STUDY NOTES

Operating Systems

From the first punch card to the AI-powered OS of 2026 — plus a step-by-step build guide

Compiled **July 2026** · **2 parts, 21 sections** · Reference edition

01 What Is an Operating System?

The one program every computer must run before it can run anything else.

An **operating system (OS)** is the software that sits between a computer's hardware and every application you use. It does not do useful work itself — a browser or a game does that. Instead, it manages the machine's resources (the processor, memory, storage, and devices) and hands them out safely and efficiently to whatever program asks for them.

Think of it as an air traffic controller for a computer. Dozens of programs (planes) want to use the same limited resources (runways) at once. The OS decides who goes when, prevents collisions, and keeps everything moving without any one program crashing the whole system.

WHY IT MATTERS

Without an OS, every application would need its own code to talk directly to the keyboard, disk, and screen. The OS provides one shared, tested layer so software developers can write "print this file" instead of controlling a printer's circuitry by hand.

The Core Job, in One Sentence

An OS converts raw, inconvenient hardware into a convenient, consistent set of services — files, windows, networking, security — that applications and people can actually use.

02 Core Functions

Every OS, from a 1960s mainframe to an iPhone in 2026, does the same six jobs.

Process Management

Decides which program gets the CPU, for how long, and switches between them so fast it looks simultaneous.

Memory Management

Allocates RAM to each running program and keeps them from reading or overwriting each other's data.

File System Management

Organizes data into files and folders on disk, and controls who can read, write, or delete them.

Device Management

Talks to hardware — printers, keyboards, GPUs, sensors — through small driver programs, hiding the hardware detail from apps.

Security & Access Control

Authenticates users, isolates programs from each other, and enforces permissions (this is most of what a modern OS update is about).

User Interface

Presents the system through a command line, a graphical desktop, or a touchscreen.

03 History & Evolution

Eight decades, condensed to the moments that actually changed how computers were used.

1940s	**No operating system at all** Operators fed instructions in directly via switches and punch cards. Each program controlled the entire machine, one at a time.
1956	**GM-NAA I/O** Widely cited as the first operating system, built by General Motors for IBM's 704 mainframe to automate job scheduling ("batch processing").
1961–69	**Time-sharing is born** MIT's CTSS (1961) and the Multics project (1969) let multiple users share one computer interactively instead of waiting for batch jobs — the idea that later shaped Unix.
1964	**IBM OS/360** The first OS designed to run across an entire family of computers, not just one model — the template for modern, portable operating systems.
1969–73	**Unix, at Bell Labs** Ken Thompson and Dennis Ritchie build Unix, then rewrite it in their new C language (1973) — making it portable to almost any hardware. Directly or indirectly, Unix is the ancestor of Linux, macOS, iOS, and Android.
1981	**MS-DOS** Microsoft's disk operating system ships on the IBM PC, becoming the standard for early personal computers — text commands only, no graphics.
1984	**The Macintosh System Software** Apple ships the first mass-market computer with a graphical user interface — windows, icons, a mouse — built on ideas developed at Xerox PARC.
1985	**Windows 1.0** Microsoft's answer to the graphical interface, initially just a shell running on top of MS-DOS.
1991	**Linux** Linus Torvalds releases the Linux kernel as a free, open-source Unix-like alternative. It goes on to run most of the internet's servers, Android phones, and eventually AI data centers.

1995	**Windows 95** Introduces the Start menu and plug-and-play hardware, and becomes the first Windows built as a full consumer OS rather than a DOS add-on.
2001	**Mac OS X & Windows XP** Apple rebuilds its OS on a Unix foundation (Mac OS X). Microsoft merges its consumer and business Windows lines into one (XP).
2007–08	**iOS and Android** The iPhone launches with what becomes iOS; Google follows with Android, built on the Linux kernel. Mobile becomes its own OS category, optimized for touch and battery life.
2015–20	**The OS becomes a service** Windows 10 moves to continuous rolling updates instead of boxed releases. Apple moves Macs to its own ARM-based silicon (2020), unifying the chip architecture across iPhone, iPad, and Mac.
2021–26	**AI moves into the OS layer** Windows 11 ships with an integrated AI assistant (Copilot); Apple adopts year-based version numbers across iOS, iPadOS, and macOS (both jumped to "26" in 2025); Android and Linux continue twice-yearly and rolling release models. See Section 6 for exact 2026 versions.

