

Build Your Own OS

This project takes you from the raw BIOS power-on sequence all the way to a preemptively scheduled multi-process kernel with user-mode isolation and system calls. You will write the bootloader that fits in 512 bytes, configure the x86 segmentation and paging hardware, wire up the programmable interrupt controller, and implement a context-switching scheduler in assembly. Every abstraction that higher-level programmers take for granted — malloc, printf, threads, file descriptors — is absent here. You build the foundation those abstractions stand on.

The project is structured as four milestones that mirror the historical evolution of OS design: bootstrap and protected mode, interrupt infrastructure, memory management, and process scheduling. Each milestone reveals a hardware constraint that software must negotiate: the CPU's privilege ring model, the TLB's stale-translation problem, the PIC's vector-collision hazard, and the TSS's role in safe ring transitions. Skipping any of these reveals results in triple faults, silent memory corruption, or security holes.

By the end, you will have a mental model precise enough to read Linux kernel source code, understand hypervisor design, and reason about why system calls have the latency they do. The OS you build is simple; the conceptual leverage it provides is enormous.

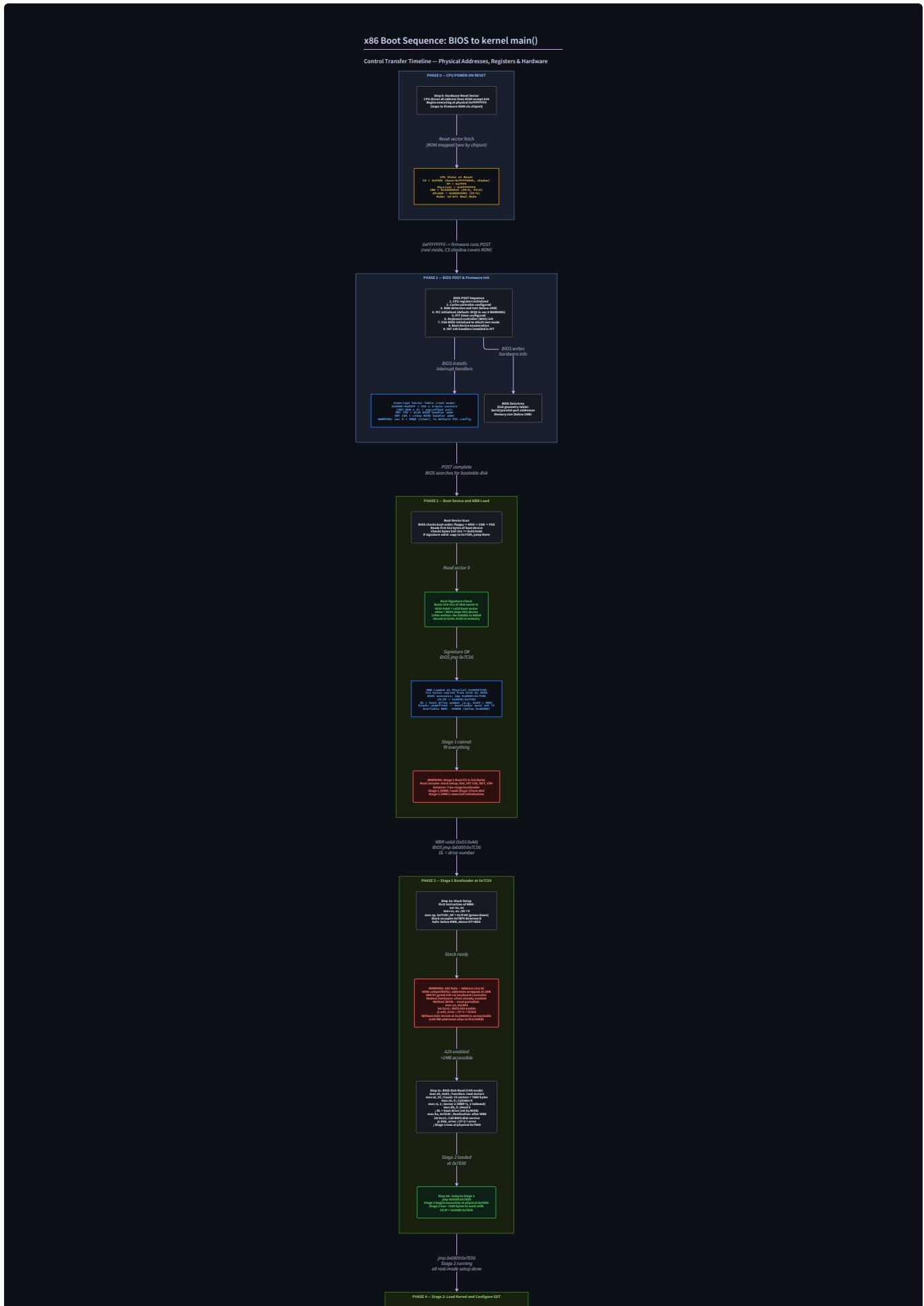
Milestone 1: Bootloader, GDT, and Kernel Entry

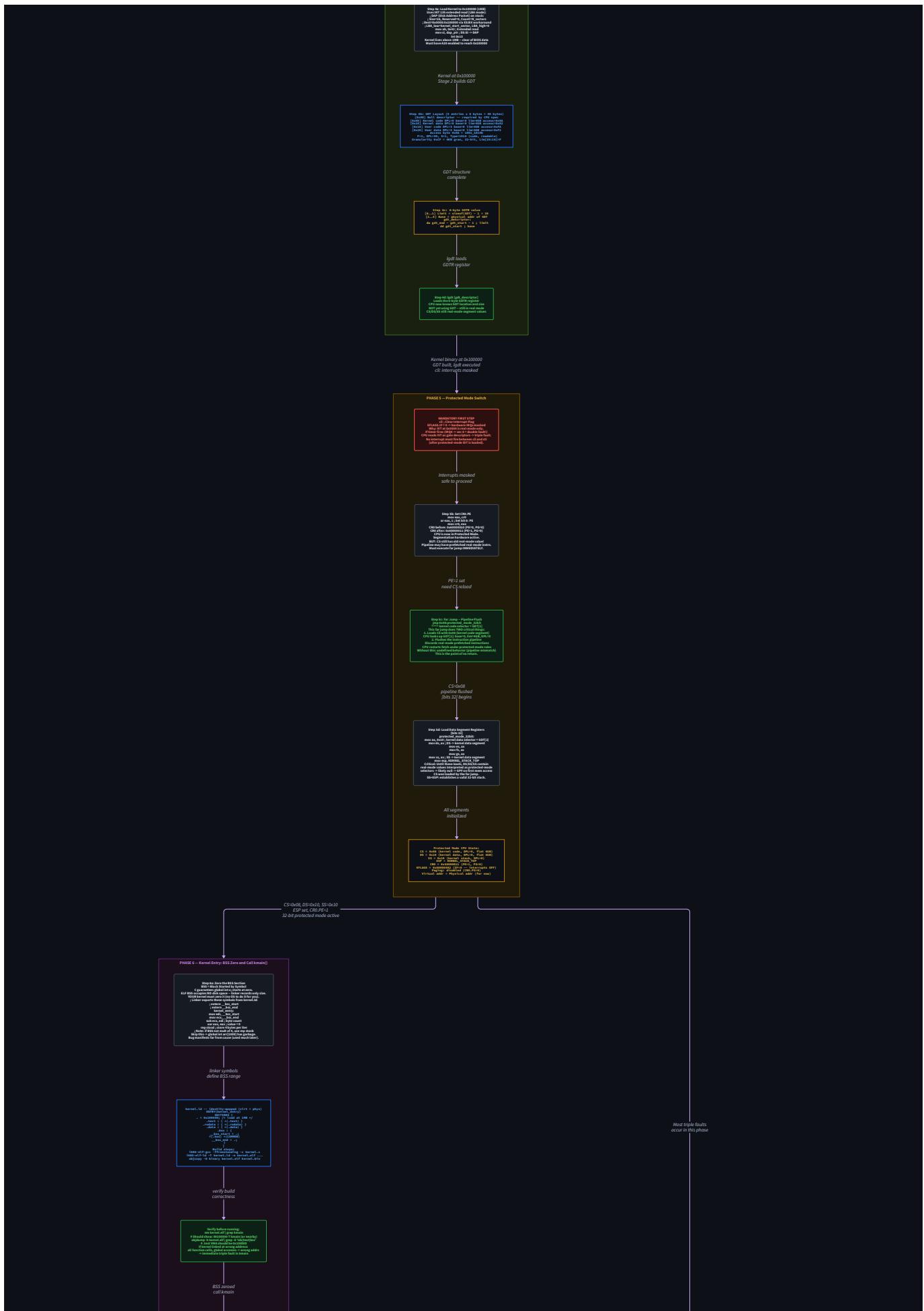
The Revelation: Your CPU Boots Into 1978

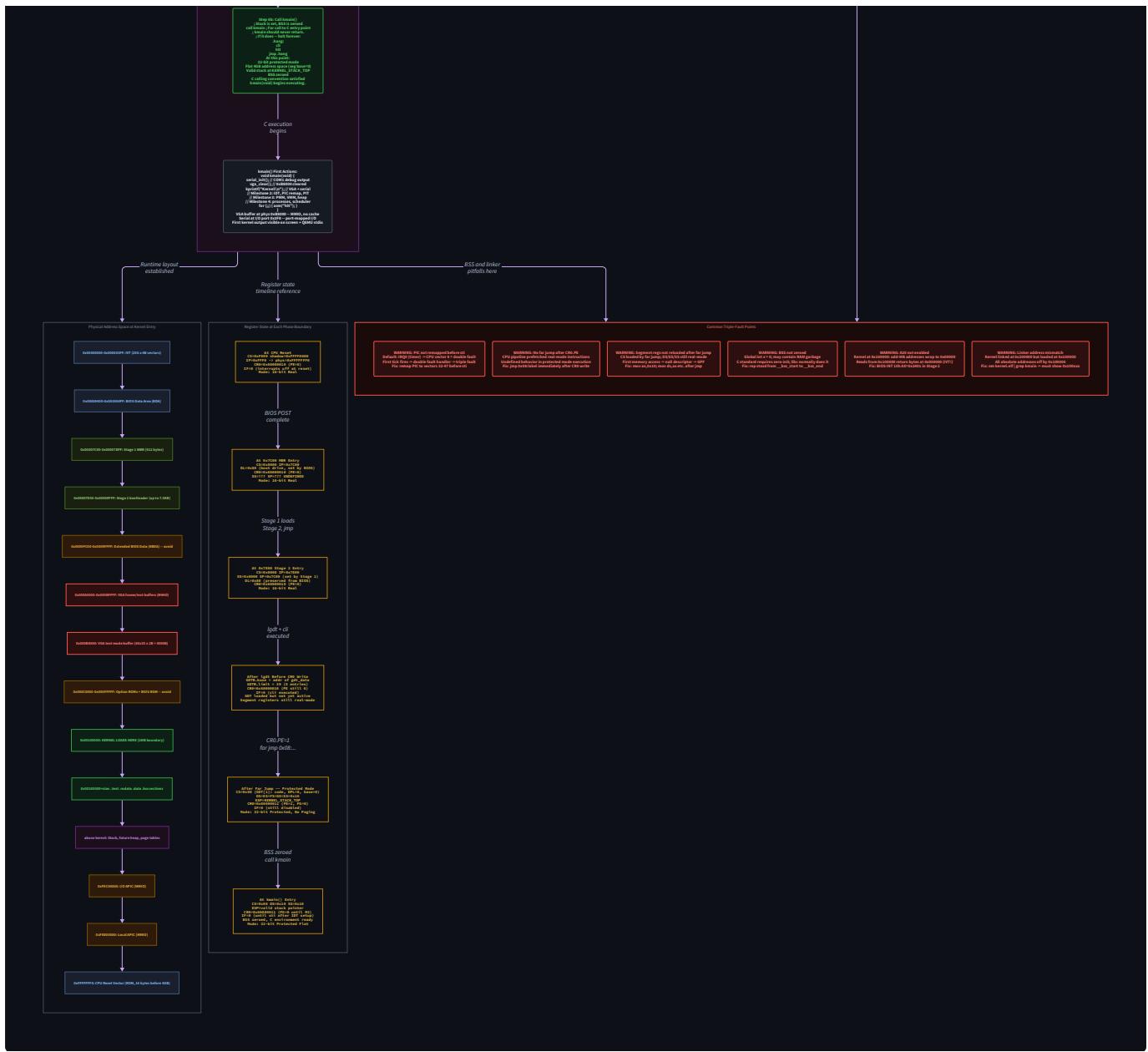
Here is what every developer who has only written userspace code believes about program startup: memory is zeroed, a stack exists, you are running 32-bit or 64-bit code, and the operating system hands your program a clean environment. This belief is correct — for every program that runs *after* an OS exists. You are about to write the program that runs *before* any of that is true.

When you press the power button, the CPU starts executing at physical address `0xFFFFFFF0` — a location in firmware ROM. No OS. No libc. No stack you chose. The processor is in **16-bit real mode**, a compatibility mode that mimics the Intel 8086 from 1978. The entire addressable universe is 1 megabyte. Memory protection does not exist. Any code can overwrite any other code. The interrupt table is a legacy structure called the IVT (Interrupt Vector Table) that assumes real-mode segment arithmetic, and if you enable interrupts while in the middle of switching to protected mode, you will immediately triple-fault with zero diagnostic information.

This is not a historical curiosity. It is the actual hardware state of every x86 machine at power-on, including the server running your cloud VM right now. The firmware runs its POST (Power-On Self-Test), finds a bootable disk, loads the first 512 bytes into memory at physical address `0x7C00`, and jumps there. Those 512 bytes are yours. That is where this milestone begins.







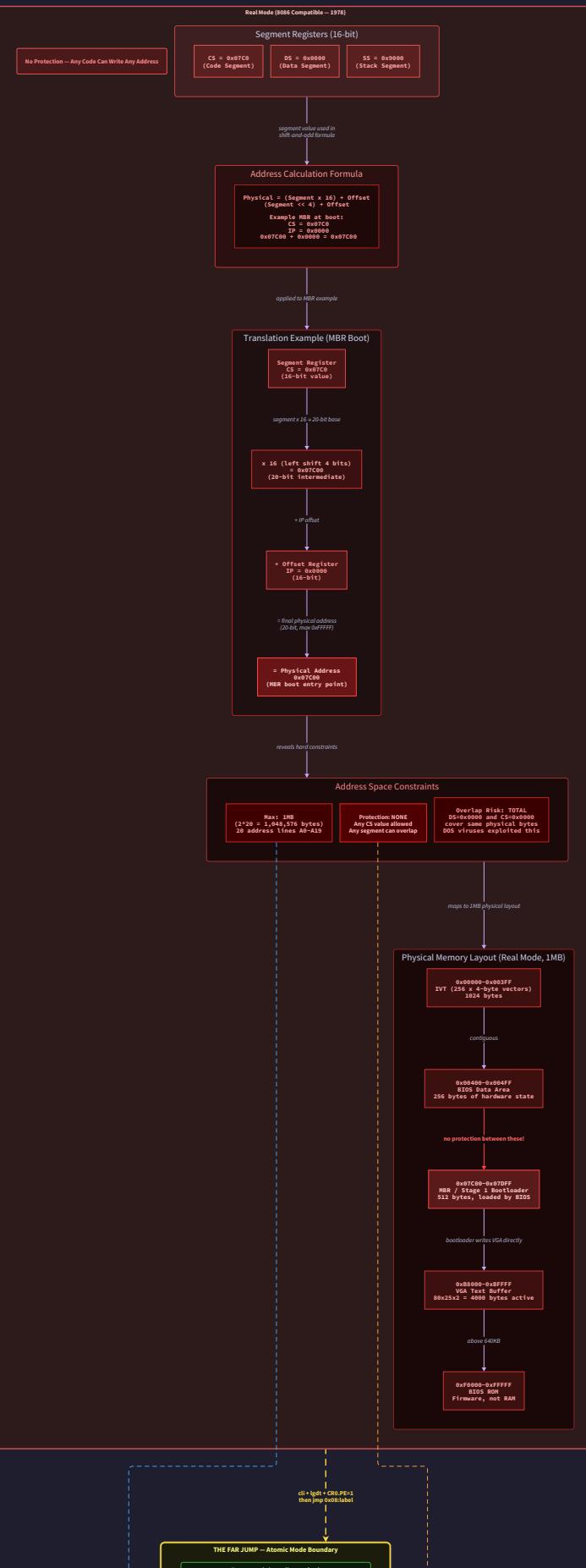
Section 1: Real Mode — The Ground You Stand On

Before you can escape real mode, you need to understand what it actually is and why it exists.

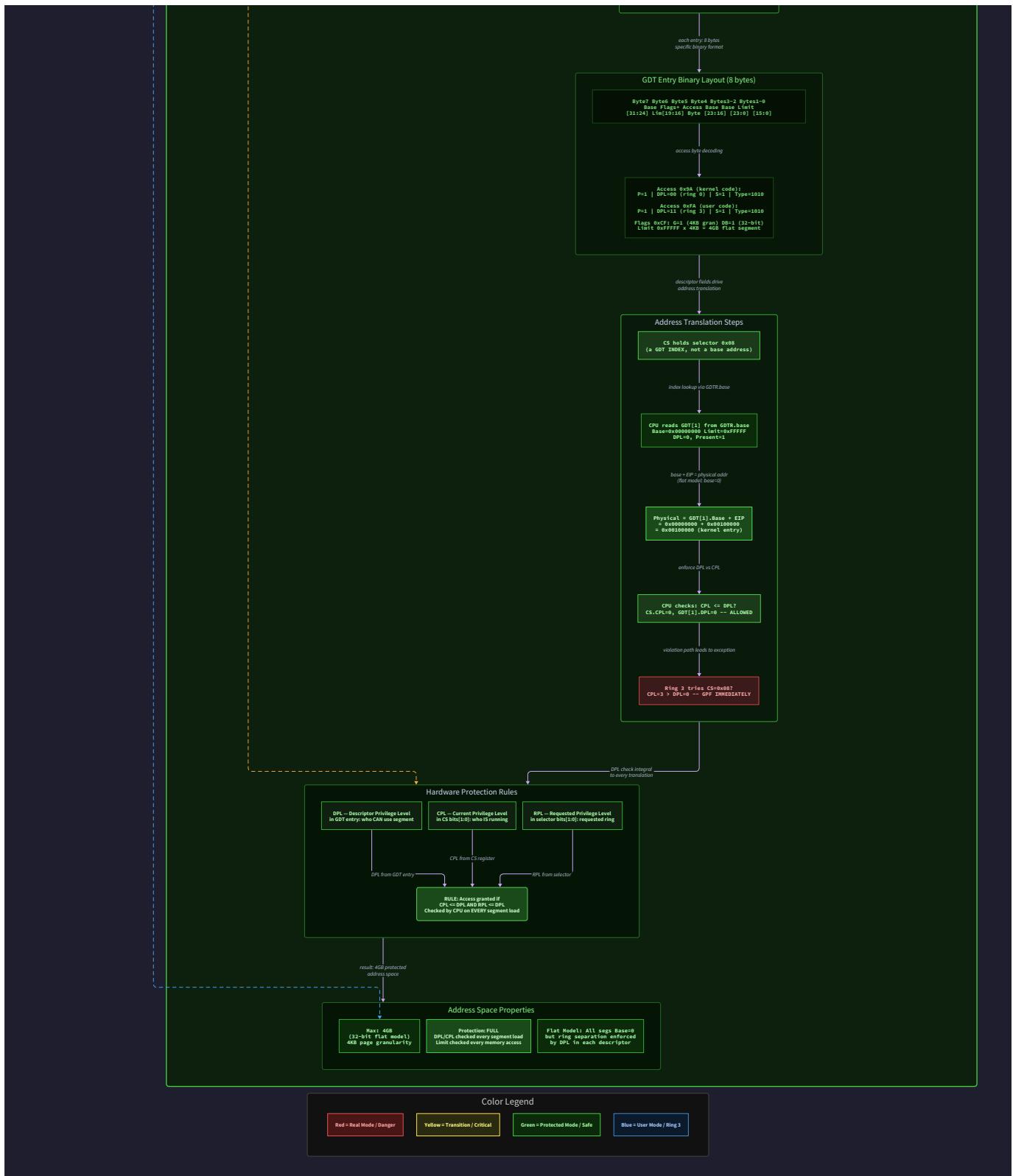
Real mode addressing works like this: the CPU has 16-bit registers but needs to address up to 1MB (20 bits of address space). Intel solved this in 1978 by combining a **segment register** with an **offset register**: $\text{physical address} = \text{segment} \times 16 + \text{offset}$. So if $\text{CS} = 0x07C0$ and $\text{IP} = 0x0000$, the physical address is $0x07C00$ — the MBR load address.

Real Mode vs Protected Mode: Address Translation Comparison

The far jump is the atomic boundary — before: segment:offset; after: GDTR(selector)base+offset







This segmentation model was the original address space mechanism. `CS` (Code Segment), `DS` (Data Segment), and `SS` (Stack Segment) are hardware-enforced boundaries in the sense that all code fetches go through `CS`, all stack operations go through `SS`, and all data accesses go through `DS`. Early DOS programs that corrupted each other's memory were abusing the fact that nothing prevented one segment from overlapping another — the hardware enforced *which register was used, not what value it contained*. Protected mode, as you will see, changes the semantics of these registers entirely.

Hardware Soul: In real mode, every memory access is a segment-register lookup plus an offset. The TLB (Translation Lookaside Buffer — the CPU's cache of address translations) does not exist yet in any meaningful sense. Every address calculation is a simple shift-and-add in hardware, costing approximately zero cycles but providing zero protection.

The BIOS, which loaded your 512-byte MBR at `0x7C00`, leaves you the following state:

- `DL` register contains the boot drive number (you need this to read more sectors from disk)
- Interrupts are enabled (BIOS IVT is active)
- The A20 line (a hardware address line that enables access above 1MB) may or may not be enabled depending on firmware vintage
- You are in 16-bit real mode

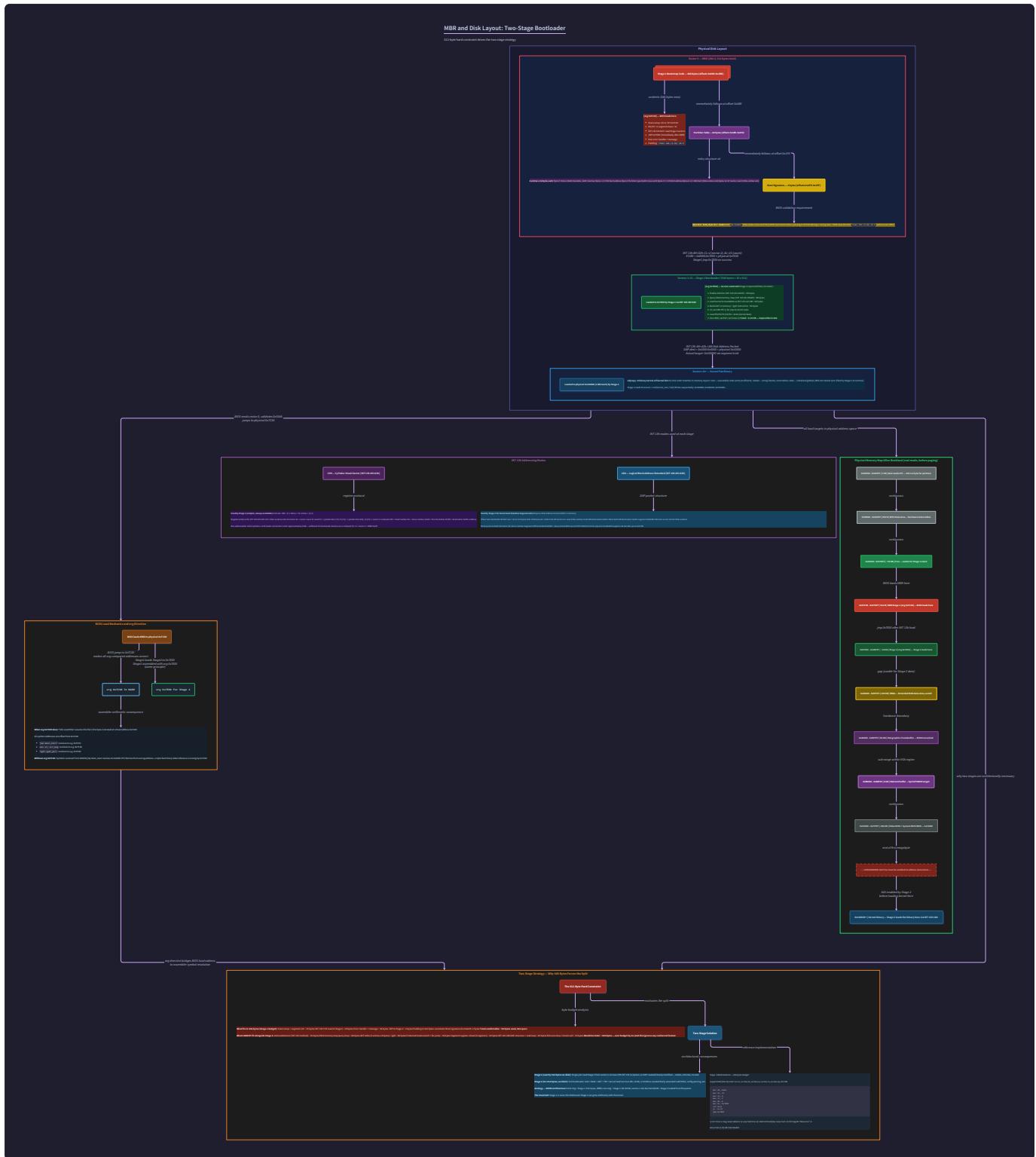
Everything else — the stack pointer, direction of memory layout, which registers contain meaningful values — is undefined. Your bootloader must establish all of it.

