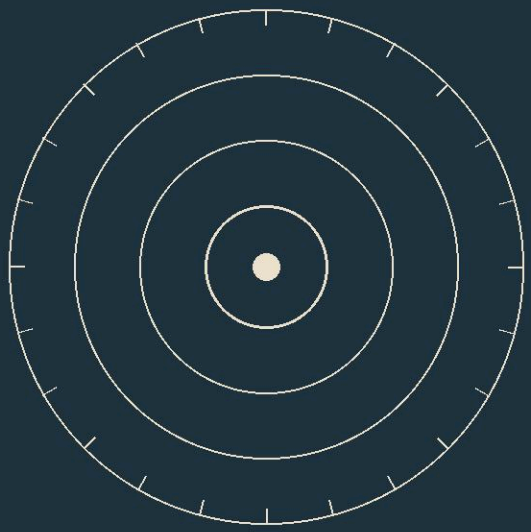
INDUSTRIES OF INDIA

THE OPERATING SYSTEM

Supervisor, Kernel, Ghost



KETINENI RAMARAO

THE OPERATING SYSTEM

Supervisor, Kernel, Ghost

Reference numbers in square brackets resolve in the companion volume, The Operating System: Source Companion.

An Industries of India study. Research and information, not investor news.
Author: Ketineni Ramarao.
Illustrations in this volume are AI-generated editorial artwork; charts are drawn from the sources cited in the companion volume.

Contents

PART I - THE PROBLEM OF THE MACHINE **4**

Chapter 1 - The program that runs the machine 4

Chapter 2 - The machine room problem 6

Chapter 3 - The first supervisors 9

Chapter 4 - The bet: one architecture to replace them all 10

PART II - THE BET **14**

Chapter 5 - OS/360: the tar pit 14

Chapter 6 - Unbundling: software becomes a thing you can sell 16

Chapter 7 - The computer learns to share 17

Chapter 8 - Multics: the cathedral 20

PART III - SHARING THE MACHINE **22**

Chapter 9 - Unix: built on a spare machine 23

Chapter 10 - C and the portable machine 25

PART IV - A BOSS FOR EVERY DESK **27**

Chapter 11 - The computer shrinks to a desk 27

Chapter 12 - The seventy-five-thousand-dollar accident 29

Chapter 13 - DOS, the toll booth 31

Chapter 14 - The friendly face 33

Chapter 15 - Windows climbs on top of DOS 35

Chapter 16 - NT: an operating system built like a mainframe 36

PART V - THE FREE STACK **39**

Chapter 17 - GNU: freedom as method 39

Chapter 18 - MINIX and the teaching kernel 41

Chapter 19 - Linux: “just a hobby” 42

Chapter 20 - The great debate 44

Chapter 21 - Distributions and the Red Hat business 46

Chapter 22 - The quiet conquest 48

PART VI - THE POCKET MAINFRAME **51**

Chapter 23 - Before the smartphone 51

Chapter 24 - Symbian’s peak, and its ceiling 53

Chapter 25 - iPhone: OS X in a pocket 55

Chapter 26 - Android: the best deal ever 57

Chapter 27 - The Java wars 59
Chapter 28 - The losers 61
Chapter 29 - The tollbooth’s reckoning 63

PART VII - THE INVISIBLE MAJORITY 66

Chapter 30 - Real time: the operating systems that cannot be late 66
Chapter 31 - The car’s hundred computers 68
Chapter 32 - Everywhere else: the operating system’s long tail 70

PART VIII - HOW THE BOSS WORKS 73

Chapter 33 - The kernel and the rings 73
Chapter 34 - Whose turn: the scheduler 75
Chapter 35 - Remembering everything: virtual memory 76
Chapter 36 - Everything is a file 78
Chapter 37 - The architecture argument, settled by practice 80

PART IX - THE BUSINESS OF BEING THE BOSS 83

Chapter 38 - The antitrust arc 83
Chapter 39 - Consolidation: what the buyers actually bought 85
Chapter 40 - Cracks in the foundation 86

PART X - WHAT COMES NEXT 89

Chapter 41 - Memory safety and the Rust wager 89
Chapter 42 - The cloud ate the operating system 91
Chapter 43 - Forks, nations and silicon 94
Chapter 44 - The ghost in everything 96
Epilogue - The dependence audit 98
GLOSSARY 99
APPENDIX A - TIMELINE 101
APPENDIX B - THE FAMILY TREE 102
APPENDIX C - THE MARKET IN NUMBERS 103

PART I - THE PROBLEM OF THE MACHINE

Every industry begins with a queue. Before the operating system existed, the queue was physical - people and punch cards waiting for the most expensive machine in the building. The machine cost more per hour than the people around it earned per month, and it spent much of its day idle while human beings walked cards from desk to reader. The first operating systems were not written for elegance. They were written because idle time on a two-million-dollar computer was a scandal with a budget line, and the scandal grew worse with every faster machine: the quicker the computer, the larger the fraction of its day disappeared into the gaps between jobs.

This part follows the decade in which the machine learned to manage its own queue. It begins with the batch monitors - GM-NAA I/O at General Motors in 1956, often cited as the first operating system in the modern sense, and the SHARE and IBSYS systems that followed on IBM’s 709 and 7090 lines. These were operations tools rather than designed software: a resident monitor that loaded one job after another, kept the accounting records the billing department needed, and took back control when each job ended. Out of that modest trick came the entire vocabulary of the field - the supervisor, the job, the queue, the system call. And out of the economics behind it came the industry: by 1964 IBM was ready to stake its existence on one architecture and one operating system for every customer it had, a bet whose cost overruns would become the founding parable of software engineering. The machine room is where the ghost first got its name.

Chapter 1 - The program that runs the machine

Every tap you make on a phone ends the same way: a program asks the operating system to do something it is not allowed to do itself. The weather app cannot read the network card. The game cannot write to the screen. The map cannot listen to the GPS chip. Each of them sends a request down to a layer of software that sits between all the applications and all the hardware, and that layer - the operating system - checks the request, schedules it, fulfils it, and hands the result back, millions of times a second, on every computing device on earth.

The scale of that arrangement is worth pausing on. Google announced in 2021 that more than three billion devices were running Android, its mobile operating system [O1]. Apple told investors in January 2025 that its installed base had passed 2.35 billion active devices, every one of them running Apple’s operating systems [O2]. Of the desktop and laptop computers that browsed the web in late 2026, about 77 percent ran Microsoft’s Windows, with Apple’s Mac systems at roughly 16 percent, Linux at about 6 percent and ChromeOS around 2 percent [O3]. On phones and tablets the picture inverts: Android carried about 72 percent of mobile web usage in October 2026 and Apple’s iOS about 28 percent [O4]. Behind the screens, the pattern inverts again: of all the websites whose server operating system could be identified in November 2026, about 92 percent ran on Unix-family systems, overwhelmingly Linux, and about 8 percent on Windows [O5]. And at

the extreme end of computing, every one of the five hundred fastest supercomputers on the TOP500 list has run Linux since November 2017 [O6].

No other manufactured thing has this reach. More people carry an operating system in their pocket today than own a bicycle, a car, or a banknote of their own national currency. Yet the operating system is invisible by design. Nobody buys a phone to admire its scheduler. The better the layer does its job, the less anyone thinks about it, and so the most widely used category of software ever built is also the least noticed. This book is about that layer: where it came from, what it actually does, who builds it, who profits from it, what constrains it, and where it is going.

Two descriptions from the textbooks frame everything that follows. In the first, the operating system is a resource manager. A computer is a bundle of scarce things - processor time, memory, storage, network capacity, screens, sensors - and many programs want them at once. Someone has to decide who gets the processor for the next fifty milliseconds, which program’s data sits in which corner of memory, and what happens when two programs want the same file. The operating system is that someone: a referee with absolute authority over the machine’s resources. In the second description, the operating system is an illusionist. It presents each program with a clean, simplified machine - files instead of spinning platters and flash cells, windows instead of framebuffers, sockets instead of radio packets - so that the person writing the weather app never has to know how a particular screen controller or a particular 5G modem actually works. Engineers call this the extended machine: the real, ugly hardware, wrapped in a tidy fiction that programs can rely on.

Both descriptions are true at once, and the tension between them is the plot of the book. The referee must be small, fast and absolutely trustworthy, because everything depends on it. The illusionist must be large and accommodating, because everything is built on top of it. For seventy years the industry has argued, through actual products and actual fortunes, about where the balance lies. Small and strict, or large and generous; one kernel doing everything, or a tiny kernel with services around it; software given away with the machine, or software sold as a product; code anyone may read, or code locked behind a licence. Every one of those arguments was fought with real operating systems, by named people, in specific years, and most of them are still being fought today under new names.

The operating system is also, not incidentally, a business - and an unusual one. The company that controls the layer everyone must pass through collects something like a toll on all computing that happens above it. IBM discovered this in the 1960s, Microsoft perfected it in the 1980s and 1990s, Apple and Google rebuilt it for the phone in the 2000s, and regulators on three continents have spent four decades trying to decide how much toll is too much. The largest acquisition in software history, IBM’s 34-billion-dollar purchase of the Linux company Red Hat, completed in 2019, was at bottom a purchase of position in this layer [O25]. When Broadcom paid for VMware in 2023, it was buying another kind of toll booth: the software that lets one physical machine pretend to be many [O66].

But the story does not begin with business. It begins with a simpler problem, almost embarrassing in hindsight: in the 1950s, computers were so expensive that the human beings operating them were the bottleneck. The machine cost hundreds of dollars an hour, and it sat idle while a person walked across the room with a deck of punched cards. The operating system was invented to fire the human from the loop, and the first versions of it were called, without poetry, monitors and supervisors. The ghost that now runs the world began as a scheduling hack.

What the boss actually does

Strip the seventy years away and the supervisor’s job description has four lines. It decides who runs: scheduling, the allocation of the processor’s attention among waiting programs. It decides who remembers: memory management, handing out and protecting the machine’s workspace. It decides who talks to what: input and output, the drivers and protocols that reach disks, networks and displays. And it decides who is allowed: protection, the rules that keep one program’s mistake or malice from becoming everyone’s. Every operating system in this book, from the GM-NAA monitor of 1956 to HarmonyOS NEXT in 2024, is a variation on those four decisions. The variations - who makes them, how fast, with what proof, at whose expense - are the plot.

A note on names

The word “operating system” itself arrived late. The 1950s said “monitor” or “supervisor”; IBM said “control program”; Burroughs said “master control program”. The historian’s study of the field’s first decade [O7] shows the modern term settling in only around 1964, precisely when System/360 made the layer impossible to ignore. Naming is a lagging indicator: a thing gets its name when it becomes indispensable. “Kernel” came later still, borrowed from mathematics, for the irreducible core that must always be present. This book uses the words loosely and the functions strictly, because the functions are what survived.

Chapter 2 - The machine room problem

To understand why the operating system had to be invented, stand for a moment in a computer installation of the mid-1950s. The computer is an IBM 704, a machine the size of a room, built from vacuum tubes, costing millions of dollars and rented for a monthly sum that would pay a small office full of engineers. It has no screen. It has no keyboard in any useful sense. Programs arrive as decks of punched cards, each card carrying a line of code or data, each deck carried by a human being.



The machine room the first supervisors replaced: tape drives, console, and the operator whose judgment became software.

That human being is the problem. A programmer hands a deck to an operator. The operator puts it in a queue with other decks. When its turn comes, the operator loads the cards, mounts the right magnetic tapes, sets switches on the console, and starts the run. The program runs - perhaps for seconds, perhaps for an hour - printing results or punching new cards, and then the machine stops and waits for the operator to notice, unload everything, and set up the next job. Between jobs, the most expensive object in the building sits idle. Studies of the era found machines idle for a painful share of the day, not because there was no work, but because the work moved at walking pace.

The economic logic was brutal. A machine that cost hundreds of dollars an hour could not afford to wait on a person carrying cards. The waste was not a rounding error; it was the difference between computing being rationed to a few grand problems and computing becoming a general utility. Every installation felt it, and because the customers of these machines included aircraft manufacturers, oil companies, weapons laboratories and insurance firms - organisations with more problems than machine hours - the pressure to fix it came from the buyers as much as from IBM.

The fix, when it came, was a piece of software with a modest name: a monitor. The idea is so ordinary now that it is hard to see as an invention. Instead of a person loading each job, a resident program stays in the machine’s memory and does the loading itself. The operator stacks a batch of jobs on the card reader. The monitor reads the first job, hands control to it, takes control back when it finishes, records what happened, and immediately starts the next. The machine stops waiting for people; people start waiting for the machine. Batch processing had arrived, and with it the first true ancestor of every operating system alive today.

What the early monitors really changed was who, or what, was in charge. Before the monitor, the computer between jobs was an inert heap of circuits awaiting a human decision. After it, the computer always had

software in control, even between jobs - a permanent skeleton crew of code that never went home. That crew grew. First it loaded jobs; then it managed the tape drives and printers so each program did not have to; then it kept accounts of who used what; then it protected one job from another’s mistakes. Each addition moved one more responsibility from people or from individual programs into the resident code. Historians who have traced the period closely note that the very phrase “operating system” only settled into its modern meaning around the early 1960s, as these accumulations of monitors, supervisors and input-output packages came to be seen as a single thing - a system for operating the machine [O7].

There is a material-thinking point here that the rest of the book keeps returning to. The operating system was not dreamed up as a grand theory. It was squeezed out of a physical constraint: the mismatch between a machine that worked in microseconds and a workforce that worked in minutes. Again and again in this story, the big abstractions - time-sharing, virtual memory, the kernel, the app store - arrive as answers to concrete bottlenecks: expensive memory, slow disks, too many users, too many programs, too much fraud. The theory came afterwards, to explain what necessity had already built.

The arithmetic of idleness

The economics that forced the operating system into existence were simple and brutal. A machine like the IBM 704 rented for the equivalent of a large house every month, and its cost per hour was measured against the only alternative: rooms full of people with desk calculators. An operator who spent ten minutes mounting tapes, loading cards and resetting switches between a five-minute job and the next had just thrown away two jobs’ worth of the most expensive time in the building. Computer managers kept logs of the gap and were judged by them. The pressure was not abstract efficiency; it was a line item that accountants could read. Every feature of the early supervisors - automatic job sequencing, standard tape labels, the resident monitor that stayed in memory between jobs - can be traced to a manager’s stopwatch. The operating system was born as a cost-accounting measure, and it never entirely stopped being one: the cloud’s millisecond billing and the mainframe’s workload manager are the same stopwatch, sixty years apart.

Who stood in the machine room

The social order of the early computing centre deserves recording, because the software replaced specific people. At the top sat the machine’s owners and their programmers; in the middle, the professional operators, whose craft was the console - the banks of lights and switches through which a skilled person could read the machine’s state like a weather map; at the bottom, the punch-card clerks, overwhelmingly women, who translated the world’s paperwork into holes. The resident monitor displaced the operator’s judgment first, automating the decisions of the highest-paid person in the room, which is why the earliest systems were sold as much on staffing savings as on speed. The programmers, meanwhile, were exiled from the machine they programmed: where a 1950s programmer booked the computer and sat at its console, a 1960s programmer handed a deck of cards through a window and collected printout hours later. Computing gained throughput and lost intimacy. The interactive terminal, when time-sharing arrived, was in part a rebellion by programmers who wanted their machine back.

Chapter 3 - The first supervisors

The distinction of being the first operating system usually goes to a piece of code written not at IBM but by its customers. In 1956, Robert L. Patrick of General Motors Research and Owen Mock of North American Aviation produced the GM-NAA I/O system for the IBM 704 [O8]. General Motors had already built a monitor for its earlier IBM 701 in 1955, and the 704 version generalised the idea: it automatically started each new program when the previous one finished, and it gave all programs a shared set of routines for talking to the input and output devices, so that every programmer did not have to reinvent control of the tape drives [O8]. Patrick came from a world of automotive production planning, and he brought its tools with him - he used Gantt charts, the bar charts of factory scheduling, to work out how the machine’s parallel activities could be made to fit together [O8]. Some version of the system ran in about forty of the 704 installations [O8].

Notice the pattern, because it repeats for seventy years. The operating system was not handed down by the machine’s maker; it grew from below, from users who were wasting money. GM and North American shared their code through SHARE, the user group that 704 customers had formed - one of the first great open exchanges of software, decades before anyone called anything open source. IBM absorbed the lesson and produced its own successors: the Fortran Monitor System, then IBSYS for the 7090 and 7094, which became the standard batch environment of the early 1960s. The customer-invented supervisor became the manufacturer-supplied supervisor, and by the middle of the decade nobody sold a large computer without one.

While American installations were perfecting the batch queue, a group at the University of Manchester was building something stranger and more forward-looking. The Atlas computer, which began operating in 1962, was one of the fastest machines in the world, and its software - called the Supervisor - was arguably the first recognisably modern operating system. The Atlas Supervisor could keep several programs in flight at once, letting the machine compute one job while it read or printed another. More remarkably, the Atlas pioneered the memory scheme that now underlies every phone and laptop on earth: paging. The machine’s physical memory was small, so the Supervisor automatically moved fixed-size pages of data between fast core memory and a slower drum, presenting programs with the fiction of one large, uniform store. The programmers called the fiction “one-level store”; the modern name is virtual memory, and Chapter 41 of this book is about it. The Atlas team also separated the machine into privileged and ordinary modes, so that only the Supervisor could perform dangerous operations - the seed of the kernel-user divide that still defines what an operating system is.

By 1964 the pieces of the modern concept existed in scattered form: the resident monitor that schedules work, the shared input-output routines, the privilege boundary, the paging trick. What did not yet exist was a single system that did all of it, for everyone, on every machine. IBM decided to build exactly that, and the decision nearly broke the company - and made the software industry.

SHARE and the users’ revolt

GM-NAA I/O mattered beyond General Motors because of what its users did next. In August 1956, the operators of the first IBM 704s - aerospace firms, mostly, clustered around Los Angeles - formed SHARE, the first computer users’ group, to swap programs and fixes rather than each write their own [O7]. The arrangement was unprecedented: competitors pooled their system software because the alternative was everyone duplicating the same scaffolding. SHARE’s members improved the GM-NAA monitor collectively, passed the improvements around on tape, and pressured IBM to take over maintenance of what became IBSYS, the standard 7090/7094 monitor. The pattern set here - users treating the base layer as a commons

because no single user benefits from hoarding scaffolding - is the direct ancestor of the free-software movement described in Part V. Stallman did not invent the instinct; he re-invented it after the industry had unlearned it.

ATLAS and the supervisor’s other root

The economics of the idle hour

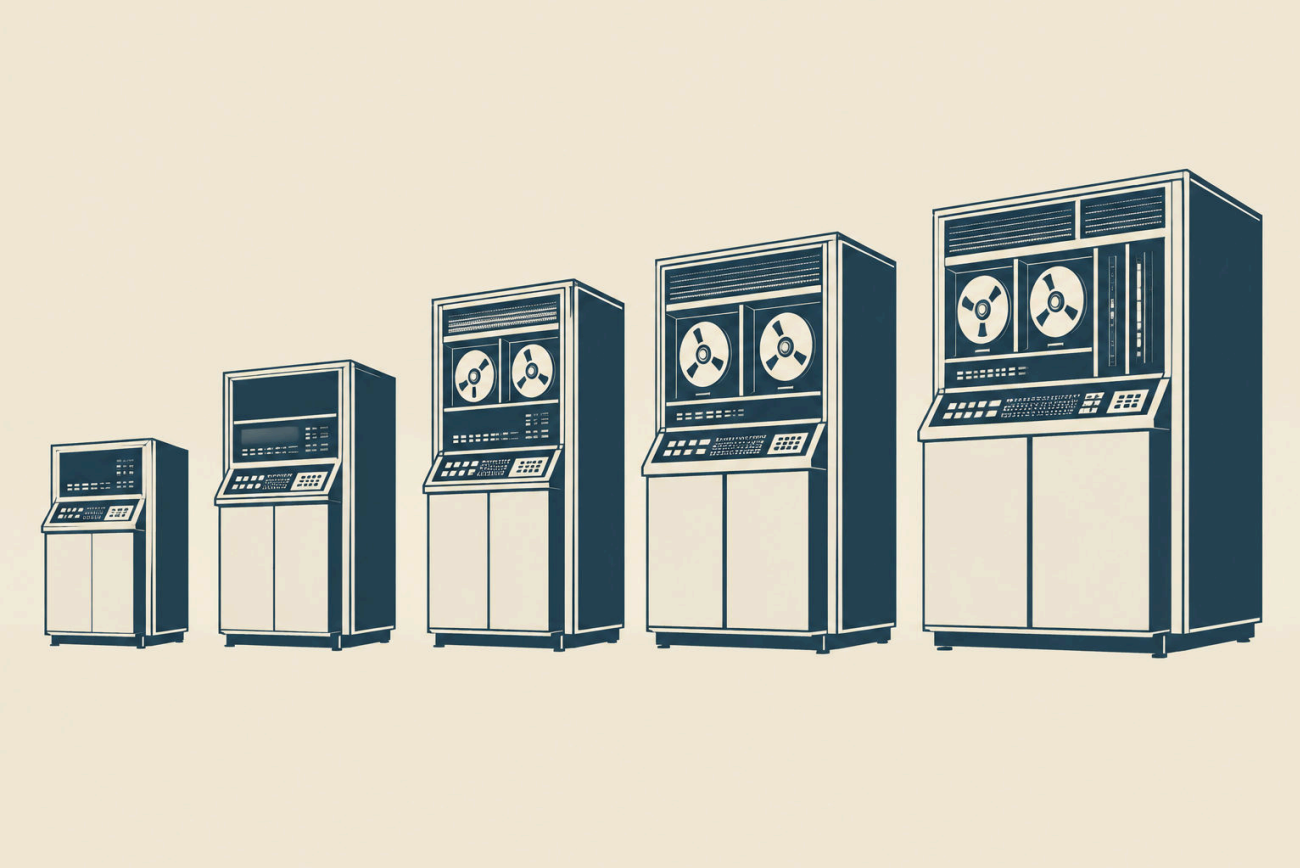
The batch monitor’s real product was not convenience but utilization, and the accounting was unforgiving. A top-of-the-line mainframe in the early 1960s rented for tens of thousands of dollars a month, which priced an idle hour in the hundreds of dollars - more than the operator watching it earned in weeks. Every minute the machine spent waiting on a human being - mounting a tape, reading a job card, noticing a run had ended - was a minute the most expensive asset in the company did nothing. The monitor attacked exactly that gap. It read the next job’s control cards itself, loaded the program itself, and took the machine back the instant the run finished. The computer’s day, which had moved at walking pace, began to move at machine pace.

Two quieter consequences followed. The first was accounting: once the monitor stood between every job and the machine, it became the natural place to record who ran what and for how long, and the billing department’s needs shaped operating-system design for decades. The second was the job control language - the terse card format in which a user declared what the job needed. Nobody loved these languages; they were unforgiving, cryptic and endlessly parodied. But they are the ancestor of every script, configuration file and cloud template since: the idea that you tell the operating system what you need in advance, in its own notation, and then get out of its way. The supervisor’s deepest habit was formed here - the machine runs best when the humans describe their work and then leave the room.

The British thread ran through Manchester. The ATLAS machine, switched on there in 1962, was supervised by a program ambitious enough to deserve the modern name: it managed one-level store - an early form of virtual memory in which fast core and slow drum appeared as a single address space - plus spooling, so that slow card readers and printers were decoupled from the fast processor through magnetic tape buffers. ATLAS’s authors called the supervisor’s services “extracodes”, instructions that existed only in software. The idea that an operating system could extend the machine’s instruction set with pretend instructions is the system call, fully formed, in 1962. Meanwhile at Burroughs, the Master Control Program for the B5000 (1961) took the boldest step of the decade: it was written in a high-level language, ESPOL, a dialect of ALGOL, at a time when “real” system programmers wrote assembler. The bet that systems software could be written above the metal would take twenty years to win, through C and Unix, but MCP placed it first.

Chapter 4 - The bet: one architecture to replace them all

In the early 1960s IBM had a problem that looked like success. It sold several incompatible families of computers - scientific machines, business machines, small machines, large ones - each with its own architecture, its own software, its own peripherals. A customer who outgrew a small IBM machine had to throw away every program when moving to a bigger one. Software, in those days, was written fresh for each installation, and there were essentially no commercial software companies to help [O10].



One architecture, many machines: the System/360 family idea - the same programs, from the smallest model to the largest.

IBM’s answer, approved after six months of heated debate by a secret task force named SPREAD - Systems Programming, Research, Engineering and Development - that met at the New Englander Motor Hotel in Greenwich, Connecticut, was to burn the whole product line down and replace it with one family: the System/360, announced in April 1964 [O10]. One architecture, from the smallest machine to the largest, so that a program written for any of them would run on all of them. The chief architect was Gene Amdahl; the project leader was Fred Brooks, who coined the very term “computer architecture”; Bob Evans and Erich Bloch rounded out the leadership, and all three later shared the first National Medal of Technology for the work [O10]. Getting the System/360 designed, built and launched cost IBM upwards of five billion dollars - the largest privately financed commercial project ever undertaken to that point [O11].

The hardware bet worked beyond anyone’s hopes; the System/360 became the dominant computer family of its era, and its descendants still run today. But the 360 needed an operating system worthy of the family - one system that would run across the whole range, on machines with wildly different memory sizes and peripherals. That system was OS/360, and its development became the most instructive disaster in software history.

OS/360 arrived late, overweight and buggy. The project consumed staggering effort and still slipped. Fred Brooks, who managed it, did the industry one last service beyond the machine itself: he wrote the disaster down. His 1975 book, The Mythical Man-Month, distilled OS/360’s pain into laws that software managers still recite - most famously that adding manpower to a late software project makes it later, because every new person must be taught, and every added person multiplies the number of conversations needed to keep everyone consistent. The book’s enduring lesson is that large software is not slowed by typing but by coordination, and the operating system is the largest software most organisations will ever attempt.

Two deeper consequences outlived the schedule overruns. First, OS/360 proved that an operating system could be a general-purpose platform: one system, many machines, countless applications written by people IBM would never meet. The idea of the platform - the base layer on which an ecosystem builds - was born here, and IBM's own history now credits the System/360 with establishing the first platform business model in computing [O10]. Second, the sheer cost and difficulty of OS/360 taught the world that software was not a free accessory to hardware but a thing of its own, with its own economics, its own risks, and soon its own companies. Within five years of the System/360 announcement, IBM would take the step that turned that lesson into an industry.

Five billion dollars, and what it bought

The size of the System/360 bet is worth pausing on. IBM's own retrospective account puts development costs at over $5 billion in mid-1960s dollars - more than the Apollo program was spending per year, staked by a private company on a single product line [O11]. The money bought simultaneous revolutions: a new circuit technology (Solid Logic Technology modules, halfway between transistors and integrated circuits), new peripherals, new factories, and the largest software project yet attempted. The people who carried it became the industry's founding generation: Gene Amdahl as chief architect; Bob Evans driving compatibility as strategy; Fred Brooks, still in his early thirties, managing OS/360 after the hardware leads had been chosen [O10]. Brooks's later verdict, that adding people to a late software project makes it later, was bought with OS/360's schedule. The bet nearly broke the company and instead made it: System/360 orders overwhelmed forecasts, IBM's revenues doubled within the decade, and "compatible with the 360" became the industry's definition of legitimacy.

Compatibility as invention

What compatibility actually bought

The technical content of the 360's promise is easy to state and was brutal to keep: one instruction set, one architecture, across machines whose prices differed by a factor of fifty. A program assembled for the smallest model had to run, unchanged, on the largest. Peripherals had to interchange. The operating system had to present the same face whether the machine filled a closet or a hall. Nothing about this was natural. Faster machines wanted wider words and bigger memory; cheaper machines wanted to cut both. Holding the family together meant the large machines carried design compromises for the small ones, and the small ones emulated instructions they could not afford in hardware - trapping them into software, which is the operating system's oldest trick for pretending the hardware is richer than it is.

The economic content was larger. Before the 360, every machine generation orphaned its customers' programs; upgrading meant rewriting, which meant the true cost of a new computer was the old software. Compatibility turned programs into durable assets - a ledger that could outlive the machine that kept it. That single change reordered the industry's power. Software stopped being part of the sale of a machine and became a reason to stay with a maker: each program written for the architecture was another strand of the moat. IBM's sales force understood this before its engineers finished proving it. The operating system, as the layer where the compatibility promise was actually kept, became the place where customer loyalty physically lived - the first clear case of the pattern this whole book traces, in which control of the base layer is worth more than any single machine it runs on.

The deepest decision was not technical but contractual with the future: every model in the family - from the small Model 30 to the large Model 75 - would run the same programs. That sounds obvious now because it won. At the time it was heresy: fast machines were supposed to be designed differently from slow ones. Amdahl’s team solved it with an architecture deliberately separated from its implementations - a definition of what programs could rely on, distinct from how any particular model achieved it. That separation is the ancestor of every platform promise since: the POSIX standard, the Android compatibility definition, the browser’s web platform. System/360 taught the industry that an architecture is a promise, and that promises, kept, are worth more than any single machine.

PART II - THE BET

In April 1964 IBM announced six computers at once, and with them a promise no computer maker had ever kept: that a program written for the smallest would run on the largest. The System/360 consumed roughly five billion dollars - more than the Apollo program was spending in the same years - and the greater share of the pain was not the hardware but OS/360, the operating system meant to be the family's common soul. Fred Brooks, who managed that project, turned its overruns into the most durable book of software wisdom ever written, and its central warning - that adding people to a late project makes it later - has outlived every technology it described.

But this part is also about what the bet bought. Compatibility turned software from a per-machine custom job into an asset that outlived any single machine, and that single change made a software industry thinkable. When IBM unbundled in June 1969, pricing programs separately from hardware, the industry became actual: ADR, Informatics, and a thousand contract houses found they could sell what had always been given away. The tar pit and the gold rush were the same event. Understanding the 360 means holding both truths at once - the worst-managed success in the history of computing, and the moment the operating system stopped being a machine's accessory and became a product, a profession, and eventually the most valuable layer of the economy.

Chapter 5 - OS/360: the tar pit

OS/360 was meant to be the software half of IBM's great unification: one operating system for the whole System/360 family, from small business machines to large scientific ones. What the project actually became is written plainly in the memoir of the man who managed it. Fred Brooks called the book The Mythical Man-Month, and its description of large software as a tar pit - the harder the beast struggles, the deeper it sinks - has never been bettered.

The difficulties were structural, not personal. The system had to run on machines with very different memory sizes and devices, so everything had to be configurable, and configuration multiplied testing beyond what any team could finish. It had to carry forward the habits of IBM's earlier systems while inventing multiprogramming - keeping several jobs in memory at once so that while one waited on a tape, another could compute. It had to be built by hundreds of programmers whose work had to fit together exactly, and the communication burden grew faster than the code. Brooks's famous conclusion fell out of the arithmetic: adding people to a late project makes it later, because the newcomers must be trained by the very people whose work they are meant to speed up.

And yet OS/360 worked, in the way that matters. Delivered in stages through the mid-1960s, it became the standard environment of the mainframe world, and its descendants - MVS, then OS/390, then today's z/OS - still run the core systems of banks, insurers, airlines and governments. IBM's own current mainframe, the z16 announced in 2022, is marketed with real-time AI inference on its Telum processor and quantum-safe cryptography, and it runs an operating system whose direct ancestor Brooks's team shipped [O14]. Few

artefacts in computing have this kind of continuity: a transaction at a bank today may complete inside a system whose design arguments date from 1963.

OS/360 left the industry two inheritances beyond the code. The first is the discipline of thinking of the operating system as a platform: a stable base, documented interfaces, applications written by strangers. The second is the discipline of humility about large software. Nearly every management idea the software world now takes for granted - small teams, incremental delivery, the suspicion of heroic schedules - has a footnote somewhere to what happened on that project.

The 360Revolution papers

The primary record of the 1966-1969 transformation survives in IBM's own historical accounting and in the engineering literature of the moment. IBM's retrospective on the System/360 revolution [O11] records the internal arithmetic: the company had become a computer company by betting its whole balance sheet, and by the late 1960s the bet had so thoroughly paid off that IBM's problem was no longer survival but governance - how to manage a market it dominated so completely that its own product decisions set industry law. The unbundling decision of June 1969 has to be read in that light: not a concession to a market that existed, but the creation of one. The IEEE Annals papers on the period [O12][O13] trace the consequences through the companies that immediately formed to sell what had been free - Applied Data Research's Autoflow being the canonical case, a flowcharting program whose commercial success proved that software could be a product with a price, a sales force and a customer. The entire software industry, Microsoft included, descends from that proof.

Multics: the overreach that taught

While OS/360 strained to be adequate, Multics strained to be perfect. A joint project of MIT, Bell Labs and General Electric from 1964, it proposed a computing utility: one system, shared by a whole community, as reliable and impersonal as the telephone exchange. Its design papers [O15][O16] are still startling - a single-level store where every file is memory, dynamic linking, security rings, all in the mid-1960s. Multics shipped late, expensive and slow, and sold in the dozens; by every commercial measure it failed. But it was the best-taught failure in computing history. Its ideas seeded Unix (whose name is a pun on it), informed the security architecture of everything after, and trained the generation - Thompson, Ritchie, Saltzer, Corbato - that would define the field. Corbato's Turing Award, when it came in 1990, was for time-sharing generally but Multics in spirit: the machine as a utility, decades before the industry could afford the vision.

Brooks's law, in context

The Mythical Man-Month, Brooks's 1975 memoir of OS/360, is the most-quoted management book in software, and its core law - adding manpower to a late software project makes it later - was bought with the experience of watching OS/360's schedule slip while its staffing swelled into the thousands. But the book's subtler lesson concerns the operating system itself: Brooks argued that conceptual integrity - one mind's coherent vision - is the scarcest resource in a big system, and that OS/360's flaws came from the compromises of committee design. The industry tested both claims against the commons half a century later: Linux, with thousands of contributors, achieves conceptual integrity through maintainer hierarchy and Linus's editorial eye; the design-by-committee systems (OS/2, Taligent, the Hurd) mostly failed. Brooks's dichotomy survives but the variables moved: the cathedral and the bazaar turned out to need the same thing - a final arbiter with taste.

The tar pit’s survivors

OS/360 shipped, late and buggy, and then refused to die: its descendant line - OS/VS2, MVS, MVS/ESA, OS/390, z/OS - has been continuously maintained for sixty years and runs, today, the majority of the world’s large banking and insurance transactions. The z16 (April 2022) runs it on silicon with on-chip AI inference [O14], and the platform’s owners still promise that programs written for System/360 run unmodified - the oldest compatibility promise in the industry, older than DOS’s, kept more strictly. The tar pit’s moral is therefore double. Big software projects do drown in their own tar; and a system that survives its drowning becomes nearly immortal, because the cost of leaving it only grows. The mainframe is this book’s first and most durable example of the base layer’s ratchet: once the world’s ledgers run on you, the world will pay almost anything rather than migrate.

Chapter 6 - Unbundling: software becomes a thing you can sell

Until the end of the 1960s, the question “what does an operating system cost?” had no answer, because software was not priced. Computer manufacturers gave software away with the machine; it was part of what made the hardware usable, and the idea of paying separately for it struck many as odd. The conventional wisdom, as one history of the period puts it, was that you could not make money selling software - it was either given away by the manufacturer or written uniquely for each installation [O13].

There were exceptions before anyone famous acted. Entrepreneurs at companies like Applied Data Research and Informatics had spent the mid-1960s quietly proving that a program could be sold off the shelf, over and over, to hundreds of customers [O13]. But the legitimising event came on 23 June 1969, when IBM - under antitrust pressure and facing its own cost arithmetic - announced that it would price software and services separately from hardware. “Unbundling” is the era’s own word for it [O12].

The consequences arrived in two waves. The first wave was the software products industry itself: companies that wrote and sold programs, including systems software, as a business in its own right. The second wave took longer to see: once software had a price, it had a margin, and once it had a margin, the operating system - the one piece of software every machine needed - became the most strategic product in computing. The company that owned the dominant operating system would own a toll position at the base of everything. IBM, constrained by a consent decree and later by a thirteen-year antitrust case that the government filed the same year and eventually dropped in 1982, never exploited that position as ruthlessly as the law might have allowed. The companies that came after studied the position carefully.

It is worth sitting with how strange the moment was. The operating system had begun, thirteen years earlier, as a scheduling hack written by customers and shared for free. Now it was becoming property - licensed, priced, defended in court. Every argument the industry has had since - about open source, about app store commissions, about who may repair or modify the software in a device you own - is an echo of the day software got a price tag.