04 Types of Operating Systems

Not one design — a family of designs, each optimized for a different constraint.

Type	What it optimizes for	Example
Batch OS	Throughput — runs jobs in queued groups with no user interaction	Early mainframes, GM-NAA I/O
Time-Sharing OS	Fairness — splits CPU time in slices so many users feel they have the computer to themselves	Unix, Multics
Distributed OS	Scale — spreads one workload across many networked machines that act as one system	Google's internal systems, Kubernetes-managed clusters
Real-Time OS (RTOS)	Guaranteed timing — a response must arrive within a fixed deadline, no exceptions	Car airbag controllers, pacemakers, factory robots
Mobile OS	Battery life & touch input	iOS, Android
Embedded OS	Minimal footprint on single-purpose hardware	Smart thermostats, routers, appliances
Network OS	Sharing files, printers, and security across a local network	Windows Server, Novell NetWare (historic)

05 The Major OS Families Today

Windows (Microsoft)

Dominant on desktop and laptop PCs worldwide. Closed-source, backward-compatible with decades of software, now ships continuous feature updates rather than numbered "versions" every few years.

macOS & iOS (Apple)

Both descend from Unix via Apple's Darwin core. Tightly integrated with Apple's own hardware (Apple Silicon chips). Since 2025, Apple names both by the year they ship (e.g., "26"), replacing the old sequential numbers (Mac OS X 10.x, iOS 1–18).

Linux (open source, many distributions)

Free and open-source kernel maintained by thousands of contributors. Powers the majority of web servers, cloud infrastructure, supercomputers, and — through Android — most of the world's smartphones. Distributions (Ubuntu, Fedora, Debian, and others) package the kernel with different tools and interfaces.

Android (Google)

Built on the Linux kernel with a custom application layer (originally Java/Kotlin-based). The most-used OS on Earth by device count, spanning budget phones to premium flagships.

06 Latest Updates — as of July 2026

Where each major OS stands right now, and what's arriving next.

OS	Current stable release	Released	What's next
Windows	Windows 11, version 25H2 (July 2026 Patch Tuesday cumulative update adds Point-in-Time Restore, Bluetooth & File Explorer fixes)	Rolling monthly	Version **26H2** ships fall 2026 as a lightweight enablement package (~200KB, installed via one restart)
macOS	macOS 26 "Tahoe" — build 26.5.2	29 Jun 2026	macOS 27 "Golden Gate" announced at WWDC 2026, ships fall 2026
iOS / **iPadOS**	iOS 26.5	11 May 2026	iOS 26.6 in beta testing now; iOS 27 ships fall 2026
Android	Android 16 (stable, most devices)	Jun 2025	Android 17 "Cinnamon Bun" in public beta on Pixel 10 series; Google has moved to two updates a year
Linux kernel	Linux 7.1 (mainline) · 6.18.21 (LTS branch)	14 Jun 2026	Kernel development continues on its usual ~9-week release cycle

PATTERN TO NOTICE

Every major vendor has moved away from "one big yearly release." Windows and Apple now ship small, frequent, low-risk updates and reserve one larger release for autumn. Android has split its yearly release into two. The Linux kernel has always worked this way. The direction of travel across the whole industry is smaller, faster, safer updates.

07 Where Operating Systems Are Heading

AI as a built-in system service

Apple Intelligence, Windows Copilot, and Gemini on Android are no longer separate apps — they are becoming OS-level services that other apps can call, the same way apps call the file system today.

Security by architecture, not by add-on

Sandboxing every app by default, verified boot chains, and hardware-backed keys are shifting from "optional feature" to baseline requirement across all major platforms.