Section 2: The MBR Constraint and Two-Stage Bootloaders

512 bytes sounds like a lot until you realize what you need to fit inside it:

- Stack setup
- A20 line enablement code
- Disk read code (BIOS INT 13h)
- GDT descriptor table and its load instruction
- Protected mode switch
- Jump to 32-bit kernel code

This is impossible in a single stage. Real bootloaders solve this with a **two-stage approach**: Stage 1 (512 bytes, in the MBR) does the minimum needed to load Stage 2 from disk. Stage 2 (a few kilobytes, unconstrained) does everything else: enables A20, reads the actual kernel binary, configures the GDT, and enters protected mode.



The 512-byte signature: The BIOS validates your MBR by checking that bytes 510-511 contain `0x55 0xAA` (called the boot signature). If these bytes are missing or wrong, the BIOS refuses to boot from that disk. This is enforced in firmware hardware; there is no workaround. Your linker script or assembler must pad to exactly 512 bytes and place the signature correctly.

```
; End of boot sector (NASM syntax)
times 510 - ($ - $$) db 0      ; Pad to 510 bytes
dw 0xAA55                      ; Boot signature (little-endian: 0x55, 0xAA)
```

NASM

Why `0x55 0xAA` little-endian? When stored as a 16-bit word on a little-endian CPU, `0xAA55` stores byte `0x55` first and byte `0xAA` second in memory. The BIOS checks bytes 510 and 511 for `0x55` and `0xAA` respectively. NASM's `dw` directive uses little-endian storage, so `dw 0xAA55` produces exactly `0x55, 0xAA` in memory. This is the first time byte order matters to you; it will not be the last.

Section 3: Loading the Kernel from Disk — BIOS INT 13h

With Stage 1 running at `0x7C00`, its entire job is to load Stage 2 from disk into memory at a known address and jump to it. The only tool available for disk I/O in real mode is **BIOS INT 13h**, a software interrupt that asks the firmware to perform a disk read on your behalf.

Software interrupt: Unlike hardware interrupts (signals from devices), a software interrupt is triggered by the `int` instruction. It looks up a handler in the IVT (using the interrupt number as an index), saves the CPU state, and jumps to firmware code that talks to the disk controller. After the firmware finishes, it executes `iret` (Interrupt Return) and your code resumes.

The INT 13h function `0x02` reads sectors using **CHS addressing** (Cylinder-Head-Sector — a legacy coordinate system for disk geometry). For simplicity, GRUB and modern bootloaders use the **LBA extension** (INT 13h `0x42`, which uses logical block addresses starting from 0). For a learning OS, CHS is fine since you are always reading sectors near the start of the disk.

```
; BIOS INT 13h: Read sectors (function 0x02)
; Reads 'count' sectors starting at CHS (cylinder=0, head=0, sector=2)
; into ES:BX (segment:offset address in real mode)
mov ah, 0x02          ; Function: read sectors
mov al, 15            ; Number of sectors to read (15 × 512 = 7680 bytes for stage2)
mov ch, 0              ; Cylinder 0
mov cl, 2              ; Sector 2 (sector 1 is the MBR, sectors are 1-indexed in CHS)
mov dh, 0              ; Head 0
; DL already contains the boot drive number from BIOS
mov bx, 0x7E00          ; Load destination: immediately after MBR in memory
int 0x13                ; Call BIOS disk service
jc disk_error          ; Carry flag set = error; halt or retry
```

NASM

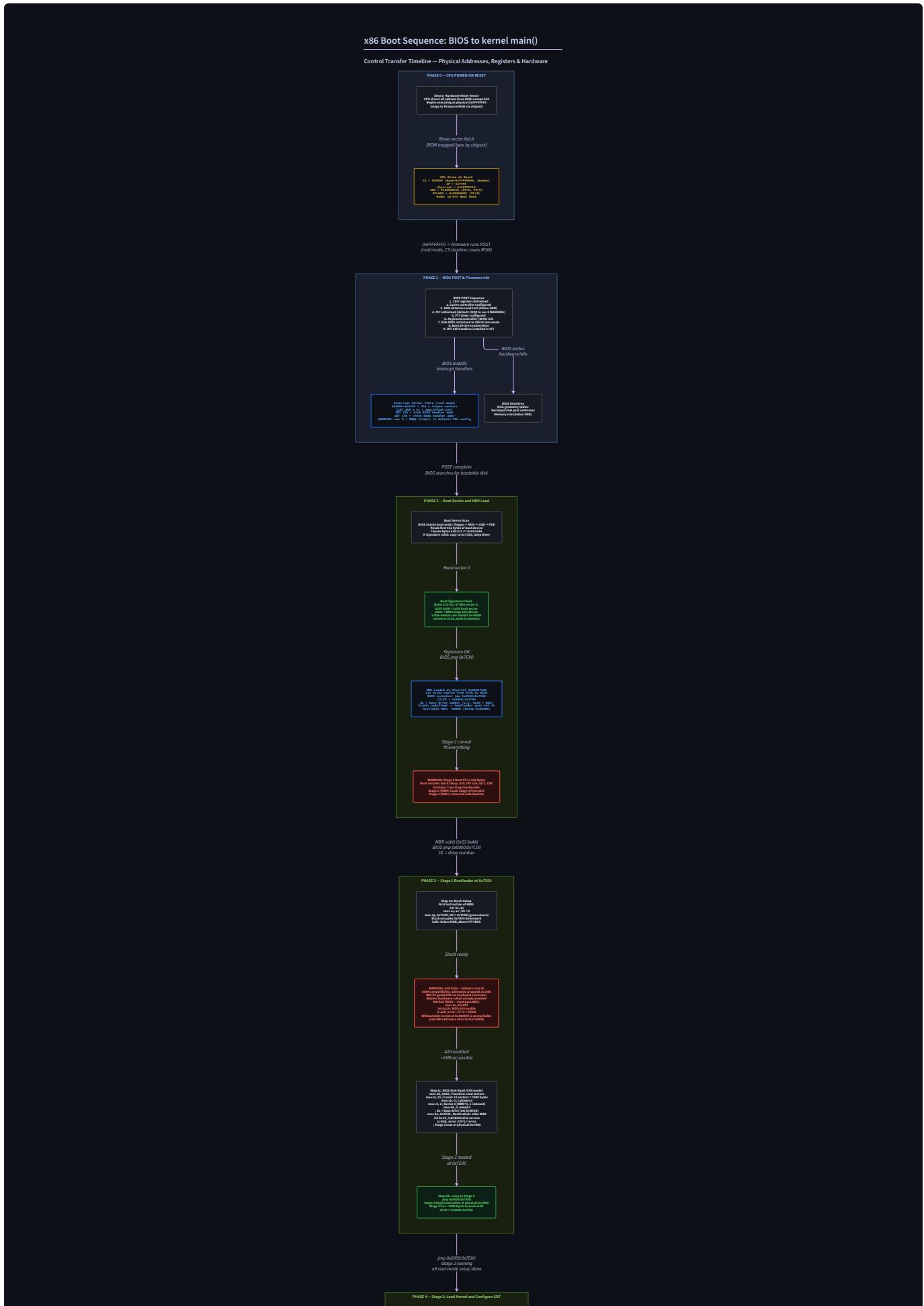
After Stage 1 loads Stage 2, it executes a `jmp 0x7E00` (or wherever Stage 2 was loaded). Stage 2 now has kilobytes of space to work with.

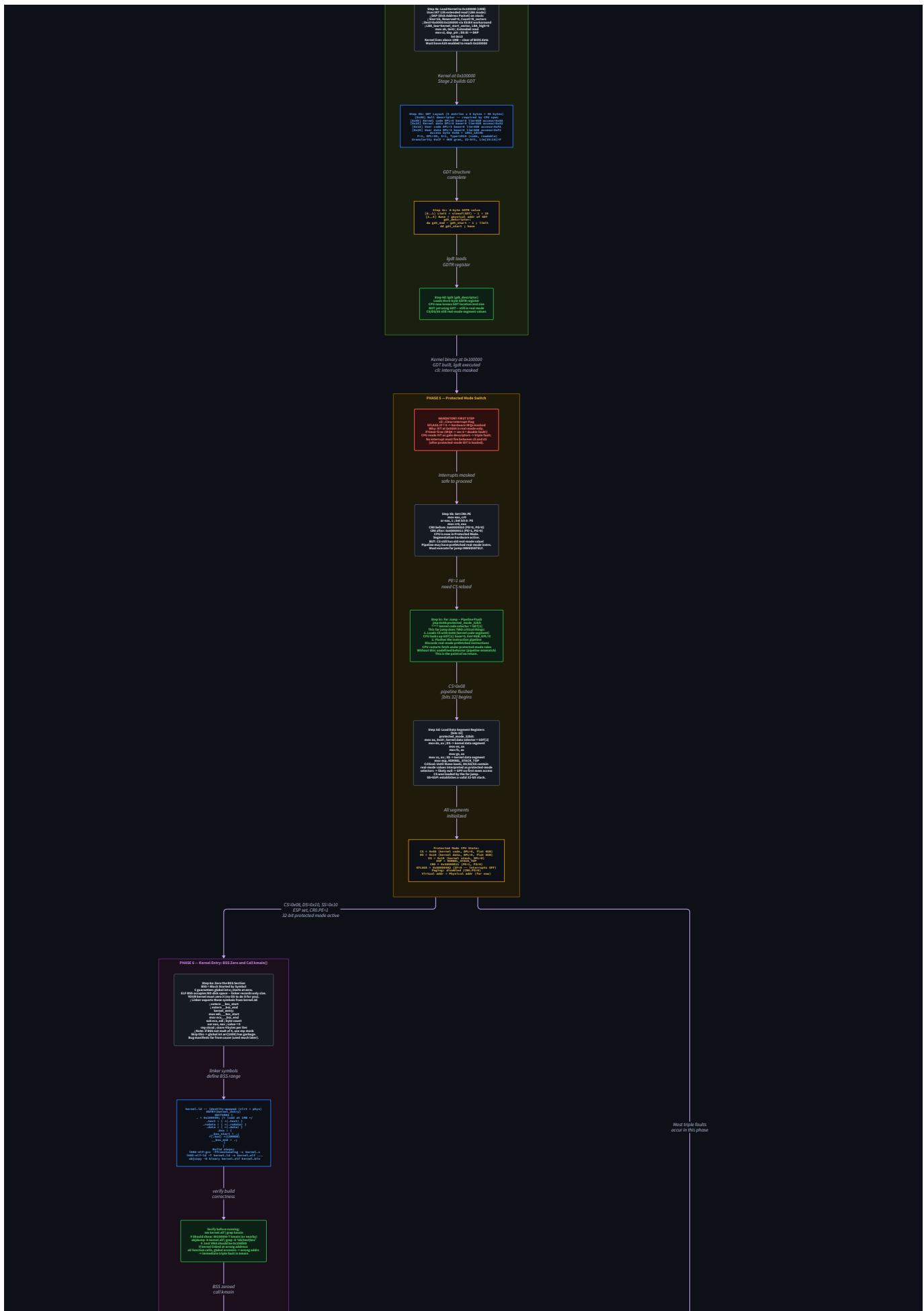
Stage 2's job, in order:

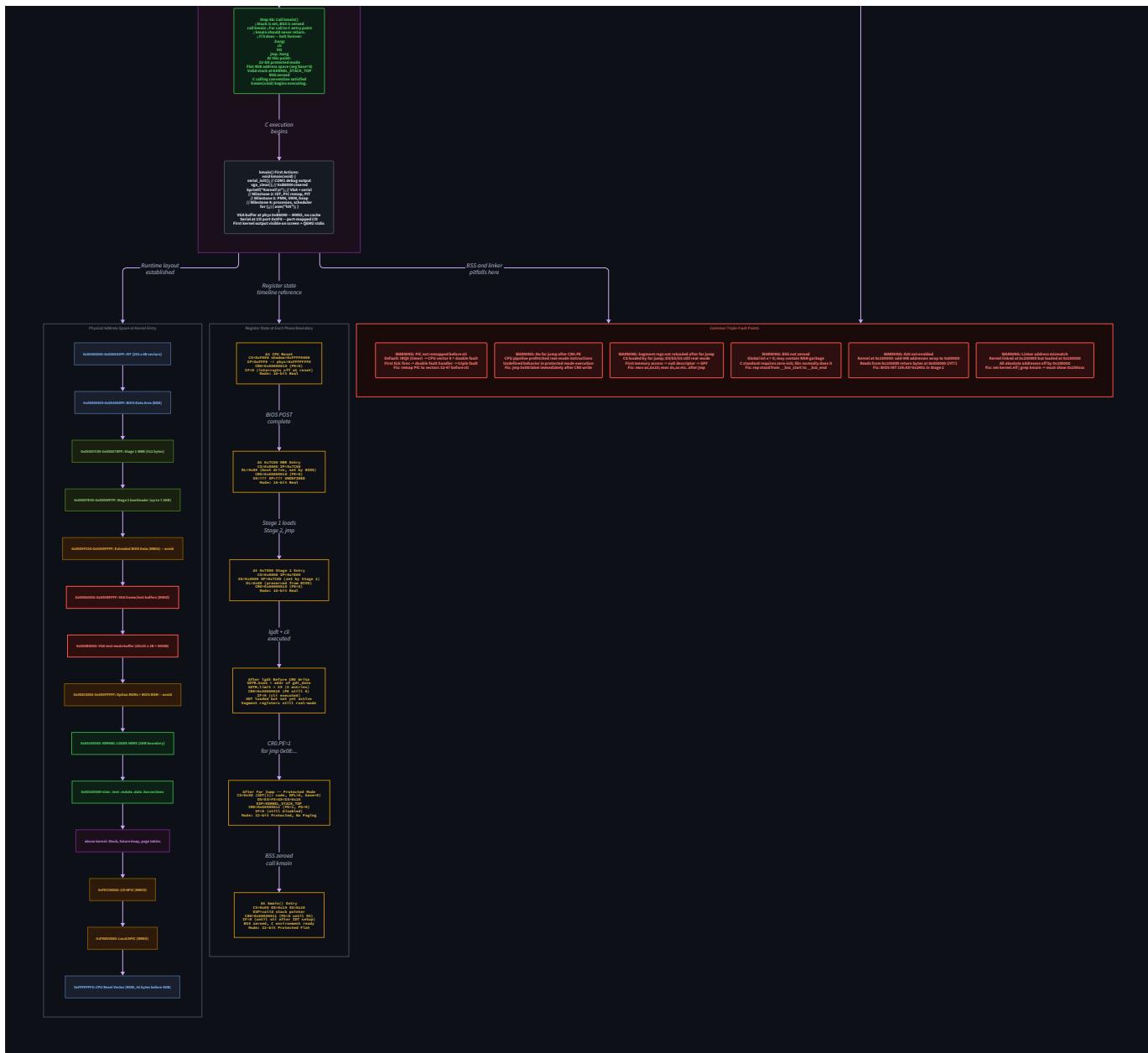
1. Enable the A20 line (allows addressing above 1MB)
2. Load the actual kernel binary from disk into memory at `0x100000` (1MB)
3. Configure and load the GDT
4. Enter protected mode
5. Jump to the 32-bit kernel entry point

The kernel loading step uses the same INT 13h mechanism, but now reading many more sectors (the entire kernel binary). The kernel is loaded to physical address `0x100000` — above the first megabyte — because the memory below 1MB is cluttered with BIOS data structures, video memory, and the bootloader itself.

A20 line: A historical accident. On the original IBM PC, the 8086 had only 20 address lines (A0–A19). Memory wraparound from `0xFFFF + 1` back to `0x00000` was a *feature* that some DOS programs depended on. When the 286 added a 21st address line (A20), IBM added a gate controlled by the keyboard controller to disable it for compatibility. Your bootloader must explicitly enable it. The most portable method is via the BIOS (INT 15h, AX=2401h). On modern hardware/VMs, A20 is often already enabled; on real hardware, skipping this check causes bizarre memory access failures above 1MB.







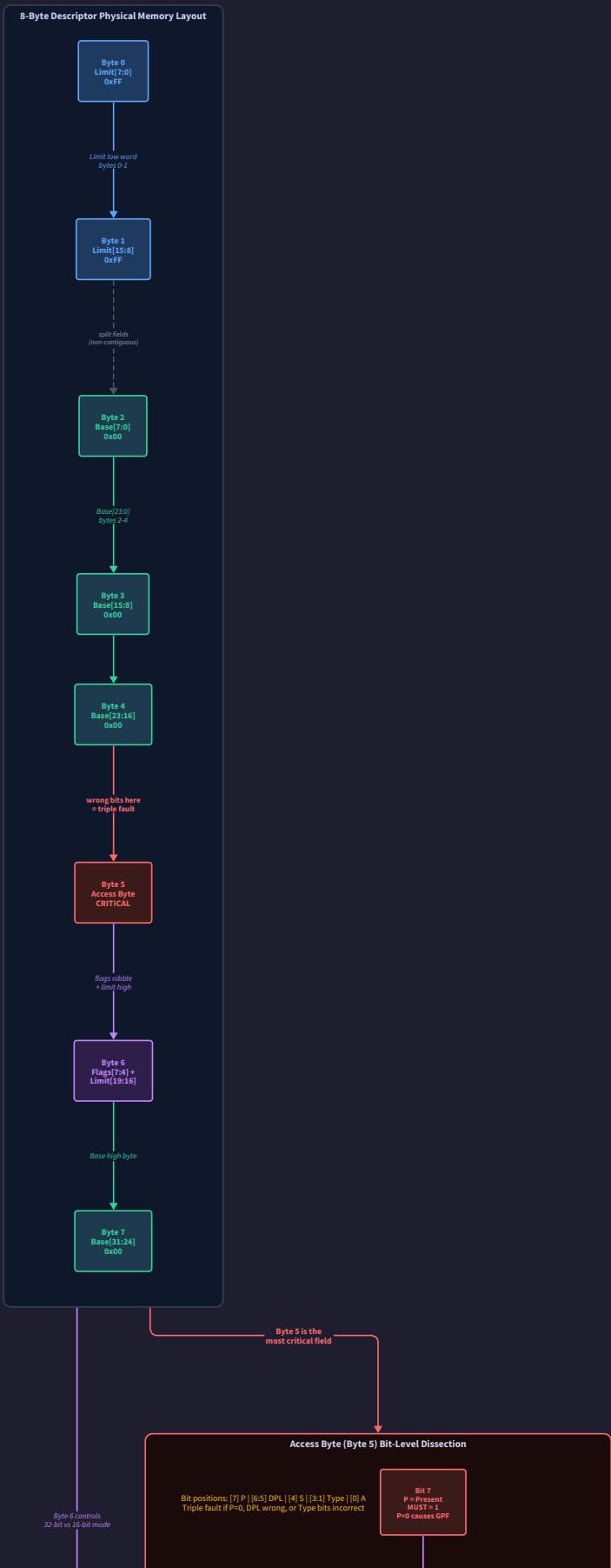
Section 4: The GDT — Hardware You Talk To, Not a Data Structure You Design

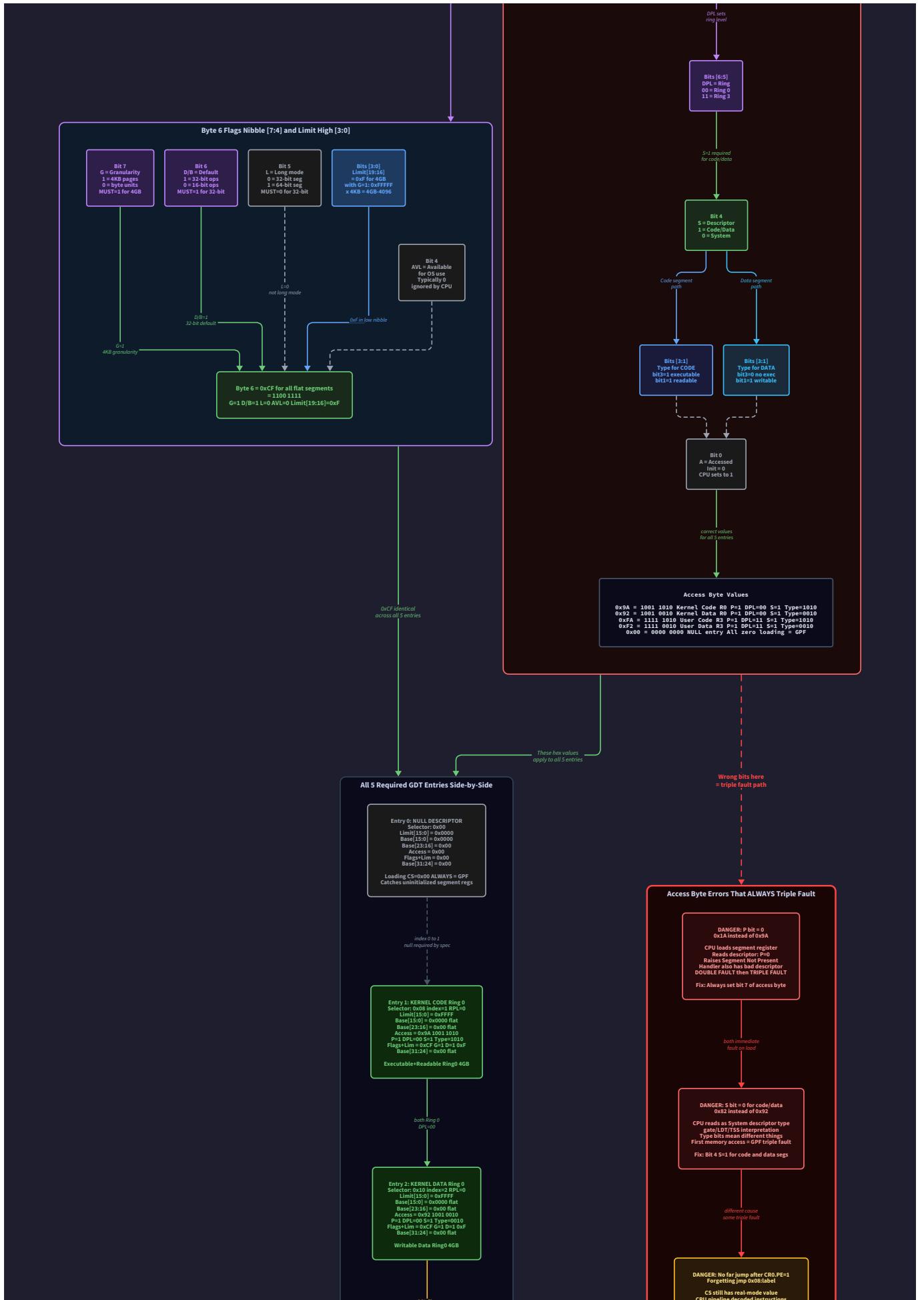
Here is the conceptual shift that trips up every systems programmer coming from userspace: the **GDT (Global Descriptor Table)** is not a software abstraction. It is a hardware register — the `GDTR` — that the CPU reads on every segment register load after protected mode is active. Misconfigure one byte of an access field, and the CPU triple-faults the instant you try to use that segment. There is no error message. There is no stack trace. The machine reboots.