The plaintiffs who priced the product

The cleanest evidence that unbundling created an industry rather than merely permitting one is the speed of the case law. Applied Data Research, a New Jersey software house, had been selling Autoflow since 1965; after 1969 it sued IBM itself, and the IEEE Annals history of the period [O13] records how ADR and its peers -

Informatics, Cullinane, Management Science America - turned the new pricing boundary into a product category: the packaged program, sold off the shelf, priced in the tens of thousands of dollars. Their customers were the same data-processing managers who had always received software free with the machine; now a line appeared in the budget, and with a budget line came vendors, salesmen, maintenance contracts and trade shows. The “software industry” as a phrase dates from these years. It is not too much to say that every software company since, from Microsoft to the smallest app-studio, operates inside the market boundary drawn on 23 June 1969 [O12].

The services exception

Unbundling also created a second, quieter industry alongside products: services. When software had a price, so did the labour of making it work, and systems integrators and contract programmers multiplied through the 1970s. The distinction matters for this book because it prefigures the industry’s permanent tension: is the operating system a product (Windows, z/OS), a service (Red Hat’s subscriptions, Chapter 21), or a commons (Linux itself)? The 1969 boundary made all three answers commercially possible at once. IBM, characteristically, learned to sell all three; the antitrust suit that accompanied unbundling, filed that January, was the state’s way of asking whether any single firm should be allowed to.

The number that was not zero

What did unbundling change in practice? IBM’s software had been “free” the way an airline seat’s meal is free: priced into the whole. After June 1969 the prices appeared, and they were real - OS/360’s successor systems and the major program products carried monthly licence fees that data-centre managers now had to justify, which instantly created the concepts the industry still runs on: software budgets, upgrade economics, vendor comparison. The IEEE Annals account [O12] records how quickly the internal arithmetic shifted: once software had revenue, it had advocates inside IBM, a roadmap, and eventually a franchise. The company’s later reluctance to open its architectures (the 1980s Micro Channel episode, Chapter 13) reads differently against this: 1969 taught IBM that software was a business; 1981 taught it that it had given the business away. The operating system’s commercial history is bracketed by those two lessons.

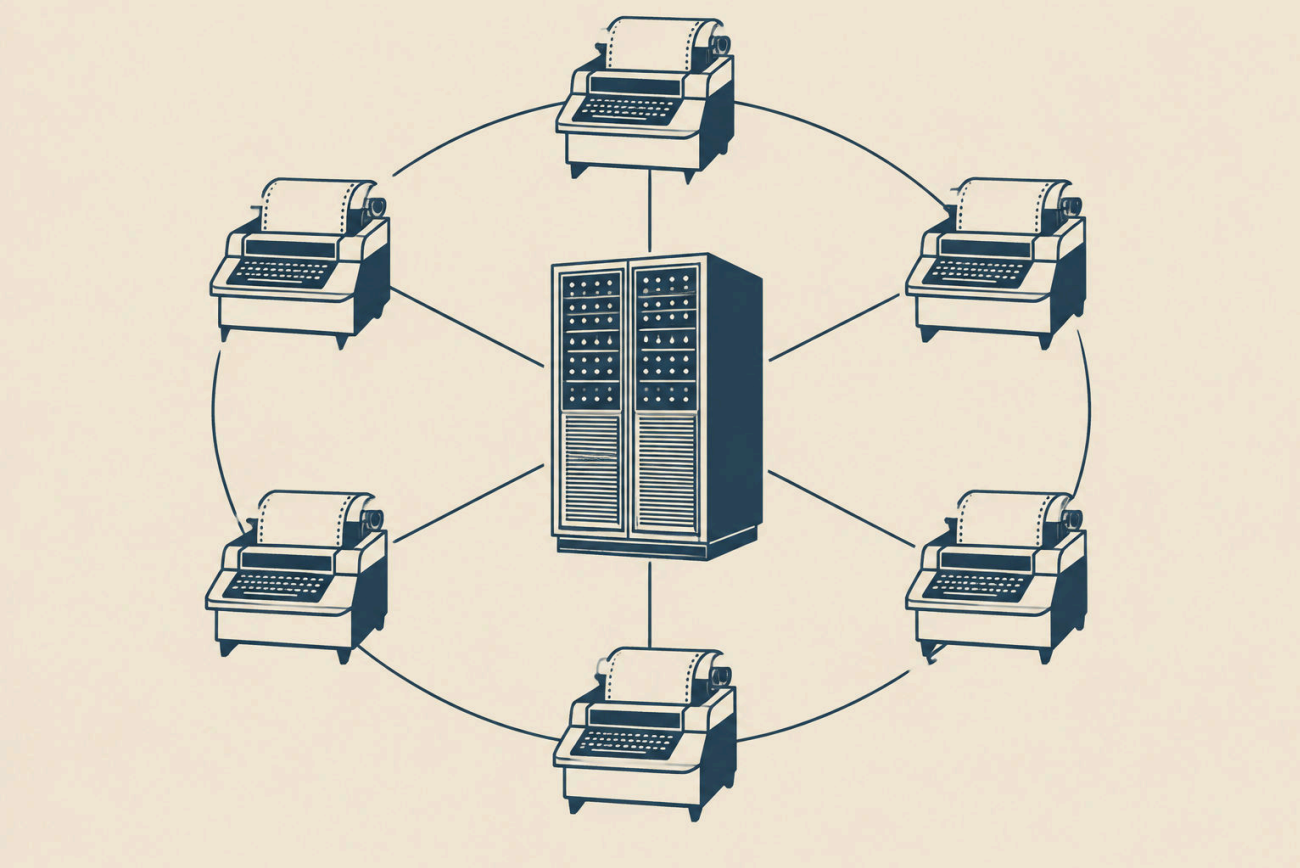
The judge’s shadow, once more

It remains impossible to say how much of unbundling was strategy and how much was the antitrust suit filed five months earlier, and IBM’s own historians are wisely noncommittal. But the sequence is the fact: the world’s dominant computer company, under legal scrutiny for tying software to hardware, voluntarily untied them - and the independent software industry exists in the space that opened. Every later gatekeeper in this book studied the sequence. Microsoft’s behaviour in the 1995 consent negotiations, Google’s and Apple’s elaborate separations between their platforms and their services, even Huawei’s insistence that HarmonyOS is an open-source project with a commercial distribution - all are architectures designed by people who know exactly what happened the last times the state asked whether the platform and the business were one thing.

Chapter 7 - The computer learns to share

Batch processing solved the idle-machine problem and created a human one: programmers now submitted a deck and waited hours for the result, and a single misplaced comma cost half a day. The obvious dream was to

go back to sitting at the machine - but to let many people sit at it at once.



Time-sharing: many terminals, one machine, and the scheduler deciding whose turn is next, many times a second.

The intellectual seed was planted early. John McCarthy had been thinking about interactive computing since the mid-1950s, and in 1959 he wrote a memo proposing a time-sharing system for MIT's IBM 709 [O15]. The idea: the machine is fast and humans are slow, so let the machine serve each human for a thin slice of time, then switch to the next. If the switching is fast enough, every user at every terminal feels alone on the machine. The computer, instead of being a queue, becomes a conversation.

Fernando Corbató and his MIT colleagues made it real. Their Compatible Time-Sharing System, CTSS, demonstrated in 1961, let users work at typewriter terminals simultaneously, each with the illusion of a private machine [O15]. The engineering that made the illusion possible was an operating system problem through and through: the supervisor had to swap whole programs in and out of memory on schedule, protect one user's program from another's, and give each user private files - which meant inventing passwords, file permissions, and accounting, inventions so useful that nobody remembers they began here.

Time-sharing changed what an operating system had to be. The batch monitor was a loader; the time-sharing supervisor was a referee in a crowded room. Scheduling, memory protection, and fair allocation - the three problems that still define the kernel's job - all became urgent at once. And the experience of using CTSS produced a generation of people who had tasted interactive computing and wanted more of it than one machine could give. Some of them, at MIT, General Electric and Bell Telephone Laboratories, launched a project to build the ultimate shared system: a computing utility, as public and reliable as the telephone network. They called it Multics.

Corbato’s demonstration

The Compatible Time-Sharing System was first demonstrated in November 1961 on an IBM 709, four typewriter consoles sharing one machine [O15]. The word “compatible” was a peace offering: CTSS ran atop a modified batch monitor, so the machine could still do its day job while the experiment ran. Fernando Corbato’s group at MIT built the thing to solve an economics problem that sounds modern: the machine cost too much for one person’s use, but each person’s actual demand came in bursts - a compile here, a run there - with long thinking pauses between. Time-sharing filled the pauses with other people’s bursts. The response-time target was a human one: answers in seconds, fast enough that the user’s train of thought survived. That target, not any machine property, defined interactive computing, and every scheduler since (Chapter 34) is measured against it.

The terminal becomes the computer

Time-sharing changed what a user bought. In the batch world, the customer purchased machine time and received printout. In the time-sharing world, the customer purchased a terminal session and received an illusion: the experience of a private computer. The 1960s literature is candid that the illusion was the product [O16]. John McCarthy’s phrase for it - computing as a utility, like electricity or the telephone - supplied the business model: rent access by the month. Commercial time-sharing bureaus (Tymshare, GE’s service, university centres) sold exactly that through the late 1960s, and their decline in the 1970s came not from failure but from the microcomputer undercutting the rental price of the illusion. When the cloud re-rented the same illusion at planetary scale half a century later (Chapter 42), it was the third time the industry had sold one machine as many; each round only changed what “one machine” meant.

The PDP-1 and the hands-on tradition

Time-sharing did not invent interactive computing; it rescued it from economics. The early small machines - the TX-0 and PDP-1 at MIT, where Spacewar! was written in 1962 - had already demonstrated what a person alone with a computer could do, and the hacker culture that grew around them treated the batch queue as an affront. The problem was arithmetic: interactivity on a PDP-1 meant dedicating a $120,000 machine to one person’s game or program. Time-sharing’s trick was to sell the same experience on a machine shared fifty ways. The two traditions - personal and shared - then chased each other for sixty years: the workstation and PC re-personalized computing in the 1980s, the Internet re-shared it in the 1990s, the cloud centralized it again in the 2000s, and the phone put it back in the pocket. Each swing was driven by the same underlying variable - the price of computation relative to communication - and the operating system swung with it, from batch monitors to personal DOS to Internet Unix to cloud kernels. The ghost is agnostic about where it lives; it follows the economics.

Project MAC and the research bet

The institutional sponsor deserves its due: Project MAC at MIT, funded by DARPA from 1963, was simultaneously the time-sharing experiment (CTSS, then Multics), the artificial-intelligence laboratory, and the seed of what became MIT’s computer-science establishment. DARPA’s funding model - generous, patient, publication-friendly - was a deliberate bet that interactive computing mattered, made when the industry consensus was batch. The bet’s returns included not just systems but the ARPANET’s culture: the protocols and habits of the Internet were designed by people who had grown up on time-shared machines, and it shows. The Internet is, in a real sense, time-sharing escaped from the machine room - many computers, shared by

many people, as if they were one resource. The operating system’s job description (Chapter 1) scales up surprisingly well.

Chapter 8 - Multics: the cathedral

Multics - the Multiplexed Information and Computing Service - was planned from 1964 as nothing less than the operating system to end operating systems: a prototype computer utility, built jointly by MIT, General Electric and, from 1965 to 1969, Bell Telephone Laboratories [O16]. Its ambitions were total. It would serve a whole community of users continuously, the way a power company serves a city. It would have a hierarchical file system - the tree of directories inside directories that every computer user now takes for granted. It would protect information with concentric rings of privilege, so that the most trusted code ran in the innermost ring and ordinary programs in the outer ones, an architecture that mainstream processors would eventually build into silicon. It would be written, as far as possible, in a high-level language rather than assembly - a radical economy at a time when system software was hand-carved per machine.

The ambition was real and the results were real; Multics worked, and installations ran it for decades. But it was enormous, expensive and late, and by 1969 Bell Labs decided to leave the project [O16]. The judgment of history is subtle. As a product, Multics missed: the future did not belong to one great shared cathedral of computing. As a school, it was unmatched. Nearly every idea that defines a modern operating system - the file tree, dynamic linking, rings of protection, the very discipline of building a system as a coherent design rather than an accretion - passed through it. And the people who left its halls carried the ideas out.

Two of those people were Ken Thompson and Dennis Ritchie. They had worked on Multics at Bell Labs, absorbed its design, and watched it collapse under its own weight. What they built next was, in spirit, Multics written by two people who had learned exactly what to leave out.

What the cathedral built

Multics’ technical legacy can be listed with unusual precision because its ideas were published as they were built [O16]. The hierarchical file system with directories nested to any depth: Multics. Dynamic linking, loading library code at call time rather than build time: Multics. The single-level store, in which a file and a region of memory are the same object: Multics, revived decades later as memory-mapped files in every modern system. Rings of protection - hardware-enforced layers of trust inside the machine, not merely between kernel and user: Multics, and the reason x86 chips still carry ring bits that modern kernels mostly leave unused. Even the modern shell’s ancestry runs through it. The design papers read like a catalogue of the next forty years of operating systems research, delivered early and in one place.

Why it failed anyway

The commercial story was grimmer. Multics ran only on large, expensive GE/Honeywell hardware; it arrived years late; it was famously heavyweight in an era when the future was tilting toward smaller machines. Honeywell sold perhaps seventy-five installations over the system’s life. Bell Labs’ 1969 withdrawal was the judgment that mattered: the laboratory that had co-designed it concluded the project could not be the utility it promised on the hardware that existed. The lesson its alumni drew - Thompson and Ritchie above all - was not that the vision was wrong but that the implementation culture was: big committees, big machine, big specification. Unix was the counter-thesis: implement the ideas one at a time, for the machine you actually

have, and let elegance emerge from constraint. The cathedral and the bazaar, twenty-five years before Eric Raymond named them, with the cathedral's own stones reused to build the bazaar.

The people Multics trained

Multics' alumni list is a partial index of the next half-century: Thompson and Ritchie (Unix, C), Saltzer and Schroeder (the protection literature - Saltzer's and Schroeder's 1975 principles of secure design remain the field's textbook), Corbato (time-sharing's public face and 1990 Turing laureate), and hundreds more who carried its ideas into DEC, Prime, Data General and later firms. The CTSS and Multics papers [O15][O16] were also the field's teaching documents: a generation learned operating systems by reading how Multics handled segments, rings and linking, because it was the system that published. The pattern repeats the IBM Cambridge experience that produced CP/CMS (Chapter 5): a research programme, commercially disappointing, whose real product was trained minds and working concepts. The industry's R&D model - the industrial lab that publishes - peaked in this era, at Bell Labs, PARC, Watson and MIT, and the operating system more than any other technology is its artefact.

Honeywell and the long tail

What Multics left behind

Multics is usually told as a failure, which misses what it was: the most concentrated dose of operating-system invention ever attempted in one project. Its hierarchical file system - directories within directories, the shape every user now takes for the texture of computing itself - was its design. So was dynamic linking, the ability to bind a program to its libraries at the moment of use rather than the moment of building, without which no modern system could update a library without recompiling the world. Its rings of protection generalized the supervisor-user divide into a graded series of privileges, an idea that survives, reduced to two rings, in every processor shipping today. And its single-level store, which treated memory and disk as one continuous space, was virtual memory carried to its logical end - too far, as the performance costs proved, but instructively too far.

Unix kept the file tree and the rings-reduced-to-two, and discarded almost everything else, on principle. Where Multics asked what a computing utility for a whole civilization should contain, Unix asked what two programmers could hold in their heads. The difference was not talent but temperament: Multics was designed by a committee with a mission, Unix by refugees with a spare machine. The lesson the industry drew - that a small system that works beats a great system that is late - became the operating system's folk wisdom, repeated in every era since. It is worth noting that the folk wisdom is only half true: Multics' ideas arrived on schedule. It was Multics itself that was late.

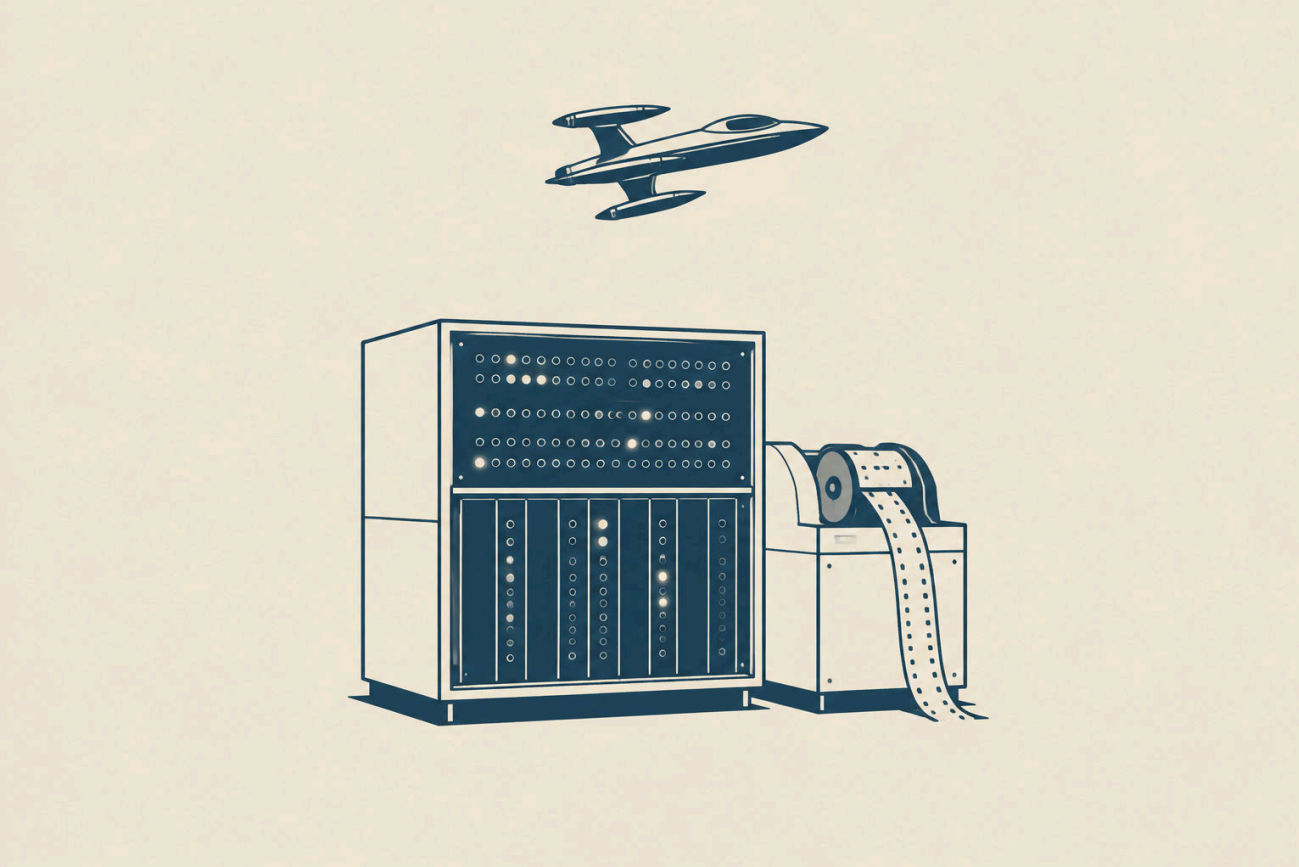
Multics itself refused to die cleanly. Honeywell, which absorbed GE's computer business in 1970, sold and maintained the system for decades; the last Multics installation (at the Canadian Department of National Defence) was shut down in October 2000. Bull, which inherited the line, even released the system's source code in 2007 for historical study. A thirty-six-year operational life for a "failed" system is a useful corrective to this book's market-share logic: systems do not die when they lose the argument; they die when their last workload is migrated. The same longevity now protects z/OS, OpenVMS and a dozen other "legacy" systems whose users see no reason to apologize for them. The base layer's economics (Chapter 39) guarantee it: migration is expensive, risk concentrates at the platform boundary, and so the old supervisors run on, in machine rooms and cloud-hosted emulators, long after their biographers arrive.

PART III - SHARING THE MACHINE

The most valuable operating system idea of the 1960s was that the machine belonged to everyone at once. Batch processing optimized the machine; timesharing optimized the people. CTSS at MIT demonstrated by 1963 that thirty users could share one computer and each feel it was theirs, and Multics - the ambitious joint project of MIT, Bell Labs and General Electric - set out to turn that trick into a computing utility for an entire community, as dependable and invisible as the telephone exchange. Multics overran, Bell Labs walked away in 1969, and the utility vision collapsed under its own ambition.

What survived was smaller and stranger. On a cast-off PDP-7, Ken Thompson and Dennis Ritchie built Unix as a deliberate rejection of Multics' grandeur - a system small enough to understand, portable enough to outlive its hardware, and generous enough to give away. The rewrite into C in 1973 made the operating system itself portable, which no one had thought to ask of one before. This part follows sharing from the 7094 to the PDP-7 to the world: how a utility dream failed upward into a design philosophy, and how a two-person skunkworks on spare hardware ended up defining what an operating system is. Every phone, server and supercomputer in this book's opening chapter runs something descended from that spare machine.

Chapter 9 - Unix: built on a spare machine



A spare PDP-7 and a space-travel game: the unlikely hardware on which Unix began.

The most consequential operating system ever written began as a game. In 1969, with Bell Labs out of Multics, Ken Thompson found himself with ideas, time, and a problem: his computer game, Space Travel - a simulation of the solar system in which the player cruised between planets and could land a ship on them - had been written for Multics and then moved to a GE 635 running GECOS, where the display was jerky and, worse, each run cost about seventy-five dollars of CPU time [O17]. Scrounging, Thompson found a little-used PDP-7, a small DEC machine with a decent display, and began moving the game to it.

Moving a program to a bare machine means writing everything underneath it. Thompson wrote, in assembly, a file system, an allocation scheme, a set of utilities - the scaffolding of an operating environment - to support one game on one orphan computer. Dennis Ritchie joined in. The little system acquired a name as a pun: it was an emasculated Multics, a castrated one-user multiplexed system - Unics, quickly respelled Unix [O16]. The first versions ran on the PDP-7 and PDP-9 around 1969-70; the system then moved to the PDP-11/20 and matured on the PDP-11/40 and /45 [O18].

What Thompson and Ritchie carried out of the Multics wreck was a taste for small, sharp tools. Unix was deliberately plain: a small kernel; a file system where everything - documents, devices, even interprocess channels - appeared as a file; a shell that let users combine simple programs rather than buy big ones. Where Multics had tried to design completeness in advance, Unix offered a few mechanisms that composed well and got out of the way. Douglas McIlroy, their department head, contributed the pipe: the ability to chain programs, each one’s output feeding the next one’s input, which turned a box of small tools into a workshop.

The design choices look technical, but they were temperament made into software. Thompson and Ritchie had watched a cathedral fail; they built a shed, and the shed outlasted every cathedral. Unix's descendants now run - under the names Linux, macOS, iOS, Android's foundations, and the BSD family - almost everything except the desktop.

The PDP-7 in the corner

The founding scene of Unix is well documented because its participants wrote it down with unusual candour. Bell Labs had withdrawn from Multics in early 1969, leaving Ken Thompson and Dennis Ritchie with habits - a hierarchical file system, a shell, the notion of computing as a shared utility - and no machine to exercise them on. Thompson found a little-used PDP-7, wrote a space-travel game for it to justify the time, and then rebuilt, in a few weeks of assembly language, the core of the environment he missed: an operating system, in the memorable retrospective phrase, "on which he could hang his ideas" [O17]. Ritchie joined; Brian Kernighan coined the pun "UNICS" for the castrated one-user echo of Multics. The official histories [O18] emphasize how much of Unix's famous elegance was poverty in disguise: a small machine forced small programs, small programs forced generality, and generality turned out to be the design's genius.

Berkeley and the great escape

Unix's second birthplace was Berkeley, where Ken Thompson's 1975 sabbatical install on a PDP-11/45 caught the attention of graduate students Bill Joy and Chuck Haley. The Berkeley Software Distribution - tapes of Joy's ex editor, the C shell, the Pascal system, and later the TCP/IP stack that became every network implementation's reference - turned AT&T's research system into a platform the world could build on. The licensing was the era's peculiar bargain: universities paid nominal fees for source code they could read, modify and teach with, but not commercialize, because AT&T, still a regulated monopoly, was barred from the computer business. The arrangement accidentally created the world's best software education program: a generation of engineers learned systems by reading Unix source, the way MINIX's book would later teach from print (Chapter 18). When the 1990s lawsuits tried to close BSD after the fact, the genie was out; the settlement left free BSD standing, and its descendants - FreeBSD, NetBSD, OpenBSD, and through NeXTSTEP the whole Apple line - remain the other half of the free-stack story that Linux headlines.

Pipes, and the philosophy

The discipline of smallness

Unix's design decisions are so familiar now that recovering how strange they were takes effort. Everything is a file: the disk, the terminal, the tape drive and later the network connection all answered to the same handful of operations - open, read, write, close - so a program did not need to know what it was talking to. Programs were small tools connected by pipes, each doing one thing, the output of one becoming the input of the next; the shell existed to arrange them. Processes were cheap to create, because on the PDP-7 they had to be. None of this was imposed by a standard or a manager. It was what two skilled programmers built when the design goal was a system they themselves would enjoy using, and its austerity was the point: every mechanism had to earn its complexity in daily use.

The portability followed from the same discipline. When the system was rewritten in C in 1973, the operating system stopped being a description of one machine and became a description of an idea - a machine-independent core with a small edge of machine-specific code. That is the reason Unix could move from the

PDP-7 to the PDP-11 to everything since, and the reason its descendants run on more kinds of hardware than any other software in existence. “Worse is better”, the later slogan for this philosophy, is usually misunderstood as an excuse for shoddiness. Its actual claim is sharper: a system simple enough to be ported, taught and repaired by ordinary programmers will outrun a perfect system that only its authors understand. The fifty years since are the demonstration.

Unix’s famous inventions are worth naming as a set, because they cohere: files as simple byte streams; directories as files; devices as files; the shell as an ordinary program; and pipes - Doug McIlroy’s 1972 suggestion, implemented overnight by Thompson - letting programs chain together, each doing one thing. The philosophy is usually quoted as “do one thing well”, but its economic content is sharper: small, composable tools let users build what designers did not foresee. That property - unanticipated composition - is the difference between a system and a platform, and it is why Unix-descended systems have hosted every computing fashion since (workstations, the Internet, the web, mobile, cloud) without being redesigned for each. The ghost of this book’s title learned, in 1969, to be a good host.

Chapter 10 - C and the portable machine

Early operating systems were welded to their hardware: written in assembly language, they embodied one machine’s quirks so completely that moving to a new model meant starting over. Unix broke the weld. In 1972-73, Dennis Ritchie designed C, a language low enough to touch the machine and high enough to ignore it, and he and Thompson rewrote Unix in it. An operating system that could be recompiled for a different computer was, at the time, a near-scandalous idea - and it changed the industry’s economics permanently. When a new machine appeared, the software no longer died with the old one. Hardware became replaceable; the operating system became the constant.

Portability had a second effect: Unix could travel to places IBM’s sales force would never go - universities. And here the story takes one of its accidental turns. AT&T, Bell Labs’ parent, was a regulated telephone monopoly bound by a 1956 consent decree that kept it out of the computer business. It could not sell Unix as a product, so it did something better for history than selling it: it licensed the system to universities for a nominal fee, source code included. A generation of students grew up reading a real operating system’s innards. At Berkeley, students led by Bill Joy turned their tape into the Berkeley Software Distribution - BSD - beginning in the late 1970s, adding the pager for virtual memory and, fatefully, the TCP/IP networking code that made Unix the native language of the growing internet.

The book-form of this education was John Lions’s commentary on Unix Version 6, a line-by-line tour of the kernel that circulated in samizdat photocopies because of licensing restrictions - probably the most photocopied operating system text ever. The lesson the generation absorbed: an operating system is not a mystery handed down by a vendor. It is a thing one can read, understand, and rewrite.

When AT&T’s empire was broken up in 1984 and the company was finally free to sell Unix, it did, aggressively - and the lawyers followed. Unix System Laboratories sued Berkeley Software Design and the University of California in 1992, claiming the freely redistributable BSD code was tainted by AT&T property [O60]. The suit landed at the worst possible moment for the free Unix world and cast a legal shadow over every free reimplementation - a shadow under which, at exactly that moment, a Finnish student quietly posted that he was writing a free operating system “just a hobby”. AT&T sold Unix to Novell, the suit settled in 1994,

and BSD lived on; its modern descendants run in Netflix's content servers (FreeBSD) and supply the world's most-used remote-login software (OpenBSD's OpenSSH). But the shadow had already done its work: it had made room for something with no AT&T code at all.

C: the portable machine

Unix's second invention mattered as much as the first. Thompson's B language became Ritchie's C between 1971 and 1973, and in 1973 the kernel itself was rewritten in C - an act of heresy against the era's certainty that operating systems lived in assembler. The rewrite bought the property that would conquer the world: portability. An operating system written in a high-level language could leave its birthplace. When AT&T's lawyers, constrained by the 1956 consent decree, allowed universities to license the source for nominal fees, Unix escaped Bell Labs entirely - to Berkeley, where graduate students added the TCP/IP networking that made it the Internet's native tongue; to Sun, SGI, HP and IBM, who built the workstation industry on it; and eventually to a student in Helsinki. Every branch of the family's dominance - the cloud's Linux, Apple's BSD core, Android - runs through that 1973 rewrite and the lawyers' accidental generosity.

C's bargain

C deserves a section of its own because it embodies the trade the whole industry would make. Ritchie's language gave programmers nearly assembler's power - direct memory access, pointers, no safety rails - with a high-level language's portability and expressiveness. The Ritchie-Thompson retrospective [O18] is unapologetic about the dangers: C trusts the programmer. That trust made Unix writable, portable and fast, and it made the next fifty years' software inherit the cost: Chapter 41's seventy percent - the share of serious vulnerabilities that are memory errors - is C's design philosophy measured in incident reports. The bargain was rational in 1973, when machines were small, programs were small, and the network was friendly. It became expensive exactly in proportion to computing's success. Rust's rise is less a new idea than a renegotiation of C's original contract, at modern stakes.

Standards and descendants

C itself became infrastructure: standardized as ANSI C in 1989, implemented for every architecture, and still the interface language of everything - Python's interpreter, the JVM, and every operating system's ABI speak C at the boundary. Its descendants sorted by ambition: C++ added abstraction without removing danger; Java and the managed languages removed danger by adding a runtime; Go and Rust remove danger by design while keeping systems-level control. The pattern mirrors the operating-system history it serves: each generation keeps the previous one's contract (fast, close to the machine) while patching the failure mode that success exposed. The kernel itself only admitted a second implementation language - Rust, in kernel 6.1, December 2022 [O64] - after forty-nine years of C alone. Infrastructure changes slowly, and then all at once, and then slowly again.

PART IV - A BOSS FOR EVERY DESK

Between 1975 and 1995 the computer shrank from a department to a desk, and the operating system shrank with it - from a staff of administrators to a floppy disk in a cardboard sleeve. The desk computer could not afford a supervisor program with payroll-sized ambitions. It needed a boss small enough for 64 kilobytes of memory and simple enough for its owner to understand, and that constraint decided everything about the next twenty years: what the personal computer could do, who could write software for it, and above all who collected the toll.

This part tells the oldest story in the industry’s modern form. Gary Kildall’s CP/M owned the first desk market and lost the decisive meeting. Microsoft bought a quick-and-dirty clone for fifty thousand dollars and licensed it to IBM as PC-DOS, keeping the right to sell it to everyone else - the most consequential clause in the history of the industry. The Macintosh proved in 1984 that ordinary people preferred windows and a mouse; Microsoft spent a decade climbing Windows on top of DOS to meet them; and Dave Cutler’s NT, built like a mainframe system in miniature, finally replaced the scaffolding with a real kernel. By 1995 one company collected a toll on nearly every desk on earth, and the desk had become the market that decided what an operating system was. The boss for every desk turned out to be one boss for nearly all of them.

Chapter 11 - The computer shrinks to a desk

While the laboratories shared big machines, a quieter revolution was cutting the machine down to personal size. The microprocessor arrived in 1971; by 1975 the Altair 8800 kit had put a crude computer within reach of hobbyists, and by the end of the decade the Apple II, the Commodore PET and the Tandy TRS-80 sat on ordinary desks. These machines were too small for anything like OS/360. They needed a new kind of operating system: tiny, single-user, built around the new floppy disk.

The person who supplied it was Gary Kildall, a computer science PhD teaching at the Naval Postgraduate School in Monterey, California. Working with an Intel development system, Kildall wrote a small control program for microcomputers - CP/M - in 1973, and founded Digital Research in 1976 to sell it [O56]. CP/M did for the first personal computers what Unix had done for the PDP-11: it hid the disk hardware behind a clean file interface, so that any program written for CP/M could run on any machine that had it. Word processors, spreadsheets, accounting packages - the first personal-computer software industry - grew up on CP/M, and by the end of the 1970s it was the standard operating system of the 8-bit world.

CP/M also wrote the grammar that personal operating systems would speak for decades: drive letters, files with names and extensions, a command line that answered to typed words. Kildall had built, almost as a side project, the layer on which the microcomputer industry stood. In 1980 the largest computer company on earth decided to enter the personal market, and it came shopping for exactly that layer.

CP/M: the first portable boss

Before the personal computer had a standard, it had Gary Kildall. Kildall, a Navy instructor and Intel consultant, wrote CP/M - Control Program for Microcomputers - in 1973 for Intel's Intellec-8 development system, after Intel itself declined to sponsor the work [O56]. The design insight that made CP/M dominant was a clean separation: a small hardware-specific layer, the BIOS, carried the machine-dependent code, while the rest of the system stayed identical across machines. Any manufacturer could port CP/M by rewriting one modest module. By the end of the 1970s it ran on hundreds of different computers - Altairs, IMSAIs, Kaypros, Osbornes - and supported an application catalogue (WordStar, dBase, SuperCalc) that buyers treated as the reason to purchase hardware. Digital Research Incorporated, the company Kildall founded in 1976 to sell it, was the first great personal-computing software firm [O56]. The lesson the industry drew - own the layer between hardware and applications and you own the market - was about to be taught again, to Kildall's cost.

The Altair and the hobbyist moment

The demand side of the story was the Altair 8800, the MITS kit computer on the January 1975 cover of Popular Electronics, which sold thousands of units to hobbyists who had never expected to own any computer at all. Two young Seattle programmers, Bill Gates and Paul Allen, called MITS within days of the cover to sell a BASIC interpreter for it; Microsoft was founded on that bluff. The early personal-computer software trade had the moral texture of the SHARE era - programs swapped freely at clubs like the Homebrew Computer Club - and Gates's 1976 "Open Letter to Hobbyists", complaining that copied BASIC was theft, was an early sign that the commons era would not last. The desk machine created a mass market for software, and mass markets, as IBM had already discovered, end up owned by whoever controls the standard.

