KEDBYTE

SOFTWARE · AI · AUTOMATION

HOW COMPUTERS WORK

From a Grain of Sand to Artificial Intelligence

VOLUME TWO

Software

Punched cards to the first compiler, how a compiler really works, languages, operating systems, the terminal, what an executable file is, one key press traced from finger to photon, and the GPU.

WRITTEN BY

Shikhar Singh

Founder & Chairman

KedByte Technologies Private Limited

Parts D and E • Chapters 15 to 22 • First Edition

HOW COMPUTERS WORK

Volume Two — Software

Parts D and E, chapters 15 to 22

Author	Shikhar Singh
Designation	Founder & Chairman, KedByte Technologies Private Limited
Published by	KedByte Technologies Private Limited
Imprint	KedByte Press
Edition	First Edition
Volume	2 of 5
Complete work	Nine parts, fifty-three chapters

About this volume. This is Volume 2 of five. It contains Parts D and E of *How Computers Work*, chapters 15 to 22. Chapter numbers are those of the complete work, so a cross-reference to any chapter means the same chapter whether you are reading a single volume or the complete edition. References to chapters outside this volume point to the other volumes in the set.

Accuracy and currency. This book describes technology that changes. Specifications, prices, version numbers and industry figures were checked at the time of writing and are dated in the text wherever they are liable to change. Where experts disagree, the book says so rather than choosing a side.

Disclaimer. The information in this book is provided on an "as is" basis, for education. Neither the author nor KedByte Technologies Private Limited shall have any liability to any person or entity with respect to any loss or damage caused, or alleged to have been caused, directly or indirectly by the information contained in this book.

BEFORE YOU START

How to read this book

Every idea in this book is explained twice. First in plain language, with an everyday comparison and a worked example, so that a beginner can follow it. Then again in the correct technical vocabulary, with real units, real specifications and the exact figures an engineer would quote. The two halves are labelled, so you can read the plain pass end to end and then come back for the engineering.

Block	What it is for
PLAIN in simple words	The idea for a complete beginner. No jargon at all.
PLAIN a picture in your head	One everyday comparison, and then a line telling you exactly where that comparison stops being true.
PLAIN a worked example	Concrete numbers, worked step by step.
PLAIN what is really happening inside	The mechanism, still in easy words, but with nothing hand-waved.
TECHNICAL the engineer's version	Correct professional vocabulary, exact units, real specifications, real figures, and the commands that observe it.
WORDS remember these	Each term twice: the plain meaning, then the technical meaning.

1. Almost everything is written as short numbered lines, one idea per line, so nothing hides inside a paragraph.
2. *The honest version* marks the places where a simple explanation is not strictly true. The accurate model follows immediately.
3. The book says whether a thing is a **standard**, a **convention**, or one product's **implementation detail**.
4. Anything that goes stale is dated. Check it again before relying on it.
5. Every chapter ends with *Common wrong ideas* and a twenty-line summary. Short of time, read those two first.

The five volumes

1. **The Machine** — chapters 1 to 14
2. **Software** — chapters 15 to 22
3. **Networks** — chapters 23 to 34

4. **Building and Shipping Software** — chapters 35 to 44
5. **Intelligence, Games and Reference** — chapters 45 to 53

Contents

PART D

Software

From punched cards to compilers, languages, operating systems, the terminal, and what an executable file really is.

15

CHAPTER

The Birth of Software — Punch Cards to the First Compiler

15.0 What this chapter gives you

1. You will be able to say exactly what a program is at the lowest level, and show a real one as raw numbers.
2. You will be able to take a handful of bytes and work out, by hand, what the machine will do with them.
3. You will be able to describe how ENIAC was programmed in 1946 with cables and switches, and how long a change took.
4. You will be able to explain a punched card completely: its size, its 80 columns, its 12 rows, and how a letter was stored as holes.
5. You will be able to say what an assembler is and why a label beats a hand-counted address.
6. You will be able to answer "who wrote the first compiler" honestly, and say why the honest answer has several parts.
7. You will be able to say what Grace Hopper's A-0 actually did in 1952, and what it did not do.
8. You will be able to explain why FORTRAN had to be fast or it would have failed, and why COBOL still runs banks in 2026.
9. You will be able to tell the moth story correctly, including the part most retellings get wrong.
10. You will be able to trace the line from a human carrying card decks to a modern operating system, and from Multics to UNIX to C.

15.1 What a program is at the bottom

15.1.1 PLAIN - in simple words

1. A computer's memory is a very long row of numbered boxes.
2. Each box holds one small number, from 0 to 255. That is one **byte**.
3. A program is a run of boxes that the processor has agreed to read as orders instead of as data.
4. The processor keeps a marker pointing at one box. It reads the number there, does what that number means, and moves the marker on.
5. That is the whole show. Read a number, do the thing, move on. Forever.
6. The same number can be an order in one place and data in another. Only position and agreement decide which.
7. Writing a program in 1948 meant choosing every one of those numbers yourself, by hand, and getting every one right.

15.1.2 PLAIN - a picture in your head

1. Think of a long street of numbered houses, and one postman.
2. Each house has one card in the letterbox with a number on it.
3. He carries a notebook with a few slots. Those are the processor's **registers**, its scratch space.

4. He never wonders whether a card is an order or a note. Where he is standing decides that. Walk him into a shopping list and he will obey the shopping list.

Where this comparison breaks: a real processor does not work in one tidy line. Modern chips fetch many instructions ahead, guess which way branches go, and run several at once, then hide the mess so it looks sequential.

15.1.3 PLAIN - a worked example

1. Here is a real program for the MOS 6502, the chip in the Apple II, the Commodore 64 and the Nintendo Entertainment System.
2. In memory it is nothing but these seven numbers.

```
Address   Bytes        What the processor does
0600      A9 05        put 5 into the accumulator
0602      69 03        add 3 to the accumulator
0604      8D 00 02     write accumulator into address 0200
0607      00           stop
```

3. At 0600 the byte is A9, meaning "load the accumulator with the next byte". The next byte is 05, so the accumulator becomes 5.
4. At 0602 the byte is 69, meaning "add the next byte". That is 03, so the accumulator becomes 8.
5. At 0604 the byte is 8D, meaning "store the accumulator at the two-byte address that follows". Those bytes are 00 then 02.
6. The 6502 stores addresses low byte first, so 00 02 means address 0200, not 0002. That ordering is called **little-endian**.
7. Address 0200 now holds 8. At 0607 the byte 00 is BRK, and the machine stops.
8. That is a complete program. Seven numbers. Nothing else exists.

15.1.4 PLAIN - what is really happening inside

1. Inside the processor is a register called the **program counter**, holding the address of the next instruction.
2. That address goes onto the address wires, and memory puts the byte at that address onto the data wires.
3. The byte lands in the instruction register, and a block of logic called the **decoder** turns on a specific set of control wires.
4. Meanwhile the program counter has already advanced to the next byte.
5. A9 does not "mean" load in any deep sense. The decoder was built so the bit pattern 1010 1001 turns on the load-immediate wires. That is all.
6. Different chip, different wiring, different meanings for the same numbers. That is why code for one processor family is noise to another.

15.1.5 TECHNICAL - the engineer's version

1. The 6502 is an 8-bit processor with a 16-bit address bus, so it addresses 65,536 bytes. MOS Technology introduced it in September 1975 at 25 United States dollars, against roughly 179 dollars for competing parts.
2. User-visible registers: A (accumulator), X and Y (index), SP (stack pointer, fixed to page 01), PC (16 bits) and P (status flags).
3. A9 is LDA immediate, 2 bytes, 2 cycles. 69 is ADC immediate, 2 bytes, 2 cycles. 8D is STA absolute, 3 bytes, 4 cycles. 00 is BRK, 1 byte, 7 cycles.
4. ADC adds the carry flag as well as the operand. That is not optional. If carry is set from earlier work, the example above yields 9, not 8. Correct 6502 code clears carry first with CLC, opcode 18.
5. Little-endian ordering for 16-bit operands is an architectural rule of the 6502, not a convention you may vary.

Item	6502 (1975)	x86-64 (today)
Opcode length	1 byte	1 to 4 bytes
Instruction length	1 to 3 bytes	1 to 15 bytes
Address bus	16 bits	48 bits used
Documented opcodes	151	over 1,000

6. To see this today: `objdump -d` on Linux, `otool -tv` on macOS, `xxd` or `hexdump -C` for raw bytes, and `gdb` with `x/8xb` to dump memory.

The honest version: "the CPU reads a byte and does it" is a fair description of a 1975 chip and a poor one of a 2026 chip. A modern x86 core decodes variable-length instructions into fixed micro-operations, renames registers, reorders execution, and retires results in order to keep up the appearance of sequence. The programmer's model is still one at a time; the hardware stopped working that way around 1995.

15.1.6 WORDS - remember these

1. Byte — a number from 0 to 255 — 8 bits, the standard addressable unit.
2. Machine code — the raw numbers a chip obeys — the binary encoding of an instruction set architecture.
3. Opcode — the number saying which operation — the operation field of an encoded instruction.
4. Program counter — the marker for the next order — a register holding the address of the next instruction to fetch.
5. Register — the processor's scratch slots — small fast storage inside the core, named in the instruction encoding.
6. Little-endian — low part of a number stored first — least significant byte at the lowest address.

15.2 Before any language existed: programming by rewiring

15.2.1 PLAIN - in simple words

1. In 1945 there was no such thing as a programming language, and barely such a thing as a program.
2. ENIAC, finished in late 1945 at the University of Pennsylvania, did not read instructions from memory. It had no memory for instructions.
3. To make it do a calculation you physically rebuilt the connections between its parts.
4. You plugged heavy cables between panels, exactly like an old telephone switchboard, and set banks of rotary switches by hand.
5. The program existed as the arrangement of cables and the positions of switches. Nothing was stored anywhere.
6. A new problem took weeks of planning on paper, days of physical work on the machine, and days more of finding what was wired wrong.
7. The people doing this were not called programmers. They were called operators, because until then "computer" meant a person who computed.

15.2.2 PLAIN - a picture in your head

1. Think of a model railway with no track laid down.
2. To run a train from the station to the mine you lay every piece of track, set every point and connect every signal wire yourself.
3. Now you want the train to go to the quarry instead. There is no button for that. You lift the track and lay it again.
4. The layout is the program. A stored-program computer is the opposite: the track is fixed forever, and you hand the driver a route on a slip of paper.

Where this comparison breaks: ENIAC's units could repeat and could branch on a condition, so it was more capable than fixed track. And in 1948 it was converted so its function tables held a stored program, cutting setup from days to hours at the cost of running about six times slower.

15.2.3 PLAIN - a worked example

1. ENIAC had 20 accumulators, each holding a 10-digit decimal number.
2. To compute "add accumulator 3 and accumulator 5 into accumulator 7" you did the following.
3. Run a program cable from a digit output socket on accumulator 3 to a digit input socket on accumulator 7.
4. Run a program pulse cable so the right control pulse triggers both transmissions at the same moment in time.
5. Do that for every step of the calculation. There were hundreds of steps.
6. Then test it, and when the answer is wrong, work out whether the fault is your wiring, a bad switch, or one of 18,000 vacuum tubes that has failed since breakfast.

15.2.4 PLAIN - what is really happening inside

1. ENIAC was a set of separate calculating units that could be joined in any pattern.
2. Two kinds of cable ran between them. Digit trunks carried the numbers. Program trunks carried timing pulses meaning "now".
3. So a program was a graph: which unit sends numbers to which, and in what order the "now" pulses fire. That is closer to circuit design than to writing instructions.
4. The three function tables were racks of switches, 1,200 ten-way switches per table, set by hand to hold constants and lookup values.
5. Because the units ran in parallel, ENIAC could do several things at once. It was a parallel machine before it was a programmable one.
6. In 1948 it was rewired once, permanently, so the function tables held a list of coded orders that the machine fetched and obeyed.
7. That change turned a rewiring job into a switch-setting job, and it is the moment ENIAC became something we would recognize as programmable.

15.2.5 TECHNICAL - the engineer's version

1. ENIAC: Electronic Numerical Integrator and Computer, built at the Moore School of Electrical Engineering, University of Pennsylvania, for the US Army Ballistic Research Laboratory.
2. First serious work: December 1945, Los Alamos thermonuclear calculations. Publicly announced and dedicated 14 to 15 February 1946.
3. About 18,000 vacuum tubes, 7,200 crystal diodes, 1,500 relays, over 30 short tons, roughly 30 by 3 metres of floor space, 150 kilowatts.
4. Clock rate 100 kHz, giving a 200-microsecond addition cycle: around 5,000 additions per second.
5. The six principal programmers were Kathleen McNulty, Jean Jennings, Betty Snyder, Marlyn Wescoff, Frances Bilas and Ruth Lichterman. They were hired as human computers and taught themselves the machine from wiring diagrams.
6. The 1948 converter-code modification, with input from John von Neumann and implemented with Richard Clippinger and Adele Goldstine, gave ENIAC a 60-order instruction set read from function table switches.

Task	ENIAC 1946	ENIAC after 1948
Set up a new program	days	hours
Fix a coding mistake	hours	minutes
Additions per second	about 5,000	about 800

The honest version: the popular line "ENIAC took two weeks to reprogram" compresses several different things. Planning could take weeks, physical setup days, and debugging days more. The 1948 conversion is the line that matters, and after it ENIAC was slower but far more usable.

15.2.6 WORDS - remember these

1. Plugboard — a panel you wire by hand — a patch panel defining data and control routing.
2. Accumulator — a box holding a running total — a register that is both operand source and result destination.
3. Function table — a rack of setting switches — read-only storage for constants, later used as instruction memory.
4. Stored program — orders kept in memory like data — the von Neumann model, with instructions and data in one addressable store.
5. Setup time — the wait before the machine can start — non-productive time between jobs, the cost batch systems later attacked.

15.3 Programming in raw numbers: switches, tape and cards

15.3.1 PLAIN - in simple words

1. Once machines stored their programs in memory, the question became how to get the numbers in there.
2. The first answer was the crudest possible: a row of switches on the front of the machine, one switch per bit.
3. The second answer was **paper tape**: a ribbon of paper with rows of holes punched across it. A hole is a 1, no hole is a 0.
4. The third answer, which ruled computing for forty years, was the **punched card**: a stiff paper rectangle holding one line of up to 80 characters.
5. A program was a stack of cards, called a **deck**, held with a rubber band.
6. You gave your deck to an operator. Hours or days later you got a printout back. One mistake meant fixing one card and queueing again.

15.3.2 PLAIN - a picture in your head

1. Imagine writing an essay where each line goes on its own index card, and you may not see the essay until a stranger reads all the cards aloud.
2. The cards must be in perfect order, there are 2,000 of them, and they are not fastened together.
3. So you draw a diagonal ink line across the top edge of the whole stack. If a card moves, the line has a jog in it.
4. You punch a fresh card 412, slide it in, and go back to the shelf. One full day for one typing mistake.

Where this comparison breaks: index cards can be read by anyone. A punched card carried its content as holes, and the characters printed along the top edge were an optional courtesy from the keypunch. Without that printing, a card was unreadable by a human.

15.3.3 PLAIN - a worked example

1. Here is exactly how characters were stored as holes. The coding is called **Hollerith code**, after Herman Hollerith, who patented punched-card tabulating in 1889 for the 1890 United States census.
2. A card has 12 rows. From the top edge down they are row 12, row 11, row 0, then rows 1 to 9.
3. Rows 12, 11 and 0 are the **zone** rows. Rows 0 to 9 are the **digit** rows. Row 0 does double duty.
4. A digit is a single punch in its own row. A letter is two punches: one zone plus one digit.
5. A to I are zone 12 with digits 1 to 9. J to R are zone 11 with digits 1 to 9. S to Z are zone 0 with digits 2 to 9.
6. Notice S starts at digit 2, not 1. There are 26 letters and 27 slots, so the last group is short by one. That quirk is real.

```
row       'A'   'J'   'S'   '5'
12  ->     X     .     .     .
11  ->     .     X     .     .
 0  ->     .     .     X     .
 1  ->     X     X     .     .
 2  ->     .     .     X     .
 3  ->     .     .     .     .
 4  ->     .     .     .     .
 5  ->     .     .     .     X
 6  ->     .     .     .     .
 7  ->     .     .     .     .
 8  ->     .     .     .     .
 9  ->     .     .     .     .
```

7. So A is holes in rows 12 and 1 of one column, S is holes in rows 0 and 2, and the digit 5 is one hole in row 5.
8. Eighty columns gives eighty characters per card, and that is where the traditional 80-character terminal width comes from.

15.3.4 PLAIN - what is really happening inside

1. A keypunch is a typewriter that makes holes instead of ink. It also prints the characters along the top edge, which is called **interpreting**.
2. A card reader pulls cards through one at a time and shines light or presses metal brushes at each of the 960 hole positions.
3. Where there is a hole, light passes or a brush completes a circuit. That is one bit.
4. So a card is not read as characters. It is read as 12 bits per column, and software decides what those 12 bits mean.
5. Your deck joined a queue with everyone else's, and the machine ran jobs one after another with no human in between. That is **batch processing**.
6. Turnaround, from handing in your deck to collecting your printout, was a few hours in a well-run university and overnight or worse elsewhere.
7. Because turnaround was slow, programmers did something we have nearly lost. They checked their code line by line on paper first, called **desk checking**.

```
write code on a coding sheet          (30 minutes)
      |
      v
keypunch each line onto a card        (1 hour)
      |
      v
hand the deck in at the window        (join the queue)
      |
      v
operator loads it, machine runs it    (2 seconds of CPU)
      |
      v
printout goes into a pigeonhole
      |
      v
you collect it next morning           (turnaround: 20 hours)
      |
      v
one missing comma  ->  do it all again
```

15.3.5 TECHNICAL - the engineer's version

1. The IBM 80-column card, introduced in 1928, measures 7 and 3/8 by 3 and 1/4 inches, that is 187 by 83 millimetres, on stock about 0.007 inches or 180 micrometres thick. Roughly 143 cards stack to one inch.
2. Rectangular holes replaced earlier round holes precisely so columns could be packed tighter, doubling capacity from 45 to 80 columns.
3. Columns 73 to 80 were reserved by convention for a card sequence number. That is why FORTRAN, by rule, ignores columns 73 to 80 of every line.
4. A dropped deck was recoverable if and only if you had punched those sequence numbers, by running it through a card sorter such as the IBM 82, 83 or 84.

Device	Role	Speed
IBM 026 keypunch	punch cards by hand	typing speed
IBM 711 (for 704)	card reader	250 cards/min
IBM 1402	card reader/punch	800 read, 250 punch
IBM 2540	card reader/punch	1,000 read, 300 punch

5. Front-panel switch entry survived into the microcomputer era. The Altair 8800 of January 1975 shipped with no keyboard: you entered the bootstrap loader on toggle switches, one byte at a time.
6. Job control was itself punched on cards. IBM's Job Control Language, JCL, survives, and its statements still begin with two slashes in columns 1 and 2 because that is where a card reader saw them.

The honest version: "turnaround was a day" is a rough average that varied enormously. A physics group with its own machine at 3 a.m. might get five runs an hour; a student at a busy service bureau might get one run a day, and in bad commercial cases two or three days.

15.3.6 WORDS - remember these

1. Punched card — stiff card storing data as holes — 80-column, 12-row IBM card using Hollerith encoding.
2. Deck — a stack of cards making one program — an ordered card file, sequence-numbered in columns 73 to 80.
3. Keypunch — the machine that punches cards — a card punch with a keyboard, for example the IBM 026 or 029.
4. Batch processing — jobs run one after another with no user present — non-interactive execution under a resident monitor.
5. Turnaround — how long you wait for your answer — elapsed time from job submission to output delivery.
6. Desk checking — reading your own code before running it — manual static verification, the ancestor of code review.

15.4 The first step up: mnemonics and assembly language

15.4.1 PLAIN - in simple words

1. Writing programs as numbers works, and it is unbearable.
2. You must remember that 169 means load and 141 means store, for hundreds of operations, with no help.
3. The fix is small and it changed everything: let people write short words instead of numbers, and let a program do the translation.
4. Instead of A9 you write LDA. Instead of 8D you write STA. These short words are **mnemonics**, which just means memory aids.

5. Instead of "jump back 10 bytes" you write "jump to loop", and put the word `loop` beside the line you meant. That word is a **label**.
6. The program that turns this text into numbers is an **assembler**.
7. An assembler is not clever. Mostly it is one line in, one instruction out. But it removes the two things humans are worst at: remembering arbitrary numbers, and counting.
8. This is the first time in history that a program's job was to write another program.

15.4.2 PLAIN - a picture in your head

1. Imagine giving directions using only distances: go 412 metres, turn, go 78 metres, turn, go 1,050 metres.
2. Now someone adds a roundabout early in the route. Every distance after it is wrong and must be recomputed.
3. Labels are street names: go to Mill Street, turn towards the bridge. Add a roundabout and the directions still work, because names do not shift when things move.
4. The assembler is the local who converts street names back into exact distances, correctly, in a second.

Where this comparison breaks: an assembler does not choose a route. It has no freedom at all. Every line you write becomes the instruction you asked for. A compiler, later in this chapter, does choose.

15.4.3 PLAIN - a worked example

1. Here is a real 6502 program that adds the numbers 1 to 5. On the left is what the assembler produces; on the right is what you write.

```
Addr   Bytes       Label   Assembly       Comment
0600   A9 00               LDA #$00       running total = 0
0602   A2 01               LDX #$01       counter = 1
0604   18          loop:   CLC            clear the carry flag
0605   86 10               STX $10        put counter in memory $10
0607   65 10               ADC $10        total = total + counter
0609   E8                  INX            counter = counter + 1
060A   E0 06               CPX #$06       is counter now 6?
060C   D0 F6               BNE loop       if not, go round again
060E   8D 00 02            STA $0200      save the total
0611   00                  BRK            stop
```

2. Trace it. Total starts at 0, counter at 1.
3. Pass 1: total = 0 + 1 = 1, counter becomes 2.
4. Pass 3: total = 3 + 3 = 6, counter becomes 4.
5. Pass 5: total = 10 + 5 = 15, counter becomes 6, the compare matches, and the branch is not taken. Address 0200 ends up holding 15.
6. Now look at the byte F6 on the BNE line. That is minus 10 in two's complement: jump back 10 bytes from the instruction after the branch.
7. Here is the whole point. Insert one extra instruction anywhere between `loop:` and the branch, and F6 becomes wrong.
8. With a label you insert the line and reassemble. The assembler recounts, and you never see the number F6 at all.

15.4.4 PLAIN - what is really happening inside

1. An assembler reads your text twice, which is why they are called **two-pass assemblers**.
2. On the first pass it walks the lines, keeps a running address counter, and writes down where every label lands. That list is the **symbol table**.
3. It has to walk first because of forward references: you can jump to a label that appears later, and on first sight its address is unknown.
4. On the second pass it converts each line to bytes, looking up every label in the symbol table.

5. For a relative branch it does one more step: subtract the address of the next instruction from the target, and check the answer fits in a signed byte.
6. And they handle **macros**: define a named block of lines once, then write its name wherever you want the block expanded.

15.4.5 TECHNICAL - the engineer's version

1. The earliest known symbolic coding scheme is in Kathleen Booth and Andrew Donald Booth's 1947 work "Coding for A.R.C." at Birkbeck College, London. Kathleen Booth is widely credited with inventing assembly language.
2. EDSAC at Cambridge first ran a program on 6 May 1949. Its **Initial Orders**, written by David Wheeler, were a bootstrap routine held permanently in uniselector switches.
3. Those Initial Orders read instructions as a single letter mnemonic, a decimal address and a length code, and turned them into binary on load. The IEEE Computer Society credits Wheeler as creator of the first assembler.
4. Wheeler also devised the **Wheeler Jump**: place the return address in the accumulator, jump to the subroutine, and let the subroutine plant that address into its own final jump. That is the first practical closed subroutine call.
5. EDSAC's subroutine library reached 87 routines by 1951, covering floating-point arithmetic, trigonometry, differential equations and matrix work. That is the first software library.
6. Maurice Wilkes, David Wheeler and Stanley Gill published "The Preparation of Programs for an Electronic Digital Computer" in 1951, the first programming textbook. It introduced the word "assemble" for joining separately written sections into one program.
7. SOAP, the Symbolic Optimal Assembly Program, written by Stan Poley for the IBM 650 in 1955, placed instructions around the rotating drum so the next one arrived under the read head just as it was needed. That is optimization by an assembler.

Year	Milestone	Who and where
1947	Coding for A.R.C.	K. and A. Booth, Birkbeck
1949	EDSAC Initial Orders	D. Wheeler, Cambridge
1951	First programming book	Wilkes, Wheeler, Gill
1955	SOAP for IBM 650	Stan Poley, IBM

8. Modern equivalents to observe: `as` (GNU assembler), `nasm`, `objdump -d` to disassemble, and `gcc -S` to see the assembly a compiler emits.

The honest version: "assembly is one-to-one with machine code" is nearly true and not exactly true. Pseudo-instructions, macros and assembler-chosen branch widths mean one written line can become several instructions, or none. On MIPS and RISC-V this is routine and documented.

15.4.6 WORDS - remember these

1. Mnemonic — a short word for an operation — the textual name of an opcode.
2. Label — a name for a place in the program — a symbol bound to an address by the assembler.
3. Assembler — the program turning mnemonics into numbers — a translator with a largely one-to-one instruction mapping.
4. Symbol table — the list of names and addresses — the assembler's map from identifiers to values, built in pass one.
5. Subroutine — code you can call from many places — a closed routine entered with a return linkage.
6. Macro — a named block that gets pasted in — textual expansion performed before assembly.

15.5 Why a higher language was needed

15.5.1 PLAIN - in simple words

1. Assembly made programming possible for ordinary mortals. It did not make it quick, and it did not make it safe.
2. Problem one: assembly is welded to one machine. Code for an IBM 704 is worthless on a UNIVAC I. Buy a new computer, rewrite everything.
3. Problem two: it is slow to write. One line is one machine operation, so a page of algebra becomes hundreds of lines.
4. Problem three: it hides your intent. Reading it tells you what the machine does, never what you meant.
5. So people asked the obvious question. Could you write the formula the way a mathematician writes it, and have a program produce the assembly?
6. Many experienced programmers said no, and they were not being silly. Machines were tiny, machine time cost more than people, and a program that wasted 30 per cent of the machine was unacceptable.
7. So the real question was never "can a machine translate". It was "can a machine translate well enough that we do not lose the machine".

15.5.2 PLAIN - a picture in your head

1. Imagine a workshop where every screw is filed by hand by a master, and the masters are proud of it.
2. Someone brings in a screw-cutting machine. The masters are not afraid of losing their jobs. They are afraid the screws will be worse.
3. If the machine's screws are 30 per cent weaker, the objection is correct and the machine should be rejected.
4. If they are as good and made in a tenth of the time, the argument ends overnight and nobody files screws again. That is exactly what happened with compilers between 1952 and 1957.

Where this comparison breaks: code quality has many dimensions. A compiler can be worse on speed and far better on correctness, portability and maintenance, and for most work those matter more. The 1950s programmer with 4,096 words of memory did not have that luxury.

15.5.3 PLAIN - a worked example

1. Take one line of mathematics: the roots of a quadratic need `d = b*b - 4*a*c`.
2. In a high-level language that is one line and you can see the formula. In 1950s assembly it is roughly this shape of work.

```
load    b           get b into the accumulator
multiply b          now accumulator holds b*b
store   temp1       park it
load    four        get the constant 4
multiply a          now 4*a
multiply c          now 4*a*c
store   temp2       park it
load    temp1       get b*b back
subtract temp2      b*b - 4*a*c
store   d           save the answer
```

3. Ten lines, two temporary locations you invented and must not reuse by accident, and one constant you placed in memory by hand.
4. Nothing in the assembly says "this is the discriminant of a quadratic". You must comment it, and comments drift out of date.

15.5.4 PLAIN - what is really happening inside

1. The gap a compiler must cross is bigger than it looks. Four problems hide inside "translate the formula".
2. **Parsing**: work out from flat text that in `a + b * c` the multiplication binds tighter, so it happens first.
3. **Temporaries**: the machine has a handful of registers and your expression has many intermediate values. Something must decide what lives where.
4. **Addressing**: `X(I)` in a loop means a different memory address every time round, so the compiler must generate address arithmetic you never wrote.
5. **Optimization**: a naive translation reloads the same value from memory again and again. A human would not, and the compiler must not either.
6. The fourth is the hard one, and it decided whether high-level languages would be accepted at all.
7. A compiler producing code twice as slow as hand assembly would have been a curiosity. One that came within a few per cent would end the argument.

15.5.5 TECHNICAL - the engineer's version

1. The economics are the whole story. In 1957 an IBM 704 rented for roughly several tens of thousands of United States dollars per month, while a programmer earned a few hundred dollars per month. Machine time dominated.
2. John Backus, who led the FORTRAN project, described the prevailing view plainly: most programmers of the time believed no automatic system could produce object code comparable to a good hand coder's.
3. The FORTRAN team's target was output within roughly a factor of one to two of hand-written assembly for typical scientific code. They largely achieved it, and that is why FORTRAN spread.
4. Earlier interpreted "automatic programming" systems, such as Backus's own Speedcoding for the IBM 701 in 1953, ran perhaps 10 to 20 times slower than hand code. Useful, but easy to dismiss.

Concern	Assembly	High-level language
Tied to one machine	yes	mostly no
Lines per formula	10 to 30	1
Type checking	almost none	some to strong
1957 speed penalty	zero	a few per cent to 2x

The honest version: the sceptics were not simply wrong and later proved foolish. For a decade they were right about specific cases, and even in 2026 there are hot loops in codecs, cryptography and kernels where hand-written assembly still beats a compiler. What changed is that those cases went from "most code" to a fraction of one per cent of it.

15.5.6 WORDS - remember these

1. Portability — code that runs on more than one machine — source-level independence from a specific instruction set.
2. Parsing — working out the structure of written text — building a syntax tree from tokens according to a grammar.
3. Optimization — making generated code faster or smaller — semantics preserving transformation of an intermediate representation.
4. Object code — the machine code a translator produces — a relocatable module containing code, data and relocation entries.
5. Intermediate value — a part-finished result — a temporary held in a register or spilled to a stack slot.

15.6 Grace Hopper and the first compilers

15.6.1 PLAIN - in simple words

1. Grace Hopper joined the United States Navy Reserve in 1943 and in 1944 was assigned to the Harvard Mark I under Howard Aiken.
2. The Mark I was an electromechanical calculator over 15 metres long. Hopper was one of its first programmers and wrote its manual, published in 1946 and running to over 500 pages.
3. In 1949 she moved to the Eckert-Mauchly Computer Corporation to work on UNIVAC I, the first commercial computer sold in the United States.
4. There she noticed something dull and important. Programmers kept copying the same routines out of notebooks by hand, and kept making copying errors.
5. Her idea: keep those routines on magnetic tape, and let a program fetch them and stitch them together.
6. She called that program a **compiler**, using the ordinary English meaning. To compile means to gather things from sources into one collection, as one compiles an anthology of poems.
7. The system was called A-0 and it worked in 1951 and 1952.
8. Here is the careful part. A-0 did not translate formulas into machine code. It gathered, positioned and joined ready-made routines.
9. By today's meanings, A-0 is much closer to what we now call a **linker** or **loader** than to a compiler.
10. That is not a criticism. The word changed meaning afterwards, partly because of her, and we are judging 1952 by a 2026 dictionary.

15.6.2 PLAIN - a picture in your head

1. Picture a cookbook where each recipe is on its own card in a filing drawer.
2. You want a three-course dinner, so you write a short list: recipe 47, recipe 12 for 4 servings, recipe 90.
3. An assistant takes the list, pulls those cards, rewrites the quantities for the servings you asked, and staples them into one set of instructions.
4. That is A-0. The assistant invented no cooking. The assistant gathered, adjusted and joined.
5. A modern compiler is a different assistant. You say "a light fish dinner for four" and it writes the recipes itself.

Where this comparison breaks: A-0 did more than staple. It adjusted the addresses inside each routine so they worked at their new position in memory, and substituted the arguments you supplied. That address adjustment is called relocation, and it is real work, not clerical work.

15.6.3 PLAIN - a worked example

1. In A-0 you wrote a program as a list of call numbers with arguments, not as machine instructions. The shape of it was like this, in spirit.

```
your input                what A-0 did with it
------------------------  ------------------------------
subroutine 015, X, Y      find routine 015 on the tape
                          copy it into place
                          fix its internal addresses
                          plug in X and Y
subroutine 032, Y         same again for routine 032
subroutine 007, Y, Z      same again for routine 007
------------------------  ------------------------------
result: one runnable program on tape, ready to load
```

2. Notice what is present: a library, selection from it, relocation, argument substitution, and output of a single runnable program.
3. Notice what is absent: no arithmetic expressions, no variables in the modern sense, no control statements, no code generation from a grammar.

4. Hopper described the work in a 1952 paper to the Association for Computing Machinery titled "The Education of a Computer".
5. A-2 followed in 1953. Remington Rand UNIVAC shipped it to customers by the end of that year with the source code included, and invited customers to send improvements back. That is an early ancestor of open source practice.
6. Then came ARITH-MATIC (A-3), MATH-MATIC (AT-3), and the important one, B-0, better known as **FLOW-MATIC**.

15.6.4 PLAIN – what is really happening inside

1. FLOW-MATIC, worked on from 1955 and substantially complete by 1959, is where Hopper's real revolution sits.
2. Her observation was about people, not machines. Business managers threw out anything full of mathematical symbols.
3. So FLOW-MATIC used English words as its statements. It was the first programming language to do so.
4. Statements looked like `COMPARE PRODUCT-NO (A) WITH PRODUCT-NO (B)` and `TRANSFER A TO D` and `READ-ITEM A ; IF END OF DATA GO TO OPERATION 14`.
5. Her management initially thought English-language programming impractical. She proposed it in 1953, and the compiler became publicly available in early 1958. It fed almost directly into COBOL.

15.6.5 TECHNICAL – the engineer's version

1. Now the credit question, precisely, because it is asked constantly and answered badly. There are at least four serious claims, and they answer different questions.

Claim	Year	What it was
Corrado Bohm, thesis	1951	translator described on paper
Hopper, A-0	1952	library linker and loader
Glennie, Autocode	1952	real translator, ran on Mark 1
Backus et al, FORTRAN	1957	first widely used compiler

2. **Corrado Bohm**, doctoral thesis at ETH Zurich, submitted 1951, defined a language and described a translator for it written in that same language. It is arguably the first self-describing compiler. It was not implemented on a machine at the time.
3. **Grace Hopper, A-0**, 1951 to 1952 on UNIVAC I. She coined and popularized the term "compiler". The system selected subroutines from a tape library by call number, relocated them, substituted arguments and produced a single runnable program.
4. In modern terminology, A-0 is a linking loader.
5. **Alick Glennie, Autocode**, 1952, for the Manchester Mark 1 at the Royal Armament Research Establishment. It accepted a symbolic, algebra-like notation and generated machine code, and many historians call it the first compiler in the modern sense.
6. Glennie's own manual claimed the efficiency loss was no more than 10 per cent. It was barely used, even at Manchester.
7. **J. Halcombe Laning and Neal Zierler**, MIT Whirlwind, from 1952 and operational by 1954. It accepted algebraic formulas in near-mathematical notation and produced machine code, with subroutine linkage, arrays and indexing, and a Runge-Kutta differential equation solver.
8. It is often called the first operating algebraic compiler.
9. **R. A. Brooker's Mark 1 Autocode**, planned 1954 and working 1955, was the Manchester system actually used heavily, unlike Glennie's.
10. So the correct answers, depending on the question asked:
11. First thing called a compiler: Hopper's A-0, 1952.

12. First compiler described in the modern sense: Bohm's 1951 thesis, on paper only.
13. First running translator from a higher notation to machine code: Glennie's Autocode, 1952.
14. First algebraic compiler in practical use: Laning and Zierler, 1954.
15. First compiler that displaced hand assembly at scale: FORTRAN, 1957.
16. Where experts disagree: some historians, including commentary published by the Association for Computing Machinery, argue plainly that A-0 was not a compiler and the popular claim overstates it.
17. Others argue that in 1952 the word meant what Hopper made it mean, and that judging her by a later definition is unfair. Both positions are defensible.
18. What is not in dispute, and is enough on its own: Hopper coined the vocabulary of automatic programming, built the first practical subroutine library system, led FLOW-MATIC, the first English-language programming language, and shaped COBOL.
19. She reached the rank of Rear Admiral, retired in 1986, and died on 1 January 1992.

The honest version: "Grace Hopper wrote the first compiler" is defensible only if you use her 1952 definition of the word and not yours. The accurate sentence is that she wrote the first system anyone called a compiler, and it did linking and loading rather than translation. The exaggeration is unnecessary. Her uncontested achievements are larger than the disputed one.

15.6.6 WORDS - remember these

1. Compiler — a program that turns your writing into machine code — a translator from a source language to machine or object code.
2. Linker — the tool joining separate pieces into one program — resolves external symbol references between object modules.
3. Loader — the tool putting a program into memory to run — performs relocation and transfers control to the entry point.
4. Relocation — fixing addresses when code moves — adjusting address fields for a load-time base other than the assembled one.
5. Subroutine library — a stored collection of reusable routines — an archive of object modules selected by name at link time.
6. FLOW-MATIC — the first English-worded language — B-0, Remington Rand, 1955 to 1959, direct ancestor of COBOL.

15.7 FORTRAN: the compiler that had to be fast

15.7.1 PLAIN - in simple words

1. In late 1953 John Backus, aged 29 and working at IBM, sent his managers a proposal to let scientists write formulas instead of assembly.
2. The project ran from 1954. A draft specification was finished in November 1954, the first manual appeared in October 1956, and the compiler was delivered in April 1957 for the IBM 704.
3. The name is short for FORmula TRANslation.
4. The team knew the project would fail unless the output was fast. Programmers would not accept a 30 per cent penalty on a machine costing a fortune per hour.
5. So most of the effort went not into the language but into the optimizer, the part that makes the generated code good.
6. It worked. FORTRAN produced code close enough to hand-written assembly that experienced programmers accepted it.
7. Within a few years, writing scientific programs in assembly became a strange thing to do. That is the moment high-level programming won.

15.7.2 PLAIN - a picture in your head

1. Imagine the first automatic gearbox. Every racing driver says a human changes gear better.
2. If it is clumsy and slow they are right, and it fails.
3. The FORTRAN team spent their effort making the gearbox change gear as well as a good driver, not on making the steering wheel prettier.
4. Once it did, the argument stopped being about speed and became about who wants to change gear 40,000 times on a journey.

Where this comparison breaks: an automatic and a manual gearbox drive the same car. A compiler changes what you can express, not just your comfort. Whole classes of program became possible because nobody had to track a thousand addresses by hand.

15.7.3 PLAIN - a worked example

1. Here is a small, period-authentic FORTRAN program that adds 1 to 5 and prints the total. Compare it with the 6502 assembly earlier in this chapter.

```
C     ADD THE WHOLE NUMBERS 1 TO 5 AND PRINT THE TOTAL
      ITOTAL = 0
      DO 10 I = 1, 5
      ITOTAL = ITOTAL + I
   10 CONTINUE
      PRINT 20, ITOTAL
   20 FORMAT (9H TOTAL = , I3)
      STOP
      END
```

2. Line 1 starts with C in column 1, which marks a comment. That is a card convention: the whole language is laid out for 80-column cards.
3. Columns 1 to 5 hold a statement number, column 6 marks a continuation card, columns 7 to 72 hold the statement, and columns 73 to 80 are ignored because that is where the card sequence number went.
4. `ITOTAL` is an integer with no declaration. In FORTRAN a name beginning with I, J, K, L, M or N is an integer by default; everything else is real.
5. `DO 10 I = 1, 5` means repeat down to statement 10 with I taking the values 1 to 5.
6. `9H TOTAL =` is a Hollerith constant: the 9 says the next 9 characters are literal text. Quoted strings arrived later, in FORTRAN 77.
7. Nine lines. The equivalent assembly is several dozen, and the assembly does not survive a change of computer.

15.7.4 PLAIN - what is really happening inside

1. The FORTRAN compiler had to do the four hard jobs listed earlier in this chapter, on a machine with 4,096 or 8,192 words of memory.
2. First it read a statement and worked out the structure of the expression, deciding what binds to what.
3. Then it turned `DO` loops into counters, comparisons and jumps, and turned array subscripts into address arithmetic.
4. Then came the part that mattered: deciding which values to keep in the 704's three index registers rather than reloading them from memory.
5. It also did what we now call common subexpression elimination and loop invariant motion: compute a thing once, and move work that does not change out of the loop.
6. Those techniques were essentially invented here, for this compiler, because there was no choice.
7. The compiler ran in several separate passes, reading and rewriting the program from tape between passes, because it could not all fit in memory.

15.7.5 TECHNICAL - the engineer's version

1. Backus's team at IBM included Irving Ziller, Harlan Herrick, Robert Nelson, Roy Nutt, Sheldon Best, Richard Goldberg, Lois Haibt, David Sayre, Peter Sheridan and Harold Stern.
2. The effort is commonly cited at about 18 person-years over roughly three years for the original IBM 704 compiler.
3. Target machine: the IBM 704, a 36-bit word machine with hardware floating point, 3 index registers, and typically 4,096 to 32,768 words of core.

Version	Year	Notable addition
FORTRAN	1957	first delivery, IBM 704
FORTRAN II	1958	user subroutines, functions
FORTRAN IV	1962	machine independence push
FORTRAN 66	1966	first ANSI standard
FORTRAN 77	1978	IF-THEN-ELSE, strings
Fortran 90	1991	free form, array syntax
Fortran 2023	2023	current ISO standard

4. FORTRAN 66 was the first standardized programming language in history. That is a standard in the strict sense: a written specification adopted by a standards body.
5. Backus received the ACM Turing Award in 1977 for FORTRAN. He had earlier written Speedcoding for the IBM 701 in 1953, an interpreted system, which taught him how badly interpretation cost.
6. Fortran is not a museum piece. As of 2026 it remains standard in weather forecasting, computational fluid dynamics, nuclear engineering and climate modelling, and the LINPACK and LAPACK numerical libraries beneath a great deal of scientific software began in Fortran.

The honest version: FORTRAN was neither the first high-level language nor the first compiler. It was the first that people used in large numbers, which is a different and more important claim. Autocode and Laning-Zierler came first and were barely adopted.

15.7.6 WORDS - remember these

1. FORTRAN — formula translation — IBM's 1957 scientific language, the first widely adopted high-level language.
2. Optimizer — the part that makes code fast — the compiler passes performing semantics-preserving transformations.
3. Implicit typing — the language guessing a variable's type from its name — FORTRAN's I-to-N integer rule, switchable off with IMPLICIT NONE.
4. Fixed-form source — code laid out in fixed columns — label 1-5, continuation 6, statement 7-72, from the punched card.
5. Loop invariant motion — moving unchanging work out of a loop — a classic optimization introduced in the 1957 FORTRAN compiler.

15.8 COBOL, LISP and ALGOL: three different futures

15.8.1 PLAIN - in simple words

1. Between 1958 and 1960 three languages appeared that between them set the shape of nearly everything since.
2. COBOL was for business. It reads like English sentences and is built around records, files and exact decimal money.
3. LISP was for symbols and for thinking about thinking. It is built around lists and functions, and programs in it are themselves lists.

4. ALGOL was for describing algorithms clearly. It was never widely used to run real work, and almost every modern language is its descendant.
5. COBOL came from a committee organized by the United States Department of Defense in 1959, with Grace Hopper as a technical adviser.
6. LISP came from one man, John McCarthy, at MIT in 1958.
7. ALGOL came from a joint European and American committee, and produced the ALGOL 60 report in January 1960.

15.8.2 PLAIN - a picture in your head

1. Think of three documents: a company's accounting ledger, a mathematician's notebook, and a legal statute.
2. COBOL is the ledger. Every field has a fixed size, every amount has exactly two decimal places, and it must balance.
3. LISP is the notebook. Everything is written in one simple form, and you are free to write notes about your own notes.
4. ALGOL is the statute: a precise written definition of what the words mean, published so anyone can implement it exactly.

Where this comparison breaks: ALGOL 60 was a real running language, not just a document. But its lasting effect really is definitional, because the ALGOL 60 report taught the world how to specify a language properly.

15.8.3 PLAIN - a worked example

1. Here is COBOL. The shape is deliberately unlike everything else.

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. PAYTOTAL.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  WS-TOTAL     PIC 9(7)V99 VALUE ZERO.
       01  WS-AMOUNT    PIC 9(5)V99 VALUE 125.50.
       PROCEDURE DIVISION.
           ADD WS-AMOUNT TO WS-TOTAL.
           DISPLAY "TOTAL IS " WS-TOTAL.
           STOP RUN.
```

2. `PIC 9(7)V99` means seven digits, then an assumed decimal point, then two digits. The V is not stored; it marks where the point sits.
3. That single feature is why COBOL still runs banks. It stores money as exact decimal digits, not binary floating point, so 0.10 plus 0.20 is exactly 0.30 and never 0.30000000000000004.
4. Here is LISP. The whole language is this one shape.

```
(defun sum-to (n)
  (if (= n 0)
      0
      (+ n (sum-to (- n 1)))))

(sum-to 5)   ; gives 15
```

5. Every LISP expression is a list in brackets: the first item is the operation, the rest are its arguments. That is all the syntax there is.
6. Because a program is a list, a program can build and run another program with no special machinery. That property is **homoiconicity**.

15.8.4 PLAIN - what is really happening inside

1. COBOL's committee met at the Pentagon on 28 and 29 May 1959 with 41 attendees, chaired by Charles Phillips. An earlier meeting had been convened by Mary K. Hawes on 8 April 1959.
2. The goal was one language running on machines from every manufacturer, so the government would stop paying to rewrite the same payroll six times.
3. Grace Hopper was a technical adviser, not a committee member. Her FLOW-MATIC was the single strongest influence on the design.
4. Jean Sammet, who was on the committee, put the record straight bluntly: Hopper was not the mother, creator or developer of COBOL.
5. LISP started differently. McCarthy wrote a mathematical notation in a 1960 paper and did not intend it as a real language.
6. Steve Russell read the paper, realized the `eval` function described in it was itself an interpreter, and hand-compiled it into IBM 704 machine code. LISP became real by accident.
7. Two 704 assembly macros gave LISP two permanent words: `car` for the first item and `cdr` for the rest.
8. ALGOL's contribution is structural: blocks marked `begin` and `end`, names visible only inside the block where they are declared, and procedures that can call themselves.
9. If your language has curly brackets, local variables and recursion, it is an ALGOL descendant. C, Java, JavaScript, Python, Go and Rust all are.

15.8.5 TECHNICAL - the engineer's version

1. COBOL: COmmon Business-Oriented Language. The COBOL 60 specification was approved by the CODASYL executive committee on 8 January 1960. The first COBOL program ran on an RCA 501 on 17 August 1960.
2. In December 1960 the same COBOL program ran on both an RCA and a UNIVAC machine. That cross-vendor portability demonstration was the point of the whole exercise.

Figure	Value	Source and date
Lines of COBOL in use	200 billion	Gartner estimate, 1997
Business programs run	about 80%	Gartner estimate, 1997
Card swipes touching it	about 95%	widely cited, 2020

3. Treat newer figures with care. Claims such as "COBOL handles 3 trillion dollars of transactions a day" circulate widely and trace back to vendor material rather than an audited study.
4. The safe statement is that COBOL still runs core banking, insurance, payroll and government systems in 2026, and nobody has a credible plan to remove most of it.
5. LISP: John McCarthy, MIT, from 1958. Key paper: "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I", 1960. LISP introduced automatic garbage collection, dynamic typing, higher-order functions, the read-eval-print loop, and code as data.
6. LISP is the second-oldest high-level language still in use, after Fortran. Living dialects include Common Lisp, Scheme, Racket and Clojure.
7. ALGOL 58, originally called IAL, came from a Zurich meeting in 1958. ALGOL 60 came from a Paris meeting in January 1960 with 13 authors: Bauer, Naur, Rutishauser, Samelson, Vauquois, van Wijngaarden and Woodger from Europe, and Backus, Green, Katz, McCarthy, Perlis and Wegstein from the United States. Peter Naur edited the report.
8. The report's grammar notation, invented by Backus and refined by Naur, is **Backus-Naur Form**. It is still how language grammars are written in 2026.
9. Tony Hoare's verdict on ALGOL 60 is the most quoted line in language design: a language so far ahead of its time that it was an improvement not only on its predecessors but on nearly all its successors.

15.8.6 WORDS - remember these

1. COBOL — the English-like business language — CODASYL, 1959 to 1960, built on records, files and fixed-point decimal.
2. PICTURE clause — the description of a field's exact shape — COBOL's PIC notation defining digits, sign and implied decimal point.
3. LISP — the list language — McCarthy, 1958: s-expressions, garbage collection and code as data.
4. Homoiconicity — programs made of the same stuff as data — source represented directly in the language's own data structures.
5. ALGOL — the algorithm description language — ALGOL 58 and 60, source of block structure, lexical scope and recursion.
6. Backus-Naur Form — the standard way of writing a grammar — a metasyntax for context-free grammars, from the ALGOL 60 report.

15.9 The moth, and what a bug really is

15.9.1 PLAIN - in simple words

1. On 9 September 1947, operators of the Harvard Mark II found the machine giving wrong answers.
2. They traced the fault to relay number 70 in panel F, and found a moth caught between the contacts.
3. They removed the moth, taped it into the logbook, and wrote beside it: "First actual case of bug being found."
4. That logbook page, moth still attached, is at the Smithsonian's National Museum of American History.
5. It is a lovely story and it is almost always told wrongly.
6. The word "bug" for a fault did not start there. It was already old engineering slang.
7. Thomas Edison used it in writing in 1878, describing how little faults and difficulties, "as such little faults and difficulties are called", show themselves and take months to clear.
8. The joke in the logbook only works because everyone already used the word. For once they had found an actual, literal bug.
9. Grace Hopper was associated with the Mark II team and told the story for decades. Accounts differ on whether she was in the room, and the entry is generally attributed to others on the team.

15.9.2 PLAIN - a picture in your head

1. Imagine a workshop where people have called every annoying fault a "gremlin" for seventy years.
2. One day someone opens a machine and finds a small carved gremlin figure that a child dropped in.
3. They pin it to the wall and write "first actual gremlin found". Everyone laughs, because the word was already worn smooth.
4. A century later people say "that is where the word gremlin comes from", which reverses the joke completely.

Where this comparison breaks: unlike gremlins, insects really did cause failures in relay machines. Relay contacts were open, mechanical and warm, and buildings were not sealed. It was a genuine failure mode, not a one-off.

15.9.3 PLAIN - a worked example

1. Now the harder question. How did you debug in 1949, when there was no screen and printing a line cost machine time you did not have?
2. Method one: the console lights. A row of lamps showed the accumulator or the program counter in binary, and you read the bits.
3. Method two: single step. A switch made the machine execute one instruction per button press, so you could watch a register change.

4. Method three: the post-mortem routine. A small program left in memory that, after your program stopped, printed a chosen block of memory.
5. Method four: the checking routine, or trace. A program that ran yours one instruction at a time and printed what each did. Stanley Gill described this for EDSAC in a 1951 paper on diagnosing mistakes in programmes.
6. Method five: sound. EDSAC and the Ferranti Mark 1 had a loudspeaker wired to a bit of a register. Programmers learned the sound of a healthy program and heard a stuck loop instantly.
7. Method six: desk checking. You read your own code on paper, pretending to be the machine, writing register values in a column.
8. All six survive today, renamed: registers view, single step, core dump, instruction trace, profiling by observation, and code review.

15.9.4 PLAIN - what is really happening inside

1. A trace routine is really an interpreter. It fetches one of your instructions, prints it, performs it itself, and moves on.
2. A post-mortem dump is a different idea. It does not slow your program at all. It captures the wreckage afterwards.
3. That trade-off has never gone away. It is exactly the choice between running under a debugger and reading a crash dump.
4. Breakpoints were originally physical. You replaced an instruction in memory with a halt, ran until the machine stopped, read the lights, then put the original instruction back.
5. Modern debuggers do the identical trick. On x86 a breakpoint is the single byte CC, the INT 3 instruction, written over your code and restored when execution stops.

15.9.5 TECHNICAL - the engineer's version

1. Harvard Mark II: an electromechanical relay machine completed in 1947 for the United States Navy at Dahlgren, Virginia, built by Howard Aiken's group.
2. The 9 September 1947 log entry sits at 15:45 in the logbook. The relay is recorded as Relay 70, Panel F.
3. Earlier uses of "bug": Edison's 1878 letter; the 1896 edition of Hawkins' New Catechism of Electricity defines "bug" as a fault or trouble in an apparatus.
4. The term was standard in radio and telephony by the 1920s, and "debug" is documented in aviation in the 1940s, before the moth.
5. Terminology today, precisely: a **fault** is the defect in the code, an **error** is the wrong internal state it causes, and a **failure** is the externally visible wrong behaviour. That three-level split is standard in dependability engineering.
6. Tools that replaced the console lights: `gdb` and `lldb` for interactive debugging, `strace` and `dtrace` for system call tracing, `perf` for sampling, `valgrind` and AddressSanitizer for memory errors, and core dumps controlled by `ulimit -c` and `/proc/sys/kernel/core_pattern` on Linux.

The honest version: the popular claim "the word bug comes from a moth in a computer in 1947" is simply false, and the people who wrote the log entry knew it, which is exactly why they wrote "first actual case". The story is true; the etymology attached to it is not.

15.9.6 WORDS - remember these

1. Bug — a fault in a program — a defect in source or design; the word is engineering slang predating computing by at least 70 years.
2. Debugging — finding and removing faults — systematic fault localization, from observed failure back to defect.
3. Post-mortem dump — a printout of memory after a crash — a core dump, the captured process image at failure time.

4. Trace — a record of each step performed — an instruction-level execution log, historically produced by interpretation.
5. Breakpoint — a place where the machine stops for you — a trap instruction patched over the target instruction, restored on hit.
6. Fault, error, failure — the defect, the bad state, the visible wrongness — the standard three-level dependability vocabulary.

15.10 The rise of the operating system as software

15.10.1 PLAIN - in simple words

1. In the early 1950s there was no operating system, because there was nothing for one to do.
2. One program ran at a time, and a human being was the operating system.
3. The machine sat idle during all of that, and the machine was the expensive part.
4. So people wrote a small program that stayed in memory permanently and did the operator's job: read the next job, run it, clean up, read the next.
5. That resident program was called a **monitor**, and it is the ancestor of every operating system.
6. The next problem was worse. A program waiting for a tape to rewind left the processor idle for seconds at a time.
7. The answer was to keep several programs in memory and switch to another whenever the current one had to wait.
8. Then came the obvious next question: if we can switch between programs, we can switch between people. Give everyone a terminal and take turns. That is **time sharing**.

15.10.2 PLAIN - a picture in your head

1. Think of one very expensive oven in a village.
2. At first each family carries its dish in, lights the oven, cooks, and carries it out, and the oven cools between families.
3. Then the village hires a permanent baker who keeps the oven hot and takes dishes in a queue. That is the batch monitor.
4. Then the baker puts several dishes in at once, turning to whichever needs attention. That is multiprogramming.
5. Then the baker gives every family thirty seconds of attention in rotation, so each feels the oven is theirs. That is time sharing.

Where this comparison breaks: the oven's heat is genuinely shared, whereas a processor gives each program the whole machine for a short slice. The illusion of simultaneity comes from switching faster than a human can notice, not from dividing capacity.

15.10.3 PLAIN - a worked example

1. Consider a 1957 job on an IBM 704 that computes for 2 seconds.
2. Without a monitor: 4 minutes of operator handling, 2 seconds of computing. Machine utilization is under 1 per cent.
3. With a batch monitor and a stack of jobs on one tape: handling drops to under a second per job, and utilization rises above 50 per cent.
4. With multiprogramming: while job A waits 30 milliseconds for a disk, job B uses those 30 milliseconds, and utilization approaches 90 per cent.
5. With time sharing: 30 users each get slices of about 100 milliseconds, and each feels alone on the machine as long as none is doing heavy work.
6. The economics justify the complexity. A machine costing tens of thousands of dollars a month sitting idle at 1 per cent is a catastrophe, and that is why operating systems exist at all.

15.10.4 PLAIN - what is really happening inside

1. A monitor needs three abilities, and each one required new hardware.
2. It must regain control from a program that will not stop. That needs a **timer interrupt**: a clock that forces the processor into the monitor.
3. It must stop one program corrupting another. That needs **memory protection**: hardware that checks every address.
4. It must stop a program driving the tape drive directly. That needs **privileged mode**: some instructions work only for the monitor.
5. Those three inventions are the boundary between a program and an operating system. Without them, a monitor is only a convention programs may ignore.
6. The word "kernel" means exactly this: the part that runs in privileged mode and that everything else must go through.

15.10.5 TECHNICAL - the engineer's version

1. GM-NAA I/O, written in 1956 by Robert Patrick of General Motors Research and Owen Mock of North American Aviation for the IBM 704, is generally described as the first operating system.
2. It was followed by the FORTRAN Monitor System, then IBSYS on the IBM 7090 and 7094 family.
3. Job Control Language exists because a batch monitor must be told where one job ends and the next begins. IBM's JCL statements still start with two slashes in columns 1 and 2, a punched-card artefact.
4. CTSS, the Compatible Time-Sharing System, was first demonstrated by Fernando Corbato at MIT in November 1961 on an IBM 709, with three user consoles.
5. It moved to an IBM 7090 in 1962 and a 7094 later, entered routine service in summer 1963, and could support up to 112 teleprinter terminals through an IBM 7750 controller.
6. CTSS was the first system with password login, and carried some of the first electronic mail and text formatting tools.
7. Multics, from 1965, was the joint MIT, General Electric and Bell Labs successor. It introduced segmented virtual memory, a hierarchical file system, dynamic linking and rings of protection.
8. Multics was late, huge and slow to arrive, and Bell Labs withdrew in 1969. That withdrawal directly caused UNIX, which is the next section.

System	Year	Idea it added
GM-NAA I/O	1956	resident batch monitor
IBSYS	1960	full job stream management
CTSS	1961	interactive time sharing
Multics	1965	virtual memory, protect rings

9. Chapter 18 takes this up properly: scheduling, virtual memory, system calls and modern kernels. Here we needed only the reason the operating system had to become software in the first place.

15.10.6 WORDS - remember these

1. Monitor — the small permanent program that runs jobs — the resident supervisor of a batch system.
2. Batch — jobs run back to back with no user present — non-interactive scheduling from a job queue.
3. Multiprogramming — several programs in memory, one running — overlapping input and output with computation.
4. Time sharing — many people using one machine at once — preemptive scheduling with short quanta over interactive terminals.
5. Kernel — the privileged core of the operating system — code executing in supervisor mode, mediating hardware access.
6. Privileged mode — an instruction level only the system may use — supervisor or ring 0 execution enforced by the processor.

15.11 UNIX and C

15.11.1 PLAIN - in simple words

1. In 1969 Bell Labs pulled out of Multics, and Ken Thompson had a large, ambitious operating system project taken away from him.
2. He had written a game called Space Travel and wanted somewhere to run it. He found a little-used PDP-7 in a corner.
3. With Dennis Ritchie and others, and with a month of free time while his wife was away, he wrote a tiny operating system for it.
4. It had a hierarchical file system, processes and a small set of tools. It was called Unics as a joke on Multics, and the spelling settled as UNIX.
5. UNIX was written in assembly at first, like everything else.
6. Then came the radical part. Thompson wrote a language called B, and Ritchie grew it into a language called C between 1971 and 1973.
7. In 1973 they rewrote the UNIX kernel itself in C.
8. Nobody did that. An operating system had to be assembly, because it touches hardware and must be fast.
9. But it worked, and the consequence was enormous: to move UNIX to a new computer you now wrote a C compiler for it, not a whole new operating system.
10. That is where portable software begins.

15.11.2 PLAIN - a picture in your head

1. Imagine every city building its trains to a different rail gauge, so every train must be rebuilt from scratch to move city to city.
2. Now someone builds an adjustable axle. The carriage design stays the same; only the axle is refitted.
3. C is the adjustable axle. The compiler is refitted per machine, and the software above it moves across.
4. The carriage is not perfectly identical everywhere. Some of it still has to be adjusted by hand, and the parts touching the rails always do.

Where this comparison breaks: C's portability is not automatic, it is a discipline. C deliberately leaves things such as integer size and byte order unspecified, and code assuming one machine's answers is not portable, no matter that it is written in C.

15.11.3 PLAIN - a worked example

1. Here is the same "add 1 to n" job from earlier, written in C, next to what a compiler produces for it.

```
int sum_to(int n) {
    int total = 0;
    for (int i = 1; i <= n; i++)
        total += i;
    return total;
}
```

2. Below is the x86-64 assembly from GCC 13.3.0 with optimization turned off, in Intel syntax. Six lines of C become nineteen instructions.

```
sum_to:
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR -20[rbp], edi    ; save n
        mov     DWORD PTR -8[rbp], 0       ; total = 0
        mov     DWORD PTR -4[rbp], 1       ; i = 1
        jmp     .L2
.L3:
```

```
        mov     eax, DWORD PTR -4[rbp]     ; load i
        add     DWORD PTR -8[rbp], eax     ; total += i
        add     DWORD PTR -4[rbp], 1       ; i++
.L2:
        mov     eax, DWORD PTR -4[rbp]     ; load i
        cmp     eax, DWORD PTR -20[rbp]    ; compare with n
        jle     .L3                        ; loop if i <= n
        mov     eax, DWORD PTR -8[rbp]     ; return total
        pop     rbp
        ret
```

3. Notice `.L2` and `.L3`. Those are labels, exactly the idea from the 6502 assembly earlier in this chapter. The compiler invents them, so no human counts bytes.
4. Notice the loop is compiled with the test at the bottom and one jump into it. That shape is a choice the compiler made, not something you wrote.
5. Turn optimization on and the function changes completely: the compiler unrolls the loop and computes two values per pass. That is the compiler choosing, which an assembler never does.
6. You can see this on any Linux machine with `gcc -S -masm=intel -O0 file.c` and then `gcc -S -masm=intel -O2 file.c`.

15.11.4 PLAIN - what is really happening inside

1. C was designed to sit exactly one small step above the machine, and no higher.
2. Its types map onto machine word sizes, its pointers are memory addresses, and its arrays are pointer arithmetic in disguise.
3. That closeness is why a C compiler is small and easy to write for a new processor, and why UNIX could be moved.
4. It is also why C is dangerous. There is no bounds checking, because the machine has none, and the language does not add what the machine lacks.
5. The 1973 rewrite made the UNIX kernel about a third larger and somewhat slower than the assembly version, and Thompson and Ritchie accepted that trade deliberately.
6. They got back something worth far more: they could read their own kernel, change it safely, and move it.
7. In 1978 Brian Kernighan and Dennis Ritchie published "The C Programming Language". The book was so precise and so short that it became the definition of the language for over a decade, known simply as K and R.

15.11.5 TECHNICAL - the engineer's version

1. Timeline: Bell Labs left Multics in 1969; UNIX started on a PDP-7 in 1969; moved to a PDP-11/20 in 1970; C developed from B and BCPL through 1971 and 1972; Version 4 UNIX rewritten in C in 1973.
2. The paper by Ritchie and Thompson, presented at the ACM Symposium on Operating Systems Principles in 1973 and published in Communications of the ACM in July 1974, is the announcement that made UNIX famous.
3. Version 6 UNIX, released in 1975, was licensed cheaply to universities. John Lions's 1976 commentary on its source code was for years the most photocopied document in computing.
4. UNIX's design rules became a culture: everything is a file, programs do one thing, programs are joined by pipes, and text is the universal interface.
5. C standards: K and R C from 1978, ANSI X3.159-1989 (C89), ISO/IEC 9899:1990 (C90), then C99, C11, C17, and C23 published in 2024.

Year	Event	People
1969	Bell Labs leaves Multics	Bell Labs

Year	Event	People
1969	UNIX begins on a PDP-7	Thompson, Ritchie
1972	C takes shape	Ritchie
1973	UNIX kernel rewritten in C	Thompson, Ritchie
1978	The C Programming Language	Kernighan, Ritchie

6. Thompson and Ritchie received the ACM Turing Award in 1983.
7. As of 2026 the direct descendants of this work include Linux, the BSDs, macOS and iOS (through NeXTSTEP and BSD), and Android's userspace model. The POSIX standards, IEEE 1003, are the written form of the UNIX interface.

The honest version: C is not portable in the sense people usually mean. The standard deliberately leaves behaviour unspecified, undefined or implementation-defined in hundreds of places. What C gives you is a portable language with a small compiler, not portable programs for free.

15.11.6 WORDS - remember these

1. UNIX — the small operating system from Bell Labs, 1969 — a multi-user, multi-process system whose interface is standardized as POSIX.
2. C — the language UNIX was rewritten in, 1972 — a statically typed systems language with direct memory access and manual lifetime management.
3. Portability — software that moves between machines — source compatibility given a conforming compiler and defined behaviour.
4. The 1973 rewrite — the moment an operating system stopped being assembly — Version 4 UNIX in C, trading size and speed for maintainability.
5. K and R — the 1978 book that defined C — Kernighan and Ritchie, "The C Programming Language", the pre-standard reference.

15.12 How software was shared, and then sold

15.12.1 PLAIN - in simple words

1. For the first fifteen years, software was not sold. It came free with the computer, because the computer was the product.
2. Users swapped programs directly. IBM's user group SHARE, founded in 1955, ran a library that members contributed to and drew from.
3. In 1969 IBM changed that. Facing antitrust pressure, it announced it would price software separately from hardware. That decision, effective in 1970, is where the software industry begins.
4. When home computers arrived in the mid-1970s, sharing came back, because the people involved were hobbyists with no money.
5. Magazines printed the full source code of programs, and you typed them in by hand, then hunted for your own typing mistakes.
6. The Homebrew Computer Club, first meeting on 5 March 1975 in a garage in Menlo Park, California, was the centre of this. Members copied paper tapes and passed them round freely.
7. In January 1976 a 20-year-old Bill Gates wrote an angry letter to that community, saying most of them were stealing his BASIC and that this would stop anyone writing good software.
8. He was making an argument nobody had needed to make before: that software by itself is a product you buy.
9. Later, others argued the opposite just as forcefully, and built an entire parallel world on it.

15.12.2 PLAIN – a picture in your head

1. Think of recipes. For centuries they spread freely; cooks copied them, improved them and passed them on.
2. Then someone opens a restaurant and says the recipe is the business, not the meal, and copying it is theft.
3. Some cooks agree and buy licences. Others say a recipe you may not read or change is not a recipe at all, and start a movement to keep them open.
4. Both worlds now exist side by side, and most restaurants use some of each.

Where this comparison breaks: recipes are not protected by copyright in most places, whereas software source code clearly is. The legal ground under the argument is completely different, and that is why licences, not manners, ended up deciding it.

15.12.3 PLAIN – a worked example

1. Here is what "sharing software" physically meant, in order of era.
2. 1955 to 1968: a magnetic tape posted to a user group, and a printed catalogue of routines you could request.
3. 1975 to 1979: a punched paper tape passed hand to hand at a club meeting, or a program listing printed in a magazine that you typed in yourself.
4. 1978 to 1990: a floppy disk, posted or copied at a club, and later a bulletin board system you dialled with a modem at 300 or 1,200 bits per second.
5. 1991 onwards: an FTP site, then the web, then a version control server.
6. 2008 onwards: a public git repository, which is where nearly all of it lives in 2026.
7. The pattern is constant. Each drop in the cost of copying produced a burst of sharing, and a matching argument about payment.

15.12.4 PLAIN – what is really happening inside

1. Gates's letter is worth understanding rather than cheering or booing.
2. The facts he cited: MITS was shipping about 1,000 Altair computers a month at the end of 1975, while paid copies of BASIC numbered in the low hundreds.
3. He said about 40,000 dollars of computer time went into writing it, and that the royalty worked out at roughly two dollars an hour for the work.
4. Critics answered that the computer time figure was inflated accounting, and that Microsoft's BASIC had itself been developed on a university machine.
5. Both things can be true. The letter is still the clearest early statement that software has a development cost that must be paid for somehow.
6. In 1983 Richard Stallman made the opposite argument, prompted by a printer whose driver he was not allowed to fix.
7. His answer was not to ask nicely. It was a licence that used copyright law against itself: you may copy and change this, provided anything you distribute carries the same permission.
8. That trick is called **copyleft**, and it is the mechanism, not the slogan, that made free software durable.

15.12.5 TECHNICAL – the engineer's version

1. IBM's unbundling announcement came on 23 June 1969 and took effect in January 1970. Before it, software and support were included in the hardware price; after it, they were priced separately.
2. Homebrew Computer Club: first meeting 5 March 1975, Gordon French's garage, Menlo Park. The Altair 8800 had appeared on the cover of Popular Electronics dated January 1975.
3. A pre-release paper tape of Altair BASIC went missing at a MITS event in Palo Alto in June 1975; Dan Sokol duplicated it, and about 50 copies appeared at the next Homebrew meeting.

4. Gates wrote "An Open Letter to Hobbyists" in January 1976. It was published in the Homebrew Computer Club Newsletter dated 3 February 1976 and in MITS Computer Notes for February 1976, and reprinted widely.
5. Richard Stallman announced the GNU Project, a recursive joke standing for "GNU's Not Unix", in a Usenet message on 27 September 1983. Development began January 1984, the GNU Manifesto appeared in 1985, and the Free Software Foundation was founded in October 1985.

Licence	Year	Core idea
GPL version 1	1989	copyleft, source must follow
GPL version 2	1991	the Linux kernel licence
GPL version 3	2007	patents and locked hardware
MIT / BSD	1980s	permissive, no copyleft

6. The four freedoms of free software: run the program for any purpose, study and change it, redistribute copies, and distribute your modified versions. Access to source code is a precondition of the second and fourth.
7. Linus Torvalds posted his announcement to the comp.os.minix newsgroup on 25 August 1991, describing a free operating system as "just a hobby, won't be big and professional like gnu". Version 0.01 followed on 17 September 1991.
8. Linux moved to the GPL with version 0.12 in February 1992 and reached version 1.0 in March 1994. Combining the Linux kernel with the GNU tools produced the first complete free operating system.
9. The term "open source" was coined at a meeting in Palo Alto on 3 February 1998, following Netscape's decision to release its browser source. The Open Source Initiative was founded that month.
10. Free software and open source are not synonyms, and the disagreement is real rather than pedantic. Free software is an ethical argument about user freedom. Open source is an argument about development quality and business practicality. They mostly cover the same code.

■ 15.12.6 WORDS - remember these

1. Unbundling — selling software separately from hardware — IBM's 1969 announcement, the origin of the commercial software industry.
2. Copyleft — a licence that forces the freedom onward — a reciprocal licence requiring derivative distributions under the same terms.
3. GPL — the GNU General Public License — the reciprocal licence family, versions 1 (1989), 2 (1991) and 3 (2007).
4. Permissive licence — you may do almost anything — MIT and BSD style terms with attribution but no reciprocity.
5. Free software — free as in freedom, not price — the four freedoms defined by the Free Software Foundation.
6. Open source — publicly readable and modifiable code — the Open Source Initiative definition, from 1998.

15.13 A short history of the idea of a "programmer"

■ 15.13.1 PLAIN - in simple words

1. Before 1945, "computer" was a job title for a person, usually a woman, who did calculations with a desk machine.
2. The first people to program electronic machines were drawn from that pool and were called operators, not programmers.
3. The work was thought of as clerical, because the design of the machine was assumed to be the intellectual part.

4. That was badly wrong, and it took about fifteen years to correct.
5. Through the 1950s the word "programmer" settled in, and "coder" came to mean a junior who turned someone else's plan into instructions.
6. By the mid-1960s, projects had grown so large that they routinely failed. Systems were years late, far over budget, and full of faults.
7. In 1968 a NATO conference gave that a name: the **software crisis**. It also made popular a deliberately provocative term for the cure: **software engineering**.
8. The provocation was the point. If bridges can be engineered with method and discipline, why not programs.
9. Almost sixty years later the argument is unfinished, and it is still the most useful question in the field.

15.13.2 PLAIN - a picture in your head

1. Think of building. First there were people who put up a shed, and it stood up or it did not.
2. Then buildings got taller, and a shed-builder's instincts stopped scaling. People fell.
3. So the trade split: architects, structural engineers, surveyors, inspectors, codes of practice and licences.
4. Software went through the first two stages and got stuck partway through the third.

Where this comparison breaks: a bridge's load is physics and does not argue. A program's requirements change every week, and it is expected to be modified constantly for decades. No civil engineer is asked to change the span of a finished bridge on a Thursday.

15.13.3 PLAIN - a worked example

1. The example everyone in 1968 had in mind was IBM's OS/360, the operating system for the System/360 family announced in 1964.
2. It ran to thousands of people and years of delay, and adding people made it later, not earlier.
3. Fred Brooks, who managed it, wrote up the lessons in "The Mythical Man-Month" in 1975.
4. His central observation: a task that takes 12 months for 1 person does not take 1 month for 12 people, because they must all talk to each other.
5. With n people there are n times (n minus 1) divided by 2 communication paths. For 12 people that is 66. For 50 people it is 1,225.
6. Brooks's law, in his own compressed form: adding people to a late software project makes it later.
7. That is the sharpest single sentence about software management ever written, and it is still true in 2026.

15.13.4 PLAIN - what is really happening inside

1. The 1968 conference proposed answers that are now everyday practice: specification before coding, modular design, reviews, testing as a distinct activity, and measurement.
2. Alongside it, a related movement argued that programs should be structured so a human can reason about them.
3. Edsger Dijkstra's letter "Go To Statement Considered Harmful", published in Communications of the ACM in March 1968, is the famous shot in that campaign.
4. His argument was not aesthetic. It was that with unrestricted jumps you cannot look at a line of code and know how you got there, so you cannot reason about the program at all.
5. What replaced the goto was exactly the block structure ALGOL 60 had introduced: sequence, choice and loops with one way in and one way out.
6. So the theory and the management story meet here. Structured programming was a response to a management crisis.

15.13.5 TECHNICAL - the engineer's version

1. The earliest known use in print of "software" in the modern sense is by John Tukey, in "The Teaching of Concrete Mathematics", American Mathematical Monthly, January 1958.

2. The NATO Software Engineering Conference was held at Garmisch, Germany, from 7 to 11 October 1968, with roughly 50 participants. A second conference followed at Rome in October 1969.
3. Who coined "software engineering" is genuinely disputed. Margaret Hamilton, who led the Apollo onboard flight software at the MIT Instrumentation Laboratory, states she began using the term during Apollo in the mid-1960s.
4. Anthony Oettinger used it in an ACM president's letter in 1966, and the 1968 NATO conference made it general.
5. All three claims can be accurate at once, and the safe statement is that the 1968 conference established the term.
6. Dijkstra's 1972 Turing Award lecture summarized the crisis: with no machines there was no programming problem, with weak machines a mild one, and with powerful machines an equally gigantic one.
7. A 1968 study by Sackman, Erikson and Grant reported differences of an order of magnitude or more between individual programmers on the same task.
8. The headline "10x programmer" comes from that work. Its method has been criticized ever since, so the size of the effect is active debate rather than established fact.

Year	Term or event	Significance
1946	"operator"	programming seen as clerical
1958	"software" in print	Tukey, AMM, January 1958
1968	NATO Garmisch	"software engineering" named
1968	Go To Considered Harmful	structured programming
1975	The Mythical Man-Month	Brooks, on team scaling

The honest version: software engineering is still not engineering in the sense that civil engineering is. There is no universal licensure, no legally binding code of practice in most jurisdictions, and no agreed body of predictive knowledge saying how long a given program will take. Experts genuinely disagree on whether that is a failing to be fixed or a reflection of a different kind of work.

15.13.6 WORDS - remember these

1. Software — the instructions, not the machine — programs and data as distinct from hardware; earliest known print use, Tukey, 1958.
2. Software crisis — projects late, costly and broken — the recognition, named in 1968, that project growth outran available method.
3. Software engineering — treating programming as a disciplined craft — the application of engineering method to software, named at NATO 1968.
4. Brooks's law — adding people to a late project makes it later — the communication overhead of n(n-1)/2 channels swamps added capacity.
5. Structured programming — code shaped so you can reason about it — single entry, single exit constructs replacing arbitrary jumps.

15.98 Common wrong ideas

1. Wrong: a program is a special kind of thing stored differently from data. Right: it is ordinary numbers in ordinary memory, treated as instructions only because the program counter points at them.
2. Wrong: Grace Hopper wrote the first compiler as we use the word today. Right: she coined the word and built A-0 in 1952, which selected, relocated and joined library routines. That is a linking loader, not a translator.
3. Wrong: nothing existed before FORTRAN. Right: Corrado Bohm's 1951 thesis described a translator on paper, Glennie's Autocode ran in 1952, and Laning and Zierler's algebraic system ran on Whirlwind by 1954.

4. Wrong: the word "bug" comes from the 1947 moth. Right: Edison used it in 1878, and the logbook entry says "first actual case" precisely because the word was already old slang.
5. Wrong: ENIAC took two weeks to reprogram, full stop. Right: planning took weeks, physical setup days and debugging days more, and the 1948 conversion changed the whole picture by cutting setup to hours.
6. Wrong: assembly language is always one line to one instruction. Right: macros, pseudo-instructions and assembler-chosen branch widths break that on most real assemblers.
7. Wrong: COBOL is obsolete. Right: it still runs core banking, insurance and government payroll in 2026, largely because its exact decimal arithmetic is correct for money in a way binary floating point is not.
8. Wrong: high-level languages won because programmers wanted comfort. Right: they won when FORTRAN's optimizer made the generated code fast enough that the speed objection collapsed.
9. Wrong: C is portable, so C programs run anywhere. Right: C is a portable language with a small compiler. Programs depending on integer size, byte order or undefined behaviour are not portable at all.
10. Wrong: free software means free of charge. Right: it means the four freedoms to run, study, share and modify. You may sell it, and the Free Software Foundation says so explicitly.

15.99 Chapter summary in 20 lines

1. A program at the bottom is a list of numbers in memory that a processor reads as instructions, one after another.
2. Seven bytes on a 6502 make a complete working program, and you can trace it by hand with nothing but an opcode table.
3. ENIAC in 1946 had no program storage at all. You wired it with cables and set 1,200-way switch banks, and a change took days.
4. Its six principal programmers, McNulty, Jennings, Snyder, Wescoff, Bilas and Lichterman, were classified as operators and taught themselves the machine.
5. The 1948 conversion gave ENIAC a stored instruction set and cut setup from days to hours at about a sixfold speed cost.
6. Programs then went in as raw numbers: front-panel switches, punched paper tape, and above all the 80-column punched card.
7. A card holds 12 rows by 80 columns; a digit is one punch, a letter is a zone punch plus a digit punch, and that is Hollerith encoding.
8. Batch processing meant handing a deck to an operator and waiting hours or a day for a printout, so programmers desk-checked everything first.
9. Assembly language replaced numbers with mnemonics and hand-counted addresses with labels, and an assembler recomputed the numbers every time.
10. EDSAC's Initial Orders by David Wheeler in 1949, the Wheeler Jump, the 87-routine library, and the 1951 Wilkes, Wheeler and Gill book built the idea of reusable software.
11. Assembly is tied to one machine, slow to write and barely checked, so people asked whether a program could write the machine code instead.
12. The objection was economic, not snobbish: machine time cost far more than programmer time, so a translator had to lose almost no speed.
13. Grace Hopper coined "compiler" and built A-0 in 1952. It gathered subroutines from a tape library, relocated them and joined them, which today we call linking and loading.
14. Bohm's 1951 thesis, Glennie's 1952 Autocode and the 1954 Laning-Zierler system all have claims to "first compiler" under different definitions.
15. FORTRAN, from Backus's team at IBM, ran from 1954 to delivery in April 1957, and won because its optimizer matched hand-written assembly closely enough for scientists to accept it.
16. COBOL came from the CODASYL committee in 1959 with FLOW-MATIC as its main ancestor, and its exact decimal arithmetic keeps it running finance in 2026.

17. LISP in 1958 gave us code as data, garbage collection and recursion, and ALGOL 60 gave nearly every later language its block structure.
18. The moth of 9 September 1947 is real and is in the Smithsonian, but the word “bug” was already 70 years old when it was taped in.
19. Batch monitors from 1956, then CTSS time sharing in 1961 and Multics, made the operating system a permanent program that manages other programs.
20. UNIX from 1969 and its 1973 rewrite in C made operating systems portable, and everything since, from Linux in 1991 to the machine in front of you, sits on that foundation.

16

CHAPTER

How a Compiler Actually Works

16.0 What this chapter gives you

1. You will be able to name every stage between the text you type and the electricity that runs, in the right order, and say what each one does.
2. You will be able to take one five-word C function and follow it through preprocessing, tokens, tree, symbol table, intermediate code, optimization, assembly, object file, executable and running process.
3. You will be able to read real assembly output and explain every single line.
4. You will be able to say why compilers use an intermediate language in the middle instead of going straight to machine instructions.
5. You will be able to explain why `-O2` turned our program into three instructions, and why that is legal.
6. You will be able to explain what a linker does, why "undefined reference" happens, and why the order of libraries on the command line matters.
7. You will be able to explain what happens at the moment you press Enter on a program, including the dynamic linker, the GOT and the PLT.
8. You will be able to compare compiled, bytecode and interpreted execution with real numbers, and explain why JavaScript got fast.
9. You will be able to explain the bootstrap problem and Ken Thompson's 1984 trusting-trust attack to somebody who has never programmed.
10. You will be able to run the commands yourself and see the same output.

Our one running example, for the whole chapter, is this file. We will call it `add.c`. Everything below is that file at a different moment of its life.

```
int add(int a, int b) { return a + b; }
int main(void) { return add(2, 3); }
```

Every command shown was run on Ubuntu 24.04 on an x86-64 machine, with GCC 13.3.0 and Clang 18.1.3. The outputs pasted here are the real outputs.

16.1 The whole pipeline in one view

16.1.1 PLAIN – in simple words

1. A computer chip does not understand the words you type. It understands numbers, and only a small fixed set of them.
2. A **compiler** is a program that reads your text and writes those numbers.
3. It does not do this in one jump. It does it in a chain of small steps.
4. Each step takes one shape of information and turns it into a simpler shape.
5. First the text is cleaned up and glued together with other files.
6. Then it is chopped into small pieces, like words in a sentence.

7. Then those pieces are arranged into a tree, the way a sentence has a subject and an object.
8. Then the meaning is checked: do these names exist, do the types match.
9. Then it is rewritten in a simple made-up language that is easy to improve.
10. Then it is improved: shortened, simplified, things that do nothing removed.
11. Then it is written as instructions for one particular chip.
12. Then those instructions become raw bytes in a file.
13. Then several such files are glued into one program.
14. Then the operating system loads it into memory and starts it.

16.1.2 PLAIN - a picture in your head

1. Think of a factory line that turns a handwritten recipe into a packed meal.
2. Station one gathers all the recipe pages that the main page refers to.
3. Station two reads the pages aloud, one word at a time.
4. Station three works out the grammar: which words are the ingredients, which are the actions, what belongs inside what.
5. Station four checks it makes sense: you cannot fry a number.
6. Station five rewrites the whole thing as a plain numbered work order.
7. Station six removes wasted steps: no need to boil water you never use.
8. Station seven writes the work order in the exact words this kitchen's machines accept.
9. Station eight packs it in a box. Station nine puts several boxes in a crate.
10. Station ten unpacks the crate onto a table and starts cooking.

Where this comparison breaks: a factory line moves one item forward and never looks back. A real compiler often loops. The optimizer runs dozens of passes over the same code, some of them many times, and a later pass can undo an earlier one. Also, several stations can be the same piece of code in memory, not separate programs. The clean chain is a teaching model, not the shape of the source code inside GCC or Clang.

16.1.3 PLAIN - a worked example

Here is the chain, drawn for `add.c`.

```
 add.c   (the text you typed)
   |
   v
[ preprocessor ]  glue in #include, expand #define
   |
   v
[ lexer ]         text -> tokens: int, add, (, int, a, ...
   |
   v
[ parser ]        tokens -> a tree of the program
   |
   v
[ semantic ]      names, scopes, types, symbol table
   |
   v
[ IR builder ]    tree -> simple three-address code
   |
   v
[ optimizer ]     fold, propagate, inline, delete
   |
   v
[ code generator ] IR -> assembly text for one CPU
   |
   v
```

```
[ assembler ]     assembly -> object file add.o (bytes)
   |
   v
[ linker ]        add.o + C library -> one executable
   |
   v
[ loader ]        exec: map into memory, bind symbols
   |
   v
 a running process
```

One line for each stage, for `add.c`:

1. Preprocessor: nothing to do here, there are no # lines. Output is the same two lines of text.
2. Lexer: produces 32 tokens plus an end marker.
3. Parser: produces a tree with two function declarations at the top.
4. Semantic analysis: records that `add` takes two `int` and returns `int`, and that the call in `main` matches.
5. IR: produces `_3 = a + b; return _3;` for `add`.
6. Optimizer at `-O2`: notices `add(2,3)` is always 5.
7. Code generator: writes `leal (%rdi,%rsi), %eax` for `add`.
8. Assembler: writes the bytes `8d 04 37` for that instruction.
9. Linker: fills in the address of `add` inside `main`'s call instruction.
10. Loader: maps the file into memory and jumps to the entry point.
11. The process returns 5. `echo $?` prints 5.

16.1.4 PLAIN - what is really happening inside

1. Most of these "stages" are functions inside one program, passing data structures to each other, not separate processes.
2. But some really are separate programs. On Linux, `gcc` is only a driver: it runs other programs and passes files between them.
3. Running `gcc -v add.c` shows the driver's real children. The three that matter are `cc1`, `as` and `collect2`.
4. `cc1` is the actual C compiler. It reads `add.c` and writes assembly text to a temporary file.
5. `as` is the assembler. It reads that text and writes a temporary `.o` file.
6. `collect2` is a wrapper around `ld`, the linker. It writes the executable.
7. So on a normal Linux box, compiling one C file starts three extra programs.
8. Each stage narrows what is possible. After lexing, you can no longer ask about spaces. After parsing, you can no longer ask about brackets. After code generation, you can no longer ask about variable names.
9. That narrowing is the point. Every stage throws away information the next stage does not need, so the next stage can be simpler.

16.1.5 TECHNICAL - the engineer's version

1. The classical decomposition is **front end**, **middle end**, **back end**. The front end is language specific, the back end is target specific, and the middle end is neither.
2. The front end covers preprocessing, lexical analysis, syntax analysis and semantic analysis, and emits an intermediate representation (IR).
3. The middle end runs target-independent optimization passes on the IR.
4. The back end performs instruction selection, instruction scheduling and register allocation, and emits assembly or machine code directly.
5. The canonical reference is *Compilers: Principles, Techniques, and Tools* by Alfred Aho, Ravi Sethi and Jeffrey Ullman, 1986, known as the Dragon Book. The 2006 second edition adds Monica Lam.

6. The first true optimizing compiler was the IBM FORTRAN compiler for the IBM 704, delivered in April 1957 under John Backus. It took about 18 staff-years and its output had to beat hand-written assembly to be accepted.
7. Grace Hopper's A-0 system of 1952 predates it but was closer to a linking loader than to a modern compiler.
8. GCC 1.0 was released by Richard Stallman in March 1987. LLVM began as Chris Lattner's work at the University of Illinois, with LLVM 1.0 in 2003.

Observing the pipeline with real commands:

```
gcc -v add.c -o prog      # show cc1, as, collect2
gcc -E add.c              # stop after the preprocessor
clang -Xclang -dump-tokens -fsyntax-only add.c
clang -Xclang -ast-dump  -fsyntax-only add.c
clang -S -emit-llvm add.c -o add.ll
gcc -S add.c -o add.s     # stop after code generation
gcc -c add.c -o add.o     # stop after the assembler
```

Stage	GCC name	Clang / LLVM name
Preprocess	cpp inside cc1	clang -E
Parse + types	cc1 front end	clang front end
Middle IR	GIMPLE, then RTL	LLVM IR
Optimize	tree and RTL passes	LLVM pass manager
Emit assembly	cc1 back end	LLVM target backend
Assemble	GNU as	integrated assembler
Link	ld via collect2	lld or ld

16.1.6 WORDS - remember these

Compiler — a program that turns your text into machine numbers — a translator from a source language to a target language, usually with static checking. Pass — one walk over the program — a single traversal of the IR performing one analysis or transformation. Front end — the part that understands your language — lexing, parsing, semantic analysis, IR generation. Back end — the part that knows the chip — instruction selection, scheduling, register allocation, code emission. Driver — the program you actually type — `gcc` or `clang`, which orchestrates the real compiler, assembler and linker as child processes.

16.2 The preprocessor

16.2.1 PLAIN - in simple words

1. Before the compiler looks at your program, another tool edits the text.
2. That tool is the **preprocessor**. It only understands lines starting `#`.
3. `#include "file.h"` means: delete this line and paste that whole file here.
4. `#define TWO 2` means: everywhere the word `TWO` appears later, write `2`.
5. `#if` and `#ifdef` mean: keep this block of text or throw it away.
6. That is all it does. It moves text around. It does not know what C is.
7. It does not know types. It does not know functions. It cannot count.
8. The compiler never sees your `#include` lines. It sees the result.

16.2.2 PLAIN - a picture in your head

1. Imagine a school essay where you are told to write `see page 40` instead of copying a long definition.
2. Before the teacher marks it, a helper goes through and physically pastes page 40 in place of every `see page 40`.

3. The helper also has a list of shorthands: wherever you wrote `WHO`, write `World Health Organization`.
4. The helper does not read for meaning. If you wrote `see page 40` inside a joke, page 40 gets pasted into the joke.
5. The teacher then marks one long essay with no shortcuts left in it.

Where this comparison breaks: the helper in this story is careful. The C preprocessor is not. It will happily paste text that produces nonsense, and it will report the error at the pasted line, not at your line. Macro expansion also happens repeatedly until nothing changes, which no human helper would do.

16.2.3 PLAIN – a worked example

Split our example into a header and a source file.

```
/* addh.h */
#ifndef ADD_H
#define ADD_H
#define TWO 2
#define THREE 3
int add(int a, int b);
#endif
```

```
/* main2.c */
#include "addh.h"
int main(void) { return add(TWO, THREE); }
```

Now run `gcc -E main2.c`. The real tail of the output is:

```
# 1 "main2.c"
# 1 "addh.h" 1

int add(int a, int b);
# 2 "main2.c" 2
int main(void) { return add(2, 3); }
```

1. The `#include` line has gone. The declaration from the header sits in its place.
2. `TWO` and `THREE` have become `2` and `3`.
3. The `#ifndef` guard lines have gone. They produced no text.
4. The lines beginning `#` with numbers are **line markers**. They tell the compiler "the next line really came from file X, line N", so error messages point at your file and not at the merged text.
5. The blank lines are the preprocessor keeping line numbers lined up.

Now a real size measurement, on the same machine:

```
$ printf '#include <stdio.h>\nint main(void){return 0;}\n' > hello.c
$ wc -l < hello.c
2
$ gcc -E hello.c | wc -l
815
$ gcc -E hello.c | wc -c
21292
```

1. Two lines of yours became 815 lines and about 21 kB of text.
2. It pulled in 29 distinct files.
3. In C++ it is far worse. `#include <iostream>` with GCC 13.3 on this machine expanded to 36,584 lines.

16.2.4 PLAIN - what is really happening inside

1. The preprocessor runs in phases defined by the C standard. The important ones, in order, are: join lines ending in a backslash, replace comments with a single space, then execute the # directives, then join adjacent string literals.
2. The **header guard** trick, `#ifndef ADD_H` / `#define ADD_H` / `#endif`, exists because a header may be included twice through different paths.
3. Without a guard, the second include pastes the same declarations again, which is an error for anything defined rather than merely declared.
4. With the guard, the second visit finds `ADD_H` already defined and skips the whole file, producing no text.
5. `#pragma once` does the same job in one line. It is not in the C or C++ standard, but every major compiler supports it. That makes it a convention, not a standard.
6. Conditional compilation, `#if defined(_WIN32)`, is how one source file serves several operating systems. The text for the other systems never reaches the compiler at all.
7. Macros are textual, so they have famous traps. `#define SQ(x) x*x` then `SQ(1+2)` becomes `1+2*1+2`, which is 5, not 9. Parentheses fix it: `#define SQ(x) ((x)*(x))`.

16.2.5 TECHNICAL - the engineer's version

1. The C preprocessor is specified in the C standard, clause 6.10. It is a standard, not an implementation detail.
2. Translation phases 1 to 4 of the standard cover trigraph mapping, line splicing, comment removal and directive execution.
3. Predefined macros include `__FILE__`, `__LINE__`, `__DATE__`, `__TIME__`, `__STDC__` and `__STDC_VERSION__`. `__STDC_VERSION__` is `199901L` for C99, `201112L` for C11, `201710L` for C17 and `202311L` for C23.
4. `#include <x.h>` searches the system include path. `#include "x.h"` searches the current directory first, then the system path. `gcc -I dir` prepends a directory. `gcc -E -v` prints the search list actually used.
5. Macro expansion is not recursive on the macro currently being expanded, which is what stops `#define X X` from looping forever.
6. # in a function-like macro stringizes its argument. ## pastes tokens together. Both are standard.
7. Preprocessing cost is real. Header expansion is the main reason C++ builds are slow, and is the motivation for precompiled headers and for C++20 modules, standardized in ISO C++20 and still only partially implemented in toolchains as of 2026.

Command	What it shows
`gcc -E f.c`	Preprocessed text
`gcc -E -P f.c`	Same, without line markers
`gcc -dM -E f.c`	Every macro defined
`gcc -H -c f.c`	Header include tree
`gcc -MMD -c f.c`	Write a `.d` dependency file

The honest version: saying "the preprocessor is a separate program" is a useful lie. It once was, as `/lib/cpp`. In GCC today it is a library linked into `cc1`, and Clang implements it as a token source feeding the lexer, so text is never fully materialized. You get the same answer either way, which is why the lie is safe.

16.2.6 WORDS - remember these

Preprocessor — the text editor that runs before the compiler — the phase that executes # directives, per C standard clause 6.10. Macro — a shorthand replaced by text — an object-like or function-like replacement list expanded during translation phase 4. Header guard — a trick to stop a file being pasted twice — an

`#ifndef` / `#define` / `#endif` idiom giving idempotent inclusion. Translation unit — one source file plus everything it pasted in — the input the compiler proper actually sees. Conditional compilation — keeping or dropping blocks of text — `#if`, `#ifdef`, `#elif`, `#else`, `#endif` evaluated on preprocessing tokens.

16.3 Lexing: characters into tokens

16.3.1 PLAIN - in simple words

1. After the text is glued together, it is still just a row of characters.
2. The **lexer** reads that row left to right and groups characters into words.
3. Each word it produces is a **token**: a small labelled piece.
4. `int` becomes a keyword token. `add` becomes an identifier token. `2` becomes a number token. `+` becomes an operator token.
5. Spaces, tabs, newlines and comments produce no tokens at all. They only separate things.
6. The lexer does not care about order. `+ + + )` is fine by the lexer. It is the next stage that objects.
7. The lexer's only real skill is to always take the longest match. In `a+++b` it produces `a`, `++`, `+`, `b`, not `a`, `+`, `+`, `+`, `b`.

16.3.2 PLAIN - a picture in your head

1. Imagine a long strip of paper with no spaces: `THECATSATONTHEMAT`.
2. Your job is to cut it into words and label each one: noun, verb, article.
3. You have a list of legal words. You scan from the left, and at each point you take the longest legal word you can.
4. You never look ahead at the sentence's meaning. You never ask if the sentence is sensible. You just cut and label.
5. You hand the next person a stack of labelled cards, in order.

Where this comparison breaks: English needs meaning to cut correctly. C is designed so that cutting never needs meaning. That was a deliberate language design decision, and languages that break it, such as C++ with its >> in nested templates, cause real trouble for their own compilers.

16.3.3 PLAIN - a worked example

This is the real token stream for our file, from `clang -Xclang -dump-tokens -fsyntax-only add.c`, trimmed to the token and the text:

```
int 'int'              identifier 'main'
identifier 'add'       l_paren '('
l_paren '('            void 'void'
int 'int'              r_paren ')'
identifier 'a'         l_brace '{'
comma ','              return 'return'
int 'int'              identifier 'add'
identifier 'b'         l_paren '('
r_paren ')'            numeric_constant '2'
l_brace '{'            comma ','
return 'return'        numeric_constant '3'
identifier 'a'         r_paren ')'
plus '+'               semi ';'
identifier 'b'         r_brace '}'
semi ';'               eof ''
r_brace '}'
```

1. Read the left column top to bottom, then the right column. That is the whole file: 32 tokens, then an end-of-file marker.
2. The keywords `int`, `void` and `return` get their own token kinds.

3. `add`, `main`, `a` and `b` are all `identifier`. The lexer does not know that `add` is a function and `a` is a parameter. It has no idea what they are.
4. `2` and `3` are `numeric_constant`. The text is kept, not the value. Working out that `2` means the number two happens later.
5. Every space and every newline vanished.
6. Clang also records the exact line and column of every token. That is why its error messages can point a caret at one character.

16.3.4 PLAIN - what is really happening inside

1. A lexer is a machine with a small memory: a **finite automaton**.
2. It has a current state, it reads one character, and that character plus the state decides the next state. That is all the memory it has.
3. Start in state "nothing". See a letter, go to state "reading an identifier". Keep reading letters and digits. See something else, stop, and emit an identifier token.
4. Start in state "nothing". See a digit, go to state "reading a number". Keep reading digits. Stop at the first non-digit.
5. Because the memory is one state and nothing else, a lexer cannot count. It cannot check that brackets match. That job belongs to the parser.
6. After building an identifier, the lexer looks the text up in a fixed table of keywords. If `int` is in the table, it is a keyword. If `add` is not, it is an identifier. This is the standard trick.
7. Handling comments, string literals with escapes, and numeric suffixes such as `10UL` are all extra states in the same machine.

16.3.5 TECHNICAL - the engineer's version

1. Lexical structure is specified with **regular expressions**. Regular expressions and finite automata describe exactly the same class of languages, a result from Stephen Kleene's work in 1951 and 1956.
2. The standard mechanical route is: regular expression -> nondeterministic finite automaton (NFA) by Thompson's construction, 1968 -> deterministic finite automaton (DFA) by subset construction -> minimized DFA -> a table.
3. That construction is why generated lexers are fast. Each input character costs one table lookup, so lexing is linear in the input size.
4. **Lex** was written by Mike Lesk and Eric Schmidt at Bell Labs, described in a 1975 technical report. **Flex**, the free rewrite, was written by Vern Paxson starting in 1987 and is what most Unix systems ship today.
5. The rules in a `.l` file are pattern-action pairs. Flex resolves conflicts by two fixed rules: longest match wins, and among equal-length matches, the rule written earliest wins.
6. Production compilers usually hand-write the lexer instead. Clang's `Lexer.cpp` and GCC's `libcpp` are hand-written, because hand-written code handles error recovery, source locations and preprocessor interaction better.
7. The **maximal munch** rule is in the C standard: the next token is the longest character sequence that could form one. It is why C++11 needed a special case so `>>` can close two templates.

Token class	Example text	C standard term
Keyword	`int`, `return`	keyword
Identifier	`add`, `a`	identifier
Number	`2`, `0x1f`, `3.5f`	constant
String	`"three"`	string literal
Operator	`+`, `==`, `->`	punctuator

16.3.6 WORDS - remember these

Token — one labelled word of the program — the smallest unit the parser consumes, carrying a kind, spelling and source location. Lexer — the tool that cuts text into tokens — the scanner implementing the lexical grammar, usually as a DFA. Regular expression — a pattern for matching text shapes — a formula denoting a regular language, equivalent in power to a finite automaton. Finite automaton — a machine whose whole memory is which state it is in — a five-tuple of states, alphabet, transition function, start state and accepting states. Maximal munch — always take the longest legal word — the longest-match rule mandated by the C and C++ lexical grammars.

16.4 Parsing: tokens into a tree

16.4.1 PLAIN - in simple words

1. A row of tokens has no shape. `return a + b ;` is a flat list.
2. The **parser** gives it shape by building a tree.
3. In the tree, `+` sits above `a` and `b`, because `+` is the thing being done and `a` and `b` are the things it is done to.
4. `return` sits above the `+`, because returning is done to the result of the addition.
5. The tree is called an **abstract syntax tree**, or AST.
6. It is called abstract because the brackets and semicolons disappear. Their only job was to tell the parser the shape, and now the shape is the tree.
7. A **syntax error** is the parser reaching a token that cannot legally follow what it has already seen.
8. That is all a syntax error is. It is not about meaning. It is about shape.

16.4.2 PLAIN - a picture in your head

1. Think of the sentence “the cat sat on the mat”.
2. A flat list of six words tells you nothing about who did what.
3. A tree tells you: the action is “sat”, the doer is “the cat”, the place is “on the mat”.
4. Now a rule set: a sentence is a noun phrase followed by a verb phrase. A noun phrase is an article followed by a noun. Follow the rules and the tree assembles itself.
5. A syntax error is “the cat sat on the”: you have run out of sentence while a rule still needs a noun.

Where this comparison breaks: English is ambiguous, so the same sentence can have several correct trees. “I saw the man with the telescope” has two. Real programming language grammars are designed hard to avoid this, and where it creeps in, the language adds a written tie-breaking rule, such as C’s “an `else` belongs to the nearest unmatched `if`”.

16.4.3 PLAIN - a worked example

The rules of a language are written in **Backus-Naur Form**. Here is a tiny slice, enough for our function:

```
function     ::= type identifier "(" params ")" block
block        ::= "{" statement* "}"
statement    ::= "return" expression ";"
expression   ::= expression "+" term | term
term         ::= identifier | number | call
call         ::= identifier "(" arglist ")"
```

Read `::=` as “is made of”, `|` as “or”, `*` as “zero or more”. Now the real AST for our file. This is a redrawing of the actual output of `clang -Xclang -ast-dump -fsyntax-only add.c`:

```
TranslationUnit
 |
 +- FunctionDecl  add : int (int, int)
 |   +- ParmVarDecl  a : int
```

```
|   +- ParmVarDecl  b : int
|   +- CompoundStmt
|       +- ReturnStmt
|           +- BinaryOperator '+' : int
|               +- DeclRefExpr a : int
|               +- DeclRefExpr b : int
|
+- FunctionDecl  main : int (void)
    +- CompoundStmt
        +- ReturnStmt
            +- CallExpr : int
                +- DeclRefExpr add : int (int, int)
                +- IntegerLiteral 2 : int
                +- IntegerLiteral 3 : int
```

1. `TranslationUnit` is the whole file.
2. Under it are exactly two things, our two functions.
3. `CompoundStmt` is the `{ ... }` block. The braces themselves are gone.
4. `BinaryOperator '+'` has two children. That is the whole meaning of `a + b`.
5. `CallExpr` has three children: what to call, then each argument in order.
6. The semicolons are gone. The parentheses are gone. Nothing is lost, because the tree already says what they said.

And a real syntax error, from `gcc -c synerr.c`:

```
synerr.c:1:36: error: expected expression before ';' token
    1 | int add(int a, int b) { return a + ; }
      |                                    ^
```

The parser had read `a` and `+`, so its rule demanded another expression. It got `;`. No rule allows that, so it stopped and reported the column.

16.4.4 PLAIN – what is really happening inside

1. There are two main families of parser: top-down and bottom-up.
2. **Top-down** starts from "this should be a function" and tries to prove it by matching smaller and smaller pieces.
3. **Recursive descent** is top-down written by hand. You write one function per grammar rule. `parseStatement` calls `parseExpression`, which calls `parseTerm`. The call stack of the compiler mirrors the tree being built.
4. **Bottom-up** does the opposite. It pushes tokens onto a stack, and whenever the top of the stack matches the right-hand side of a rule, it replaces those items with the left-hand side. That is called a reduce.
5. Bottom-up parsers are usually generated by a tool from a grammar file, because the tables are far too tedious to write by hand.
6. **Precedence** is the rule that `2 + 3 * 4` is 14, not 20. A hand-written parser gets it by layering: the `+` function calls the `*` function, so `*` binds tighter and sits deeper in the tree.
7. **Associativity** is the rule that `a - b - c` means `(a - b) - c`. It decides whether the tree leans left or right.

16.4.5 TECHNICAL – the engineer's version

1. Programming language syntax is described by a **context-free grammar** (CFG), level 2 in the Chomsky hierarchy from Noam Chomsky's 1956 classification.
2. Backus-Naur Form came from John Backus in 1959 and Peter Naur's editing of the ALGOL 60 report in 1960. Donald Knuth proposed the name in 1964. Extended BNF adds `*`, `+`, `?` and grouping.

3. **LL(k)** parsers read Left to right and build a Leftmost derivation, with k tokens of lookahead. Recursive descent is the hand-written form of LL(1).
4. **LR(k)** parsers read Left to right and build a Rightmost derivation in reverse. LALR(1) is the practical variant used by Yacc and Bison. LR grammars are strictly more powerful than LL grammars.
5. **Yacc** was written by Stephen C. Johnson at Bell Labs, published as a 1975 technical report. **GNU Bison** is the free replacement, from Robert Corbett in the mid-1980s. ANTLR, by Terence Parr from 1989, generates adaptive ALL(*) parsers.
6. Real compilers have moved back to hand-written recursive descent. GCC replaced its Bison C++ parser in GCC 3.4, 2004, and its C parser in GCC 4.1, 2006. Clang was recursive descent from the start. The reason is error messages and recovery, not speed.
7. Two classic grammar problems. The **dangling else** is settled by a rule in the C standard. The C **typedef ambiguity** makes `(A)*b` a cast or a multiplication depending on whether `A` is a type.
   (a) C parsers settle the second by feeding symbol table facts back into the parser. That breaks the clean stage separation, and is called the lexer hack.
8. Operator precedence in C has 15 levels. Precedence climbing, also called Pratt parsing after Vaughan Pratt's 1973 paper, handles all of them in one compact function instead of 15 nested ones.

Parser style	Reads	Typical tool
Recursive descent	LL(1), hand	none, written by hand
LALR(1) table	bottom-up	Yacc, Bison
GLR	ambiguous CFG	Bison in GLR mode
PEG / packrat	ordered choice	pegjs, Rust pest
ALL(*)	adaptive LL	ANTLR 4

16.4.6 WORDS - remember these

Parser — the part that finds the shape of the program — the syntax analyser that builds a parse tree or AST from a token stream. Abstract syntax tree — a tree of what the code does — a tree of language constructs with punctuation removed, the front end's main data structure. Grammar — the written rules of the language's shape — a context-free grammar, usually given in BNF or EBNF. Recursive descent — one function per grammar rule — a hand-written predictive top-down parser, normally LL(1) with local backtracking. Syntax error — the shape is wrong — no production allows the current token given the parser state; reported with the source location of that token.

16.5 Semantic analysis: does it make sense

16.5.1 PLAIN - in simple words

1. A sentence can have correct shape and still be nonsense. "The number ate the Tuesday" is grammatical and meaningless.
2. **Semantic analysis** is the stage that catches the meaningless ones.
3. It asks: does this name exist here? Have you used it before declaring it? Have you declared it twice?
4. It asks: do the types fit? Can you add these two things? Can you pass this thing where that thing is expected?
5. It builds a **symbol table**: a list of every name, what it is, what type it has, and where it is visible.
6. Most of the errors a working programmer sees come from this stage, not from the parser.

16.5.2 PLAIN - a picture in your head

1. Imagine a hotel with a guest register at the front desk.
2. When somebody checks in, you write their name and room in the register.

3. When a letter arrives for "Mr Smith", you look the name up. If he is not in the register, the letter cannot be delivered.
4. The hotel has floors. Each floor has its own small register. When looking up a name you check the floor you are on, then the floor below, then the ground floor register.
5. When a floor closes for the night, its small register is thrown away, and those names stop meaning anything.

Where this comparison breaks: a hotel register is one flat list per floor. A real symbol table also records types, storage class, linkage, whether the thing is a function or a variable, and how the compiler will refer to it later. And hotel names are unique; C deliberately lets an inner name hide an outer one.

16.5.3 PLAIN - a worked example

The symbol table for our file, after analysing it, holds these entries:

Name	Kind	Type	Scope
add	function	int (int, int)	file
a	parameter	int	body of add
b	parameter	int	body of add
main	function	int (void)	file

1. When the parser reaches `a + b` inside `add`, semantic analysis looks up `a`. It finds the parameter, whose type is `int`.
2. It looks up `b`. Also `int`.
3. It checks that `+` accepts two `int` values. It does. The result type of the whole expression is recorded as `int`.
4. It checks that the type being returned matches the function's declared return type. `int` and `int`. Fine.
5. In `main`, it looks up `add`, finds a function of type `int (int, int)`, checks the call has two arguments, and checks each argument's type.

Now break it on purpose. Change the call to `add(2, "three")`. Real output from `clang -c typeerr.c`:

```
typeerr.c:2:32: error: incompatible pointer to integer
  conversion passing 'char[6]' to parameter of type 'int'
    2 | int main(void) { return add(2, "three"); }
      |                                ^~~~~~~
typeerr.c:1:20: note: passing argument to parameter 'b' here
    1 | int add(int a, int b) { return a + b; }
      |                    ^
1 error generated.
```

1. The shape of this program is perfectly legal. The parser was happy.
2. Only the type checker complains, and it can quote two places: the call and the declaration. It could only do that because the symbol table remembered where `add` was declared.
3. GCC 13.3 makes this a warning by default and still compiles. Clang 18 makes it an error. Experts disagree on default strictness; GCC 14 turned several such warnings into errors in 2024. Use `-Werror` to be strict.

16.5.4 PLAIN - what is really happening inside

1. The symbol table is usually a stack of hash tables, one per scope.
2. Entering a `{` pushes a new table. Leaving `}` pops it.
3. Looking up a name searches the top table, then the one below, and so on. The first hit wins. That is exactly why an inner variable hides an outer one.

4. Type checking walks the AST from the leaves upward. Each node computes its own type from its children's types.
5. `IntegerLiteral 2` says "I am int". `DeclRefExpr a` asks the symbol table and says "I am int". `BinaryOperator +` sees two ints and says "I am int".
6. Where types nearly match, C inserts a silent conversion. The Clang AST for our file shows `ImplicitCastExpr <LValueToRValue>` around `a` and `b`. That node means "read the value out of the variable". The compiler added it; you did not write it.
7. **Type inference** is the same walk in reverse: the compiler works out a type instead of checking a written one. In C++ `auto x = 2 + 3;` the type of `x` is deduced from the right-hand side.

16.5.5 TECHNICAL – the engineer's version

1. Semantic analysis covers name resolution, scope and lifetime rules, type checking, implicit conversion insertion, constant expression evaluation, and language-specific rules such as C's requirement that a `case` label be a constant.
2. C has four scopes in the standard: file, block, function prototype and function (labels only). Linkage is separate from scope: `static` at file level means internal linkage, no `static` means external linkage.
3. Clang stores this in a `Sema` object with an `IdentifierResolver` and a chain of `DeclContexts`. GCC uses `binding_level` structures. Both amount to a scoped hash table.
4. Full **Hindley-Milner** inference, from J. Roger Hindley in 1969 and Robin Milner in 1978, infers types for a whole program without annotations. ML, OCaml and Haskell use it. Rust and C++ use weaker local inference, so errors stay near the mistake.
5. The dividing line matters for error quality. Missing semicolon is a parse error. Undeclared identifier, wrong argument count, wrong argument type, assigning to a `const`, duplicate definition and returning the wrong type are all semantic errors.
6. Borrow checking in Rust and lifetime analysis are semantic analysis too, done on a control-flow graph after the AST. They are the reason `rustc` is slower than a C front end.
7. Useful flags: `gcc -Wall -Wextra -Werror`, `gcc -fsyntax-only` to stop after this stage, `clang -Xclang -ast-dump` to see the typed tree.

Error	Stage that catches it
Missing ;	Parser
Unbalanced {	Parser
Undeclared variable	Semantic
Wrong argument type	Semantic
Wrong number of args	Semantic
Missing function body	Linker
Divide by zero at run time	Nobody, it crashes

16.5.6 WORDS – remember these

Symbol table — the list of every name and what it means — a scoped mapping from identifier to declaration, type, storage class and linkage. Scope — the region where a name is visible — the block, file, prototype or function region defined by the language standard. Type checking — making sure the pieces fit — verifying every expression's type against the operator or context that consumes it. Implicit conversion — a change the compiler adds silently — a cast node inserted into the AST, such as C's integer promotions and lvalue-to-rvalue conversion. Name resolution — deciding which declaration a name refers to — lookup through the scope chain, obeying shadowing and linkage rules.

16.6 Intermediate representation

16.6.1 PLAIN - in simple words

1. After the tree is checked, the compiler could write chip instructions straight away. Almost none do.
2. Instead they rewrite the program in a simple made-up language of their own. That is the **intermediate representation**, or IR.
3. The IR is deliberately boring. Every line does one tiny thing. No nesting, no shortcuts, no cleverness.
4. Boring is useful. It is much easier to spot a wasted step in a flat list of tiny operations than in a tree full of nested expressions.
5. It is also useful because the IR does not belong to any chip. The same IR can later be turned into instructions for many different chips.
6. And it does not belong to any language either. C, C++, Rust and Swift can all produce the same IR, and then share one optimizer and one code generator.

16.6.2 PLAIN - a picture in your head

1. Think of an international airport with one central hub.
2. There are many airlines flying in from many countries, and many destinations flying out.
3. Without a hub you would need a direct route from every origin to every destination. Ten origins and ten destinations means a hundred routes.
4. With a hub, every origin flies to the hub, and the hub flies to every destination. Ten plus ten. Twenty routes instead of a hundred.
5. The IR is the hub. Languages fly in, chips fly out.

Where this comparison breaks: a passenger arrives at the hub unchanged. A program does not. Most of the real work, and most of the compiler's total running time, happens while the program is sitting in the hub being rewritten.

16.6.3 PLAIN - a worked example

Here is our `add` function in GCC's IR, called GIMPLE, from `gcc -O1 -fdump-tree-gimple=/dev/stdout -S add.c`:

```
int add (int a, int b)
{
  int D.2746;
  D.2746 = a + b;
  return D.2746;
}

int main ()
{
  int D.2748;
  D.2748 = add (2, 3);
  return D.2748;
}
```

1. Notice the temporary name `D.2746`. The compiler invented it.
2. Every statement has at most one operator and at most three names: one destination and two sources. That is why this style is called **three-address code**.
3. Now the same thing in Static Single Assignment form, from `gcc -O1 -fdump-tree-ssa=/dev/stdout -S add.c`:

```
int add (int a, int b)
{
  int _3;
  <bb 2> :
```

```
  _3 = a_1(D) + b_2(D);
  return _3;
}
```

4. Every name now has a version number. `a_1`, `b_2`, `_3`.
5. `<bb 2>` is a **basic block**: a run of instructions with no branch in the middle. Control enters at the top and leaves at the bottom.
6. And here is our whole file in LLVM IR, real output from `clang -O0 -S -emit-llvm add.c`, with the attribute lines removed:

```
define dso_local i32 @add(i32 noundef %0, i32 noundef %1) {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 %0, ptr %3, align 4
  store i32 %1, ptr %4, align 4
  %5 = load i32, ptr %3, align 4
  %6 = load i32, ptr %4, align 4
  %7 = add nsw i32 %5, %6
  ret i32 %7
}

define dso_local i32 @main() {
  %1 = alloca i32, align 4
  store i32 0, ptr %1, align 4
  %2 = call i32 @add(i32 noundef 2, i32 noundef 3)
  ret i32 %2
}
```

7. `i32` means a 32-bit integer. `%0` and `%1` are the two parameters.
8. `alloca` reserves space on the stack. At `-O0` the compiler dutifully stores both parameters to the stack and loads them straight back. That is pointless, and the optimizer will delete it.
9. `add nsw` means add with "no signed wrap": the compiler is allowed to assume signed overflow never happens. Hold on to that; it comes back in 16.7.

16.6.4 PLAIN - what is really happening inside

1. **Static Single Assignment**, or SSA, means every name is written exactly once in the whole function.
2. If your source assigns `x` three times, the IR has `x_1`, `x_2` and `x_3`.
3. Why bother? Because now, if you see a use of `x_2`, there is exactly one place it could have come from. No searching. No guessing.
4. That single property makes constant propagation, dead code removal and value numbering enormously simpler and faster.
5. Branches create a problem: after an `if`, which version of `x` is live? SSA solves it with a **phi node**, written `PHI`, which says "take the version from whichever block we came from".
6. Here is a real phi from a loop, `for (i = 0; i < n; i++) s += i;`, dumped by GCC with `-fdump-tree-ssa`:

```
  <bb 4> :
  # s_1 = PHI <s_3(2), s_8(3)>
  # i_2 = PHI <i_4(2), i_9(3)>
  if (i_2 < n_5(D))
    goto <bb 3>;
  else
    goto <bb 5>;
```

7. Read `s_1 = PHI <s_3(2), s_8(3)>` as: if we arrived from block 2, `s_1` is `s_3`; if from block 3, it is `s_8`.

8. Phi nodes are not real instructions. Late in the back end they are removed by inserting copies into the predecessor blocks.

16.6.5 TECHNICAL - the engineer's version

1. SSA was formalized by Ron Cytron, Jeanne Ferrante, Barry Rosen, Mark Wegman and Kenneth Zadeck in *Efficiently computing static single assignment form and the control dependence graph*, ACM TOPLAS, October 1991.
2. Placing phi nodes minimally requires **dominance frontiers**, computed by the Lengauer-Tarjan dominator algorithm of 1979 or Cooper, Harvey and Kennedy's simpler 2001 method.
3. GCC uses two IRs in sequence. GIMPLE is a three-address, SSA-capable, tree-ish IR added in GCC 4.0, 2005. RTL, Register Transfer Language, is the older low-level IR used by the back end.
4. LLVM uses a single strongly typed SSA IR with three equivalent forms: an in-memory form, a human-readable `.ll` text form, and a compact `.bc` bitcode form. `llvm-as` and `llvm-dis` convert between text and bitcode.
5. LLVM IR is not a portable virtual machine. It embeds the target data layout and pointer sizes, visible in our dump as `target triple = "x86_64-pc-linux-gnu"`. Shipping `.bc` and expecting it to run anywhere is a common misunderstanding.
6. The front-end / middle-end / back-end split is what makes the ecosystem work. Clang, Rust, Swift, Julia, Zig and `flang` all emit LLVM IR. LLVM 18 registers over 30 target backends, from x86 and AArch64 to RISC-V, WebAssembly, NVPTX and AVR.
7. Other IR designs exist. Java bytecode and CPython bytecode are stack machines, not three-address. WebAssembly is a structured stack machine. Cranelift, used by Wasmtime, uses its own SSA IR built for compile speed rather than peak output quality.

IR	Owner	Style
GIMPLE	GCC	3-address, SSA
RTL	GCC back end	low-level, registers
LLVM IR	LLVM	typed SSA
Java bytecode	JVM	stack machine
CPython bytecode	CPython	stack machine
Wasm	W3C	structured stack

16.6.6 WORDS - remember these

Intermediate representation — the compiler's own simple language — a program form between source and machine code, target and language neutral in the middle end. Three-address code — one operation, one result, two inputs per line — a linearized IR form of the shape `t1 = a op b`. SSA — every name written exactly once — static single assignment form, with phi functions merging values at control-flow joins. Basic block — a straight run with no branches inside — a maximal instruction sequence with a single entry and a single exit. Phi node — "take whichever value we came in with" — a pseudo-instruction at a join point selecting a value by predecessor block.

16.7 Optimization

16.7.1 PLAIN - in simple words

1. The optimizer rewrites your program so it does less work but produces the same visible result.
2. It is allowed to change anything you cannot observe. It is not allowed to change what you can observe.
3. If the answer is always the same number, compute it now and store the number.
4. If a value never changes, replace the name with the value.
5. If a result is never used, delete the code that made it.

6. If the same sum is computed twice, compute it once.
7. If a function is tiny, paste its body into the caller instead of calling it.
8. Do all of these repeatedly, because each one exposes new chances for the others.

16.7.2 PLAIN – a picture in your head

1. Imagine editing a set of instructions for making tea for a guest.
2. Step 3 says "boil water". Step 9 says "boil water" again. Delete step 9 and reuse the first pot. That is common subexpression elimination.
3. Step 5 says "count the sugar packets in the drawer" every time round the stirring loop, but nobody adds packets. Move it outside the loop. That is loop-invariant code motion.
4. Step 7 says "fetch a lemon" and no later step uses the lemon. Delete step 7. That is dead code elimination.
5. Step 2 says "look up the boiling point of water in the book". You know it is 100. Write 100. That is constant folding.
6. Step 8 says "consult the separate one-line card called stir". Just write "stir" here. That is inlining.

Where this comparison breaks: you may delete "fetch a lemon" only if fetching lemons has no other effect. In a real program, a "useless" call might print something, write a file or set a flag another thread reads. Deciding whether a step is observable is the hard part, and it is most of what an optimizer does.

16.7.3 PLAIN – a worked example

This is the heart of the chapter. Compile our unchanged file two ways.

`gcc -O0 -S add.c`, the whole `add` function, with directives removed:

```
add:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -4(%rbp)
    movl    %esi, -8(%rbp)
    movl    -4(%rbp), %edx
    movl    -8(%rbp), %eax
    addl    %edx, %eax
    popq    %rbp
    ret

main:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    movl    $3, %esi
    movl    $2, %edi
    call    add
    popq    %rbp
    ret
```

`gcc -O2 -S add.c`, the same file:

```
add:
    endbr64
    leal    (%rdi,%rsi), %eax
    ret

main:
    endbr64
```

```
    movl    $5, %eax
    ret
```

1. Seventeen instructions became five.
2. Why is `add` now one instruction? At `-O0` the compiler stores both arguments to the stack and reloads them, because `-O0` keeps every variable in memory for the debugger. At `-O2` that is deleted.
3. `leal (%rdi,%rsi), %eax` computes `rdi + rsi` and puts it in `eax`. `lea` means load effective address: it does the address arithmetic but does not touch memory. It is used here as a free three-operand add.
4. Why is `main` now `movl $5, %eax`? Three optimizations in sequence.
   (a) **Inlining**: `add` is tiny, so its body is pasted into `main`, giving `return 2 + 3;`.
   (b) **Constant folding**: `2 + 3` is computed at compile time, giving `return 5;`.
   (c) **Dead code elimination**: nothing else in `main` is needed, so the frame setup, the call, and the stack work all disappear.
5. `add` still exists as a separate function because it has external linkage: another file might call it. Mark it `static` and the whole function vanishes from the output.
6. The program still returns 5. `./a.out; echo $?` prints `5`. That is the only thing that had to stay true.

Other optimizations, each verified on this machine with GCC 13.3 at `-O2`:

Source	Output at -O2
`3 * 4 + 10 / 2`	`movl $17, %eax`
`x * 8`	`leal 0(,%rdi,8), %eax`
`int u = a*999; return a+1;`	`leal 1(%rdi), %eax`
`a*b+1` and `a*b+2` summed	one `imull`, then `leal`

1. Row one is constant folding: the whole expression is a compile-time constant.
2. Row two is **strength reduction**: multiply by 8 became a scaled address computation, which is cheaper than a general multiply.
3. Row three is dead code elimination: the multiply by 999 is gone.
4. Row four is common subexpression elimination: `a*b` was written twice and is computed once.

16.7.4 PLAIN - what is really happening inside

1. **Loop unrolling** copies a loop body several times so the loop test runs less often. Our recursive `tail(n, acc)` function was turned by GCC into a loop and then unrolled by a factor of two.
2. **Tail call elimination** is what made that possible. When the last thing a function does is call something and return its result, the call can reuse the current stack frame. A self tail call then becomes a jump, that is, a loop.
3. Even across separate files, GCC at `-O2` turned `call add; ret` into a plain `jmp add`. That is a tail call: `add`'s own `ret` returns straight to `main`'s caller.
4. **Vectorization** does several elements at once. For `for (i=0;i<n;i++) out[i] = in[i] * 2.0f;` GCC at `-O3` emitted `movups` and `addps`. `addps` is "add packed single", four 32-bit floats in one instruction, using the 128-bit SSE registers.
5. **Register allocation** decides which values live in registers. x86-64 has 16 general purpose registers, a few reserved, so roughly 12 to 14 are usable. Values that do not fit must be **spilled** to the stack.
6. Chaitin's method models this as **graph colouring**. Each value is a node, and two nodes are joined if they are live at the same moment. Colouring with k colours, k being the register count, is exactly a valid allocation.
7. Graph colouring is NP-complete in general, so compilers use heuristics: repeatedly remove any node with fewer than k neighbours, and if none exists, pick a node to spill and try again.

16.7.5 TECHNICAL - the engineer's version

1. The legal limit on all of this is the **as-if rule**. An implementation may do anything, provided observable behaviour matches the abstract machine. The C standard defines that as volatile accesses, input and output, and the state at termination.
2. GCC optimization levels, as documented for GCC 13: `-O0` none, `-O1` about 47 passes, `-O2` most non-size-increasing passes, `-O3` adds vectorization and heavier inlining, `-Os` optimizes for size, `-Ofast` is `-O3` plus `-ffast-math`, `-Og` keeps debugging usable.
3. **Floating point is not associative.** IEEE 754 arithmetic rounds each operation, so `(a+b)+c` and `a+(b+c)` can differ. Real run on this machine with `a = 1e20f`, `b = -1e20f`, `c = 1.0f`:

```
(a+b)+c = 1
a+(b+c) = 0
```

4. That is why compilers must not reassociate floating point by default. `-ffast-math` grants permission to reassociate, to assume no NaN or infinity, and to flush denormals. On this machine, adding `0.1f` ten times prints `1.00000012`, not `1`.
5. **Undefined behaviour** lets the compiler assume the impossible never happens. Signed integer overflow is undefined in C. Real GCC 13.3 output at `-O2`:

```
int check(int x) { return x + 1 > x; }
check:
    movl    $1, %eax        # always true, folded
    ret

unsigned check_u(unsigned x) { return x + 1 > x; }
check_u:
    xorl    %eax, %eax      # real comparison kept
    cmpl    $-1, %edi
    setne   %al
    ret
```

6. The signed version was folded to a constant `1`. The unsigned version was not, because unsigned overflow is defined to wrap, so `x + 1 > x` really is false when `x` is `UINT_MAX`.
7. Compiling the signed version with `-fwrapv` restores the comparison, against `2147483647`. `-fwrapv` is a GCC and Clang extension defining signed overflow as wrapping. It is an implementation choice, not the C standard.
8. Undefined behaviour also erases null checks after a dereference, deletes infinite loops with no side effects, and assumes strict aliasing. Tools: `-fsanitize=undefined`, `-fsanitize=address`, and Clang's `-Rpass=inline` to see what was inlined.

Optimization	What it removes
Constant folding	Compile-time arithmetic
Constant propagation	Reads of known values
Dead code elimination	Unused computations
Common subexpr elim	Repeated identical work
Strength reduction	Expensive operators
Loop-invariant motion	Repeated loop work
Inlining	Call overhead
Tail call elimination	Stack frame growth

16.7.6 WORDS - remember these

Constant folding — do the sums now, not later — evaluating constant expressions at compile time. Inlining — paste the function body in — replacing a call with the callee's body, enabling further optimization at the call site. Dead code elimination — delete what nobody uses — removing instructions whose results are not live and have no side effects. Spilling — running out of registers and using memory — storing a live value to the stack frame because the allocator could not colour it. As-if rule — you may change anything nobody can see — the standard's permission to transform a program while preserving observable behaviour. Undefined behaviour — the standard gives no rules for this — a construct for which the standard imposes no requirements, letting the optimizer assume it never occurs.

16.8 Code generation

16.8.1 PLAIN - in simple words

1. Now the compiler must write instructions for one actual chip.
2. Three jobs happen here, tangled together.
3. **Instruction selection**: pick which real instructions do what the IR said. The IR said "add". The chip may offer `add`, `lea` or `inc`.
4. **Instruction scheduling**: pick the order. Some instructions wait for results; put unrelated work in the gap.
5. **Register allocation**: pick which of the chip's few fast slots holds which value, and put the rest in memory.
6. There is also a rule book that everybody must obey: where arguments go, where the answer comes back, who is allowed to break which register.
7. That rule book is the **calling convention**. It is what lets a function compiled today call a library compiled ten years ago.

16.8.2 PLAIN - a picture in your head

1. Think of handing work to a colleague through a hatch with numbered trays.
2. Everyone in the building has agreed: the first thing you are passing goes in tray 1, the second in tray 2, and the answer comes back in tray 0.
3. Some trays are yours to scribble on; the colleague may empty them. Other trays they must hand back exactly as they found them.
4. If you both follow the agreement, neither of you needs to know anything about how the other works inside.
5. If you disagree about which tray is which, everything breaks in ways that look like random corruption.

Where this comparison breaks: real conventions also cover stack alignment, variable argument lists, structures too big for a register, floating point in separate registers, and where the return address lives. And there is no single agreement: Linux, Windows and ARM each chose different trays.

16.8.3 PLAIN - a worked example

Here is our `-O0` `add` function again, and this time every line is annotated.