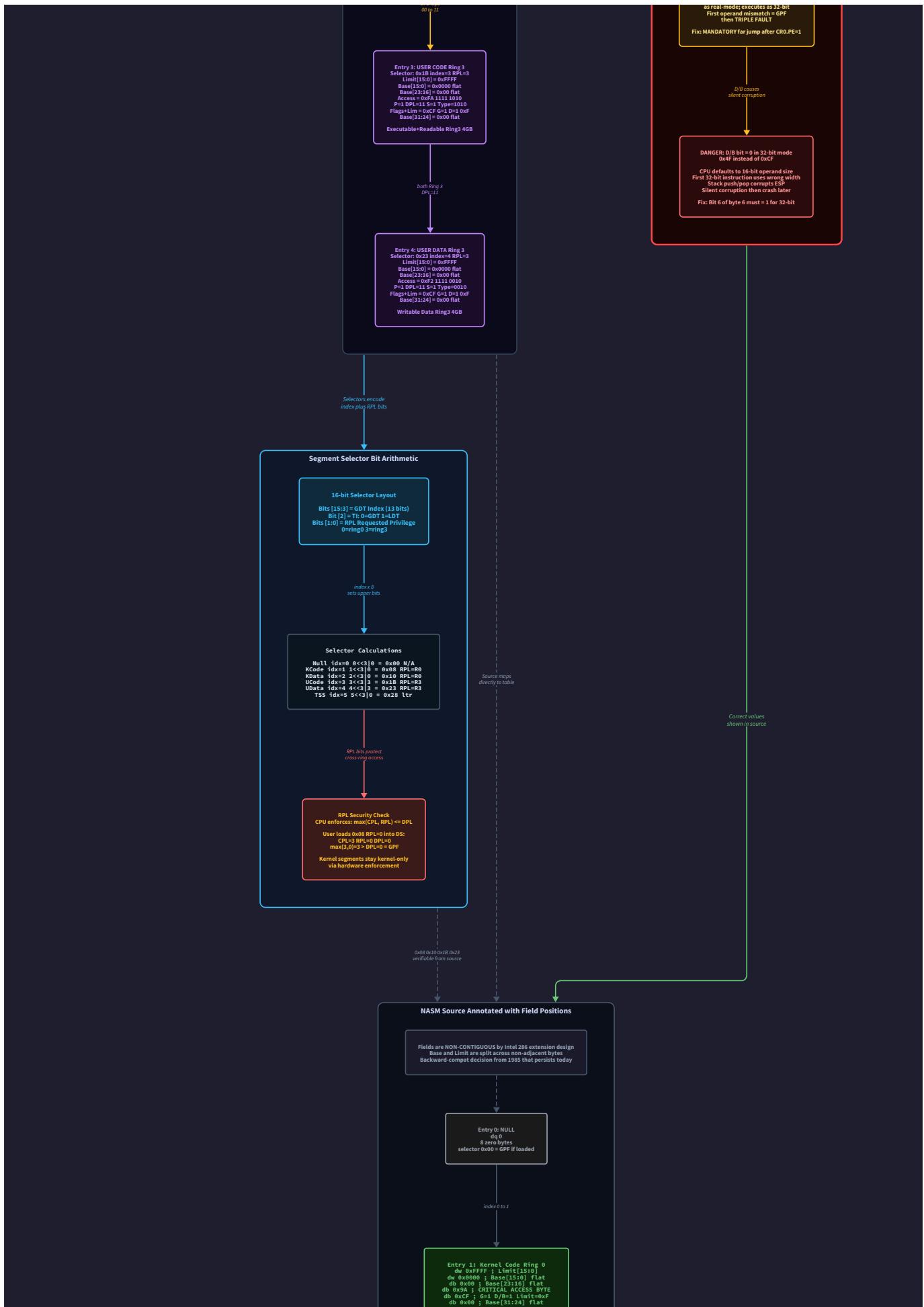
Triple fault: When the CPU encounters a fault (like a General Protection Fault from a bad segment), it tries to invoke the fault handler. If that invocation itself causes a fault (double fault), it tries to invoke the double-fault handler. If *that* causes a fault, the CPU has no more recovery options and performs a hard reset. This is a triple fault. In QEMU, it causes an immediate restart. On real hardware, you see a reboot. Triple faults during protected mode setup are the most common failure mode for new OS developers.

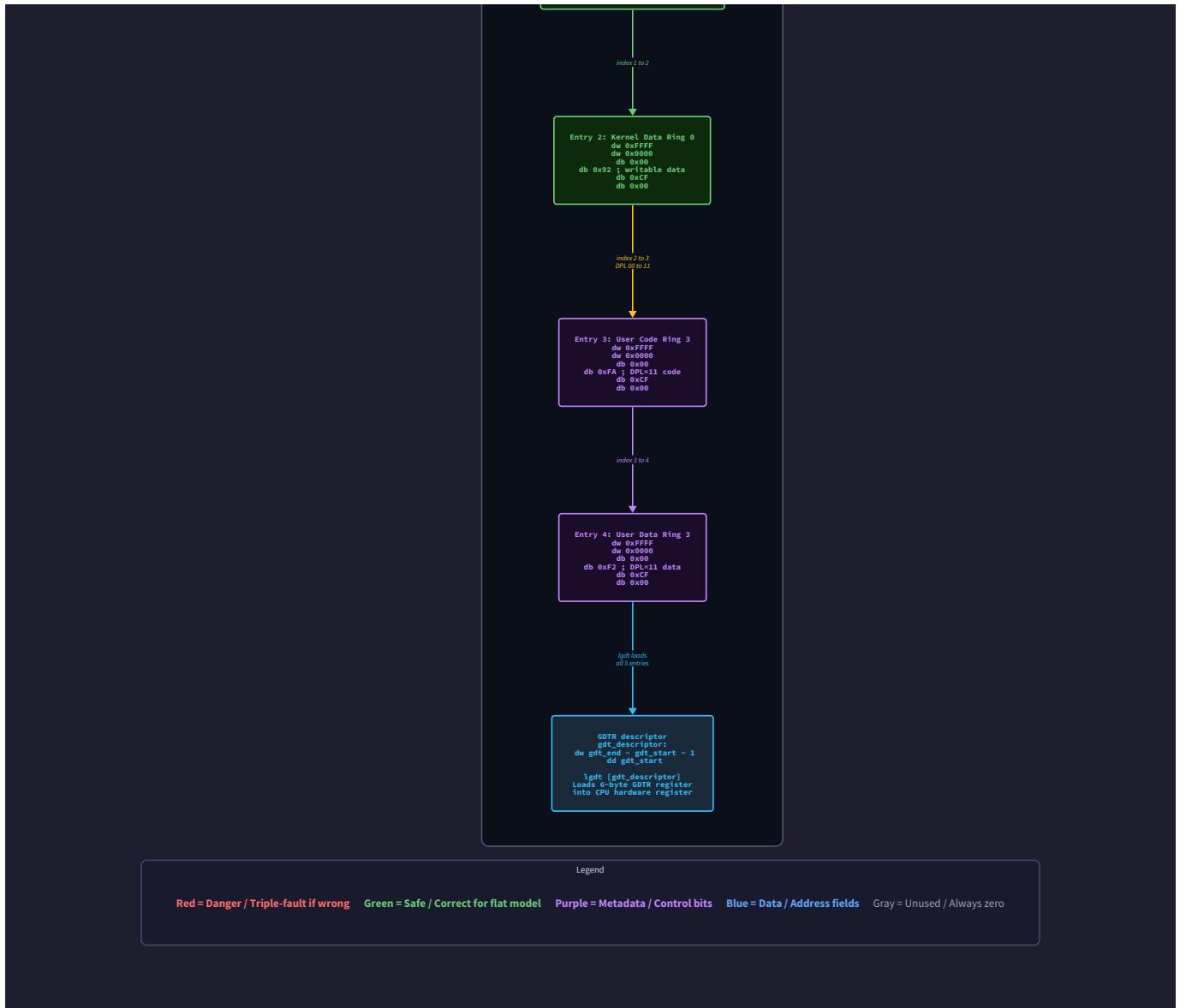
In protected mode, segment registers (`CS`, `DS`, `SS`, etc.) no longer contain base addresses directly. Instead, they contain **segment selectors** — 16-bit values that are indices into the GDT. When you load a value into `CS`, the CPU looks up the corresponding GDT entry and reads the base address, limit, and access rights from it. The hardware enforces those rights on every memory access.

GDT Descriptor Binary Layout — 8-Byte Microscopic









The GDT Descriptor — Byte by Byte

Each GDT entry is exactly 8 bytes. The layout is famously non-contiguous because Intel extended the 286-era descriptor format in the 386 without breaking backward compatibility, splitting the limit and base fields across non-adjacent bytes:

Byte 7	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0
Base [31:24]	Flags+ Limit [19:16]	Access Byte	Base [23:16]	Base [23:0]		Limit [15:0]	

- Limit [19:0]:** Maximum addressable unit. With the 4KB granularity flag set (`G=1`), a limit of `0xFFFF` means $0xFFFF \times 4096 = 4GB - 4096$. That is effectively a 4GB segment — the **flat memory model** that modern 32-bit code assumes.
- Base [31:0]:** Where the segment starts in physical memory. For a flat model, all segments have `base = 0`.
- Access byte:** The most critical field. Contains the DPL (Descriptor Privilege Level — ring 0 or ring 3), the segment type (code or data), and the present bit.

Access Byte (bit 7 to bit 0):

P	DPL	S	Type	A
(1)	(2b)	(1)	(4 bits)	(1)

P = Present (must be 1 for valid descriptor)

DPL = Descriptor Privilege Level (00=ring 0, 11=ring 3)

S = Descriptor type (1=code/data, 0=system)

Type= For code: bit3=executable, bit1=readable

For data: bit3=0, bit1=writable

A = Accessed (CPU sets this; initialize to 0)

Your kernel needs exactly **5 GDT entries** (entries are indexed from 0):

Index	Selector	Description	DPL	Base	Limit	Access
0	0x00	Null descriptor (required by spec)	—	0	0	0x00
1	0x08	Kernel code	0	0	4GB	0x9A
2	0x10	Kernel data	0	0	4GB	0x92
3	0x18	User code	3	0	4GB	0xFA
4	0x20	User data	3	0	4GB	0xF2

Why is a null descriptor required? The CPU specification reserves GDT entry 0 as invalid. Any attempt to load a segment register with selector `0x00` causes a General Protection Fault. This is the hardware's way of catching uninitialized segment register bugs — if you forget to set `DS` before using data memory, the null descriptor immediately signals the error.

Selector arithmetic: A segment selector is a 16-bit value. Bits [15:3] are the GDT index. Bits [2:2] select GDT vs LDT (Local Descriptor Table — not used here). Bits [1:0] are the **RPL** (Requested Privilege Level). Kernel code selector: index 1, GDT, RPL 0 → $(1 \ll 3) | 0 = 0x08$. User code selector: index 3, GDT, RPL 3 → $(3 \ll 3) | 3 = 0x1B$. Note the user selectors have RPL=3 in the low bits, distinguishing them from kernel selectors at the hardware level.

```

; GDT in NASM syntax
gdt_start:
    ; Entry 0: Null descriptor (required)
    dq 0

    ; Entry 1: Kernel code (ring 0)
    ; Base=0, Limit=0xFFFF, 4KB granularity, 32-bit, executable+readable
    dw 0xFFFF      ; Limit [15:0]
    dw 0x0000      ; Base [15:0]
    db 0x00        ; Base [23:16]
    db 0x9A        ; Access: present, ring 0, code, executable, readable
    db 0xCF        ; Flags (4KB gran, 32-bit) + Limit [19:16] = 0xF
    db 0x00        ; Base [31:24]

    ; Entry 2: Kernel data (ring 0)
    dw 0xFFFF
    dw 0x0000
    db 0x00
    db 0x92        ; Access: present, ring 0, data, writable
    db 0xCF
    db 0x00

    ; Entry 3: User code (ring 3)
    dw 0xFFFF
    dw 0x0000
    db 0x00
    db 0xFA        ; Access: present, ring 3, code, executable, readable
    db 0xCF
    db 0x00

    ; Entry 4: User data (ring 3)
    dw 0xFFFF
    dw 0x0000
    db 0x00
    db 0xF2        ; Access: present, ring 3, data, writable
    db 0xCF
    db 0x00
gdt_end:

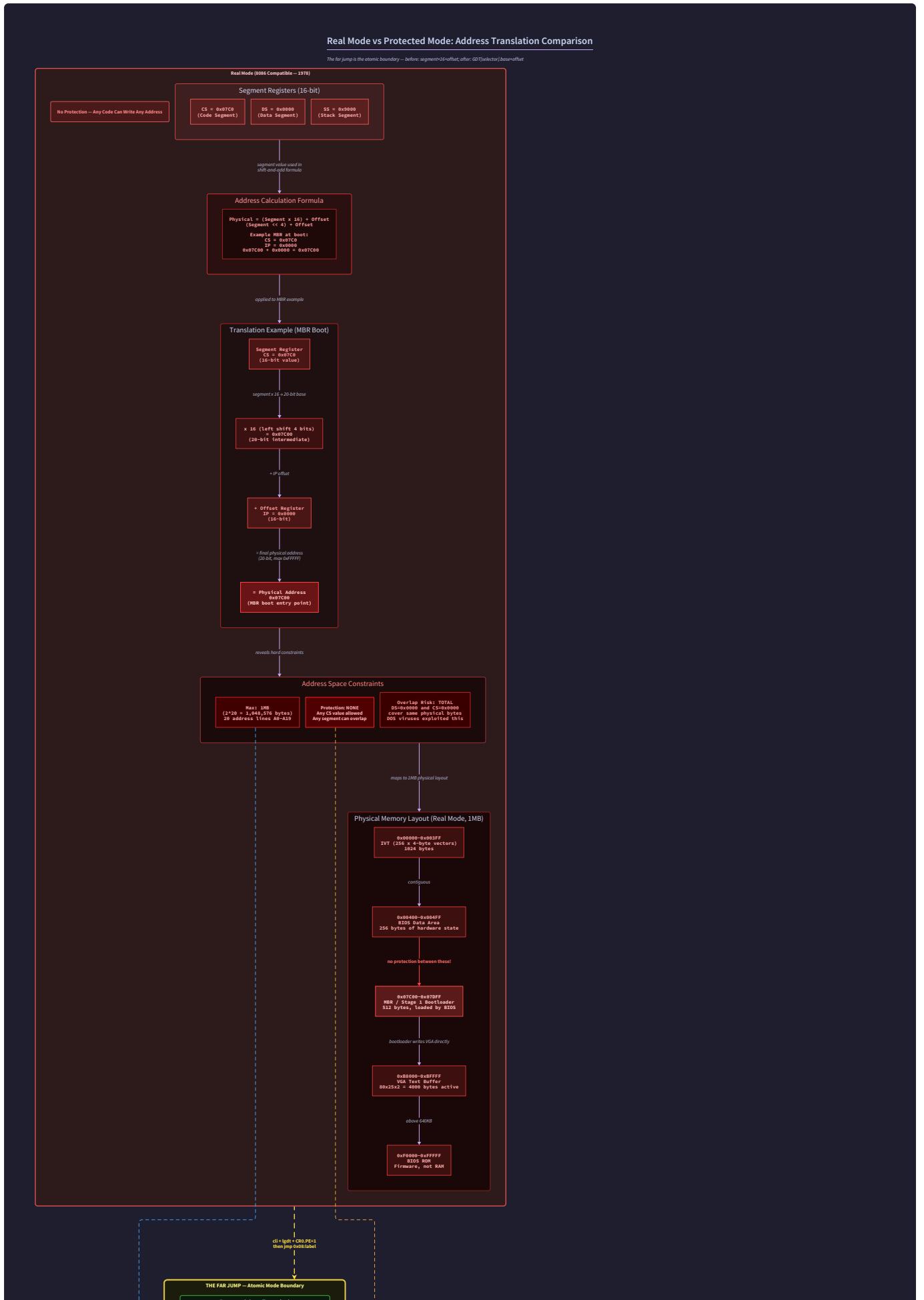
; GDTR – the value loaded into the hardware GDTR register
gdt_descriptor:
    dw gdt_end - gdt_start - 1    ; Limit: size in bytes minus 1
    dd gdt_start                 ; Base: physical address of GDT

```

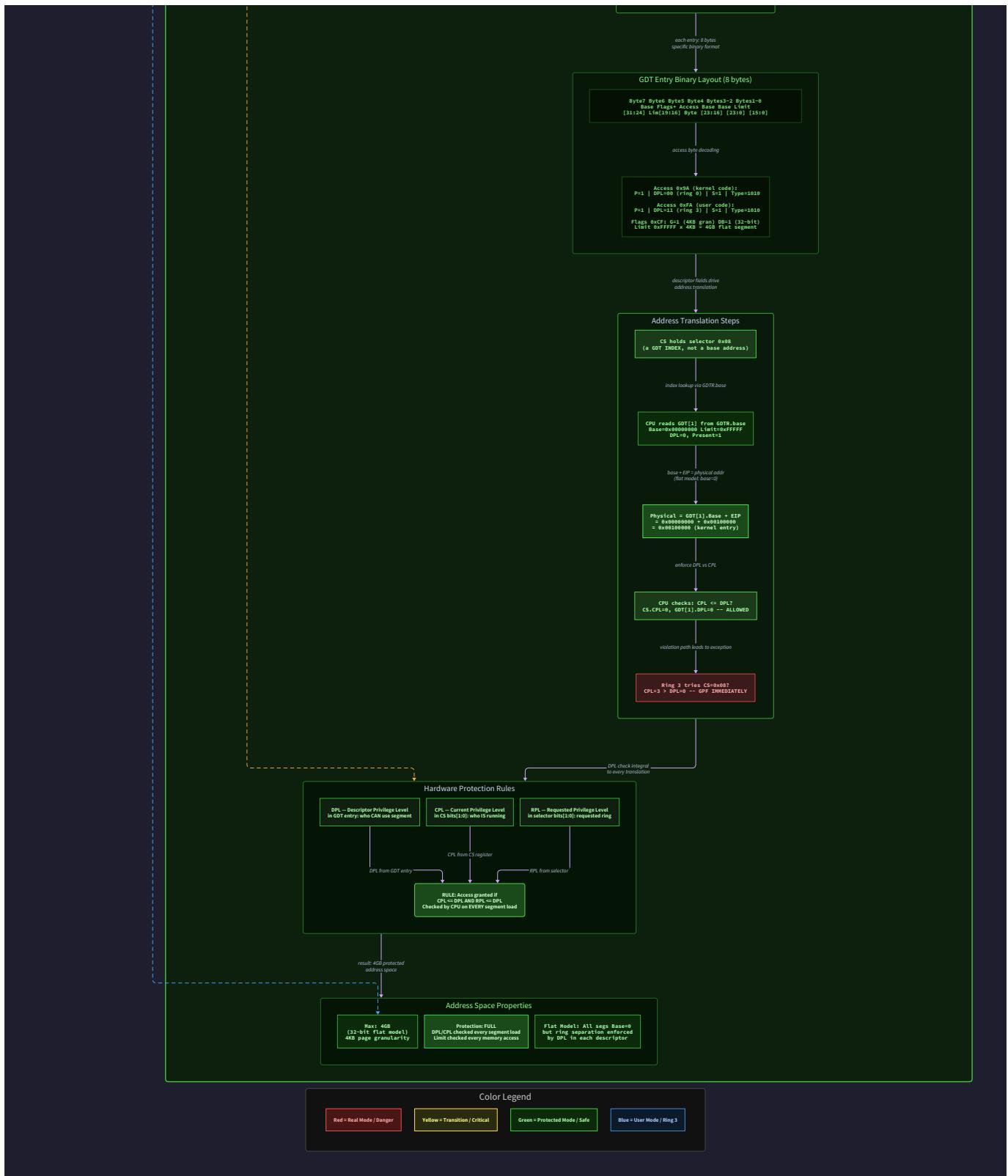
The access byte values — `0x9A`, `0x92`, `0xFA`, `0xF2` — are the most error-prone part. One wrong bit (forgetting the present bit `P=1`, or setting the wrong DPL) causes an immediate triple fault. Build a function or macro that constructs descriptors from named fields, not magic numbers, once you are writing the C-level kernel.

Section 5: The Protected Mode Transition — Four Precise Steps

The transition from real mode to protected mode is a choreographed sequence. The CPU does not know you are about to switch; it is simply executing instructions. You must ensure that the transition is instantaneous from the CPU's perspective — no interrupt can fire in the middle of it.







Step 1: Disable interrupts

```
cli      ; Clear Interrupt Flag – hardware interrupts are now masked
```

NASM

This is non-negotiable. The BIOS IVT (Interrupt Vector Table) is a real-mode structure at physical address `0x0000`. If a hardware interrupt fires after you have loaded the GDTR but before you have set up an IDT (Interrupt Descriptor Table) for protected mode, the CPU will try to handle it using the IVT — which contains real-mode addresses that are meaningless as protected-mode gate descriptors. The result is an immediate triple fault.

Step 2: Load the GDTR

```
lgdt [gdt_descriptor] ; Load the GDT register with base address and size
```

NASM

`lgdt` loads the 6-byte GDTR register: a 16-bit limit (size - 1) and a 32-bit base address pointing to your GDT. After this instruction, the CPU knows where your GDT lives, but it is not yet *using* it — you are still in real mode.

Step 3: Set CR0.PE (Protection Enable)

```
mov eax, cr0  
or eax, 1          ; Set bit 0: Protection Enable  
mov cr0, eax
```

NASM

`CR0` is a **control register** — a special-purpose CPU register that controls fundamental processor behavior. Setting bit 0 (`PE = Protection Enable`) switches the CPU from real mode to protected mode. This change is immediate and atomic. The CPU is now in protected mode, but `CS` still contains the old real-mode value. This is a brief inconsistent state that must be resolved immediately.

Step 4: Far jump to flush the pipeline

```
jmp 0x08:protected_mode_entry ; Far jump: segment selector 0x08 (kernel code), offset = label
```

NASM

A **far jump** (`jmp segment:offset`) explicitly loads a new value into `CS`. This does two critical things simultaneously:

1. It loads `CS` with `0x08` — the kernel code segment selector — causing the CPU to look up GDT entry 1 and use its base/limit flags for all subsequent code fetches.
2. It **flushes the instruction pipeline**. Modern CPUs prefetch and decode instructions ahead of the current instruction pointer. Some of those prefetched instructions were decoded under real-mode assumptions. The far jump forces the pipeline to discard all prefetched instructions and restart fetching from the new address under protected-mode rules.

Without this far jump, the pipeline might execute a few real-mode-decoded instructions in protected mode — undefined behavior that manifests as random corruption.

After the far jump — loading data segment registers:

```
protected_mode_entry:  
    mov ax, 0x10      ; Kernel data segment selector (GDT entry 2)  
    mov ds, ax  
    mov es, ax  
    mov fs, ax  
    mov gs, ax  
    mov ss, ax  
    mov esp, KERNEL_STACK_TOP ; Set up a stack
```

NASM

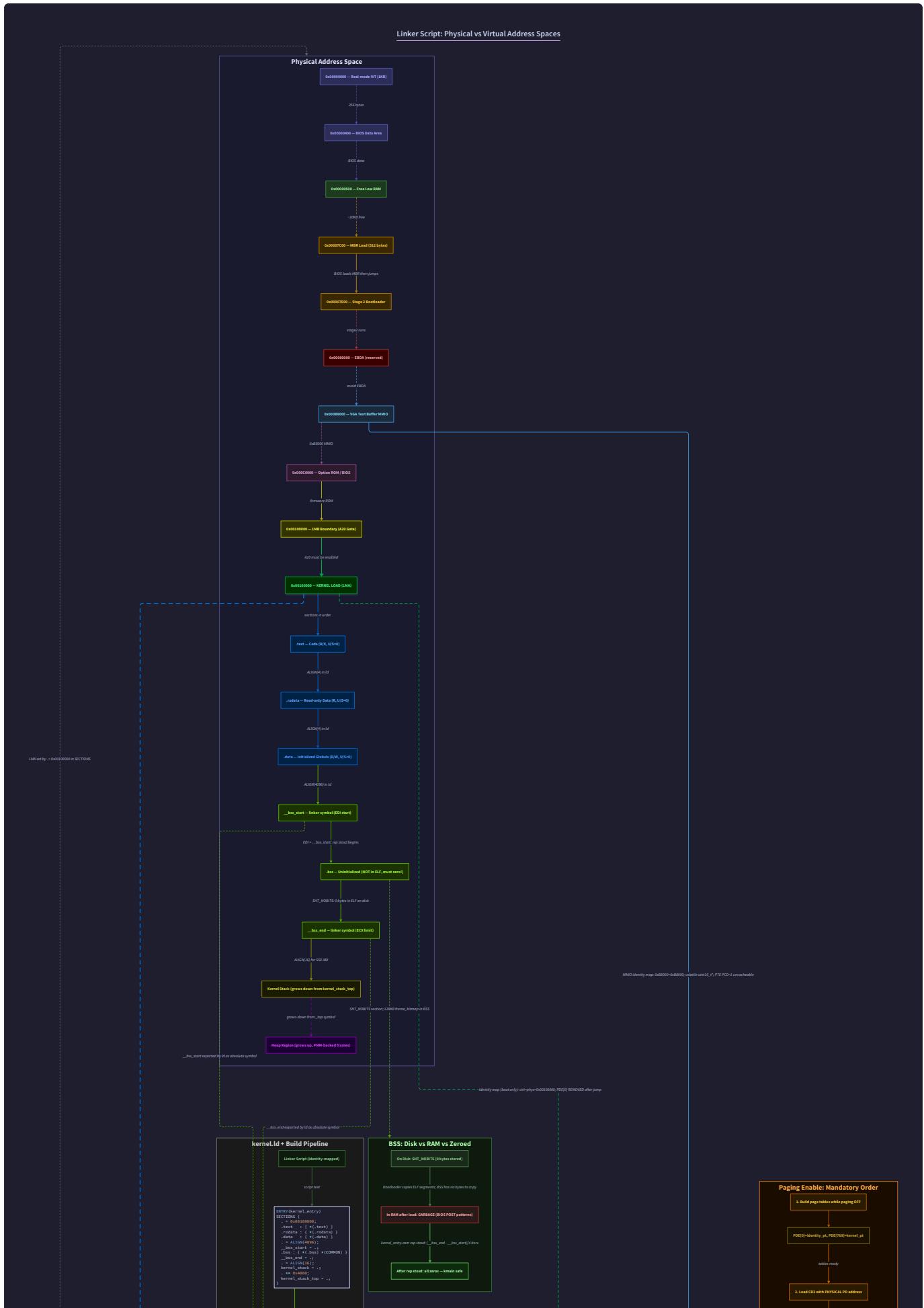
`CS` is loaded by the far jump. Every other segment register (`DS`, `ES`, `FS`, `GS`, `SS`) must be explicitly loaded with the kernel data selector. Until you do this, they still contain real-mode segment values that the CPU will now misinterpret as protected-mode selectors — likely pointing to the null descriptor or garbage, causing a GPF on the first memory access.