One identity across many devices

Phone, laptop, watch, and headset increasingly behave as one continuous system (Handoff, Continuity, cross-device clipboards) rather than separate machines that happen to share an account.

Spatial and ambient computing

visionOS and similar platforms point toward operating systems designed around 3D space and always-on sensors, not just a rectangular screen.

08 Quick Glossary

Kernel

The core of the OS that runs with full hardware privileges and manages CPU, memory, and devices directly.

Shell

The command interpreter (text-based, like Bash) that lets a user or script instruct the kernel.

GUI

Graphical User Interface — windows, icons, and pointers, as opposed to typed commands.

Multitasking

Running multiple programs by rapidly switching the CPU between them.

Virtual memory

A technique that lets programs use more memory than physically exists by borrowing disk space.

Driver

A small program that lets the OS talk to a specific piece of hardware.

API

Application Programming Interface — the set of functions an OS exposes so apps can request its services.

Distribution (distro)

A complete, installable package built around the Linux kernel plus tools and a desktop environment.

Sources: Microsoft Support & Windows Insider Blog, WindowsLatest, Windows Central (Windows 11 25H2/26H2) · Macworld & MacRumors (macOS 26/27, iOS 26) · 9to5Google & Android Developers (Android 16/17) · Kernel Newbies & LWN.net (Linux 7.1). Compiled July 2026 — check each vendor's site for changes after this date.

PART TWO

Building an Operating System — Step by Step

A hands-on, ground-up walkthrough of how a real (if small) operating system gets built: from power-on to a working shell. Same ideas real kernels like Linux and the Windows/macOS cores use, stripped down to teaching size.

09 Before Any Code: Design Decisions

Every real OS project fails or succeeds on decisions made before the first line of code.

1. Pick a kernel architecture

Architecture	How it works	Used by
Monolithic	Entire OS (scheduler, file system, drivers, network stack) runs in one privileged address space. Fast — no messaging overhead — but a crashing driver can crash everything.	Linux, original Unix
Microkernel	Only the bare minimum (scheduling, inter-process messaging, basic memory management) runs privileged. Drivers and file systems run as normal, isolated processes.	Minix, QNX, seL4
Hybrid	Monolithic core for speed, with some services (drivers, graphics) pushed out to isolated components for safety.	Windows (NT kernel), macOS (XNU)

2. Pick a target and a language

Most teaching kernels target **x86-64** (documented, emulator-friendly) or increasingly **ARM64**. The kernel itself is written "freestanding" — C or Rust with no standard library, since there is no OS underneath it yet to provide one.

3. Write down goals and non-goals

Before coding, decide what the project will *not* do. A teaching kernel that tries to support Wi-Fi, USB, a GUI, and a package manager on day one never boots. The real skill in OS design is scoping — the same discipline that made Unix's "do one thing well" philosophy last fifty years.

> **RULE OF THUMB**
>
> Build the smallest thing that can print "Hello, kernel" to the screen and halt cleanly. Every step after that adds exactly one capability at a time.

10 Step 1 — Set Up the Toolchain

You cannot compile a kernel with your everyday compiler — it assumes an OS is already there to link against.

Cross-compiler (x86_64-elf-gcc) | NASM / GAS assembler | GNU binutils & ld | QEMU | GDB | Make

1. **Build a cross-compiler**
   A compiler that runs on your machine but targets "no operating system" (`--target=x86_64-elf`). Using your host's regular gcc would silently assume Linux/macOS system headers exist.

2. **Install an assembler**
   NASM (Intel syntax) or GAS (AT&T syntax) for the small hand-written assembly stubs — boot code and context switches — that C cannot express.

3. **Install an emulator**
   QEMU boots your kernel image in seconds without touching real hardware. Bochs is slower but simulates the CPU instruction-by-instruction, which is invaluable when debugging boot failures.

4. **Wire up GDB**
   QEMU can expose a GDB remote-debugging port (`-s -S`), letting you set breakpoints inside your kernel exactly like a normal program.

The **OSDev Wiki** is the de facto reference for every step in this part — treat it as the primary source, not this summary.