```
add:
    endbr64                 # landing pad for indirect jumps (CET)
    pushq   %rbp            # save caller's frame pointer
    movq    %rsp, %rbp      # this function's frame starts here
    movl    %edi, -4(%rbp)  # store arg 1 (a) into the frame
    movl    %esi, -8(%rbp)  # store arg 2 (b) into the frame
    movl    -4(%rbp), %edx  # load a back into edx
    movl    -8(%rbp), %eax  # load b back into eax
    addl    %edx, %eax      # eax = a + b
```

```
    popq    %rbp            # restore caller's frame pointer
    ret                     # jump back to the return address
```

1. `%edi` and `%esi` are the low 32 bits of `rdi` and `rsi`. On Linux those are the first two integer argument registers.
2. `%eax` holds the return value. The result of the addition is already there, so no extra move is needed.
3. The two stores and two loads are pure `-O0` overhead, as explained in 16.7.
4. `pushq %rbp` and `movq %rsp, %rbp` are the **prologue**. `popq %rbp` and `ret` are the **epilogue**.
5. `endbr64` is not part of the calling convention. It is Intel Control-flow Enforcement Technology, enabled by default on Ubuntu with `-fcf-protection=full`. It marks a legal target for an indirect jump.

And `main`:

```
main:
    endbr64
    pushq   %rbp            # prologue
    movq    %rsp, %rbp
    movl    $3, %esi        # second argument = 3
    movl    $2, %edi        # first argument  = 2
    call    add             # push return address, jump to add
    popq    %rbp            # epilogue; eax already holds 5
    ret                     # return to the C runtime
```

1. Arguments are loaded second-first here. That order is the compiler's choice, not a rule. Only which register holds which argument is fixed.
2. `call` pushes the address of the next instruction onto the stack and jumps.
3. Nothing copies the result. `add` left it in `eax` and `main` returns it from `eax`.

The stack while `add` is running:

```
higher addresses
  +-----------------------------+
  | main's frame                |
  +-----------------------------+
  | return address into main    | <- pushed by call
  +-----------------------------+
  | saved rbp                   | <- pushed by prologue
  +-----------------------------+ <- rbp now points here
  | -4(%rbp)  copy of a         |
  | -8(%rbp)  copy of b         |
  +-----------------------------+ <- rsp
  | 128-byte red zone           |    usable without moving rsp
  +-----------------------------+
lower addresses
```

16.8.4 PLAIN – what is really happening inside

1. Instruction selection is usually done by **tree pattern matching**. The IR is covered with the cheapest set of instruction patterns that fits.
2. The cost model matters. `imul` by 8 costs about 3 cycles on many x86 cores; `lea` with a scale of 8 costs 1. That is why GCC chose `lea`.
3. Instruction scheduling matters most on machines that issue several instructions per cycle. Modern out-of-order x86 chips reorder anyway, so scheduling is worth less there than on an in-order core.
4. The same source compiled for AArch64 with `clang --target=aarch64-linux-gnu -O2` gives:

```
add:
    add     w0, w1, w0
    ret
```

```
main:
    mov     w0, #5
    ret
```

5. Same structure, different names. `w0` and `w1` are the low 32 bits of `x0` and `x1`, the first two AArch64 argument registers, and `x0` is also the return register.
6. The same source targeting Windows with `clang --target=x86_64-pc-windows-msvc -O0` stores from `%ecx` and `%edx` instead of `%edi` and `%esi`, because Windows chose different argument registers.
7. One source file, three sets of rules, all correct.

16.8.5 TECHNICAL - the engineer's version

1. The three conventions side by side. All three are written specifications, not conventions in the loose sense: System V AMD64 ABI, Microsoft x64 calling convention, and Arm's AAPCS64.

Role	SysV AMD64	Win x64	AArch64
Int arg 1	rdi	rcx	x0
Int arg 2	rsi	rdx	x1
Int arg 3	rdx	r8	x2
Int arg 4	rcx	r9	x3
Int args 5-6	r8, r9	stack	x4, x5
Float args	xmm0-xmm7	xmm0-xmm3	v0-v7
Return	rax (rdx:rax)	rax	x0 (x1 too)
Shadow space	none	32 bytes	none
Red zone	128 bytes	none	none
Frame pointer	rbp	rbp	x29
Return address	on stack	on stack	x30 (lr)

2. **Callee-saved** registers on System V AMD64 are rbx, rbp, r12, r13, r14, r15. A function that uses them must restore them. Everything else is **caller-saved**: if you need it after a call, save it yourself.
3. Stack alignment: System V AMD64 requires `rsp + 8` to be 16-byte aligned at a `call`, so `rsp` is 16-byte aligned on entry after the return address is pushed. AAPCS64 requires `sp` 16-byte aligned at all public interfaces.
4. The **red zone** is 128 bytes below `rsp` that a leaf function may use without adjusting `rsp`. Signal handlers must not clobber it. Kernel code is compiled with `-mno-red-zone` for exactly this reason.
5. Structures larger than 16 bytes are passed in memory on System V AMD64; smaller ones are classified field by field into INTEGER and SSE classes. The classification algorithm in the ABI document is one of the fiddliest parts of the whole convention.
6. Register allocation in production: GCC uses IRA, an integrated regional allocator, plus LRA for reload, replacing the old `reload` pass in GCC 4.8, 2013. LLVM's default is a greedy allocator based on priority and live range splitting, not classical Chaitin colouring. Linear scan allocation, from Massimiliano Poletto and Vivek Sarkar in 1999, is preferred by JITs because it is far faster to run.
7. Useful commands: `gcc -S -masm=intel` for Intel syntax, `objdump -d --no-show-raw-insn`, `perf annotate` to see which instruction is hot, and `llvm-mca` to model instruction throughput on a named CPU.

16.8.6 WORDS - remember these

Calling convention — the shared rules about who puts what where — the ABI specification of argument registers, return registers, stack alignment and register preservation. Prologue — the few instructions that set up a function — saving the frame pointer and reserving stack space on entry. Epilogue — the few instructions that clean up — restoring saved registers and returning. Stack frame — one function's private

scratch area — the region between the frame pointer and the stack pointer holding locals, spills and saved registers. Callee-saved register — one you must hand back unchanged — a register the ABI requires the called function to preserve across the call. Instruction selection — choosing which real instructions to use — covering the IR with target patterns under a cost model.

16.9 The assembler

16.9.1 PLAIN - in simple words

1. The compiler's output is still text. `addl %edx, %eax` is nine characters.
2. The **assembler** turns each such line into the bytes the chip actually reads.
3. It is a much simpler program than the compiler. Mostly it is a lookup table plus arithmetic on addresses.
4. It also collects your program into named piles called **sections**.
5. Code goes in one pile. Numbers you gave starting values go in another. Numbers that start at zero go in a third that takes no file space at all.
6. It records every name you defined and every name you used but did not define.
7. Where an address is not yet known, it writes zeros and leaves a note. Those notes are called **relocations**.
8. Its output is an **object file**, ending `.o` on Unix and `.obj` on Windows.

16.9.2 PLAIN - a picture in your head

1. Think of typesetting a book chapter that refers to other chapters.
2. You can set every word of your own chapter into metal type immediately.
3. But "see chapter 9, page ..." cannot be set, because you do not know the page number yet. Other people are still setting chapter 9.
4. So you leave a gap of the right size and write a note in the margin: "fill this gap with the start page of chapter 9".
5. You hand the printer your typeset pages plus your list of margin notes.
6. Later, somebody assembling the whole book fills every gap.

Where this comparison breaks: some relocations are not simple substitutions. They store a difference between two addresses, or an offset from the current position, or an index into a table filled at program start. The note in the margin has a type, and there are dozens of types.

16.9.3 PLAIN - a worked example

Run `gcc -c add.c -o add.o`, then `objdump -d add.o`:

```
0000000000000000 <add>:
   0:  f3 0f 1e fa     endbr64
   4:  55              push   %rbp
   5:  48 89 e5        mov    %rsp,%rbp
   8:  89 7d fc        mov    %edi,-0x4(%rbp)
   b:  89 75 f8        mov    %esi,-0x8(%rbp)
   e:  8b 55 fc        mov    -0x4(%rbp),%edx
  11:  8b 45 f8        mov    -0x8(%rbp),%eax
  14:  01 d0           add    %edx,%eax
  16:  5d              pop    %rbp
  17:  c3              ret

0000000000000018 <main>:
  18:  f3 0f 1e fa     endbr64
  1c:  55              push   %rbp
  1d:  48 89 e5        mov    %rsp,%rbp
  20:  be 03 00 00 00  mov    $0x3,%esi
  25:  bf 02 00 00 00  mov    $0x2,%edi
  2a:  e8 00 00 00 00  call   2f <main+0x17>
```

```
  2f:  5d              pop    %rbp
  30:  c3              ret
```

1. The left column is the offset inside the section. The middle column is the actual bytes. The right column is the text form.
2. `01 d0` is two bytes. That is the entire `add %edx,%eax` instruction.
3. `c3` is one byte: `ret`.
4. Look at offset `2a`. The `call` is `e8` followed by `00 00 00 00`. The assembler did not know where `add` would end up, so it wrote four zero bytes.
5. `objdump` guesses the target as `2f`, which is simply "the next instruction", because the offset is zero. That is not the real target. It is a hole.
6. Now the note in the margin, from `readelf -r add.o`:

```
Relocation section '.rela.text':
  Offset          Type              Sym. Name + Addend
  00000000002b  R_X86_64_PLT32    add - 4
```

7. Read it as: at byte offset `0x2b`, which is the four zero bytes, write the distance from here to `add`, minus 4.
8. `nm add.o` shows the symbols: `0000000000000000 T add` and `0000000000000018 T main`. `T` means defined in the text section.

16.9.4 PLAIN - what is really happening inside

1. The assembler makes two passes. Pass one measures every instruction and works out where each label lands. Pass two writes the bytes, using the label addresses from pass one.
2. Two passes are needed because of forward references: a jump to a label further down the file cannot be encoded until that label's position is known.
3. Sections keep unlike things apart so the operating system can protect them differently:
   (a) `.text` is machine code. It ends up readable and executable, not writable.
   (b) `.rodata` is constants, such as string literals. Readable, not writable.
   (c) `.data` is variables with a non-zero starting value. Readable and writable, and its contents occupy space in the file.
   (d) `.bss` is variables starting at zero. Readable and writable, and it takes no file space at all: only a size is recorded.
4. `readelf -S add.o` on our file shows `.text` at 0x31 bytes and both `.data` and `.bss` at zero bytes, because our program has no global variables.
5. That is why a program declaring a 10 MB zeroed array does not have a 10 MB executable. The `.bss` entry says "10 MB of zeros" in a few bytes.

16.9.5 TECHNICAL - the engineer's version

1. The object file format on Linux and most Unix systems is **ELF**, Executable and Linkable Format, introduced with UNIX System V Release 4 around 1988 and adopted by Linux in the mid-1990s, replacing a.out. macOS uses Mach-O, from NeXTSTEP. Windows uses PE/COFF.
2. Our `add.o` contains 13 section headers. The important ones are `.text`, `.rela.text`, `.data`, `.bss`, `.symtab`, `.strtab`, `.shstrtab` and `.eh_frame`.
3. `.symtab` is the symbol table, `.strtab` holds the symbol name strings, and `.shstrtab` holds the section name strings. Names are stored once and referenced by offset.
4. `.eh_frame` holds unwind information generated from the `.cfi_*` directives in the assembly. It is what lets a debugger produce a backtrace and what lets C++ exceptions unwind the stack. It is generated even for C.
5. Common x86-64 relocation types: `R_X86_64_64` absolute 64-bit, `R_X86_64_PC32` 32-bit program-counter relative, `R_X86_64_PLT32` call through the procedure linkage table, `R_X86_64_GOTPCREL` load through

the global offset table. `R_X86_64_RELATIVE` is applied at load time for position-independent executables.

6. Symbol binding is `LOCAL`, `GLOBAL` or `WEAK`. `nm` prints `T` for a global text symbol, `t` for local, `U` for undefined, `B` for `.bss`, `D` for `.data`, `W` for weak.
7. Our whole object file is 169 bytes of text, data and bss combined, per `size add.o`. The linked executable reports 1302 bytes of text, because the C runtime start-up code has been added.

Command	What it shows
`objdump -d f.o`	Disassembled code
`objdump -h f.o`	Section headers, sizes
`readelf -S f.o`	Full section table
`readelf -r f.o`	Relocation entries
`readelf -s f.o`	Symbol table
`nm f.o`	Symbols, short form
`size f.o`	text, data, bss totals

16.9.6 WORDS - remember these

Assembler — turns instruction text into bytes — a two-pass translator from assembly mnemonics to encoded machine instructions plus metadata. Object file — a half-finished program with holes — a relocatable ELF, Mach-O or COFF file containing sections, symbols and relocations. Section — one named pile of similar bytes — a named region such as `.text`, `.data`, `.bss` or `.rodata` with its own permissions. Relocation — a note saying "fill this hole later" — a record naming an offset, a symbol and a formula for computing the value to patch in. Symbol — a name the outside world can see — a named entry binding an identifier to a section and offset, with a binding and a visibility.

16.10 The linker

16.10.1 PLAIN - in simple words

1. Real programs are many files. Each is compiled separately into an object file full of holes.
2. The **linker** puts them all together and fills every hole.
3. It stacks all the `.text` sections into one `.text`, all the `.data` into one `.data`, and so on.
4. Now every function has a final address, so every hole can be filled with a real number.
5. Then it checks that every name that was used somewhere is defined somewhere.
6. If a name is used and never defined, it stops and says **undefined reference**. That is the most famous error message in C.
7. It also pulls in code from **libraries**: bundles of ready-made object files.

16.10.2 PLAIN - a picture in your head

1. Think of assembling one book from chapters written by different authors.
2. Each author numbered their pages from 1. You must renumber so the whole book runs from 1 to 400.
3. Every "see page 12" inside chapter 3 must be adjusted by however much chapter 3 moved.
4. Every "see the chapter on rivers" must be turned into a real page number by looking up which chapter that is.
5. If somebody wrote "see the chapter on volcanoes" and no such chapter exists, the book cannot be finished. That is undefined reference.
6. A **static library** is a shelf of spare chapters. You copy in only the ones somebody referred to.
7. A **dynamic library** is a separate book that stays separate. Your book says "look this up in the other book", and the reader must own that book.

Where this comparison breaks: the linker does more than renumber. It merges duplicate definitions of inline functions and templates, discards unused sections, can reorder functions for cache locality, and with link-time optimization can re-run the whole optimizer across chapter boundaries.

16.10.3 PLAIN – a worked example

Split our example across two files: `main2.c` calling `add`, and `addonly.c` defining it. Compile `main2.c` alone and try to link:

```
$ gcc main2.c -o prog2
/usr/bin/ld: /tmp/cckot5dw.o: in function `main':
main2.c:(.text+0x13): undefined reference to `add'
collect2: error: ld returned 1 exit status
```

1. The compiler was perfectly happy: the header declared `add`, so the call type checked.
2. The linker was not, because no object file defines `add`.
3. Now link both: `gcc main2.o addonly.o -o prog2`. It works, and returns 5.

Look at what relocation actually did. In `add.o` the call was `e8 00 00 00 00`. In the linked executable, `objdump -d`:

```
0000000000001129 <add>:
    1129:  f3 0f 1e fa   endbr64
    ...
0000000000001141 <main>:
    1141:  f3 0f 1e fa   endbr64
    ...
    1153:  e8 d1 ff ff ff   call   1129 <add>
```

4. The four zero bytes became `d1 ff ff ff`, which as a signed 32-bit little-endian number is `-47`.
5. The instruction after the call starts at `0x1158`. And `0x1158 - 47 = 0x1129`, which is exactly where `add` ended up.
6. That is the whole of relocation, in one number.

Now libraries. Real commands and real sizes on this machine:

```
$ ar rcs libadd.a addonly.o          # static library
$ gcc main2.c -L. -ladd -o prog_static

$ gcc -fPIC -shared addonly.c -o libadd.so   # shared library
$ gcc main2.c -L. -ladd -o prog_dyn -Wl,-rpath,'$ORIGIN'
```

File	Size in bytes
libadd.a	1372
libadd.so	15112
prog_static	15840
prog_dyn	15944

Both programs return 5. In `prog_static` the `add` code is inside the executable. In `prog_dyn` it is not, and `ldd prog_dyn` lists `libadd.so => /tmp/kb/./libadd.so`.

16.10.4 PLAIN – what is really happening inside

1. A static library `.a` file is just an archive of object files, made by `ar`. `ar -t libadd.a` prints `addonly.o`.
2. The linker treats an archive specially: it does not include everything. It scans the archive and pulls in only the members that resolve a symbol still undefined at that moment.

3. That is why **link order matters**. The linker walks the command line left to right, keeping a set of currently undefined symbols.
4. Real proof on this machine. `gcc -L. -ladd main2.c` fails with "undefined reference to `add`", while `gcc main2.c -L. -ladd` succeeds. Same files, different order.
5. The reason: when the library was scanned, nothing needed `add` yet, so no member was pulled in. `main2.c` came later and asked for a symbol nobody was still looking for. Put objects first and libraries last.
6. In **C++** the situation is harder, because C++ allows several functions with the same name. The linker only handles plain names, so the compiler encodes the full signature into the name. That is **name mangling**.
7. Real output from `nm` on a C++ file, alongside `nm -C` which demangles:

```
_Z3addii          ->  add(int, int)
_Z3adddd          ->  add(double, double)
_ZN2kb1T3addEii   ->  kb::T::add(int, int)
add_c             ->  add_c
```

8. Read `_Z3addii` as: mangled name, 3-letter name `add`, arguments `i`, `i`. The last one is `extern "C"`, which turns mangling off, which is exactly how C and C++ call each other.

16.10.5 TECHNICAL - the engineer's version

1. Linker phases: read inputs, resolve symbols, assign section addresses via a layout script, apply relocations, write output, optionally strip.
2. Library naming, all conventions rather than standards: `.a` static and `.so` shared on Linux, `.lib` static and `.dll` dynamic on Windows, `.a` static and `.dylib` dynamic on macOS. Windows also uses an import `.lib` that describes a `.dll` without containing its code.
3. **Symbol visibility** controls what a shared library exports. GCC and Clang accept `-fvisibility=hidden` plus `__attribute__((visibility("default")))` on the few symbols you mean to export. Smaller export tables mean faster loading and better optimization. Windows uses `__declspec(dllexport)` and `__declspec(dllimport)`.
4. Shared library code must be position independent, built with `-fPIC`, because it can be mapped at different addresses in different processes. Most Linux distributions have built executables as position independent (PIE) by default since around 2017.
5. **Link-time optimization** keeps the IR in the object file instead of only machine code, so the optimizer can run across file boundaries at link time. Real measurement on this machine, with `add` in a separate file from `main`:
   (a) `gcc -O2 main2.c addonly.c` gave `main` a tail call: `mov $3,%esi; mov $2,%edi; jmp 1150 <add>`.
   (b) `gcc -O2 -flto main2.c addonly.c` gave `main` exactly `mov $0x5,%eax; ret`.
6. That is the same inline-then-fold sequence from 16.7, now working across translation units, which plain `-O2` cannot do. GCC's LTO arrived in GCC 4.5, 2010. LLVM offers full LTO and ThinLTO, the latter published by Teresa Johnson and colleagues in 2017 for much better build parallelism.
7. Linker choices as of 2026: GNU `ld`, the original; `gold`, 2008, ELF only and dropped from recent binutils; LLVM `lld`, the default in many toolchains; and `mold` by Rui Ueyama, 1.0 in 2022, the fastest widely used linker for large C++ builds.
8. Weak symbols let a definition be overridden. `__attribute__((weak))` is how a library provides a default that a program may replace.

Symptom	Usual cause
undefined reference to X	Object or library missing
undefined reference in C++	Missing `extern "C"`, mangling
multiple definition of X	Definition in a header

Symptom	Usual cause
library before object fails	Wrong command-line order
runs, then “cannot open .so”	Library not on the run path

16.10.6 WORDS - remember these

Linker — the tool that glues object files into a program — the program that merges sections, resolves symbols and applies relocations. Static library — a shelf of object files you copy from — an `ar` archive whose members are pulled in only to satisfy undefined symbols. Shared library — a separate file loaded at run time — a `.so`, `.dll` or `.dylib` mapped into the process by the dynamic linker. Name mangling — encoding the full signature into the symbol name — the compiler scheme that gives overloaded and namespaced C++ entities distinct linker names. Link-time optimization — optimizing across file boundaries — keeping IR in object files so the optimizer runs at link time. Undefined reference — somebody used a name nobody defined — an unresolved symbol remaining after all inputs have been scanned.

16.11 The loader and dynamic linking

16.11.1 PLAIN - in simple words

1. A finished executable is a file on disk. It is not yet a running program.
2. When you type its name and press Enter, the operating system creates a process and puts the file’s contents into that process’s memory.
3. The part of the operating system that does this is the **loader**.
4. It does not copy the whole file. It maps it, meaning it says “this range of memory corresponds to this part of this file”, and lets the pages arrive when they are first touched.
5. If the program uses shared libraries, they are not inside it, so somebody must find them, load them too, and connect the calls.
6. That job belongs to a small program called the **dynamic linker**, which the loader starts first.
7. Only after all that does your `main` actually run.

16.11.2 PLAIN - a picture in your head

1. Think of a play. The script is the executable file. The performance is the process.
2. The stage manager, the loader, sets out the scenery from the script.
3. The script says “at this point, the orchestra plays the theme”. The orchestra is a shared library and is not in your script.
4. So before the curtain, an assistant goes and finds the orchestra, checks they have the right music, and writes their seat numbers into a small card at the side of the stage.
5. Every time the script says “orchestra”, an actor glances at the card to see where to look.
6. If the assistant is lazy, the card starts blank, and the first time the actor glances at it, the assistant is fetched to fill in that one entry.

Where this comparison breaks: the card, the global offset table, is per process, not per library, and it is normally made read-only after start-up for security. And the assistant is itself a shared library, loaded by a special path that does not need an assistant.

16.11.3 PLAIN - a worked example

`readelf -l` on our executable shows what the loader is told to do:

```
Elf file type is DYN (Position-Independent Executable)
Entry point 0x1040
  Type    VirtAddr    FileSiz  MemSiz  Flg  Align
  INTERP  0x00000318  0x00001c 0x00001c R    0x1
```

```
      [Requesting interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD    0x00000000  0x0005f0 0x0005f0 R    0x1000
  LOAD    0x00001000  0x000169 0x000169 R E  0x1000
  LOAD    0x00002000  0x0000ec 0x0000ec R    0x1000
  LOAD    0x00003df0  0x000220 0x000228 RW   0x1000
```

1. Four `LOAD` entries, four memory regions with different permissions.
2. The second is `R E`, readable and executable: that is your code.
3. The last is `RW` and its `MemSiz` (0x228) is larger than its `FileSiz` (0x220). The extra 8 bytes are `.bss`, zero-filled at load and stored in no file bytes at all.
4. `INTERP` names the dynamic linker. The kernel loads that, not your program.

Now watch it happen. `strace -e trace=execve,openat,mmap ./prog_dyn`, trimmed:

```
execve("./prog_dyn", ["./prog_dyn"], ...) = 0
openat(".../glibc-hwcaps/x86-64-v4/libadd.so") = -1 ENOENT
openat(".../glibc-hwcaps/x86-64-v3/libadd.so") = -1 ENOENT
openat("/tmp/kb/libadd.so", O_RDONLY|O_CLOEXEC) = 3
mmap(NULL,  16400, PROT_READ, ...)              = 0x7f88..
mmap(0x..1c, 4096, PROT_READ|PROT_EXEC, ...)
mmap(0x..1d, 4096, PROT_READ, ...)
mmap(0x..1e, 8192, PROT_READ|PROT_WRITE, ...)
openat("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC)  = 3
openat("/lib/x86_64-linux-gnu/libc.so.6", ...)  = 3
+++ exited with 5 +++
```

5. It searched several CPU-feature-specific directories first, then found the library, then mapped it in four pieces with different permissions.
6. It consulted `/etc/ld.so.cache`, a prebuilt index of every library on the system, before touching the real directories, because scanning directories for every library on every program start would be slow.

16.11.4 PLAIN – what is really happening inside

1. Calls into a shared library do not jump directly. They jump to a small local stub. Real disassembly of `main` in `prog_dyn`:

```
    115b:  e8 f0 fe ff ff   call   1050 <add@plt>

0000000000001050 <add@plt>:
    1050:  f3 0f 1e fa      endbr64
    1054:  ff 25 76 2f 00 00  jmp *0x2f76(%rip)  # 3fd0
```

2. `main` calls a stub called `add@plt`. That stub jumps to whatever address is stored at `0x3fd0`.
3. The table of such addresses is the **global offset table**, the GOT. The table of stubs is the **procedure linkage table**, the PLT.
4. `readelf -r prog_dyn` shows the note that fills slot `0x3fd0`:

```
Relocation section '.rela.plt' contains 1 entry:
  Offset        Type                Sym. Name
  000000003fd0  R_X86_64_JUMP_SLO   add + 0
```

5. **Lazy binding**: by default, the GOT slot initially points back into the PLT, which calls the dynamic linker to resolve the symbol, patch the slot, and jump on. Every later call goes straight through.
6. Setting `LD_BIND_NOW=1` resolves everything up front. On this machine, `LD_DEBUG=statistics LD_BIND_NOW=1 ./prog_dyn` reported 100 relocations and about 155,000 cycles of total dynamic loader start-up time.
7. Lazy binding trades a slightly slower first call for a faster start. Hardened builds turn it off, because a writable GOT is an attack target; `-z now` plus `-z relro` makes the GOT read-only after start-up.

16.11.5 TECHNICAL - the engineer's version

1. Search order for a shared library on glibc: `DT_RPATH` if there is no `DT_RUNPATH`, then `LD_LIBRARY_PATH`, then `DT_RUNPATH`, then `/etc/ld.so.cache`, then the default directories `/lib` and `/usr/lib` with their architecture subdirectories.
2. `DT_RPATH` is deprecated in favour of `DT_RUNPATH`, because `RUNPATH` is overridable by `LD_LIBRARY_PATH` and `RPATH` is not. Our example used `-Wl,-rpath,'$ORIGIN'` and `readelf -d` shows `RUNPATH Library runpath: [$ORIGIN]`. `$ORIGIN` expands to the directory of the executable, which is how self-contained application bundles work.
3. Setting `LD_LIBRARY_PATH` globally in a shell profile is a well-known source of confusing bugs, because it affects every program you start. Prefer `RUNPATH` baked into the binary.
4. `LD_PRELOAD` loads a library before all others, letting you interpose on any function. It is the mechanism behind many profilers and sanitizer shims. For safety, it is ignored for set-user-ID programs.
5. Versioned sonames are how Linux avoids most DLL hell. A library records a `SONAME` such as `libc.so.6`, and glibc additionally versions individual symbols, so `__libc_start_main@GLIBC_2.34` and an older version can coexist in one file. Our binary's relocation table shows exactly that symbol.
6. **Dependency hell** is when two of your dependencies need incompatible versions of a third. **DLL hell** was the Windows form, where an installer overwrote a shared `system32` DLL and broke other programs. Windows answered with side-by-side assemblies from Windows XP, 2001.
7. Static linking avoids all of it and costs disk space and the loss of shared library security updates. Go links statically by default; Rust links its own standard library statically but the system C library dynamically unless you choose a musl target.
8. This chapter has treated ELF, Mach-O and PE only as far as needed to explain linking and loading. Chapter 20 covers the file formats themselves in detail.

Tool	Purpose
`ldd prog`	List shared library deps
`readelf -d prog`	NEEDED, SONAME, RUNPATH
`LD_DEBUG=all ./prog`	Full dynamic linker trace
`LD_DEBUG=bindings`	Which symbol bound where
`ldconfig -p`	Contents of ld.so.cache
`otool -L prog`	Same as ldd, on macOS

16.11.6 WORDS - remember these

Loader — the part of the OS that turns a file into a process — the `execve` path that maps `PT_LOAD` segments and transfers control to the entry point. Dynamic linker — the helper that finds and connects libraries — `ld.so`, named in the `PT_INTERP` segment, which maps dependencies and applies relocations. GOT — the little table of "where is it really" — the global offset table, an array of addresses patched at load or first use. PLT — the little stubs that read that table — the procedure linkage table, one trampoline per imported function. Lazy binding — only look it up the first time it is called — resolving PLT entries on first call rather than at start-up; disabled by `-z now`. RUNPATH — a search path baked into the program — the `DT_RUNPATH` dynamic entry, searched after `LD_LIBRARY_PATH`.

16.12 Interpreters and virtual machines

16.12.1 PLAIN - in simple words

1. Compiling is not the only way to run a program.
2. An **interpreter** reads your program and does what it says, immediately, without ever producing machine code.

3. The simplest kind walks the syntax tree from 16.4. At a + node it evaluates the left child, evaluates the right child, and adds.
4. That is easy to write and slow to run, because it re-inspects the tree every single time round a loop.
5. So most real "interpreted" languages do something in between. They compile your program once into a simple made-up instruction set, then run those.
6. Those instructions are **bytecode**, and the program that runs them is a **virtual machine**.
7. Bytecode is not for any real chip. It is designed to be easy to produce and quick to run in a loop.

16.12.2 PLAIN – a picture in your head

1. Imagine a recipe in a foreign language and a cook who does not read it.
2. Option one: a translator stands beside the cook and translates each line aloud, every time, including every time round a repeated step. That is a tree interpreter.
3. Option two: the translator writes out the whole recipe in simple numbered steps first, and the cook then follows the numbered steps. Translation happens once. That is bytecode.
4. Option three: the translator rewrites the recipe into this exact kitchen's own house notation, tuned for these exact pans, before anyone starts. That is ahead-of-time compilation.

Where this comparison breaks: option two's numbered steps are still not what the kitchen's machines understand. Somebody, the virtual machine, is still reading each numbered step and deciding what to do. That reading is the cost that a JIT, covered in 16.13, removes.

16.12.3 PLAIN – a worked example

Our example in Java, compiled with `javac` and disassembled with `javap -c`:

```
static int add(int, int);
  Code:
     0: iload_0
     1: iload_1
     2: iadd
     3: ireturn

public static void main(java.lang.String[]);
  Code:
     0: iconst_2
     1: iconst_3
     2: invokestatic  #7   // Method add:(II)I
     5: invokestatic  #13  // Method System.exit:(I)V
     8: return
```

1. This is a **stack machine**. `iload_0` pushes local variable 0. `iload_1` pushes local variable 1. `iadd` pops two and pushes their sum. `ireturn` pops and returns.
2. Every opcode is one byte, which is where the name bytecode comes from.
3. No registers are named. That is deliberate: the JVM does not know what chip it is on.
4. The first eight bytes of the `.class` file, from `od -An -tx1 -N 8`, are `ca fe ba be 00 00 00 41`. The first four are the magic number 0xCAFEBABE, chosen at Sun in the early 1990s. `0x41` is 65: Java 21.

And the same idea in CPython 3.11, from the `dis` module:

```
def add(a, b): return a + b
  LOAD_FAST     0 (a)
  LOAD_FAST     1 (b)
  BINARY_OP     0 (+)
  RETURN_VALUE

def main(): return add(2, 3)
```

```
  LOAD_GLOBAL    1 (NULL + add)
  LOAD_CONST     1 (2)
  LOAD_CONST     2 (3)
  PRECALL        2
  CALL           2
  RETURN_VALUE
```

5. Same shape: push, push, operate, return.
6. Python caches this in `__pycache__/m.cpython-311.pyc`, whose first four bytes here are `a7 0d 0d 0a`: a version magic, then a carriage return and newline chosen so that a text-mode file transfer corrupts the magic and the file is rejected rather than misread.
7. The huge difference is what the opcodes mean. `iadd` in Java adds two 32-bit integers, full stop. `BINARY_OP +` in Python must inspect both objects' types at run time and find the right method. That is most of the speed gap.

16.12.4 PLAIN – what is really happening inside

1. A bytecode virtual machine is a loop: fetch the next opcode, jump to the code for it, do it, repeat.
2. The jump is usually a computed jump into a table of handler addresses. The cost per opcode is that jump plus the work.
3. The jump is hard for the CPU's branch predictor, because the next opcode is essentially unpredictable. That mispredict cost is the main tax of interpretation.
4. **Threaded code** reduces it. Instead of one jump at the top of the loop, each handler ends with its own jump to the next handler, giving the predictor more context. CPython uses computed gotos on GCC and Clang for this reason.
5. Bytecode is portable because it names no registers, no addresses and no instruction encodings. One `.class` file runs on x86, AArch64 and anything else with a JVM.
6. It also starts fast, because there is nothing to compile at start-up if the bytecode is already cached, which is exactly what `.pyc` files are for.

16.12.5 TECHNICAL – the engineer's version

1. Java was released by Sun Microsystems in January 1996. The JVM specification defines the class file format, the instruction set, verification rules and the memory model. It is a written standard.
2. The JVM has about 200 defined opcodes out of 256, leaving room reserved for internal use. Class file major versions map to releases: 52 is Java 8, 55 is Java 11, 61 is Java 17, 65 is Java 21.
3. Bytecode verification happens at class load. The verifier proves type safety and stack discipline before any code runs. That is a real security property, and it is why JVM bytecode is not simply machine code in disguise.
4. CPython compiles to bytecode at import and caches it. `__pycache__` and the `.pyc` naming scheme were introduced in Python 3.2 by PEP 3147, 2010. The bytecode format is explicitly unstable between minor releases, which is why the file name includes `cpython-311`.
5. Python 3.11, October 2022, added a specializing adaptive interpreter, PEP 659, which rewrites hot opcodes into type-specialized versions at run time. The CPython team reported roughly 1.25 times faster on their benchmark suite versus 3.10.
6. Rough figures for an arithmetic loop, order of magnitude only, because the real ratio depends heavily on the workload. Treat these as approximate.
   (a) A tree-walking interpreter: roughly 100 to 1000 times slower than C.
   (b) A bytecode VM with no JIT: roughly 10 to 100 times slower.
   (c) A good JIT: within about 1 to 3 times.

Trait	Compiled AOT	Bytecode VM	Tree walker
Start-up	Fastest	Medium	Fast
Peak speed	Fastest	Near AOT, if JIT	Slowest
Portability	Rebuild per CPU	One artifact	One artifact
Debug info	Needs symbols	Built in	Built in
Examples	C, Rust, Go	Java, C#, Python	Early Ruby, bash

16.12.6 WORDS - remember these

Interpreter — a program that carries out your program directly — an evaluator that executes source or IR without emitting native code. Bytecode — compact instructions for a made-up machine — a serialized instruction set for a virtual machine, typically one byte per opcode. Virtual machine — the program that runs bytecode — an abstract machine specification plus its implementation, such as the JVM or CPython's `ceval` loop. Stack machine — a machine with no named registers — an architecture where operands are pushed and popped on an operand stack. Dispatch — choosing which handler runs next — the fetch-decode-jump step of an interpreter loop, often implemented with computed gotos.

16.13 Just-in-time compilation

16.13.1 PLAIN - in simple words

1. A **just-in-time** compiler, or JIT, compiles your program while it is already running.
2. It does not compile everything. Most code runs once or twice and is not worth the effort.
3. So the system counts. Every time a function is entered or a loop goes round, a counter goes up.
4. When a counter crosses a threshold, that code is **hot**, and the JIT compiles it to real machine instructions.
5. The next time it is reached, the fast version runs instead.
6. A JIT knows something an ordinary compiler cannot: what actually happened. It knows which types really turned up, which branch really got taken, and which function really got called.
7. So it compiles a specialized version based on those observations, and adds a cheap check that the observation still holds.
8. If the check ever fails, it throws that version away and goes back to the slow safe path. That is called **deoptimization**.

16.13.2 PLAIN - a picture in your head

1. Imagine a new receptionist at a large office. On day one they look up every visitor's destination in the directory.
2. After a week they notice that nearly everyone who arrives asks for the third floor, so they start saying "third floor" before checking.
3. But they keep a quick glance at the visitor's badge, in case somebody different turns up.
4. If a badge does not match, they stop guessing, go back to the directory, and look it up properly for that visitor.
5. Guessing is a huge win only because it is nearly always right and because being wrong is safe.

Where this comparison breaks: the receptionist keeps their memory. A real JIT must be able to abandon an optimized version mid-execution, reconstruct the exact state the slow interpreter would have had at that point, and resume. Getting that reconstruction correct is one of the hardest parts of JIT engineering.

16.13.3 PLAIN – a worked example

Take our `add`, but in a dynamic language, where `a + b` could mean anything.

1. First call: the interpreter runs `BINARY_OP +`. It looks at the type of `a`, looks at the type of `b`, finds both are whole numbers, and adds them.
2. The system records what it saw: two whole numbers, at this exact spot.
3. Calls two to about a thousand: the same thing, and the recording is confirmed each time.
4. The counter crosses the threshold. The JIT compiles this function assuming both arguments are whole numbers.
5. It emits something close to our `-O2` output: a check, then one `add` instruction, then a return.
6. The check is a **guard**: "is `a` really a whole number, is `b`?" A few instructions, usually predicted correctly, essentially free.
7. Call 100,000: someone passes a string. The guard fails. The compiled version is abandoned for that call, the state is rebuilt, and the interpreter takes over from that exact point.
8. If it keeps failing, the compiled version is discarded and recompiled with the new information.

Real tiering in the HotSpot JVM, which is a written design, not a guess:

Tier	What runs	Roughly when
0	Interpreter	Always at first
1–3	C1 compiler	After ~1,500 invocations
4	C2 compiler	After ~10,000

The exact thresholds are controlled by `-XX:CompileThreshold` and related flags and vary between JVM versions; treat the numbers as typical defaults rather than fixed values.

16.13.4 PLAIN – what is really happening inside

1. Compilation happens on a background thread, so the program keeps running on the old version while the new one is prepared.
2. **On-stack replacement**, or OSR, handles the awkward case of a loop that is already running and will not be entered again. The system swaps the running frame over to compiled code partway through.
3. **Inline caching**, invented for Smalltalk by L. Peter Deutsch and Allan Schiffman in 1984, remembers at each call site which method was called last time, so the lookup is skipped.
4. A **polymorphic inline cache**, from Urs Hölzle, Craig Chambers and David Ungar's work on Self in 1991, remembers several possibilities at one site.
5. **Hidden classes**, also from Self, give dynamically shaped objects a fixed internal layout, so a field access can become a single load at a fixed offset instead of a dictionary lookup.
6. Those three ideas together are the reason JavaScript became fast. They were research results from the 1980s and 1990s that were finally applied to the web in the 2008 to 2012 period.

16.13.5 TECHNICAL – the engineer's version

1. HotSpot came to Sun through the 1997 acquisition of Animorphic and shipped in 1999. It introduced tiered compilation with the C1 client compiler and the C2 server compiler, plus deoptimization back to the interpreter.
2. Google's V8 shipped with Chrome in September 2008, developed in Aarhus under Lars Bak, who had worked on Self and HotSpot. The lineage is direct.
3. V8's pipeline as of 2026 is Ignition, a bytecode interpreter, then Sparkplug, a baseline compiler added in 2021, then Maglev, a mid-tier compiler added in 2023, then TurboFan. Ignition plus TurboFan replaced Crankshaft in Chrome 59, 2017.

4. Mozilla's SpiderMonkey, the original JavaScript engine written by Brendan Eich in 1995, went through TraceMonkey in 2008, and now uses Baseline plus IonMonkey with WarpBuilder.
5. Deoptimization needs a precise mapping from optimized machine state back to interpreter state at every safepoint. This is stored as side tables and is a large part of why JIT compilers are hard to write correctly.
6. The trade against ahead-of-time compilation:
   (a) JIT wins on peak speed for dynamic languages, because it can specialize on observed types and inline through virtual calls speculatively.
   (b) AOT wins on start-up time, on memory, on predictable latency, and on platforms that forbid writable-and-executable memory, such as iOS.
   (c) JIT costs memory for the compiler, the profiling data and the code cache.
   (d) JIT hurts tail latency, because compilation and deoptimization happen at unpredictable moments. This matters for trading systems and for games.
7. Hybrid approaches now dominate. Java has AppCDS class data sharing and GraalVM native-image ahead-of-time compilation, first released 2019. Android moved from Dalvik's pure JIT to ahead-of-time ART in Android 5.0, 2014, then to a profile-guided mix in Android 7.0, 2016.
8. Established fact: JIT reliably beats plain interpretation for long-running dynamic code. Active research: cheap start-up, predictable latency, and sharing profiles across runs. Marketing claim: any flat statement that a JIT language is "as fast as C".

16.13.6 WORDS - remember these

JIT — compiling while the program runs — dynamic translation of hot code to native instructions, guided by run-time profiles. Hot path — the code that runs most — a method or loop whose execution counter has crossed a compilation threshold. Tiered compilation — several compilers of increasing quality — a pipeline from interpreter to baseline to optimizing compiler, per method. Deoptimization — abandoning fast code when a guess turns out wrong — reconstructing interpreter state at a safepoint and resuming in the slower tier. Inline cache — remembering what was called here last time — a per-call-site cache of receiver type to target method, monomorphic or polymorphic. On-stack replacement — swapping code under a running loop — transferring an active frame from interpreted to compiled code, or back.

16.14 Modern build reality

16.14.1 PLAIN - in simple words

1. In real projects, nobody types `gcc` by hand.
2. A **build system** works out what needs rebuilding and runs the commands.
3. It does this by comparing times: if the source is newer than the output, the output is stale and must be rebuilt.
4. Anything that depended on that output is stale too, and so on up the chain.
5. Rebuilding only what changed is an **incremental build**, and it is the difference between a two-second edit-run cycle and a twenty-minute one.
6. A **cross-compiler** runs on one kind of machine and produces code for another. That is how phone apps are built on laptops.
7. A **transpiler** compiles from one high-level language to another, rather than to machine code.
8. A **toolchain** is the whole matched set: compiler, assembler, linker, standard library and headers for one target.

16.14.2 PLAIN - a picture in your head

1. Think of a kitchen preparing a large set menu.
2. Some dishes depend on a stock that takes an hour. Some depend on that stock's sauce. Some depend on nothing.

3. A good kitchen keeps a chart of what depends on what.
4. If the stock is remade, everything downstream of it must be remade. If only a garnish changed, only the garnish is redone.
5. A busy kitchen also keeps a fridge of finished components labelled with exactly which ingredients went into them. If someone orders the same component again with identical ingredients, it comes out of the fridge. That is a build cache.

Where this comparison breaks: a kitchen can smell that the stock has gone off. A build system has only timestamps and hashes. If a timestamp lies, for example because clocks differ between machines or a file was restored from backup, the build system will confidently reuse something wrong. This is a real and common class of bug.

16.14.3 PLAIN - a worked example

Here is our two-file version, driven by `make`.

```
prog: main2.o addonly.o
    gcc main2.o addonly.o -o prog
main2.o: main2.c addh.h
    gcc -c main2.c -o main2.o
addonly.o: addonly.c
    gcc -c addonly.c -o addonly.o
```

Real session on this machine:

```
$ make
gcc -c main2.c -o main2.o
gcc -c addonly.c -o addonly.o
gcc main2.o addonly.o -o prog

$ make
make: 'prog' is up to date.

$ touch addonly.c && make
gcc -c addonly.c -o addonly.o
gcc main2.o addonly.o -o prog

$ touch addh.h && make
gcc -c main2.c -o main2.o
gcc main2.o addonly.o -o prog
```

1. The first build compiles both files.
2. The second build does nothing, because nothing is newer than its output.
3. Touching `addonly.c` rebuilds only that object, then relinks.
4. Touching the header rebuilds `main2.o`, because the rule lists `addh.h` as a dependency. That listing is the whole trick, and forgetting it is the classic cause of "it did not pick up my header change". Generate it automatically with `gcc -MMD`.

16.14.4 PLAIN - what is really happening inside

1. Make builds a graph of files and rules, sorts it so dependencies come first, and runs any rule whose output is older than any of its inputs.
2. Modern build systems replace timestamps with **content hashes**, which is more reliable: the same input bytes plus the same command always give the same output.
3. That is what makes a **build cache** possible. Hash the compiler version, the flags and the preprocessed source; if that hash was seen before, reuse the stored object file. `ccache` does exactly this locally, and Bazel and Gradle do it across a whole team.

4. **Cross-compilation** works because the pipeline in 16.1 is already split. The front end does not care about the target. Only the back end, the assembler, the linker and the libraries do.
5. Our chapter already did it. `clang --target=aarch64-linux-gnu -O2 -S add.c` produced ARM instructions on an x86 machine, with no special install, because LLVM ships every backend in one binary.
6. To produce a runnable ARM binary you also need ARM headers and libraries. That is the difference between a compiler and a toolchain, and it is why cross-toolchain setup is more annoying than cross-compiling itself.

16.14.5 TECHNICAL - the engineer's version

1. `make` was written by Stuart Feldman at Bell Labs in 1976, reportedly after losing a morning to a program that had failed to relink. Its tab-indentation rule has never been fixed for compatibility reasons.
2. `CMake` was created at Kitware around 2000 for the Insight Toolkit, funded by the US National Library of Medicine. It is a **build generator**: it does not build, it writes Makefiles, Ninja files or Visual Studio projects.
3. `Ninja`, written by Evan Martin for the Chrome build in the early 2010s, is deliberately not human-authored. Its input is machine-generated, its dependency handling is fast, and it is the usual back end for CMake and for Meson on large projects.
4. `Gradle` targets the JVM world, uses a Groovy or Kotlin domain-specific language, supports incremental and cached tasks, and reached 1.0 in 2012. `Bazel` is the open-sourced version of Google's Blaze, released in 2015, and is built around hermetic, hash-keyed, remotely cacheable actions.
5. `ccache`, from Andrew Tridgell in the early 2000s, and `distcc`, from Martin Pool around the same time, remain the cheapest large wins for C and C++ projects that cannot adopt Bazel.
6. **Transpilers**: TypeScript, announced by Microsoft in October 2012 under Anders Hejlsberg, compiles to JavaScript. Babel, started as 6to5 by Sebastian McKenzie in 2014, compiles newer JavaScript to older JavaScript. The very first C++ implementation, Bjarne Stroustrup's Cfront from 1983, was a transpiler to C.
7. **WebAssembly** is a portable compilation target with a formal specification. Its Core Specification 1.0 became a W3C Recommendation in December 2019.
   (a) Our example built with `clang --target=wasm32 -O2 -nostdlib -c add.c` gave a 333-byte module.
   (b) Its first eight bytes are `00 61 73 6d 01 00 00 00`, that is `\0asm` followed by version 1.
8. Why build times matter: they set the length of the edit-compile-test loop, which sets how often a developer tries an idea. As a scale marker, a full Chromium build is commonly reported in hours on one workstation and minutes with a large distributed cache.

Tool	Year	Kind
make	1976	Build executor
Cfront	1983	C++ to C transpiler
CMake	2000	Build generator
ccache	early 2000s	Compiler cache
Ninja	early 2010s	Build executor
Gradle 1.0	2012	Build system, JVM
TypeScript	2012	Transpiler
Bazel	2015	Build system, cached
Wasm 1.0	2019	Portable target

16.14.6 WORDS - remember these

Build system — the thing that decides what to recompile — a dependency graph executor keyed on timestamps or content hashes. Incremental build — rebuild only what changed — recomputing the minimal set of stale targets from the dependency graph. Build cache — reuse a result somebody already computed

— a content-addressed store keyed on inputs, tool version and flags. Cross-compilation — building on one machine for another — using a toolchain whose target triple differs from the host triple. Transpiler — a compiler whose output is another language — a source-to-source translator, such as TypeScript to JavaScript. Toolchain — the whole matched set of tools — compiler, assembler, linker, headers and run-time libraries for one target.

16.15 The bootstrap problem and trusting trust

16.15.1 PLAIN - in simple words

1. A C compiler is a program. Programs must be compiled. So what compiled the first C compiler?
2. This is the **bootstrap problem**, and there are only three honest answers.
3. One: write the first version in something else, such as assembly or another existing language.
4. Two: write a small, limited version by hand, then use it to compile a bigger version, then use that to compile a bigger one still.
5. Three: use a compiler on a different machine to produce code for yours. That is cross-compilation from 16.14.
6. A compiler written in the language it compiles is **self-hosting**. Almost every serious compiler ends up this way.
7. Self-hosting is a strong test: the compiler is its own largest and most demanding user.

16.15.2 PLAIN - a picture in your head

1. Imagine you want a workshop that can build any tool, including its own tools.
2. You cannot start there. You start with a rough hammer made by hand.
3. With the rough hammer you make a better hammer. With the better hammer you make a decent lathe. With the lathe you make precise tools.
4. Eventually the workshop can make every tool in it, including exact copies of its own machines.
5. The first rough hammer is then thrown away, and nobody alive has seen it.

Where this comparison breaks: a hammer's shape is visible. A compiler can carry an invisible instruction inside it that is not written in any source file anybody keeps, and that instruction can copy itself into every tool the workshop makes. That is the whole point of the next block.

16.15.3 PLAIN - a worked example

1. Start with our own file. `gcc add.c` worked because a `gcc` binary already existed on this machine. That binary is itself written in C and C++, so some earlier compiler must have built it. Follow that chain back far enough and you reach the question of where the first one came from.
2. Around 1972 to 1973, Dennis Ritchie's C compiler at Bell Labs was written in an earlier language, first B and then an intermediate step usually called NB or "new B", and grew into C by stages, each version compiling the next.
3. In 1973 the Unix kernel was rewritten in C for Version 4 Unix, which is the moment portable operating systems become possible.
4. Corrado Böhm's 1951 PhD thesis at ETH Zurich described the first compiler for a language written in that same language, so the idea is older than C by two decades.
5. Tim Hart and Mike Levin wrote a Lisp compiler in Lisp at MIT in 1962, then ran it through the Lisp interpreter to compile itself.
6. Modern examples: the Rust compiler became self-hosted in April 2011. Go removed the last of its C compiler in Go 1.5, August 2015, and has been written in Go since.
7. Today's route for a brand-new C compiler is easier. Compile version one with GCC, compile version one with itself, compile once more, and check the last two outputs are byte-identical. That is a three-stage bootstrap, and GCC does it in its own build.

16.15.4 PLAIN - what is really happening inside

1. Now Ken Thompson's argument, in plain words. He gave it in his 1984 Turing Award lecture, *Reflections on Trusting Trust*, printed in Communications of the ACM in August 1984. He and Dennis Ritchie shared the 1983 Turing Award.
2. Step one. Suppose you modify a C compiler so that when it notices it is compiling the login program, it silently adds a back door accepting a secret password.
3. That is easy but obvious: the extra code is right there in the compiler's source, and any reviewer would see it.
4. Step two. So add a second rule: when the compiler notices it is compiling a C compiler, it silently inserts both rules into the output.
5. Now compile the compiler with the modified compiler. The new binary contains both rules.
6. Step three. Remove all of it from the source. Delete every line. The source is clean and reviewable and contains nothing suspicious.
7. But the compiled compiler still carries both rules, because it was built by a compiler that inserts them. Compile the clean source with that binary, and the new binary has them again.
8. The attack now lives only in the binary, reproducing itself forever, with no trace in any source file anybody reads.
9. Thompson's conclusion: you cannot trust code you did not totally write yourself, and "totally" reaches all the way down through the compiler, the assembler, the linker, the loader, the operating system and the hardware.
10. He also noted that no amount of source-level inspection protects you, because the source is genuinely clean.

16.15.5 TECHNICAL - the engineer's version

1. The paper's technique is a quine-like self-reproducing program combined with two pattern-matching triggers. Thompson stated that he had actually built a working version, though it was never released.
2. The practical defence is **diverse double-compiling**, from David A. Wheeler's 2005 paper and 2009 PhD dissertation. Compile the suspect compiler's source with a second, independently written compiler, then use both binaries to compile that source again. Identical final binaries mean neither carries the attack.
3. This only works if the compiler is deterministic: the same source and flags must always give byte-identical output. That requirement is precisely the goal of the **Reproducible Builds** project, which began within Debian around 2013.
4. Sources of non-determinism that have to be removed: embedded build timestamps, build paths in debug information, file ordering from directory reads, random hash seeds and thread scheduling. GCC and Clang both accept `SOURCE_DATE_EPOCH` and `-ffile-prefix-map` to help.
5. As of 2026, Debian reports that the large majority of its source packages build reproducibly, and several distributions publish rebuild verification. Exact percentages move month to month, so treat any single figure as dated.
6. The related project **Bootstrappable Builds** attacks the other end: shrinking the trusted seed binary to something a human can audit. Its chain starts from a few hundred bytes of hand-checkable machine code and climbs through stage0, M2-Planet, TinyCC and GCC.
7. Established fact: the attack works and is well understood. Active research: verifying the whole chain from hardware upward, including proved compilers such as CompCert, machine-checked in Coq since 2008. Marketing claim: any product said to have solved supply-chain trust.
8. The honest version: reproducible builds and diverse double-compiling reduce the trusted base, they do not eliminate it. You still trust the CPU microcode, the firmware and the fabrication process. Thompson's point survives; we have only made the untrusted part smaller.

16.15.6 WORDS - remember these

Bootstrapping — getting a compiler off the ground with no compiler — building a language's compiler through successive stages from an existing toolchain. Self-hosting — a compiler written in its own language — a compiler that can compile its own source, usually verified by a multi-stage build. Three-stage bootstrap — build it three times and compare — stage 2 and stage 3 outputs must be byte-identical, a standard part of the GCC build. Trusting trust — the compiler can lie and the source will not show it — Thompson's 1984 self-reproducing compiler back door. Reproducible build — same source in, same bytes out, every time — a build whose output is a deterministic function of its declared inputs. Diverse double-compiling — check one compiler using a different one — Wheeler's method for detecting a trusting-trust attack in a deterministic compiler.

16.98 Common wrong ideas

1. Wrong: the compiler translates your code line by line into machine code. Right: it goes through the whole chain of 16.1, and the optimizer may delete, merge, reorder and duplicate your lines. Our two-line file became three instructions with no call in it.
2. Wrong: `#include` imports a module the way Python's `import` does. Right: it pastes the file's text at that point. Two lines of C with `#include <stdio.h>` became 815 lines of text on this machine.
3. Wrong: a syntax error means the compiler does not understand what you meant. Right: a syntax error means no grammar rule allows the token it just saw. Meaning is checked later, in semantic analysis, and produces different errors.
4. Wrong: undefined reference is a compiler error. Right: it is a linker error. The compiler was satisfied by the declaration. The linker could not find a definition. That is why the fix is usually an extra object file or library, not a code change.
5. Wrong: `-O2` makes your program do the same steps, faster. Right: it makes the program produce the same observable result by doing different steps. Our `main` never calls `add` at all after `-O2`.
6. Wrong: undefined behaviour means the program does something unpredictable at that point. Right: it means the compiler may assume it never happens, which can change code far away from the mistake. `x + 1 > x` folded to a constant `1` at `-O2`.
7. Wrong: floating point differences between `-O0` and `-O2` are compiler bugs. Right: they can be legitimate, especially with `-ffast-math`, because IEEE 754 addition is not associative. On this machine `(a+b)+c` gave 1 and `a+(b+c)` gave 0 for the same three values.
8. Wrong: interpreted languages are slow because interpreting is slow. Right: mostly they are slow because operations must decide their meaning at run time. Java bytecode is interpreted too and is far faster, because `iadd` already knows it is adding two 32-bit integers.
9. Wrong: a JIT is always faster than ahead-of-time compilation. Right: it wins on peak speed for dynamic code and loses on start-up time, memory use and latency predictability. That is why iOS forbids it and why GraalVM native-image exists.
10. Wrong: reading a compiler's source proves it has no back door. Right: Thompson showed in 1984 that a compiler binary can carry an attack that appears in no source file. Diverse double-compiling, not reading, is the defence.

16.99 Chapter summary in 20 lines

1. A compiler is a chain of stages, each turning one shape of information into a simpler one, from text to running process.
2. The preprocessor is a text editor: it pastes files in and substitutes macros, and it knows nothing about C.
3. The lexer cuts characters into labelled tokens using a finite automaton, and always takes the longest legal match.

4. The parser arranges tokens into an abstract syntax tree, guided by a context-free grammar written in Backus-Naur Form.
5. A syntax error means no grammar rule allows the current token. It is about shape, not meaning.
6. Semantic analysis builds a scoped symbol table, resolves names, checks types and inserts implicit conversions.
7. Compilers then rewrite the program into an intermediate representation: flat three-address code, usually in static single assignment form.
8. SSA gives every value exactly one definition, with phi nodes at joins, which makes most optimizations simple and fast.
9. The front-end, middle-end, back-end split lets many languages share one optimizer and one set of processor backends.
10. The optimizer may do anything that preserves observable behaviour: fold, propagate, delete, inline, unroll, vectorize, and eliminate tail calls.
11. Our `int main(void) { return add(2,3); }` became `movl $5, %eax; ret` through inlining, then constant folding, then dead code elimination.
12. Undefined behaviour lets the optimizer assume the impossible never happens, which is why `x + 1 > x` folds to `1` for signed integers but not unsigned.
13. Code generation performs instruction selection, scheduling and register allocation, and must obey a calling convention.
14. System V AMD64 passes the first two integers in rdi and rsi, Windows x64 uses rcx and rdx, AArch64 uses x0 and x1. All return in the first of those.
15. The assembler turns instruction text into bytes, groups them into sections, records symbols, and leaves relocations where addresses are unknown.
16. The linker merges sections, resolves symbols and patches relocations. Our `e8 00 00 00 00` became `e8 d1 ff ff ff`, a relative jump of minus 47 bytes.
17. Library order on the command line matters, because archives are scanned once against the symbols still undefined at that moment.
18. At exec time the kernel maps segments and starts the dynamic linker, which loads shared libraries and fills the GOT, lazily by default.
19. Bytecode virtual machines trade speed for portability, and a JIT wins it back by compiling hot paths using types observed at run time, with deoptimization as the safety net.
20. Compilers bootstrap themselves, and Ken Thompson showed in 1984 that a compiler binary can hide a back door that no source file reveals, which is why reproducible and diverse builds matter.

17

CHAPTER

Programming Languages and How to Think in Them

17.0 What this chapter gives you

1. You will be able to say what a programming language actually is, and why there are thousands of them instead of one.
2. You will be able to read a small program in six different languages and see that they are all saying the same thing.
3. You will be able to explain what a type is, why static and dynamic typing argue with each other, and what the billion dollar mistake was.
4. You will be able to describe the three ways a program can manage memory, and name the exact failure each one prevents and each one allows.
5. You will be able to name the main programming paradigms and write the same task in two of them.
6. You will be able to pick the right data structure for a job and state its cost in Big-O notation without looking it up.
7. You will be able to explain why concurrency is genuinely hard, and show a race condition line by line.
8. You will be able to write a test, read someone else's code, and judge a dependency before you install it.
9. You will be able to choose a language for a real project and defend the choice with reasons instead of taste.

17.1 What a programming language is

17.1.1 PLAIN – in simple words

1. A programming language is a way of writing down instructions that both a person and a machine can work with.
2. It has a **grammar**: rules about which arrangements of symbols are legal.
3. It has a **meaning**: rules about what each legal arrangement does.
4. Those two things are separate. A sentence can be perfectly formed and still say something silly or wrong.
5. The grammar is called **syntax**. The meaning is called **semantics**.
6. A language is designed for humans first. Machines only understand numbers, and something else turns your text into those numbers.
7. That something else is a compiler or an interpreter. Chapter 16 covered how a compiler works. Here we care about the language itself.
8. A language is not the same as its tools. C is a written standard. GCC and Clang are two programs that implement it.

17.1.2 PLAIN - a picture in your head

1. Think of a knitting pattern.
2. A pattern has a strict notation: `k2, p2, rep to end`. Every symbol means one exact action.
3. The notation has a grammar. `k2` is legal. `k` on its own with no number is not, because the pattern must say how many.
4. The notation also has a meaning. `k2` means knit two stitches. Follow it and a real object appears.
5. Two knitters in different countries use different pattern notations. Both make the same jumper. The garment is the program, the notation is the language.
6. A pattern can be written correctly and still produce a jumper with three sleeves. The notation was obeyed. The intention was wrong.

Where this comparison breaks: a knitting pattern is followed by a patient human who silently fixes obvious mistakes. A computer fixes nothing. It does exactly what you wrote, at billions of steps per second, including the wrong thing. A pattern also has no way to say "repeat until the wool runs out and then decide what to do next". Real languages do, and that ability to make decisions is what separates a program from a recipe.

17.1.3 PLAIN - a worked example

1. Here are three lines of Python. Each is broken in a different way.

```
print("hello"
x = 10 / 0
average = total / count
```

2. Line 1 is a **syntax error**. The closing bracket is missing. The grammar was violated. Python refuses to run the file at all.
3. Line 2 is legal grammar but impossible meaning. Division by zero. Python starts running, reaches this line, and stops with an error.
4. Line 3 has perfect grammar and perfect meaning. It will run happily. But if `count` is the number of rows in the file and `total` is the sum of a different column, the answer is nonsense and nothing complains.
5. That third case is the one that costs money. No tool caught it because nothing was broken. The program did exactly what you said.
6. This is the whole of programming in one example: getting the syntax right is easy and the machine helps you. Getting the meaning right is hard and you are mostly on your own.

17.1.4 PLAIN - what is really happening inside

1. When you save a source file you have saved plain text. Bytes. Nothing more.
2. A tool reads that text left to right and chops it into **tokens**: names, numbers, symbols, keywords.
3. It then checks whether those tokens form a legal shape, using the grammar. This is parsing, and it builds a tree of the program's structure.
4. If the shape is illegal you get a syntax error, with a line number.
5. If the shape is legal, the tool works out what it means, and produces either machine instructions or a set of actions it will perform itself.
6. Because a language is a written definition and not a program, several different tools can implement it. Python has CPython, PyPy and others.
7. This is why "the language is slow" is usually the wrong sentence. The implementation is slow. The language is a document.
8. Why thousands of languages exist: every language is a set of trade-offs. Fast to write against fast to run. Safe against flexible. Small against expressive. Nobody has found one point that wins everywhere.
9. A new language is also cheap to start and expensive to finish. Thousands are started. About twenty are widely used at any moment.

17.1.5 TECHNICAL - the engineer's version

1. Syntax is normally specified as a context-free grammar in Backus-Naur Form. John Backus described the notation in 1959 for ALGOL 58, and Peter Naur refined it in the ALGOL 60 report of 1960, which is why it carries both names.
2. Noam Chomsky published the hierarchy of formal grammars in 1956. Type 3 is regular (what a lexer handles), type 2 is context-free (what a parser handles), type 1 is context-sensitive, type 0 is unrestricted.
3. Real languages are not purely context-free. C requires a symbol table to decide whether `A * B;` is a multiplication or a pointer declaration. This is called the lexer hack, and it is an implementation detail, not a standard.
4. Semantics has three classical formal styles: operational (Gordon Plotkin's structural operational semantics, 1981), denotational (Dana Scott and Christopher Strachey, around 1970), and axiomatic (Tony Hoare, in the October 1969 paper "An Axiomatic Basis for Computer Programming").
5. In practice most language specifications define semantics in careful English prose, not in formal mathematics. Standard ML and WebAssembly are among the very few with a fully formal specification.
6. Undefined behaviour is a specification concept, not a bug: the standard declines to say what happens, so the compiler may assume it never occurs. Signed integer overflow in C is the classic case.

Language	Governing document	Latest as of Aug 2026
C	ISO/IEC 9899	C23, published 2024
C++	ISO/IEC 14882	C++26, March 2026
JavaScript	ECMA-262	ECMAScript 2025
SQL	ISO/IEC 9075	SQL:2023
Python	PEPs, no ISO standard	3.14, Oct 2025

7. There is no census of languages. The Online Historical Encyclopaedia of Programming Languages lists roughly 9,000 entries, which is an approximate figure and includes long-dead research languages.
8. Tools that show you the grammar in action: `python -m ast`, `gcc -fdump-tree-original`, `clang -Xclang -ast-dump`, and `node --print-ast` style flags in various runtimes.

17.1.6 WORDS - remember these

1. Syntax — the spelling and shape rules — the context-free grammar of the language.
2. Semantics — what it means when it runs — the operational or denotational definition of program behaviour.
3. Token — one word or symbol — the smallest unit produced by lexical analysis.
4. Parsing — checking the shape — building an abstract syntax tree from tokens according to the grammar.
5. Specification — the official rulebook — the ISO, ECMA or community document that defines the language, separate from any implementation.
6. Undefined behaviour — the rules do not say — a construct the standard leaves unconstrained, allowing any compiler output at all.

17.2 The building blocks every language has

17.2.1 PLAIN - in simple words

1. Almost every language is built from the same ten or so ideas.
2. A **variable** is a named box that holds a value you can change.
3. A **literal** is a value written down directly, like `42` or `"cat"`.
4. An **operator** is a symbol that combines values, like `+` or `<`.
5. An **expression** is anything that produces a value: `2 + 3`, `name`, `total > 100`.

6. A **statement** is a complete instruction that does something: assign, print, return.
7. A **conditional** chooses between paths: if this, do that, otherwise do the other.
8. A **loop** repeats a block, either a fixed number of times or until some test stops being true.
9. A **function** is a named piece of program you can call from elsewhere, optionally handing it values and getting one back.
10. **Scope** is the region of the program where a name is visible.
11. A **comment** is text the machine ignores completely, written for humans.

17.2.2 PLAIN – a picture in your head

1. Think of a kitchen with labelled jars on a shelf.
2. A jar is a variable. The label is the name, the contents are the value.
3. A literal is an ingredient you bring in fresh, not from a jar.
4. An operator is a kitchen action: mix these two, compare these two.
5. A recipe step that produces something you can hold is an expression. A step that just says "turn on the oven" is a statement.
6. A conditional is "if the dough is sticky, add flour".
7. A loop is "stir for ten minutes".
8. A function is a sub-recipe on its own card: "make the sauce". You can use it from any recipe without rewriting it.
9. Scope is which shelf you can reach from where you are standing. Jars in the locked pantry are out of scope.

Where this comparison breaks: a cook can ignore a step or improvise. A program cannot. Also, a real recipe never calls itself, and functions often do, which is called recursion and has no clean kitchen equivalent.

17.2.3 PLAIN – a worked example

1. Here is the same tiny program in six languages. It adds the numbers 1 to 10 and prints 55.
2. Read them one after another. Notice that the punctuation changes and nothing else does.

```c
#include <stdio.h>

int main(void) {
    int total = 0;
    for (int i = 1; i <= 10; i++) {
        total = total + i;
    }
    printf("%d\n", total);
    return 0;
}
```

```python
total = 0
for i in range(1, 11):
    total = total + i
print(total)
```

```javascript
let total = 0;
for (let i = 1; i <= 10; i++) {
  total = total + i;
}
console.log(total);
```

```java
public class Sum {
    public static void main(String[] args) {
        int total = 0;
        for (int i = 1; i <= 10; i++) {
```

```
            total = total + i;
        }
        System.out.println(total);
    }
}
```

```
package main

import "fmt"

func main() {
    total := 0
    for i := 1; i <= 10; i++ {
        total = total + i
    }
    fmt.Println(total)
}
```

```
fn main() {
    let mut total = 0;
    for i in 1..=10 {
        total = total + i;
    }
    println!("{}", total);
}
```

3. Every one of them declares a variable called `total`, sets it to zero, loops `i` from 1 to 10, adds `i` to `total`, and prints the result.
4. The differences are: whether you write the type (`int`) or not, whether lines end in a semicolon, whether blocks use braces or indentation, and how much ceremony wraps the main body.
5. Java has the most ceremony because every piece of code must live inside a class. Python has the least because the file itself is the program.
6. Rust needs `mut` because in Rust a variable cannot be changed unless you say so explicitly.
7. Go uses `:=` to mean "make a new variable and work out its type for me".
8. If you can read one of these, you can read all six with about an hour of effort. That is the real point of this section.

17.2.4 PLAIN - what is really happening inside

1. A variable is a name the compiler maps to a place: a CPU register, a slot on the stack, or an address in the heap.
2. In `total = total + i`, the machine loads two values, adds them in a register, and stores the result back. Three steps, one line of your text.
3. `i++` is one instruction on most processors. `total = total + i` is usually one instruction too, once the values are in registers.
4. A loop is a conditional jump. At the bottom of the block, compare `i` with 10, and if the test passes, jump back to the top.
5. A function call pushes the return address and arguments somewhere the callee can find them, jumps to the function's code, and later jumps back.
6. Scope is a compile-time idea, not a runtime one. The compiler simply refuses to resolve a name that is not visible, and produces an error.
7. Comments never reach the machine. The lexer throws them away before the parser ever sees them.
8. Here is what the loop becomes in spirit:

```
    total <- 0
    i <- 1
top:
```

```
    if i > 10 goto done
    total <- total + i
    i <- i + 1
    goto top
done:
    print total
```

9. Every one of the six languages above compiles or interprets down to something with this shape. The high-level `for` is a convenience.

17.2.5 TECHNICAL – the engineer's version

1. Expressions have precedence and associativity, fixed by the grammar. In C, `a + b * c` parses as `a + (b * c)` because * binds tighter. `a - b - c` parses as `(a - b) - c` because - is left-associative.
2. Statement versus expression is a real language design axis. In Rust and in most functional languages, `if` is an expression and returns a value. In C and Java it is a statement and returns nothing, which is why those languages need a separate ternary operator `? :`.
3. Scope rules split into lexical (static) and dynamic. Almost every modern language uses lexical scope: a name resolves by where it is written. Emacs Lisp and Bash use dynamic scope in places, where a name resolves by who called you.
4. Variable lifetime is separate from scope. A C `static` local has function scope but program lifetime.
5. Parameter passing conventions: call by value (C, Java for primitives, Go), call by reference (C++ with &, C# with `ref`), and call by sharing (Python, JavaScript, Java for objects, where the reference is copied but the object is not).
6. This is the source of a common confusion. In Python, reassigning a parameter inside a function does not affect the caller, but calling `list.append` on it does. Both are consistent with call by sharing.

Construct	C	Python	Rust
Block delimiter	braces	indentation	braces
Statement end	semicolon	newline	semicolon
Declare variable	`int x = 1;`	`x = 1`	`let x = 1;`
Mutable by default	yes	yes	no

7. Tools: `python -m dis` shows the bytecode a Python function becomes. `godbolt.org` style compiler explorers show the assembly for C, C++, Rust and Go. `javap -c` disassembles a Java class file.

17.2.6 WORDS – remember these

1. Variable — a named box for a value — a binding from an identifier to a storage location with a scope and lifetime.
2. Literal — a value written directly — a constant token whose value is fixed at compile time.
3. Expression — something that produces a value — a grammar production that evaluates to a value of some type.
4. Statement — an instruction that acts — a grammar production executed for its effect rather than its value.
5. Scope — where a name can be seen — the lexical region in which an identifier binding is visible.
6. Call by sharing — the object is shared, the variable is not — the argument reference is copied, so mutation is visible and rebinding is not.

17.3 Types

17.3.1 PLAIN – in simple words

1. A **type** is the answer to "what kind of thing is this value".
2. `42` is a whole number. `"42"` is a piece of text. `4.2` is a decimal number. They look similar and behave completely differently.
3. Types exist to stop you doing meaningless things, like subtracting a name from a date.
4. They also tell the machine how many bytes to set aside and which instructions to use. Adding two whole numbers uses different hardware from adding two decimals.
5. Some languages check types before the program runs. That is **static typing**.
6. Some languages check types while the program runs. That is **dynamic typing**.
7. Some languages refuse to mix types silently. That is **strong typing**.
8. Some languages quietly convert one type into another to make an expression work. That is **weak typing**.
9. Static and strong are different questions. A language can be any of the four combinations.

17.3.2 PLAIN – a picture in your head

1. Think of an airport with two kinds of security.
2. Static typing is the check at the gate before boarding. Nobody without a valid ticket gets on the plane at all.
3. Dynamic typing is the check by the cabin crew mid-flight. Most people are fine, but if someone is in the wrong seat you find out at 30,000 feet.
4. Strong typing is a strict officer who says a bus ticket is not a plane ticket, full stop.
5. Weak typing is an officer who looks at your bus ticket, decides it is probably fine, and lets you through.
6. Static checking catches problems earlier and cheaper. Dynamic checking lets you board faster and change plans later.

Where this comparison breaks: static checks are not free and they are not complete. A type checker proves that the shapes fit together. It does not prove your program is correct. A program that computes the wrong average with perfectly matched types passes every static check ever written.

17.3.3 PLAIN – a worked example

1. Watch the same expression in three languages.

```
Expression:  1 + "2"

Python   -> TypeError, refuses to run this line
JavaScript -> "12"  (number turned into text)
PHP      -> 3       (text turned into number, PHP 8+)
```

2. Python is dynamically typed and strongly typed. It waits until runtime to check, and then refuses to guess.
3. JavaScript is dynamically typed and weakly typed. It converts the number to text and joins them.
4. PHP is dynamically typed and weakly typed in the other direction for `+`, because `+` in PHP is only arithmetic and `.` is the joining operator.
5. Now the static side. In Java, `int x = "hello";` will not compile. The error arrives in under a second, at your desk, before anyone else sees the code.
6. In Python, `x = "hello"` followed a hundred lines later by `x + 1` will run fine until it reaches that line, possibly in production at 3am.
7. **Type inference** means the compiler works out the type so you do not have to write it. In Rust, `let x = 5;` gives an `i32` without you saying so. This is still static typing. The type is fixed, it just was not typed by hand.

17.3.4 PLAIN - what is really happening inside

1. A type is mostly a compile-time idea. After compilation, memory holds bytes and nothing else.
2. The type told the compiler which machine instruction to emit. `ADD` for integers, `ADDSD` for double-precision decimals on x86.
3. In a dynamic language, every value carries a tag at runtime saying what it is. That tag costs memory and a check on every operation.
4. That is why a Python integer needs about 28 bytes while a C integer needs 4. The extra bytes hold the type pointer, a reference count and the digits.
5. **Composite types** build bigger shapes from smaller ones: a struct or record groups named fields, an array groups values of one type, a tuple groups a fixed number of possibly different types.
6. **Generics** let you write one piece of code that works for many types. A list of integers and a list of strings share one implementation. Without generics you either copy the code or throw away the type information.
7. **Null** is a special value meaning "there is nothing here". It is allowed where a real value is expected, which is exactly the problem.
8. The fix is an **option type**: a value that is explicitly either "something" or "nothing", and the language will not let you use it without checking which.

17.3.5 TECHNICAL - the engineer's version

1. Sizes. The C standard guarantees only minimum widths, so the exact sizes below are the near-universal LP64 arrangement used by Linux and macOS on x86-64 and ARM64. Windows uses LLP64, where `long` is 4 bytes.

Type	Size	Range or precision
int8, signed char	1 byte	-128 to 127
uint8, unsigned char	1 byte	0 to 255
int16, short	2 bytes	-32,768 to 32,767
int32, int	4 bytes	-2,147,483,648 up
int64, long (LP64)	8 bytes	about -9.2e18 to 9.2e18
float, binary32	4 bytes	about 7 decimal digits
double, binary64	8 bytes	about 15 to 17 digits
bool	1 byte	true or false
Rust char	4 bytes	one Unicode scalar value

2. Floating point follows IEEE 754, first published in 1985 and revised in 2008 and 2019. binary32 has 24 bits of significand, binary64 has 53. This is why `0.1 + 0.2` is `0.30000000000000004` in every language that uses binary64, including Python, JavaScript and Java.
3. JavaScript numbers are all binary64. Integers are exact only up to 2^53 - 1, which is 9,007,199,254,740,991, exposed as `Number.MAX_SAFE_INTEGER`. BigInt was added in ECMAScript 2020 for larger values.
4. Type system taxonomy: static or dynamic (when checking happens), strong or weak (how much implicit conversion is allowed), nominal or structural (do two types match by name or by shape). Go interfaces and TypeScript are structural. Java classes are nominal.
5. Type inference in ML-family languages uses Hindley-Milner, published by Roger Hindley in 1969 and independently by Robin Milner in 1978. Rust and Swift use local inference, which is weaker but gives better error messages.
6. Generics are implemented either by monomorphization (Rust, C++ templates, C# value types), which duplicates code per type and is fast, or by erasure (Java, which added generics in Java 5 in 2004), which keeps one copy and loses the type at runtime.

7. Null. Tony Hoare introduced the null reference in 1965 while designing the type system of ALGOL W. At a software conference in 2009 he called it "my billion-dollar mistake", saying it had led to "innumerable errors, vulnerabilities and system crashes".
8. Option types are the fix: `Option<T>` in Rust, `Optional<T>` in Java 8 and Swift, `Maybe a` in Haskell, `T?` with null-safety checks in Kotlin, and `strictNullChecks` in TypeScript since version 2.0 in 2016.

```
fn find(name: &str) -> Option<u32> {
    if name == "ada" { Some(1815) } else { None }
}

match find("ada") {
    Some(year) => println!("born {}", year),
    None => println!("not found"),
}
```

9. The compiler will not let you reach `year` without handling `None`. That is the entire trick: the checking is not optional and not forgettable.
10. Tools: `mypy` and `pyright` add static checking to Python. `tsc` does it for JavaScript via TypeScript. `sorbet` does it for Ruby.

17.3.6 WORDS - remember these

1. Type — what kind of value it is — a set of values plus the operations legal on them.
2. Static typing — checked before running — type checking performed at compile time by the language's type system.
3. Dynamic typing — checked while running — types carried on values and verified at each operation.
4. Strong typing — no silent conversions — the language rejects operations between incompatible types rather than coercing.
5. Type inference — the compiler guesses correctly — deducing types from context without explicit annotations, still statically.
6. Generics — one implementation for many types — parametric polymorphism, realized by monomorphization or by erasure.
7. Option type — a value that may be absent, safely — a sum type with a present and an absent case, forcing the caller to handle both.

17.4 Memory management

17.4.1 PLAIN - in simple words

1. A running program needs space to keep things. That space is memory.
2. Some space is easy. A number you use inside one function lives on the **stack**, and disappears by itself when the function ends.
3. Some space is hard. Anything whose size you only learn while running, or that must outlive the function that made it, lives on the **heap**.
4. Heap space has to be asked for, and later given back.
5. There are exactly three answers to "who gives it back".
6. Answer one: you do, by hand. This is C and old C++. It is fast and it is easy to get wrong.
7. Answer two: the language does, automatically, by watching your values while the program runs. This is Java, C#, Go, Python, JavaScript and most others.
8. Answer three: the compiler works it out before the program runs, from rules you followed while writing. This is Rust.
9. Every one of the three is a real trade. None of them is free.

17.4.2 PLAIN - a picture in your head

1. Think of a library with a room of borrowed books.
2. Manual memory management is a library with no due dates. You borrow a book and you must remember to return it. If you forget, the room fills up. If you return the same book twice, the desk gets confused. If you return a book and then keep reading it, you are reading a book that has been given to someone else.
3. Reference counting is a library where each book has a tally on the cover. Add one when someone borrows it, subtract one when they finish. At zero, it goes back on the shelf immediately.
4. Tracing garbage collection is a librarian who periodically walks the whole building, notes every book anyone is actually holding, and reshelves the rest. While she walks, nobody may move.
5. Rust's ownership is a library rule that each book has exactly one owner, and the owner must return it when they leave the room. The rule is checked at the door, before you enter.

Where this comparison breaks: books are visible and countable, and a librarian can see who is holding one. Real memory has no such property. The collector can only follow pointers from a known starting set, and anything not reachable that way is assumed dead, even if you meant to keep it.

17.4.3 PLAIN - a worked example

1. Here is manual allocation in C, done correctly.

```
#include <stdlib.h>

int *make_array(int n) {
    int *a = malloc(n * sizeof(int));
    if (a == NULL) return NULL;
    for (int i = 0; i < n; i++) a[i] = i;
    return a;
}

/* caller must eventually call free(a) */
```

2. `malloc` asks for a block of bytes and hands back its address. `free` gives it back. Nothing else does.
3. Now the four ways this goes wrong.
4. **Leak**: you never call `free`. The block stays reserved for the life of the process. Do it in a loop and memory grows until the process is killed.
5. **Double free**: you call `free(a)` twice. The allocator's internal list is corrupted. This is a classic route to a security exploit.
6. **Use after free**: you call `free(a)` and then read `a[0]`. The bytes may still look right for a while, which makes this bug appear and disappear at random.
7. **Dangling pointer**: you return the address of a local variable, which lived on the stack and vanished the moment the function ended.

```
int *bad(void) {
    int x = 42;
    return &x;   /* x dies here; the pointer is dangling */
}
```

8. All four compile without error in plain C. Three of the four are silent during testing and loud in production.

17.4.4 PLAIN - what is really happening inside

1. Reference counting. Each object carries a small number. Every time a new name points at it, add one. Every time a name stops pointing at it, subtract one. At zero, free it at once.
2. That is simple, immediate and predictable. It has two costs. Every assignment does extra arithmetic, and in a multi-threaded program that arithmetic must be atomic, which is slow.
3. It also cannot free a cycle. If A points at B and B points at A, both counts stay at one forever even though nothing outside can reach them.
4. Tracing garbage collection solves the cycle problem by asking a different question: not "how many people point at this" but "can I still reach this".
5. Mark and sweep works in two phases. Start from the **roots** (global variables, stack slots, registers). Follow every pointer and mark everything you can reach. Then walk the whole heap and free everything unmarked.
6. Most objects die very young. Measured across many programs, the large majority of allocations become garbage almost immediately. This observation is called the generational hypothesis.
7. So a **generational collector** splits the heap. New objects go in a small young area, collected often and quickly. Objects that survive a few collections are promoted to an old area, collected rarely.
8. A **GC pause** is the time the program is stopped so the collector can work without objects moving under its feet. Modern collectors do most of the work concurrently and stop the world only briefly.
9. Rust's answer removes the question. Every value has exactly one **owner**. When the owner goes out of scope, the value is freed. You can lend a value out as a **borrow**, and the compiler checks that no borrow outlives the owner.
10. There is no runtime cost at all, because all of this is checked before the program runs. The price is that the compiler rejects programs a human can see are fine, and you must restructure them.

```
fn main() {
    let s = String::from("hello");
    let r = &s;              // borrow, does not take ownership
    println!("{} {}", s, r); // both usable
}                            // s dropped here, memory freed
```

17.4.5 TECHNICAL - the engineer's version

1. `malloc` and `free` are C standard library functions, not system calls. They sit on top of `brk`, `sbrk` or `mmap`. glibc uses ptmalloc2; alternatives include jemalloc, tcmalloc and mimalloc, chosen for different fragmentation and threading behaviour.
2. Fragmentation is the memory management failure nobody warns you about: the allocator holds plenty of free bytes, but no single contiguous run large enough for the next request.
3. Reference counting in CPython is the primary mechanism, with a separate cycle detector for the case counting cannot handle. Swift uses ARC, automatic reference counting, inserted by the compiler at compile time, and requires the programmer to mark cycles with `weak` or `unowned`.
4. Tracing collectors in production, with real characteristics:

Collector	Runtime	Typical pause target
G1 (default since Java 9)	JVM	tens of milliseconds
ZGC (production, Java 15)	JVM	under 1 ms
Shenandoah	JVM	under 10 ms
Go's collector	Go	under 1 ms since Go 1.8

5. Go's collector is concurrent, tri-colour and non-moving. The Go 1.5 release in August 2015 was the point where sub-10 ms pauses became normal, and Go 1.8 in 2017 removed the stop-the-world stack rescanning that had been the main remaining source of long pauses.

6. ZGC and Shenandoah are concurrent compacting collectors using load or store barriers, so pause time is largely independent of heap size. This is the headline claim, and it holds for pause length, not for total throughput cost.
7. Rust's rules, stated exactly: each value has one owner; there may be either any number of immutable borrows or exactly one mutable borrow at a time, not both; every borrow's lifetime must be contained within the owner's. The checker that enforces this is the borrow checker, rewritten as NLL (non-lexical lifetimes) in the Rust 2018 edition.
8. Escape hatches exist and are honest: `Rc<T>` and `Arc<T>` add reference counting when single ownership does not fit, and `unsafe` blocks let you do what C does, in clearly marked regions.

Approach	Speed	Safety	Predictability
Manual (C)	fastest possible	none, all bugs live	fully predictable
Ref counting	steady small cost	leaks cycles	very predictable
Tracing GC	fast alloc, pauses	memory-safe	unpredictable
Ownership (Rust)	same as manual	memory-safe	fully predictable

9. The honest version: "Rust has no runtime cost" is nearly true but not entirely. Bounds checks on slice indexing remain at runtime unless the optimizer can prove them redundant, and `Rc/Arc` cost the same as any other reference count.
10. Microsoft reported in 2019 that around 70 percent of the security vulnerabilities it assigned CVEs to were memory safety issues. The Chromium project has published a very similar figure. This is the single strongest argument for the second and third approaches.
11. Tools: `valgrind --leak-check=full`, AddressSanitizer (`-fsanitize=address`), `heaptrack`, `jcmd GC.heap_info`, `GODEBUG=gctrace=1`, and `tracemalloc` in Python.

17.4.6 WORDS - remember these

1. Stack — automatic space that cleans itself — a contiguous region managed by push and pop, with frame lifetime tied to call depth.
2. Heap — space you ask for and give back — a dynamically managed region served by an allocator with explicit or automatic reclamation.
3. Memory leak — space never given back — allocated memory that remains reachable-but-unused or unreachable-but-unfreed for the process lifetime.
4. Use after free — reading a returned book — dereferencing a pointer to memory already released, a leading cause of exploitable vulnerabilities.
5. Reference counting — keep a tally per object — incrementing and decrementing a per-object counter, freeing at zero, unable to reclaim cycles.
6. Mark and sweep — find what is reachable, free the rest — a tracing algorithm that marks from roots then sweeps unmarked objects.
7. GC pause — the moment everything stops — a stop-the-world phase during which mutator threads are suspended for collector work.
8. Ownership — one owner, checked at compile time — Rust's affine type discipline enforced by the borrow checker with zero runtime cost.

17.5 Paradigms

17.5.1 PLAIN - in simple words

1. A **paradigm** is a style of organizing a program. It is a habit of thought, not a feature.
2. **Imperative** means you write a sequence of commands that change state. Do this, then this, then this.
3. **Procedural** is imperative plus the idea of grouping commands into named procedures you can call.

4. **Object-oriented** groups data together with the operations on that data into objects.
5. **Functional** treats computation as evaluating functions, avoids changing things in place, and passes functions around as ordinary values.
6. **Declarative** means you describe what you want, not how to get it. SQL and HTML are declarative.
7. **Event-driven** means the program sits waiting and reacts when something happens: a click, a message, a timer.
8. Most real languages support several of these. The paradigm is a choice you make per program, and sometimes per file.

17.5.2 PLAIN - a picture in your head

1. Imagine you want a cup of tea and there are five ways to get one.
2. Imperative: you write out every step yourself. Fill kettle, switch on, wait, pour, add bag, wait three minutes, remove bag.
3. Procedural: you write "boil water", "brew", "serve" as three named jobs, then call them in order.
4. Object-oriented: you build a Kettle that knows how to boil itself and a Cup that knows how to be filled. You tell the kettle to boil, not the water.
5. Functional: you write `serve(brew(boil(water)))`. Each step takes something and returns a new something, and nothing is modified in place.
6. Declarative: you write "I want tea, medium strength, with milk" and something else works out the steps.
7. Event-driven: you sit down, and when the kettle clicks off, you react.

Where this comparison breaks: real programs mix all of these in one file, and the boundaries are much blurrier than five clean styles suggest. Also the declarative version only exists because somebody wrote the imperative version underneath it. Declarative never removes the work, it moves it.

17.5.3 PLAIN - a worked example

1. The task: given a list of orders, find the total value of the ones over 100.
2. First in an object-oriented style, in Python.

```
class OrderBook:
    def __init__(self):
        self.orders = []

    def add(self, value):
        self.orders.append(value)

    def total_over(self, limit):
        total = 0
        for v in self.orders:
            if v > limit:
                total += v
        return total

book = OrderBook()
for v in [50, 120, 300, 80]:
    book.add(v)
print(book.total_over(100))     # 420
```

3. Now the same task in a functional style.

```
from functools import reduce

orders = [50, 120, 300, 80]
```

```
big = filter(lambda v: v > 100, orders)
total = reduce(lambda a, b: a + b, big, 0)
print(total)                    # 420
```

4. Both print 420. The difference is where the state lives.
5. In the first, `self.orders` is a thing that exists, holds state, and is changed by `add`. Behaviour is attached to that thing.
6. In the second there is no thing. There is a list going in and a number coming out, through two functions that each produce a new value.
7. The functional version is easier to test, because there is nothing to set up. The object version is easier to extend when you later need customer names, dates and discounts hanging off each order.
8. Neither is better. They fail differently.

17.5.4 PLAIN - what is really happening inside

1. Object-oriented programming rests on three ideas.
2. **Encapsulation**: the data inside an object is private, and you touch it only through the object's own methods. That way the object can guarantee its own rules stay true.
3. **Inheritance**: a new class can be defined as "an existing class, plus these changes". A `SavingsAccount` is an `Account` with an interest rate.
4. **Polymorphism**: many different objects can respond to the same message in their own way. Call `area()` on a Circle and a Square and each does the right thing without the caller knowing which it holds.
5. The honest criticism of inheritance: deep inheritance chains are one of the most reliable ways to make a codebase hard to change.
6. When class E inherits from D inherits from C inherits from B inherits from A, reading one method means reading five files, and changing A silently changes the behaviour of dozens of classes you have never opened.
7. This is why the modern advice, common since the 1994 book Design Patterns by Gamma, Helm, Johnson and Vlissides, is "favour composition over inheritance": hold an object rather than inherit from it.
8. Go took this seriously and has no inheritance at all, only embedding and interfaces. Rust has traits and no class inheritance. Both are widely used, which shows inheritance is not required for large software.
9. Functional programming rests on different ideas.
10. A **pure function** returns the same answer for the same inputs and changes nothing outside itself. It cannot write a file, print, or read the clock.
11. **Immutability** means values are never changed in place. To "change" a list you produce a new list. Nothing you already handed to someone else can shift under them.
12. A **higher-order function** takes a function as an argument or returns one. `map`, `filter` and `reduce` are the three you meet constantly.
13. `map` applies a function to every item and gives a new sequence. `filter` keeps the items passing a test. `reduce` folds a sequence down to one value by combining two at a time.

17.5.5 TECHNICAL - the engineer's version

1. Timeline. Fortran, 1957, John Backus at IBM: imperative. Lisp, 1958, John McCarthy at MIT: the first functional language. Simula 67, 1967, Ole-Johan Dahl and Kristen Nygaard in Norway: the first object-oriented language, and the source of the word "class". Smalltalk, 1972 at Xerox PARC, Alan Kay and colleagues: the language that defined object-oriented as a whole worldview.
2. Alan Kay, who coined the term "object-oriented", said later that he meant messaging between independent objects rather than classes and inheritance, and that C++ and Java were not what he had in mind. Experts genuinely disagree about which reading is the useful one.
3. SOLID, named by Michael Feathers around 2004 from principles Robert Martin collected, is the standard object-oriented design checklist: single responsibility, open-closed, Liskov substitution, interface segregation, dependency inversion. Barbara Liskov stated the substitution principle in a 1987 keynote.
4. Substitution in practice: if `Square` inherits from `Rectangle` and `Rectangle` has `setWidth` and `setHeight`, a `Square` cannot honour both independently. This is the classic demonstration that "is a" in English is not "is a" in a type system.
5. Functional core concepts with names: referential transparency (an expression can be replaced by its value without changing behaviour), persistent data structures (updates share most of the old structure, so a "copy" is cheap), currying, closures, and algebraic data types.
6. Persistent data structures are what make immutability affordable. Clojure's vectors use a 32-way branching trie, so an update copies about five small nodes rather than the whole vector.
7. Functional ideas have won in practice even where the paradigm has not. `map` and `filter` are in every mainstream language. Java 8 added lambdas and streams in March 2014. C++11 added lambdas in 2011. Python has had `lambda` since 1994.

Paradigm	Core unit	Example language
Procedural	the procedure	C, 1972
Object-oriented	the object	Smalltalk, 1972
Functional	the pure function	Haskell, 1990
Declarative	the description	SQL, 1974
Logic	the rule	Prolog, 1972

8. Event-driven is a control-flow architecture rather than a paradigm in the same sense. Its engine is a loop that pulls from a queue and dispatches to handlers, which we take apart in section 17.8.

17.5.6 WORDS - remember these

1. Paradigm — a style of organizing code — a coherent model of computation and program structure.
2. Encapsulation — keep the insides private — restricting direct access to state so invariants are enforced by the type's own operations.
3. Inheritance — a class built on another — subtyping with implementation reuse from a parent class.
4. Polymorphism — one call, many behaviours — dispatch of an operation based on the runtime or static type of its operand.
5. Pure function — same input, same output, no side effects — referentially transparent, with no observable interaction outside its return value.
6. Immutability — never change, always replace — values that cannot be modified after construction, usually with structural sharing for efficiency.
7. Higher-order function — a function taking or returning a function — a function whose domain or codomain includes function types.

17.6 Data structures, properly

17.6.1 PLAIN - in simple words

1. A **data structure** is a way of arranging values in memory so that certain operations are cheap.
2. Every structure is good at some things and bad at others. There is no best one.
3. An **array** is a fixed row of slots, all the same size, side by side. You can jump straight to slot 500.
4. A **dynamic array** is an array that grows when it fills up, by making a bigger one and copying.
5. A **linked list** is a chain: each item holds a value and the address of the next item. Items can be anywhere in memory.
6. A **stack** is last in, first out. Like a pile of plates.
7. A **queue** is first in, first out. Like a queue at a counter.
8. A **deque** is a queue you can push to and pop from at both ends.
9. A **hash table** maps a key to a value with near-instant lookup.
10. A **set** is a collection with no duplicates and fast "is this in here".
11. A **binary search tree** keeps items in order so you can find, insert and delete in a small number of steps.
12. A **heap** keeps the smallest (or largest) item instantly available.
13. A **graph** is nodes joined by edges. Roads, friendships, dependencies.
14. A **trie** stores strings by shared prefix, so all words starting "car" sit under one path.

17.6.2 PLAIN - a picture in your head

1. Think of ways to store books in a house.
2. An array is a shelf of identical slots, numbered. You want book 40, you walk straight to slot 40. But inserting a book in the middle means shifting every book after it.
3. A linked list is a treasure hunt. Each book has a note saying where the next one is. Inserting is trivial, you just rewrite two notes. Finding book 40 means following 39 notes.
4. A hash table is a set of drawers labelled A to Z, where a rule turns the title into a drawer number. You compute the drawer and open it. If two books land in the same drawer, you keep a small pile there.
5. A binary search tree is a house where each room says "smaller titles that way, larger titles that way". Twenty questions gets you to any of a million books.
6. A heap is a pile where the lightest book is always on top, and nothing else is sorted at all.

Where this comparison breaks: real memory has a property no shelf has, called locality. Reading one byte pulls its neighbours into cache for free. So an array is far faster than its step count suggests, and a linked list far slower, even when the theory says they are equal. Measured on modern hardware, scanning an array can be several times faster than walking a linked list of the same length.

17.6.3 PLAIN - a worked example

1. Take a hash table holding names against phone numbers.
2. You insert `"ada" -> 5551234`.
3. The hash function turns the text "ada" into a number. Suppose it gives 3,481,092,776.
4. The table has 8 buckets. Take the hash modulo 8: 3,481,092,776 mod 8 = 0.
5. So "ada" goes into bucket 0, storing both the key and the value.
6. Now insert `"bob" -> 5559876`. Suppose `hash("bob") mod 8` is also 0.
7. That is a **collision**. Two different keys, one bucket.
8. Handling it by **chaining**: bucket 0 holds a tiny list, and now that list has two entries. Looking up "bob" finds bucket 0, then compares keys within it.
9. Handling it by **open addressing**: if bucket 0 is taken, try bucket 1, then 2, until an empty one is found. Lookup follows the same walk.

```
buckets, 8 slots, chaining:

 0 -> ["ada" 5551234] -> ["bob" 5559876]
```

```
1 -> empty
2 -> empty
3 -> ["cy" 5550000]
...
7 -> empty
```

10. As the table fills, collisions get more common and lookups slow down.
11. So the table watches its **load factor**: entries divided by buckets. When that crosses a threshold, typically 0.7 to 0.9, it allocates a bigger table and rehashes everything into it.
12. That one resize is expensive, but it happens rarely, so the average cost of an insert stays constant. This is called amortized constant time.

17.6.4 PLAIN – what is really happening inside

1. An array's superpower is arithmetic. The address of item `i` is `base + i * item_size`. One multiply, one add, done. That is why access is constant time regardless of size.
2. A dynamic array grows by a factor, not by a fixed amount. Typically it doubles, or grows by 1.5 times. Growing by one each time would make appending n items cost n-squared work in total.
3. Because it doubles, appending n items costs about 2n copies overall, so each append averages constant time even though some individual appends are expensive.
4. A linked list's superpower is that inserting needs no shifting. Its weakness is that every item costs an extra pointer of memory and lives at an unpredictable address, which defeats the CPU cache.
5. A stack and a queue are usually not separate structures at all. They are restricted interfaces on top of an array or a linked list.
6. A binary search tree can degenerate. Insert 1, 2, 3, 4, 5 in order into a plain tree and you get a straight line: a linked list with extra steps, and lookups become linear.
7. A **balanced tree** fixes this by rotating itself after inserts to keep the height near the logarithm of the item count. AVL trees (1962) and red-black trees (1978) are the two you meet.
8. A heap is stored as a plain array with an implicit shape rule: the children of index `i` sit at `2i+1` and `2i+2`. No pointers at all.
9. A trie stores one character per edge. All words sharing a prefix share a path, so lookup costs the length of the word and not the size of the dictionary.

17.6.5 TECHNICAL – the engineer's version

1. Complexities below are average case unless marked. `n` is the number of elements, `k` the key length.

Structure	Access	Search	Insert
Array	O(1)	O(n)	O(n)
Dynamic array	O(1)	O(n)	O(1) amortized end
Linked list	O(n)	O(n)	O(1) at a known node
Stack, queue, deque	O(1) ends	O(n)	O(1)
Hash table	n/a	O(1) avg, O(n) worst	O(1) avg
Balanced BST	O(log n)	O(log n)	O(log n)
Binary heap	O(1) min	O(n)	O(log n)
Trie	n/a	O(k)	O(k)

Structure	Delete	Ordered	Extra space
Array	O(n)	no	none
Dynamic array	O(n)	no	up to 100 percent
Linked list	O(1) at node	no	1 pointer each

Structure	Delete	Ordered	Extra space
Hash table	O(1) avg	no	buckets plus slack
Balanced BST	O(log n)	yes	2 pointers each
Binary heap	O(log n)	partial	none
Trie	O(k)	yes, by prefix	high

2. Real implementations. C++ `std::vector` is a dynamic array, `std::map` is a red-black tree, `std::unordered_map` is a hash table with chaining. Java `ArrayList` is a dynamic array, `HashMap` uses chaining and converts a bucket to a red-black tree once it holds 8 or more entries, a change made in Java 8 in 2014 to blunt hash-collision denial-of-service attacks.
3. Python `dict` has been insertion-ordered since CPython 3.6 (an implementation detail then) and by language guarantee since 3.7 in 2018. It uses open addressing with a compact index array.
4. Go maps are hash tables with 8-slot buckets and randomized iteration order, deliberately, so that nobody writes code depending on the order.
5. Hash functions in production: SipHash-1-3 in Python and Rust's default hasher, chosen for resistance to hash-flooding attacks rather than raw speed; FNV-1a and xxHash where speed matters more; wyhash in several newer maps.
6. Python's `hash()` for strings is randomized per process by default since Python 3.3 in 2012, after the 2011 hash-collision denial-of-service disclosure. Set `PYTHONHASHSEED` to make it repeatable.
7. Cache behaviour dominates the constant factors. A cache line is 64 bytes on x86-64 and on Apple silicon. An array of 4-byte integers gets 16 per line for free. A linked list of nodes scattered across the heap gets one useful item per cache miss, and a main-memory miss costs on the order of 60 to 100 nanoseconds while an L1 hit costs about 1 nanosecond.
8. Graph representations: adjacency list uses O(V + E) space and is right for sparse graphs; adjacency matrix uses O(V^2) and gives O(1) edge lookup, right only for dense graphs.
9. Tools: `sys.getsizeof` in Python, `jol` for Java object layout, `perf stat -e cache-misses` on Linux, and `heaptrack` for allocation profiles.

17.6.6 WORDS - remember these

1. Array — numbered slots side by side — contiguous storage with O(1) indexed access by address arithmetic.
2. Dynamic array — an array that grows — geometric reallocation giving amortized O(1) append.
3. Linked list — a chain of items — nodes with pointers, O(1) splice, poor cache locality.
4. Hash table — compute the drawer, open it — key-to-bucket mapping with collision resolution by chaining or open addressing.
5. Collision — two keys, one bucket — distinct keys hashing to the same index, unavoidable by the pigeonhole principle.
6. Load factor — how full it is — entries divided by buckets, the trigger for resize, typically 0.7 to 0.9.
7. Balanced tree — a tree that stays short — a BST with rebalancing keeping height O(log n), such as AVL or red-black.
8. Amortized — expensive rarely, cheap usually — the average cost per operation over a worst-case sequence, not a probabilistic average.

17.7 Algorithms and Big-O

17.7.1 PLAIN - in simple words

1. An **algorithm** is a finite recipe for turning an input into an answer.
2. The same job usually has many algorithms, and they differ enormously in how the work grows as the input grows.
3. **Big-O notation** is a short way of saying how the work grows.

4. It ignores constants and small terms on purpose. It answers "what happens when the input gets ten times bigger", not "how many milliseconds".
5. O(1) means the work does not depend on the size at all.
6. O(n) means ten times the data takes ten times as long.
7. O(n^2) means ten times the data takes a hundred times as long.
8. O(log n) means doubling the data adds one step. This is the good one.
9. O(2^n) means adding one item doubles the time. This is the one that kills you.
10. Big-O is about scale, not speed. For 10 items, anything works. For 10 million, the choice decides whether the program finishes today.

17.7.2 PLAIN – a picture in your head

1. You are looking for a name in a phone book of one million entries.
2. Reading every page from the front is **linear search**, O(n). Worst case, a million looks.
3. Opening the middle, deciding which half, and repeating is **binary search**, O(log n). Twenty looks, because 2^20 is just over a million.
4. Twenty against a million. That is the whole reason algorithms matter.
5. Now imagine comparing every entry with every other entry to find duplicates. That is O(n^2), which is a million million comparisons. At a billion comparisons a second, about 11 days.
6. Sorting first and then scanning once is O(n log n), which is about 20 million steps, or a fiftieth of a second.

Where this comparison breaks: binary search only works because the phone book is already sorted. Big-O hides the cost of getting there. It also hides constants, and constants matter in the real world: an O(n log n) algorithm with a huge constant can lose to an O(n^2) one for every input you will ever actually see. This is exactly why library sorts switch to insertion sort for small chunks.

17.7.3 PLAIN – a worked example

1. Assume a rough one hundred million simple operations per second, which is a reasonable figure for interpreted code and pessimistic for compiled code. These figures are approximate and meant for ranking, not for prediction.

Complexity	Name	n done in about 1 s
O(1)	constant	any size
O(log n)	logarithmic	effectively unlimited
O(n)	linear	100,000,000
O(n log n)	linearithmic	about 4,000,000
O(n^2)	quadratic	about 10,000
O(n^3)	cubic	about 460
O(2^n)	exponential	about 26
O(n!)	factorial	about 11

2. Read the bottom two rows again. An exponential algorithm handles 26 items in a second. Twenty-six. Not 26 thousand.
3. Now dynamic programming, with the classic example. Fibonacci numbers, where each is the sum of the two before it.