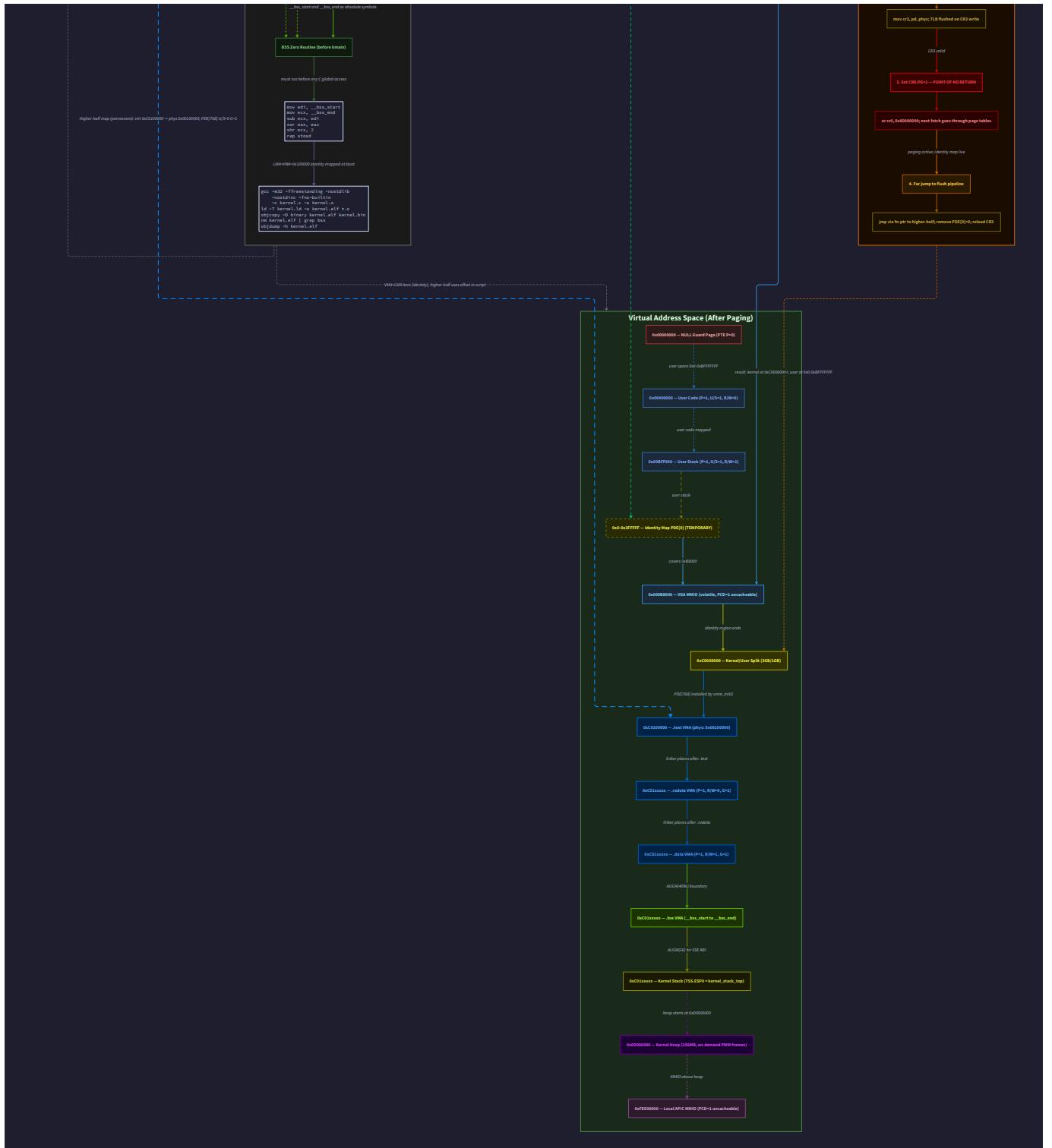
Hardware Soul: The far jump costs approximately 20-30 cycles on modern CPUs due to pipeline flush and branch target prediction miss — but this happens exactly once at boot. The TLB is irrelevant here (paging is not enabled yet). The critical timing constraint is that `cli` → `lgdt` → `CR0.PE` → `jmp` must be a contiguous sequence with no interrupts possible between them.

Section 6: The Linker Script — Virtual vs Physical Addresses

You are about to confront one of the most fundamental concepts in systems programming: **the kernel's linked virtual address and its physical load address are different things**.

The bootloader loaded your kernel binary to physical address `0x100000` (1MB). But your C code, compiled to run at a different address (say, `0x100000` for now, or `0xC0100000` for a higher-half kernel), has all its symbol addresses embedded by the linker. If the linker placed `main` at virtual address `0xC0100000` but the CPU is fetching instructions from physical address `0x100000`, every function call, every global variable access, every constant's address is wrong.





For this milestone, we use the **identity-mapped** approach: the kernel is linked *and* loaded at `0x1000000`. Virtual address = physical address. This sidesteps the complexity of higher-half mapping (which requires paging, covered in Milestone 3). Your linker script:

```
/* kernel.ld - simple identity-mapped kernel */
ENTRY(kernel_entry)      /* Symbol name of the first function to call */

SECTIONS {
    . = 0x100000;          /* Start placing sections at 1MB physical/virtual */

    .text : {
        *(.text)        /* All code from all object files */
    }

    .rodata : {
        *(.rodata)       /* Read-only data: string literals, const arrays */
    }

    .data : {
        *(.data)         /* Initialized globals */
    }

    .bss : {
        __bss_start = .;   /* Symbol marking start of BSS */
        *(.bss)           /* Uninitialized globals */
        *(COMMON)          /* Tentative definitions (C commons) */
        __bss_end = .;     /* Symbol marking end of BSS */
    }
}
```

LD

What is BSS? The `.bss` section (Block Started by Symbol — a name from 1950s assembly) contains uninitialized global variables. In an ELF binary, BSS consumes no space on disk — the file just records "there are N bytes of zeros starting here." Your libc normally zeros BSS before `main()` runs. In a freestanding kernel, *no one* zeros BSS — that is your job. If you skip this, global variables that the C standard guarantees are zero start as whatever bytes happened to be in RAM.

The kernel C entry point, written in assembly (to control the stack before calling C):

```
; kernel_entry.asm - first 32-bit code after protected mode switch          NASM
[bits 32]
[extern kmain]
[extern __bss_start]
[extern __bss_end]

kernel_entry:
    ; Zero the BSS section
    mov edi, __bss_start
    mov ecx, __bss_end
    sub ecx, edi          ; ECX = number of bytes to zero
    xor eax, eax          ; EAX = 0
    rep stosd            ; Store EAX to [EDI], increment EDI, decrement ECX; repeat

    ; Set up the stack (grows downward, so top of reserved region)
    mov esp, kernel_stack_top

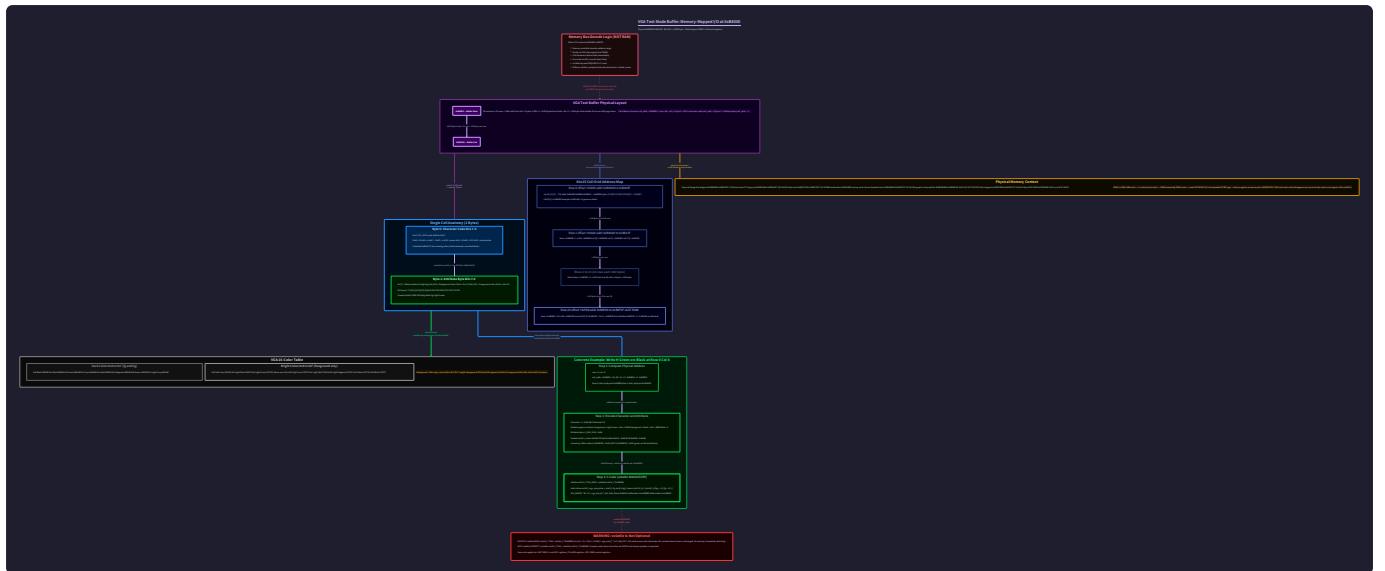
    ; Call the C kernel main
    call kmain

    ; If kmain ever returns, halt the CPU
    cli
    hlt
```

`rep stosd` is an x86 string instruction: it stores the value in `EAX` to the memory address in `EDI`, increments `EDI` by 4, decrements `ECX` by 1, and repeats until `ECX` reaches 0. This zeros BSS four bytes at a time. Note: `sub ecx, edi` gives the byte count; `rep stosd` decrements by 1 per iteration but advances by 4 bytes — if BSS size is not a multiple of 4, use `rep stosb` instead.

Section 7: The VGA Text Mode Driver — Memory-Mapped I/O

At physical address `0xB8000`, the hardware places a 2D array of character cells. Writing bytes there makes them appear on screen immediately — no syscall, no driver stack, no DMA. This is **memory-mapped I/O (MMIO)**: the CPU's memory bus is wired so that certain physical address ranges do not connect to RAM but to device registers.



The VGA text buffer is 80 columns × 25 rows = 2000 cells. Each cell is **2 bytes**: the ASCII character code and an attribute byte controlling color.

Cell layout (2 bytes):

Character (byte 0)	Attribute (byte 1)
ASCII code 0-255	Background Fore
	[7:4] [3:0]

Attribute byte:

Bits [3:0] = Foreground color (0-15)
 Bits [6:4] = Background color (0-7)
 Bit [7] = Blink enable (or bright background)

Color values:

0=Black, 1=Blue, 2=Green, 3=Cyan, 4=Red, 5=Magenta, 6=Brown, 7=Light Gray
 8=Dark Gray, 9=Light Blue, A=Light Green, B=Light Cyan, C=Light Red, D=Light Magenta, E=Yellow, F=White

```

/* vga.c - minimal VGA text mode driver */

#include <stdint.h>

#define VGA_BASE ((volatile uint16_t *)0xB8000)

#define VGA_COLS 80
#define VGA_ROWS 25

static int cursor_row = 0;
static int cursor_col = 0;

/* Encode character + attribute into a 16-bit VGA cell */
static inline uint16_t vga_entry(char c, uint8_t fg, uint8_t bg) {
    return (uint16_t)c | ((uint16_t)((bg << 4) | fg) << 8);
}

void vga_putchar(char c, uint8_t fg, uint8_t bg) {
    if (c == '\n') {
        cursor_col = 0;
        cursor_row++;
    } else {
        VGA_BASE[cursor_row * VGA_COLS + cursor_col] = vga_entry(c, fg, bg);
        cursor_col++;
        if (cursor_col >= VGA_COLS) {
            cursor_col = 0;
            cursor_row++;
        }
    }
}

/* TODO: scroll when cursor_row >= VGA_ROWS */
}

```

The **volatile** keyword is critical here. Without **volatile**, the C compiler may observe that you are writing to a memory location but never reading it back, and optimize the writes away as "dead stores." **volatile** tells the compiler: "this memory access has observable side effects that you cannot see — do not optimize it." Every MMIO access must be through **volatile** pointers.

Hardware Soul: Writing to `0xB8000` does not cost a cache miss in the L1/L2/L3 sense — MMIO regions are typically mapped as uncacheable by the memory type range registers (MTRRs) in the CPU. Every write goes directly to the device. This makes MMIO writes slower than RAM writes (hundreds of nanoseconds vs tens), but it guarantees the device sees the update immediately. For a terminal driver, this is irrelevant; for high-bandwidth MMIO (like GPU framebuffers), it matters enormously.

Section 8: The Serial Port Debug Driver — A Lifeline

VGA output is visible on the screen, but it cannot be logged, captured, or piped to a file. The **serial port (UART — Universal Asynchronous Receiver-Transmitter)** — specifically COM1 at I/O port `0x3F8` — is your debug lifeline. QEMU can map the serial port to stdout with `-serial stdio`, meaning every byte your kernel writes to COM1 appears in your terminal. When your kernel triple-faults before VGA is initialized, the last serial output tells you where it died.

Unlike MMIO (which uses regular memory read/write instructions), serial port communication uses **port-mapped I/O** — the `in` and `out` instructions, which address a separate 16-bit I/O address space distinct from the memory address space. In C (freestanding, no `stdlib`), you access port I/O via inline assembly:

```
/* io.h - port I/O primitives */

static inline void outb(uint16_t port, uint8_t value) {
    __asm__ volatile ("outb %0, %1" : : "a"(value), "Nd"(port));
}

static inline uint8_t inb(uint16_t port) {
    uint8_t value;
    __asm__ volatile ("inb %1, %0" : "=a"(value) : "Nd"(port));
    return value;
}
```

The 16550 UART (the chip behind COM1) is initialized by configuring eight registers via port I/O. The most critical settings: baud rate divisor, 8N1 framing (8 data bits, no parity, 1 stop bit), and FIFO enablement.

```

/* serial.c - COM1 initialization */

#define COM1_BASE 0x3F8

void serial_init(void) {

    outb(COM1_BASE + 1, 0x00); // Disable interrupts

    outb(COM1_BASE + 3, 0x80); // Enable DLAB (Divisor Latch Access Bit)

    outb(COM1_BASE + 0, 0x03); // Divisor low byte: 3 → 38400 baud

    outb(COM1_BASE + 1, 0x00); // Divisor high byte

    outb(COM1_BASE + 3, 0x03); // 8 bits, no parity, 1 stop bit; clear DLAB

    outb(COM1_BASE + 2, 0xC7); // Enable FIFO, clear, 14-byte threshold

    outb(COM1_BASE + 4, 0x0B); // IRQs enabled, RTS/DSR set

}

void serial_putchar(char c) {

    /* Wait until transmitter holding register is empty */

    while ((inb(COM1_BASE + 5) & 0x20) == 0);

    outb(COM1_BASE, c);

}

```

Why busy-wait here? The UART has a small transmit buffer. Before sending a byte, you poll the Line Status Register (offset +5) until bit 5 (Transmitter Holding Register Empty — THRE) is set. This is a **busy-wait (spin loop)** — the CPU does nothing useful while waiting. In a production kernel, you would use interrupts and a transmit queue. At boot time, before interrupts are configured, busy-waiting is the only option. Every character costs approximately 260 microseconds at 38400 baud ($1 / 38400$ baud \times 10 bits per character $\approx 260\mu\text{s}$). For debug output, this is acceptable.

Combining both outputs into a `kprintf`-style function:

```

/* kprintf - simplified: formats and writes to both VGA and serial */

void kputchar(char c) {

    vga_putchar(c, VGA_COLOR_WHITE, VGA_COLOR_BLACK);

    serial_putchar(c);

    if (c == '\n') serial_putchar('\r'); /* Serial terminals need CR+LF */

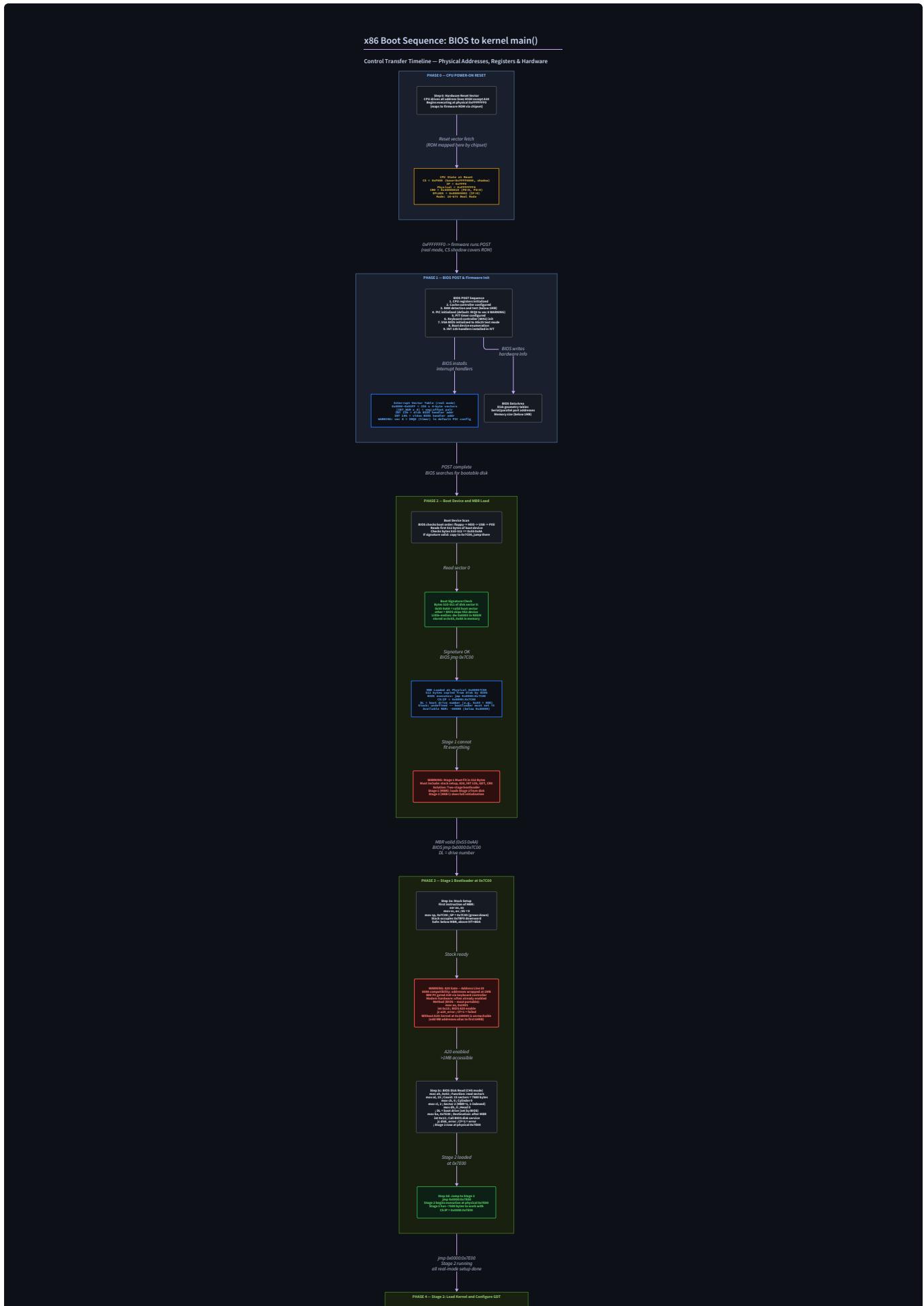
}

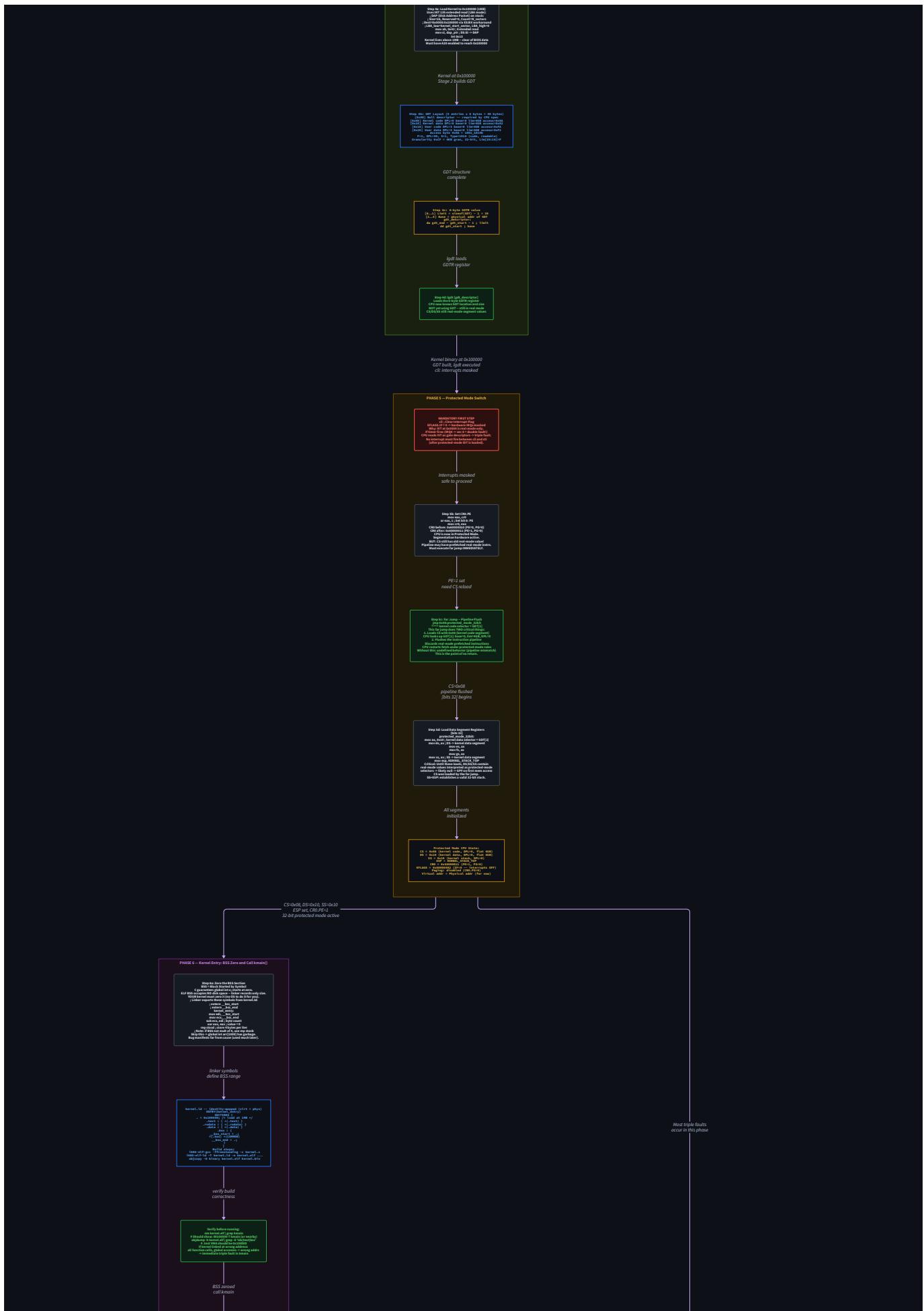
```

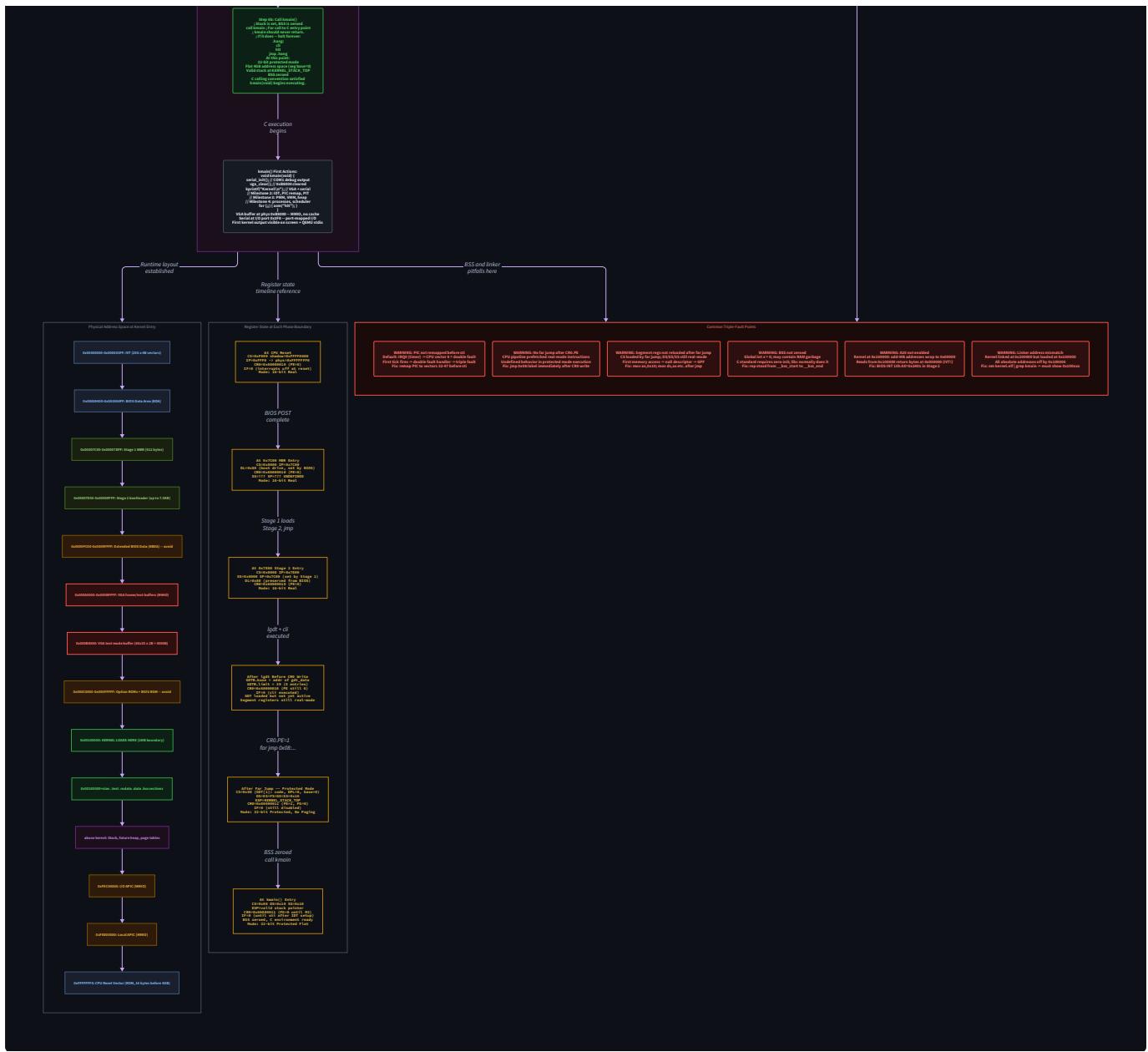
Section 9: The Complete Boot Sequence in One View