VisiCalc and the killer application

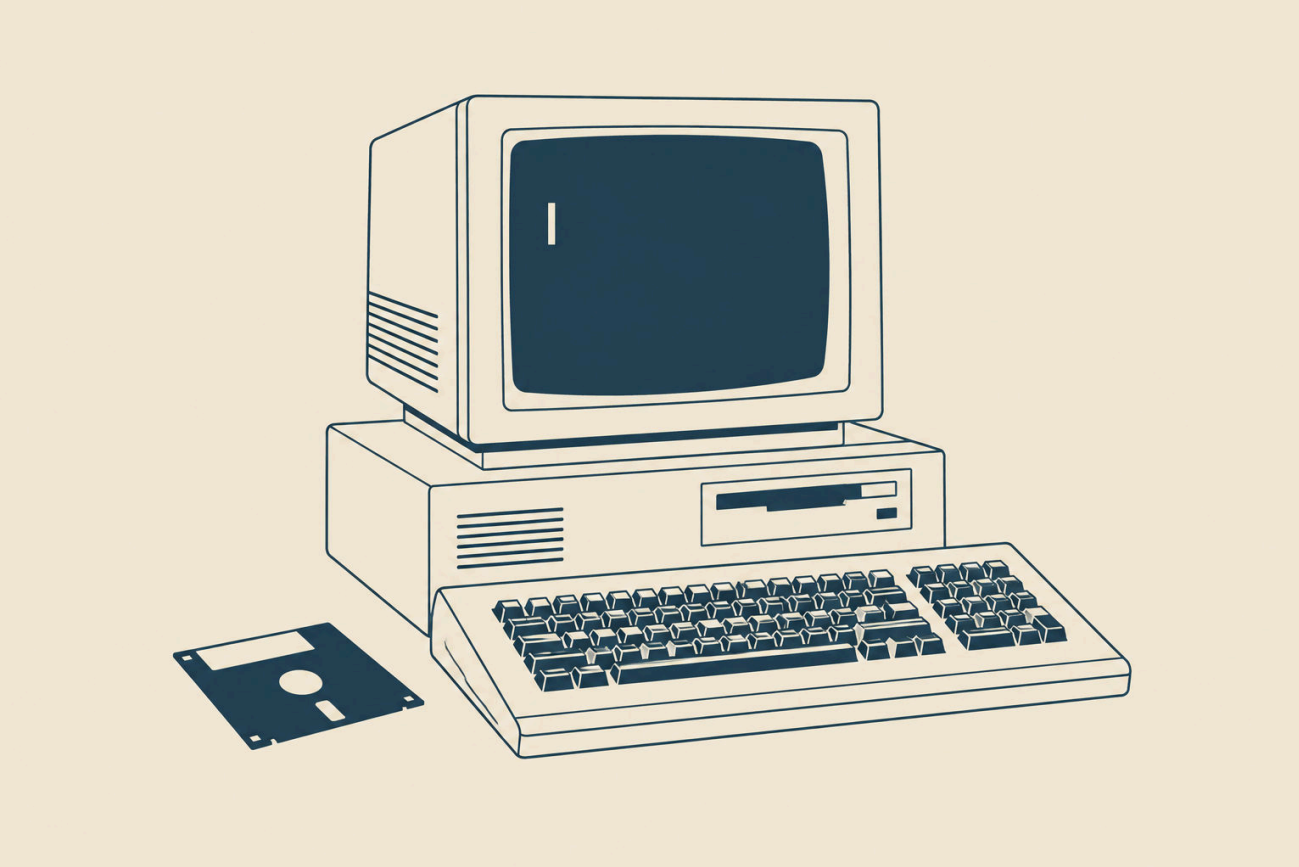
The personal computer's demand shock came from software, not hardware. VisiCalc, the 1979 spreadsheet written for the Apple II by Dan Bricklin and Bob Frankston, was the first program that justified buying the machine it ran on: small-business owners purchased Apple IIs to run it, the first recorded instance of the "killer application" pattern that would govern platform economics thereafter - Lotus 1-2-3 for the IBM PC, Office for Windows, the browser for the Internet PC, and the app ecosystem for the phone. The pattern matters to this book because it fixes the operating system's true product: not its own features but its applications' abundance. Every platform decision since - Apple's 500-app store launch [O30], Google's developer-first Android tools, Microsoft's compatibility obsession - descends from VisiCalc's demonstration that buyers purchase the machine to get the program, and the program exists only where the platform's owner made it easy and profitable to write.

The home-computer interlude

Between the Altair and the IBM PC lay the home-computer boom - Commodore PET, Apple II, TRS-80, then the Spectrum and Commodore 64 - machines whose operating systems were BASIC interpreters burned into ROM, booting instantly to a programming prompt. A generation's first experience of computing was that blinking cursor: the machine waiting to be told, in a language a child could learn. The home boom collapsed in the mid-1980s price war and the PC's standardization, but its legacy is measurable in the demographics of the software industry it seeded. It also fixed an expectation the industry later forgot and the smartphone restored: that the machine starts instantly, without ceremony. The pocket mainframe's instant-on is the home computer's ROM BASIC, reborn as a hardware requirement.

Chapter 12 - The seventy-five-thousand-dollar accident

The story of how IBM's PC got its operating system has been polished by legend; the version that survives documentary scrutiny is remarkable enough. In July 1980, IBM contacted Microsoft about supplying programming languages for its secret personal computer project. IBM also needed an operating system, and Microsoft's Bill Gates pointed Boca Raton toward Kildall's Digital Research, whose 16-bit CP/M-86 was in preparation [O56]. IBM and Digital Research failed to agree on licence terms - on exactly why, the record holds competing memories and one documented outcome: no deal [O56].



The desk computer and its blinking cursor: the market QDOS was written for, and DOS came to own.

Microsoft, now on the hook for an operating system it did not have, found one in its own neighbourhood. Tim Paterson at Seattle Computer Products had, in April 1980, written a 16-bit system that mirrored CP/M's function calls - he called it QDOS, the Quick and Dirty Operating System [O56]. On 6 January 1981 Microsoft licensed it for $25,000, obtaining the right to sublicense it, without disclosing that IBM was the customer. Then, just before the PC's launch, Microsoft bought the system outright for a further $50,000 on 27 July 1981 [O56]. Total price: $75,000.

The IBM Personal Computer appeared on 12 August 1981, and buyers could choose PC DOS - Microsoft's renamed purchase - as its operating system [O56]. The deal's decisive feature was not the price but the retained rights: IBM received a royalty-free licence, while Microsoft kept ownership and, crucially, the right to license the same system, as MS-DOS, to every other manufacturer [O56]. When the IBM design became an industry standard and the clone makers arrived - Compaq first and most famously, then a flood - each clone

needed the operating system, and each paid Microsoft. The PC belonged to IBM; its toll booth belonged to Microsoft.

It is hard to find a better illustration of the operating system's strange economics. The software itself was, by design, quick and dirty - a small disk operating system in the CP/M mould, no multiprogramming, no memory protection, nothing that OS/360's engineers would have admired. Its power came not from its sophistication but from its position. Whoever owns the layer every application must sit on owns the market above it. Kildall's CP/M had proved the layer's value; Microsoft understood, as Digital Research did not, that the layer was the business.

The meeting that did not happen

The events of 1980 are among the most retold in business history, and the Digital Research corporate chronology fixes the verifiable bones [O56]. IBM's Boca Raton team, building the secret "Acorn" personal computer, needed an operating system and first approached the market leader: Digital Research, for CP/M. The negotiations failed - accounts differ on whether Kildall flew that day, on the non-disclosure terms, on Dorothy Kildall's reluctance to sign - but the fact of failure is not in dispute. IBM turned back to Microsoft, which was supplying BASIC. Microsoft did not own an operating system either, but knew who had written one: Tim Paterson at Seattle Computer Products, whose QDOS - "Quick and Dirty Operating System", written in April 1980 as a stopgap while waiting for a CP/M-86 that was late [O56] - cloned CP/M's interface for Intel's 8086. In January 1981 Microsoft licensed QDOS for $25,000; on 27 July 1981 it bought it outright for $50,000 more, without disclosing the customer [O56]. Seventy-five thousand dollars, for what became the most valuable software franchise in history.

Paterson's six weeks and their consequences

Fairness to Paterson matters, because QDOS was not a trivial knock-off: he implemented CP/M's application interface on a different processor with a different file system - FAT, the File Allocation Table, which Microsoft had developed for its BASIC and which would survive, in descendants like FAT32, into the flash drives of the 2020s. Paterson later joined Microsoft. Kildall, for his part, maintained until his death in 1994 that DOS copied his work; the industry litigated the question only in memoirs. What is certain is the structural outcome: the operating system for the machine that would define personal computing was written in six weeks by one engineer as a placeholder, and the company that owned the placeholder - not the company that built the machine - captured the industry.

Kildall's afterlife

Gary Kildall's story did not end in 1981, and its remainder is part of the record. Digital Research continued to innovate: DR DOS competed credibly with MS-DOS through the early 1990s - Novell bought the company in 1991 largely for that fight - and the subsequent antitrust litigation (Caldera v. Microsoft, settled in 2000 for a confidential sum reported around $150-280 million) alleged that Microsoft had used vaporware announcements and per-processor licensing to smother it. Kildall himself sold out, hosted television's Computer Chronicles, and died in 1994 at fifty-two. The industry's memory of him has been contested ever since: the Microsoft-era narrative treated him as the man who "went flying" and fumbled history; the engineers' memory is of the person who invented the personal computer's operating system, the BIOS portability trick, and early versions of everything from disk caching to preemptive multitasking on micros.

Both memories are true. The lesson the industry metabolized was not “be first” or “be best” but the colder one this book keeps meeting: control the licence, and history is written by the licensor.

What IBM thought it was buying

It is worth reconstructing IBM’s side of the 1980 negotiation, because the mistake was a category error, not stupidity. IBM’s culture priced components, and it treated the operating system as a component: necessary, cheap, undifferentiated - like the power supply. What the PC’s architects did not model was that software, unlike hardware, has zero marginal cost and network effects: the same program on every machine becomes a standard, and a standard’s owner collects rent from the whole market. IBM had itself demonstrated the principle with System/360’s architecture (Chapter 4) and then forgotten it in Boca Raton, because the PC was a side project on a one-year schedule, built from outside parts by a team exempted from IBM’s normal controls. The exemption made the product fast and the concession invisible. By the time IBM understood, in the late 1980s, that the value had migrated to the software layer, the OS/2 effort to reclaim it was already late - and Windows was already shipping.

Chapter 13 - DOS, the toll booth

Through the 1980s MS-DOS became the most widely used operating system on earth by the simple mechanism of riding every clone. The machine makers competed on price and hardware; Microsoft collected from each of them, per machine shipped. The operating system market, such as it was, consolidated around the firm that had paid $75,000 for its entry ticket.

DOS itself evolved slowly - it remained a single-tasking system with the famous 640-kilobyte ceiling - but that conservatism was a feature: software written in 1981 still ran in 1991, and the installed base of programs became the deepest moat in computing. Competitors tried. Digital Research answered with DR DOS, technically respectable and cheaper; it won some makers and then withered under commercial pressure so contested that Digital Research’s successor eventually sued, and the antitrust files from that case are among the best surviving records of how the DOS market actually worked [O56].

Two durable lessons came out of the toll-booth decade. The first is that in operating systems, compatibility beats elegance. DOS won not because it was good but because everything ran on it, and everything ran on it because it won - the self-reinforcing loop that economists now call a network effect and that every later platform war would reproduce. The second is that the visible machine is a commodity and the invisible layer is the prize. The clone makers fought savage price wars and many died; the company that sold each of them the same forty-dollar licence outlasted almost all of them.

But DOS had a ceiling of its own: it was ugly. It spoke in commands typed into a black screen, and the wider world would never love a machine that answered back in cryptic prompts. The next fight was not about who owned the layer but about what the layer looked like - and it had begun, improbably, in a photocopier company’s research lab.

The royalty machine

The IBM agreement’s crucial clause was the one IBM’s lawyers barely noticed: Microsoft retained the right to license the operating system to others [O56]. IBM, which thought of itself as a hardware company and had the

1969 antitrust suit still open in the background, conceded the software layer to a 30-person vendor. When the IBM PC shipped in August 1981, its open specification invited clones; the clone makers - Compaq first, then dozens - needed PC-compatible software, and Microsoft sold them MS-DOS on identical terms, per machine shipped. The economics were superb: zero marginal cost, prepaid royalties, and a customer list that grew every time IBM succeeded or was copied. By the mid-1980s the “IBM PC standard” had quietly become the MS-DOS standard, and IBM discovered that it had built a market whose gate was owned in Redmond. The per-processor licence of the early 1990s - clone makers paying for DOS on every machine whether shipped with it or not - later became Exhibit A in the antitrust record [O67], but the foundation was the original 1981 paper: the boss was sold by the seat, and every desk became a toll.

What DOS actually was

Technically, DOS was modest, and the modesty was the point. It fit on a floppy, booted in seconds, and left the hardware almost bare: a file system, a command line, a loader, and the BIOS services. It had no memory protection, no multitasking, one user, one program at a time - less capability than CTSS had in 1961. Critics measured it against Unix and found a toy; customers measured it against a typewriter and found a revolution. DOS’s real function was standardization: it gave WordPerfect, Lotus 1-2-3, dBase and Turbo Pascal one stable target, and the applications, not the operating system, were what buyers fell in love with. The lesson recurs throughout this book: the winning base layer is not the most sophisticated one but the one that makes the most other things possible, cheaply, soon.

The clone wars

DOS’s franchise was tested once, seriously, and the test fixed its dominance. Compaq’s 1983 Portable - the first legal IBM-compatible - required reverse-engineering IBM’s BIOS, done by clean-room separation: one team read IBM’s published code and wrote a specification, another wrote new code from it. The technique legitimized an industry; within five years, clones outsold IBM’s own PCs, and every clone shipped MS-DOS. Phoenix Technologies commercialized the clean-room BIOS so any maker could join. IBM’s counter-move, the PS/2 line of 1987 with its proprietary Micro Channel architecture and OS/2 partnership, attempted to re-proprietarize the standard it had created; the market declined, preferring the open bus and the old operating system. The lesson inscribed itself into every platform strategy since: once an ecosystem forms around an open specification, its creator cannot reclose the gate without handing the ecosystem to whoever keeps it open. Microsoft kept it open, collected the toll, and let IBM learn the price of its own invention twice - once with the architecture, once with the bus.

The DOS that would not die

DOS’s technical afterlife is longer than its product life. PC DOS 2000 closed the IBM line; MS-DOS 8.0 shipped inside Windows ME in 2000; and then the franchise dissolved into heritage - FreeDOS, the free reimplementation begun in 1994, still maintains it for industrial controllers, embedded devices and retrocomputing. FAT, DOS’s file system [O56], became the world’s interchange format: every camera card, USB stick and EFI system partition speaks a descendant of Tim Paterson’s 1980 design. The boot process of every PC until the 2010s began in DOS’s real mode, a 16-bit ritual preserved in silicon for thirty years past DOS’s commercial death, because compatibility’s ratchet only turns one way. UEFI finally retired the ritual. The base layer never really dies; it becomes the floor the next layer stands on, invisible and load-bearing, which is this book’s whole argument in miniature.

Chapter 14 - The friendly face

The graphical user interface - windows, icons, menus, a pointer - was invented in the 1970s at Xerox's Palo Alto Research Center, where the Alto computer (1973) and its commercial descendant the Star (1981) demonstrated that a machine could present its file system as a desktop of visible objects and its commands as things you pointed at rather than memorised. Xerox famously failed to convert the invention; the conversion happened down the road, at Apple. Steve Jobs toured PARC in late 1979, understood at once what he was seeing, and set Apple on the path that produced the Lisa in 1983 and, on 24 January 1984, the Macintosh.



The graphical face: overlapping windows, menus, and the mouse - the vocabulary the Macintosh taught the world.

The Macintosh mattered to operating systems for a subtle reason. Its System software was not powerful - a single-tasking system at first, with cooperative multitasking bolted on later, in which each application had to voluntarily yield the machine and a badly written one could freeze everything. What the Macintosh System proved was that the interface itself was now the product. The operating system stopped being the machine's private manager and became the machine's personality: the desktop metaphor, the trash can, the menu bar, the double-click. Microsoft saw it as clearly as Apple had seen the Alto. Within a year it had shipped its own graphical shell.

For the operating system story, the GUI era's deepest change is that usability became a mass-market requirement, and the mass market would now decide which operating systems lived. Technical merit had never been sufficient; from 1984 on, it was not even the main battlefield.

PARC’s invention and Apple’s price

The graphical interface was invented once, at Xerox’s Palo Alto Research Center, in the Alto of 1973: bitmapped screen, overlapping windows, mouse, menus, Ethernet, laser printing - the entire future in one lab, a decade early. Xerox famously failed to sell the future it had built. Apple’s December 1979 demonstrations - the visits in which Steve Jobs saw the Alto and, by his own later account, had its meaning “shown” to him - are the industry’s most consequential tour. Apple did not copy the Alto; it hired its people, including Larry Tesler, and spent five years re-engineering the ideas into something a consumer machine could afford. The Lisa (1983) proved the concept at $10,000 and failed; the Macintosh (January 1984) proved it at $2,495 and founded a religion. The Macintosh System Software’s real breakthrough was cultural: it asserted that ordinary people should not have to learn the machine’s language, that the machine should learn theirs. Everything since - Windows, the web browser, the smartphone - is a variation on that assertion.

The look-and-feel wars

Apple’s response to Microsoft’s pursuit was litigation: Apple v. Microsoft, filed 1988, argued that Windows 2.0 stole the Macintosh’s “look and feel”. The courts largely sided with Microsoft in 1992, holding that most of the contested elements were licensed, unprotectable ideas, or both. The judgment settled the industry’s rules: user-interface ideas could be imitated; only code and branding were fenced. On those rules, the graphical interface commoditized - which is why every desk and pocket now uses the same grammar of windows, icons and touch, and why the monopoly money moved from the interface’s inventor to its bundlers. Apple learned the lesson so well that its later iPhone strategy - own the whole stack, license nothing - reads as a deliberate inversion of the Macintosh mistake.

The 128K constraint

The original Macintosh was an engineering argument conducted under a memory ceiling. The 1984 machine had 128 kilobytes of RAM - less than a modern icon file - into which had to fit a bitmapped screen buffer, a graphical toolbox, the application and the document. The solution was the Macintosh Toolbox: the interface’s core routines burned into 64K of ROM, written in hand-tuned assembly by Andy Hertzfeld, Bill Atkinson and a small team who treated cycles like water in a desert. QuickDraw, Atkinson’s graphics engine, performed clipping and region operations at speeds that seemed impossible on the hardware, because the whole graphical illusion - windows overlapping, menus pulling down - depended on it never visibly stuttering. The modern OS instinct is to abstract and layer; the Macintosh was built the opposite way, every layer squeezed into the one below. It worked: the machine that reviewers called a toy ran a graphical interface more fluidly than workstations costing twenty times more, and the discipline of the Toolbox set the baseline expectation for every consumer interface after it.

The Finder and the desk metaphor

The Macintosh’s other inheritance is conceptual: the desktop. Files as documents in folders, the trash can, drag-and-drop, the clipboard - the metaphors were designed for people who had never seen a computer, tested on ordinary users in a way no system had been before. The metaphors were so successful that they outlived their purpose: forty years later, users who have never worked at a paper desk still save documents in folders and drag things to a trash icon. The smartphone’s grid of apps is the desk metaphor’s final reduction - the desk with no drawers, just the tools laid out. Interface metaphors are the operating system’s user-facing vocabulary, and the Macintosh wrote most of the words. When later chapters describe the antitrust fights over defaults and

stores, it is worth remembering that the first fight over this interface was about who was allowed to use the vocabulary at all.

Chapter 15 - Windows climbs on top of DOS

Microsoft’s first answer to the Macintosh was modest: Windows 1.0, released in November 1985, was less an operating system than a graphical environment laid over MS-DOS, its windows unable even to overlap [O58]. Windows 2.0 in 1987 let them overlap. The breakthrough was Windows 3.0 in May 1990, which sold in the millions and made the PC’s black command screen optional for ordinary work [O58]. Windows 3.1 and 3.11 consolidated the position; by the mid-1990s, “the computer” for most of humanity meant a PC running Windows - which still meant, underneath, DOS.

The climax of this lineage was Windows 95, launched on 24 August 1995 with the first true mass-media campaign for a piece of software: the Start button introduced to the sound of the Rolling Stones’ “Start Me Up”, licensed for a fee that rumour inflated to as much as $14 million before Microsoft’s retired chief operating officer set the record straighter [O59]. Windows 95 was, technically, a hybrid - 32-bit ambitions still partly running on DOS foundations - but it was the first high-volume version of Windows that did not sit on top of a separately installed MS-DOS, and Microsoft itself presented that integration as its defining feature [O57]. Windows 98 and the ill-fated Millennium Edition continued the line to its end in 2000.

The DOS-Windows family did something no operating system had done: it put one company’s layer on essentially every desk. By the late 1990s, more than nine out of ten personal computers ran Windows. That dominance would soon bring the regulators (Chapter 44). But Microsoft itself knew the old codebase could not carry the future. In parallel with the DOS line, it had spent years building a second, clean-slate operating system - one designed by a man recruited from the minicomputer world for exactly that purpose.

Windows 3.0 and the turning point

Windows spent its first five years as a curiosity. Version 1.0 (November 1985) tiled rather than overlapped its windows and required DOS underneath; version 2.0 (1987) added overlap and drew the Apple lawsuit [O58]. The product that changed the industry was Windows 3.0, May 1990: a usable interface, memory management that finally exploited the Intel 386, and Solitaire to teach the mouse. It sold four million copies in its first year. Microsoft’s distribution weapon was the OEM channel perfected for DOS - Windows preloaded on every clone - and its pricing made refusal irrational for manufacturers. Windows 3.1 (1992) and 95’s forerunners completed the conquest: by the mid-1990s Microsoft controlled both the operating system and, through Office, the application layer on top of it, a vertical grip no PC rival could match.

August 24, 1995

Windows 95 deserves its folklore. It was still DOS underneath (an MS-DOS 7.0 kernel wearing a graphical system), but the experience was genuinely new: the Start button, the taskbar, long filenames, Plug and Play. The marketing was unprecedented for software - the Rolling Stones’ “Start Me Up” licensed as the launch anthem (the press delighted in debunking the rumoured $8-14 million fee without ever settling the real number [O59]), the Empire State Building lit in the logo’s colours, midnight store openings, customers buying a software upgrade like concert tickets. The contemporary launch documentation [O57] shows a company teaching the public to want an operating system release as an event, a trick Apple would later perfect for

phones. Windows 95 sold seven million copies in five weeks. The consumer operating-system launch as mass spectacle - queues, countdowns, keynote theatre - dates from that August, and the industry has never fully stopped staging it.

The DOS underneath

Windows 95's engineering compromise deserves a clear statement, because it explains both its triumph and its eventual replacement. The system booted DOS, then loaded the 32-bit graphical system on top; old DOS and 16-bit Windows programs ran in a shared, unprotected memory space, so one misbehaving application could still freeze the whole machine - the infamous blue screen was the architecture's confession. Microsoft chose this deliberately: the installed base of DOS and Windows 3.x applications was the moat, and breaking compatibility (as NT demanded) would have drained it. The consumer line therefore carried 1981's kernel architecture - no memory protection, no privilege separation - into the Internet era, at exactly the moment machines began inviting hostile code from the network. The security record of Windows 98 and ME is that decision compounding. The merger into NT at XP (2001) [O58] was the correction: the compatibility moat had been rebuilt inside a mainframe-grade kernel, at the cost of thirteen years of consumer exposure. Platform transitions are always paid for; the question is only who pays, and when.

Cairo, Chicago, and the vapour wars

The competitive texture of the early 1990s was shaped by Microsoft's habit of announcing the future to freeze the present. Cairo, the promised object-oriented next-generation system, was announced in 1991 and never shipped as such; Chicago (Windows 95) shipped three years late. Competitors - IBM's OS/2, Novell's DR DOS, Apple's Copland-era Macintosh - spent those years competing against products that did not exist, while customers deferred purchases to wait for them. The tactic's legality was never tested directly, but its effect is in the record: OS/2, technically superior in its 2.0 (1992) incarnation, sold into a market that had decided to wait for Chicago. The vapour wars are the darker complement to the standards story: the same credibility that made Microsoft's promises valuable made its threats expensive. The antitrust case that followed [O67] charged the formal instruments - per-processor licences, exclusionary OEM terms, browser tying - but the informal instrument, the announcement, was just as effective and remains perfectly legal.

Chapter 16 - NT: an operating system built like a mainframe

Dave Cutler had already built VMS, the respected operating system for DEC's VAX minicomputers, when a cancelled DEC project sent him looking for a new home. On 31 October 1988 he started at Microsoft; the core of his DEC West team followed, giving notice on a Friday and starting in Redmond the next Monday [O60]. Their charter was the opposite of DOS: a modern, portable, secure, 32-bit operating system with protected memory, preemptive multitasking and a hardware abstraction layer that would let it run on processors beyond Intel's. The small team spent its first six months writing the specification - a document that now resides in the Smithsonian Institution [O60].

Windows NT 3.1 - numbered to match the consumer Windows of its day - shipped in July 1993 [O58]. It was everything DOS was not: a hybrid-kernel design in which applications could not crash the system, every process had its own protected address space, and the same source tree ran on Intel, MIPS, Alpha and PowerPC machines. NT 4.0 followed in 1996, moving the graphics system into the kernel for speed at some cost in

purity; Windows 2000 hardened the line for the enterprise; and in October 2001 Windows XP finally retired the DOS family, unifying consumer and business Windows on the NT kernel [O58].

From that point, “Windows” meant NT - on the desktop, on servers, and eventually in places DOS never dreamed of: the Xbox, the Azure cloud, cash registers, ATMs. The two families tell the whole dialectic of operating system design. DOS/Windows 95 grew from below: small, pragmatic, compatible, beloved of the market. NT descended from above: principled, portable, secure, designed by veterans of bigger machines. In the end the principled system won the technology while the pragmatic one had won the market - and Microsoft, uniquely, owned both halves of the sentence.

The NT lineage also carried an old-world inheritance inside it. Cutler’s design bore the stamp of VMS, itself a product of the minicomputer era that Unix had sprung from. The industry’s two great operating-system cultures - the DEC camp and the Unix camp - thus both survive, embedded, in the systems running today. The third culture, which no corporation owned, was about to announce itself from a student dormitory in Helsinki.

Cutler’s team and the VMS echo

Dave Cutler arrived at Microsoft in October 1988 from Digital Equipment, where he had led VMS, the VAX operating system, and he brought a core of DEC’s western-software engineers with him [O60]. NT - officially “New Technology” - was designed with mainframe habits: hardware abstraction through the HAL, a microkernel-flavoured executive, preemptive multitasking, security with a reference monitor, and portability targets that initially included the i860 and MIPS before settling on x86. DEC veterans noted the family resemblance to VMS so openly that jokes about “WNT = VMS+1” (increment each letter) became industry legend, and a 1993 settlement between Microsoft and DEC quietly covered the overlap. The resemblance is a compliment: NT was a workstation-class system built by people who had done it before, transplanted into a PC company whose existing product was a DOS extender.

One kernel to rule the product line

Windows NT 3.1 shipped in July 1993 [O58] aimed at servers and workstations; the consumer line (95, 98, ME) limped on its DOS ancestry for eight more years. The two lines merged in Windows XP (October 2001) [O58], when the NT kernel finally took over the consumer desktop - the point at which Microsoft’s whole franchise stood on Cutler’s architecture. That architecture then outlasted every prediction: Windows 11, the Xbox system software, and the kernel under Azure’s Hyper-V all descend from the 1993 codebase’s design. A mainframe engineer’s insistence on doing it properly - stability over compatibility, architecture over expedience - turned out to be the longest-lived software decision Microsoft ever made.

The specification in the Smithsonian

NT’s ambition is documented in an unusual place: Cutler’s original design specification is held by the Smithsonian, and Microsoft’s own retrospective feature on Cutler records the programme’s shape - begun 31 October 1988, built by a hand-picked team that grew to about two hundred, targeting RISC and x86 simultaneously to enforce portability [O60]. The design principles read like a rebuke to DOS: a hardware abstraction layer so the kernel never touches silicon directly; an executive with object-style management of processes, files and devices; a security model built for government evaluation from day one; and environment subsystems so the same kernel could host Win32, OS/2 and POSIX personalities - three operating systems’ applications on one kernel, a direct descendant of the compatibility logic that had made DOS valuable. The

subsystems faded (Win32 won), but the discipline remained: NT was the first Microsoft product designed to be maintained by strangers, decades into the future. It has been, and it is.

NT 3.1 to XP: the long merge

The consumer and enterprise lines ran in parallel for eight years, and the comparison is instructive. NT 3.1 (1993), 3.5, 4.0 (1996) and Windows 2000 built the server franchise - Active Directory, clustering, the enterprise sales motion - while 95, 98 and ME held the consumer desktop with DOS underneath (Chapter 15). XP (October 2001) merged them on the NT kernel [O58], and the merged product became the longest-lived desktop operating system in history: supported for twelve years, still visibly present in the installed base two decades later [O72]. The merge also settled Microsoft’s internal architecture argument permanently - one kernel, one driver model, one security model - and that consolidation is what made the later transitions (64-bit, Secure Boot, the cloud’s Hyper-V) possible at all. Consolidation at the base is what buys the freedom to change everything above it.

PART V - THE FREE STACK

The industry’s central institution in this part is not a company but a contradiction: free software. Richard Stallman’s 1983 announcement of GNU - a complete Unix-compatible system, free as in freedom - proposed to rebuild the most valuable kind of software there was and give it away, on the principle that users must be able to study, modify and share what runs their machines. Andrew Tanenbaum’s MINIX showed what a teaching kernel could be. And in 1991 a Finnish student announced a hobby kernel that would not be “big and professional like gnu”, and named it Linux.

The free stack won more completely than its champions predicted and less purely. The great 1992 debate between Tanenbaum and Torvalds - microkernel against monolith, design against development velocity - settled the architecture argument in public. The distributions, Debian and Ubuntu and a hundred others, turned a raw kernel into systems people could actually run, and Red Hat proved you could charge billions for supporting what was free. By the time the conquest was noticed, it was over: every one of the five hundred fastest computers on earth, the overwhelming majority of web servers, and the phone in most pockets ran on this stack. This part asks how software no one owns became the foundation everyone stands on - and what that means for who maintains it, who funds it, and who answers when it breaks.

Chapter 17 - GNU: freedom as method

Richard Stallman worked at MIT’s Artificial Intelligence Laboratory through the 1970s and watched its hacker commons die: the lab’s own operating system was abandoned, colleagues left for companies, and code that had once circulated freely began arriving under licences that forbade sharing. His response, announced in September 1983 on Usenet, was a declaration of method: he would write a complete Unix-compatible operating system and give it away as free software - free not in price but in the user’s freedom to run, study, modify and share it [O61]. He called it GNU, a recursive pun: GNU’s Not Unix.

The GNU project produced, through the 1980s, superb components - the Emacs editor, the GCC compiler, the debugger, the shell, the core utilities - everything, in fact, except its own finished kernel. The GNU kernel, called Hurd, was ambitious and slow to arrive. Stallman’s other creation compensated: the GNU General Public License of 1989, whose copyleft clause says that anyone redistributing the software, modified or not, must pass on the same freedoms. The GPL turned generosity into an enforceable system - a licence engineered so that free code stays free, and it would soon matter far beyond GNU.

By 1990 the free world had a compiler, tools, utilities, an editor, a licence - and no kernel. The stage was set for someone to supply the missing piece without knowing he was doing it.

The announcement and the license

Richard Stallman’s initial GNU announcement, posted to net.unix-wizards in September 1983, is one of computing’s founding documents, and it is disarmingly plain: “I am going to write a complete Unix-compatible software system called GNU (for Gnu’s Not Unix), and give it away free to everyone who can use

it” [O61]. The method was as radical as the goal. Where SHARE had pooled utilities and the universities had shared Unix under license, Stallman designed a legal instrument - the GNU General Public License - whose “copyleft” clause made freedom sticky: you may copy, modify and redistribute, but you must pass on the same rights, source code included. Copyleft converted copyright, the industry’s fence, into the commons’ wall: code under the GPL can be enclosed only by breaking the license. The Emacs editor, the GCC compiler and the GNU C library that followed became the standard toolkit of serious Unix work, free or not.

Everything but the kernel

By 1991 the GNU project had built nearly an entire operating system - editor, compiler, debugger, shell, libraries, utilities - except the piece at the centre. GNU’s own kernel, the Hurd, had been begun in 1990 as an ambitious microkernel-based design and was proceeding slowly; the project’s history records the gap with rueful candour. The irony is structural: Stallman had correctly seen that freedom required a whole system, and the hardest single piece of a system is the kernel. Into that precisely shaped hole, in August 1991, dropped a Finnish student’s hobby kernel (Chapter 19). The combination - GNU userland, Linux kernel - is what the world now runs, and the argument over what to call it (“GNU/Linux”, Stallman insists) is really an argument over which half of the story matters more: the vision that defined the goal, or the accident that finished it.

The printer that started a movement

The origin story matters because it explains the licence’s absolutism. In the early 1980s, Stallman at MIT’s AI Lab wanted to fix a jam-prone Xerox printer’s driver - the lab had always patched its printers’ software - and Xerox refused to supply the source code. A colleague who had signed a non-disclosure agreement refused to share it. The episode, small in itself, taught Stallman the shape of the new world: software was becoming property, and property meant strangers could forbid you from fixing your own tools. The AI Lab’s own collapse - its hacker community hired away by Lisp-machine startups whose licenses forbade the old sharing - completed the education. GNU was designed as a system that could never be enclosed the way the lab’s world had been: the GPL’s share-alike clause is, in form, a promise extracted in advance from everyone who touches the code. Stallman’s biography is unusual among this book’s founders in that the money was never the point; the point was that users should be able to see and change what their machines do. The industry’s later adoption of free software for commercial reasons (Chapter 21) did not soften his position, and the split between “free software” (a moral stance) and “open source” (a development method, named in 1998) records the movement’s own fork.

The compiler that compiled the world

GCC - the GNU Compiler Collection - deserves its own paragraph in any history of the base layer, because it quietly became the industry’s foundation slab. Stallman released it in 1987; by the mid-1990s it was the standard compiler on every Unix variant, the enabler of every free operating system, and the toolchain that built the Internet’s servers. Its existence altered the economics of new hardware: any company designing a processor could port GCC and immediately inherit the entire free-software ecosystem, which is part of how ARM’s ecosystem and later RISC-V’s (Chapter 43) became viable. The kernel itself is GCC-compiled to this day (with Clang recently admitted as an alternative). The lesson repeats the book’s deepest pattern: the most strategic software is the software nobody thinks of as a product. The compiler is the operating system’s silent partner, and it, too, was built by the movement that believed tools should be free.

Chapter 18 - MINIX and the teaching kernel

Andrew Tanenbaum, a professor in Amsterdam, faced a pedagogical problem in the mid-1980s: Unix had become commercially restricted, and students could no longer read its source. His answer was to write a small Unix-compatible system for the IBM PC and publish every line. MINIX, released in 1987, was about 12,000 lines of code with a memory manager, a file system and core Unix utilities, its system-call interface compatible with Unix Version 7; the source accompanied his textbook Operating Systems: Design and Implementation [O20]. For the first time, anyone who could buy a book could study a complete working operating system. Students worldwide did exactly that; the ACM, honouring Tanenbaum, credits MINIX with instructing a generation - and notes its most famous side effect: it inspired the initial development of Linux [O20].

MINIX was also an architectural statement. Tanenbaum built it as a microkernel: the kernel proper did the bare minimum - messaging, low-level scheduling, interrupt handling - while the file system and device drivers ran as ordinary user processes. A crashed driver could, in principle, be restarted without crashing the machine. This was the opposite of the Unix tradition, in which everything lived in one big kernel, and the disagreement would soon be fought out in public, in front of the entire internet, between Tanenbaum and a student who had bought a PC to run MINIX on.

Tanenbaum’s teaching tool

Andrew Tanenbaum, an American computer scientist at Amsterdam’s Vrije Universiteit, wrote MINIX in 1987 because AT&T’s tightening Unix licenses had made real source code unusable in the classroom [O20]. MINIX was a from-scratch Unix workalike, small enough to teach - about twelve thousand lines - and architecturally radical for its day: a true microkernel, with the file system and device drivers running as ordinary user processes, so a driver crash need not take down the machine. Its book-and-source package, Operating Systems: Design and Implementation, taught a generation of students, and its influence reached further than anyone intended. Tanenbaum’s ACM biography notes MINIX’s role as the inspiration for Linux [O20].

The professor’s restraint

MINIX’s licensing history prefigures the open-source era’s central tension. Tanenbaum initially distributed it under a permissive publisher’s license - source available, modification restricted by book terms - and only in 2000 relicensed it freely under the BSD license. By then Linux, GPL-protected and aggressively developed, had taken the audience MINIX educated. Tanenbaum’s public position was always that MINIX was a teaching instrument, not a competitor; his debate with Torvalds (Chapter 20) was architectural, not commercial. But the episode teaches the same lesson as CP/M and BSD: the technically superior or earlier system does not inherit the earth. The one with the right license, momentum and timing does.

The classroom that built an industry

MINIX’s influence operated through a specific mechanism: the book. Tanenbaum’s Operating Systems: Design and Implementation paired the theory with complete, compilable source code - every line of a real system, printed and explained. Universities worldwide built courses on it; students read an actual scheduler, an actual file system, an actual memory manager, instead of pseudocode. The approach created a common technical literacy across a generation of systems programmers, and when the Internet connected that

generation in the early 1990s, the audience for a free, hackable Unix was already trained. Torvalds was one of those students - he bought the Tanenbaum book specifically to learn operating systems, ran MINIX on his own 386, and built Linux partly out of frustration with MINIX's restrictions. The student surpassing the teacher is the oldest story in universities; it is worth recording that the teacher, in this case, supplied the tools for his own surpassing, and said so with grace [O20].