```
def fib(n):
    if n < 2:
        return n
    return fib(n - 1) + fib(n - 2)
```

4. This is correct and it is O(2^n). `fib(40)` makes about 331 million calls, because `fib(38)` is computed twice, `fib(37)` three times, and so on.
5. The insight of **dynamic programming**: the sub-problems repeat, so solve each one once and write the answer down.

```
def fib(n, seen={}):
    if n < 2:
        return n
    if n not in seen:
        seen[n] = fib(n - 1, seen) + fib(n - 2, seen)
    return seen[n]
```

6. That change makes it O(n). `fib(40)` now takes 40 additions instead of 331 million calls. The algorithm did not get cleverer. It stopped repeating itself.
7. Writing answers down as you go is called memoization, and it is the top-down form. Filling a table from the smallest case upward is the bottom-up form. They compute the same thing.

17.7.4 PLAIN - what is really happening inside

1. Recursion means a function calls itself. Each call gets its own **stack frame**: a slice of the stack holding its arguments, its local variables and the address to return to.

```
main            -> frame 1
  fib(4)        -> frame 2
    fib(3)      -> frame 3
      fib(2)    -> frame 4
        fib(1)  -> frame 5   returns 1, frame popped
        fib(0)  -> frame 5   returns 0, frame popped
```

2. Frames stack up as you go deeper and are popped as each call returns. If you never stop going deeper, the stack runs out. That is a stack overflow.
3. Default limits: CPython refuses at about 1,000 nested calls by default; the main thread stack on Linux is typically 8 MB, which is thousands of frames for compiled code.
4. Now sorting, in order of how they work.
5. **Bubble sort**: repeatedly walk the list swapping neighbours that are out of order. O(n^2). Never use it, but understand it.
6. **Insertion sort**: take each item and slide it back into its place among the already-sorted front. O(n^2) in general, but O(n) if the data is nearly sorted, and very fast for small lists.
7. **Merge sort**: split in half, sort each half, then merge the two sorted halves in one pass. Always O(n log n). Needs extra space. Stable, meaning equal items keep their original order.
8. **Quicksort**: pick a pivot, move everything smaller left and larger right, then sort each side. O(n log n) on average, O(n^2) if you keep picking bad pivots. Sorts in place, and is usually the fastest in practice.
9. Greedy against exhaustive. A **greedy** algorithm takes the best-looking step at each moment and never reconsiders. It is fast and sometimes optimal.
10. Giving change with British coins greedily always gives the fewest coins. With a hypothetical set of 1, 3 and 4, greedy makes 6 as 4+1+1, three coins, while the best answer is 3+3, two coins. Greedy is a gamble that needs proof.
11. **Exhaustive** search tries every possibility. It always finds the best answer and it is usually exponential.

17.7.5 TECHNICAL - the engineer's version

1. Formally, f(n) is O(g(n)) if there exist positive constants c and n0 such that f(n) <= c*g(n) for all n >= n0. Big-O is an upper bound. Big-Omega is a lower bound, and Big-Theta is both.
2. Comparison sorting has a proven lower bound of Omega(n log n) comparisons, because a decision tree with n! leaves has height at least log2(n!). Counting sort and radix sort beat it by not comparing.
3. What real libraries actually ship:

Library	Algorithm	Since
Python `sorted`	Timsort	2002, Tim Peters
Java objects	Timsort	Java 7, 2011
Java primitives	dual-pivot quicksort	Java 7, 2011
C++ `std::sort`	introsort	Musser, 1997
Go `slices.Sort`	pattern-defeating quicksort	Go 1.19, 2022
Rust `sort_unstable`	quicksort variant	in-place, no alloc

4. Timsort was written by Tim Peters for CPython in 2002. It finds existing sorted runs, extends short ones with binary insertion sort, and merges them under strict invariants. It is stable and it is O(n) on already-sorted data. Java adopted it for object arrays in Java 7 in 2011.
5. Introsort, described by David Musser in 1997, starts as quicksort, switches to heapsort if recursion gets too deep, and finishes with insertion sort on small partitions. It gets quicksort's speed with heapsort's O(n log n) worst-case guarantee.
6. Dual-pivot quicksort was contributed by Vladimir Yaroslavskiy in 2009 and is used for Java primitive arrays, where stability is meaningless because equal integers are indistinguishable.
7. Dynamic programming was named by Richard Bellman around 1953 at RAND. He chose the words partly because they sounded impressive to a defence secretary who disliked mathematical research. It requires two properties: optimal substructure and overlapping subproblems.
8. Standard dynamic programming problems and their complexities: 0/1 knapsack O(nW), longest common subsequence O(nm), edit distance O(nm), Floyd-Warshall all-pairs shortest paths O(V^3).
9. Greedy algorithms that are provably optimal: Dijkstra's shortest path (1959, Edsger Dijkstra), Kruskal's and Prim's minimum spanning trees, and Huffman coding (David Huffman, 1952). Each has a proof, usually via a matroid or an exchange argument.
10. Tools: `timeit` in Python, JMH for Java, `perf record` and `perf report` on Linux, `cargo bench` and `pprof` for Go.

17.7.6 WORDS - remember these

1. Algorithm — a finite recipe that terminates — a well-defined procedure mapping input to output in finite steps.
2. Big-O — how the work grows with size — an asymptotic upper bound on a cost function, ignoring constants and lower-order terms.
3. Binary search — halve the range each time — O(log n) search on a sorted sequence with random access.
4. Stable sort — equal items keep their order — a sort preserving the relative order of records with equal keys.
5. Stack frame — one call's private space — the activation record holding arguments, locals and the return address.
6. Memoization — write the answer down — caching results of pure function calls keyed by arguments.
7. Greedy — best-looking step, never reconsider — a locally optimal choice strategy, globally optimal only where provable.

17.8 Concurrency and parallelism

17.8.1 PLAIN - in simple words

1. **Concurrency** is dealing with many things at once. **Parallelism** is doing many things at once.
2. One cook juggling three pans is concurrent. Three cooks each with one pan is parallel.
3. A single-core machine can be concurrent and cannot be parallel. Concurrency is about structure. Parallelism is about hardware.
4. A **process** is a running program with its own private memory. Two processes cannot touch each other's data by accident.

5. A **thread** is a strand of execution inside a process. Threads in one process share all memory, which is fast and dangerous.
6. When two threads touch the same data at the same time and the result depends on who got there first, that is a **race condition**.
7. A **lock** is a token. Only the thread holding it may touch the protected data. Everyone else waits.
8. If two threads each hold a lock the other needs, both wait forever. That is **deadlock**.
9. Concurrency is genuinely hard because the bugs depend on timing, appear once in ten thousand runs, and vanish when you add a print statement.

17.8.2 PLAIN - a picture in your head

1. Picture a shared kitchen notebook holding a running total.
2. Two people each have twenty receipts to add.
3. Adding one receipt takes three actions: read the total, work out the new total in your head, write it back.
4. Person A reads 41. Before A writes, person B also reads 41. A works out 42 and writes it. B works out 42 and writes it.
5. Two receipts were added. The total went up by one. Nothing was broken, nobody made a mistake, and the answer is wrong.
6. A lock is a rule: pick up the pen before reading, put it down after writing. Only one person can hold the pen.
7. Deadlock: A holds the pen and needs the calculator; B holds the calculator and needs the pen. Neither will let go. The kitchen stops.

Where this comparison breaks: people notice when they are stuck. Threads do not. Also, real processors and compilers may reorder your reads and writes for speed, so the order you wrote is not always the order that happens. That is a whole extra layer of difficulty the notebook has no equivalent for.

17.8.3 PLAIN - a worked example

1. Here is the notebook, in Python, with two real threads.

```
import threading

counter = 0

def bump():
    global counter
    for _ in range(200000):
        counter += 1

t1 = threading.Thread(target=bump)
t2 = threading.Thread(target=bump)
t1.start(); t2.start()
t1.join();  t2.join()
print(counter)      # expected 400000, often less
```

2. `counter += 1` looks like one action. It is three: load, add, store.
3. Here is one bad interleaving, step by step.

```
 step  thread A        thread B        counter
   1   load 41                           41
   2                   load 41           41
   3   add -> 42                         41
   4                   add -> 42         41
   5   store 42                          42
   6                   store 42          42
```

4. Two increments happened. The counter rose by one. This is a **lost update**.
5. The fix is a lock around the three steps.

```
lock = threading.Lock()

def bump():
    global counter
    for _ in range(200000):
        with lock:
            counter += 1
```

6. Now the three steps are one indivisible unit. The section of code the lock protects is called the **critical section**.
7. The cost is real: this version is several times slower, because threads spend time waiting instead of working.

17.8.4 PLAIN - what is really happening inside

1. A **mutex** is a lock allowing exactly one holder. The name is short for mutual exclusion. Acquiring an uncontended mutex is cheap, tens of nanoseconds. Acquiring a contended one means the operating system puts your thread to sleep and wakes it later, costing microseconds.
2. An **atomic operation** is one the hardware guarantees cannot be split. `lock xadd` on x86 does load-add-store as a single unit. For a simple counter this is faster than a lock and needs no waiting.
3. Deadlock needs four conditions at once, described by Edward Coffman in 1971: mutual exclusion, hold and wait, no preemption, and circular wait. Break any one and deadlock becomes impossible.
4. The usual practical fix is lock ordering: every thread acquires locks in the same global order, which breaks circular wait.
5. **Async and await** are a different approach entirely. Instead of many threads, you have one thread and an **event loop**.
6. The loop holds a queue of ready tasks. It runs one until that task hits something slow, like waiting for the network. The task says "I am waiting" and hands control back. The loop runs another task.

```
        +----------------+
        |  ready queue   |
        +-------+--------+
                |
                v
   +------------------------+
   |  run task until await  |
   +-----------+------------+
               |
     waiting?  |  yes -> park it, register callback
               |  no  -> task finished, drop it
               v
       back to ready queue when the wait completes
```

7. So one thread can juggle ten thousand network connections, because almost all of them are waiting almost all of the time.
8. This is concurrency without parallelism, and without locks, because only one thing runs at a time. The trade is that any task doing real computation blocks everything.
9. A **coroutine** is a function that can pause in the middle and resume later, keeping its local variables. That is the machinery `await` is built on.
10. **Message passing** avoids shared memory entirely. Two workers never touch the same data. They send copies to each other through a channel. Nothing is shared, so nothing needs locking.
11. The **actor model** takes that further: an actor is an isolated unit with private state and a mailbox. It only reacts to messages. Erlang and Elixir are built entirely this way.

17.8.5 TECHNICAL - the engineer's version

1. Processes versus threads on Linux: both are tasks to the kernel, created by `clone` with different sharing flags. A process context switch costs more because it changes the page tables and flushes TLB entries. Rough figures on modern x86-64: thread switch about 1 to 2 microseconds, process switch somewhat more.
2. Memory ordering is a specification, not an implementation detail. The Java Memory Model was rewritten in JSR-133 for Java 5 in 2004. The C++11 memory model, adopted in 2011, defines `memory_order_relaxed`, `acquire`, `release` and `seq_cst`. x86-64 gives total store order almost for free; ARM and RISC-V are weakly ordered and need explicit barriers.
3. Amdahl's law, stated by Gene Amdahl in 1967: if a fraction p of a program is parallelizable, maximum speedup on N cores is $1 / ((1 - p) + p/N)$. With $p = 0.95$, the ceiling is 20 times no matter how many cores you buy.
4. Concurrency models and their origins:

Model	Origin	Used by
Threads and locks	1960s, POSIX 1995	C, C++, Java
CSP channels	Hoare, 1978	Go, occam
Actors	Hewitt, 1973	Erlang, Akka
Event loop	Unix `select`, 1983	Node.js, asyncio
Ownership + Send/Sync	Rust, 2015	Rust

5. Go's goroutines are user-space threads multiplexed onto operating-system threads by the runtime, starting at an 8 KB stack that grows as needed, which is why a million goroutines is routine and a million OS threads is not.
6. Erlang, created by Joe Armstrong and colleagues at Ericsson from 1986, runs millions of isolated processes with no shared memory and a "let it crash" supervision model. The AXD301 switch built on it is the source of the often-quoted nine nines availability figure, which is a marketing claim about one deployment and not a property of the language.
7. The Global Interpreter Lock in CPython is a single mutex that ensures only one thread executes Python bytecode at a time. It makes reference counting safe and single-threaded code fast. It means CPU-bound Python threads do not scale across cores. Input/output releases it, so I/O-bound threading works.
8. This is changing, with dates. PEP 703 was accepted in July 2023. Python 3.13, released October 2024, shipped an optional experimental free-threaded build. Python 3.14, released 7 October 2025, moved that build to officially supported, though it is still not the default. Established fact: the option exists. Active question: what the single-threaded performance cost settles at, and how quickly C extensions adapt.
9. Rust prevents data races at compile time using the `Send` and `Sync` marker traits plus the borrow checker. It does not prevent deadlock, which is a liveness property, and the documentation says so plainly.
10. Tools: ThreadSanitizer (`-fsanitize=thread`), Helgrind under Valgrind, `perf sched`, Go's `-race` flag, and `jstack` for JVM thread dumps.

17.8.6 WORDS - remember these

1. Concurrency — dealing with many things at once — composing independently executing tasks, whether or not they overlap in time.
2. Parallelism — doing many things at once — simultaneous execution on multiple hardware execution units.
3. Race condition — the answer depends on timing — unsynchronized access where at least one access is a write.

4. Mutex — a token only one may hold — a mutual exclusion primitive serializing entry to a critical section.
5. Deadlock — everyone waiting forever — a cycle of threads each holding a resource another needs, requiring all four Coffman conditions.
6. Atomic — cannot be split in half — an operation the hardware performs indivisibly, such as compare-and-swap.
7. Event loop — one worker, many waits — a dispatcher polling readiness and running callbacks or resuming coroutines on a single thread.
8. GIL — one Python thread at a time — CPython's global interpreter lock, optional to disable since Python 3.13 and supported since 3.14.

17.9 Errors and correctness

17.9.1 PLAIN - in simple words

1. Things go wrong. Files are missing, networks drop, users type nonsense.
2. There are two main ways a function tells you it failed.
3. **Return codes**: the function returns a value meaning "it did not work", and the caller must check it. Simple, explicit, and easy to ignore.
4. **Exceptions**: the function throws, and control jumps out to whoever declared they would handle it. Hard to ignore, but the jump is invisible when reading the code.
5. A **panic** is a third thing: the program has found a state that should be impossible and stops rather than continuing wrongly.
6. The rule of thumb: expected problems are errors, and you handle them. Impossible states are bugs, and you crash on them.
7. An **assertion** is a statement of something you believe must be true. If it is false, stop immediately.
8. A **test** is code that runs your code and checks the answer, automatically, every time.

17.9.2 PLAIN - a picture in your head

1. Think about a parcel delivery.
2. A return code is the driver leaving a card saying "not delivered, reason 3". You have to read the card. Nothing forces you to.
3. An exception is the driver pulling a fire alarm that keeps sounding until someone on some floor takes responsibility. If nobody does, the whole building is evacuated, which is the program crashing.
4. A panic is the driver finding the building on fire and refusing to enter.
5. An assertion is a sign on the loading bay saying "this door must be unlocked during working hours". Nobody expects it to be locked. If it is, something deeper is wrong and continuing makes it worse.

Where this comparison breaks: fire alarms are rare and exceptions are not. In many programs, a file not existing is completely normal and gets thrown dozens of times a second, which is why some languages treat exceptions as a bad fit for ordinary control flow and use return values instead.

17.9.3 PLAIN - a worked example

1. Four languages, the same job: open a file that may not be there.

```
FILE *f = fopen("data.txt", "r");
if (f == NULL) {
    perror("cannot open");
    return 1;
}
```

```
try:
    f = open("data.txt")
```

```
except FileNotFoundError as e:
    print("cannot open:", e)
```

```
f, err := os.Open("data.txt")
if err != nil {
    return fmt.Errorf("cannot open: %w", err)
}
```

```
let f = match File::open("data.txt") {
    Ok(f) => f,
    Err(e) => return Err(e.into()),
};
```

2. C returns a null pointer. Nothing stops you using it, and the crash comes later somewhere else.
3. Python throws. If you write no `try`, the program stops with a stack trace pointing at the exact line.
4. Go returns two values and the second is the error. You can ignore it, but the compiler complains about the unused variable and every linter shouts.
5. Rust returns a `Result`, which is either `Ok` or `Err`, and you cannot get the file out without dealing with both. Rust also has the `?` operator, which is shorthand for exactly the `match` above.
6. Now a real test, in Python's built-in style.

```
def slugify(text):
    return text.strip().lower().replace(" ", "-")

def test_slugify_basic():
    assert slugify("Hello World") == "hello-world"

def test_slugify_trims():
    assert slugify("  Spaced Out  ") == "spaced-out"
```

7. Run it with `pytest`. Two tests, two green ticks, in about 10 milliseconds.
8. Now a property-based test, which invents inputs instead of using yours.

```
from hypothesis import given, strategies as st

@given(st.text())
def test_slug_has_no_whitespace(s):
    assert not any(c.isspace() for c in slugify(s))
```

9. This one fails, quickly, with the input `"\t"`. A tab is whitespace, it is not a space, and `replace(" ", "-")` never touches it.
10. That is the value of property-based testing in one example. It found a real bug from a rule, not from a guess about which inputs to try.

17.9.4 PLAIN - what is really happening inside

1. Throwing an exception is not a normal return. The runtime walks back up the stack, frame by frame, looking for a handler that matches, running cleanup code for each frame it discards. This is called stack unwinding.
2. That is why exceptions are cheap when not thrown and expensive when thrown. Modern implementations use zero-cost tables, so the non-throwing path has no overhead at all and the throwing path costs microseconds.
3. Assertions are usually compiled out of release builds. In C, `NDEBUG` removes them. This means an assertion must never contain something the program needs to do, because in production it will not happen.
4. **Defensive programming** means checking inputs at the boundary of your code, where data arrives from outside, and trusting them inside.

5. The mistake is checking everywhere. Every internal function re-validating makes code longer, slower, and harder to read, without making it safer.
6. Testing comes in layers.
7. A **unit test** tests one function in isolation, in milliseconds, with no database and no network.
8. An **integration test** tests several parts together, usually with a real database, in seconds.
9. An **end-to-end test** drives the whole system the way a user would, in tens of seconds to minutes, and is the most fragile.
10. A **property-based test** states a rule that should always hold and lets the tool generate hundreds of inputs trying to break it.
11. The rough guidance is a pyramid: many unit tests, fewer integration tests, very few end-to-end tests. The reason is cost and flakiness, not purity.

17.9.5 TECHNICAL - the engineer's version

1. Error strategies by language, stated exactly:

Language	Mechanism	Ignorable
C	int or null return, `errno`	yes, silently
Java	checked and unchecked	checked, no
Python	exceptions only	yes, uncaught crashes
Go	explicit error value	yes, linters object
Rust	`Result<T, E>`	no, `#[must_use]`

2. Java's checked exceptions, present since Java 1.0 in 1996, force the caller to declare or handle them. Experts genuinely disagree about this: supporters say it makes failure part of the type signature; critics, including the C# design team who deliberately left them out, say it produces empty catch blocks and `throws Exception` on every method.
3. Rust distinguishes recoverable errors (`Result`) from unrecoverable ones (`panic!`). A panic unwinds by default and can be configured to abort with `panic = "abort"` in Cargo, which produces smaller binaries.
4. Go added error wrapping with `%w` and the `errors.Is` and `errors.As` functions in Go 1.13, released September 2019.
5. Exception implementation: Itanium C++ ABI zero-cost exceptions use side tables consulted only during unwinding, so entering a `try` block costs nothing. Setjmp/longjmp implementations cost on entry instead and are largely historical.
6. Testing frameworks and origins: JUnit, Kent Beck and Erich Gamma, 1997, which defined the xUnit shape copied by almost everything since. QuickCheck, the first property-based tester, by Koen Claessen and John Hughes for Haskell in 2000. Hypothesis brought it to Python from 2013.
7. Coverage is a diagnostic, not a target. 100 percent line coverage proves every line ran, not that any behaviour was checked. Branch coverage is stricter and still not proof. Tools: `coverage.py`, JaCoCo, `go test -cover`, `cargo llvm-cov`.
8. Formal methods sit beyond testing: TLA+, created by Leslie Lamport in the 1990s, is used by AWS to check distributed protocol designs, and the seL4 microkernel has a machine -checked functional correctness proof, completed in 2009. These are real and they are expensive, so they are used where failure is catastrophic.
9. Tools: `pytest -x --lf`, `go test -race ./...`, `cargo test`, `mvn verify`, and mutation testers such as `mutmut` and PIT, which change your code deliberately to see whether any test notices.

17.9.6 WORDS - remember these

1. Return code — a value meaning it failed — an out-of-band or sentinel result the caller must inspect.
2. Exception — a jump to whoever handles it — a non-local transfer of control with stack unwinding and frame cleanup.
3. Panic — stop, this should be impossible — an unrecoverable fault that aborts or unwinds rather than returning.
4. Assertion — a claim that must be true — a runtime check of an invariant, typically compiled out in release builds.
5. Unit test — check one piece alone — an isolated, fast, deterministic test of a single unit of behaviour.
6. Property-based test — state a rule, let it hunt — automated generation of inputs against an invariant, with shrinking to a minimal failing case.
7. Coverage — how much code the tests ran — the proportion of lines or branches executed, a diagnostic rather than a goal.

17.10 A tour of the major languages

17.10.1 PLAIN - in simple words

1. There are about twenty languages you will actually meet, and they sort into a few families.
2. **Systems languages** compile straight to machine code and give you control of memory: C, C++, Rust, and Go in a gentler way.
3. **Managed languages** run on a virtual machine with automatic memory: Java, C#, Kotlin.
4. **Scripting languages** run from source with no separate build step: Python, JavaScript, Ruby, PHP.
5. **Platform languages** exist mainly to build for one place: Swift for Apple, Kotlin for Android.
6. **Specialist languages** do one job extremely well and are bad at everything else: SQL for data, Bash for gluing commands, R and MATLAB for numbers.
7. Underneath all of them is **assembly**, one line per machine instruction.
8. For each language below: the year, the person, what it was made for, what it is genuinely good at, its real weakness, and where you meet it in 2026.

17.10.2 PLAIN - a picture in your head

1. Think of vehicles rather than a ladder of quality.
2. C is a motorcycle: light, fast, nothing between you and the road, and no protection when you fall.
3. Java is a coach: heavy, slow to start, carries a hundred passengers reliably for twenty years.
4. Python is a bicycle: anyone can ride it today, it goes almost anywhere, and it will not win a race.
5. Rust is a motorcycle with a mandatory riding test that you must pass before the engine will start.
6. SQL is a train: it goes only where the rails go, and on those rails nothing beats it.
7. Nobody asks which vehicle is best. They ask what the journey is.

Where this comparison breaks: vehicles cannot be combined, and languages are combined constantly. A single 2026 web product routinely uses TypeScript in the browser, Go on the server, SQL in the database, Python for the model, and C underneath all four.

17.10.3 PLAIN - a worked example

1. The systems languages, with a sample each.
2. **C**. 1972, Dennis Ritchie at Bell Labs. Made to rewrite the Unix operating system in something portable. Genuinely good at: direct hardware access, a near-zero runtime, and being the language every other language talks to. Main weakness: no memory safety of any kind. In 2026 you meet it in the Linux kernel, embedded firmware, and inside the runtime of nearly every language in this list.

```
for (int i = 0; i < n; i++) sum += a[i];
```

3. **C++**. Begun 1979 as "C with Classes" by Bjarne Stroustrup at Bell Labs, released 1985. Made to add Simula's classes to C for large simulations. Good at: zero-overhead abstraction, the fastest code you can still structure. Weakness: enormous, with forty years of features that never leave. In 2026: game engines such as Unreal, browsers, databases, and trading systems. The C++26 standard replaced C++23 in March 2026.

```
std::sort(v.begin(), v.end());
auto n = std::count_if(v.begin(), v.end(), is_big);
```

4. **Go**. Announced November 2009 by Robert Griesemer, Rob Pike and Ken Thompson at Google. Made for large server software with fast builds. Good at: simplicity you can hold in your head, built-in concurrency, one static binary with no dependencies, and compiles measured in seconds. Weakness: deliberately plain, and error handling is repetitive on purpose. In 2026: Docker, Kubernetes, and most cloud infrastructure. Go 1.26 landed 10 February 2026.

```
go worker(ch)          // start a goroutine
msg := <-ch            // receive from a channel
```

5. **Rust**. Started 2006 as Graydon Hoare's personal project, sponsored by Mozilla from 2009, version 1.0 on 15 May 2015. Made for systems work without memory bugs. Good at: memory safety with no garbage collector, and concurrency the compiler checks. Weakness: a steep learning curve and slow compiles. In 2026: Firefox, Cloudflare, Android system components, and Linux kernel drivers, since Rust support was merged into Linux 6.1 in December 2022. Stable release in August 2026 is 1.97.

```
let names: Vec<String> =
    people.iter().map(|p| p.name.clone()).collect();
```

17.10.4 PLAIN – what is really happening inside

1. The managed and scripting languages.
2. **Java**. 1995, James Gosling at Sun Microsystems, from a 1991 project called Oak. Made so one compiled program could run on any machine with a virtual machine. Good at: very large long-lived server systems, superb tooling and profilers, mature garbage collectors. Weakness: verbose, slow to start, heavy on memory. In 2026: banking, enterprise back-ends, and Android underneath. Java 25 is the current long-term-support release, from 16 September 2025.

```
List<String> big = names.stream()
    .filter(n -> n.length() > 3).toList();
```

3. **C#**. Announced 2000, released 2002 with .NET 1.0, designed by Anders Hejlsberg at Microsoft. Made as Microsoft's answer to Java. Good at: a clean modern design, LINQ for querying collections, and it shipped `async/await` in C# 5 in 2012, which most other languages then copied. Weakness: was tied to Windows until .NET Core in 2016. In 2026: Unity games, Windows desktop, and enterprise services on .NET 10, released November 2025.

```
var big = names.Where(n => n.Length > 3).ToList();
```

4. **Kotlin**. Announced July 2011 by JetBrains, version 1.0 in February 2016. Made as a less verbose Java that runs on the same virtual machine. Good at: null safety in the type system, coroutines, and complete Java interop. Weakness: still bound to the JVM tooling world, and compiles more slowly than Java. In 2026: Google named it the preferred Android language in 2019, and most new Android code is Kotlin.

```
val name: String? = user.name
println(name?.length ?: 0)
```

5. **Swift**. Announced June 2014 at Apple's developer conference, led by Chris Lattner. Made to replace Objective-C. Good at: safe and modern while compiling to fast native code, with automatic reference

counting instead of a collector. Weakness: in practice you meet it almost only on Apple platforms. In 2026: essentially all new iOS and macOS applications.

```
let big = names.filter { $0.count > 3 }
```

6. **Python**. First release February 1991, by Guido van Rossum at CWI in the Netherlands. Made as a readable scripting language for people who found the alternatives painful. Good at: readability, gluing other systems together, and an unmatched scientific and machine-learning ecosystem. Weakness: slow, and packaging has been a long-running sore point. In 2026: machine learning, data work, automation, and back-ends. Version 3.14 arrived 7 October 2025.

```
big = [n for n in names if len(n) > 3]
```

7. **JavaScript**. 1995, Brendan Eich at Netscape, with the first version written in about ten days in May 1995. Made to add small interactions to web pages. Good at: it runs everywhere, in every browser on earth, with the largest package ecosystem in existence. Weakness: weak typing and a set of historical oddities that can never be removed. In 2026: every browser, plus servers through Node.js, Deno and Bun.
8. **TypeScript**. Released 1 October 2012, designed by Anders Hejlsberg and Luke Hoban at Microsoft. It is JavaScript plus a static type system, erased before running. Good at: catching whole classes of mistakes before the code ships. Weakness: another build step, and the types are checked but not enforced at runtime. In 2026 it is the default for serious front-end work. TypeScript 6.0 shipped 23 March 2026, and version 7.0, a rewrite of the compiler in Go for roughly a tenfold speedup, is in preview.

```
const big: string[] = names.filter(n => n.length > 3);
```

9. **PHP**. 1995, Rasmus Lerdorf, originally a set of tools he called Personal Home Page. Made to put dynamic content into web pages. Good at: the simplest deployment story there is, and universal cheap hosting. Weakness: an inconsistent standard library and a poor security reputation earned in its early years. In 2026: WordPress, which runs a large fraction of all websites, plus Wikipedia and the Laravel framework. PHP 8.5 arrived 20 November 2025.
10. **Ruby**. 1995, Yukihiro Matsumoto in Japan, designed explicitly for programmer happiness. Good at: expressive code and building small domain-specific languages, which is why Rails felt like magic in 2004. Weakness: slow, and the ecosystem shrank after about 2015. In 2026: GitHub, Shopify, and a large body of Rails applications.

```
big = names.select { |n| n.length > 3 }
```

17.10.5 TECHNICAL – the engineer's version

1. The specialist languages.
2. **SQL**. Designed 1974 as SEQUEL by Donald Chamberlin and Raymond Boyce at IBM San Jose, implementing Edgar Codd's 1970 relational model. Declarative: you state the result, the query planner chooses the method. Good at: set operations over large relational data with an optimizer you did not write. Weakness: dialects differ enough that portable SQL is a discipline, and it is not a general-purpose language. Current standard is SQL:2023. Everywhere data is stored in 2026.

```
SELECT country, COUNT(*) AS n
FROM users WHERE active GROUP BY country
HAVING COUNT(*) > 100 ORDER BY n DESC;
```

3. **Bash**. 1989, Brian Fox for the GNU project, as a free replacement for the 1979 Bourne shell. Good at: joining programs with pipes and automating a machine in a few lines. Weakness: quoting rules are genuinely treacherous and error handling is weak, which is why `set -euo pipefail` is standard advice. In 2026: continuous-integration pipelines, container images, and every server login.

```
set -euo pipefail
grep -c ERROR /var/log/app.log || echo 0
```

4. **R**. 1993, Ross Ihaka and Robert Gentleman at the University of Auckland, as a free implementation of the S language John Chambers created at Bell Labs in 1976. Good at: statistics, and plotting through ggplot2. Weakness: unusual evaluation semantics and poor performance outside vectorized work. In 2026: academic statistics, biostatistics and epidemiology.
5. **MATLAB**. Written by Cleve Moler in the late 1970s at the University of New Mexico as a friendly front end to the LINPACK and EISPACK Fortran libraries, and sold commercially from 1984 when MathWorks was founded. Good at: matrix and numerical work, and Simulink for control system modelling. Weakness: a paid commercial licence, and 1-based indexing that surprises everyone else. In 2026: control engineering, signal processing, and university courses.
6. **Assembly**. Not one language but one per instruction set. Kathleen Booth is credited with writing the first assembly language, working at Birkbeck College in London from 1947. One mnemonic maps to one machine instruction. Good at: exact control and hand-tuning the few functions that matter. Weakness: not portable at all, and enormous effort per line. In 2026: boot code, cryptographic primitives, and compiler output you read while debugging.

```
mov  eax, [rdi]     ; load a 32-bit value
add  eax, 1         ; add one
mov  [rdi], eax     ; store it back
```

7. Verified summary of origins:

Language	Year	Creator
Assembly	1947	Kathleen Booth
C	1972	Dennis Ritchie
SQL (SEQUEL)	1974	Chamberlin and Boyce
MATLAB	late 1970s	Cleve Moler
C++	1979 / 1985	Bjarne Stroustrup
Bash	1989	Brian Fox
Python	1991	Guido van Rossum
R	1993	Ihaka and Gentleman
Java	1995	James Gosling
JavaScript	1995	Brendan Eich
PHP	1995	Rasmus Lerdorf
Ruby	1995	Yukihiro Matsumoto
C#	2000 / 2002	Anders Hejlsberg
Go	2009	Griesemer, Pike, Thompson
Rust	2010 / 1.0 in 2015	Graydon Hoare
Kotlin	2011 / 1.0 in 2016	JetBrains
TypeScript	2012	Hejlsberg and Hoban
Swift	2014	Chris Lattner

8. Release status checked in August 2026:

Language	Current release	Date
Python	3.14	7 Oct 2025
Java	25 LTS, 26 current	Sep 2025, Mar 2026
Go	1.26	10 Feb 2026

Language	Current release	Date
Rust	1.97	Jul 2026
PHP	8.5	20 Nov 2025
TypeScript	6.0, 7.0 in preview	23 Mar 2026

9. Popularity indexes such as TIOBE, the Stack Overflow developer survey and the GitHub Octoverse report disagree with each other every year, because they measure different things: search volume, respondent self-report, and public repository activity. Treat any single ranking as one weak signal.

17.10.6 WORDS - remember these

1. Systems language — you manage the memory — compiles to native code with direct memory access and no mandatory runtime.
2. Managed language — the runtime manages memory — executes on a virtual machine with automatic memory management.
3. Interop — languages calling each other — a foreign function interface, most often through the C ABI.
4. Static binary — one file, no dependencies — an executable with all libraries linked in, as Go produces by default.
5. Erasure — types vanish before running — compile-time-only types removed before execution, as in TypeScript and Java generics.
6. Dialect — the same language, different rules — vendor-specific extensions to a standard, most visible across SQL implementations.

17.11 How to choose a language for a job

17.11.1 PLAIN - in simple words

1. "Which language is best" is not a real question, because it has no fixed subject. Best at what, for whom, with what deadline.
2. A better question has four parts: what must this program do, where must it run, who will maintain it, and how long must it live.
3. Constraints usually decide it before taste does. An iPhone app is Swift. A browser front end is JavaScript or TypeScript. A database query is SQL.
4. Where nothing forces the choice, the strongest factor is the team. A language nobody on the team knows costs six months of slower work.
5. The second strongest factor is the ecosystem. The right library in an average language beats a great language with nothing written for it.
6. Performance matters far less often than people believe, and far more when it matters.
7. You are also allowed to pick two. Most real systems use several.

17.11.2 PLAIN - a picture in your head

1. Choosing a language is like choosing a material for a building.
2. Nobody argues that steel is better than wood. They ask about the span, the climate, the budget, and who is available to work it.
3. A material also implies its trades. Choosing steel means hiring welders. Choosing Rust means hiring or training Rust programmers.
4. And once the frame is up, changing material means rebuilding. Language choices are cheap on day one and very expensive on day 800.

Where this comparison breaks: buildings cannot be half steel and half wood in the same wall, and software routinely is. A Python service calling a Rust extension is completely normal and often the right answer.

17.11.3 PLAIN - a worked example

1. **Scenario one: a machine-learning model for a research paper, six weeks.** Answer: Python. Not because it is fast, but because PyTorch, NumPy and pandas are there and every collaborator already reads it. The heavy numerical work happens in C++ and CUDA inside those libraries anyway.
2. **Scenario two: firmware for a battery-powered sensor with 64 KB of RAM.** Answer: C, or Rust if the team has the appetite. No garbage collector will fit, and you need exact control over every byte and every interrupt.
3. **Scenario three: an internal web tool for 200 staff, needed next month.** Answer: whatever the team already runs in production. Python with Django, Ruby with Rails, PHP with Laravel, or TypeScript with Node. The differences between these for this job are noise next to familiarity.
4. **Scenario four: a payments back-end that must run for fifteen years and be audited.** Answer: Java or C#. Long-term-support releases, deep tooling, large hiring pools, and a strong culture of backwards compatibility. Java 25 is supported into the 2030s.
5. **Scenario five: a network proxy handling 100,000 connections per second.** Answer: Go or Rust. Go if you want it working next month with cheap concurrency. Rust if the last 20 percent of performance and predictable latency without garbage collection pauses are worth the extra effort.
6. Notice that in five real scenarios, "which language do I like" appeared zero times.

17.11.4 PLAIN - what is really happening inside

1. The hidden costs of a language choice, in the order they bite you.
2. Hiring. A language with 50,000 practitioners in your city and one with 500 are different business risks.
3. Libraries. Check whether the exact thing you need exists and is maintained, before you decide, not after.
4. Operations. Can your team debug it at 3am. Do profilers, tracing and crash reporting exist for it.
5. Build and deploy. Go gives you one file. Python gives you a dependency tree and a virtual environment. That difference shows up every single day.
6. Longevity. Ask whether the language has a paid maintainer, a standards body, or a large company depending on it. Languages with none of those have died before.
7. Interop. If you can call C from it, you can reach almost anything. Nearly every language on the list can.
8. Rewrites are the most expensive way to change your mind. The usual honest advice is to rewrite one component, measure, and only then decide.

17.11.5 TECHNICAL - the engineer's version

1. Rough performance ranking on CPU-bound work, taking optimized C as 1. These are order-of-magnitude figures from public benchmark suites and vary hugely by workload, so treat them as ranking only.

Language	Relative CPU time	Startup
C, C++, Rust	1x	under 1 ms
Go	1x to 2x	a few ms
Java, C# (warm)	1x to 2x	100 ms to 1 s
JavaScript (V8)	2x to 10x	tens of ms
Python (CPython)	20x to 100x	tens of ms

2. Java and C# reach near-native speed only after the just-in-time compiler has warmed up, which is why start-up and steady-state must be quoted separately, and why ahead-of-time options such as GraalVM native image exist.
3. Python's figure collapses when the work is inside NumPy or PyTorch, because the loop then runs in C or CUDA and Python only issues the calls.

4. Decision inputs that are checkable rather than opinion: long-term-support policy and dates, the security advisory process, the package registry's provenance guarantees, the presence of an official formatter and linter, and whether the specification is a published standard or a single implementation.
5. Conway's law, published by Melvin Conway in 1968, applies: your system will mirror your communication structure, so a language choice that splits the team also splits the architecture.
6. Where experts genuinely disagree: whether a strong static type system pays for itself on small teams. Studies exist on both sides and none is conclusive, because the confounding variables are enormous.

17.11.6 WORDS - remember these

1. Ecosystem — the libraries and tools around it — the package registry, frameworks, profilers and community that surround a language.
2. Long-term support — a version kept safe for years — a release with a published multi-year window for security patches.
3. Warm-up — the first minute is slower — the period before a JIT compiler has optimized the hot paths.
4. Interop — calling other languages — a foreign function interface, usually via the C application binary interface.
5. Rewrite risk — changing your mind costs years — the cost of porting a working system, historically underestimated by a large factor.

17.12 Reading other people's code and writing readable code

17.12.1 PLAIN - in simple words

1. You will read far more code than you write, including your own from six months ago, which is somebody else's code.
2. So the real skill is not writing clever code. It is writing code the next person understands on the first read.
3. **Naming** is most of it. A name should say what the thing is, not what type it is and not how it works.
4. `d` tells you nothing. `days` tells you the unit. `days_since_signup` tells you the meaning.
5. A function should do **one thing**. If you need the word "and" to describe it, it is two functions.
6. Comments should explain **why**, not what. The code already says what.
7. A **style guide** is an agreement about layout so that nobody argues about it again.
8. A **formatter** rewrites your code into that layout automatically. A **linter** warns about patterns known to cause bugs.
9. **Code review** is another person reading your change before it ships.
10. **Technical debt** is a shortcut you took on purpose, that will cost more later, like a loan.

17.12.2 PLAIN - a picture in your head

1. Think of code as directions written for a stranger arriving at night.
2. "Turn left at the thing" is a bad name. "Turn left at the petrol station" is a good one.
3. A step that says "drive to the roundabout and also buy milk" is two steps.
4. A note saying "this road is one-way northbound" is a good comment. A note saying "turn left here" next to the instruction "turn left here" is noise.
5. A style guide is everyone agreeing to write distances in kilometres.
6. Technical debt is the shortcut through the field. Faster tonight. Impassable after rain, and eventually you must build the road anyway, having also paid for the field.

Where this comparison breaks: directions are read once and code is read for years, by people who will change it. Code must be not only understandable but safely modifiable, which is a much higher bar.

17.12.3 PLAIN - a worked example

1. Here is a real function, written badly.

```
def proc(d, f):
    r = []
    for x in d:
        if x[2] > f and x[4] == 1:
            r.append(x[0])
    return r
```

2. It works. Nobody can maintain it. What is `x[2]`, and what does `1` mean.
3. The same thing, rewritten.

```
def active_user_ids_above(users, min_age):
    """Return ids of active users older than min_age."""
    return [
        user.id
        for user in users
        if user.age > min_age and user.is_active
    ]
```

4. What changed: the function name says what comes out, the parameters are named, the fields are named instead of numbered, and the magic `1` became `is_active`.
5. Nothing got slower. The rewrite is the same number of machine operations.
6. Now comments. Here is a bad one and a good one.

```
i = i + 1              # add one to i
i = i + 1              # skip the header row
```

7. The first repeats the code and will rot the moment the line changes. The second records a fact about the data that the code cannot express.

17.12.4 PLAIN - what is really happening inside

1. How to read code you did not write, in order.
2. Start at the entry point: `main`, the route handler, the command. Do not start at the top of a file.
3. Find the data structures before the functions. Once you know what the program holds, the functions usually explain themselves.
4. Run it. Put a breakpoint or a print in the middle and look at real values. Five minutes of running beats an hour of reading.
5. Use the tools: jump to definition, find all references, and `git log -p` on a file to see how it grew and why.
6. Read the tests. A good test suite is a specification with worked examples.
7. Now writing for review. Keep changes small. A 50-line change gets a real review. A 2,000-line change gets a shrug and an approval, and that is a measured effect, not a joke.
8. Separate a refactor from a behaviour change. Two commits, two reviews. Mixed together, nobody can see which lines actually changed the program.
9. Technical debt is only debt if you intend to repay it. Write it down: a comment, a ticket, or a `TODO` with a name and a date. Undocumented shortcuts are not debt, they are just a mess.

17.12.5 TECHNICAL - the engineer's version

1. Formatters and linters, with dates:

Tool	Language	Since
lint	C	1978, Stephen Johnson
gofmt	Go	2009, no options at all
ESLint	JavaScript	2013, Nicholas Zakas
Prettier	JS, CSS, more	2017, James Long
Black	Python	2018, Lukasz Langa
rustfmt, clippy	Rust	2016 onward
Ruff	Python	2022, written in Rust

2. `gofmt` made a design decision worth copying: one canonical format with no configuration. That ends the discussion permanently, which was the point. Black and rustfmt followed the same approach.
3. Ruff is roughly one to two orders of magnitude faster than the Python-based linters it replaces, because it is written in Rust and does a single pass. That speed changes behaviour: a linter fast enough to run on every keystroke gets used.
4. PEP 8, the Python style guide, was written by Guido van Rossum, Barry Warsaw and Nick Coghlan in 2001. Its 79-character line limit is a convention, not a standard, and teams commonly raise it to 88 or 100.
5. Code review as a formal practice comes from Michael Fagan's inspection method at IBM, published in 1976. Modern lightweight pull-request review is a descendant with much of the ceremony removed.
6. Published review guidance, including Google's engineering practices documents, converges on: review under 400 lines at a time, under 60 minutes at a stretch, and expect defect detection to fall sharply outside those bounds.
7. "Technical debt" was coined by Ward Cunningham in a 1992 OOPSLA experience report. His original meaning was deliberately shipping a not-yet-right design to learn from it, then refactoring. The common modern usage, meaning any sloppy code, is a drift from what he wrote, and he said so publicly.
8. Cyclomatic complexity, defined by Thomas McCabe in 1976, counts independent paths through a function. A value above roughly 10 is a common review threshold. It is a signal, not a rule.
9. Tools: `git blame`, `git log -S` to find when a string appeared, `radon cc` for Python complexity, `golangci-lint`, `clang-tidy`, and `pre-commit` to run all of them before a commit lands.

17.12.6 WORDS - remember these

1. Refactor — change the shape, not the behaviour — a behaviour-preserving code transformation, ideally covered by existing tests.
2. Linter — a tool that warns about risky patterns — static analysis flagging likely defects and style violations without running the code.
3. Formatter — a tool that fixes layout — a deterministic rewriter producing a canonical source form.
4. Magic number — an unexplained literal — a bare constant with no name, carrying meaning nowhere recorded.
5. Code review — someone reads it before it ships — structured peer inspection of a change set prior to merge.
6. Technical debt — a shortcut with interest — deliberately deferred design work, tracked and intended to be repaid.
7. Cyclomatic complexity — how many ways through — the count of linearly independent paths through a function, McCabe 1976.

17.13 The standard library, packages and dependencies

17.13.1 PLAIN – in simple words

1. The **standard library** is the set of code that ships with the language itself. Sorting, files, dates, networking, text.
2. Everything else you use is a **package**: code someone else wrote, published somewhere, that you download.
3. A **package manager** is the tool that downloads packages and keeps track of which versions you have.
4. Your package depends on other packages, which depend on others. Those are **transitive dependencies**, and there are usually far more of them than you expect.
5. A **version number** tells you how much a new release might break. That is what semantic versioning is for.
6. A **lock file** records the exact versions you actually used, so that the build on your machine and the build on the server are identical.
7. **Supply chain risk** is the plain fact that installing a package means running someone else's code, and trusting everyone they trusted.
8. In March 2016, an eleven-line package was deleted and thousands of builds around the world broke within minutes.

17.13.2 PLAIN – a picture in your head

1. Think of building a car from parts catalogues.
2. The standard library is the parts that come in the box with the chassis.
3. A package is a part you order from a supplier. It arrives quickly and works.
4. But your supplier orders their bolts from someone else, who orders steel from someone else. You have contracts with one company and dependence on three hundred.
5. A lock file is writing down every part number and batch, so that next year's car is identical to this year's.
6. Supply chain risk is what happens when one bolt supplier decides to stop, or is bought by someone hostile, or ships a bad batch.

Where this comparison breaks: car parts are inspected, certified and physically present. Software dependencies are downloaded automatically by a script, often without a human ever looking, and a compromised one runs with all the privileges of your build machine.

17.13.3 PLAIN – a worked example

1. Semantic versioning, from the SemVer 2.0.0 specification written by Tom Preston-Werner. A version is three numbers.

```
2 . 7 . 3
|   |   |
|   |   +-- PATCH: bug fixes only, safe
|   +------ MINOR: new features, still compatible
+---------- MAJOR: something broke, read the notes
```

2. So going from 2.7.3 to 2.7.4 should be safe. Going from 2.7.3 to 3.0.0 means the authors deliberately broke something.
3. In an npm `package.json`, `"^2.7.3"` means "any 2.x.y at or above 2.7.3", and `"~2.7.3"` means "any 2.7.x at or above 2.7.3".
4. This is a convention, backed by a written specification, and it depends entirely on authors honouring it. They sometimes do not.
5. Now the transitive problem. Install one popular web framework and count.

```
npm ls --all | wc -l          # often over 1,000 lines
```

6. You chose one package. You installed several hundred, from several hundred different people, and you have read none of them.

7. The lock file is what makes this survivable. `package-lock.json`, `Cargo.lock`, `poetry.lock`, `go.sum` and `Gemfile.lock` all record the exact resolved version of every package, and usually a cryptographic hash of its contents.
8. Rule of thumb: commit the lock file for applications, so deployments are reproducible. For libraries, do not, so that whoever uses your library can resolve versions themselves.

17.13.4 PLAIN – what is really happening inside

1. The left-pad incident, with dates.
2. On 22 March 2016, a developer named Azer Koculu had a trademark dispute over a package named `kik`. The npm registry transferred that name to the company.
3. In protest he unpublished all 273 of his packages from npm.
4. One of them was `left-pad`: eleven lines of code that pad a string on the left with spaces.
5. It was a dependency of a dependency of Babel, and through Babel of React, webpack and thousands of build systems.
6. Within minutes, builds failed worldwide with a 404 error for a package almost nobody had chosen to install.
7. npm restored the package from backup about two hours later, which was itself unprecedented, and then changed policy: a package cannot be unpublished after 24 hours if anything depends on it.
8. The technical lesson is not "do not use small packages". It is that your build had a single point of failure you did not know existed, controlled by a person you had never heard of.
9. The security version of the same lesson is worse. If someone takes over that account, they do not break your build. They add three lines that read your environment variables and send them somewhere.
10. This has happened repeatedly. In November 2018 the `event-stream` package was handed to a new maintainer who added code targeting a cryptocurrency wallet. In late 2021 the accounts behind `ua-parser-js`, `coa` and `rc` were hijacked and used to ship malware.
11. The most sophisticated known case is the xz-utils backdoor, CVE-2024-3094, found on 29 March 2024 by Andres Freund, a PostgreSQL developer who noticed that SSH logins were taking half a second longer than expected. It was the result of a patient multi-year effort to become a trusted maintainer.

17.13.5 TECHNICAL – the engineer's version

1. Package managers by language:

Language	Manager	Registry
JavaScript	npm, yarn, pnpm	npmjs.com
Python	pip, uv, poetry	PyPI
Rust	Cargo	crates.io
Go	go modules	proxy.golang.org
Java	Maven, Gradle	Maven Central
C#	NuGet	nuget.org
Ruby	Bundler, gem	RubyGems
PHP	Composer	Packagist

2. Dates: npm launched January 2010, created by Isaac Schlueter. Maven 1.0 arrived in 2004. Cargo has shipped with Rust since 1.0 in 2015 and has never had a competitor, which is a deliberate design outcome. Go modules arrived in Go 1.11 in 2018 and became the default build mode in Go 1.16 in 2021, replacing GOPATH.
3. Go's approach differs on purpose: minimal version selection picks the lowest version satisfying all requirements, rather than the highest. Combined with a checksum database, this makes builds reproducible without a traditional resolver.

4. `go.sum` and the public checksum database, plus the module proxy, mean Go verifies that a module's contents have never changed since first seen. That is a stronger guarantee than most registries provide.
5. Diamond dependency conflicts occur when A needs C version 1 and B needs C version 2. npm resolves it by installing both, nested. Maven picks one by nearest-wins. Cargo allows two major versions to coexist. Python historically cannot, which is why virtual environments exist.
6. Supply chain defences that exist in 2026, with their status. Established: lock files with hashes, npm provenance attestations using Sigstore from 2023, PyPI trusted publishing, and two-factor authentication mandates on major registries. Active work: SLSA build-integrity levels and reproducible builds, which are real projects with partial adoption. Marketing claim: any product promising to "eliminate" supply chain risk.
7. Log4Shell, CVE-2021-44228, disclosed 9 December 2021 with a CVSS score of 10.0, was a vulnerability in a logging library that almost no Java team had consciously chosen. It is the clearest demonstration that transitive dependencies are your responsibility whether you know their names or not.
8. Standard library size is a design position. Go and Python ship large ones, Rust ships a deliberately small one and pushes the rest to crates.io, JavaScript ships almost nothing, which is exactly why `left-pad` existed.
9. Tools: `npm audit`, `pip-audit`, `cargo audit`, `govulncheck`, Dependabot, `syft` to generate a software bill of materials, and `grype` to scan one.

17.13.6 WORDS - remember these

1. Standard library — what comes in the box — the modules specified and shipped with the language implementation.
2. Package manager — the tool that fetches code — resolves version constraints, downloads artefacts and records them.
3. Transitive dependency — your dependency's dependency — a package pulled in indirectly, usually the majority of the tree.
4. Semantic versioning — major, minor, patch — SemVer 2.0.0, where a major bump signals a deliberate breaking change.
5. Lock file — the exact versions you used — a pinned resolution with content hashes, giving reproducible installs.
6. Supply chain risk — trusting strangers by default — the attack surface of every upstream package, maintainer account and build step.
7. Software bill of materials — a parts list — a machine-readable inventory of every component in a build, in SPDX or CycloneDX format.

17.98 Common wrong ideas

1. Wrong: some languages are simply better than others. Right: every language is a set of trade-offs, and the sensible question is what the job demands in speed, safety, team skill, ecosystem and lifespan.
2. Wrong: Python is a slow language. Right: a language is a written specification and has no speed. CPython is a slow implementation, and the same Python calling NumPy runs the actual work in optimized C.
3. Wrong: static typing proves a program is correct. Right: it proves the shapes fit together. A program that computes the wrong average with perfectly matched types passes every type check ever written.
4. Wrong: garbage collection means you cannot leak memory. Right: it means you cannot leak unreachable memory. A cache or a listener list that keeps growing is fully reachable, and will exhaust the heap exactly as a C leak would.
5. Wrong: `x += 1` is a single operation, so two threads cannot corrupt it. Right: it is a load, an add and a store, and a thread can be interrupted between any two of them, which is precisely how lost updates happen.

6. Wrong: Big-O tells you which code is faster. Right: it tells you how cost grows with input size. For small inputs a worse complexity with a smaller constant routinely wins, which is why library sorts fall back to insertion sort on short runs.
7. Wrong: recursion is always more elegant than a loop. Right: it costs a stack frame per call, and without tail-call elimination, which most mainstream languages do not guarantee, deep recursion overflows the stack.
8. Wrong: a null check everywhere is the same as an option type. Right: a null check can be forgotten and the compiler will not notice, whereas an option type cannot be unwrapped without handling the empty case.
9. Wrong: more tests always mean better code. Right: coverage measures which lines ran, not which behaviours were checked. A property-based test that states a real rule can be worth a hundred example tests.
10. Wrong: a tiny dependency is harmless because it is only eleven lines. Right: you are trusting an account, a registry and a maintainer, as the left-pad breakage of 22 March 2016 and the xz-utils backdoor of March 2024 both showed.

17.99 Chapter summary in 20 lines

1. A programming language is a notation with a grammar (syntax) and a meaning (semantics), defined by a document and implemented by separate tools.
2. Syntax errors are caught for you; semantic errors run happily and give the wrong answer, which is where the real difficulty lives.
3. Every language is built from the same parts: variables, literals, operators, expressions, statements, conditionals, loops, functions, scope and comments.
4. The same ten-line program in C, Python, JavaScript, Java, Go and Rust differs only in punctuation and ceremony, not in ideas.
5. A type is a set of values plus the legal operations on them; static means checked before running, dynamic means checked while running, and strong versus weak is the separate question of implicit conversion.
6. Tony Hoare added the null reference to ALGOL W in 1965 and called it his billion-dollar mistake in 2009; option types are the fix because they cannot be used without handling the absent case.
7. Manual memory management is fastest and admits leaks, double frees, use after free and dangling pointers; roughly 70 percent of reported vulnerabilities at Microsoft and Chromium were memory safety issues.
8. Reference counting is immediate and predictable but cannot reclaim cycles; tracing collection marks from roots and sweeps the rest, and generational collectors exploit the fact that most objects die young.
9. Rust’s ownership and borrowing move the whole question to compile time, giving memory safety with no runtime cost and a much stricter compiler.
10. Paradigms are habits of organization: imperative, procedural, object-oriented, functional, declarative and event-driven, and real programs mix them.
11. Encapsulation, inheritance and polymorphism are the object-oriented trio; deep inheritance is the honest weak point, which is why composition is preferred and why Go and Rust omit class inheritance entirely.
12. Pure functions, immutability and higher-order functions with map, filter and reduce have spread into every mainstream language, whether or not the language calls itself functional.
13. Data structures are trades: arrays give O(1) access and O(n) insertion, hash tables give O(1) average lookup by computing a bucket and resolving collisions, balanced trees give O(log n) with order preserved.
14. Big-O describes growth, not speed; O(n log n) sorting handles millions in a second while O(2^n) handles about 26 items, and constants and cache behaviour decide the rest.

15. Real libraries ship Timsort (Python 2002, Java 7 in 2011), introsort (C++, Musser 1997), dual-pivot quicksort (Java primitives, 2009) and pattern-defeating quicksort (Go 1.19, 2022).
16. Dynamic programming turns exponential recomputation into linear work by solving each overlapping subproblem once and writing the answer down.
17. Concurrency is structure and parallelism is hardware; a race condition is unsynchronized access where one access writes, fixed by locks, atomics, message passing, or by not sharing at all.
18. Async and await are one thread and an event loop, which scales to tens of thousands of waiting connections and stalls completely on real computation; CPython's GIL became optional in 3.13 in 2024 and supported in 3.14 in 2025.
19. Errors are reported by return codes, exceptions, error values or Result types; assertions state invariants, and unit, integration, end-to-end and property-based tests each catch a different class of mistake.
20. Choose a language from constraints, team and ecosystem rather than taste, write code the next reader understands, and treat every dependency as code you are choosing to run, as 22 March 2016 taught the whole industry.

18

CHAPTER

K

Operating Systems

18.0 What this chapter gives you

1. You will be able to say exactly what an operating system is, and name the three jobs it exists to do.
2. You will be able to describe what a kernel is, what lives inside it, and why almost every kernel is written in C with a little assembly.
3. You will be able to narrate a computer's boot, from the power supply's ready signal to the login prompt, with real timings from a real machine.
4. You will be able to explain a process, a thread, and the difference, and read the output of `ps` without guessing.
5. You will be able to run a scheduling algorithm by hand on paper and produce the same answer the machine would.
6. You will be able to explain what happens mechanically when your program asks the kernel for something, and read a real `strace` line by line.
7. You will be able to describe file descriptors, permissions, mounting, pipes, signals and shared memory, and say which one to reach for.
8. You will be able to explain containers, virtual machines and kernel panics in terms of one idea: privilege.
9. You will be able to tell the history of UNIX, Linux, Windows, macOS and Android with correct names and years.
10. You will know what building your own operating system actually involves, and which learning resources to start from.

18.1 What an operating system is and why it must exist

■ 18.1.1 PLAIN – in simple words

1. An operating system is a program whose job is to run other programs.
2. It is the first big program that starts when you switch the machine on, and the last one running when you switch it off.
3. It has three jobs, and only three. Everything else it does is a detail of one of them.
4. Job one: **share the hardware**. One processor, one disk, one screen, many programs that all want them. Somebody must decide who gets what and when.
5. Job two: **keep programs apart**. Your music player must not be able to read or wreck your bank app's memory, whether by accident or on purpose.
6. Job three: **hide the hardware behind something simple**. Your program says "write these bytes to this file". It does not say "spin up the disk, move the head, wait".
7. Without an operating system you can still run one program. That program must then do every one of those three jobs itself, for itself.

8. The operating system is not the desktop, not the windows, not the icons. Those sit on top. The operating system is the part underneath that decides.

18.1.2 PLAIN – a picture in your head

1. Think of a shared workshop with one lathe, one drill and one bench.
2. Twenty people want to build things. Left alone they would fight over the lathe, put each other's parts in the wrong bin and ruin each other's work.
3. So the workshop hires a manager.
4. The manager keeps a list of who wants the lathe and gives each person a turn of a few minutes. That is sharing the hardware.
5. The manager gives each person a locked drawer and refuses to let anyone open somebody else's drawer. That is keeping people apart.
6. The manager takes requests in plain words. You say "cut me a 40 mm disc". You do not set the lathe's speeds and feeds yourself. That is hiding the hardware behind something simple.
7. The manager also produces nothing. No customer ever pays for the manager's own work. The manager exists purely so that twenty people can share.

Where this comparison breaks: the workshop manager is a person with judgement, who can be argued with. The operating system is code, and it decides by fixed rules written years earlier by strangers. It also switches between jobs thousands of times a second, far faster than a human could hand over a lathe. And a real workshop manager cannot physically stop you opening a drawer. The operating system can, because the processor itself refuses the instruction.

18.1.3 PLAIN – a worked example

1. You double-click a music player. Here is what the operating system does that you never see.
2. It finds the program file on the disk, reads its header, and works out how much memory it needs.
3. It sets aside memory for it, and arranges that the player can only see its own memory and nothing else.
4. It creates a **process**: a record saying "this program is running, here is its memory, here are its open files, here is its identity".
5. It starts the program's first instruction, and hands the processor to it.
6. A few milliseconds later it takes the processor away again and gives it to your browser, then to a background updater, then back to the player.
7. The player asks to open a sound file. It does not know which disk, which cable or which controller. It says a name. The operating system does the rest.
8. The player asks to send sound to the speakers. The operating system mixes it with the alert sound from your mail app so both are heard.
9. You close the player. The operating system reclaims the memory, closes the files, and removes the record.
10. Count the operating system's work in that story: sharing, separating and simplifying. There is nothing else.

18.1.4 PLAIN – what is really happening inside

1. The trick that makes all of this possible is a feature of the processor itself, not of the software.
2. The processor can run in at least two modes. In one mode every instruction is allowed. In the other, dangerous instructions are refused.
3. The operating system's core runs in the all-allowed mode. Your programs run in the restricted mode.
4. In the restricted mode a program cannot change which memory it can see, cannot talk to devices directly, and cannot switch off interrupts.
5. So when a program needs any of those things it must ask. The asking mechanism is a special instruction that jumps into the operating system at a fixed, pre-arranged address.

6. The program does not choose where it lands. That is the whole point. It lands where the operating system decided, running the operating system's code.
7. A second hardware feature makes sharing possible: the timer interrupt. A chip pokes the processor at a fixed rate, say 250 times a second.
8. Each poke forces the processor to stop what it is doing and run operating system code. That is how a program that never gives up control is taken off the processor anyway.
9. Take away those two hardware features and no operating system can enforce anything. It becomes a library of helpful routines that programs may ignore.
10. That is exactly what life is like on a small embedded chip with no protection hardware, and it is why such systems trust their code completely.

18.1.5 TECHNICAL - the engineer's version

1. Formally, an operating system is a resource manager and a virtual machine provider. Both descriptions appear in the standard textbooks: Silberschatz, Galvin and Gagne's *Operating System Concepts*, and Tanenbaum and Bos's *Modern Operating Systems*.
2. The resource manager view: it multiplexes CPU time, physical memory, storage bandwidth, network bandwidth and device access among competing requesters.
3. The virtual machine view: it presents each process with the illusion of a private processor, a private flat address space and a private set of files.
4. The enforcement primitives are hardware. On x86-64 these are the four privilege rings (only 0 and 3 are used in practice), the paging unit driven by CR3, and the `syscall` and `sysret` instruction pair.
5. On ARMv8-A the equivalents are exception levels EL0 (application), EL1 (kernel), EL2 (hypervisor) and EL3 (secure monitor), plus the `svc` instruction for supervisor calls.
6. Without an OS you are on **bare metal**. Real categories of bare-metal software: bootloaders, PC firmware, and the tightest embedded control loops.
7. A microcontroller such as an ATmega328P (used in the Arduino Uno) has 2 KB of SRAM and no memory management unit. There is no meaningful isolation to enforce, so the usual answer is a `main` with an infinite loop.
8. Between bare metal and a full OS sits the real-time operating system (RTOS). FreeRTOS is roughly 6,000 to 9,000 lines of C and gives you tasks, a scheduler, queues and semaphores, with no memory protection by default.

Layer	Isolation	Typical size
Bare metal loop	None	100 to 5,000 lines
RTOS (FreeRTOS)	Optional MPU	under 10,000 lines
Linux kernel 6.12	Full MMU + rings	over 30 million lines
Windows NT kernel	Full MMU + rings	tens of millions

9. The Linux source tree line count is the whole tree including every driver and every architecture. Any single running kernel compiles a small fraction of it. Quoting "Linux is 30 million lines" as the size of a running kernel is wrong.
10. Tools that show you the boundary in action: `strace` on Linux, `dtruss` on macOS, Process Monitor on Windows, and `perf trace` on Linux.

18.1.6 WORDS - remember these

1. Operating system — the program that runs other programs — the resource manager and abstraction layer between hardware and applications.
2. Bare metal — no operating system at all — code executing directly on hardware with no supervisor and no runtime services.

3. Kernel mode — the mode where everything is allowed — the privileged execution state, ring 0 on x86-64 or EL1 on ARMv8-A.
4. User mode — the mode where programs are restricted — the unprivileged execution state, ring 3 or EL0.
5. Timer interrupt — a regular poke from a chip — a periodic hardware interrupt used to force scheduler entry and enable preemption.
6. Multiplexing — sharing one thing among many users — time-division or space-division allocation of a single resource among several requesters.

18.2 The kernel: the part that is really the operating system

18.2.1 PLAIN - in simple words

1. The **kernel** is the core of the operating system. It is the part that runs in the all-allowed mode of the processor.
2. It is a single program file on disk. On Linux it is often called `vmlinuz`. On Windows it is `ntoskrnl.exe`. On macOS it is the `kernel` file inside a kernel collection.
3. When people say "Linux" strictly, they mean only this file and its source code. Everything else you use is separate software.
4. Things that are inside the kernel: the scheduler, memory management, the file systems, the network stack, and most device drivers.
5. Things that are outside the kernel: the shell, the window system, the web browser, the compiler, the login screen, the package manager.
6. The dividing line is privilege, not importance. Your window system is very important and still runs as an ordinary program.
7. Code inside the kernel can do anything, so a mistake there can destroy the whole machine. Code outside can only wreck itself.
8. Designers therefore argue endlessly about how much should be inside.

18.2.2 PLAIN - a picture in your head

1. Think of a hospital. The kernel is the operating theatre and the staff allowed inside it.
2. Inside the theatre you may do anything to the patient. Nothing stops you. That power is needed, and it is dangerous.
3. Everything else in the hospital — reception, the cafe, the pharmacy counter, the ward — happens outside, under normal rules.
4. A **monolithic kernel** is a hospital where the theatre is huge: pharmacy, imaging, pathology and the blood bank are all inside the sterile zone. Fast, because nobody walks anywhere. Risky, because one contaminated glove contaminates everything.
5. A **microkernel** is a hospital where the theatre is a tiny room containing only the surgeon. Pharmacy, imaging and pathology are separate departments you telephone. Safer. Slower, because of all the telephoning.
6. A **hybrid** is a hospital that says "tiny theatre" in the brochure, and in practice keeps imaging and pathology inside anyway.

Where this comparison breaks: hospital departments genuinely cannot infect each other from down the corridor, whereas a microkernel's separated services still share the same physical memory chips and the same processor. The isolation is enforced by the memory management unit, not by distance. And a phone call between hospital departments costs seconds; a message between microkernel services costs about a microsecond. The relative costs are completely different.

18.2.3 PLAIN - a worked example

1. Ask: where does the code that understands the ext4 filesystem live?
2. In Linux, inside the kernel. It is a module, `ext4.ko`, loaded into kernel memory and running with full privilege.
3. In MINIX 3, a filesystem server is an ordinary user-mode process. If it crashes, MINIX restarts it and the machine keeps running.
4. In Windows, NTFS is `ntfs.sys`, a kernel-mode driver. Full privilege.
5. Now ask: where does the code that draws your windows live?
6. On Linux, outside the kernel entirely, in the X server or a Wayland compositor. The kernel only provides the low-level graphics device interface.
7. On Windows since NT 4.0 in 1996, a large part of the window manager and the graphics engine was moved **into** kernel mode for speed. That decision was controversial and caused years of graphics-driver blue screens.
8. Same job, three different sides of the line, for reasons of speed and history rather than principle.

18.2.4 PLAIN - what is really happening inside

1. A kernel is not a process. Nothing schedules it as a job. It is a body of code that gets entered when something happens.
2. There are exactly three ways in. Remember these three and kernel behaviour stops being mysterious.
3. Way one: a **system call**. A program deliberately executes the special instruction that jumps into the kernel.
4. Way two: an **interrupt**. A device raises a line, the processor stops the current instruction stream and runs a kernel handler.
5. Way three: an **exception** or fault. The program did something the processor could not complete: touched an unmapped page, divided by zero, executed an illegal instruction.
6. In all three cases the processor switches mode, switches to a kernel stack, and jumps to an address the kernel published in advance.
7. The kernel does the work, then returns, and the processor drops back to the restricted mode at the exact instruction that follows.
8. Between those events the kernel is doing nothing at all. An idle Linux machine really is idle. The kernel is a set of reactions, not a running loop.
9. There are exceptions to that: kernels also run their own threads for background work. On Linux you can see them in `ps` with names in square brackets, like `[kworker/0:1]`.

18.2.5 TECHNICAL - the engineer's version

1. **Monolithic**: all OS services execute in a single kernel address space in supervisor mode. Communication is a plain function call. Examples: Linux, FreeBSD, all classic UNIX.
2. **Microkernel**: the kernel provides only address spaces, threads and inter-process communication. Drivers, filesystems and network stacks run as user-mode servers. Examples: MINIX 3, QNX Neutrino, seL4, GNU Hurd.
3. **Hybrid**: a microkernel-derived structure with performance-critical services linked into kernel space. Examples: Windows NT, XNU. Some engineers reject "hybrid" as a marketing word for "monolithic with a Mach heritage". Say that both readings are defensible.
4. **Exokernel**: the kernel only multiplexes and protects hardware, exposing it almost raw; libraries in user space implement the abstractions. From MIT, Dawson Engler and Frans Kaashoek, 1995. Nearly all influence, almost no deployment. The idea resurfaced in unikernels and in Linux `io_uring`.
5. The **Tanenbaum-Torvalds debate** began on 29 January 1992 in the `comp.os.minix` newsgroup, in a post by Andrew S. Tanenbaum, author of MINIX and of the standard textbook, titled "LINUX is obsolete".
6. Tanenbaum's two claims: monolithic kernels are the wrong design, and writing one in 1991 was "a giant step back into the 1970s"; and Linux was too tied to the Intel 386 to survive changes in hardware.

7. Torvalds's replies: microkernels are theoretically nicer but MINIX had real design faults, targeting the 386 was a deliberate choice for a learning project, and application-level portability is what users actually feel.
8. How it turned out, honestly: Tanenbaum was right about portability mattering and Linux was later ported to more than 20 architectures. He was wrong about the market outcome. Torvalds was right that a working monolithic kernel beat an elegant unfinished one. Neither side won the design argument outright, and seL4's formal proof of correctness in 2009 showed the microkernel case is still very much alive.

Kernel	Structure	Primary languages
Linux	Monolithic, modular	C, asm, Rust
Windows NT	Hybrid	C, C++, asm
XNU (macOS)	Hybrid, Mach + BSD	C, C++, asm
MINIX 3	Microkernel	C
seL4	Microkernel, verified	C, asm

9. Why C: it compiles to predictable machine code with no hidden runtime, no garbage collector, no exceptions and no implicit allocation. A kernel cannot depend on services that only exist once the kernel is running.
10. Why assembly is still needed: the very first instructions after reset, context switch, interrupt entry and exit, setting control registers, and atomic primitives. In Linux 6.x the `arch/` directory holds this, a few tens of thousands of assembly lines against millions of lines of C.
11. Why C++ appears in places: XNU's IOKit driver framework uses a restricted subset of Embedded C++ with no exceptions, no templates and no multiple inheritance. Windows drivers use C++ widely.
12. **Rust in Linux**: Miguel Ojeda's Rust for Linux work was merged for Linux 6.1, released 11 December 2022. The first production Rust drivers landed in Linux 6.8 in March 2024, including the Android Binder driver and two Ethernet PHY drivers. As of December 2025 Rust is an official kernel language alongside C and assembly, not an experiment.
13. Why Rust: about two-thirds of serious kernel security bugs historically come from memory-safety errors — use after free, buffer overflow, data races. Rust's borrow checker rejects most of those at compile time with no runtime cost. This is established, not marketing: Microsoft and Google have both published the roughly 70 percent figure for their own codebases.
14. Windows also ships Rust in the kernel: Microsoft stated in 2023 that parts of the Windows kernel, including a rewritten `win32kbase` font parser and DWM core components, are now Rust.

18.2.6 WORDS - remember these

1. Kernel — the core that has full power — the privileged portion of the OS that executes in supervisor mode.
2. Monolithic kernel — one big privileged block — all services share a single kernel address space and call each other directly.
3. Microkernel — a tiny privileged core — only address spaces, threads and IPC are privileged; other services run as user-mode servers.
4. Kernel module — a piece of kernel you can add later — dynamically loadable object linked into the running kernel's address space.
5. Interrupt — a device raising its hand — an asynchronous hardware signal that diverts the processor into a registered handler.
6. Exception — the processor could not continue — a synchronous fault such as a page fault, divide by zero or invalid opcode.

18.3 The boot sequence, step by step

18.3.1 PLAIN - in simple words

1. Booting is the process of getting from "no power" to "a working machine", using nothing but what is already stored in the hardware.
2. The word comes from "pulling yourself up by your own bootstraps", which is the impossible thing this process appears to do.
3. It is a chain. Each stage is small and stupid, and its only real job is to find and start the next, slightly cleverer stage.
4. Stage 0: the power supply tells the rest of the machine that the voltages are stable and it is safe to start.
5. Stage 1: the processor starts executing at one fixed address that is burned into the chip's design. It always starts there. It cannot start anywhere else.
6. Stage 2: at that address is **firmware**, a small program in a chip on the board. It tests the machine and finds something to boot from.
7. Stage 3: the firmware loads a **bootloader** from disk. The bootloader knows how to find the operating system.
8. Stage 4: the bootloader loads the kernel into memory and jumps into it.
9. Stage 5: the kernel sets itself up, finds the real disk, and starts the very first ordinary program.
10. Stage 6: that first program starts everything else, and eventually you get a login prompt.

18.3.2 PLAIN - a picture in your head

1. Think of arriving at a large office building at night, with only a note in your pocket saying "go to the front desk".
2. At the front desk is a laminated card. It says: check the lights work, check the lifts work, then take the lift to floor 3.
3. On floor 3 is another card: it says which of several offices you want, and what the key code is.
4. In that office is a thick manual. Reading it takes a while. It tells you how to run the whole building.
5. Once you have read the manual, you telephone the staff and they arrive one by one and start their own jobs.
6. Finally the reception desk opens and members of the public can come in. That is your login prompt.
7. Notice that no stage had to know everything. Each stage only had to know enough to find the next.

Where this comparison breaks: you are one person walking through a building, whereas the real sequence involves several processors. On a modern x86 machine the main processor is not even the first thing that runs; a separate management engine inside the chipset starts first and holds the main processor in reset. And the "cards" are not just instructions, they are complete programs with drivers, network stacks and graphics, sometimes larger than early operating systems.

18.3.3 PLAIN - a worked example

1. Here is a real boot, measured on the Linux machine this chapter was written on. The times are seconds since the kernel took control, printed by `dmesg`.

```
0.000000  Linux version 6.18.5, gcc 15.2.0
0.000064  tsc: Detected 2100.000 MHz processor
0.165101  Booting paravirtualized kernel on KVM
0.173953  random: crng init done
0.475547  smpboot: CPU0: Intel(R) Xeon(R) @ 2.10GHz
0.494363  smpboot: Total of 2 processors activated
0.633029  Memory: 8203096K/8388216K available
0.636116  devtmpfs: initialized
0.927746  Unpacking initramfs...
1.039733  virtio_blk virtio1: [vda] 536870912 blocks
1.162061  Freeing unused kernel image memory: 2892K
```

```
1.163192  Write protecting kernel read-only data: 26624k
1.174639  Run /process_api as init process
2.551159  EXT4-fs (vda): mounted filesystem r/w
```

2. Read that from the top. For the first 0.165 seconds the kernel is doing nothing but working out what machine it is on.
3. At 0.475 the first processor is identified, and by 0.494 the second processor has been woken up and put to work. Before that moment the machine had one working core no matter how many it owns.
4. At 0.633 the memory manager reports what it has: 8,203,096 KB usable out of 8,388,216 KB total. The missing 185 MB is the kernel itself plus reserved regions.
5. At 0.927 the kernel unpacks the **initial ramdisk**, a small filesystem held in memory that contains the drivers needed to reach the real disk.
6. At 1.039 the block device driver finds the disk. Only now does a disk exist as far as the kernel is concerned.
7. At 1.162 the kernel frees its own setup code, which will never run again. 2,892 KB returned to the pool.
8. At 1.174 the kernel starts the first user-mode program. On this machine that is `/process_api`. On a normal Linux desktop the line reads `Run /sbin/init as init process`.
9. At 2.551 the real root filesystem is mounted. Everything after this point is ordinary software.
10. The whole kernel phase took 2.55 seconds, and 1.37 of those seconds were spent after the first user program had already started.

18.3.4 PLAIN - what is really happening inside

1. **Power good.** The power supply raises a signal when its outputs are within tolerance. Until then the whole board is held in reset. On a desktop this takes roughly 100 to 500 milliseconds after you press the button.
2. **Reset vector.** The processor comes out of reset with its registers at fixed values and starts fetching from one address chosen by the chip designers. Nothing is loaded from disk yet, because nothing can read a disk.
3. That address is wired so that it lands in the firmware chip, not in RAM. RAM is not even usable yet, because the memory controller has not been set up.
4. **Firmware.** The firmware runs. It configures the memory controller, tests the machine, and builds tables describing what hardware exists.
5. The self test at this stage is called POST, power-on self test. If it fails before the screen works, the machine reports the failure by beeping in a pattern or flashing a light, because that is all it has.
6. **Boot device selection.** The firmware works down a list of places to look: internal disk, USB, network. You can edit this list in the firmware settings.
7. **Bootloader.** The firmware loads a small program from the chosen device and jumps to it. This program's whole job is to find the kernel.
8. Why a separate bootloader exists at all: the firmware does not know what a Linux kernel is, and the kernel is too big and too clever to be started directly by dumb firmware.
9. **Loading the kernel.** The kernel file on disk is usually compressed. The bootloader loads it, and a small decompressor stub at its front expands the real kernel into memory and jumps to it.
10. **Initial ramdisk.** The bootloader also loads a second file: a compressed archive of a tiny root filesystem. The kernel unpacks it into memory.
11. Why: the real disk might need a driver, and that driver might live on the real disk. The ramdisk breaks the circle by carrying the driver with it.
12. **Kernel initialization.** The kernel sets up its page tables, starts the other processors, initializes the scheduler and memory manager, and probes for devices.
13. **Mounting the root filesystem.** With a working disk driver, the kernel mounts the real root and switches to it, then throws away the ramdisk.

14. **The first user process.** The kernel executes one program in user mode and gives it process ID 1. If that program ever exits, the kernel panics, because there is nothing left to run.
15. **Services.** Process 1 reads its configuration and starts everything else: logging, networking, the sound daemon, the display manager.
16. **Login.** The display manager or the terminal getty draws a prompt. The boot is over.

18.3.5 TECHNICAL - the engineer's version

1. On x86-64 the reset vector is physical address `0xFFFFFFF0`, sixteen bytes below the top of the 4 GiB space, executed in 16-bit real mode with CS base set so the first fetch lands in the firmware flash. This is fixed by Intel and AMD, not by software.
2. The processor comes out of reset in real mode even on a 2026 machine, and the firmware must walk it through protected mode to long mode. That is 1978 compatibility still costing time in every boot.
3. On ARMv8-A the reset address is implementation defined and supplied by the SoC; the boot core starts at EL3 and steps down to EL2 and EL1.
4. **BIOS** is the legacy firmware interface, from the IBM PC of 1981. It reads the first 512-byte sector, the master boot record, checks for the signature `0x55AA` at offset 510, loads it at `0x7C00` and jumps there. 446 bytes of code is all you get.
5. **UEFI** replaced it. Intel began the work in 1998 for Itanium as the Intel Boot Initiative, later EFI; Intel stopped at version 1.10 and handed it to the Unified EFI Forum, which published UEFI 2.0 in 2006. The current specification is UEFI 2.11, published in December 2024.
6. UEFI boots differently: it reads a GPT-partitioned disk, mounts the EFI System Partition, which the specification requires to be FAT12, FAT16 or FAT32, and executes a PE-format `.efi` executable from it.
7. Standard ESP paths: `\EFI\BOOT\BOOTX64.EFI` is the fallback the specification mandates; `\EFI\Microsoft\Boot\bootmgfw.efi` is Windows Boot Manager; `\EFI\ubuntu\grubx64.efi` is a distribution's GRUB. The first is a standard. The others are conventions.
8. **Secure Boot** is a UEFI feature: firmware verifies the signature on the loaded image against keys in the `db` variable. Most distributions use a small Microsoft-signed `shim` that then verifies the distribution's own key.
9. **GRUB 2** is the common Linux bootloader. It reads the filesystem, presents a menu, and uses the Linux boot protocol: it loads `vmlinuz` and `initrd`, fills in a `boot_params` structure, and jumps to the kernel entry point.
10. The kernel command line is passed as a string. On the machine used above it contained `rdinit=/process_api`, which is what caused PID 1 to be that program rather than `/sbin/init`.
11. The initial ramdisk on modern Linux is an **initramfs**: a cpio archive, usually compressed with gzip, zstd or lz4, unpacked into a tmpfs. The older `initrd` was a block-device image. Both names are still used loosely.
12. **PID 1** in practice: `systemd` on most Linux distributions since Fedora 15 in May 2011, Arch in October 2012, and Debian and Ubuntu from 2015; `launchd` on macOS since Mac OS X 10.4 Tiger in 2005, written by Dave Zarzycki; `wininit.exe` and `smss.exe` on Windows; `init` from BusyBox or OpenRC on smaller Linux systems.
13. Typical timings on a real x86-64 laptop, measured with `systemd-analyze`. These are approximate and vary widely by machine.

Phase	Typical time	Tool to measure
Firmware (UEFI)	1 to 12 s	systemd-analyze
Bootloader	0.1 to 1 s	systemd-analyze
Kernel to PID 1	1 to 4 s	dmesg timestamps
Userspace to login	1 to 15 s	systemd-analyze blame

14. Commands that observe this: `dmesg` for kernel timestamps, `systemd-analyze` for the phase split, `systemd-analyze blame` for the slowest services, `systemd-analyze critical-chain` for the dependency path, `bootctl` for UEFI state, and `efibootmgr -v` for the boot entry list.
15. The honest version: the sequence described here starts at the main processor's reset vector, and on modern x86 that is not the true beginning. Intel's Management Engine and AMD's Platform Security Processor run first, verify the firmware, and release the main cores. You cannot observe or disable that from the operating system.

```
power good
    |
    v
CPU reset vector  0xFFFFFFF0  (real mode)
    |
    v
firmware: POST, memory init, ACPI tables
    |
    v
boot device list -> ESP -> BOOTX64.EFI / GRUB
    |
    v
load vmlinuz + initramfs, fill boot_params
    |
    v
kernel: decompress, page tables, SMP, drivers
    |
    v
unpack initramfs -> find real root -> switch_root
    |
    v
exec PID 1 (systemd / launchd / init)
    |
    v
services -> display manager -> login prompt
```

18.3.6 WORDS - remember these

1. Bootstrapping — starting from nothing by stages — a chain loader sequence in which each stage loads and transfers control to the next.
2. Reset vector — the one address a CPU starts at — the architecturally fixed fetch address after reset, `0xFFFFFFF0` on x86-64.
3. Firmware — the program in a chip on the board — non-volatile platform code implementing BIOS or UEFI services.
4. POST — the switch-on self test — power-on self test, reporting faults by beep codes or diagnostic LEDs before video exists.
5. Bootloader — the program that finds the kernel — a chain-loading stage such as GRUB 2, Windows Boot Manager or `boot.efi`.
6. Initramfs — a tiny filesystem carried in memory — a compressed cpio archive unpacked into tmpfs, holding the drivers needed to mount the real root.
7. PID 1 — the first ordinary program — the init process, ancestor of all user processes; its exit causes a kernel panic.

18.4 Processes

18.4.1 PLAIN – in simple words

1. A **program** is a file on disk. It does nothing. It is just bytes.
2. A **process** is that program while it is running: the code, plus its memory, plus its current position, plus everything the system is holding for it.
3. The same program can be running as many processes at once. Ten terminal windows are ten processes from one program file.
4. Each process gets a number, the **process ID**, usually written PID. It is how everything else refers to it.
5. Every process except the very first has a **parent**: the process that created it. That gives you a family tree with PID 1 at the root.
6. A process is not always running. Most of the time most processes are asleep, waiting for a key press, a disk read or a network packet.
7. When a process finishes, it does not vanish immediately. It leaves behind a small note saying how it ended, and waits for its parent to read it.
8. If the parent never reads that note, the dead process stays in the table as a **zombie**: dead, but still listed.
9. If a parent dies before its child, the child is adopted by PID 1. Such a child is called an **orphan**.

18.4.2 PLAIN – a picture in your head

1. Think of a cookery book on a shelf. That is the program.
2. Now think of you, in a kitchen, halfway through a recipe from it. That is the process.
3. The process is not the recipe. It is the recipe **plus** the ingredients on the bench, the pan on the heat, your finger on the current line, and the oven timer you set.
4. Your sister can cook the same recipe in a different kitchen at the same time. Same program, two processes, completely separate benches.
5. The doorbell rings. You mark your place, wipe your hands and answer it. When you come back you read your mark and continue. That is a context switch.
6. If you leave the kitchen for good, someone must still come and clear up and note whether the dish worked. Until they do, the mess stays on the bench. That is a zombie.

Where this comparison breaks: you have one pair of hands, so you genuinely stop cooking when you answer the door. A modern processor has several cores and can truly cook several dishes at once. And your memory of where you were is in your head; a process's memory of where it was is written into a data structure that another program could, with enough privilege, read or alter.

18.4.3 PLAIN – a worked example

1. Here is a real program that creates a child and replaces the child with a different program. This is how every UNIX shell starts every command.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int main(void) {
    printf("before fork: pid=%d ppid=%d\n",
           getpid(), getppid());
    pid_t p = fork();
    if (p == 0) {
        printf("child : pid=%d ppid=%d fork gave %d\n",
               getpid(), getppid(), p);
        execl("/bin/echo", "echo", "child is now echo",
              NULL);
        _exit(127);
    }
    printf("parent: pid=%d fork gave %d\n", getpid(), p);
    int st; waitpid(p, &st, 0);
    printf("parent: child %d exited with %d\n",
```

```
            p, WEXITSTATUS(st));
    return 0;
}
```

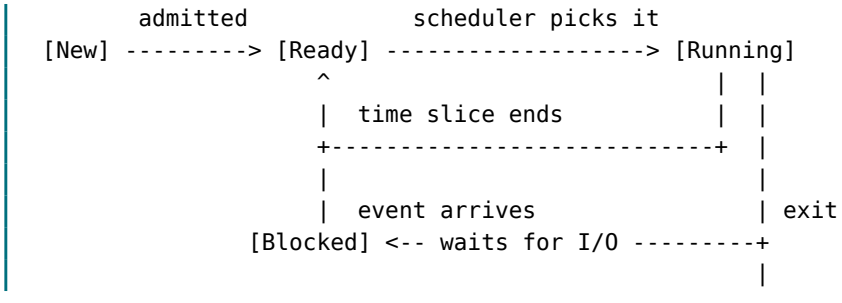
2. Compiled with `gcc` and run, it printed this, exactly:

```
before fork: pid=5344 ppid=4984
child : pid=5346 ppid=5344 fork gave 0
child is now echo
parent: pid=5344 fork gave 5346
parent: child 5346 exited with 0
```

3. Read it carefully. `fork` was called once and returned twice, in two different processes.
4. In the child it returned 0. In the parent it returned 5346, the child's PID. That is the only way each side can tell which one it is.
5. The child's own PID is 5346 and its parent PID is 5344, which is the parent. The family link is real and visible.
6. Then `execl` replaced the child completely. Same PID 5346, entirely different program. The line "child is now echo" was printed by `/bin/echo`.
7. The parent waited, and collected the exit status 0.
8. One honest detail. When the same program's output was sent into a pipe instead of a screen, the line `child : pid=...` never appeared at all. The C library buffers output when it is not a terminal, the text was still sitting in the buffer, and `execl` threw the whole buffer away when it replaced the process image. This is a real bug people hit; the fix is `fflush` before `exec`.

18.4.4 PLAIN - what is really happening inside

1. The kernel keeps one record per process. Textbooks call it the **process control block**. Linux calls it `struct task_struct`.
2. That record holds, at minimum: the process ID, the parent's ID, the state, the saved register values, a pointer to the memory map, the table of open files, the owning user, the scheduling priority and the accounting counters.
3. On Linux 6.x `struct task_struct` is a few kilobytes and has well over a hundred fields. All processes live in a linked structure the kernel walks.
4. A process moves between a small number of **states**. There are five that matter.
5. **New**: the record exists but the process has not started running.
6. **Ready**: it could run right now, it just does not have a processor.
7. **Running**: it is executing on a processor this instant.
8. **Blocked** or waiting: it asked for something that has not arrived, such as a disk read, and cannot proceed.
9. **Terminated**: it has finished but the record has not yet been removed.
10. Only three transitions are interesting. Ready to running is the scheduler choosing it. Running to ready is the scheduler taking the processor away. Running to blocked is the process itself asking for something.
11. Note the asymmetry: blocked goes to ready, never straight to running. When your disk read finishes you become eligible; you do not jump the queue.



```
                                                    v
                                                [Terminated]
```

12. Creating a process on UNIX is two steps, deliberately. `fork` makes a copy of the calling process. `exec` throws away the copy's program and loads a new one in its place.
13. That looks wasteful and is not, because `fork` does not really copy the memory. It marks every page as shared and read-only, and only makes a real copy of a page when one side writes to it. This is **copy on write**.
14. Windows does it in one step instead. `CreateProcess` takes the program name and a pile of options and makes a new process running that program. There is no fork.
15. The two-step design is why UNIX shells are simple: between `fork` and `exec` the child can quietly rearrange its own file descriptors, and the new program inherits the arrangement without knowing anything about it.

18.4.5 TECHNICAL - the engineer's version

1. Linux does not really have separate `fork`, `vfork` and thread creation. All three are `clone()` with different flag sets. `fork` is `clone` with `SIGCHLD` and nothing shared.
2. This trace, captured with `strace` on the shell command `sh -c 'echo hello world | tr a-z A-Z'`, shows a real fork:

```
clone(child_stack=NULL,
      flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID
            |SIGCHLD,
      child_tidptr=0x7f26e1971a10) = 18232
```

3. Process states in `ps` on Linux, from the `STAT` column: `R` runnable or running, `S` interruptible sleep, `D` uninterruptible sleep (usually disk), `T` stopped, `Z` zombie, `I` idle kernel thread. Suffixes: `s` session leader, `l` multi-threaded, `+` foreground, `<` high priority, `N` low.
4. `D` state deserves special mention. A process in uninterruptible sleep cannot be killed, not even with signal 9, because it is inside a kernel path that cannot safely unwind. Persistent `D` state means broken storage or a hung network filesystem.
5. A real zombie, produced deliberately by forking a child that exits while the parent does not call `wait`:

```
  PID  PPID STAT COMMAND
20066 20065 Z    zombie
```

6. A real orphan, produced by a parent that exits before its child. The child printed its parent ID twice, one second apart:

```
parent 23414: exiting immediately
child 23415: ppid is 23414
child 23415: ppid is now 1
```

7. Reparenting to PID 1 is the kernel's rule. On systemd systems a process can instead register itself as a subreaper with `prctl(PR_SET_CHILD_SUBREAPER)`, and orphans go to it rather than to PID 1.
8. PID limits: the default maximum on 64-bit Linux is 4,194,304, set in `/proc/sys/kernel/pid_max`. The historical default of 32,768 came from a 16-bit signed field. PIDs wrap around and are reused, which is a real source of race conditions in scripts that kill by PID.
9. Per-process limits are visible in `/proc/PID/limits`. On the machine used here:

Limit	Soft	Hard
Max stack size	8388608	unlimited
Max processes	32056	32056
Max open files	20000	20000

Limit	Soft	Hard
Max core file size	0	unlimited

10. `/proc/PID/stat` is the raw record, 52 space-separated fields. The first seven are pid, comm, state, ppid, pgrp, session, tty. Field 14 and 15 are user and system jiffies consumed. `ps` and `top` read exactly this file.
11. Windows: `CreateProcessW` takes 10 parameters including `STARTUPINFO` and `PROCESS_INFORMATION`. A Windows process starts with one thread and a PEB (process environment block). Handles are inherited only if marked inheritable and `bInheritHandles` is true.
12. Observation tools by platform: `ps aux`, `ps -ef`, `pstree`, `top`, `htop` and `/proc` on Linux; `ps`, `top` and Activity Monitor on macOS; `dtruss` and `sample` on macOS; Task Manager, Process Explorer, Process Monitor and `Get-Process` on Windows.
13. Useful `ps` incantations. `ps -eo pid,ppid,stat,pri,ni,etime,comm` prints the fields this section talks about. `ps -eLf` shows one line per thread. `ps -eo pid,rss,vsz,comm --sort=-rss` finds memory hogs.

18.4.6 WORDS - remember these

1. Program — a file of instructions — a passive executable image on storage.
2. Process — a program while running — an execution instance with its own address space, descriptor table, credentials and scheduling state.
3. Process control block — the kernel's record of a process — `task_struct` on Linux, EPROCESS on Windows.
4. PID — a process's number — a reusable integer identifier, capped by `/proc/sys/kernel/pid_max`.
5. fork — make a copy of yourself — `clone()` creating a new address space via copy-on-write page sharing.
6. exec — become a different program — replace the current process image while keeping the PID and inherited descriptors.
7. Zombie — dead but still listed — a terminated process whose exit status has not yet been reaped by `wait`.
8. Orphan — a child whose parent died — a process reparented to PID 1 or to the nearest subreaper.
9. Copy on write — pretend to copy, copy only if changed — shared read-only page mappings duplicated lazily on the first write fault.

18.5 Threads

18.5.1 PLAIN - in simple words

1. A **thread** is one flow of execution: one finger moving down the list of instructions.
2. A process has at least one thread. It may have many.
3. All the threads in one process share the same memory. If one writes to a variable, the others see the change instantly.
4. Each thread has its own **stack**, because each thread is in a different place in the program and has its own local variables and its own chain of calls.
5. That is the entire difference. Shared memory, separate position and stack.
6. Threads are used when several parts of one job must happen at once: one thread drawing the screen, another loading a file, another talking to the network.
7. Because they share memory, threads are fast to create and cheap to talk between.
8. Because they share memory, one bad thread can corrupt the data of all the others, and there is no protection at all.
9. That trade is the whole story of threads: speed and sharing, bought with danger.

18.5.2 PLAIN - a picture in your head

1. A process is a flat with a locked front door. A thread is a person living in it.
2. Two flats cannot see into each other. That is process isolation, enforced by the lock.
3. Two flatmates share the kitchen, the fridge and every cupboard. Neither needs permission to open anything. That is thread sharing.
4. Each flatmate still has their own bedroom with their own things. That is the per-thread stack.
5. If one flatmate takes the milk while the other is pouring it, you get a mess. Nobody broke a rule. They just both used one thing at once. That is a race condition.
6. Moving into a spare bedroom is quick. Getting a whole new flat takes longer and costs more. That is why threads are cheaper than processes.

Where this comparison breaks: flatmates can talk and agree who uses the kitchen. Threads cannot agree about anything unless you write the agreement yourself, in the form of locks. And a flat has one kitchen; a multi-core processor gives each thread its own set of caches, so two threads can genuinely believe different things about the same variable until the caches are made to agree.

18.5.3 PLAIN - a worked example

1. Measured on the machine used for this chapter, a 2.10 GHz Intel Xeon with two cores, Linux 6.18, glibc on Ubuntu 24.04.
2. One thousand threads created, then all joined. One thousand processes forked, then all reaped.

```
pthread_create only        :    32.8 us
fork only (small process)  :   182.5 us
fork only (256 MB dirty)   :  2620.1 us
```

3. Creating a thread cost 32.8 microseconds. Creating a process cost 182.5 microseconds. The thread is about 5.6 times cheaper.
4. Now the third line. The same fork, after the parent had allocated and written to 256 MB of memory, cost 2,620 microseconds. Fourteen times more.
5. Why: copy-on-write does not copy the data, but it must still copy the page tables that describe it. 256 MB is 65,536 pages of 4 KB, and every entry must be duplicated and marked read-only.
6. The honest version: "a thread is far cheaper than a process" is only true sometimes. Fork's cost depends on how much memory the parent has mapped. Thread creation does not depend on that at all.
7. A separate measurement, thread create and join one at a time, gave 80.2 microseconds against 134.3 for fork and wait. The gap narrows because the join and the wait dominate.
8. Take the lesson, not the numbers: threads are cheaper, but on Linux the gap is a factor of a few, not a factor of a thousand. On Windows the gap is larger, because Windows process creation is genuinely heavier.

18.5.4 PLAIN - what is really happening inside

1. Here is exactly what is shared between two threads of one process, and what is not.
2. Shared: the code, the global variables, the heap, the open file table, the current directory, the user identity, the signal handlers.
3. Not shared: the stack, the register values including the program counter, the thread ID, the signal mask, and any thread-local storage.
4. When the kernel switches from thread A to thread B in the same process, it saves A's registers and loads B's, and that is nearly all.
5. When it switches between threads of different processes, it must also switch the memory map, which means loading a new page table root and, on older processors, flushing the address translation cache.
6. That flush is why cross-process switches cost more than same-process ones. Modern processors avoid most of it with tagged translation caches: PCID on x86-64 since Westmere in 2010, ASID on ARM.
7. There are two places threads can be implemented.

8. **Kernel-level threads**: the kernel knows about each one and schedules them individually. If one blocks on disk, the others keep running. This is what Linux, Windows and macOS do.
9. **User-level threads**: a library inside your process switches between them, and the kernel sees only one thread. Switching is very fast because no mode change happens. But if one blocks in the kernel, all of them stop.
10. Modern languages have revived the user-level idea under new names: goroutines in Go, virtual threads in Java 21, `async` tasks in Rust and Python. They pair it with a runtime that never blocks the underlying kernel thread, which removes the old flaw.
11. A **thread pool** exists because creating a thread costs tens of microseconds and most tasks are shorter than that. You create a fixed set of threads once and feed them work from a queue for the life of the program.

18.5.5 TECHNICAL - the engineer's version

1. The POSIX threads API is IEEE Std 1003.1c-1995, universally called pthreads. The core calls are `pthread_create`, `pthread_join`, `pthread_detach`, `pthread_mutex_lock` and `pthread_cond_wait`.
2. On Linux a pthread is created by `clone()` with `CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLON` The kernel calls the result a task; there is no separate thread object.
3. Because of `CLONE_THREAD`, all threads of a process share a thread group ID, which is the value `getpid()` returns. The per-thread identifier is the TID, returned by `gettid()`. `ps -eLf` shows both as `PID` and `LWP`.
4. Default thread stack size on glibc is the process stack rlimit, typically 8 MiB, reserved as virtual address space and committed lazily. On musl it is 128 KiB. On Windows the default reserve is 1 MiB with 4 KiB committed.
5. That 8 MiB reservation is why you cannot have a million pthreads on a 64-bit system with default settings and a modest `vm.max_map_count`, even though the memory is never touched.
6. Threading models, in the classic taxonomy: 1:1 (one kernel thread per user thread; Linux NPTL, Windows), N:1 (all user threads on one kernel thread; green threads), M:N (a runtime multiplexes M user threads onto N kernel threads; Go, Erlang, Java 21 virtual threads).
7. Linux abandoned M:N deliberately. NPTL, the Native POSIX Thread Library by Ulrich Drepper and Ingo Molnár, shipped in glibc 2.3.2 in 2003 and replaced LinuxThreads with a strict 1:1 model, because the kernel scheduler was made good enough that a userspace one added nothing.

Model	Kernel sees	Blocking one thread
1:1 NPTL, Windows	Every thread	Others keep running
N:1 green threads	One thread	All stop
M:N Go, Erlang	N carriers	Runtime reschedules

8. Thread pool sizing rules of thumb. CPU-bound work: pool size equal to the core count. I/O-bound work: cores multiplied by one plus the ratio of wait time to compute time. Both are heuristics, not laws.
9. Real pool defaults: Java's `ForkJoinPool.commonPool` uses cores minus one; .NET's ThreadPool starts at the core count and grows by one thread every 500 ms under starvation; nginx uses one worker process per core with an event loop rather than a thread pool.
10. Observation: `ps -eLf`, `top -H`, `htop` with thread view on Linux; `/proc/PID/task/` lists one directory per thread; `perf sched` shows switch events; Windows Performance Analyzer and Process Explorer on Windows.

18.5.6 WORDS - remember these

1. Thread — one flow through the code — a schedulable execution context with its own stack and register set.

2. Thread group — the threads of one process — tasks sharing a TGID, which is what `getpid()` returns on Linux.
3. Race condition — two threads touching one thing at once — an outcome that depends on unsynchronized interleaving of accesses.
4. Thread pool — reuse threads instead of making new ones — a fixed set of worker threads fed by a task queue.
5. 1:1 model — one kernel thread per program thread — the NPTL and Windows model, where the kernel schedules every thread directly.
6. M:N model — a runtime shuffles many onto few — user-space scheduling of green threads onto a smaller pool of kernel carriers.

18.6 Scheduling

18.6.1 PLAIN - in simple words

1. Your machine seems to run fifty programs at once. If it has eight cores, at most eight instructions are actually being executed at any instant.
2. The trick is speed. The system gives each program a tiny slice of time, then switches to the next, and goes round and round.
3. Switch fast enough and it looks continuous, in the same way that separate photographs shown quickly look like a moving picture.
4. The part of the operating system that decides who goes next is the **scheduler**.
5. Its job is to answer one question, over and over, thousands of times a second: of everything that could run right now, which one runs next, and for how long.
6. There is no perfect answer, because the goals fight each other.
7. You want the machine to be busy. You want short jobs to finish quickly. You want the mouse pointer to move the instant you move the mouse. You want no program to be forgotten forever.
8. A scheduler that is good at one of those is usually worse at another. Every real scheduler is a compromise that someone chose.

18.6.2 PLAIN - a picture in your head

1. Picture a doctor's surgery with one doctor and a waiting room.
2. **First come first served**: strictly in arrival order. Perfectly fair, and one person with a forty-minute problem makes everyone else late.
3. **Shortest job first**: call in whoever will be quickest. The waiting room empties fastest. Someone with a long problem may sit there all day.
4. **Round robin**: everyone gets five minutes, then goes to the back of the queue if they are not finished. Nobody waits forever. Everyone comes back several times, and each return wastes a minute settling in.
5. **Priority**: emergencies first. Right, and if emergencies keep arriving the ordinary patient never gets seen. That is **starvation**.
6. **Multilevel feedback**: several waiting rooms. Everyone starts in the fast one. If you use your whole slot without finishing you are moved to a slower room with longer slots. Short visits stay fast, long ones do not clog things.

Where this comparison breaks: the receptionist knows roughly how long each appointment will take. A scheduler does not, and cannot: predicting how long a program will run is equivalent to solving the halting problem. Every real scheduler that claims to run the shortest job first is in fact guessing from recent history. And the doctor takes a minute to switch patients; a processor switches in about a microsecond, which changes every calculation.

18.6.3 PLAIN - a worked example

1. Four jobs arrive at a single core. Times are in arbitrary equal units.

Job	Arrives	Needs
A	0	7
B	2	4
C	4	1
D	5	4

2. Total work is 16 units, so whatever we do, the last job finishes at time 16. Only the waiting changes.
3. **First come first served.** A runs 0 to 7, B 7 to 11, C 11 to 12, D 12 to 16.
4. Waiting times: A 0, B 5, C 7, D 7. Average wait 4.75.
5. **Shortest job first, without interrupting.** A runs 0 to 7 because it is the only one there. At 7 the choices are B (4), C (1), D (4). C is shortest, so C runs 7 to 8, then B 8 to 12, then D 12 to 16.
6. Waiting times: A 0, B 6, C 3, D 7. Average wait 4.00.
7. **Shortest remaining time first**, which is the same idea but allowed to interrupt. A starts. At time 2 B arrives needing 4, A has 5 left, so B takes over. At 4 C arrives needing 1, B has 2 left, so C takes over and finishes at 5. B then finishes 5 to 7. D runs 7 to 11. A finally finishes 11 to 16.
8. Waiting times: A 9, B 1, C 0, D 2. Average wait 3.00. Best of the four, and A paid for it.
9. **Round robin with a slice of 2 units.** New arrivals join the back of the queue before a job that has just been interrupted.

Slice	Runs	Queue afterwards
0 to 2	A	B, A
2 to 4	B	A, C, B
4 to 6	A	C, B, D, A
6 to 7	C done	B, D, A
7 to 9	B done	D, A
9 to 11	D	A, D
11 to 13	A	D, A
13 to 15	D done	A
15 to 16	A done	empty

10. Waiting times: A 9, B 3, C 2, D 6. Average wait 5.00.
11. Put the four side by side.

Algorithm	Avg wait	Avg turnaround
First come first served	4.75	8.75
Shortest job first	4.00	8.00
Shortest remaining first	3.00	7.00
Round robin, slice 2	5.00	9.00

12. Round robin came last on both averages. It is still what real interactive systems use, because averages are not what a human notices.
13. What a human notices is that under round robin, C started 2 units after it arrived and B started immediately. Under first come first served, C sat still for 7 units. Responsiveness is not in the average.

18.6.4 PLAIN - what is really happening inside

1. A **context switch** is the act of taking the processor away from one thread and giving it to another.
2. Here is exactly what gets saved, in order.
3. The general-purpose registers. On x86-64 that is 16 of them, 8 bytes each.
4. The program counter, so the thread resumes at the right instruction. On x86-64 this is already on the kernel stack, pushed by the interrupt.
5. The flags register, which holds the results of the last comparison.
6. The stack pointer.
7. The floating point and vector register state. On x86-64 with AVX-512 this is the big one: over 2,500 bytes. Modern kernels defer this and only save it if the new thread actually uses those registers.
8. Then, if the new thread belongs to a different process, the page table root register is reloaded, which changes what memory is visible.
9. The direct cost of all that is roughly one to three microseconds.
10. The indirect cost is larger and invisible: the new thread finds the caches full of the old thread's data, and runs slowly until it has refilled them. Measured cache-warming costs of 10 to 100 microseconds are common.
11. This is why a scheduler that switches too often makes a machine slower even though it looks fairer.
12. The **time slice** or **quantum** is how long a thread may run before the scheduler is asked again. Typical values are 1 to 10 milliseconds, which is a thousand to ten thousand times the switch cost.
13. **Preemptive** multitasking means the system can take the processor away whether or not the program agrees. This requires the timer interrupt.
14. **Cooperative** multitasking means a program keeps the processor until it chooses to give it up. One program with an infinite loop freezes everything.
15. Cooperative was not a theory. Classic Mac OS up to version 9 and Windows 3.x both worked that way, and both froze exactly as described.

18.6.5 TECHNICAL - the engineer's version

1. Scheduling metrics, defined precisely. Turnaround time is completion minus arrival. Waiting time is turnaround minus service time. Response time is first-run minus arrival. Throughput is completed jobs per unit time.
2. **Linux CFS**, the Completely Fair Scheduler, was written by Ingo Molnár, inspired by Con Kolivas's Rotating Staircase Deadline scheduler, and merged in Linux 2.6.23 in October 2007.
3. CFS abolished fixed time slices. It tracked `vruntime`, virtual runtime, per task, scaled by the task's weight, and always ran the task with the smallest `vruntime`. The ready set was a red-black tree, so picking the next task was O(log n).
4. Nice values map to weights by a table where each step of 1 changes CPU share by about 10 percent, and nice 0 has weight 1024. Nice ranges from -20 to +19.
5. **EEVDF** replaced CFS in Linux 6.6, released 29 October 2023. The name is Earliest Eligible Virtual Deadline First, from a 1995 paper by Ion Stoica and Hussein Abdel-Wahab.
6. EEVDF gives each task a virtual deadline computed from its requested slice and its lag, and runs the eligible task with the earliest deadline. It handles latency-sensitive tasks properly, which CFS could only do through a growing pile of heuristics and out-of-tree patches.
7. Linux scheduling classes, in strict priority order: `SCHED_DEADLINE`, then the real-time classes `SCHED_FIFO` and `SCHED_RR`, then `SCHED_OTHER` and `SCHED_BATCH`, then `SCHED_IDLE`. Verified on the machine used here with `chrt -m`:

```
SCHED_OTHER min/max priority  : 0/0
SCHED_FIFO  min/max priority  : 1/99
SCHED_RR    min/max priority  : 1/99
SCHED_BATCH min/max priority  : 0/0
SCHED_IDLE  min/max priority  : 0/0
SCHED_DEADLINE min/max        : 0/0
```

8. A `SCHED_FIFO` task at priority 99 that never blocks will hang the machine, or would if not for the throttle: `sched_rt_runtime_us` defaults to 950000 out of `sched_rt_period_us` of 1000000, so real-time tasks are capped at 95 percent of each second.
9. `SCHED_RR` round-robin slice on this machine, from `/proc/sys/kernel/sched_rr_timeslice_ms`, is 100 ms.
10. **Windows** uses 32 priority levels: 0 is the zero-page thread, 1 to 15 are dynamic, 16 to 31 are real-time. The level is computed from a process priority class and a thread priority offset.

Priority class	Base level
Idle	4
Below normal	6
Normal	8
Above normal	10
High	13
Real-time	24

11. Windows boosts a thread's priority when its wait completes, and boosts the foreground window's thread, then decays the boost by one level per quantum. Quantum on Windows client is 2 clock intervals, roughly 20 to 30 ms; on Server it is 12 intervals, roughly 120 to 180 ms, favouring throughput.
12. macOS uses a multilevel feedback queue with four bands: normal, system high priority, kernel mode only, and real-time. It also exposes Grand Central Dispatch quality-of-service classes, from `QOS_CLASS_USER_INTERACTIVE` down to `QOS_CLASS_BACKGROUND`, which map onto scheduler parameters and, on Apple silicon, onto the choice of performance or efficiency core.
13. Real context switch counts. On the machine used here, `/proc/stat` reported `ctxt 4153017` after 95 minutes of light use, roughly 730 switches per second across two cores.
14. Per-process counters live in `/proc/PID/status` as `voluntary_ctxt_switches` (the task blocked) and `nonvoluntary_ctxt_switches` (the scheduler preempted it). A high nonvoluntary count means CPU contention; a high voluntary count means I/O.
15. Observation and control: `chrt` to read and set policy, `nice` and `renice` for weights, `taskset` for CPU affinity, `perf sched latency` and `perf sched map` for switch-level traces, `pidstat -w` for switch rates, `vmstat 1` for the `cs` column, and `schedtool` for everything at once.
16. Where experts disagree: whether a general-purpose kernel should ever adopt a pluggable scheduler interface. Linux merged `sched_ext`, which allows BPF programs to implement schedulers, in Linux 6.12 in November 2024, after years of objection that it would fragment behaviour. Both sides had a real point.

18.6.6 WORDS - remember these

1. Scheduler — the chooser of who runs next — the kernel component implementing a policy over the set of runnable tasks.
2. Context switch — swapping one thread for another — saving and restoring architectural state, and possibly the address space.
3. Quantum — how long you get before being asked to stop — the scheduling time slice, typically 1 to 10 ms on desktop systems.
4. Preemption — being interrupted whether you like it or not — involuntary removal of a running task, driven by the timer interrupt.
5. Starvation — never getting a turn — indefinite postponement of a low-priority task by a stream of higher-priority ones.
6. Priority inversion — a low task blocking a high one — a high-priority task waiting on a lock held by a low-priority task; fixed by priority inheritance.

7. CFS — Linux's fair scheduler from 2007 — weight-proportional virtual runtime scheduling over a red-black tree.
8. EEVDF — its 2023 replacement — earliest eligible virtual deadline first, merged in Linux 6.6.

18.7 System calls

18.7.1 PLAIN - in simple words

1. Your program cannot open a file. It genuinely cannot. The instruction that would talk to the disk is refused in user mode.
2. So it asks the operating system to do it. That request is a **system call**.
3. A system call is not a function call into a library. It is a controlled jump across the boundary between your program and the kernel.
4. There are only a few hundred of them, and they are the complete list of everything your program can ask the machine to do.
5. Everything else your program does — arithmetic, loops, string handling — needs no permission and never crosses the boundary.
6. So the full power of a program is: any computation it likes, plus this fixed menu of requests.
7. When people say a language "can do system programming", they mean it can make these calls directly rather than through several layers.
8. Every file you open, every byte you send over the network, every window you draw and every key you read passes through this menu.

18.7.2 PLAIN - a picture in your head

1. Think of a bank counter with thick glass and a small hatch.
2. You are outside. The money is inside. You cannot reach it.
3. There is a printed list of things you may ask for: withdraw, deposit, balance, transfer. Numbered 1 to 40.
4. You fill in a slip: request number, then the details in fixed boxes. You push it through the hatch.
5. The clerk checks the slip, does the work inside where you cannot see, and pushes back a result: either what you asked for, or a refusal code.
6. You cannot ask for anything not on the list. You cannot go behind the counter. You cannot see how the vault works.
7. Filling in a slip and waiting takes longer than counting your own coins. So you learn to ask for a lot at once rather than making forty small requests.