Let us trace exactly what happens from power-on to your kernel's `kmain()` function executing:

1. **CPU resets** → starts at `0xFFFFFFFF0` (firmware ROM) in real mode
2. **Firmware POST** → memory test, device enumeration, boot device selection
3. **Firmware loads MBR** → reads 512 bytes from disk to `0x7C00`, verifies `0x55 0xAA`, jumps to `0x7C00`
4. **Stage 1 runs** at `0x7C00` → uses INT 13h to load Stage 2 from disk to `0x7E00`, jumps to `0x7E00`
5. **Stage 2 runs** at `0x7E00`:
 - Enables A20 line
 - Loads kernel binary from disk to `0x100000` using INT 13h
 - Loads GDTR with `lgdt`
 - Disables interrupts with `cli`
 - Sets `CR0.PE = 1`
 - Far jumps to `0x08:protected_mode_32bit`
6. **32-bit protected mode** begins:
 - Loads `DS`, `ES`, `FS`, `GS`, `SS` with kernel data selector `0x10`
 - Sets `ESP` to kernel stack top
 - Zeros BSS from `__bss_start` to `__bss_end`
 - Calls `kmain()`
7. **`kmain()` runs:**
 - Calls `serial_init()` — COM1 is now available for debug output
 - Calls `vga_clear()` — clears VGA buffer
 - Calls `kprintf("Kernel booted successfully!\n")` — visible on screen and serial console







Section 10: Debugging in QEMU

QEMU is your primary development environment. Three essential invocations:

```
# Basic boot with serial output to terminal
```

BASH

```
qemu-system-i386 -drive format=raw,file=os.img -serial stdio
```

```
# With GDB stub (allows attaching a debugger)
```

```
qemu-system-i386 -drive format=raw,file=os.img -serial stdio \
-s -S      # -s: GDB on port 1234; -S: pause at startup
```

```
# In a separate terminal, attach GDB:
```

```
gdb kernel.elf
```

```
(gdb) target remote :1234
```

```
(gdb) break kmain
```

```
(gdb) continue
```

The most important debugging technique for this milestone: **if QEMU reboots immediately**, you have a triple fault. Add `-d int` to QEMU's flags to log every interrupt and exception to stderr. The last exception before the reset is your triple fault. Common causes at this stage:

Symptom	Likely Cause
Triple fault immediately after <code>CR0.PE</code> set	GDT access byte wrong; forgot far jump
Triple fault immediately after far jump	Segment registers not loaded with protected-mode selectors
Triple fault in <code>kmain</code>	BSS not zeroed; global variable assumed to be 0 but is not; or stack not set up
Black screen but no reboot	VGA write to wrong address; <code>volatile</code> missing; cursor position calculation wrong
Serial silent	DLAB not cleared before setting framing; wrong baud divisor

The **linker step** is where many silent bugs live. Your build command must produce a raw binary at exactly the right addresses:

```
# Compile kernel (freestanding: no stdlib, no default includes, no startup files)

i686-elf-gcc -m32 -ffreestanding -nostdlib -nostdinc \
-fno-builtin -fno-stack-protector \
-c kernel.c -o kernel.o

# Link with explicit linker script

i686-elf-ld -T kernel.ld -o kernel.elf kernel.o vga.o serial.o

# Extract raw binary (strip ELF headers; bootloader loads the raw kernel)

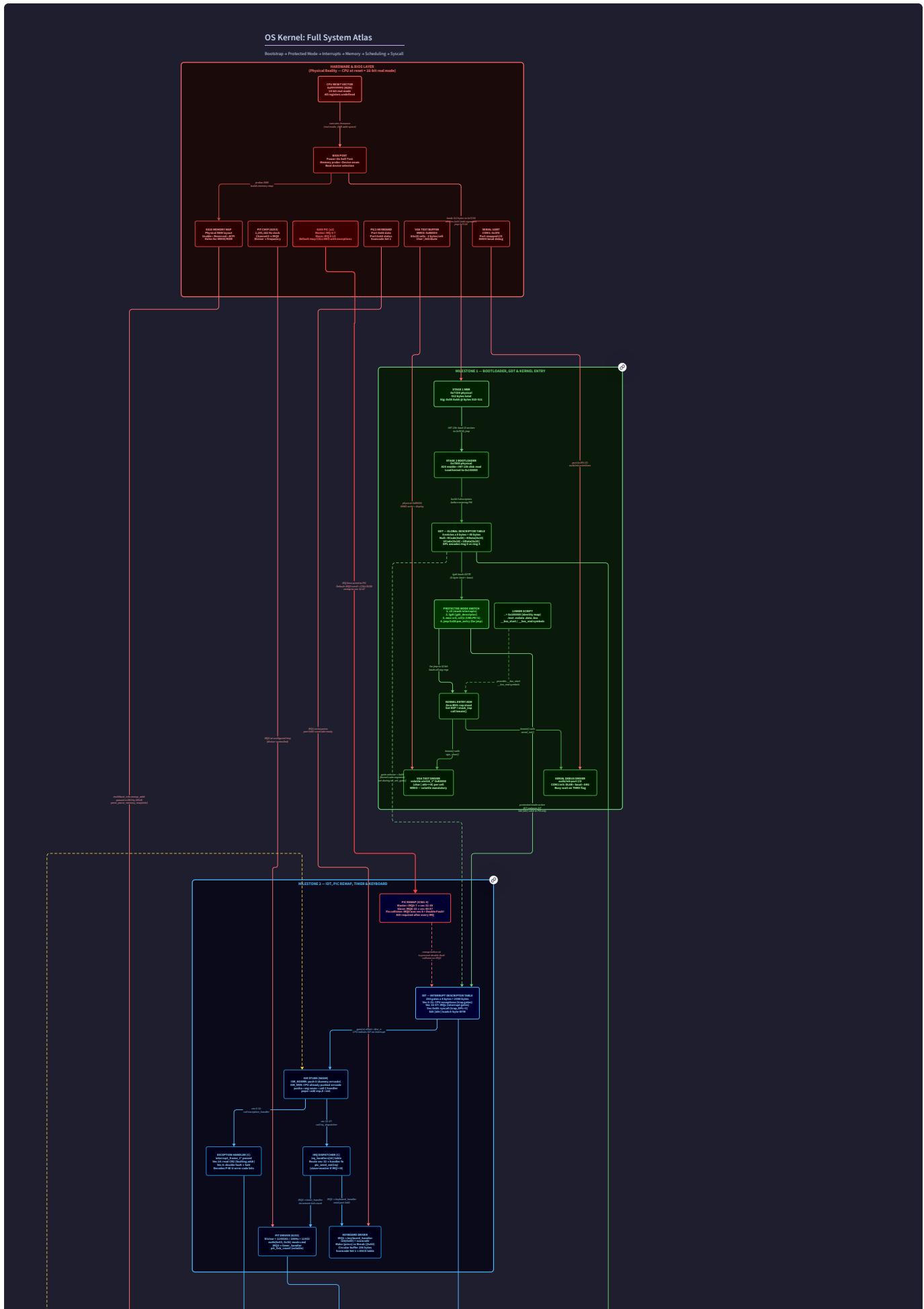
objcopy -O binary kernel.elf kernel.bin

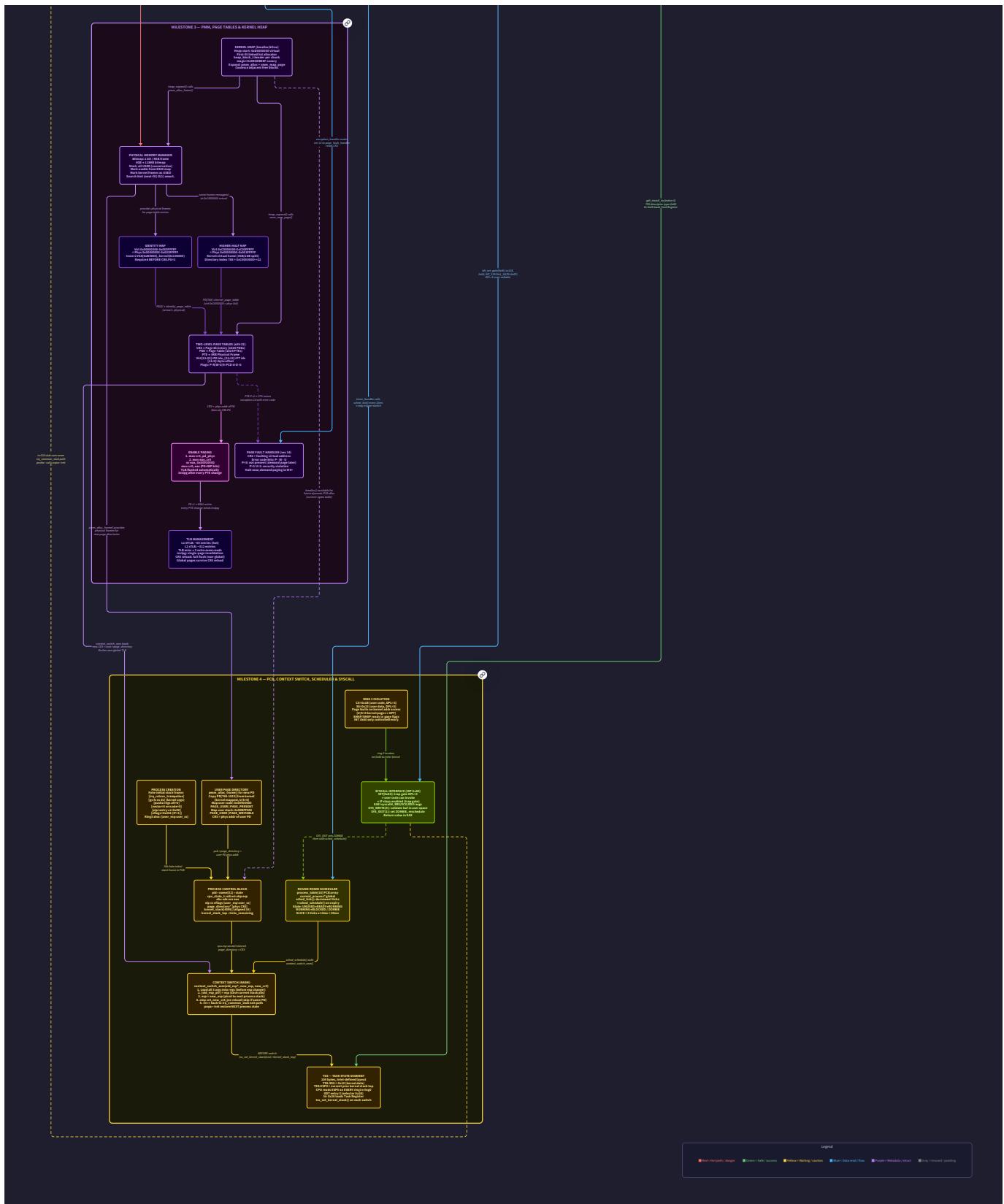
# Verify entry point address

nm kernel.elf | grep kmain      # Should show address 0x100000 or near
```

Cross-compiler: The host compiler (`gcc` on your Linux machine) targets the host OS's ABI — it links against the host libc, uses the host calling convention, and may generate instructions not available on your target. For OS development, you must use a **cross-compiler** targeting `i686-elf` (32-bit x86, ELF binary format, no host OS). The OSDev Wiki has a build script. This is one setup step worth doing correctly once.

The Three-Level View: What "Load a Kernel from Disk" Really Means





Level 1 — The Operation: Your bootloader calls INT 13h with AH=0x02, specifying a cylinder/head/sector address and a destination buffer. The BIOS returns with the data in memory.

Level 2 — BIOS and Firmware: INT 13h is a software interrupt that vectors through the IVT to BIOS code in ROM. The BIOS firmware contains drivers for legacy ATA/IDE controllers, floppy disk controllers, and (via INT 13h extensions) USB mass storage and SATA drives emulated as legacy disks. The firmware maintains an internal data structure (the BIOS Data Area at `0x400`) with disk geometry information discovered during POST. The BIOS firmware uses PIO (Programmed I/O) or DMA to transfer sectors from the disk controller's buffer to your specified memory address.

Level 3 — Hardware: The disk controller receives an ATA command (for SATA drives in legacy emulation mode: READ SECTORS, LBA28 or CHS addressing). The drive's internal controller reads data from spinning platters (HDD) or NAND flash (SSD), places it in the drive's sector buffer, and signals completion via IRQ14 (primary ATA channel). The BIOS handles the IRQ, copies data from the I/O ports (`0x1F0 - 0x1F7` for primary ATA) into memory using `rep insw` (read string from port), and returns to your code. On a real spinning disk, this read costs 3–10ms of seek time. On a VM using a file-backed disk image, it costs microseconds — the host OS has already cached it.

This entire chain — INT 13h software interrupt → BIOS firmware → ATA command → disk hardware → memory DMA — is the conceptual ancestor of every I/O abstraction that follows. Linux's block layer, NVMe command queues, and io_uring are all adding buffering, reordering, and concurrency on top of this same physical read.

Pitfalls Reference: The Seven Most Common Triple Faults

Pitfall	Why It Happens	Fix
GDT access byte wrong	Magic numbers are easy to misinform; one wrong bit	Build a descriptor macro; cross-check with OSDev wiki table
No far jump after <code>CR0 . PE</code>	Pipeline has real-mode prefetched instructions	The <code>jmp 0x08:label</code> instruction is <i>mandatory</i>
Interrupts not disabled	Hardware interrupt fires mid-transition with no IDT	<code>cli</code> must precede <code>lgdt</code>
A20 not enabled	Kernel loads to <code>0x100000</code> ; odd addresses wrap to first 64KB	Enable via BIOS INT 15h AX=2401h or I/O port method
BSS not zeroed	Global <code>int x = 0</code> has garbage value; logic errors far from cause	<code>rep stosd</code> or <code>rep stosb</code> from <code>__bss_start</code> to <code>__bss_end</code>
<code>volatile</code> missing on VGA	Compiler eliminates "dead" MMIO writes	Every MMIO access via <code>volatile</code> pointer
Linker script address mismatch	Kernel linked at <code>0x200000</code> but loaded at <code>0x100000</code> ; all jumps are wrong	Verify with <code>nm</code> , <code>objdump -h</code> ; check load address = link address

Knowledge Cascade — Ten Things This Unlocks

This milestone introduces foundational mechanisms that reappear everywhere in systems software. Each one below is a direct extension of what you just built.

1. Linker scripts → shared libraries, ASLR, and PIE binaries The difference between link-time address and load-time address is exactly how Linux shared libraries work. A `.so` file is compiled as **position-independent code (PIC)** and linked without fixed addresses. The dynamic linker relocates it to wherever in virtual memory is available. **ASLR (Address Space Layout Randomization)** randomizes the load address of every binary and library to make buffer overflow exploits harder. You just experienced the root concept: the address the linker embeds and the address the loader uses are separate concerns.

2. Freestanding environments → embedded firmware, WASM runtimes, UEFI drivers The discipline you just applied — no libc, no CRT0, explicit BSS zeroing, no floating-point ABI assumptions — is identical to what embedded firmware developers do for microcontrollers (ARM Cortex-M, RISC-V) and what WASM runtimes implement as their "OS-like" substrate. Every UEFI driver is also a freestanding binary that runs before an OS exists. You now understand why embedded developers care about startup code.

3. CPU privilege rings → browser process isolation, SGX enclaves, hypervisors Real mode vs protected mode is your first encounter with hardware-enforced privilege boundaries. CPL 0 (ring 0) vs CPL 3 (ring 3) is why a browser tab cannot read your kernel memory — the CPU refuses to execute user-mode code that accesses supervisor-only pages. Intel SGX (Software Guard Extensions) creates a ring -1 of sorts where even the OS cannot read enclave memory. Hypervisors add a VMX (Virtual Machine Extensions) layer below ring 0. All of these extend the same mechanism you just configured.

4. GDT segment registers → the death of segmentation in 64-bit mode In 64-bit x86 (long mode), segmentation is almost entirely disabled. `CS`, `DS`, `SS` are ignored for address calculation (base=0 always, limit irrelevant). The GDT still exists (the CPU reads it for privilege level and some system segments), but the flat-memory model you configured is now the *only* model. Understanding why 32-bit protected mode needed segments — and why 64-bit mode abandoned them — is a prerequisite for understanding why ARM64 and RISC-V never added segmentation at all.

5. BIOS INT 13h → Linux block layer → io_uring The disk read you just did is the bottom of an abstraction stack that has grown 50 years of complexity. Linux's block layer adds request queuing, elevator scheduling, and merge logic. NVMe command queues allow

65,535 queued commands per queue with zero-copy DMA. `io_uring` adds a shared ring buffer between kernel and userspace to eliminate syscall overhead entirely. The physical read at the bottom is the same; everything else is batching, reordering, and removing round-trips.

6. MMIO at 0xB8000 → GPU framebuffers → PCIe BAR mapping Writing to `0xB8000` and seeing characters appear is your first MMIO device. Modern GPUs expose tens of gigabytes of framebuffer and register space as MMIO via **PCIe BARs (Base Address Registers)**. The OS maps these physical addresses into virtual address space using page tables (flagged as uncacheable MMIO), and graphics drivers write directly to GPU registers and VRAM the same way you wrote to `0xB8000`. The mechanism scales to terabytes; the concept is identical.

7. The BSS section → demand-zeroing and zero pages in Linux Linux is lazy about zeroing memory. When you `mmap` anonymous memory, Linux maps all pages to a single read-only **zero page**. The first write to any of these pages triggers a copy-on-write page fault that allocates a real page and zeros it on demand. BSS segments in ELF executables are handled the same way — the kernel does not physically zero them at load time; it marks the pages as zero-fill-on-demand. You did this eagerly because you had no page fault handler yet. Linux does it lazily for efficiency.

8. volatile and MMIO → memory ordering and mfence / sfence in SMP kernels `volatile` prevents the compiler from reordering or eliminating MMIO accesses. On a multi-core machine, you also need **memory barriers** to prevent the CPU's out-of-order execution engine from reordering MMIO writes relative to other memory operations. `mfence` (Memory Fence), `sfence` (Store Fence), and `lfence` (Load Fence) are the hardware instructions for this. Device drivers in Linux use `wmb()`, `rmb()`, and `mb()` macros that compile to the appropriate fence instructions. Your `volatile` VGA driver is the conceptual root.

9. The bootloader size constraint → UEFI and the death of the MBR The 512-byte MBR is a constraint from 1983. **UEFI (Unified Extensible Firmware Interface)** replaces it with a firmware that reads a FAT32 partition (the EFI System Partition), loads a full ELF or PE executable (the bootloader), and provides a rich API for disk access, graphics, and device enumeration — all before the OS starts. GRUB 2 supports both BIOS MBR and UEFI. The Rust OS tutorial (phil-opp) targets UEFI. Understanding why the MBR is so constrained makes UEFI's design choices obvious.

10. Serial port debug output → GDB stubs → kernel crash reporters Your serial port is the simplest form of out-of-band debug output. Linux's **kgdb** (Kernel GNU Debugger) works over serial: the kernel includes a GDB stub that, on a kernel panic, starts listening on the serial port for GDB remote protocol commands. You can then attach from a second machine, inspect kernel memory, and walk the call stack — exactly like QEMU's `-s -S` flags. Production kernel crash reporters (`kdump`, `crash`) use a separate kernel loaded into reserved memory to capture the main kernel's state after a panic.

Milestone 2: Interrupts, Exceptions, and Keyboard

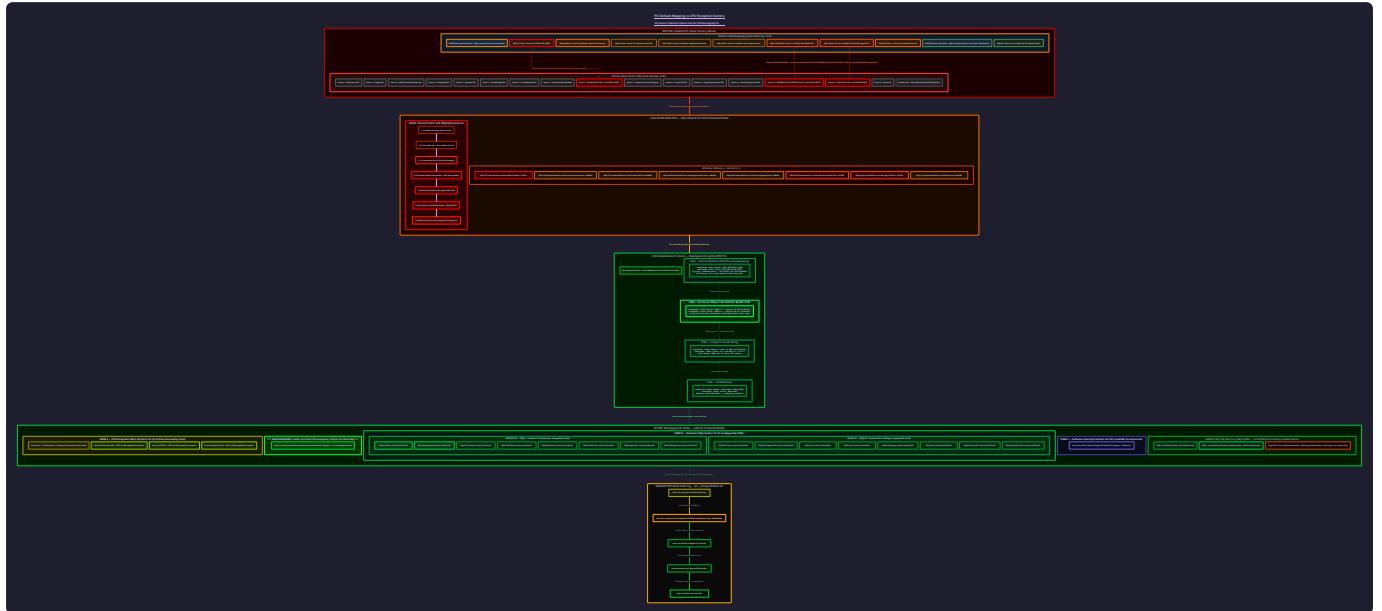
The Revelation: The PIC Was Built Before CPU Exceptions Existed

Here is what every developer who has written event-driven code believes about hardware interrupts: the CPU pauses whatever it is doing, runs a handler function, and resumes — like a JavaScript event listener or a POSIX signal handler, but lower-level. The interrupt numbers assigned to hardware devices just work, because surely someone designed the system as a coherent whole. This belief is understandable. It is also wrong in a way that will triple-fault your kernel the moment you enable interrupts without reading this section.

The Intel 8259 PIC (Programmable Interrupt Controller — the chip that translates hardware signals from devices into CPU interrupt vectors) was designed in 1976 for the 8080 processor. The x86 CPU exception model — the numbered exceptions for divide-by-zero, page faults, general protection faults — was designed in 1982 for the 286. These two teams did not coordinate. The result: the 8259 PIC, *out of the factory*, maps **IRQ0 (the timer interrupt) to CPU vector 8**. CPU vector 8, by the 286 exception specification, is the **double fault handler** — the CPU's last-resort recovery mechanism before a triple fault.

The consequence is direct: if you configure the IDT and enable interrupts without first remapping the PIC, the first timer tick — arriving approximately 55 milliseconds after you call `sti` — causes the CPU to invoke your double fault handler. Your double fault handler tries to display an error message. While it does so, another timer tick arrives. The CPU tries to invoke the double fault handler *again*, but it is already executing — this is a fault inside a fault handler, which is itself a double fault. The CPU looks for a double fault handler, which it cannot invoke because it is busy. **Triple fault. Reboot. No error message.**

This is not an obscure edge case. It is the default behavior of x86 hardware. Every OS developer encounters it. Now you know why it happens before it happens to you.