A microkernel's quiet revenge

The coda is one of computing's better ironies. From roughly 2008 onward, Intel processors have carried a Management Engine - a separate small computer inside the chipset, running whether the main machine is on or off, for remote administration and security functions. Its operating system, security researchers revealed in 2017, is a modified MINIX. Tanenbaum responded with an open letter noting, with characteristic dryness, that MINIX might therefore be "the most widely used computer operating system in the world" - and that no one had told him [O9]. The microkernel that lost the 1992 argument on performance grounds ended up exactly where its design logic said it belonged: small, isolated, and trusted with the machine's most sensitive control functions. Chapter 37's settlement - big kernels for capability, small kernels for trust - was being implemented by the market even where nobody could see it.

The syllabus that outlived the lesson

MINIX's deeper contribution may be curricular: it fixed what "an operating systems course" means. Before MINIX, the subject was taught from theory and vendor manuals; after it, from a working system small enough to hold in one head. The tradition produced not just Linux but a culture of reading real code as the final authority - the culture visible in every "the source is the spec" argument on every kernel mailing list since. When Tanenbaum and Torvalds argued in 1992 [O19], they were, in a sense, the syllabus arguing with its most gifted student, in public, for the field to watch. Computer-science education's centre of gravity - build a small real thing rather than study a large abstract one - is MINIX's lasting architecture. The microkernel's revenge (this chapter) is the joke; the syllabus is the legacy.

Chapter 19 - Linux: "just a hobby"

On 25 August 1991, a 21-year-old University of Helsinki student posted to the comp.os.minix newsgroup. Linus Benedict Torvalds announced "a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones", asked for things people liked and disliked in MINIX, and noted that his system "somewhat resembles" MINIX but that it had already been ported to no other machine and "probably never will support anything other than AT-harddisks" [O21]. Every hedged prediction in the message turned out to be wrong.

Torvalds had started by writing a task-switching experiment for his new 386, grown it into a terminal emulator, and then - because he wanted to read files and save work - into a kernel with a file system. He posted version 0.01 in September 1991. Early adopters contributed fixes; in early 1992 he relicensed the system under the GNU GPL, and the two incomplete projects - GNU's tools without a kernel, Torvalds's kernel without a full toolset - clicked together like halves of a hinge. The combination, GNU plus Linux, was a complete, free, modern Unix for ordinary PCs: no licence fee, no restricted source, no vendor.

The timing deserves attention. In 1991-92 the free-Unix world was frozen by the USL-BSDi lawsuit; AT&T's System V was expensive; Microsoft's NT was a year away; the 386 PC was suddenly powerful enough for real work. Into that gap walked a hobby project that explicitly promised to stay unprofessional. It grew by the mechanism Stallman had designed: anyone could take it, improve it, and had to share the improvements. Within a few years the hobby had distributions, companies, and the first faint outlines of an empire - and its first famous public argument, about whether its very architecture was obsolete.

The comp.os.minix post

The founding document of the modern software commons is 428 words long. On 25 August 1991, Linus Torvalds, a 21-year-old University of Helsinki student, posted to the comp.os.minix newsgroup: "I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones" [O21]. The post asked MINIX users what features they liked, admitted the system was "not portable" and might never support anything but his own hard disk, and signed off with the humility of a side project. Every qualifier proved false. The original thread is preserved in Google's archive of the group [O21], and it rewards reading precisely because it is so unprophetic: the largest collaborative engineering project in human history announced itself as a footnote.

Version 0.01 and the snowball

The first upload, in September 1991, was ten thousand lines and booted to two shells on one machine. What made it grow was not the code's polish but its licence and its cadence: Torvalds adopted the GPL in early 1992 (he has called it the best decision he ever made), and he released constantly - "release early, release often" before the phrase existed - folding strangers' patches back within days. Contributors arrived for the reason economists later struggled to model: each needed the thing for their own work, and sharing the improvement cost nothing. Within two years the system ran X Windows, TCP/IP and the GNU toolchain; within five, it was a credible server; within ten, it was the dot-com boom's default engine. The hobby had become infrastructure, and the infrastructure belonged to no one, which turned out to be its commercial superpower: every company could bet on it without strengthening a rival.

Git: the tool the community needed

Linux's scaling problem was social before it was technical: by the early 2000s, hundreds of developers were feeding patches to one maintainer through a proprietary tool (BitKeeper) whose licence terms kept tightening. When the arrangement collapsed in 2005, Torvalds wrote a replacement in weeks: git, a distributed version-control system designed for the kernel's actual workflow - thousands of parallel branches, email-based review, no central authority required. Git's design is the kernel's politics in software form: every developer holds the whole history; trust is established by signature and review, not by server permission. It outgrew its origin almost immediately - through GitHub it became the default infrastructure of all software, open or not - and it completes the era's toolkit: the GPL made the code free, the Internet made collaboration free, and git made coordination free. The kernel's later velocity - a release every nine to ten weeks, seventy-five to eighty-eight thousand commits a year [O26] - runs on that stack. The hobby built its own factory.

Releases and the maintainer's discipline

The modern kernel's rhythm is worth one paragraph because it is the commons' answer to the certification cultures of Part VII: a two-week merge window in which new work lands, then seven or eight weekly release

candidates of stabilization, then release - nine times a year, for two decades, with long-term-support kernels maintained for years after. The discipline is enforced socially: Torvalds’s release notes, sometimes profane, are the industry’s most-read code reviews, and the maintainers below him run their subsystems with comparable rigour. The system’s weakness is the mirror of its strength: everything depends on the judgment and stamina of a few hundred maintainers, and the 2018 episode in which Torvalds temporarily stepped back to work on his own conduct showed how personalized the commons’ governance still is. Institutions accrete slowly around talent. The kernel is now old enough - thirty-five years - to be facing the same succession questions as every other institution in this book.

Chapter 20 - The great debate

On 29 January 1992, Tanenbaum posted a message titled “LINUX is obsolete” to comp.os.minix [O19]. His case was architectural: Linux was a monolithic kernel - file system, drivers, everything inside one privileged blob - and writing such a thing in 1991 was, he said, “a giant step back into the 1970s”. Worse, Linux was welded to the x86 line, which would surely be superseded; the future belonged to portable microkernels like GNU’s Hurd, which Tanenbaum predicted people would be running within five years [O19].

Torvalds’s reply was unvarnished: MINIX was the toy, Linux worked, the kernel was free, and portability could wait until someone needed it. Others piled in. Theodore Ts’o, an early Linux contributor, delivered the line that settled the argument in practice if not in theory: “Linux is here, and GNU isn’t” - and people had been working on Hurd far longer than Torvalds had been working on Linux [O19].

Three decades allow a verdict with evidence, and it is more interesting than either side expected. On the server, the phone, the supercomputer and the embedded device, monolithic and hybrid kernels won completely: Linux and the NT/XNU hybrids run nearly everything, valued for speed and pragmatism. Tanenbaum’s specific prediction - the x86 would be superseded, Hurd ascendant - failed; Linux outported everyone, now running on more processor families than any system in history. But the microkernel did not die. It retreated to the places where reliability outranks features: QNX runs hundreds of millions of car computers, seL4 carries mathematical proof of correctness into military helicopters, and MINIX itself lives on inside Intel’s processors’ management engine. The debate’s real legacy is the question it canonised - how much should a kernel do? - which every new system, from cloud microVMs to verified kernels, still has to answer.

The debate, on the record

On 29 January 1992, Andrew Tanenbaum posted “LINUX is obsolete” to comp.os.minix, and the most famous technical argument in operating-system history began [O19]. Tanenbaum’s case was architectural: monolithic kernels were “a giant step back into the 1970s”; tying Linux to the Intel 386 made it unportable; microkernels, with drivers in user space, were the research consensus and the future. Torvalds, three weeks from adopting the GPL and in no mood for reverence, replied that MINIX’s own design choices had their costs, that Linux existed and worked while the microkernel future did not, and that being free beat being theoretically right. The exchange ran for weeks and is preserved in full [O19]. Both men were right in their frame: Tanenbaum about where the theory pointed, Torvalds about what actually gets used.

What the argument decided

History's verdict arrived in pieces, and Chapter 37 assembles them. In the 1990s, performance killed the pure microkernel: Mach-based systems, including early NeXTSTEP, suffered message-passing overheads that monolithic rivals did not, and the winning commercial systems (NT, Apple's eventual XNU) hybridized instead. In the 2000s, the reliability argument resurfaced where performance mattered less than correctness: QNX in cars, seL4's formal proof, and MINIX itself inside Intel's chips. And Linux, the "obsolete" monolith, absorbed nearly every useful idea from the other side - loadable kernel modules, user-space drivers, and eventually the eBPF virtual machine that lets programs run safely inside the kernel. The debate's real content was never architecture but method: designed-by-committee elegance versus released-today utility. The industry keeps re-running the same argument - cathedral versus bazaar, proof versus shipping - and it keeps reaching the same mixed verdict.

The sequel nobody scheduled

The 1992 debate never really ended; it forked into industry practice. Mach, the microkernel Tanenbaum championed in spirit, became the core of Apple's XNU after NeXT - but only after its designers bolted BSD directly into the kernel's address space, conceding the performance point. The Hurd, GNU's own microkernel, remained a research project; Debian still ships an experimental GNU/Hurd port, a living fossil of the argument's designed-elegance side. Meanwhile the "obsolete" monolith grew the largest development community in software history - 1.3 million commits from over 29,000 authors by the end of 2024 [O26] - while quietly absorbing microkernel ideas wherever they proved out: loadable modules, user-space driver frameworks, FUSE file systems, and the eBPF sandbox. And the purest vindication of Tanenbaum's reliability argument, seL4's mathematical proof of correctness [O52], arrived from researchers who cited exactly the tradition he represented. The debate's final form is not a verdict but a division of labour, and both professors - the one in Amsterdam, the one in Helsinki - can claim to have been right about the part they cared most about.

What the fight looked like from inside

The debate's texture is worth one more note, because it set the culture of the commons. It was conducted in public, in plain text, between a professor and a student, and it was won on evidence - running code - rather than seniority. Torvalds's own later reflections are notably gracious: he has said that without MINIX he would never have started, and Tanenbaum has said he is proud of what his student-built audience produced. The argument became part of Linux's founding myth because it dramatized the commons' core claim: that in software, authority must re-earn itself against working code, continuously. Every mailing-list flame war since, every kernel maintainer showdown, every rejected patch with a technical justification, is a descendant of that January argument. It is not always polite. It has, on the evidence, mostly worked.

What the debate actually decided

Tanenbaum's opening post is worth reading in full, because its charges were specific rather than rhetorical [O19]. Linux, he wrote, was a monolithic system - "a giant step back into the 1970s," like taking a working C program and rewriting it in BASIC. It was tied closely to the 80x86, "not the way to go," when MINIX had already been ported from the Intel line to the 680x0, SPARC and the NS32016. And it was premature, since in his view the GNU system would arrive soon enough. Torvalds answered point by point, with the impatience of a twenty-one-year-old whose hobby had just been called obsolete in public: MINIX itself had first been

written for a specific machine, the code that existed beat the code that was promised, and “if the GNU kernel had been ready last spring, I’d not have bothered to even start my project.”

The deeper thing the debate decided was social, and neither participant fully saw it. Tanenbaum assumed that serious systems were built by organized teams with an architect’s authority; Torvalds demonstrated that a loosely coordinated crowd, given source code and a fast communication channel, could out-ship the organized team. The comp.os.minix thread is the founding document of that discovery - two visions of how operating systems get made, argued out in public, in the medium that would decide it. Every verdict the industry later reached on the technical question, Chapter 37 assembles; the social verdict arrived first, and it was not close.

Chapter 21 - Distributions and the Red Hat business

A kernel plus GNU tools is not yet something an organisation can run: someone must assemble, test, package and maintain the whole. That someone is a distribution. The first great ones appeared in 1993 - Slackware in July, and in August the Debian Project, founded by Ian Murdock as a fully volunteer, fully free distribution [O22]. The same year, Marc Ewing and Bob Young founded Red Hat. Each made a bet on how free software should be assembled; Red Hat added the more improbable bet on how it should be sold.

Red Hat’s model answered the question every sceptic asked: if the software is free, what is the business? The answer turned out to be certainty. Companies would not bet their payroll systems on a hobbyist’s download, but they would pay a subscription for a tested, certified, security-patched, supported Linux - the same bits anyone could download, wrapped in accountability. Red Hat Enterprise Linux became the default Linux of the corporate world; the company grew to a full-year revenue of $3.4 billion in fiscal 2019, up 15 percent on the year [O24]. On 9 July 2019 IBM completed its acquisition of Red Hat for $190 a share, roughly $34 billion - one of the largest software acquisitions ever made [O25]. IBM was explicit about what it was buying: position at the base of the hybrid cloud.

The family kept branching. SUSE anchored Linux in European enterprises. In October 2004, Mark Shuttleworth’s Canonical released Ubuntu 4.10, “Warty Warthog”, a Debian derivative tuned for human beings, which carried Linux onto more desktops and cloud machines than any other distribution [O23]. The pattern that Red Hat proved - give away the software, sell the confidence - became open source’s standard business, and it only works because the GPL guarantees that no one, not even Red Hat, can fence the software off again.

Debian and the social contract

Ian Murdock founded Debian in August 1993 with an explicit critique: existing Linux distributions were assembled carelessly, and a system this important deserved careful, principled, community maintenance [O22]. Debian’s innovations were institutional as much as technical: a package format (deb) with dependency management, a social contract and free-software guidelines that became the working definition of “open source”, and a constitution with elected leadership - a governance structure for a volunteer nation-state. Debian’s own history [O22] records the intent: not a product but a project. Three decades later, Debian remains the reference distribution, non-commercial by design, and the root of the largest branch of the family tree: Ubuntu, founded by Mark Shuttleworth and announced in October 2004 with the promise of a free,

polished desktop released every six months [O23], is a Debian derivative, as are hundreds of others, including the system running the International Space Station's laptops [O49].

Ubuntu and the polish question

Ubuntu's wager, from its "Warty Warthog" 4.10 release [O23], was that Linux's remaining barrier was not capability but finish: installation, fonts, hardware detection, sane defaults. It shipped free CDs worldwide, standardized the six-month release rhythm, and made "Linux on the desktop" at least plausible for the non-specialist. The desktop market never flipped - Chapter 42's numbers show why - but Ubuntu's deeper impact was in the cloud, where its server images became the default starting point for a generation of deployments, and in credibility: it demonstrated that a Debian-derived system could carry commercial support without commercial capture of the commons underneath.

The distribution as constitution

A distribution is more than packaging; it is a governance act. Every choice a distribution makes - which kernel version, which C library, which init system, which default desktop, which patches carried forward - is a constitutional decision about what its users' common platform is. Debian's institutional forms (constitution, elected leader, technical committee, public bug tracker) exist because a volunteer project holding the dependencies of thousands of downstream systems needs legitimacy, not just code. The init-system argument of 2014-2015 - Debian's vote to adopt systemd over the objections of a vocal minority - showed both the cost and the function of that legitimacy: the decision was contentious, fully public, and final, and the project survived it. Proprietary operating systems make these decisions in private, by org chart. The commons makes them in public, by argument, and the argument is part of the product: the record of why things are the way they are, available to anyone who needs to depend on them.

The long tail of forks

The distribution family tree now numbers in the hundreds, and its structure teaches the same lesson as the book's larger history. Most distributions are derivatives of three roots - Debian, Red Hat, and (increasingly) independent systemd-era builds - because maintenance is expensive and derivative work is cheap. Specialization happens at the edge: Raspbian for the Pi, Alpine for containers, Tails for privacy, Kali for security work, Android's AOSP for everything Google-adjacent. The economics mirror Chapter 4's: the architecture is the promise, the distribution is the fulfilment, and the long tail exists because the commons lets anyone keep a promise the mainstream has stopped making - a distribution for old hardware, for a language community, for a country. Proprietary platforms prune; the commons branches. Both strategies concentrate power at the trunk; only one of them lets the twigs survive.

The ecology around the kernel

What Debian, Red Hat and their successors actually sold or gave away was curation. A Linux kernel plus the GNU tools plus an X server plus a thousand libraries is not a product; it is a warehouse. The distribution is the editorial act: choosing versions that work together, writing the installer, keeping the security patches moving, and standing behind the result. Debian made that act a constitution. Its Social Contract, written in 1997, committed the project to remain entirely free software and to give its improvements back upstream, and its volunteer structure - roughly a thousand developers, no company, decisions by debate and vote - made it the nearest thing the industry has to a public utility [O22]. When the International Space Station's laptops moved

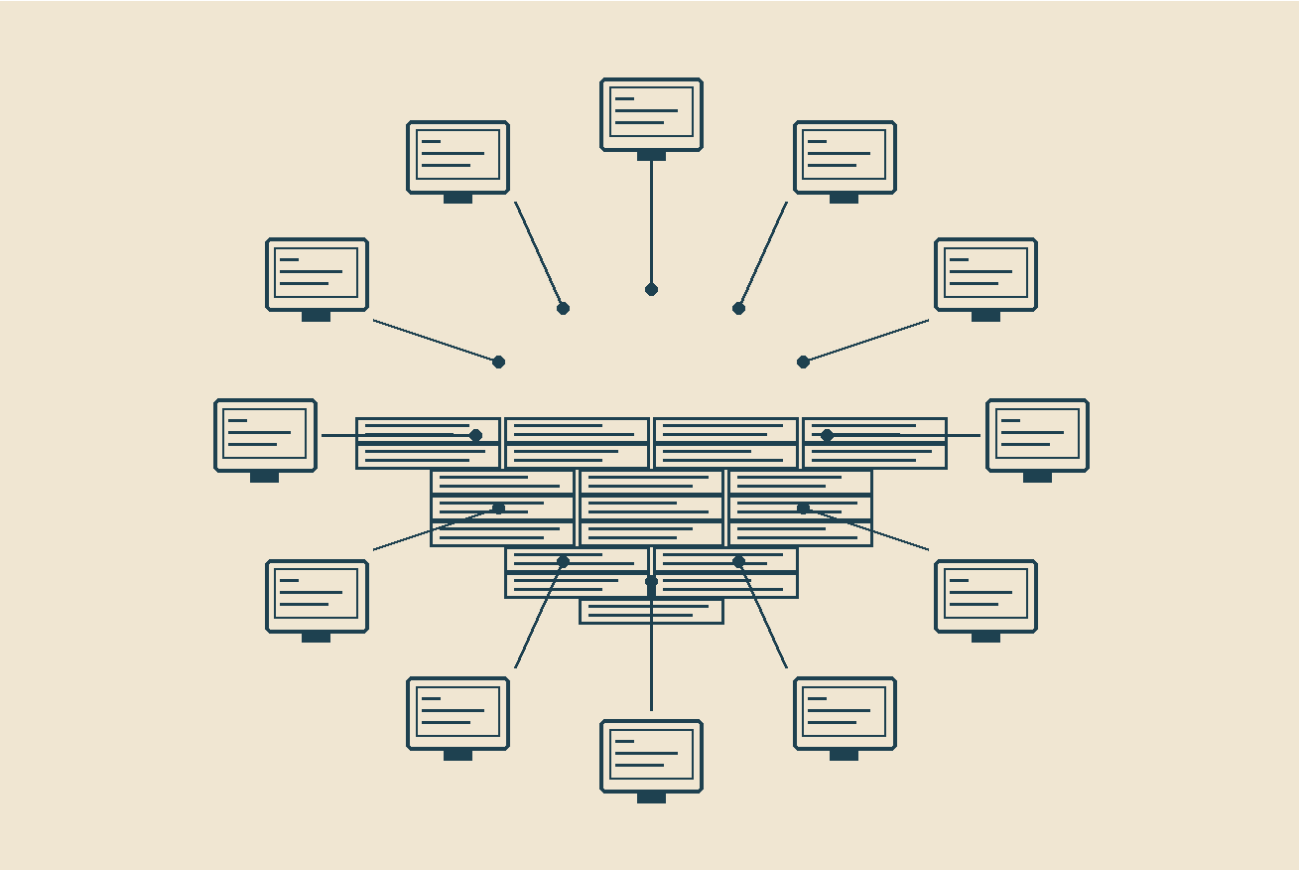
off Windows in 2013, they moved to Debian for exactly that reason: stability and an in-house ability to patch, with no vendor to negotiate with [O49].

Ubuntu made the opposite bet with the same raw material. Canonical, funded by Mark Shuttleworth and launched in 2004, took Debian's unstable branch, froze it on a predictable six-month rhythm, and aimed the result at people who had never compiled anything [O23]. The bet worked well enough that Ubuntu became the default Linux for cloud instances and developer laptops, and its long-term-support releases became infrastructure in their own right. Red Hat, meanwhile, had already proved the commercial version of the model: give the software away, sell the certainty. Its subscription to Red Hat Enterprise Linux was never really a license; it was a promise that someone accountable had tested this exact combination and would answer the phone for a decade [O24].

The 2019 verdict on that model was IBM's $34 billion purchase of Red Hat at $190 a share [O25] - the largest software acquisition in history at the time, and a strange homecoming. The company that had invented the operating system as a bundled accessory to hardware was paying a record sum for a firm whose product was given away free. What IBM bought was not code but trust: the editorial standing described above, accumulated release by release over twenty-five years.

Chapter 22 - The quiet conquest

Nobody voted for Linux to take over the world's machines; it happened in layers, each invisible to the layer above.



The commons at work: thousands of contributors, one shared edifice of code.

First the web. Unix-family systems - overwhelmingly Linux - run about 92 percent of the websites whose operating system can be identified [O5]. The LAMP stack (Linux, Apache, MySQL, PHP) made serving a web page nearly free, and the economics compounded: cheap servers made more websites, more websites made more Linux expertise, more expertise made Linux the default. Then the clouds: the virtual machines that Amazon, Google and Microsoft rent by the hour run mostly Linux, because it is free to replicate at any scale and nobody pays a licence per instance. Then the extremes: since November 2017, every single machine on the TOP500 list of the world's fastest supercomputers has run Linux [O6] - all one hundred percent, a shutout no proprietary system has ever achieved at any scale of computing.

Then places no one thinks to look. The International Space Station's laptops were migrated from Windows to Debian Linux in 2013 because, as the NASA contractor leading the migration put it, the crew needed an operating system that was stable and gave them in-house control to "patch, adjust or adapt" as required [O49]. On 19 April 2021, when NASA's Ingenuity helicopter made the first powered flight on another planet, it did so running Linux and JPL's open-source F Prime framework - the first Linux-powered flight on Mars [O48]. The New York Stock Exchange's trading systems, national railway networks, most of the world's phones (Chapter 24), the majority of car infotainment units (Chapter 29): the same kernel, everywhere, mostly unacknowledged.

The scale of the collaboration is worth a pause. At the end of 2024, the Linux kernel's source tree held about 39.8 million lines across more than 87,000 files, accumulated through 1.32 million commits from some 29,380 different authors [O26]. In 2024 alone it absorbed 75,314 commits and 3.7 million new lines, with Torvalds himself still its most prolific merger of other people's code, thirty-three years after the hobby post [O26]. No company employs this workforce; no single organisation could afford it. The operating system that was "just a hobby" is now the largest collaborative engineering artefact in human history, and it is, in the most literal sense, common property.

Red Hat and the discovery that free is a business

The company's insight, articulated by Bob Young and Marc Ewing in the mid-1990s, was that Linux's problem for businesses was not price but accountability: an enterprise cannot file a support ticket with a newsgroup. Red Hat sold the box, then the subscription - certified, tested, patched, indemnified, with an engineer who answers. The model matured into the numbers IBM eventually bought: $3.4 billion in revenue in fiscal 2019, growing at fifteen percent, with $3.0 billion of deferred revenue on the balance sheet - customers paying in advance for future certainty [O24]. The economics are subtle: Red Hat's engineers are among the largest contributors to the upstream kernel, so the company funds the commons it sells. The subscription buys not code but a promise about the code, and the promise is credible because Red Hat employs the people who keep it.

The acquisition and its meaning

IBM completed its purchase of Red Hat on 9 July 2019, paying $190 per share in cash - roughly $34 billion, the largest software acquisition to that date [O25]. The buyer's logic was strategic: IBM's own middleware had lost the cloud era, and Red Hat's OpenShift gave it an enterprise Kubernetes story with the one asset IBM still uniquely possessed, the trust of the world's largest IT departments. The market's verdict on the model was the multiple: a company selling subscriptions to free software commanded a price per share representing about ten times annual revenue. The lesson for this book's larger argument is direct. When the base layer is a commons, the business moves to the edges - support, certification, integration, cloud hosting - but it does not

shrink. Free software did not kill the operating-system business; it forced the business to sell something more durable than licences: stewardship.

The numbers of the conquest

The conquest is measurable in layers, and each layer's source is on the record. On the public web, Unix-family systems power about 92 percent of sites whose operating system can be identified, against Windows' 8 percent [O5]. In supercomputing, the sweep is total: since November 2017, every one of the TOP500 machines runs Linux [O6] - a category Unix workstations contested through the 1990s and Linux took entirely in two decades. In cloud instances, Linux's share is estimated by the providers themselves at well over three-quarters. The common explanation - "free as in price" - is half the story; the other half is adaptability. A system whose source can be cut down, tuned and rebuilt by the customer's own engineers fits every niche the proprietary vendors could not economically serve: routers, TVs, cars, satellites, supercomputers, stock exchanges. The quiet conquest was not won in the market Linux's founders cared about (the desktop) but in every market they did not know existed.

The desktop that stayed out of reach

The honest counterpoint is the one market where the conquest failed. Linux's desktop share has hovered in low single digits for twenty years - 5.6 percent as of November 2026 [O3] - despite free cost, capable interfaces and periodic prophecies of the "year of the Linux desktop". The reasons are this book's recurring machinery: the incumbent's application moat (Office, Adobe, games), OEM preinstallation economics, and the switching-cost ratchet. What changed the desktop's terms instead was the browser - as applications moved to the web, the desktop operating system mattered less, which is precisely why ChromeOS (a Linux variant) found its niche in education. And the desktop's long decline in the industry's centre of gravity meant the failure mattered less every year: Linux lost the desk and won everything behind the glass the desk looks through. Torvalds himself has said the desktop was never the point. The numbers say the point found him anyway.

PART VI - THE POCKET MAINFRAME

The computer’s second migration - from the desk to the pocket - replayed the first at ten times the speed. By 2006 the mobile industry had a settled order: Symbian’s empire on Nokia’s handsets, BlackBerry’s keyboards in every boardroom, Palm and Windows Mobile contesting the organizers. It looked like the mainframe era looked in 1978: permanent. Within eight years every one of those platforms was dead or dying, and the pocket had become the largest software market in history.

This part follows the fastest platform war ever fought. The iPhone’s 2007 announcement reset what a phone was - OS X shrunk into glass, with a real browser and a real operating system where the industry had shipped toys. Android, bought by Google for about fifty million dollars and called by its own executives the best deal ever, became the free ammunition of everyone Apple threatened, and took three quarters of the world’s handsets. The Java wars ran in the courts for a decade over whether an API could be owned. Nokia’s burning platform, BlackBerry’s secure silence, Palm’s elegant last throw - the losers are as instructive as the winners, because they were not stupid, and they lost anyway. And the settlement produced the tollbooth: two gatekeepers, thirty percent, and the antitrust reckoning that this book’s ninth part takes up in earnest.

Chapter 23 - Before the smartphone

The idea that a telephone should contain a real operating system predates the iPhone by a decade, and its history is a graveyard of companies that were right too early.

The British pioneer was Psion, whose organisers sold in the millions. For its Series 5 handheld in 1997, Psion built EPOC32, a proper modern system - protected memory, preemptive multitasking - written for devices that ran for weeks on two AA batteries [O27]. In June 1998, Psion joined with Ericsson, Motorola and Nokia to form Symbian Ltd, a joint venture to turn EPOC into the industry’s smartphone operating system [O27]. The design was prescient: the phone makers would compete on hardware while sharing the software layer, exactly the PC industry’s clone economics applied to mobile.

Palm took the opposite path: radical simplicity. The PalmPilot of 1996 and its Palm OS did few things - calendar, contacts, notes - instantly and reliably, synchronising with a PC through a cradle and one button. At its peak in 2000, Palm OS held 74 percent of its market [O27]. In Canada, Research In Motion’s BlackBerry fused a thumb keyboard with a operating system and network built around one killer use: email that arrived by itself, pushed to the pocket, encrypted well enough for presidents. And Microsoft, predictably, shrank Windows: Windows CE and its Windows Mobile descendants carried the Start-menu metaphor into phones, with Microsoft’s enterprise relationships as the lever.

By 2006 the mobile industry’s centre of gravity was Symbian, at about 73 percent of the smartphone market, in a year when the world bought 64 million smartphones - a number worth remembering, because it is roughly what a single strong quarter would add a decade later [O27]. Every incumbent was profitable, established, and convinced the game was about adding features to a telephone. None of them survived the next three years with their operating system intact. The smartphone as they understood it - an organiser that could make calls, or a

phone that could do email - was about to be replaced by something none of them had priced in: a full general-purpose computer whose operating system was the product, with the phone as merely one of its radios.

EPOC and the Psion school

The smartphone's operating-system lineage runs through a British handheld before it reaches California or Finland. Psion's EPOC, released in 1997 for the Series 5 organizer, was designed for constraints the desktop never knew: a small battery, a few megabytes of memory, and a user who expected the device to be instant-on for weeks [O27]. EPOC was 32-bit, preemptively multitasking, object-oriented, with a graphical interface - a pocket operating system engineered with a seriousness the desktop industry only later appreciated. Psion's problem was that it was a hardware company with a world-class software asset. The June 1998 solution was Symbian, a joint venture with Ericsson, Motorola and Nokia to carry EPOC forward as the industry's smartphone operating system [O27][O28]. The handset makers, watching Microsoft's PC toll booth from outside, deliberately chose a structure in which no single owner could tax them all. The design of the company encoded the lesson of 1981.

Palm and the other path

The parallel American thread was Palm: the Pilot of 1996, with Graffiti handwriting, HotSync one-button synchronization, and a Zen-like discipline about what a pocket device should not attempt. Palm OS peaked near three-quarters of the US handheld market in 2000 [O27], on the strength of doing less, better. Its ceiling was architectural: built for organizers, it struggled into the telephony and multimedia era, and Palm's repeated reboots (PalmSource, Cobalt, Access) never landed. Between Psion's over-engineering and Palm's under-engineering sat the question the iPhone would answer in 2007: what happens when you build a pocket operating system with no compromises at all, because you own the silicon, the software and the sale?

The PDA's real lesson

The pre-iPhone decade's deeper lesson was that the pocket market punished every half-measure. The PDA era's devices - Psion's organizers, Palm's Pilots, HP's iPAQs, Apple's own Newton (1993, too early, too expensive, handwriting not ready) - each proved one element and missed the rest: Psion had the operating system without the phone, Palm had the simplicity without the architecture, Windows Mobile had the desktop's weight without its reason, and the Newton had the vision a decade before the components. Reading the period with hindsight, the checklist for the smartphone was already written: a real operating system, a real browser, capacitive touch, a developer story, and integration with a service ecosystem. What no one could schedule was the hardware - the ARM SoCs, flash densities and battery chemistries all crossed their thresholds in the mid-2000s at once. The iPhone looks, in retrospect, less like an invention than like the first complete assembly of parts the industry had spent fifteen years preparing. That is not a criticism; it is how platform shifts actually happen, and it is why so many companies were simultaneously close.

The carriers' world

The commercial structure the iPhone broke also needs recording. Before 2007, the customer relationship in mobile belonged to the carrier: the carrier chose which phones its network allowed, which applications were preinstalled, which features (Wi-Fi, GPS, Bluetooth file transfer) were disabled, and what the "deck" - the phone's portal - promoted. Device makers were suppliers; operating systems were firmware. Apple's 2007 deal with Cingular/AT&T was revolutionary precisely because it inverted this: Apple controlled the device, the

software and the updates; the carrier sold the pipe. Android's alliance structure [O32] offered carriers a way to answer without surrendering to Apple, which is why they adopted it so fast. The smartphone wars of 2008-2012 were fought on this commercial axis as much as on technology - and the Epic trial record [O37] shows how completely the victors then rebuilt a toll structure of their own on the ruins of the carriers'.

Chapter 24 - Symbian's peak, and its ceiling

Symbian's dominance deserves a chapter of its own, because its fall became the canonical lesson in how operating system platforms die. Through the mid-2000s it was the smartphone: Nokia's N-series flagships, Sony Ericsson's communicators, a full catalogue of applications, and the backing of the largest phone company on earth. In 2008 Nokia bought out its partners to take full control of Symbian Ltd [O28], and in a gesture that looked magnanimous, the code was opened to the world through a new Symbian Foundation.

But the system had been built for scarcity, and its age was showing. Symbian's programming model, honed for two AA batteries and four megabytes, was legendarily arduous for developers. Its interface layer had fragmented across vendors, so an application written for one Symbian phone often stumbled on another. The ecosystem - that self-reinforcing loop of users attracting developers attracting users - was spinning the wrong way: developers were tiring of Symbian's difficulty just as two newcomers were promising them modern tools and modern economics.

There is a deeper structural point that this book returns to. An operating system's competitive position is not its feature list; it is the cost, in developer effort, of making the next application appear. Symbian in 2008 had ten years of features and the industry's largest installed base, but writing for it was slow, testing across its variants was slower, and its store infrastructure was years behind. When the new platforms turned developer time into the scarcest resource in the industry, Symbian's incumbency converted from asset to drag with startling speed. Its peak share, about 73 percent in 2006, evaporated within five years [O27] - the fastest collapse of a dominant operating system the industry had ever recorded, and a preview of how quickly the same gravity could reverse against anyone.

The numbers at the summit

Symbian's peak was real and large. Around 2006 the platform held roughly two-thirds of the smartphone market - some 73 percent on some measures - with over 60 million Symbian phones shipped that year alone [O27]. Nokia's S60 interface ran across dozens of models from multiple manufacturers; NTT DoCoMo's MOAP carried it in Japan. The installed base of Symbian devices passed 100 million before the iPhone existed. This was the incumbent the iPhone faced: not a weak one but the most successful smartphone platform ever built, backed by the largest handset maker in the world, shipping in volumes the PC industry had taken twenty years to reach.

The ceiling, honestly described

Symbian's architecture had been designed for the 1997 constraints and carried them into a different era. Its C++ dialect with manual memory idioms made applications hard to write and easy to crash; its UIQ/S60/MOAP fragmentation meant "a Symbian app" barely existed as a category; and its development tools were miserable compared to what desktop programmers expected. The platform's certificate-based signing system, designed for carrier security, made casual development nearly impossible - the exact opposite

of the App Store's signed-but-frictionless model. When Nokia bought out its partners in 2008 for 264 million euros and then open-sourced the whole platform through the Symbian Foundation [O28], it was attempting to keep the scale while shedding the complexity. The move failed for a reason that recurs in this book: openness cannot retroactively fix ergonomics. Developers follow toolchains and money, and after 2007 both were elsewhere.

Nokia's operating system, Nokia's problem

The structural issue underneath Symbian's ergonomics was ownership. Nokia accounted for the overwhelming majority of Symbian shipments, which meant the platform's other partners - the original anti-Microsoft coalition - were effectively licensing their largest competitor's software. The 2008 buyout and open-sourcing [O28] was designed to fix exactly this: make Symbian neutral property, and the coalition could hold. Timing defeated the design. The iPhone was already a year old; Android shipped months after the buyout; and the Symbian Foundation's open release (2010) arrived into a market that had already chosen its two horses. The Foundation closed in 2011. The episode is the clearest test case in this book for a claim made in Chapter 22: openness is necessary but not sufficient. Android was open AND modern AND backed by a full-stack company with no handset business to protect; Symbian became open while remaining architecturally dated and politically suspect. The licence is the ticket, not the race.

The market that vanished under it

Symbian's decline speed still astonishes. From roughly two-thirds of a growing smartphone market in 2006-2007, the platform fell to irrelevance by 2012 - not because its phones stopped selling (Nokia shipped record volumes into 2010) but because the basis of competition shifted underneath it, from hardware features and carrier relationships to developer ecosystems and touch interfaces. Symbian had an app store, a touchscreen UI, and global distribution, and none of it mattered because the developers' energy had moved. Chapter 28's losers all share this signature: the market did not gradually leave; it tipped. Platform economics are unforgiving in both directions - the same network effects that made Symbian's rise look inevitable made its fall look sudden. Every incumbent since, including today's, has studied this chapter of the story with personal interest.