Where this comparison breaks: the clerk is a separate person, whereas the kernel runs on the very same processor as your program, just in a different mode. And the bank clerk cannot look inside your wallet; the kernel can read and write every byte your program owns, and does, in order to fill in your answers. The boundary protects the kernel from you. It does not protect you from the kernel.

18.7.3 PLAIN - a worked example

1. Here is a complete C program:

```
#include <stdio.h>
int main(void) { printf("hello\n"); return 0; }
```

2. Run under `strace`, which prints every system call, it made exactly these, with the boring middle removed:

```
execve("./hello", ["./hello"], 0x7ffd... ) = 0
brk(NULL)                               = 0x5587a3d30000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY) = 3
```

```
fstat(3, {st_mode=S_IFREG|0644, st_size=53283}) = 0
mmap(NULL, 53283, PROT_READ, MAP_PRIVATE, 3, 0) = ...
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6",
       O_RDONLY|O_CLOEXEC)              = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0..."..., 832) = 832
mmap(NULL, 2170256, PROT_READ, MAP_PRIVATE, 3, 0) = ...
mmap(..., PROT_READ|PROT_EXEC, MAP_FIXED, 3, 0x28000)
close(3)                                = 0
arch_prctl(ARCH_SET_FS, 0x7f22105ce740) = 0
set_tid_address(0x7f22105cea10)         = 18134
mprotect(0x7f22103ff000, 16384, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024})
fstat(1, {st_mode=S_IFIFO|0600, st_size=0}) = 0
getrandom("\x8a\x08\xd7\x55...", 8, GRND_NONBLOCK) = 8
brk(0x5587a3d51000)                     = 0x5587a3d51000
write(1, "hello\n", 6)                  = 6
exit_group(0)                           = ?
```

3. Now every line, in order.
4. `execve` is the call that loaded this program. It is the first line of every trace because the trace starts in the shell, before the program exists.
5. `brk(NULL)` asks where the heap currently ends. It is a query, not a change.
6. `access("/etc/ld.so.preload")` returned `-1 ENOENT`, meaning the file does not exist. That is not an error; the dynamic linker is checking for an optional override and finding none.
7. `openat` on `/etc/ld.so.cache` returned 3. That number is a **file descriptor**: the handle you use for every later operation on that file. 0, 1 and 2 are already taken by standard input, output and error.
8. `fstat(3, ...)` asks how big it is: 53,283 bytes.
9. `mmap` maps those bytes into memory rather than reading them, so the library cache can be looked at as an array.
10. `close(3)` releases the descriptor. Descriptor 3 is now free again, which is why the very next `openat` also returns 3.
11. The second `openat` finds the C library itself, all 2,125,328 bytes of it.
12. `read(3, "\177ELF...", 832)` reads the first 832 bytes: the ELF header. The first four bytes are `0x7F` then `ELF`, the magic number that identifies the file format.
13. Then four `mmap` calls place the library into memory with different permissions: read-only for constants, read plus execute for code, read plus write for data.
14. `arch_prctl(ARCH_SET_FS, ...)` sets the FS segment base, which is how thread-local storage is found on x86-64.
15. `set_tid_address` tells the kernel where to clear a word when this thread dies, which is how `pthread_join` learns the news.
16. `mprotect(..., PROT_READ)` makes the relocation table read-only after it has been filled in. That is a hardening measure called RELRO.
17. `prlimit64` reads the stack limit: 8,192 times 1,024 bytes, that is 8 MiB.
18. `fstat(1, ...)` checks what standard output is. Here it reported `S_IFIFO`, a pipe. That is why the C library chose full buffering rather than line buffering.
19. `getrandom(..., 8, ...)` fetches 8 random bytes for the stack canary, the guard value that detects buffer overflows.
20. `brk` grows the heap by 132 KiB for the output buffer.
21. `write(1, "hello\n", 6)` is the one call the program actually meant to make. Six bytes to descriptor 1. It returned 6, meaning all six were accepted.
22. `exit_group(0)` ends every thread of the process with status 0.
23. Count them: about twenty-five system calls to print one word, and only one of them was the point. The

rest is the cost of dynamic linking.

18.7.4 PLAIN - what is really happening inside

1. Mechanically, on a 64-bit Intel or AMD processor, a system call is this.
2. The program puts a number in the register `rax`. That number says which service it wants. `write` is 1. `read` is 0. `openat` is 257.
3. It puts the arguments in registers, in a fixed order: `rdi`, `rsi`, `rdx`, `r10`, `r8`, `r9`. Up to six.
4. It executes one instruction: `syscall`.
5. The processor then does several things at once, in hardware. It saves the return address into `rcx` and the flags into `r11`. It switches to privilege ring 0. It loads the instruction pointer from a control register that only the kernel could write.
6. That control register, `MSR_LSTAR`, was set during boot to the address of the kernel's entry stub. So the program lands exactly where the kernel decided, and nowhere else.
7. The kernel stub switches to a kernel stack, saves the rest of the registers, checks that the number in `rax` is within range, and calls the function in the system call table at that index.
8. The function does the work. It may block, in which case the scheduler picks somebody else and this thread resumes here later.
9. The result goes into `rax`. Errors are returned as small negative numbers.
10. The kernel restores registers and executes `sysret`, which puts the processor back into ring 3 at the address in `rcx`.
11. The C library then looks at `rax`. If it is between -1 and -4095, the library stores its negation in `errno` and returns -1 to you.
12. That is why C code checks for -1 and then reads `errno`. The kernel never set `errno`; the library did.
13. Cost, measured on the machine used here: a bare `getpid` system call took 0.089 microseconds, that is 89 nanoseconds, about 187 cycles at 2.10 GHz.
14. That is genuinely expensive next to an ordinary function call of one or two nanoseconds, and cheap next to a disk read of 100 microseconds.
15. Some calls are so common that the kernel arranges for them to need no boundary crossing at all. It maps a small shared page called the **vDSO** into every process, containing code and data that answer them in user mode.
16. Measured on this machine: `clock_gettime` through the vDSO took 0.030 microseconds; forced through the real system call it took 0.144 microseconds. Nearly five times cheaper for doing the same thing.

18.7.5 TECHNICAL - the engineer's version

1. The x86-64 Linux calling convention for system calls: number in `rax`, args in `rdi`, `rsi`, `rdx`, `r10`, `r8`, `r9`, return in `rax`. Note `r10` rather than `rcx`, because `syscall` overwrites `rcx` with the return address.
2. On AArch64 Linux: number in `x8`, args in `x0` to `x5`, instruction `svc #0`, return in `x0`. On RISC-V: number in `a7`, args `a0` to `a5`, instruction `ecall`.
3. Older x86 mechanisms, in historical order: `int 0x80` software interrupt (slow, about 1,000 cycles), `sysenter` and `sysexit` from Pentium II, `syscall` and `sysret` from AMD64. Windows used `int 0x2E` before switching to `sysenter`.
4. Linux has 300 to 460 system calls depending on architecture and version. Verified numbers from `asm/unistd_64.h` on this machine:

Call	Number	Purpose
read	0	Read from a descriptor
write	1	Write to a descriptor
close	3	Release a descriptor
mmap	9	Map memory or a file
clone	56	Create process or thread

Call	Number	Purpose
execve	59	Replace process image
exit_group	231	End all threads
openat	257	Open relative to a dir fd

5. Grouped by purpose, the ones worth memorizing:

Group	Calls
Process	fork, clone, execve, wait4
Process	exit_group, kill, getpid
Files	openat, read, write, close
Files	lseek, statx, unlink, rename
Memory	mmap, munmap, mprotect, brk
Network	socket, bind, listen, accept
Network	connect, sendto, recvfrom
Sync	futex, poll, epoll_wait
Devices	ioctl, fcntl

6. `ioctl` is the escape hatch: a single call with a device-specific command number, used for everything that does not fit the read and write model. It is widely regarded as an ugly but necessary design.
7. Counting calls with `strace -c` on `ls /tmp` on this machine gave 76 calls total, 5 of which returned errors. The largest single group was 17 `mmap` calls, all from dynamic linking.
8. Windows equivalents are in `ntdll.dll` as `Nt` or `Zw` functions: `NtCreateFile`, `NtReadFile`, `NtWriteFile`, `NtAllocateVirtualMemory`. The documented API you are meant to call is Win32, in `kernel32.dll`, which calls these. Microsoft does not guarantee the numbers, and they change between builds, which is why malware that hardcodes them breaks after updates.
9. macOS syscall numbers are split into classes by the top byte: class 1 is Mach traps, class 2 is BSD calls. `write` is `0x2000004`.
10. Reducing the cost: `io_uring`, merged in Linux 5.1 in May 2019, replaces per operation calls with two shared ring buffers, allowing batched submission and completion with, in the best case, zero system calls per operation.
11. The security cost: Meltdown and Spectre, disclosed 3 January 2018, forced kernel page-table isolation, which roughly doubled or tripled system call cost on affected processors. Measured overheads of 5 to 30 percent on system call heavy workloads were common in 2018.
12. Tools: `strace` and `ltrace` on Linux, `dtruss` and `dtrace` on macOS (which require disabling System Integrity Protection for many targets), Process Monitor and `Event Tracing for Windows` on Windows, and `bpftrace` on modern Linux for anything the others cannot do.

18.7.6 WORDS - remember these

1. System call — asking the kernel for something — a controlled mode transition into a numbered kernel service routine.
2. File descriptor — a small number naming an open thing — an index into the per-process open file descriptor table.
3. errno — the reason the last request failed — a thread-local integer set by the C library from the kernel's negative return value.
4. vDSO — a shortcut that skips the kernel — a kernel-provided shared object mapped into every process, serving calls like `clock_gettime` in user mode.
5. ioctl — the request that covers everything else — a device-specific control call carrying an opaque command number and argument.

6. io_uring — batching requests instead of asking one at a time — shared submission and completion ring buffers, in Linux since 5.1.

18.8 Memory management from the operating system's side

18.8.1 PLAIN - in simple words

1. Chapter 11 covered memory in depth. One line to bring it back: every program sees its own private set of addresses starting at zero, and the hardware translates those into real addresses in the physical chips.
2. That translation is done by a unit in the processor using tables the kernel builds. The kernel owns those tables. No program can edit its own.
3. Here we care about the part the kernel does, which is three jobs.
4. Job one: hand out **page frames**. Physical memory is cut into fixed blocks, usually 4 KB. The kernel keeps track of which are used and which are free.
5. Job two: decide what to do when there is not enough. Either free something, or write something to disk, or kill a program.
6. Job three: enforce limits, so one program cannot take everything.
7. A surprising fact: the kernel routinely promises more memory than it has.
8. It does that because programs ask for far more than they use, and most of the promise is never called in.
9. When the promise is called in and the memory does not exist, something has to die. The kernel picks a victim and kills it.

18.8.2 PLAIN - a picture in your head

1. Think of an airline selling seats on a plane with 180 seats.
2. Experience says about 5 percent of people never turn up. So the airline sells 189 tickets. That is **overcommit**.
3. Almost always it works and the plane goes out full. Occasionally everyone turns up, and someone gets removed from the flight.
4. The airline does not remove people at random. It has a rule: latest booking, cheapest fare, travelling alone.
5. The kernel does the same. It has a scoring rule and it kills the highest scorer. That code is called the **out of memory killer**.
6. Memory the program asked for but never touched is a ticket that was never used. The kernel only allocates a real page frame the first time you actually write to the address.

Where this comparison breaks: the airline can offer you money to take a later flight. The kernel cannot negotiate. It sends signal 9 and the process ends instantly with no chance to save anything. And an airline's overbooking is calculated from real statistics; Linux's default overcommit setting is a rough heuristic that guesses, and it can be switched off entirely.

18.8.3 PLAIN - a worked example

1. A program calls `malloc(1024 * 1024 * 1024)`, asking for one gigabyte, on a machine with 8 GB of RAM and 6.7 GB free.
2. The C library sees a large request and calls `mmap` rather than `brk`.
3. The kernel checks its overcommit policy. The default on this machine, from `/proc/sys/vm/overcommit_memory`, is 0: the heuristic mode. It agrees.
4. The kernel records a mapping one gigabyte long in the process's list of memory regions and returns a pointer. **Not one byte of physical memory has been allocated.**
5. `free` reports no change. The program's virtual size, `VSZ` in `ps`, jumps by 1 GB. Its resident size, `RSS`, does not move.

6. The program writes one byte at the start. The processor tries to translate that address, finds no page table entry, and raises a **page fault**.
7. The kernel handles the fault, takes one free 4 KB frame, zeroes it, points the table entry at it, and restarts the instruction. `RSS` goes up by 4 KB.
8. Only when the program walks the whole gigabyte does the kernel hand over 262,144 frames and the memory is really consumed.
9. Now suppose it does that when only 200 MB is free. The kernel first tries to reclaim: it drops cached file pages, and swaps out unused anonymous pages.
10. If that is not enough, the out-of-memory killer runs. It reads a score for every process and kills the highest.
11. On this machine the score for a small shell process was 666 out of 1000, from `/proc/self/oom_score`, and the tunable `/proc/self/oom_score_adj` was 0. Setting the adjustment to -1000 makes a process immune.
12. In the kernel log you would then see a line beginning `Out of memory: Killed process 1234 (chrome)`, followed by a full report of who had what.

18.8.4 PLAIN - what is really happening inside

1. The kernel divides physical memory into page frames and keeps one small record per frame. On Linux that is `struct page`, about 64 bytes each.
2. For 8 GB of RAM with 4 KB pages, that is 2,097,152 frames and about 128 MB of records. The kernel's own bookkeeping costs about 1.6 percent of memory.
3. Free frames are managed by the **buddy allocator**: free memory is held in lists of blocks of 1, 2, 4, 8 up to 1024 pages. Splitting and merging is done by halving and pairing, which keeps fragmentation manageable.
4. Small kernel objects do not each take a page. A second allocator, the slab allocator, carves pages into same-sized objects with free lists. You can see it in `/proc/slabinfo`.
5. Memory that holds file contents is not wasted. The **page cache** keeps recently read file data in otherwise free memory and hands it back instantly when reclaimed. That is why "free memory" on a healthy Linux box is near zero and this is correct, not a problem.
6. When memory runs short the kernel walks its lists of pages and decides what to evict, approximately least-recently-used. Clean file pages are cheapest to drop, because the copy on disk is still valid. Dirty pages must be written first. Anonymous pages, which have no file, must go to swap.
7. If there is no swap and nothing left to reclaim, the only remaining action is to kill something.
8. The out-of-memory score is roughly proportional to the process's resident memory plus its swap use, expressed out of 1000, then adjusted by `oom_score_adj`. The biggest user usually dies. That is deliberate: killing the biggest recovers the most memory for the fewest deaths.

18.8.5 TECHNICAL - the engineer's version

1. Linux overcommit modes, in `/proc/sys/vm/overcommit_memory`:

Value	Name	Behaviour
0	Heuristic	Refuse only absurd requests
1	Always	Never refuse anything
2	Strict	Refuse beyond the limit

2. In mode 2, the limit is swap plus `overcommit_ratio` percent of RAM. The default ratio is 50, verified on this machine. With 8 GB and no swap, that allows only 4 GB of committed address space in total, which is why mode 2 is rarely used without tuning.

3. Windows does not overcommit in the Linux sense. It maintains a commit charge against a commit limit equal to RAM plus the pagefile, and `VirtualAlloc` with `MEM_COMMIT` fails when the limit is reached. The failure is returned to the program rather than delivered later as a kill.
4. That is a genuine design difference with consequences. Linux gives you better memory utilization and a risk of sudden death; Windows gives you honest allocation failures and requires a large pagefile.
5. Page sizes: 4 KiB base on x86-64 and on Linux ARM64 by default; 2 MiB and 1 GiB huge pages on x86-64; 16 KiB base on Apple silicon macOS and on iOS. Transparent Huge Pages on Linux promotes eligible 2 MiB regions automatically and can be controlled per process with `madvise`.
6. Per-process limits come from `setrlimit` and are visible in `/proc/PID/limits`. `RLIMIT_AS` caps total address space, `RLIMIT_DATA` the heap, `RLIMIT_STACK` the main stack (8 MiB soft on this machine), `RLIMIT_MEMLOCK` how much may be pinned.
7. Cgroups give stronger limits than rlimits, because they apply to a group and are enforced by the reclaim path. In cgroup v2, `memory.max` is a hard cap that triggers cgroup-local OOM kill, `memory.high` is a throttling threshold, and `memory.low` protects a floor during reclaim.
8. Useful counters, all real files: `/proc/meminfo` for the global picture, `/proc/PID/status` for `VmRSS`, `VmSize` and `VmSwap`, `/proc/PID/smaps` for per-mapping detail, and `/proc/pressure/memory` for PSI stall percentages, which is the single best early warning of memory trouble.
9. Reading `/proc/meminfo` on this machine gave `MemTotal: 8216168 kB`, `MemFree: 6709900 kB`, `MemAvailable: 7407208 kB` and `Cached: 847024 kB`. `MemAvailable` is the number to trust; `MemFree` ignores reclaimable cache.
10. Tools: `free -m`, `vmstat 1`, `smem`, `ps -eo pid,rss,vsz --sort=-rss`, `pmap`, `slabtop`, and `dmesg | grep -i "killed process"` to find OOM events after the fact. On macOS, `vm_stat`, `footprint` and Activity Monitor's memory pressure graph.

18.8.6 WORDS - remember these

1. Page frame — one fixed block of real memory — a physical page, usually 4 KiB on x86-64.
2. Page fault — the translation was missing — a trap raised when a virtual address has no valid mapping, handled by the kernel.
3. Overcommit — promising more than you have — allowing total mapped address space to exceed RAM plus swap.
4. OOM killer — the code that picks who dies — the out-of-memory handler that selects and SIGKILLs the highest scoring process.
5. Page cache — free memory holding file data — the unified cache of file-backed pages, reclaimable without I/O when clean.
6. Buddy allocator — splitting and pairing free blocks — the power-of-two physical page allocator underneath the kernel's memory subsystem.

18.9 The filesystem from the operating system's side

18.9.1 PLAIN - in simple words

1. A disk stores numbered blocks. It knows nothing about files, names or folders. Those are entirely the operating system's invention.
2. The filesystem is the code that turns "a file called notes.txt in a folder called work" into "blocks 41,922 to 41,930 on this disk".
3. There are dozens of filesystems: ext4, NTFS, APFS, XFS, Btrfs, FAT32, ZFS. They all store data differently.
4. Your programs do not know which one they are using, and should not care.
5. That works because the kernel has a layer in the middle that presents one uniform set of operations: open, read, write, close, list.
6. Every filesystem implements that same set. The layer picks the right one based on where the file lives.

7. **Mounting** is how a filesystem gets attached to the tree of names. You say "this disk appears at /media/photos" and from then on it does.
8. When you open a file, you get back a small number. Every later operation uses that number. The kernel remembers everything else.
9. Permissions decide who may do what. On UNIX they are simple and were designed in 1971. On Windows they are elaborate lists.

18.9.2 PLAIN - a picture in your head

1. Think of a large library.
2. The shelves hold numbered boxes. Box 41,922 contains some pages. Boxes know nothing about titles. That is the disk.
3. The card catalogue maps a title to a list of box numbers. That is the filesystem's index, called the inode table on UNIX.
4. The front desk is where you ask for a book. You give a title; they hand you a ticket with a number on it, say ticket 3. That is your file descriptor.
5. From then on you say "more from ticket 3" and they know exactly which book and which page you were on. You never say the title again.
6. Two people can hold two tickets for the same book, each on a different page. Two descriptors, one file, two positions.
7. Adding a new wing to the library and declaring that its shelves count as part of aisle F is mounting.

Where this comparison breaks: a library has one physical copy of a book, so two readers cannot both have it. A filesystem hands out as many descriptors as are asked for, and if two writers write at once without coordinating, the result is a mixture of both. The filesystem does not lock anything for you unless you ask.

18.9.3 PLAIN - a worked example

1. Real permission bits, produced on the machine used here by creating files and running `ls -l`:

```
-rw-r--r-- 1 root root    0 a.txt      644
-rwxr-x--- 1 root root    0 b.sh       750
drwxr-sr-x 2 root root 4096 d          2755
-rwsr-xr-x 1 root root    0 s          4755
-rwxrwxrwt 1 root root    0 t          1777
```

2. Read the first column in four pieces. First character: file type. `-` is an ordinary file, `d` a directory, `l` a symbolic link, `c` a character device, `b` a block device, `s` a socket, `p` a pipe.
3. Then three groups of three: what the **owner** may do, what the **group** may do, what **everyone else** may do.
4. In each group: `r` read, `w` write, `x` execute. A dash means not allowed.
5. So `-rw-r--r--` is: owner may read and write; group may read; others may read. Nobody may execute.
6. The numeric form packs each group into one octal digit, with read worth 4, write worth 2, execute worth 1. Owner 4+2=6, group 4, others 4, giving 644.
7. `b.sh` at 750 is owner read, write and execute (7), group read and execute (5), others nothing (0).
8. Now the odd ones. `s` shows `-rwsr-xr-x`, mode 4755. The `s` where the owner's `x` should be is the **setuid** bit. It means: when this program runs, it runs as the file's owner, not as you.
9. That is how `/usr/bin/passwd` works. Verified on this machine, it really is `-rwsr-xr-x 1 root root 64152`. You need to change a root-owned file, so the program briefly becomes root.
10. Setuid is the single most dangerous bit in UNIX. A bug in a setuid-root program is a way for any user to become root. Modern systems replace it with capabilities wherever they can.
11. `d` shows `drwxr-sr-x`, mode 2755. The `s` in the group position on a directory is **setgid**: new files created inside inherit the directory's group. That is how shared project folders work.

12. `t` shows `-rwxrwxrwt`, mode 1777. The final `t` is the **sticky bit**. On `/tmp` it means: anyone may create files here, but you may only delete your own. Without it, any user could delete anyone's temporary files.
13. On a directory the bits mean something different from a file: `r` lists the names, `w` creates and deletes entries, `x` allows you to traverse into it.
14. A directory with `x` but not `r` is real and useful: you can open a known path inside it but cannot see what is there.

18.9.4 PLAIN – what is really happening inside

1. The **virtual filesystem** layer, VFS, is a set of function pointers. Each filesystem registers its own implementations of open, read, write, lookup and about forty others.
2. When your program calls `read`, the kernel finds the file's inode, follows its pointer to the operations table, and calls whichever function is there. ext4's read or NFS's read or procfs's read.
3. That indirection is why `/proc/self/status` can be read like a file even though there is no such file on any disk. Reading it runs a kernel function that generates the text on the spot.
4. There are three tables involved in open files, and confusing them is the most common source of misunderstanding.
5. Table one: the **file descriptor table**, one per process. It maps small integers to entries in table two.
6. Table two: the **open file table**, system-wide. Each entry holds the current position, the access mode, and a pointer into table three.
7. Table three: the **inode table**. One entry per actual file, holding size, owner, permissions, timestamps and the block map.
8. Now the behaviours make sense. `dup` copies a descriptor entry, so both point at the same open file entry, so both share one position. `open` on the same file twice makes two open file entries, so two independent positions.
9. And after `fork`, parent and child have separate descriptor tables that point at the same open file entries, so their positions move together.
10. Real descriptors, from `/proc/self/fd` on this machine:

```
0 -> /dev/null
1 -> /tmp/.../output
2 -> /tmp/.../output
3 -> /proc/6536/fd
```

11. Note that 1 and 2 point at the same file: standard output and standard error were both redirected to one place.
12. `open` does this: parse the path one component at a time, checking traverse permission on each directory; find the final inode; check the requested access against the permission bits; allocate an open file entry; find the lowest free descriptor number; return it. The rule that it is always the lowest free number is a standard, and shell redirection depends on it.
13. `read` does this: look up the descriptor, take the position, ask the filesystem for those bytes, which usually come from the page cache, copy them into the program's buffer, advance the position, return the count.
14. `write` usually does not touch the disk at all. It copies into the page cache and marks it dirty. A kernel thread writes it out later. `fsync` is what forces it out now.
15. `close` releases the descriptor and decrements a reference count on the open file entry. Only when that count reaches zero is the entry freed.

18.9.5 TECHNICAL - the engineer's version

1. Linux VFS core objects: `struct super_block` for a mounted filesystem, `struct inode` for a file, `struct dentry` for a name-to-inode cache entry, and `struct file` for an open instance.
2. The dentry cache is the reason repeated path lookups are fast. Without it, opening `/usr/share/doc/x/y` would require reading four directories from disk every time.
3. Mounting attaches a superblock at a mountpoint. On this machine `/proc/mounts` showed real entries:

```
proc /proc proc rw,relatime 0 0
sysfs /sys sysfs rw,relatime 0 0
devtmpfs /dev devtmpfs rw,size=4102960k,mode=755 0 0
tmpfs /dev/shm tmpfs rw,size=8216168k 0 0
devpts /dev/pts devpts rw,mode=600 0 0
```

4. Note that `proc`, `sysfs`, `devtmpfs` and `tmpfs` have no disk at all. They are kernel interfaces wearing a filesystem's clothes. That is the VFS abstraction paying off.
5. UNIX permission checking order is strict: if you are the owner, only the owner bits apply, even if the group bits are more permissive. Then group, then other. First match wins, and it is not a union.
6. `umask` masks off bits from newly created files. The default 022 verified here turns a requested 666 into 644 and a requested 777 into 755.
7. POSIX access control lists, from the withdrawn POSIX 1003.1e draft 17 but implemented anyway, add named users and groups: `setfacl -m u:alice:rw file`, read back with `getfacl`. A file with an ACL shows a + after its mode in `ls -l`.
8. Windows uses a completely different model. Every securable object has a security descriptor containing an owner SID, a group SID, a DACL (discretionary access control list) and a SACL (system ACL, for auditing).
9. A DACL is an ordered list of access control entries, each granting or denying specific rights to a SID. There are far more than three rights: `FILE_READ_DATA`, `FILE_WRITE_ATTRIBUTES`, `DELETE`, `WRITE_DAC`, `WRITE_OWNER`, `SYNCHRONIZE` and more.
10. Evaluation order matters: deny entries are processed before allow entries, and inherited entries after explicit ones. Tools: `icacls` on the command line, the Security tab in Explorer.
11. Filesystem comparison, all figures from the filesystems' own documentation:

Filesystem	Max file size	Journal
ext4	16 TiB	Yes
XFS	8 EiB	Yes
NTFS	8 PiB (Win 10+)	Yes
APFS	8 EiB	Copy on write
Btrfs	16 EiB	Copy on write
FAT32	4 GiB minus 1	No

12. FAT32's 4 GiB limit is the reason a single large video file cannot be copied to many USB sticks as sold. It is a 32-bit size field, not a bug.
13. Tools: `lsof` and `fuser` to see who has a file open, `df` and `du` for space, `stat` for one file's metadata, `mount` and `findmnt` for the mount tree, `tune2fs -l` for ext4 superblock details, `debugfs` for raw inspection, `fsck` for repair.

18.9.6 WORDS - remember these

1. Filesystem — the scheme that turns names into blocks — the on-disk format plus the driver implementing it.
2. VFS — one interface over many filesystems — the virtual filesystem switch, dispatching operations through per-filesystem function tables.

3. Inode — the record describing one file — metadata and block map, identified by number rather than by name.
4. File descriptor — the ticket you get from open — a per-process index into the descriptor table, always the lowest free number.
5. Mount — attaching a filesystem into the name tree — binding a superblock to a directory so paths below it resolve there.
6. setuid — the program runs as its owner, not you — mode bit 4000, causing the effective UID to be set from the file's owner on exec.
7. Sticky bit — you may only delete your own files here — mode bit 1000 on a directory, restricting deletion to owners.
8. Page cache write-back — write returns before the disk is touched — buffered writes marked dirty and flushed later, forced by `fsync`.

18.10 Inter-process communication

18.10.1 PLAIN - in simple words

1. Processes are deliberately kept apart. That is the whole point of the design.
2. But they often need to work together, so the operating system provides controlled ways to pass information across the wall.
3. There are six that matter, and they differ mainly in three questions: how fast, how much can you send, and can the two sides be on different machines.
4. **Pipes**: a one-way stream of bytes from one process to another. Made by the shell every time you use the | symbol.
5. **Named pipes**: the same thing, but with a name in the filesystem, so unrelated processes can find it.
6. **Signals**: a single number sent to a process to interrupt it. Almost no information, but it arrives even if the target is stuck.
7. **Message queues**: a post box holding whole messages with boundaries preserved, so a 20-byte message arrives as 20 bytes and not as part of a stream.
8. **Shared memory**: two processes agree to see the same physical memory. The fastest possible, because nothing is copied.
9. **Sockets**: a two-way connection, which works between processes on one machine and between machines across a network, with the same code.
10. **Remote procedure call**: a layer on top that makes a call to another machine look like an ordinary function call in your program.

18.10.2 PLAIN - a picture in your head

1. Two offices in the same building need to exchange work.
2. A pipe is a chute in the wall. Papers go in one end and come out the other, in order, one way only. Simple and fast.
3. A named pipe is the same chute with a label on it, so anyone in the building can find it rather than only the two who installed it.
4. A signal is a fire alarm. It carries one bit of meaning, you cannot send data with it, and it interrupts whatever the other office was doing.
5. A message queue is a set of pigeonholes. Each item stays a separate item. Nothing merges.
6. Shared memory is knocking a hole in the wall and putting one desk half in each room. Both sides see the same paper immediately. Nobody carries anything. And both sides can grab the same sheet at once, so you need a rule about whose turn it is.
7. A socket is a telephone line, which happens to work identically whether the other office is next door or in another country.

Where this comparison breaks: a chute holds an unlimited pile of paper. A real pipe holds a fixed amount, 64 KiB by default on Linux, and when it is full the writer simply stops until the reader catches up. That blocking behaviour is not a flaw, it is the flow control that makes pipelines work on files larger than memory.

18.10.3 PLAIN - a worked example

1. This shell command:

```
sh -c 'echo hello world | tr a-z A-Z'
```

2. Traced with `strace -f`, the kernel calls were exactly these, with process IDs at the left:

```
18231 pipe2([3, 4], 0)                  = 0
18231 clone(...SIGCHLD...)              = 18232
18231 close(4)                          = 0
18232 close(3)                          = 0
18232 dup2(4, 1)                        = 1
18232 close(4)                          = 0
18232 write(1, "hello world\n", 12)     = 12
18232 +++ exited with 0 +++
18231 clone(...SIGCHLD...)              = 18233
18231 close(3)                          = 0
18233 dup2(3, 0)                        = 0
18233 close(3)                          = 0
18233 execve("/usr/bin/tr", ["tr","a-z","A-Z"]) = 0
18233 read(0, "hello world\n", 8192)    = 12
18233 read(0, "", 8192)                 = 0
18233 write(1, "HELLO WORLD\n", 12)     = 12
18231 wait4(-1, [exited 0], 0, NULL)    = 18232
18231 wait4(-1, [exited 0], 0, NULL)    = 18233
```

3. Line by line. `pipe2` creates the pipe and returns two descriptors at once: 3 is the read end, 4 is the write end. One object, two handles.
4. `clone` makes the first child, PID 18232, which will run `echo`. It inherits both descriptors.
5. The parent closes 4, its copy of the write end, because it will not write.
6. The child closes 3, its copy of the read end, because it will not read.
7. `dup2(4, 1)` is the key move. It makes descriptor 1, standard output, point at the same open file entry as descriptor 4. Now anything written to standard output goes into the pipe.
8. The child then closes 4, because 1 already refers to the pipe. Two handles to the same thing are no longer needed.
9. `write(1, "hello world\n", 12)` puts 12 bytes into the pipe buffer.
10. The second child, 18233, does the mirror image: `dup2(3, 0)` makes standard input read from the pipe, then `execve` replaces it with `/usr/bin/tr`.
11. Notice that `tr` does none of this. By the time `tr` starts, the plumbing is already done and it just reads standard input like any program. That is why every UNIX tool composes.
12. `read(0, "hello world\n", 8192)` gets the 12 bytes. The next `read` returns 0.
13. That zero is end-of-file, and it happens only because every copy of the write end has been closed. If any process had kept one open, `tr` would have waited forever. Forgetting to close the write end is the classic pipe bug.
14. Finally `write(1, "HELLO WORLD\n", 12)` sends the result to the terminal, and the shell reaps both children with `wait4`.

18.10.4 PLAIN - what is really happening inside

1. A pipe is a fixed-size circular buffer in kernel memory with two ends and no name. 64 KiB on Linux by default, adjustable with `fcntl` and `F_SETPIPE_SZ`.

2. Writing to a full pipe blocks. Reading an empty pipe with writers still open blocks. Reading an empty pipe with no writers returns 0. Writing to a pipe with no readers delivers `SIGPIPE`, which kills the writer by default.
3. That last rule is why `head` on a huge file stops quickly rather than reading everything: `head` exits, the pipe loses its reader, the producer is killed.
4. A signal is a bit set in the target's task structure. Nothing is delivered immediately. The next time that process is about to return to user mode, the kernel notices the bit and diverts it to the handler.
5. That is why a process stuck in uninterruptible kernel sleep does not respond to signals: it is never about to return to user mode.
6. Signal 9, `SIGKILL`, and signal 19, `SIGSTOP`, cannot be caught or ignored, because they are handled by the kernel itself rather than delivered.
7. Shared memory is the only mechanism with no copying. The kernel maps the same physical page frames into two processes' page tables. After setup, the kernel is not involved at all.
8. That is also its danger. With no kernel in the loop there is no ordering, so the two sides must agree on synchronization themselves, usually with a semaphore or an atomic flag plus memory barriers.
9. Sockets carry a protocol stack even when both ends are on one machine. A UNIX domain socket skips the network stack and is roughly twice as fast as a loopback TCP socket for the same data.

18.10.5 TECHNICAL - the engineer's version

Mechanism	Copies	Boundaries	Across machines
Pipe	2	Stream	No
Named pipe / FIFO	2	Stream	No
Signal	0	1 number	No
POSIX mq	2	Messages	No
Shared memory	0	You decide	No
UNIX socket	2	Both modes	No
TCP socket	2+	Stream	Yes

1. Two copies means user to kernel and kernel to user. `splice` and `vmsplice` on Linux can reduce pipe transfers to one copy or zero by moving page references instead of bytes.
2. Signals: 31 standard signals plus 32 real-time signals on Linux. Real-time signals queue and carry a small value; standard signals do not queue, so two `SIGUSR1` sent quickly may be delivered once.
3. A demonstration on this machine sent `SIGUSR1` to itself three times in a tight loop and the handler ran three times, because each was delivered before the next was raised. Send them faster than delivery and you would lose some.
4. The first six signals as printed by `kill -l` here: 1 SIGHUP, 2 SIGINT, 3 SIGQUIT, 4 SIGILL, 5 SIGTRAP, 6 SIGABRT. Also worth knowing: 9 SIGKILL, 11 SIGSEGV, 13 SIGPIPE, 15 SIGTERM, 17 SIGCHLD, 19 SIGSTOP.
5. `SIGTERM` asks politely and can be caught for cleanup. `SIGKILL` cannot. A shutdown script that sends only `SIGKILL` loses data; systemd sends `SIGTERM`, waits `DefaultTimeoutStopSec` (90 s by default), then `SIGKILL`.
6. System V IPC, from AT&T System V in 1983: `msgget`, `semget`, `shmget`, with keys instead of file descriptors and no automatic cleanup. Limits on this machine from `ipcs -l`: 32,000 message queues maximum, 8,192-byte maximum message size, 4,096 shared memory segments.
7. POSIX IPC, IEEE 1003.1b-1993, is the modern replacement: `mq_open`, `sem_open`, `shm_open`, all using file-descriptor semantics and names under `/dev/shm` and `/dev/mqueue`.
8. Modern Linux additions: `eventfd`, `signalfd` and `timerfd` turn events, signals and timers into readable descriptors, so they can be waited on by `epoll` alongside sockets. `memfd_create` gives an anonymous file for sharing.

9. Higher-level buses: D-Bus on Linux desktops, Mach ports on macOS, Binder on Android, ALPC on Windows. All are message passing with identity and permissions on top.
10. Remote procedure call history: Birrell and Nelson's 1984 paper *Implementing Remote Procedure Calls* set the model. Sun RPC, RFC 1057 in 1988, powered NFS. Modern systems use gRPC over HTTP/2 with Protocol Buffers, or JSON over HTTP.
11. The honest version, from Waldo, Wyant, Wollrath and Kendall's 1994 Sun paper *A Note on Distributed Computing*: a remote call is not a local call. It can be slow, it can fail halfway, and it can succeed while the reply is lost. Any RPC layer that hides this will eventually mislead you.
12. Tools: `lsof -p PID` to see pipes and sockets, `ss -x` for UNIX sockets, `ipcs` and `ipcrm` for System V objects, `strace -e trace=ipc`, and `dbus-monitor` for D-Bus traffic.

18.10.6 WORDS - remember these

1. Pipe — a one-way chute between processes — an anonymous kernel ring buffer with a read end and a write end, 64 KiB by default on Linux.
2. FIFO — a pipe with a name — a filesystem object that unrelated processes can open to reach the same buffer.
3. Signal — a numbered interruption — an asynchronous notification delivered on return to user mode; SIGKILL and SIGSTOP cannot be caught.
4. Shared memory — the same physical pages in two processes — zero-copy IPC requiring caller-supplied synchronization.
5. Socket — a connection endpoint — a descriptor supporting stream or datagram communication, local via AF_UNIX or remote via AF_INET.
6. RPC — calling a function on another machine — a stub-and-skeleton layer that marshals arguments over a transport, with failure modes a local call lacks.

18.11 Device management and drivers

18.11.1 PLAIN - in simple words

1. Chapter 13 covered devices and buses. One line to bring it back: a device is a lump of hardware that the processor talks to through registers, memory regions and interrupts.
2. A **driver** is the piece of software that knows how to talk to one specific device. It is the translator between the general and the particular.
3. The operating system's job here is to make every disk look like a disk, so that the filesystem code does not need to know which brand it is.
4. It does that by defining an interface, and requiring every driver to implement it: open, close, read, write, control.
5. On UNIX systems, devices then appear as files in `/dev`, so programs can talk to hardware using the same calls they use for files.
6. **Hot plug** is the ability to add and remove hardware while running. The kernel notices, loads a driver, and tells userspace.
7. The **I/O scheduler** decides in what order pending disk requests are sent, because order matters enormously for speed on a spinning disk and matters less on a solid-state one.

18.11.2 PLAIN - a picture in your head

1. Think of an electrical socket standard.
2. A kettle, a lamp and a laptop charger are completely different inside. They all end in the same plug.
3. The socket is the interface. The plug is the driver. The building's wiring does not know or care what is on the other end.
4. A new appliance arrives, you plug it in, and the building needs no changes. That is a loadable driver.

5. If a badly made appliance shorts out, it can trip the whole building, because it is connected directly to the mains. That is why a bad driver crashes the kernel.

Where this comparison breaks: an appliance can only draw power, whereas a driver runs inside the kernel with the same privileges as the kernel itself. A closer comparison would be an appliance that could rewire the building. This is exactly why driver signing exists, and why microkernels move drivers out of the kernel.

18.11.3 PLAIN – a worked example

1. Real device files from `/dev` on the machine used here:

```
crw-rw-rw- 1 root root   1, 3 /dev/null
crw-rw-rw- 1 root root   1, 5 /dev/zero
crw-rw-rw- 1 root root   1, 8 /dev/random
brw------- 1 root root 254, 0 /dev/vda
```

2. The first character is the device class. `c` means **character device**: a stream of bytes, read one after another, no seeking. `b` means **block device**: a fixed-size array of blocks you can jump around in.
3. Where a normal file would show its size, a device file shows two numbers. `/dev/null` is 1, 3.
4. The first is the **major number**: which driver handles this. Major 1 is the kernel's memory device driver. Major 254 here is virtio block.
5. The second is the **minor number**: which device that driver should use. Minor 3 is null, minor 5 is zero, minor 8 is random. Same driver, three behaviours.
6. So `/dev/null` contains no data at all. It is a name, a type and two integers. Everything it does is code in the kernel.
7. Writing to `/dev/null` calls a function that discards the bytes and reports success. Reading from it returns end of file immediately.
8. Permissions on device files are ordinary UNIX permissions, and they are the access control. `/dev/vda`, the raw disk, is `brw-------`: only root may open it. `/dev/null` is `crw-rw-rw-`: anyone.

18.11.4 PLAIN – what is really happening inside

1. When you open `/dev/null`, the VFS sees the inode is a character device, reads major 1 minor 3 from it, and looks up major 1 in the character device table.
2. It replaces the file's operations table with that driver's, so every later read and write goes to driver code.
3. The device file is therefore not a link to hardware. It is a name that carries two numbers, and the numbers select code.
4. Drivers do their work in two halves, because interrupts must be short.
5. The top half runs when the interrupt arrives. It does the minimum: acknowledge the device, grab the data, and schedule the rest.
6. The bottom half runs slightly later with interrupts enabled, and does the real work. Linux calls these softirqs, tasklets and workqueues.
7. Hot plug on Linux works like this: the bus controller raises an interrupt, the bus driver reads the new device's identity numbers, the kernel matches those numbers against every driver's table of supported devices, and loads the one that claims it.
8. It then sends an event up to userspace over a netlink socket, where `udev` receives it and creates the device file, sets permissions and may run rules.
9. The I/O scheduler sits between the filesystem and the driver. It holds requests briefly so it can merge adjacent ones and reorder them.
10. On a spinning disk that reordering is worth an enormous amount, because a head seek costs about 5 to 10 milliseconds and a reordering that avoids one saves more time than all the software involved.
11. On an SSD there is no head, so reordering saves much less, and the modern default is a scheduler that mostly stays out of the way.

18.11.5 TECHNICAL - the engineer's version

1. Linux device model: `struct bus_type`, `struct device`, `struct device_driver` and `struct class`, all exposed under `/sys`. `/sys/bus/pci/devices` lists every PCI device; `/sys/class/net` lists every network interface.
2. Driver binding is by ID table. A PCI driver declares a `struct pci_device_id[]` of vendor and device IDs; the core matches and calls the driver's `probe`. Modules carry these tables in a `.modalias` section so `udev` can load them on demand.
3. Character devices are registered with `alloc_chrdev_region` and `cdev_add`; block devices go through the block layer, which today is `blk-mq`, the multi-queue block layer merged in Linux 3.13 in 2014 and made the only path in Linux 5.0 in 2019.
4. Linux I/O schedulers available in the multi-queue layer:

Scheduler	Best for	Note
none	NVMe SSD	No reordering at all
mq-deadline	Mixed, SATA SSD	Deadline per request
bfq	Desktop, HDD	Fair queueing, latency
kyber	Fast multi-queue	Target latency tuning

5. Read and set it per device: `cat /sys/block/sda/queue/scheduler` shows the list with the active one in brackets; writing a name changes it.
6. Windows uses the Windows Driver Model and, since Vista, the Windows Driver Frameworks. KMDF drivers run in kernel mode; UMDF drivers run in user mode, which is Windows quietly adopting the microkernel argument for classes of device where the performance cost is acceptable.
7. Driver signing: 64-bit Windows has required kernel-mode driver signatures since Vista in 2006, and since Windows 10 version 1607 in 2016 new kernel drivers must be signed through the Microsoft hardware developer portal.
8. macOS deprecated kernel extensions and replaced them with DriverKit and System Extensions from macOS 10.15 Catalina in 2019. DriverKit drivers run in user space. On Apple silicon, loading a kext requires lowering the security policy and rebooting.
9. Real interrupt counts from `/proc/interrupts` on the machine used here: IRQ 26 is `ttyS0`, the serial port, with 12,584 interrupts on CPU0 and 0 on CPU1. Interrupt affinity, not chance, put them all on one core.
10. Tools: `lspci -v`, `lsusb`, `lsblk`, `lsmod`, `modinfo`, `udevadm monitor`, `dmesg` on Linux; `system_profiler` and `ioreg` on macOS; Device Manager, `pnputil` and `driverquery` on Windows.

18.11.6 WORDS - remember these

1. Driver — code that knows one specific device — a kernel or user-mode module implementing a standard interface for particular hardware.
2. Device file — a name that selects driver code — a filesystem node carrying a type plus major and minor numbers.
3. Major number — which driver — the index selecting a registered driver in the character or block device table.
4. Minor number — which instance — the value the driver interprets to pick a device or behaviour.
5. Top half and bottom half — do the urgent bit now, the rest later — interrupt handler versus deferred processing in softirq or workqueue context.
6. I/O scheduler — deciding the order of disk requests — the block layer elevator merging and reordering requests, chosen per device.

18.12 Protection and isolation

18.12.1 PLAIN – in simple words

1. Everything in this chapter that stops one program hurting another comes from one idea: some code is trusted and some is not, and the hardware knows which.
2. The processor runs in **user mode** or **kernel mode**. In user mode the dangerous instructions simply fail.
3. A program cannot reach another program's memory because the address it uses is not a real address. It is translated, and the translation table only contains its own memory.
4. If it tries an address that is not in its table, the processor raises a fault, and the kernel usually kills it. That is the segmentation fault.
5. On top of that base, systems add finer tools.
6. **Capabilities** cut root's absolute power into pieces, so a program can be allowed to bind a low port without being allowed to do everything.
7. **Sandboxes** shrink what a program may ask for at all, by filtering the system call list.
8. **Containers** give a process a private view of the filesystem, the process list, the network and the users, while sharing one kernel.
9. **Virtual machines** go further and give a whole fake computer, with its own kernel, on top of a program called a hypervisor.
10. When the trusted code itself fails, there is nothing above it to catch the error, so the machine stops. That is a kernel panic or a blue screen.

18.12.2 PLAIN – a picture in your head

1. Think of a building with staff passes.
2. Kernel mode is the master key. User mode is a pass that opens only your own office.
3. Capabilities are what happens when you cut the master key into forty separate keys and hand out only the ones each person needs.
4. A sandbox is an office where most of the doors have been bricked up, so even a valid key opens nothing.
5. A container is a floor of the building with its own numbering, its own directory board and its own staff list. It feels like a whole building. Walk down the fire stairs far enough and it is still the same building, with one boiler serving everything. That boiler is the shared kernel.
6. A virtual machine is a separate building, with its own boiler, built inside a warehouse. Much stronger separation, much more concrete.

Where this comparison breaks: a wall in a building is physical, and no clever argument gets you through it. Processor isolation is enforced by logic that can have design flaws, which is exactly what Spectre and Meltdown were in 2018: correct permission checks defeated by timing side effects of speculation. The walls were real. Information still leaked around them.

18.12.3 PLAIN – a worked example

1. Namespaces are the mechanism containers are built from. Every process on Linux belongs to one of each kind. Real listing from `/proc/self/ns` on this machine:

```
cgroup -> cgroup:[4026531835]
ipc    -> ipc:[4026531839]
mnt    -> mnt:[4026531832]
net    -> net:[4026531833]
pid    -> pid:[4026531836]
time   -> time:[4026531834]
user   -> user:[4026531837]
uts    -> uts:[4026531838]
```

2. Those numbers are namespace identifiers. Two processes with the same number share that view of the world.

3. Run a container and it gets different numbers for some of these. Same kernel, different view.
4. A process in a new PID namespace sees itself as PID 1 and cannot see any process outside. `ps` inside a container genuinely shows only the container's processes, because the kernel is answering a different question.
5. A process in a new mount namespace has its own filesystem tree. That is how a container's `/` can be an Ubuntu image on a Fedora host.
6. A process in a new network namespace has its own interfaces, its own routing table and its own port numbers. Two containers can both listen on port 80.
7. Namespaces control what you can see. **Cgroups** control how much you can use. Real controllers mounted on this machine include `cpu`, `cpuacct`, `cpuset`, `memory`, `blkio`, `pids`, `devices` and `freezer`.
8. Put the two together and you have a container: a normal process, with a private view from namespaces and a resource cap from cgroups. There is no "container" object in the Linux kernel at all.

18.12.4 PLAIN - what is really happening inside

1. Privilege rings on x86 were designed with four levels, 0 to 3. Every general purpose operating system uses exactly two: 0 for the kernel, 3 for programs.
2. Rings 1 and 2 were meant for drivers and services. They were abandoned because they do not exist on other architectures, and portable kernels could not rely on them.
3. Virtualization added a level below 0. On Intel it is VMX root mode, on AMD it is SVM host mode, on ARM it is EL2. The hypervisor runs there, so a guest kernel can believe it is in ring 0 while actually being supervised.
4. That hardware support arrived in 2005 and 2006, with Intel VT-x and AMD-V. Before it, virtualization required rewriting guest instructions on the fly.
5. Capabilities on Linux split root into about 40 separate powers, such as `CAP_NET_BIND_SERVICE` for low ports, `CAP_SYS_ADMIN` for a great deal, and `CAP_DAC_OVERRIDE` for ignoring file permissions.
6. Sandboxing with seccomp works by giving the kernel a filter program that is run on every system call, deciding allow, deny, kill or trap. Once installed it cannot be removed, and the process cannot regain the lost ability.
7. A kernel panic happens when kernel code hits a condition it cannot recover from: a null pointer dereference in kernel space, a corrupted internal structure, or an unhandled exception in an interrupt handler.
8. There is no higher authority to catch the error and no isolated context to kill, so the only safe action is to stop and print what happened.
9. Windows does exactly the same thing and calls it a bug check. The blue screen shows a stop code and, on modern versions, writes a crash dump to disk before restarting.

18.12.5 TECHNICAL - the engineer's version

1. Linux namespace types and the `clone` flags that create them:

Namespace	Flag	Isolates
Mount	CLONE_NEWNS	Filesystem tree
PID	CLONE_NEWPID	Process IDs
Network	CLONE_NEWNET	Interfaces, ports
User	CLONE_NEWUSER	UID/GID mapping
UTS	CLONE_NEWUTS	Hostname
IPC	CLONE_NEWIPC	SysV IPC, mqueues
Cgroup	CLONE_NEWCGROUP	Cgroup root view
Time	CLONE_NEWTIME	Boot and monotonic

2. Dates: the mount namespace arrived in Linux 2.4.19 in 2002, the rest through 2.6.x, user namespaces completed in 3.8 in 2013, and the time namespace in 5.6 in 2020.

3. Cgroups came from Google, written by Paul Menage and Rohit Seth, merged as "control groups" in Linux 2.6.24 in January 2008. Cgroup v2 with a unified hierarchy became the default in most distributions from around 2021.
4. Container runtimes are userspace assemblers of these primitives: Docker from 2013, then the OCI runtime specification, `runc`, `containerd`, `podman`, and Kubernetes as the orchestrator above them.
5. The honest version: a container shares the host kernel, so a kernel privilege escalation escapes it. A virtual machine shares only the hypervisor, which has a much smaller attack surface. Anyone who tells you containers are as isolating as VMs is selling something.
6. The middle ground is real: Firecracker, released by AWS in 2018, is a minimal virtual machine monitor of roughly 50,000 lines of Rust, with a device model of five devices, booting in about 125 milliseconds. gVisor intercepts system calls in a userspace kernel instead. Kata Containers puts a real VM under a container interface.
7. Mandatory access control layers sit above discretionary permissions: SELinux (originating from the United States National Security Agency, merged in Linux 2.6 in 2003), AppArmor, and on macOS the Sandbox subsystem plus System Integrity Protection, introduced in OS X 10.11 El Capitan in 2015.
8. Hypervisor types: type 1 runs on the hardware directly (Xen from 2003, VMware ESXi, Microsoft Hyper-V, KVM once you accept that Linux becomes the hypervisor); type 2 runs as an application (VirtualBox, VMware Workstation, older Parallels). The line is blurry and the classification is a convention, not a standard.
9. Panic and bug check diagnostics: `kdump` and `crash` on Linux, `/var/crash` dumps, `panic=1` on the kernel command line to reboot instead of hanging (which was set on the machine used here); WinDbg with `!analyze -v` on Windows minidumps in `C:\Windows\Minidump`.
10. Common Windows stop codes worth knowing: `IRQL_NOT_LESS_OR_EQUAL` usually a driver touching paged memory at high interrupt level; `PAGE_FAULT_IN_NONPAGED_AREA` bad pointer or failing RAM; `SYSTEM_SERVICE_EXCEPTION` a fault in a system call path. All three most often mean a driver, not Windows itself.

18.12.6 WORDS - remember these

1. Privilege ring — how much the processor will let you do — hardware protection level, 0 for kernel and 3 for user on x86-64.
2. Capability — one slice of root's power — a per-thread bit such as `CAP_NET_BIND_SERVICE`, granted independently of UID 0.
3. Seccomp — a filter on which system calls you may make — a BPF program applied to every syscall, irreversible once installed.
4. Namespace — a private view of one kind of resource — the kernel mechanism giving a process its own PIDs, mounts, network or users.
5. Cgroup — a cap on how much you may use — hierarchical resource accounting and limiting for CPU, memory, I/O and process count.
6. Hypervisor — the thing that runs whole operating systems — software at a privilege level below the guest kernel, using VT-x, AMD-V or EL2.
7. Kernel panic — the trusted code failed — an unrecoverable kernel fault with no higher authority to handle it; a bug check or blue screen on Windows.

18.13 UNIX: the full story

18.13.1 PLAIN - in simple words

1. Nearly every operating system you will use descends from one project, or was built to imitate it.
2. It began as a failure. In the mid-1960s MIT, General Electric and Bell Labs tried to build Multics, an ambitious time-sharing system for the GE-645.
3. Multics was late, huge and complicated. Bell Labs pulled out in 1969.

4. Two of the Bell Labs people, Ken Thompson and Dennis Ritchie, missed the pleasant working environment they had lost.
5. Thompson found a little-used PDP-7 in a corner and, in 1969, wrote a small system for it, helped by Ritchie and Rudd Canaday.
6. It was a joke on the name: Multics was big and did everything at once, so theirs was **Unics**, later spelled UNIX.
7. In 1970 the group got a PDP-11, and UNIX became a real working system.
8. In 1973 they rewrote almost the whole thing in C, a language Ritchie had made for the purpose. Almost nobody wrote operating systems in a high-level language then.
9. That single decision is why UNIX spread. Move the C compiler to a new machine and most of the operating system comes with it.

18.13.2 PLAIN - a picture in your head

1. Think of a kitchen with a drawer full of single-purpose tools: a peeler, a grater, a sieve, a whisk. Each does one thing well.
2. The alternative is one huge machine with forty buttons that peels, grates, sieves and whisks, badly, and cannot be repaired.
3. UNIX chose the drawer. Small programs, each doing one job, that can be connected end to end.
4. The connection is the pipe: the output of one becomes the input of the next, without either knowing the other exists.
5. To make that work, everything must be the same shape, so the shape chosen was plain lines of text.
6. That is why `ls | grep report | wc -l` works, and why you can invent a combination nobody has ever tried and it will still work.

Where this comparison breaks: kitchen tools are physical and cannot be combined into new tools. UNIX programs can be composed into scripts that become new programs, so the drawer grows. And the text-stream convention has real costs: programs must parse each other's output, which is fragile, and this is exactly the argument PowerShell makes by passing objects instead.

18.13.3 PLAIN - a worked example

1. The UNIX philosophy, as Doug McIlroy, the head of the Bell Labs computing research department and the inventor of the pipe, stated it in the 1978 Bell System Technical Journal:
2. Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new features.
3. Expect the output of every program to become the input to another, as yet unknown, program. Do not clutter output with extraneous information.
4. Design and build software, even operating systems, to be tried early, ideally within weeks. Do not hesitate to throw away the clumsy parts and rebuild them.
5. Use tools in preference to unskilled help to lighten a programming task, even if you have to detour to build the tools and expect to throw some out after you have finished using them.
6. Two further principles are usually added, from Rob Pike and others: everything is a file, and text is the universal interface.
7. The first is why a disk, a terminal, a random number source and a running process all appear as things you can `open` and `read`.

18.13.4 PLAIN - what is really happening inside

1. AT&T could not sell UNIX. A 1956 consent decree, settling an antitrust case, forbade the Bell System from entering businesses outside common carrier communications.
2. So AT&T licensed UNIX to universities for a nominal fee, with source code. That accident is why an entire generation of computer scientists learned operating systems by reading a real one.

3. At Berkeley, students and staff extended their copy heavily: the C shell, the vi editor, virtual memory, the fast filesystem, job control, and the sockets API that the whole internet uses.
4. Their distribution was the Berkeley Software Distribution, BSD, from 1978.
5. The 1984 breakup of AT&T removed the restriction, and AT&T began selling UNIX commercially as System V. Suddenly the free university tradition and a commercial product were the same code.
6. Then came the **UNIX wars**: through the late 1980s, vendors split into rival camps, each with an incompatible UNIX, each claiming to be the standard.
7. Customers responded by demanding a written specification that any vendor could implement, which produced POSIX.
8. In 1992 AT&T's Unix System Laboratories sued Berkeley Software Design over BSD code. The case settled in February 1994, requiring a handful of files to be removed, and 4.4BSD-Lite was released free of AT&T code.
9. The delay mattered. During the two years that BSD's legal status was unclear, a student in Helsinki released a kernel with no legal cloud over it at all.

18.13.5 TECHNICAL - the engineer's version

1. Key dates, all verifiable: Multics development from 1964 at MIT Project MAC with Bell Labs and General Electric; Bell Labs withdrew in 1969; Thompson's PDP-7 work began in 1969; the PDP-11/20 arrived in 1970 and the name UNIX was adopted; Version 4 was rewritten in C in 1973.
2. Ritchie and Thompson's paper *The UNIX Time-Sharing System* appeared in Communications of the ACM, volume 17 number 7, July 1974, and is the document that made the outside world take notice.
3. Thompson and Ritchie received the ACM Turing Award in 1983 for UNIX and for the theory of operating systems.
4. **POSIX** is IEEE Std 1003.1, first published in 1988. The name was suggested by Richard Stallman when the committee wanted something pronounceable. It standardizes the C API, the shell and utilities, not the kernel design.
5. The Single UNIX Specification, maintained by The Open Group, is what confers the right to use the UNIX trademark. macOS is certified UNIX. Linux is not, and has never applied. AIX, HP-UX and Solaris are.
6. The family tree, with founding dates:

System	From	Started
Research UNIX	Bell Labs	1969
BSD	UC Berkeley	1978
System V	AT&T	1983
SunOS / Solaris	Sun	1982 / 1992
AIX	IBM	1986
HP-UX	HP	1984
MINIX	Tanenbaum	1987
Linux	Torvalds	1991
FreeBSD, NetBSD	386BSD	1993
OpenBSD	NetBSD fork	1996
macOS	NeXTSTEP + BSD	2001

7. The split that matters technically: System V gave us `init` runlevels, System V IPC and the `/etc/init.d` layout; BSD gave us sockets, the fast filesystem, `rc` scripts and much of the networking code that every system, including Windows, borrowed.
8. Both branches are still running in production. FreeBSD powers Netflix's content delivery servers and is the base of the PlayStation 4 and 5 system software; OpenBSD produces OpenSSH, which almost every server on earth runs.

18.13.6 WORDS - remember these

1. Multics — the ambitious project UNIX reacted against — the 1964 MIT, GE and Bell Labs time-sharing system that Bell Labs abandoned in 1969.
2. UNIX philosophy — small tools, joined together — composition of single-purpose programs over a universal text interface.
3. BSD — the Berkeley branch of UNIX — the distribution that produced sockets, the fast filesystem and today's FreeBSD, NetBSD and OpenBSD.
4. System V — the AT&T commercial branch — the source of runlevels, System V IPC and much enterprise UNIX practice.
5. POSIX — the written rules any UNIX-like system can follow — IEEE Std 1003.1, first published in 1988, defining API, shell and utilities.
6. Single UNIX Specification — what lets you call it UNIX — The Open Group's certification, held by macOS, AIX, HP-UX and Solaris, not by Linux.

18.14 Linux

18.14.1 PLAIN - in simple words

1. In 1991 a 21-year-old student in Helsinki, Linus Torvalds, wanted a UNIX-like system for his new 386 PC. The teaching system MINIX was restricted and he could not change it freely.
2. On 25 August 1991 he posted to the newsgroup `comp.os.minix`. The message is worth quoting exactly, because it is famous for being wrong:

> ```
> Hello everybody out there using minix -
>
> I'm doing a (free) operating system (just a hobby, won't be
> big and professional like gnu) for 386(486) AT clones.
> ```

3. The post continued that he had ported bash 1.08 and gcc 1.40, asked what features people wanted, and added a postscript: the system was free of any MINIX code, had a multi-threaded filesystem, and was "NOT protable" because it used 386 task switching.
4. Version 0.01 was released on 17 September 1991.
5. He wrote only the kernel. Everything else that made it usable — the shell, the compiler, the text tools, the C library — already existed, from the GNU project.
6. The GNU project had been started by Richard Stallman in 1983 to build a complete free UNIX-like system. By 1991 GNU had almost everything except a working kernel.
7. So the two halves fitted together exactly. That is not luck; it is what happens when both sides target the same POSIX interface.
8. In January 1992, with version 0.12, Torvalds put Linux under the GNU General Public License. He has repeatedly called this the single best decision he made.

18.14.2 PLAIN - a picture in your head

1. Think of an engine and a car.
2. The Linux kernel is the engine. On its own it moves nothing and has no seats.
3. GNU supplied the gearbox, the wheels, the steering and the dashboard.
4. A **distribution** is the finished car: someone chose an engine version, bolted on a particular set of parts, painted it, tested it and put a badge on it.
5. Ubuntu, Debian, Fedora, Arch and Android are all cars built around the same family of engines, and they do not look or drive alike.
6. This is why "which is better, Linux or Ubuntu" is a confused question. One is an engine, the other is a car containing it.

Where this comparison breaks: a car engine is useless outside a car, whereas the Linux kernel really does run alone in embedded systems with a single program on top. And a car maker builds one car; a distribution assembles tens of thousands of independently developed pieces it did not write and cannot fully test.

18.14.3 PLAIN – a worked example

1. What a distribution actually adds to the kernel, concretely:
2. A C library, usually glibc, sometimes musl. The kernel does not provide one.
3. A userland: GNU coreutils, or BusyBox on small systems, or Toybox on Android.
4. An init system and service manager, usually systemd since about 2015.
5. A package manager and a repository of tested, signed, compiled packages: apt with .deb, dnf with .rpm, pacman, apk.
6. A default configuration for thousands of programs, which is most of the real work and almost none of the credit.
7. A desktop environment, if any: GNOME, KDE Plasma, Xfce.
8. A release policy: Debian stable freezes and supports for years; Arch ships updates continuously; Ubuntu LTS releases every two years in April with five years of support.
9. Security updates, which is the service people actually pay for.
10. Numbers for scale: Debian 12, released June 2023, contains over 64,000 binary packages. The kernel is one of them.

18.14.4 PLAIN – what is really happening inside

1. How a change gets into Linux, in order:
2. You write a patch and send it as plain text email to the mailing list for the relevant subsystem, with a `Signed-off-by` line certifying its origin.
3. The subsystem maintainer reviews it. Most patches are rejected or sent back several times. Review is public and often blunt.
4. If accepted, it goes into the maintainer's tree, then into `linux-next`, where it is built and tested against everything else nightly.
5. When the merge window opens, the maintainer sends a pull request to Torvalds, who merges it.
6. The merge window is two weeks. Then come seven or eight weekly release candidates, and then the release. The whole cycle is nine to ten weeks and has been remarkably regular since 2005.
7. Torvalds wrote Git in 2005 in about ten days, specifically because this workflow needed a tool that did not exist. The tool exists because of the process, not the other way round.
8. Version numbers carry no meaning beyond ordering. Torvalds increments the first digit when the second gets "too big", by his own description. That is why 2.6.39 became 3.0, 3.19 became 4.0, and 6.17 became 7.0 in April 2026.
9. Long-term support kernels are chosen once a year and maintained for two to six years, or longer under the Civil Infrastructure Platform.

18.14.5 TECHNICAL – the engineer's version

1. Verified version history and current state, from kernel.org as of 9 August 2026: mainline 7.2-rc7, stable 7.1.8, longterm 6.18.44, 6.12.103, 6.6.151, 6.1.182, 5.15.215 and 5.10.264.

Version	Released	Note
0.01	17 Sep 1991	First release
0.12	Jan 1992	Relicensed to GPL
1.0	14 Mar 1994	First stable
2.6.0	17 Dec 2003	O(1) scheduler era
6.1	11 Dec 2022	Rust support, LTS

Version	Released	Note
6.6	29 Oct 2023	EEVDF replaces CFS
6.12	17 Nov 2024	sched_ext, LTS
7.0	12 Apr 2026	Numbering rollover

2. Licensing: the kernel is GPL version 2 only, not "version 2 or later". That deliberate choice means it can never be relicensed to GPLv3 without the agreement of thousands of copyright holders.
3. The user-space boundary is covered by a syscall exception note, which is why proprietary applications on Linux are legal while proprietary in-kernel drivers are legally contested.
4. The GNU/Linux naming argument, stated fairly. The Free Software Foundation's position: the system is mostly GNU with a Linux kernel, so calling the whole thing Linux erases the project that supplied most of it and the freedom philosophy that motivated it. The common counter-position: the name refers to the kernel and to common usage, most modern systems contain far less GNU code than in 1993, and Android contains none. Both sides are factually correct about different things; the disagreement is about naming, not about facts.
5. Where Linux runs, with real figures. Supercomputers: on the June 2026 TOP500 list all 500 systems run a Linux-based operating system, as they have since November 2017. The list is led by LineShine at the National Supercomputing Centre in Shenzhen at 2,198.40 PFlop/s, ahead of El Capitan at Lawrence Livermore at 1,809.00 PFlop/s and Frontier at Oak Ridge at 1,353.00 PFlop/s.
6. Mobile: Android, which is a Linux kernel with a completely different userland, accounts for roughly 70 percent of smartphone operating system share worldwide, with iOS most of the rest. Share figures move and vary by source, so treat the exact number as approximate.
7. Servers and cloud: Linux is the majority operating system on public cloud instances and on web-facing servers, by a wide margin in every survey; exact percentages vary by methodology and are worth checking before quoting.
8. Embedded: routers, televisions, cars, cameras, industrial controllers, spacecraft. The Mars helicopter Ingenuity ran Linux on a Snapdragon 801 with the F Prime framework, the first Linux flight on another planet, in 2021.
9. Desktop: the smallest share, in the low single digits of percent, and rising slowly. Steam's hardware survey and Statcounter both place it around 2 to 5 percent in 2025 and 2026 depending on method.
10. Contribution scale: roughly 1,500 to 2,000 developers from over 200 companies contribute to each release. The largest corporate contributors in recent years include Intel, Red Hat, Google, AMD, Linaro and Huawei. The image of Linux as hobbyist code has been wrong for over fifteen years.

18.14.6 WORDS - remember these

1. Kernel versus distribution — the engine versus the finished car — the Linux kernel is one component; a distribution assembles it with libc, userland, init, packaging and configuration.
2. GPL — the licence that requires sharing changes — the GNU General Public License version 2 for the kernel, requiring derived works to be distributed under the same terms with source.
3. Merge window — the two weeks when new features are accepted — the first phase of each nine to ten week Linux release cycle.
4. LTS kernel — the version kept alive for years — a long-term support release chosen annually and maintained with backported fixes.
5. Signed-off-by — the line certifying where a patch came from — the Developer Certificate of Origin attestation required on every kernel patch.
6. Userland — everything above the kernel — the libraries, shells and utilities that make a kernel into a usable system.

18.15 Windows

18.15.1 PLAIN – in simple words

1. Windows has two completely separate ancestries that were merged only in 2001.
2. The first is **MS-DOS**. Tim Paterson at Seattle Computer Products wrote 86-DOS in about six weeks in 1980, as a port of the ideas in Digital Research's CP/M to the 8086 processor.
3. Microsoft licensed it, hired Paterson in May 1981, bought the rights that July for $25,000, and licensed it to IBM, which shipped it as PC DOS 1.0 in August 1981.
4. MS-DOS had no protection, no multitasking and no memory management. One program ran and owned the machine.
5. Windows 1.0, in November 1985, was not an operating system. It was a program you ran from DOS that drew overlapping panels and let you switch between applications. Windows 2.0, 3.0 in 1990 and 3.1 in 1992 were the same idea, much improved.
6. The second ancestry is **Windows NT**, started from nothing in 1988 by a team under Dave Cutler, whom Microsoft had recruited from Digital Equipment Corporation.
7. Cutler had designed the RSX-11M and VMS operating systems at Digital. NT is full of VMS ideas, and the resemblance was noticed immediately.
8. Windows NT 3.1 shipped on 27 July 1993. It had protection, preemptive multitasking, proper security and portability across processors.
9. For eight years Microsoft sold two families: the NT line for business, and Windows 95, 98 and Me for home users, which were still DOS underneath.
10. Windows XP in 2001 put everyone on the NT kernel. Every Windows since is NT.

18.15.2 PLAIN – a picture in your head

1. Think of a town with an old wooden high street and, on the edge, a new engineered town centre built to modern codes.
2. The wooden high street is the DOS line. Everything is convenient, nothing is fireproof, and one careless shop burns the row down.
3. The new centre is NT: fire doors, separate services, inspections. Slower to build, much heavier, and it does not burn down.
4. For years people kept shopping on the wooden street because that is where all the shops they liked were.
5. Eventually the new centre was made to look exactly like the old street, so nobody had to change their habits, and the wooden buildings were removed.
6. That disguise is the Win32 API, and it is the real reason the migration worked.

Where this comparison breaks: the disguise is not cosmetic. Win32 is a genuine programming interface with defined semantics, and NT implements it as one of several personalities over a lower interface. And the wooden street was never fully removed: enough compatibility behaviour survives in Windows 11 that programs from 1995 often still run.

18.15.3 PLAIN – a worked example

1. The **registry** is the part of Windows most often described wrongly, so here it is properly.
2. It is a hierarchical database of settings, stored in a small number of binary files called hives, and exposed as a tree of keys and values.
3. Before it, settings lived in hundreds of `.ini` text files scattered everywhere. There was no way to set permissions on them, no transactions, and no way to find them all.
4. The registry gave one namespace, per-value access control, and a defined API. That was a real engineering improvement, not a bureaucratic one.
5. The visible top-level keys and what they actually are:

Key	What it really is
HKEY_LOCAL_MACHINE	Machine-wide settings
HKEY_CURRENT_USER	A link into HKEY_USERS
HKEY_USERS	Every loaded user hive
HKEY_CLASSES_ROOT	Merge of machine and user
HKEY_CURRENT_CONFIG	A link to a hardware key

6. Only `HKEY_LOCAL_MACHINE` and `HKEY_USERS` hold real data. The others are views and links, which is why the same value appears in two places.
7. On disk the hives are `C:\Windows\System32\config\SYSTEM`, `SOFTWARE`, `SAM` and `SECURITY`, plus `NTUSER.DAT` in each user's profile.
8. Value types include `REG_SZ` (text), `REG_DWORD` (32-bit number), `REG_BINARY` and `REG_MULTI_SZ`.
9. Registry cleaners are, with rare exceptions, useless. A registry with 200,000 unused keys costs a modern machine no measurable time, because lookups are indexed. This is settled, not controversial.

18.15.4 PLAIN - what is really happening inside

1. The NT kernel is layered, and the layers have names worth knowing.
2. The **HAL**, hardware abstraction layer, is `hal.dll`. It hides differences between motherboards: interrupt controllers, timers, bus access. Above it, the rest of the kernel sees one abstract machine.
3. The **kernel** proper, sometimes called the microkernel, handles scheduling, interrupts, synchronization primitives and multiprocessor coordination. It is small and never paged out.
4. The **executive** sits above it and contains the real subsystems: the object manager, process manager, memory manager, I/O manager, cache manager, security reference monitor and configuration manager. All of these live in `ntoskrnl.exe`.
5. The **object manager** is NT's most distinctive idea. Processes, threads, files, mutexes, registry keys and devices are all objects in one namespace with one set of rules for naming, referencing and securing.
6. Above the executive, in user mode, are **subsystems**: personalities that present a particular API. Win32 was one, and OS/2 and POSIX subsystems existed early on and were later removed.
7. That structure is why the Windows Subsystem for Linux was possible. WSL 1, in 2016, was a new subsystem translating Linux system calls to NT ones. WSL 2, from 2019, gave up and shipped a real Linux kernel in a lightweight Hyper-V virtual machine instead, because full syscall compatibility proved harder than virtualization.

18.15.5 TECHNICAL - the engineer's version

1. NT 3.1 shipped for four architectures in 1993: Intel x86, DEC Alpha, MIPS and later PowerPC. That portability was designed in from the start, in contrast to early Linux.
2. NT's design lineage from VMS is direct: asynchronous I/O request packets, the interrupt request level scheme, the object and handle model, and the layered driver stack all have VMS counterparts.
3. The version numbering is a source of confusion. NT 3.1, 3.5, 3.51, 4.0, then 5.0 was Windows 2000, 5.1 was XP, 6.0 Vista, 6.1 Windows 7, 6.2 Windows 8, 6.3 Windows 8.1, then 10.0 for both Windows 10 and Windows 11.
4. Current state as of August 2026: Windows 11 version 26H1, build 10.0.28000, released 10 February 2026, a platform release focused on ARM devices including Snapdragon X2. Version 25H2, build 26200, shipped 30 September 2025.
5. **NTFS**, shipped with NT 3.1 in 1993, is a journaling filesystem built around a Master File Table in which every file, including the MFT itself, is a record with a set of attributes. Small files live entirely inside their MFT record. It supports hard links, alternate data streams, compression, encryption, quotas and sparse files.
6. Alternate data streams are worth knowing about: `file.txt:hidden` is a second stream on the same file, invisible to `dir`, and historically a malware hiding place.

7. **Win32** is the documented API, in `kernel32.dll`, `user32.dll` and `gdi32.dll`. It calls the undocumented native API in `ntdll.dll`. Microsoft's compatibility promise attaches to Win32, not to the native layer.
8. Driver signing timeline: 64-bit editions have required signed kernel-mode drivers since Vista in 2006; since Windows 10 version 1607 in 2016, new kernel drivers must be submitted to and signed by the Microsoft hardware developer portal. Attestation signing covers most, EV certificates are required to submit.
9. Modern additions: virtualization-based security uses Hyper-V to isolate credentials in a separate virtual trust level; Windows 11 requires TPM 2.0 and UEFI Secure Boot; the July 2024 CrowdStrike incident, in which a faulty configuration update to a kernel-mode security driver bug-checked roughly 8.5 million Windows machines worldwide, restarted the industry argument about how much security software belongs in the kernel at all.
10. Tools: Sysinternals suite (Process Explorer, Process Monitor, Autoruns, WinObj, Handle), WinDbg with the public symbol server, Performance Monitor, Event Viewer, `wevtutil`, and PowerShell's `Get-Process` and `Get-Service`.

18.15.6 WORDS - remember these

1. MS-DOS — the single-tasking ancestor — the 16-bit real-mode system bought from Seattle Computer Products in 1981 and shipped as PC DOS 1.0.
2. Windows NT — the engineered line that survived — Dave Cutler's 1993 kernel with protection, portability and preemptive multitasking.
3. HAL — the layer that hides the motherboard — `hal.dll`, abstracting interrupt controllers, timers and bus access.
4. Executive — the main body of kernel services — object, process, memory, I/O, cache, security and configuration managers inside `ntoskrnl.exe`.
5. Subsystem — a personality presenting one API — user-mode environment such as Win32, and formerly OS/2 and POSIX.
6. Registry — one database instead of thousands of ini files — hierarchical configuration store in binary hives with per-value access control.
7. NTFS — the Windows filesystem since 1993 — journaling filesystem organized around a Master File Table of attributed records.

18.16 macOS, iOS, Android and the rest

18.16.1 PLAIN - in simple words

1. Modern Apple systems come from a company Apple did not own in 1996.
2. Steve Jobs left Apple in 1985 and founded NeXT, which built expensive workstations and an operating system called NeXTSTEP.
3. NeXTSTEP combined the Mach kernel from Carnegie Mellon with BSD UNIX code and a set of programming frameworks written in Objective-C.
4. Apple, meanwhile, had spent most of the 1990s failing to replace the ageing classic Mac OS, which had no memory protection and cooperative multitasking.
5. Apple announced it would buy NeXT on 20 December 1996, and closed the deal on 7 February 1997 for about $427 million. Jobs returned with it.
6. NeXTSTEP became the base of Mac OS X, released for desktops on 24 March 2001.
7. The same foundation, with a different interface layer, is iOS, iPadOS, watchOS, tvOS and visionOS. One core, many products.
8. Android took a different route entirely: the Linux kernel, with none of the GNU userland, and an application layer designed from scratch by Google.

18.16.2 PLAIN – a picture in your head

1. Think of two ways of putting a new engine into an old car brand.
2. Apple bought a whole different car, rebadged it, and gradually made it look like the old one. The dashboard was familiar, everything under it was new.
3. Google took a well-known engine, threw away every other part of the donor car, and built a completely new vehicle around it.
4. So an Android phone runs Linux, and almost nothing you know about a Linux desktop applies to it: different C library, different init, different graphics, different IPC, different package format.
5. Calling Android "Linux" is true about the engine and misleading about the car.

Where this comparison breaks: Apple did not simply rebadge. XNU is a genuine hybrid whose Mach layer and BSD layer are welded together in ways neither original design intended, and it has been rewritten heavily over 25 years.

18.16.3 PLAIN – a worked example

1. **XNU** is the Apple kernel. The name stands for "X is Not Unix", a joke from the NeXT era, and it is made of three parts.
2. Part one: **Mach**, from Carnegie Mellon University, originally a microkernel research project by Rick Rashid and Avie Tevanian. Apple uses a heavily modified OSF MK 7.3. It provides virtual memory, tasks, threads, scheduling and Mach ports for messaging.
3. Part two: **BSD**, derived from 4.3BSD and refreshed from FreeBSD. It provides the POSIX API, the process model, users and permissions, the network stack and the VFS.
4. Part three: **IOKit**, the driver framework, written in a restricted subset of Embedded C++ with no exceptions, no templates and no multiple inheritance.
5. All three run in one address space in kernel mode. Mach's separation is a structure, not an isolation boundary. That is why XNU is called hybrid.
6. **Darwin** is the open-source part: XNU plus the BSD userland plus the core libraries. Apple publishes it. The frameworks, the interface and the applications on top are not open.
7. Above Darwin sit the frameworks: Cocoa and AppKit on macOS, UIKit on iOS, written in Objective-C, and SwiftUI, from 2019, written in Swift.
8. Objective-C came from NeXT. Swift was announced in June 2014 and is now the default for new Apple development, with Objective-C still fully supported.

18.16.4 PLAIN – what is really happening inside

1. **System Integrity Protection**, introduced in OS X 10.11 El Capitan in 2015, is a rule enforced by the kernel that even root may not modify protected system locations, load unsigned kernel extensions or attach a debugger to Apple binaries.
2. It is configured from the recovery environment, not from the running system, which is the point: a compromised running system cannot turn it off.
3. Since macOS 10.15 Catalina in 2019, the system volume is a separate read-only volume, and since Big Sur in 2020 it is a cryptographically sealed snapshot.
4. The **Apple silicon transition** was announced at WWDC on 22 June 2020. The first M1 Macs shipped in November 2020, and the transition finished with the Mac Pro in June 2023.
5. Two translation layers made it survivable: Rosetta 2, which translates x86-64 binaries ahead of time on install, and universal binaries containing both architectures in one file.
6. **Android** is a Linux kernel plus an entirely separate userland.
7. Its C library is Bionic, written by Google, not glibc. Its init is Android's own. Its IPC is **Binder**, a kernel driver originally from Be Incorporated's OpenBinder, doing object-oriented calls between processes.
8. Its application runtime is **ART**, the Android Runtime, introduced as a preview in Android 4.4 KitKat in 2013 and made the only runtime in Android 5.0 Lollipop in 2014, replacing the older Dalvik virtual machine.

9. ART started as purely ahead-of-time compilation. Since Android 7.0 Nougat it is a hybrid: new apps run interpreted and JIT-compiled while a profile is collected, and the hot code is compiled ahead of time when the device is idle and charging.
10. Between the kernel and the framework sits Android's **HAL**, a set of defined interfaces every vendor must implement, so that Android can be ported without Google seeing the vendor's driver source.
11. Project Treble, from Android 8.0 in 2017, split the vendor implementation from the framework so the framework can be updated independently. It was a response to the update problem, and it helped without solving it.

18.16.5 TECHNICAL - the engineer's version

1. Current versions as of August 2026: macOS 26 Tahoe, released 15 September 2025, introducing the Liquid Glass design across Apple platforms.

System	Kernel	Userland
macOS	XNU (Mach+BSD)	BSD + Apple
iOS, iPadOS	XNU	BSD + Apple
Android	Linux	Bionic, Toybox, ART
ChromeOS	Linux	Chrome, Ash, crosvm
FreeBSD	FreeBSD	BSD
QNX	QNX microkernel	POSIX

2. Apple filesystem history: HFS from 1985, HFS+ from 1998, and APFS announced in 2016, shipped on iOS 10.3 in March 2017 and macOS 10.13 High Sierra in September 2017. APFS is copy-on-write with cheap snapshots, native encryption and space sharing across volumes in a container.
3. Security hardware: the Secure Enclave, a separate co-processor with its own boot ROM, present since the A7 in 2013, holding keys and biometric templates that the main processor never sees.
4. On Apple silicon, memory pages are 16 KiB rather than 4 KiB, which changes memory footprint measurements noticeably compared with x86 Macs.
5. Android's kernel is a fork with substantial out-of-tree patches, though the Generic Kernel Image programme since Android 11 has pushed vendors towards a common core. Android security patches are shipped monthly on a published bulletin schedule.
6. ChromeOS is Gentoo-derived Linux with a hardened design: verified boot with a read-only rootfs, automatic background updates to an alternate partition, and Linux applications run inside a virtual machine using crosvm and Termina. Announced 7 July 2009, first Chromebooks June 2011.
7. **Real-time operating systems** are a different category with a different promise: not speed, but a bounded worst-case response time. Names worth knowing: VxWorks (which ran the Mars rovers Spirit, Opportunity, Curiosity and Perseverance), QNX Neutrino (a true microkernel, dominant in automotive infotainment), FreeRTOS, Zephyr and RTEMS.
8. The distinction that matters: hard real-time guarantees a deadline and treats a miss as a failure; soft real-time treats a miss as degraded quality. Linux with the PREEMPT_RT patch set, merged into mainline in Linux 6.12 in November 2024 after roughly twenty years of development, offers strong soft real-time and, with care, hard real-time on suitable hardware.

18.16.6 WORDS - remember these

1. XNU — Apple's kernel — a hybrid of Mach for memory and messaging, BSD for POSIX and networking, and IOKit for drivers.
2. Darwin — the open part of macOS — XNU plus BSD userland and core libraries, published by Apple without the frameworks or interface.
3. Mach port — Apple's basic IPC handle — a capability referring to a message queue, underlying most macOS inter-process communication.

4. SIP — root is not allowed either — System Integrity Protection, kernel enforced since OS X 10.11 in 2015, configurable only from recovery.
5. Binder — Android's IPC — a kernel driver providing object-oriented calls between processes, derived from OpenBinder.
6. ART — Android's application runtime — profile-guided hybrid of interpretation, JIT and ahead-of-time compilation, replacing Dalvik in Android 5.0.
7. RTOS — an operating system that promises deadlines — bounded worst-case latency rather than best average throughput.

18.17 How you would actually write one

18.17.1 PLAIN - in simple words

1. Writing an operating system is not mysterious. It is a long series of small, well-documented steps, each of which is achievable in an evening.
2. The reason few people finish is not difficulty. It is that there are hundreds of steps and no user to please until quite far in.
3. The minimum thing that deserves the name is small: it must start on its own, take control of the processor, put something on the screen, and respond to a key press.
4. That is achievable in a weekend, in about 300 lines, and it is a real operating system in the same sense that a paper aeroplane is a real aircraft.
5. After that the work has a natural order, because each part needs the one before it.
6. You need interrupts before you can have a keyboard. You need a memory allocator before you can have processes. You need processes before a scheduler. You need a disk driver before a filesystem. You need a filesystem before a shell.
7. You do this on a simulated machine, not real hardware, because a simulated machine restarts in a second and lets you inspect every register.
8. Everything you need is documented, free, and has been done by thousands of people who wrote down what went wrong.

18.17.2 PLAIN - a picture in your head

1. Think of building a house on empty ground where nothing exists, not even roads.
2. First you need a track wide enough to bring in a digger. That is the bootloader: just enough to get the next thing in.
3. Then foundations: level ground, marked out, load bearing. That is switching the processor into its proper mode and setting up the tables it needs.
4. Then the wiring loom, before any room is finished, because everything depends on it. That is the interrupt table.
5. Then plumbing: a system for allocating what is scarce. That is the memory manager.
6. Only then do rooms appear, and people move between them. That is processes and the scheduler.
7. The kitchen and the front door come last, and they are the only parts a visitor ever notices. That is the filesystem and the shell.

Where this comparison breaks: a builder can inspect a half-built house by walking through it. When your kernel fails, the machine simply stops, with no message, because the code that would print a message is the code that broke. Building a way to see inside your own system is a real early task, not a luxury.

18.17.3 PLAIN - a worked example

1. The smallest complete thing: a boot sector that prints a character. This is real, assembles with NASM, and runs in QEMU.

```
bits 16
org 0x7c00
```

```
start:
    mov ah, 0x0e        ; BIOS teletype function
    mov al, 'K'
    int 0x10            ; call BIOS video service
    jmp $               ; loop here forever

    times 510-($-$$) db 0
    dw 0xaa55           ; boot signature
```

2. Line by line. `bits 16` says generate 16-bit code, because the processor starts in real mode.
3. `org 0x7c00` tells the assembler that this code will live at address 0x7C00, which is where BIOS loads a boot sector. That is fixed, not a choice.
4. `mov ah, 0x0e` selects the BIOS teletype output service, `mov al, 'K'` is the character, and `int 0x10` calls the BIOS video interrupt.
5. `jmp $` jumps to itself forever, because there is nothing to return to.
6. `times 510-($-$$) db 0` pads with zeros up to byte 510.
7. `dw 0xaa55` writes the two-byte signature that BIOS checks before it will treat the sector as bootable.
8. Build and run it with two commands:

```
nasm -f bin boot.asm -o boot.bin
qemu-system-i386 -drive format=raw,file=boot.bin
```

9. A window opens and shows the letter K. Nothing else on that machine exists. No operating system, no library, no memory manager, no C runtime. You wrote every instruction that ran.
10. That is 512 bytes and it is the honest starting point of every hobby OS.

18.17.4 PLAIN - what is really happening inside

1. The realistic order of work, with what each stage costs an average learner working evenings.
2. **Boot and print** — one weekend. The example above.
3. **Protected mode or long mode** — one to two weeks. Build a global descriptor table, set a bit in a control register, and jump. Getting the jump right is the classic first wall.
4. **A C environment** — one week. Write a linker script, a small startup stub, and build with a cross-compiler so you are not accidentally linking against your host's C library.
5. **Interrupts** — two to three weeks. An interrupt descriptor table, entry stubs that save registers, remapping the interrupt controller, a timer and a keyboard.
6. **Physical and virtual memory** — one month. A page frame allocator, page tables, and a heap. This is where most projects stall, because bugs here produce silent corruption rather than error messages.
7. **Processes and a scheduler** — one month. Save and restore context, a task list, round robin. The first successful switch between two tasks is the most satisfying moment in the project.
8. **User mode** — two to four weeks. Separate address spaces, a system call entry, and a separate stack per privilege level.
9. **A disk driver and a filesystem** — one to two months. ATA PIO is simplest. FAT16 is the easiest real filesystem to read, and writing is much harder than reading.
10. **A shell** — two weeks, and it feels like finishing, because for the first time you can type at your own system and it answers.
11. Total, for one person working evenings: roughly six months to a year to reach a usable shell. Two weekends to reach something that boots and responds.

18.17.5 TECHNICAL - the engineer's version

1. Tooling, all free. A cross-compiler built as `i686-elf-gcc` or `x86_64-elf-gcc`, so headers and libraries from your host cannot leak in. NASM or GAS for assembly. GNU ld with a custom linker script. GNU make.

2. Emulators: QEMU for speed and GDB support, Bochs for its built-in debugger which can single-step at the hardware level, and VirtualBox for a final check against something closer to real firmware.
3. Debugging technique that changes everything: run QEMU with `-s -S` and attach GDB to port 1234. You can then single-step your kernel from the very first instruction, with symbols.
4. Bootloader choices: write your own for learning, use GRUB with the Multiboot2 specification to skip straight to a 32-bit or 64-bit C environment, or use Limine for a modern UEFI-capable option.
5. Learning resources, by name.

Resource	Kind	Focus
OSDev Wiki	Reference	Everything, x86 detail
Three Easy Pieces	Textbook	Free, thorough
xv6, MIT 6.1810	Teaching OS	Readable UNIX in C
Writing an OS in Rust	Tutorial	Modern, step by step
The little book about	Short book	Minimal x86 kernel
MINIX 3	Microkernel	Tanenbaum's system
SerenityOS	Live project	Desktop, from nothing
Redox OS	Live project	Microkernel in Rust

6. Notes on those. The OSDev wiki is the single most useful page collection in this field, and its "Beginner Mistakes" page will save you weeks. *Operating Systems: Three Easy Pieces*, by Remzi and Andrea Arpaci-Dusseau at Wisconsin, is free online and is the best modern textbook. *The little book about OS development*, by Erik Helin and Adam Renberg, is the shortest complete path from nothing to a small x86 kernel. xv6 is MIT's teaching reimplementation of Sixth Edition UNIX in ANSI C for RISC-V, with a line-by-line commentary book. Philipp Oppermann's *Writing an OS in Rust* is the best written tutorial series of the last decade. SerenityOS was started by Andreas Kling in 2018 and is now a complete graphical UNIX-like system written from nothing including its own browser engine.
7. Specifications you will actually read: the Intel Software Developer's Manual volume 3 for system programming, the AMD64 Architecture Programmer's Manual volume 2, the Multiboot2 specification, the UEFI specification, the ATA and NVMe specifications, and the OSDev wiki's summaries of all of them.
8. The honest scope warning. A hobby kernel that boots, schedules and runs a shell is perhaps 5,000 to 20,000 lines. A system you could daily-drive needs drivers for real hardware, a network stack, a graphics stack, a browser and a toolchain, and that is where the number reaches millions. SerenityOS took a full-time effort and many contributors over years to reach a usable desktop.
9. Why do it anyway: after building a scheduler you will never again be confused about what a context switch is, and after building a page allocator you will never again be confused about virtual memory. The understanding is disproportionate to the code.

18.17.6 WORDS - remember these

1. Cross-compiler — a compiler that targets a different system — a toolchain such as `i686-elf-gcc` that cannot accidentally use the host's libraries.
2. Boot sector — the 512 bytes BIOS will run — loaded at 0x7C00 and required to end with the signature 0xAA55.
3. Global descriptor table — the table that describes memory regions — the x86 structure that must exist before entering protected mode.
4. Interrupt descriptor table — the list of handler addresses — the table mapping interrupt and exception numbers to entry points.
5. Multiboot2 — a standard way for a bootloader to hand over — the specification GRUB implements, delivering a kernel into a known state with a memory map.
6. Linker script — the file that decides where code lands — the ld input that defines sections and load addresses for a freestanding binary.

18.98 Common wrong ideas

1. Wrong: the operating system is the desktop, the icons and the windows. Right: those are ordinary programs. The operating system is the kernel plus the services underneath, and a machine with no graphics at all still has a complete operating system.
2. Wrong: Linux is an operating system. Right: Linux is a kernel. What you install is a distribution, which adds a C library, a userland, an init system and tens of thousands of packages. Android proves the point: same kernel, none of the rest.
3. Wrong: a process and a program are the same thing. Right: a program is a file that does nothing. A process is that program while running, with its own memory, descriptors and scheduling state. One program can be many processes.
4. Wrong: multitasking means the processor runs several programs at once. Right: on one core it runs one thing at a time and switches quickly. Real simultaneity requires several cores. On this machine the kernel had performed 4,153,017 context switches in 95 minutes to produce the illusion.
5. Wrong: threads are always much cheaper than processes. Right: measured here, creating a thread cost 32.8 microseconds against 182.5 for a fork, about 5.6 times. But forking after dirtying 256 MB cost 2,620 microseconds, because page tables must still be copied. The ratio depends on what the process holds.
6. Wrong: `malloc` gives you memory. Right: it gives you addresses. Physical memory is allocated one 4 KiB page at a time, on the first write to each page, through a page fault. Asking for a gigabyte moves `VSZ` and leaves `RSS` unchanged.
7. Wrong: the out-of-memory killer means your machine ran out of RAM through bad management. Right: it means the kernel's earlier promises could not all be kept at once. Overcommit is deliberate, because programs reserve far more than they touch.
8. Wrong: a system call is just a function call into a library. Right: it is a hardware mode transition. The `syscall` instruction changes privilege level and jumps to an address only the kernel could set. It cost 89 nanoseconds here against one or two for a normal call.
9. Wrong: `write` returning success means the data is on the disk. Right: it means the data is in the page cache and marked dirty. Only `fsync`, `fdatasync` or `O_DIRECT` involve the storage device. Databases care about this difference more than anything else.
10. Wrong: containers are lightweight virtual machines. Right: containers are ordinary processes with a restricted view, built from namespaces and cgroups, sharing the host kernel. A kernel vulnerability escapes a container and does not escape a virtual machine.
11. Wrong: a blue screen or kernel panic means the operating system is badly written. Right: it usually means a driver failed. Drivers run with full privilege, there is no higher authority to catch their errors, and stopping is safer than continuing with corrupted state.
12. Wrong: `kill -9` is the proper way to stop a program. Right: `SIGKILL` cannot be caught, so the program cannot flush buffers, release locks or write its state. `SIGTERM` first, wait, then `SIGKILL`. A process stuck in `D` state ignores both.

18.99 Chapter summary in 20 lines

1. An operating system exists to do three things: share the hardware, keep programs apart, and hide devices behind a simple interface.
2. All of its power comes from two hardware features: a privileged processor mode, and a timer interrupt that forces control back to the kernel.
3. The kernel is the privileged part. It is entered only three ways: a system call, an interrupt, or a fault. Between those it does nothing.
4. Kernel designs run from monolithic (Linux) through hybrid (Windows NT, XNU) to microkernel (MINIX 3, QNX, seL4); the 1992 Tanenbaum-Torvalds debate settled nothing and both approaches ship today.

5. Kernels are written in C for predictable code with no hidden runtime, plus assembly for entry paths, and since Linux 6.1 in December 2022, Rust.
6. Booting is a chain: power good, reset vector at 0xFFFFFFF0, firmware and POST, boot device, bootloader, kernel, initramfs, root filesystem, PID 1.
7. On the machine used here the kernel phase took 2.55 seconds, and PID 1 started at 1.17 seconds, before the real root was mounted at 2.55.
8. A process is a running program with its own address space, descriptors and state; UNIX creates one with `fork` then `exec`, Windows with `CreateProcess`.
9. Threads share everything except stack and registers, which makes them cheap and dangerous; the kernel schedules them individually in the 1:1 model.
10. The scheduler answers one question thousands of times a second, and every algorithm trades throughput against responsiveness; round robin loses on averages and wins on how the machine feels.
11. Linux used CFS from 2.6.23 in 2007 and replaced it with EEVDF in 6.6 in October 2023; Windows uses 32 priority levels with dynamic boosts.
12. A system call puts a number in `rax`, arguments in six registers, executes `syscall`, and lands where `MSR_LSTAR` points; printing "hello" took about twenty-five of them, of which one was the point.
13. The kernel allocates physical page frames lazily on first touch, overcommits deliberately, and kills the highest-scoring process when the promise fails.
14. The virtual filesystem layer dispatches through per-filesystem function tables, which is why `/proc` can be read like a file that does not exist.
15. Three tables sit behind every open file: the per-process descriptor table, the system-wide open file table, and the inode table.
16. UNIX permissions are owner, group, other with read, write, execute, plus setuid, setgid and sticky; Windows uses ordered access control lists.
17. Pipes, FIFOs, signals, message queues, shared memory, sockets and RPC differ in copies, message boundaries and whether they cross machines.
18. Containers are namespaces plus cgroups on a shared kernel; virtual machines add a hypervisor below the guest kernel and isolate far more strongly.
19. The lineage is one family: Multics failed, UNIX began on a PDP-7 in 1969, was rewritten in C in 1973, split into BSD and System V, was standardized as POSIX in 1988, and produced Linux in 1991, macOS via NeXT in 2001, and Android on a Linux kernel in 2008.
20. Every one of the 500 machines on the June 2026 TOP500 list runs Linux, as has been true since November 2017, while Windows NT's 1993 design still runs most desktops and XNU runs every Apple device.

19

CHAPTER

The Terminal and the Shell

19.0 What this chapter gives you

1. You will be able to say what a terminal actually was, as a physical machine, and why the word survived after the machine died.
2. You will be able to tell apart five things that people constantly mix up: terminal, terminal emulator, shell, console, and command line.
3. You will be able to explain what a pseudo-terminal is, and describe exactly what happens inside the machine when you press Ctrl-C.
4. You will be able to describe the shell's read-parse-expand-execute loop, and name the two system calls that start every program you run.
5. You will be able to use redirection and pipes correctly, including the case where `2>&1` in the wrong place does nothing useful.
6. You will be able to read an exit code, chain commands with `&&` and `||`, and say why continuous integration systems only look at that number.
7. You will be able to fix a broken `PATH`, and say which file to edit on macOS, on a Linux server, and why the answer differs between the two.
8. You will be able to explain the Windows registry properly, and say why a macOS or Linux developer almost never thinks about one.
9. You will be able to use around forty commands with confidence, and read a manual page to find the fortieth-first.
10. You will be able to write a real shell script with error handling that stops instead of continuing into damage.

19.1 What a terminal actually is

19.1.1 PLAIN – in simple words

1. Today, a terminal is a window on your screen with text in it.
2. That is a costume. The word means something older and physical.
3. A **terminal** was a machine that sat at the end of a wire.
4. The other end of the wire went to a computer, often in another building.
5. The terminal had a keyboard and a way to show what came back.
6. It had almost no brain of its own. It sent characters out and printed characters in. Nothing more.
7. The earliest ones printed onto a paper roll, like a till receipt that never ends.
8. Later ones had a small screen instead of paper, and that was a huge saving in paper and noise.
9. When personal computers arrived, nobody needed the physical machine any more. The computer was on your desk.
10. But all the software already knew how to talk to a terminal. So we kept pretending. A program on your screen now acts like the old machine.

11. That is why the window is called a terminal, and why it still obeys commands designed for a device made in 1978.

19.1.2 PLAIN - a picture in your head

1. Imagine a hotel in 1970 with one telephone switchboard in the basement.
2. Every room has a simple phone. The phone has no brain. It has a handset and a dial.
3. All the intelligence, all the routing, all the connections, live in the basement.
4. The phone in the room is the terminal. The switchboard is the computer.
5. Ten rooms can be connected at once. The basement machine handles all ten, giving each a slice of its attention.
6. Now the hotel modernizes. Every room gets its own full computer. The basement machine is thrown away.
7. But the guests learned to use the handset. So the new computers put a picture of a handset on their screens, and it works the same way.
8. That picture is the terminal emulator you use today.

Where this comparison breaks: a hotel phone carries sound as a continuous wave, while a terminal carries discrete characters, one byte at a time, with a strict alphabet. And the old terminal was not entirely brainless. It knew how to move its own cursor, clear its own screen, and pick a colour, when told to by special character sequences. That small amount of local intelligence is exactly the part that survives in your terminal window today.

19.1.3 PLAIN - a worked example

1. Picture the Teletype Model 33, sold from 1963. Engineers called it the ASR-33.
2. ASR stands for Automatic Send-Receive: it could also punch and read paper tape.
3. It weighed around 25 kilograms. It printed onto a roll of paper. It was loud, like a typewriter that never stopped.
4. It ran at 110 bits per second. That is 10 characters per second.
5. Type a line of 80 characters and wait 8 seconds for the reply to finish printing. That was the normal rhythm of computing.
6. Because printing was slow, command names were made short. This is the real reason UNIX has `ls`, `cp`, `mv`, `rm` and not `list`, `copy`, `move`, `remove`.
7. There is no scrolling back on a paper terminal. What is printed is printed. To see something again, you print it again.
8. There is no clearing the screen either. There is no screen.
9. Now compare the DEC VT100, introduced in 1978. It had a glass screen showing 24 rows of 80 characters.
10. Suddenly the cursor could be moved anywhere, text could be erased, and the screen could be redrawn. Full-screen text editors became possible.
11. Below is a real capture of the bytes a modern terminal still uses to turn text red and bold. This was taken from the sandbox used to write this chapter.

```
$ printf '\033[1;31mRED BOLD\033[0m\n' | od -c
0000000 033   [   1   ;   3   1   m   R   E   D       B   O
0000020   L   D 033   [   0   m  \n
```

12. `033` is the octal number for the ESC character, decimal 27, hex 1B.
13. `ESC [ 1 ; 3 1 m` means "bold, red foreground". `ESC [ 0 m` means "reset everything".
14. That exact sequence was defined for terminals of the VT100 era. Your terminal in 2026 still speaks it.

19.1.4 PLAIN – what is really happening inside

1. A terminal and a computer are joined by a serial line: one wire for characters going out, one for characters coming in.
2. When you press the A key, the terminal sends the single byte 65 down the wire. It does not draw an A on its own.
3. The computer receives 65, decides what to do, and usually sends 65 straight back. That is called echo.
4. Only when the byte comes back does the terminal print it. So on a slow or broken line, your typing does not appear.
5. This is why an old terminal feels like a conversation: everything you see is something the far end chose to send you.
6. Most bytes mean "print this character". A few mean "do this action".
7. Byte 7 rings the bell. Byte 8 moves the print head back one space. Byte 10 moves down a line. Byte 13 returns to the left margin.
8. Byte 27, ESC, means "the next few bytes are not text, they are an instruction".
9. `ESC [` starts a control sequence. Numbers and a final letter follow.
10. `ESC [ 2 J` clears the screen. `ESC [ 10 ; 5 H` puts the cursor on row 10, column 5.
11. Every manufacturer invented their own sequences at first, which was chaos. A standard fixed that, and the VT100 was the terminal that made the standard popular.
12. In UNIX, every device is a file. The file that represents the connection to a terminal was named after the machine on the other end.
13. Teletype was shortened to tty. The device is `/dev/tty`. That is the whole origin of the name.

19.1.5 TECHNICAL – the engineer's version

1. The Teletype Corporation Model 33 was introduced as a commercial product in 1963, after an original design for the United States Navy.
2. It was one of the first products to use ASCII, first published in 1963. Over half a million Model 33 units had been built by 1975.
3. The Model 33 defined by accident two long-lived conventions: code 17 (Ctrl-Q, DC1) as XON and code 19 (Ctrl-S, DC3) as XOFF for flow control.
4. Ctrl-S still freezes many terminals today, and confused users still think their machine has crashed. It has not. Press Ctrl-Q.
5. The DEC VT52 arrived in September 1975 with a proprietary escape set. The DEC VT100 arrived in 1978 and implemented the emerging ANSI sequences.
6. The relevant standards, in order: ECMA-48 adopted 1976; ANSI X3.41-1974 and ANSI X3.64-1977 cited in the VT100 manual; the name "ANSI escape sequence" dates from ANSI's 1979 adoption of X3.64.
7. ECMA-48 and X3.64 were merged into ISO/IEC 6429. ANSI withdrew X3.64 in 1994 in favour of the international standard. Japan adopted JIS X 0211.
8. So the correct name in 2026 is ECMA-48 or ISO/IEC 6429. "ANSI escape codes" is a convention of speech, not a live standard name.
9. The VT100 used an Intel 8080 processor, showed 24 lines of 80 columns, and supported 132-column mode. Its market success created a clone industry: Zenith Z-19 in 1979, Qume QVT-108, Televideo TVI-970, Wyse WY-99GT.
10. Because sequences still varied, UNIX grew a capability database: termcap, then terminfo. Programs ask the database, not the hardware.
11. The `TERM` environment variable names your terminal type. `tput` and `infocmp` read the database. Real output from this sandbox:

```
$ TERM=xterm-256color tput colors
256
$ TERM=xterm-256color tput setaf 1 | od -c
0000000 033   [   3   1   m
$ infocmp xterm | head -5
```

```
xterm|xterm-debian|xterm terminal emulator (X Window System),
        am, bce, km, mc5i, mir, msgr, npc, xenl,
        colors#8, cols#80, it#8, lines#24, pairs#64,
        bel=^G, blink=\E[5m, bold=\E[1m, cbt=\E[Z,
```

12. Note the honest detail: plain `xterm` claims 8 colours, `xterm-256color` claims 256. Setting `TERM` wrongly is a common cause of broken colours and broken arrow keys over SSH.

Machine	Year	Output	Speed
Teletype 33ASR	1963	paper roll	110 bit/s
DEC VT52	1975	24x80 screen	up to 19200
DEC VT100	1978	24x80 screen	up to 19200
Modern emul.	2026	window	memory speed

13. The honest version: your terminal is not emulating a VT100 exactly. It emulates a superset, usually announced as `xterm-256color`. Mouse reporting, bracketed paste, true colour and Unicode are all later additions that no VT100 ever had.

19.1.6 WORDS - remember these

1. Terminal — a machine at the end of a wire for typing and reading — a character-oriented I/O device with a keyboard and display.
2. Teleprinter — a typewriter that talks over a wire — an electromechanical send-receive device using a serial character code.
3. ASR-33 — the famous noisy paper terminal — Teletype Model 33 Automatic Send-Receive, 1963, 110 bit/s, 10 characters per second.
4. Escape sequence — a few characters that mean “do something” not “print something” — a control string beginning with ESC, per ECMA-48.
5. tty — the name of the terminal device file — abbreviation of teletype, exposed as `/dev/tty` and `/dev/ttyN`.
6. terminfo — a list of what each terminal model can do — a compiled capability database consulted through `tput` and curses.
7. Echo — the far end sending your keystroke back so you can see it — local or remote reflection of input characters, controlled by the line discipline.

19.2 Terminal, terminal emulator, shell, console and command line

19.2.1 PLAIN - in simple words

1. Five words get used as if they mean the same thing. They do not.
2. A **terminal** is the device, real or pretended, that carries characters in and out.
3. A **terminal emulator** is a program that draws a terminal on your screen. Terminal.app and iTerm2 on macOS are terminal emulators.
4. A **shell** is a different program entirely. It reads what you type and runs programs for you. `bash` and `zsh` are shells.
5. A **console** originally meant the one screen physically attached to the machine, used for its most important messages.
6. A **command line** is not a program at all. It is a style of working: you type a line, you press Enter, something happens.
7. When you open Terminal.app on a Mac, you start a terminal emulator, and it starts a shell inside itself.
8. The window is the emulator. The `%` or `$` prompt inside it comes from the shell.
9. If the shell crashes, the window stays but nothing responds. If the window closes, the shell dies with it.

19.2.2 PLAIN - a picture in your head

1. Think of a theatre.
2. The building is the terminal emulator. It has seats, lighting, a stage, and a door to the street.
3. The play performed on the stage is the shell. It is what you actually came to watch.
4. The building does not know the plot. It only supplies a place to perform and a way for the audience to see and be heard.
5. You can put a different play in the same building. Run `zsh` instead of `bash` and nothing about the window changes.
6. You can also put the same play in a different building. Run `bash` under iTerm2 instead of Terminal.app and the play is identical.
7. The console is the stage manager's desk backstage: the one place that gets the emergency announcements even when the audience does not.

Where this comparison breaks: a theatre and a play are physically separate, while your emulator and shell are joined by a specific and strange piece of kernel plumbing, not by air. And the shell can run with no building at all, reading commands from a file, with nobody watching. That is a shell script.

19.2.3 PLAIN - a worked example

1. Open Terminal.app on macOS. You now have four things stacked up.
2. Terminal.app itself is a normal macOS application, written with Apple's UI toolkit. It draws characters and reads your keyboard.
3. It asked the operating system for a pseudo-terminal, a fake wire.
4. It started `/bin/zsh` on the far end of that fake wire.
5. `zsh` printed a prompt into the fake wire, and Terminal.app drew it.
6. Prove it to yourself. Type `tty` and press Enter. On macOS you will see something like `/dev/ttys003`. On Linux you will see `/dev/pts/0`.
7. Real output from the Linux sandbox used to write this chapter, running a shell inside a real pseudo-terminal:

```
$ tty
/dev/pts/0
$ echo $SHELL
/bin/bash
$ ps -o pid,ppid,pgid,sid,tpgid,stat,tty,comm
  PID  PPID  PGID   SID TPGID STAT TT       COMMAND
30511 30510 30511 30511 30512 Ss   pts/0    bash
30512 30511 30512 30511 30512 R+   pts/0    ps
```

8. Read that table. `bash` has process ID 30511. `ps` has 30512 and its parent is 30511, so the shell started it.
9. Both are attached to terminal `pts/0`. That is the fake wire.
10. `TPGID` is 30512, meaning the terminal's foreground group is currently `ps`, not the shell. That is what "the shell is waiting" looks like from outside.

19.2.4 PLAIN - what is really happening inside

1. The emulator owns one end of the fake wire. The shell owns the other.
2. Everything you type goes into the emulator, which writes those bytes into its end.
3. The operating system carries them across and hands them to the shell.
4. Everything the shell or its programs print goes the other way, and the emulator draws it.
5. The emulator has one extra job: it must understand escape sequences. When bytes `ESC [ 2 J` arrive, it clears its own drawing area.
6. The shell has one extra job: it must decide what your line of text means.
7. Neither knows much about the other. You can replace either one.

8. The console is a separate idea. On a Linux server it is where kernel messages appear, even before any emulator exists.
9. On a physical Linux machine, pressing Ctrl-Alt-F2 gives you a real text console, `/dev/tty2`, drawn by the kernel itself with no emulator involved.
10. On macOS there is no such thing for the user. There is a boot-time verbose console and a system log, but no switchable text consoles.
11. Windows is different again. Its window is `conhost.exe` or Windows Terminal, and the thing inside is `cmd.exe` or `powershell.exe`.

19.2.5 TECHNICAL - the engineer's version

Thing	What it is	Examples
Terminal	character device	tty, pty, serial
Terminal emulator	GUI application	Terminal.app, iTerm2
Shell	command language	bash, zsh, fish
Console	primary device	/dev/console, tty1
Command line	interaction mode	not a program

1. On Linux, virtual consoles are `/dev/tty1` through `/dev/tty63`, driven by the kernel VT subsystem, with no user-space emulator.
2. `/dev/console` is the kernel's message destination, set by the `console=` kernel command-line parameter at boot.
3. `/dev/tty` with no number is a magic alias meaning "the controlling terminal of the process reading it".
4. Pseudo-terminal slaves on Linux are `/dev/pts/N`, provided by the `devpts` filesystem. On macOS and BSD they are `/dev/ttysNNN`.
5. Common emulators, with real facts: xterm, first released 1984, still the compatibility reference. GNOME Terminal, KDE Konsole, Alacritty, kitty, WezTerm, iTerm2 on macOS, Windows Terminal from Microsoft.
6. Windows Terminal reached version 1.0 on 19 May 2020. Before it, Windows had only the legacy console host.
7. macOS Terminal.app ships with macOS. It defaults `TERM` to `xterm-256color` in recent versions. iTerm2 is a third-party replacement with split panes and its own escape extensions.
8. The distinction matters operationally. Colour problems and key problems are emulator or `TERM` problems. Completion and prompt problems are shell problems. Never debug one by changing the other.

19.2.6 WORDS - remember these

1. Terminal emulator — the window program — a user-space application that allocates a pty and renders ECMA-48 output.
2. Shell — the program that runs your commands — a command language interpreter, specified for `sh` by POSIX IEEE Std 1003.1.
3. Console — the machine's own primary screen — `/dev/console`, the kernel's message device, selected at boot.
4. Virtual console — a full-screen text session on Linux — `/dev/ttyN`, implemented in the kernel VT layer.
5. Controlling terminal — the terminal that owns your session — the tty a session leader has opened, reachable as `/dev/tty`.

19.3 The pseudo-terminal

19.3.1 PLAIN – in simple words

1. There is no wire and no machine any more. So the operating system fakes one.
2. The fake is called a **pseudo-terminal**, usually shortened to PTY.
3. It comes as a pair of ends that are joined inside the kernel.
4. One end is held by the terminal emulator. Whatever it writes appears as keyboard input on the other end.
5. The other end is held by the shell. Whatever the shell prints comes out of the first end for the emulator to draw.
6. In between sits a piece of kernel code called the **line discipline**.
7. The line discipline is not a pipe. It is an active thing that changes the characters as they pass.
8. It is why backspace deletes a letter instead of printing a strange symbol.
9. It is why the shell does not see your line until you press Enter.
10. It is why Ctrl-C stops a program, instead of being delivered to it as the letter it technically is.

19.3.2 PLAIN – a picture in your head

1. Imagine two people passing notes through a slot in a wall.
2. On the wall's slot sits a clerk. The clerk reads every note before passing it on.
3. When you write a letter, the clerk copies it onto a public board so you can see what you wrote. That is echo.
4. When you scribble out a letter, the clerk erases it from the board and from the note. That is backspace.
5. The clerk holds each note until you write a full stop. Only then is the note pushed through. That is line buffering.
6. If you write a special mark meaning "stop", the clerk does not pass the mark along. The clerk runs to the other room and pulls the person out of their chair. That is Ctrl-C.
7. The clerk can be told to stop doing all of this. In that mode every letter goes through instantly, unchanged, uncopied.
8. Text editors and games ask for exactly that mode, because they want to react to every single key.

Where this comparison breaks: the clerk is not slow and not optional. It is kernel code running in microseconds, and there is always one, even in the mode where it does almost nothing. Also, the clerk does not understand your notes. It only recognizes about a dozen specific characters and treats every other byte as data to be passed along.

19.3.3 PLAIN – a worked example

1. Here are the special characters the line discipline is watching for, taken from a real `stty -a` run inside a pseudo-terminal on Linux.

```
$ stty -a
speed 38400 baud; rows 0; columns 0; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D;
start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W;
lnext = ^V; discard = ^O; min = 1; time = 0;
isig icanon iexten echo echoe echok -noflsh -tostop
```

2. Read the important ones. `intr = ^C` means Ctrl-C is the interrupt key.
3. `susp = ^Z` means Ctrl-Z suspends. `eof = ^D` means Ctrl-D signals end of input.
4. `erase = ^?` means the Delete character erases one letter. `kill = ^U` erases the whole line.
5. `werase = ^W` erases one word. `lnext = ^V` means "take the next key literally, do not treat it specially".
6. Now the flags on the last line, which are the settings, not the keys.
7. `icanon` means canonical mode is on: input is collected into lines.
8. `echo` means the kernel prints your keystrokes back for you.

9. `isig` means the special keys generate signals. Turn this off with `stty -isig` and Ctrl-C becomes an ordinary byte.
10. Notice `speed 38400 baud`. There is no wire, so this number is fiction. It is kept only because programs still ask for it.