Section 1: The Interrupt Taxonomy — Not All Interrupts Are Created Equal

Before configuring hardware, you need a clear mental model of what the CPU actually does when something demands its attention. The x86 architecture has three distinct interrupt categories, and conflating them is the source of most beginner confusion.

Exceptions are synchronous events generated by the CPU itself in response to an instruction. They happen because of what code is doing right now, at this instruction. Examples: dividing by zero (exception 0), accessing a non-present page (exception 14 — the page fault), executing a privileged instruction from ring 3 (exception 13 — General Protection Fault). Exceptions can be further classified:

- **Faults:** The CPU reports the exception *before* completing the faulting instruction. If the handler fixes the problem (like a page fault handler that maps the missing page), the instruction retries from the beginning. The saved `EIP` points to the faulting instruction.
- **Traps:** The CPU reports the exception *after* completing the triggering instruction. The saved `EIP` points to the next instruction. Breakpoints (exception 3) and overflow (exception 4) are traps.
- **Aborts:** Unrecoverable errors. The CPU cannot reliably report the faulting instruction's address. Double fault (exception 8) and machine check (exception 18) are aborts. You cannot resume from an abort; you halt.

Hardware IRQs (Interrupt Requests) are asynchronous events generated by external devices. The keyboard pressed a key. The timer ticked. The network card received a packet. These arrive between instructions — the CPU finishes the current instruction, saves state, and jumps to the handler. The device generating the IRQ does not know or care what the CPU was doing; it just signals the PIC, which decides whether to forward the signal to the CPU based on its priority mask.

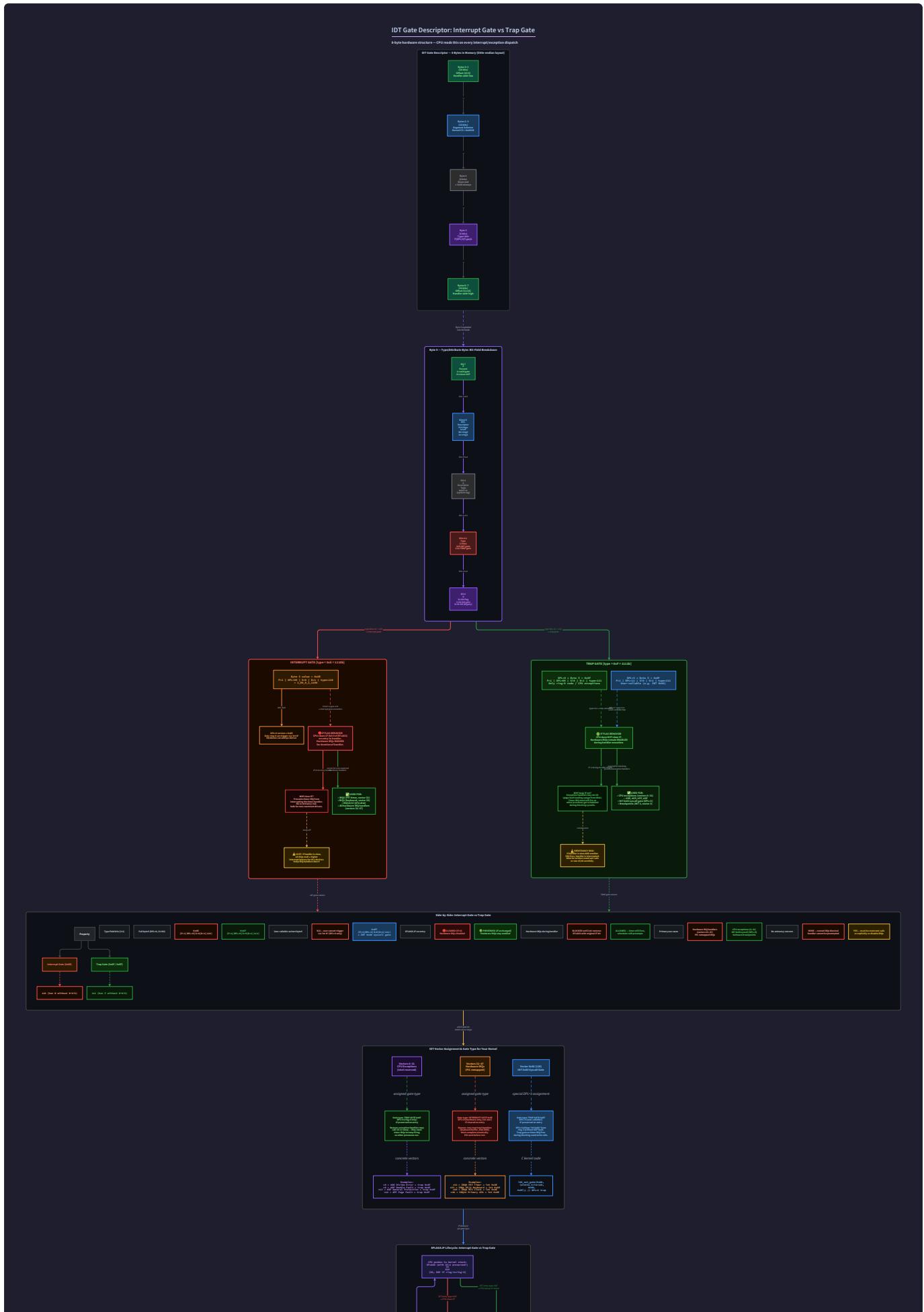
Software interrupts are explicitly triggered by the `int N` instruction. They look like hardware interrupts to the CPU but are generated intentionally by code. `int 0x80` is the Linux system call interface (historically); `int 0x03` is the software breakpoint. You will implement `int 0x80` as your system call gate in Milestone 4.

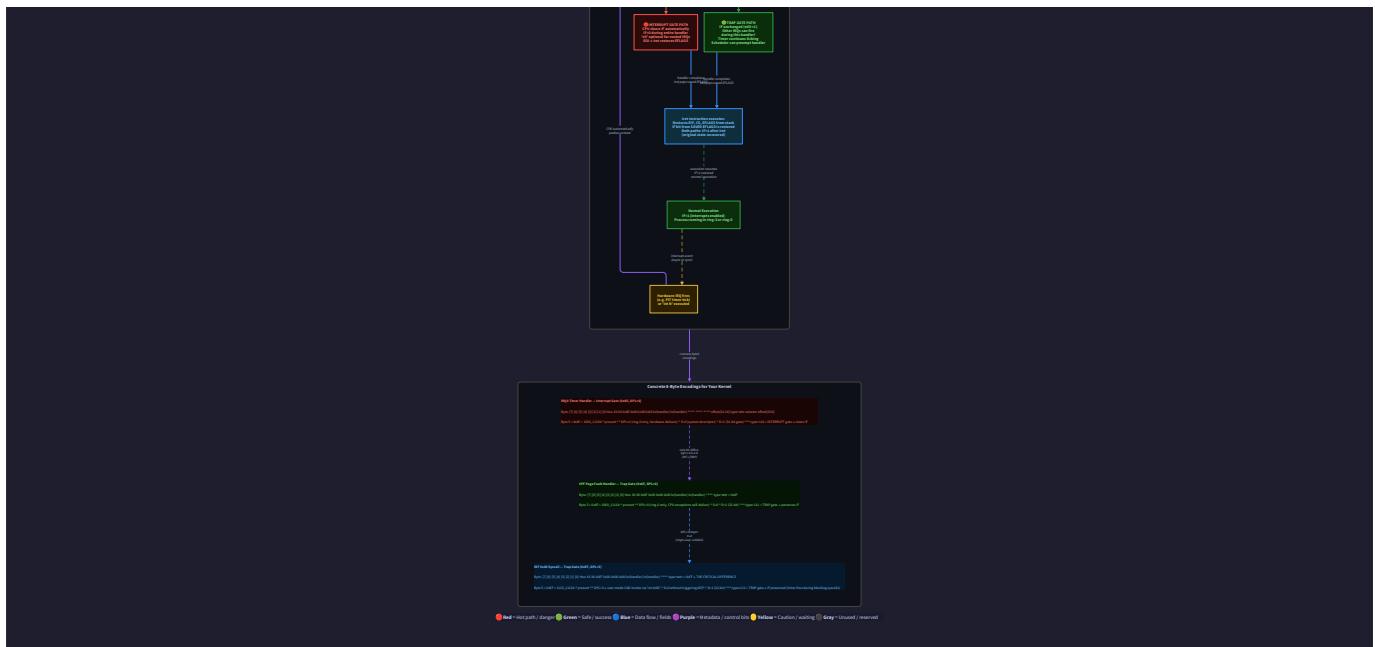
The CPU's perspective on all three: From the CPU's point of view, all three types go through the same mechanism — the IDT. The CPU uses the interrupt vector number as an index into the IDT, loads the gate descriptor, switches stacks if privilege level changes, pushes state, and jumps to the handler. The source of the interrupt (exception logic, PIC signal, or `int` instruction) is irrelevant to the dispatch mechanism. This uniformity is elegant; the complexity is in the details of each type.

Section 2: The IDT — 256 Slots, Each a Gate to a Handler

The **IDT (Interrupt Descriptor Table)** is the protected-mode analog of the real-mode IVT. Where the IVT was a flat array of 4-byte real-mode addresses (segment:offset pairs), the IDT is an array of 8-byte **gate descriptors** that specify not just where to jump, but *how* to jump: which privilege level is required to invoke this gate, whether to clear the interrupt flag on entry, and which code segment to use for the handler.

The IDT has exactly 256 entries, addressing every possible 8-bit interrupt vector. Entries 0–31 are reserved by Intel for CPU exceptions (though not all are currently used — unused ones should still have handlers that print a meaningful error and halt). Entries 32–255 are available for hardware IRQs and software interrupts.





The Gate Descriptor — Eight Bytes That Control Everything

Each IDT entry is an 8-byte gate descriptor. The most important gate types for a kernel are:

Interrupt Gate (type = 0xE for 32-bit): When the CPU invokes an interrupt gate, it automatically **clears the Interrupt Flag (IF)**. This means hardware interrupts are masked for the duration of the handler. This is the correct type for hardware IRQ handlers — you do not want a timer interrupt to interrupt your timer handler.

Trap Gate (type = 0xF for 32-bit): When invoked, the IF flag is *not* cleared. Hardware interrupts can arrive while the handler is running. This is the correct type for CPU exceptions (which should be debuggable — you want to be able to break into a debugger while a page fault is being handled) and for system calls (so that other processes can be interrupted while one is making a syscall).

IDT Gate Descriptor (8 bytes):								
Byte 7	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0	
Offset [31:16]	P DPL 0 Type 0 0 0 (0)			Segment		Offset [15:0]		
				Selector				

P = Present bit (1 = valid gate)
DPL = Descriptor Privilege Level (0 = only ring 0 can explicitly invoke; 3 = any ring)
Type = Gate type: 0xE = 32-bit interrupt gate, 0xF = 32-bit trap gate
Offset [31:16] = High 16 bits of handler address
Offset [15:0] = Low 16 bits of handler address
Segment Selector = Which code segment the handler runs in (usually 0x08: kernel code)

DPL on IDT gates: The DPL field controls which privilege level can *explicitly invoke* this gate with the `int N` instruction. For hardware IRQ handlers (DPL=0), user-mode code cannot trigger them via `int N` — only the hardware can. For the system call gate (DPL=3), user-mode code can explicitly call `int 0x80`. This is the gating mechanism that prevents user processes from directly triggering arbitrary kernel handlers.

In C, a gate descriptor is a packed struct:

```
/* idt.h */

#include <stdint.h>

typedef struct __attribute__((packed)) {

    uint16_t offset_low;      /* Handler address bits [15:0] */
    uint16_t selector;       /* Code segment selector (0x08) */
    uint8_t zero;            /* Always 0 */
    uint8_t type_attr;       /* P|DPL|0|Type bits */
    uint16_t offset_high;    /* Handler address bits [31:16] */

} idt_gate_t;

typedef struct __attribute__((packed)) {

    uint16_t limit;          /* Size of IDT in bytes minus 1 */
    uint32_t base;           /* Physical address of IDT */

} idtr_t;

#define IDT_ENTRIES 256

static idt_gate_t idt[IDT_ENTRIES];

static idtr_t idtr;

/* Gate type + attribute bytes */

#define IDT_INTERRUPT_GATE 0x8E /* P=1, DPL=0, type=0xE (32-bit interrupt gate) */
#define IDT_TRAP_GATE      0x8F /* P=1, DPL=0, type=0xF (32-bit trap gate) */
#define IDT_SYSCALL_GATE   0xEF /* P=1, DPL=3, type=0xF (user-callable trap) */

void idt_set_gate(uint8_t vector, uint32_t handler, uint16_t selector, uint8_t type_attr) {

    idt[vector].offset_low = handler & 0xFFFF;
    idt[vector].selector = selector;
    idt[vector].zero = 0;
    idt[vector].type_attr = type_attr;
    idt[vector].offset_high = (handler >> 16) & 0xFFFF;
}

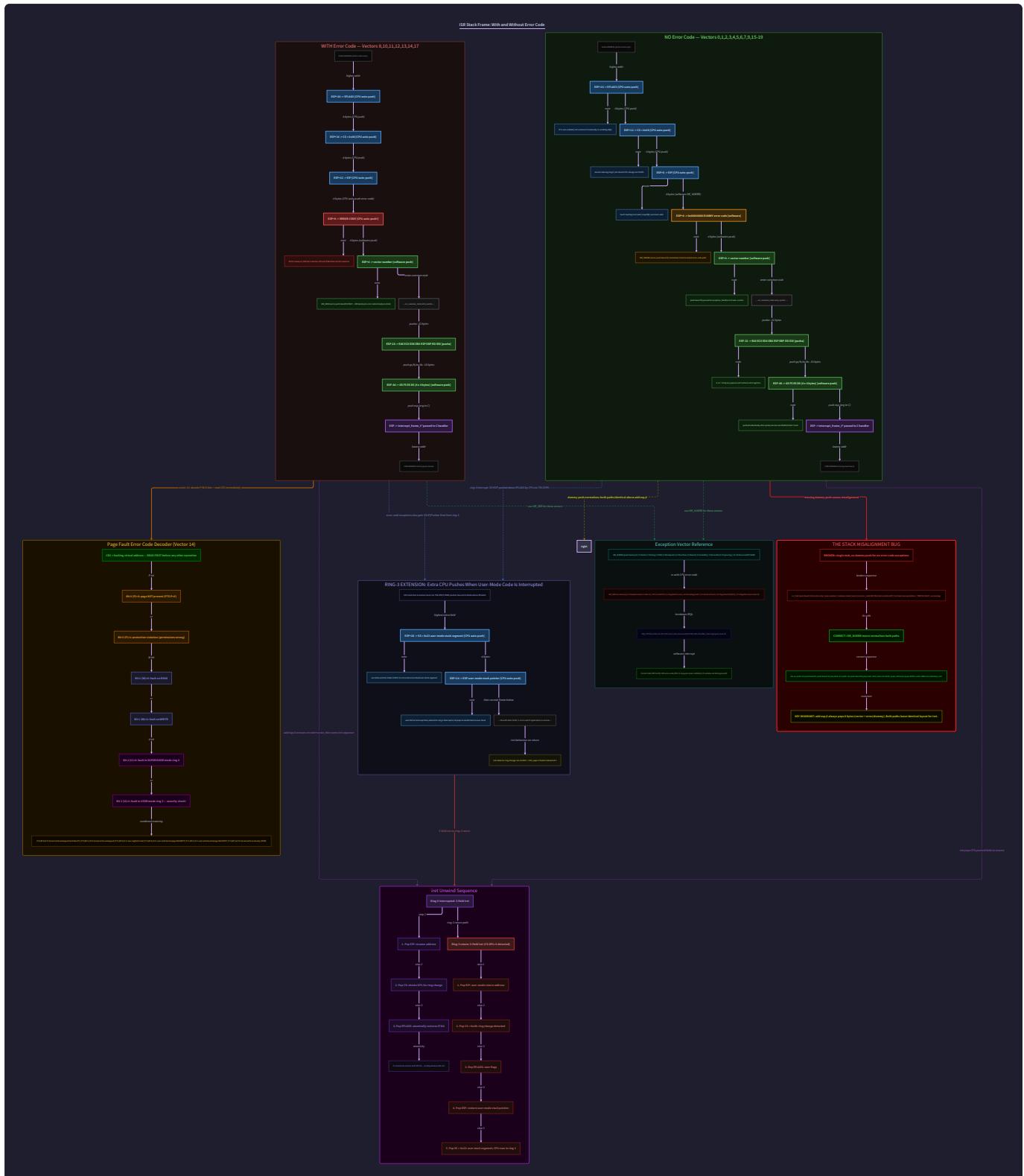
void idt_install(void) {
    idtr.limit = sizeof(idt) - 1;
    idtr.base = (uint32_t)&idt;
```

```
__asm__ volatile ("lidt %0" : : "m"(idtr));  
}
```

`__attribute__((packed))` : GCC normally inserts padding bytes between struct fields to align them to their natural alignment (4-byte fields at 4-byte-aligned offsets). The IDT gate descriptor's layout is defined by hardware specification — the bytes must be in exactly the positions Intel specifies, with no padding. `packed` instructs GCC to pack fields with no padding. This is essential for any struct whose layout is dictated by hardware rather than software convention.

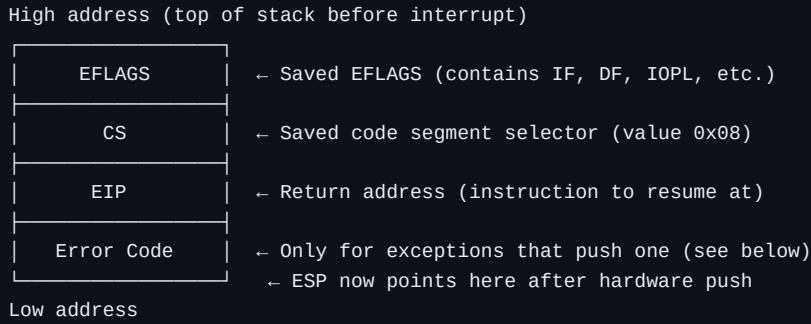
Section 3: The ISR Stack Frame — Why `iret` Is Not `ret`

When the CPU invokes an interrupt handler, it does something that `call` does not: it automatically saves not just the return address but the entire privilege-transition context. Understanding this stack frame is essential — miscounting the items on the stack is one of the most common sources of silent corruption in kernel development.



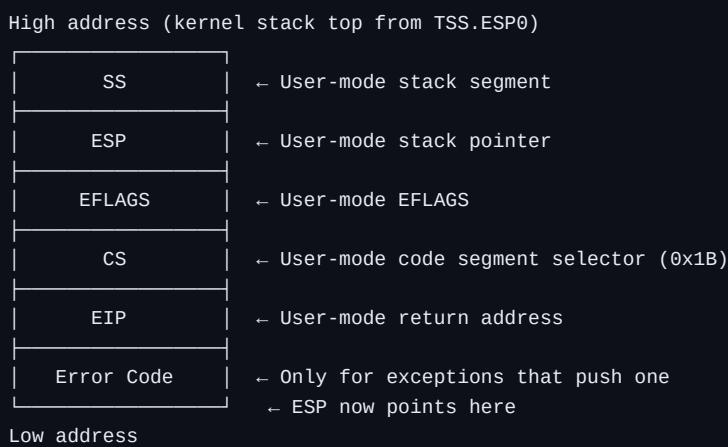
For interrupts arriving while in ring 0 (kernel mode):

The CPU pushes onto the *current* stack (no stack switch required since we are already in ring 0):



For interrupts arriving while in ring 3 (user mode):

The CPU also performs a privilege-level transition. It reads the kernel stack pointer from the TSS (Task State Segment — configured in Milestone 4), switches to the kernel stack, and pushes additional values:



`iret` (Interrupt Return) is the counterpart to this automatic push. It pops `EIP`, `CS`, and `EFLAGS` from the stack and restores them atomically. If a privilege transition occurred (CS's DPL changed), `iret` also pops `ESP` and `SS` and switches back to the user stack. This is why you cannot use `ret` to return from an interrupt handler — `ret` only pops `EIP`, leaving `CS` and `EFLAGS` on the stack, corrupting the stack pointer for the resumed code.

The Error Code Asymmetry — The Source of Countless Stack Misalignments

Of the 32 CPU exception vectors, some push an error code onto the stack *before* invoking the handler; others do not. The asymmetry is not random — it reflects whether the exception provides meaningful fault information:

Vector	Name	Error Code?
0	Divide Error	No
1	Debug	No
2	NMI	No
3	Breakpoint	No
4	Overflow	No
5	Bound Range	No
6	Invalid Opcode	No
7	Device Not Available	No
8	Double Fault	Yes (always 0)
10	Invalid TSS	Yes
11	Segment Not Present	Yes
12	Stack-Segment Fault	Yes
13	General Protection Fault	Yes
14	Page Fault	Yes
17	Alignment Check	Yes

If you write a single ISR stub that does not account for this asymmetry, exceptions without error codes will have a misaligned stack when the handler reads `esp+8` for the error code — it will read garbage, and `iret` will restore the wrong `EIP`. The standard solution: for exceptions without error codes, push a **dummy zero** as an error code before calling the generic handler.

```
; isr_stubs.asm - NASM syntax
; For exceptions WITHOUT error code: push dummy 0 first
%macro ISR_NOERR 1
    global isr%1
    isr%1:
        cli
        push dword 0      ; Dummy error code (maintains uniform stack frame)
        push dword %1      ; Exception vector number
        jmp isr_common_stub
%endmacro

; For exceptions WITH error code: CPU already pushed it
%macro ISR_ERR 1
    global isr%1
    isr%1:
        cli
        push dword %1      ; Exception vector number (error code already on stack)
        jmp isr_common_stub
%endmacro

ISR_NOERR 0    ; Divide error
ISR_NOERR 1    ; Debug
ISR_NOERR 2    ; NMI
ISR_NOERR 3    ; Breakpoint
ISR_NOERR 4    ; Overflow
ISR_NOERR 5    ; Bound range
ISR_NOERR 6    ; Invalid opcode
ISR_NOERR 7    ; Device not available
ISR_ERR 8      ; Double fault (error code = 0, but CPU pushes it)
ISR_NOERR 9    ; Coprocessor segment overrun (legacy)
ISR_ERR 10     ; Invalid TSS
ISR_ERR 11     ; Segment not present
ISR_ERR 12     ; Stack-segment fault
ISR_ERR 13     ; General protection fault
ISR_ERR 14     ; Page fault
ISR_NOERR 15   ; Reserved
ISR_NOERR 16   ; x87 FPU error
ISR_ERR 17     ; Alignment check
ISR_NOERR 18   ; Machine check
ISR_NOERR 19   ; SIMD floating-point
; ... ISR_NOERR 20 through 31 for remaining reserved vectors
```

The common stub — called by all exception stubs — saves all registers, calls the C handler, and restores:

```

; isr_common_stub - unified entry point for all exception handlers
isr_common_stub:
; At this point the stack contains (top to bottom):
;   vector number, error code, EIP, CS, EFLAGS [, ESP, SS if ring-3]

pusha           ; Push EAX, ECX, EDX, EBX, ESP(before pusha), EBP, ESI, EDI
; This saves all general-purpose registers

; Save segment registers
mov ax, ds
push eax
mov ax, es
push eax
mov ax, fs
push eax
mov ax, gs
push eax

; Load kernel data segment (in case we were interrupted in user mode)
mov ax, 0x10
mov ds, ax
mov es, ax
mov fs, ax
mov gs, ax

; Call C handler: pass pointer to stack frame as argument
push esp        ; Argument: pointer to the saved register state
call exception_handler
add esp, 4      ; Clean up argument

; Restore segment registers
pop eax
mov gs, ax
pop eax
mov fs, ax
pop eax
mov es, ax
pop eax
mov ds, ax

popa           ; Restore general-purpose registers
add esp, 8      ; Pop the error code and vector number we pushed
iret           ; Restore EIP, CS, EFLAGS (and ESP, SS if ring-3 transition)

```

pusha / popa (Push/Pop All): `pusha` pushes `EAX`, `ECX`, `EDX`, `EBX`, `ESP` (the value *before* `pusha` executed), `EBP`, `ESI`, `EDI` — eight registers, 32 bytes — in a single instruction. `popa` restores them in reverse order. This saves and restores the entire general-purpose register state. If you omit `pusha / popa`, the interrupted code resumes with garbage in any register that your handler modified — a bug that manifests far from the interrupt, because the corrupted register might not be used until many instructions later.