The squeeze

Symbian's fall is usually told as a single event - the iPhone - but the mechanics were a pincer. From the top, the iPhone's 2007 arrival redefined what a phone's software owed its user: a real browser, a real touch interface, a development kit that treated the phone as a computer [O29]. From the bottom, Android gave every second-tier manufacturer a modern system for nothing, starting with the HTC Dream in October 2008 [O33], which meant Symbian was undercut in the market segment that had always been its volume. Nokia's response in 2008 - buying out its Symbian co-owners for hundreds of millions of euros and putting the platform's code under an open-source foundation - was an attempt to out-open Android before Android shipped [O28]. It moved too slowly. The source release took years to complete, the foundation's membership never became a developer movement, and the gesture read as what it was: a consortium rearranging its paperwork while two companies with a single controlling mind each lapped it.

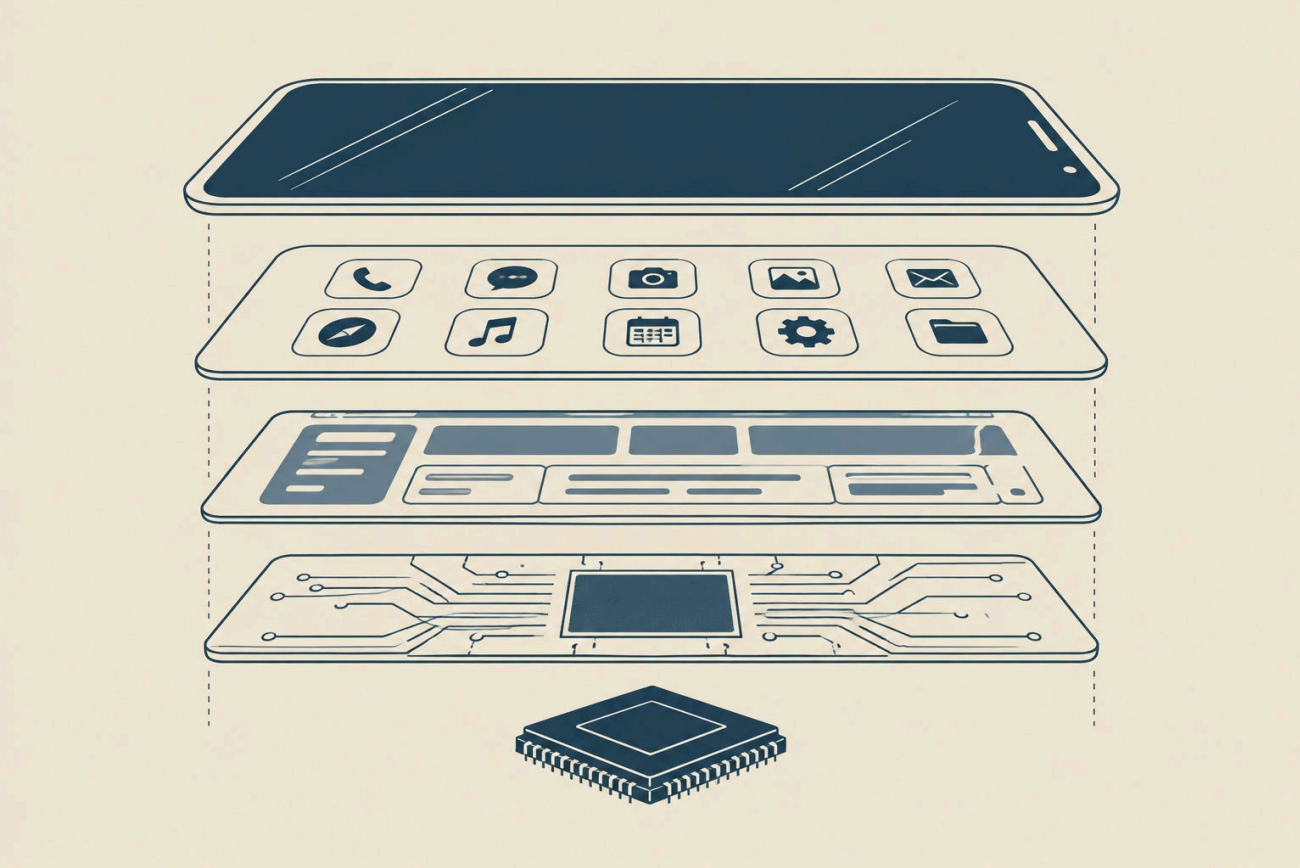
Inside Nokia the problem was never ignorance. Engineers demonstrated touch prototypes before the iPhone shipped; the memos said the right things. The constraint was structural: Symbian had to serve the whole Nokia range, from commodity handsets to flagships, on hardware generations apart, and every operator and licensee

had a vote. Each requirement was defensible; the sum was paralysis. Stephen Elop’s leaked 2011 memo named the result with unusual honesty for a chief executive: Nokia was standing on a burning platform, and the man who had made the fire said the company had to jump [O35]. The jump - to Windows Phone, an operating system with no ecosystem of its own - traded a fading asset for a partner’s unproven one, and the handset business that Microsoft bought in 2014 had to be almost entirely written off the following year [O36].

The lesson the industry took was not about code quality. Symbian’s kernel was respectable engineering. The lesson was that by 2008 an operating system was no longer judged as an operating system: it was judged as a market, with developers, applications and a retail shelf, and a technically fine system with a slow market lost to a good-enough system with a fast one.

Chapter 25 - iPhone: OS X in a pocket

On 9 January 2007, at Macworld in San Francisco, Steve Jobs introduced the iPhone by naming its software before its hardware: “iPhone runs OS X”, he told the audience - a claim, Apple noted in its press release, that the phone ran the same core operating system as the Mac [O29]. The claim was more true than marketing usually is. Beneath the touch interface, the iPhone ran a system derived from Mac OS X, which was itself the 2001 rebirth of the Macintosh on a Unix foundation: the XNU kernel, built from the Mach microkernel that Avie Tevanian’s team had developed at Carnegie Mellon from 1985 [O51] and the BSD layer NeXT had assembled at the company Apple bought in December 1996 for $429 million - the acquisition that returned Jobs to Apple [O34]. The chain of custody is remarkable: the phone in a billion pockets descends from a 1969 PDP-7 in New Jersey, by way of a Pittsburgh research kernel and a Redwood City workstation company.



The pocket mainframe in layers: glass, apps, operating system, kernel, silicon.

What the iPhone changed was not the kernel but the contract. The multi-touch screen made the interface direct - no stylus, no keys, the software object itself under the finger - and the operating system was designed around that directness from the first pixel. The mobile browser was the real web, not a stripped imitation. And the device was deliberately closed: at launch, third parties could not write native applications at all.

That closure lasted eighteen months, and its reversal created the modern app economy. On 10 July 2008, alongside the iPhone 3G, Apple opened the App Store with about five hundred applications [O30] - a single, curated, operating-system-integrated shop where any developer could sell to every iPhone user, with Apple reviewing each app and keeping 30 percent of the price. The commission rate would become one of the most contested numbers in business (Chapter 27); the model - the operating system as marketplace owner - became the industry's default within two years. The operating system had completed a journey: from scheduler of machine time, to platform for applications, to shopkeeper of a two-sided market, collecting rent on every sale above it.

NeXT's long way back

The iPhone's operating system was a homecoming twenty years in the making. When Apple bought NeXT in December 1996 for $429 million [O34], it acquired two things: Steve Jobs, and NeXTSTEP - a Unix operating system built on Carnegie Mellon's Mach microkernel (begun at CMU in 1985 [O51]) with a BSD userland and the object-oriented frameworks that became Cocoa. Mac OS X, which shipped to consumers in 2001, was NeXTSTEP in all but name. The crucial detail for 2007 is that Apple spent the early 2000s making that system small: OS X ran on a 400MHz PowerPC, then on Intel, and the engineering discipline of paring a workstation Unix to consumer hardware meant that when the iPhone project needed an operating system, the answer was

already in the building. iPhone OS - renamed iOS in 2010 - was OS X's kernel (XNU), its core frameworks, and a new touch interface layer. The pocket computer shipped with a thirty-year-old Unix inside, which is why it could browse the real web, run real email, and crash like a workstation rather than like a phone.

The App Store inversion

The iPhone launched, famously, without third-party native applications - Jobs's "sweet solution" was web apps. The reversal came in a year: the App Store opened 10 July 2008 with about five hundred applications [O30], and the design choices made that weekend defined mobile economics for two decades: Apple as sole retailer, a 70/30 revenue split, review as gate, distribution friction near zero. Developers who had suffered Symbian's signing bureaucracy and carrier decks could reach every iPhone user by uploading a binary. Ten million downloads came in the first weekend. The store was an operating-system decision - Apple controlled the platform APIs, the signing keys and the payment rail as one integrated layer - and it converted the operating system's gatekeeping power into retail margin. The 30 percent commission that Epic would later attack in court [O37] was born as the price of that integration.

The 2.35 billion

The scale Apple built on that 1996 rescue is now the industry's benchmark for reach: 2.35 billion active devices, the figure Apple disclosed in January 2025 [O2]. The number matters for what it includes - iPhones, iPads, Macs, Watches, Apple TVs, all running derivatives of the same XNU kernel and the same frameworks - and for what it enables: services revenue (the App Store, iCloud, Music, Pay) that has become Apple's second business, all taxed at the platform's boundary. The NeXT acquisition's return on investment is, by any honest accounting, the best $429 million ever spent in the industry [O34]: it purchased the operating system that became macOS, iOS, iPadOS, watchOS, tvOS and visionOS, plus the executive who made them into the world's most valuable product line. Chapter 4's observation completes: the platform is the product, and the company that owns its whole stack owns the terms of every market above it.

The sealed system

Apple's post-2007 design philosophy deserves a clear statement, because it is the industry's great counter-example to the commons. Where Google published Android's source, Apple sealed everything: the bootloader, the kernel extensions, the App Store's signing keys, the Secure Enclave's keys fused into silicon. The seal buys real security - iOS's exploit costs are the highest in the industry - and real control: no carrier shovelware, no sideloaded competition, updates delivered directly to a billion devices within days. The costs are Chapter 29's subject: the same seal that keeps malware out keeps rivals out, and the DMA's designation of iOS as a gatekeeper [O39] is the state's judgment that the distinction has stopped mattering. Apple's response - the minimum compliance the law allows - is the strategy Chapter 12 would predict from any rational gate owner: the gate is the business; everything else is negotiation.

Chapter 26 - Android: the best deal ever

Android began in 2003 not at Google, and not as a phone project with a finished plan: Andy Rubin and his co-founders started Android Inc. in Palo Alto to build smarter software for cameras, then pivoted to the bigger target of mobile phones. By 2005 the startup was running short of money and shopping itself around. Google

bought it in August of that year for a price never officially disclosed and estimated at about fifty million dollars [O31]. Years later, Google's vice president of corporate development, David Lawee, would call it the company's "best deal ever" [O31]. It is hard to argue: the purchase secured Google a position at the base of the device that was about to displace the personal computer as humanity's main way of reaching the internet.

Google's strategy inverted Microsoft's. Windows was sold per machine; Android would be given away. The logic was not charity but traffic: Google's business was advertising against attention, and attention was moving to phones. If any single company - Apple, Microsoft, Nokia - controlled the phone's operating system, it could tax or throttle Google's services. Android was insurance, priced at fifty million. On 5 November 2007, Google and thirty-three other companies - handset makers, chip firms, carriers, software houses - announced the Open Handset Alliance, committing to Android as an open platform [O32]. The first phone arrived on 23 September 2008: the T-Mobile G1, built by HTC [O33].

The engineering was a deliberate collage, and each choice mattered. The kernel was Linux, which gave Android a free, proven, hardware-hungry foundation - drivers, memory management, networking - though heavily patched. Above it, applications were written in Java, not by accident: Sun's language carried the world's largest population of mobile and enterprise developers. Because Oracle's predecessor Sun's Java runtime was encumbered, Google built its own virtual machine, Dalvik, and its own implementation of the Java libraries - a decision that would cost it eleven years in court (Chapter 27). Above that, the familiar grid of apps.

The business model completed the design. Phone makers received Android free, but the commercially vital pieces - the Play Store, Maps, Gmail, YouTube, Search - were licensed through the Mobile Application Distribution Agreement, which effectively required manufacturers wanting the store to preinstall Google's apps, with the Play Store prominent on the home screen [O37]. The operating system was open; the storefront was controlled. This double structure - open kernel, gated garden - defined Android's economics and would eventually define its legal troubles.

The numbers settled the argument. Android passed a billion users, then two; at Google I/O in May 2021 the company announced more than three billion active devices [O1] - the largest installed base of any operating system ever built, on phones and tablets across every price band on earth, from fifty-dollar handsets in Lagos and Jakarta to flagships in Seoul. In October 2026 it carried about 72 percent of global mobile web usage [O4]. The system that began as a hedge against exclusion had become the default computer of the species.

The $50 million insurance policy

Google bought Android Inc. in 2005 for what Reuters later reported as an estimated $50 million - a sum Google's own deal-maker, David Lawee, described at a 2010 conference as the company's "best deal ever" [O31]. Andy Rubin's startup had originally aimed at cameras, then pivoted to phone software with the thesis that the mobile industry's problem was a missing open platform. For Google, the strategic fear was specific: Microsoft's playbook - own the operating system, own the defaults, tax everything above - applied to the mobile internet, where Google's advertising business would live or die on default search placement. Android was bought to ensure that no one could put a toll booth between Google and the mobile user. It worked beyond all intention: by May 2021 Google was announcing three billion active Android devices at its I/O conference [O1].

The alliance as the moat

The launch mechanism was the Open Handset Alliance, announced 5 November 2007 [O32]: thirty-four companies - handset makers (HTC, LG, Motorola, Samsung), carriers (T-Mobile, Sprint), chip firms (Qualcomm, Intel) - committed to an open-source platform with no licensing fee. The first phone, the T-Mobile G1 built by HTC, shipped in September 2008 with the Android Market along for the ride [O33]. The structure was Symbian's insight executed correctly: share the base layer so no single owner taxes the industry - but this time with a single engineering centre of gravity (Google), modern tools (Java syntax, Eclipse), and a store that welcomed developers instead of certifying them. The cost, which surfaced a decade later in Epic v. Google, was that "open" described the code, not the economics: the Play Store, the Google apps bundle, and the default-placement deals were the proprietary toll layer sitting atop the open road [O37].

The fragmentation tax

Android's openness had a price that became visible by 2011: fragmentation. Every manufacturer shipped its own version, on its own schedule, with its own modifications; carriers delayed updates for certification; and millions of devices ran years-old releases with unpatched vulnerabilities. Google's countermeasures were architectural, and they redrew the line between the commons and the product: the Android Compatibility Programme and CTS test suite defined what "Android" legally meant for device makers who wanted the Play Store; the platform's moving parts migrated into Google Play Services, updated over the air, bypassing manufacturers; Project Treble (2017) separated the vendor's silicon layer from the system image so updates could flow; Project Mainline (2019) made core OS modules updatable through the store itself. Each step made Android more updatable and, not coincidentally, moved more of it into Google's controlled, proprietary layer. The open base served its strategic purpose - ubiquity - and the control migrated upward, to the layer Chapter 27 and Chapter 29 describe. The pattern is now a recognized playbook: open the substrate, close the services, collect at the boundary.

The numbers at the summit

The summit's dimensions: three billion active devices by May 2021 [O1], roughly 72 percent of the world's mobile operating-system market in October 2026 [O4], and across all device types the single most-used operating system on Earth [O72]. The ecosystem the G1 seeded [O33] now spans phones, tablets, watches, televisions, cars and laptops in all but name. And the platform's economics matured exactly where Chapter 13 predicted: the Play Store's margin trajectory - 26 percent in 2014 to 71 percent in 2021 [O38] - is the DOS toll booth with an additional twenty years of refinement, and Epic v. Google [O37] is the antitrust arc arriving on schedule. The best deal ever [O31] bought not a product but a position, and positions of this kind, this book has argued, are the most durable assets the technology industry knows how to build.

Chapter 27 - The Java wars

Android's Java decision produced the longest, most consequential software-copyright fight of the century so far, and it is worth following closely, because at stake was a question every operating system faces: who owns the interface?

When Google built Android, it needed millions of Java programmers to feel at home. Java's attraction as a platform was its library of prewritten tasks, summoned by exact command lines - the application programming

interface, or API. Google wrote its own implementations of the libraries but copied the API's declaring code: about 11,500 lines, the commands' names and organisation, so that a programmer's existing knowledge would work unchanged [O73]. Oracle, which acquired Sun - Java's owner - in January 2010, sued Google that August, claiming copyright in the API itself.

The case ran for eleven years, through two jury trials, two reversals by the Federal Circuit, and a final appeal. The legal question sounds dry and was in fact foundational: an API is a system of operation - the words through which programmers command a platform. If one company could copyright the system of commands, every compatible reimplementation, every emulator, every clean-room clone in computing history - from Compaq's IBM-compatible BIOS to Linux's re-creation of Unix's calls - stood on shakier ground. On 5 April 2021 the Supreme Court decided for Google: assuming the lines could be copyrighted, copying them was fair use, because what Google took was a user interface - "a way for programmers to access prewritten computer code" - and programmers had invested their own learning in it, which the law weighs differently from ordinary creative work [O73]. Justice Breyer's opinion walked through the four fair-use factors; the deciding logic was that locking up the API would grant copyright holders monopoly power not over expression but over other people's acquired skill.

The industry exhaled, but the deeper lesson is older than the case. Control of an operating system's interface is control of its ecosystem. DEC learned it when Compaq cloned the PC's BIOS legally; Microsoft profited from it when it licensed QDOS's CP/M-compatible calls; Apple spent the 1990s suing over the "look and feel" of windows and menus and largely lost. Google's win kept the door open for compatible systems - and, not coincidentally, left Android free to grow into the world's largest operating system. The interface, the courts finally said, belongs partly to the people who learned it.

Oracle v. Google: the API question

The first mobile-platform war was fought over copyright, not market share. Oracle, having acquired Sun in 2010, sued Google the same year, claiming Android's re-implementation of Java's APIs infringed both patents and copyright. After a decade of litigation and two trips to the Supreme Court, the Court held 6-2 in April 2021 that Google's copying of roughly 11,500 lines of declaring code from the Java SE API was fair use as a matter of law [O73]. The reasoning mattered more than the verdict: Justice Breyer's opinion treated APIs as an interface whose reimplementation serves programmers' accumulated investment in learning them - a legal recognition that the value of a platform lives partly in its developers' heads. The decision removed a legal cloud over every clean-room reimplementation in the industry, from Wine to AWS-compatible APIs, and it retroactively legitimized the way Android was built.

The Java that Android actually used

Technically, Android's Java story was always a fork of the community's design: applications written in Java syntax, compiled not to Java virtual-machine bytecode but to Dalvik (later ART), a register-based runtime designed for low-memory phones, built on Apache Harmony's class libraries. Sun praised Android publicly in 2007 while privately smarting at the fork; Oracle chose litigation. The episode taught the industry a lesson in platform politics that HarmonyOS NEXT (Chapter 43) is now applying in reverse: if your stack can be cut off by a rival's lawyers or a government's order, you do not own your platform. Huawei's Ark compiler and own-kernel bet is Oracle v. Google, plus the Entity List, converted into engineering policy.

The other fronts

Oracle v. Google was the loudest front of the Java wars, not the only one. Sun's original mobile Java - J2ME - had fragmented across hundreds of carrier-mandated variants by the mid-2000s, a cautionary tale in platform governance that Android's compatibility programme was explicitly designed to avoid. Apple's prohibition of Java and Flash on the iPhone, announced in Jobs's 2010 "Thoughts on Flash", killed the mobile plugin era and asserted that the platform owner, not the developer-tool vendor, would set the runtime rules. Microsoft's .NET and C# - Java's direct answer, born of Sun's lawsuit over Microsoft's own Java extensions - became the enterprise stack Java might have been. Each front turned on the same question the Supreme Court finally answered [O73]: who owns an interface? The law's answer - interfaces can be reused, fairly, for compatibility - arrived twenty years into the industry's practice, and mostly ratified what engineers had assumed since Compaq's BIOS: the vocabulary of a platform belongs to everyone who has invested in learning it.

What the verdict changed

The practical consequences sorted quickly. Clean-room reimplementation - the technique behind Compaq's BIOS, Linux's Unix compatibility, Wine's Windows compatibility, and every "API-compatible" cloud service - stood on firmer ground. Platform owners adjusted by moving value out of the API: into services, app stores, signing keys and cloud backends, layers copyright cannot easily reach. Google's own post-verdict strategy reflects this: the Play services layer, not the Android API surface, is where the platform's control now lives [O37]. The law, arriving last as usual, settled the question of 2010; the market had already moved the fight to 2020's ground. Chapter 38's observation completes itself: each legal round governs the previous technical round, and the gate migrates upward faster than the law can follow.

Chapter 28 - The losers

Between 2007 and 2015, the mobile industry's old guard was liquidated - not gradually, but with a speed that reset everyone's model of how long platform dominance lasts. The casualties repay study, because each died of a different failure of the same organ: the operating system ecosystem.

Nokia was the giant. In February 2011 its new CEO, Stephen Elop, sent the company a memo that leaked immediately and became famous. He told the story of a man on a burning oil platform in the North Sea, who had to jump into freezing water to survive: "We too, are standing on a burning platform," Elop wrote, arguing that Nokia's problem was no longer devices but ecosystems - "the battle of devices has now become a war of ecosystems" [O35]. Days later Nokia bet on Microsoft's Windows Phone. It did not work. Microsoft, which had been building Windows Phone 7 since 2010 as a genuinely fresh, well-reviewed system, could not close the app gap: developers wrote for the two platforms with users, users bought the phones with apps, and no amount of engineering quality broke the circle. In 2013 Microsoft bought Nokia's phone business outright; in July 2015 it wrote the acquisition down by approximately $7.6 billion and cut up to 7,800 positions, mostly in phones [O36]. Windows Phone, technically admired to the end, was dead of ecosystem starvation - the clearest modern proof that in this industry, the layer above the operating system decides the operating system's fate.

BlackBerry fell differently: it mistook its feature for its franchise. Push email, the thumb keyboard, and corporate security had made it the executive's phone, but its operating system aged, its browser lagged, and its belated clean-slate replacement, BlackBerry 10, arrived in 2013 to find the app shelves bare. The company

eventually abandoned its own OS, shipped Android phones, and retreated to software - where, in a quiet irony, its QNX division ended up running the computers inside hundreds of millions of cars (Chapter 29).

Palm's webOS, introduced in 2009, was the tragedy's purest case: a beautiful, web-native, card-based system whose ideas - swipe gestures, cards for multitasking, wireless charging - were years ahead, owned by a company too small to survive one weak launch. HP bought Palm in 2010 and killed the hardware line in 2011 after forty-nine days on the tablet market. webOS survives in an afterlife almost no one notices: it is the operating system inside LG's smart televisions.

The lesson the decade burned into the industry: an operating system is not defeated by a better operating system. It is defeated by the migration of the people who build on it. Features can be copied in a year; an ecosystem is a standing army that takes a decade to raise and cannot be purchased at any price - as Microsoft proved by spending billions to learn it.

The burning platform, verbatim

Nokia's end deserves its own paragraph because its CEO wrote it down. In Stephen Elop's internal memo of early 2011, preserved in the contemporary record, the outgoing market leader compared his company to a man on a burning oil platform: Apple had "redefined the smartphone", Android had come from nowhere to take the mid-range in two years, and Nokia had "fallen behind, missed big trends, and lost time" [O35]. The memo's candour was unmatched and its remedy - abandoning Symbian and the MeeGo Linux project for Microsoft's Windows Phone, announced days later - was a bet that Microsoft's chequebook and Nokia's hardware could buy a third ecosystem. Elop, a former Microsoft executive, returned to Microsoft when it bought Nokia's phone business in 2014. The suspicion of designed failure that trailed the deal was never proven; the failure itself was total and public.

The $7.6 billion write-off

Microsoft's own accounting closes the story. In July 2015, fourteen months after completing the Nokia devices acquisition, Microsoft announced an impairment charge of approximately $7.6 billion - essentially the entire purchase - and 7,800 job cuts concentrated in the phone hardware business [O36]. Windows Phone's final share was a rounding error. The losses of the mobile wars were not the also-rans of a growing market; they were, in Nokia's case, the largest company Finland had ever produced, and in BlackBerry's, the firm that had invented the category's business end. The lesson the survivors drew was about switching costs: once a platform's apps, habits and developer base accumulate, even tens of billions of dollars and a famous brand cannot buy back the entry ticket. Platform markets tip, and after they tip they do not untip.

webOS: the best-designed corpse

Palm's last throw deserves its own eulogy because it was, by wide agreement, the most elegant loser. webOS (2009) was built on a Linux kernel with an application layer written in the web's own languages - HTML, CSS, JavaScript - a decade before that became orthodoxy; its card-based multitasking and gesture interface were so good that both Apple and Google later adopted their essential gestures. It failed on the other axes: weak hardware, a confused owner sequence (Palm, then HP, which bought it for $1.2 billion in 2010 and open-sourced it eighteen months later), and the chicken-and-egg of a third app ecosystem arriving last. webOS survives, improbably, as the interface of LG's televisions - tens of millions of sets running the phone system the market rejected. The base layer's afterlives (Chapter 13's FreeDOS, Chapter 31's QNX) are always in devices nobody designed it for.

BlackBerry and Windows Phone: the capital test

The two richest losers test the claim that money can buy a platform late. BlackBerry poured its franchise's profits into BlackBerry 10 (2013) - a QNX-based [O45] system with genuinely modern multitasking - and into a secure-services pivot, and the market did not turn; the secure niches (government, finance) were too small to carry an ecosystem. Microsoft's case was starker: Windows Phone had excellent reviews, Nokia's hardware, billions in marketing and carrier subsidies, and Nokia's entire smartphone volume behind it, and it peaked under 4 percent before the $7.6 billion write-off [O36]. Post-mortems converge on the same variable: applications. The two surviving ecosystems had crossed the threshold where users' apps, data and habits made switching irrational, and no amount of capital substitutes for the network effect once tipped. The mobile wars' peace treaty was not signed anywhere; it simply became true - a duopoly, with the world's third-largest company and its most valuable brand both outside it. Chapter 43 asks who might still force a new war: the candidates are a state-backed stack and an AI interface, and both know exactly what happened to the last challengers.

Chapter 29 - The tollbooth's reckoning

The mobile duopoly settled into place by 2013: Apple owned the profitable top of the market, Android owned the rest of the planet, and both taxed their ecosystems through their stores. The App Store's 30 percent commission, introduced in 2008 as table stakes, became over the following decade one of the highest-margin tolls in business history. At the Epic v. Google trial, an expert for Epic calculated that the Google Play Store's operating margin climbed from 26 percent in 2014 to 71 percent in 2021 - generating more than $12 billion in operating profit in that year alone [O38]. The trial record showed Google had originally set its own 30 percent commission simply to match Apple's [O37].

How the toll was defended turned out to be as important as its size. Epic v. Google - decided by a jury on 11 December 2023, and affirmed by the Ninth Circuit on 31 July 2025 - put on the record the machinery of platform protection [O37]. Android technically permitted installing apps from outside the Play Store, but the path was fenced: default settings disabled direct downloads, and changing them triggered a gauntlet of warning screens, sometimes as many as fourteen or fifteen, none of which reflected any actual security assessment of the source. When Epic tried to distribute its game Fortnite directly, 35 percent of users who began the process abandoned it after hitting the warnings; finishing required more than fifteen steps [O37]. Meanwhile the MADA contracts kept the Play Store on the home screen, "Project Hug" paid top game developers not to leave it, and a proposal to Samsung called "Project Banyan" aimed to neutralise Samsung's own store [O37]. By 2020, the Play Store handled 95 percent of Android app downloads in the United States and more than 80 percent worldwide outside China [O37]. The jury found Google had illegally maintained monopoly power; the court's October 2024 injunction ordered catalog sharing and the distribution of rival stores through Play; the appeals court affirmed both verdict and injunction [O37].

Apple's parallel case, Epic v. Apple, ended mostly in Apple's favour on antitrust but found its anti-steering rules wanting under California law. Europe went further by statute: the Digital Markets Act requires designated gatekeepers to allow app distribution through third-party stores and the web (Article 6(4)) and to let developers steer users to outside purchase channels (Article 5(4)) [O39]. In 2025 the European Commission adopted a formal non-compliance decision against Apple over exactly those steering restrictions [O40].

The arc across two decades is the operating system's economic story in miniature. The layer that began as a scheduler became a platform, then a marketplace, then a regulated utility in everything but name. The regulators' question is no longer whether the toll booth may exist, but how high the toll may run, and what the owner may do to keep travellers from taking the other road.

The Play Store's margin, on the record

The Epic v. Google trial did something rare: it put the mobile toll's actual economics into the public record. The Verge's trial coverage reported figures Google had never disclosed: the Play Store's operating profit margin climbed from 26 percent in 2014 to 71 percent in 2021, on revenue exceeding $12 billion a year [O38]. A 71 percent margin is not a marketplace competing for sellers; it is a gate collecting rent. The Ninth Circuit's July 2025 opinion [O37] records the mechanics that produced it: the MADA agreements that required manufacturers to preload Google's apps as a bundle; the "Project Hug" and "Project Banyan" programmes paying leading developers and device makers not to build competing stores; and the scare screens - the court counted fourteen to fifteen warning steps - that confronted any user who tried to install an app from outside the store, a gauntlet that roughly 35 percent of users abandoned. Google collected about 95 percent of US Android app-distribution revenue through Play.

The DMA writes it into law

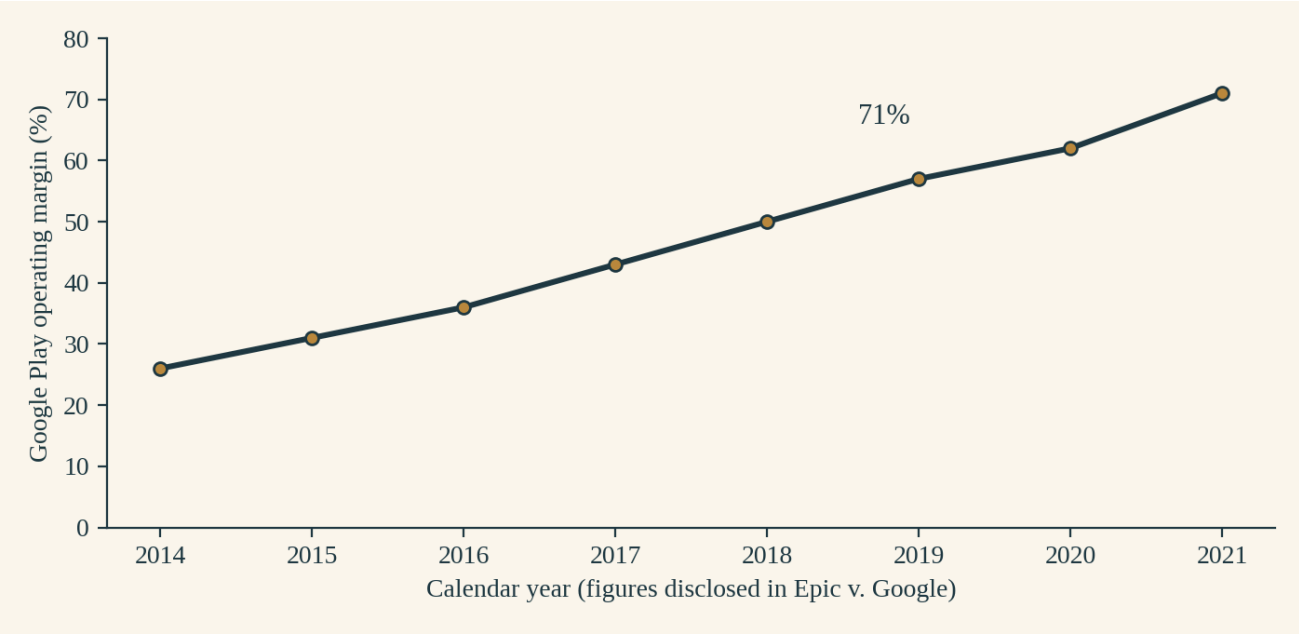
Europe legislated what the American cases litigated. The Digital Markets Act, in force for designated gatekeepers since 2023-2024, names the operating system itself a "core platform service" and instructs its owner in plain terms: Article 6(4) requires gatekeepers to allow third-party app stores and sideloading; Article 5(4) prohibits anti-steering rules that stop developers telling users about cheaper options [O39]. The Commission's subsequent decisions have begun enforcing the list [O40]. The DMA is the antitrust arc's latest form - the observation of Chapter 38 made statute: when a private gate becomes infrastructure, the state eventually publishes the gate's operating rules. Apple's response - grudging European sideloading, new fees with new names - and Google's injunction appeal [O37] show the industry's counter-move: comply with the letter at maximum friction. The contest between the toll booth and the regulator is now permanent, which is its own kind of verdict on how durable the toll's economics are.

The verdict and the injunction

The trial's outcome was the first judicial finding that a mobile gate's operation was illegal monopoly maintenance. On 11 December 2023 a San Francisco jury found for Epic on every count [O37]: Google had monopoly power in Android app distribution and in-app billing, had maintained it anticompetitively, and had unlawfully tied the store to its billing system. The remedy phase produced, in October 2024, an injunction ordering Google to open Play to rival stores and to stop requiring Google Play Billing - the order whose implementation the Ninth Circuit's 2025 opinion reviewed while largely leaving its core intact [O37]. The injunction's practical content is remarkable: for the first time, a dominant gate was ordered not merely to stop specific abuses but to host its competitors - rival app stores distributed through the store itself. Implementation has been predictably contested, and the DMA's parallel requirements in Europe [O39] mean the world's two largest regulatory zones now both require the mobile gates to open. The toll booth still stands; it is required, for the first time since 2008, to let other collectors through.

What the 30 percent bought

The defence’s best argument deserves honest statement, because the Ninth Circuit took it seriously even while affirming: the commission funded security review, distribution infrastructure, developer tools, and the Android platform’s continued development - and Apple’s matching 30 percent [O37] showed the rate was an industry standard, not a Google artifact. The court’s answer was not that 30 percent is inherently illegal but that the market was never allowed to test it: the tying, the exclusivity payments and the scare screens prevented the competition that would have discovered the real price. That is the antitrust arc’s deepest principle for the base layer: the problem is never the gate’s existence - gates are how platforms are governed - but the gatekeeper’s suppression of the market in gates. The industry built two of the most profitable businesses in history behind walls that competition law was, for fifteen years, too slow to examine. The examination is now permanent, and it is part of the operating system’s cost of doing business.



Google Play operating margin, 2014-2021, as disclosed in Epic v. Google [O38]. Chart drawn from the cited figures.

PART VII - THE INVISIBLE MAJORITY

Measured by units, the famous operating systems are a minority of computing. The majority works invisibly: in cars, spacecraft, cash machines, routers, thermostats and infusion pumps, under deadlines measured in milliseconds and certification regimes measured in years. Down there the virtues of Part VIII reverse. The general-purpose system optimizes throughput and fairness; the real-time system optimizes the worst case, because a brake controller that is usually on time is a brake controller that is sometimes late.

This part visits the invisible majority. VxWorks, born in a Berkeley garage in 1981, flew to Mars on Pathfinder - where the famous priority-inversion bug of 1997 taught the whole industry why the scheduler’s edge cases are not edge cases when the machine is on another planet. QNX’s microkernel became the default instrument cluster of the car industry, hundreds of millions of vehicles deep. The long tail runs further: FreeRTOS in tens of billions of microcontrollers, seL4 with a mathematical proof instead of a test suite, the space station’s laptops quietly migrated to Debian. And the automotive industry’s hundred computers per car are forcing the deepest rethink of what a vehicle operating system is since the engine control unit was invented. The ghost is already everywhere. This part is where it works without ever being seen.

Chapter 30 - Real time: the operating systems that cannot be late

All the systems discussed so far share an assumption so deep it goes unsaid: they are allowed to be a little late. A phone that redraws its screen a hundred milliseconds behind schedule annoys nobody in particular. But a vast category of computing lives under the opposite rule. An anti-lock brake controller that is late is wrong. A flight-control computer that is late is wrong. For these machines, the operating system’s first duty is not fairness or features but determinism: the guarantee, provable in advance, that the highest-priority task will run within a known number of microseconds, every time.