11 Step 2 — The Bootloader

The first 512 bytes the CPU ever runs, before anything resembling an operating system exists.

When a PC powers on, firmware (BIOS or UEFI) reads the first sector of the boot disk into memory address `0x7C00` and jumps to it — but only if the sector's last two bytes are the signature `0x55 0xAA`. Everything from here up is code you write.

```
; minimal 16-bit boot sector (NASM)
[org 0x7C00]
start:
    mov si, msg
    call print_string
    jmp $                ; hang forever

print_string:
    lodsb
    or al, al
    jz done
    mov ah, 0x0E         ; BIOS teletype interrupt
    int 0x10
    jmp print_string
done:
    ret

msg db "Booting...", 0

times 510-($-$$) db 0    ; pad to 510 bytes
dw 0xAA55                ; boot signature
```

Two practical paths from here:

- **Write your own Stage 2 loader** — the 16-bit boot sector loads a bigger second-stage program from disk, which then does the real work of finding and loading the kernel. Full control, more work.
- **Use GRUB with the Multiboot2 specification** — GRUB does the messy disk-loading and mode-switching for you and hands off to your kernel with a clean, documented machine state. Recommended for beginners.

12 Step 3 — Leaving Real Mode

The CPU boots in a 1970s-compatible mode. Modern kernels need to leave it almost immediately.

Real mode (the CPU's power-on state) only addresses 1 MB of memory and has no memory protection — any program can overwrite any other. A real kernel must step the CPU through two transitions:

1. **Enable the A20 line**
   A historical quirk: early CPUs wrapped memory addresses above 1MB back to zero. The 21st address line has to be explicitly switched on before you can use more memory.

2. **Load a minimal GDT and enter Protected Mode**
   Set bit 0 of the CR0 register after loading a Global Descriptor Table (see Section 13), then perform a far jump to flush the CPU's instruction pipeline into 32-bit mode.

3. **Build page tables and enter Long Mode**
   To reach full 64-bit mode, paging must already be enabled (Section 15) and the `EFER.LME` bit set — then a second far jump lands the CPU in 64-bit Long Mode, where the real kernel finally runs.

WHY THIS FEELS BACKWARDS

Modern 64-bit chips still boot as if they were a 1978 8086, for full backward compatibility. Every 64-bit OS you have ever used — Windows, Linux, macOS — walks through this exact same real → protected → long mode sequence on every single boot.

13 Step 4 — Kernel Entry & Linking

Getting from "raw assembly" to "a C function is now running."

The bootloader (or GRUB) needs a fixed place to jump to. A tiny assembly stub sets up a stack pointer — C code cannot run without one — then calls the C `kmain()` function:

```
; boot.asm
global _start
extern kmain
section .bss
  stack_bottom: resb 16384
  stack_top:
section .text
_start:
    mov esp, stack_top     ; C needs a valid stack
    call kmain
    cli
.hang: hlt
    jmp .hang
```

A **linker script** then tells the linker exactly where in memory each section (`.text` code, `.rodata` constants, `.data` and `.bss` variables) should live, and where execution should begin. Many production kernels use a **higher-half design** — mapping the kernel to the top of the virtual address space (e.g. `0xFFFFFFFF80000000`) so the low addresses stay free for user programs.

14 Step 5 — The GDT in Detail

The table that tells the CPU which memory ranges exist and who is allowed to touch them.

The Global Descriptor Table defines segments — regions of memory with a base address, a size limit, and permissions. Every real OS defines at minimum:

Entry	Purpose
Null descriptor	Required filler; the CPU rejects a GDT that doesn't start with one
Kernel code segment	Executable, ring 0 (most-privileged) — where kernel instructions run
Kernel data segment	Read/write, ring 0 — kernel variables and stack
User code segment	Executable, ring 3 (least-privileged) — where application instructions run
User data segment	Read/write, ring 3 — application variables and stack
Task State Segment (TSS)	Holds the kernel stack pointer to use when an interrupt arrives while in ring 3

After loading the table with the `lgdt` instruction, every segment register must be reloaded — this is the actual reason the far jump in Step 3 is required, not just a formality.