The C handler receives a pointer to the saved register frame on the stack:

```
/* interrupt.c */

typedef struct {

    uint32_t gs, fs, es, ds;                      /* Segment registers (pushed last → restored first) */

    uint32_t edi, esi, ebp, esp;                  /* From pusha */

    uint32_t ebx, edx, ecx, eax;                  /* From pusha */

    uint32_t vector, error_code;                  /* Pushed by ISR stub */

    uint32_t eip, cs, eflags;                     /* Pushed by CPU */

    /* uint32_t user_esp, user_ss; — only present on ring-3 → ring-0 transition */

} interrupt_frame_t;

static const char *exception_names[] = {

    "Divide Error",           /* 0 */

    "Debug",                 /* 1 */

    "Non-Maskable Interrupt", /* 2 */

    "Breakpoint",            /* 3 */

    "Overflow",              /* 4 */

    "Bound Range Exceeded",  /* 5 */

    "Invalid Opcode",        /* 6 */

    "Device Not Available",  /* 7 */

    "Double Fault",          /* 8 */

    "Coprocessor Seg Overrun", /* 9 */

    "Invalid TSS",           /* 10 */

    "Segment Not Present",   /* 11 */

    "Stack-Segment Fault",   /* 12 */

    "General Protection Fault", /* 13 */

    "Page Fault",            /* 14 */

    "Reserved",              /* 15 */

    "x87 Floating-Point",    /* 16 */

    "Alignment Check",       /* 17 */

    "Machine Check",         /* 18 */

    "SIMD Floating-Point",   /* 19 */

    /* 20-31: reserved */

};


```

```

void exception_handler(interrupt_frame_t *frame) {
    if (frame->vector < 32) {
        kprintf("\n==== KERNEL EXCEPTION ====\n");
        kprintf("Exception: %s (vector %d)\n",
               exception_names[frame->vector], frame->vector);
        kprintf("Error code: 0x%08x\n", frame->error_code);
        kprintf("EIP: 0x%08x CS: 0x%04x EFLAGS: 0x%08x\n",
               frame->eip, frame->cs, frame->eflags);
        kprintf("EAX: 0x%08x EBX: 0x%08x ECX: 0x%08x EDX: 0x%08x\n",
               frame->eax, frame->ebx, frame->ecx, frame->edx);

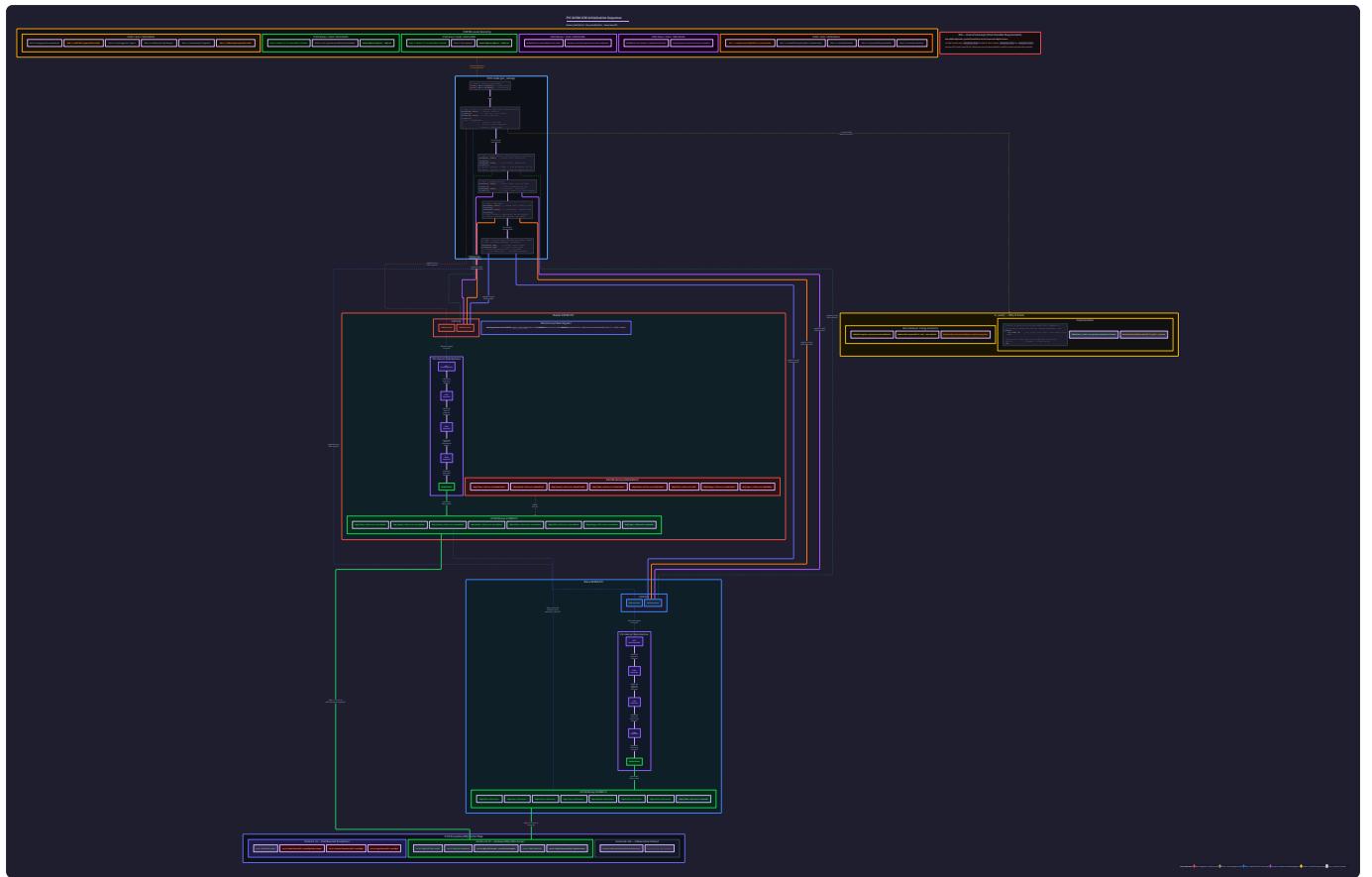
        /* For page faults, CR2 contains the faulting virtual address */
        if (frame->vector == 14) {
            uint32_t cr2;
            __asm__ volatile ("mov %%cr2, %0" : "=r"(cr2));
            kprintf("Faulting address (CR2): 0x%08x\n", cr2);
        }
    }

    /* Halt – cannot recover from unhandled kernel exceptions */
    kprintf("System halted.\n");
    for (;;) { __asm__ volatile ("cli; hlt"); }
}

```

Section 4: The 8259 PIC — Remapping Before It Kills You

The **8259 PIC (Programmable Interrupt Controller)** is a chip that sits between hardware devices and the CPU. Up to 15 hardware IRQ lines feed into the PIC (or a cascade of two PICs — master and slave), and the PIC translates them into CPU interrupt vectors. The critical word is *programmable*: the mapping from IRQ number to vector number is configurable. It just ships from the factory configured incorrectly for protected-mode x86.



The standard PC architecture uses **two 8259 PICs in cascade**:

- **Master PIC** (ports `0x20` / `0x21`): handles IRQ0–IRQ7
- **Slave PIC** (ports `0xA0` / `0xA1`): handles IRQ8–IRQ15; connected to master via IRQ2

The IRQ-to-device mapping:

IRQ	Device
0	PIT Timer (Programmable Interval Timer)
1	PS/2 Keyboard
2	Cascade (slave PIC)
3	COM2 Serial
4	COM1 Serial
5	LPT2 / Sound
6	Floppy
7	LPT1 / Spurious
8	CMOS Real-Time Clock
9	Legacy SCSI / Free
10	Free
11	Free
12	PS/2 Mouse
13	FPU
14	Primary ATA
15	Secondary ATA / Spurious

Default mapping (wrong for protected mode): IRQ0 → vector 8, IRQ1 → vector 9, ..., IRQ7 → vector 15, IRQ8 → vector 112 (or 0x70), ..., IRQ15 → vector 119. Vectors 8–15 collide directly with CPU exceptions (double fault, invalid TSS, segment not present, stack fault, GPF, page fault, reserved, coprocessor error).

Target mapping (correct for protected mode): IRQ0 → vector 32, IRQ1 → vector 33, ..., IRQ7 → vector 39, IRQ8 → vector 40, ..., IRQ15 → vector 47. This places all hardware IRQs above the 32 reserved exception vectors.

The ICW Initialization Protocol

Reprogramming the PIC requires sending a precise sequence of **ICW (Initialization Command Words)** to specific I/O ports. The PIC enters initialization mode when it receives ICW1, then expects ICW2, ICW3, and ICW4 in order — the chip has a state machine that advances through these command words automatically.

```

/* pic.c - PIC remapping */

#include "io.h" /* outb/inb from Milestone 1 */

#define PIC1_CMD    0x20    /* Master PIC command port */
#define PIC1_DATA   0x21    /* Master PIC data port */
#define PIC2_CMD    0xA0    /* Slave PIC command port */
#define PIC2_DATA   0xA1    /* Slave PIC data port */

#define PIC_EOI     0x20    /* End-of-Interrupt command */
#define ICW1_INIT   0x10    /* Initialize bit (required) */
#define ICW1_ICW4   0x01    /* ICW4 will be provided */
#define ICW4_8086   0x01    /* 8086/88 mode (vs MCS-80/85) */

/* Small I/O delay - some hardware needs time between PIC writes */

static void io_wait(void) {
    outb(0x80, 0); /* Port 0x80 is unused POST diagnostic port; write is ~1µs delay */
}

void pic_remap(uint8_t master_offset, uint8_t slave_offset) {
    /* Save current interrupt masks (we will restore them after remapping) */

    uint8_t master_mask = inb(PIC1_DATA);
    uint8_t slave_mask = inb(PIC2_DATA);

    /* ICW1: Start initialization sequence, cascade mode, ICW4 needed */
    outb(PIC1_CMD, ICW1_INIT | ICW1_ICW4); io_wait();
    outb(PIC2_CMD, ICW1_INIT | ICW1_ICW4); io_wait();

    /* ICW2: Set vector offsets

        Master PIC: IRQ0-7 → vectors master_offset to master_offset+7
        Slave PIC:  IRQ8-15 → vectors slave_offset to slave_offset+7 */

    outb(PIC1_DATA, master_offset); io_wait(); /* e.g., 0x20 = 32 */
    outb(PIC2_DATA, slave_offset); io_wait(); /* e.g., 0x28 = 40 */

    /* ICW3: Configure cascade wiring

        Master: bitmask of which IRQ line the slave is connected to (IRQ2 = bit 2 = 0x04)
        Slave: numeric IRQ line it is connected to on master (2 = 0x02) */
}

```

```

outb(PIC1_DATA, 0x04); io_wait(); /* Slave on master's IRQ2 */

outb(PIC2_DATA, 0x02); io_wait(); /* Slave ID = 2 */

/* ICW4: Set 8086 mode (vs legacy 8080 mode) */

outb(PIC1_DATA, ICW4_8086); io_wait();

outb(PIC2_DATA, ICW4_8086); io_wait();

/* Restore saved masks */

outb(PIC1_DATA, master_mask);

outb(PIC2_DATA, slave_mask);

}

```

Call this as: `pic_remap(0x20, 0x28)` — mapping IRQ0–7 to vectors 32–39, IRQ8–15 to vectors 40–47. This must be called **before** `sti` (enabling interrupts) and ideally before setting up IRQ handlers in the IDT, so there is never a window where an IRQ can arrive at the wrong vector.

EOI — The Acknowledgment That Must Never Be Forgotten

After the PIC delivers an IRQ to the CPU and the CPU starts executing the handler, the PIC sets an **ISR bit (In-Service Register bit)** for that IRQ line and masks further interrupts on that line. The PIC will not deliver another interrupt on that line — or, for a master PIC, will not deliver lower-priority IRQs — until it receives an **EOI (End of Interrupt)** signal.

If you forget to send EOI, the PIC permanently suppresses that IRQ line. The timer stops firing. The keyboard goes silent. The system does not crash — it simply stops responding to hardware events, with no error message. This is one of the most insidious bugs in kernel development.

The rule: if the IRQ came from the **slave PIC** (IRQ8–15), you must send EOI to **both the slave and the master**. If it came from the **master PIC** (IRQ0–7, excluding IRQ2), send EOI only to the master.

```

void pic_send_eoi(uint8_t irq) {

    if (irq >= 8) {

        /* IRQ came from slave PIC – slave must be acknowledged first */

        outb(PIC2_CMD, PIC_EOI);

    }

    /* Always acknowledge master PIC */

    outb(PIC1_CMD, PIC_EOI);

}

```

Spurious IRQs — the phantom interrupt: IRQ7 (master) and IRQ15 (slave) are special. Due to electrical characteristics of the PIC, a noise spike or very short IRQ pulse can cause the PIC to issue an interrupt but then clear it before the CPU reads the ISR. The CPU receives vector 39 (IRQ7) or 47 (IRQ15) — a "spurious" interrupt with no real source. Your handler must check the PIC's ISR register to distinguish real IRQ7/15 from spurious ones. A spurious IRQ7 must *not* receive an EOI (the PIC did not actually set the ISR bit); a spurious IRQ15 requires an EOI to the *master* but not the slave. Most beginner kernels ignore this and get away with it — until they run on hardware that generates noise.

Section 5: The IRQ Stub Framework

Hardware IRQ handlers follow the same save/restore pattern as exception handlers, but they also need to send EOI. The cleanest approach: separate stubs for each IRQ (vectors 32–47), each pushing the IRQ number and jumping to a common handler:

```

; irq_stubs.asm - IRQ handler stubs (vectors 32-47)

%macro IRQ_STUB 1
    global irq%1
    irq%1:
        cli
        push dword 0          ; No error code for hardware IRQs
        push dword (%1 + 32); Vector number (IRQ0 → 32, IRQ1 → 33, etc.)
        jmp irq_common_stub
%endmacro

IRQ_STUB 0    ; Timer (PIT, IRQ0 → vector 32)
IRQ_STUB 1    ; Keyboard (IRQ1 → vector 33)
IRQ_STUB 2    ; Cascade (IRQ2 → vector 34, should never fire)
IRQ_STUB 3    ; COM2
IRQ_STUB 4    ; COM1
IRQ_STUB 5    ; LPT2
IRQ_STUB 6    ; Floppy
IRQ_STUB 7    ; LPT1 / Spurious
IRQ_STUB 8    ; RTC (IRQ8 → vector 40)
IRQ_STUB 9
IRQ_STUB 10
IRQ_STUB 11
IRQ_STUB 12    ; PS/2 Mouse
IRQ_STUB 13    ; FPU
IRQ_STUB 14    ; Primary ATA
IRQ_STUB 15    ; Secondary ATA / Spurious

irq_common_stub:
    pusha
    push ds
    push es
    push fs
    push gs
    mov ax, 0x10
    mov ds, ax
    mov es, ax
    mov fs, ax
    mov gs, ax
    push esp
    call irq_dispatcher      ; C function: void irq_dispatcher(interrupt_frame_t*)
    add esp, 4
    pop eax
    mov gs, ax
    pop eax
    mov fs, ax
    pop eax
    mov es, ax
    pop eax
    mov ds, ax
    popa
    add esp, 8              ; Pop dummy error code and vector number
    iret

```

The C dispatcher routes to registered handlers:

```

/* irq.c */

typedef void (*irq_handler_t)(interrupt_frame_t *frame);

static irq_handler_t irq_handlers[16]; /* Handlers for IRQ0-15 */

void irq_install_handler(uint8_t irq, irq_handler_t handler) {
    irq_handlers[irq] = handler;
}

void irq_dispatcher(interrupt_frame_t *frame) {
    uint8_t irq = frame->vector - 32; /* Convert vector back to IRQ number */

    if (irq_handlers[irq]) {
        irq_handlers[irq](frame);
    }

    pic_send_eoi(irq); /* ALWAYS send EOI, even if no handler registered */
}

```

EOI before or after the handler? Sending EOI before calling the handler allows the PIC to deliver another interrupt on the same IRQ line while the handler is still running. This is risky for non-reentrant handlers (like the keyboard buffer handler) — a second keyboard interrupt while the first is being processed can corrupt the buffer. Sending EOI *after* the handler is safer for simple kernels: the PIC waits until the handler completes. In production kernels (Linux), EOI timing is more nuanced and depends on interrupt threading. For this kernel: send EOI after the handler.

Now register the IDT gates for all 48 vectors (0–31 exceptions, 32–47 IRQs):

```

void idt_setup_all(void) {
    /* CPU exceptions: trap gates (IF not cleared, allows debugger interruption) */

    idt_set_gate(0, (uint32_t)isr0, 0x08, IDT_TRAP_GATE);

    idt_set_gate(1, (uint32_t)isr1, 0x08, IDT_TRAP_GATE);

    /* ... for all 0-31 */

    idt_set_gate(8, (uint32_t)isr8, 0x08, IDT_TRAP_GATE); /* Double fault */

    idt_set_gate(13, (uint32_t)isr13, 0x08, IDT_TRAP_GATE); /* GPF */

    idt_set_gate(14, (uint32_t)isr14, 0x08, IDT_TRAP_GATE); /* Page fault */

    /* Hardware IRQs: interrupt gates (IF cleared on entry – no nested IRQs) */

    idt_set_gate(32, (uint32_t)irq0, 0x08, IDT_INTERRUPT_GATE);

    idt_set_gate(33, (uint32_t)irq1, 0x08, IDT_INTERRUPT_GATE);

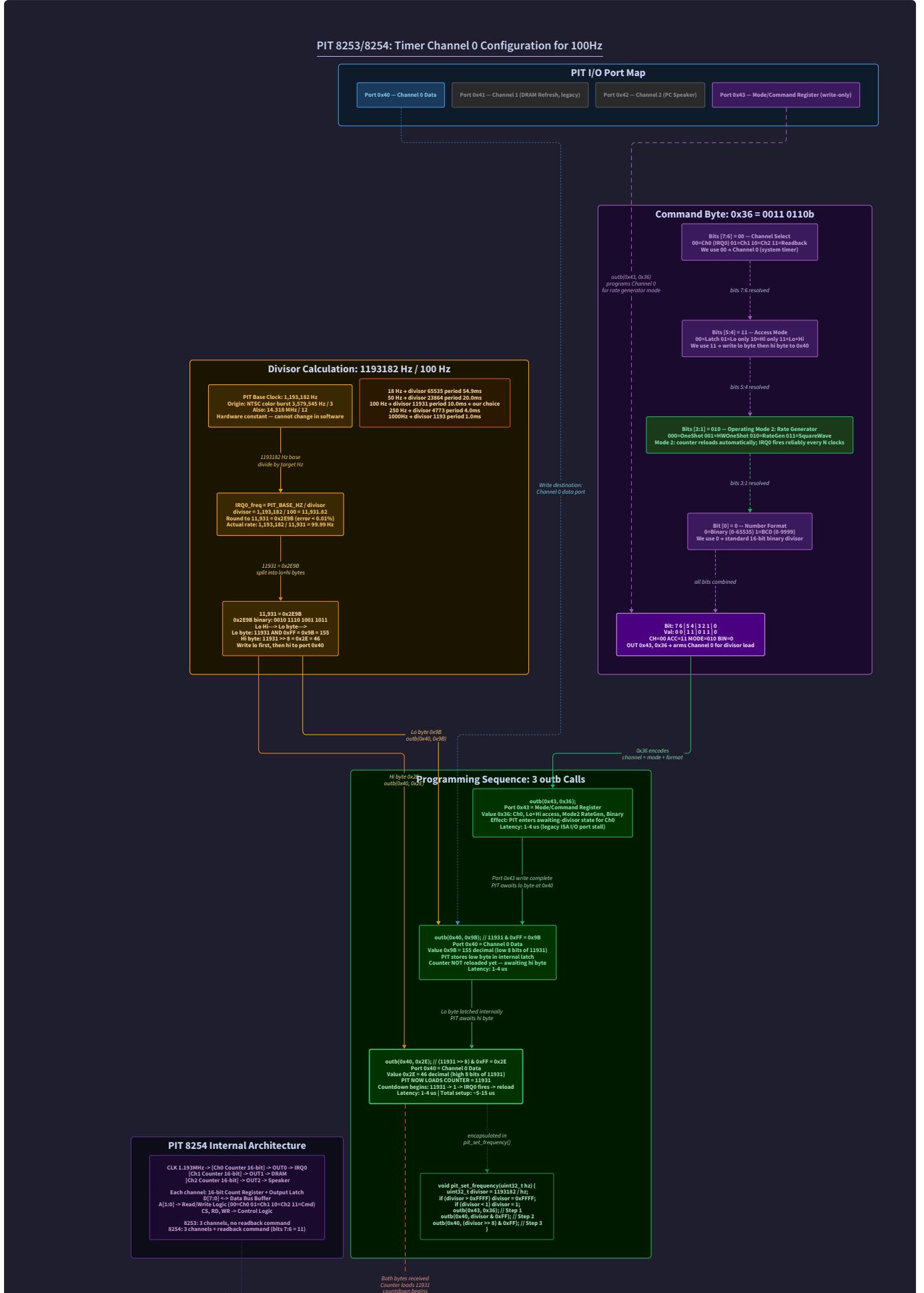
    /* ... for all IRQ stubs */

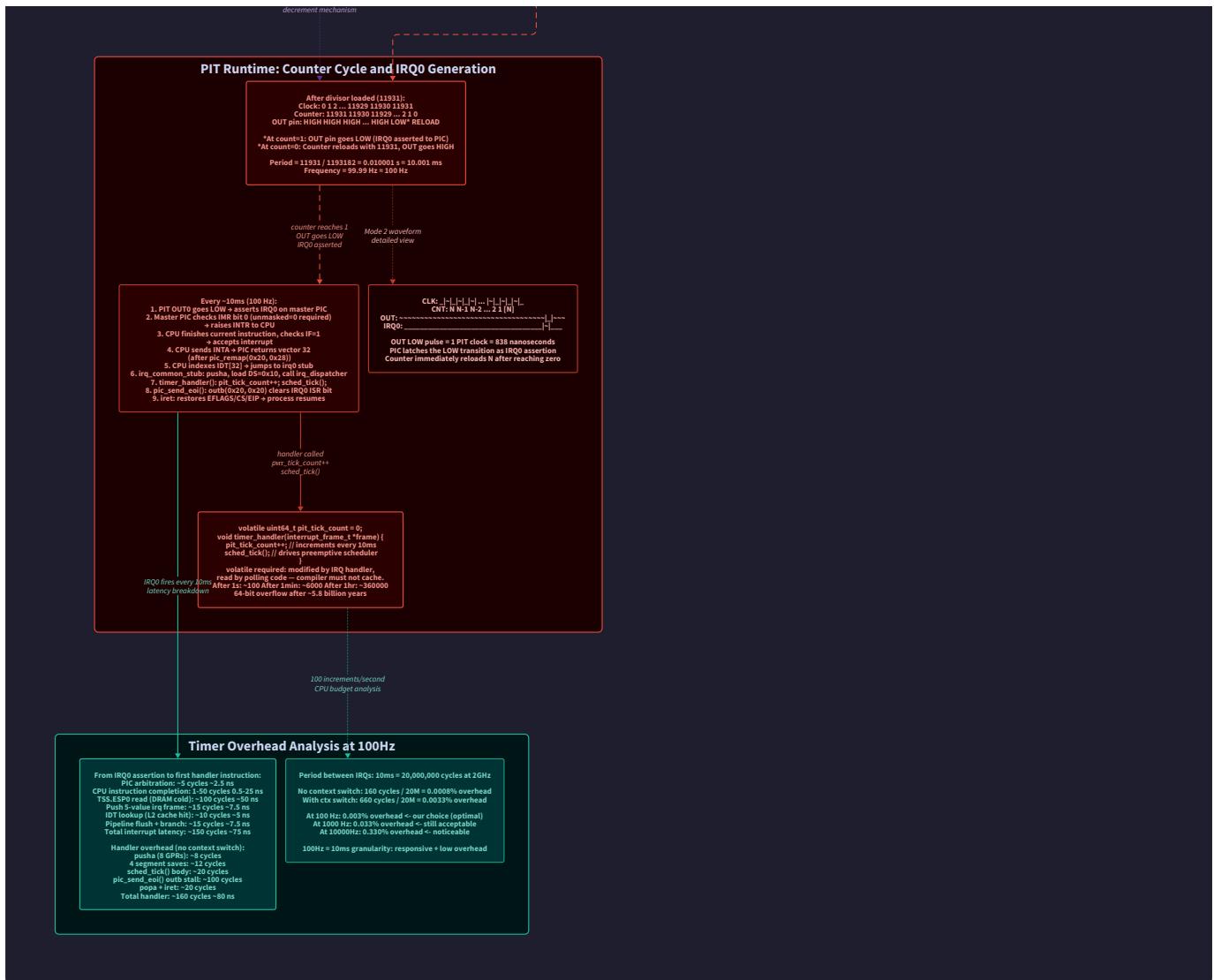
    idt_install();
}

```

Section 6: The PIT — Building a 100Hz Clock from a Counting Chip

With interrupts wired up and the PIC remapped, you need a source of periodic interrupts to drive future scheduling. The **PIT (Programmable Interval Timer — Intel 8253/8254)** is a chip with three independent 16-bit countdown counters clocked at **1,193,182 Hz** (a frequency inherited from television hardware standards — NTSC broadcast frequency divided down through several stages to land on this oddly specific number). Channel 0 is wired to IRQ0, making it the system timer.