The systems built to that guarantee are called real-time operating systems, and the archetype is VxWorks, created by Wind River - a company founded in a Berkeley garage in 1981 by Jerry Fiddler, a former Lawrence Berkeley National Laboratory computer scientist, and named for the Wyoming mountains where he vacationed [O43]. VxWorks became the industry’s leading RTOS, ranked first by VDC Research, certified across more than 600 safety projects to standards like DO-178C for aviation and ISO 26262 for cars, and trusted, in Wind River’s own summary, for systems “where it simply cannot fail” [O43].

Its most famous workplace is not on Earth. NASA’s Curiosity and Perseverance rovers both drive across Mars on VxWorks [O43]. And the lineage’s most instructive hour came on an earlier rover, and deserves the telling it gets in Chapter 40: in July 1997, days after Pathfinder landed, the spacecraft began resetting itself unpredictably. The fault, diagnosed from Earth by reading the machine’s own error logs, was priority inversion - a subtle scheduling pathology in which a low-priority weather task, holding a resource, was blocking a high-priority bus task while medium-priority tasks ran instead. The resets were the system’s watchdog firing on the missed deadline. The fix - enabling priority inheritance, a mechanism already present in VxWorks but switched off in that build - was uploaded to a spacecraft 190 million kilometres away, and the mission was

saved by what amounted to a one-line change in scheduling policy [O44]. Few episodes show so clearly what an operating system is: the pathology was not in any instrument or program, but in the invisible rules about whose turn it was.

The real-time world has its own economy. Its customers do not buy downloads; they buy certification evidence, long-term support, and the certainty that the vendor will still answer the phone in fifteen years. Prices are small per unit and enormous in aggregate, because the units are planes, trains, infusion pumps and robots - the machines whose failure modes are measured in bodies. It is the least glamorous and most demanding corner of the operating system industry, and almost nobody outside engineering knows it exists.

What “real time” actually means

The phrase is abused enough to deserve a definition. A real-time operating system is not a fast one; it is a predictable one. The guarantee on offer is not “quick” but “by this deadline, always” - a scheduler whose worst-case latency can be bounded and proven. The distinction sorts the industry: Linux and Windows optimize throughput and fairness on average, and their worst case can be arbitrarily bad (a disk error storm, a memory compaction); VxWorks, QNX, INTEGRITY and their peers optimize the worst case itself, and sell the bound. An anti-lock brake controller, a flight computer or an infusion pump does not care about average response; it cares that the response never, ever misses. The engineering consequence is a different culture: smaller kernels, static analysis, certification evidence, and release cycles measured in years, because in these markets the operating system is a safety component, and safety components are proven, not patched.

Wind River and the garage

VxWorks, the market’s flagship, began in a Berkeley garage in 1981, where Jerry Fiddler and David Wilner founded Wind River - named, the company likes to record, after the Wyoming range where Fiddler had worked [O43]. The product’s lineage runs through NASA’s 1980s embedded work, and its customer list became a roll-call of high-stakes engineering: aerospace, defence, industrial control, automotive, networking. The company’s own positioning has not changed in four decades: “the intelligent edge” is the modern phrase for the same market - machines that must compute correctly where they are, without a datacenter’s forgiveness [O43]. Wind River’s sale history (to Intel in 2009, then TPG, then Aptiv in 2022) tracks the market’s steady revaluation: the invisible majority of operating systems kept getting more valuable as computing spread into the physical world.

Certification as a product feature

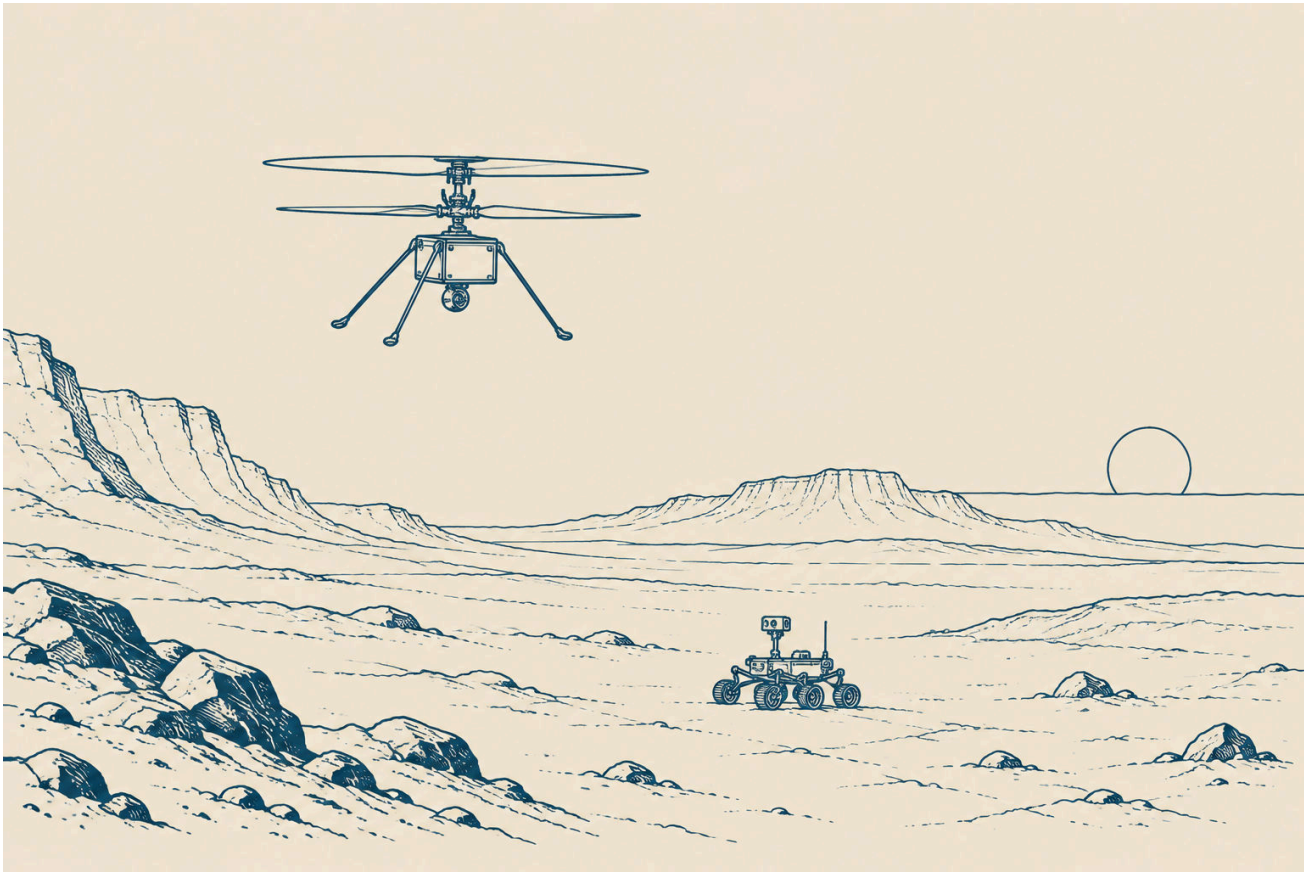
The real-time vendors’ brochures read differently from consumer software’s, and the difference is the product. Where consumer platforms sell features, RTOS vendors sell evidence: the DO-178C certification artifacts for avionics, the ISO 26262 safety case for automotive, the IEC 62304 documentation for medical. Wind River’s product pages [O43] and QNX’s [O45] foreground exactly this - safety certifications, long-term support, security evaluation - because their customers are not buying software but the right to put software in a certified machine. The business consequence is the longest customer relationships in the industry: an avionics program chooses an RTOS at design start and lives with it for forty years, through the airframe’s entire production and service life. The operating system that cannot be late also cannot be replaced on a whim. In no other software market does the 1969 lesson - the base layer is a commitment, not a product - bind so completely.

Fuchsia and the new entrants

The real-time world is not immune to the industry's larger currents. Google's Fuchsia - a from-scratch, non-Linux, capability-based microkernel system (Zircon) - shipped quietly in the Nest Hub from 2021, the first new production operating system from a major platform company in decades; its ambitions remain deliberately unstated, but its design (capability security, driver isolation, updatability) reads as a study in what Android's architecture would be if designed after the lessons of Chapter 40. Amazon's stewardship of FreeRTOS [O50] and Azure RTOS (ThreadX) gave the clouds a path to the edge's smallest devices. And seL4's lineage [O52] keeps appearing wherever assurance is the requirement. The invisible majority, in other words, is where the industry's next architecture is being prototyped - as it was in 1969, when the future of interactive computing was being sketched on a spare PDP-7 while the mainframe world looked the other way.

Chapter 31 - The car's hundred computers

A modern car is not a machine with a computer; it is dozens of computers with wheels, and they run on operating systems almost no driver has heard of. The dominant one is QNX, and its story is a small perfect arc of the industry's history.



Linux on another world: the Ingenuity helicopter and the Perseverance rover on Mars.

QNX came out of Kanata, Ontario, where Dan Dodge and Gordon Bell - graduates of the University of Waterloo - founded Quantum Software Systems in 1980. Their product was first called QUNIX, "Quantum UNIX", until a polite letter from AT&T's trademark lawyers persuaded them to shorten it [O45]. From the

start it was a microkernel: the kernel did almost nothing but pass messages and schedule, while drivers, file systems and networking ran as ordinary processes outside it. Crashes stayed local. The design won a devoted niche in factories, medical devices and nuclear instrumentation - places where a reboot is a production incident.

Its route to ubiquity ran through BlackBerry, which bought QNX in 2010 as its own phone business crested. The phone failed; the kernel thrived. Cars were absorbing computers by the dozen - dashboard infotainment, digital instrument clusters, driver-assistance controllers, telematics - and automakers wanted a system with decades of safety certification behind it. By BlackBerry's fiscal 2026 results, QNX software was embedded in more than 275 million vehicles worldwide; the division posted a record $78.7 million quarter, up 20 percent year on year, with a royalty backlog of $950 million [O46] - a number that means hundreds of millions of already-signed future cars.

Around the specialist kernels, the car industry built a standard of its own: AUTOSAR, the automotive open system architecture, whose Classic Platform serves the predictable, microcontroller-based functions - windows, brakes, engine - and whose Adaptive Platform specifies a runtime environment for the new, powerful domain computers that handle automated driving, with standardized interfaces letting applications be developed independently of the hardware beneath them [O47]. The industry's stated destination is the "software-defined vehicle": fewer, larger computers, updated over the air, running mixes of QNX, AUTOSAR, automotive Linux, and Android Automotive for the screens. The operating system question Microsoft and Nokia fought over in phones is being refought, quietly, in every car company on earth - with the difference that here a scheduling mistake can kill.

Priority inversion on Mars

The most instructive bug in real-time history happened 200 million kilometres from the debugger. On 4 July 1997, Mars Pathfinder landed running VxWorks, and within days its computer began resetting itself, losing data each time [O44]. JPL's engineers reproduced the fault on their Earth-bound replica and traced it to priority inversion: a low-priority task held a resource (the information bus's mutual-exclusion lock) that a high-priority task needed, while medium-priority tasks kept preempting the low one - so the highest-priority work waited indefinitely, the watchdog fired, and the system reset. The fix was elegant: enable priority inheritance (a VxWorks switch that temporarily lends the blocking task the blocked task's priority) by uploading new parameters. The account survives in Glenn Reeves's widely cited write-up and the ACM version of the post-mortem [O44]. Two lessons earned their keep. First, the rocket-science failure mode was a textbook concurrency hazard that the operating system's designers already knew. Second - and this is why every spacecraft since has carried the capability - the system was fixable because it was designed to be debugged and patched from Earth. Real-time does not mean never wrong; it means engineered all the way down.

Ingenuity: Linux at 1.8 percent atmosphere

Two dozen years later, Mars flew again with a different stack. The Ingenuity helicopter, which made the first powered flight on another world on 19 April 2021, ran Linux on a Qualcomm Snapdragon 801 - a phone-class processor - with NASA/JPL's open-source F Prime flight-software framework, which JPL had used since Canham's team began it in 2013 [O48]. The choice was deliberate: Ingenuity was a technology demonstration, and flying commercial parts and open software was part of what it demonstrated. The significance for this book is the inversion of assumptions: in 1997, spaceflight meant VxWorks and radiation-hardened bespoke

hardware; in 2021, the Linux kernel - the hobby of Chapter 19 - was judged robust enough for the first aircraft on another planet, because the ecosystem's accumulated testing had become its own qualification. F Prime is now a Linux Foundation project, used beyond JPL. The commons reached Mars.

VxWorks and QNX: the duopoly's terms

The commercial real-time market settled into a structure the desktop would recognize: a proprietary duopoly plus a rising commons. VxWorks [O43] anchors aerospace, defence and industrial - markets where certification evidence and thirty-year support commitments are the product. QNX [O45] took automotive and medical on the strength of its microkernel's fault isolation: a crashed driver in QNX is a process to restart, not a kernel to reboot, and a dashboard that reboots its navigation app while the engine controller never notices is the architecture's sales pitch. The terms of competition are certifications (ISO 26262 ASIL D, IEC 61508 SIL 3, Common Criteria EAL levels) that cost millions to obtain and therefore defend the incumbents - Chapter 40's moat in its purest form. Against them, embedded Linux and the RTOS commons (FreeRTOS, Zephyr, and automotive's own AGL and SOAFEE efforts) advance from below, winning design starts where certification is lighter and connectivity matters more. The car, consolidating a hundred ECUs into a handful of domain computers running hypervisors, is where the two worlds now meet: one silicon, several operating systems, an RTOS for the brakes and Linux for the screen, separated by the same virtual-machine machinery IBM built in 1966.

BlackBerry's second life

QNX's corporate story contains the industry's strangest second act. BlackBerry - the fallen smartphone king of Chapter 28 - acquired QNX in 2010 for its own devices, lost the phone market anyway, and found that it owned the automotive industry's operating system of record. The company's filings put numbers on the resurrection: QNX revenue of $78.7 million in one recent quarter, up 20 percent year on year; fiscal-year growth of 14 percent; a design-win backlog around $950 million; software in over 275 million vehicles [O46]. A company whose consumer brand died with the keyboard phone now supplies the safety-certified plumbing of the software-defined vehicle. The lesson is one this book keeps finding: in the base layer, the survivors are the ones whose technology was built for constraints the market eventually arrived at - QNX's 1982 microkernel was waiting, forty years early, for the car to become a computer.

Chapter 32 - Everywhere else: the operating system's long tail

Beyond cars and spacecraft stretches the true mass of the industry: the long tail of devices that no one thinks of as computers at all.

The automatic teller machine is a personal computer in an armoured cabinet, and for two decades it ran overwhelmingly on Windows XP: when Microsoft ended XP support in April 2014, 95 percent of the world's 2.2 million ATMs still ran it, and major banks negotiated and paid for extended support while they struggled to migrate fleets whose hardware, drivers and certifications were all pinned to the old system [O53]. Point-of-sale terminals, ticket kiosks, and industrial controllers repeat the pattern: operating systems chosen for a decade's deployment and then locked in place by the cost of revalidation.

The television became a computer too: Samsung's sets run Tizen, LG's run the exiled Palm webOS. Routers and access points run Linux, often the community-built OpenWrt; e-readers and smart speakers run stripped

Linux. In the pocket-sized cheap end, KaiOS - a descendant of Mozilla's abandoned Firefox OS - put apps and 4G onto keypad feature phones, including India's JioPhone, serving users for whom a smartphone was out of reach.

At the very bottom of the scale, beneath even these, sit the bare-metal and tiny-kernel systems: the microcontroller in a washing machine, a hearing aid, a mouse. Here lives FreeRTOS, created by Richard Barry in 2003 as a free, minimal real-time kernel, and adopted in November 2017 by Amazon, which relicensed it under the permissive MIT licence and built its device-cloud strategy around it [O50]; here too the Linux Foundation's Zephyr project. These kernels are a few thousand lines long. They will never be reviewed, photographed or loved. And they are, by unit count, the majority of all operating systems ever shipped - billions of quiet instances, each one still doing the job the GM-NAA monitor was invented for: keeping a machine busy, in order, without a human standing over it.

The ATM and the kiosk

The most numerous computers most people touch are the ones they do not think of as computers, and their operating systems are usually the industry's afterthoughts. The automatic teller machine is the canonical case: for years after Microsoft ended Windows XP's support in April 2014, a large share of the world's ATMs still ran it, and banks paid Microsoft for custom extended patches because certifying a fleet of cash machines on a new platform cost more than the protection money [O53]. The pattern generalizes: point-of-sale terminals, airport kiosks, ticket machines and hospital devices run whichever Windows or Linux was certified when the device was designed, and they run it for a decade past its obituary. The long tail is not neglect; it is the economics of certification, the same force that keeps mainframe COBOL alive. Chapter 40's end-of-life cliffs - Windows 10's October 2025 retirement and the ESU bridge to October 2027 [O54] - are, for this market, a recurring procurement event with a known price.

The Space Station's laptops

Even low Earth orbit runs the desktop's systems. In 2013 the United States operational segment of the International Space Station migrated its fleet of laptops from Windows XP to Debian 6, with the Linux Foundation providing training; the foundation's account quotes the station's computing manager, Keith Chuvala: "We needed an operating system that was stable and reliable - one that would give us in-house control" [O49]. The reasons are the whole argument of Part V in one sentence: in orbit there is no field service, no vendor escalation path, and no tolerance for a blue screen, so the ability to inspect and patch the source is a mission requirement. The station's laptops run the astronauts' procedures, inventory and email - ordinary work, extraordinary environment, same conclusion as the supercomputers (Chapter 22): when the requirement is control, the answer is the commons.

Routers, radios and the network's own systems

The Internet's plumbing runs its own operating systems, historically distinct from both the desktop and the datacenter. Cisco's IOS - the Internetwork Operating System, not Apple's - has configured the world's routers since the 1980s; Juniper's Junos is FreeBSD underneath, a fact its documentation has always stated plainly; and the modern shift to "white-box" networking runs Linux-based NOS stacks (SONiC, originally Microsoft's, now a Linux Foundation project) through the hyperscalers' fabrics. The pattern rhymes with supercomputing: a specialized, reliability-obsessed market that standardized on Unix-descended systems because nothing else was both dependable and adaptable. Your home router almost certainly runs Linux; the

broadband remote-access servers that terminate a city’s connections run Linux or FreeBSD; the mobile core is migrating to cloud-native network functions on Linux. The network is made of operating systems talking to operating systems - the ghost, multiplied and talking to itself.

The device census

An honest census of the long tail defies precision, but the magnitudes are known. Tens of billions of microcontrollers ship each year, and the share running an RTOS or embedded Linux rather than bare-metal firmware grows steadily as chips get cheaper and radios get included: FreeRTOS (whose kernel stewardship passed to AWS in 2017 [O50]), Zephyr, ThreadX, VxWorks, QNX and their peers divide the market roughly by criticality - toys and thermostats at one end, avionics and medical at the other. The long tail’s aggregate weight is the argument of this part: measured by unit count, the famous operating systems - Windows, macOS, even Android - are a minority of the world’s computing. The majority is invisible, unbranded, unpatched and working. The security implications of that last clause are Chapter 40’s subject; the engineering achievement of the first three is the subject here.

PART VIII - HOW THE BOSS WORKS

How does the boss actually work? The first eight parts followed the industry; this one opens the machine. The ideas here are fifty years old and completely current, because the operating system's basic settlement - rings of privilege, virtual memory, the scheduler, everything-is-a-file - was reached early and has never been replaced, only refined. The phone in a pocket and the supercomputer in Part I run on the same four ideas.

The kernel and the rings: why some instructions are forbidden to ordinary programs, and how the system call became the narrow gate between them. The scheduler: whose turn it is, decided thousands of times a second, from round-robin courtesy to the datacenter's market in millicores. Virtual memory: the best con in computing, giving every process a private machine that does not exist, and paying for the illusion with page faults. Everything is a file: the Unix design decision that made devices, pipes and networks one interface, and its limits. And the architecture argument settled by practice - monolith against microkernel, the 1992 debate revisited thirty years on, when the winners turned out to borrow from both sides and the proof-carrying microkernel found its niche where failure is not an option. Read this part and the rest of the book becomes visible as what it is: the business and history of these four ideas.

Chapter 33 - The kernel and the rings

Strip an operating system to its irreducible core and one arrangement appears, unchanged since the Atlas Supervisor of 1962: the machine is divided into two worlds. In one world, code may do anything - touch any memory, command any device, change the rules. In the other, code may act only by asking. The first world is the kernel; the second is everything else; and the border between them is enforced not by software but by the processor itself, which carries a privilege bit - several privilege levels, in modern designs descended from Multics' rings - that flips only at carefully guarded crossing points.

Those crossing points are the system calls. A system call is a controlled trap: the program executes a special instruction, the processor flips to privileged mode and jumps to a fixed, kernel-owned address, the kernel validates the request, performs it, and returns. Open this file. Allocate memory. Send this packet. Every interaction between the billion applications and the trillion devices in the world squeezes through these narrow gates. The complete set of calls an operating system offers is its true constitution - more fundamental than any logo or marketing name - and the reason Linux can run decades-old Unix software is that it faithfully reimplements a constitution first drafted in 1973.

Around the constitutional core, an actual kernel carries an enormous staff: the scheduler, which chooses who runs next; the memory manager, which hands out and reclaims RAM; the file systems; the network stack; and - the bulk of the code by far - the device drivers, the per-hardware translators that let one kernel speak to a million kinds of silicon. Of the Linux kernel's roughly forty million lines, the great majority is drivers [O26]. The pattern holds everywhere: Windows, macOS, Android. The glamorous centre of the system is tiny; the unglamorous periphery, endlessly grown to meet new hardware, is vast. This is why "writing an operating

system” is easy enough for a student to do in a semester and why making one usable is beyond the reach of almost any company: the kernel is a weekend project, the drivers are an industry.

Rings and gates

The hardware contract underneath every modern operating system is a bargain the processor makes: it will run fast, provided the software agrees to be two kinds of program. Unprivileged code - applications - executes in the processor’s restricted mode (ring 3 on x86), where it cannot touch hardware, other programs’ memory, or the kernel’s tables; privileged code - the kernel - runs in ring 0 with the machine’s full instruction set, including the instructions that change which mode the machine is in. Crossing the boundary is deliberately expensive: a system call is a controlled gate, a special instruction that traps into the kernel at a fixed entry point where the kernel decides whether the request is legal. The entire security model of the world’s computing reduces to this: billions of lines of application code, all untrusted, asking for services through a doorway a few hundred instructions wide. Intel and AMD actually provide four rings; Multics designed for them; almost everyone uses two. The unused rings are a monument to the 1960s’ more ambitious security architecture (Chapter 8), kept in silicon out of compatibility.

The kernel’s working day

What lives in ring 0 is surprisingly small and surprisingly busy. The scheduler (Chapter 34) decides the next program to run, typically every few milliseconds, whenever a timer interrupt, a blocked process or a newly ready one forces the question. The memory manager (Chapter 35) maintains the page tables that make every process’s private address space real. The interrupt handlers catch the hardware’s demands for attention - disk finished, packet arrived, key pressed - and route them. The system-call dispatcher fields requests: open, read, write, fork, map. And the drivers, historically the largest and buggiest part, translate between the kernel’s abstractions and ten thousand kinds of device. Everything else the user sees - the desktop, the browser, the apps - is a guest of this few-hundred-thousand-line core, living or dying by its judgment. The kernel is the only software on the machine that is always awake; that is the definition of the boss.

The syscall, examined once

It is worth slowing down on one system call, because everything else is repetition. When a program calls open(“notes.txt”), the library packages the request, executes the trap instruction, and the processor switches to ring 0 and jumps to the kernel’s single entry point. The dispatcher validates the request: is the pointer legal, is the string readable, does the process hold rights to this directory? The file-system layer resolves the path through its caches to an inode, checks permissions against the process’s credentials, locates the blocks on disk, and returns a small integer - the file descriptor, a capability in number form. Control returns to ring 3; the whole crossing takes microseconds and happens millions of times per second per machine. Every property the user depends on - privacy between users, integrity of files, mediation of devices - is enforced at this doorway and nowhere else. The kernel’s entire security claim is that there is no other door. Attackers’ entire art is finding one: a driver bug, a speculative-execution side channel, a misconfigured permission. The siege has run continuously for sixty years; the door still holds, on average, which is the most that can honestly be said.

Drivers: the soft underbelly

The kernel’s own code is reviewed, fuzzed and argued over by the best systems programmers alive. Its drivers are not. Device drivers - historically the majority of the kernel’s lines, written by vendors to ship hardware on

schedule - account for the large majority of kernel crashes and a disproportionate share of vulnerabilities; Windows's blue screens and Linux's oopses are mostly driver events. The industry's responses have been architectural: user-space driver frameworks (Linux's UIO, Apple's DriverKit) that let a crashed driver die without taking the kernel; signed-driver requirements; and the microkernel and Rust programmes of Chapters 37 and 41, both aimed largely at the driver layer. The boss's weakest employees are the ones it hires from outside. This was true of the 360's device handlers in 1966 and is true of the GPU driver in your phone today.

Chapter 34 - Whose turn: the scheduler

The scheduler is the operating system's referee, called upon thousands of times a second to answer one question: who runs next? On a machine with eight processor cores and eight hundred runnable programs, the answer is the product of policy dressed as mathematics, and the policy encodes what the system's designers believed computing was for.

The batch monitors answered it crudely: whoever is next in the queue. Time-sharing demanded more, and produced the round-robin and the time slice - each user in turn, for a few dozen milliseconds, the illusion of simultaneity purchased by switching so fast that no human notices. Modern interactive systems refined the goal from fairness to responsiveness: the program you are looking at must feel instant, the backup job in the background can wait. Linux's current scheduler distributes processor time so that every task advances at a weighted share of the machine, tracking each task's received service down to nanoseconds; Windows and the phone systems play further tricks with priorities that boost the thread under your fingertip.

The pathologies are where the theory earns its keep, and none is more instructive than priority inversion. Give tasks strict priorities and a shared resource - say, a data structure only one task may touch at a time - and a nightmare ordering becomes possible: a low-priority task holds the resource; a high-priority task wakes and must wait for it; meanwhile medium-priority tasks, which care about neither, run freely and starve the low-priority holder. The high-priority task ends up waiting not on the task it contends with but on unrelated traffic - priority, inverted. This is not a classroom curiosity. It is exactly what froze the Mars Pathfinder's bus task in July 1997 while its low-priority weather task held the bus and medium-priority communications work ran on; the spacecraft's watchdog saw the deadline pass and reset the computer, again and again, until engineers on Earth read the logged failure, understood it, and switched on the ready-made cure, priority inheritance - the low task temporarily borrows the blocked high task's priority while it holds the resource [O44]. The whole pathology, its cure, and a mission saved by remote diagnosis: the scheduler's entire moral universe in one episode. The lesson generalises far beyond rockets. Every system that queues human work - hospitals, ports, call centres - has a scheduler, explicit or not, and every one of them can invert.

From round-robin to completely fair

Scheduling is where the operating system's oldest function meets its most modern economics. CTSS used multilevel feedback queues in 1961; the problem was already fully formed: interactive users need instant response, batch jobs need throughput, and the scheduler cannot know in advance which is which. Unix's classic scheduler sorted processes by "niceness" and aged them. Linux's Completely Fair Scheduler, introduced in 2007, replaced queues with a red-black tree ordered by "virtual runtime" - each task runs in proportion to its share, and the least-run task goes next - an accountant's solution to a resource-allocation problem. In 2023-2024 the kernel moved again, to EEVDF (Earliest Eligible Virtual Deadline First), bringing

latency-sensitive fairness into the same framework. The constant across sixty years is the workload's invisibility: the scheduler must infer a program's character - interactive, batch, bursty - from its behaviour alone, in microseconds, millions of times a day.

The phone changes the question

Mobile added a new axis: energy. A scheduler that keeps the processor's cores awake to be responsive drains the battery; one that batches work to let cores sleep risks a stuttering interface. Android's answer involves the scheduler cooperating with the power manager - "energy-aware scheduling" - and with the display's deadline: a frame every 16.7 milliseconds or the user sees a jank. The datacenter's version of the same pressure is utilization: a hyperscaler wants every expensive core busy, which is why Google built Borg and then Kubernetes [O71][O62] - a scheduler above the scheduler, allocating whole machines to workloads the way the kernel allocates milliseconds to threads. Chapter 42 follows that stack upward. The pattern to notice is fractal: at every scale, from nanoseconds to datacenters, someone is running Corbato's 1961 loop - who is waiting, who is urgent, whose turn is next.

Priority, and how it is earned

Every scheduler answers two questions - who is next, and for how long - and the design space between answers is where six decades of practice live. Fixed priorities (the real-time world's choice, Chapter 30) make the answer provable and risk the starvation of low-priority work; dynamic priorities (the desktop and server world's choice) adapt to behaviour and make proofs harder. Timeslice length trades responsiveness against overhead: CTSS's 200-millisecond quantum, modern Linux's millisecond-scale preemption, and the phone's frame-deadline scheduling are the same dial set for different latencies of impatience. Multiprocessor scheduling added the hardest wrinkle: moving a task between cores forfeits its warm cache, so the scheduler must balance fairness against affinity - a memory-management problem wearing a scheduler's clothes. The general lesson is that there is no free lunch, only accounting: every scheduling policy is a choice of which workloads to favour, and the choice is therefore always partly political.

When the scheduler is the product

The datacenter turned the scheduler from a component into a market. Borg's internal scheduling decisions were, for a decade, one of Google's core competitive secrets - how to pack thousands of machines to 60-plus percent utilization while keeping latency-sensitive services responsive [O71]. Kubernetes productized the insight [O62]: declare the workload's needs, and the scheduler finds the placement. Above that, the autoscaler adjusts capacity to demand; above that, the spot market prices it. A 1961 construct - who runs next - now has a price, an SLA and a derivatives market of reserved instances and savings plans. Corbato's loop closes: time-sharing was invented to sell one expensive machine to many users, and the cloud is time-sharing with a billing API. The ghost was always an accountant; now it invoices.

Chapter 35 - Remembering everything: virtual memory

Of all the illusions the operating system maintains, virtual memory is the most audacious: it promises every program a private, vast, contiguous expanse of memory that does not exist.

The promise is kept with pages. Memory is cut into fixed-size pages, typically four kilobytes; each program's address space is a table mapping its private page numbers to real locations, which may be anywhere in physical RAM - or nowhere in RAM at all, parked on disk. When a program touches a page that is not resident, the hardware raises a page fault, and the kernel quietly fetches the page from disk, perhaps evicting another to make room, and restarts the instruction as if nothing happened. The program never knows. Every computer you have ever used is running this con, billions of times a day.

The invention belongs to the Atlas team in Manchester, who built the first paged, one-level store in 1962 because their fast memory was small and their drum was slow; the theory of how to keep the con efficient belongs to Peter Denning, whose 1968 working-set model explained why paging works at all: programs, at any moment, touch only a small, slowly shifting set of pages, so the hot set fits in RAM and the cold rest can live on disk. When the working set outgrows the machine, the system thrashes - spending its life swapping instead of computing - a failure mode every programmer has met as the beach ball or the frozen hourglass.

Virtual memory's side effects reshaped computing. Because each program has a private address space, no program can scribble on another's - memory protection, the foundation of all security, arrives as a by-product. Because addresses are fictions, the kernel can overcommit: promise more memory than exists, on the actuarial observation that not every program draws on its promise at once, a piece of bookkeeping that behaves exactly like fractional-reserve banking and fails the same way under a run. And because mapping is flexible, tricks like shared libraries, memory-mapped files and copy-on-write fall out for free. The modern phone with "8 gigabytes" is, in truth, running twenty gigabytes of promises on eight gigabytes of substance, with the kernel as the clearing house - and it works for the same reason banking works: almost everyone almost always leaves their deposits alone.

The page-fault economy

Virtual memory is the operating system's best con: every process is given a private, contiguous, enormous address space that does not exist. The hardware's memory-management unit translates each address through page tables; pages not in physical memory trigger a page fault, and the kernel quietly fetches them from disk or creates them from zero. The con pays because programs exhibit locality: at any moment they touch only a small working set of their pages, so the machine can run programs far larger than its memory with only a modest tax - until the working set exceeds the memory, and the tax becomes the famous thrash, the disk-light-on, everything-frozen state every long-serving user knows. Swapping was invented at Manchester (the ATLAS one-level store, Chapter 3) and perfected on the VAX; the paging algorithms - least-recently-used and its approximations, clock hands, working-set estimation - are sixty years of refinements on one insight: the future resembles the recent past.

Overcommit and the OOM killer

Linux's most notorious memory policy follows from a pragmatic observation: programs routinely allocate memory they never touch (fork before exec doubles a process's apparent footprint instantly), so the kernel overcommits, handing out promises of memory it may not have. Usually the promises are never called. When they are - when physical memory and swap genuinely run out - the kernel's OOM killer selects a process and shoots it, choosing by heuristics that try to kill the greedy rather than the vital. Server administrators treat OOM events as failures of capacity planning; embedded engineers treat overcommit itself as the failure. Both are right: it is a policy tuned for the datacenter's economics, where slightly overselling memory across thousands of machines is worth more than the occasional casualty. The episode shows something general

about the base layer: even its technical mechanisms encode business models. The same kernel that powers a pacemaker-adjacent medical device and a hyperscaler’s fleet must be configured by people who know which economy they are in.

The fork trick

Unix’s process-creation design is worth a section because it is simultaneously beautiful and absurd, and both properties shaped history. fork() clones the calling process - address space, open files, everything - by marking the memory copy-on-write: parent and child share physical pages until either writes, so the clone costs almost nothing until it diverges. The child then usually exec()s a new program, discarding the cloned world it just made. Ritchie and Thompson’s retrospective [O18] records the design’s virtue: no new concepts - the clone is an ordinary process, with the same pipes, files and privileges, which is why shell redirection and pipelines fell out naturally. The absurdity is scale: a 40-gigabyte database process that wants to run /bin/echo must conceptually clone 40 gigabytes (copy-on-write makes it cheap, but the accounting - Chapter 35’s overcommit - must assume the debt might be called). Modern systems offer posix_spawn and vfork as escapes. The design’s longevity is the point: an elegant hack from a small machine became the process model of the world, and the world has been engineering around its corner cases, fondly, ever since.

The phone’s answer to memory pressure

The pocket mainframe inherited virtual memory’s machinery but not its safety valve. Swapping pages to disk, the sixty-year-old release valve, sits badly on a phone: flash storage wears out under constant write traffic, a page fetch from flash costs power the battery cannot spare, and a user holding the device in hand will not forgive the half-second stall a desktop user never noticed. So the mobile systems rebuilt the policy. Android compresses instead of swapping - a region of RAM (zRAM) holds squeezed pages, trading processor cycles for capacity, on the observation that a phone has compute to spare and flash endurance to protect. When compression runs out, both platforms kill: iOS’s jetsam and Android’s low-memory killer rank background applications and shoot the most expendable, having first asked each to serialize its state to storage so it can be resurrected later, looking, with luck, as if it had been running all along.

That is why the app you left an hour ago sometimes returns to exactly your place and sometimes starts over: underneath, a page-level accounting decided the app’s fate without the user or the developer having any say. The desktop’s thrash - the frozen hourglass of Chapter 35’s working-set failure - has no mobile equivalent, because the phone refuses to pay it; it spends applications instead. The policy inverts the mainframe’s economics. The original supervisors sacrificed jobs to keep the machine busy; the phone sacrifices programs to keep the human unbothered. Virtual memory’s con survives intact - every process still believes in its private, vast, contiguous expanse - but on the pocket machine the clearing house has a kill list, and being backgrounded is a state of probation, not rest.

Chapter 36 - Everything is a file

Unix’s most quoted design decision sounds like a slogan and is actually an engineering economy: everything is a file. Documents are files. Disks are files. Terminals are files. Network connections, process information, hardware devices - all presented through the same small set of operations: open, read, write, close. One interface, learned once, applies to the whole machine; programs and people stop caring what is underneath.

The organising metaphor above the interface is the tree: directories inside directories, one root, a path from root to leaf that names every object on the system. Multics pioneered the hierarchical file system; Unix made it canonical; every successor kept it. The tree is one of those ideas that looks obvious only after it has won - earlier systems had flat namespaces, per-device islands of files, or both.

Underneath the metaphor, file systems are among the hardest software humans build, because they must keep promises across catastrophe. Pull the power cord mid-write and a naive file system is left with a half-updated structure - the file gone, or worse, silently wrong. The industry's answers accumulated over decades: careful write ordering, then journaling (write the intended change to a log first, then do it; after a crash, replay the log), then copy-on-write designs that never modify data in place at all, so that the on-disk state is always a valid snapshot. The modern champions - ZFS from Sun, Btrfs, Apple's APFS, Microsoft's NTFS and ReFS - add checksums over everything, because disks and flash do not merely fail, they occasionally lie, returning bits other than the ones they were given. A modern file system is less a filing cabinet than an auditor that expects its own filing cabinets to commit fraud.