19.3.4 PLAIN - what is really happening inside

1. Here is the whole chain, from your finger to a character on the glass.

```
 your keyboard
      |
      v
[ window system: macOS AppKit / X11 / Wayland ]
      |  key event
      v
[ terminal emulator process ]
      |  writes bytes
      v
+-----------------------------+
|  PTY MASTER  (/dev/ptmx)    |
+-----------------------------+
      |    kernel
      v
+-----------------------------+
|  LINE DISCIPLINE            |
|  echo, erase, line buffer,  |
|  Ctrl-C -> SIGINT,          |
|  Ctrl-Z -> SIGTSTP          |
+-----------------------------+
      |
      v
+-----------------------------+
|  PTY SLAVE   (/dev/pts/0)   |
+-----------------------------+
      |  read() returns a line
      v
[ shell process: bash or zsh ]
      |  fork + exec
      v
[ the command you ran ]
      |  writes to fd 1
      v
 back up through slave, line
 discipline, master, emulator,
 and onto the screen
```

2. Step by step, when you type `l` then `s` then Enter:
3. The emulator writes the byte `l` into the master.
4. The line discipline copies it back towards the master, so the emulator draws `l`. It also stores it in a small buffer, not yet given to the shell.
5. Same for `s`. The buffer now holds `ls`.
6. You press Enter, which sends carriage return, byte 13. The line discipline translates it to newline, byte 10, and now considers the line complete.
7. The shell's `read()` call, which was blocked and waiting, returns the three bytes `l`, `s`, newline.
8. Now the interesting one. You press Ctrl-C. That is byte 3.
9. The line discipline sees byte 3 matches `intr`. It does not put it in the buffer.
10. Instead it sends the signal SIGINT to every process in the terminal's foreground process group.

11. The default action of SIGINT is to end the process. So your running command stops. The shell was not in the foreground group, so it survives.
12. Ctrl-D is different. It is not a signal. It means "end of file now".
13. If the buffer has text in it, Ctrl-D pushes that text through immediately.
14. If the buffer is empty, `read()` returns zero bytes, which means end of input, and the shell exits.
15. That is why Ctrl-D on an empty line logs you out, and why pressing it in the middle of a typed line does nothing visible.
16. Ctrl-Z sends SIGTSTP, which suspends. The process freezes in place. The shell notices, takes back the terminal, and prints your prompt.

19.3.5 TECHNICAL - the engineer's version

1. A PTY is a bidirectional character device pair. On Linux the master is obtained by opening `/dev/ptmx`, which allocates a new slave in `/dev/pts`.
2. The relevant calls are `posix_openpt`, `grantpt`, `unlockpt`, `ptsname`, then `open` on the slave. `openpty` and `forkpty` wrap all of it.
3. The slave is made the controlling terminal by the child calling `setsid` then `ioctl(TIOCSCTTY)`, or implicitly on first open by a session leader.
4. Terminal settings live in `struct termios` with four flag sets: `c_iflag` input, `c_oflag` output, `c_cflag` control, `c_lflag` local, plus the `c_cc` array of control characters.
5. `ICANON` in `c_lflag` selects canonical mode. Cleared, you are in raw mode and `VMIN` and `VTIME` in `c_cc` control read behaviour.
6. `ECHO`, `ECHOE`, `ECHOK`, `ISIG`, `IEXTEN` are the other `c_lflag` bits you will actually touch. `OPOST` and `ONLCR` in `c_oflag` translate newline to carriage-return newline on output.
7. Signals generated by the line discipline, with their Linux numbers:

Key	Byte	Signal	Default action
Ctrl-C	0x03	SIGINT (2)	terminate
Ctrl-\	0x1C	SIGQUIT (3)	terminate, core
Ctrl-Z	0x1A	SIGTSTP (20)	stop
none	none	SIGTTIN (21)	stop background
none	none	SIGTTOU (22)	stop background

8. Signals go to the foreground process group of the controlling terminal, found with `tcgetpgrp` and set with `tcsetpgrp`. This is the field shown as `TPGID` by `ps`.
9. Job control was first implemented in the C shell by Jim Kulp at IIASA in Austria, using features of the 4.1BSD kernel, released 1981.
10. A background process that reads from the terminal gets SIGTTIN and stops. That is why a backgrounded interactive program mysteriously freezes.
11. Window size is not part of `termios`. It is `struct winsize` set with `ioctl(TIOCSWINSZ)`, and changing it delivers SIGWINCH, signal 28, to the foreground group.
12. Real job-control evidence from a pseudo-terminal session in this sandbox:

```
demo$ sleep 300 &
[1] 30514
demo$ jobs
[1]+  Running                 sleep 300 &
demo$ ps -o pid,pgid,stat,tty,comm --ppid $$
  PID  PGID STAT TT       COMMAND
30514 30514 S    pts/0    sleep
30515 30515 R+   pts/0    ps
demo$ kill %1
```

13. Note that `sleep` has its own PGID equal to its PID. Each job gets its own process group, which is what makes Ctrl-C hit one job and not all of them.
14. The `+` in `R+` means "in the foreground process group". `sleep` has no plus, because it is a background job.
15. Windows had no equivalent until ConPTY, the Windows Pseudo Console API, which first shipped in the Windows 10 October 2018 update, version 1809. Before that, tools faked it by screen-scraping the console.
16. The honest version: a PTY is not a perfect terminal. There is no real baud rate, no parity, no modem control lines. `stty` will happily accept settings for hardware that does not exist and silently ignore them.

19.3.6 WORDS - remember these

1. PTY — a fake wire between a window and a shell — a pseudo-terminal device pair, master plus slave, joined in the kernel.
2. Line discipline — the kernel code that edits your typing — the tty layer implementing canonical mode, echo and signal generation.
3. Canonical mode — input is handed over one whole line at a time — `ICANON` set in `termios.c_lflag`, with line editing by the kernel.
4. Raw mode — every key reaches the program instantly — `ICANON` and `ECHO` cleared, `VMIN` and `VTIME` controlling `read` behaviour.
5. Foreground process group — the job that owns the keyboard right now — the process group returned by `tcgetpgrp` on the controlling terminal.
6. SIGINT — the polite "stop that" signal from Ctrl-C — signal 2, default action terminate, catchable and ignorable.
7. Job control — running several programs from one terminal — process groups, sessions and the SIGTSTP, SIGCONT, SIGTTIN, SIGTTOU family.

19.4 What a shell is

19.4.1 PLAIN - in simple words

1. A shell is a program with a very small job description.
2. It reads a line of text. It works out what you meant. It asks the operating system to run something. It waits. It prints a prompt again.
3. That is the whole loop. Everything else is decoration on those four steps.
4. The shell is not part of the operating system. It is an ordinary program.
5. You can have several shells installed and switch between them freely.
6. The shell is also a programming language, with variables, conditions and loops. That is what makes shell scripts possible.
7. A few commands are built into the shell itself, because they must be. `cd` is the main one.
8. `cd` cannot be a separate program, because a separate program cannot change its parent's current directory.
9. Everything else, `ls` and `grep` and `python`, is a real file on disk that the shell finds and launches.

19.4.2 PLAIN - a picture in your head

1. Think of a restaurant with one waiter and a kitchen.
2. You say "two coffees and the soup". The waiter does not cook.
3. The waiter interprets. "Two coffees" becomes two separate drink orders. "The soup" means today's soup, so the waiter fills in the detail.
4. The waiter walks to the kitchen and passes the orders. The kitchen is the operating system.
5. The waiter then stands and waits until the food is ready, and brings it back.

6. Then the waiter returns to your table and stands ready again. That is the prompt.
7. If you ask for something not on the menu, the waiter comes back and says so. That is `command not found`.
8. The waiter can also do a few things without the kitchen: move you to another table, remember your name. Those are the built-in commands.

Where this comparison breaks: the waiter understands meaning, and will guess sensibly if you are vague. The shell does not understand anything. It applies fixed textual rules in a fixed order. If those rules turn your line into nonsense, it passes the nonsense to the kitchen without hesitation. A waiter who behaved like a shell would fetch a chair when you said "chair" in the middle of a sentence about the weather.

19.4.3 PLAIN - a worked example

1. You type this and press Enter:

```
grep -c 500 access.log
```

2. Step 1, read. The shell has the line as a string of 24 characters.
3. Step 2, split into words. It uses spaces and tabs, giving four words: `grep`, `-c`, `500`, `access.log`.
4. Step 3, expand. It checks each word for things it must rewrite: variables, wildcards, braces, tildes, backticks. Here there are none, so nothing changes.
5. Step 4, find the program. `grep` has no slash in it, so the shell searches the directories listed in `PATH`, in order.
6. It finds `/usr/bin/grep` and stops looking.
7. Step 5, run it. The shell makes a copy of itself, and the copy replaces itself with `grep`.
8. Step 6, wait. The shell sleeps until `grep` finishes.
9. `grep` prints `3` and exits with status 0.
10. Step 7, record the result. The shell stores 0 in the variable `?` and prints the prompt.
11. The real run, from this sandbox:

```
$ grep -c 500 access.log
3
$ echo $?
0
```

19.4.4 PLAIN - what is really happening inside

1. The shell cannot simply "become" `grep`, because then it would be gone and you would have no shell left.
2. So it uses a two-step trick that UNIX has used since the beginning.
3. First it calls **fork**. The operating system makes a near-identical copy of the shell process. Two processes now exist, running the same code.
4. Both come back from `fork`, but with different answers. The parent gets the child's process number. The child gets zero.
5. That difference is how each one knows who it is.
6. The child then calls **exec**. This does not create a process. It throws away the current program's memory and loads a new program in its place.
7. The process number does not change. Open files do not change. Environment variables do not change. Only the code and data are replaced.
8. This is exactly why redirection works. Between fork and exec, the child can quietly rearrange its own files, and the new program inherits the arrangement without knowing.
9. The parent calls **wait**. It sleeps until the child ends, and collects the child's exit number.
10. If you typed `&` at the end, the parent skips the waiting and prints the prompt immediately. That is a background job.

11. Between pressing Enter and seeing output, then: line read, words split, expansions applied, program located, fork, file descriptors set up, exec, program runs, output travels back up the pseudo-terminal, parent collects the exit status, prompt printed.
12. On a modern machine that whole sequence takes well under a millisecond, plus however long your program actually takes.

19.4.5 TECHNICAL - the engineer's version

1. The canonical loop, in C-like pseudocode, is short enough to memorize:

```
for (;;) {
    print_prompt();
    line = read_line();          /* from fd 0 */
    words = tokenize(line);
    words = expand(words);       /* order matters, see 19.12 */
    if (is_builtin(words[0])) { run_builtin(words); continue; }
    pid = fork();
    if (pid == 0) {
        setup_redirections();    /* dup2 on 0,1,2 */
        execvp(words[0], words); /* only returns on failure */
        _exit(127);              /* command not found */
    }
    waitpid(pid, &status, 0);
    last_status = WEXITSTATUS(status);
}
```

2. `fork` is defined in POSIX. Linux implements it with `clone`. Modern implementations use copy-on-write, so the copy is cheap: page tables are duplicated, physical pages are shared until written.
3. `vfork` and `posix_spawn` exist as cheaper variants. Real shells often use `fork` anyway because the child needs to run arbitrary setup code.
4. `execvp` is the variant that searches `PATH` and takes an argument vector. The `e` variants take an explicit environment; without them the current `environ` is inherited.
5. A successful `exec` never returns. Any code after it runs only on failure, which is why the example calls `_exit(127)`.
6. Exit status is packed into an integer by `wait`. `WEXITSTATUS` extracts the low 8 bits. `WIFSIGNALED` and `WTERMSIG` cover death by signal.
7. Builtins are required for anything that must change shell state: `cd`, `export`, `exec`, `set`, `unset`, `shift`, `trap`, `read`, `wait`, `eval`, `source`, `umask`, and the job control commands.
8. Some builtins exist only for speed. `echo`, `test`, `[`, `pwd`, `kill` and `printf` all exist as real binaries in `/usr/bin` as well.
9. You can see the duplication:

```
$ type -a echo
echo is a shell builtin
echo is /usr/bin/echo
echo is /bin/echo
$ type cd
cd is a shell builtin
```

10. This matters. `/usr/bin/echo -e` and the bash builtin `echo -e` behave differently, and scripts that assume one get the other under `sh`. Use `printf` when it matters. That advice is a convention, not a standard, but it is close to universal among careful script authors.
11. Instrument the whole thing with `strace -f -e trace=execve,clone,wait4` on Linux, or `dtruss` on macOS, which needs elevated privileges and System Integrity Protection considerations.

19.4.6 WORDS - remember these

1. Shell — the program that runs your typed commands — a command language interpreter and scripting language.
2. Built-in — a command the shell performs itself — a function inside the shell binary, not a file in `PATH`.
3. fork — make a copy of the running program — the POSIX system call that duplicates a process, returning 0 to the child.
4. exec — replace this program with another — the `execve` family, which overwrites the address space and keeps the PID.
5. wait — pause until the child is finished — `waitpid`, which reaps the child and returns its termination status.
6. Prompt — the text asking you for a command — a shell-generated string from `PS1`, printed to the terminal before each read.

19.5 The shells themselves

19.5.1 PLAIN - in simple words

1. There have been many shells. A handful matter.
2. The first UNIX shell was written by Ken Thompson in 1971. It could run programs and redirect their input and output. It could not do much else.
3. Stephen Bourne wrote a much better one at Bell Labs, and it shipped as the default in Version 7 UNIX. That is the **Bourne shell**, the program `sh`.
4. It added variables, loops, conditions and functions. It made shell scripting a real thing. Everything since is measured against it.
5. Bill Joy wrote the **C shell** at Berkeley, with a syntax that looked more like the C language, and with job control and command history.
6. David Korn at Bell Labs wrote the **Korn shell**, taking the good ideas from the C shell and putting them into a Bourne-compatible shell.
7. Brian Fox wrote **bash** for the GNU project, free of licence restrictions, and it became the shell of Linux.
8. Paul Falstad wrote **zsh**, which is Bourne-compatible but with far better completion and matching. Apple made it the macOS default in 2019.
9. **fish** broke compatibility on purpose, to be friendly out of the box.
10. **PowerShell** from Microsoft is not in this family at all. It moves objects between commands instead of text.

19.5.2 PLAIN - a picture in your head

1. Think of English and its descendants over centuries.
2. Old English is the Thompson shell: recognizable, but you cannot read a newspaper with it.
3. Middle English is the Bourne shell: the grammar settles, and texts written in it are still readable today with effort.
4. Then two dialects grow in different towns. One is the Korn shell, which keeps the old grammar and adds new words. The other is the C shell, which changes the grammar and confuses travellers.
5. bash is the modern standard dialect, taught in schools, understood everywhere, slightly dull.
6. zsh is the same language spoken by people who care about pronunciation, with a large vocabulary of convenience.
7. fish is a constructed language: cleaner and easier, but nobody else's documents work in it.
8. PowerShell is not a dialect of English. It is a different language family entirely, from a different continent.

Where this comparison breaks: languages drift by accident, but shells were designed on purpose, by named people, with written justifications. And unlike English, there is a formal written standard for

shell grammar: POSIX. A script written to that standard genuinely runs on all of the Bourne-family shells, which is not something you can say about human dialects.

19.5.3 PLAIN – a worked example

1. The same task, count files ending in `.md`, in four shells.

```
# sh, bash, zsh, ksh: the Bourne family
n=$(ls *.md | wc -l)
echo "there are $n files"

# csh and tcsh: different assignment syntax entirely
set n = `ls *.md | wc -l`
echo "there are $n files"

# fish: no dollar on assignment, no equals sign
set n (ls *.md | wc -l)
echo "there are $n files"

# PowerShell: no text at all, real objects
$n = (Get-ChildItem *.md).Count
"there are $n files"
```

2. Notice that the Bourne family line is identical across four shells. That is the value of POSIX compatibility.
3. Notice that the C shell needs spaces around = and uses `set`. Scripts do not port between the families.
4. Notice that PowerShell never counted lines of text. `Get-ChildItem` returned a list of file objects, and `.Count` asked the list how long it was.
5. That last point is the whole design difference. In UNIX shells, the thing passing between commands is bytes. In PowerShell it is typed objects with properties.

19.5.4 PLAIN – what is really happening inside

1. Why did so many shells appear? Because the shell is small, personal, and used constantly. Small annoyances are worth fixing.
2. The C shell's improvements were real: job control, history with `!!`, aliases, directory stacks.
3. Its scripting was genuinely bad. A widely circulated essay by Tom Christiansen, "Csh Programming Considered Harmful", listed the reasons, and the argument was largely won.
4. The Korn shell showed you could have both: Bourne syntax plus the interactive features. It became the commercial UNIX default.
5. bash exists for a legal reason as much as a technical one. GNU needed a shell it could ship freely, without AT&T code.
6. zsh went further on interactive quality: completion that understands the command you are typing, spelling correction, shared history, better globbing.
7. Apple's move to zsh in 2019 was also partly legal. bash version 4 changed to the GPL version 3 licence, which Apple will not ship, so macOS was frozen on bash 3.2 from 2007.
8. fish decided that compatibility was the thing holding shells back. It has syntax highlighting and autosuggestions with no configuration at all.
9. PowerShell came from a different problem. Windows configuration is not text files, so parsing text was useless. Passing objects was the natural answer on that system.

19.5.5 TECHNICAL – the engineer's version

Shell	Year	Author	Note
sh (V6)	1971	Ken Thompson	first UNIX shell

Shell	Year	Author	Note
sh	1979	Stephen Bourne	shipped in V7 UNIX
csh	1978	Bill Joy	job control, history
ksh	1983	David Korn	Bourne plus csh ideas
bash	1989	Brian Fox	GNU, Linux default
zsh	1990	Paul Falstad	macOS default 2019
fish	2005	A. Liljencrantz	not POSIX by design
PwrShell	2006	Jeffrey Snover	object pipeline

1. Precise dates worth knowing. Bash 1.0 was released on 8 June 1989 by Brian Fox at the Free Software Foundation. The name is Richard Stallman's pun: Bourne-again shell.
2. zsh 1.0 was released in 1990 by Paul Falstad, then a sophomore at Princeton University. The name comes from the login ID of a Princeton teaching assistant, Zhong Shao.
3. fish 1.0 was released on 13 February 2005 by Axel Liljencrantz. The name is "friendly interactive shell".
4. PowerShell's design was published in the Monad Manifesto by Jeffrey Snover in August 2002. It was demonstrated in October 2003, renamed PowerShell on 25 April 2006, and version 1.0 shipped on 14 November 2006.
5. PowerShell Core 6.0, released January 2018, made it cross-platform and open source. It runs on Linux and macOS today.
6. macOS 10.15 Catalina, released 7 October 2019, made zsh the default login shell for new accounts. Existing accounts kept bash. Apple ships `/bin/bash` as version 3.2.57 for licence reasons and it is not updated.
7. The POSIX shell is specified in IEEE Std 1003.1, the Shell and Utilities volume. `sh` on Debian and Ubuntu is `dash`, not bash, which is a common source of "works on my machine" script failures.
8. Verified in this sandbox:

```
$ ls -l /bin/sh
lrwxrwxrwx 1 root root 4 Mar 31  2024 /bin/sh -> dash
$ bash --version | head -1
GNU bash, version 5.2.21(1)-release (x86_64-pc-linux-gnu)
```

9. Practical advice, and this is opinion held by most working engineers rather than a rule: write scripts for `sh` or `bash`, use whatever you like interactively. Never write scripts in `csh`. Never assume `#!/bin/sh` gives you bash.
10. Where experts disagree: some argue that fish and zsh's improvements should be adopted in scripts too, and that POSIX compatibility is a museum concern. Others point out that servers, containers and rescue images often contain only `sh`, so portable scripts still pay. Both are right in their own context.

19.5.6 WORDS - remember these

1. Bourne shell — the original serious shell — `sh`, from Version 7 UNIX 1979, ancestor of the POSIX shell grammar.
2. POSIX shell — the written standard for shell syntax — IEEE Std 1003.1 Shell and Utilities volume.
3. bash — the common Linux shell — Bourne-again shell, GNU project, first released June 1989.
4. zsh — the macOS default since 2019 — Z shell, Bourne-compatible with advanced completion and globbing.
5. dash — a small fast `sh` — Debian Almquist shell, POSIX-only, `/bin/sh` on Debian and Ubuntu.
6. Object pipeline — passing structured data instead of text — PowerShell's model, where commands emit and consume .NET objects.

19.6 stdin, stdout and stderr

19.6.1 PLAIN - in simple words

1. Every program starts life with three channels already open.
2. Channel 0 is **standard input**. It is where the program reads from. By default, your keyboard.
3. Channel 1 is **standard output**. It is where results go. By default, your screen.
4. Channel 2 is **standard error**. It is where complaints go. Also your screen, by default.
5. Output and errors are separate on purpose. This is one of the best design decisions in UNIX.
6. If they were mixed, saving a program's results to a file would also save its error messages into the same file, ruining the data.
7. Because they are separate, you can capture the results and still see the errors on your screen, live.
8. The shell lets you point any of these channels somewhere else before the program starts.
9. That is called **redirection**, and the program never knows it happened.

19.6.2 PLAIN - a picture in your head

1. Think of a factory machine with three pipes attached.
2. One pipe brings raw material in at the top.
3. One pipe sends finished product out of the front.
4. One pipe sends scrap and warning notes out of the side.
5. The machine does not know or care where the pipes go. It just pushes things into them.
6. Before switching the machine on, you can move any pipe. Put the product pipe into a barrel. Leave the scrap pipe pointing at the floor so you notice it.
7. Or connect the product pipe of one machine to the input pipe of the next machine. Now you have a production line.
8. Nothing inside either machine changed. Only the plumbing.

Where this comparison breaks: real pipes have no memory, but these have a small buffer, so a fast machine can run ahead of a slow one for a while. And the scrap pipe is not really for scrap. Progress messages, prompts and warnings all come out of it, including from programs that are working perfectly.

19.6.3 PLAIN - a worked example

1. Here is a real run. The directory has `a.txt` and does not have `nope.txt`.
2. First, no redirection at all. Both streams reach the screen and get mixed:

```
$ ls a.txt nope.txt
ls: cannot access 'nope.txt': No such file or directory
a.txt
```

3. Now send only standard output to a file:

```
$ ls a.txt nope.txt > out.txt
ls: cannot access 'nope.txt': No such file or directory
$ cat out.txt
a.txt
```

4. The error still appeared live on the screen. The file has only the good result. That is exactly what you want.
5. Now send only standard error to a file:

```
$ ls a.txt nope.txt 2> err.txt
a.txt
$ cat err.txt
ls: cannot access 'nope.txt': No such file or directory
```

6. Now send both to the same file, correctly:

```
$ ls a.txt nope.txt > both.txt 2>&1
$ cat both.txt
ls: cannot access 'nope.txt': No such file or directory
a.txt
```

7. Now the same two pieces in the wrong order. Watch what happens:

```
$ ls a.txt nope.txt 2>&1 > wrong.txt
ls: cannot access 'nope.txt': No such file or directory
$ cat wrong.txt
a.txt
```

8. The error went to the screen, not the file. The redirection did not work.
9. Why: `2>&1` means "make channel 2 point wherever channel 1 points right now". At that moment channel 1 still points at the screen.
10. Only afterwards did `> wrong.txt` move channel 1 to the file. Channel 2 was already aimed at the screen and stayed there.
11. The rule to remember: redirections are applied left to right, and `2>&1` copies the current destination, not a promise to follow.
12. So `> file 2>&1` is right and `2>&1 > file` is almost always a mistake.

19.6.4 PLAIN - what is really happening inside

1. The three channels are numbers, not names. The number is a **file descriptor**: an index into a table the kernel keeps for your process.
2. Entry 0, 1 and 2 are filled in before your program starts. They are not special to the kernel. They are special only by agreement.
3. When you run a program from a terminal, all three entries point at the same pseudo-terminal device. That is why output and errors both land on screen.
4. When you write `> out.txt`, the shell does this between fork and exec:
5. It opens `out.txt` for writing, creating or emptying it. The kernel gives back the lowest free number, say 3.
6. It calls `dup2(3, 1)`. That copies entry 3 over entry 1. Entry 1 now points at the file.
7. It closes 3, which is no longer needed. Then it calls exec.
8. The new program starts with entry 1 already pointing at the file. It writes to 1 as usual, with no idea anything is different.
9. `2>&1` is just `dup2(1, 2)`. Copy whatever entry 1 currently holds into entry 2.
10. That is the entire mechanism. There is no magic and no cooperation from the program.
11. You can see the table on Linux. Every process has one under `/proc`:

```
$ ls -l /proc/self/fd
lr-x------ 0 -> /dev/null
l-wx------ 1 -> pipe:[159625]
l-wx------ 2 -> /dev/null
lr-x------ 3 -> /proc/10189/fd
```

12. In that real capture, the process was reading nothing, writing into a pipe, and throwing errors away.

19.6.5 TECHNICAL - the engineer's version

Syntax	Meaning	Underlying call
`> f`	stdout to f, truncate	open O_TRUNC, dup2
`>> f`	stdout to f, append	open O_APPEND
`< f`	stdin from f	open O_RDONLY
`2> f`	stderr to f	dup2 to fd 2

Syntax	Meaning	Underlying call
`2>&1`	stderr to current stdout	dup2(1, 2)
`&> f`	both to f (bash, zsh)	two dup2 calls
`<<< str`	here-string as stdin	temp file or pipe
`<< EOF`	here-document as stdin	temp file or pipe
`2>/dev/null`	discard errors	open the null dev
`>&-`	close the descriptor	close(1)

1. &> and &>> are bash and zsh extensions. The portable POSIX spelling is `> file 2>&1`. Use the portable form in scripts with `#!/bin/sh`.
2. `|&` in bash is shorthand for `2>&1 |`. It is also not POSIX.
3. Buffering is a libc behaviour, not a kernel one, and it surprises people. The C standard library uses line buffering for stdout when it is a terminal and full buffering, typically 4096 or 8192 bytes, when it is a pipe or file.
4. stderr is unbuffered by default. That is why error messages can appear before the output that logically came first.
5. Fix it when it matters with `stdbuf -o0 -e0 command`, or `unbuffer` from the expect package, or in Python with `python3 -u`.
6. `/dev/null` is the discard device, major 1 minor 3 on Linux. Writes succeed and vanish, reads return end of file immediately.
7. `/dev/stdout`, `/dev/stderr` and `/dev/fd/N` exist on Linux and macOS and let you name descriptors as paths. Useful with tools that only accept a filename.
8. Descriptors above 2 are yours to use. `exec 3< file` opens a private read channel that survives across commands until `exec 3<&-` closes it.
9. Inspect a live process's descriptors with `lsof -p PID`, or on Linux read `/proc/PID/fd`. Real `lsof` output from this sandbox:

```
$ lsof -i -P -n | head -3
COMMAND  PID USER FD  TYPE NAME
claude   462 root 11u IPv4 TCP 192.0.2.2:39400->160.79.104.10:443
claude   462 root 13u IPv4 TCP 127.0.0.1:43279 (LISTEN)
```

10. The honest version: "everything is a file" is a slogan, not a fact. Descriptors point to file descriptions, which may be regular files, pipes, sockets, devices, epoll instances, timers or signal queues. They all support `read` and `write` but not equally: you cannot seek in a pipe, and `write` to a socket can partially succeed.

19.6.6 WORDS - remember these

1. File descriptor — the number a program uses to name an open channel — a small non-negative integer indexing the process file descriptor table.
2. stdin — where a program reads from — file descriptor 0, `FILE *stdin` in C.
3. stdout — where results go — file descriptor 1, line buffered to a terminal, block buffered otherwise.
4. stderr — where complaints go — file descriptor 2, unbuffered by default, kept separate so results stay clean.
5. Redirection — pointing a channel somewhere else — `open` plus `dup2` performed by the shell between fork and exec.
6. `/dev/null` — the bin — the null device, discards writes, returns EOF on read.
7. Here-document — inline text fed to a program as input — the `<< WORD` construct, terminated by WORD on its own line.

19.7 Pipes

19.7.1 PLAIN - in simple words

1. A pipe connects the output of one program straight to the input of the next.
2. You write it with the vertical bar: `a | b`.
3. Nothing is saved on disk. The data goes from one program to the other through memory.
4. Both programs run at the same time. The second does not wait for the first to finish.
5. This means you can process something enormous without ever holding all of it at once.
6. It also means each program can be small, do one job, and know nothing about the others.
7. That combination is generally considered the single best idea in UNIX.
8. Doug McIlroy proposed it at Bell Labs, and it was added to UNIX in 1973.
9. Before pipes, you saved to a temporary file and read it back. Everyone did it. Nobody enjoyed it.

19.7.2 PLAIN - a picture in your head

1. Think of a bucket chain at a fire, before fire engines existed.
2. Ten people stand in a line from the well to the fire.
3. The first person does not fill a thousand buckets and then start passing them. They pass each bucket as soon as it is full.
4. Water starts arriving at the fire within seconds, not hours.
5. Nobody in the chain knows where the water comes from or where it goes. Each person knows only "take from the left, give to the right".
6. If a person in the middle is slow, the people behind them naturally wait. Nobody needs to coordinate this. The line just backs up.
7. If the fire is put out, the person at the end walks away. The next person tries to hand over a bucket, finds nobody there, and stops too.
8. That last case is important, and it has a name in UNIX. We come back to it.

Where this comparison breaks: real people can hold one bucket, while a pipe holds a fixed amount of data, 65536 bytes on Linux by default. And the writer does not "see" that the reader has gone. It gets told, forcefully, by a signal.

19.7.3 PLAIN - a worked example

1. Here is a real log file, eight lines, from this sandbox. We will find which client caused the most server errors.

```
10.0.0.7  ... "GET /index.html HTTP/1.1" 200 5120
10.0.0.9  ... "GET /style.css HTTP/1.1" 200 812
10.0.0.7  ... "GET /missing HTTP/1.1" 404 152
10.0.0.12 ... "POST /api/login HTTP/1.1" 500 90
10.0.0.7  ... "GET /api/user HTTP/1.1" 500 90
10.0.0.9  ... "GET /index.html HTTP/1.1" 200 5120
10.0.0.12 ... "GET /api/user HTTP/1.1" 500 90
10.0.0.3  ... "GET /index.html HTTP/1.1" 200 5120
```

2. Stage 1, keep only the server errors. Real output:

```
$ grep " 500 " access.log
10.0.0.12 ... "POST /api/login HTTP/1.1" 500 90
10.0.0.7  ... "GET /api/user HTTP/1.1" 500 90
10.0.0.12 ... "GET /api/user HTTP/1.1" 500 90
```

3. Stage 2, keep only the first field, which is the client address:

```
$ grep " 500 " access.log | cut -d' ' -f1
10.0.0.12
```

```
10.0.0.7
10.0.0.12
```

4. Stage 3, sort, so identical lines sit together:

```
$ ... | sort
10.0.0.12
10.0.0.12
10.0.0.7
```

5. Stage 4, collapse the runs and count them:

```
$ ... | uniq -c
      2 10.0.0.12
      1 10.0.0.7
```

6. Stage 5, sort by that count, largest first:

```
$ ... | sort -rn
      2 10.0.0.12
      1 10.0.0.7
```

7. Stage 6, keep the top two:

```
$ grep " 500 " access.log | cut -d' ' -f1 | sort | uniq -c \
  | sort -rn | head -2
      2 10.0.0.12
      1 10.0.0.7
```

8. Six programs. None of them knows about logs. Each does one small thing.
9. `uniq -c` only collapses adjacent duplicates, which is exactly why `sort` must come before it. This trips up everyone once.
10. The same pipeline works unchanged on a log of eight lines or eight hundred million, and uses the same tiny amount of memory either way.

19.7.4 PLAIN - what is really happening inside

1. When the shell sees `a | b`, it asks the kernel for a pipe.
2. The kernel creates a small buffer in memory and gives back two file descriptors: one you can read from, one you can write to.
3. The shell forks twice, once for `a` and once for `b`.
4. In the child that will run `a`, it points descriptor 1 at the write end.
5. In the child that will run `b`, it points descriptor 0 at the read end.
6. Both children close the ends they do not need. This matters enormously.
7. Then both children exec. Both are now running at the same time.
8. `a` writes as usual to descriptor 1, thinking it is writing to the screen.
9. `b` reads as usual from descriptor 0, thinking it is reading a keyboard.
10. When the buffer is full, `a`'s next write simply blocks. The kernel puts it to sleep until `b` takes some data out.
11. When the buffer is empty, `b`'s read blocks until `a` puts something in.
12. This automatic waiting is called back pressure, and it is why a pipeline cannot run out of memory no matter how much data flows through.
13. Here is real proof from this sandbox that a pipe streams rather than waits:

```
$ time (seq 1 50000000 | head -1)
1

real    0m0.002s
```

14. Producing fifty million numbers would take seconds. It finished in two thousandths of a second, because `head` stopped after one line and `seq` was killed.
15. Now the ending case. When `b` exits, the read end is closed.
16. The next time `a` writes, the kernel sees there is no reader left and sends `a` the signal SIGPIPE.
17. The default action of SIGPIPE is to kill the process silently. That is by design: it stops a producer from filling the world with data nobody wants.
18. This is why `yes | head -3` ends instead of running forever.

19.7.5 TECHNICAL – the engineer's version

1. The system call is `pipe(int fd[2])`, giving `fd[0]` for reading and `fd[1]` for writing. `pipe2` adds flags such as `O_CLOEXEC`.
2. On Linux the default pipe capacity is 65536 bytes, which is sixteen pages of 4096 bytes. Verified here:

```
$ python3 -c 'import os, fcntl
> r, w = os.pipe()
> print(fcntl.fcntl(w, 1032))'
65536
$ cat /proc/sys/fs/pipe-max-size
1048576
```

3. `PIPE_BUF`, which is 4096 on Linux and 512 minimum by POSIX, is a different number. It is the largest write guaranteed to be atomic when several writers share one pipe.
4. Capacity can be changed per pipe with `fcntl(fd, F_SETPIPE_SZ, n)` up to `/proc/sys/fs/pipe-max-size`. On macOS the pipe buffer starts at 16384 bytes and the kernel may grow it; there is no `F_SETPIPE_SZ`.
5. Closing unused ends is mandatory. If the shell left the write end open in the reader, the reader would never see end of file and the pipeline would hang forever. This is the classic pipe bug.
6. SIGPIPE is signal 13. The shell reports death by signal N as exit status 128 plus N, so a SIGPIPE death shows as 141.

```
$ yes | head -3 >/dev/null; echo "${PIPESTATUS[@]}"
141 0
$ seq 1 100000000 | head -2 >/dev/null; echo "${PIPESTATUS[@]}"
141 0
```

7. `PIPESTATUS` is a bash array holding every stage's status. zsh calls it `pipestatus`. The plain `$?` gives only the last stage, which is why a failing first stage is invisible by default.
8. `set -o pipefail` makes the pipeline's status the rightmost non-zero status. Verified:

```
$ set -o pipefail; yes | head -3 >/dev/null; echo $?
141
```

9. Servers and daemons usually call `signal(SIGPIPE, SIG_IGN)` and check for `EPIPE` from `write` instead, because a killed web server is worse than a dropped client.
10. Doug McIlroy proposed pipes at Bell Labs; Ken Thompson implemented them in Version 3 UNIX in 1973. The vertical bar notation and the `tee` command date from that period.
11. Named pipes, or FIFOs, are the same buffer with a filesystem name, created with `mkfifo`. Opening one for writing blocks until a reader appears, which is a real behaviour, demonstrated in this sandbox: a writer sat blocked for a full second until `cat` opened the other end.
12. Process substitution, `<(command)`, is bash and zsh only. It creates a FIFO or a `/dev/fd` entry and substitutes its path, letting you feed a command's output to a tool that demands a filename.

19.7.6 WORDS - remember these

1. Pipe — a direct connection from one program's output to another's input — an anonymous unidirectional kernel buffer with two file descriptors.
2. Back pressure — a fast producer waiting for a slow consumer — blocking of `write` when the pipe buffer is full.
3. SIGPIPE — the "nobody is listening" signal — signal 13, sent to a writer whose read end has closed, default action terminate.
4. PIPESTATUS — the list of results from every stage — a bash array of exit statuses for the last foreground pipeline.
5. FIFO — a pipe with a name on disk — a named pipe created by `mkfifo`, type `p` in `ls -l`.
6. `tee` — split the flow so you can see it and save it — a filter that copies stdin to stdout and to named files.

19.8 Exit codes

19.8.1 PLAIN - in simple words

1. Every program, when it finishes, hands back a single number.
2. That number is called the **exit code** or exit status.
3. It has nothing to do with what the program printed. It is a separate, private answer to one question: did this work?
4. Zero means success. Anything else means failure.
5. That feels backwards. It is not. There is one way to succeed and many ways to fail, so zero is the single success and the rest label the failure.
6. The number is small: 0 to 255 only. It cannot carry a message, only a code.
7. You see the last one with `echo $?`.
8. The shell uses it to decide whether to run the next command when you chain commands together.
9. Automated build systems test only this number. Not your output. Not your colours. The number.

19.8.2 PLAIN - a picture in your head

1. Think of a factory quality inspector at the end of a line.
2. The inspector does not describe the product. There is a label for that.
3. The inspector stamps one thing on the crate: a code.
4. Code 0 means "passed". Any other code means "rejected", and different codes say which test it failed.
5. The next station on the line reads only the stamp. It never opens the crate.
6. If the stamp is 0, the crate moves on. If not, the line stops and an alarm sounds.
7. A crate can contain a beautiful description of a disaster and still be stamped 0, if the inspector was careless. Then the disaster moves down the line unnoticed.

Where this comparison breaks: the stamp is applied by the program itself, and a badly written program can stamp 0 on anything. This happens constantly and is the single most common cause of a build system reporting success on a broken build. The check is only as honest as the program doing the checking.

19.8.3 PLAIN - a worked example

1. Real runs from this sandbox. Each shows the command, the message, and the code.

```
$ ls /nonexistent
ls: cannot access '/nonexistent': No such file or directory
$ echo $?
2

$ true;  echo $?      -> 0
$ false; echo $?      -> 1
```

```
$ grep -q zzz /etc/hostname; echo $?
1

$ bash -c 'exit 42'; echo $?
42

$ /etc/hostname; echo $?      # a file that is not executable
126

$ nosuchcommand123; echo $?
127

$ bash -c 'kill -INT $$'; echo $?
130

$ bash -c 'kill -TERM $$'; echo $?
143
```

2. Read the meanings. `grep` returning 1 is not an error. It means "I searched correctly and found nothing".
3. That distinction is why `grep -q pattern file && do_something` works cleanly. It is also why `grep` returning 2 means a real error, like an unreadable file.
4. 126 means "found it, but could not run it". Usually a missing execute permission.
5. 127 means "could not find it at all". If you see 127 in a build log, look at `PATH` first.
6. 130 is 128 plus 2, and signal 2 is SIGINT. Somebody pressed Ctrl-C.
7. 143 is 128 plus 15, and signal 15 is SIGTERM. Something asked it to stop, usually a timeout or an orchestrator.
8. 137 is 128 plus 9, SIGKILL. In a container that almost always means the memory limit was hit and the kernel killed it.
9. Now chaining. Real output:

```
$ true  && echo "ran because true"
ran because true
$ false && echo "not printed"
$ false || echo "ran because false"
ran because false
$ true  || echo "not printed"
$ false ; echo "semicolon always runs"
semicolon always runs
```

10. `A && B` runs B only if A succeeded. `A || B` runs B only if A failed.
11. `A ; B` runs B either way. This is the one that hides failures.

19.8.4 PLAIN - what is really happening inside

1. When a program calls `exit(3)`, the number 3 goes to the kernel.
2. The process dies, but a small record stays behind holding that number.
3. The parent calls `wait`, collects the record, and the record is freed. A process whose record has not yet been collected is a **zombie**.
4. The kernel packs more than the exit code into that record. It also stores whether the process was killed by a signal, and which one.
5. The shell unpacks it. If the process exited normally, `$?` is the code. If it was killed by signal N, the shell reports 128 plus N.
6. That 128 rule is a shell convention, not a kernel fact. The kernel keeps the two cases genuinely separate.
7. Why the limit of 255: the exit status field is eight bits wide in the traditional `wait` encoding.
8. So `exit 256` becomes 0 and `exit -1` becomes 255. Both are silent traps.

9. In a pipeline, `$?` reports only the last command. The others are lost unless you ask for them.
10. `set -e` makes the shell exit as soon as any command returns non-zero. It is how you stop a script from carrying on after a disaster.

19.8.5 TECHNICAL - the engineer's version

Code	Meaning	Typical source
0	success	everything
1	general failure	most tools
2	misuse or real error	grep, ls, bash
64-78	sysexits.h categories	BSD-style tools
126	found but not executable	shell
127	command not found	shell
128+N	killed by signal N	shell reporting
130	Ctrl-C, SIGINT	interactive use
137	SIGKILL, often out of memory	containers
143	SIGTERM	timeouts, systemd
255	out of range or -1	buggy exit calls

1. The kernel encoding is defined by POSIX macros: `WIFEXITED`, `WEXITSTATUS`, `WIFSIGNALED`, `WTERMSIG`, `WCOREDUMP`, `WIFSTOPPED`.
2. `WEXITSTATUS` is only valid when `WIFEXITED` is true. Reading it otherwise returns rubbish, and this is a real bug in real code.
3. The 64 to 78 range comes from `sysexits.h`, introduced in 4.3BSD. `EX_USAGE` is 64, `EX_DATAERR` 65, `EX_NOINPUT` 66, `EX_UNAVAILABLE` 69, `EX_SOFTWARE` 70, `EX_CONFIG` 78. It is a convention, widely ignored, but useful if you follow it consistently within one project.
4. `grep` documents its own contract in its manual page, and the exact wording from the real page in this sandbox is worth reading:

```
EXIT STATUS
       Normally the exit status is 0 if a line is selected, 1 if
       no lines were selected, and 2 if an error occurred.
```

5. `diff` uses 0 for identical, 1 for different, 2 for trouble. `curl` has about 100 documented codes; 6 is "could not resolve host", 7 "failed to connect", 28 "operation timed out", 56 "failure receiving data". A real failure captured here returned 56.
6. Continuous integration works exactly like this. A GitHub Actions step fails when the process exits non-zero. Nothing else is inspected.
7. Consequences that bite in practice. A step that ends with a pipeline reports only the last stage, so `run_tests | tee log.txt` reports the status of `tee`, which basically always succeeds.
8. The fix is `set -o pipefail`, or checking `PIPESTATUS`, or restructuring so the important command is last.
9. `set -e` has genuine sharp edges. It does not trigger inside a condition, inside && or || chains except the last element, or inside a command whose status is being tested. Experts disagree about whether it is safe.
10. Both sides of that argument: the "always use it" camp says most scripts are short and unhandled failure is worse than surprising exits. The "never use it" camp says the exceptions are too subtle to remember and explicit `if ! cmd; then` checks are honest. The compromise most teams settle on is `set -euo pipefail` plus explicit handling around known-noisy commands.

19.8.6 WORDS - remember these

1. Exit code — the single number a program leaves behind — the 8-bit exit status collected by `wait`.
2. `$?` — the shell variable holding the last one — expands to the exit status of the most recently completed foreground command.
3. Zombie — a finished process whose result nobody collected — a process in state `Z`, holding only its exit record.
4. `&&` — run the next one only if this worked — the AND list operator, short circuits on non-zero status.
5. `||` — run the next one only if this failed — the OR list operator, short circuits on zero status.
6. `set -e` — stop the script at the first failure — `errexit`, which exits on any untested non-zero status.
7. sysexits — an agreed table of failure categories — the 64 to 78 range from the BSD `sysexits.h` header.

19.9 Environment variables

19.9.1 PLAIN - in simple words

1. Every running program carries a small list of name and value pairs.
2. That list is the **environment**. It is just text, copied into the program when it starts.
3. `HOME` says where your home directory is. `PATH` says where to look for commands. `LANG` says which language and character set to use.
4. When a program starts another program, the child gets a copy of the list.
5. It is a copy, not a link. The child can change its own copy and the parent never notices.
6. There is no way for a child to change its parent's environment. This is a hard rule, and it explains many confusing situations.
7. Inside the shell you can also have plain variables that are not in the environment. They exist only in that shell.
8. To move one into the environment you `export` it. That is the whole difference.
9. Where these get set is a separate mess, and it is why your `PATH` can be right in one window and wrong in another.

19.9.2 PLAIN - a picture in your head

1. Think of a person leaving home for work with a small notebook in their pocket.
2. The notebook lists their address, their language, and the list of shops they are allowed to visit.
3. When they hire an assistant, they photocopy the notebook and hand the copy over.
4. The assistant can scribble in their copy. The original is untouched.
5. When the assistant hires their own assistant, another photocopy is made, including any scribbles.
6. So changes flow downwards only, and only to people hired after the change.
7. If you edit the master notebook at home, everybody already out working still has the old copy.
8. That is exactly why changing a setting file does not affect terminal windows you already have open.

Where this comparison breaks: a notebook can hold anything, but the environment holds only text, with a size limit, and no structure. There are no lists, no numbers and no nesting. Everything is a string, and any structure you think you see, like the colons in `PATH`, is a convention agreed by the programs reading it.

19.9.3 PLAIN - a worked example

1. Real run showing the difference between a shell variable and an exported one:

```
$ MYVAR=hello
$ echo "in this shell: $MYVAR"
in this shell: hello
$ bash -c 'echo child sees: [$MYVAR]'
child sees: []
$ export MYVAR
```

```
$ bash -c 'echo child sees: [$MYVAR]'
child sees: [hello]
```

2. Before `export`, the child saw nothing. After `export`, it saw the value. Nothing else changed.
3. Now `PATH`. Here is the real one from this sandbox, shortened:

```
$ echo $PATH
/home/claude/.npm-global/bin:/root/.local/bin:/root/.cargo/bin:
/usr/local/go/bin:/opt/node22/bin:/usr/local/sbin:/usr/local/bin:
/usr/sbin:/usr/bin:/sbin:/bin
```

4. It is one string. Directories separated by colons. Searched strictly left to right, first match wins.
5. Find out which one wins:

```
$ which ls
/usr/bin/ls
$ type ls
ls is /usr/bin/ls
$ command -v grep
/usr/bin/grep
```

6. Order matters enormously. If `/usr/local/bin` comes before `/usr/bin` and both contain `python3`, you get the one in `/usr/local/bin`.
7. This is the whole explanation for "it works in my terminal but not in the cron job". Different `PATH`, different program.

19.9.4 PLAIN - what is really happening inside

1. The environment is passed to `execve` as an array of strings, each shaped `NAME=value`, ending with a null pointer.
2. The kernel copies that array onto the new program's stack. The C library makes it available as the global `environ`.
3. Nothing validates it. Any string with an `=` is accepted.
4. When you type a command with no slash in it, the shell splits `PATH` on colons and tries each directory in turn.
5. For each, it builds a full path and tries to execute it. First success wins. If none work, you get 127.
6. An empty entry in `PATH`, such as a leading colon or two colons together, means the current directory. That is a security hazard and should be avoided.
7. The current directory is not in `PATH` by default on any sane system. That is deliberate.
8. If it were, someone could leave a file called `ls` in a shared directory, and you would run it by accident.
9. So to run a program in the directory you are standing in, you must say `./program`. The slash tells the shell not to search at all.
10. Searching `PATH` for every command would be slow, so shells remember. This is called **hashing**.

```
$ hash -r          # forget everything
$ ls > /dev/null
$ grep --version > /dev/null
$ hash
hits    command
   1    /usr/bin/grep
   1    /usr/bin/ls
```

11. The cache is why installing a new program sometimes does not take effect until you run `hash -r` or open a new shell. bash and zsh both do this.

19.9.5 TECHNICAL - the engineer's version

Variable	Holds	Typical value
PATH	command search list	/usr/local/bin:/usr/bin
HOME	your home directory	/Users/name, /home/name
PWD	current directory	maintained by the shell
OLDPWD	previous directory	used by `cd -`
SHELL	your login shell	/bin/zsh
USER	your login name	from the passwd entry
TERM	terminal capability name	xterm-256color
LANG	locale for everything	en_US.UTF-8
EDITOR	preferred text editor	vim, nano, code -w
TMPDIR	scratch directory	/tmp, or per-user
LD_LIBRARY_PATH	extra library dirs	avoid unless forced

1. `PWD` is maintained by the shell, not the kernel. The kernel truth is `getcwd`. They can disagree when symbolic links are involved, which is what `cd -P` and `pwd -P` exist to resolve.
2. `SHELL` records your login shell from the password database. It does not tell you which shell is currently running. To find that, check `$0`, or `$BASH_VERSION` and `$ZSH_VERSION`.
3. `LANG` and the `LC_*` family change program behaviour in ways people do not expect. Sort order is the classic case. Real measurement from this sandbox:

```
$ printf 'b\nA\na\nB\n' > letters.txt
$ LC_ALL=C sort letters.txt          -> A B a b
$ LC_ALL=en_US.UTF-8 sort letters.txt -> a A b B
```

4. That is the same data, the same command, and a different answer. Scripts that must be reproducible set `LC_ALL=C` explicitly.
5. Startup file order is the part everyone gets wrong. It depends on two independent questions: is this a login shell, and is it interactive.

Shell	Login shell reads	Interactive non-login
bash	/etc/profile, then the first of ~/.bash_profile ~/.bash_login, ~/.profile	~/.bashrc only
zsh	/etc/zprofile, ~/.zprofile then /etc/zshrc, ~/.zshrc, /etc/zlogin, ~/.zlogin	/etc/zshrc, ~/.zshrc
sh	/etc/profile, ~/.profile	ENV file if set

6. bash reads only one of `~/.bash_profile`, `~/.bash_login`, `~/.profile`, in that order, and stops at the first that exists. If you have both a `.bash_profile` and a `.profile`, the `.profile` is silently ignored.
7. zsh always reads `~/.zshrc` for interactive shells, login or not. That is why zsh setup is simpler and why macOS advice usually says "put it in `.zshrc`".
8. This explains the classic complaint. An SSH login is a login shell. A new tab in Terminal.app on macOS is also a login shell, by Apple's choice, which differs from most Linux terminal emulators where a new tab is an interactive non-login shell.
9. So a `PATH` line added to `~/.bashrc` on a Linux server works in new tabs but not over `ssh host command`, which is non-interactive and reads neither.
10. Real check of which files exist, from this sandbox:

```
$ for f in /etc/profile ~/.bash_profile ~/.bashrc \
>          ~/.profile ~/.zshrc
> do
>   [ -e $f ] && s=EXISTS || s=missing
>   printf '%-20s %s\n' "$f" "$s"
> done
/etc/profile         EXISTS
/root/.bash_profile  missing
/root/.bashrc        EXISTS
/root/.profile       EXISTS
/root/.zshrc         EXISTS
```

11. GUI applications on macOS do not read your shell startup files at all. They inherit the environment from `launchd`. This is why a program launched from the Dock cannot find a tool that works fine in Terminal.
12. Environment size is limited. On Linux the combined size of arguments and environment is capped by `MAX_ARG_STRLEN` at 128 KiB per string, and the total by the stack limit, typically a quarter of `ulimit -s`. Exceeding it gives `E2BIG`, seen as "Argument list too long".
13. Never put secrets in environment variables on a shared machine. On Linux `/proc/PID/environ` is readable by the owner, and process listings can leak command-line arguments to everyone.

19.9.6 WORDS - remember these

1. Environment — the list of settings a program inherits — a null-terminated array of `NAME=value` strings passed through `execve`.
2. Export — move a shell variable into the environment — mark it for inclusion in the child's environment on the next `exec`.
3. PATH — where the shell looks for commands — a colon-separated directory list, searched left to right, first match wins.
4. Hashing — remembering where a command was found — the shell's command location cache, cleared with `hash -r`.
5. Login shell — the shell started when you sign in — a shell whose argv[0] begins with `-`, reading the profile files.
6. Locale — language and formatting rules — the `LANG` and `LC_*` variables controlling collation, case, dates and messages.

19.10 Configuration on each system

19.10.1 PLAIN - in simple words

1. Every operating system needs somewhere to keep settings.
2. Windows keeps almost all of them in one giant database called the **registry**.
3. macOS and Linux keep them in ordinary files scattered around the disk.
4. The Windows way is one place, one format, one tool. It is fast to read and hard to inspect by eye.
5. The UNIX way is many places, many formats, and any text editor works.
6. On Linux, system-wide settings live in `/etc`, and your personal settings live in files in your home directory whose names begin with a dot.
7. A leading dot means "hidden" by convention. `ls` will not show it unless you ask with `-a`.
8. These are called **dotfiles**, and people keep them in version control and copy them between machines.
9. macOS does both. It has UNIX dotfiles and `/etc` underneath, and Apple's own settings system on top, using files called property lists.
10. That is why a Mac developer edits `~/.zshrc` for the shell and never thinks about a registry, even though macOS has a settings database of its own.

19.10.2 PLAIN – a picture in your head

1. Imagine two libraries.
2. The first library has one enormous card catalogue in the entrance hall. Every fact about every book is in a drawer somewhere in it.
3. Finding anything is fast, if you know the drawer. Browsing is impossible. The drawers are labelled in code.
4. If the catalogue is damaged, the whole library stops working, because no book can be located.
5. The second library writes each subject's notes on a sheet of paper and pins it to the shelf that subject lives on.
6. Finding a note means walking to the right shelf. Slower, but you can read it with your eyes, and you can photocopy one shelf's notes and carry them elsewhere.
7. If one sheet is torn, only that shelf is affected.
8. Library one is the Windows registry. Library two is `/etc` and dotfiles.

Where this comparison breaks: the registry is far better engineered than a card catalogue. It is transactional, it supports permissions per entry, and it can be changed by policy across ten thousand machines at once. The scattered-files approach has no transactions at all: a half-written config file is simply a broken config file.

19.10.3 PLAIN – a worked example

1. Say you want to change the shell prompt colour and the editor a tool opens.
2. On Linux or macOS, you edit one text file:

```
# in ~/.zshrc or ~/.bashrc
export EDITOR=vim
export PS1='%n@%m %1~ %# '
```

3. You then run `source ~/.zshrc` to apply it to the current shell, or open a new window.
4. You can email that file to a colleague. It is 20 lines of text.
5. On macOS, an application setting is different. To make Finder show hidden files:

```
defaults write com.apple.finder AppleShowAllFiles -bool true
killall Finder
```

6. That wrote into a property list file at `~/Library/Preferences/com.apple.finder.plist`.
7. On Windows, the same class of change is a registry edit. In `regedit` you would navigate to a path such as:

```
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion
    \Explorer\Advanced
```

8. Then set the value `Hidden` to the number 1.
9. Or from PowerShell, without the graphical tool:

```
Set-ItemProperty -Path 'HKCU:\Software\Microsoft\Windows\
  CurrentVersion\Explorer\Advanced' -Name Hidden -Value 1
```

10. Three systems, one idea, three completely different mechanisms.

19.10.4 PLAIN – what is really happening inside

1. The registry is a tree, like a filesystem, but stored in a few binary files.
2. The top-level branches are called **hives**. The two you meet are:
3. `HKEY_LOCAL_MACHINE`, shortened to HKLM. Settings for the whole computer. Changing it needs administrator rights.

4. `HKEY_CURRENT_USER`, shortened to HKCU. Settings for the person signed in right now. You can change your own.
5. Inside a hive are **keys**, which are like folders, and **values**, which are like files.
6. A value has a name, a type and data. Types include string, expandable string, 32-bit number, 64-bit number, binary blob and multi-string.
7. `regedit.exe` is the graphical browser. It is a plain tree view with the values on the right.
8. Why it exists: before it, Windows kept settings in hundreds of `.ini` text files. There was no locking, no types, no permissions and no network management.
9. Why it became a problem: it is a single shared namespace with no ownership rules. Any program can write anywhere it has rights to.
10. Uninstalling a program often leaves its keys behind. Over years the registry accumulates dead entries, and a whole industry of "registry cleaner" products grew up around this, most of them useless or harmful.
11. On macOS, the equivalent store is thousands of separate property list files, most in `~/Library/Preferences` and `/Library/Preferences`.
12. Each is named after a reverse domain identifier, such as `com.apple.finder.plist`, so applications cannot collide by accident.
13. They are usually stored in a compact binary format, not readable text. That is a storage choice, not a philosophy choice.
14. On Linux, there is no central store at all. Each program picks its own file and its own format, guided loosely by an agreed set of directories.

19.10.5 TECHNICAL - the engineer's version

System	Store	Tool to read
Windows	registry hives	regedit, reg.exe
macOS	plist files	defaults, plutil
Linux	/etc and dotfiles	any text editor
Modern	XDG directories	any text editor

1. Registry history. It first appeared in Windows 3.1, released 1992, as a single file `REG.DAT` limited to 64 KB, holding only OLE registration and file associations.
2. Windows NT 3.1 in 1993 and Windows 95 in 1995 expanded it into the full hive structure with `HKEY_LOCAL_MACHINE` and `HKEY_CURRENT_USER`, and moved registry access into kernel mode.
3. On modern Windows the hive files live in `C:\Windows\System32\config` for machine hives, and `NTUSER.DAT` in each user profile for HKCU. They are not editable as text.
4. The five root keys are HKLM, HKCU, HKCR (classes, a merged view), HKU (all loaded user hives) and HKCC (current hardware profile). Only HKLM and HKU are truly stored; the others are views.
5. Value types are named `REG_SZ`, `REG_EXPAND_SZ`, `REG_DWORD`, `REG_QWORD`, `REG_BINARY`, `REG_MULTI_SZ`.
6. Windows also has Group Policy, which writes into the registry under `Software\Policies` and is enforced centrally. That central-management ability is the strongest argument for the registry's existence.
7. macOS plists conform to Apple's property list format. The XML variant has a DOCTYPE from Apple; the binary variant starts with the eight bytes `bplist00`.
8. `defaults read com.apple.finder` prints a domain. `defaults write` sets a value. `plutil -convert xml1 file.plist` makes a binary file readable, and `plutil -lint` validates one.
9. There is a caching daemon, `cfprefsd`. Editing a plist file by hand while an application is running can be overwritten by the cache. Use `defaults` or quit the application first. This is a real and frequent trap.
10. Linux and modern macOS command-line tools increasingly follow the XDG Base Directory Specification, version 0.8, published 8 May 2021 by freedesktop.org.

XDG variable	Default	For
XDG_CONFIG_HOME	$HOME/.config	settings
XDG_DATA_HOME	$HOME/.local/share	app data
XDG_STATE_HOME	$HOME/.local/state	logs, history
XDG_CACHE_HOME	$HOME/.cache	rebuildable data
XDG_CONFIG_DIRS	/etc/xdg	system settings
XDG_DATA_DIRS	/usr/local/share/	system data

11. The point of XDG is that a home directory used to fill with dozens of top-level dotfiles. Now well-behaved tools use `~/.config/toolname/`.
12. Adoption is partial. `git` supports `~/.config/git/config`. `ssh` still insists on `~/.ssh`. bash still uses `~/.bashrc`. This is a convention with incomplete uptake, not a standard anyone enforces.
13. System configuration on Linux lives in `/etc`, a hierarchy defined loosely by the Filesystem Hierarchy Standard, current version 3.0 from 2015. Examples: `/etc/passwd`, `/etc/hosts`, `/etc/ssh/sshd_config`, `/etc/fstab`, `/etc/systemd/system`.
14. Why a macOS or Linux developer never thinks about a registry: everything they touch daily, the shell, the editor, the compiler, git, ssh, docker, reads a text file at a documented path. Nothing they use has a central database, so the concept never comes up.
15. The honest version: macOS does have a registry-like system, and its problems are the same as Windows'. Stale preference domains accumulate, `cfprefsd` caching causes lost edits, and there is no reliable uninstall. Developers avoid it only because their tools are UNIX tools, not Mac applications.

19.10.6 WORDS - remember these

1. Registry — Windows' one big settings database — a hierarchical binary store of keys and typed values, held in hive files.
2. Hive — a top-level branch of the registry — a discrete file-backed subtree such as HKLM or a user's NTUSER.DAT.
3. HKLM and HKCU — machine settings and my settings — HKEY_LOCAL_MACHINE requires elevation, HKEY_CURRENT_USER does not.
4. plist — a macOS settings file — an Apple property list, XML or binary, keyed by reverse-domain identifier.
5. Dotfile — a hidden settings file in your home directory — a file whose name starts with `.`, omitted by `ls` unless `-a` is given.
6. XDG base directories — the agreed places for config, data and cache — the freedesktop.org specification defining `XDG_CONFIG_HOME` and friends.
7. `/etc` — where system settings live on UNIX — the machine-wide configuration hierarchy described by the Filesystem Hierarchy Standard.

19.11 The essential commands

19.11.1 PLAIN - in simple words

1. There are thousands of commands. About forty carry almost all daily work.
2. They fall into groups, and learning them by group is far easier than learning them alphabetically.
3. Navigation: where am I, what is here, move somewhere else.
4. Files: create, copy, move, delete.
5. Viewing: show me the contents, the top, the bottom, one page at a time.
6. Searching: find files by name, find text inside files.
7. Text processing: cut columns, replace text, sort, count.
8. Permissions: who is allowed to do what.
9. Processes: what is running, stop it.

10. Disk: how much space, what is using it.
11. Network: is it reachable, fetch it, log in to it.
12. Archives: bundle a folder into one file, unpack it again.
13. Help: what does this command do, where does it live, what did I type before.

19.11.2 PLAIN – a picture in your head

1. Think of a workshop with a pegboard of hand tools.
2. You do not learn a pegboard by reading every label. You learn it by doing three or four jobs.
3. The saw, the hammer and the tape measure get used in every job. Those are `ls`, `cd` and `cat`.
4. Some tools look alike but are for different materials. `grep` searches inside files; `find` searches for files. Beginners reach for the wrong one constantly.
5. Some tools are power tools with a manual. `sed` and `awk` are those. You will use ten percent of them forever, and that is fine.
6. Every tool has flags, which are like the settings on a drill. Two or three settings per tool are worth memorizing. The rest you look up.

Where this comparison breaks: hand tools are shaped so you cannot use them wrongly. Command flags are one letter long and unforgiving. `rm -rf /` and `rm -rf ./` differ by one character and by everything else.

19.11.3 PLAIN – a worked example

1. Navigation and files, run for real in this sandbox:

```
$ pwd
/tmp/shdemo
$ ls -l
-rw-r--r-- 1 root root 610 Aug 13 01:55 access.log
prw-r--r-- 1 root root   0 Aug 13 01:55 f
-rw-r--r-- 1 root root   0 Aug 13 01:56 note1.txt
```

2. Read a long listing left to right. The first character is the type: `-` for a regular file, `d` for a directory, `l` for a symbolic link, `p` for a named pipe, which is what that `f` is.
3. The next nine characters are permissions in three groups of three: owner, group, everyone else. `r` read, `w` write, `x` execute.
4. Then the link count, the owner, the group, the size in bytes, the modification time and the name.
5. Permissions changing for real:

```
$ echo 'echo hi' > s.sh
$ ls -l s.sh
-rw-r--r-- 1 root root 8 Aug 13 01:56 s.sh
$ chmod +x s.sh
$ ls -l s.sh
-rwxr-xr-x 1 root root 8 Aug 13 01:56 s.sh
$ chmod 640 s.sh
$ stat -c '%a %A %n' s.sh
640 -rw-r----- s.sh
```

6. The number form is three octal digits. Read 4 for read, 2 for write, 1 for execute, added together. So 6 is read plus write, 4 is read only, 0 is nothing. 640 means owner read and write, group read, others nothing.
7. Archives for real:

```
$ tar -czf proj.tar.gz proj
$ tar -tzf proj.tar.gz
proj/
```

```
proj/src/
proj/src/main.py
proj/README.md
$ tar -xzf proj.tar.gz -C out
```

8. Remember the letters: `c` create, `t` list, `x` extract, `z` gzip, `f` the filename follows, `v` be chatty. Always put `f` last, because the filename comes straight after it.
9. Networking for real, using the reader's own recorded session on their home connection in India:

```
$ dig +short github.com
20.207.73.82
$ curl -v https://github.com
* Trying 20.207.73.82:443...
  ... times out after 15 seconds, no response at all
```

10. That address is in a Microsoft-owned range, because GitHub is owned by Microsoft, acquired 2018. The name resolved fine. The connection did not complete. Those are two different failures and the commands separate them.
11. A trace from the same session showed the path leaving the home router at `192.168.0.1`, crossing private ISP addresses, reaching Microsoft's network at hop 7, and then falling silent.

19.11.4 PLAIN - what is really happening inside

1. Almost every one of these commands is a small program in `/usr/bin` or `/bin`. The shell finds it in `PATH` and runs it.
2. Most read from standard input when given no filename, which is why they all work in pipes without any special support.
3. Most write results to standard output and complaints to standard error, so redirection works uniformly.
4. Most return 0 on success and non-zero on failure, so && chains work.
5. These four properties are the entire reason the toolkit composes. Nothing else is shared between them.
6. `cd` is the exception. It is a shell builtin, because changing directory must affect the shell itself.
7. `ls` sorts alphabetically because it chooses to, not because directories are sorted. On disk, directory entries are in whatever order the filesystem likes.
8. `rm` does not erase data. It removes a name. If another name points at the same data, the data survives. Only when the last name goes and no process has the file open is the space freed.
9. `mv` within one filesystem does not move data either. It just changes which directory holds the name. Across filesystems it must copy and then delete, which is why it is slow there.
10. `kill` does not necessarily kill. It sends a signal. The default is SIGTERM, which is a polite request the program may handle or ignore. `kill -9` sends SIGKILL, which the program cannot refuse.

19.11.5 TECHNICAL - the engineer's version

1. Navigation and inspection.

Command	Plain job	Flags worth knowing
pwd	print current path	-P resolve symlinks
cd	change directory	- go back, no flag
ls	list directory	-l -a -h -t -r -S
tree	show nested structure	-L depth, -a

2. Real note: `ls -lt` sorts newest first, `ls -ltr` reverses so newest is at the bottom, which is what you want on a long log directory.
3. Files.

Command	Plain job	Flags worth knowing
mkdir	make a directory	-p make parents
touch	create or update time	-t set a time
cp	copy	-r -a -i -v
mv	move or rename	-i -n -v
rm	delete	-r -f -i
ln	make another name	-s symbolic link

4. `cp -a` means archive: recursive, preserve permissions, times and links. `rm -i` asks before each delete and is worth aliasing on shared machines.
5. There is no undelete. `rm` is final. The only protection is backups and care.
6. Viewing.

Command	Plain job	Flags worth knowing
cat	print whole file	-n number lines
less	page through a file	-N -S, / to search
head	first lines	-n N, -c bytes
tail	last lines	-n N, -f follow
wc	count lines and words	-l -w -c

7. `tail -f logfile` is the standard way to watch a log grow. `tail -F` also survives the file being rotated and recreated.
8. Inside `less`: `/text` searches forward, `n` next match, `G` end, `g` start, `q` quit. It is also the default pager for `man`.
9. Searching.

Command	Plain job	Flags worth knowing
grep	find text in files	-i -n -r -v -c -E
find	find files by attribute	-name -type -mtime
which	show which file will run	-a show all matches
locate	fast name search	needs an index

10. Real `grep` and `find` runs from this sandbox:

```
$ grep -c "500" access.log
3
$ grep -n "404" access.log
3:10.0.0.7 ... "GET /missing HTTP/1.1" 404 152
$ grep -v "200" access.log | wc -l
4
$ find /home/claude/book/chapters -name 'ch1*.md' -size +100k \
    -printf '%s %p\n' | sort -rn | head -3
131233 /home/claude/book/chapters/ch14.md
124198 /home/claude/book/chapters/ch17.md
123297 /home/claude/book/chapters/ch16.md
```

11. `find` syntax is unusual: path first, then tests, then actions. `-exec cmd {} \;` runs once per file; `-exec cmd {} +` batches them and is much faster. `-print0` with `xargs -0` handles filenames with spaces.
12. Text processing.

Command	Plain job	Flags worth knowing
cut	take columns	-d delim, -f fields
sort	order lines	-n -r -u -k -t
uniq	collapse adjacent dups	-c -d -u
tr	swap or delete chars	-d delete, -s squash
sed	edit a stream	-n, s///, -i
awk	field-aware processing	-F, patterns, END

13. Real runs:

```
$ cut -d: -f1,7 /etc/passwd | head -3
root:/bin/bash
daemon:/usr/sbin/nologin
bin:/usr/sbin/nologin
$ printf '10\n9\n100\n2\n' | sort      -> 10 100 2 9
$ printf '10\n9\n100\n2\n' | sort -n   -> 2 9 10 100
$ echo "Hello World" | tr 'a-z' 'A-Z'
HELLO WORLD
$ sed 's/GET/FETCH/' access.log | head -1
10.0.0.7 ... "FETCH /index.html HTTP/1.1" 200 5120
$ awk '{bytes += $10} END {print "total:", bytes, "lines:", NR}' \
    access.log
total: 16594 lines: 8
$ awk '$9 == 500 {print $1}' access.log
10.0.0.12
10.0.0.7
10.0.0.12
```

14. Note `sort` without `-n` puts 100 before 2, because it is comparing text. That single default has produced more wrong reports than any other flag in UNIX.
15. `sed -i` edits in place. GNU sed takes `-i` alone; BSD and macOS sed require an argument, so `sed -i '' 's/a/b/' f` on macOS and `sed -i 's/a/b/' f` on Linux. This difference breaks scripts crossing the two systems constantly.
16. Permissions and ownership.

Command	Plain job	Flags worth knowing
chmod	change permissions	-R, +x, octal 644
chown	change owner and group	-R, user:group
umask	default for new files	022 typical
id	who am I, which groups	-u -g -G
sudo	run as another user	-u, -i, -l

17. Real: `umask` printed `0022` here, meaning new files get 644 and new directories 755, because the mask removes write from group and others.
18. `chown` needs root for changing the owner. Changing only the group is allowed if you belong to the target group.
19. Processes.

Command	Plain job	Flags worth knowing
ps	snapshot of processes	aux, -ef, -o fmt
top	live process view	-b -n batch mode

Command	Plain job	Flags worth knowing
kill	send a signal	-TERM -KILL -HUP
pkill	signal by name	-f match full line
jobs	this shell's jobs	%1 refers to job 1
nohup	survive terminal close	with &

20. Real, from this sandbox:

```
$ ps aux --sort=-%mem | head -3
USER  PID  %CPU %MEM     VSZ    RSS STAT  TIME COMMAND
root  462   3.8  9.0 6045508 746736 Rsl   3:46 claude
root  473   0.3  0.4 1800580  33860 Sl    0:21 environment-manager
$ ps -o pid,ppid,stat,etime,comm -p 1
  PID  PPID STAT     ELAPSED COMMAND
    1     0 SLl     01:38:18 process_api
```

21. `ps aux` is BSD-style, `ps -ef` is System V style. Both work on Linux and macOS. `RSS` is resident memory in kilobytes, the number that actually matters. `VSZ` is address space reserved and is usually meaningless.
22. Escalation order for stopping something: `kill PID` first, wait a few seconds, then `kill -9 PID`. Going straight to `-9` skips the program's chance to flush data and clean up.
23. Disk.

Command	Plain job	Flags worth knowing
df	free space per mount	-h -i inodes
du	space used by a tree	-sh, -d1, -h
ncdu	interactive disk usage	not always installed

24. Real:

```
$ df -h | head -4
Filesystem      Size  Used Avail Use% Mounted on
/dev/vda        252G   12G   30G  29% /
/dev/vdc        327M  295M   26M  93% /opt/claude-code
$ du -sh /home/claude/book/chapters
1.6M    /home/claude/book/chapters
```

25. If `df` shows space free but writes still fail, check `df -i`. You may have run out of inodes, which are the records that name files, not the bytes.
26. Network.

Command	Plain job	Flags worth knowing
curl	fetch a URL	-I -L -o -s -v
wget	download a file	-c continue, -O
ssh	log in to another host	-p port, -i key, -v
scp	copy files over ssh	-r, -P port
ping	is the host answering	-c count
dig	ask DNS a question	+short, @server
traceroute	show the path there	-m max hops
netstat	sockets and listeners	-tlnp
ss	modern netstat	-tlnp
lsof	what has this file open	-i, -p PID

27. Real runs from this sandbox:

```
$ dig +short github.com
140.82.112.4
$ dig @1.1.1.1 +short pypi.org
151.101.64.223
151.101.192.223
$ curl -s -o /dev/null \
   -w 'code=%{http_code} time=%{time_total}s ip=%{remote_ip}\n' \
   pypi.org
code=301 time=0.039756s ip=151.101.192.223
$ netstat -tlnp | head -3
Proto Local Address       State   PID/Program name
tcp   127.0.0.1:43279     LISTEN  462/claude
tcp   0.0.0.0:2024        LISTEN  -
$ ssh -V
OpenSSH_9.6p1 Ubuntu-3ubuntu13.18, OpenSSL 3.0.13 30 Jan 2024
```

28. `netstat` is deprecated on Linux in favour of `ss` from the iproute2 package, but it is still present on macOS and on many servers. Learn both spellings of the same idea.
29. `ping` needs ICMP to be allowed. Many cloud hosts drop ICMP by policy, so a failed ping proves nothing on its own. Prove reachability with a TCP connection instead: `curl -v` or `nc -vz host port`.
30. Archives and transfer.

Command	Plain job	Flags worth knowing
tar	bundle a directory	-czf -xzf -tzf
gzip	compress one file	-d decompress, -9
zip	Windows-friendly bundle	-r recursive
unzip	unpack a zip	-l list, -d dir
rsync	sync trees efficiently	-av –delete -n

31. `tar` bundles then compresses the whole bundle, so it compresses better but cannot extract one file without reading the stream. `zip` compresses each file separately, so it is worse at compression but supports random access. Real measurement here: the same tiny tree gave 213 bytes as `.tar.gz` and 640 bytes as `.zip`.
32. Help and history.

Command	Plain job	Flags worth knowing
man	read the manual	-k search, section
which	path of the command	-a all matches
type	what kind of thing is it	-a all definitions
history	what did I type before	Ctrl-R to search
apropos	search manual summaries	same as man -k

33. Real history behaviour, captured in a pseudo-terminal here:

```
demo$ history
    1  echo alpha
    2  ls note1.txt
    3  pwd
    4  history
demo$ !2
ls note1.txt
note1.txt
```

34. `!!` repeats the previous command. `sudo !!` is the standard recovery after forgetting `sudo`. `!$` is the last argument of the previous line.

19.11.6 WORDS - remember these

1. Flag — a short option that changes a command's behaviour — a single-letter or long option parsed by `getopt`.
2. Recursive — apply to everything underneath — the `-r` or `-R` option, descending the directory tree.
3. Inode — the record that describes a file — the filesystem structure holding permissions, times and block pointers, not the name.
4. Signal — a message sent to a running process — an asynchronous notification such as SIGTERM 15 or SIGKILL 9.
5. Pager — a program that shows text one screen at a time — `less` or `more`, the target of the `PAGER` variable.
6. Symbolic link — a file that points at another path — created with `ln -s`, shown as type `l` by `ls -l`.

19.12 Globbing, quoting and expansion

19.12.1 PLAIN - in simple words

1. The shell rewrites your line before any program sees it.
2. By the time `ls *.txt` reaches `ls`, the star is gone. `ls` receives a list of real filenames.
3. This surprises people. `ls` has no wildcard support at all. It never needed any.
4. The rewriting has several kinds, and they happen in a fixed order.
5. ~ becomes your home directory.
6. `{a,b}` becomes two separate words.
7. `$NAME` becomes the value of a variable.
8. `$(command)` becomes whatever that command printed.
9. `*` and `?` become matching filenames.
10. Quotes switch parts of this off. That is their only job.

19.12.2 PLAIN - a picture in your head

1. Think of a mail room that opens every letter before delivery.
2. Wherever the letter says "the boss", the clerk writes in the boss's actual name. Wherever it says "all the branches", the clerk writes out every branch address in full.
3. The recipient never sees the shorthand. They see a fully spelled-out letter.
4. Quotation marks are an instruction to the clerk: leave this part exactly as written.
5. Single quotes mean "change nothing at all inside here".
6. Double quotes mean "you may still fill in names, but do not split the text into pieces or expand wildcards".

Where this comparison breaks: the clerk works in a strict order and cannot go back. Once a variable has been replaced by text containing a space, the shell may split that text into two words, and there is no way to un-split it afterwards. That one-way order is the cause of nearly every quoting bug.

19.12.3 PLAIN - a worked example

1. All real output from this sandbox. The directory holds `note1.txt`, `note2.txt`, `notes.md`, `report.csv` and `my file.txt`.

```
$ echo *.txt
my file.txt note1.txt note2.txt
$ echo note?.txt
note1.txt note2.txt
$ echo note[12].txt
note1.txt note2.txt
```

```
$ echo nomatch*.zzz
nomatch*.zzz
$ echo file{1..4}.log
file1.log file2.log file3.log file4.log
$ echo {a,b,c}.txt
a.txt b.txt c.txt
$ echo ~
/root
$ echo "there are $(ls | wc -l) entries"
there are 7 entries
$ echo $((3 + 4 * 2))
11
```

2. Note the fourth one. When a wildcard matches nothing, bash leaves it unchanged. The program then receives a literal star and usually complains. zsh instead reports an error and runs nothing, which is arguably safer.
3. Note that brace expansion produced filenames that do not exist. Braces are pure text generation; they never look at the disk.
4. Quoting, for real:

```
$ NAME=world
$ echo "double: $NAME"
double: world
$ echo 'single: $NAME'
single: $NAME
$ echo backslash: \$NAME
backslash: $NAME
```

5. Now the spaces problem, which is the one that destroys data:

```
$ f="my file.txt"
$ printf "[%s]\n" $f
[my]
[file.txt]
$ printf "[%s]\n" "$f"
[my file.txt]
$ rm $f
rm: cannot remove 'my': No such file or directory
rm: cannot remove 'file.txt': No such file or directory
```

6. Unquoted, one filename became two arguments. With `rm` in a directory that happened to contain a file called `my`, that command would have deleted the wrong file silently.
7. The rule is simple and absolute: put double quotes around every variable expansion unless you have a specific reason not to.

19.12.4 PLAIN - what is really happening inside

1. You can watch the rewriting happen. `set -x` makes bash print each line after expansion. Real output:

```
$ bash -xc 'x=5; ls note*.txt
>           echo "x is $x, dir $(basename /tmp/shdemo)"'
+ x=5
+ ls note1.txt note2.txt
++ basename /tmp/shdemo
+ echo 'x is 5, dir shdemo'
```

2. Look at line two. `ls note*.txt` became `ls note1.txt note2.txt` before running. The wildcard is gone.

3. Look at the double plus. That is a nested expansion: the command substitution ran first, at one level deeper.
4. The order the shell uses is fixed and worth memorizing:
5. Brace expansion first. Purely textual, no filesystem involved.
6. Then tilde expansion.
7. Then, all at the same time and left to right: parameter expansion (`$VAR`), command substitution (`$(...)`) and arithmetic expansion (`$((...))`).
8. Then word splitting, which cuts the results on spaces, tabs and newlines.
9. Then filename expansion, the wildcards.
10. Finally quote removal, which strips the quote characters themselves so the program never sees them.
11. Two consequences follow directly. Word splitting happens after variable expansion, which is why unquoted variables split.
12. And filename expansion happens after word splitting, which is why a variable holding a star can still expand to filenames unless quoted.

19.12.5 TECHNICAL - the engineer's version

Form	Name	Consults disk
`{a,b}`	brace expansion	no
`~user`	tilde expansion	passwd file
`$VAR`	parameter expansion	no
`$(cmd)`	command substitution	runs a program
`$((x+1))`	arithmetic expansion	no
`* ? []`	pathname expansion	yes
`<(cmd)`	process substitution	creates a fifo

1. The controlling variable for word splitting is `IFS`, the internal field separator. Its default is space, tab and newline. Setting `IFS=$'\n'` restricts splitting to line boundaries.
2. Glob characters are `*` any string including empty, `?` any single character, `[abc]` one of a set, `[!abc]` or `[^abc]` none of a set. Character classes such as `[[:digit:]]` are POSIX.
3. A leading dot is never matched by `*` by default. `shopt -s dotglob` in bash changes that. This is why `rm *` does not remove `.git`.
4. `shopt -s nullglob` makes a non-matching pattern expand to nothing instead of itself. `shopt -s failglob` makes it an error, matching zsh's default.
5. `shopt -s globstar` enables `**` for recursive matching in bash 4 and later. macOS's bundled bash 3.2 does not have it; zsh has it always.
6. Parameter expansion has a large sub-language worth learning: `${VAR:-default}` use a default, `${VAR:?message}` fail if unset, `${VAR#prefix}` and `${VAR%suffix}` strip, `${VAR//old/new}` replace all, `${#VAR}` length, `${VAR:2:3}` substring.
7. Backticks are the old command substitution syntax. `$(...)` is POSIX, nests cleanly, and is what you should write. Backticks are a legacy convention.
8. Single quotes protect everything, including backslashes. There is no way to put a single quote inside single quotes; you must close, escape, and reopen: `'it'\''s'`.
9. Double quotes still allow `$`, backtick and backslash. They suppress word splitting and globbing.
10. `"$@"` expands to each argument as a separate quoted word. `"$*"` joins them into one word. This distinction matters in every wrapper script ever written.
11. For filenames from `find`, use `-print0` with `xargs -0` or `while IFS= read -r -d ''`. Newlines are legal in UNIX filenames, and any line-based loop over filenames is broken in principle.
12. `shellcheck` is a static analyser that catches unquoted expansions and dozens of similar faults. Running it on every script is the cheapest quality improvement available in shell programming.

19.12.6 WORDS - remember these

1. Glob — a wildcard filename pattern — pathname expansion performed by the shell, not by the command.
2. Expansion — the shell rewriting your line before running it — the ordered sequence from brace expansion to quote removal.
3. Word splitting — cutting expanded text into separate arguments — splitting on the characters in `IFS`, after parameter expansion.
4. IFS — the characters that separate words — the internal field separator, defaulting to space, tab and newline.
5. Command substitution — replacing a command with its output — `$(cmd)`, with trailing newlines removed.
6. Quote removal — the final step that deletes the quote marks — performed after all expansions, so programs never see quotes.

19.13 Shell scripting

19.13.1 PLAIN - in simple words

1. A shell script is a file containing the same commands you would type.
2. Put them in a file, mark the file as runnable, and you can run the whole sequence with one word.
3. The first line should say which program interprets the file. That line starts with `#!` and is called the **shebang**.
4. `#!/usr/bin/env bash` is the usual choice, because it finds bash wherever it is installed.
5. Inside, you get variables, `if` for decisions, `for` and `while` for repetition, and functions for naming a block of steps.
6. Arguments arrive as `$1`, `$2`, and so on. `$@` is all of them. `$#` is how many.
7. `exit N` ends the script with the code N, which the caller can test.
8. One line near the top prevents most disasters: `set -euo pipefail`.
9. It means: stop on the first failure, stop if I use a variable I never set, and notice failures inside pipelines.

19.13.2 PLAIN - a picture in your head

1. Think of a recipe card taped to the kitchen wall.
2. Typing commands is cooking from memory. A script is writing the recipe down so tomorrow's version is identical.
3. Without `set -e`, the recipe says "if the oven fails to heat, carry on anyway", which produces a raw cake presented as finished.
4. With `set -e`, the cook stops and tells you the oven failed.
5. Functions are named sub-recipes: "make the sauce" written once, used three times.
6. Arguments are the parts you leave blank: how many people, which flavour.

Where this comparison breaks: a human cook applies judgement and notices when something looks wrong. A script notices nothing at all. It will pour salt for an hour if you tell it to, and its only safety comes from checks you wrote in advance.

19.13.3 PLAIN - a worked example

1. Here is the real difference `set -euo pipefail` makes. Both scripts were run in this sandbox.

```
#!/usr/bin/env bash
cd /nonexistent-dir
echo "still running, and about to work in $(pwd)"
echo "value is: $UNDEFINED_VAR"
```

2. Real output:

```
./noset.sh: line 2: cd: /nonexistent-dir: No such file or directory
still running, and about to work in /tmp/shdemo
value is:
exit=0
```

3. Read that carefully. The `cd` failed. The script continued in the wrong directory. It used an undefined variable as an empty string. And it reported success.
4. If line 3 had been `rm -rf ./*`, it would have deleted the wrong directory and told the caller everything was fine.
5. Now the same script with the guard:

```
#!/usr/bin/env bash
set -euo pipefail
cd /nonexistent-dir
echo "this line never runs"
```

6. Real output:

```
./withset.sh: line 3: cd: /nonexistent-dir: No such file
  or directory
exit=1
```

7. It stopped at the failure and reported failure. That is the whole argument for the line.

19.13.4 PLAIN – what is really happening inside

1. The `#!` line is read by the kernel, not the shell. When you execute a file, the kernel looks at its first two bytes.
2. If they are `#!`, the kernel reads the rest of the line, runs that program, and hands it the script's path as an argument.
3. So `./script.sh` really becomes `/usr/bin/env bash ./script.sh`.
4. `env` then searches `PATH` for `bash` and executes it. That indirection is why the `env` form works on machines where bash is not in `/bin`.
5. Without the execute permission bit, the kernel refuses, and you get exit code 126.
6. `set -e` sets the `errexit` shell option. After every simple command the shell checks the status and exits if it is non-zero and untested.
7. `set -u` sets `nounset`. Expanding an unset variable becomes an error instead of an empty string.
8. `set -o pipefail` changes a pipeline's status from "the last command" to "the rightmost command that failed".
9. Functions are not separate processes. They run inside the same shell, so they can change its variables and its directory.
10. A subshell, written `( ... )`, is a forked copy. Changes inside it are thrown away when it ends. That is why `cd` inside `( )` does not move you.

19.13.5 TECHNICAL – the engineer's version

1. Here is a complete, working script, run for real in this sandbox. It reads a web server log and reports error counts, with a meaningful exit code.

```
#!/usr/bin/env bash
set -euo pipefail

usage() {
  echo "usage: $(basename "$0") LOGFILE [MIN_ERRORS]" >&2
  exit 64
}
```

```
[ $# -ge 1 ] || usage
logfile=$1
min_errors=${2:-1}

if [ ! -r "$logfile" ]; then
  echo "error: cannot read $logfile" >&2
  exit 66
fi

total=$(wc -l < "$logfile")
echo "file:  $logfile"
echo "lines: $total"

echo "status counts:"
awk '{print $9}' "$logfile" | sort | uniq -c | sort -rn

echo "clients with at least $min_errors error(s):"
found=0
while read -r count ip; do
  if [ "$count" -ge "$min_errors" ]; then
    printf '  %-12s %s\n' "$ip" "$count"
    found=$((found + 1))
  fi
done < <(awk '$9 >= 400 {print $1}' "$logfile" \
          | sort | uniq -c | sort -rn)

if [ "$found" -eq 0 ]; then
  echo "  none"
  exit 0
fi
exit 1
```

2. Line by line. `#!/usr/bin/env bash` selects bash via `PATH`.
3. `set -euo pipefail` turns on the three guards described above.
4. `usage()` defines a function. `>&2` sends its message to standard error, so it is not captured by a caller collecting output. `exit 64` is `EX_USAGE`.
5. `[ $# -ge 1 ] || usage` is an idiom: if there is not at least one argument, run `usage`. `$#` is the argument count.
6. `min_errors=${2:-1}` uses parameter expansion to default the second argument to 1 when it is absent. Without this, `set -u` would abort.
7. `[ ! -r "$logfile" ]` tests readability. The quotes are essential; a path with a space would otherwise become two arguments and break the test.
8. `wc -l < "$logfile"` uses redirection rather than passing the filename, so `wc` prints only the number without the filename after it.
9. The `while read -r count ip` loop reads two fields per line. `-r` stops backslashes being interpreted, and should always be present.
10. `done < <(...)` is process substitution. The pipeline runs in a separate process and its output is fed to the loop's standard input.
11. This matters: a plain `pipeline | while read` puts the loop in a subshell, so `found` would be lost when the loop ends. Process substitution keeps the loop in the main shell.
12. `$((found + 1))` is arithmetic expansion. No external `expr` is needed.
13. The script exits 0 when nothing was found and 1 when errors were found, which lets a monitoring system use it directly in an `if`.
14. Real runs:

```
$ ./logsum.sh access.log 2
```

```
file:  access.log
lines: 8
status counts:
      4 200
      3 500
      1 404
clients with at least 2 error(s):
  10.0.0.7     2
  10.0.0.12    2
exit=1

$ ./logsum.sh
usage: logsum.sh LOGFILE [MIN_ERRORS]
exit=64

$ ./logsum.sh /nope.log
error: cannot read /nope.log
exit=66
```

15. Portability notes. `[[ ]]` is a bash and zsh feature with better quoting behaviour; `[ ]` is POSIX and is really the program `test`. Process substitution and arrays are bash and zsh only, not `sh`.
16. `trap 'rm -f "$tmp"' EXIT` is the standard way to clean up on any exit path, including errors. Create temporary files with `mktemp`, never with a fixed name in `/tmp`.
17. Where experts disagree: some hold that anything over roughly 100 lines should be rewritten in Python. Others keep shell scripts of a thousand lines running happily. The practical dividing line most teams use is data structures: the moment you want a list of records with fields, leave shell.

19.13.6 WORDS - remember these

1. Shebang — the first line naming the interpreter — the `#!` magic number read by the kernel's `execve`.
2. `set -e` — stop at the first failure — the `errexit` option, with documented exceptions inside conditions.
3. `set -u` — treat unset variables as errors — the `nounset` option.
4. `pipefail` — a pipeline fails if any stage fails — a bash and zsh option, not in POSIX.
5. Subshell — a forked copy of the shell — created by `( )`, pipelines and command substitution; its variable changes do not escape.
6. Process substitution — treat a command's output as a file — `<(cmd)`, implemented with a FIFO or `/dev/fd`, bash and zsh only.
7. `trap` — run cleanup code when the script ends — a handler registered for a signal or the pseudo-signal `EXIT`.

19.14 Getting comfortable

19.14.1 PLAIN - in simple words

1. Speed at the command line comes from about a dozen habits, not from typing faster.
2. Tab completion: type the first few letters and press Tab. The shell finishes the name.
3. History search: press Ctrl-R and type part of an old command. It appears. Press Enter to run it.
4. Line editing: Ctrl-A jumps to the start of the line, Ctrl-E to the end, Ctrl-U deletes to the start, Ctrl-W deletes the last word.
5. Aliases: give a long command a short name you choose.
6. Sessions that survive: `tmux` keeps your work running when your connection drops.
7. Reading manuals properly: every manual page has the same sections, and you only ever need three of them.

19.14.2 PLAIN - a picture in your head

1. Think of learning a musical instrument.
2. Nobody plays fast by moving their fingers faster. They play fast because common patterns became automatic.
3. Tab completion is like a scale you no longer think about.
4. Ctrl-R is like remembering a phrase you played last week instead of re-inventing it.
5. tmux is like leaving your instrument set up on its stand, tuned, so tomorrow you sit down and continue mid-piece.

Where this comparison breaks: an instrument gives immediate feedback when you get it wrong. A shell often gives no feedback at all, so bad habits survive for years. The only cure is deliberately learning the correct form once.

19.14.3 PLAIN - a worked example

1. Keyboard shortcuts worth memorizing. These come from the readline library, used by bash, and zsh has the same set in emacs mode.

Keys	What it does	Keys	What it does
Ctrl-A	start of line	Ctrl-E	end of line
Ctrl-U	delete to line start	Ctrl-K	delete to end
Ctrl-W	delete previous word	Ctrl-Y	paste last delete
Ctrl-R	search history back	Ctrl-G	cancel the search
Ctrl-L	clear the screen	Ctrl-C	cancel this line
Ctrl-D	end of input, log out	Ctrl-Z	suspend the job
Alt-B	back one word	Alt-F	forward one word
Alt-.	last argument of prev	Ctrl-_	undo the edit

2. On macOS, Alt is the Option key, and Terminal.app needs "Use Option as Meta key" enabled in its settings before Alt-B and Alt-F work.
3. Aliases, real output from this sandbox:

```
$ alias ll="ls -lah"
$ alias
alias l='ls -CF'
alias la='ls -A'
alias ll='ls -lah'
alias ls='ls --color=auto'
$ type ll
ll is aliased to `ls -lah'
```

4. Aliases go in `~/.zshrc` or `~/.bashrc` so they exist in every new shell.
5. An alias is text substitution on the first word only. For anything needing arguments in the middle, write a function instead.

19.14.4 PLAIN - what is really happening inside

1. Tab completion is a shell feature, not a terminal feature. The terminal just sends byte 9.
2. The shell's line editor intercepts it, looks at what you have typed, and works out the candidates.
3. bash uses the GNU readline library for this. zsh has its own editor, ZLE.
4. Modern completion is programmable. When you type `git che` and press Tab, the shell runs a completion function shipped with git, which knows about branches.
5. History is kept in memory during the session and written to a file when the shell exits. bash uses `~/.bash_history`; zsh uses `~/.zsh_history`.

6. That end-of-session write is why history from one window can be missing in another. `setopt INC_APPEND_HISTORY` and `setopt SHARE_HISTORY` in zsh, or `shopt -s histappend` with `PROMPT_COMMAND` in bash, fix it.
7. tmux works by putting your shell inside a pseudo-terminal that tmux itself owns, rather than the one your emulator made.
8. When your SSH connection dies, your emulator's pty is destroyed, but tmux's is not, because the tmux server process is still running on the far machine.
9. Reconnect, run `tmux attach`, and tmux re-draws its saved screen contents into your new terminal.
10. Without tmux, losing the connection sends SIGHUP to your shell, which normally kills everything it started.

19.14.5 TECHNICAL – the engineer's version

1. tmux, real version and behaviour from this sandbox:

```
$ tmux -V
tmux 3.4
$ tmux new-session -d -s demo 'sleep 60'
$ tmux ls
demo: 1 windows (created Thu Aug 13 02:00:38 2026)
$ tmux kill-session -t demo
```

2. The commands worth knowing: `tmux new -s name`, `tmux ls`, `tmux attach -t name`, `tmux kill-session -t name`. Inside, the prefix key is Ctrl-B by default: `Ctrl-B d` detach, `Ctrl-B c` new window, `Ctrl-B "` split horizontally, `Ctrl-B %` split vertically, `Ctrl-B [` scrollback mode.
3. GNU Screen is the older equivalent, first released 1987, prefix Ctrl-A. tmux, first released 2007 by Nicholas Marriott, is the more actively developed choice today.
4. `nohup cmd &` and `disown` are lighter alternatives: they detach a single job from the terminal's hangup signal but give you no way to reattach and see it.
5. Manual page structure is standardized. The sections, in the order they always appear, and confirmed against the real `grep` page in this sandbox:

```
GREP(1)                 User Commands              GREP(1)
NAME
SYNOPSIS
DESCRIPTION
OPTIONS
REGULAR EXPRESSIONS
EXIT STATUS
ENVIRONMENT
NOTES
COPYRIGHT
BUGS
EXAMPLE
SEE ALSO
```

6. How to read one properly: read NAME to confirm it is the right tool, read SYNOPSIS to learn the argument shape, then jump straight to EXAMPLES at the bottom. Read DESCRIPTION only when the examples are not enough.
7. Inside `less`, which pages the manual: `/word` searches, `n` repeats, `q` quits, `G` goes to the end where the examples are.
8. Manual sections are numbered. 1 user commands, 2 system calls, 3 library functions, 4 devices, 5 file formats, 7 miscellaneous and overviews, 8 administration commands.
9. That numbering matters. `man 1 printf` is the command; `man 3 printf` is the C function. `man 5 passwd` is the file format; `man 1 passwd` is the tool.

10. `man -k word` or `apropos word` searches all the NAME lines when you do not know the command's name.
11. GNU tools often have fuller documentation in `info` rather than `man`. `curl --help all` and `git help -a` are other real sources.
12. `tldr`, a community project, gives example-first summaries and is worth installing when a manual page is 900 lines long.

19.14.6 WORDS - remember these

1. readline — the library giving bash its line editing — GNU readline, configured through `~/.inputrc`.
2. Tab completion — the shell finishing a word for you — programmable completion driven by shell functions.
3. Reverse search — finding an old command by fragment — Ctrl-R, an incremental backward search through the history list.
4. Alias — a short name for a longer command — first-word text substitution performed before expansion.
5. tmux — keeps sessions alive across disconnection — a terminal multiplexer holding its own ptys in a persistent server process.
6. SIGHUP — the "your terminal has gone" signal — signal 1, sent to the foreground group when the controlling terminal is lost.

19.15 Package managers

19.15.1 PLAIN - in simple words

1. A package manager installs software for you and keeps track of what it installed.
2. It knows which files belong to which program, so it can remove them cleanly.
3. It knows which programs depend on which other programs, and installs those too.
4. It gets the software from a **repository**: a server holding a catalogue and the files.
5. It checks a cryptographic signature, so you can tell the files really came from the people who claim to have made them.
6. Linux distributions each have one. Debian and Ubuntu use `apt`. Fedora and Red Hat use `dnf`. Arch uses `pacman`.
7. macOS has no official one for developer tools, so the community built **Homebrew**.
8. Windows now has `winget`, built by Microsoft.
9. The pattern is always: update the catalogue, then install by name.

19.15.2 PLAIN - a picture in your head

1. Think of a library with a strict lending system.
2. You ask for one book. The librarian knows that book refers constantly to two others, so brings all three.
3. Every book has a stamp proving it came from a real publisher, not a forgery somebody left on a shelf.
4. When you return the first book, the librarian checks whether anyone else is using the other two before shelving them.
5. Installing software by downloading it yourself is walking into the building and taking a book without telling anyone. Nothing tracks it and nothing removes it later.

Where this comparison breaks: books do not conflict. Software versions do. Two programs may need incompatible versions of the same library, and no librarian can satisfy both from one shelf. That problem is called dependency hell, and it is the reason containers and per-language package managers exist.

19.15.3 PLAIN – a worked example

1. Real `apt` output from this sandbox, showing what a package manager actually knows:

```
$ apt-cache policy curl
curl:
  Installed: 8.5.0-2ubuntu10.9
  Candidate: 8.5.0-2ubuntu10.11
  Version table:
     8.5.0-2ubuntu10.11 500
        500 archive.ubuntu.com/ubuntu noble-updates/main
 *** 8.5.0-2ubuntu10.9 100

$ apt-cache depends curl
curl
  Depends: libc6
  Depends: libcurl4t64
  Depends: zlib1g

$ dpkg -L curl | head -4
/usr
/usr/bin
/usr/bin/curl
/usr/share
```

2. Three separate facts there: which version is installed, which is available, what it needs, and exactly which files it owns.
3. The repositories it trusts are listed in `/etc/apt/sources.list.d/`, and the signing keys in `/etc/apt/trusted.gpg.d/`. In this sandbox those held `ubuntu.sources`, `docker.list`, and the Ubuntu archive keyring files.
4. Equivalent commands across systems:

Task	apt	dnf / pacman
refresh	apt update	dnf check-update
install	apt install curl	dnf install curl
remove	apt remove curl	dnf remove curl
search	apt search curl	dnf search curl
upgrade	apt upgrade	dnf upgrade

5. On Arch the same five are `pacman -Sy`, `pacman -S curl`, `pacman -R curl`, `pacman -Ss curl`, `pacman -Syu`.
6. On macOS: `brew update`, `brew install curl`, `brew uninstall curl`, `brew search curl`, `brew upgrade`.
7. On Windows: `winget search curl`, `winget install curl`, `winget upgrade --all`.

19.15.4 PLAIN – what is really happening inside

1. `apt update` downloads an index file listing every package, its version, its dependencies and a hash of its contents.
2. That index is signed. The manager verifies the signature against keys you already trust, and refuses to proceed if it fails.
3. `apt install X` reads the index and solves a puzzle: pick versions of X and everything it needs so that all constraints hold at once.
4. This is genuinely a hard computational problem. Real solvers use SAT solving techniques, and they can fail with a message about held broken packages.

5. It then downloads each chosen package, verifies each hash, and unpacks the files to their recorded locations.
6. It runs the package's own install scripts, and writes down which files were placed, so removal is exact.
7. Homebrew works differently. It downloads pre-built binaries, called bottles, into its own directory tree and symlinks them into place, so it never touches Apple's system files.
8. Now the warning. You will see installation instructions of this shape:

```
curl -fsSL some-site.example/install.sh | sh
```

9. That downloads a script and runs it immediately, with your permissions, with no chance to read it.
10. There is no signature check, no record of what it installed, and no way to uninstall it cleanly.
11. Worse, a hostile server can send different content to `curl` than it sends to a browser, so reading the page first proves nothing.
12. The safer form is two steps: download to a file, read it, then run it. That is not paranoia, it is the minimum you would apply to any other code you were about to give full access to your account.

19.15.5 TECHNICAL - the engineer's version

Manager	Year	System	Format
dpkg	1994	Debian	.deb
RPM	1995	Red Hat 2.0	.rpm
APT	1998	Debian	wraps dpkg
pacman	2002	Arch Linux	.pkg.tar
YUM	2002	RPM systems	wraps rpm
Homebrew	2009	macOS	formulae
DNF	2015	Fedora 22	wraps rpm
winget	2020	Windows	manifests

1. Precise history. Ian Murdock created dpkg for Debian in January 1994. Red Hat Linux 2.0 shipped RPM on 20 September 1995. APT 0.0.1 was released by Scott K. Ellis in 1998; APT 1.0 came on 1 April 2014.
2. YUM was created by Seth Vidal and Michael Stenner at Duke University on 7 June 2002. DNF replaced it as Fedora's default in May 2015 with Fedora 22.
3. Judd Vinet created pacman alongside the launch of Arch Linux in March 2002.
4. Max Howell created Homebrew on 21 May 2009. Version 1.0 arrived on 21 September 2016.
5. Microsoft released winget in preview at Build on 19 May 2020, and version 1.0 on 27 May 2021. Chocolatey, the community predecessor, released 0.6.0 on 23 March 2011.
6. Two distinct layers exist and confusing them causes real problems. System package managers own `/usr` and system libraries. Language package managers, `pip`, `npm`, `cargo`, `gem`, own their own trees.
7. Installing a Python library with the system manager and another with `pip` into the same interpreter is a known way to break a machine. Use virtual environments, or `pipx`, or containers.
8. Signature mechanics. Debian signs the `Release` file with OpenPGP; the file contains hashes of the index files, which contain hashes of the packages. So one signature check covers everything by chaining hashes.
9. RPM signs individual packages as well. Both models are documented, both are sound, and both fail if you add an untrusted key without thinking.
10. Homebrew does not sign bottles with OpenPGP. It relies on HTTPS transport and on checksums recorded in the formula in a public git repository. That is weaker than Debian's model, and it is a fair criticism.
11. Reproducibility is active work rather than settled fact. The Reproducible Builds project, running since 2013, aims to make a package's binary byte-for-byte derivable from its source. Debian reports high but not complete coverage. Treat any claim of full reproducibility as a target, not a delivered guarantee.

12. Supply chain attacks are the reason all of this matters. Real incidents include the event-stream npm package compromise in 2018 and the xz-utils backdoor discovered in March 2024, which reached Debian and Fedora testing branches before being caught.

19.15.6 WORDS - remember these

1. Package — one installable unit of software — an archive plus metadata describing version, dependencies and file list.
2. Repository — the server holding the catalogue — a signed index plus the package files it describes.
3. Dependency resolution — working out what else must be installed — constraint solving over the version graph, often SAT-based.
4. Signature — proof the files came from who they claim — an OpenPGP signature over the index or the package.
5. Bottle — a pre-built Homebrew package — a relocatable binary archive built by Homebrew's own infrastructure.
6. Supply chain attack — hostile code inserted upstream — compromise of a package, its maintainer or its build system, rather than of your machine.

19.98 Common wrong ideas

1. Wrong: the terminal and the shell are the same program. Right: the terminal emulator draws the window; the shell runs inside it and can be swapped for another without changing the window.
2. Wrong: `ls` understands the * wildcard. Right: the shell expands * into a list of filenames first; `ls` only ever receives real names.
3. Wrong: Ctrl-C sends the letter C to the program. Right: the kernel's line discipline sees byte 3 and sends SIGINT to the foreground process group; the byte itself is never delivered.
4. Wrong: `2>&1 > file` sends both streams to the file. Right: redirections are applied left to right, so stderr is aimed at the screen first and stays there. Write `> file 2>&1`.
5. Wrong: a pipeline runs the first command fully, then the second. Right: all stages run at the same time, connected by a kernel buffer, with the writer blocking when it is full.
6. Wrong: exit code 1 always means something crashed. Right: many tools use 1 for a normal negative answer. `grep` returns 1 when it searched correctly and found nothing.
7. Wrong: a child process can change its parent's environment variables. Right: the child gets a copy at `exec` time. Nothing it does can reach back. That is why `cd` must be a shell builtin.
8. Wrong: editing `~/.zshrc` changes shells that are already open. Right: it is read at startup. Existing shells keep the old values until you `source` the file or open a new window.
9. Wrong: `kill -9` is the normal way to stop a program. Right: `kill` sends SIGTERM, which lets the program flush data and clean up. SIGKILL cannot be handled and should be the second attempt, not the first.
10. Wrong: piping a downloaded script into a shell is fine because you trust the website. Right: there is no signature, no record and no clean removal, and a server can serve different bytes to `curl` than to a browser.

19.99 Chapter summary in 20 lines

1. A terminal was a physical machine at the end of a wire, with a keyboard and a printer or screen, and almost no intelligence of its own.
2. The Teletype Model 33 of 1963 ran at 110 bits per second, printed on paper, and is why UNIX commands have two-letter names.
3. The DEC VT100 of 1978 popularized the escape sequences, standardized as ECMA-48 and ISO/IEC 6429, that still control your terminal today.
4. The device file is called tty because the machine on the other end was a teletype.

5. Terminal, terminal emulator, shell, console and command line are five different things; you use an emulator containing a shell.
6. A pseudo-terminal is a kernel-made fake wire with a master end for the emulator and a slave end for the shell.
7. The line discipline sits between them, doing echo, line editing and buffering, and turning Ctrl-C into SIGINT for the foreground process group.
8. A shell reads a line, expands it, forks, execs and waits. That loop is the whole job.
9. The shell family runs from Thompson 1971 and Bourne 1979 through bash 1989 to zsh 1990, which macOS made the default in Catalina in 2019.
10. PowerShell, from 2006, is a different design entirely: it passes typed objects between commands instead of text.
11. Every process starts with descriptors 0, 1 and 2 for input, output and errors, kept separate so results stay clean.
12. Redirection is `open` plus `dup2` performed between fork and exec, applied strictly left to right, which is why `2>&1` order matters.
13. A pipe is a kernel buffer, 65536 bytes on Linux, joining two processes that run at the same time, with blocking as natural back pressure.
14. When a reader exits early the writer gets SIGPIPE, signal 13, reported by the shell as exit status 141.
15. Exit codes are 0 for success and 1 to 255 for failure, with 126 not executable, 127 not found, and 128 plus N for death by signal N.
16. `&&` runs the next command only on success, `||` only on failure, and `;` always. Build systems test only these numbers.
17. The environment is a copied list of strings; `PATH` is searched left to right, the result is cached, and `./` is required for the current directory.
18. Windows keeps settings in one registry of hives, keys and typed values; macOS and Linux use plists, `/etc` and dotfiles, increasingly under the XDG directories.
19. The shell rewrites your line in a fixed order, and word splitting happens after variable expansion, which is why you quote every expansion.
20. `set -euo pipefail` at the top of a script turns silent wrong behaviour into a loud stop, which is the single highest-value line you can write.

20

CHAPTER

Files, Formats, Extensions and the .exe

20.0 What this chapter gives you

1. You will be able to say exactly what a file is, and why there is no such thing as a "picture file" at the level of the disk.
2. You will be able to explain what a file extension really does on Windows, on macOS and on Linux, and why the three answers are different.
3. You will be able to read the first bytes of any file and name the format, using a table of real magic numbers you will have memorized parts of.
4. You will be able to explain why a text file written on Windows shows strange marks on Linux, and fix it.
5. You will be able to open a DOCX, an XLSX, a JAR and an APK with a zip tool, because they are all zip files, and show the layout inside.
6. You will be able to describe what an .exe actually contains, section by section, including why it still says "This program cannot be run in DOS mode".
7. You will be able to trace, in thirty numbered steps, everything that happens between your double-click and the first line of your program.
8. You will be able to explain why a `.py` or `.sh` file also runs, and what the two characters `#!` do at the level of the kernel.
9. You will be able to identify a file of an unknown format using only `file`, a hex dump, and `strings`.
10. You will be able to name what an installer does, why unsigned software warns you, and what a photo silently tells the world about where you were.

20.1 What a file actually is

20.1.1 PLAIN – in simple words

1. A file is a run of numbered boxes. Each box holds one byte, a number from 0 to 255.
2. That is the whole of it. A file is a sequence of bytes with a length.
3. The file also has a name, so that a human and a program can find it again.
4. Around that sequence, the system keeps a few extra facts: how big it is, when it was last changed, who owns it, who is allowed to read it.
5. Those extra facts are called **metadata**, which just means "data about the data".
6. Nothing in the bytes says "I am a picture". Nothing in the metadata says it either, on most systems.
7. A file becomes a picture only when a program chooses to read those bytes using picture rules.
8. If a different program reads the same bytes using sound rules, it will produce noise. Nothing was broken. The bytes never claimed anything.
9. So a file is not a thing with a nature. It is raw material plus an agreement about how to read it.

20.1.2 PLAIN - a picture in your head

1. Think of a long shelf of numbered pigeonholes in a post room.
2. Each pigeonhole holds one small card with a number written on it, from 0 to 255.
3. The shelf has a label on the front: "note.txt". That is the file name.
4. A clipboard hangs beside the shelf. It records: 26 cards, filled on 13 August at 01:57, owned by the office manager, readable by everyone.
5. That clipboard is the metadata. The pigeonholes are the file contents.
6. Now, one clerk has been told "these cards spell out English letters". Another clerk has been told "these cards are pixel colours".
7. Both clerks read the exact same cards. They produce completely different results. Neither clerk is cheating.

Where this comparison breaks: real pigeonholes are physically next to each other, and a real file usually is not. The bytes of one file can be scattered across thousands of separate places on a disk, and the file system quietly stitches them into one continuous-looking sequence for you. The shelf is a fiction the file system maintains. Also, a real clipboard is attached to the shelf, whereas on most file systems the metadata lives somewhere else entirely, in a small record called an inode, and the file name is not even part of it.

20.1.3 PLAIN - a worked example

1. Here is a file with 26 bytes in it, viewed as raw numbers and then as characters.

```
$ stat note.txt
  File: note.txt
  Size: 26          Blocks: 8       IO Block: 4096  regular file
Device: 254,0    Inode: 794747      Links: 1
Access: (0644/-rw-r--r--)  Uid: (0/root)   Gid: (0/root)
Access: 2026-08-13 01:57:42.022877781 +0000
Modify: 2026-08-13 01:57:42.020159532 +0000
Change: 2026-08-13 01:57:42.020159532 +0000
 Birth: 2026-08-13 01:57:42.016211454 +0000
```

2. `Size: 26` is the only fact about the contents. Twenty-six bytes. That is all the system knows about what is inside.
3. `Inode: 794747` is the number of the little record that holds this metadata. The name `note.txt` is not stored there. It is stored in the folder.
4. `Blocks: 8` means eight 512-byte units are actually reserved on disk, so 4096 bytes are used to store 26. Storage is handed out in whole blocks.
5. There are four timestamps, not one, and they mean different things:

Field	Plain meaning	Changed by
Access	last time it was read	reading the file
Modify	last time contents changed	writing bytes
Change	last time metadata changed	chmod, rename
Birth	when it was created	creation only

6. Notice what is absent from every line above: any statement about what kind of file this is. The word "text" appears nowhere.

20.1.4 PLAIN - what is really happening inside

1. When you save a file, the file system does three separate jobs.
2. Job one: find free blocks on the storage device and write your bytes into them.

3. Job two: create or update a small fixed-size record holding the size, the owner, the permissions, the timestamps, and a list of which blocks hold the data. On Linux file systems this record is the **inode**.
4. Job three: add an entry to a directory. A directory is itself just a file whose contents are a list of pairs: a name, and an inode number.
5. This is why the name is not part of the file. The name lives in the folder, pointing at the file.
6. This is also why one file can have two names. Two directory entries can point at the same inode. That is a **hard link**.
7. Deleting a name does not delete the bytes. It removes one directory entry and reduces a counter. The bytes go only when the counter reaches zero.
8. The `Links: 1` line in the output above is exactly that counter.
9. None of these three jobs asks or records what the bytes mean.

20.1.5 TECHNICAL - the engineer's version

1. A regular file in POSIX terms is an unstructured byte stream addressed by offset, with no record structure imposed by the kernel.
2. The metadata returned by the `stat(2)` system call is defined in POSIX.1 and is filled from the file system's own inode structure.
3. On ext4 the inode is 256 bytes by default and holds mode, uid, gid, size, four timestamps with nanosecond precision, link count, and either extents or block pointers.
4. `st_size` is the logical length in bytes. `st_blocks` is the allocated storage in 512-byte units and can be smaller than `st_size` for a sparse file or larger because of block rounding.
5. Directory entries on ext4 are stored in `linux_dirent` structures containing an inode number, a record length, a name length, a one-byte file type hint, and the name. There is no type field beyond regular, directory, symlink, FIFO, socket, block device and character device.
6. NTFS differs. Its Master File Table record holds a set of named attributes, and small files are stored resident inside the MFT record itself. NTFS also supports multiple named data streams per file, which is covered in 20.13.
7. Apple's APFS, shipped in 2017 with macOS 10.13 High Sierra, stores extended attributes in the same B-tree as the file record.
8. Real figures for common file systems:

File system	Max file size	Max name length
ext4	16 TiB	255 bytes
NTFS	8 PiB (Win11)	255 UTF-16 code units
APFS	8 EiB	255 UTF-8 characters
exFAT	128 PiB	255 UTF-16 code units

9. Tools that observe this layer: `stat`, `ls -li`, `df -i`, `debugfs` on ext4, `fsutil file layout` on Windows, `diskutil` and `stat -f` on macOS.

20.1.6 WORDS - remember these

1. File — a named run of bytes — an unstructured byte stream addressed by offset, with an associated metadata record.
2. Metadata — facts about a file that are not its contents — the `struct stat` fields: mode, uid, gid, size, timestamps, link count.
3. Inode — the small record holding a file's facts — an on-disk structure indexed by number, containing everything except the name.
4. Directory — a folder — a file whose contents map names to inode numbers.
5. Hard link — a second name for the same file — an additional directory entry referencing the same inode, incrementing `st_nlink`.