15 Step 6 — Interrupts & the IDT

How a kernel reacts to a key press, a timer tick, or a program dividing by zero.

The **Interrupt Descriptor Table** maps each of the CPU's 256 interrupt numbers to a handler function. Three categories matter:

CPU exceptions (0–31)

Divide-by-zero, invalid opcode, general protection fault, and — most important — the page fault (#14), which the memory manager relies on directly (Section 16).

Hardware IRQs (32+)

The timer and keyboard fire these. The legacy 8259 PIC (or modern APIC) must be reprogrammed first, since it defaults to IDT slots that collide with CPU exceptions.

Software interrupts

A program deliberately triggers one (historically `int 0x80`, or the dedicated `syscall` instruction on x86-64) to ask the kernel for a service — this is the actual mechanism behind every system call in Section 21.

DEBUGGING TIP

An unhandled CPU exception with no valid handler triggers a **double fault**; if that handler is also broken, the CPU gives up entirely with a **triple fault** — which looks like an instant, silent reboot in QEMU. It is the single most common failure new OS developers hit.

16 Step 7 — Memory Management & Paging

Turning a flat pile of RAM chips into isolated, protected address spaces for every process.

Physical memory: what exists

The bootloader (or Multiboot2 info structure) reports which RAM regions are usable versus reserved. The kernel tracks free physical pages (4 KB blocks) with a bitmap or a free-list allocator.

Virtual memory: what a program sees

Paging translates the addresses a program uses into real physical addresses, through a chain of lookup tables (four levels on x86-64: PML4 → PDPT → PD → PT). This buys three things at once: isolation between processes, the illusion of more memory than physically exists, and the ability to mark pages read-only or non-executable.

1. **Identity-map the kernel**
   So the code doesn't disappear out from under itself the instant paging turns on.

2. **Load CR3 with the top-level page table**
   Then set the paging bit in CR0 — from this instruction forward, every memory access goes through translation.

3. **Write a page fault handler**
   When a program touches an unmapped or protected address, the CPU calls interrupt 14 instead of crashing — the kernel decides whether to map in a new page, swap from disk, or kill the offending process.

17 Step 8 — Dynamic Memory (the Heap)

Giving the kernel its own version of `malloc()` — there is no libc to borrow one from.

The simplest working allocator is a **bump allocator**: keep a pointer to the next free byte, and move it forward on every allocation. It never frees memory, but it's enough to get early kernel structures running. The next step is a **free-list allocator** that tracks freed blocks and reuses them, and eventually a **slab allocator** (as Linux uses) that pre-carves memory into fixed-size chunks for common object types to avoid fragmentation entirely.

Allocator	Trade-off
Bump	Trivial to write, cannot free memory — fine only for permanent kernel data
Free-list	Can reuse freed memory; suffers fragmentation over time
Slab	Fast, low fragmentation for same-sized objects; more complex to implement

18 Step 9 — Processes & Privilege Levels

Turning a file on disk into a running, isolated program.

Each process is represented by a **Process Control Block (PCB)** — a struct holding its saved registers, page table pointer, process ID, state (ready / running / blocked), and open file handles. Creating a process means: allocate a PCB, build it a fresh page table, load its code (typically an ELF binary) into memory, and set up an initial stack.

Ring 0 vs. Ring 3

The kernel runs in ring 0 (full hardware access); ordinary programs run in ring 3 (restricted — no direct hardware I/O, no arbitrary memory access). Moving from ring 3 to ring 0 only happens through controlled doors: interrupts, exceptions, or a system call — never by the program's own choice.

WHY THIS IS THE WHOLE POINT

Every security property people expect from an OS — one app can't read another's password, a crashing app doesn't crash the whole machine — comes directly from this ring separation, enforced by hardware and set up by exactly the paging and GDT work done in Sections 14 and 16.

19 Step 10 — CPU Scheduling

Deciding, dozens of times per second, which process gets the processor next.