The file system's sibling idea, the database, grew up to handle the cases where files are too blunt; and the cloud era added object stores, where the tree dissolves into a flat sea of named blobs. But walk into any machine room or open any phone, and the tree is still there, forty years on, still the way civilisation keeps its working papers in order.

Everything is a file - and its limits

Unix's most quoted design slogan was a deliberate provocation with a practical core: present every resource - disks, terminals, printers, processes, network connections - through the same four verbs: open, read, write, close. The payoff was composability: programs that knew nothing of each other could be chained with pipes, each reading the last one's output as a file. The 1978 CACM retrospective by Ritchie and Thompson states the philosophy plainly [O18]: a small set of interfaces, uniform across devices, written in a portable language. The limits arrived with the interactive and networked era: graphics hardware, IOCTL-heavy devices and asynchronous network I/O never fit the read/write grammar gracefully, and Plan 9, the Bell Labs successor to Unix, spent the 1990s pursuing the slogan to its logical end (everything really is a file, including the network and the window system) and was admired more than adopted. Linux and macOS today are pragmatists: the file metaphor where it fits, specialized APIs where it does not.

Journaling and the power-cut contract

The file system's hardest promise is crash consistency: at any instant - including the instant the power fails - the disk must hold either the old state or the new one, never a scrambled half-write. The industry's answer, from IBM's JFS through ext3, NTFS, XFS and btrfs, is the journal: write the intention first, then the change, then replay the journal after a crash. Copy-on-write systems (ZFS, btrfs, APFS) took the further step of never overwriting in place at all, making snapshots nearly free - which is why a modern Mac can roll its whole system back across a failed update. The engineering is invisible until it fails, which is the point: the file system's success is measured in the data losses that do not happen, across billions of devices, every day. It is the purest example of the operating system's working arrangement with its users: they will never know what it does, and they must never have to.

ext4, XFS, btrfs and the Linux way

Linux’s file-system history is a lesson in the commons’ style of engineering: not one answer but several, adopted at different speeds for different risks. ext2 (1993) was the simple, fast, unjournaled start; ext3 (2001) added the journal conservatively; ext4 (2008) remains the default workhorse - boring, proven, everywhere. XFS, SGI’s contribution, serves the large-file workloads; btrfs and ZFS (via OpenZFS) brought copy-on-write, checksums and snapshots to the mainstream; F2FS serves flash’s peculiar physics. The pattern contrasts sharply with Windows’ single NTFS line or Apple’s HFS+-to-APFS succession: where the proprietary vendors manage one file system as a product, the commons maintains a portfolio, and users - usually without knowing - pick by distribution default. The portfolio’s cost is fragmentation of effort; its benefit is that no vendor’s roadmap can strand your data. Android, the largest single deployment, runs ext4 and F2FS under the glass - the pocket mainframe’s ledgers are kept by the commons.

Permissions and the social contract of files

The file system is also where the operating system’s trust model touches the user most directly. Unix’s nine permission bits - read, write, execute for owner, group, other - were designed for 1970s timesharing among colleagues and have carried the Internet’s security ever since, extended by access-control lists, SELinux’s mandatory policies, and the phone’s per-app sandboxes where every application is its own “user”. The progression tells the trust story of the whole book in miniature: from colleagues sharing a machine, to strangers sharing a network, to adversaries sharing a pocket. Each era kept the file metaphor and hardened the boundary around it. When Chapter 29’s courts ordered Google to let rival stores onto its devices [O37], the implementation question underneath was, literally, which file-system permissions and package-signing rules would change. Law at the top, inode semantics at the bottom, one system.

Chapter 37 - The architecture argument, settled by practice

The monolithic-versus-microkernel debate of 1992 never got a referee’s decision, so history issued one instead, and it is double-edged. Where raw capability and hardware breadth mattered, the big kernel won utterly: Linux and NT are monolithic or hybrid, and they run the datacenter, the phone and the desktop. Where provable reliability matters more than features, the small kernel won completely: QNX’s message-passing microkernel runs the car fleet, and one microkernel has gone further than any system in history toward mathematical certainty.

That system is seL4, developed by Gernot Heiser’s group at NICTA in Australia, which in 2009 became the first general-purpose operating-system kernel with a machine-checked, functional correctness proof: a formal demonstration, carried out by theorem-proving software, that the kernel’s code implements its specification - behaviour pinned down with the rigour of a mathematical theorem rather than the hope of testing. Its most public examination came from DARPA’s HACMS program, which built seL4-based flight software and then invited professional red teams to attack it; the program’s own case study records that the seL4-protected systems resisted, and HACMS software now protects the Army’s MH-6 Little Bird helicopter from malware-based attacks [O52].

Between the two poles lies the hybrid mainstream, and its most instructive lineage is Mach. Developed at Carnegie Mellon from 1985 with goals including kernel-level interprocess communication and user-level

virtual-memory servers [O51], Mach was meant to replace the monolithic Unix kernel with something cleaner. Pure microkernel performance disappointed, so the winning systems compromised: Apple's XNU grafts BSD services onto a Mach core for speed; Windows NT keeps its services in a hybrid executive. The purest idea of 1985 survives everywhere, but always alloyed. The debate's real resolution is that "kernel architecture" turned out to be not a religion but a dial - and different industries set it at different points for measurable reasons: throughput on one end, proof on the other.

POSIX: the treaty

The practical settlement that let the Unix family survive its own success was standardization. By the mid-1980s the commercial Unices had diverged enough that a program written for one needed surgery for another, and the fragmentation was Microsoft's opportunity: one stable target (DOS, then Windows) beat twenty squabbling better ones. POSIX - the Portable Operating System Interface, IEEE's family of standards from 1988 - fixed a common floor: system calls, shell, utilities, that every Unix-like system would provide. Its achievement was negative but decisive: it stopped the divergence. Linux implemented POSIX from the start, which is why the GNU toolchain ran on it in 1992 and why the world's server software moved to it with so little friction. Standards are the unglamorous machinery of the commons: no company owns POSIX, every company can meet it, and the user of the standard is free to leave. Chapter 39's contrast - Broadcom's VMware rent, extracted precisely because customers could not leave - is the same observation from the other side.

The hybrid everywhere

Apple's XNU is the hybrid design with the largest installed base: Mach handles the lowest-level tasks and threads, a BSD layer provides POSIX, networking and the file system, and the two share one address space for speed - the 1990s performance lesson absorbed into the 1985 microkernel [O51]. Windows NT made a parallel set of compromises for parallel reasons: its executive is structured like a microkernel's client-server design but runs in kernel mode, again for speed. The two systems that dominate desks and pockets are thus both descendants of the 1992 debate's losing side, wearing the winning side's clothes. The honest summary is that architecture debates are settled by the market's constraints - performance in the 1990s, certification in the 2000s, formal proof in the 2010s - and each constraint crowned a different design. seL4's 8,700 lines can be proved correct; Linux's tens of millions of lines can run the world. The industry, being an industry, buys both.

The performance question, honestly

Why did the monolith win the 1990s? The honest answer is measurement, not ideology. Every boundary in a microkernel - between kernel and file server, between driver and application - costs context switches and message copies, and on 1990s hardware those costs were visible in benchmarks that customers ran. Microsoft's own Research group spent years on exactly this problem, and its eventual answer was not a microkernel at all but Singularity and Midori: research systems that bought isolation with language safety instead of address spaces - software isolated processes, verified by the compiler rather than the MMU. The lineage from that research to Chapter 41's Rust wager is direct: if the language guarantees safety, you can co-locate components without trusting them, and the microkernel's cost largely vanishes. The 1992 argument, fully resolved, says: the monolith won on the hardware of its day; the microkernel won where proofs matter; and the future probably belongs to systems that make the distinction irrelevant.

POSIX, once more, as the quiet winner

The settlement’s least dramatic participant did the most work. Because POSIX standardized the interface rather than the implementation, every combatant - monolith, microkernel, hybrid - could present the same face to applications, which is why a 1995 Unix program compiles today on Linux, macOS, QNX and seL4-based systems with equal ease. The architecture wars were fought below the interface; the interface, like System/360’s architecture thirty years earlier (Chapter 4), was the promise that held. It is the final rhyme of this part: the industry argues about implementations and standardizes promises. The promises - POSIX, the C ABI, the socket API - are the true platform, and they have outlived every kernel ever written.

PART IX - THE BUSINESS OF BEING THE BOSS

Whoever owns the boss owns the building. The operating system's position between hardware and every application is not just an engineering fact; it is a commercial chokepoint, and the law has spent forty years learning to see it. This part follows the money and the courts: the antitrust arc from IBM's 1969 unbundling under government pressure, through the Microsoft consent decree of 2001, to the mobile cases - Epic v. Google, the Digital Markets Act - that are dismantling the app-store tollbooth in the 2020s. Each era's case looks different and turns out to be the same case: the platform's owner may not use the gate to decide which other businesses may exist.

Then consolidation: what the buyers actually bought. IBM did not buy Red Hat's code - no one can buy what is already free; it bought the maintenance relationship, the trust of the Fortune 500's infrastructure teams. Broadcom did not buy VMware's hypervisor; it bought the switching cost of every enterprise datacenter that runs on it. The pattern is the operating system's deepest economics: the product is position, and position is rented, never sold. And finally the cracks in the foundation - the xz backdoor, where one exhausted maintainer nearly handed the world's servers to an unknown patient adversary, and the end-of-support cliff that left cash machines running Windows XP years after its funeral. Being the boss is a business. This part prices it, and prices what happens when the business fails.

Chapter 38 - The antitrust arc

Because the operating system is a gate, every owner of a dominant one has eventually met the state. The encounters form a forty-year curriculum.

The first case ran against IBM. The government sued in January 1969 - six months before unbundling - alleging monopoly in general-purpose computers; the case dragged on for thirteen years and was dismissed in 1982 as without merit, but the long shadow of it shaped IBM's behaviour for a generation, including, on many accounts, its accommodating 1981 deal that left Microsoft owning the PC's operating system. The second case made law. The United States sued Microsoft in 1998 over the bundling of Internet Explorer with Windows; the trial court found monopoly maintenance, ordered a breakup, and saw the remedy overturned; the case settled on 2 November 2001 with conduct restrictions [O67]. Europe then did what America had not: in 2004 the Commission found against Microsoft and imposed a fine of 497 million euros, and followed with further penalties over the browser ballot and protocol licensing. The third round arrived with the phone: the Epic cases against Apple and Google (Chapters 25 and 29), and then legislation - the EU's Digital Markets Act, which simply lists what a gatekeeper operating system may no longer do [O39].

Read together, the cases trace the industry's own migration. The IBM case was about hardware. The Microsoft case was about the operating system itself. The mobile cases are about the layer above it: the store, the default, the payment rail. Each round, the contested gate moved one level up the stack - and each round confirmed the original observation from 1981: the money and the power live at the layer everyone must pass through.

The browser ballot and the fine that followed

Europe's Microsoft case deserves its own detail because it established the template the DMA later codified. The Commission's 2004 decision found two abuses: tying Windows Media Player to Windows, and refusing interoperability information to workgroup-server rivals; the remedy included a 497-million-euro fine and a version of Windows without the media player, which the market ignored - a lesson in the limits of unbundling remedies. The follow-on was more consequential: after Opera complained in 2007, the 2009 settlement forced Microsoft to show European users a browser-choice ballot, and Chrome's rise on the desktop dates visibly from the years that followed. When Microsoft "forgot" the ballot in a 2011 service pack, the Commission fined it again - 561 million euros in 2013 - for the omission. The episode taught regulators that conduct remedies need policing, which is why the DMA [O39] is written as an ex-ante rulebook: no case-by-case proof, just obligations that apply from designation day.

What the cases could not do

The sobering record is how little the judgments changed market structure. Microsoft kept Windows; IBM kept the mainframe; the remedies constrained behaviour at the margin while the network effects did the work. What actually dislodged each gate was a platform shift the incumbent missed: the PC dislodged the mainframe's gate, the browser dislodged the desktop's, the phone dislodged the PC's - and in every case the state arrived years after the market had moved. The mobile cases [O37] are the first attempt to act while the gate is still standing, and the DMA's designation of the operating system itself as regulated infrastructure is the first statute to say the quiet part: the base layer is not a product anymore; it is a utility, and utilities get rulebooks.

The IBM case's thirteen years

The first great case deserves its chronology, because its duration was itself the lesson. Filed in January 1969 at the close of the Johnson administration, United States v. IBM consumed thirteen years, hundreds of witnesses and millions of pages before the Reagan Justice Department withdrew it in 1982 as "without merit". During those thirteen years the market it was meant to govern transformed completely: the minicomputer rose, the PC was invented, and the mainframe monopoly the suit described stopped being the industry's centre. The case's real effects were internal: IBM's pricing and product decisions were made with counsel in the room, and many historians read the 1981 PC's outsourced, open architecture (Chapter 12) as partly an antitrust-era reflex - the safest way to ship fast was to own as little of the stack as the contract allowed. If so, the state's longest case against a computing monopoly had the paradoxical effect of creating the next one, in softer hands. Antitrust in this industry does not move slower than technology because lawyers are slow; it moves slower because the question it is answering has usually changed by the time the answer arrives.

Consent decrees and their shadow

The older instrument was the 1956 consent decree that confined AT&T to regulated communications - the reason Unix could be licensed to universities for tape costs but never sold as a product, and therefore the reason it spread as source code through a generation of graduate schools (Chapter 9). The decree's 1982 dissolution, breaking up the Bell System, liberalized Unix commercially just as the workstation market opened; Sun, SGI and the Unix wars were its children. The through-line is that operating-system history's biggest commercial events were often legal events wearing technical clothes: the 1956 decree, the 1969 unbundling [O12], the 2001 Microsoft settlement [O67], the 2023-2025 mobile rulings [O37], the DMA

[O39]. Markets for the base layer are made by whoever writes the rules of the gate - sometimes the gatekeeper, sometimes the state, and the contest between them is permanent.

Chapter 39 - Consolidation: what the buyers actually bought

The operating-system industry’s maturity can be read in its mergers, and the two largest tell complementary stories.

In July 2019, IBM paid roughly $34 billion, $190 a share in cash, for Red Hat [O25] - a company whose entire product was free to download. What IBM purchased was not code but position: the subscription relationship with nearly every large enterprise running Linux, the engineers who steer key upstream projects, and the hybrid-cloud story that Red Hat’s OpenShift (its enterprise Kubernetes) made sellable. Red Hat’s own numbers the year before the deal - $3.4 billion in revenue growing at 15 percent annually, a deferred-revenue balance of $3.0 billion [O24] - showed what the model had become: the software is free, the certainty is the product, and certainty renews.

Four years later, on 22 November 2023, Broadcom completed its acquisition of VMware [O66]. VMware had invented the modern x86 hypervisor business - the company was founded in 1998 by Diane Greene, Mendel Rosenblum and three colleagues, shipped Workstation in 1999, and then ESX Server, which let one physical server pretend to be many [O65]. The hypervisor is, functionally, an operating system under the operating system: it schedules whole machines instead of programs. Broadcom bought the toll booth underneath the datacenter - and then exercised the classic landlord’s privilege, restructuring licenses and prices in ways that sent many customers looking for exits. The lesson rhymes with Red Hat’s story from the other side: when the base layer is free software, the buyer sells trust; when it is proprietary, the buyer collects rent.

Microsoft’s own arc belongs in the same register. The company that once licensed DOS per machine now gives Windows away in its consumer upgrade program and makes its money from cloud, subscriptions and devices; its 2025 annual report files Windows OEM licensing inside “More Personal Computing”, one line in a company whose centre of gravity has moved to Azure [O55]. Even the greatest toll booth in software history became, in the end, a feeder for a cloud.

CentOS and the limits of stewardship

The consolidation story has a cautionary middle chapter. CentOS - the community rebuild of Red Hat Enterprise Linux, bit-for-bit compatible and free - had for fifteen years been the standard answer to “what do we run when we cannot pay Red Hat?” In December 2020, Red Hat, now inside IBM, announced CentOS Linux would end in favour of CentOS Stream, a rolling preview just ahead of RHEL rather than a copy behind it. The reaction was immediate: the founders of the replacement projects (Rocky Linux, AlmaLinux) raised institutional backing within weeks, and the clones re-formed. The episode is the commons’ antibody response: when a steward’s interests drift from its community’s, the code’s licence lets the community walk. Red Hat acted within its rights and possibly its necessities - CentOS had become a free substitute more than a funnel - but the trust cost is unquantifiable and real. Chapter 40’s argument about maintenance economics has a corporate mirror: the commons is only as reliable as its stewards’ incentives, and incentives change with ownership.

What the balance sheets say

The numbers, assembled, describe a mature industry that monetizes in four distinct ways. Apple takes hardware margin on a vertically integrated stack and services margin on the store. Microsoft takes cloud and subscription revenue with Windows as a feeder. IBM/Red Hat and the Linux vendors take subscription support and certification. Broadcom/VMware takes infrastructure rent under proprietary licences. The four models coexist because each exploits a different property of the base layer: integration, network effects, trust, and lock-in respectively. The market's total size is rarely stated as one number, but its components are public: Play Store profits exceeding $12 billion a year at 71 percent margins [O38], Red Hat's $3.4 billion at acquisition [O24], Windows's OEM line inside a $245-billion Microsoft [O55], and the embedded segment's design-win backlogs approaching a billion dollars per vendor [O46]. The base layer is not the biggest business in technology; it is the most leveraged - every other business sits on it.

The IBM-Red Hat integration, assessed

Six years on, the largest software acquisition ever made shows a specific pattern worth naming: the acquired commons-survivor kept its culture, and the acquirer changed around it. Red Hat still ships RHEL, Fedora and OpenShift on its own cadence; IBM's hybrid-cloud strategy - its entire modern positioning - is built on Red Hat's stack, and the older IBM middleware lines were quietly subordinated or wound down. The acquisition's logic, visible in hindsight, was that IBM bought the one asset it could not rebuild: credibility with the open-source developers and enterprise architects who decide infrastructure. Whether the CentOS Stream decision (this chapter) spent some of that credibility is the open question; the Rocky and Alma migrations measured it. The lesson for consolidation generally: when you buy a trust business, the asset is the trust, and it is re-earned daily in public.

VMware under Broadcom: the landlord's playbook

The VMware case is younger but sharper. Broadcom completed the acquisition on 22 November 2023 [O66] and moved immediately to the classic infrastructure-rent playbook: end perpetual licences in favour of subscriptions, consolidate the catalogue into a few bundles, reprice renewals toward the largest customers, and cut the partner channel's smaller tiers. Customer reaction was loud and measurable - public-sector and mid-market buyers began evaluating exits to KVM-based alternatives, Nutanix, and the public clouds - but the installed base of VMware virtualized workloads is so large, and migration so risky, that the repricing largely held. It is Chapter 13's lesson in modern dress: whoever owns the layer everything runs on can raise the rent, and the tenants' anger is a cost line, not an exit. The base layer's morality is economics all the way down, and it has not changed since 1969.

Chapter 40 - Cracks in the foundation

For all its scale, the industry's base layer rests on supports thinner than its users assume. Three cracks matter most.

The first crack is maintenance economics. Critical pieces of the world's operating infrastructure are looked after by handfuls of volunteers, paid little or nothing, invisible until they break. The clearest near-catastrophe came in March 2024, when a Microsoft engineer named Andres Freund, puzzled by half-second delays in SSH logins on a development machine, traced the cause into the release tarballs of xz Utils - a compression library

embedded in nearly every Linux system - and found a deliberately planted backdoor, assigned CVE-2024-3094 [O69]. The backdoor had been inserted by a contributor using the name Jia Tan, who had spent two years patiently earning the project's trust and maintainer access [O70]. Freund's public disclosure on 29 March 2024 [O70] landed days before the poisoned versions would have flowed into the major distributions. The investigation that followed read like a thriller and audited like a balance sheet: a load-bearing component of the world's computing was defended, in the end, by one overworked maintainer and one curious engineer with a free afternoon. Log4j in 2021 told the same story from another corner: the plumbing is communal, the plumbing is priceless, and the plumbing is understaffed.

The second crack is the end-of-life cliff. Operating systems outlive their makers' willingness to maintain them, and the retirements arrive on schedule whether the world is ready or not. Microsoft ended Windows 10 support on 14 October 2025; consumers and firms may buy Extended Security Updates only through October 2027 [O54]. Hundreds of millions of machines faced replacement or risk - a forced migration measured in e-waste as much as in licences. Windows 11 passed Windows 10 in installed share only in mid-2025, years after release [O72]. Beneath the headline versions, the long tail is older still: ATMs that ran Windows XP for years after its 2014 death [O53], factory controllers on systems whose vendors no longer exist.

The third crack is trust in the supply chain itself. A modern operating system assembles code from tens of thousands of contributors and hundreds of vendors; the xz episode showed how patiently that trust can be farmed. The industry's answers - reproducible builds, software bills of materials, signed provenance - are real but young. The base layer of the world economy is, in the end, a social artefact: it holds because enough of the right people keep showing up. Chapter 43 asks who, exactly, is still showing up.

Log4j and the volunteer economy

The Log4Shell vulnerability of December 2021 - a trivially exploitable remote-code-execution flaw in Log4j, a logging library present in a staggering share of enterprise Java systems - triggered the largest coordinated patching effort in software history. Log4j was maintained by a small volunteer team; its flaw lived in a feature (JNDI lookups) almost no user knew existed. The episode forced an accounting the industry had avoided: critical infrastructure, including government systems, ran on components maintained by people with no contract, no funding and no obligation. The responses - the OpenSSF's funding programs, the US executive order's software-bill-of-materials requirements, the EU's Cyber Resilience Act - all aim at the same target: making the supply chain's invisible dependencies visible and paid for. Progress is real and slow; the fundamental mismatch - value created in the commons, value captured at the applications - persists.

The certification moats

A subtler crack runs the other way: not too little assurance but assurance as barrier. In safety-critical domains - automotive ISO 26262, avionics DO-178C, common criteria for government systems - certification of an operating system costs millions of dollars and years of evidence-gathering. The cost is justified by the stakes and also functions as a moat: a certified RTOS incumbent is nearly impossible to displace, whatever a newcomer's merits, because recertification reprices the switch. This is the embedded world's version of lock-in, and it explains both the longevity of VxWorks and QNX and the enthusiasm of their corporate acquirers. The base layer's economics, once again: wherever switching is expensive - by network effect, by contract, by certificate - the layer's owner collects. The industry's most durable profits are not in being the best system, but in being the certified one.

The update treadmill

A fourth crack deserves its own paragraph because it compounds the other three: the patching economy. Every operating system in this book now ships security fixes on a drumbeat - Microsoft's second-Tuesday, Apple's rapid responses, the kernel's weekly stable updates - and the drumbeat is the price of the connected world: the code has defects, the adversaries are professional, and the window between disclosure and exploitation has shrunk from months to hours. The treadmill's hidden costs fall unevenly. Consumers get seamless background updates because Apple and Google rebuilt their platforms to permit them. Enterprises gate updates through change control and absorb the risk gap. And the long tail of Part VII - the ATMs, the vehicles, the medical devices - mostly cannot be patched at operational tempo at all, so its defence is isolation: network segmentation, signed firmware, and prayer. The industry's honest summary is that it built a world on software it must continuously repair, and the repair capacity is not evenly distributed. Whoever owns the update channel owns the machine's last line of defence - another gate, another toll, another set of arguments about who is allowed through.

The maintainer problem

The deepest of the cracks is organizational rather than technical. The modern stack rests on unpaid custodianship: libraries and tools maintained by one or two people, used by every company on earth, noticed only when they fail. The near-miss that made this visible was the xz Utils backdoor of March 2024. A contributor who had spent years building trust inside the project shipped versions 5.6.0 and 5.6.1 with a hidden modification to the build process that compromised the SSH authentication path - the front door of nearly every Linux server on the planet [O69]. It was caught by accident, by a Microsoft engineer who noticed an unaccountably slow login while benchmarking. Had the poisoned versions completed their journey into the stable releases of the major distributions, the result would have been the worst supply-chain compromise in the industry's history.

What the xz affair exposed is that the operating system's supply chain is not a chain at all; it is a gift economy that the largest corporations in history treat as a loading dock. The person at the center had been handed commit rights partly because the original maintainer - one individual, maintaining a compression library used by the world in his spare time - was exhausted and said so publicly. The response from the companies whose revenue rests on that gift economy was swift in proportion to the scare: Openwall and similar hardened-distribution projects published their assessments [O70], and the major platform vendors expanded funding programs and audit requirements for critical open-source components. But the structural condition remains. A system is only as sound as the least-rested person who can commit to it, and the industry has not yet found a market mechanism that pays for rest.

This is the maintenance chapter of the operating system's economics: the toll booths described earlier collect enormous rents from the finished stack, while the foundations underneath are held up by habit, pride and patchwork sponsorship. It is not a stable arrangement, and everyone who has examined it knows it is not a stable arrangement.

PART X - WHAT COMES NEXT

The final part looks forward with the field’s own tools: memory safety, verified kernels, the cloud’s quiet annexation of the operating system’s job, and the new politics of the stack. Rust’s arrival inside the Linux kernel in 2022 ended a fifty-year argument about C by changing the subject; seL4’s proofs ask whether correctness can be mathematics rather than testing; and Firecracker’s microVMs suggest the cloud has already split the operating system in two - a thin safe shell above, a planetary scheduler below.

Then the map shifts. Nations have decided that the operating system is strategic territory: HarmonyOS NEXT’s clean break from Android is the first mass-market phone system built explicitly so that no other country’s platform owner can switch it off, and post-quantum standards are forcing the world’s crypto plumbing to be rebuilt in place. The closing chapters ask what the ghost becomes when it is in everything - cars, bodies, grids, weapons - and end with the dependence audit: an honest reckoning of what this book’s subject now holds up, what it would cost to lose, and why the least visible layer of the modern world deserves the most attention of any of them.

Chapter 41 - Memory safety and the Rust wager

Most serious operating-system vulnerabilities share one root cause, and it is a detail of the C language: C trusts the programmer. A program that writes past the end of a buffer, or reads memory after freeing it, compiles happily and fails catastrophically - and Microsoft’s and Google’s security teams have each reported that around seventy percent of their severe bugs are such memory-safety errors. Half a century of the world’s most trusted software rests on the discipline of individual programmers never slipping.

The industry’s answer is a language that refuses the slip. Rust enforces memory safety at compile time through its ownership rules: the compiler, not the programmer’s care, guarantees that no two references fight over the same memory and no reference outlives its data. In 2021 Rust for Linux entered the kernel tree as an experimental subsystem; by the end of 2022, kernel 6.1 - released 11 December 2022 - shipped with initial Rust support in mainline [O64], the first new implementation language the Linux kernel has admitted in over three decades. Android has pushed further and reports results: memory-safety vulnerabilities in Android fell sharply as new code shifted to memory-safe languages. Microsoft is rewriting parts of the Windows kernel in Rust. The transition will take a generation - tens of millions of lines of C do not retire - but the direction is set. The wager underneath it is philosophical: that reliability should be engineered into the language, not begged from the programmer. It is the same instinct, carried in syntax instead of theorems, that drives seL4’s proofs and QNX’s tiny kernel - and it suggests the industry’s centre of gravity is finally drifting toward the reliability pole of Chapter 37’s dial.

The seventy percent

The statistic that moved the industry was assembled independently by its two largest security teams. Microsoft’s MSRC and Google’s Chromium project each reported, in 2019, that about seventy percent of their serious vulnerabilities were memory-safety issues - the same classes of bug (buffer overflow, use-after-free)

that Morris’s worm exploited in 1988. Thirty years of safer libraries, static analysis, fuzzing and exploit mitigations had moved the number barely at all. The conclusion the platforms drew was that the problem was the language, not the programmers: C and C++ make the correct thing the careful thing, and at a hundred million lines of code, careful is not a plan. Android’s data since adopting Rust for new code is the strongest natural experiment in the field: as the share of new code in memory-safe languages rose, memory-safety vulnerabilities fell from 76 percent of Android’s total in 2019 to 24 percent in 2024. The mechanism is unglamorous - the compiler rejects entire bug classes before they ship - and that is its strength: it does not depend on anyone’s vigilance.

What a language migration means at kernel scale

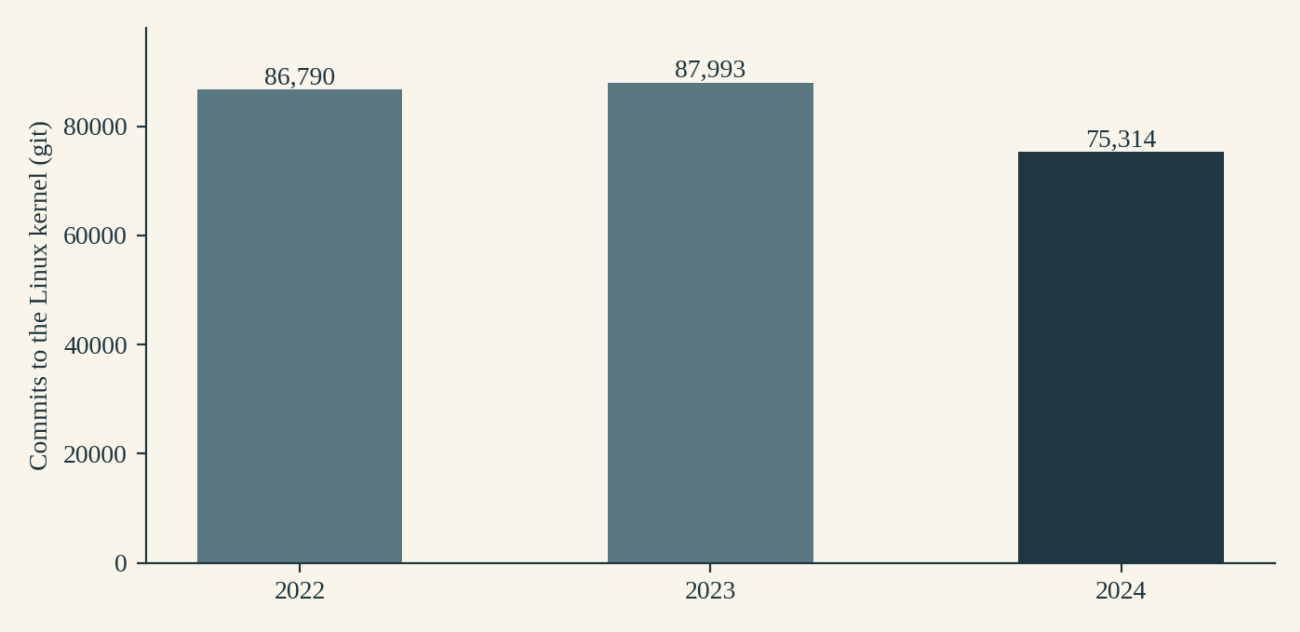
The Linux kernel’s Rust support, landing in 6.1 in December 2022 [O64], is deliberately modest: drivers and new subsystems first, bound by strict rules about where “unsafe” code may live. The social negotiation was harder than the technical one - maintainers who have reviewed C for thirty years must now review Rust, and the kernel’s development model assumes any maintainer can read any code. The project’s own statistics [O26] give the scale: 1.3 million commits, 29,380 authors, nearly 40 million lines. No migration will rewrite that mass; the strategy is generational - new code safer by default, old code gradually contained by tooling and hardware (memory-tagging extensions are arriving in ARM and future Intel silicon for the same reason). The wager’s horizon is decades. But the direction is notable precisely because kernels are conservative institutions: when the most cautious engineering culture in software changes its language, the argument is over.

Beyond Rust: the formal frontier

Memory-safe languages are the middle path between C’s trust and the extreme end, where the frontier has also moved. seL4’s 2009 proof [O52] established that a kernel’s code can be shown, machine-checked, to implement its specification; subsequent work extended the guarantees toward security properties and down to the compiled binary. CompCert did the same for a C compiler. The cost remains brutal - person-decades per ten thousand lines, and proofs age as code changes - so formal verification is not coming to the mainstream kernel; but its existence changes what “assurance” can mean in the markets that buy it (Chapter 30), and its techniques keep leaking into ordinary engineering as stronger type systems, verified micro-libraries and model-checked protocols. The trajectory of the whole arc is worth stating plainly: the industry spent fifty years treating reliability as a testing problem, discovered that testing does not scale with consequences, and is now - slowly, expensively, and for the first time with real momentum - treating it as a mathematics problem. The 1992 debate’s losing side (Chapter 20) is winning the long game.

What Rust feels like in the kernel

The practitioner’s version is less grand and more telling. A Rust driver in the Linux kernel comes with bounds-checked slices, ownership-checked lifetimes, and compiler errors that read like a pedantic maintainer - which is precisely the point: the reviewer’s burden moves from memory to logic. Early adopters report the same pattern Android’s data shows at scale: whole bug classes stop appearing in review, and the bugs that remain are the interesting ones - wrong hardware assumptions, race conditions of intent, not accidents of the heap. There is a cultural cost: the kernel community’s shared language is C, and a second language splits reviewing attention. The kernel’s compromise - Rust for new drivers and subsystems where the safety payoff is highest, C for the core - is conservative in the way infrastructure should be. The ghost changes slowly. But it changes.



Commits to the Linux kernel, 2022-2024 [O26]. Chart drawn from the cited figures.

Chapter 42 - The cloud ate the operating system

The last two decades' biggest operating-system story is the operating system's partial disappearance. Three technologies did the erasing.



The cloud’s machine room: the operating system industrialized, a million schedulers out of sight.

Containers came first. Where a virtual machine carries a whole guest operating system per application, a container shares the host kernel and isolates only the process and its filesystem view - milliseconds to start, megabytes not gigabytes. The Linux kernel features that made it possible (namespaces, cgroups) date from the 2000s; Docker, from 2013, made them usable; and Kubernetes made them an economy. The first commit to Kubernetes landed on 6 June 2014 - 47,501 lines [O62] - as an open-sourced descendant of Google’s internal Borg system, whose design lessons a small team pitched inside Google in the summer of 2013 [O71]. A decade later it orchestrates a large share of the world’s containerized workloads, and its real significance is architectural: the unit of management stopped being the machine and became the workload. The operating system below became, for many developers, an implementation detail.

Serverless completed the erasure. AWS Lambda and its peers run functions without any visible machine at all; underneath, Amazon built Firecracker, a stripped-down virtual-machine monitor written in Rust, written up publicly in 2018, that boots microVMs in about 125 milliseconds and carries Lambda and Fargate [O63]. Even the hypervisor shrank toward a kernel’s outline.

What remains is telling. The cloud did not abolish the operating system; it industrialized it. Somewhere under every function sits a Linux kernel and a scheduler doing the same work CTSS did in 1961 - deciding who runs next - now for tenants who will never know its name. The ghost became infrastructure: everywhere, essential, and invisible unless you go looking.

Borg to Kubernetes

Google’s internal container story began years before the public one: Borg, the cluster manager that has scheduled Google’s workloads since the early 2000s, taught the company that the unit of datacenter computing

is the task, not the machine. When a small team pitched an open-source successor in the summer of 2013 [O71], the design goal was to give everyone else the operational lessons of Borg without its Google-specific scaffolding. Kubernetes' first commit, 6 June 2014 [O62], was 47,501 lines; within three years it had won the orchestration market against Docker Swarm, Mesos and Cloud Foundry, largely by being the neutral ground every cloud could support - the same no-owner logic as Linux, one layer up. The Linux Foundation's CNCF became its home, and the major cloud providers all ship managed Kubernetes despite it commoditizing their own differentiation: the commons again, this time in the orchestration layer.

What remains yours

The honest accounting of the cloud era is that "the operating system disappeared" only from some people's view. Developers who deploy functions never see a kernel; the hyperscalers' fleets run millions of them, patched live, at a scale of systems administration humanity has never previously attempted. The skills concentrated: a few thousand engineers at Amazon, Microsoft and Google now perform work that a million sysadmins once did locally, with automation and discipline that the old machine-room operators (Chapter 2) would recognize as the final victory of their craft. The risk concentrated too: a bad kernel update at a hyperscaler, or a faulty agent pushed to a fleet - as the July 2024 CrowdStrike incident demonstrated when a security agent's update crashed millions of Windows machines worldwide - becomes a global event in minutes. The cloud did not remove the base layer's risks; it pooled them. Chapter 44's question - who guards the ghost - is, in the cloud era, a question about a handful of companies' operational discipline, on which everyone now depends.