6. Block — the smallest chunk of disk given out — the allocation unit, typically 4096 bytes on ext4 and NTFS.

20.2 File extensions: a convention, not a rule

20.2.1 PLAIN - in simple words

1. The bit after the last dot in a file name is the **extension**: `.txt`, `.jpg`, `.exe`.
2. Say this once and remember it: on almost every system, the extension is part of the name and nothing else.
3. It is a **convention**. Everyone agreed to use it. It is not enforced by the disk, and mostly not enforced by the kernel either.
4. You can rename `holiday.jpg` to `holiday.txt`. Not one byte inside the file changes.
5. What changes is the guess that programs make about the file when they see the name.
6. On Windows, that guess is very strong. The extension decides which program opens the file, full stop.
7. On macOS, the extension matters too, but the system also keeps a richer type idea behind the scenes.
8. On Linux, the usual tools ignore the name and look at the bytes instead.
9. So the same rename does three different things on three systems, and none of them altered the file.

20.2.2 PLAIN - a picture in your head

1. Think of a hospital where every patient wears a wristband with a written label: “diabetes”, “fracture”, “asthma”.
2. The wristband is fast and convenient. A nurse can glance and act.
3. But the wristband is written by hand. It is not the patient. Swap two wristbands and the patients are unchanged.
4. A careful doctor does not trust the wristband. She examines the patient. That takes longer and is always right.
5. Windows is the nurse acting on the wristband. Linux is the doctor examining the patient.
6. macOS is a hospital that reads the wristband first but also keeps a proper file in the records office.

Where this comparison breaks: a real patient has an objective illness that exists whether or not anyone examines it. A file does not have an objective format in the same way. Its format only exists as an agreement between whoever wrote it and whoever reads it. Two different agreements can both fit the same bytes, which is why a valid file can genuinely be two formats at once, a fact attackers use.

20.2.3 PLAIN - a worked example

1. Here is a real one-pixel PNG image. We copy it to a `.txt` name and ask the system what it is.

```
$ cp tiny.png tiny.txt
$ file tiny.txt
tiny.txt: PNG image data, 1 x 1, 8-bit/color RGB, non-interlaced

$ file --mime-type tiny.txt tiny.png
tiny.txt:   image/png
tiny.png:   image/png

$ sha256sum tiny.png tiny.txt
b1ff9c8ea3a780bad09b346c423d2d0e46815926879b18e841d928376a946640
  tiny.png
b1ff9c8ea3a780bad09b346c423d2d0e46815926879b18e841d928376a946640
  tiny.txt
```

2. The two checksums are identical to the last character. The contents are the same file.

3. The Linux `file` command did not care about the name at all. It read the bytes and answered PNG both times.
4. On Windows, double-clicking `tiny.txt` would open Notepad, which would show a screenful of rubbish and then possibly corrupt the file if you saved it.
5. Nothing about the file was ever wrong. The name simply stopped matching the contents, and one operating system trusts names.

20.2.4 PLAIN – what is really happening inside

1. Windows keeps a large lookup table called the **registry**. One part of it, `HKEY_CLASSES_ROOT`, maps extensions to type identifiers.
2. `.txt` maps to a name like `txtfile`. `txtfile` then has a `shell\open\command` entry giving the exact command line to run.
3. Double-clicking runs `ShellExecute`, which walks that chain: extension, then program identifier, then command. The file's contents are never consulted.
4. macOS uses **Uniform Type Identifiers**, reverse-domain strings such as `public.png` or `com.adobe.pdf`. Every application declares which identifiers it can open.
5. macOS decides a file's identifier mostly from its extension, then registers the answer in a database called Launch Services.
6. Linux desktops use the freedesktop.org shared MIME database, which combines name patterns with content tests, and content tests usually win.
7. Command-line Linux tools such as `file` use a library called **libmagic**, which holds thousands of rules of the form "at offset X, expect these bytes, therefore this format".
8. So there are two strategies in the world: trust the name, or read the bytes. Reading the bytes is slower and needs an open file. Trusting the name is instant and needs nothing.
9. That speed difference is the whole reason the extension convention survives.

20.2.5 TECHNICAL – the engineer's version

1. The 8.3 filename with a three-character extension comes from CP/M in 1974 and was carried into MS-DOS in 1981 and into the FAT file system.
2. In FAT, the name and extension were stored as two fixed fields, 8 bytes and 3 bytes, with no dot stored on disk. The dot was purely a display convention.
3. Modern Windows stores the whole name including dots as one UTF-16 string. The extension is defined as the text after the final dot.
4. Windows Explorer hides extensions for registered types by default. This setting is `HideFileExt` under `HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\Advanced`.
5. That default is a documented security weakness. A file named `invoice.pdf.exe` displays as `invoice.pdf` with a PDF-looking icon supplied by the executable's own resource section.
6. The ILOVEYOU worm of May 2000 used exactly this. Its attachment was named `LOVE-LETTER-FOR-YOU.TXT.vbs` and displayed as a `.TXT` file. It caused damage estimated in the billions of dollars.
7. Windows also supports right-to-left override characters in file names, which can display `exe.txt` when the real name ends in `.exe`. Modern mail gateways strip these.
8. Apple introduced Uniform Type Identifiers in Mac OS X 10.4 Tiger in 2005, and replaced the older four-character type and creator codes inherited from the 1984 Macintosh. A modern Swift framework, UniformTypeIdentifiers, arrived in macOS 11 in 2020.
9. The `file` command was written by Ian Darwin in 1986 and rewritten and maintained by Christos Zoulas since 1993. Its rule database is compiled into `magic.mgc`.
10. Behaviour by platform:

Platform	Primary decision	Fallback
Windows	extension plus registry	none
macOS	extension to UTI	type metadata
Linux CLI	content sniffing	extension
Linux desktop	shared MIME db	glob patterns

11. Web browsers are a fourth case. They obey the HTTP `Content-Type` header, with a defined sniffing algorithm in the WHATWG MIME Sniffing Standard, which servers can disable with `X-Content-Type-Options: nosniff`.

20.2.6 WORDS - remember these

1. Extension — the letters after the last dot — a filename suffix used as a type hint, not enforced by the file system.
2. Convention — everyone does it this way — behaviour that is customary but not specified or enforced.
3. Registry — the Windows settings database — a hive-structured key-value store; `HKEY_CLASSES_ROOT` maps extensions to ProgIDs.
4. UTI — Apple's name for a file kind — Uniform Type Identifier, a reverse-DNS string in a declared inheritance hierarchy.
5. libmagic — the library that guesses a file's type from its contents — a rules engine matching byte patterns at given offsets.
6. Double extension — a name with two dots used to deceive — an attack relying on `HideFileExt` and on icon resources embedded in the executable.

20.3 Magic numbers: the real identity of a file

20.3.1 PLAIN - in simple words

1. Most file formats begin with a short, fixed pattern of bytes that says which format it is.
2. That pattern is called a **magic number**, or a signature.
3. It is at the very start, so a program can read a few bytes, decide, and stop if it is wrong.
4. A PNG image always begins with the byte 137 and then the letters P, N, G.
5. A ZIP archive always begins with the letters P and K, the initials of Phil Katz who invented the format.
6. A Linux program always begins with the byte 127 and then the letters E, L, F.
7. A Windows program always begins with the letters M and Z, the initials of Mark Zbikowski who designed the old DOS executable header.
8. This is the closest thing a file has to a real identity, and it is still only a convention, because nothing stops you writing those bytes yourself.
9. But it is a far better guess than the name, because the bytes were written by the program that made the file.

20.3.2 PLAIN - a picture in your head

1. Think of the first line of a formal letter.
2. "Dear Sir" tells you it is a letter. "Once upon a time" tells you it is a story. "Ingredients:" tells you it is a recipe.
3. You do not need to read the rest. Three or four words settle it.
4. Better still, some documents begin with a printed crest or a watermark, which is harder to produce by accident.
5. A magic number is that crest. It is at the top, it is short, and it is very unlikely to appear at the top of something else by chance.

Where this comparison breaks: a letter's opening is meant for humans and can be argued about. A magic number is exact. Either the bytes are `89 50 4E 47` or they are not. There is no partial match and no interpretation. Also, unlike a crest, a magic number carries no proof of who wrote it. Anyone can type the same four bytes at the start of any file, which is exactly how polyglot files, files valid as two formats at once, are built.

20.3.3 PLAIN – a worked example

1. Here is the first sixteen bytes of a real PNG file we made, shown as hex on the left and as printable characters on the right.

```
$ xxd -l 16 tiny.png
00000000: 8950 4e47 0d0a 1a0a 0000 000d 4948 4452  .PNG........IHDR
```

2. Read it byte by byte. `89` is the number 137, chosen deliberately to be above 127 so that a program treating the file as text notices immediately.
3. `50 4e 47` is `P`, `N`, `G` in ASCII.
4. `0d 0a` is carriage return and line feed. If a broken file transfer converted line endings, this pair would be damaged and the file would fail its own check.
5. `1a` is the old DOS end-of-file character. If you type this file on DOS, it stops printing right there instead of spewing binary.
6. `0a` is a lone line feed, catching the opposite conversion.
7. Then `00 00 00 0d`, which is the number 13 in big-endian order: the length of the first chunk.
8. Then `49 48 44 52`, which is `IHDR`, the image header chunk.
9. The whole eight-byte PNG signature was designed in the mid-1990s to survive damage by naive file transfer software, and it works.

20.3.4 PLAIN – what is really happening inside

1. A program that must identify a file opens it and reads the first block, usually the first 512 or 4096 bytes.
2. It then runs down a list of rules. Each rule says: at this offset, compare these bytes, and if they match, report this type.
3. Rules can be nested. Matching `RIFF` at offset 0 leads to a second test at offset 8, which decides between WAV, AVI and WebP.
4. Rules can also be negative. A ZIP that begins with the entry name `mimetype` is treated as an OpenDocument file rather than a plain archive.
5. Some formats have no fixed signature at all. Plain text, CSV and raw MP3 audio are examples, so the tool falls back to statistics: are all bytes printable, are there consistent separators.
6. When nothing matches, the answer is `data`, which honestly means "I do not know".

20.3.5 TECHNICAL – the engineer's version

1. The following signatures were all verified by hex-dumping real files on the machine used to write this chapter, except where the row says otherwise.

Format	First bytes (hex)	As text
PNG	89 50 4E 47 0D 0A 1A 0A	.PNG....
JPEG/JFIF	FF D8 FF E0	....
GIF87a	47 49 46 38 37 61	GIF87a
GIF89a	47 49 46 38 39 61	GIF89a
PDF	25 50 44 46 2D	%PDF-
ZIP local	50 4B 03 04	PK..
ZIP central	50 4B 01 02	PK..
ZIP end record	50 4B 05 06	PK..

Format	First bytes (hex)	As text
GZIP	1F 8B 08	…
ELF	7F 45 4C 46	.ELF
PE / MZ	4D 5A	MZ
Mach-O 64-bit	CF FA ED FE	….
Java class	CA FE BA BE	….
WebAssembly	00 61 73 6D 01 00 00 00	.asm….
RIFF / WAV	52 49 46 46	RIFF
MP3 with ID3	49 44 33	ID3
SQLite 3	53 51 4C 69 74 65 20 66	SQLite f
7-Zip	37 7A BC AF 27 1C	7z..'.
BZIP2	42 5A 68	BZh
XZ	FD 37 7A 58 5A 00	.7zXZ.

2. Notes on the trickier rows. Mach-O stores its magic as a 32-bit little-endian integer, so `MH_MAGIC_64`, whose value is `0xFEEDFACF`, appears on disk as `CF FA ED FE`. The 32-bit form is `0xFEEDFACE`.
3. A macOS universal binary starts with `0xCAFEBABE` in big-endian order, which is byte-for-byte identical to a Java class file. Tools disambiguate by checking the next field: Java stores a version number below 100, a fat Mach-O stores an architecture count. This collision is real and long-standing.
4. Mach-O was not produced in this sandbox, which is Linux. Those two rows come from the `loader.h` header in Apple's open-source XNU sources, not from a local dump.
5. Java class files carry a major version after the magic. Version 52 is Java 8, 55 is Java 11, 61 is Java 17, 65 is Java 21. A real dump taken here:

```
$ unzip -p commons-lang3-3.14.0.jar \
    org/apache/commons/lang3/AnnotationUtils\$1.class | xxd -l 16
00000000: cafe babe 0000 0034 00bc 0a00 0200 0307
```

`0x34` is 52, so this class targets Java 8. 6. WebAssembly's magic is the four bytes `\0asm` followed by a 32-bit version, currently 1. WebAssembly 1.0 became a W3C Recommendation in December 2019. 7. Raw MP3 has no header signature. An MPEG audio frame starts with an 11-bit sync pattern of all ones, so bytes such as `FF FB`. Most real files start with an ID3v2 tag instead, whose first three bytes are `ID3`. 8. The `file` command in action on the machine used here:

```
$ file sample.wav sample.db sample.wasm hello hello.exe
sample.wav:  RIFF (little-endian) data, WAVE audio,
             Microsoft PCM, 16 bit, mono 8000 Hz
sample.db:   SQLite 3.x database, last written using
             SQLite version 3045001, database pages 2
sample.wasm: WebAssembly (wasm) binary module version
             0x1 (MVP)
hello:       ELF 64-bit LSB pie executable, x86-64,
             dynamically linked, interpreter
             /lib64/ld-linux-x86-64.so.2, not stripped
hello.exe:   PE32+ executable (console) x86-64,
             for MS Windows, 19 sections
```

9. Note how much `file` extracted beyond the type: image dimensions, sample rate, SQLite version, section count, interpreter path. Those come from deeper rules in the same database.
10. The magic database ships as text in `/usr/share/file/magic/` and is compiled into `magic.mgc`. You can add your own rules and point `file` at them with the `-m` option.

20.3.6 WORDS - remember these

1. Magic number — a fixed pattern at the start of a file naming its format — a signature matched at a specified offset by a type-detection rule.
2. Signature — another word for magic number — same thing; the term "signature" is also used for cryptographic signing, so context matters.
3. Polyglot file — one file that is validly two formats — a file crafted so that two parsers each find a complete valid structure.
4. Endianness — which end of a number comes first — byte order; little-endian stores the least significant byte at the lowest address.
5. Sniffing — guessing a type by reading contents — content-based type detection, as opposed to name-based.
6. libmagic rule — one line in the guessing database — an offset, a type, a test value and a message, optionally nested by indentation depth.

20.4 Text files versus binary files

20.4.1 PLAIN - in simple words

1. People say a file is either "text" or "binary". That split is useful but it is not a real property of the file.
2. Every file is binary. Every file is bytes. There is no text mode on the disk.
3. What people mean by "text" is: if you show each byte as a character, you get something a human can read.
4. So "text" is a statement about how the bytes happen to look, not about how they are stored.
5. Text files also have one extra habit: they are split into lines.
6. And here the trouble starts, because the world never agreed on how to mark the end of a line.
7. Linux and macOS end a line with one invisible byte. Windows ends it with two. Very old Macs used one different byte.
8. That is why a file written on Windows sometimes shows odd marks when opened on Linux, and why a Linux file sometimes appears as one giant line on Windows.
9. On top of that, there is the question of which byte means which letter, which is called the **encoding**, and that is a second, larger mess.

20.4.2 PLAIN - a picture in your head

1. Imagine an old mechanical typewriter with a paper carriage.
2. To start a new line you must do two separate physical things.
3. First, push the carriage back to the left edge. That is the **carriage return**.
4. Second, roll the paper up by one line. That is the **line feed**.
5. Two actions, two separate control codes, because the machine really did have two separate mechanisms.
6. Windows still sends both codes, faithfully imitating the typewriter.
7. Unix decided one code should mean "do both", and sends only the line feed.
8. The old Macintosh decided the opposite, and sent only the carriage return.
9. Three reasonable choices, made in different decades, and we live with all three forever.

Where this comparison breaks: on a typewriter you could send carriage return without line feed on purpose, to type over a line and make bold text. Software no longer does that, so the two codes have collapsed into one meaning, "new line", and only the spelling differs. Also, a typewriter had no concept of encoding. Which key made which shape was fixed by the metal. In a file, the mapping from byte to character is a separate choice on top of the line-ending choice.

20.4.3 PLAIN – a worked example

1. We made the same two lines of text in three different line-ending styles and dumped the raw bytes.

```
$ file unix.txt dos.txt mac9.txt
unix.txt: ASCII text
dos.txt:  ASCII text, with CRLF line terminators
mac9.txt: ASCII text, with CR line terminators

$ od -c unix.txt
l i n e   o n e \n l i n e   t w o \n

$ od -c dos.txt
l i n e   o n e \r \n l i n e   t w o \r \n

$ od -c mac9.txt
l i n e   o n e \r l i n e   t w o \r

$ wc -l unix.txt dos.txt mac9.txt
 2 unix.txt
 2 dos.txt
 0 mac9.txt
```

2. `\n` is byte 10, the line feed. `\r` is byte 13, the carriage return.
3. Notice the last line. The tool counted two lines in the Unix file, two in the Windows file, and zero in the old Mac file.
4. It is not broken. `wc -l` counts line feed bytes, and the old Mac file contains none at all.
5. This is exactly how a real file can be "empty" to one tool and full of text to another.
6. When a Windows-made file is read by a Linux tool that only strips `\n`, the `\r` stays on the end of every value. A version number `1.2.3` becomes `1.2.3\r`, and comparisons silently fail.

20.4.4 PLAIN – what is really happening inside

1. There are three different layers where line endings can be changed, and confusing them causes most of the pain.
2. Layer one: the program that wrote the file chose what to put there.
3. Layer two: a transfer tool may translate on the way. Old FTP had an explicit text mode that rewrote line endings. Git can do the same on checkout.
4. Layer three: the program reading the file may quietly accept both, or may not.
5. Encoding is a separate question: which byte value stands for which character.
6. ASCII fixed 128 characters in the 1960s using values 0 to 127. That covers English and nothing else.
7. Everyone then invented their own meaning for values 128 to 255, producing dozens of incompatible sets, and the same bytes rendered differently in different countries.
8. Unicode solved the character question by giving every character in every script a number, and UTF-8 solved the byte question by encoding those numbers in one to four bytes.
9. UTF-8 was designed so that all the old ASCII values keep their old meaning. An English-only file is identical in ASCII and UTF-8.
10. Some tools put three special bytes at the very start of a UTF-8 file to announce the encoding. That is the **byte order mark**, and it causes its own set of problems.

20.4.5 TECHNICAL – the engineer's version

1. Carriage return is US-ASCII 0x0D, decimal 13. Line feed is 0x0A, decimal 10. Both are in the C0 control set standardized in ASCII, published in 1963 and revised in 1967.
2. The two-code sequence comes from teleprinters such as the Teletype Model 33, introduced in 1963, which needed roughly 200 milliseconds to return the carriage. Sending two codes gave the mechanism time to finish.

3. Multics and then Unix reduced this to a single 0x0A in the file and let the terminal driver expand it, which is why the C standard library still has the notion of translating in text mode.
4. Line-ending conventions by system:

System	Bytes	Escape	Era
Unix, Linux, macOS X+	0A	\n	1970 onward
Windows, DOS, HTTP	0D 0A	\r\n	1981 onward
Classic Mac OS 1-9	0D	\r	1984 to 2001

5. Several network protocols mandate CRLF regardless of platform, including HTTP/1.1 in RFC 9112, SMTP in RFC 5321, and FTP in RFC 959. This is a standard, not a convention, and sending a bare LF is a protocol violation.
6. Encodings and their byte order marks, all dumped from real files created for this chapter:

```
$ xxd -l 12 bom_utf8.txt
00000000: efbb bf68 656c 6c6f 0a                  ...hello.

$ xxd -l 12 utf16.txt
00000000: fffe 6800 6500 6c00 6c00 6f00           ..h.e.l.l.o.

$ xxd -l 6 latin1.txt
00000000: 6361 66e9 0a                            caf..

$ xxd -l 6 utf8.txt
00000000: 6361 66c3 a90a                          caf...
```

7. Read those carefully. The word `café` is five bytes in Latin-1 and six bytes in UTF-8, because the accented é needs one byte in the first and two in the second.
8. The UTF-8 byte order mark is `EF BB BF`. It encodes U+FEFF, a character that originally meant zero-width non-breaking space.
9. UTF-8 has no byte order to mark, so the mark is purely a flag. The Unicode standard permits it but does not recommend it, and POSIX tools do not expect it.
10. A UTF-8 byte order mark breaks a shebang line, breaks a shell script, breaks a JSON parser that follows RFC 8259 strictly, and appears as ï»¿ when a file is misread as Latin-1. Microsoft tools add it by default; most others do not.
11. UTF-16 requires a mark or external knowledge, because `fffe` and `feff` mean opposite byte orders. The dump above shows `fffe`, which is little-endian.
12. Useful commands: `file -i`, `iconv -f X -t Y`, `dos2unix`, `unix2dos`, `sed -i 's/\r$//'`, and in Git, `core.autocrlf` and a `.gitattributes` file with `* text=auto`.
13. The honest version: `file` reporting "ASCII text" is a guess based on statistics. It checks whether all bytes fall in printable and common control ranges. A binary file made only of printable bytes will be called text, and a UTF-8 file truncated mid-character will be called data.

20.4.6 WORDS - remember these

1. Text file — a file whose bytes read as readable characters — a byte stream interpretable under a character encoding and divided by line terminators.
2. CR — carriage return — US-ASCII 0x0D, historically returning the print head to column zero.
3. LF — line feed — US-ASCII 0x0A, historically advancing the paper one line.
4. CRLF — the Windows and network line ending — the two-byte sequence 0x0D 0x0A, mandated by HTTP, SMTP and FTP.
5. Encoding — the rule mapping bytes to characters — a character encoding scheme such as US-ASCII, ISO-8859-1 or UTF-8.

6. BOM — three or two marker bytes at the start of a text file — the byte order mark, U+FEFF, encoded as EF BB BF in UTF-8.
7. Mojibake — text that has turned into garbage symbols — the result of decoding bytes with a different encoding than they were written in.

20.5 Common formats and how they are built

20.5.1 PLAIN - in simple words

1. Most real formats are built from two separate ideas: a **container** and the things inside it.
2. The container decides how to store several pieces, where each one starts, and what each one is called.
3. The pieces inside can each be squeezed or encoded in their own way.
4. A ZIP file is the most successful container ever made. It holds a list of named items, each optionally compressed.
5. That is why so many things you use every day are secretly ZIP files.
6. A Word document, an Excel sheet, a PowerPoint deck, a Java program and an Android app are all ZIP files with agreed contents inside.
7. A PDF is a different shape. It is a numbered list of objects with a directory at the end telling you where each object lives.
8. Then there are the small human-readable data formats, JSON, XML, YAML, CSV and TOML, each with a different set of traps.

20.5.2 PLAIN - a picture in your head

1. Think of a filing cabinet with a card index drawer at the bottom.
2. Each folder in the cabinet has its own label card stapled to the front, and the contents may be folded up tightly to save space.
3. The card index at the bottom lists every folder and says exactly how far in it sits.
4. To find one folder, you read the index, walk straight to that spot, and pull it out. You never touch the others.
5. To add a folder, you push it in at the end, then write a new index and put it after everything. The old index is simply ignored.
6. That is a ZIP file exactly: local labels on each item, and a central directory at the very end.

Where this comparison breaks: with a real cabinet you would throw the old index away. A ZIP file keeps it, sitting in the middle of the file, unreferenced. That leftover is not harmless: some tools read the file forwards and see the old listing, while correct tools read the central directory at the end and see the new one. Attackers have used this exact disagreement to smuggle content past scanners. It is known in the Android world as one of the master key bugs.

20.5.3 PLAIN - a worked example

1. A Microsoft Word document is a ZIP file. Here is a real one, created with a library, then opened with a plain unzip tool.

```
$ file report.docx
report.docx: Microsoft Word 2007+

$ xxd -l 16 report.docx
00000000: 504b 0304 1400 0000 0800 2e0f 0d5d ad52  PK...........].R

$ unzip -l report.docx
  Length      Date    Time    Name
---------  ---------- -----   ----
     1738  2026-08-13 01:57   [Content_Types].xml
      734  2026-08-13 01:57   _rels/.rels
```

```
      721  2026-08-13 01:57   docProps/core.xml
     1132  2026-08-13 01:57   docProps/app.xml
     1693  2026-08-13 01:57   word/document.xml
     1227  2026-08-13 01:57   word/_rels/document.xml.rels
   349458  2026-08-13 01:57   word/styles.xml
   438131  2026-08-13 01:57   word/stylesWithEffects.xml
     2535  2026-08-13 01:57   word/settings.xml
     2811  2026-08-13 01:57   word/fontTable.xml
    10939  2026-08-13 01:57   word/theme/theme1.xml
     5513  2026-08-13 01:57   word/numbering.xml
     8324  2026-08-13 01:57   docProps/thumbnail.jpeg
---------                     -------
   826305                     17 files
```

2. The first four bytes are `504b 0304`, the ZIP local file header signature. It is a ZIP and nothing else.
3. Your actual words live in `word/document.xml`. Everything else is styles, relationships, fonts, and a preview image.
4. Look at the sizes. The text of a two-line document is 1693 bytes. The style definitions are 349458 bytes. That is why an empty Word file is not empty.
5. The `docProps/thumbnail.jpeg` entry is a real JPEG, which is why a document can leak a picture of an earlier version of itself.
6. You can extract, edit the XML with a text editor, and re-zip, and Word will open the result. This is how many document-generation tools work.

20.5.4 PLAIN – what is really happening inside

1. A ZIP file is stored as three kinds of record, in this order in the file.
2. First, for every item: a **local file header** giving the name, the compression method, the sizes and a checksum, immediately followed by the item's bytes.
3. Then, after all the items: a **central directory**, one record per item, repeating the same information plus the exact offset where that item's local header starts.
4. Last, a small **end of central directory record** saying how many entries there are and where the central directory begins.
5. A correct reader starts at the end, finds the end record, jumps to the central directory, and reads the list. It never scans forwards.
6. That design has three consequences worth knowing.
7. You can pull out one file from a five-gigabyte archive without reading the rest, because you know the exact offset.
8. You can add data before the archive without breaking it, because all offsets are found through the end record, which self-corrects. That is how self-extracting archives work: a program at the front, a ZIP at the back.
9. You can append junk after the archive and many tools still open it, because they search backwards for the end record.

20.5.5 TECHNICAL – the engineer's version

1. ZIP was created by Phil Katz in 1989 for PKZIP. The specification is Katz's APPNOTE.TXT, maintained by PKWARE, not by a standards body. It is a de facto standard.
2. Signature values, all confirmed by parsing a real archive built for this chapter: local header `0x04034B50`, central directory `0x02014B50`, end of central directory `0x06054B50`. Stored little-endian, so `PK\3\4` on disk.
3. The structure of a real three-entry archive, parsed with a short Python script written for this chapter:

```
total size: 556 bytes
LOCAL   @    0 name=note.txt  method=0 csize=  26 usize=  26
LOCAL   @   92 name=data.csv  method=0 csize=  12 usize=  12
```

```
LOCAL   @  170 name=tiny.png  method=8 csize=  64 usize=  69
EOCD    @  534 entries=3 cd_size=234 cd_offset=300
CENTRAL @  300 name=note.txt  local_hdr_at=   0
CENTRAL @  378 name=data.csv  local_hdr_at=  92
CENTRAL @  456 name=tiny.png  local_hdr_at= 170
```

4. Method 0 is Stored, meaning no compression. Method 8 is Deflate. The tool chose Stored for the tiny text files because compressing them made them larger.
5. Deflate is specified in RFC 1951, published in 1996 by Peter Deutsch. It combines LZ77 matching over a 32 KiB window with Huffman coding.
6. Appending to an archive was tested directly:

```
$ cp demo.zip appended.zip
$ printf 'EXTRA_TRAILING_DATA' >> appended.zip
$ unzip -l appended.zip
       26  note.txt
       12  data.csv
       69  tiny.png
```

The archive still lists correctly with 19 extra bytes glued on the end. 7. Prepending was also tested. Concatenating a Linux executable and a ZIP produced a file that `file` calls an ELF executable and `unzip` reads as a valid archive. Both tools are correct. 8. The original ZIP format uses 32-bit sizes and offsets and a 16-bit entry count, capping it at 4 GiB and 65535 entries. Zip64 extensions, added to APPNOTE around 2001, raise those to 64-bit fields. 9. ZIP-based formats and what identifies each:

Extension	Contents marker	Standard
.docx .xlsx .pptx	[Content_Types].xml	ECMA-376, ISO 29500
.odt .ods .odp	mimetype entry first	ISO/IEC 26300
.jar .war	META-INF/MANIFEST.MF	Java SE spec
.apk	AndroidManifest.xml	Android platform
.epub	mimetype entry first	EPUB 3, W3C
.xpi	manifest.json	Mozilla add-on

10. OOXML, the DOCX family, was standardized as ECMA-376 in 2006 and ISO/IEC 29500 in 2008. OpenDocument was standardized by OASIS in 2005 and as ISO/IEC 26300 in 2006.
11. PDF has a completely different shape. Adobe released PDF 1.0 in 1993. It became an open ISO standard, ISO 32000-1, in 2008, and ISO 32000-2, known as PDF 2.0, in 2017.
12. A PDF is a header, then a body of numbered indirect objects, then a cross-reference table listing the byte offset of every object, then a trailer. Real tail of a 33-page PDF found on this machine:

```
0000683320 00000 n
0000683847 00000 n
trailer
<<
/Size 302
/Root 301 0 R
/Info 300 0 R
>>
startxref
683904
%%EOF
```

13. `startxref` gives the byte offset of the cross-reference table, and `%%EOF` ends the file. A reader also starts at the end here, exactly like ZIP.

14. `/Root 301 0 R` is an indirect reference to object 301, which is the document catalogue. The `0` is a generation number used by incremental updates.
15. PDF supports incremental update: append new objects and a new cross-reference table, leaving the old ones in place. This is why a redacted PDF can still contain the unredacted original.
16. The small data formats compared, with the trap that catches people:

Format	Year, spec	Main trap
JSON	RFC 8259, 2017	no comments, no trailing comma
XML	W3C Rec, 1998	entity expansion attacks
YAML	1.2, 2009	indentation and type guessing
CSV	RFC 4180, 2005	quoting, commas, encodings
TOML	1.0.0, 2021	deep nesting is awkward

17. Detail on each trap. JSON forbids comments and trailing commas by design, and RFC 8259 requires UTF-8 for network interchange. JSON numbers have no defined precision limit, so a 64-bit integer can lose accuracy in a parser that uses doubles.
18. XML supports entity definitions. A document defining nested entities that each expand ten times can expand to gigabytes from a few kilobytes. That is the billion laughs attack, first publicized in 2003. Parsers must disable external entity resolution, or an XML file can read local files off a server.
19. YAML 1.1 interpreted unquoted `NO`, `ON`, `OFF` and `YES` as booleans. In a country list, `NO` for Norway became `false`. YAML 1.2's core schema fixed this in 2009, but many libraries still default to 1.1 behaviour. Check your library, not the specification.
20. CSV is not one format. RFC 4180 is informational and describes common practice: comma separator, CRLF line endings, double quotes doubled inside quoted fields. Real files use semicolons in European locales, tabs, and embedded newlines inside quoted fields.
21. TOML was created by Tom Preston-Werner in 2013 and reached version 1.0.0 in January 2021. It is used by Rust's Cargo and by Python packaging in `pyproject.toml`.
22. Containers versus codecs, restated: a `.mp4` file is a container defined by ISO/IEC 14496-12. It says nothing about how the video is encoded. Inside it can be H.264, H.265, AV1 or several others, plus AAC or Opus audio. "It is an MP4" tells you nothing about whether a device can play it.

20.5.6 WORDS - remember these

1. Container — a format that holds other pieces — a structure defining framing, naming and offsets for enclosed streams.
2. Codec — the rule for squeezing one stream — coder-decoder, the algorithm encoding a media stream inside a container.
3. Central directory — the index at the end of a ZIP — the authoritative list of entries with their local header offsets.
4. Deflate — the usual ZIP compression method — LZ77 with a 32 KiB window plus Huffman coding, specified in RFC 1951.
5. Indirect object — a numbered item in a PDF — an object addressed as "number generation R" and located through the cross-reference table.
6. Incremental update — adding to a PDF without rewriting it — appending new objects and a new xref section pointing back to the previous one.
7. Billion laughs — an XML file that explodes in size — a recursive entity expansion denial-of-service attack.

20.6 What an executable file really is

20.6.1 PLAIN – in simple words

1. There is nothing magical about a program file. It is a file, like a photo or a letter.
2. What makes it special is what the bytes are, and a small label at the front that explains how to arrange them.
3. Most of the bytes are **machine instructions**: the exact numeric codes that the processor knows how to obey.
4. The rest is data the program needs: text it will print, tables it will read, space it will want.
5. The label at the front is called a **header**. It says things like "the code starts at this offset and should be placed at this address" and "start running at this point".
6. The part of the operating system that reads that header and does what it says is called the **loader**.
7. So running a program is: read a header, copy or map the pieces into memory in the right places, then jump to the starting point.
8. Three families of headers dominate the world, one per operating system, and they do the same job in different words.
9. Windows uses PE. Linux uses ELF. macOS uses Mach-O.

20.6.2 PLAIN – a picture in your head

1. Think of a flat-pack wardrobe delivered in a box.
2. The box holds planks, screws and panels. None of it is a wardrobe yet.
3. On top sits an instruction sheet: put panel A here, panel B there, this side faces the wall, start with step one.
4. The planks are the machine instructions and data. The instruction sheet is the header.
5. The person following the sheet is the loader. It never invents anything. It only does what the sheet says.
6. A wardrobe from a different manufacturer comes with a different style of sheet. The planks might be almost identical. The sheet is not.
7. Give a Swedish sheet to a fitter trained on a different system and nothing gets built, even though the wood is fine.

Where this comparison breaks: a wardrobe is assembled once and then stands on its own. A program is re-assembled from the file every single time you run it, and usually most of the planks are never actually moved. The loader mostly just says "pretend these pages of the file are at these addresses" and the pages are read from disk only when first touched. It is less like building a wardrobe and more like handing someone a catalogue and promising to fetch each plank the moment they reach for it.

20.6.3 PLAIN – a worked example

1. We compiled the same six-line C program twice on the same Linux machine, once for Linux and once for Windows, and asked what each file is.

```
$ gcc -O2 -o hello hello.c
$ x86_64-w64-mingw32-gcc -O2 -o hello.exe hello.c

$ file hello
hello: ELF 64-bit LSB pie executable, x86-64, version 1
       (SYSV), dynamically linked, interpreter
       /lib64/ld-linux-x86-64.so.2, not stripped

$ file hello.exe
hello.exe: PE32+ executable (console) x86-64,
           for MS Windows, 19 sections

$ ls -l hello hello.exe
```

```
-rwxr-xr-x 1 root root  15960 hello
-rwxr-xr-x 1 root root 249671 hello.exe
```

2. Same source code. Same processor family, x86-64. Same instructions inside for the parts we wrote.
3. Yet the Linux file will not run on Windows and the Windows file will not run on Linux without a compatibility layer.
4. The reason is not the instructions. It is the header format and, more importantly, which library functions the file expects to find.
5. The Windows file is fifteen times larger here because the cross-compiler statically included more of its runtime and debug information.
6. Here is the first line of each file, side by side:

```
hello      7f45 4c46 0201 0100  .ELF....
hello.exe  4d5a 9000 0300 0000  MZ......
```

7. Four bytes and two bytes. That is the visible difference between an entire operating system's idea of a program and another's.

20.6.4 PLAIN - what is really happening inside

1. A processor does not know about files. It knows about memory addresses and instruction bytes.
2. Every program file therefore has to answer four questions for the loader.
3. Question one: which processor are these instructions for? An ARM chip cannot run x86 codes, so the header names the architecture.
4. Question two: which pieces of the file go where in memory? The header lists chunks, each with a file offset, a memory address, a length, and permissions.
5. Question three: what permissions does each chunk need? Code needs to be readable and executable but not writable. Data needs to be readable and writable but not executable.
6. Question four: where do I start? The header holds one address, the **entry point**.
7. There is a fifth question for most modern programs: which shared libraries do I need? The header lists them by name.
8. The loader answers these in order, builds a fresh empty memory space for the new program, maps in the chunks, and jumps to the entry point.
9. That is the entire trick. Everything else in an executable format is bookkeeping around those five answers.

20.6.5 TECHNICAL - the engineer's version

1. The three formats and their lineage:

Format	Origin	Used by
PE/COFF	Windows NT 3.1, 1993	Windows, UEFI
ELF	System V R4, circa 1989	Linux, BSD, Solaris
Mach-O	Mach at CMU, 1980s	macOS, iOS

2. PE stands for Portable Executable. It extends COFF, the Common Object File Format from AT&T Unix System V, and was introduced with Windows NT 3.1 in 1993. The specification is published by Microsoft as the PE Format document.
3. ELF stands for Executable and Linkable Format. It was defined by Unix System Laboratories for System V Release 4 and published in the System V Application Binary Interface. Linux moved from the older a.out format to ELF during 1995 and 1996.
4. Mach-O comes from the Mach microkernel project at Carnegie Mellon University, was used by NeXTSTEP from 1989, and was inherited by Mac OS X in 2001.

5. A structural comparison:

Concept	PE name	ELF name
Shared library	.dll	.so
Static library	.lib	.a
Loadable unit	section	segment
Import list	Import Directory	DT_NEEDED entries
Start address	AddressOfEntryPoint	e_entry
Base address	ImageBase	p_vaddr of first LOAD
Runtime linker	ntdll loader	ld.so

6. Mach-O uses `.dylib` for shared libraries, `.a` for static archives, load commands instead of a fixed table, `LC_MAIN` for the entry point and `/usr/lib/dyld` as the dynamic linker.
7. All three support position-independent code so that the operating system can place the image at a randomized base address. This is Address Space Layout Randomization, standard on Windows since Vista in 2007, on Linux since the mid-2000s, and on macOS since 10.5 in 2007.
8. UEFI firmware executables also use PE/COFF, with subsystem values 10 to 13. This is why a Linux bootloader signed for Secure Boot is a PE file even though Linux itself uses ELF.
9. There is genuine disagreement among engineers about whether a modern executable format should keep separate section and segment tables at all. ELF keeps both, one for the linker and one for the loader. Some newer formats such as WebAssembly use a single section list. The ELF side argues the split allows stripping without touching load behaviour; the critics argue it duplicates information and has caused parser confusion bugs.

20.6.6 WORDS - remember these

1. Executable — a file the system can run — an image containing machine code plus a header describing its memory layout and entry point.
2. Header — the label at the front of a program file — a fixed structure giving architecture, layout tables and the entry point.
3. Loader — the part of the system that sets a program up — kernel code that parses the image, creates an address space and maps segments.
4. Entry point — where execution begins — the virtual address stored in `e_entry` on ELF or `AddressOfEntryPoint` on PE.
5. Machine code — the numbers a processor obeys directly — encoded instructions for one instruction set architecture.
6. ASLR — placing a program at a random address each run — Address Space Layout Randomization, requiring position-independent code and relocations.

20.7 Anatomy of a PE file

20.7.1 PLAIN - in simple words

1. Open any Windows `.exe` in a hex viewer and the first two letters are `MZ`.
2. Those are the initials of Mark Zbikowski, a Microsoft engineer who designed the executable header for MS-DOS in the early 1980s.
3. Right after that sits a tiny complete DOS program. Its only job is to print "This program cannot be run in DOS mode" and quit.
4. That stub is still there in 2026, in every Windows program you own, even though no one has run DOS in decades.
5. About 128 bytes in comes the letters `PE` followed by two zero bytes. That is where the modern file really starts.

6. After that come three tables: one describing the machine and how many pieces there are, one describing the whole image, and one listing the pieces.
7. Each piece is called a **section**, and each has a short name beginning with a dot: `.text` for code, `.data` for changeable data, `.rdata` for read-only data.
8. There is also a list of functions the program needs from other files, and if it is a library, a list of what it offers.
9. And one small number decides whether Windows opens a black console window for the program or not.

20.7.2 PLAIN - a picture in your head

1. Think of a very old building that has been refurbished many times.
2. At the front door there is still a Victorian gas lamp bracket. It has not carried gas since 1930. It is bolted to the wall and nobody removes it.
3. It costs almost nothing to leave, and taking it off risks damaging the wall.
4. The DOS stub is that gas lamp bracket. Sixty-four bytes of dead code plus a sentence.
5. Once you walk past it, the building inside is completely modern: a directory board in the lobby listing every floor, and every floor labelled by purpose.
6. The directory board is the section table. Each floor is a section.

Where this comparison breaks: the gas bracket really is useless, whereas the DOS stub has one live job left. The last four bytes before it, at offset 0x3C, hold the offset of the real PE header. Every Windows loader reads that number. So the stub region is not dead space; it contains the pointer that makes the whole format work. Removing the stub without fixing that pointer breaks the file.

20.7.3 PLAIN - a worked example

1. Here are the first 128 bytes of a real Windows executable we compiled for this chapter.

```
$ xxd -l 128 hello.exe
00000000: 4d5a 9000 0300 0000 0400 0000 ffff 0000  MZ..............
00000010: b800 0000 0000 0000 4000 0000 0000 0000  ........@.......
00000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000030: 0000 0000 0000 0000 0000 0000 8000 0000  ................
00000040: 0e1f ba0e 00b4 09cd 21b8 014c cd21 5468  ........!..L.!Th
00000050: 6973 2070 726f 6772 616d 2063 616e 6e6f  is program canno
00000060: 7420 6265 2072 756e 2069 6e20 444f 5320  t be run in DOS
00000070: 6d6f 6465 2e0d 0d0a 2400 0000 0000 0000  mode....$.......
```

2. Bytes 0 and 1 are `4d 5a`, the letters `MZ`. That is the DOS header magic.
3. Look at offset `0x3c`, which is on the fourth line: the four bytes `8000 0000`. Little-endian, that is the number `0x00000080`, decimal 128.
4. That is the promise: the real header begins at byte 128 of this file.
5. Between offset `0x40` and `0x78` sits the stub. The bytes `0e 1f ba 0e 00 b4 09 cd 21` are real 16-bit DOS instructions.
6. Decoded: push cs, pop ds, load the address of the message into dx, put 9 in ah, then `int 21h` which is the DOS system call to print a string.
7. Then `b8 01 4c cd 21` is: put `0x4C01` in ax and call DOS again, which means terminate with exit code 1.
8. Then the message itself, ending with `0d 0d 0a 24`. The `24` is a dollar sign, which is how DOS marked the end of a string.
9. Now jump to byte 128 and look:

```
$ xxd -s 128 -l 32 hello.exe
00000080: 5045 0000 6486 1300 5e24 7d6a 0032 0300  PE..d...^$}j.2..
00000090: 5307 0000 f000 2600 0b02 0229 006c 0000  S.....&....).l..
```

10. `5045 0000` is `PE\0\0`. The modern file starts here, exactly where the pointer said.

20.7.4 PLAIN - what is really happening inside

1. Read the bytes after `PE\0\0` in order and every field tells you something useful.
2. `6486` little-endian is `0x8664`, the code for the x86-64 processor family. An ARM Windows binary would say `0xAA64`.
3. `1300` is `0x0013`, decimal 19. The file has nineteen sections.
4. `5e24 7d6a` is `0x6A7D245E`, a Unix timestamp. Converted, that is 13 August 2026 at 01:56:46 UTC, which is when we compiled it.
5. `f000` is `0x00F0`, decimal 240: the size of the next header.
6. `0b02` is `0x020B`, the marker meaning this is a 64-bit image. A 32-bit one would say `0x010B`.
7. The 240-byte header that follows holds the entry point, the preferred load address, the alignment rules, the stack size, and a table of sixteen directories pointing at imports, exports, resources and relocations.
8. After that comes the section table: nineteen fixed-size records, each with a name, a size, a memory address, and a file offset.
9. The loader reads exactly this and nothing more before it starts mapping.

20.7.5 TECHNICAL - the engineer's version

1. Real dump of the optional header of the file compiled for this chapter, trimmed to fit the page:

```
$ objdump -x hello.exe
architecture: i386:x86-64
start address 0x0000000140001410
Characteristics 0x26
  executable, line numbers stripped, large address aware
Time/Date        Thu Aug 13 01:56:46 2026
Magic            020b   (PE32+)
SizeOfCode               0000000000006c00
SizeOfInitializedData    0000000000009600
AddressOfEntryPoint      0000000000001410
BaseOfCode               0000000000001000
ImageBase                0000000140000000
SectionAlignment         00001000
FileAlignment            00000200
MajorSubsystemVersion    5
SizeOfImage              0003f000
CheckSum                 00041982
Subsystem                00000003   (Windows CUI)
DllCharacteristics       00000160
    HIGH_ENTROPY_VA, DYNAMIC_BASE, NX_COMPAT
SizeOfStackReserve       0000000000200000
SizeOfHeapReserve        0000000000100000
NumberOfRvaAndSizes      00000010
```

2. `Subsystem 3` is `IMAGE_SUBSYSTEM_WINDOWS_CUI`, a console program. Value 2 is `IMAGE_SUBSYSTEM_WINDOWS_GUI`. This single 16-bit field is the entire difference between a program that opens a black console window and one that does not.
3. `ImageBase 0x140000000` is the default preferred base for a 64-bit executable. A 32-bit executable defaults to `0x400000` and a DLL to `0x10000000`.
4. `DYNAMIC_BASE` means the image opts in to ASLR, so the loader will place it elsewhere and apply base relocations. `NX_COMPAT` means data pages can be marked non-executable. `HIGH_ENTROPY_VA` allows 64-bit address randomization.
5. `SectionAlignment 0x1000` is the page size in memory. `FileAlignment 0x200` is the alignment on disk. Sections are therefore padded differently in the file and in memory, which is why file offsets and virtual addresses differ.

6. The section table from the same binary:

```
$ objdump -h hello.exe
Idx Name          Size      VMA               File off
  0 .text         00006b68  0000000140001000  00000600
  1 .data         000000c0  0000000140008000  00007200
  2 .rdata        00000da0  0000000140009000  00007400
  3 .pdata        00000468  000000014000a000  00008200
  4 .xdata        0000050c  000000014000b000  00008800
  5 .bss          00000b80  000000014000c000  00000000
  6 .idata        000006d0  000000014000d000  00008e00
  7 .CRT          00000060  000000014000e000  00009600
  8 .tls          00000010  000000014000f000  00009800
  9 .reloc        00000084  0000000140010000  00009a00
 10 .debug_aranges ...
```

7. What the standard sections hold:

Section	Contents	Permissions
.text	machine code	read, execute
.rdata	constants, import tables	read only
.data	initialized variables	read, write
.bss	zero-filled variables	read, write
.rsrc	icons, strings, manifest	read only
.reloc	base relocation entries	read only
.idata	import descriptors	read
.edata	export descriptors	read
.pdata	unwind function table	read only

8. Note `.bss` has file offset `00000000`. It occupies no space in the file at all. The loader simply allocates zeroed pages of the stated size. This is why a program with a 100 MB static buffer can be a 50 KB file.
9. The import table, dumped from the same binary:

```
$ objdump -p hello.exe
  DLL Name: KERNEL32.dll
  d350   281  DeleteCriticalSection
  d368   317  EnterCriticalSection
  d380   628  GetLastError
  d3d8  1034  MultiByteToWideChar
  d422  1489  VirtualProtect
  DLL Name: msvcrt.dll
  d49e    82  __getmainargs
  d4c8    97  __set_app_type
  d51c   219  _fmode
  d526   285  _initterm
```

10. Each imported name has a slot in the **Import Address Table**. Before loading, each slot holds a pointer to the name. After loading, the loader overwrites each slot with the real address of the function inside the loaded DLL. Calls go through the slot, so they cost one extra memory read.
11. The IAT location is Data Directory entry 12. In this file it is at RVA `0xd1c8`, length `0x188`, which is 392 bytes, or 49 pointers.
12. Because the IAT is a table of writable function pointers, hooking it is a classic technique for both debugging tools and malware. Overwrite the slot and every call to that function goes to your code instead.
13. A DLL adds an export table. We built a real DLL with two functions:

```
$ x86_64-w64-mingw32-gcc -shared -O2 -o mylib.dll mylib.c
$ file mylib.dll
mylib.dll: PE32+ executable (DLL) (console) x86-64,
           for MS Windows, 20 sections
$ objdump -p mylib.dll
Export Address Table -- Ordinal Base 1
  [0] +base[1] 1390 Export RVA
  [1] +base[2] 13a0 Export RVA
[Ordinal/Name Pointer] Table
  [0] add
  [1] mul
```

14. Every export has both a name and an ordinal number. Importing by ordinal is smaller and faster but breaks if the library is rebuilt with a different order. Importing by name is the normal choice.
15. The addresses in these tables are **RVAs**, relative virtual addresses: offsets from `ImageBase`. Add `0x140000000` to `0x1390` to get the real address if the image loads at its preferred base.
16. Tools that read PE files: `objdump -x` and `objdump -p` from GNU binutils, `dumpbin /headers` from Microsoft, `llvm-readobj --coff-all`, and the Python `pefile` library.

20.7.6 WORDS - remember these

1. MZ header — the two letters at the start of every Windows program — the DOS `IMAGE_DOS_HEADER`, whose `e_lfanew` field at offset 0x3C locates the PE header.
2. DOS stub — the small dead program after the MZ header — 16-bit code invoking `int 21h` to print a message and terminate.
3. Section — one labelled region of a program image — a record in the section table with a name, RVA, size, file offset and characteristic flags.
4. RVA — an address measured from the start of the image — Relative Virtual Address, added to `ImageBase` to get a linear address.
5. IAT — the table of addresses of borrowed functions — the Import Address Table, patched by the loader with resolved function pointers.
6. Subsystem — the flag deciding console or window — a 16-bit field; 2 is `WINDOWS_GUI`, 3 is `WINDOWS_CUI`, 10 to 13 are UEFI.
7. Base relocation — a fix-up applied when the image loads elsewhere — an entry in `.reloc` naming an address that must be adjusted by the load delta.

20.8 Anatomy of an ELF file

20.8.1 PLAIN - in simple words

1. Every Linux program begins with byte 127 followed by the letters `E`, `L`, `F`. That is the whole signature.
2. After it comes a small header of 64 bytes that answers the basic questions: 32-bit or 64-bit, which byte order, which processor, where to start.
3. Then the file has two separate tables of contents, and this surprises people.
4. One table is for the loader: it lists a handful of big chunks to map into memory. These are called **segments**.
5. The other table is for the linker and for tools: it lists many small named pieces such as `.text` and `.data`. These are called **sections**.
6. Both tables describe the same bytes, grouped differently, for two different readers.
7. When a program runs, only the first table matters. The second can be deleted entirely and the program still runs.
8. Most Linux programs also name a helper program inside themselves: the dynamic linker, which finds the shared libraries at start-up.
9. That helper is usually `/lib64/ld-linux-x86-64.so.2`, and it is itself an ELF file.

20.8.2 PLAIN – a picture in your head

1. Think of a large printed cookbook that has two indexes at the back.
2. The first index is by chapter: Starters, Mains, Puddings. Three entries, big blocks, useful when you want to open the book roughly in the right place.
3. The second index is by recipe name: four hundred entries, precise, useful when you want one exact thing.
4. Both indexes describe the same pages. Neither is wrong. They serve different readers.
5. A cook in a hurry uses the chapter index. A librarian cataloguing the book uses the recipe index.
6. The loader is the cook in a hurry: it wants three or four big blocks and the permissions for each.
7. The linker and the debugger are the librarian: they want the exact name and boundary of every small piece.

Where this comparison breaks: you cannot tear the recipe index out of a cookbook and still have the chapter index work, because they refer to page numbers that would not change. In ELF you genuinely can delete the whole section table. The `strip` command does this, the file gets smaller, and the program runs identically. There is no equivalent to that in a book, and it is the clearest proof that the two tables truly serve different masters.

20.8.3 PLAIN – a worked example

1. Here is the start of a real Linux executable compiled for this chapter, sixty-four bytes, which is exactly the ELF header.

```
$ xxd -l 64 hello
00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF............
00000010: 0300 3e00 0100 0000 8010 0000 0000 0000  ..>.............
00000020: 4000 0000 0000 0000 9836 0000 0000 0000  @........6......
00000030: 0000 0000 4000 3800 0d00 4000 1f00 1e00  ....@.8...@.....
```

2. `7f 45 4c 46` is the magic: 127, `E`, `L`, `F`.
3. `02` means 64-bit. `01` would mean 32-bit.
4. The next `01` means little-endian byte order. `02` would mean big-endian.
5. `01` again is the ELF version, which has been 1 since 1989 and has never changed.
6. `0300` is `0x0003`, the type. Type 3 is a shared object, which is what a modern position-independent executable is.
7. `3e00` is `0x003E`, decimal 62, the code for x86-64.
8. `8010 0000 ...` is the entry point: address `0x1080`.
9. `4000 ...` is 64: the program header table starts at byte 64, immediately after this header.
10. `9836 0000` is `0x3698`, decimal 13976: where the section header table starts, near the end of the file.
11. And the tool agrees with our hand reading:

```
$ readelf -h hello
  Class:                             ELF64
  Data:                              2's complement, little endian
  Type:                              DYN (Position-Independent Executable)
  Machine:                           Advanced Micro Devices X86-64
  Entry point address:               0x1080
  Start of program headers:          64 (bytes into file)
  Start of section headers:          13976 (bytes into file)
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         31
```

20.8.4 PLAIN – what is really happening inside

1. Thirteen program headers, thirty-one section headers. Only the thirteen matter at run time, and of those only four are of type LOAD.
2. Each LOAD entry says: take this many bytes from this file offset, put them at this virtual address, and give them these permissions.
3. One LOAD entry has read and execute permission. That is the code. One has read and write. That is the data.
4. A separate entry of type INTERP holds a plain text string: the path of the dynamic linker. On this machine it is `/lib64/ld-linux-x86-64.so.2`.
5. If that entry is present, the kernel loads the named program too, and gives control to it, not to your program.
6. The dynamic linker then reads a special region called the **dynamic section**, which is a list of tagged values.
7. One tag, `NEEDED`, appears once per required library. Our program has exactly one: `libc.so.6`.
8. Other tags point at the symbol table, the string table, the relocation lists, and the initialization function.
9. The dynamic linker opens each needed library, maps it, then walks the relocation lists patching addresses until every borrowed function has a real address.
10. Only then does it jump to your entry point.

20.8.5 TECHNICAL – the engineer's version

1. Program headers of the same binary, as a table. `FileSiz` and `MemSiz` differ only for the writable segment, because `.bss` occupies memory but not file.

Type	File off	Virt addr	Flags
PHDR	0x000040	0x0000040	R
INTERP	0x000318	0x0000318	R
LOAD	0x000000	0x0000000	R
LOAD	0x001000	0x0001000	R E
LOAD	0x002000	0x0002000	R
LOAD	0x002db8	0x0003db8	RW
DYNAMIC	0x002dc8	0x0003dc8	RW
GNU_EH_FRAME	0x002014	0x0002014	R
GNU_STACK	0x000000	0x0000000	RW
GNU_RELRO	0x002db8	0x0003db8	R

2. `GNU_STACK` with RW and no E is what makes the stack non-executable. If this header were missing or had E set, the kernel would map an executable stack.
3. `GNU_RELRO` marks a region that the dynamic linker makes read-only after relocation is finished. It covers `.init_array`, `.fini_array`, `.dynamic` and, with full RELRO, the global offset table. This blocks a whole class of pointer-overwrite attacks.
4. The interpreter string is stored literally in the file and reported by `readelf -l` as `[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]`.
5. Section headers, abbreviated from the real output of `readelf -SW hello`:

Section	Type	Address	Size
.interp	PROGBITS	0x318	0x1c
.dynsym	DYNSYM	0x3d8	0xa8
.rela.dyn	RELA	0x550	0xc0

Section	Type	Address	Size
.rela.plt	RELA	0x610	0x18
.init	PROGBITS	0x1000	0x1b
.plt	PROGBITS	0x1020	0x20
.text	PROGBITS	0x1060	0x109
.rodata	PROGBITS	0x2000	0x11
.dynamic	DYNAMIC	0x3dc8	0x1f0
.got	PROGBITS	0x3fb8	0x48
.data	PROGBITS	0x4000	0x10
.bss	NOBITS	0x4010	0x8
.symtab	SYMTAB	0	0x360

6. `.bss` has type `NOBITS`: no bytes in the file. `.symtab` has address 0: it is never loaded into memory.
7. The dynamic section, real output, trimmed:

```
$ readelf -d hello
 (NEEDED)      Shared library: [libc.so.6]
 (INIT)        0x1000
 (FINI)        0x116c
 (INIT_ARRAY)  0x3db8
 (GNU_HASH)    0x3b0
 (STRTAB)      0x480
 (SYMTAB)      0x3d8
 (PLTGOT)      0x3fb8
 (JMPREL)      0x610
 (RELA)        0x550
 (FLAGS)       BIND_NOW
 (FLAGS_1)     Flags: NOW PIE
```

8. `BIND_NOW` with full RELRO means every symbol is resolved at start-up rather than lazily on first call. This is the default on Ubuntu and most modern distributions. Lazy binding is faster to start and weaker to attack.
9. Relocations from the same file:

```
$ readelf -r hello
Relocation section '.rela.dyn':
  R_X86_64_RELATIVE                    1160
  R_X86_64_RELATIVE                    1120
  R_X86_64_GLOB_DAT  __libc_start_main@GLIBC_2.34
  R_X86_64_GLOB_DAT  __cxa_finalize@GLIBC_2.2.5
Relocation section '.rela.plt':
  R_X86_64_JUMP_SLOT puts@GLIBC_2.2.5
```

10. `R_X86_64_RELATIVE` entries need no symbol lookup. They just say “add the load base to the value stored here”, and there are three because the image is position independent.
11. Static versus dynamic linking, measured on this machine with the same source:

Build	File size	text bytes
dynamic, PIE	15,960	1,369
static	785,360	668,329
dynamic, stripped	14,472	1,369

12. The static binary is 49 times larger because the entire C library is copied in. It has no `INTERP` header, no `NEEDED` entries, and `ldd` reports it as “not a dynamic executable”.

13. Stripping removed `.symtab` and `.strtab`, cutting 1,488 bytes and reducing the section count from 31 to 29. `nm` then reports "no symbols". The program runs identically; only debugging is harder.
14. PIE, position-independent executable, is `Type: DYN` with an entry point of `0x1080`, a small number, because the real base is decided at load time. Building with `-no-pie` produced `Type: EXEC` with entry `0x401070`, a fixed address. PIE is the default on Ubuntu since 16.10 and on Debian since Stretch in 2017.
15. Tools that read ELF: `readelf`, `objdump`, `nm`, `ldd`, `eu-readelf` from elfutils, `llvm-readelf`, and `LD_DEBUG=libs` to watch the dynamic linker work.

20.8.6 WORDS - remember these

1. Segment — one chunk the loader maps — a program header entry of type LOAD with a file offset, virtual address, sizes and permission flags.
2. Section — one named region for tools — a section header entry used by the linker, debugger and stripper, ignored at run time.
3. Dynamic section — the run-time instruction list — an array of tagged values including DT_NEEDED, DT_STRTAB, DT_RELA and DT_INIT.
4. Interpreter — the helper that starts your program — the path in the PT_INTERP header, normally `ld.so`, loaded by the kernel before your code.
5. PIE — a program with no fixed address — Position-Independent Executable, type ET_DYN, relocated to a random base by the kernel.
6. Stripping — deleting the names from a binary — removing `.symtab` and `.strtab`, shrinking the file without changing behaviour.
7. RELRO — making some data read-only after start-up — RELocation Read-Only, applied by `ld.so` via `mprotect` after relocations complete.

20.9 What happens when you double-click a program

20.9.1 PLAIN - in simple words

1. You double-click an icon. A fraction of a second later a window appears. A great deal happened in between.
2. First, something has to work out which file you meant and whether you are allowed to run it.
3. Then a request goes to the core of the operating system: please replace a process with this program.
4. The core opens the file, reads the first bytes, and checks that it recognizes the format and that the instructions are for this processor.
5. It then throws away the old program's memory entirely and builds a fresh, empty memory space.
6. It maps the pieces of the file into that space at the addresses the header asked for.
7. It writes your command line arguments and the environment settings onto the new program's stack, so the program can find them.
8. If the program needs shared libraries, the core starts a helper program first and hands control to it.
9. The helper finds each library, maps it in, and fills in every borrowed address.
10. Only then does control reach your program's real starting point, and even that is not your code yet. A short start-up routine runs first, and it calls your `main`.

20.9.2 PLAIN - a picture in your head

1. Think of an empty theatre and a script arriving at the stage door.
2. The stage manager checks the envelope: right theatre, right company, do we have permission to perform this.
3. The stage crew clears the stage completely. Everything from the last performance goes.
4. The set pieces are then brought in and placed at marked positions on the floor. The script says exactly where each goes.

5. A list of props is pinned up: the ones we own, and the ones we must borrow from the props house next door.
6. A runner is sent to the props house. He fetches each borrowed item and puts it in its marked place.
7. Only when every marked place is filled does the stage manager call “curtain up”, and the first line is spoken.

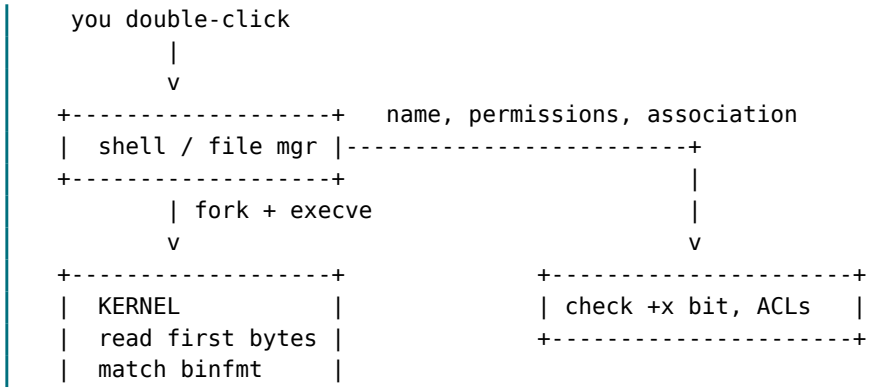
Where this comparison breaks: a real stage is fully dressed before the curtain rises. A program is not. The loader mostly writes down promises rather than moving anything. A page of code is fetched from disk only when the processor first tries to execute it, causing a page fault that the kernel quietly services. A large program can start running having read only a few dozen kilobytes of its own file. The set is built while the play is already running.

20.9.3 PLAIN – a worked example

1. We traced the system calls of the tiny program from earlier. This is the real output, lightly trimmed to fit the page.

```
$ strace ./hello
execve("./hello", ["./hello"], 0x7ffe.. /* 151 vars */) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, ...)  = 0x7ff5dfab6000
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
mmap(NULL, 54027, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7ff5dfaa8000
close(3)                                        = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\2\1\1\3\0\0..."..., 832)    = 832
mmap(NULL, 2170256, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7ff5df800000
mmap(0x7ff5df828000, 1605632, PROT_READ|PROT_EXEC, ...)
mmap(0x7ff5df9b0000, 323584, PROT_READ, ...)
mmap(0x7ff5df9ff000, 24576, PROT_READ|PROT_WRITE, ...)
mmap(0x7ff5dfa05000, 52624, PROT_READ|PROT_WRITE|ANONYMOUS)
close(3)                                        = 0
arch_prctl(ARCH_SET_FS, 0x7ff5dfaa5740)      = 0
write(1, "hello, world\n", 13)               = 13
exit_group(0)                                   = ?
+++ exited with 0 +++
```

2. Count the lines. Seventeen system calls to print thirteen characters. Only one of them, the `write`, is our program doing our work.
3. Look at the four `mmap` calls for `libc.so.6`. Each one maps a different segment with different permissions: read only, read plus execute, read only again, then read plus write.
4. The last one has `ANONYMOUS` and no file. That is `.bss`: zeroed memory backed by nothing on disk.
5. `read(3, "\177ELF...", 832)` is the dynamic linker reading the first 832 bytes of the library to parse its ELF header and program headers.
6. Here is the same journey as a picture.



```
|  new address space|
|  map LOAD segments|
|  build stack:     |   argv, envp, auxv
|  load PT_INTERP   |
+-------------------+
        | jump to ld.so entry
        v
+-------------------+
|  ld.so            |  find libs, mmap them,
|  dynamic linker   |  apply relocations, run
+-------------------+  library init functions
        | jump to e_entry
        v
+-------------------+
|  _start           |  set up argc/argv
|  __libc_start_main|  init stdio, TLS, locale
+-------------------+
        | call
        v
     main(argc, argv)
        | return
        v
 exit -> atexit handlers -> _exit -> exit_group
        |
        v
 kernel frees memory, notifies parent
```

20.9.4 PLAIN - what is really happening inside

1. Here is the whole sequence in thirty numbered steps. Steps are given for Linux; the Windows path is noted in the technical block below.
2. Step 1. The file manager receives a double-click on an icon and resolves it to a full path, following any symbolic links or shortcuts.
3. Step 2. It decides how to run it. On Linux it checks whether the file is executable; on Windows it consults the registry association for the extension.
4. Step 3. It checks permission. On Linux, the execute bit must be set for you. On Windows, an access control list is evaluated.
5. Step 4. The file manager calls `fork`, creating a child process that is a copy of itself.
6. Step 5. The child calls `execve` with the path, the argument list, and the environment list.
7. Step 6. The kernel opens the file and reads the first 256 bytes into a buffer.
8. Step 7. The kernel walks its list of registered binary format handlers, offering the buffer to each in turn.
9. Step 8. The ELF handler checks for `7f 45 4c 46` and accepts.
10. Step 9. It checks the class field, 64-bit here, and the machine field, 62 for x86-64, against the running kernel. A mismatch fails with `ENOEXEC`.
11. Step 10. The kernel checks the set-user-id and set-group-id bits and decides the new credentials.
12. Step 11. The kernel destroys the old address space: all mappings from the previous program are released.
13. Step 12. A brand-new, empty address space is created for the process.
14. Step 13. The kernel reads the program header table and iterates over the `PT_LOAD` entries.
15. Step 14. For each, it creates a mapping from the file into memory at the stated address, with the stated permissions. Nothing is copied yet.
16. Step 15. For a `PT_LOAD` whose memory size exceeds its file size, the extra is mapped as anonymous zeroed pages. That is `.bss`.
17. Step 16. For a position-independent executable, the kernel picks a random base address and adds it to every mapping address.

18. Step 17. The kernel creates the stack, at a randomized address, sized by the process resource limit, usually 8 MB.
19. Step 18. It writes onto that stack, from the top down: the environment strings, the argument strings, then pointer arrays for each, then a table of auxiliary values called the **auxiliary vector**.
20. Step 19. The auxiliary vector carries facts the program cannot easily discover otherwise: page size, the address of the program headers, the entry point, the real and effective user ids, and a pointer to sixteen random bytes.
21. Step 20. If the file has a `PT_INTERP` header, the kernel opens that path, maps its segments too, and remembers its entry point.
22. Step 21. The kernel sets the instruction pointer to the interpreter's entry point, not the program's, and returns to user mode.
23. Step 22. The dynamic linker begins. It relocates itself first, since nothing has fixed its own addresses yet.
24. Step 23. It reads the program's dynamic section and collects every `DT_NEEDED` library name.
25. Step 24. For each name, it searches: `DT_RPATH` and `DT_RUNPATH` in the file, then `LD_LIBRARY_PATH`, then the cache in `/etc/ld.so.cache`, then the default directories.
26. Step 25. Each found library is opened and mapped exactly as the main program was, and its own dependencies are added to the queue.
27. Step 26. With every object loaded, the linker resolves symbols. For each relocation entry, it looks up the symbol name in the load order and writes the resolved address into the named slot.
28. Step 27. With full RELRO enabled, it then calls `mprotect` to make the global offset table and the dynamic section read-only.
29. Step 28. It runs the initialization functions of every library, in dependency order, then those of the program itself, from `DT_INIT` and `DT_INIT_ARRAY`.
30. Step 29. It jumps to the program's real entry point, `_start`, which is not your code. `_start` arranges the arguments into registers and calls `__libc_start_main`.
31. Step 30. `__libc_start_main` sets up thread-local storage, initializes the standard input, output and error streams, registers the cleanup handler, and finally calls `main`. When `main` returns, `exit` runs the registered handlers, flushes buffers, calls destructors, and issues `exit_group`, at which point the kernel frees everything and stores the exit status for the parent.

20.9.5 TECHNICAL - the engineer's version

1. On Linux the kernel entry point is `execve(2)`, implemented in `fs/exec.c`. It fills a `struct linux_binprm`, reads `BINPRM_BUF_SIZE` bytes, currently 256, and calls `search_binary_handler`.
2. Registered handlers in a typical kernel: `binfmt_elf`, `binfmt_script`, `binfmt_misc`, and `binfmt_elf_fdpic` on some embedded targets. The old `binfmt_aout` was removed from mainline Linux in 2022.
3. Mapping is done with `vm_mmap` using `MAP_PRIVATE`, so pages are copy-on-write and shared between every process running the same binary until written.
4. The auxiliary vector entries most often used: `AT_PHDR`, `AT_PHENT`, `AT_PHNUM`, `AT_ENTRY`, `AT_PAGESZ`, `AT_RANDOM`, `AT_SECURE`, `AT_HWCAP` and `AT_HWCAP2`. You can print them with `LD_SHOW_AUXV=1 ./program`.
5. `AT_RANDOM` points at sixteen kernel-supplied random bytes. glibc uses them to seed the stack canary. This is why the canary differs on every run.
6. The dynamic linker's work can be observed directly:

```
$ LD_DEBUG=libs ./hello
  find library=libc.so.6 [0]; searching
   search cache=/etc/ld.so.cache
    trying file=/lib/x86_64-linux-gnu/libc.so.6
  calling init: /lib64/ld-linux-x86-64.so.2
  calling init: /lib/x86_64-linux-gnu/libc.so.6
  initialize program: ./hello
```

```
$ LD_DEBUG=bindings ./hello
  binding file ./hello [0] to libc.so.6 [0]:
    normal symbol `__libc_start_main' [GLIBC_2.34]
  binding file ./hello [0] to libc.so.6 [0]:
    normal symbol `puts' [GLIBC_2.2.5]
```

7. Note that our source called `printf` but the binding is to `puts`. The compiler replaced a `printf` with a constant string and no format specifiers by the cheaper `puts`. Verified in the disassembly:

```
$ objdump -d hello
0000000000001060 <main>:
  1060: f3 0f 1e fa    endbr64
  1064: 48 83 ec 08    sub    $0x8,%rsp
  1068: 48 8d 3d 95..  lea    0xf95(%rip),%rdi
  106f: e8 dc ff ff ff call   1050 <puts@plt>
  1074: 31 c0          xor    %eax,%eax
  107a: c3             ret
```

8. The resulting address space, from `/proc/PID/maps` of a running process, shows the pattern clearly: five mappings for the executable and five for libc, each with different permissions.

```
55ffa11de000-55ffa11e0000 r--p /usr/bin/sleep
55ffa11e0000-55ffa11e4000 r-xp /usr/bin/sleep
55ffa11e4000-55ffa11e5000 r--p /usr/bin/sleep
55ffa11e6000-55ffa11e7000 rw-p /usr/bin/sleep
55ffdd305000-55ffdd326000 rw-p [heap]
7f9296600000-7f9296628000 r--p libc.so.6
7f9296628000-7f92967b0000 r-xp libc.so.6
7f92967b0000-7f92967ff000 r--p libc.so.6
7f9296803000-7f9296805000 rw-p libc.so.6
```

9. On Windows the equivalent path is: `CreateProcess` in kernel32 calls `NtCreateUserProcess` in ntdll, the kernel creates a section object for the image, maps it, and starts a thread at `RtlUserThreadStart`. The user-mode loader in ntdll, often called `LdrpInitializeProcess`, then walks the import directory, loads each DLL, calls each DLL's `DllMain` with `DLL_PROCESS_ATTACH`, patches the IAT, and finally jumps to `AddressOfEntryPoint`.
10. On macOS the kernel's `execve` recognizes Mach-O, maps segments, and loads `/usr/lib/dyld`, which reads `LC_LOAD_DYLIB` commands, consults the shared cache, binds symbols and calls `LC_MAIN`.
11. Approximate cost on the machine used here: `execve` plus dynamic linking for a one-library program completes in under two milliseconds. A large application linking against forty shared libraries can spend tens of milliseconds purely in symbol resolution, which is why `prelink` existed and why static linking is popular again for command-line tools.
12. Tools that observe each stage: `strace -f`, `ltrace`, `LD_DEBUG`, `LD_SHOW_AUXV`, `perf trace`, `/proc/PID/maps`, and on Windows Process Monitor and the `!dlls` command in WinDbg.

20.9.6 WORDS - remember these

1. exec — replacing a process with a new program — the `execve(2)` family, which discards the old address space and never returns on success.
2. fork — making a copy of a process — `fork(2)`, creating a child with a copy-on-write duplicate of the parent's address space.
3. Auxiliary vector — a list of facts the kernel hands the new program — the `auxv` array of type-value pairs on the initial stack.
4. Page fault — the moment a page is actually fetched — a processor exception the kernel services by reading from the mapped file or allocating a zero page.

5. Copy-on-write — sharing memory until someone writes — mapping pages read-only and duplicating them on the first write fault.
6. Lazy binding — resolving a function's address on first call — PLT stubs that call into the linker once, then patch the GOT slot.
7. C runtime start-up — the code that runs before `main` — `_start` and `__libc_start_main`, which initialize TLS, stdio and `atexit`.

20.10 Installers and packaging

20.10.1 PLAIN - in simple words

1. An installer is just a program whose job is to put other files in the right places.
2. There is nothing sacred about installing. You could do all of it by hand with a file manager, given enough patience.
3. A typical installer does five things.
4. It copies the program's files into a folder, usually somewhere central rather than in your downloads.
5. It tells the system which file types this program can open, so double-clicking works.
6. It creates shortcuts, in a menu, on a desktop, or in a dock.
7. It writes settings somewhere the program can find them later.
8. And it records enough information for the program to be removed cleanly.
9. That last one is the part most often done badly, which is why uninstalling often leaves rubbish behind.

20.10.2 PLAIN - a picture in your head

1. Think of a new appliance being delivered to a flat.
2. The delivery team carries it to the right room. That is copying the files.
3. They plug it in and register it with the building's electrics. That is registering file types.
4. They put a label on the fuse box saying which switch controls it. That is the shortcut.
5. They fill in a form for the building manager: what was installed, when, and what has to be undone to remove it. That is the uninstall record.
6. A careless team dumps the box in the hallway and leaves. Later, nobody knows what to remove.

Where this comparison breaks: an appliance stays where it was put. Software changes the building. It can alter shared settings, replace shared libraries that other programs also use, and add background services that start every time the building's power comes on. Removing it is therefore not the reverse of a delivery. It is closer to undoing a renovation, which is why package managers that track every file exist at all.

20.10.3 PLAIN - a worked example

1. A Debian package is not a mysterious binary. It is an `ar` archive, the same ancient Unix format used for static libraries.

```
$ xxd -l 24 zstd_1.5.5+dfsg2-2build1.1_amd64.deb
00000000: 213c 6172 6368 3e0a 6465 6269 616e 2d62  !<arch>.debian-b
00000010: 696e 6172 7920 2020                      inary

$ ar t zstd_1.5.5+dfsg2-2build1.1_amd64.deb
debian-binary
control.tar.zst
data.tar.zst

$ dpkg -I zstd_1.5.5+dfsg2-2build1.1_amd64.deb
 new Debian package, version 2.0.
 size 644020 bytes: control archive=1069 bytes.
     755 bytes,  17 lines   control
    1064 bytes,  17 lines   md5sums
```

```
Package: zstd
Version: 1.5.5+dfsg2-2build1.1
Architecture: amd64
Installed-Size: 1802
Depends: libc6 (>= 2.34), libgcc-s1 (>= 3.3.1),
         liblz4-1 (>= 1.8.0), liblzma5, libstdc++6,
         zlib1g (>= 1:1.1.4)
Section: utils
Priority: optional
```

2. Three members. `debian-binary` is a text file containing `2.0`, the format version.
3. `control.tar.zst` holds the metadata: package name, version, dependencies, checksums, and optional scripts to run before and after install.
4. `data.tar.zst` holds the actual files, with their full destination paths inside.
5. Installing is therefore: read the control data, check the dependencies are satisfied, unpack the data archive at the root of the file system, record the file list, run the post-install script.
6. Uninstalling is exact, because the file list was recorded. This is the whole advantage of a package manager over an installer program.

20.10.4 PLAIN - what is really happening inside

1. Different systems answer the same problem with different shapes.
2. Windows historically used self-contained installer programs: a single `.exe` that unpacks itself and does the work in code.
3. Microsoft's own answer, the Windows Installer, replaced code with data. An `.msi` file is a small relational database of tables: which files, which registry keys, which shortcuts, in which order.
4. Because it is data, the system itself performs the installation, can roll back a failure, and can uninstall precisely.
5. macOS mostly avoids installers. An application is a folder whose name ends in `.app`, containing the executable, the icons and everything else.
6. The Finder shows that folder as a single item. Dragging it to Applications is the entire installation.
7. Linux distributions use package managers with a central database and hard dependency rules, which is why installing one thing can pull in twenty others.
8. The newer Linux formats invert this. Instead of sharing libraries, each app carries its own, so it works on any distribution at the cost of size.
9. And on every platform there is now a signing layer: the operating system checks a cryptographic signature before running the installer, and warns you loudly if there is not one.

20.10.5 TECHNICAL - the engineer's version

1. Windows packaging formats:

Format	What it is	Notes
MSI	relational database	Windows Installer, 1999
NSIS	scripted self-extractor	Nullsoft, 2000, open source
InstallShield	commercial toolkit	from the 1990s
MSIX	signed app container	since Windows 10, 2018
Portable exe	no install at all	user copies one file

2. Windows Installer shipped in 1999 alongside Office 2000. An MSI file is actually an OLE compound document holding roughly eighty defined tables such as `File`, `Component`, `Registry`, `Shortcut` and `InstallExecuteSequence`.

3. NSIS was written at Nullsoft for distributing Winamp and released in 2000. It produces a single executable containing a compressed payload and a compiled script. It is the most common format for open-source Windows applications.
4. Uninstall information on Windows lives under `HKLM\Software\Microsoft\Windows\CurrentVersion\Uninstall`, one key per product, with `DisplayName`, `UninstallString` and `InstallLocation`.
5. macOS packaging:

Format	What it is	Notes
.app	a directory, not a file	bundle from NeXTSTEP
.dmg	disk image	mounts as a volume
.pkg	installer package	XAR archive, since 10.5
.mpkg	several pkgs together	metapackage

6. An `.app` bundle contains `Contents/Info.plist` declaring the identifier, version and document types, `Contents/MacOS/` with the Mach-O executable, and `Contents/Resources/` with icons and localizations. It is an ordinary directory; `cd` into it from a terminal and everything is visible.
7. Linux packaging: `.deb` from Debian, described above; `.rpm` from Red Hat, created in the mid-1990s, consisting of a lead, a signature header, a metadata header, and a compressed cpio payload.
8. The distribution-independent formats:

Format	Started	Isolation model
AppImage	klik 2004, renamed 2011	none, single file
Flatpak	2015, was xdg-app	bubblewrap sandbox
Snap	Canonical, 2014–2016	AppArmor confinement

9. An AppImage is a single executable file: a small runtime concatenated with a SquashFS image. Running it mounts the image with FUSE and executes the application inside. It requires no installation and no root.
10. Flatpak shares common runtimes between applications through OSTree deduplication and sandboxes each app with bubblewrap and portals. Snap uses compressed SquashFS images mounted as loop devices and confined by AppArmor.
11. Code signing by platform:

Platform	Mechanism	Introduced
Windows	Authenticode	1996
macOS	codesign + Gatekeeper	2012
macOS	notarization required	2019, Catalina
Linux	repository GPG signing	Debian, early 2000s

12. Authenticode embeds a PKCS#7 signature in the PE file, referenced by Data Directory entry 4, the Security Directory. The signed hash deliberately excludes the checksum field and the certificate table itself, so those can change without invalidating the signature.
13. Extended Validation certificates and, since 2023, the requirement that code-signing private keys live on hardware tokens or in a hardware security module, exist because stolen signing keys were used to sign malware. The Stuxnet worm, discovered in 2010, was signed with certificates stolen from two Taiwanese hardware companies.
14. Apple's Gatekeeper arrived with OS X 10.8 Mountain Lion in 2012. From macOS 10.15 Catalina in 2019, software distributed outside the App Store must also be notarized: uploaded to Apple, scanned, and issued a ticket that the system checks.

15. Windows SmartScreen, part of Windows since version 8 in 2012, warns about executables with little reputation history regardless of signing. A newly signed application still triggers it until enough installs accumulate. This is a reputation system, not a signature check, and the two are often confused.
16. Linux distributions sign the package index rather than each package. `apt` verifies a detached GPG signature on the `Release` file, which contains hashes of the package lists, which contain hashes of the packages. Trust flows down that chain.

20.10.6 WORDS - remember these

1. Installer — a program that puts software in place — a bootstrapper performing file placement, registration and uninstall bookkeeping.
2. MSI — the Windows database installer format — an OLE compound file of installation tables executed by the Windows Installer service.
3. Bundle — a macOS application folder shown as one item — a directory with a defined layout and an `Info.plist` describing it.
4. Package manager — a tool that tracks every installed file — a database-backed system resolving dependencies and enabling exact removal.
5. Sandbox — a fence around a running application — a confinement mechanism such as bubblewrap, AppArmor or the macOS App Sandbox.
6. Code signing — proving who built a program — attaching a signature over a hash of the image, verified against a trusted certificate chain.
7. Notarization — Apple checking your app before users run it — submission to Apple's service, producing a ticket stapled to the distribution.

20.11 Scripts versus binaries

20.11.1 PLAIN - in simple words

1. A `.sh` or `.py` file contains no machine instructions at all. It is plain text a human wrote.
2. Yet you can mark it executable and run it exactly like a program. Something must be bridging that gap.
3. The bridge is two characters at the very start of the file: a hash and an exclamation mark, written `#!`.
4. Those two characters are followed by the path of a real program.
5. When you run the file, the system reads that first line, and instead of trying to execute your text, it runs the named program and hands it your file.
6. So `./script.sh` really becomes `/bin/sh ./script.sh` behind your back.
7. The named program is called an **interpreter**. It reads your text line by line and does what it says.
8. The interpreter itself is an ordinary compiled program, with an ELF or PE header, exactly as described earlier.
9. So nothing was ever magic. One real program is running, and your file is its input.

20.11.2 PLAIN - a picture in your head

1. Think of a sealed envelope handed to a receptionist.
2. On the outside is written: "Please pass to the French translator."
3. The receptionist does not read the letter. She reads only the instruction on the outside and walks the envelope to the right desk.
4. The translator opens it and works through the contents.
5. The `#!` line is the writing on the outside of the envelope. The kernel is the receptionist. The interpreter is the translator.
6. If the writing names a desk that does not exist, the receptionist comes back with an error, and the error is confusingly about the envelope, not the desk.

Where this comparison breaks: a receptionist can see the letter is in French even without the label, and would work it out. The kernel cannot and does not try. It reads exactly two characters. If they are not `#!`, it

gives up with an error called "exec format error", and any success after that is the shell doing extra work of its own accord, not the kernel being clever.

20.11.3 PLAIN - a worked example

1. We wrote two shell scripts, identical except that one has a `#!` line.

```
$ cat s1.sh
#!/bin/sh
echo "I am a script, arg1=$1"

$ xxd -l 16 s1.sh
00000000: 2321 2f62 696e 2f73 680a 6563 686f 2022  #!/bin/sh.echo "

$ ./s1.sh hello
I am a script, arg1=hello

$ file s1.sh
s1.sh: POSIX shell script, ASCII text executable
```

2. The first two bytes are `23 21`, which are `#` and `!`. That is the whole mechanism.
3. Now the file without a `#!` line, run two ways.

```
$ cat s2.sh
echo "no shebang here"

$ ./s2.sh
no shebang here

$ python3 -c "import os; os.execve('/tmp/ch20/s2.sh',
              ['s2.sh'], {})"
execve failed: [Errno 8] Exec format error: 's2.sh'
```

4. Read that carefully. The shell ran it fine. A direct `execve` refused.
5. The kernel genuinely could not run the file. Error 8 is `ENOEXEC`, no recognized format.
6. When the shell got the same error, it did something extra: it decided to run the file itself, as shell commands. That is a rule in the shell, defined by POSIX, not a kernel feature.
7. So the file "worked" for a reason that has nothing to do with the file being executable at the system level.

20.11.4 PLAIN - what is really happening inside

1. Inside the kernel, running any file goes through a list of format handlers, tried in order.
2. One of them, on Linux called `binfmt_script`, is very simple.
3. It checks whether the first two bytes are `#` and `!`.
4. If so, it reads up to a limit of characters, stopping at the first newline.
5. It splits that into the interpreter path and, on Linux, at most one further argument.
6. It then restarts the whole exec operation, but with the interpreter as the program and with the argument list rewritten.
7. The new argument list is: the interpreter path, the optional argument from the `#!` line, the original file path, and then your original arguments.
8. That restart is limited. A script whose interpreter is itself a script is allowed, but only to a fixed depth, to stop infinite loops.
9. Windows has no `#!` mechanism in the kernel. It uses file associations instead: `.py` is registered to the Python launcher, which then reads a `#!` line itself as a convention borrowed from Unix.

20.11.5 TECHNICAL - the engineer's version

1. The mechanism is implemented in `fs/binfmt_script.c` in the Linux kernel. The `#!` sequence is often called the **shebang**, or hash-bang.
2. It was added to Unix by Dennis Ritchie around 1980 and first appeared in the Eighth Edition of Research Unix. It is specified in POSIX only as an unspecified behaviour, meaning portable scripts may rely on it in practice but it is not formally guaranteed.
3. Line length limits differ and this matters for portability:

System	`#!` line limit	Extra args
Linux	255 bytes	at most one
FreeBSD	8192 bytes	several
macOS	512 bytes	at most one
Solaris	1023 bytes	at most one

4. Because Linux allows only one argument, `#!/usr/bin/env python3 -u` passes `python3 -u` to `env` as a single string on Linux and fails. Writing `#!/usr/bin/env -S python3 -u` works on GNU coreutils 8.30 and later, released in 2018.
5. `#!/usr/bin/env python3` is the common idiom because it searches `PATH` instead of hard-coding a location, which differs across distributions and inside virtual environments.
6. A shebang must be an absolute path. The kernel does no `PATH` search. That is the entire reason `env` is used as an indirection.
7. If a script has CRLF line endings, the interpreter path becomes `/bin/sh\r`, which does not exist. The resulting message, `bad interpreter: No such file or directory`, is one of the most confusing errors in Unix, because the path printed looks correct on screen.
8. `binfmt_misc`, added to Linux in 1997, generalizes this. You register a magic number or extension with an interpreter path by writing to `/proc/sys/fs/binfmt_misc/register`.
9. This is how a Linux system transparently runs ARM binaries under QEMU, and how Windows executables can be handed to Wine automatically. It was not mounted in the sandbox used for this chapter, which is common inside containers.
10. The interpreter receives the script by path, not on standard input. It can therefore seek in the file, which is why Python can read a `.pyc` cache and why a shell script can safely have binary data appended to it after an `exit` line.
11. Cost comparison, approximate and measured informally: starting a compiled C binary that prints one line takes about 1 ms on the machine used here. Starting the same task through `python3` takes roughly 20 to 40 ms, dominated by importing the interpreter's own startup modules.
12. The honest version: saying "scripts are interpreted and binaries are compiled" is a useful lie. CPython compiles your source to bytecode before running it. Java compiles to bytecode and then to machine code at run time. Modern JavaScript engines do the same. The real distinction is when the translation happens and who ships the result, not whether translation happens.

20.11.6 WORDS - remember these

1. Shebang — the `#!` at the top of a script — the two-byte magic recognized by `binfmt_script`, followed by an absolute interpreter path.
2. Interpreter — the program that reads and runs your text — an ordinary executable receiving the script path as an argument.
3. ENOEXEC — the error meaning "I cannot run this" — errno 8, returned by `execve` when no binary format handler accepts the file.
4. binfmt_misc — the Linux feature for registering new formats — a kernel module mapping magic bytes or extensions to interpreter paths.

5. env — the tool used to find an interpreter on PATH — `/usr/bin/env`, run from the shebang so the real path need not be hard-coded.
6. Bytecode — a compact instruction set for a virtual machine — the intermediate form produced by CPython, javac and similar compilers.

20.12 Archives and compression in practice

20.12.1 PLAIN - in simple words

1. Two different jobs get muddled together: putting many files into one, and making things smaller.
2. Putting many files into one is called **archiving**. Making things smaller is called **compression**.
3. ZIP does both jobs in one format. It squeezes each file separately and then packs the squeezed pieces together.
4. Unix split the jobs. `tar` packs files together and does no squeezing at all. `gzip` squeezes one stream and knows nothing about files.
5. That is why you see `.tar.gz`: first tar joined everything into one stream, then gzip squeezed that whole stream.
6. The order matters enormously. Squeezing one big stream finds repetition between files. Squeezing each file alone cannot.
7. Squeezing everything as one stream is called **solid** compression. It gives smaller results and makes extracting a single file slower.
8. Different squeezing methods trade speed against size, and the trade is real and measurable.
9. There is no best one. There is a fastest, a smallest, and several sensible middles.

20.12.2 PLAIN - a picture in your head

1. Think of packing a suitcase for a family of four.
2. Method one: each person rolls their own clothes tightly, and the four bundles go in. Anyone can pull out their own bundle without disturbing the others.
3. Method two: all the clothes are laid out together and vacuum-packed as one slab. It takes far less space, because socks fill the gaps between shirts.
4. But to get one shirt you must open the whole slab and repack it.
5. Method one is ZIP. Method two is tar plus gzip, or a solid 7z archive.
6. If everyone's clothes are similar, method two wins by a lot, because the machine notices the repetition across people.

Where this comparison breaks: vacuum packing removes air, which is a fixed saving. Compression removes repetition, which is not fixed at all. A slab of already-random data compresses by nothing, and can come out slightly larger because of the bookkeeping added. There is no method that shrinks every possible input. That is not an engineering limit; it is a counting argument, and it is provably true.

20.12.3 PLAIN - a worked example

1. We compressed the same 47,575,040 bytes of C header files, all real text, with six settings on one machine, and timed both directions.

Tool	Size (bytes)	Ratio	Compress
zstd -3	8,220,027	5.79x	0.19 s
gzip -6	8,466,301	5.62x	1.30 s
gzip -9	8,384,098	5.67x	2.93 s
bzip2 -9	6,632,497	7.17x	3.67 s
zstd -19	5,895,562	8.07x	19.20 s
xz -6	5,760,568	8.26x	16.09 s

Tool	Size (bytes)	Ratio	Compress

2. Read the first two rows together. `zstd -3` produced a smaller file than `gzip -6` and did it about seven times faster.
3. Now read the top and bottom rows. `xz -6` is 30 percent smaller than `zstd -3`, and took 85 times longer.
4. Decompression times tell a different story again, and this is the number that matters for software you ship.

Tool	Decompress	Notes
zstd -3	0.06 s	same for -19
gzip -6	0.26 s	same for -9
xz -6	0.38 s	slower to build
bzip2 -9	1.12 s	slowest both ways

5. Note that zstd decompresses at the same speed whether it was compressed at level 3 or level 19. Higher levels cost the packer more, not the unpacker.
6. That single property is why zstd replaced gzip in Debian and Ubuntu packages, in Arch Linux packages, and in the Linux kernel's own build options.

20.12.4 PLAIN - what is really happening inside

1. Nearly all general compression works by finding repeats and referring back to them instead of writing them again.
2. The reference says "go back this many bytes and copy this many". That is the LZ77 idea, published by Abraham Lempel and Jacob Ziv in 1977.
3. How far back the tool may look is the **window**. A bigger window finds more repeats and needs more memory.
4. After that, the remaining symbols are re-coded so that common ones use fewer bits. That is Huffman coding, or the newer arithmetic and range coders.
5. Solid compression matters because of the window. If every file is compressed separately, the window is reset at each file boundary and cross-file repetition is invisible.
6. In a directory of four thousand C header files that all begin with a similar licence comment, that repetition is enormous.
7. This is why the same data, packed two ways, can differ in size by more than a factor of two.

20.12.5 TECHNICAL - the engineer's version

1. We measured solid versus per-file directly, on the same 4,394 files.

Method	Size (bytes)	Time
zip -9, per file	13,580,396	3.38 s
tar + gzip -9, solid	8,384,087	2.98 s
7z -mx=5 -ms=off	11,690,306	9.43 s
7z -mx=5 -ms=on	6,224,950	12.58 s

2. Compare rows one and two. Identical Deflate algorithm, identical level. The solid version is 38 percent smaller purely because the window spans file boundaries.
3. Compare rows three and four. Identical LZMA2 algorithm, identical level. Solid mode is 47 percent smaller.

4. The cost of solid mode is random access. Extracting the last file from a solid archive requires decompressing everything before it. ZIP can seek directly.
5. History and specifications:

Tool	Author, year	Specification
tar	Version 7 Unix, 1979	POSIX.1 ustar
gzip	Gailly and Adler, 1992	RFC 1952, DEFLATE RFC 1951
bzip2	Julian Seward, 1996	no formal spec
7-Zip	Igor Pavlov, 1999	LZMA, documented
xz	Tukaani project, 2009	xz file format spec
zstd	Yann Collet, 2016	RFC 8478

6. `tar` means tape archive. Its record size of 512 bytes and its habit of padding to 10,240-byte blocks come directly from nine-track magnetic tape drives. A modern `.tar` file is still padded to a multiple of 10,240 bytes, which is why a tar of one small file is 10,240 bytes.
7. Window sizes: Deflate uses 32 KiB, fixed by RFC 1951. bzip2 uses blocks of up to 900 KiB. LZMA defaults to 8 to 64 MiB depending on level. zstd's window grows with level, up to 128 MiB at level 22 with long mode enabled.
8. bzip2 uses a completely different approach, the Burrows-Wheeler transform published in 1994, which reorders data to group similar bytes before coding. It is slow in both directions and has largely been displaced.
9. zstd's speed comes from finite state entropy coding, an implementation of asymmetric numeral systems published by Jaroslaw Duda in 2009. It achieves arithmetic-coding compression ratios at table-lookup speeds.
10. A practical rule that holds up: choose zstd for anything you compress once and decompress often, gzip when compatibility with everything matters, and xz only when the transfer saving genuinely outweighs a long build.
11. None of these is encryption. ZIP's original password scheme, from 1990, is broken and can be attacked with known plaintext. ZIP AES-256 and 7z AES-256 are sound, but 7z still leaves file names visible unless header encryption is switched on with `-mhe=on`.

20.12.6 WORDS - remember these

1. Archive — many files joined into one — a container preserving names, sizes, permissions and directory structure.
2. Compression — making data smaller — removing redundancy so the original can be reconstructed exactly, in lossless schemes.
3. Solid — compressing everything as one stream — allowing back-references to cross file boundaries, at the cost of random access.
4. Window — how far back the packer may look — the maximum LZ77 match distance, 32 KiB in Deflate, up to 128 MiB in zstd.
5. Ratio — how many times smaller it got — original size divided by compressed size, quoted with the exact corpus and level.
6. Lossless — nothing is thrown away — the decompressed output is byte-identical to the input, unlike JPEG or MP3.

20.13 File metadata beyond the basics

20.13.1 PLAIN – in simple words

1. Beyond size and dates, a file can carry extra labels that most tools never show you.
2. On Linux and macOS these are called **extended attributes**: small named pieces of data attached to a file but not part of its contents.
3. macOS uses them heavily. When you download a file, the system attaches a label meaning "this came from the internet".
4. That label is why macOS asks you whether you are sure the first time you open a downloaded application.
5. Windows has a stranger version. A file on NTFS can have several separate contents, each with its own name.
6. The main one is what you see. The extra ones are invisible in Explorer, do not change the reported file size, and were abused for years to hide code.
7. Photographs carry a third kind: a block of camera information inside the image file itself.
8. That block can include the exact time, the camera model, the settings, and often the precise place the photo was taken.
9. Sharing a photo can therefore share your home address without you saying a word.

20.13.2 PLAIN – a picture in your head

1. Think of a book returned to a library.
2. The book has a title and pages. That is the name and contents.
3. A librarian has stuck a small note inside the back cover: "returned late, donated by the Smith family, check for water damage".
4. The note is not part of the book. Photocopying the book does not copy the note. Lending it might.
5. Those notes are extended attributes. Useful, invisible, and easily lost.
6. Now imagine a book where extra chapters are bound in behind the endpapers, invisible unless you know to look. The spine still says 300 pages.
7. That is a Windows alternate data stream.

Where this comparison breaks: the librarian's note is written by a human who knows what it means. Extended attributes are written by programs, in binary, with names like `com.apple.quarantine`, and are silently dropped by many everyday operations: uploading, emailing, extracting from a plain zip, or copying with the wrong flag. The library at least knows the note existed.

20.13.3 PLAIN – a worked example

1. Extended attributes on Linux, all real commands run for this chapter.

```
$ setfattr -n user.kedbyte -v "chapter20" note.txt
$ getfattr -d note.txt
# file: note.txt
user.kedbyte="chapter20"

$ ls -l note.txt
-rw-r--r-- 1 root root 26 Aug 13 01:57 note.txt

$ cp note.txt note_copy.txt
$ getfattr -d note_copy.txt
(no output)

$ cp --preserve=xattr note.txt note_copy2.txt
$ getfattr -d note_copy2.txt
# file: note_copy2.txt
user.kedbyte="chapter20"
```

2. The reported size is still 26 bytes. The attribute is stored outside the data.

3. A plain copy lost it silently. No warning, no error. Only the explicit flag preserved it.
4. Now the photo case. We took a real JPEG and added camera and location tags.

```
$ exiftool -Make="ACME" -Model="Phone X" \
    -DateTimeOriginal="2026:03:14 09:41:00" \
    -GPSLatitude=18.5204 -GPSLatitudeRef=N \
    -GPSLongitude=73.8567 -GPSLongitudeRef=E photo.jpg

$ exiftool -G1 photo.jpg
[IFD0]      Make                : ACME
[IFD0]      Camera Model Name   : Phone X
[ExifIFD]   Date/Time Original : 2026:03:14 09:41:00
[Composite] GPS Position        : 18 deg 31' 13.44" N,
                                  73 deg 51' 24.12" E
```

5. That is a location accurate to a few metres, a timestamp to the second, and the device, all inside an ordinary picture.
6. Removing it is one command, and it shrank the file by 2,311 bytes:

```
$ ls -l photo.jpg
-rw------- 1 root root 8520 photo.jpg
$ exiftool -all= photo.jpg
$ ls -l photo.jpg
-rw------- 1 root root 6209 photo.jpg
$ exiftool -Make -Model -GPSLatitude photo.jpg
(nothing listed)
```

20.13.4 PLAIN – what is really happening inside

1. Extended attributes are stored by the file system next to the inode, or in a separate block if they grow too large.
2. They are name-value pairs. The name has a prefix declaring a namespace: `user.` for anything, `security.` for SELinux labels, `system.` for access control lists, `trusted.` for privileged data.
3. macOS uses the same mechanism with Apple-defined names. Two matter most.
4. `com.apple.quarantine` is set by browsers and mail clients on every downloaded file, and holds flags, a timestamp and the downloading application.
5. `com.apple.ResourceFork` is the modern home of the old resource fork, a second data stream that classic Macintosh files had from 1984.
6. Windows does it differently. On NTFS, every file is a set of named attributes, and the contents are just the unnamed `$DATA` attribute.
7. Adding a second named `$DATA` attribute gives the file a second, hidden set of contents. The syntax is a colon: `report.txt:hidden`.
8. EXIF is not an operating system feature at all. It is a block of data placed inside the image file, in a JPEG marker segment, using the TIFF tag structure.
9. That is why EXIF survives copying, emailing and zipping, whereas extended attributes usually do not. It is part of the file's own bytes.

20.13.5 TECHNICAL – the engineer's version

1. Linux extended attribute namespaces and their rules:

Namespace	Who can write	Typical use
user.	file owner	application metadata
trusted.	CAP_SYS_ADMIN	privileged tooling
security.	policy modules	SELinux labels

Namespace	Who can write	Typical use
system.	kernel	POSIX ACLs

2. On ext4 the total size of all extended attributes for one file is limited to one file system block, normally 4096 bytes. XFS allows more. This is why attributes are used for labels, not for storage.
3. macOS quarantine values look like `0083;5f2b1c40;Safari;UUID`, giving flags, a hexadecimal timestamp, the agent name and an event identifier. Gatekeeper reads this before first launch and removes it after approval. `xattr -d com.apple.quarantine app` clears it manually.
4. NTFS alternate data streams have existed since NTFS shipped with Windows NT 3.1 in 1993, originally to support Macintosh resource forks through Services for Macintosh.
5. The reported file size is the length of the unnamed `$DATA` stream only. `dir` shows no sign of the others. `dir /r`, added in Windows Vista, does. `Get-Item -Stream *` in PowerShell lists them.
6. Windows itself uses one: `Zone.Identifier`, the mark of the web, written by browsers and by Outlook. Office reads it and opens the document in Protected View. Removing it is what "Unblock" does in the file properties dialog.
7. Malware families including Backdoor.Rustock in 2006 and later commodity loaders stored payloads in alternate data streams to evade scanners that read only the default stream. Every serious scanner now enumerates streams. Streams are lost the moment a file is copied to FAT32, exFAT, a network share without support, or into a plain ZIP.
8. EXIF was first published by JEIDA in 1995 and is maintained today by CIPA and JEITA. It stores TIFF-format image file directories inside a JPEG APP1 marker segment beginning with the bytes `Exif\0\0`.
9. Commonly present tags, taken from the real file inspected here: `Make`, `Model`, `DateTimeOriginal`, `ExposureTime`, `FNumber`, `ISO`, `Orientation`, `LensModel`, plus a full GPS sub-directory with latitude, longitude, altitude and a UTC timestamp.
10. The `Orientation` tag is the reason a photo appears rotated in one program and upright in another: the pixels are unrotated and the tag says how to display them.
11. Most large social platforms strip EXIF on upload. Most chat applications strip it when sending a photo as an image and keep it when sending as a file. Cloud storage links and direct email attachments keep it. Treat any file you send as carrying its metadata unless you removed it yourself.
12. Tools: `getfattr`, `setfattr`, `attr` on Linux; `xattr -l` and `mdls` on macOS; `Get-Item -Stream`, `dir /r` and `streams.exe` on Windows; `exiftool` and `exiv2` for image metadata.

20.13.6 WORDS - remember these

1. Extended attribute — a small named label attached to a file — a namespaced name-value pair stored outside the file's data.
2. Quarantine flag — the macOS mark on downloaded files — the `com.apple.quarantine` extended attribute read by Gatekeeper.
3. Resource fork — the classic Macintosh second stream — historically an HFS fork, now the `com.apple.ResourceFork` extended attribute.
4. Alternate data stream — a hidden second set of contents on NTFS — an additional named `$DATA` attribute addressed as `file:stream`.
5. Mark of the web — the Windows record of where a file came from — the `Zone.Identifier` stream, driving Protected View and SmartScreen.
6. EXIF — camera information stored inside a photo — TIFF image file directories in the JPEG APP1 segment, including an optional GPS directory.

20.14 Reading a file you do not understand

20.14.1 PLAIN – in simple words

1. Sooner or later you are handed a file with no extension, or a wrong one, and asked what it is.
2. There is a reliable order of steps, and it takes about a minute.
3. Step one: ask the type-guessing tool. It is right most of the time and costs nothing.
4. Step two: look at the first sixteen bytes yourself. Compare them with the magic number table.
5. Step three: pull out the readable text. Format names, version strings, file paths and error messages usually appear in plain sight.
6. Step four: look at the very end. Many formats keep their index there.
7. Step five: if the file looks like nothing, check whether it is compressed or encrypted, because both look like random noise.
8. Step six: scan the whole file for magic numbers, not just the start, because files are often nested inside other files.
9. Then unwrap one layer and start again from step one.

20.14.2 PLAIN – a picture in your head

1. Think of a parcel with no address label.
2. You shake it. Something rattles. That is the type-guessing tool: fast, rough, usually right.
3. You cut off one corner and peek. That is the hex dump of the first bytes.
4. You read the customs sticker for words you recognize. That is `strings`.
5. You find another wrapped parcel inside, and you start again. Nested formats are the norm, not the exception.

Where this comparison breaks: shaking a parcel gives you a hint about the whole thing at once. A file gives you nothing unless you look in the right place. A 1 GB file whose first bytes are unrecognized may be a perfectly ordinary format that simply begins with a length field, and no amount of staring at the start will tell you. Real identification means checking several places: the start, the end, and a scan across the middle.

20.14.3 PLAIN – a worked example

1. We were handed a file called `mystery.dat` and asked what it was. Here is the whole session.

```
$ ls -l mystery.dat
-rw-r--r-- 1 root root 259 mystery.dat

$ file mystery.dat
mystery.dat: gzip compressed data, was "inner.tar",
             last modified Thu Aug 13 02:02:54 2026,
             max compression, from Unix,
             original size modulo 2^32 10240

$ xxd -l 16 mystery.dat
00000000: 1f8b 0808 ce25 7d6a 0203 696e 6e65 722e  .....%}j..inner.
```

2. `1f 8b` is the gzip magic. `08` is the compression method, Deflate. The fourth byte `08` is a flag meaning "an original filename follows".
3. That is why the readable text `inner.` appears at offset 10. The gzip header literally stores the original name.
4. So it is one layer of gzip around something called `inner.tar`. Unwrap it.

```
$ gzip -dc mystery.dat > mystery.out
$ file mystery.out
mystery.out: POSIX tar archive (GNU)

$ tar tvf mystery.out
```

```
-rw-r--r-- root/root  69 2026-08-13 01:55 tiny.png
-rw-r--r-- root/root  26 2026-08-13 01:57 note.txt

$ tar xf mystery.out -C out && file out/*
out/note.txt: ASCII text
out/tiny.png: PNG image data, 1 x 1, 8-bit/color RGB
```

5. Three layers, three commands, one minute. Nothing was guessed.
6. Note that `file` told us the original name and the uncompressed size without decompressing anything, because gzip stores both in its header and trailer.

20.14.4 PLAIN – what is really happening inside

1. Scanning for embedded formats is worth doing by hand, because tools like `binwalk` are not always installed.
2. The method is simple: search the whole file for every known magic byte sequence and report the offsets.
3. We wrote a twenty-line script that does exactly this and ran it on a file made by gluing a Linux executable and a ZIP archive together.

```
scanning combo.bin (16516 bytes)
  offset       0  ELF object
  offset   15960  ZIP local file header
  offset   16052  ZIP local file header
  offset   16130  ZIP local file header
  offset   16260  ZIP central directory
  offset   16338  ZIP central directory
  offset   16416  ZIP central directory
  offset   16494  ZIP end of central dir
```

4. The structure is now completely clear without opening the file in an editor: an executable of 15,960 bytes, then a three-entry ZIP archive.
5. That layout is exactly how self-extracting archives and single-file applications are built, and it is what to look for when a program seems suspiciously large.
6. False positives happen. Two-byte signatures like `MZ` appear by chance roughly once every 65,536 bytes of random data. Always confirm a hit by checking a second field, not just the magic.

20.14.5 TECHNICAL – the engineer's version

1. The identification toolkit, and what each answers:

Tool	Question it answers	Cost
file	what format is this	milliseconds
xxd, od	what are the exact bytes	instant
strings	what text is inside	one pass
binwalk	what is embedded where	one pass
7z l, unzip -l	what is in the archive	fast

2. `strings` by default prints runs of four or more printable characters. Use `-n 8` to cut noise and `-e l` or `-e b` for UTF-16 text, which is where Windows binaries hide most of their readable content.
3. Real `strings` output from the Linux binary we built tells you almost everything about it:

```
$ strings -n 6 hello
/lib64/ld-linux-x86-64.so.2
__libc_start_main
libc.so.6
GLIBC_2.2.5
```

```
GLIBC_2.34
hello, world
GCC: (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0
hello.c
```

4. From eight lines we learned the interpreter, the library, the minimum glibc version, the program's output, the exact compiler build, and the source file name. This is why release builds are stripped.
5. Practical checks for the "it looks like random noise" case. Compute the byte entropy. Compressed data typically measures 7.9 to 8.0 bits per byte; encrypted data measures the same. Distinguish them by structure: compressed formats have headers, encrypted blobs usually do not.
6. Check the length. AES in a block mode produces output that is a multiple of 16 bytes, often plus a fixed header. That is a strong hint.
7. If the file is a container you half-recognize, list rather than extract. `unzip -l`, `tar tvf`, `7z l`, `ar t`, `dpkg -I` and `rpm -qip` all read metadata without writing anything to disk.
8. For executables, `objdump -x`, `readelf -a`, `nm -D`, `ldd` and `otool -L` on macOS answer what it links against. For unknown binary blobs inside firmware, `binwalk -e` automates the scan-and-extract loop described above.
9. Never run an unknown executable to find out what it is. Read it statically first, and if you must run it, do so in a disposable virtual machine with no network.
10. When nothing matches, the remaining evidence is structure: look for repeating record lengths, plausible 32-bit little-endian sizes near the start, and offsets that point at other offsets. Most private formats are a header, a count, and an array.

20.14.6 WORDS - remember these

1. Hex dump — the raw bytes shown as pairs of hex digits — an offset-addressed listing produced by `xxd`, `od -t x1` or a hex editor.
2. strings — the readable text pulled out of a binary — runs of printable characters above a minimum length, in a chosen encoding.
3. Carving — pulling embedded files out of a bigger one — locating known signatures at arbitrary offsets and extracting the ranges between them.
4. Entropy — how random the bytes look — Shannon entropy in bits per byte, near 8.0 for compressed or encrypted data.
5. Nested format — a file inside a file inside a file — layered containers, each requiring its own identification pass.
6. False positive — a magic match that is a coincidence — an accidental byte sequence, more likely with short signatures.

20.98 Common wrong ideas

1. Wrong: the extension tells the computer what a file is. Right: the extension is part of the name. Windows chooses to trust it, Linux tools read the bytes instead, and the file itself is unchanged either way.
2. Wrong: renaming a file converts it. Right: renaming changes a directory entry. Every byte of contents stays exactly the same, as two identical SHA-256 sums in 20.2.3 demonstrated.
3. Wrong: `.exe` files are a special kind of thing the computer understands natively. Right: an `.exe` is an ordinary file whose bytes happen to be machine instructions plus a header. The loader does all the work.
4. Wrong: a text file and a binary file are different at the storage level. Right: both are byte sequences. "Text" describes how the bytes happen to render under an encoding, nothing more.
5. Wrong: an empty line ending is just one invisible character everywhere. Right: it is one byte on Linux and macOS, two on Windows, and a different single byte on classic Mac OS, and `wc -l` reported zero lines for a file full of text because of it.

6. Wrong: a DOCX is a Microsoft binary format. Right: it is a ZIP archive of XML files, openable with any unzip tool, as shown in 20.5.3.
7. Wrong: bigger compression level always means slower decompression. Right: for zstd, decompression took 0.06 seconds at both level 3 and level 19. Level affects the packer's search effort, not the unpacker's work.
8. Wrong: a ZIP is read from the front. Right: correct readers start at the end of central directory record and jump backwards. That is why you can prepend a program to an archive and both still work.
9. Wrong: stripping a binary makes it faster. Right: stripping removes the symbol table, which is never loaded at run time. The program is smaller on disk and identically fast, only harder to debug.
10. Wrong: deleting metadata from a photo before sharing is paranoid. Right: a single command revealed a location accurate to metres, a timestamp and the device model from an ordinary JPEG.

20.99 Chapter summary in 20 lines

1. A file is a named sequence of bytes plus a small metadata record. Nothing in it declares what the bytes mean.
2. Metadata lives in an inode or MFT record: size, permissions, owner, link count and four separate timestamps. The name lives in the directory, not the file.
3. File extensions are a convention inherited from CP/M in 1974, not a rule.
4. Windows decides types from the extension through the registry, macOS from Uniform Type Identifiers introduced in 2005, Linux tools from content sniffing with libmagic.
5. Hiding known extensions by default made `invoice.pdf.exe` display as `invoice.pdf`, and the ILOVEYOU worm of May 2000 used exactly that.
6. Most formats begin with a magic number: `89 PNG` for PNG, `PK` for ZIP, `7f ELF` for Linux binaries, `MZ` for Windows binaries.
7. Java class files and macOS universal binaries share the magic `CAFEBABE`, a real collision resolved by checking the next field.
8. "Text" means the bytes render as readable characters. Line endings are LF on Unix, CRLF on Windows and network protocols, CR on classic Mac OS, all descended from teleprinter mechanics.
9. A UTF-8 byte order mark is three bytes, `EF BB BF`, permitted but not recommended, and it breaks shebangs, shell scripts and strict JSON parsers.
10. ZIP stores local headers, then the data, then a central directory, then an end record. Readers start at the end, which is why appending and prepending both work.
11. DOCX, XLSX, PPTX, ODT, JAR, APK, EPUB and XPI are all ZIP archives with an agreed internal layout.
12. PDF is a body of numbered objects plus a cross-reference table of byte offsets and a trailer, read from the end, supporting incremental update.
13. An executable is machine code plus a header describing segments, permissions and one entry point. PE on Windows, ELF on Linux, Mach-O on macOS.
14. Every PE still begins with an MZ header and a DOS stub printing "This program cannot be run in DOS mode", because offset 0x3C inside it holds the pointer to the real header.
15. ELF carries two tables over the same bytes: program headers for the loader and section headers for the linker. `strip` deletes the second and the program runs identically.
16. Running a program is thirty steps: resolve, permission-check, `execve`, match the format, build a fresh address space, map segments, build the stack with arguments and the auxiliary vector, load `ld.so`, resolve libraries and relocations, then `_start`, then `__libc_start_main`, then `main`.
17. A `.sh` or `.py` file runs because of two bytes, `#!`, handled by `binfmt_script`, which re-execs the named interpreter with your file as an argument. Without them, `execve` returns `ENOEXEC`.
18. Installers copy files, register types, create shortcuts, write settings and record uninstall data. Package managers do the same with an exact file list, which is why removal is clean.
19. tar archives without compressing and gzip compresses without archiving; combining them gives solid compression, which measured 38 percent smaller than the same algorithm applied per file.

20. Files carry more than you see: extended attributes, macOS quarantine flags, NTFS alternate data streams, and EXIF blocks that can hold your exact location. `file`, `xxd`, `strings` and a signature scan will identify almost anything in about a minute.

PART E

Seeing and Showing

One key press traced from your finger to the light in your eye, and how the picture is computed.