Algorithm	How it decides
First-Come, First-Served	Runs processes in arrival order; simple but one long process blocks everything behind it
Round Robin	Each process gets a fixed time slice, then moves to the back of the queue — the easiest scheduler to implement first
Priority Scheduling	Higher-priority processes run first; needs an aging mechanism or low-priority work starves forever
Completely Fair Scheduler (CFS)	Linux's default — tracks each process's accumulated runtime and always picks whichever has run the least, approximating equal CPU shares

A working first scheduler needs only: a ready queue, a timer interrupt (Section 15) firing at a fixed rate, and a **context switch** — save the current process's registers to its PCB, load the next process's registers, and return into different code than the interrupt originally came from.

20 Step 11 — Device Drivers

The code that actually talks to hardware, one register at a time.

Port-mapped I/O

Dedicated CPU instructions (`in` / `out`) read and write hardware registers on a separate address space from RAM — used by the legacy PS/2 keyboard controller and PIT timer.

Memory-mapped I/O

Hardware registers appear at fixed addresses inside normal memory — used by VGA text mode (`0xB8000`) and virtually all modern devices.

First three drivers to write

Screen: write character + color bytes directly to VGA text memory.
Keyboard: IRQ1 fires per keypress; translate the scancode it delivers into ASCII.
Timer (PIT): IRQ0 fires at a fixed rate, driving both the scheduler's clock and elapsed-time tracking.

21 Step 12 — System Calls

The one and only doorway a user program has into the kernel.

A system call table maps small integer numbers to kernel functions (`0 = read`, `1 = write`, `60 = exit`, and so on — Linux's actual table has several hundred entries). A user-space program loads the syscall number and arguments into fixed registers, then executes the dedicated `syscall` instruction (or the older `int 0x80`), which triggers a controlled jump into ring 0 at exactly the address the kernel registered — never anywhere else.

```
; user-space wrapper, x86-64 SysV calling convention
write:
    mov rax, 1        ; syscall number for write
    ; rdi, rsi, rdx already hold the arguments
    syscall
    ret
```

Everything a program can do to the outside world — print text, open a file, create another process — ultimately funnels through this narrow, checked interface. It is the single most important security boundary in the entire system.

22 Step 13 — File Systems

Turning a flat block device into files, folders, and permissions.

A disk is just a numbered sequence of fixed-size blocks — the file system layer imposes structure on top:

Concept	Role
Superblock	Fixed location describing the whole file system: total size, block size, free block count
Inode	One per file — stores metadata (size, permissions, timestamps) and pointers to the data blocks holding its contents
Directory entry	Maps a human-readable filename to an inode number
Free-space map	Bitmap tracking which blocks are unused and available

Most teaching kernels start with a trivial **read-only initial RAM disk (initrd)** — the whole file system loaded into memory at boot, no real disk driver required — before attempting a real, persistent format like FAT32 or ext2. A **Virtual File System (VFS)** layer above this lets the kernel support multiple file system formats behind one common `open/read/write/close` interface, exactly as Linux does.

23 Step 14 — User Space & the Shell

The moment the kernel stops being the whole computer and starts being a platform for other programs.

Reaching "user space" requires a minimal freestanding C library (a tiny `libc`) providing wrappers around the system calls from Section 21, then loading and running the first user program — traditionally called **init**. Everything else, including a command shell that reads text and runs programs like `ls`, `cat`, and `echo`, is an ordinary ring-3 program built on that library — no different in principle from any app you install today.

MILESTONE

The instant a typed shell command produces output on screen, every layer below it — bootloader, paging, interrupts, scheduler, syscalls, file system — has been proven to work end to end.

24 Step 15 — Testing, Debugging & Emulation

A kernel bug doesn't throw a stack trace — it reboots the machine.

1. **Run in QEMU first, always**

   `qemu-system-x86_64 -kernel mykernel.bin` boots in under a second and is safe to crash repeatedly.