The hypervisor's second life

Underneath the cloud's first decade sat a technology this book met in 1966: the virtual machine. IBM's CP/CMS had invented the idea of one machine pretending to be many [O11]; VMware industrialized it for x86 from 1998 [O65]; and the hyperscalers built their original cloud instance model on Xen and later KVM - the Linux kernel itself turned hypervisor. The instance was the unit of sale: a virtual machine, billed by the hour, that looked and behaved like hardware. Containers and then serverless (Firecracker's 125-millisecond microVMs [O63]) compressed the same idea toward the workload's true size, but the accounting unit of the modern economy - the "instance-hour", the function invocation - descends directly from CP-40's discovery that time-sharing could be nested: a machine sharing machines. IBM's Cambridge laboratory would recognize the business model immediately.

The kernel's datacenter phenotype

The Linux that runs the cloud is recognizably the hobby of 1991 and visibly a different animal: tuned by hyperscaler engineers for million-core fleets, live-patched without reboots, stripped of drivers no VM will ever need, and instrumented so deeply that a single scheduler regression becomes a measurable revenue event. The kernel's own statistics mark the shift: of the roughly 75,000-88,000 commits landing each year [O26], the plurality now comes from companies - Intel, Google, Red Hat, Microsoft, Meta, Amazon - whose business is the fleet, not the box. The commons adapted to being infrastructure by professionalizing: subsystem maintainers are mostly employed to do what volunteers once did. Whether that is the commons succeeding or being absorbed is a live argument, and Chapter 40's cracks - the xz maintainer alone at the bottom of the stack - are its sharpest evidence that the absorption is incomplete where it matters most.

The machine room returns

The physical fact of the cloud is a return to the book’s opening image. Behind every serverless function and every rented instance is a building full of machines, and on those machines runs a supervisor that allocates hardware among strangers - the hypervisor, which is an operating system whose only processes are other operating systems. VMware built the first great business on this, founding in 1998 on the proposition that one x86 machine could be safely subdivided [O65]; the hyperscalers then rebuilt the proposition at a scale VMware never imagined, on the open-source Xen and KVM hypervisors, and the economics inverted: the guest systems are the product, the host is the cost.

The clearest sign of where that leads is Firecracker, the virtual-machine monitor Amazon wrote for Lambda and released openly in 2018 [O63]. Firecracker is a deliberate minimalism: a microVM stripped to roughly fifty thousand lines, built on KVM, that starts in a fraction of a second, because Amazon’s metered business needed millions of tiny, fast, disposable supervisors rather than a few large durable ones. The operating system inside those microVMs is Linux, usually invisible to the customer, patched by the provider, billed by the millisecond. The user of a cloud function has approximately the relationship to their operating system that a 1965 time-sharing customer had to theirs: it is there, it is shared, it is someone else’s machine, and the bill is computed by the supervisor itself.

What the cloud did not change is the queue. The names are new - availability zones instead of machine rooms, capacity reservations instead of batch windows - but the discipline underneath is the one GM-NAA and CTSS invented: keeping the most expensive machines in the building from standing idle between jobs. The machine room learned to rent itself out by the second. That is the whole transformation, and it is enormous, and it is also the same supervisor’s arithmetic, sixty years on.

Chapter 43 - Forks, nations and silicon

Three slower shifts will shape the next decade of operating systems more than any product launch.

The first is geopolitical. Operating systems have become strategic assets, and nations now act accordingly. China’s HarmonyOS NEXT, announced in January 2024 and shipped commercially with the Mate 70 series that November [O41][O42], dropped the Linux kernel and Android compatibility entirely - a deliberate bid for a sovereign software stack, with Huawei claiming its own Ark compiler, its own kernel, and an ecosystem rebuilding from zero. The bet is expensive and unprecedented in scale: no platform since iPhone OS has asked the world’s developers to start over, and no government-backed one has ever carried this many users. Whether it succeeds or becomes the industry’s most instructive failure, it establishes that operating-system independence is now industrial policy.

The second shift is in silicon. The open RISC-V instruction set is moving from microcontrollers toward application processors, and with it the prospect of a fully open stack - open hardware design, open kernel, open userland - that no single company or country can revoke. Linux has run on RISC-V for years; the question is when a competitive RISC-V phone or laptop arrives, and whose.

The third shift is cryptographic. On 13 August 2024, NIST published the first three finalized post-quantum encryption standards - FIPS 203, 204 and 205 [O68] - and every operating system now faces a decade-long migration: TLS libraries, VPN stacks, disk encryption, secure boot and update signing must all learn new

mathematics before a cryptographically relevant quantum computer exists. It is the largest coordinated change to the base layer since the move to 64 bits, and almost no user will ever notice it happened - which is the operating system's oldest story. Its greatest victories are the disasters that never occur.

HarmonyOS: the sovereign stack

Huawei's path is the purest test of whether a new operating system can still be born at scale. Cut off from Google's licensed Android by the 2019 Entity List, Huawei first shipped a compatibility play - HarmonyOS built on the open-source Android core - and then, in January 2024, announced HarmonyOS NEXT, which removes the Android compatibility layer and the Linux kernel entirely [O41]. The commercial version shipped with the Mate 70 series in November 2024 [O42]. The stack is wholly Huawei's: kernel, Ark compiler, distributed frameworks, and an app ecosystem that had to be rebuilt application by application - tens of thousands ported in the first year, an industrial effort with no precedent since 2007. The strategic reading is Chapter 27's lesson internalized: a platform whose legal or supply-chain foundation can be revoked is not a platform but a tenancy. The engineering question - can a from-scratch stack match two decades of Linux hardening - remains genuinely open, and the answer will be measured in the world's largest consumer market, with state encouragement behind it.

RISC-V and the open substrate

The silicon layer underneath is opening at the same time. RISC-V, the Berkeley-born open instruction set, removes the last proprietary dependency in the stack: anyone may design a processor without licensing ARM or x86. Linux has run on RISC-V since the late 2010s; Android's port is underway; China's chip strategy has embraced it as the escape from export controls on ARM and Intel designs. The full-open stack - RISC-V silicon, Linux kernel, open userland - would be the first computing platform in history with no proprietary chokepoint from transistor to application. Whether it becomes a commercial force or remains a strategic hedge is the decade's question; its existence already changes negotiations, as open alternatives always do. And in cryptography, the migration mandated by NIST's August 2024 post-quantum standards [O68] will touch every kernel, TLS stack and secure-boot chain on earth - the kind of invisible, decade-long, civilization-scale maintenance the base layer exists to perform.

The AI question

The newest claimant on the operating system's future is artificial intelligence, and the honest assessment is two-layered. At the hardware layer, the shift is already here: NPUs in every phone and laptop, GPU clusters as the new mainframes, and kernels learning to schedule heterogeneous silicon - the z16's on-chip AI accelerator [O14] and Android's neural-network interfaces are early examples of the OS absorbing a new resource class, as it once absorbed floating-point units and GPUs. At the interface layer, the claim is larger: that natural-language agents will replace applications as the way users command the machine, demoting the app model that has governed since 2008. If that happens, the gate moves again - from the store to the agent - and the antitrust arc (Chapter 38) gets a fourth round. The base layer's record counsels humility about predictions and attention to incentives: whoever owns the agent's runtime will own the new gate, and everyone in this book's story knows it.

The constants

Against three genuine unknowns - sovereign stacks, open silicon, the AI interface - the constants are worth restating. Someone will own the scheduler. Someone will sign the updates. Someone will decide what the hardware is allowed to be told. Those decisions will be made, as they always have been, by a mixture of engineering judgment, commercial incentive and state pressure, and they will matter to everyone who never thinks about them. The form the operating system takes in 2040 is unknowable; that there will be one, doing CTSS's job on hardware CTSS's designers would find indistinguishable from magic, is as close to a certainty as this industry offers.

When nations fork

The newest forks in the family tree are drawn by borders rather than by engineering taste. Huawei's HarmonyOS NEXT is the sharpest case: after the 2019 export restrictions cut its access to Google's licensed Android, Huawei rebuilt its phone operating system from its own kernel upward, and the Mate 70 series, shipped in late 2024, runs a HarmonyOS with the Android Open Source Project compatibility removed entirely - not a reskin of Android, as earlier versions substantially were, but a separate operating system with its own application format and its own store [O41][O42]. A fork that starts as sanctions compliance becomes, if it survives, a separate ecosystem with its own developers, its own habits and its own toll booth. That has not happened at handset scale since Symbian.

The regulatory fork runs the other way: not a new system, but new terms imposed on old ones. The European Union's Digital Markets Act, applied from 2023, names gatekeepers and prescribes what their operating systems must permit - alternative app stores, sideloading, choice screens, access to the same hardware features the platform's own software uses [O39][O40]. The compliance work lands in the kernel's neighborhood: every capability that must be opened to competitors must first be made safe to open, which means new permission machinery, new attestation paths, new surfaces to secure. Apple and Google now maintain measurably different operating systems inside the EU than outside it. The operating system, for sixty years a product shaped by its maker's judgment and its customers' pressure, has acquired a third author.

Neither fork is finished, and both may fail - HarmonyOS may remain a domestic system, and the DMA's openings may prove too narrow to change habits. But the direction is set: the era of one global operating system per device class is ending, and the map of operating systems is starting to look like the map of everything else.

Chapter 44 - The ghost in everything

Stand back from the seventy years and a shape appears. The operating system began as a timesaving hack on a machine that cost more than a jet - an operator's assistant on the IBM 704, a "supervisor" on ATLAS. It became a product when IBM bet $5 billion on making one span a whole family; a profession when the same company, under legal pressure, priced software separately in June 1969; an industry when Microsoft kept the rights to the PC's nervous system; a commons when a student in Helsinki posted a hobby note and the world's engineers answered; a utility when the datacenter, the supercomputer and the phone quietly standardized on it; and a utility again, differently, when Apple and Google turned it into the gate between three billion people and everything digital.

At every stage, the same logic repeated: whoever owns the base layer owns the economics above it, and the base layer wins not by being the best technology but by being the one everyone builds on. That is why the arguments over it never stop - Tanenbaum and Torvalds in 1992, Epic and Google in 2023, Beijing and Cupertino now. The subject of those arguments keeps changing costume: batch monitor, time-sharing supervisor, PC ROM, Internet kernel, app-store gatekeeper, cloud substrate. The function never changes. Something must stand between the application and the metal, deciding, protecting, pretending.

The next time a phone wakes to a face, a car brakes itself, a trade clears, or a rover on Mars waits out a dust storm for its flight computer to come back, the thing doing the waking and braking and clearing and waiting is a direct descendant of programs written to stop people standing around watching a computer idle. The most successful machine humanity has ever built for organizing other machines was invented to manage queues. It has been managing them ever since - seventy years, and counting.

The people

A word about the humans, because this industry tends to erase them. The operating system was built by identifiable people working in small rooms: Patrick and the GM team; Corbato’s group at MIT; Thompson and Ritchie in an attic lab; Kildall at his naval postgraduate desk; Paterson’s six weeks; Cutler’s DEC transplants; Stallman’s one-man campaign; Tanenbaum’s teaching project; a Helsinki student; the unnamed thousands whose commits [O26] now carry the kernel forward at seventy-five thousand changes a year. It was also built by institutions that behaved, at crucial moments, against their own interests: GM giving away its monitor, IBM publishing its architecture and then unbundling its software, AT&T’s lawyers letting Unix escape, Berkeley sharing TCP/IP, Google open-sourcing Android and Kubernetes. The mix - individual obsession, institutional accident, and the commons’ arithmetic - is not a flaw in the story. It is the story. The base layer of the digital world is a social achievement before it is a technical one.

The wager this book has made

The series’ premise is that industries are best understood through their material logic - what things are made of, who must pass through whose gates, where the physics and the economics bind. The operating system has no material: it is weightless, copyable, invisible. And yet it has behaved exactly like the physical industries of the other volumes - concentration at the gate, standards as treaties, consolidation at maturity, the state arriving last. The ghost, it turns out, obeys the economics of the bridge and the railway. That is the final reason it deserves a book: not because it is exceptional, but because it is the purest case - an industry with no raw material except coordination, which nonetheless built the same toll booths, fought the same wars, and ended, like all of them, as infrastructure: indispensable, regulated, and forgotten by the people it serves.

The unbuilt alternatives

A history defined by winners should spare a paragraph for the roads the industry did not take, because they were real. Plan 9 from Bell Labs pushed Unix’s file abstraction to its limit and stayed a cult. The Lisp machines and Smalltalk workstations ran the language as the operating system - no kernel boundary at all - and lost to C’s portability. BeOS, Jean-Louis Gassee’s multimedia system of the mid-1990s, demonstrated a responsive design the incumbents took a decade to match, and nearly became Apple’s future before NeXT did. OS/2 ran Windows programs better than Windows and died of the arrangement. Each failure had the same shape: a better answer to a question the market had stopped asking, arriving after the gate had moved. The lesson generalizes beyond software. In layered industries, the decisive layer is won early and defended by

switching costs, and the window for a new entrant opens only when the whole stack shifts - to the desktop, to the network, to the pocket. Chapter 43’s contenders are all betting that the stack is shifting again. They may be right. The losers’ shelf says the bet is expensive.

A field guide to noticing the ghost

The book closes where it began: the operating system is invisible by design, but not undetectable. You can feel the scheduler in the split second before a heavy app wakes. You can hear the file system in the fan that spins up when a thousand small writes flush at once. You can watch virtual memory’s con in the momentary freeze when a swapped-out browser tab is touched. You can see the protection boundary in every permission dialog, the driver layer in every printer that refuses, the update mechanism in every “restart to finish”. The interface between you and the machine is mostly this book’s subject, wearing its best suit. Once you have seen it, the modern world reads differently: not a cloud, not an app, not a phone, but seventy years of supervisors - in the data centre, in the dashboard, in the pocket - still deciding, still protecting, still pretending. The machine room got bigger. The boss never left.

Epilogue - The dependence audit

Every book in this series ends by asking what the thing’s ubiquity means for the people who depend on it. The operating system’s answer is uncomfortable in a particular way: it is the one industry in this series where dependence is total and attention is zero.

Consider what would stop if the base layer stopped. Not applications - everything. The ledger of the world economy, the control loops of its power grids and aircraft, the communications of its governments and the memories of its families all execute inside schedulers, virtual memory managers and file systems written by a few hundred thousand people, living and dead, most of them unthanked and some of them unknown. The Transistor book in this series describes the hardware that made computing cheap; the Submarine Cable book describes the glass that carries the traffic. This book’s subject is the layer in between - the part that decides, at every nanosecond, what the silicon is actually doing. The transistor made computation abundant; the operating system made it trustworthy enough to build a civilization on.

Three honest conclusions follow. First, the base layer is more concentrated than the free surface suggests: two kernels - Linux and NT - plus Apple’s hybrids and a handful of embedded specialists carry nearly everything, and consolidation is still running. Second, the commons model that rescued the industry from single-company control has its own fragility, and the xz backdoor showed how thin the margin can be. Third, the contest is not over: memory safety, quantum migration, sovereign stacks and open silicon are live fights, and their outcomes will be set in the next decade by people working, mostly, out of sight.

The user said it best in 1991, unknowingly, in a newsgroup post about a hobby. The thing that “won’t be big and professional” became the most professional thing there is - by becoming, like the best infrastructure, invisible. This book has tried to make it visible for a few hundred pages. The next time the screen obeys, you will know what you are looking at: seventy years of supervisors, kernels and ghosts, still standing between the machine and everyone who needs it.

The audit, itemized

Concretely, the dependence this book has documented resolves into a short inventory of single points of failure. Two kernels - Linux and NT - carry the overwhelming majority of the world's server, cloud, embedded and desktop computation; Apple's XNU derivatives carry the wealthiest slice of the client world; a handful of RTOS kernels carry its safety-critical machines. Two signing authorities - Apple and Google - control software installation for three-quarters of humanity's personal computing [O1][O2][O4]. Three clouds operate the majority of the shared fleet. One open-source compression library nearly carried a backdoor into all of it [O69]. Each concentration emerged from rational local decisions - standardization's genuine efficiencies, the commons' genuine advantages - and each is individually defensible. The inventory is still the inventory: the species has placed its memory, its money, its communications and its machines' reflexes into a narrower set of hands than it has ever tolerated in any physical industry, and it has done so without ever quite deciding to.

What to watch

The closing advice is therefore not alarm but attention. Four signals will announce the next turn of this industry, and all four are visible in this book's last chapters. Watch the Rust migration's velocity: if memory-safe code reaches the kernel's core paths, the vulnerability economy changes [O64]. Watch the DMA's enforcement and the mobile injunctions' effects: if genuine store competition survives contact with the incumbents' compliance strategies, the toll model reprices [O37][O39]. Watch HarmonyOS NEXT's trajectory in the world's largest market: a successful sovereign stack ends forty years of the Unix-Windows duopoly's assumption of permanence [O41][O42]. And watch the maintainers: the commons' succession - who reviews the kernel when its founders retire, who funds the libraries when the grants lapse - is the least technological and most consequential variable in the whole system. The ghost in everything turns out to be, in the end, people. It always was.

GLOSSARY

Address space - The private range of memory addresses a process is allowed to see, created by the memory-management hardware and the kernel working together.

API (application programming interface) - The published set of services an operating system offers programs; stable APIs are what let software written decades ago still run.

Batch processing - The 1950s practice of collecting jobs and feeding them to the machine in a stream without an operator between each one; the operating system's original purpose.

Boot (bootstrap) - The self-starting sequence that loads the operating system when a machine powers on, named for pulling oneself up by one's bootstraps.

Container - An isolated workload that shares the host machine's kernel; lighter and faster than a virtual machine, and the unit of the modern cloud.

Context switch - The kernel's act of saving one running program's state and loading another's; the heartbeat of multitasking.

Daemon - A background process that waits for work - printing requests, network connections, scheduled tasks - named for Maxwell's helpful demon.

File system - The layer that organizes raw disk blocks into named files and directories, and keeps them intact across crashes and power cuts.

Hypervisor (virtual-machine monitor) - Software that runs whole operating systems as guests on one physical machine; an operating system for operating systems.

Interrupt - A hardware signal that diverts the processor to the kernel to handle an event - a keystroke, a disk completion, a timer tick.

Kernel - The always-resident core of the operating system with full hardware privilege; everything else asks it for services.

Microkernel - A design that keeps the kernel minimal and moves drivers and services into isolated user-space processes; slower historically, safer structurally.

Monolithic kernel - A design where the kernel and its services share one privileged address space; fast, and the mainstream choice of Unix, Linux and (hybrid) Windows.

POSIX - The portable operating-system interface standard that let Unix-like systems share programs; the treaty that ended the Unix wars.

Privilege ring - Hardware-enforced levels of trust; user programs run unprivileged, the kernel in ring 0.

Process - A running program with its own address space and resources; the operating system's basic unit of protection and scheduling.

Scheduler - The kernel component that decides which ready process runs next; the oldest continuous function in computing.

Shell - The command interpreter between user and system; on the first Unix it was itself just another program, a small revolution in flexibility.

Supervisor - The 1950s-60s word for the resident control program; the direct ancestor of "kernel".

System call - The controlled doorway through which an unprivileged program requests a privileged service - open a file, send a packet, fork a process.

Time-sharing - Interleaving many users on one machine fast enough that each feels sole possession; invented at MIT in 1961.

Unbundling - IBM's June 1969 decision to price software separately from hardware, creating the independent software industry.

Virtual machine - A software-simulated computer; IBM's CP/CMS pioneered it in the 1960s, VMware industrialized it, the cloud consumed it.

Virtual memory - The illusion, maintained by hardware and kernel together, that every process owns a large private, contiguous memory.

Copyleft - The licensing mechanism, pioneered by the GNU GPL, that requires modified versions of free code to remain free; copyright turned against enclosure.

Ecosystem - The platform's true moat: the accumulated applications, developers, habits and data that make leaving irrational.

Firmware - Software burned into a device's permanent memory; the operating system's smaller, slower-changing sibling.

Gatekeeper - The platform owner who controls passage between users and developers; the regulatory category the DMA made law.

Jailbreak / root - Obtaining the operating system's highest privilege against the vendor's wishes; the user's counter-move against the sealed platform.

Live patch - A kernel fix applied without rebooting; the datacenter's answer to the restart's cost.

OEM (original equipment manufacturer) - The device maker who preinstalls the operating system; the channel through which DOS, Windows and Android were each won.

Open core - The business model of an open base with proprietary services on top; Android's, and increasingly the industry's, standard structure.

Real time - See Chapter 30: not fast, but provably on time.

Root of trust - The first piece of code the hardware unconditionally believes, from which all secure boot chains descend.

Sandbox - An execution compartment that limits what a program can reach; the phone's unit of distrust.

Sideloading - Installing software outside the platform's official store; the act the mobile antitrust cases made a right.

System image - The complete, signed snapshot of an operating system as shipped to devices; the unit the update mechanism replaces.

Toll booth - This book's name for the recurring economic position: ownership of the layer everyone must pass through.

APPENDIX A - TIMELINE

1956 - GM-NAA I/O, the first production operating system, runs on the IBM 704 at General Motors Research [O8]. **1961** - CTSS demonstrated at MIT; compatible time-sharing begins [O15]. **1964** - April 7: IBM announces System/360 and OS/360, one architecture and one operating system for a whole family [O10]. **1966** - CP/CMS virtual machines at IBM's Cambridge Scientific Center. **1969** - June 23: IBM unbundles software from hardware; the software industry begins [O12]. The first Unix system takes shape at Bell Labs on a PDP-7 [O17]. **1973** - CP/M written by Gary Kildall [O56]. **1980** - Microsoft buys QDOS and licenses PC DOS to IBM; August: Digital Research misses the meeting [O56]. **1981** - August: IBM PC ships with PC DOS; Microsoft keeps the rights [O56]. **1983** - September: Richard Stallman announces GNU [O61]. **1984** - January: Macintosh; the graphical mainstream begins. **1985** - November: Windows 1.0 [O58]. Mach microkernel work begins at CMU [O51]. **1991** - August: Linus Torvalds announces his hobby kernel [O21]. **1992** - January: the Tanenbaum-Torvalds debate [O19]. 386BSD. **1993** - July: Windows NT 3.1, Dave Cutler's all-new kernel [O58][O60]. Debian founded. **1995** - August: Windows 95; the Rolling Stones soundtrack a taskbar [O59]. **1996** - December: Apple buys NeXT for $429 million; Steve Jobs returns [O34]. **1997** - July: Mars Pathfinder lands running VxWorks; priority inversion nearly ends the mission [O44]. **1998** - June:

Symbian founded [O27]. VMware founded [O65]. **2000** - QNX released for embedded systems (founded 1980 as QUNIX) [O45]. **2001** - October: Mac OS X and Windows XP ship, months apart [O58]. November: US v. Microsoft settles [O67]. **2004** - October: Ubuntu 4.10 [O23]. **2005** - July: Google buys Android for an estimated $50 million [O31]. **2007** - January 9: the iPhone; November: the Open Handset Alliance [O29] [O32]. **2008** - July: the App Store opens with 500 apps [O30]. September: the T-Mobile G1 [O33]. Symbian Foundation announced; Nokia buys Symbian outright [O28]. **2009** - seL4 functional-correctness proof completed. **2010** - February: webOS; Symbian's source briefly open. **2011** - February: Nokia's burning-platform memo; Windows Phone bet [O35]. **2013** - May: ISS laptops switch to Debian [O49]. Summer: Kubernetes pitched inside Google [O71]. **2014** - April: Windows XP support ends; ATMs scramble [O53]. June: first Kubernetes commit [O62]. **2015** - July: Microsoft writes off $7.6 billion on Nokia; phone hardware exits [O36]. **2017** - November: all 500 fastest supercomputers run Linux [O6]. **2018** - November: Firecracker published; serverless gets a microkernel-shaped hypervisor [O63]. **2019** - July: IBM completes the $34 billion Red Hat acquisition [O25]. **2020** - December: CentOS refocused to Stream; Rocky and Alma born. **2021** - May: Android passes three billion active devices [O1]. **2022** - April: IBM z16 with on-chip AI inference [O14]. December: Rust enters the Linux kernel in 6.1 [O64]. **2023** - November: Broadcom completes the VMware acquisition [O66]. December: Epic v. Google verdict [O37]. **2024** - March: the xz Utils backdoor is caught [O69][O70]. August: NIST publishes post-quantum standards [O68]. November: HarmonyOS NEXT ships commercially on Mate 70 [O42]. **2025** - January: Apple reports 2.35 billion active devices [O2]. October 14: Windows 10 support ends [O54]. **2026** - Android holds roughly 72 percent of mobile; Windows about 77 percent of desktop [O4][O3].

The timeline records the industry's habit of arriving early and mattering late: most entries above were noted, at the time, as minor product news. Read vertically, the seven decades describe one migration - from one machine per institution, to one per desk, to one per person, to one per thing - with the operating system re-founding itself at each scale.

APPENDIX B - THE FAMILY TREE

Almost every operating system in use today descends from one of four ancestors. The tree below is a simplification - code, ideas and people cross-pollinated constantly - but it is the right first map.

The IBM line - GM-NAA I/O (1956) and IBSYS lead to OS/360 (1966), whose descendants still run: z/OS on the mainframe, and the VM/CMS line through CP-40 and CP-67 to today's z/VM. CICS and DB2 run atop. The property that defined the family - backward compatibility as religion - means some production binaries trace their lineage to punch-card-era code.

The Unix line - Bell Labs Unix (1969) forks into the commercial Unices (AIX, HP-UX, Solaris, IRIX) and the Berkeley line (BSD, 1978 onward). BSD forks into NetBSD, FreeBSD and OpenBSD, and - through NeXTSTEP's Mach-plus-BSD hybrid - into Apple's XNU family: macOS, iOS, iPadOS, watchOS, tvOS. From the GNU project's userland plus Torvalds' kernel (1991) comes Linux: Android, ChromeOS, the cloud, the supercomputers, the routers, the televisions. Minix (1987) was the teaching spark; seL4 shares the microkernel instinct though not the code. Nearly all mobile, cloud and embedded computing lives on this branch.

The Microsoft line - QDOS/86-DOS (1980) becomes MS-DOS/PC DOS (1981), which grows Windows (1985) as a shell, then an environment, then - through the 95/98/ME line, ended 2000 - a consumer operating system. In parallel, Cutler's NT (1993), built with OS/2-era ambitions and DEC VMS experience, becomes Windows 2000, XP, 7, 10, 11 and every Windows Server. Since XP (2001), one kernel family has carried all of Microsoft's clients; DOS survives only as heritage and emulation.

The embedded line - The real-time branch runs on different ancestry: VxWorks (1987, from Wind River, founded 1981), QNX (1982, founded 1980), INTEGRITY, ThreadX, FreeRTOS (kernel lineage to 2003, stewarded by AWS from 2017), Zephyr and AUTOSAR stacks. These systems rarely meet consumers directly, but they outnumber everything else on the tree: cars, spacecraft, medical devices, routers, factory floors. seL4 (2009) anchors the formally verified end.

Reading the tree teaches one lesson above all: survival belongs to portable designs. The operating systems alive in 2026 are the ones that let go of one machine - Unix left the PDP-7, Linux left the 386, NT left the i860, iOS left the phone for the wrist and the tablet. The ones that stayed loyal to hardware - CP/M, classic Mac OS, Palm OS, Symbian, DOS itself - are museum pieces, however beloved.

APPENDIX C - THE MARKET IN NUMBERS

Figures below are snapshots from the sources cited in the companion volume; shares move monthly and are stated with their period. They are offered as orientation, not precision.

Mobile, worldwide, October 2026 (StatCounter [O4]): Android 72.4 percent; iOS 27.6 percent; all others under 1 percent combined. Android passed three billion active devices in May 2021 [O1]; Apple reported 2.35 billion active devices of all kinds in January 2025 [O2].

Desktop, worldwide, November 2026 (StatCounter [O3]): Windows about 76.7 percent; OS X and macOS together about 15.7 percent; Linux about 5.6 percent; ChromeOS about 2.0 percent. Windows 11 overtook Windows 10 only in mid-2025 [O72]; Windows 10 exited support on 14 October 2025 [O54].

Across all device types (StatCounter, mid-2025 [O72]): Android leads all operating systems at about 47.7 percent; Windows follows near 24.7 percent; iOS near 16.9 percent. The most-used operating system in the world is a phone system.

Servers and infrastructure (W3Techs, November 2026 [O5]): Unix-family systems power about 92 percent of websites whose operating system is known; Windows about 8 percent. All 500 of the TOP500 supercomputers have run Linux since November 2017 [O6]. Kubernetes, first committed 6 June 2014 [O62], orchestrates containerized workloads across the major clouds; AWS's Firecracker microVM monitor carries Lambda and Fargate [O63].

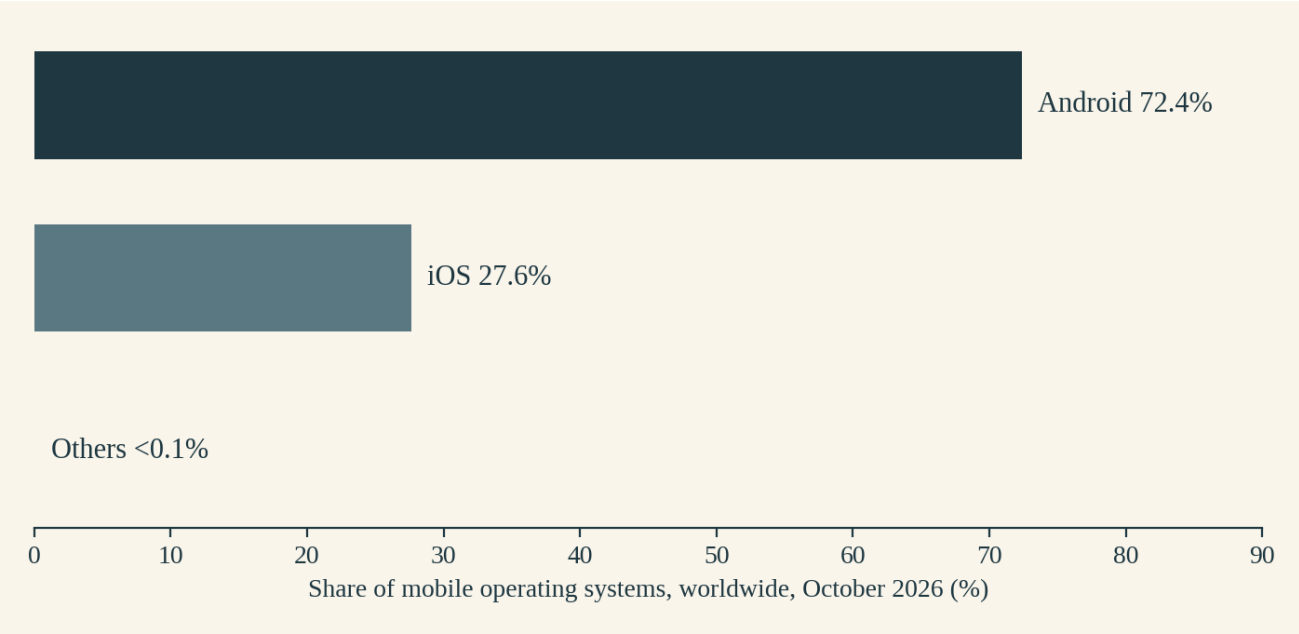
Embedded : QNX reports its software in over 275 million vehicles, with a design-win backlog around $950 million [O46]. AUTOSAR's Adaptive Platform standardizes the high-performance end of the car [O47]. FreeRTOS and Zephyr carry the microcontroller billions. VxWorks flies on Mars - Pathfinder in 1997 [O44], Ingenuity's companion systems later, with the helicopter itself on Linux and F Prime [O48].

The economics : Google's Play Store operating margin rose from 26 percent in 2014 to 71 percent in 2021 on over $12 billion in annual revenue, per court-disclosed figures [O38][O37]. Red Hat's revenue reached $3.4 billion before IBM's $34 billion acquisition [O24][O25]. Microsoft files Windows OEM licensing inside a

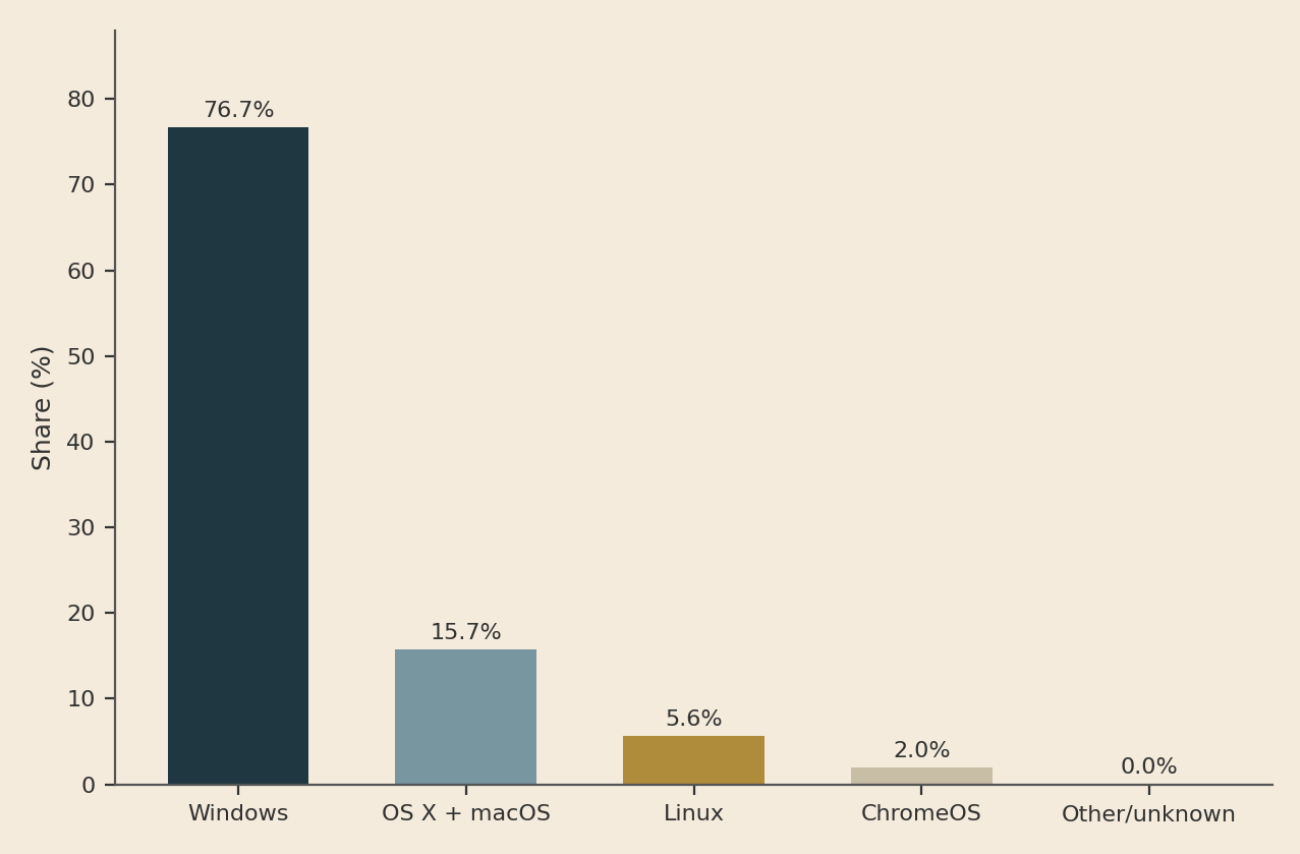
segment whose centre of gravity is now cloud [O55]. IBM’s z16 (2022) shows the mainframe still shipping new silicon [O14]. The Linux kernel received roughly 75,000-88,000 commits per year in 2022-2024 from thousands of contributors [O26].

The pattern : every layer where one operating system achieved majority share, the economics pooled at the gate - IBM’s account control in 1969, Microsoft’s per-PC royalty in the 1980s, the app-store commission in the 2010s. The numbers above are the current state of an old game.

A note on method: web-measured shares (StatCounter, W3Techs) count usage, not machines; vendor-disclosed figures (device counts, margins) count whatever the vendor chooses to disclose, on the vendor’s definitions; and court-disclosed figures, while sworn, describe the period of the case. Where the three disagree, this book prefers court records, then vendor disclosures, then panel measurements - and treats all of them as the current reading of a moving quantity, which is why each figure above carries its period with it.



Mobile operating-system share, worldwide, October 2026 [O4]. Chart drawn from the cited figures.



Desktop operating-system share, India, November 2026 [O3]. Chart drawn from the cited figures.