21

CHAPTER

From Key Press to Pixel — The Complete Trace

21.0 What this chapter gives you

1. You will be able to follow one key press, the letter A, from your fingertip to the light entering your eye, without a single gap in the story.
2. You will be able to name every piece of hardware and every piece of software that touches that press, in order, and say what each one does to it.
3. You will be able to explain why a switch that closes once produces several electrical changes, and what the machine does about that.
4. You will be able to read a raw USB keyboard report by hand and say which keys are down.
5. You will be able to explain why the keyboard does not know what letter you pressed, and where the letter is actually decided.
6. You will be able to describe what a window system does with an event and why the same key goes to a different program depending on where you clicked.
7. You will be able to give a real time budget for the whole chain, stage by stage, and say which stage is costing you the most.
8. You will be able to do the same trace for a mouse click and a touchscreen tap and say exactly what is different.
9. You will be able to argue honestly about input lag, with numbers, instead of with feelings.

21.1 The finger and the switch

21.1.1 PLAIN - in simple words

1. Under every key on your keyboard there is a switch.
2. A switch is a thing that joins two pieces of metal when you push it, and separates them when you let go.
3. When the metal pieces touch, electricity can flow through. When they are apart, it cannot.
4. That is the whole job of the key. Nothing more. It is a gate for electricity.
5. The key does not know it is the letter A. It has no idea. It only knows “touching” or “not touching”.
6. Different keyboards use different ways of making that touch happen, and they feel very different under the finger.
7. The cheapest way uses a small rubber dome that squashes flat. The most expensive ways use a spring and a sliding plastic part, or a magnet, or a beam of light.
8. All of them are trying to solve the same problem: turn a finger moving a couple of millimetres into a clean electrical signal.

21.1.2 PLAIN - a picture in your head

1. Think of a garden gate with a spring on it.
2. You push the gate, it swings open, and when you let go the spring pulls it shut again.
3. Now imagine the gate has a metal latch that touches a metal post when the gate reaches a certain point.
4. You do not have to push the gate all the way to the wall for the latch to touch. It touches partway through the swing.
5. That partway point is what engineers call the actuation point: the exact moment the key counts as pressed.
6. Now imagine the gate is light and springy. When the latch first hits the post it does not stop dead. It taps, bounces off a fraction of a millimetre, taps again, and only then settles.
7. Anyone watching only the latch would see it touch, break, touch, break, touch, and finally stay touching. All from one push of the gate.
8. That tapping is exactly what a key switch does, and it is the first real problem in this whole chapter.

Where this comparison breaks: a real gate bounces over tens of milliseconds and you can see it. A key switch bounces over a handful of milliseconds and the movements are microscopic. Also, a gate has one latch, while some switch types have no moving contact at all: an optical switch just blocks a light beam, and a Hall-effect switch just moves a magnet past a sensor. Those types cannot bounce mechanically, because there is nothing to bounce.

21.1.3 PLAIN - a worked example

1. Take a Cherry MX Red switch, one of the most common mechanical switches sold.
2. Its published numbers, from the Cherry datasheet, are these.

Property	Value
Actuation force	45 cN (about 45 g)
Pre-travel to actuation	2.0 mm
Total travel	4.0 mm
Initial force	30 cN minimum
Bounce time	under 5 ms

3. So the key sits at rest. You start pushing.
4. After 2.0 mm of downward movement, the sliding plastic part has let the metal contact leaf spring back into contact. The circuit closes.
5. You are still moving. There are 2.0 mm left before the key hits the bottom. Those 2.0 mm change nothing electrically.
6. This is important. The key fires at 2.0 mm, not at the bottom. If you type by slamming keys to the bottom, you are wasting 2.0 mm of your own time.
7. At the moment of contact, the metal leaf vibrates. For under 5 milliseconds the connection flickers on and off.
8. The keyboard's own chip therefore sees something like this over those few milliseconds.

```
time (ms):  0.0  0.4  0.7  1.1  1.5  2.0  2.4  3.0 ... 5.0
contact:    off  ON   off  ON   off  ON   ON   ON      ON
```

9. If the keyboard reported every one of those changes, you would get "AAAA" from one press.
10. So the firmware waits. It only accepts the change once the signal has held steady for a set period. That waiting is called debouncing.

21.1.4 PLAIN - what is really happening inside

1. Let us go through the four main switch families, because they behave differently and cost you different amounts of time.
2. **Rubber dome / membrane.** Three flexible plastic sheets are stacked. The top and bottom sheets have printed conductive traces. The middle sheet has holes.
3. A moulded rubber dome sits over each key. Pressing collapses the dome, which pushes the top sheet through the hole in the middle sheet, so the top trace touches the bottom trace.
4. The collapse is sudden, which is why a cheap keyboard has a soft top and then a sharp give. Actuation and bottoming out are almost the same moment, typically at 3.5 to 4 mm.
5. **Mechanical.** Each key has its own sealed plastic housing with a spring, a sliding stem, and a springy metal contact leaf.
6. The stem has a ramp cut into it. As the stem descends, the ramp lets the leaf spring back into contact with a fixed metal terminal.
7. Because the contact is a springy metal blade hitting a metal terminal, it bounces. Cherry specifies under 5 ms of bounce.
8. **Scissor.** Used on laptops and thin keyboards. A rubber dome still does the electrical work, but two plastic arms crossed like scissors hold the keycap level.
9. That means the key works even if you press a corner, and it allows travel of roughly 1 to 2 mm instead of 4 mm. Apple's butterfly design of 2015 pushed this to about 0.5 mm and was abandoned by 2021 because of reliability complaints.
10. **Optical.** The stem carries a small flag. At rest the flag blocks an infrared beam between an emitter and a photo-transistor. Pressing lifts or lowers the flag and the beam changes.
11. There is no metal-to-metal contact, so there is no contact bounce. The signal still has to cross a threshold, and comparators near their threshold can chatter, so firmware still applies a small filter.
12. **Hall effect / magnetic.** A magnet in the stem moves past a sensor that measures magnetic field strength. The output is not on or off. It is a number that says how far down the key is.
13. That means the actuation point is a software setting. Wooting's Lekker switch is specified at 40 gf actuation, 60 gf bottom out, 4.0 mm total travel, and an actuation point adjustable anywhere from 0.1 mm to 4.0 mm.
14. It also allows "rapid trigger": the key releases the moment you start moving up, rather than waiting to cross a fixed point. That is a firmware behaviour, not a physical one.

21.1.5 TECHNICAL - the engineer's version

1. Contact bounce is a mechanical resonance of the moving contact against the fixed contact, with the contact modelled as a damped mass-spring system. Bounce duration scales with contact mass and inversely with damping.
2. Cherry's MX1A-Lxx datasheet, which covers MX Red, specifies bounce time under 5 ms measured during actuation at a closing speed of 0.4 m/s. The MX2A revision keeps the same 45 cN, 2.0 mm, 4.0 mm figures.
3. Typical debounce strategies in keyboard firmware:

Strategy	How it works	Added latency
Eager / asym	Fire on first edge	~0 ms on press
Defer / symmetric	Wait N stable scans	3 to 8 ms
Per-key timer	5 ms lockout per key	up to 5 ms
Integrator	Count up/down to limit	2 to 6 ms

4. QMK, the widely used open-source keyboard firmware, ships `DEBOUNCE` defaulting to 5 milliseconds, with selectable algorithms named `sym_defer_g`, `sym_eager_pk`, `asym_eager_defer_pk` and others. The

`eager` variants report the press immediately and then lock the key out, giving zero added press latency at the cost of possible false presses on a dirty contact.

5. Actuation force is quoted in centinewtons (cN) by Cherry and in grams-force (gf) by most others. 1 cN is 0.01 N; 1 gf is 0.00980665 N. So 45 cN is about 45.9 gf. Treating cN and gf as interchangeable introduces about 2 percent error, which is inside the manufacturing tolerance anyway.
6. Rated lifetime: Cherry MX2A claims 100 million actuations per key. Rubber domes are typically rated 5 to 20 million. Buckling spring keyboards, the IBM Model M design of 1985, are commonly rated at 25 million and many are still working after forty years.
7. IBM patented the buckling spring mechanism in 1978. In it, a coil spring under compression buckles sideways, and the buckling motion flips a hammer that strikes a capacitive sense pad on the PCB. Actuation and the audible click are the same event, which is why the feedback is honest, unlike a clicky mechanical switch where the click and the electrical actuation are produced by different parts and can drift apart.
8. Optical switches used in production include Razer's opto-mechanical design (2018) and Bloody's LK Libra series. Vendors claim 0.2 ms actuation latency. That is a **marketing claim** about the sensor alone and does not include scan, debounce filtering, or USB, which dominate.
9. Hall-effect switches use a linear Hall sensor such as the Allegro A1304 or equivalent, sampled by the microcontroller's ADC. Resolution of the reported depth is limited by ADC bits and calibration drift, not by the magnet.
10. Tools: an oscilloscope across the switch terminals shows the bounce directly. On a keyboard you control, a logic analyser on the matrix pins shows the scan and the debounce window together.

21.1.6 WORDS - remember these

1. Switch — the thing under a key that joins two wires — an electromechanical or non-contact single-pole momentary contact.
2. Actuation point — how far down the key must go before it counts — pre-travel distance at which the contact closes, typically 2.0 mm on Cherry MX.
3. Travel — total distance the key can move — total actuator travel, typically 4.0 mm full size, 1 to 2 mm scissor.
4. Actuation force — how hard you must press — operating force in cN or gf, typically 45 to 60.
5. Bounce — the flickering of the contact when it first touches — contact chatter caused by mechanical resonance, under 5 ms on Cherry MX.
6. Debounce — waiting until the flickering stops — firmware filtering of contact chatter, typically a 5 ms stability window.
7. Hall effect — sensing a magnet instead of touching metal — a linear magnetic field sensor giving continuous key depth rather than a binary state.

21.2 The key matrix

21.2.1 PLAIN - in simple words

1. A full-size keyboard has 104 keys. It does not have 104 wires going to the chip inside.
2. That would need 104 pins on the chip plus a common return, and cheap chips do not have that many pins.
3. Instead the keys are arranged in a grid, like squares on graph paper.
4. There are wires running across, called rows, and wires running down, called columns.
5. Each key sits where one row crosses one column, and when pressed it joins that row to that column.
6. With 8 rows and 18 columns you get 144 crossing points, enough for 104 keys, using only 26 wires.
7. The chip cannot look at all crossings at once. So it checks them in turn, very fast, over and over.
8. It energizes one row, reads all the columns, then moves to the next row. Doing all rows once is called a scan.

9. A good keyboard does a complete scan a thousand times a second. A cheap one might do it a hundred times a second.

21.2.2 PLAIN - a picture in your head

1. Think of a block of flats with a caretaker who has to know which flats have someone at home.
2. There are 8 floors and 18 flats on each floor. The caretaker cannot see all 144 flats at once.
3. So the caretaker switches on the light in the corridor of floor 1 only, then walks along and looks at which doorways show light spilling out.
4. Then floor 1's light goes off, floor 2's comes on, and the same check happens. And so on to floor 8.
5. One full trip through all 8 floors is one scan. If the caretaker can do it a thousand times a second, nothing gets missed for more than a millisecond.
6. Now the problem. Suppose three flats are occupied and they happen to form three corners of a rectangle. Light can leak from floor to floor through the open doors and make the fourth corner look occupied too.
7. That false fourth key is called ghosting, and it is a real thing that happens on cheap keyboards.
8. The fix is a one-way door on every flat: light can go out but not back in. In electronics, a one-way door is a diode.

Where this comparison breaks: the caretaker is slow and the keyboard is not. A scan of the whole matrix takes tens of microseconds of actual work; the rest of the millisecond is the chip waiting on purpose. And the "light leaking" is not light, it is a current finding a path backwards through two other closed switches.

21.2.3 PLAIN - a worked example

1. Here is a tiny 3x3 matrix. Rows R0 to R2 run across. Columns C0 to C2 run down. A key is at every crossing.

```
        C0     C1     C2
       |      |      |
R0 ----+------+------+----
       |      |      |
R1 ----+--Q---+--W---+----      Q at R1/C0, W at R1/C1
       |      |      |
R2 ----+--A---+--S---+----      A at R2/C0, S at R2/C1
       |      |      |
```

2. The chip drives R0 low and reads C0, C1, C2. All read high, so no key on row 0 is down.
3. The chip drives R1 low. C0 reads low, so Q is down. C1 and C2 read high.
4. The chip drives R2 low. C0 reads low, so A is down.
5. Now hold three keys at once: Q (R1/C0), W (R1/C1) and A (R2/C0).
6. Drive R2 low. Current goes out of R2, through A, up C0, back through Q into R1, then out through W into C1.
7. So C1 reads low as well, and the chip believes S (R2/C1) is pressed. It was not. That is a ghost key.
8. Put a diode in series with every switch, all facing the same way. Now current can go from row to column but never from column back to row.
9. The path through Q backwards is blocked. No ghost. Every combination of keys is now readable. That property is called N-key rollover.
10. Some cheap keyboards do not add diodes. Instead the firmware detects the rectangle pattern and refuses to report anything, which is called masking or blocking. You press three keys and get two. That is safer than a ghost and cheaper than 104 diodes.

21.2.4 PLAIN - what is really happening inside

1. The chip's row pins are outputs. The column pins are inputs with pull-up resistors, meaning each column reads high unless something pulls it down.

2. To scan a row, the chip sets that one row pin low and leaves all the others floating or high.
3. It then waits a short settle time, typically a few microseconds, because the wires have capacitance and the voltage does not change instantly.
4. It reads all the column pins in one operation, usually a single register read, giving 18 bits at once.
5. It stores those bits in a row of a bitmap in memory. After 8 rows it has the full state of all 144 crossings.
6. It compares that bitmap with the previous one. Any bit that changed is a candidate key event.
7. Candidates go into the debounce filter. Only after a candidate has held its new value for the debounce window does it become a real event.
8. Real events go into a queue for the next stage, which is turning a matrix position into a code.
9. The scan rate is a choice. Scanning faster costs power and leaves less CPU time for lighting effects and macros. Scanning slower adds latency.
10. On average, waiting for the scan costs you half a scan period, because your press lands at a random point in the cycle. At 1000 Hz that average is 0.5 ms. At 100 Hz it is 5 ms.

21.2.5 TECHNICAL - the engineer's version

1. A typical full-size layout uses an 8 x 18 or 6 x 21 matrix. Pin count is rows plus columns, so 26 or 27 general-purpose I/O pins.
2. Diodes are small-signal types, commonly 1N4148 in through-hole builds or BAV70 / 1N4148W in surface mount, chosen for low forward voltage (about 0.7 V) and fast recovery.
3. Orientation is a convention, and both exist. `COL2ROW` means the diode cathode faces the row; `ROW2COL` is the reverse. QMK has a compile-time `DIODE_DIRECTION` setting for exactly this reason.
4. Scan rates in real firmware:

Board / firmware	Scan rate	Note
Apple IIe (AY-3600)	~556 Hz	1978 dedicated chip
Ergodox (original fw)	~167 Hz	claimed by project
QMK on ARM, typical	1000+ Hz	often 2 to 5 kHz
Cheap membrane board	100 to 200 Hz	common, not specified

5. Dan Luu's 2017 analysis computed the Apple IIe's General Instrument AY-3600 keyboard encoder at a 1.8 ms scan delay and a 6.8 ms debounce delay, under 8.6 ms of internal keyboard logic, using the RC values on the board.
6. N-key rollover over USB requires more than diodes. The 8-byte boot report carries only six key codes. Full NKRO needs a report-protocol descriptor with a bitmap of usages, and most NKRO keyboards expose two HID interfaces: a boot-compatible one for BIOS use and an NKRO one for the operating system.
7. Ghosting terminology: strictly, **ghosting** is a phantom key appearing; **masking** or **jamming** is a real key being suppressed. Consumer marketing uses "anti-ghosting" for both, which is imprecise.
8. Matrix settle time matters at high scan rates. With long traces and no series resistors, a 1 microsecond settle can produce false reads; QMK exposes `DEBOUNCE`, `MATRIX_IO_DELAY` (default 30 microseconds) and `DIRECT_PINS` for keyboards small enough to skip the matrix entirely.
9. Hall-effect boards do not have a conventional matrix scan of digital states. They multiplex analogue sensors into an ADC. The limiting rate is ADC conversion time times the number of keys, which is why 8 kHz analogue boards are recent and expensive.
10. Tools: `sudo evtest`, `wev` on Wayland or `xev` on X11 will show whether keys are being lost. A logic analyser on the row pins shows the scan period directly.

21.2.6 WORDS - remember these

1. Matrix — a grid of wires so a few pins can read many keys — an m x n row/column arrangement needing m+n GPIO pins for m*n keys.

2. Scan — one full pass over all rows — the periodic sequential strobe and sample loop, typically 100 Hz to 5 kHz.
3. Ghosting — a key that appears pressed but is not — a phantom detection from a sneak current path through three closed switches.
4. Masking — keys deliberately dropped to avoid ghosts — jamming, the firmware suppressing an ambiguous rectangle pattern.
5. Diode — a one-way valve for current — a small-signal rectifier per key blocking the reverse sneak path, giving NKRO.
6. N-key rollover — every key readable at once — unlimited simultaneous key detection, requiring both per-key diodes and a suitable HID report format.

21.3 The keyboard's own computer

21.3.1 PLAIN - in simple words

1. Inside your keyboard there is a small computer. A real one, with a processor, memory and a program.
2. It is not powerful. It might run at 16 to 100 million cycles a second, with a few kilobytes of memory. Your phone is a hundred thousand times bigger.
3. Its whole job is: scan the grid, clean up the signal, decide what changed, and tell the host machine.
4. When it decides that the key at row 2, column 0 has gone down, it does not send "row 2, column 0". It looks up a table.
5. The table says: this position on this keyboard corresponds to code number such-and-such.
6. That number is called a scan code, or in USB language a usage code.
7. Here is the part people get wrong. That number is not a letter. It is not the letter A.
8. It is a name for a physical position on the keyboard. "The key that is one to the right of Caps Lock", roughly.
9. If you switch your computer to a French layout, that same key produces Q on screen. The keyboard sent the identical number both times.
10. So the keyboard does not know your language, your layout, or what you typed. It only knows which physical key moved.

21.3.2 PLAIN - a picture in your head

1. Imagine a hotel with numbered rooms and a bell board behind the reception desk.
2. When someone in room 214 presses their bell, a flag drops on the board labelled "214".
3. The board does not say "Mrs Patel wants tea". It says "214".
4. Reception has a separate list mapping rooms to guests. Guests change every week. The bell board never changes.
5. The bell board is the keyboard. The room number is the scan code. The guest list is your keyboard layout, and it lives at reception, not in the wiring.
6. This split is why a hotel can rewire nothing when new guests arrive, and why your computer can switch between English, French and Devanagari instantly without you touching the keyboard.

Where this comparison breaks: the hotel board reports only "someone pressed". The keyboard reports both press and release, and reports the full set of keys currently held, so the computer can know that Shift and A are down at the same moment. Also, some keyboards do relabel themselves in firmware, which would be like the bell board printing guest names. That is a firmware remap, and it is the exception, not how layouts normally work.

21.3.3 PLAIN - a worked example

1. You press the A key on a standard 104-key board wired to a US layout.
2. The firmware's key map is an array. Position [row 2][column 0] holds the value `KC_A`, which in QMK is the number 0x04.

3. That 0x04 is the USB HID usage code for "Keyboard a and A" on Usage Page 0x07.
4. The name in the specification is literally "Keyboard a and A". The specification refuses to pick one, because that choice is not the keyboard's to make.
5. Now compare the three older PS/2 scan code sets for the same physical key.

System	Make code for A	Break code
Set 1 (XT, 1981)	0x1E	0x9E
Set 2 (AT, 1984)	0x1C	0xF0 0x1C
Set 3 (PS/2, 1987)	0x1C	0xF0 0x1C
USB HID (1996 on)	0x04	absent from report

6. Four different numbers for one physical key, across four standards. None of them is the ASCII code for A, which is 0x41, or for a, which is 0x61.
7. Notice the last row. USB does not send a release code at all. It sends the full list of keys currently held down. A key disappearing from the list is the release.
8. Now hold Shift and press A. The firmware does not combine them. Shift is reported as a separate bit in a separate byte, and A is still 0x04.
9. Combining them into a capital letter is somebody else's job, several stages later.

21.3.4 PLAIN - what is really happening inside

1. The microcontroller runs a loop, forever, in this order.
2. Scan the matrix into a bitmap. Compare with the last bitmap. Feed changes into the debounce filter.
3. For each debounced change, look up the key map for the current layer.
4. Layers are a firmware idea. Holding a Fn key switches to a second key map array, so the same physical key gives a different code. This is how a 60-key board offers arrow keys.
5. Sort the resulting codes into modifiers and normal keys. The eight modifiers, left and right Ctrl, Shift, Alt and GUI, go into a single byte as bits.
6. Normal keys go into a list, in the order they were pressed, oldest first.
7. Build the report. Byte 0 is the modifier bits. Byte 1 is reserved and always zero. Bytes 2 to 7 are up to six normal key codes.
8. Hand the report to the USB stack, which puts it in a buffer attached to an endpoint and waits for the host to ask.
9. If more than six normal keys are held, the boot protocol has no room. The specification says to fill all six slots with 0x01, the code named ErrorRollOver.
10. There is no auto-repeat in the keyboard. Hold a key and the keyboard just keeps reporting the same list. The repeating you see when you hold a key is generated by the operating system with a timer.

21.3.5 TECHNICAL - the engineer's version

1. Common keyboard microcontrollers: Atmel ATmega32U4 (8-bit AVR, 16 MHz, 2.5 KB SRAM, native USB) in older custom boards; STM32F072/F103 and RP2040 in newer ones; proprietary parts from Holtek, SONiX and Nuvoton in mass-market boards.
2. The USB HID Usage Tables document defines Usage Page 0x07 as the Keyboard/ Keypad page. Selected values:

Usage ID	Name in the specification
0x00	Reserved (no event)
0x01	ErrorRollOver
0x04	Keyboard a and A
0x1D	Keyboard z and Z

Usage ID	Name in the specification
0x28	Keyboard Return (ENTER)
0xE1	Keyboard LeftShift

3. Note 0xE1. Left Shift has a usage ID, but in the boot report it is carried as bit 1 of byte 0 rather than as a key code. Both representations exist and report-protocol descriptors normally declare the modifier byte as a bitmap of usages 0xE0 to 0xE7.
4. PS/2 scan code set 2 is what AT-class keyboards actually send on the wire. The 8042 keyboard controller on the motherboard translates set 2 into set 1 before the operating system sees it, when translation is enabled. That translation is why so much documentation disagrees about which set is "the" scan code set. Set 3 was defined for the IBM 3270 terminal style and is rarely used, though it has the useful property that every key can be individually configured as make-only, make-break or typematic.
5. USB was published as version 1.0 in January 1996 by a group including Intel, Microsoft, Compaq, IBM, DEC, NEC and Nortel. The HID class specification version 1.11 dates from 2001 and is still the current document.
6. Legacy support: the BIOS or UEFI cannot run a full USB stack early in boot, so the boot protocol exists. `SET_PROTOCOL` with value 0 selects boot protocol, value 1 selects report protocol. Firmware also historically used SMM-based USB legacy emulation to make a USB keyboard appear at I/O ports 0x60 and 0x64 like an 8042.
7. Firmware options in wide use: QMK (2015 on, from Jun Wako's TMK), ZMK for wireless, VIA and Vial for runtime remapping, and closed vendor firmware. QMK's `keymap.c` is a C array of arrays, one per layer.
8. Latency inside the microcontroller for the lookup and report build is small: on a 16 MHz AVR, a full 8x18 scan plus debounce plus report build measures in the tens of microseconds, so under 0.1 ms. The scan period, not the computation, is the cost.
9. Where the layout is NOT decided, said plainly: not in the switch, not in the matrix, not in the scan code table, and not in the USB report. All of those carry physical position only.

21.3.6 WORDS - remember these

1. Microcontroller — the tiny computer in the keyboard — an MCU with CPU, flash, SRAM and a USB device peripheral, typically 8 to 100 MHz.
2. Scan code — a number naming a physical key — a make/break code from PS/2 sets 1, 2 or 3, or a HID usage ID on Usage Page 0x07.
3. Usage code — the USB name for a key — a 16-bit usage page plus usage ID pair identifying a control, defined in the HID Usage Tables.
4. Modifier — Shift, Ctrl, Alt, Cmd — usages 0xE0 to 0xE7, carried as a bitmap in byte 0 of the boot keyboard report.
5. Layer — a second key map reached by holding a key — a firmware-level alternate keycode array, selected by momentary or toggle actions.
6. ErrorRollOver — the code meaning "too many keys" — usage 0x01 placed in all six slots when more keys are held than the report can carry.

21.4 Getting the data out: USB HID

21.4.1 PLAIN - in simple words

1. The keyboard now has an 8-byte message ready. It has to get it to the computer.
2. USB does not let a device speak whenever it wants. The computer is in charge of the wire, always.
3. The keyboard cannot interrupt. It can only put its message in a small mailbox and wait to be asked.
4. The computer asks on a fixed schedule. "Anything for me?" Over and over, forever, whether you are typing or not.

5. If the mailbox is empty, the keyboard answers with a tiny "nothing", which costs almost no time on the wire.
6. If the mailbox has the message, the keyboard hands it over and the computer acknowledges it.
7. The schedule is normally once every millisecond. Some fast devices are asked every 125 microseconds, which is eight times a millisecond.
8. So on average your key press waits half of one asking-interval before it can leave the keyboard.
9. That mailbox has a name: an endpoint. The kind of asking is called an interrupt transfer, which is a confusing name because nothing is interrupted. The device is polled.

21.4.2 PLAIN - a picture in your head

1. Think of a postal round in a village with one postman and many houses.
2. The postman walks the same route every minute, stopping at every house that has asked to be visited.
3. At each stop the postman says "anything to send?" The householder either hands over a letter or says "nothing today".
4. A householder cannot run down the street and shove a letter into the postman's bag out of turn. They must wait for the visit.
5. If you write a letter just after the postman leaves, it waits nearly a full minute. If you write it just before he arrives, it goes almost at once.
6. On average it waits half a minute. That is exactly the average USB polling delay.
7. A house can ask to be visited more often, and the postman will agree if the route has room. That is what setting a shorter polling interval means.

Where this comparison breaks: the postman visits in a fixed order, whereas USB visits in a schedule computed by the host controller that can be reordered and that reserves bandwidth in advance. And the postman carries the letter away; USB copies the data and leaves the device's buffer to be refilled.

21.4.3 PLAIN - a worked example

1. Here is the exact 8-byte boot keyboard report format.

```
byte 0 : modifier bits
byte 1 : reserved, always 0x00
byte 2 : key code slot 1
byte 3 : key code slot 2
byte 4 : key code slot 3
byte 5 : key code slot 4
byte 6 : key code slot 5
byte 7 : key code slot 6
```

2. The modifier byte, bit by bit, lowest bit first:

```
bit 0 : Left  Ctrl     bit 4 : Right Ctrl
bit 1 : Left  Shift    bit 5 : Right Shift
bit 2 : Left  Alt      bit 6 : Right Alt
bit 3 : Left  GUI      bit 7 : Right GUI
```

3. You press A alone. The report on the wire is:

```
00 00 04 00 00 00 00 00
```

4. Read it: no modifiers, reserved zero, key code 0x04 in slot 1, nothing else.
5. You now hold Left Shift and press A. Two reports go out, because Shift lands first.

```
02 00 00 00 00 00 00 00     Left Shift down, no keys
02 00 04 00 00 00 00 00     Left Shift down, A down
```

6. Byte 0 is 0x02, which is binary 00000010, so bit 1 is set, which is Left Shift. Byte 2 is 0x04, which is the A key.
7. Notice what is not there. There is no capital A anywhere. There is no character at all. Two independent physical facts: a modifier is held, and a key position is down.
8. You release A but keep Shift. The report becomes `02 00 00 00 00 00 00 00` again. The disappearance of 0x04 is the release event.
9. You release Shift. The report becomes all zeros.
10. Now type A, then B without releasing A, then C without releasing either:

```
00 00 04 00 00 00 00 00     A
00 00 04 05 00 00 00 00     A then B
00 00 04 05 06 00 00 00     A then B then C
00 00 05 06 00 00 00 00     A released, B and C shuffle down
```

11. Press more than six normal keys at once and the boot report cannot carry them, so all six slots become 0x01:

```
20 00 01 01 01 01 01 01     Right Shift held, rollover error
```

21.4.4 PLAIN - what is really happening inside

1. When you plug the keyboard in, the host asks it to describe itself. This is called enumeration.
2. The device returns a device descriptor, a configuration descriptor, one or more interface descriptors, endpoint descriptors and a HID report descriptor.
3. The interface descriptor says class 3, which means HID. Subclass 1 means it supports the boot protocol. Protocol 1 means keyboard, protocol 2 means mouse.
4. One of the endpoint descriptors says “interrupt IN”, meaning data flows device to host on a polled schedule. It carries a field called `bInterval`.
5. `bInterval` sets how often the host asks. On a full-speed device it is a number of 1 ms frames, from 1 to 255. On a high-speed device it is an exponent: the period is 2 to the power of (bInterval minus 1), counted in 125 microsecond microframes.
6. So the fastest possible polling is 125 microseconds, which is 8000 times a second, and that requires a high-speed connection.
7. Once running, the host controller issues an IN token to the device’s address and endpoint number at each scheduled slot.
8. If the keyboard has a new report, it replies with a DATA0 or DATA1 packet containing the 8 bytes, and the host replies ACK.
9. If nothing changed, the keyboard replies NAK, which is 3 bytes on the wire and takes under a microsecond.
10. Bluetooth keyboards do none of this. They use HID over GATT, a profile on Bluetooth Low Energy, where reports become notifications on a GATT characteristic.
11. Bluetooth Low Energy has its own polling clock called the connection interval, and it is much coarser. That is the single biggest reason wireless keyboards feel slower.

21.4.5 TECHNICAL - the engineer’s version

1. USB speeds and their interrupt endpoint limits:

Speed	Bit rate	Min interrupt period
Low speed	1.5 Mbit/s	10 ms (bInterval 10)
Full speed	12 Mbit/s	1 ms (bInterval 1)
High speed	480 Mbit/s	125 us (bInterval 1)
SuperSpeed	5 Gbit/s	125 us (bInterval 1)

2. Per the USB 2.0 specification, section 9.6.6, a low-speed interrupt endpoint must declare `bInterval` between 10 and 255 in 1 ms frames; a full-speed one between 1 and 255 in 1 ms frames; and a high-speed one between 1 and 16 as an exponent, giving a period of 2^(bInterval-1) microframes of 125 us.
3. Most plain USB keyboards enumerate as low speed or full speed. Dan Luu's 2017 measurements list many as "USB FS". A low-speed keyboard with `bInterval` 10 is polled 100 times a second and adds an average of 5 ms.
4. Gaming peripherals advertising 1000 Hz are full speed with `bInterval` 1. Those advertising 4000 Hz or 8000 Hz must be high speed with `bInterval` 1 or 2, because full speed physically cannot go below a 1 ms frame.
5. The report descriptor for a boot-compatible keyboard is a well-known 63-byte blob. Its meaning, decoded:

```
Usage Page (Generic Desktop), Usage (Keyboard)
Collection (Application)
  Usage Page (Keyboard/Keypad)
  Usage Min (0xE0) Usage Max (0xE7)      modifiers
  Logical Min (0) Logical Max (1)
  Report Size (1) Report Count (8)
  Input (Data, Variable, Absolute)       -> byte 0
  Report Size (8) Report Count (1)
  Input (Constant)                       -> byte 1 reserved
  Report Size (8) Report Count (6)
  Logical Min (0) Logical Max (101)
  Usage Min (0) Usage Max (101)
  Input (Data, Array)                    -> bytes 2..7
End Collection
```

6. Note the difference in the two Input items. The modifier byte is `Variable`, meaning each bit is its own control. The six key slots are `Array`, meaning each byte holds the index of whichever control is active. That single distinction is why modifiers are a bitmap and keys are a list.
7. NKRO keyboards add a second report, or a second interface, declaring a `Variable` bitmap over usages 0x00 to 0xF7, which is 31 bytes. Every key then has its own bit and there is no six-key limit.
8. Bluetooth: HID over GATT Profile (HOGP) 1.0 was adopted by the Bluetooth SIG in 2011. Connection interval is negotiated in units of 1.25 ms, with a permitted range of 7.5 ms to 4 s. Apple's accessory guidelines historically asked for intervals no shorter than 15 ms for most accessories, with exceptions for HID. Vendors using proprietary 2.4 GHz dongles, such as Logitech Lightspeed or Razer HyperSpeed, bypass Bluetooth entirely and can hold a 1 ms interval.
9. Tools: `lsusb -v` on Linux prints `bInterval` and the full descriptor tree. `usbhid-dump` prints report descriptors. On Windows, USB Device Tree Viewer shows the same. Wireshark with `usbmon` captures the actual reports.
10. Honest note on 8000 Hz polling: it removes at most about 0.44 ms of average wait compared with 1000 Hz. It also multiplies host interrupt load by eight. Whether that trade is worth it is contested, and blind tests have not shown a consistent human benefit above roughly 1000 Hz.

21.4.6 WORDS - remember these

1. Endpoint — a mailbox on the device — a uniquely addressed source or sink of data, identified by number and direction.
2. Interrupt transfer — the host asking on a timer — a guaranteed-bandwidth, bounded-latency polled transfer type, despite the name.
3. bInterval — how often the host asks — the polling interval field of the endpoint descriptor, in frames or as a power-of-two microframe exponent.
4. Boot protocol — the simple fixed 8-byte format — the reduced report format defined by HID subclass 1 so firmware can drive a keyboard without a parser.
5. Report descriptor — the device explaining its own data format — a self-describing byte stream of HID items parsed into a control tree.
6. NAK — "nothing to send" — a handshake packet meaning the endpoint has no data ready, costing about 3 bytes of bus time.
7. HOGP — Bluetooth's version of HID — HID over GATT Profile, carrying reports as BLE notifications on a negotiated connection interval.

21.5 Into the machine: the host controller

21.5.1 PLAIN - in simple words

1. The 8 bytes have left the keyboard and travelled down the cable. They arrive at a chip on the motherboard called the host controller.
2. The host controller is the thing that actually runs the USB schedule. It is the postman from the earlier picture.
3. It is not the CPU. It is a separate piece of hardware that works while the CPU is doing other things.
4. The operating system has already told the host controller, in advance, where in memory to put anything that arrives from this keyboard.
5. So when the bytes arrive, the host controller writes them straight into main memory itself, without asking the CPU.
6. Hardware writing to memory on its own is called DMA, direct memory access.
7. Then the host controller writes a small note into another area of memory saying "a transfer finished, here is what happened".
8. Only then does it poke the CPU, by sending a special message that makes the CPU stop what it is doing and run some kernel code.
9. That poke is an interrupt. This time the word is accurate: something really is interrupted.

21.5.2 PLAIN - a picture in your head

1. Think of a busy kitchen with a chef and a delivery hatch.
2. The chef does not stand at the hatch waiting for deliveries. That would waste the whole day.
3. Instead there are labelled shelves. The chef has already written a list: "onions go on shelf 3, flour goes on shelf 7".
4. The delivery person puts the goods on the right shelf without speaking to the chef at all. That is DMA.
5. Then the delivery person writes a line in a notebook by the door: "onions, shelf 3, 10:42, all fine". That notebook is the event ring.
6. Finally, they ring a bell. The chef stops chopping, wipes their hands, reads the notebook, and deals with whatever arrived.
7. The bell is the interrupt. Ringing it costs the chef a few seconds of disruption, so you do not ring it for every single carrot.

Where this comparison breaks: the chef can ignore the bell for a while; a CPU takes the interrupt at the very next instruction boundary unless interrupts are explicitly masked. And the notebook is a fixed-size

circular list, so if the chef reads too slowly, the delivery person eventually has to stop and wait.

21.5.3 PLAIN - a worked example

1. Modern machines use xHCI, the eXtensible Host Controller Interface, released by Intel in 2010. It replaced the older UHCI, OHCI and EHCI designs.
2. Before any keys are pressed, the driver has set up these structures in ordinary system memory.

Structure	What it holds
Device Context Base Array	pointer per USB device
Transfer Ring	queued work per endpoint
Event Ring	completed-work notices
Command Ring	driver-to-controller orders

3. Each entry in these rings is a 16-byte block called a TRB, a Transfer Request Block.
4. The driver puts a Normal TRB on the keyboard's transfer ring saying: "when data arrives on this endpoint, write up to 8 bytes to physical address 0x1A2B3000, then post an event".
5. The driver then writes to a doorbell register on the controller to say "there is new work on this ring". Doorbells are the only place the driver pokes the hardware directly.
6. Time passes. You press A. At the next 1 ms slot, the controller polls the keyboard and gets the 8 bytes.
7. The controller performs a DMA write of those 8 bytes to 0x1A2B3000. This goes over PCI Express, through the IOMMU if one is enabled, into RAM.
8. The controller then writes a Transfer Event TRB onto the event ring, saying which endpoint, how many bytes, and a completion code of Success.
9. The controller then raises an MSI interrupt, which is itself just a PCI Express memory write to a special address that the interrupt controller watches.
10. The CPU takes the interrupt. Total elapsed time from the last bit on the USB wire to the first instruction of the kernel handler: roughly 2 to 10 microseconds on a modern desktop.

21.5.4 PLAIN - what is really happening inside

1. Let us be precise about "the CPU is interrupted mid-instruction".
2. On x86 and on Arm, the processor checks for pending interrupts at instruction boundaries, not in the middle of an instruction.
3. So the current instruction finishes and retires. A few very long instructions, such as x86 repeated string moves, are designed to be interruptible and restartable partway.
4. The core then does the following, in hardware, with no software involved.
5. It saves the current instruction pointer, the flags register and the stack information onto a stack.
6. It reads the interrupt vector number, a value between 0 and 255 on x86, that came with the MSI message.
7. It uses that number as an index into the Interrupt Descriptor Table, a table the kernel built at boot, to find the address of the handler.
8. It switches privilege level from user mode to kernel mode if it was in user code, and switches to the kernel stack.
9. It jumps to the handler. From the program's point of view, absolutely nothing happened; when the handler returns, execution resumes at the exact next instruction with all registers intact.
10. Whatever the CPU was running has now had its pipeline flushed and quite possibly its caches disturbed. That is the real cost of an interrupt, and it is why interrupt moderation exists.

21.5.5 TECHNICAL - the engineer's version

1. xHCI 1.0 was published by Intel in 2010; the current revision is 1.2 with errata. It unified low, full, high and SuperSpeed handling in one register interface, which the earlier split of UHCI/OHCI (USB 1.x) and EHCI (USB 2.0) did not.

2. Structures, precisely: the Device Context Base Address Array (DCBAA) is indexed by slot ID. Each device context contains a slot context and up to 31 endpoint contexts. Each endpoint context points at a transfer ring.
3. Rings are circular arrays of 16-byte TRBs. The last TRB in a segment is a Link TRB pointing back to the start. A Cycle Bit is toggled each time the producer wraps, which is how consumer and producer distinguish new entries from stale ones without a separate index.
4. The Event Ring Segment Table (ERST) describes the event ring's memory. The controller is the producer of events; the driver is the consumer and updates the Event Ring Dequeue Pointer register when it has processed them.
5. Doorbell registers: one per device slot, plus doorbell 0 for the command ring. Writing a doorbell tells the controller to re-examine a ring. This is the only per-transfer MMIO write in the fast path, which matters because MMIO writes to a PCIe device cost hundreds of nanoseconds.
6. Interrupts: MSI was introduced in PCI 2.2 (1998) and MSI-X in PCI 3.0 (2004). Both replace a physical interrupt line with a posted memory write to an address decoded by the interrupt controller, which removes the shared-line ambiguity and the need to read a device register to find out who interrupted.
7. xHCI provides an Interrupt Moderation register per interrupter, expressed in 250 nanosecond units, which sets a minimum gap between interrupts. Raising it reduces interrupt rate and increases worst-case latency. Exact defaults are an **implementation detail** that differs between the specification's reset value and what Linux, Windows and macOS drivers actually program.
8. IOMMU involvement: with an IOMMU enabled (Intel VT-d, AMD-Vi, Arm SMMU) the controller's DMA addresses are I/O virtual addresses translated per device. This costs an IOTLB lookup, normally tens of nanoseconds when hot, and it is what stops a malicious USB controller from reading arbitrary memory.
9. Rough interrupt entry costs on x86-64, order of magnitude:

Step	Typical cost
DMA write of 8 bytes	under 1 us
MSI write to APIC	~0.1 to 0.5 us
Hardware IDT dispatch	~0.1 to 1 us
Handler entry with mitigations	+0.2 to 1 us

10. Tools: `cat /proc/interrupts` shows per-CPU counts per IRQ, including `xhci_hcd`. `sudo perf stat -e irq:irq_handler_entry` counts handler entries. `sudo trace-cmd record -e xhci-hcd` traces xHCI ring activity.

21.5.6 WORDS - remember these

1. Host controller — the chip that runs the USB bus — the xHCI device that schedules transfers and DMAs data, independent of the CPU.
2. DMA — hardware writing to memory by itself — direct memory access, a bus master writing to system RAM without CPU involvement.
3. TRB — one entry in a work list — a 16-byte Transfer Request Block on an xHCI transfer, command or event ring.
4. Ring — a circular list in memory — a producer/consumer queue with a Cycle Bit distinguishing fresh from stale entries.
5. Doorbell — the poke that says "new work" — a memory-mapped register write telling the controller to re-scan a ring.
6. MSI — an interrupt sent as a memory write — Message Signaled Interrupt, a posted PCIe write to an address the interrupt controller decodes.
7. IDT — the table of interrupt handlers — the Interrupt Descriptor Table, 256 gate descriptors indexed by vector number on x86.

21.6 The kernel takes over

21.6.1 PLAIN - in simple words

1. The CPU is now running kernel code because of the interrupt. This code is called the interrupt service routine, or handler.
2. The handler must be fast. While it runs, other things are held up, so it does the minimum and defers the rest.
3. It reads the event note the controller left, finds out which device and which transfer finished, and marks that transfer complete.
4. It then schedules the real work to happen a moment later, outside the handler. On Linux that deferred half is a softirq or a work queue.
5. The real work starts in the USB core, the part of the kernel that knows about USB in general but nothing about keyboards.
6. The USB core looks at which driver claimed this interface and hands the 8 bytes to it. For a keyboard, that is the HID class driver.
7. The HID driver already read the device's report descriptor at plug-in time and built a map of where every field lives inside the report.
8. It uses that map to pull out "modifier bits" and "the six key slots", then compares with the previous report to work out what changed.
9. A key present now but not before is a press. A key present before but not now is a release.
10. Finally it translates each USB usage code into the operating system's own key code, and hands that to the input subsystem, which is the thing applications can actually read from.

21.6.2 PLAIN - a picture in your head

1. Imagine a large office building's post room.
2. A courier drops a sack at the loading bay and rings the bell. Somebody signs for it in ten seconds flat and goes back to work. That is the interrupt handler: sign, do not open.
3. Later, a sorter opens the sack. They do not read the letters. They look at the department code on each envelope and put it in the right pigeonhole. That is the USB core.
4. The department clerk opens the envelope. They know this department's forms, so they know that box 3 on the form is "employee number" and box 7 is "date". That is the HID driver using the report descriptor.
5. The clerk then rewrites the information onto the building's own standard form, because every department in the building uses different paper but the building's computer only accepts one format. That is the translation from HID usages to kernel key codes.
6. The standard form goes into a tray that anybody with permission can read from. That tray is the input subsystem.

Where this comparison breaks: the post room is sequential and slow. The kernel does all of this in a few microseconds, and the "tray" is a shared memory structure with a wake-up mechanism, so readers are woken the instant something lands rather than checking periodically.

21.6.3 PLAIN - a worked example

1. On Linux you can watch this directly. `evtest` opens a device in `/dev/input/` and prints every event.
2. Press and release A on a US keyboard and you see this.

```
Event: time 1755105143.201933, type 4 (EV_MSC),
       code 4 (MSC_SCAN), value 70004
Event: time 1755105143.201933, type 1 (EV_KEY),
       code 30 (KEY_A), value 1
Event: time 1755105143.201933, -------- SYN_REPORT --------
Event: time 1755105143.298114, type 4 (EV_MSC),
```

```
       code 4 (MSC_SCAN), value 70004
Event: time 1755105143.298114, type 1 (EV_KEY),
       code 30 (KEY_A), value 0
Event: time 1755105143.298114, -------- SYN_REPORT --------
```

3. Line by line. `EV_MSC` with `MSC_SCAN` and value `70004` is the raw hardware code: usage page 0x0007, usage 0x0004. The kernel passes it through so tools can remap unusual keys.
4. `EV_KEY` with code 30 is `KEY_A`, the Linux input key code for that physical key. It is 30, not 4 and not 65. A third numbering.
5. `value 1` means pressed. `value 0` means released. `value 2` means auto-repeat, generated by a kernel timer, not by the keyboard.
6. `SYN_REPORT` marks the end of one batch of simultaneous events. Everything between two `SYN_REPORT` lines happened at the same instant.
7. The gap between the two timestamps is 96.181 ms. That is how long the finger was on the key, which is a completely normal typing dwell time.
8. Notice what is still not present. Nothing anywhere says the letter A. Still no character.

21.6.4 PLAIN - what is really happening inside

1. Order of events inside Linux, in detail.
2. `xhci_irq` runs as the hard interrupt handler. It reads the event ring, finds the Transfer Event TRB, and calls the completion function attached to the URB.
3. An URB, USB Request Block, is the kernel's object for one transfer. Its completion runs the HID driver's `usb_hid_irq_in` callback.
4. The HID driver calls `hid_input_report` with the 8 bytes. This is where the parsed report descriptor is used.
5. `hid_input_report` walks the report's fields. For each field it extracts the bits and compares with the stored previous value.
6. For each changed usage it calls `hid_input_field`, which maps the HID usage to a Linux input event through `hidinput_hid_event`.
7. The mapping is a table. Usage page 0x07, usage 0x04 becomes `EV_KEY`, `KEY_A`, which is 30.
8. `input_event` is called, which puts the event into the input core.
9. The input core hands the event to every registered handler. `evdev` is one such handler and it appends to a per-open-file circular buffer, then wakes any process sleeping on that file descriptor.
10. Interrupt handlers must not sleep, so all of this after the initial read runs in softirq context or a tasklet. Under a normal load this whole path takes 5 to 50 microseconds.
11. Under a heavily loaded system with the CPU busy, the wake-up of the reading process is at the mercy of the scheduler, and that is where a millisecond can quietly disappear.

21.6.5 TECHNICAL - the engineer's version

1. Three operating systems, three input stacks, doing the same job:

OS	Layer names
Linux	usbcore, hid-core, input, evdev
macOS	IOUSBHostFamily, IOHIDFamily
Windows	USBXHCI, HIDCLASS, kbdclass

2. Linux: character devices appear as `/dev/input/eventN` with a stable alias under `/dev/input/by-id/`. The protocol is a stream of `struct input_event { struct timeval time; __u16 type; __u16 code; __s32 value; }`. Key codes live in `include/uapi/linux/input-event-codes.h`. `KEY_A` is 30, `KEY_LEFTSHIFT` is 42, `KEY_ENTER` is 28.

3. macOS: `IOHIDFamily` parses the report descriptor into `IOHIDElement` objects. Events become `IOHIDEvents`, then `NSEvents` once they reach the window server. Since macOS 10.15 (Catalina, 2019) reading raw HID input from another application requires Input Monitoring permission, which is why key loggers and remapping tools prompt for access.
4. Windows: `HIDCLASS.SYS` parses reports. `kbdhid.sys` translates HID usages into PS/2-style scan codes for backward compatibility, then `kbdclass.sys` feeds `Win32k`, which posts a `WM_KEYDOWN` message to a thread's message queue. Alternatively an application can call `RegisterRawInputDevice` and receive `WM_INPUT` with the raw HID report, bypassing the translation. Games do this, because the message path applies key repeat, focus filtering and accessibility features that a game does not want.
5. The Windows raw input path is genuinely lower latency than the message path, mostly because it avoids waiting for the message pump and avoids the translation layers, not because the reports arrive earlier.
6. Linux key code space: `KEY_MAX` is 0x2FF (767), so there is room for many more keys than any keyboard has. Codes above 255 cannot be expressed in the older X11 protocol without an offset of 8, which is where the famous "X keycode equals evdev code plus 8" rule comes from.
7. Timestamps: `evdev` timestamps are taken in the input core when the event is generated, not when userspace reads it. Since Linux 4.4 you can request `CLOCK_MONOTONIC` timestamps with `EVIOCSCLOCKID`, which is what compositors do so that input times and frame times are comparable.
8. Tools: `evtest`, `libinput debug-events`, `sudo showkey -s` for raw scan codes on a console, `xinput test-xi2` on X11, `wev` on Wayland, `dmesg` for enumeration messages, `hid-recorder` from the hid-tools project for capturing raw reports.

21.6.6 WORDS - remember these

1. ISR — the code that runs when hardware pokes the CPU — the interrupt service routine, running with interrupts partly disabled and forbidden to sleep.
2. URB — the kernel's object for one USB transfer — USB Request Block, submitted asynchronously with a completion callback.
3. HID class driver — the code that understands report layouts — the driver that parses report descriptors and emits generic input events.
4. evdev — the Linux input file interface — the event device layer exposing `/dev/input/eventN` as a stream of `input_event` structures.
5. Key code — the operating system's own number for a key — a `KEY_*` constant on Linux, a virtual key on Windows, distinct from both scan codes and characters.
6. SYN_REPORT — the marker for "that batch is finished" — an `EV_SYN` event grouping simultaneous events into one atomic update.

21.7 From key code to character

21.7.1 PLAIN - in simple words

1. So far, at every single stage, nothing has known what letter you typed.
2. The switch knew "closed". The matrix knew "row 2, column 0". The keyboard knew "usage 0x04". The kernel knew "KEY_A, value 1".
3. All of those are names for a position on a slab of plastic.
4. Now, finally, something turns that position into a character.
5. The thing that does it is a table called a keymap, and it belongs to your user account, not to your hardware.
6. The keymap says: for this key, with no modifiers, produce a. With Shift, produce A. With Alt Gr, produce something else again.
7. Change your keymap to French and the same key produces q and Q instead. You have not touched the keyboard.

8. Some keys produce nothing on their own. Press the acute accent key on a French keyboard and nothing appears. Press e afterwards and é appears. That is a dead key.
9. Some systems let you type a sequence: a special Compose key, then o, then c, giving the copyright sign. That is a compose sequence.
10. And for languages with thousands of characters, a whole extra program sits in the middle, taking your keystrokes and offering you candidate words to pick from. That is an input method editor.

21.7.2 PLAIN – a picture in your head

1. Think of a piano with 88 keys and an orchestra behind a curtain.
2. Each piano key sends a number to the conductor. Key 40, key 41, and so on.
3. The conductor holds a sheet that says what each number means today. Today's sheet says number 40 means a violin note.
4. Tomorrow the conductor swaps in a different sheet, and number 40 means a drum. The piano is unchanged. The keys are unchanged. The sound is different.
5. Now imagine a language with 50,000 sounds and only 88 keys. No sheet can map one key to one sound.
6. So instead the pianist plays a short phrase, and an assistant listens, guesses which of several possible sounds was meant, and holds up a small list to choose from.
7. That assistant is the input method editor. It sits between the keys and the result, and it is a real program with its own memory and its own predictions.

Where this comparison breaks: a piano key really does have a fixed physical string, and a keyboard key genuinely has nothing fixed at all. Also, the conductor's sheet in real systems is not one flat list. It is a two-dimensional table with a group axis, for which language you are in, and a level axis, for which modifiers you hold.

21.7.3 PLAIN – a worked example

1. One physical key. Four different keymaps. Four different results.

Keymap	Plain	With Shift
US QWERTY	a	A
French AZERTY	q	Q
German QWERTZ	a	A
Dvorak	a	A

2. The same physical key on a French AZERTY board is where Q lives, so the letter differs even though the code sent was identical.
3. Now dead keys. On a French keyboard, the circumflex key sends its own key code. The keymap marks it as a dead key.
4. Press it: nothing appears. The system remembers "a circumflex is pending".
5. Press e: the system combines pending circumflex with e and produces ê.
6. Press the space bar instead: the system produces the bare ^ character.
7. Now compose sequences. On Linux, if you set a Compose key, this file controls it:

```
/usr/share/X11/locale/en_US.UTF-8/Compose

<Multi_key> <o> <c>          : "(c)"   copyright
<Multi_key> <a> <e>          : "ae"    ae ligature
<Multi_key> <minus> <minus> <period> : "--"  en dash
```

8. Now an input method editor. To type the Japanese word for Tokyo you might type `toukyou` on a plain US keyboard.

9. The IME converts those letters into kana as you go, showing the five kana to-u-kyo-u underlined as pre-edit text. Nothing is committed yet.
10. You press space. The IME offers a candidate list of kanji spellings that all read "toukyou": the city name, a rarer compound, and others. You pick one with the arrow keys and press Enter.
11. Only then does the application receive the finished text. Everything before that was a conversation between you and the IME.

21.7.4 PLAIN – what is really happening inside

1. A keymap is not a simple list. It has two axes.
2. The first axis is the **group**, which means which layout is active. You can have several loaded and switch between them with a hotkey.
3. The second axis is the **level**, which is chosen by which modifiers you hold. Level 1 is plain, level 2 is Shift, level 3 is Alt Gr, level 4 is Alt Gr with Shift.
4. So a lookup is: this key code, in this group, at this level, gives this symbol.
5. Modifiers themselves are stateful. The system tracks which are held, which are latched (pressed once, applying to the next key), and which are locked (Caps Lock).
6. Caps Lock is not Shift. Caps Lock normally affects only letters, whereas Shift affects every key. That is why Shift-2 gives @ but Caps Lock plus 2 still gives 2.
7. Dead keys are implemented as a small state machine. The keymap marks certain symbols as combining. When one is produced, the system stores it and waits for the next symbol.
8. If the next symbol has no defined combination, most systems emit both separately rather than losing your keystroke.
9. An input method editor is a separate process. It gets keystrokes first, before the application, and decides whether to consume them or pass them on.
10. While it is composing, it sends the application "pre-edit" text, which the application is expected to display inline but not to treat as committed.
11. When you confirm, the IME sends a commit event with the final string, and the pre-edit is cleared.
12. Why this cannot live in the keyboard: an IME needs a dictionary of hundreds of thousands of words, learns from your habits, needs megabytes of storage, and must know what the application is doing. A 2 KB microcontroller cannot and should not do that.

21.7.5 TECHNICAL – the engineer's version

1. Linux and X11 use XKB, the X Keyboard Extension, standardized in X11R6.1 in 1996. Its data lives in the `xkeyboard-config` package under `/usr/share/X11/xkb/` split into `keycodes`, `types`, `compat`, `symbols` and `geometry`.
2. A symbols entry looks like this, and this is the actual syntax:

```
key <AC01> { [ a, A, ae, AE ] };
key <AD01> { [ q, Q, at,  Omega ] };
```

3. `<AC01>` is a position name meaning alphanumeric section, row C, key 01, counting from the bottom-left. The four entries are levels 1 to 4. Position names, not letters, are the identifiers. That is the whole point.
4. `libxkbcommon` is the modern library that implements this without needing an X server; Wayland compositors and many terminals use it directly.
5. macOS uses `.keylayout` XML bundles in `/Library/Keyboard Layouts/`, evaluated by `UCKeyTranslate` from the Carbon Unicode Utilities. Ukelele is the common editor for them.
6. Windows keyboard layouts are DLLs, historically built with the Microsoft Keyboard Layout Creator. `kbdus.dll` is the US layout. Applications call `ToUnicodeEx` to run a virtual key plus keyboard state through the active layout and get characters back.
7. Input method frameworks:

Platform	Framework	Common engines
Linux	IBus, Fcitx5	Mozc, Anthy, Pinyin
Windows	TSF (since 2001)	MS-IME, Pinyin
macOS	InputMethodKit	Kotoeri, Pinyin
Android	InputMethodService	Gboard

8. Scale of the problem: Unicode 16.0, published September 2024, defines 154,998 characters. A keyboard has around 104 keys. The ratio is roughly 1,500 characters per key. No hardware solution exists, and none can exist.
9. Korean is a special case worth knowing: Hangul syllables are algorithmically composed from jamo, so a 2-set keyboard plus a composition automaton produces all 11,172 modern syllable blocks with no dictionary at all. It is an input method, but a deterministic one.
10. Indic scripts need reordering as well as composition. In Devanagari, the vowel sign i is typed after its consonant but displayed before it, so the input method and the text shaper both have work to do on the same keystroke.
11. Tools: `setxkbmap -print -verbose 10` shows the active XKB configuration; `xkbcomp $DISPLAY out.xkb` dumps the compiled keymap; `xkbcli interactive-wayland` shows live keysym resolution; `ibus engine` and `fcitx5-remote` query the active input method.
12. Honest note on where experts disagree: some argue that firmware remapping, as offered by QMK and VIA, is better because the layout follows the keyboard between machines. Others argue it is worse because the operating system then cannot know what the layout is, breaking shortcut display, accessibility tools and input methods. Both are right about their own failure mode.

21.7.6 WORDS - remember these

1. Keymap — the table turning key positions into characters — a per-user mapping from key code plus modifier state to keysyms and text.
2. Group — which language layout is active — the XKB axis selecting among several loaded layouts, switched by a hotkey.
3. Level — which modifier row of the table applies — the XKB axis selected by Shift, Alt Gr and their combinations, typically four levels.
4. Dead key — a key that waits for the next one — a combining accent that produces no character alone but modifies the following one.
5. Compose sequence — a short recipe of keys giving one character — a multi-key sequence resolved by a Compose table.
6. IME — the helper program for large scripts — an input method editor converting keystrokes to candidate text through a dictionary and a pre-edit/commit protocol.
7. Pre-edit — text you are still composing — uncommitted composition text sent to the application for display but not inserted into the document.

21.8 The window system

21.8.1 PLAIN - in simple words

1. Your screen has many windows on it. Only one of them should receive what you type.
2. The program that decides this is the display server, or on modern Linux, the compositor.
3. It keeps a note of which window has focus, meaning which window is currently listening.
4. Focus changes when you click on a window, or use a keyboard shortcut to switch, or when a program asks for it and is allowed.
5. So the display server takes the key event from the operating system, looks at its focus note, and sends the event to that one program.

6. For a mouse click there is an extra step, because a click has a position. The server must work out which window is under that point, and then which part of that window.
7. Working out what is under a point is called hit testing.
8. Programs do not sit staring at their input. They sleep. The display server wakes them by delivering a message.
9. A program that is waiting for messages, handling them, and waiting again, is running an event loop. Almost every interactive program on earth has one.

21.8.2 PLAIN – a picture in your head

1. Picture a large open-plan office with a single receptionist and many desks.
2. Post arrives addressed to "the person currently working on the Henderson file". The receptionist keeps a note of who that is.
3. The note changes during the day. When it changes, the receptionist tells the old person "you are no longer on it" and the new person "you are now on it". Those are the focus-out and focus-in events, and programs really do get them.
4. Every worker has an in-tray on their desk and is asleep at their desk.
5. The receptionist drops a letter in a tray and taps the sleeper's shoulder. They wake, read the whole tray, deal with everything, and go back to sleep.
6. That is the event loop: sleep, wake, drain the tray, act, sleep again.
7. A worker who takes twenty minutes on one letter is not asleep and not reading their tray. Their tray fills up and their window appears frozen. That is exactly what an unresponsive application is.

Where this comparison breaks: a real receptionist can be bypassed. On a modern compositor the routing is enforced, and one application cannot read another application's keystrokes. On the old X11 design it could, which is a real security difference and one of the main reasons Wayland exists.

21.8.3 PLAIN – a worked example

1. Here is what an event loop actually looks like. This is pseudo-code, but it is very close to real code in every toolkit.

```
for (;;) {
    /* sleep until something happens */
    event = wait_for_next_event(connection);

    switch (event.type) {

    case KEY_PRESS:
        widget = focused_widget(window);
        text   = keymap_translate(event.keycode,
                                  event.modifiers);
        widget->on_key(widget, event.keycode, text);
        break;

    case BUTTON_PRESS:
        widget = hit_test(window, event.x, event.y);
        set_focus(widget);
        widget->on_click(widget, event.x, event.y);
        break;

    case RESIZE:
        relayout(window, event.width, event.height);
        break;

    case FRAME_CALLBACK:
        if (window->damaged)
```

```
            redraw(window);
        break;
    }

    if (window->damaged)
        request_frame_callback(window);
}
```

2. `wait_for_next_event` is where the program spends over 99 percent of its life. On Linux it is a `poll` or `epoll_wait` on a file descriptor.
3. Note there is no loop spinning and checking. The process is genuinely asleep and consumes no CPU until the kernel wakes it.
4. Note also that `hit_test` is called for the mouse and not for the keyboard. The keyboard goes straight to the focused widget.
5. And note the last two blocks. Drawing does not happen inside the key handler. The key handler only marks the window as needing redraw. The actual painting happens later, once, even if fifty keys arrived.

21.8.4 PLAIN - what is really happening inside

1. Four systems, four architectures. It helps to see them side by side.
2. **X11.** The X server owns the screen and all input. Applications are clients connected over a socket. The server sends a `KeyPress` event containing a keycode and a modifier mask.
3. In classic X11 the client then does the keymap lookup itself using XKB data fetched from the server. Any client could also ask to receive all key events through `XGrabKeyboard` or passive grabs, which is why keyloggers were trivial on X.
4. **Wayland.** There is no separate server. The compositor is the display server, the window manager and the compositor in one process.
5. At connect time the compositor sends the client the entire keymap as a file descriptor holding XKB text. The client maps it and uses `libxkbcommon` to translate.
6. Then `wl_keyboard.key` carries the raw evdev key code and a state, and `wl_keyboard.modifiers` carries the modifier state separately. Only the focused client gets them at all.
7. **Quartz on macOS.** The WindowServer process owns the screen. Events become `CGEvents`, then `NSEvents`. Each application has a `CFRunLoop`, and `NSApplication` pulls events from it.
8. **Windows.** Every GUI thread has a message queue. `GetMessage` blocks on it. `TranslateMessage` looks at a `WM_KEYDOWN` and, if the key maps to a character, posts an additional `WM_CHAR` with the character in it. `DispatchMessage` calls the window's procedure.
9. That Windows split is worth remembering: `WM_KEYDOWN` is the physical key, `WM_CHAR` is the character, and they are two separate messages produced by one press.
10. In all four systems, the moment the event is handed over, a context switch happens: the compositor's process stops running and the application's process starts. That switch costs 1 to 10 microseconds when the machine is idle, and can cost milliseconds when it is busy.

21.8.5 TECHNICAL - the engineer's version

1. History: X Version 11 Release 1 shipped on 15 September 1987 from MIT's Project Athena, with Robert Scheifler and Jim Gettys as the principal authors of the protocol. The protocol is still wire-compatible today, which is nearly forty years of stability and also the source of most of its problems.
2. Wayland was started by Kristian Hoegsberg in 2008 while at Red Hat. Version 1.0 of the protocol was released in October 2012. GNOME and KDE both default to Wayland sessions on current distributions, and Fedora dropped the GNOME X11 session in Fedora 41 (2024).
3. Input handling on modern Linux normally goes through `libinput`, which owns pointer acceleration, tap-to-click, palm rejection and gesture recognition, so that every compositor behaves the same way. It reads `/dev/input/eventN` directly.
4. Security model comparison:

System	Can app A see app B keys
X11	Yes, by design
Wayland	No, focus-only delivery
Windows	Only with raw input hooks
macOS	Only with granted permission

5. Wayland's frame callback (`wl_surface.frame`) is the mechanism that ties drawing to the display refresh. A client that draws only on frame callbacks never renders frames that will not be shown, which saves power but does add the wait to the latency.
6. X11 uses an offset in key codes. An X keycode is the evdev code plus 8, for historical reasons dating to the original X server key code range. So evdev `KEY_A` (30) is X keycode 38.
7. Event delivery cost: on Linux with a Wayland compositor, the path from `evdev` read to the client's socket becoming readable is typically 100 to 500 microseconds when idle. Under load with the compositor competing for CPU it can exceed 5 ms, which is why compositors run their input handling at elevated scheduling priority where they can.
8. Tools: `wev` prints Wayland events including the resolved keysym. `xev` does the same for X11. `libinput debug-events --verbose` shows the layer below. On Windows, Spy++ shows the message queue. On macOS, `Quartz Debug` and the `CGEventTap` API let you observe the stream.

21.8.6 WORDS - remember these

1. Display server — the program that owns the screen and input — the process holding the DRM master or equivalent and arbitrating input delivery.
2. Compositor — the thing that merges all windows into one image — on Wayland, the same process as the display server.
3. Focus — which window receives keyboard input — the keyboard focus target, tracked as state and changed by explicit focus events.
4. Hit testing — finding what is under a point — geometric resolution of a coordinate to a surface and then to a widget, respecting stacking and input regions.
5. Event loop — sleep, handle, sleep — the blocking dispatch loop at the heart of every interactive program, typically built on `poll` or `epoll`.
6. Frame callback — permission to draw for the next frame — a compositor notification synchronizing client rendering with display refresh.

21.9 The application reacts

21.9.1 PLAIN - in simple words

1. The text editor is now awake with a key event in its hand.
2. It asks: which part of me has the keyboard right now? A window can have many text boxes, buttons and menus.
3. The answer is the focused widget, which in a text editor is the main text area.
4. The text area receives the event and asks the toolkit: what character is this, given the current layout and modifiers?
5. The answer comes back: the character A.
6. The editor then modifies its text. It has a big structure in memory holding every character of the document.
7. It inserts A at the cursor position, and then moves the cursor one place to the right.
8. That change may push words onto the next line, which means the text has to be laid out again from that point.
9. Laying out text means deciding, for every character, which picture to draw and exactly where on the screen to put it.

10. Finally the editor says "this rectangle of my window is now wrong, please ask me to redraw it". It does not draw yet.

21.9.2 PLAIN - a picture in your head

1. Think of a typesetter in an old print shop, arranging metal letters in a frame.
2. You hand them a new letter to insert into the middle of a line.
3. They do not just drop it in. They push everything after it along.
4. If the line now runs past the edge of the frame, the last word has to move down to the next line, and that might push a word off that line too.
5. This cascade stops when a line still has room. Usually that is one or two lines. Occasionally, adding one letter to the first line of a page reflows the entire page.
6. The typesetter also knows that certain letter pairs sit badly next to each other. A capital A followed by a capital V looks better if they are pushed slightly closer. They adjust the spacing by hand.
7. That adjustment has a name and it is still used today: kerning.

Where this comparison breaks: the typesetter has one physical piece of metal per letter. Modern text has no fixed one-to-one relation between characters and shapes. One character can produce several shapes, several characters can produce one shape, and in some scripts the shapes get reordered relative to the characters.

21.9.3 PLAIN - a worked example

1. The document currently says `Hello world` and the cursor is after `Hello`, at index 5.
2. The editor stores the text. A simple editor uses a plain array, which is easy but means every insert copies everything after it.
3. Real editors use smarter structures.

Structure	Used by	Insert cost
Gap buffer	Emacs	O(1) near the gap
Piece table	MS Word, VS Code	O(log n)
Rope	some editors, Xi	O(log n)
Flat array	toy editors	O(n)

4. The insert happens. The buffer now holds `HelloA world` and the cursor moves to index 6.
5. Now layout. The editor walks the paragraph and splits it into runs of text that share a font, a size, a direction and a script.
6. Each run goes to a shaping engine. The shaper takes characters and produces glyphs with positions.
7. For `HelloA` in a normal Latin font, the mapping is nearly one to one:

```
char 'H' -> glyph 43, advance 722 units
char 'e' -> glyph 72, advance 444 units
char 'l' -> glyph 79, advance 222 units
char 'l' -> glyph 79, advance 222 units
char 'o' -> glyph 82, advance 500 units
char 'A' -> glyph 36, advance 667 units
```

8. Those advances are in font design units. A typical font has 1000 or 2048 units per em. At 16 pixels per em with 1000 units, one unit is 0.016 pixels.
9. So the A occupies 667 * 0.016 = 10.67 pixels of width. Fractional. That fraction matters and we come back to it when we rasterize.
10. The editor now knows the pixel rectangle that changed, expands it a little for safety, and calls something like `invalidate_rect(x, y, w, h)`.
11. Nothing has been drawn. One flag has been set and one rectangle recorded.

21.9.4 PLAIN - what is really happening inside

1. Let us be precise about characters, glyphs and shaping, because this is where most people's mental model is wrong.
2. A **character** is a unit of text meaning. The letter A is a character. It has a Unicode code point, U+0041.
3. A **glyph** is a picture in a font. It has a number that is meaningful only inside that one font file.
4. The mapping between them is not one to one, and here are the four ways it breaks.
5. **Ligatures.** The characters f and i can be drawn as one joined shape. Two characters, one glyph.
6. **Contextual forms.** In Arabic, most letters have four shapes depending on whether they are alone, at the start, in the middle, or at the end of a word. One character, four possible glyphs.
7. **Decomposition.** In some Indic scripts one character is drawn as several marks placed around a base. One character, several glyphs.
8. **Reordering.** In Devanagari the vowel sign for i is stored after its consonant but drawn before it. The glyph order does not match the character order.
9. Shaping is the process that resolves all of this. It reads tables inside the font that describe substitutions and positioning rules.
10. Kerning is part of the positioning step. The font contains a table saying that when glyph A is followed by glyph V, shift V left by some units.
11. After shaping the editor has a list of glyph IDs, each with an x and y offset in fractional pixels. That is the layout result.
12. The layout is cached. Editing one line does not re-shape the whole document, which is why a fast editor stays fast on a large file.

21.9.5 TECHNICAL - the engineer's version

1. Font format history: TrueType was created by Apple around 1991 and licensed to Microsoft. PostScript Type 1 came from Adobe in 1984. OpenType, announced by Microsoft and Adobe in 1996, unified them and added the `GSUB` and `GPOS` tables that make real shaping possible. OpenType became ISO/IEC 14496-22 as part of MPEG-4 in 2003.
2. Key OpenType tables in this path:

Table	Purpose
cmap	code point to glyph ID
glyf / CFF2	the outlines themselves
hmtx	advance widths
GSUB	ligatures, contextual forms
GPOS	kerning, mark attachment

3. HarfBuzz is the dominant open-source shaping engine, used by Chrome, Firefox, Android, GNOME, LibreOffice and Qt. Its name is Persian for "open type". Apple uses CoreText, and Microsoft uses DirectWrite with its own shaping engine, `usp10`'s successor.
4. The `hb-shape` command-line tool shows shaping output directly:

```
hb-shape --font-file=/usr/share/fonts/DejaVuSans.ttf \
         --unicodes=0048,0065,006C,006C,006F,0041
```

5. Text buffer structures, precisely. A gap buffer keeps a movable hole at the cursor, giving amortized O(1) inserts at the cursor and O(n) cursor jumps. A piece table keeps the original text immutable and stores an ordered list of spans into an original buffer and an append-only add buffer, which makes undo nearly free. Visual Studio Code moved to a piece tree, a balanced tree of pieces, in 2018 and documented the change publicly.

6. Damage tracking terminology differs by toolkit. GTK calls it `gtk_widget_queue_draw`, Qt calls it `QWidget::update`, Cocoa calls it `setNeedsDisplay:`, Win32 calls it `InvalidateRect`. All four mean the same thing: record a dirty region and return immediately.
7. Coalescing is the reason this design exists. If you hold a key and the system generates 30 repeats per second, but the display refreshes 60 times per second, drawing per event would be wasteful. Marking damage and drawing once per frame is strictly better.
8. Latency cost of this section, measured on a modern desktop with a warm cache: widget dispatch under 10 microseconds, buffer insert under 1 microsecond, shaping a single line 20 to 200 microseconds depending on script and cache state, damage marking under 1 microsecond.
9. The honest version: those are the good numbers. In a browser-based editor, this same section can take 5 to 50 milliseconds, because the key event goes through JavaScript, a virtual DOM diff, a style recalculation and a layout pass, and can be delayed by garbage collection. That is a real and measurable difference, and it is why native editors still feel sharper.

21.9.6 WORDS - remember these

1. Widget — one interactive part of a window — a node in the toolkit's hierarchy with its own geometry, focus state and event handlers.
2. Character — a unit of meaning in text — a Unicode code point such as U+0041.
3. Glyph — a drawing in a font — an indexed outline within one font file, meaningless outside it.
4. Shaping — turning characters into positioned glyphs — the application of `GSUB` and `GPOS` rules by an engine such as HarfBuzz or CoreText.
5. Kerning — nudging two letters closer or apart — pairwise or class-based positioning adjustment from the font's `GPOS` or legacy `kern` table.
6. Advance — how far to move before the next glyph — the horizontal advance width in font design units, scaled by point size.
7. Damage — the part of the window that is now wrong — the invalidated region recorded for repainting on the next frame.

21.10 Drawing

21.10.1 PLAIN - in simple words

1. The editor now knows it must draw the letter A at a certain spot.
2. Inside the font, A is not a picture of dots. It is a description of its outline: a set of curves and straight lines forming the edge of the shape.
3. Turning that outline into coloured dots is called rasterizing.
4. The rasterizer works out, for every pixel, how much of that pixel falls inside the shape.
5. If a pixel is completely inside, it is fully dark. Completely outside, untouched. Half covered, half dark.
6. That partial darkness is anti-aliasing, and it is why text on a screen looks smooth instead of like a staircase.
7. There is a trick that goes further. Each pixel on most screens is actually three tiny lights side by side: red, green and blue.
8. If you treat those three as three separate samples instead of one, you get three times the horizontal detail. That is sub-pixel rendering.
9. Once the letter is drawn, the editor's window is a finished rectangle of pixels sitting in memory.
10. Then the compositor takes that rectangle, and every other window's rectangle, and stacks them into one single image the size of your screen.

21.10.2 PLAIN - a picture in your head

1. Imagine drawing a shape on graph paper and then colouring in the squares.
2. A square fully inside the shape gets fully coloured. A square fully outside stays white.

3. A square the edge passes through is the hard case. If you only allow black or white, the edge looks jagged.
4. So instead you shade it grey in proportion to how much of the square the shape covers. Seventy percent covered, seventy percent grey.
5. Step back and the eye blends the greys into a smooth edge. That is anti-aliasing in one sentence.
6. Now imagine each square on your graph paper is actually three thin vertical strips, coloured red, green and blue, that only look white together.
7. If the shape's edge covers only the left strip, you darken only red. The eye still reads it as a slightly-left-shifted edge, at a third of a square's precision.

Where this comparison breaks: darkening only the red strip really does tint the edge, and on a large low-density screen you can see coloured fringes on text. This is a genuine trade of colour accuracy for apparent sharpness, and it only works if the software knows the exact stripe order of your panel.

21.10.3 PLAIN – a worked example

1. Rasterize the letter A at 16 pixels per em from a font with 1000 units per em.
2. Scale factor is 16/1000 = 0.016 pixels per font unit.
3. The outline is a list of points. In TrueType the curves are quadratic Beziers, each with one control point. In CFF and PostScript fonts they are cubic, with two.
4. The rasterizer flattens each curve into short straight segments, fine enough that the error is under a fraction of a pixel.
5. It then sweeps down the shape one scanline at a time, finding where the outline crosses that line, and filling between crossings.
6. For the anti-aliased version it computes area coverage per pixel instead of a yes/no fill. The result is a small grey bitmap:

```
. . 3 9 9 3 . .        digits are coverage
. . 8 4 4 8 . .        0 = empty, 9 = full
. 3 9 . . 9 3 .
. 8 9 9 9 9 8 .        the crossbar of the A
3 9 . . . . 9 3
9 4 . . . . 4 9
```

7. Sub-pixel rendering triples the horizontal sampling first, producing three coverage values per pixel, then applies a filter across neighbouring sub-pixels to reduce colour fringing.
8. Those coverages are blended with the background. Doing that blend on the raw stored values is wrong, because screen values are gamma encoded. Correct blending happens in linear light, and getting it wrong makes text look too thin on dark backgrounds and too fat on light ones.
9. The glyph bitmap is cached in a glyph cache keyed by font, size, hinting mode and sub-pixel position. Typing a second A costs a cache lookup, not a rasterization.

21.10.4 PLAIN – what is really happening inside

1. There are two places the drawing can happen and modern applications use both.
2. **On the CPU.** The application writes pixel values into a buffer in ordinary memory using a library such as FreeType plus Cairo, Skia's software backend, or CoreGraphics.
3. This is simple, predictable and works everywhere. It costs CPU time proportional to the number of pixels touched.
4. **On the GPU.** The application does not write pixels. It writes commands into a command buffer: bind this texture, draw these triangles, use this shader.
5. Glyphs are pre-rasterized into a big texture called a glyph atlas. Drawing a letter becomes drawing two triangles with the right texture coordinates.

6. The command buffer is submitted to a queue, and the GPU executes it independently. The CPU does not wait.
7. Either way the result is a buffer of pixels representing this one window.
8. Now the compositor. It has one buffer per visible window, plus a cursor, plus wallpaper, plus panels.
9. It composites: for each output pixel, work out which window is on top there, apply any transparency, scaling or rotation, and write the final colour.
10. This is almost always done on the GPU, because it is exactly the kind of massively parallel per-pixel work a GPU is built for.
11. There is a shortcut worth knowing. If a window is full-screen, opaque, and already in the right format, the compositor can skip compositing entirely and tell the display hardware to scan out that window's buffer directly.
12. That is called direct scanout, and it removes one full copy and often one full frame of latency. It is why full-screen games are faster than windowed ones.

21.10.5 TECHNICAL - the engineer's version

1. Hinting: TrueType fonts carry a bytecode program per glyph that moves points onto the pixel grid so that stems have consistent widths at small sizes. Apple's patents on the TrueType bytecode interpreter expired in 2010, which is why FreeType enabled the interpreter by default from version 2.4 onward.
2. FreeType's own autohinter is used when a font has no useful hints. Full hinting produces crisper but distorted shapes; the current preference on high-density displays is slight or no hinting, keeping shapes faithful.
3. Sub-pixel rendering: Microsoft announced ClearType at COMDEX in November 1998. The work was led by Bill Hill with Greg Hitchcock, and Microsoft published "Displaced Filtering for Patterned Displays" in May 2000. The idea itself is older; Apple's 1976 Apple II used a related trick on colour artefacts, and IBM described sub-pixel addressing earlier still.
4. Sub-pixel rendering has three real limitations. It requires knowing the stripe order, which is RGB on most desktop LCDs and BGR on some. It breaks on a rotated display. And it is meaningless on OLED panels with non-striped sub-pixel layouts such as PenTile.
5. Apple removed sub-pixel anti-aliasing from macOS in Mojave (10.14, 2018), on the reasoning that Retina-class pixel densities make it unnecessary. This remains contested by people using non-Retina external monitors.
6. Signed distance field text, described by Chris Green of Valve at SIGGRAPH 2007 in "Improved Alpha-Tested Magnification for Vector Textures and Special Effects", stores distance-to-edge instead of coverage, allowing one atlas entry to scale smoothly. It is widely used in games and in map rendering, and it is slightly less accurate for small body text.
7. Buffer sharing on Linux is by `dma-buf` file descriptors passed over the Wayland socket, so no pixel data is copied between processes. macOS uses `IOSurface`. Windows uses DXGI shared surfaces.
8. Compositing cost, order of magnitude at 2560x1440 on a mid-range 2023 GPU:

Operation	Typical cost
Rasterize one glyph, CPU	5 to 50 us
Glyph cache hit	under 1 us
Composite 6 windows, GPU	0.3 to 1.5 ms
Direct scanout instead	~0 ms

9. Tools: `fc-list` and `fc-match` for font selection; `ftview` and `ftbench` from FreeType; `GALLIUM_HUD` or `mangohud` for frame timing; `weston-debug` and `drm_info` for plane and scanout state; `about:gpu` in Chrome for its compositing decisions.

21.10.6 WORDS - remember these

1. Outline — the shape of a letter as curves — quadratic or cubic Bezier contours stored in `glyf` or `CFF2`.
2. Rasterize — turn a shape into pixels — scan conversion with area coverage computation.
3. Anti-aliasing — grey edges instead of jagged ones — using fractional coverage as an alpha value.
4. Sub-pixel rendering — using R, G and B as separate samples — horizontal tripling with a filter, as in ClearType.
5. Hinting — nudging outlines onto the pixel grid — bytecode or automatic grid fitting for stem consistency at small sizes.
6. Glyph atlas — a texture full of pre-drawn letters — a packed cache of rasterized glyphs for GPU text drawing.
7. Direct scanout — showing a window's buffer with no compositing — assigning a client buffer straight to a display plane.

21.11 Out to the screen

21.11.1 PLAIN - in simple words

1. The compositor has produced one image the size of your whole screen. That image is called the framebuffer.
2. A piece of hardware called the display controller reads that image out of memory, continuously, forever, whether or not anything changed.
3. It reads it in a fixed order: the top-left pixel first, then along the top row, then the next row, and so on to the bottom-right.
4. Then it starts again at the top. On a 60 Hz screen it does this 60 times a second. On a 240 Hz screen, 240 times.
5. Between finishing the bottom and starting the top again there is a short gap. That gap is called the vertical blanking interval.
6. If you swap in a new image during the gap, the screen shows one whole image at a time. If you swap in the middle, the top of the screen shows the old image and the bottom shows the new one, with a visible line between them. That is tearing.
7. So the system usually waits for the gap. That waiting is called vsync, and it is one of the largest single delays in this whole chapter.
8. The pixel values travel out of the cable, one after another, as fast electrical signals.
9. At the other end, the monitor has its own computer. It may need to resize the image, adjust the colours, and then drive the actual panel.
10. Finally, tiny amounts of liquid crystal twist, or tiny organic diodes emit, and light leaves the screen and reaches your eye.

21.11.2 PLAIN - a picture in your head

1. Picture a very fast painter repainting a fence, left to right, top plank to bottom plank, over and over.
2. The painter never stops. Even if nothing has changed, the whole fence is repainted 60 times a second.
3. You want to change the design. If you hand over the new design while the painter is halfway down the fence, the top half keeps the old design and the bottom half gets the new one. Half and half.
4. So you wait until the painter reaches the bottom and is walking back to the top. That walk back is the blanking interval. Hand over the design then and the whole fence changes together.
5. The cost of politeness is the wait. If you finish your design just after the painter starts a pass, you wait almost a whole pass.

Where this comparison breaks: the painter's walk back is short, but the pass itself takes a full frame, and a change to the top of the screen appears before a change to the bottom of the same frame. That means

the physical position of your text on the screen genuinely affects when you see it, by up to a whole frame time.

21.11.3 PLAIN - a worked example

1. Take 1920x1080 at 60 Hz over HDMI with standard CEA-861 timings.
2. The controller does not scan 1920x1080 pixels. It scans 2200 x 1125, including the blanking areas where no pixel is shown.
3. Total pixel clock: 2200 * 1125 * 60 = 148,500,000 pixels per second, which is the well-known 148.5 MHz pixel clock for 1080p60.
4. One full frame takes 1/60 second = 16.67 ms.
5. One scanline takes 16.67 ms / 1125 = 14.8 microseconds.
6. Your text is on line 400 of the screen. Once the frame starts scanning out, that line is transmitted at 400 * 14.8 us = 5.9 ms into the frame.
7. Now compare refresh rates:

Refresh	Frame time	Average vsync wait
60 Hz	16.67 ms	8.33 ms
120 Hz	8.33 ms	4.17 ms
144 Hz	6.94 ms	3.47 ms
240 Hz	4.17 ms	2.08 ms
360 Hz	2.78 ms	1.39 ms

8. The average wait is half the frame time because your frame becomes ready at a random point in the cycle.
9. That is only the wait. On top of it comes the scanout position, the panel's own processing, and the pixel's response time.
10. And if the system is triple buffered, your finished frame may sit behind one or two other finished frames, adding one or two whole frame times on top. At 60 Hz that is another 16.67 or 33.3 ms.

21.11.4 PLAIN - what is really happening inside

1. The display controller, called a CRTC on Linux, holds the memory address of the framebuffer, the timing numbers, and the format.
2. Changing which buffer is displayed is called a page flip. The driver programs the new address and the hardware latches it at the next blanking interval.
3. A page flip is atomic and cheap. Nothing is copied. Only a pointer changes.
4. The pixel data is serialized onto the cable. HDMI and DVI use TMDS, transition-minimized differential signalling, on three data lanes plus a clock. DisplayPort uses packetized micro-packets on one to four lanes with an embedded clock.
5. At the monitor, the received stream first meets a scaler if the incoming resolution does not match the panel's native resolution. Scaling costs real time, often several milliseconds, and is a common hidden source of lag.
6. Then the timing controller, the TCON, converts the stream into the signals the panel needs, driving gate lines row by row and source lines column by column.
7. On an LCD, the backlight is always on. Each sub-pixel is a tiny cell of liquid crystal between polarizers, with a colour filter. Applying voltage twists the crystal, changing how much light passes.
8. The crystal takes time to twist. That is response time, and it is why a fast-moving object smears on some panels.
9. On an OLED, each sub-pixel emits its own light. There is no backlight and no crystal to twist, so response is under 0.1 ms.

10. Light then leaves the screen. At 60 cm, the photons take about 2 nanoseconds to reach your eye. That is two millionths of a millisecond, and it is the only stage in this chapter you can safely ignore.

21.11.5 TECHNICAL - the engineer's version

1. Standards and dates: HDMI 1.0 in December 2002; DisplayPort 1.0 from VESA in 2006; HDMI 2.1 in November 2017 at 48 Gbit/s; DisplayPort 2.0 in 2019 and 2.1 in 2022, with UHBR20 giving 80 Gbit/s raw.
2. Variable refresh rate: NVIDIA G-SYNC shipped in 2013 with a hardware module in the monitor. VESA added Adaptive-Sync to DisplayPort 1.2a in 2014, and AMD's FreeSync built on it from 2015. HDMI added VRR in HDMI 2.1 in 2017.
3. VRR changes the latency picture. Instead of the frame waiting for the display, the display waits for the frame, within a supported range such as 48 to 240 Hz. The average vsync wait largely disappears while the frame rate is inside that window.
4. Linux exposes all of this through DRM/KMS. The atomic modesetting API, merged in Linux 4.2 (2015), lets a compositor commit a whole new display state, including plane assignments, in one transaction.
5. Panel response times, honestly stated:

Panel type	Gray-to-gray	Notes
OLED	under 0.1 ms	near instant
TN LCD	1 to 3 ms	poor colour
Fast IPS	3 to 6 ms	common now
VA LCD	4 to 15 ms	slow dark shifts

6. Marketing "1 ms" figures are usually the fastest single transition with overdrive at its most aggressive, not an average, and overdrive causes overshoot artefacts. Dan Luu measured that a monitor advertising 1 ms switching took roughly 10 ms to fully settle a character.
7. Display processing latency is a separate number from response time and is dominated by the scaler and any image processing. Televisions in normal picture modes commonly add 20 to 100 ms; game mode disables most of it and typically brings it under 10 ms.
8. Sample and hold: on an LCD or OLED the image is held static for the whole frame, so your eye, tracking a moving object, smears it across the retina. This persistence blur is independent of response time and is why backlight strobing exists.
9. Tools: `modetest` and `drm_info` show CRTC timings and plane usage; `xrandr --verbose` prints modelines including blanking; `sudo powermetrics` on macOS and `PresentMon` on Windows show present and display timings; the Blur Busters test patterns measure real panel behaviour.

21.11.6 WORDS - remember these

1. Framebuffer — the finished image in memory — a region of memory holding one full screen of pixels in a defined format and stride.
2. Scanout — the controller reading the image to the cable — continuous sequential transmission of the framebuffer at the pixel clock.
3. Vblank — the gap between frames — the vertical blanking interval, when no visible pixel is being transmitted.
4. Page flip — swapping which buffer is shown — an atomic change of the scanout address latched at vblank.
5. Tearing — two frames visible at once — the artefact of changing the scanout source mid-frame.
6. TCON — the panel's own controller — the timing controller driving gate and source lines from the received video stream.
7. Response time — how long a pixel takes to change — the gray-to-gray transition time of the pixel element, distinct from display processing lag.

21.12 The complete time budget

21.12.1 PLAIN – in simple words

1. Here is the whole chain in one picture. Read it from top to bottom. This is every stage in this chapter, in order.

```
              FINGER PRESSES THE KEY
                       |
                       |  2.0 mm of travel, 45 cN
                       v
 [1]  SWITCH CONTACT CLOSES     bounce for up to 5 ms
                       |
                       v
 [2]  KEY MATRIX     8 rows x 18 cols, diode per key
                       |  scanned 100 to 1000 times a second
                       v
 [3]  KEYBOARD MICROCONTROLLER
      debounce -> keymap lookup -> scan code 0x04
                       |  builds the 8-byte HID report
                       v
 [4]  USB ENDPOINT 1 IN (interrupt)
                       |  host polls every 1 ms or 125 us
                       v
 [5]  xHCI HOST CONTROLLER
      DMA into RAM -> event TRB -> MSI interrupt
                       |
                       v
 [6]  CPU TAKES INTERRUPT -> kernel ISR
      usbcore -> HID class driver -> input core
                       |  EV_KEY code 30 (KEY_A) value 1
                       v
 [7]  DISPLAY SERVER / COMPOSITOR
      focus lookup -> route to one client only
                       |
                       v
 [8]  TOOLKIT + KEYMAP
      key code + modifiers -> the character 'A'
                       |
                       v
 [9]  APPLICATION
      insert into text buffer -> move cursor
                       |
                       v
[10]  TEXT LAYOUT
      itemize -> shape (GSUB/GPOS) -> positions
                       |
                       v
[11]  RASTERIZE
      outline -> coverage -> anti-aliased pixels
                       |
                       v
[12]  COMPOSITE
      all window buffers -> one framebuffer
                       |
                       v
[13]  WAIT FOR VBLANK -> PAGE FLIP
                       |
                       v
[14]  DISPLAY CONTROLLER SCANS OUT
```

```
                      |  148.5 MHz pixel clock at 1080p60
                      v
[15]  CABLE -> SCALER -> TCON -> ROW/COL DRIVERS
                      |
                      v
[16]  SUB-PIXELS CHANGE STATE
                      |  ~2 ns of flight over 60 cm
                      v
                  PHOTONS -> EYE
```

2. Now the numbers. Two columns: a carefully built fast setup, and an ordinary badly configured one.
3. The clock starts at the moment the switch closes, not when your finger starts moving.

21.12.2 PLAIN - a picture in your head

1. Think of a relay race with sixteen runners, each handing a baton to the next.
2. The total time is not decided by how fast the runners are. Most of them are very fast indeed.
3. It is decided by the four or five runners who stand still and wait for a whistle before setting off.
4. In our chain the whistles are: the matrix scan, the USB poll, the frame callback, and vsync.
5. Making the fast runners faster changes almost nothing. Making the whistles more frequent changes everything.

Where this comparison breaks: some of our runners genuinely are slow, not just waiting. A browser-based editor doing a layout pass, or a television doing image processing, is real work taking real time, and no amount of scheduling fixes it.

21.12.3 PLAIN - a worked example

Stage	Good ms	Bad ms
1 Contact settle	1.0	5.0
2 Debounce hold-off	0.0	5.0
3 Matrix scan wait	0.5	5.0
4 Firmware build report	0.05	0.5
5 USB poll wait	0.5	5.0
6 Transfer, DMA, IRQ	0.05	0.2
7 Kernel HID and evdev	0.05	1.0
8 Compositor routing	0.2	2.0
9 App wake and handle	0.3	15.0
10 Layout and shaping	0.1	5.0
11 Rasterize and draw	0.5	5.0
12 Compositing	0.5	8.0
13 Extra queued frames	0.0	33.3
14 Vsync wait	2.1	8.3
15 Scanout to that row	1.0	6.0
16 Panel scaler and TCON	1.0	20.0
17 Pixel response	1.0	10.0
18 Photons to eye	0.000002	0.000002
TOTAL	8.85	134.3

1. The good column assumes: eager debounce, 1000 Hz matrix scan, 1 ms USB polling, a lightly loaded machine, a native application, direct scanout, a 240 Hz monitor with no scaling, and a fast panel.

2. The bad column assumes: 5 ms symmetric debounce, 100 Hz scan, a low-speed USB keyboard polled every 10 ms, a busy machine, a browser-based editor, triple buffering, a 120 Hz refresh, and a television doing image processing.
3. The ratio between them is about fifteen to one. Nothing in that table is exotic. Both configurations are extremely common.
4. Notice where the milliseconds actually are in the bad column. Queued frames at 33.3 ms, panel processing at 20.0 ms, and the application itself at 15.0 ms. Those three are more than half the total.
5. Notice what is almost free in both columns. The USB transfer, the DMA, the interrupt and the kernel. All the parts people worry about most are the parts that cost least.

21.12.4 PLAIN – what is really happening inside

1. It is important to compare that table with real measured numbers, because a sum of estimates is not evidence.
2. Dan Luu, in 2017, measured keyboards alone with a logic analyser on a cut USB cable, from the key starting to move to the USB packet leaving.

Keyboard	Latency
Apple Magic (USB)	15 ms
HHKB Lite 2	20 ms
Unicomp Model M	30 ms
Razer Ornata Chroma	35 ms
Kinesis Advantage	50 ms
Logitech K360	60 ms

3. Those are much larger than the first six rows of our table, which sum to about 2 ms in the good case. Why?
4. Because his clock starts when the key starts moving, not when the contact closes. The finger's own travel to the actuation point is several milliseconds and it is genuinely part of the experience.
5. And because real keyboards use conservative debounce and modest scan rates. A 35 ms gaming keyboard is not a measurement error. It is firmware choosing safety over speed.
6. Dan Luu also measured whole systems, key press to the character finishing on screen, with a high-speed camera.

System	Year	End to end
Apple IIe	1983	30 ms
Custom PC, 165 Hz	2014	50 ms
SGI Indy	1993	60 ms
MacBook Pro	2014	100 ms
PowerSpec G405, Windows	2017	200 ms
Symbolics 3620	1986	300 ms

7. A 1983 Apple IIe beats a 2017 desktop by a factor of nearly seven. That is the single most useful fact in this chapter.
8. For gaming setups, NVIDIA's Reflex tooling reports click-to-photon system latency. TFTCentral, testing an Asus ROG Swift PG259QNR in 2021, measured about 13 ms of PC-plus-display latency at 360 Hz with Reflex on, plus 0.8 ms of mouse latency, for 13.8 ms total, versus about 26 ms at 60 Hz with Reflex off.
9. So our "good" column of 8.85 ms is optimistic but the right order of magnitude, and our "bad" column of 134 ms sits comfortably inside the range of real measured systems.

21.12.5 TECHNICAL - the engineer's version

1. Sources of the buffering penalty, which is the largest single controllable item:
2. Triple buffering with a queue depth of 3 lets the GPU render ahead by two frames. Throughput improves, latency worsens by up to two frame times.
3. Dan Luu computed this from his own data: going from 24 Hz to 165 Hz gave a 90 ms improvement, whereas pure vsync mathematics predicts only about 18 ms. The residual implies roughly 2.5 frames of buffering, independent of refresh rate, on that Windows machine running PowerShell.
4. Mitigations that actually work, with their real mechanism:

Mitigation	Mechanism
Reduce queue depth to 1	fewer frames in flight
VRR (FreeSync, G-SYNC)	display waits for frame
Direct scanout / fullscreen	skips a composite pass
Monitor game mode	disables scaler processing
Raise app scheduling prio	shortens wake-up delay

5. NVIDIA Reflex, announced 1 September 2020, works by having the driver and the game agree to delay the start of CPU work so that the CPU finishes just in time for the GPU, instead of queueing frames ahead. It is a back-pressure mechanism, not a rendering speed-up.
6. Measurement methods and what each one actually includes:

Method	Starts at	Ends at
High-speed camera	key movement	pixel change
Logic analyser on USB	key movement	USB packet
LDAT / photodiode	click signal	light change
In-engine counters	input read	present call

7. Those four methods do not measure the same thing and their results are not comparable. Most arguments about input lag on the internet are two people quoting different methods at each other.
8. Tools: `PresentMon` on Windows for present-to-display timing; `LatencyFlex` and Reflex counters in supported games; `libinput measure` for device-side timing; a 1000 fps phone camera, which most phones now have, for a genuine end-to-end check that requires no trust in software.

21.12.6 WORDS - remember these

1. End-to-end latency — the total wait from press to light — motion-to-photon or click-to-photon latency, measured externally.
2. Queue depth — how many finished frames wait in line — the maximum frames in flight, typically 1 to 3, trading latency against throughput.
3. Back pressure — making the producer wait for the consumer — the technique behind Reflex and anti-lag features.
4. Click to photon — the gamer's version of the same measurement — latency from a switch event to a measured luminance change at the panel.
5. Jitter — the variation in the wait, not the average — the spread of latency across repeated events, often more noticeable than the mean.

21.13 The same trace for a mouse and a touchscreen

21.13.1 PLAIN - in simple words

1. A mouse click follows almost the same road. Switch, debounce, matrix or direct pin, microcontroller, USB HID, kernel, compositor, application.
2. Two things differ. First, the mouse also carries movement, and movement comes from a camera, not a switch.
3. Under an optical mouse there is a tiny camera and a light. It photographs the surface thousands of times a second.
4. It compares each photo with the one before, works out how far the pattern shifted, and reports that shift.
5. So a mouse never reports where it is. It only reports how far it moved since last time. The position on screen is the computer adding those up.
6. Second, a click has a place. The system must work out what is under the pointer before it can deliver the click.
7. A touchscreen is different again. It reports an actual position on the glass, not a movement.
8. Under the glass is a grid of transparent wires. Your finger changes the electrical capacity where it touches, and the controller finds the centre of that change.
9. The controller then works out whether you tapped, held, dragged, or made a gesture with two fingers, and reports that.

21.13.2 PLAIN - a picture in your head

1. Imagine walking across a field in fog with a torch and a camera pointed at the ground.
2. Every fraction of a second you photograph the grass. You cannot see where you are in the field, but you can see that this photo is shifted two centimetres left of the last one.
3. Add up all the shifts and you have a path. That is an optical mouse, and it is also why a mouse loses tracking on glass: no grass, nothing to compare.
4. Now imagine instead a giant chessboard where every square reports when somebody stands on it. You do not need to add anything up. You are told where you are.
5. That is a touchscreen. Absolute, not relative.

Where this comparison breaks: the chessboard squares are coarse and a touchscreen is not. The controller looks at how strongly several neighbouring squares respond and computes a weighted centre, giving a position much finer than the spacing of the wires.

21.13.3 PLAIN - a worked example

1. Real optical mouse sensor numbers, from the Agilent ADNS-3080 datasheet of 2005, a design still widely copied:

Property	Value
Image sensor	30 x 30 pixels
Maximum frame rate	6469 frames/s
Resolution	up to 1600 counts/inch
Maximum speed	40 inches/second
Maximum acceleration	15 g

2. At 6469 frames per second, one frame takes 155 microseconds. That is the sensor's own contribution to latency and it is tiny.
3. DPI, properly called CPI or counts per inch, is how many counts the sensor reports per inch of movement. At 1600 CPI, moving one inch produces 1600 counts.
4. Higher CPI is not "more accurate". It is finer granularity. If the sensor's noise is larger than one count, extra CPI just reports noise more precisely.

5. The USB HID boot mouse report is three bytes:

```
byte 0 : buttons, bit 0 left, bit 1 right, bit 2 middle
byte 1 : X movement, signed 8-bit, negative is left
byte 2 : Y movement, signed 8-bit, negative is up
```

6. Left click with no movement gives `01 00 00`. Release gives `00 00 00`. Moving right 5 and down 3 with no buttons gives `00 05 03`.
7. Signed 8-bit caps movement at 127 counts per report. At 1000 Hz polling and 1600 CPI, that caps out at about 79 inches per second, which is enough. At 125 Hz polling it would cap at about 10 inches per second, which is not. That is why real mice use the report protocol with 12 or 16-bit deltas.
8. Click versus drag is decided in software, not hardware. A press, then movement beyond a threshold before release, is a drag. A press and release inside the threshold, within a time limit, is a click.
9. Mouse switches bounce too, and a worn Omron D2FC-F-7N switch is the usual cause of a mouse that double-clicks by itself. It is the debounce window failing to hide a degraded contact.

21.13.4 PLAIN – what is really happening inside

1. Optical mouse tracking, step by step. An LED or a laser illuminates the surface at a shallow angle so that microscopic texture casts shadows.
2. The sensor captures a small grey image. The processor cross-correlates it with the stored previous image, testing candidate shifts, and picks the shift with the best match.
3. Sub-pixel interpolation of the correlation peak gives finer resolution than the pixel spacing. The result is a pair of integers, dx and dy.
4. Frame rate is usually variable. The sensor speeds up when moving fast and slows down when still, to save power.
5. Touchscreen, step by step. Projected capacitive screens have two layers of transparent conductor, usually indium tin oxide, in rows and columns.
6. The controller drives one row with a signal and measures the coupling into each column. Your finger steals some of that coupling.
7. This produces a small grid of numbers, a capacitance image, typically at 60 to 240 scans per second.
8. The controller finds connected regions above a threshold, computes each region's weighted centre, and assigns each a tracking identifier so that a finger keeps its identity across frames.
9. Then it decides what the touches mean: a tap, a long press, a swipe, a pinch, a rotation. Some controllers do this themselves; on phones, most of it is done by the operating system.
10. Palm rejection, edge rejection and moisture handling all live here too, and they add processing time.
11. Coordinates are reported as absolute values in a device coordinate space, then scaled into screen coordinates. This is the deepest difference from a mouse: no accumulation, no acceleration curve, no pointer to lose.

21.13.5 TECHNICAL – the engineer's version

1. History: Douglas Engelbart and Bill English built the first mouse at SRI in the mid-1960s; the patent, US 3,541,541, "X-Y Position Indicator for a Display System", was granted in 1970, and the public demonstration on 9 December 1968 is remembered as the Mother of All Demos.
2. The first optical mice needed a special printed mat. Surface-independent optical tracking arrived with Agilent's sensors and the Microsoft IntelliMouse with IntelliEye in 1999.
3. Capacitive touch sensing was described by E. A. Johnson at the Royal Radar Establishment in 1965. Multi-touch projected capacitive reached the mass market with the iPhone in 2007.
4. HID reporting: mice use Generic Desktop usages X (0x30) and Y (0x31) with `Relative` set. Touchscreens use the Digitizer usage page (0x0D) with `Absolute` coordinates, contact identifiers and a contact count, defined by the Windows Precision Touchscreen and Precision Touchpad specifications.

5. On Linux, touch arrives as multitouch protocol B events: `ABS_MT_SLOT`, `ABS_MT_TRACKING_ID`, `ABS_MT_POSITION_X`, `ABS_MT_POSITION_Y`, grouped by `SYN_REPORT`. Protocol A, the older form without slots, still exists on some hardware.
6. Polling rates and scan rates in current products:

Device	Rate
Standard USB mouse	125 Hz
Gaming mouse	1000 Hz
HyperPolling class mouse	8000 Hz
Phone touch digitizer	120 to 240 Hz
Apple Pencil scan	240 Hz

7. Dan Luu's 2017 scroll-latency measurements put the 2017 iPad Pro 10.5 inch at 70 ms with a finger and 30 ms with the Pencil, against 110 to 150 ms for most Android phones of the same period. The Pencil's advantage comes from the higher scan rate and a shorter dedicated path.
8. Pointer acceleration is a software transfer function mapping counts per report to pixels, and it is a genuine source of disagreement. Games disable it and read raw counts; desktops keep it. `libinput` implements it in userspace, which is why acceleration behaviour became consistent across Linux desktops after 2015.
9. Tools: `sudo libinput debug-events` shows both relative and absolute events; `evtest` shows raw multitouch slots; `xinput list-props` shows acceleration settings; the `mousetester` utility plots real polling intervals and reveals a mouse whose 1000 Hz claim is not honoured.

21.13.6 WORDS - remember these

1. Relative input — reporting movement, not position — delta reporting, as from a mouse or trackball, accumulated by the host.
2. Absolute input — reporting a position directly — coordinate reporting, as from a touchscreen or tablet digitizer.
3. CPI / DPI — counts reported per inch moved — sensor resolution, commonly 400 to 26,000, independent of accuracy.
4. Cross-correlation — matching one photo against the last — the block-matching step that converts two images into a displacement.
5. Projected capacitance — sensing a finger through glass — mutual capacitance measurement across a transparent row/column electrode grid.
6. Tracking identifier — the number that says "this is the same finger" — the slot identity that keeps contacts distinguishable between scans.

21.14 Why this matters

21.14.1 PLAIN - in simple words

1. It is tempting to say that a few milliseconds cannot matter, because a millisecond is a thousandth of a second.
2. That turns out to be wrong, and we can show it with measurements rather than opinions.
3. In a shooting game, the enemy you are aiming at is where they were, plus however long your whole chain took.
4. If your chain is 100 ms and theirs is 20 ms, they see the truth 80 ms earlier than you do. At normal movement speeds that is a body width.
5. In music, the same problem shows up as feel. You hit a drum pad and hear the sound late. Your hands compensate by playing early, and the timing drifts.

6. In plain typing, a slow chain does not stop you working, but it does make the machine feel unresponsive, and there is evidence that it makes people less accurate.
7. The honest part is this: for most people, most of the delay is not in the keyboard. It is in the screen and in the software.
8. Buying a faster keyboard when your monitor adds 40 ms of image processing is spending money in the wrong place.

21.14.2 PLAIN - a picture in your head

1. Think of a conversation over a satellite phone with half a second of delay.
2. Nothing is broken. Every word arrives perfectly. Yet the conversation is exhausting, because you cannot interrupt, agree or laugh at the right moment.
3. Now think of the same delay applied to your own hand. You move it and see it move half a second later. You would immediately feel that something was badly wrong.
4. Your body has an extremely tight expectation for the delay between an action and its result, because that loop is how you learned to move at all.
5. Input lag is that same violation, just smaller. Small enough to be unnamed, large enough to be felt.

Where this comparison breaks: satellite delay is constant and you adapt to it. Input lag on a computer varies from press to press, and variation is harder to adapt to than a constant offset. A steady 40 ms often feels better than a wandering 20 to 60 ms.

21.14.3 PLAIN - a worked example

1. NVIDIA Research ran an aiming study and reported it in 2020 alongside Reflex.
2. They compared PCs at 12 ms and at 20 ms of system latency, an 8 ms difference.
3. The average difference in the time taken to acquire and shoot a target was 182 ms. That is roughly 22 times the latency difference.
4. On a 128-tick server, that is about 23 ticks earlier that your shots land.
5. NVIDIA also stated that most gamers play on systems with 50 to 100 ms of system latency, which puts the 12 ms machine in a different world.
6. Now music. MIDI over the original 5-pin cable runs at 31,250 bits per second. A three-byte note-on message with start and stop bits is 30 bits, so about 960 microseconds.
7. A digital audio interface with a 128-sample buffer at 48 kHz adds 128/48000 = 2.67 ms per buffer, and there are usually at least two, in and out.
8. Real round-trip latency on a well-configured system is 5 to 12 ms. Above roughly 10 ms most players start to feel the instrument is not under their fingers.
9. A useful check: sound travels about 343 metres per second, so 10 ms is 3.4 metres. A 10 ms delay is the same as standing three and a half metres back from your amplifier, which musicians do all the time without complaint. That is why 10 ms is roughly the boundary rather than 1 ms.

21.14.4 PLAIN - what is really happening inside

1. Where do the milliseconds actually go, honestly, for a normal person on a normal machine?
2. The keyboard contributes 15 to 60 ms depending on the model, mostly in debounce and scan rate, per Dan Luu's 2017 measurements.
3. The USB layer contributes 0.5 to 5 ms, and almost never more.
4. The kernel contributes well under 1 ms unless the machine is loaded.
5. The application contributes anything from 0.3 ms for a native editor to tens of milliseconds for a browser-based one.
6. The compositor and buffering contribute one to three frame times, which at 60 Hz is 16.7 to 50 ms.
7. The display contributes the vsync wait plus its own processing plus pixel response, commonly 10 to 40 ms, and far more on a television not in game mode.

8. So the honest ranking, for a typical desktop, is: display and buffering first, application second, keyboard third, and everything else a rounding error.
9. That is why the 1983 Apple IIe beat a 2017 desktop. It had no compositor, no buffering, no process boundaries, and a display that was written to directly.
10. It is also why the fix is usually free. Turning on game mode, disabling a compositor's extra buffering, using a native application, and raising the refresh rate cost nothing and beat any hardware purchase.

21.14.5 TECHNICAL - the engineer's version

1. Human perception limits, from published work rather than folklore:
2. Albert Ng, Paul Dietz and colleagues, in "Designing for Low-Latency Direct-Touch Input" at UIST 2012, built a touch system with about 1 ms of latency and found that people could discriminate latencies down to approximately 2 ms in direct-touch dragging tasks.
3. That result directly contradicts the widely repeated Nielsen Norman claim that anything under 100 ms feels instantaneous. The 100 ms figure describes perceived system responsiveness for discrete actions, not the sensorimotor loop of continuous input.
4. Where experts disagree: whether polling rates above 1000 Hz produce a human-detectable improvement. Manufacturers say yes and sell 8000 Hz products. Independent blind testing has not established a consistent benefit, and the arithmetic caps the possible gain at about 0.44 ms of average wait. Treat "8K polling" claims as **marketing** until a blind study says otherwise.
5. Established fact: total system latency measurably affects task performance in aiming, per NVIDIA's published study and per earlier work in human factors going back to Sheridan and Ferrell's teleoperation studies in 1963.
6. Active research: perceptual thresholds for latency in VR and AR, where motion-to-photon budgets under 20 ms are considered necessary to avoid sickness, and where John Carmack's 2013 essay "Latency Mitigation Strategies" set much of the agenda.
7. Practical budget for a competitive setup, achievable in 2026 with ordinary parts:

Component	Realistic contribution
8 kHz analogue keyboard	1 to 3 ms
Kernel and compositor	under 1 ms
Game with Reflex-style cap	5 to 10 ms
240 Hz OLED, game mode	3 to 6 ms
Total	roughly 10 to 20 ms

8. Tools that let you argue from data: a 1000 fps phone camera for end-to-end; `PresentMon` for the present pipeline; Reflex or LDAT counters for click-to-photon; `cyclictest` to check whether your machine's scheduling latency is the real problem.

21.14.6 WORDS - remember these

1. Input lag — the delay between acting and seeing the result — end-to-end system latency across the full input and display pipeline.
2. Motion to photon — the VR name for the same thing — the interval from head or hand movement to the corresponding light emission.
3. Sensorimotor loop — the body's action-and-result cycle — the closed control loop the nervous system uses, sensitive to delays of a few milliseconds.
4. Round-trip audio latency — in through the microphone and out the speaker — the sum of input buffer, processing and output buffer, typically 5 to 12 ms.
5. Tick rate — how often a game server updates the world — the server simulation frequency, commonly 64 or 128 Hz, adding its own delay on top.

21.98 Common wrong ideas

1. Wrong: the keyboard sends the letter A. Right: it sends usage code 0x04 on usage page 0x07, which names a physical position. The letter is decided by the operating system's keymap, several stages later.
2. Wrong: a key press is a single clean electrical event. Right: the contact bounces for up to 5 ms on a Cherry MX switch, and firmware must filter that into one event.
3. Wrong: a 104-key keyboard needs 104 wires to its chip. Right: an 8 by 18 matrix needs 26 pins, and the chip scans the grid repeatedly rather than watching all keys at once.
4. Wrong: "anti-ghosting" and "N-key rollover" mean the same thing. Right: anti-ghosting often means the firmware suppressing ambiguous combinations, whereas true N-key rollover needs a diode per key and a HID report format that can carry more than six keys.
5. Wrong: a USB interrupt transfer means the device interrupts the computer. Right: the device can never speak first. The host polls it on a fixed schedule, and the name is historical.
6. Wrong: the CPU is stopped in the middle of an instruction by the keyboard. Right: the keyboard's data is written into RAM by the host controller using DMA, and the CPU takes the resulting interrupt at an instruction boundary.
7. Wrong: an 8000 Hz polling rate is eight times better than 1000 Hz. Right: it removes at most about 0.44 ms of average wait, which is small next to the frame and buffering costs that dominate the chain.
8. Wrong: one character equals one glyph. Right: ligatures merge two characters into one glyph, Arabic gives one character four contextual shapes, and Devanagari reorders glyphs relative to characters.
9. Wrong: a monitor advertising 1 ms response has 1 ms of latency. Right: response time is only the pixel transition. Display processing, scaling and the vsync wait are separate and usually much larger.
10. Wrong: new computers respond faster than old ones. Right: Dan Luu measured a 1983 Apple IIe at 30 ms end to end and a 2017 Windows desktop at 200 ms. Raw speed went up and latency went up with it, because of buffering and process boundaries.

21.99 Chapter summary in 20 lines

1. A key switch is only a gate for electricity; a Cherry MX Red closes after 2.0 mm of a 4.0 mm travel at 45 cN and bounces for under 5 ms.
2. Switch families differ in how they close: membrane domes, mechanical leaves, scissor domes, optical beams, and Hall-effect magnets that report depth as a number rather than as on or off.
3. Firmware must debounce, and that choice costs between 0 and 8 ms depending on whether the algorithm fires eagerly or waits for stability.
4. Keys are wired in a row and column matrix so a few pins can read many keys, scanned 100 to 1000 times a second, costing half a scan period on average.
5. Without a diode per key, three pressed keys can create a false fourth, which is ghosting; diodes give true N-key rollover.
6. The keyboard contains a small computer that turns a matrix position into a scan code through a lookup table, and that code is a position, not a letter.
7. USB HID's boot keyboard report is exactly 8 bytes: a modifier bitmap, a reserved byte, and six key code slots; Shift plus A is `02 00 04 00 00 00 00 00`.
8. The host polls the keyboard's interrupt endpoint every 1 ms at full speed or every 125 microseconds at high speed, so the report waits half a period on average.
9. Bluetooth keyboards use HID over GATT with a negotiated connection interval in 1.25 ms units, which is the main reason they feel slower.
10. The xHCI host controller DMAs the report into RAM, posts an event TRB, and raises an MSI interrupt; the CPU takes it at an instruction boundary.
11. The kernel's USB core hands the bytes to the HID class driver, which uses the parsed report descriptor to find fields and emits `EV_KEY` with `KEY_A`, code 30, through evdev.
12. The keymap, not the keyboard, decides the character, using a group axis for layout and a level axis for

modifiers, with dead keys and compose sequences as small state machines on top.

13. Scripts with more characters than keys need an input method editor, a separate program with a dictionary that sends pre-edit text and then a commit.
14. The display server routes the event to the focused window only, and every interactive program is a loop that sleeps, wakes, drains events and sleeps again.
15. The application inserts the character, moves the cursor, re-shapes the affected line into positioned glyphs, and marks a rectangle as damaged without drawing anything yet.
16. Rasterizing turns an outline into per-pixel coverage; anti-aliasing uses that coverage as transparency, and sub-pixel rendering treats the red, green and blue stripes as three horizontal samples.
17. The compositor merges all window buffers into one framebuffer, unless it can hand a single window's buffer straight to the display hardware.
18. The display controller scans that framebuffer out continuously, so a new frame waits for the vertical blanking interval, costing half a frame time on average plus any queued frames.
19. A carefully built chain totals under 10 ms from contact to light; an ordinary badly configured one totals over 130 ms, and real measurements put a 1983 Apple IIe at 30 ms against a 2017 desktop at 200 ms.
20. The milliseconds live in the display, the buffering and the application, not in the wire, so measure before you spend, and remember that NVIDIA measured an 8 ms latency difference producing a 182 ms difference in aiming time.