2. **Attach GDB for real breakpoints**

   `qemu -s -S` pauses at boot and waits for `gdb -ex "target remote :1234"` — step through kernel code exactly like an application.

3. **Log over the serial port**

   Screen output vanishes on a crash; writing debug text to the emulated serial port (captured to a host file) survives a triple fault.

4. **Drop to Bochs for the hard cases**

   Slower than QEMU but simulates every instruction individually and can dump full CPU/register state on a triple fault — the only reliable way to see exactly what caused one.

25 Case Study — Mapping This to a Real Boot

Every step above has a direct counterpart in the OS you're reading this on.

Stage	Toy kernel (this guide)	Real-world equivalent
Firmware	BIOS reads sector 0	UEFI firmware (modern PCs), iBoot (Apple)
Bootloader	Hand-written stage1/stage2, or GRUB	GRUB2/systemd-boot (Linux), Windows Boot Manager, iBoot stages (Apple)
Mode switch	Real → Protected → Long mode	Identical sequence on every x86-64 OS
Kernel init	kmain(), GDT, IDT, paging	Linux's `start_kernel()`, Windows' `ntoskrnl.exe` init, XNU's bootstrap
Early file system	initrd in RAM	Linux `initramfs`; Windows Boot Manager's boot image
First user process	Hand-written init/shell	Linux `systemd` (PID 1); macOS `launchd`; Windows `wininit.exe`

26 Common Pitfalls

- **Triple faults with no error message.** Almost always a broken IDT, an unhandled exception, or a bad stack pointer. Add serial logging before debugging further.
- **Forgetting to reload segment registers** after loading a new GDT — the CPU keeps using stale segment descriptors until every register (CS, DS, ES, SS...) is explicitly reloaded.
- **Misaligned stack on function calls.** The x86-64 ABI requires the stack to be 16-byte aligned at a `call` instruction — violating this silently corrupts SSE instructions deep in library code.
- **Off-by-one errors in page table indices.** A single wrong bit shift in the four-level paging math maps the wrong physical page — and looks exactly like random memory corruption.
- **Enabling interrupts too early.** If the IDT, PIC remapping, and a valid kernel stack are not all in place first, the very first timer tick crashes the kernel before it has done anything visible.

27 Tools, Resources & Where to Go Next

Reference material

The **OSDev Wiki** is the definitive, continuously updated reference for every topic in this part. *Operating Systems: Three Easy Pieces* (free online) covers the theory behind processes, memory, and file systems in more depth than any single guide can. MIT's **6.828**/**6.S081** course materials, built around the teaching kernel **xv6**, are widely used as a structured, guided path.

Real teaching kernels worth reading

xv6 (MIT) | SerenityOS | ToaruOS | Minix | Redox OS (Rust)

Reading a small, complete kernel's source end to end teaches more than any tutorial — every concept in this guide appears there as real, working code.

28 Build Checklist

The fifteen steps from this part, as a single project roadmap.

Step	Milestone	Difficulty
1	Cross-compiler + assembler + QEMU installed and building a stub	Low
2	Boot sector prints text and hangs cleanly	Low
3	CPU reaches 64-bit Long Mode	Medium
4	C `kmain()` runs, linked at the correct address	Low
5	Custom GDT loaded, segment registers reloaded	Medium
6	IDT installed; exceptions and IRQs print a message instead of rebooting	Medium-High
7	Paging enabled; page fault handler working	High
8	Kernel heap allocator (`kmalloc` / `kfree`) working	Medium
9	First user process runs in ring 3	High
10	Round-robin scheduler switching between two+ processes	High
11	Keyboard, screen, and timer drivers working	Medium
12	System call interface (at least `write` and `exit`)	Medium
13	Read-only initrd file system mounted	Medium
14	Shell running as a normal ring-3 program	Medium
15	Serial logging + GDB workflow in place	Low

Part Two is an educational simplification of concepts documented in depth by the OSDev Wiki, MIT 6.828/6.S081, and the published source of Linux, xv6, and similar teaching kernels. It describes general, well-established OS construction techniques, not the internal source of any specific commercial OS.