22

CHAPTER

Graphics and the GPU

22.0 What this chapter gives you

1. You will be able to explain why a graphics chip is built differently from a processor, and what "throughput" buys that "speed" does not.
2. You will be able to draw a line, fill a shape and blend two transparent images by hand, using the same arithmetic the machine uses.
3. You will be able to describe how a three-dimensional shape is stored, and trace one point of it through every coordinate space to a pixel on screen.
4. You will be able to multiply a point by a 4x4 matrix and say exactly why the fourth row and column are there.
5. You will be able to name every stage of the rasterisation pipeline, in order, and say which stages you can write code for.
6. You will be able to read a short shader program and say what runs once per corner and what runs once per pixel.
7. You will be able to explain mipmaps, depth testing, shadow maps and physical material parameters without reaching for magic words.
8. You will be able to say how ray tracing differs from rasterisation, why it was impossible in real time for forty years, and what changed in 2018.
9. You will be able to read a real graphics card specification sheet line by line and predict which number will limit your program.
10. You will be able to explain why the same chip that draws a game also trains a neural network, which is the bridge into Chapters 46 to 51.

22.1 Why a separate chip

22.1.1 PLAIN – in simple words

1. A screen at 1920 by 1080 has 2,073,600 little coloured squares called pixels.
2. At 60 pictures per second, that is about 124 million pixel values to work out every second. At 4K and 120 pictures per second it is nearly a billion.
3. Every one of those pixels is worked out almost independently of its neighbours. Nobody has to wait for anybody else.
4. A main processor is built to do one job very quickly, one step after another, with clever tricks to guess what comes next.
5. That cleverness is expensive in chip area and power, and it is wasted when the same simple sum has to run two million times.
6. So we build a second chip full of very simple workers instead of a few very clever ones.
7. A graphics chip in 2026 has thousands of these simple workers. A desktop processor has eight to twenty-four clever ones.

8. Each simple worker is slower than a clever one. Together they finish the pile of work far sooner.

22.1.2 PLAIN - a picture in your head

1. Imagine a post office that has to stamp one million envelopes today.
2. Option one: hire four brilliant clerks. Each can read handwriting, fix wrong addresses, phone customers and think around problems.
3. Each brilliant clerk stamps four envelopes a second. Four clerks give sixteen a second. The million envelopes take seventeen hours.
4. Option two: hire two thousand school students. Each can only do one thing: press the stamp where the arrow points.
5. Each student stamps one envelope a second. Two thousand students give two thousand a second. The million envelopes take about eight minutes.
6. If a student has to wait for a new box of envelopes, the supervisor simply turns to another student who already has a box. Nobody stands idle.
7. This is the whole idea. The brilliant clerks are a processor. The students are a graphics chip.

Where this comparison breaks: the students are not independent. They are lined up in rows of thirty-two, and every student in a row must press the stamp at the same instant on the same command. If half the row needs a different action, the other half must stand still and wait its turn. A processor core has no such restriction, and this is the single biggest reason some tasks are hopeless on a graphics chip even though they contain a lot of work.

22.1.3 PLAIN - a worked example

1. Suppose we need to add 1 to every number in a list of 16,384 numbers.
2. A processor core running one instruction per cycle at 5 GHz, with no vector tricks, needs 16,384 cycles, about 3.3 microseconds.
3. With vector instructions it handles 16 numbers per instruction, so 1,024 instructions, about 0.2 microseconds.
4. A graphics chip with 16,384 lanes gives every number its own lane. One pass. At 2.4 GHz that pass is under a microsecond including overheads.
5. But the work has to get to the chip first. Sending 16,384 four-byte numbers over the connection to the card takes about 65 kilobytes of transfer.
6. Over a PCIe 5.0 x16 link at roughly 55 gigabytes per second of real throughput, 65 kilobytes takes about 1.2 microseconds each way.
7. So for this tiny job, the copying costs more than the computing. The graphics chip loses.
8. Change the list to 100 million numbers and the picture inverts completely. The chip wins by a factor of tens.
9. This is the rule: a graphics chip pays a fixed toll and then charges very little per item. It only wins when there are a great many items.

22.1.4 PLAIN - what is really happening inside

1. Inside a processor core, most of the silicon is not doing arithmetic.
2. It is guessing which way a branch will go, reordering instructions, keeping large caches full, and undoing work that turned out to be wrong.
3. All of that exists to make one chain of instructions finish sooner. We call this design latency-optimised: it minimizes the waiting time of one task.
4. A graphics chip does almost none of that. It is throughput-optimised: it maximizes how many tasks finish per second, and does not care that any single one is slow.
5. When a processor core asks memory for a value and memory takes 300 cycles to answer, the core tries to find other work in the same instruction stream. Often it cannot, and it stalls.

6. When a graphics chip asks memory for a value, it simply parks that group of threads and runs a different group that already has its data.
7. Because every group's working values sit in a huge on-chip register file, the switch costs nothing. There is no saving and restoring.
8. So a processor hides waiting with caches and guessing. A graphics chip hides waiting with sheer numbers of ready threads.
9. The threads are not free-running. They are grouped, and each group shares one instruction pointer. Thirty-two threads, one instruction, thirty-two different pieces of data.
10. When threads in one group need to take different branches, the hardware runs both paths and switches off the lanes that should not be active. This is called divergence and it wastes time.

22.1.5 TECHNICAL – the engineer's version

1. **SIMD** means Single Instruction, Multiple Data. One instruction operates on a fixed-width vector register. Michael Flynn defined the category in 1966.
2. On x86, AVX-512 gives 512-bit registers, that is 16 lanes of 32-bit float per instruction. The width is visible in the instruction set.
3. **SIMT** means Single Instruction, Multiple Threads. NVIDIA coined the term for the G80 architecture in 2006. It is a programming model over SIMD hardware.
4. In SIMT you write scalar code for one thread. The hardware bundles threads into fixed groups and issues one instruction for the whole group.
5. NVIDIA calls a group a **warp**, and it has been 32 threads on every NVIDIA architecture since G80. That is an implementation detail, not a standard, but it has not changed in twenty years.
6. AMD calls a group a **wavefront**. GCN used 64 threads. RDNA, from 2019, defaults to 32 (wave32) and can also run wave64. Intel Xe issues SIMD8, SIMD16 or SIMD32 depending on the compiled kernel.
7. Divergence is handled by an execution mask. Both sides of a branch execute in sequence with inactive lanes predicated off. Worst case cost is the sum of both paths.
8. Since Volta in 2017, NVIDIA hardware keeps a per-thread program counter, called independent thread scheduling. Threads in a warp can reconverge more flexibly, but they still issue as a warp.
9. Latency hiding is quantified by **occupancy**: resident warps divided by the maximum the hardware supports. Registers and shared memory per thread limit it. NVIDIA ships an occupancy calculator with the CUDA toolkit.
10. Approximate access latencies on recent NVIDIA parts, measured by microbench marks rather than published by the vendor, and therefore approximate:

Level	Typical latency	Note
Register	~1 cycle	Per-thread, huge file
Shared memory	~20-30 cycles	Software managed
L1 cache	~30-40 cycles	Per SM
L2 cache	~200-300 cycles	Chip-wide
GDDR7 VRAM	~400-600 ns	Hundreds of cycles

11. The register file on an NVIDIA H100 streaming multiprocessor is 256 KB. Across 132 of them that is 33 MB of registers, larger than the L2 cache of most processors. That inversion is the signature of a throughput design.
12. Tools to observe this: `nvidia-smi` for utilization and clocks, Nsight Compute for per-kernel occupancy and stall reasons, `rocm-smi` and `rocprof` on AMD, Xcode's Metal debugger on Apple silicon.

22.1.6 WORDS – remember these

1. Latency — how long one job takes — the delay from issue to completion of a single operation or memory request.

2. Throughput — how much work finishes per second — sustained operations or bytes per unit time, independent of any one job's delay.
3. SIMD — one order, many hands — one instruction applied across a fixed-width vector register file.
4. SIMT — scalar code, grouped execution — NVIDIA's model where per-thread programs are executed in lockstep warps on SIMD hardware.
5. Warp / wavefront — a row of workers — the hardware scheduling group: 32 threads on NVIDIA, 32 or 64 on AMD.
6. Divergence — the row splits — threads in one group taking different branches, forcing serialized execution with lane masking.
7. Occupancy — how many rows are waiting to work — resident warps per streaming multiprocessor as a fraction of the architectural maximum.

22.2 Two-dimensional graphics first

22.2.1 PLAIN - in simple words

1. Before any three-dimensional trickery there is a very simple idea: a big block of memory, one entry per pixel.
2. That block is called the **framebuffer**. Write a number into a slot and that pixel changes colour. That is all drawing ever is.
3. For a 1920 by 1080 screen with four bytes per pixel, the block is 1920 times 1080 times 4, which is 8,294,400 bytes, about 7.9 mebibytes.
4. Drawing a line means deciding which slots to write. There is no such thing as a diagonal line in a grid, only a staircase of squares that looks like one.
5. Filling a shape means finding, for each row, where the shape starts and stops, and writing every slot between.
6. Copying a rectangle of pixels from one place to another is called a **blit**. It is the workhorse of all older graphics.
7. A small movable picture, drawn over the background without disturbing it, is called a **sprite**.
8. Making one image show through another is **blending**, and it uses a fourth number per pixel telling how solid that pixel is.
9. Refusing to draw outside a rectangle is **clipping**, and it is what stops one window scribbling over another.

22.2.2 PLAIN - a picture in your head

1. Think of a very large sheet of squared paper, 1920 squares wide and 1080 squares tall.
2. You have coloured pencils and one rule: you may only fill in whole squares. You may never draw between them.
3. To draw a line from one corner to another you hold a ruler over the paper and fill the square nearest the ruler in each column.
4. To draw a triangle you find, in each row, the leftmost and rightmost squares inside it, and fill everything between.
5. To move a picture you rub out the squares it was on and fill in a new set. The paper does not remember anything, so you must repaint the background.
6. To make something look like coloured glass you mix the new colour with what was already in the square instead of replacing it.

Where this comparison breaks: real drawing never rubs anything out. The machine paints the whole sheet again from scratch, sixty or more times a second, into a fresh sheet while you are looking at the previous one. There are always at least two sheets, and they swap. Nothing is ever edited in place while it is visible, because you would see the edit happening.

22.2.3 PLAIN - a worked example

1. Let us draw a line from pixel (0,0) to pixel (6,4) using Bresenham's algorithm, published by Jack Bresenham of IBM in 1965.
2. The idea: step one column at a time, keep a running error, and step down a row only when the error says we have drifted too far.
3. Set dx = 6, dy = 4. Set the decision value D = 2 times dy minus dx, so D = 8 - 6 = 2. Start at y = 0.
4. Rule at each column: plot the pixel, then if D is greater than 0, increase y by 1 and add (2dy - 2dx) to D; otherwise add 2dy to D.
5. Here is every step, with the true mathematical line for comparison:

x	D before	pixel plotted	true y
0	2	(0,0)	0.00
1	-2	(1,1)	0.67
2	6	(2,1)	1.33
3	2	(3,2)	2.00
4	-2	(4,3)	2.67
5	6	(5,3)	3.33
6	2	(6,4)	4.00

6. Every chosen pixel is the one nearest the true line. No division, no floating point, no multiplication inside the loop. Only integer addition.
7. Here is the whole thing in C, for the case where the line goes right and down and is wider than it is tall:

```
void line(int x0, int y0, int x1, int y1) {
    int dx = x1 - x0, dy = y1 - y0;
    int D = 2 * dy - dx;
    int y = y0;
    for (int x = x0; x <= x1; x++) {
        plot(x, y);
        if (D > 0) { y++; D += 2 * dy - 2 * dx; }
        else       { D += 2 * dy; }
    }
}
```

8. Now blending. Suppose a red marker at 25 percent opacity is drawn over a pale grey background.
9. Source colour is (200, 30, 30) with alpha 0.25. Destination is (240,240,240).
10. The standard source-over formula is: result = source times alpha, plus destination times (1 minus alpha).
11. Red: 200 x 0.25 + 240 x 0.75 = 50 + 180 = 230.
12. Green: 30 x 0.25 + 240 x 0.75 = 7.5 + 180 = 187.5, which rounds to 188.
13. Blue is the same as green: 188. The final pixel is (230, 188, 188), a pale pink. Exactly what a thin red wash over grey looks like.

22.2.4 PLAIN - what is really happening inside

1. The framebuffer is ordinary memory. On a modern machine it lives in the graphics card's own memory, not in main memory.
2. Pixels are stored row by row. The distance in bytes from one row to the next is called the **pitch** or stride, and it is often larger than width times bytes per pixel, because rows are padded for alignment.
3. A blit is a rectangle copy that respects both pitches. Dan Ingalls wrote the first widely copied version, BitBLT, as a microcoded routine on the Xerox Alto in 1975. The Commodore Amiga shipped a famous dedicated engine for it in 1985.

4. A sprite, on very old hardware, was not drawn into the framebuffer at all. A separate circuit overlaid it as the picture was being sent to the screen.
5. Today a sprite is simply a small textured rectangle drawn by the normal pipeline, and the word survives only as a habit.
6. Alpha is a fourth channel next to red, green and blue. 0 means fully see through, 1 (or 255) means fully solid.
7. Clipping is done by comparing each pixel's coordinates with a rectangle before writing. Modern hardware does it with a scissor test that costs nothing.
8. Your window manager does not let applications write to the screen at all. Each window draws into its own private buffer.
9. A separate program, the **compositor**, then draws all those buffers onto the real screen in the right order, with the right transparency, at the right moment.
10. This is why a frozen application leaves a stale image rather than a hole, and why shadows and rounded corners on windows are cheap.

22.2.5 TECHNICAL - the engineer's version

1. The first true framebuffer is usually credited to Richard Shoup's SuperPaint at Xerox PARC, working in 1973, holding 640 by 486 pixels at 8 bits each in 307 kilobytes of shift-register memory that cost a fortune.
2. Bresenham's algorithm appeared as "Algorithm for computer control of a digital plotter", IBM Systems Journal volume 4 number 1, 1965. He wrote it in 1962.
3. The compositing algebra everyone uses comes from Thomas Porter and Tom Duff, "Compositing Digital Images", SIGGRAPH 1984. It defines twelve operators; the one you meet daily is `over`.
4. Straight alpha stores colour and alpha separately. **Premultiplied** alpha stores colour already multiplied by alpha, which makes `over` cheaper and makes filtering correct:

```
straight:      out = src.rgb * src.a + dst.rgb * (1 - src.a)
premultiplied: out = src.rgb          + dst.rgb * (1 - src.a)
out alpha:     out.a = src.a + dst.a * (1 - src.a)
```

5. Filtering straight-alpha images produces dark or bright halos, because the colour of fully transparent texels leaks in. Premultiplied alpha does not have this fault. This is a convention question that bites everybody once.
6. Blending must happen in linear light, not in the sRGB values you store. Blending sRGB numbers directly is wrong and produces dark edges. This is a standard requirement, defined in IEC 61966-2-1 for sRGB, and routinely ignored.
7. The hardware blend stage is configurable but not programmable. In OpenGL you set it with `glBlendFuncSeparate` and `glBlendEquation`; in Vulkan through `VkPipelineColorBlendAttachmentState`.
8. Real framebuffer sizes, with a 32-bit colour buffer and a 32-bit depth buffer:

Resolution	Colour buffer	Colour + depth
1280x720	3.5 MiB	7.0 MiB
1920x1080	7.9 MiB	15.8 MiB
2560x1440	14.1 MiB	28.1 MiB
3840x2160	31.6 MiB	63.3 MiB

9. Compositors in the wild: Quartz Compositor on macOS since 2001, the Desktop Window Manager on Windows since Vista in 2006, Mutter and KWin on Wayland, SurfaceFlinger on Android.
10. Tools: `xwd` and `import` capture X11 windows; `weston-info` reports Wayland compositor capabilities; RenderDoc captures the actual draw calls of any frame on Windows, Linux and Android.

22.2.6 WORDS - remember these

1. Framebuffer — the memory that is the picture — a linear array of pixel values with a defined format and pitch, scanned out to the display.
2. Pitch — bytes per row — the stride between the start of consecutive rows, including alignment padding.
3. Blit — copy a rectangle — a two-dimensional block transfer between memory regions, historically by a dedicated engine.
4. Alpha — how solid a pixel is — a per-pixel coverage or opacity channel used by the Porter-Duff compositing operators.
5. Premultiplied alpha — colour already dimmed by alpha — storage where RGB is pre-scaled by A, required for correct filtering.
6. Clipping — do not draw outside here — restricting rasterisation to a region, implemented as a scissor rectangle test.
7. Compositor — the program that assembles the desktop — the system process that blends per-window buffers into the scanout surface.

22.3 The three-dimensional problem

22.3.1 PLAIN - in simple words

1. A screen is flat. A world is not. So we need a way to write down a solid shape as numbers, and a recipe for flattening it.
2. A shape is stored as a list of points in space. Each point is three numbers: how far right, how far up, how far forward.
3. A point on its own is invisible. Points are joined into flat patches, and almost every patch in every game is a triangle.
4. Three points always lie on one flat surface. Four points may not. That single fact is why the whole industry uses triangles.
5. A whole object is a **mesh**: a list of points plus a list of which three points make each triangle.
6. Each point also carries extra facts, most importantly a **normal**: an arrow saying which way the surface faces at that point.
7. Without normals, lighting is impossible, because light depends entirely on the angle between the surface and the lamp.
8. To get from a stored shape to a picture we move the numbers through several different ways of measuring, one after another.
9. First relative to the object itself, then to the world, then to the camera, then to the flat screen, then to actual pixel positions.

22.3.2 PLAIN - a picture in your head

1. Think of a paper model of a house that you built on your desk.
2. When you measured the pieces you measured them from a corner of the house. The chimney is 3 centimetres up from the roof. That is the house's own ruler.
3. Then you put the house on a model railway board. Now the chimney is 40 centimetres from the left edge of the board. That is the board's ruler.
4. Then you kneel down and look at the board through a cardboard tube. Now what matters is how far the chimney is in front of your eye and how far off to the side. That is your ruler.
5. Then you trace what you see onto the flat end of the tube. Things far away trace smaller. That is the flattening.
6. Finally you photocopy the tracing onto a page of a particular size. That is turning it into pixels.
7. Five rulers, one after the other. The house never moved. Only the way we measured it changed.

Where this comparison breaks: the tracing step is not a passive act of looking. It is a specific piece of

arithmetic that also throws away everything behind you, everything too close, and everything too far, and it records how far away each traced point was so that nearer things can hide farther ones later. A cardboard tube does none of that bookkeeping.

22.3.3 PLAIN - a worked example

1. Here is the smallest useful mesh: a square made of two triangles.
2. Four corners, listed once each. Position, then the normal arrow, then a pair of numbers for where to sample a picture (we meet those in section 22.7).

```
index  position (x,y,z)   normal        uv
  0    (-1, -1, 0)        (0, 0, 1)     (0, 0)
  1    ( 1, -1, 0)        (0, 0, 1)     (1, 0)
  2    ( 1,  1, 0)        (0, 0, 1)     (1, 1)
  3    (-1,  1, 0)        (0, 0, 1)     (0, 1)

triangles: (0, 1, 2) and (0, 2, 3)
```

3. Notice corners 0 and 2 are used by both triangles. That is the point of the index list: store each corner once, refer to it many times.
4. A naive format would store 6 corners for 2 triangles. This stores 4. On a real mesh the saving is close to a factor of three, because in a closed surface each vertex is shared by about six triangles.
5. Storage cost here: 4 vertices times (3 + 3 + 2) floats times 4 bytes = 128 bytes, plus 6 indices times 2 bytes = 12 bytes. Total 140 bytes.
6. A detailed game character is roughly 50,000 to 150,000 triangles. At about 32 bytes per vertex and roughly as many vertices as half the triangle count, a 100,000-triangle character costs on the order of 2 to 3 megabytes before textures.
7. The order of the three indices matters. (0,1,2) walked round the triangle is anticlockwise when seen from the front. That is how the hardware later works out which side you are looking at.

22.3.4 PLAIN - what is really happening inside

1. A model file on disk holds, at minimum: vertex positions, indices, and one or more sets of the extra per-vertex facts.
2. Common extras: normal, tangent (needed for surface-detail textures), one or two sets of texture coordinates, vertex colour, and for animated models a list of which bones move this vertex and by how much.
3. It also holds material references: which texture files, which shader, which numeric settings.
4. Now the five coordinate spaces, each reached by multiplying by a matrix.
5. **Model space**, also called object space. Coordinates measured from the object's own origin. This is what the artist made and what the file stores.
6. **World space**. Multiply by the model matrix. This places, rotates and scales the object into the shared scene. Every object in the level now shares one ruler.
7. **View space**, also called eye or camera space. Multiply by the view matrix. This re-measures everything relative to the camera, which now sits at the origin looking down one axis.
8. **Clip space**. Multiply by the projection matrix. This applies perspective and defines a box: anything outside it is not visible and is thrown away.
9. **Normalized device coordinates**. Divide every component by the fourth component, w. This is the step that actually makes distant things smaller. The visible region becomes a tidy cube from -1 to 1.
10. **Screen space**. Apply the viewport transform, which stretches that tidy range onto the real pixel grid, for example 0 to 1919 across.
11. Every one of these steps except the divide is a matrix multiply, and matrices can be multiplied together beforehand, so in practice three matrices are combined into one and applied once per vertex.

22.3.5 TECHNICAL - the engineer's version

1. Triangles win for four concrete reasons: three points are always coplanar; three points are always convex; barycentric interpolation across a triangle is well defined and cheap; and any polygon can be triangulated.
2. Quads and higher polygons are used in modelling tools for editing convenience and subdivision. They are triangulated before reaching the hardware. This is a convention, not a limit of the mathematics.
3. Index buffers are usually 16-bit (up to 65,535 vertices) or 32-bit. The post-transform vertex cache, an implementation detail typically holding 16 to 32 entries, rewards index orders that reuse recent vertices. Tools such as Tom Forsyth's linear-speed vertex cache optimizer and `meshoptimizer` reorder indices for this.
4. Winding order defines facing. OpenGL defaults to counter-clockwise front faces (`GL_CCW`); Direct3D historically defaults to clockwise. Both are configurable. Getting it wrong makes objects vanish.
5. Vertex normals are usually the area-weighted or angle-weighted average of the adjacent face normals. Hard edges are made by splitting the vertex so the two sides carry different normals, which is why a cube needs 24 vertices, not 8.
6. Common interchange formats:

Format	Year	Nature
OBJ (Wavefront)	1980s	Text, no animation
FBX (now Autodesk)	1996	Binary, closed, rich
glTF 2.0 (Khronos)	2017	JSON + binary, open
USD (Pixar)	2016	Scene graph, open

7. glTF 2.0 is deliberately close to what a graphics API wants: buffers, buffer views, accessors, and a physically based material model. It is called "the JPEG of 3D" for that reason.
8. The matrix chain, written the way most maths texts and OpenGL write it, with column vectors on the right:

```
clip = P * V * M * v_model
ndc  = clip.xyz / clip.w        (the perspective divide)
win  = viewport(ndc)
```

9. Handedness and depth range are the two places every engine disagrees. OpenGL traditionally uses a right-handed view space and clip depth from -1 to 1. Direct3D, Vulkan and Metal use depth 0 to 1. Vulkan also flips the Y axis of clip space relative to OpenGL. These are specification differences, not opinions, and porting code without fixing them produces upside-down or inside-out images.
10. Reversed-Z, storing near objects at depth 1.0 and far at 0.0 with a floating-point depth buffer, drastically improves precision. It is now standard practice in serious engines and requires a 0-to-1 depth range.

22.3.6 WORDS - remember these

1. Vertex — a corner point — a record holding a position plus arbitrary attributes, consumed by the vertex shader.
2. Mesh — a shape as a list of corners and triangles — indexed geometry: a vertex buffer plus an index buffer plus a topology.
3. Normal — which way the surface faces — a unit vector perpendicular to the surface, used in all lighting terms.
4. Winding order — the direction you walk round a triangle — the vertex order that defines front and back faces for culling.
5. Model space — measured from the object — object-local coordinates as authored.
6. View space — measured from the camera — coordinates after the view matrix, with the eye at the origin.

7. Clip space — the pre-divide box — homogeneous coordinates where visibility is tested against -w <= x,y <= w.
8. Normalized device coordinates — the tidy cube — post-divide coordinates in [-1,1] for x and y, resolution independent.

22.4 The mathematics, taught gently

22.4.1 PLAIN - in simple words

1. Three ideas carry all of three-dimensional graphics: the vector, the dot product, and the matrix. Everything else is decoration.
2. A **vector** is just a list of numbers. In graphics it is usually three or four. It can mean a position, or a direction, or a colour.
3. The **dot product** of two vectors is: multiply matching entries, add up the results. One number comes out.
4. That one number tells you how much two directions agree. If they point the same way it is large. At right angles it is exactly zero. Opposite, negative.
5. The **cross product** of two vectors gives a third vector at right angles to both. It is how you find which way a surface faces.
6. A **matrix** is a grid of numbers that describes a transformation: a rotation, a scaling, a shear, a move.
7. Multiplying a point by a matrix gives the moved point. Multiplying two matrices gives one matrix that does both jobs.
8. For three-dimensional work we use a 4x4 grid, not 3x3, and the reason is surprisingly simple: a 3x3 matrix cannot move something.
9. The same multiply, done on much bigger grids, is what a neural network does. The chip does not know or care which one you meant.

22.4.2 PLAIN - a picture in your head

1. Think of a matrix as a machine that eats an arrow and spits out a different arrow, always in a consistent way.
2. Feed it the arrow pointing one step east. It gives you back some arrow. Write that down as the first column.
3. Feed it the arrow pointing one step up. Write that answer down as the second column. Same for one step forward, third column.
4. Now you know everything the machine does, because any arrow is just so many easts plus so many ups plus so many forwards.
5. That is literally what a matrix is: the columns are where the basic direction arrows land.
6. And here is the problem. Feed such a machine the arrow of length zero, the origin. Multiply zero by anything and you get zero. The origin can never move.
7. So a 3x3 machine can turn, stretch and squash, but never shift. To allow shifting we add a fourth number to every arrow and a fourth row and column to every machine.

Where this comparison breaks: the fourth number is not a fake. It is a real coordinate in a real four-dimensional space, and the division by it at the end is the whole of perspective. Treating it as a bookkeeping trick works right up until you meet the projection matrix, at which point you need the honest story, which is in the technical block below.

22.4.3 PLAIN - a worked example

1. First the dot product. Take a surface facing straight up, normal N = (0,1,0). Take a direction to the lamp, L = (0.6, 0.8, 0).
2. Check L is length 1: 0.6 squared is 0.36, 0.8 squared is 0.64, they sum to 1.00. Good.
3. Dot product: (0 x 0.6) + (1 x 0.8) + (0 x 0) = 0.8.

4. So this surface receives 80 percent of the light it would receive if the lamp were straight overhead. That is Lambert's cosine law in one multiplication.
5. If the lamp were below the surface, the dot product would come out negative, and we clamp it to zero. Surfaces do not receive negative light.
6. Now the full journey of one point. Take a vertex at (1, 1, 0) in model space.
7. The model matrix moves the object 2 units right and 5 units away from the camera. So world position is (3, 1, -5).
8. The camera sits at the origin looking down the negative z axis, so the view matrix is the identity and view position is also (3, 1, -5).
9. The projection matrix: 90 degree vertical field of view, 16:9 aspect, near plane 0.1, far plane 100. Its non-zero entries are

```
P[0][0] = 1 / (tan(45 deg) * (16/9)) = 0.5625
P[1][1] = 1 / tan(45 deg)            = 1.0
P[2][2] = (far+near)/(near-far)      = -1.002002
P[2][3] = (2*far*near)/(near-far)    = -0.2002002
P[3][2] = -1
```

10. Apply it to the homogeneous point (3, 1, -5, 1): x = 0.5625 x 3 = 1.6875. y = 1.0 x 1 = 1.0. z = (-1.002002 x -5) + (-0.2002002 x 1) = 5.01001 - 0.2002 = 4.80981. w = -1 x -5 = 5.
11. Clip space point is (1.6875, 1.0, 4.80981, 5). Note w is now 5, not 1. The projection matrix stashed the distance into w.
12. Divide everything by w. Normalized device coordinates are (0.3375, 0.2, 0.961962). All three are inside -1 to 1, so the point is visible.
13. Viewport transform onto a 1920 by 1080 window, with y counted downwards: screen x = (0.3375 + 1) / 2 x 1920 = 0.66875 x 1920 = 1284. screen y = (1 - 0.2) / 2 x 1080 = 0.4 x 1080 = 432.
14. The vertex lands on pixel (1284, 432), and its stored depth is 0.981. Done.
15. Now move the same object twice as far away, to world z = -10. Redo it: w becomes 10, x stays 1.6875, so ndc x = 0.16875 and screen x = 1082.
16. The point moved from 1284 to 1082, that is 202 pixels closer to the centre at 960. Half the distance from centre, for twice the depth. That is perspective, and it fell out of one division.

22.4.4 PLAIN - what is really happening inside

1. A 4x4 matrix has a clean structure. The top-left 3x3 block does rotation, scale and shear. The rightmost column does translation. The bottom row is usually (0,0,0,1) and does nothing.
2. When the bottom row is not (0,0,0,1), you get perspective. The projection matrix puts -1 in the bottom row so that w ends up holding the view-space distance.
3. A point is written with w = 1. A direction is written with w = 0.
4. That single difference makes translation apply to points and not to directions, which is exactly right: moving a house does not change which way north is.
5. Matrix multiplication is not commutative. Rotate then move is a different result from move then rotate. Ninety percent of beginner bugs are this.
6. Because matrices combine, an engine computes one combined matrix on the processor per object per frame, and the graphics chip applies it once per vertex.
7. For a 100,000-vertex model that is one 4x4 by 4x4 multiply on the processor and 100,000 4x4 by 4x1 multiplies on the graphics chip.
8. Each of those is 16 multiplies and 12 adds. That is 28 arithmetic operations with no branching, no dependence on neighbours, and perfectly predictable memory access.
9. That shape of work is exactly what thousands of simple lanes are for. It is also, and this is the point of the chapter, exactly the shape of the work in a neural network.

22.4.5 TECHNICAL - the engineer's version

1. Homogeneous coordinates were introduced by August Ferdinand Möbius in 1827, long before computers, in his work on projective geometry.
2. Larry Roberts applied them to computer graphics in his 1963 MIT doctoral thesis on machine perception of solids, which is where the projection matrix as we use it comes from.
3. The geometric meaning of the dot product: a dot b equals |a||b| cos(theta). For unit vectors it is simply cos(theta).
4. The geometric meaning of the cross product: a cross b is perpendicular to both, with magnitude |a||b| sin(theta), which equals the area of the parallelogram they span. Direction follows the right-hand rule in a right-handed system.
5. Triangle face normal from vertices A, B, C is normalize(cross(B - A, C - A)). Half the magnitude of that cross product is the triangle's area, which is why the same computation drives area-weighted normal averaging.
6. Column-major versus row-major is a storage question, not a mathematical one. OpenGL and GLSL store matrices column-major; Direct3D's older maths libraries are row-major and write vectors on the left. The same transform then appears transposed. This is a convention clash and a permanent source of bugs.
7. The general perspective projection matrix, OpenGL convention, depth mapped to [-1, 1]:

```
[ f/aspect  0   0                    0                  ]
[ 0         f   0                    0                  ]
[ 0         0   (far+near)/(near-far) 2*far*near/(near-far) ]
[ 0         0  -1                    0                  ]
where f = 1 / tan(fovy / 2)
```

8. Vulkan and Direct3D map depth to [0, 1], which changes the third row. Reversed -Z swaps near and far in that row and pairs with a `GREATER` depth test.
9. Depth precision: with a 24-bit fixed-point depth buffer, near = 0.1 and far = 1000, 90 percent of the stored range is used up within the first metre. Raising the near plane helps far more than raising the buffer's bit depth. Reversed-Z with a 32-bit float buffer largely solves it.
10. The connection to machine learning, stated plainly. A vertex transform is a 4x4 matrix times a 4x1 vector. A fully connected neural network layer is an N x M weight matrix times an M x 1 activation vector, usually batched into an N x M times M x B matrix multiply.
11. Both are the same primitive: multiply-accumulate over contiguous data with no control flow. The hardware unit that does one does the other. Tensor cores, introduced on NVIDIA Volta in 2017, are simply matrix-multiply units with a fixed small tile size, and they are used by both graphics and machine learning code. Chapter 46 picks this up in detail.
12. Libraries: GLM for C++ (header-only, mirrors GLSL), `numpy` for experimenting, `cglm` for C. To check a matrix by hand, print it and verify that the fourth column is your translation.

22.4.6 WORDS - remember these

1. Vector — a list of numbers with a direction meaning — an element of a real vector space, here R3 or R4.
2. Dot product — how much two directions agree — the scalar product, which equals |a||b|cos(theta) and is zero for perpendicular vectors.
3. Cross product — a direction at right angles to two others — the vector product, magnitude |a||b|sin(theta), used for face normals.
4. Matrix — a grid that transforms — a linear map expressed in a chosen basis; its columns are the images of the basis vectors.
5. Homogeneous coordinates — the extra fourth number — projective coordinates where (x,y,z,w) and (kx,ky,kz,kw) name the same point, enabling translation and perspective as linear operations.
6. Perspective divide — dividing by w — the non-linear step that converts clip space to normalized device coordinates and makes distance shrink things.

7. Model-view-projection matrix — the three rulers combined — the product P * V * M, uploaded as a uniform and applied per vertex.

22.5 The rasterisation pipeline, stage by stage

22.5.1 PLAIN - in simple words

1. Rasterisation is the answer to one question, asked once per triangle: which pixels does this triangle cover?
2. The whole machine is a production line. Corners go in one end, coloured pixels come out the other.
3. First the machine reads the corners out of memory.
4. Then it runs your program on each corner, which is where the matrices from the last section get applied.
5. Then, optionally, it can create extra corners to add detail, or make whole new shapes.
6. Then it throws away anything outside the visible box, and cuts triangles that are half in and half out.
7. Then it does the perspective divide and maps everything onto real pixel coordinates.
8. Then it works out, for each triangle, exactly which pixel centres fall inside it.
9. For each of those it can check the depth early and skip the expensive part.
10. Then it runs your second program once per surviving pixel, which decides the colour.
11. Then it blends that colour with what is already there and writes it into memory.
12. Some of those stages you write yourself. The rest are fixed circuits you can only configure.

22.5.2 PLAIN - a picture in your head

1. Think of a car factory line.
2. At the start, a robot picks parts out of bins. That is vertex fetch.
3. The first station is staffed by a worker who can do anything you train them to do. That is the vertex shader.
4. The next stations can add extra parts or duplicate the chassis. Those are tessellation and the geometry shader, and most cars skip them.
5. Then a quality gate throws out anything that will not fit in the showroom. That is clipping.
6. Then a stamping press converts the shape into a fixed grid pattern. That is rasterisation, and it is a fixed machine, not a person.
7. Then another trainable worker paints each square. That is the fragment shader.
8. Then the final fixed machine decides whether the new paint covers the old and files the result. That is depth test and blending.

Where this comparison breaks: a factory line handles one car at a time per station. This line has thousands of cars at every station simultaneously, and the stations are not physically separate. On real hardware the same shader cores run the vertex program and the fragment program, switching between them as work arrives. The line is a logical order, not a physical layout.

22.5.3 PLAIN - a worked example

1. Draw one triangle whose screen positions after the divide and viewport transform are A = (100, 100), B = (300, 100), C = (100, 300).
2. The bounding box is x from 100 to 300 and y from 100 to 300. That is 201 by 201, so 40,401 candidate pixels.
3. The triangle is half of that box, so about 20,200 pixels are actually inside.
4. Each candidate is tested with three edge functions. For edge AB the function is E(x,y) = (B.y - A.y)(x - A.x) - (B.x - A.x)(y - A.y).
5. Substituting A and B: E(x,y) = 0 x (x - 100) - 200 x (y - 100), which is -200(y - 100). It is negative for y above 100 and zero on the line.
6. A pixel is inside when all three edge functions have the same sign. Test the pixel centre at (150.5, 150.5) and it passes all three.

7. Once inside, the hardware computes barycentric weights. For the pixel (150.5, 150.5): u = 0.2525 toward B, v = 0.2525 toward C, and 0.495 toward A.
8. Those weights interpolate every attribute. If A, B and C had depths 0.5, 0.7 and 0.9, this pixel's depth is 0.495 x 0.5 + 0.2525 x 0.7 + 0.2525 x 0.9 = 0.6515.
9. That depth is compared against what is already in the depth buffer. If the stored value is nearer, this pixel is discarded before the fragment shader ever runs.
10. Fragments are always processed in 2x2 groups, called quads, even at the triangle edge where one or two of the four are outside. The wasted ones are called helper lanes. On thin triangles this can waste half the work, which is why very high triangle counts stop paying off.

22.5.4 PLAIN - what is really happening inside

1. Here is the full order, with which parts you can program:

```
[ vertex + index buffers in VRAM ]
              |
              v
     vertex fetch                    fixed function
              |
              v
     VERTEX SHADER                   you write this
              |
              v
   tessellation (optional)           2 of 3 stages yours
              |
              v
   GEOMETRY SHADER (optional)        you write this
              |
              v
     clipping                        fixed function
              |
              v
   perspective divide                fixed function
              |
              v
   viewport transform                fixed function
              |
              v
   triangle setup                    fixed function
              |
              v
   rasterisation                     fixed function
              |
              v
   early depth test                  fixed, configurable
              |
              v
     FRAGMENT SHADER                 you write this
              |
              v
   depth / stencil test              fixed, configurable
              |
              v
      blending                       fixed, configurable
              |
              v
[ framebuffer in VRAM ]
```

2. **Vertex fetch** reads the index buffer, looks up each index in the vertex buffer, and assembles a struct for each vertex.
3. **Vertex shader** runs once per vertex. Its only required output is the clip space position. It may output anything else you want passed along.
4. **Tessellation** is three stages: a control shader you write, a fixed tessellator that subdivides a patch by the factors you gave, and an evaluation shader you write that positions each new vertex.
5. **Geometry shader** can emit zero, one or many primitives per input primitive. It is flexible and, on most hardware, slow. Modern engines avoid it.
6. **Clipping** removes primitives fully outside the view volume and cuts those that straddle it, producing new vertices with interpolated attributes.
7. **Perspective divide** divides x, y and z by w.
8. **Viewport transform** maps the -1 to 1 range onto the pixel rectangle and maps depth into the depth buffer's range.
9. **Triangle setup** computes the edge functions and the per-attribute interpolation gradients once per triangle, so the per-pixel work is cheap.
10. **Rasterisation** walks tiles of the bounding box and tests pixel centres against the edge functions, emitting fragments.
11. **Early depth test** compares depth before the fragment shader. It is only safe when the shader does not change depth and does not discard pixels.
12. **Fragment shader** runs once per fragment and outputs colour, and optionally depth.
13. **Blending** combines the shader's output with the existing pixel, then the result is written to the framebuffer.

22.5.5 TECHNICAL - the engineer's version

1. The programmable stages, in Direct3D 11 and OpenGL 4 terms, are: vertex, hull (tessellation control), domain (tessellation evaluation), geometry, and pixel (fragment). Compute exists outside the graphics pipeline.
2. Everything else is fixed function. Fixed does not mean unconfigurable. The depth test comparison, blend factors, cull mode, scissor rectangle, viewport and stencil operations are all set through pipeline state.
3. Modern GPUs are **tile-based** or use binning even on desktop. AMD RDNA's Draw Stream Binning Rasterizer and NVIDIA's tiled caching, first observed on Maxwell in 2014, sort triangles into screen tiles to improve cache locality. These are implementation details and are not exposed in the APIs.
4. Rasterisation rules are specified precisely so that adjacent triangles neither double-draw nor leave gaps. Direct3D and Vulkan both define a top-left fill rule and fixed-point vertex snapping, typically 8 subpixel bits.
5. Early depth test is called Early-Z or Hi-Z. Writing to `gl_FragDepth`, calling `discard`, or using certain blend modes disables it. A single stray `discard` in a shader can cost a large fraction of frame time.
6. Attribute interpolation must be perspective correct: the hardware interpolates attribute/w and 1/w linearly in screen space, then divides. Doing it without the divide produces the warped textures of 1990s consoles. The original PlayStation of 1994 famously lacked perspective correction.
7. The mesh shader pipeline, introduced with NVIDIA Turing in 2018 and standardized in Direct3D 12 Ultimate in 2020 and Vulkan's `VK_EXT_mesh_shader` in 2022, replaces vertex, tessellation and geometry with two programmable stages: task and mesh. It is a genuine architectural change, not a rename.
8. Fragment quads: derivatives such as `dFdx` and `dFdy`, used for texture mipmap selection, only exist because fragments are shaded in 2x2 groups. This is why the quad is a hardware invariant and not an optimization.
9. Observation tools: RenderDoc and NVIDIA Nsight Graphics let you step through a captured frame stage by stage, inspect the post-vertex-shader output, and see per-draw pixel counts. `GL_ARB_pipeline_statistics_query` and Direct3D's pipeline statistics queries report primitives in, primitives clipped, and fragment shader invocations.

22.5.6 WORDS - remember these

1. Rasterisation — deciding which pixels a shape covers — converting primitives into fragments by testing sample positions against edge functions.
2. Fragment — a candidate pixel — a potential contribution to a pixel, carrying interpolated attributes, before depth and blend decide its fate.
3. Barycentric coordinates — how far toward each corner — the three weights that express a point inside a triangle as a combination of its vertices.
4. Early-Z — check depth before painting — a depth test executed before the fragment shader, disabled by depth writes or discard in the shader.
5. Fixed function — a circuit you configure, not program — a hardware stage with settable state but no user code.
6. Quad — the 2x2 shading group — the minimum granularity of fragment shading, required for screen-space derivatives.
7. Mesh shader — the new front end — a compute-like stage that emits meshlets directly, replacing vertex, tessellation and geometry stages.

22.6 Shaders

22.6.1 PLAIN - in simple words

1. A shader is a small program that runs on the graphics chip instead of the processor.
2. The name is misleading. A shader does not only handle shading. It is just the name for any program that runs at one of the programmable stages.
3. A **vertex shader** runs once for each corner. Its job is to say where that corner ends up.
4. A **fragment shader** runs once for each candidate pixel. Its job is to say what colour that pixel should be.
5. Both are written in a small C-like language. The most common are GLSL for OpenGL and Vulkan, HLSL for Direct3D, and MSL for Apple's Metal.
6. Values that are the same for every invocation, such as the camera matrix or the time, are called **uniforms**. You set them from the processor.
7. Values that the vertex shader computes and that get smoothly blended across the triangle for the fragment shader are the varying outputs.
8. Pictures the shader can look things up in are **textures**, and the object describing how to look them up is a **sampler**.
9. Thousands of copies of the same shader run at the same instant on different data. No copy can see another copy's variables.
10. A **compute shader** is the same machinery without any drawing at all. You just say how many invocations you want and they run. That is what opened the chip to work that has nothing to do with pictures.

22.6.2 PLAIN - a picture in your head

1. Imagine a huge examination hall with twenty thousand desks.
2. Every candidate has the exact same question paper. That is the shader program.
3. On the front board are some facts everyone needs: today's date, the exchange rate. Those are uniforms. Nobody may change them.
4. In each candidate's envelope is their own individual data: a corner position, or a pixel coordinate. That is the per-invocation input.
5. There is a reference library at the side of the hall that anyone may consult. That is texture memory.
6. Candidates may not talk to each other. Each writes one answer and leaves.
7. The invigilator does not wait for candidate 1 before starting candidate 2. Everyone works at once.

Where this comparison breaks: candidates in the same row of thirty-two must turn the page at the same moment. If one candidate needs to read a different section of the paper, the whole row goes there and the others sit idle. And in a compute shader, small groups of candidates do get a shared scratch table they can all write on, which is the one exception to the no-talking rule and the reason compute shaders can do things fragment shaders cannot.

22.6.3 PLAIN – a worked example

1. Here is a complete, working vertex shader in GLSL. It transforms a position and passes a texture coordinate and a normal onward.

```
#version 330 core
layout(location = 0) in vec3 aPos;
layout(location = 1) in vec3 aNormal;
layout(location = 2) in vec2 aUV;

uniform mat4 uMVP;
uniform mat3 uNormalMatrix;

out vec3 vNormal;
out vec2 vUV;

void main() {
    vNormal     = normalize(uNormalMatrix * aNormal);
    vUV         = aUV;
    gl_Position = uMVP * vec4(aPos, 1.0);
}
```

2. Line by line: the three `in` variables are read from the vertex buffer. The two `uniform` values were set once by the processor.
3. `gl_Position` is the one required output: the clip space position from section 22.4.
4. `vNormal` and `vUV` are outputs that the hardware will interpolate across the triangle.
5. Here is the matching fragment shader. It looks up a texture and applies one diffuse light.

```
#version 330 core
in vec3 vNormal;
in vec2 vUV;

uniform sampler2D uAlbedo;
uniform vec3 uLightDir;

out vec4 fragColour;

void main() {
    vec3  base = texture(uAlbedo, vUV).rgb;
    float ndotl = max(dot(normalize(vNormal), uLightDir), 0.0);
    fragColour = vec4(base * (0.1 + 0.9 * ndotl), 1.0);
}
```

6. `vNormal` and `vUV` arrive already interpolated. They are not the same values the vertex shader wrote; they are a weighted blend of three vertices' values.
7. `sampler2D` is the combined texture and sampler. `texture()` does the lookup, including filtering and mipmap selection, in one hardware instruction.
8. `dot(N, L)` is the Lambert term from section 22.4. The 0.1 is a crude ambient floor so that unlit sides are not pure black.
9. Now count the work. At 1920 by 1080 with every pixel covered once, this fragment shader runs 2,073,600 times per frame. At 60 frames per second that is 124 million invocations per second, of a

program with one texture fetch and about ten arithmetic operations.

10. A compute shader, for comparison, invents its own grid. This one inverts an image:

```
#version 430
layout(local_size_x = 16, local_size_y = 16) in;
layout(rgba8, binding = 0) uniform image2D uImage;

void main() {
    ivec2 p = ivec2(gl_GlobalInvocationID.xy);
    vec4  c = imageLoad(uImage, p);
    imageStore(uImage, p, vec4(1.0 - c.rgb, c.a));
}
```

11. There is no triangle, no vertex, no framebuffer. You dispatch a grid of workgroups and each of the 256 invocations per group handles one pixel.

22.6.4 PLAIN - what is really happening inside

1. Your shader text is not what the chip runs. It goes through two compilations.
2. First, the source is compiled to a portable intermediate form. For Vulkan that is SPIR-V, a binary format standardized by Khronos in 2015. For Direct3D it is DXIL, based on LLVM bitcode.
3. Second, the graphics driver compiles that intermediate form into the actual machine code of the specific chip installed in your machine.
4. That second step is why the same game runs on chips whose instruction sets are completely different and undocumented.
5. Uniforms are not registers. They live in a small buffer in memory that every invocation reads. Because every invocation reads the same address, the read is broadcast, and it is close to free.
6. Varying outputs are written into on-chip storage by the vertex shader, then the triangle setup stage turns them into interpolation gradients, and the fragment shader reads interpolated values.
7. A texture fetch is not a normal memory read. Dedicated texture units handle address wrapping, format decoding, filtering between neighbouring texels, and mipmap level choice, all in hardware.
8. Because a texture fetch may take hundreds of cycles, the scheduler issues it and immediately switches to another warp. This is exactly the latency hiding from section 22.1.
9. The number of registers your shader needs directly limits how many warps can be resident. A shader with too many live variables reduces occupancy and can run slower despite doing less arithmetic.

22.6.5 TECHNICAL - the engineer's version

1. History, briefly. Fixed-function transform and lighting arrived on the NVIDIA GeForce 256 in August 1999. Programmable vertex and pixel shaders were specified in Direct3D 8 in November 2000 and first shipped on the GeForce 3 in February 2001, written in assembly with instruction-count limits in the low tens.
2. High level languages followed: Cg and HLSL in 2002, GLSL in OpenGL 2.0 in 2004. Unified shader hardware, where one pool of cores runs all stages, arrived with the ATI Xenos in the Xbox 360 in 2005 and NVIDIA G80 in November 2006.
3. Compute shaders were standardized in Direct3D 11 in 2009 and OpenGL 4.3 in 2012. Metal and Vulkan had them from the start.
4. Shader stage inputs and outputs are declared with explicit locations, and the interface between stages must match or linking fails. In Vulkan this is validated at pipeline creation, not at draw time.
5. Uniform buffer objects have a guaranteed minimum size of 16 KiB in OpenGL and Vulkan; shader storage buffers are much larger and writable. Push constants in Vulkan give a small, fast path, with a guaranteed minimum of 128 bytes.
6. Interpolation qualifiers change behaviour: `smooth` is perspective correct and is the default, `noperspective` is linear in screen space, `flat` takes the value from the provoking vertex with no interpolation.

7. Workgroup limits are guaranteed minimums, not typical values. OpenGL 4.3 guarantees at least 1024 invocations per workgroup and 32 KiB of shared memory; Vulkan guarantees only 128 invocations and 16 KiB. Real desktop hardware offers 1024 and 32 KiB or more, but relying on that is not portable.
8. Subgroup or wave intrinsics let invocations in one warp exchange values directly through registers. `subgroupAdd`, `subgroupBallot` and friends were added in Vulkan 1.1 in 2018. They are the fastest reduction primitive available and are heavily used in machine learning kernels.
9. Tools: `glslangValidator` and `glslc` compile GLSL to SPIR-V; `spirv-cross` converts between shader languages; `dxc` compiles HLSL to DXIL and SPIR-V; NVIDIA's `nvdisasm` and AMD's Radeon GPU Analyzer show the real machine code and register counts.

22.6.6 WORDS - remember these

1. Shader — a program that runs on the graphics chip — user code executed at a programmable pipeline stage in SIMT fashion.
2. Uniform — a value that is the same everywhere — read-only per-draw constant data, uploaded from the host into a constant or uniform buffer.
3. Varying — a value blended across the triangle — a vertex shader output interpolated per fragment, usually perspective correct.
4. Sampler — the rules for reading a picture — the object holding filter mode, wrap mode, mip bias and comparison state used by a texture fetch.
5. SPIR-V — the portable half-compiled shader — the Khronos binary intermediate representation consumed by Vulkan and OpenCL, standardized in 2015.
6. Compute shader — a shader with no drawing — a general kernel dispatched over an explicit grid of workgroups with shared local memory.
7. Workgroup — a small team with a shared table — a block of invocations that share on-chip memory and can synchronize with a barrier.

22.7 Textures and surface detail

22.7.1 PLAIN - in simple words

1. A triangle mesh gives shape. It does not give pattern, colour or fine detail.
2. Those come from **textures**: ordinary images wrapped onto the surface.
3. To wrap an image you need to say, for each corner of each triangle, which point of the image it corresponds to.
4. That pair of numbers is called a **UV coordinate**, running from 0 to 1 across the image in each direction.
5. Between corners, the machine blends UV values, so every pixel of the triangle knows where to look in the image.
6. The image pixel you land on is called a **texel**, to keep it separate from a screen pixel. They are almost never the same size.
7. If a texel is bigger than a pixel the image looks blocky. If it is much smaller, distant surfaces sparkle and crawl horribly.
8. The fix for the second problem is to store the same image several times, each half the size of the last. That stack is a **mipmap**.
9. Textures do more than colour. One can store which way each tiny bump of the surface faces, giving fine detail with no extra triangles.
10. Textures are the single biggest consumer of graphics memory in any real game.

22.7.2 PLAIN - a picture in your head

1. Think of wrapping a present with patterned paper.
2. The box is the mesh. The paper is the texture. Deciding which part of the paper touches which face of the box is UV mapping.

3. To do it properly you first cut the paper into a flat pattern that unfolds to fit the box. Artists really do this, and call it unwrapping.
4. Now step back across the room. The pattern is too fine to see and appears to shimmer. So instead you use a version of the paper printed with a coarser pattern for viewing from far away.
5. You keep a whole set: full detail, half detail, quarter detail, down to a single average-coloured square. You choose by distance.

Where this comparison breaks: paper has no thickness and no lighting behaviour. A modern surface uses four or five separate images at once for the same patch: one for colour, one for the direction of tiny bumps, one for how metallic it is, one for how rough it is, and often one for how much light gets trapped in crevices. They are all sampled with the same UV coordinates.

22.7.3 PLAIN - a worked example

1. Take a 4096 by 4096 texture. That is 16,777,216 texels.
2. Stored as four bytes per texel, uncompressed, it needs 64 mebibytes.
3. Now add the mipmap chain: 2048 squared, then 1024, then 512, and so on to 1x1. Each level is a quarter of the previous, so the whole chain adds one third. Total 85.3 mebibytes.
4. Now compress it. Here are the real costs for the same image:

Format	Bits per texel	With full mipmaps
RGBA8 uncompressed	32	85.3 MiB
BC7 (high quality)	8	21.3 MiB
BC1 (old, no alpha)	4	10.7 MiB
ASTC 8x8 (mobile)	2	5.3 MiB

5. A game character might use five such textures. Uncompressed that is 426 mebibytes for one character. In BC7 it is 107 mebibytes. Compression is not optional.
6. Now filtering. Suppose a pixel lands at UV (0.30012, 0.50008) on a 1024 by 1024 texture. That is texel position (307.32, 512.08).
7. Nearest filtering picks texel (307, 512) and ignores the fractions.
8. Bilinear filtering blends four texels: (307,512), (308,512), (307,513), (308,513), with weights (1-0.32)(1-0.08) = 0.6256, 0.32 x 0.92 = 0.2944, 0.68 x 0.08 = 0.0544, and 0.32 x 0.08 = 0.0256. Those four sum to 1.
9. Trilinear filtering does that twice, on the two nearest mipmap levels, then blends between them by the fractional level. Eight texels, one result.
10. Anisotropic filtering takes several trilinear samples along the direction the surface is stretched. At 16x it may take up to sixteen of them. It is the single best-value image quality setting in any game.

22.7.4 PLAIN - what is really happening inside

1. Mipmaps exist because of sampling theory, not because of laziness. If the texture has more detail than the pixel grid can represent, that detail folds back as false patterns. That is aliasing.
2. The hardware picks a mipmap level automatically. It compares the UV of the pixel to the right and the pixel below, using the 2x2 quad from section 22.5, and takes the base-2 logarithm of the larger change.
3. A **normal map** stores, in the red, green and blue channels, the x, y and z of a tiny surface normal, in a coordinate frame attached to the surface.
4. Because it is only a normal, not real geometry, the silhouette of the object never changes. Look along a bumpy wall and the edge is still perfectly flat.
5. A **bump map** or height map stores only height, and the normal is derived from the slope. It is older, smaller, and less exact.

6. A **cube map** is six square textures forming a box around a point. Look up with a direction vector instead of a UV pair. This is how skies and reflections are stored.
7. Block compression works on 4x4 blocks. Each block stores two endpoint colours and a small index per texel that says how far between them the texel lies.
8. That fixed block size is the point: any texel's address can be computed directly, so the texture unit can decompress a single block on demand without touching the rest of the image.

22.7.5 TECHNICAL - the engineer's version

1. Texture mapping was introduced in Edwin Catmull's 1974 University of Utah doctoral thesis, alongside the depth buffer. Mipmapping came from Lance Williams, "Pyramidal Parametrics", SIGGRAPH 1983. The name is from the Latin "multum in parvo", much in little.
2. Bump mapping is James Blinn, "Simulation of Wrinkled Surfaces", SIGGRAPH 1978. Environment mapping is Blinn and Newell, 1976.
3. Compression families by platform, all standards:

Family	Where used	Typical rate
BC1-BC7 (DXT/S3TC)	PC, consoles	4 or 8 bpp
ETC2 / EAC	Android, OpenGL ES 3	4 or 8 bpp
ASTC	Mobile, Vulkan	0.89 to 8 bpp
BC6H	HDR textures	8 bpp

4. BC6H stores high dynamic range data and is the correct choice for environment and light probe maps. BC5 stores two channels and is the standard choice for normal maps, with z reconstructed in the shader as sqrt(1 - x^2 - y^2).
5. Anisotropic filtering degree is capped by hardware, commonly 16x on desktop. The cost is extra texture fetches, so it is bandwidth-bound, not arithmetic-bound. Measured cost in modern games is typically under 3 percent of frame time at 16x.
6. Real memory budgets, observed rather than specified. In 2026, a demanding PC title at 4K with ray tracing and the highest texture setting commonly reports 10 to 14 gigabytes of video memory allocated, of which the large majority is textures. At 1080p with medium textures the same title fits in 6 to 8 gigabytes.
7. This is why 8 gigabyte cards became controversial from 2023 onward. The argument is not about raw speed; it is about what happens when the working set does not fit and textures must stream over PCIe every frame, producing stutter and visibly blurred surfaces.
8. Virtual texturing, sparse textures and the Direct3D 12 Sampler Feedback feature, added in 2020, let an engine load only the mip levels and tiles actually sampled. Unreal Engine 5's Nanite and Virtual Shadow Maps rely on this class of technique.
9. Tools: `texconv` and `compressonator` for offline compression; RenderDoc shows every bound texture with its exact format and mip chain; `nvidia-smi` and `radeontop` report video memory in use, though allocated is not the same as actively needed.

22.7.6 WORDS - remember these

1. Texel — a pixel of a texture — an addressable element of a texture image, distinct from a screen pixel.
2. UV coordinate — where on the picture this corner sits — normalized texture coordinates in [0,1], interpolated per fragment.
3. Mipmap — the stack of shrinking copies — a prefiltered image pyramid, each level half the linear size, selected by screen-space derivatives.
4. Bilinear filtering — blend the four nearest dots — weighted average of the 2x2 texel neighbourhood within one mip level.

5. Trilinear filtering — blend between two shrink levels too — bilinear in each of two adjacent mip levels, then linearly interpolated between them.
6. Anisotropic filtering — extra samples along the stretched direction — multiple trilinear taps along the axis of greatest UV derivative, up to the hardware cap.
7. Normal map — a picture of which way each speck faces — a texture encoding tangent-space normals, giving lighting detail without geometry.
8. Block compression — fixed-size squeezed tiles — lossy fixed-rate texture formats decoded in the texture unit, allowing random access.

22.8 Making it look real

22.8.1 PLAIN - in simple words

1. Once you can put a coloured triangle on screen, everything else is about deciding what colour, and which triangle wins.
2. Which triangle wins is settled by the **depth buffer**: a second image that stores, for every pixel, how far away the nearest thing drawn so far is.
3. Draw order stops mattering. Each pixel simply keeps the nearest.
4. When two surfaces are at almost the same distance, rounding error makes the winner flicker between them. That is **z-fighting**.
5. You can also skip any triangle whose back is facing you, since you cannot see the inside of a solid object. That is **back-face culling**, and it removes about half of all triangles for free.
6. Colour comes from a lighting model: a formula combining a base ambient level, a part that depends on the angle to the lamp, and a shiny highlight.
7. Shadows are handled by rendering the scene once from the lamp's position, storing only depth, then checking each pixel against that.
8. Modern materials are described not by made-up numbers but by physical ones: base colour, whether it is metal, and how rough it is.
9. Light that has bounced off other surfaces is called global illumination and is the hardest part of the whole subject.
10. Finally, effects applied to the finished image, such as blur and colour grading, are called post-processing.

22.8.2 PLAIN - a picture in your head

1. Think of painting a stage set with an assistant holding a tape measure.
2. Every time you are about to paint a square, the assistant tells you how far away the last thing painted on that square was.
3. If your new object is nearer, you paint and the assistant writes down the new distance. If it is further, you skip it.
4. That is the depth buffer, and it means you can paint the scenery in any order at all.
5. Now for shadows: you walk to where the lamp is, look at the set from there, and write down how far the nearest surface is in every direction.
6. Back at your easel, for each square you ask: is this point further from the lamp than the nearest thing the lamp could see in that direction? If yes, it is in shadow.

Where this comparison breaks: the tape measure has limited precision. Two surfaces a millimetre apart at a hundred metres get the same reading, and the painter flickers between them. That is z-fighting, and it is not a bug in the idea, it is a limit of storing distance in a finite number of bits.

22.8.3 PLAIN - a worked example

1. Take a surface with normal N = (0,1,0), light direction L = (0.6, 0.8, 0), and view direction V = (0, 1, 0). Shininess exponent 32.
2. Diffuse term: N dot L = 0.8. Lambert's law, straight from section 22.4.
3. Phong specular uses the mirror-reflected light direction: R = 2(N dot L)N - L = (0, 1.6, 0) - (0.6, 0.8, 0) = (-0.6, 0.8, 0).
4. R dot V = (-0.6 x 0) + (0.8 x 1) = 0.8. Raise to the power 32: 0.8^32 = 0.00079. Almost no highlight.
5. Blinn-Phong instead uses the halfway vector between light and view: H = normalize(L + V) = normalize(0.6, 1.8, 0) = (0.3162, 0.9487, 0).
6. N dot H = 0.9487. Raise to the power 32: 0.9487^32 = 0.185. A clearly visible highlight.
7. Those two answers differ by a factor of 230. The models are not interchangeable at the same exponent.
8. To match Blinn-Phong to Phong you need roughly four times the exponent. Check: 0.9487^128 = 0.00118, which is close to Phong's 0.00079. The rule of thumb holds.
9. Full Blinn-Phong for this pixel, with ambient 0.05, diffuse strength 0.8, specular strength 0.5: result = 0.05 + 0.8 x 0.8 + 0.5 x 0.185 = 0.05 + 0.64 + 0.0925 = 0.7825.

22.8.4 PLAIN - what is really happening inside

1. The formulas, written out plainly. Lambert diffuse: `kd * max(dot(N, L), 0)`. Phong specular: `ks * pow(max(dot(R, V), 0), shininess)` with `R = reflect(-L, N)`. Blinn-Phong specular: `ks * pow(max(dot(N, H), 0), shininess)` with `H = normalize(L + V)`.
2. Blinn-Phong is cheaper, because computing H is fewer operations than reflecting a vector, and it behaves better at grazing angles where Phong produces an unnatural hard cut-off.
3. Shadow mapping in exact steps: render the scene from the light, store depth only; then when shading, transform the pixel's world position into the light's space, look up the stored depth, and compare.
4. Two errors follow immediately. If the comparison is exact, surfaces shadow themselves in a stripey pattern called shadow acne, fixed with a small bias. Too much bias and shadows detach from their objects, called peter-panning.
5. Physically based rendering replaces made-up shininess with measurable parameters: **albedo** (the base colour a surface reflects diffusely), **metalness** (0 for non-metal, 1 for metal, nothing in between in the standard model), and **roughness** (0 mirror, 1 completely scattered).
6. The reason it works is energy conservation: a surface may not reflect more light than it receives. Older models happily broke that and artists corrected it by eye, which is why old games look different under different lighting.
7. Global illumination is light that has bounced at least once. Direct lighting is one lamp to one surface to the eye. Indirect is lamp to wall to floor to eye, and it is what makes a room look like a room.
8. Ambient occlusion is a cheap stand-in: darken creases and contact points, because less bounced light reaches them.

22.8.5 TECHNICAL - the engineer's version

1. The depth buffer, or z-buffer, is from Edwin Catmull's 1974 thesis, with Wolfgang Strasser describing the same idea independently in 1974.
2. Z-fighting is a precision problem. With a 24-bit fixed-point buffer, near 0.1 and far 1000, the gap between representable depths is about 6 millimetres at 100 metres and about 60 centimetres at 1000 metres. Two surfaces closer together than that gap cannot be told apart.
3. Fixes, in order of preference: move the near plane out; use reversed-Z with a 32-bit float depth buffer; use a depth bias with `glPolygonOffset`; last resort, physically separate the surfaces.
4. Back-face culling uses the sign of the signed screen-space area of the triangle after projection. It removes about half the triangles of a closed mesh and costs nothing, so it is on by default in every engine.
5. Lighting model history: Henri Gouraud published per-vertex interpolated shading in 1971. Bui Tuong Phong published his reflection model and per-pixel normal interpolation in Communications of the

ACM in June 1975, dying of leukaemia in the same year at 32. James Blinn published the halfway-vector variant at SIGGRAPH in 1977.

6. Cook and Torrance published the first physically motivated microfacet model in 1982. The parameterization the industry actually uses comes from Brent Burley's "Physically Based Shading at Disney", SIGGRAPH 2012, usually called the Disney principled BRDF.
7. The standard real-time specular term today is the Cook-Torrance form D * F * G / (4 (N dot L)(N dot V)), with GGX for the normal distribution D (Walter and others, 2007), the Schlick approximation for Fresnel F (1994), and a Smith height-correlated term for G.
8. Shadow map filtering: percentage closer filtering (Reeves, Salesin and Cook, 1987) samples several nearby depths and averages the comparison results, not the depths. Cascaded shadow maps split the view frustum into 3 or 4 ranges, each with its own map, and are the standard for outdoor sunlight.
9. Screen space ambient occlusion was introduced by Crytek in Crysis in 2007. Ground truth ambient occlusion (GTAO, 2016) is the common modern choice.
10. Post-processing effects and their real costs at 4K, order of magnitude, on a mid-range 2025 card: tone mapping under 0.1 ms, bloom 0.3 to 0.8 ms, motion blur 0.5 to 1.5 ms, depth of field 0.5 to 2 ms, screen space reflections 1 to 3 ms. In a 16.7 ms frame budget these add up quickly.

22.8.6 WORDS - remember these

1. Depth buffer — a per-pixel record of nearest distance — the z-buffer, tested and optionally written per fragment, making draw order irrelevant.
2. Z-fighting — flickering between two surfaces — depth quantization error when two surfaces fall within one representable depth step.
3. Back-face culling — skip triangles facing away — discarding primitives by the sign of their projected signed area.
4. Diffuse — brightness by angle to the lamp — the Lambertian term, proportional to max(N dot L, 0).
5. Specular — the shiny highlight — the mirror-like lobe, modelled by Phong, Blinn-Phong or a microfacet distribution such as GGX.
6. Shadow map — a depth picture taken from the lamp — a depth buffer rendered from the light, compared against per fragment to test occlusion.
7. Albedo, metalness, roughness — colour, is-it-metal, how scattered — the three core parameters of the metallic-roughness physically based material model.
8. Ambient occlusion — darkening in the creases — an estimate of how much of the surrounding hemisphere is blocked at each point.

22.9 Ray tracing

22.9.1 PLAIN - in simple words

1. Rasterisation asks: for each triangle, which pixels does it cover?
2. Ray tracing asks the opposite: for each pixel, which triangle does it see?
3. To answer that, you shoot an imaginary straight line from the eye through that pixel and find the first thing it hits.
4. Once you have hit something, you can shoot more lines: toward the lamp to test for shadow, or in the mirror direction to find a reflection.
5. That is why ray tracing gets reflections, refraction and correct shadows almost for free, while rasterisation needs a separate trick for each.
6. The cost is brutal. Testing one line against every triangle in a scene of ten million triangles is ten million tests, for every one of two million pixels.
7. The fix is to sort the triangles into a tree of nested boxes, so that one test against a big box can eliminate a million triangles at once.
8. Even with the tree, this was far too slow for real time for about forty years.

9. In 2018 graphics chips gained circuits dedicated to walking that tree and testing lines against triangles, and real-time ray tracing became possible.
10. Games do not use it for everything. They rasterise most of the image and use rays only for shadows, reflections or bounced light. That is hybrid rendering.

22.9.2 PLAIN – a picture in your head

1. Imagine finding which shop in a city a laser pointer is aimed at.
2. The stupid method is to visit all forty thousand shops and check each one.
3. The sensible method is a nested set of boxes. Is the beam inside the city boundary? Yes. Which district? North. Which street in that district? Which building on that street? Which shop in the building?
4. Five or six questions instead of forty thousand. Each question halves or better the remaining candidates.
5. That nested set of boxes is called a bounding volume hierarchy.

Where this comparison breaks: the city does not move. A game scene does, every frame, and the hierarchy has to be rebuilt or repaired for anything that moved, by the processor or by the chip itself. That rebuild cost is a real and significant part of the frame time and is invisible in screenshots.

22.9.3 PLAIN – a worked example

1. A ray is written as P(t) = O + tD, where O is the origin, D is a unit direction, and t is distance travelled.
2. A sphere of radius r centred at C is every point where the distance to C equals r.
3. Substituting one into the other gives a quadratic in t: t^2 + 2t(D dot (O - C)) + |O - C|^2 - r^2 = 0.
4. Real numbers. Eye at the origin O = (0,0,0). Direction D = (0,0,-1). Sphere at C = (0,0,-10) with radius 2.
5. O - C = (0,0,10). D dot (O - C) = -10. So b = -20 and c = 100 - 4 = 96.
6. Discriminant = b^2 - 4c = 400 - 384 = 16. Its square root is 4.
7. t = (20 plus or minus 4) / 2, so t = 8 or t = 12.
8. The nearer hit is t = 8, at the point (0, 0, -8). Correct: the front of the sphere is at z = -10 + 2 = -8.
9. A negative discriminant means the ray misses. A discriminant of exactly zero means it grazes the surface.
10. For triangles the standard method is Moller-Trumbore, published in 1997. It solves for the barycentric coordinates and the distance in one go, using two cross products and needing no precomputed plane equation.

22.9.4 PLAIN – what is really happening inside

1. A bounding volume hierarchy is a tree. Each node holds a box that contains everything below it. Leaves hold a handful of triangles.
2. Traversal keeps a small stack. Test the ray against the two child boxes; descend into the nearer one first; if you find a hit closer than the far box, you never open the far box at all.
3. Ray-box tests are cheap: six comparisons using the slab method. Ray-triangle tests are more expensive, so the tree exists to minimize how many you do.
4. In a well-built tree, a ray touches roughly 20 to 60 nodes and 2 to 10 triangles in a scene of millions.
5. The 2018 hardware does two specific things. One unit walks the tree and does box tests. Another does triangle intersection. Both run while the shader cores do something else.
6. Path tracing is ray tracing taken to its logical end: at every hit, pick a random new direction, keep going, and average over many random paths. It converges to the true answer given enough samples.
7. Real time can afford about one to two paths per pixel. That gives a violently noisy image.
8. So the last stage is a **denoiser**: an algorithm, today usually a small neural network, that turns a noisy one-sample image plus depth and normal information into a clean one.
9. Without the denoiser, real-time path tracing would be useless. It is not a polish step, it is a load-bearing part of the system.

22.9.5 TECHNICAL - the engineer's version

1. Arthur Appel described ray casting for shading solids in 1968. Turner Whitted added recursive reflection and refraction in "An Improved Illumination Model for Shaded Display", Communications of the ACM, June 1980. His famous test image took 74 minutes on a VAX 11/780.
2. James Kajiya formalized the rendering equation and path tracing in "The Rendering Equation", SIGGRAPH 1986. Everything since is a method of approximating that one integral.
3. Bounding volume hierarchies for ray tracing go back to Rubin and Whitted in 1980 and Kay and Kajiya in 1986. The surface area heuristic, the standard build quality metric, is from Goldsmith and Salmon, 1987.
4. Hardware acceleration arrived with NVIDIA Turing, announced 20 August 2018 and shipping 20 September 2018, in the RTX 2080 and 2080 Ti. NVIDIA quoted 10 billion rays per second for the 2080 Ti.
5. AMD added ray accelerators with RDNA 2 in November 2020; RDNA 4, in March 2025, doubled ray-triangle throughput per compute unit and added oriented bounding boxes. Intel shipped ray tracing units with Arc Alchemist in 2022.
6. API support is a standard: DirectX Raytracing (DXR) announced March 2018, Vulkan Ray Tracing extensions ratified November 2020, and Metal ray tracing from 2020. All expose a two-level acceleration structure: a bottom level per mesh, a top level of instances.
7. The shader types DXR defines are ray generation, intersection, any-hit, closest-hit and miss. Traversal itself is fixed function and deliberately not programmable, which is what allows it to be a dedicated circuit.
8. Denoising: the practical breakthrough was Chaitanya and others, "Interactive Reconstruction of Monte Carlo Image Sequences Using a Recurrent Denoising Autoencoder", SIGGRAPH 2017. NVIDIA's Ray Reconstruction, shipped in DLSS 3.5 in September 2023, replaces the hand-written denoiser with a trained network.
9. Honest performance picture, as of 2026. Full path tracing at 4K on the fastest consumer card of the day still typically renders internally at 1080p or lower and relies on both denoising and upscaling to reach the output resolution. Native-resolution real-time path tracing is not a solved problem. Vendors showing "4K path traced" numbers are almost always showing an upscaled image, and independent testers routinely point this out.

22.9.6 WORDS - remember these

1. Ray casting — shoot a line, find what it hits — solving for the nearest surface intersection along a parameterized ray.
2. Bounding volume hierarchy — nested boxes for fast searching — a tree of axis-aligned boxes over primitives, traversed to prune intersection tests.
3. Path tracing — follow random bounces and average — Monte Carlo integration of the rendering equation.
4. Hybrid rendering — rasterise most, trace some — using rasterisation for primary visibility and rays only for selected effects.
5. RT core — the circuit that walks the tree — fixed-function hardware for ray-box and ray-triangle intersection, introduced in 2018.
6. Denoiser — clean up a noisy estimate — a spatial and temporal filter, now usually a neural network, reconstructing an image from few samples.

22.10 Upscaling and frame generation

22.10.1 PLAIN - in simple words

1. Rendering costs roughly scale with pixel count. Going from 1080p to 4K is four times the pixels and roughly four times the cost.

2. So there is an obvious cheat: render at a lower resolution and enlarge the result cleverly.
3. Naive enlargement looks soft and blurry, so for a long time nobody did it.
4. The clever version uses information from previous frames. Slightly shift the camera by a fraction of a pixel each frame, and over several frames you have sampled the scene far more finely than one frame allows.
5. That trick, on its own, is called temporal anti-aliasing. It reuses history to remove jagged edges.
6. Upscalers use the same history but also target a higher output resolution, and the modern ones use a trained neural network to decide how to combine old and new samples.
7. There is a second, more controversial idea: instead of rendering more frames, invent them. Look at two real frames and manufacture the picture in between.
8. That makes the counter say a bigger number and makes motion look smoother, but it cannot make the game respond faster. It usually makes it respond slower.
9. Neither trick is free. Both introduce specific, visible artefacts.

22.10.2 PLAIN - a picture in your head

1. Think of photographing a page of small print in poor light.
2. One photo is grainy and unreadable. But take eight photos, each shifted by a fraction of a millimetre, and stack them, and the text becomes sharp.
3. That is temporal accumulation. You bought detail with time instead of with sensor resolution.
4. Now suppose the page is being pulled away while you shoot. To stack the photos you must first work out how far the page moved between each one, and slide each photo back into place.
5. That correction is called reprojection, and the arrows telling you how far things moved are motion vectors.
6. When a corner of the page was hidden behind your thumb in the earlier photos, there is no history for that region, and the stack has nothing to add.

Where this comparison breaks: a page moves rigidly. A game scene has objects moving in front of each other, transparent surfaces with no motion vectors at all, shadows that move differently from the objects casting them, and particles. Each of these breaks the correction in its own way, which is why upscaler artefacts cluster around fast movement, thin objects, and things seen through glass.

22.10.3 PLAIN - a worked example

1. Output resolution 3840 by 2160, which is 8,294,400 pixels.
2. The standard quality presets and their internal render sizes:

Preset	Scale factor	Internal at 4K
Quality	0.667	2560 x 1440
Balanced	0.58	2227 x 1253
Performance	0.50	1920 x 1080
Ultra Performance	0.33	1280 x 720

3. Performance mode renders 2,073,600 pixels instead of 8,294,400. That is 25 percent of the work for the parts of the frame that scale with resolution.
4. In practice the measured speed-up is smaller, typically 1.6x to 2.2x rather than 4x, because vertex work, shadow map rendering and the upscaler itself do not shrink.
5. Now frame generation, and the latency arithmetic that people get wrong.
6. Suppose the game natively runs at 60 frames per second, one frame every 16.7 milliseconds.
7. To insert a frame between real frames N and N+1, the system must already have rendered N+1. So it holds N+1 back, shows the invented frame first, then shows N+1.

8. The counter now reads 120 frames per second. But the newest real image reaches your eye about one frame later than it would have, plus the time to generate the fake one.
9. Measured end-to-end click-to-photon latency typically rises by 8 to 15 milliseconds with single frame generation, before any latency-reduction feature claws some back.
10. So the honest statement is: frame generation improves smoothness, not responsiveness, and slightly harms responsiveness. It is at its best when the base frame rate is already comfortable, and at its worst below about 45 frames per second where it is most tempting.

22.10.4 PLAIN - what is really happening inside

1. The renderer jitters the projection matrix by a sub-pixel offset each frame, following a low-discrepancy sequence such as Halton.
2. It also outputs a motion vector for every pixel: where that surface point was in the previous frame, in screen coordinates.
3. The upscaler takes this frame's low-resolution colour, its depth, its motion vectors, and the previous output frame.
4. It warps the previous output using the motion vectors, then decides per pixel how much of that warped history to trust.
5. The old approach used hand-written rules: colour clamping to a neighbourhood box, rejecting history that looks too different. It causes ghosting when it trusts too much and flicker when it trusts too little.
6. The new approach feeds all of that into a small neural network trained on matched pairs of low-resolution input and high-quality reference images.
7. Frame generation additionally computes an optical flow field between two real frames, warps both toward the midpoint, and fills disocclusions with a network.
8. The generated frame is never used as history for the next real frame. It is displayed and discarded.

22.10.5 TECHNICAL - the engineer's version

1. Timeline, all verifiable release dates. DLSS 1.0, February 2019, per-game trained, widely judged poor. DLSS 2.0, March 2020, generic temporal network, the version that made the idea work. DLSS 3, October 2022, adds Frame Generation, RTX 40 only. DLSS 3.5, September 2023, adds Ray Reconstruction. DLSS 4, January 2025, replaces the convolutional network with a transformer model and adds Multi Frame Generation on RTX 50.
2. As of August 2026 NVIDIA's current release is DLSS 4.5, with a second generation transformer model and a dynamic multi-frame multiplier advertised up to 6x, meaning five generated frames per rendered frame.
3. AMD FidelityFX Super Resolution: FSR 1.0, June 2021, spatial only. FSR 2.0, May 2022, temporal, open source, runs on competitors' hardware. FSR 3, September 2023, adds frame generation. FSR 4, March 2025, machine learning based and limited to RDNA 4.
4. Intel XeSS launched August 2022 with two code paths: XMX matrix instructions on Intel Arc, and a DP4a integer path elsewhere that is measurably weaker. XeSS 2, December 2024, added frame generation and a low-latency mode.
5. Latency reduction features are separate technologies: NVIDIA Reflex (2020), AMD Anti-Lag, Intel Xe Low Latency. They shorten the render queue. They do not remove the frame-holding cost of frame generation.
6. Honest quality comparison, as of 2026. Independent testing outlets including Digital Foundry and Hardware Unboxed consistently report the transformer-based DLSS as the most stable and detailed at a given preset, FSR 4 as close behind on supported hardware and a large jump over FSR 3, and XeSS as good on Intel hardware and weaker on its fallback path. This is expert consensus from controlled comparison, not a measured standard, and reasonable people disagree about specific titles.
7. The known artefact list, which applies to all of them in some degree: ghosting trails behind fast movement; smearing where an object uncovers background; shimmer on thin geometry such as fences and

wires; loss of fine texture detail at aggressive presets; and, for frame generation, warped heads-up-display elements and text in the generated frames.
8. The cost of running the upscaler itself is real: roughly 1 to 2 milliseconds per frame at 4K output on recent hardware. Below about 40 frames per second this overhead eats a noticeable share of the gain.
9. There is a genuine industry disagreement worth stating. One camp argues these techniques are legitimate reconstruction, no different in kind from mipmaps or texture compression. The other argues they let publishers ship games that cannot run acceptably at native resolution, and that quoting frame rates with generated frames included is misleading. Both positions are held by serious people.

22.10.6 WORDS - remember these

1. Temporal anti-aliasing — reuse earlier frames to smooth edges — accumulation of sub-pixel jittered samples across frames with motion-vector reprojection.
2. Motion vector — where this pixel was last frame — a per-pixel screen-space displacement produced by the renderer for reprojection.
3. Reprojection — slide the old frame into place — warping history buffers by motion vectors before reuse.
4. Ghosting — trails behind moving things — an artefact of trusting stale history whose motion vectors were wrong or missing.
5. Disocclusion — newly revealed area with no history — a region uncovered this frame, which temporal methods must invent.
6. Frame generation — invent a picture between two real ones — interpolating an intermediate frame from optical flow, raising frame counters and latency together.

22.11 Inside a real graphics chip

22.11.1 PLAIN - in simple words

1. A graphics chip is not one big pool of lanes. It is many small identical blocks, each a small computer in its own right.
2. NVIDIA calls a block a streaming multiprocessor. AMD calls it a compute unit. Intel calls it an Xe core. They are the same idea.
3. Each block has its own instruction scheduler, its own arithmetic lanes, its own registers, its own small fast memory, and its own texture unit.
4. The marketing number, “21,760 CUDA cores”, is the total count of arithmetic lanes, not the number of independent computers.
5. The real count of independent computers on that chip is 170.
6. Alongside the general lanes sit two kinds of special circuit: matrix units for machine learning, and ray tracing units for the tree walking of section 22.9.
7. All the blocks share a large cache and a memory controller that talks to the card’s own memory chips.
8. That memory is the real bottleneck of almost every workload, and its speed is the number you should look at first.

22.11.2 PLAIN - a picture in your head

1. Think of a large open-plan office split into 170 identical pods.
2. Each pod has 128 desks, one supervisor per 32 desks, a shared whiteboard, and a shelf of reference books.
3. The whiteboard is shared memory: fast, tiny, and only visible inside the pod.
4. Down the corridor is the building library, shared by all pods. That is the L2 cache.
5. Outside the building is the city archive. That is the card’s memory. Fetching from there takes a long walk, so you send someone and get on with other work.

Where this comparison breaks: the desks in a pod are not independent workers. Groups of 32 share one supervisor who reads out one instruction at a time. The pod is more like a rowing eight than an office.

22.11.3 PLAIN - a worked example

1. Here is a real specification sheet, the NVIDIA GeForce RTX 5090, launched 30 January 2025 at 1,999 US dollars, decoded line by line.

Line on the sheet	Value	What it really means
Architecture	Blackwell	Design generation, 2025
GPU die	GB202	The specific silicon
CUDA cores	21,760	FP32 lanes in total
SM count	170	Independent blocks
Tensor cores	680	4 per SM, matrix units
RT cores	170	1 per SM, tree walkers
Boost clock	2.41 GHz	Peak, not sustained
Memory	32 GB GDDR7	Card's own memory
Memory bus	512-bit	Wires to the memory
Bandwidth	1,792 GB/s	Bytes readable per second
L2 cache	96 MB	Chip-wide shared cache
Transistors	92.2 billion	Complexity
Die size	750 mm2	Physical area of silicon
Interface	PCIe 5.0 x16	Link to the processor
Total board power	575 W	Heat you must remove

2. Now the arithmetic that the sheet does not show you.
3. 21,760 divided by 170 gives 128 FP32 lanes per block. 680 divided by 170 gives 4 matrix units per block.
4. Peak FP32 rate = lanes x 2 (a multiply-add counts as two) x clock = 21,760 x 2 x 2.41 GHz = 104.8 TFLOPS. This matches the published figure exactly, which tells you it is a paper number from a formula, not a measurement.
5. Bandwidth = bus width x per-pin data rate / 8 = 512 x 28 Gbps / 8 = 1,792 GB/s. The 28 gigabits per second per pin is the GDDR7 speed grade used.
6. A 512-bit bus with 32-bit memory chips means 16 chips. 32 GB across 16 chips is 2 GB each. The board layout follows directly from the bus width.
7. Now the ratio that matters most: 104.8 TFLOPS against 1,792 GB/s is about 58 floating-point operations available for every byte you can read.
8. So any calculation doing fewer than 58 operations per byte of data is limited by memory, not by arithmetic. Most real ones are. This ratio is the whole of the roofline model in one line.
9. Power: 575 watts over 750 square millimetres is about 0.77 watts per square millimetre, which is why these cards carry three-slot coolers and vapour chambers.

22.11.4 PLAIN - what is really happening inside

1. Inside one block, the lanes are split into four partitions of 32. Each partition has one warp scheduler that issues one instruction per cycle to its own group.
2. The register file is enormous by processor standards: 256 KB per block on recent NVIDIA parts, so more register storage than L1 cache.
3. That is deliberate. Registers are what let dozens of thread groups stay resident so the scheduler always has someone ready to run.
4. Shared memory is a scratchpad the programmer controls, not a cache. On recent NVIDIA consumer parts it shares a 128 KB pool with L1, split as you choose.
5. Tensor cores do not do one multiply. They do a whole small matrix multiply per instruction, for example a 16x8 by 8x16 tile, accumulating into a result.
6. RT cores do box and triangle intersection while the general lanes are free to do other work.

7. The memory controller is split into many channels. Each GDDR chip has its own path, and requests from many blocks are coalesced and reordered to keep every channel busy.
8. The PCIe link is not for the drawing. It is for uploading geometry and textures, sending commands, and reading results back.

22.11.5 TECHNICAL - the engineer's version

1. Memory technologies and real bandwidth, all from published specifications:

Part and memory	Bus width	Bandwidth
Arc B580, GDDR6 19 Gbps	192-bit	456 GB/s
RX 9070 XT, GDDR6 20 Gbps	256-bit	640 GB/s
RTX 5080, GDDR7 30 Gbps	256-bit	960 GB/s
RTX 5090, GDDR7 28 Gbps	512-bit	1,792 GB/s
H100 SXM, HBM3	5120-bit	3,350 GB/s
B200, HBM3e	wide stacks	about 8,000 GB/s

2. JEDEC published the GDDR7 standard, JESD239, in March 2024. It is the first JEDEC DRAM standard to use PAM3 signalling, three voltage levels carrying 1.5 bits per symbol, and it specifies up to 32 Gb/s per pin.
3. JEDEC published HBM4, JESD270-4, in April 2025, with revision 4A in December 2025. It doubles the per-stack interface to 2048 bits, runs up to 8 Gb/s per pin, reaches up to 2 TB/s per stack, and allows up to 64 GB per stack with 16-high 32 Gb dies.
4. HBM is stacked vertically on an interposer beside the die, which is why it is confined to expensive datacentre parts. GDDR is separate packages on the board, which is why it is on everything else.
5. Naming honesty. A "CUDA core" is one FP32 lane inside a SIMD datapath with no independent program counter. AMD's "stream processor" is the same thing. Neither is a core in the sense used for a processor. This is marketing vocabulary that stuck.
6. TFLOPS honesty. AMD publishes 48.7 TFLOPS FP32 for the RX 9070 XT, which has 4,096 stream processors at up to 2.97 GHz. The plain formula gives 24.3. The published figure counts dual-issue FP32, which only applies when the compiler can pair instructions. NVIDIA's 104.8 TFLOPS for the RTX 5090 uses the plain formula. Comparing the two numbers directly is not valid.
7. PCIe 5.0 x16 has a raw signalling rate of 32 GT/s per lane, giving a theoretical 63 GB/s each way after 128b/130b encoding, and roughly 50 to 55 GB/s achievable in practice.
8. Power delivery on high-end 2025 cards uses the 12V-2x6 connector, the revised form of 12VHPWR, rated at 600 watts. The original 12VHPWR connector had well-documented melting failures on RTX 4090 cards in 2022, traced to incomplete insertion and uneven current sharing between pins.
9. Tools: `nvidia-smi -q` dumps clocks, power, temperature and memory; `nvtop` and `radeontop` give live views; Nsight Compute reports achieved occupancy, memory throughput as a percentage of peak, and the exact stall reasons per kernel.

22.11.6 WORDS - remember these

1. Streaming multiprocessor — one pod of lanes — NVIDIA's independent scheduling and execution block; AMD's compute unit, Intel's Xe core.
2. CUDA core — one arithmetic lane — an FP32 lane within a SIMD datapath, not an independently scheduled core.
3. Tensor core — the matrix multiplier — a fixed-tile matrix multiply-accumulate unit, first shipped on NVIDIA Volta in 2017.
4. Shared memory — the pod's whiteboard — programmer-managed on-chip scratchpad shared by a workgroup.

5. GDDR — fast memory chips on the board — graphics DDR DRAM; GDDR7 uses PAM3 and reaches 32 Gb/s per pin under JESD239.
6. HBM — memory stacked next to the die — high bandwidth memory on an interposer; HBM4 under JESD270-4 reaches 2 TB/s per stack.
7. Roofline — the arithmetic-per-byte break-even — the ratio of peak compute to peak bandwidth that decides whether a kernel is compute or memory bound.

22.12 The software stack

22.12.1 PLAIN - in simple words

1. Your program never talks to the graphics chip directly. It talks to a library with a defined set of functions, called a graphics API.
2. Behind that library sits the **driver**, written by the chip maker, which turns those calls into the chip's own commands.
3. Old APIs let you say "draw this" one thing at a time, and the driver did an enormous amount of guessing and bookkeeping.
4. New APIs make you state everything up front, in explicit objects, and then record long lists of commands yourself.
5. That is more work for the programmer and much less guessing for the driver, which means less overhead and more predictable timing.
6. Commands are not executed as you write them. They are recorded into a buffer, submitted in a batch, and executed later by the chip.
7. Because the chip runs behind the processor, at any moment one of them is waiting for the other. Which one is waiting decides how you make the program faster.
8. Shaders must be turned into the chip's real machine code at some point. If that happens the first time you enter a new area, the game stutters.

22.12.2 PLAIN - a picture in your head

1. Think of ordering from a kitchen through a waiter.
2. The old API is a waiter who takes one instruction at a time, remembers the whole table's preferences, and silently fixes contradictions.
3. That waiter is convenient and unpredictable. You never know when they will disappear to sort something out.
4. The new API makes you write the entire order on a card, in full, and hand it over. The waiter only carries cards.
5. It is more effort. But nothing surprising happens, and several people can write cards at the same time.

Where this comparison breaks: the kitchen is always about one full order behind. By the time your card is being cooked you are already writing the next one. If you ever ask "is my food ready yet" and wait for the answer, both of you stop. That is what a synchronization stall costs, and it is the single most common performance mistake in graphics programming.

22.12.3 PLAIN - a worked example

1. A frame is CPU-bound or GPU-bound. Here is how to tell, with numbers.
2. Measure two things: how long the processor takes to prepare the frame, and how long the chip takes to draw it.

```
case A   CPU 12.0 ms  |============|
         GPU  6.0 ms  |======|              -> CPU-bound, 83 fps
case B   CPU  4.0 ms  |====|
         GPU 15.0 ms  |===============|     -> GPU-bound, 67 fps
```

3. In case A, buying a faster graphics card changes nothing. The chip already finishes early and waits.
4. In case A, lowering the resolution also changes nothing, because resolution costs the chip, not the processor.
5. What helps in case A: fewer draw calls, fewer objects, better batching, a lower-overhead API.
6. In case B, resolution and shader quality are exactly the right dials.
7. A quick field test: drop the resolution by half. If the frame rate barely moves, you are CPU-bound.

22.12.4 PLAIN - what is really happening inside

1. The API validates your calls, tracks state, and hands work to the driver.
2. The driver contains a shader compiler that turns the portable intermediate form into the specific chip's instructions, and a command translator that builds the hardware command packets.
3. Those packets go into a ring buffer in memory. The chip reads the ring buffer independently through direct memory access.
4. Work is submitted to **queues**. A modern chip has separate queues for graphics, for compute, and for copying, and they can run at the same time.
5. Because they run at the same time, you need explicit fences and barriers to say "this must finish before that starts". Getting those wrong causes corruption that only appears on some hardware.
6. Shader compilation is the classic stutter cause. A game may contain tens of thousands of shader variants. Compiling one takes a few milliseconds to a few hundred. Doing it at the moment an effect first appears drops a frame.
7. The fix is to compile everything ahead of time and cache it, which is why many games now show a "compiling shaders" screen on first run or after a driver update.
8. A driver update invalidates the cache, because the generated machine code may change. That is why the stutter returns after updating a driver.

22.12.5 TECHNICAL - the engineer's version

1. API history, with dates. OpenGL 1.0 from Silicon Graphics in 1992. Direct3D in 1996. OpenGL 2.0 with GLSL in 2004. Direct3D 11 in 2009. AMD Mantle in 2013, which was donated and became the basis of Vulkan. Apple Metal in June 2014. Direct3D 12 with Windows 10 in July 2015. Vulkan 1.0 on 16 February 2016. Vulkan 1.4 in December 2024. WebGPU shipped in Chrome 113 in May 2023 and its W3C specification was still at Candidate Recommendation stage in January 2026.
2. What changed with the low-level APIs, precisely: pipeline state is baked into immutable objects instead of being set piecemeal; resource lifetime and residency become the application's job; memory is allocated and sub-allocated by the application; synchronization is explicit; and command buffers can be recorded from many threads at once.
3. The measured payoff is draw call overhead. A Direct3D 11 draw call costs roughly 1 to 10 microseconds of processor time depending on how much state changed with it; Direct3D 12 and Vulkan reduce this to well under 1 microsecond and let the cost be spread across threads.
4. The cost is code volume. A minimal triangle in OpenGL is about 100 lines. In Vulkan it is 800 to 1,500. This is a real and widely acknowledged trade, and it is why most people use an engine rather than the API directly.
5. Pipeline state objects are the mechanism behind first-run stutter. Direct3D 12 added Pipeline State Object libraries and Vulkan added pipeline caches; both Vulkan's `VK_EXT_graphics_pipeline_library` (2022) and Direct3D 12's state object work aim to reduce combinatorial explosion. Valve's Steam Deck distributes precompiled pipeline caches to users precisely for this reason.
6. Queue families in Vulkan are advertised by the device. A typical discrete card exposes one graphics-and-compute queue family, one or more compute-only, and one transfer-only queue using the copy engines.
7. Diagnosis tools: RenderDoc for frame capture and per-draw timing; PIX on Windows; Nsight Systems and Radeon GPU Profiler for timeline views showing exactly where the processor and the chip wait for each other; `VK_LAYER_KHRONOS_validation` for correctness.

22.12.6 WORDS - remember these

1. Graphics API — the agreed set of drawing functions — the specified interface between application and driver, such as Vulkan or Direct3D 12.
2. Driver — the translator to real hardware — vendor software containing the shader compiler, memory manager and command packet builder.
3. Command buffer — a written list of orders — a recorded sequence of GPU commands, submitted to a queue for later execution.
4. Queue — a lane of work into the chip — an independent submission channel; graphics, compute and transfer queues can run concurrently.
5. Pipeline state object — the whole draw configuration frozen — an immutable object bundling shaders, blend, depth and raster state, compiled once.
6. CPU-bound — the processor is the limit — frame time dominated by host-side preparation, unaffected by resolution.
7. GPU-bound — the chip is the limit — frame time dominated by device execution, responsive to resolution and shader cost.

22.13 Graphics chips for everything else

22.13.1 PLAIN - in simple words

1. For thirty years these chips did one thing. Then people noticed the shape of the work, not the subject of the work, was what suited them.
2. Anything made of millions of independent, identical, arithmetic-heavy steps fits. Weather models, molecular simulation, oil exploration, cryptography and, above all, neural networks.
3. Before 2007 you had to disguise your problem as a drawing problem. People really did pack scientific data into textures and read the answer out of a picture.
4. In 2007 NVIDIA released CUDA, which let you write ordinary C-like code for the chip with no pretence of drawing. That is the moment the field opened.
5. Speed is quoted in FLOPS, floating-point operations per second. A TFLOP is a trillion of them per second.
6. But the number depends entirely on how precise the numbers are. Halve the precision and the same silicon roughly doubles the rate.
7. And the FLOPS number is usually not what limits you. Memory bandwidth is. Most real work spends its life waiting for data.
8. The chips sold for datacentres and the chips sold for games are the same family with very different memory, very different precision support, and very different prices.

22.13.2 PLAIN - a picture in your head

1. Think of a factory that can assemble 100,000 items per hour.
2. The loading bay can only bring in enough parts for 20,000 items per hour.
3. The factory's rated capacity is a fiction. You will build 20,000 items per hour, and buying faster assembly robots will change nothing.
4. The loading bay is memory bandwidth. The robots are FLOPS.
5. The only real fixes are a bigger loading bay, or redesigning the product so each part gets used in more items before it is thrown away.

Where this comparison breaks: you can genuinely redesign the work. Reusing data that is already on the chip, by tiling a matrix multiply so each loaded block feeds many operations, is the single most important optimization in the field, and it is why a well-written matrix multiply reaches near peak while a naive one reaches a few percent.

22.13.3 PLAIN - a worked example

1. Real throughput of one NVIDIA H100 SXM, at each precision, dense (that is, not counting the sparsity doubling that vendors like to quote):

Precision	Dense throughput
FP64 tensor	67 TFLOPS
FP32 vector	67 TFLOPS
TF32 tensor	494 TFLOPS
FP16 / BF16 tensor	989 TFLOPS
FP8 tensor	1,979 TFLOPS

2. From FP64 to FP8 is a factor of about 30 on the same chip. Nothing changed except how many bits each number uses.
3. This is why the AI field moved to lower precision so aggressively. It is not a small saving.
4. Now the bandwidth check. The H100 SXM has 3,350 GB/s. At FP16 it can do 989 trillion operations per second.
5. That is 989e12 / 3350e9 = about 295 operations for every byte read. Any calculation with a lower ratio than that is bandwidth-limited.
6. Generating one token from a 70-billion-parameter model in 16-bit precision requires reading roughly 140 gigabytes of weights, and does about two operations per weight. That is 1 operation per byte. Hopelessly bandwidth-limited.
7. At 3,350 GB/s, reading 140 GB takes about 42 milliseconds. That is the physical floor for one token on one such chip, no matter how fast its arithmetic is.
8. A gaming card against a datacentre accelerator, same generation family:

Item	RTX 5090	B200 SXM
Memory	32 GB GDDR7	180 GB HBM3e
Bandwidth	1.79 TB/s	about 8 TB/s
FP4 tensor, sparse	about 3.4 PFLOPS	18 PFLOPS
Board power	575 W	up to 1,000 W
Launch price	1,999 US dollars	not sold singly

22.13.4 PLAIN - what is really happening inside

1. A general-purpose program for a graphics chip is called a kernel. You launch a grid of thread blocks; each block runs on one streaming multiprocessor.
2. The programming model is deliberately the same as a compute shader. Same hardware, different front door.
3. Data must be copied to the card's memory first, unless the system has a shared memory design. That copy crosses PCIe and is often the slowest part.
4. Matrix multiply is the workhorse. The efficient version splits the matrices into tiles that fit in shared memory, loads each tile once, and reuses it across many multiply-accumulate operations.
5. Tensor cores make this even more extreme: one instruction consumes a whole tile and produces a whole tile of results.
6. Lower precision helps twice: the arithmetic units go faster, and the same bytes of bandwidth carry twice as many numbers.
7. That second effect is often the larger one, which is exactly the point about bandwidth being the true limit.

22.13.5 TECHNICAL - the engineer's version

1. Timeline. NVIDIA announced CUDA with the G80 architecture in November 2006; the public toolkit shipped in 2007. Khronos released OpenCL 1.0 in December 2008, initiated by Apple. Both are still maintained; CUDA has by far the larger ecosystem, which is a business fact as much as a technical one.
2. The scientific computing shift came first. GPU-accelerated entries appeared in the TOP500 supercomputer list from 2008 onward, and Tianhe-1A took the number one spot in November 2010 using NVIDIA Tesla parts.
3. The machine learning shift is usually dated to AlexNet, Krizhevsky, Sutskever and Hinton, which won the ImageNet competition in 2012 trained on two GeForce GTX 580 cards with 3 GB each. That result is established fact and is the hinge of the modern field.
4. Precision formats in use, all standards or de facto standards: IEEE 754 binary 64 and binary 32; IEEE 754 binary16; bfloat16, from Google Brain, which keeps FP32's exponent range with 8 fewer mantissa bits; TF32, NVIDIA's 19-bit internal tensor format from 2020; FP8 in the E4M3 and E5M2 variants, standardized by an Arm, Intel and NVIDIA joint proposal in 2022; and FP4 from Blackwell in 2024.
5. Marketing claim, flagged as such: every headline TOPS figure from every vendor in this space is quoted at the lowest supported precision, with structured sparsity assumed. NVIDIA's 3,352 AI TOPS for the RTX 5090 is FP4 with sparsity. Dividing by four gives a fairer comparison against an older FP16 figure.
6. Active research, flagged as such: whether FP4 and lower training is generally viable rather than viable for specific layers is not settled as of 2026. Inference at 4 bits with careful quantization is established practice; 4-bit training is not.
7. The roofline model, from Williams, Waterman and Patterson in 2009, is the standard tool: plot achievable performance against arithmetic intensity, with a slanted bandwidth roof and a flat compute roof. Nsight Compute draws it for you per kernel.
8. Chapter 46 takes this forward into what a neural network actually is and why these numbers govern what can be trained and served.

22.13.6 WORDS - remember these

1. FLOPS — sums per second — floating-point operations per second; TFLOPS is 10^12 and PFLOPS is 10^15.
2. Kernel — a program launched over a grid — a device function executed by many threads organized into blocks.
3. Arithmetic intensity — sums per byte fetched — the ratio of operations to bytes moved, which determines the binding limit under the roofline model.
4. Sparsity — assume half the weights are zero — NVIDIA's 2:4 structured sparsity, which doubles quoted tensor throughput when applicable.
5. bfloat16 — 16 bits with FP32's range — a truncated FP32 with 8 exponent bits and 7 mantissa bits, from Google Brain.
6. CUDA — the first non-graphics door into the chip — NVIDIA's proprietary parallel computing platform and C-like language, public from 2007.

22.14 Integrated, discrete and mobile chips

22.14.1 PLAIN - in simple words

1. A **discrete** graphics card is a separate board with its own memory chips and its own power supply, plugged into the processor over a link.
2. An **integrated** graphics chip is built into the same package as the processor and has no memory of its own. It borrows the system's memory.
3. That borrowing is the whole story of integrated graphics. System memory is several times slower than a graphics card's memory.
4. Apple's approach is different: one large pool of fast memory that both the processor and the graphics chip address directly, with no copying.
5. Phone and tablet chips use a further trick. Instead of drawing the whole screen at once, they cut it into small tiles and finish each tile completely in tiny on-chip memory.
6. That saves enormous amounts of memory traffic, which saves power, which is why a phone can render a detailed scene on a battery.
7. A phone chip can do almost everything a desktop chip can do, including ray tracing on recent models. What it cannot do is sustain it, because of heat.

22.14.2 PLAIN - a picture in your head

1. Imagine painting a large mural.
2. A desktop chip paints the whole wall at once, walking back to the paint store for every colour.
3. A phone chip masks off one square metre, keeps that square's paints on a small tray at hand, finishes it completely, then moves the tray to the next square.
4. Far fewer trips to the store. The store is memory, and every trip costs battery.

Where this comparison breaks: the tile has to know which parts of the mural touch it before it starts, so all the shapes must be sorted into tiles first. That sorting step is real work and real memory traffic, and it is why tile-based designs do best on scenes with moderate geometry and lose their advantage as triangle counts explode.

22.14.3 PLAIN - a worked example

1. Memory bandwidth, the number that decides everything here:

System	Memory	Bandwidth
Desktop, DDR5-6000	System DDR5	96 GB/s
Apple M5	Unified LPDDR5X	153.6 GB/s
Apple M5 Max, 40-core	Unified LPDDR5X	614 GB/s
Intel Arc B580	12 GB GDDR6	456 GB/s
RTX 5090	32 GB GDDR7	1,792 GB/s

2. An integrated chip on a normal desktop shares that 96 GB/s with the processor, which is also using it. In practice it sees perhaps 60 to 70 GB/s.
3. That is about 4 percent of an RTX 5090's bandwidth. No amount of arithmetic capability rescues that.
4. Apple's M5 Max at 614 GB/s is in the same range as a mid-range discrete card, and unlike a discrete card it can address up to 128 GB of it.
5. That combination is why Apple laptops became popular for running large models locally: not raw speed, but a large pool of reasonably fast memory with no copying.

22.14.4 PLAIN - what is really happening inside

1. Integrated graphics carve a region out of system memory. Some is reserved at boot, more is allocated on demand. The processor and graphics chip may share a last-level cache, which helps.
2. Unified memory on Apple silicon means one physical pool and one address space. A buffer written by the processor is visible to the graphics chip with no copy and no PCIe transfer.
3. Tile-based deferred rendering, the mobile approach, works in two passes. The first pass runs all the vertex work and records which triangles fall in which tile, writing that list to memory.
4. The second pass takes one tile at a time, typically 16x16 or 32x32 pixels, loads nothing, rasterises every triangle in that tile's list into on-chip memory, and writes only the final colours out.
5. Because the depth buffer for a tile lives on-chip, depth traffic to main memory is almost eliminated. So is blending traffic.
6. "Deferred" here refers to deferring fragment shading until visibility within the tile is resolved, so hidden surfaces are never shaded at all.
7. What a phone cannot do is sustain power. A phone chip may draw 10 watts for a few seconds and then must fall back to 3 to 5 watts or the case becomes too hot to hold. Frame rates fall accordingly.

22.14.5 TECHNICAL - the engineer's version

1. Tile-based rendering came from Imagination Technologies' PowerVR, whose first parts shipped in 1996 and which powered the Sega Dreamcast in 1998 and every iPhone until Apple's own designs from the A11 in 2017.
2. Arm Mali and Qualcomm Adreno both use tiling; Adreno can also switch to immediate mode per render pass, which Qualcomm calls FlexRender.
3. Vulkan and Metal expose tiling directly through render pass load and store operations. Declaring `LOAD_OP_DONT_CARE` and `STORE_OP_DONT_CARE` correctly is the single biggest mobile optimization, because it tells the driver a tile never needs to be read from or written to main memory.
4. Apple silicon current figures, from Apple's own specifications: M5, launched 15 October 2025, has an 8 or 10 core GPU with a neural accelerator in each core and 153.6 GB/s of unified bandwidth. M5 Pro and M5 Max followed on 3 March 2026, with up to 20 and up to 40 GPU cores, and 307 GB/s and 614 GB/s respectively.
5. AMD's integrated designs in handheld consoles, and Apple's designs, blur the old boundary: both are integrated in the strict sense yet perform like entry-level to mid-range discrete parts.
6. What a phone GPU genuinely cannot do, as of 2026: hold large working sets, because bandwidth and thermal budget both punish it; sustain high clocks for more than a few minutes; and match the fixed-function ray tracing throughput of a 300-watt desktop part, though recent Arm, Qualcomm, Imagination and Apple designs all have hardware ray tracing.
7. Tools: Arm Performance Studio and Qualcomm Snapdragon Profiler report tile counts, overdraw and bandwidth per render pass; Xcode's Metal debugger shows per-tile memory traffic on Apple silicon.

22.14.6 WORDS - remember these

1. Discrete GPU — a separate card with its own memory — a device with dedicated VRAM connected over PCIe.
2. Integrated GPU — built into the processor package — a device sharing system DRAM and its bandwidth with the host.
3. Unified memory — one pool, no copying — a single physical memory and address space shared coherently by CPU and GPU, as on Apple silicon.
4. Tile-based deferred rendering — finish one small square at a time — binning primitives into screen tiles and shading each tile entirely in on-chip memory.
5. Binning pass — sorting shapes into squares — the first pass of a tiled renderer, producing per-tile primitive lists.

6. Thermal throttling — slowing down to stay cool — clock reduction driven by sustained power and temperature limits, dominant on mobile devices.

22.98 Common wrong ideas

1. Wrong: a GPU is just a faster CPU. Right: it is a slower processor repeated thousands of times. Single-thread performance on a GPU lane is several times worse than on a modern CPU core; the win is entirely in parallel count.
2. Wrong: 21,760 CUDA cores means 21,760 independent processors. Right: it means 21,760 arithmetic lanes across 170 independently scheduled blocks, with groups of 32 lanes forced to execute the same instruction.
3. Wrong: more TFLOPS means a faster card. Right: TFLOPS is a formula, not a measurement, vendors count it differently, and most real workloads are limited by memory bandwidth long before they run out of arithmetic.
4. Wrong: the fourth number in a 4x4 matrix is a mathematical trick with no meaning. Right: it is a real projective coordinate, and the division by it is literally what produces perspective.
5. Wrong: a normal map adds geometric detail. Right: it only changes the direction used in lighting. Silhouettes stay perfectly flat, and the illusion collapses at grazing angles.
6. Wrong: ray tracing replaced rasterisation. Right: every shipping game in 2026 rasterises primary visibility and uses rays for selected effects. Full path tracing runs at reduced internal resolution and depends on denoising.
7. Wrong: frame generation makes a game more responsive. Right: it raises the frame counter and smoothness while increasing click-to-photon latency by roughly 8 to 15 milliseconds, because a real frame must be held back.
8. Wrong: upscaling from a lower resolution always looks worse than native. Right: because native rendering has no anti-aliasing history, a good temporal upscaler often resolves fine detail and edges better than native with simple anti-aliasing, while being worse in fast motion.
9. Wrong: the depth buffer stores distance in metres. Right: it stores a non-linear function of distance, which is why precision is concentrated near the camera and why reversed-Z with a float buffer is a large improvement.
10. Wrong: shader stutter means the game is badly optimized in general. Right: it usually means shader machine code is being generated on first use, which precompilation and a persistent cache fix without changing anything else.

22.99 Chapter summary in 20 lines

1. A screen is millions of independent small calculations per frame, which is a different shape of problem from a single fast instruction stream.
2. A processor is latency-optimised: a few clever cores with caches, branch prediction and reordering, built to finish one task quickly.
3. A graphics chip is throughput-optimised: thousands of simple lanes that hide memory waiting by switching to another ready group of threads.
4. Threads run in fixed groups, warps of 32 on NVIDIA and wavefronts of 32 or 64 on AMD, sharing one instruction at a time; divergence costs real time.
5. All drawing ends in a framebuffer. Lines come from integer stepping algorithms such as Bresenham's from 1965, and transparency from the Porter-Duff over operator of 1984.
6. Three-dimensional shapes are meshes of triangles with per-vertex normals, because three points are always flat, always convex, and always interpolatable.
7. A vertex travels through model, world, view, clip, normalized device and screen space, one matrix multiply per step plus one division by w.

8. Four-by-four matrices with homogeneous coordinates, an idea from Möbius in 1827, are used because a 3x3 matrix cannot express translation or perspective.
9. That same matrix multiply, on larger matrices, is exactly what a neural network layer computes, which is why one chip serves both fields.
10. The rasterisation pipeline has programmable stages, vertex, tessellation, geometry and fragment, separated by fixed-function clipping, divide, viewport transform, triangle setup, rasterisation, depth test and blending.
11. Shaders are compiled twice: once to a portable form such as SPIR-V, once by the driver into the installed chip's own machine code.
12. Textures supply detail, and mipmaps exist to prevent aliasing, not to save memory; block compression is what makes real texture budgets possible.
13. The depth buffer, from Catmull in 1974, makes draw order irrelevant, and its finite precision is the sole cause of z-fighting.
14. Lighting moved from Phong in 1975 and Blinn in 1977 to physically based models parameterized by albedo, metalness and roughness, following Disney's 2012 formulation.
15. Ray tracing answers the opposite question from rasterisation, needs a bounding volume hierarchy to be tractable, and became real time only with dedicated hardware from 2018.
16. Path tracing at real-time rates produces a noisy image that a neural denoiser reconstructs; the denoiser is load-bearing, not cosmetic.
17. Upscaling and frame generation trade image quality and latency for pixel throughput; the trades are real, measurable and worth stating honestly.
18. Inside a chip: independent blocks with their own schedulers, registers, shared memory and texture units, sharing an L2 cache and a wide memory controller feeding GDDR7 or HBM.
19. The ratio of peak arithmetic to peak bandwidth, about 58 operations per byte on an RTX 5090, decides whether any given program is compute or memory bound.
20. Integrated chips are limited by shared system bandwidth, Apple's unified memory sidesteps copying entirely, and mobile chips save power by finishing the picture one small on-chip tile at a time.

ABOUT THE AUTHOR

Shikhar Singh

Founder & Chairman

KedByte Technologies Private Limited

Shikhar Singh is the Founder & Chairman of KedByte Technologies Private Limited.

He wrote this book to solve a problem he kept meeting: capable people concluding they were not technical, when in truth nobody had ever shown them the bottom of the stack. His conviction is that computing has no genuinely hard idea in it, only a very long chain of simple ones, and that anybody who is walked along that chain in order can hold the whole of it in their head.

That conviction shapes the method used here. Explain it plainly, with a comparison and a worked example, so that it lands. Then explain it again in the exact professional vocabulary, with the real units and the real figures, so that it is usable. Say clearly where the plain version stops being true. Never let a reader carry away a confident misunderstanding.

Under his direction, KedByte Technologies Private Limited treats technical education as infrastructure rather than as a product feature: knowledge that should be transferable, verifiable, and free of the vocabulary barriers that keep capable people out of the field.

ABOUT THE PUBLISHER

KedByte Technologies Private Limited

KedByte Technologies Private Limited publishes technical work under the KedByte Press imprint.

The editorial standard applied to this volume is simple to state and difficult to meet. Every explanation must work twice: once for a reader who has never opened a terminal, and once for an engineer who needs the exact figure. Every claim that can be checked must be checked. Every figure that will go stale must carry a date. Where the field genuinely disagrees, both positions are given rather than one being quietly chosen.

Where this book uses real evidence, it uses it unaltered. The network trace in Part F, the failed pushes in Part G and the error messages quoted throughout are reproduced exactly as they were observed. Diagnosis is taught against ambiguous real data, because that is the only kind there is.

KEDBYTE

TECHNOLOGIES PRIVATE LIMITED

COLOPHON

This first edition of *How Computers Work* runs to nine parts and fifty-three chapters, and was set entirely from plain text.

The body text is set in TeX Gyre Pagella, a digital revival of Hermann Zapf's Palatino, chosen for long-form reading. Headings, tables and the KedByte identity are set in Poppins. Code, command output and diagrams are set in DejaVu Sans Mono, and every diagram in this book is drawn in plain ASCII so that it survives being copied anywhere.

The book was composed with pandoc and typeset with XeLaTeX on A4 stock. The single accent colour used on rules, chapter numbers and section headings is KedByte navy.

Shikhar Singh

Founder & Chairman, KedByte Technologies Private Limited

First Edition • KedByte Press