The PIT works by loading a divisor into the counter. The counter decrements on each clock cycle. When it reaches zero, it triggers IRQ0 and reloads the divisor. The interrupt frequency is therefore:

IRQ0 frequency = 1,193,182 Hz ÷ divisor

For 100Hz (10ms per tick): divisor = 1,193,182 ÷ 100 = 11,931.82 → round to 11,932


```
/* pit.c - PIT channel 0 configuration */

#define PIT_CHANNEL0 0x40 /* Channel 0 data port */

#define PIT_CMD        0x43 /* Mode/command register */

#define PIT_FREQUENCY 1193182

volatile uint64_t pit_tick_count = 0; /* Incremented every IRQ0 */

void pit_set_frequency(uint32_t hz) {

    uint32_t divisor = PIT_FREQUENCY / hz;

    if (divisor > 0xFFFF) divisor = 0xFFFF; /* Clamp to 16-bit maximum */

    if (divisor < 1)      divisor = 1;       /* Divisor of 0 means 65536 */

    /* Command byte: channel 0, lo/hi byte access, mode 2 (rate generator), binary */

    /* 0x36 = 00 11 011 0:
       ^^ channel 0 (bits 7:6 = 00)
       ^^ lo/hi access (bits 5:4 = 11): send low byte, then high byte
       ^^^ mode 2 = rate generator (bits 3:1 = 011): continuous square wave
          ^ binary counting (bit 0 = 0) */

    outb(PIT_CMD, 0x36);

    outb(PIT_CHANNEL0, divisor & 0xFF); /* Low byte */

    outb(PIT_CHANNEL0, (divisor >> 8) & 0xFF); /* High byte */

}

void timer_handler(interrupt_frame_t *frame) {

    (void)frame; /* Unused for now; scheduler will use it in M4 */

    pit_tick_count++; /* Atomic on single-core; needs lock on SMP */

}

void pit_init(uint32_t hz) {

    pit_set_frequency(hz);

    irq_install_handler(0, timer_handler); /* IRQ0 → timer_handler */

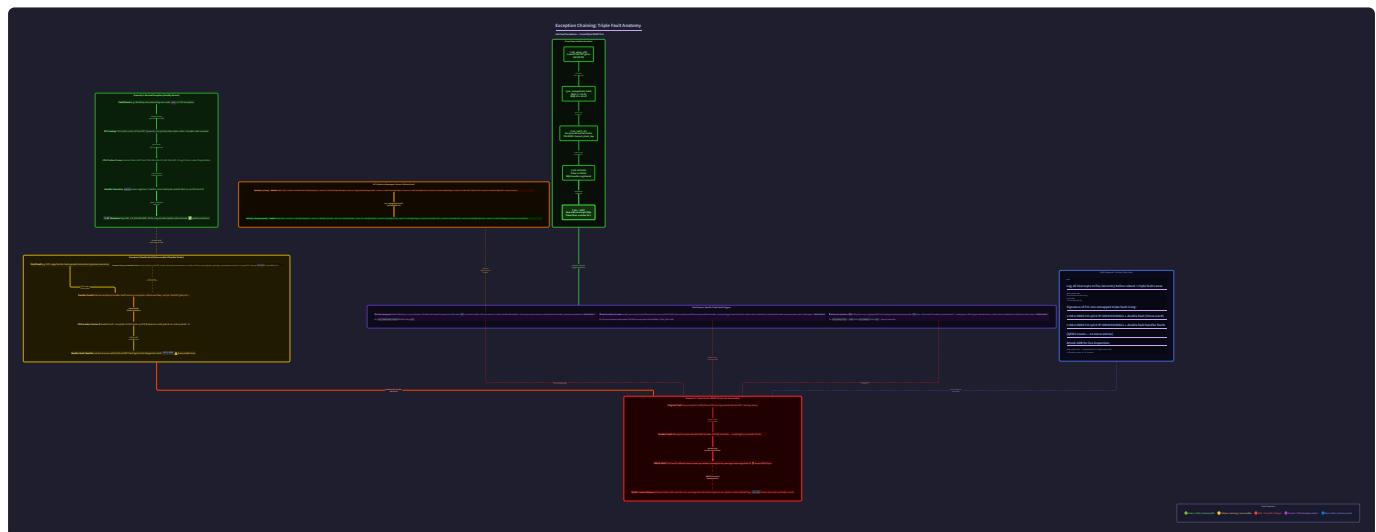
}
```

volatile uint64_t pit_tick_count: The `volatile` qualifier is required here for the same reason it was required for MMIO in Milestone 1 — without it, the compiler may cache the value in a register and never re-read from memory. Since `pit_tick_count` is modified by the interrupt handler and read by other code, the compiler cannot see this dependency (interrupt handlers are invisible to the optimizer's control flow analysis). `volatile` forces every access to go through memory. On a multi-core system, `volatile` is insufficient — you need atomic operations. For a single-core kernel, `volatile` is correct.

Hardware Soul: At 100Hz, the PIT fires every 10ms. From the CPU's perspective: the PIT asserts IRQ0 → the PIC samples it at the end of the current instruction → the CPU acknowledges, saves state (EFLAGS, CS, EIP pushed: 20 cycles), reads the IDT entry (1 memory access, likely L2 cache hit: 10 cycles), jumps to the stub, `pusha` (~8 cycles), C handler call (~15 cycles). Total interrupt latency from IRQ assertion to first instruction of handler: roughly 50–100 cycles, or 25–50 nanoseconds at 2GHz. This is negligible for a 10ms period. The interrupt overhead at 100Hz is approximately 50ns/10ms = 0.0005% CPU time.

Section 7: The Double Fault Handler — The Last Line of Defense

Exception 8 (double fault) deserves special treatment. A double fault occurs when the CPU tries to deliver an exception but encounters another exception while doing so — for instance, a GPF while trying to invoke the GPF handler because the handler's address in the IDT is invalid, or a stack fault while pushing the exception frame because the stack pointer is invalid.



The double fault handler is unique: **it always pushes an error code of zero** (even though the error code is always zero, the CPU pushes it, so your stub must use the `ISR_ERR` macro — it accounts for the CPU-pushed error code). If the double fault handler itself faults, you get a triple fault (unrecoverable reset). Therefore, the double fault handler must be bulletproof: no function calls, no stack-heavy operations, no paging assumptions.

```

/* In exception_handler in interrupt.c: special case for vector 8 */

if (frame->vector == 8) {

    /* Double fault: system is in an inconsistent state.

       Print what we can, then halt. Do not attempt complex operations. */

    kprintf("\n\n!!! DOUBLE FAULT !!!\n");

    kprintf("Error code: 0x%08x (always 0 for double fault)\n", frame->error_code);

    kprintf("EIP at time of original fault: 0x%08x\n", frame->eip);

    kprintf("System cannot recover. Halting.\n");

    /* Disable interrupts and halt all CPU activity */

    __asm__ volatile ("cli; hlt");

    for (;;) /* Should never reach here; satisfy compiler */

}

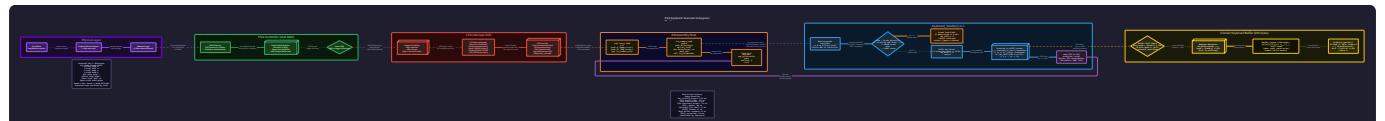
```

An advanced technique: the double fault TSS. Because a double fault may occur due to a corrupted stack (ESP is invalid), the normal ISR stack frame push will itself fault. A robust kernel uses a separate **Task State Segment** with its own stack for the double fault handler, configured in the IDT as a **task gate** rather than an interrupt gate. The CPU hardware switches to this separate stack before pushing the fault frame, guaranteeing the handler always has a valid stack. This is the technique Linux uses. For this milestone, a standard interrupt gate is sufficient — just ensure your kernel stack is healthy before enabling interrupts.

Section 8: The PS/2 Keyboard Driver — Make Codes, Break Codes, and the Scancode Table

With the interrupt framework in place, the keyboard driver is its first real application. The **PS/2 keyboard** (the classic DIN/mini-DIN connector, still emulated by USB HID on modern hardware) communicates via two I/O ports:

- **Port 0x60** (PS/2 Data Port): Read a byte to get the next scancode; write a byte to send a command to the keyboard controller.
- **Port 0x64** (PS/2 Status/Command Port): Read to get controller status; write to send controller commands.



When a key is pressed or released, the PS/2 controller generates IRQ1 and places a **scancode** in the data register at port **0x60**. Scancodes are not ASCII codes — they are hardware-level codes that represent physical key positions on the keyboard, independent of the key's character or the current modifier state. The three scancode sets have different code assignments; PS/2 keyboards default to **Set 1** on IBM-compatible PCs.

The critical behavior: every key generates **two events**:

- **Make code:** sent when the key is *pressed*. For most keys in Set 1, this is a single byte in the range **0x01 – 0x58**.

- **Break code:** sent when the key is *released*. This is the make code with bit 7 set (ORed with `0x80`). So the `A` key's make code is `0x1E`, and its break code is `0x9E`.

Some keys (extended keys like arrow keys, Numpad Enter, right Ctrl) send a two-byte sequence starting with `0xE0`, followed by the actual code. You can handle these later; for a first implementation, handle single-byte codes only.

```
/* keyboard.c */

#define KB_DATA_PORT    0x60
#define KB_STATUS_PORT  0x64
#define KB_BUFFER_SIZE  256

/* Scancode Set 1: index = make code, value = ASCII character
   0x00 = no mapping (shift, ctrl, extended, etc.) */

static const char scancode_to_ascii[128] = {

/*00*/  0,    27,   '1', '2', '3', '4', '5', '6', '7', '8', '9', '0', '-', '=', '\b',
/*0F*/  '\t', 'q', 'w', 'e', 'r', 't', 'y', 'u', 'i', 'o', 'p', '[', ']', '\n',
/*1D*/  0,    'a', 's', 'd', 'f', 'g', 'h', 'j', 'k', 'l', ';', '\'', ``',
/*2A*/  0,    '\\', 'z', 'x', 'c', 'v', 'b', 'n', 'm', ',', '.', '/', 0, '*',
/*38*/  0,    ' ', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
/*48*/  0,    0, 0, '-' , 0, 0, 0, '+', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
/*58*/  0,    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
/*68*/  0,    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

};

/* Circular keyboard input buffer */

static char    kb_buffer[KB_BUFFER_SIZE];
static uint8_t  kb_head = 0;    /* Next position to write */
static uint8_t  kb_tail = 0;    /* Next position to read */
static uint8_t  kb_shift = 0;   /* Non-zero if Shift is held */

/* Called from IRQ1 handler */

void keyboard_handler(interrupt_frame_t *frame) {
    (void)frame;

    uint8_t scancode = inb(KB_DATA_PORT);

    if (scancode & 0x80) {
        /* Break code (key release): clear make code, check for shift release */

        uint8_t make = scancode & 0x7F;

        if (make == 0x2A || make == 0x36) { /* Left or right Shift */
            kb_shift = 0;
        }
    }
}
```

```

/* Ignore other break codes – we do not need to act on key releases

for a simple single-key ASCII driver */

return;

}

/* Make code (key press) */

if (scancode == 0x2A || scancode == 0x36) { /* Left or right Shift */

kb_shift = 1;

return;

}

char c = scancode_to_ascii[scancode];

if (c == 0) return; /* Unmapped key (function keys, arrow keys, etc.) */

/* Apply shift: uppercase letters and shifted symbols */

if (kb_shift && c >= 'a' && c <= 'z') {

c = c - 'a' + 'A';

}

/* Write to circular buffer – discard if full */

uint8_t next_head = (kb_head + 1) % KB_BUFFER_SIZE;

if (next_head != kb_tail) {

kb_buffer[kb_head] = c;

kb_head = next_head;

}

}

/* Called from kernel code to read a character; returns 0 if buffer is empty */

char keyboard_getchar(void) {

if (kb_head == kb_tail) return 0; /* Buffer empty */

char c = kb_buffer[kb_tail];

kb_tail = (kb_tail + 1) % KB_BUFFER_SIZE;

return c;

}

```

```
void keyboard_init(void) {  
    irq_install_handler(1, keyboard_handler); /* IRQ1 → keyboard_handler */  
}
```

The circular buffer (ring buffer): `kb_buffer` is accessed by two "parties" — the interrupt handler writes to it, and kernel polling code reads from it. The head pointer (`kb_head`) advances on write; the tail pointer (`kb_tail`) advances on read. The buffer is full when `(head + 1) % SIZE == tail`; empty when `head == tail`. This structure avoids needing to shift bytes around — it is O(1) per operation. It is the same data structure used for kernel pipe buffers, network socket receive queues, and audio sample ring buffers. On a single-core system, the interrupt handler cannot be interrupted by `keyboard_getchar` (since interrupts are disabled in the IRQ handler), so no lock is needed. On SMP, you need an atomic compare-and-swap.

Section 9: Wiring It All Together

The complete initialization sequence, to be called from `kmain()` after the VGA and serial drivers:

```
void kmain(void) {  
    /* Milestone 1: output drivers */  
  
    serial_init();  
  
    vga_clear();  
  
    kprintf("Kernel booted. Setting up interrupt infrastructure...\n");  
  
    /* Milestone 2: interrupt infrastructure */  
  
    /* Step 1: Load the IDT with 256 entries (all pointing to handlers) */  
  
    idt_setup_all();  
  
    kprintf("[IDT] 256 entries loaded.\n");  
  
    /* Step 2: Remap PIC BEFORE enabling interrupts */  
  
    pic_remap(0x20, 0x28); /* IRQ0-7 → 32-39, IRQ8-15 → 40-47 */  
  
    kprintf("[PIC] Remapped: IRQ0-7 → vectors 32-39, IRQ8-15 → 40-47.\n");  
  
    /* Step 3: Configure PIT at 100Hz */  
  
    pit_init(100);  
  
    kprintf("[PIT] Timer configured at 100Hz (10ms per tick).\n");  
  
    /* Step 4: Initialize keyboard driver */  
  
    keyboard_init();  
  
    kprintf("[KBD] PS/2 keyboard driver ready.\n");  
  
    /* Step 5: Enable interrupts – only AFTER everything above is configured */  
  
    __asm__ volatile ("sti");  
  
    kprintf("[INT] Interrupts enabled.\n");  
  
    /* Spin and echo keyboard input */  
  
    kprintf("Type something: ");  
  
    while (1) {  
  
        char c = keyboard_getchar();  
  
        if (c) kprintf("%c", c);  
  
    }  
}
```

The ordering is not arbitrary:

1. IDT before PIC: prevents a window where an IRQ arrives before there is a valid handler in the IDT.
 2. PIC remapping before `sti`: absolutely mandatory to avoid the double-fault collision described at the start of this milestone.
 3. `sti` last: the atomic boundary. Nothing fires until you say so.
-

Section 10: The Three-Level View — What "Keyboard Press" Truly Means

Tracing a single keypress from your finger to your kernel's character buffer:

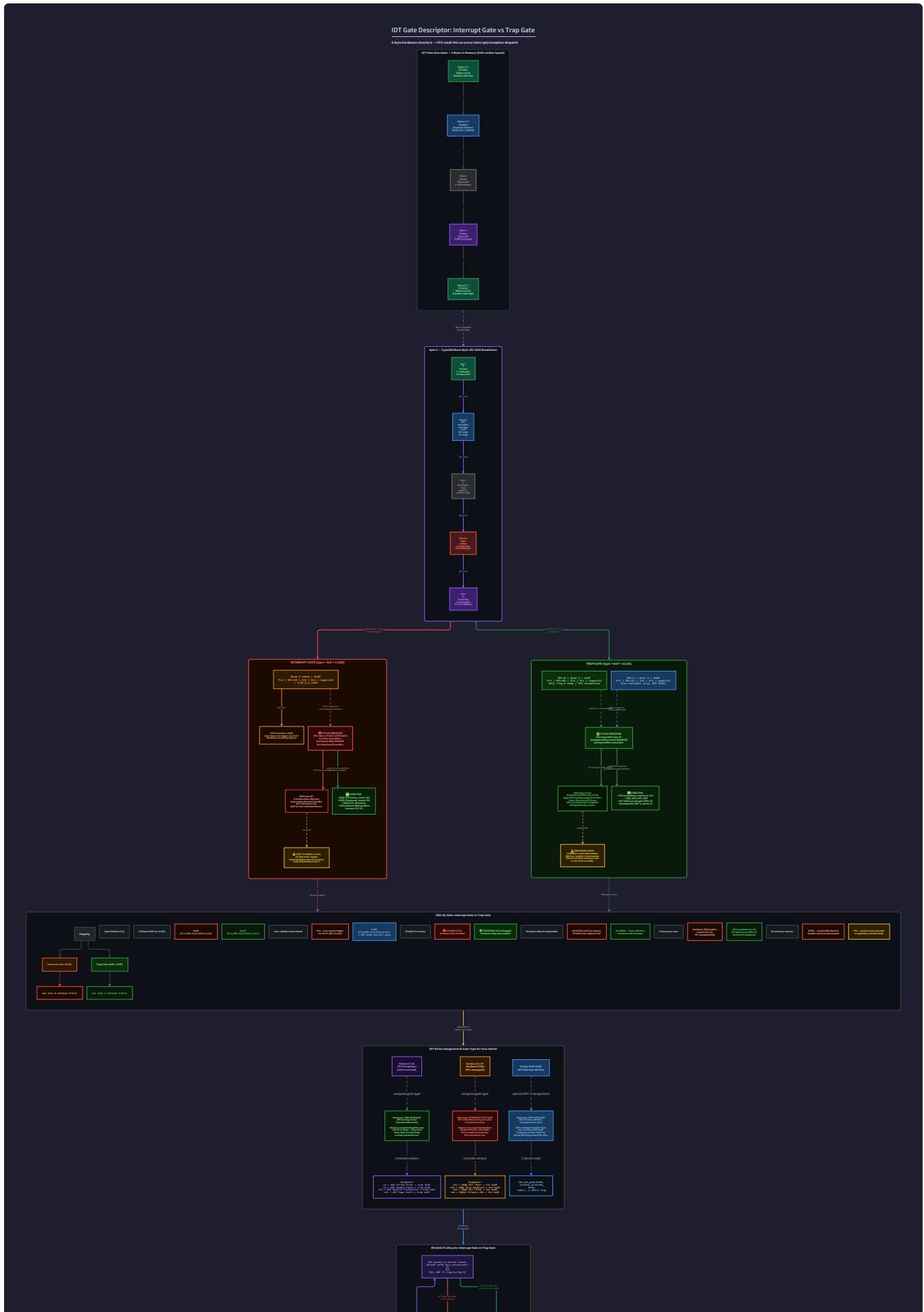
Level 1 — The Kernel API: Your code calls `keyboard_getchar()`, which reads from the circular buffer. If the buffer is non-empty, a character is returned immediately. From the caller's perspective: a character appeared, asynchronously, via some mechanism.

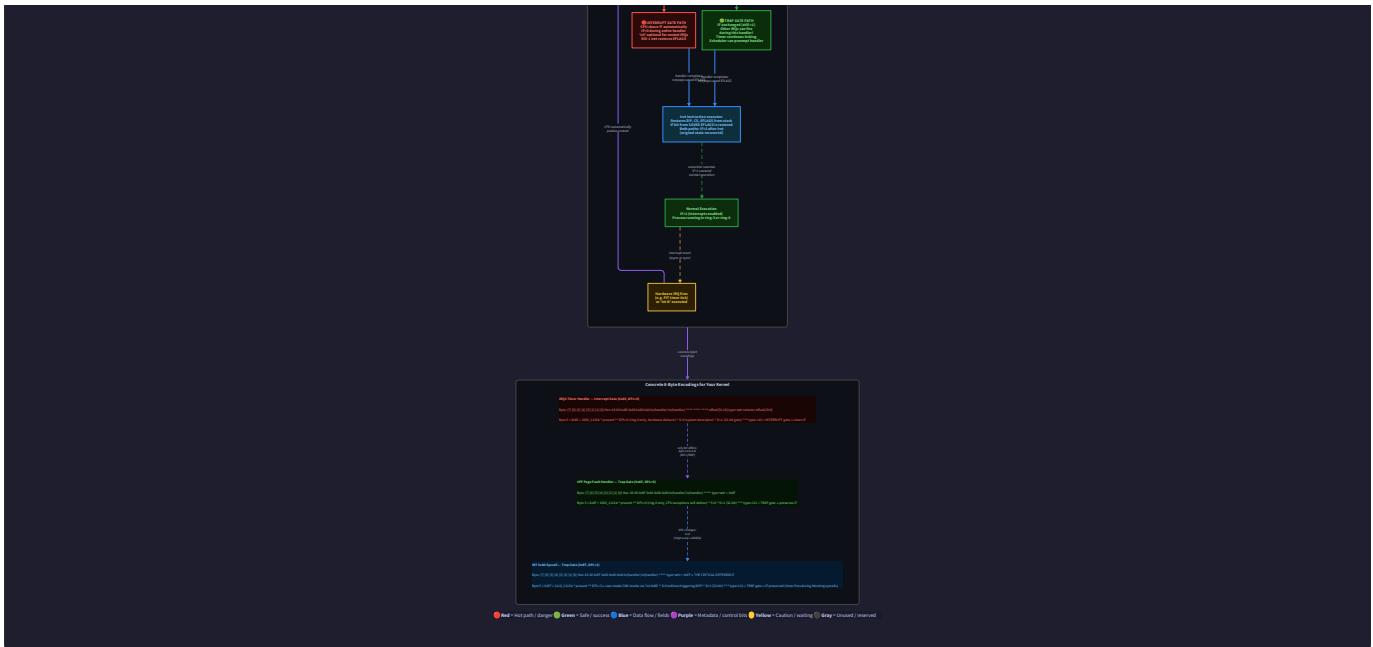
Level 2 — The Interrupt Path: The PS/2 keyboard controller (an Intel 8042-compatible chip) received a serial bit stream from the keyboard matrix, assembled it into a scancode byte, and asserted IRQ1. The PIC sampled IRQ1, determined it was not masked, and raised the `INTR` line to the CPU between instructions. The CPU finished the current instruction, saved EFLAGS/CS/EIP to the kernel stack, indexed into the IDT at vector 33 ($32 + \text{IRQ1}$), loaded the handler address, and jumped to `irq1` stub. The stub saved all registers (`pusha`), called `irq_dispatcher`, which called `keyboard_handler`, which read port `0x60` (consuming the scancode from the controller's buffer), translated it, and wrote to the circular buffer. Then EOI was sent to the master PIC, all registers were restored, and `iret` resumed the interrupted code.

Level 3 — Hardware Timing: The keyboard matrix scanner runs at roughly 1kHz, debounces key contacts (physical contacts bounce for 5–20ms when pressed), and serializes the scancode at 10–16kHz to the PS/2 controller. The PS/2 controller buffers the scancode and asserts IRQ1. The PIC's interrupt arbitration circuit checks priority (lower IRQ number = higher priority) and the master's IMR (Interrupt Mask Register) to decide whether to raise `INTR`. The CPU's interrupt recognition happens at instruction boundaries — some long instructions (like `rep movs`) check for interrupts between iterations. The total latency from key press to handler executing: 5–20ms of debounce + ~60µs of PS/2 serial transfer + ~1µs of PIC arbitration + ~50ns of CPU pipeline flush = effectively human-imperceptible.

Hardware Soul: Every `inb(0x60)` in `keyboard_handler` crosses the I/O bus — it is not cached, it stalls the CPU pipeline waiting for the I/O port to respond (typically 1–4µs for legacy I/O ports). On the 8259 PIC, the EOI write (`outb(PIC1_CMD, 0x20)`) similarly crosses the I/O bus. Total I/O bus accesses per keypress: 2 (read scancode + write EOI) \times ~2µs each \approx 4µs of pure I/O stall. Irrelevant at human typing speeds; significant if you were handling millions of IRQs per second (which is why modern NICs use MSI-X and interrupt coalescing rather than per-packet IRQs).

The Interrupt vs Trap Gate Distinction — A Design Decision With Consequences





The difference between an interrupt gate and a trap gate is a single bit in the IDT entry (`type` field: `0xE` vs `0xF`). But the behavioral consequence shapes every I/O subsystem in the kernel.

Interrupt gate (IF cleared on entry): When the CPU invokes an interrupt gate, it clears the Interrupt Flag. Hardware interrupts cannot arrive while your handler runs. This prevents timer interrupts from interrupting a timer handler (no reentrancy problem), and prevents keyboard IRQs from interrupting keyboard IRQ handling. The cost: if your handler is slow, you add latency to *all* other interrupts. No preemption occurs inside the handler.

Trap gate (IF not cleared on entry): Interrupts remain enabled. The handler can itself be interrupted by higher-priority hardware events. This is correct for CPU exceptions (a page fault handler might perform I/O that requires timer interrupts to complete), and it is essential for system calls — a `read()` syscall might block for seconds; you must allow other processes' timer interrupts to fire during that wait.

Gate Type	IF on Entry	Use Case	Risk
Interrupt Gate (<code>0x8E</code>)	Cleared (interrupts off)	Hardware IRQ handlers	Interrupt latency if handler is slow
Trap Gate (<code>0x8F</code> , DPL=0)	Not cleared (interrupts on)	CPU exception handlers	Reentrancy if not careful
Trap Gate (<code>0xEF</code> , DPL=3)	Not cleared (interrupts on)	System call gate (INT <code>0x80</code>)	User code can invoke via <code>int</code>

Linux uses interrupt gates for hardware IRQs (to prevent handler reentrancy) and trap gates for system calls (so timer interrupts still fire during blocking syscalls). The specific gate type for the system call vector (`0x80`) has DPL=3, allowing user-mode code to invoke it with `int 0x80` without triggering a General Protection Fault. You will wire this up in Milestone 4.

Debugging Interrupt Problems

When interrupt code goes wrong, the failure modes are particularly opaque. A taxonomy of what you will encounter:

Symptom	Diagnosis	Fix
Immediate triple fault after <code>sti</code>	PIC not remapped; first timer tick hits double fault vector	Call <code>pic_remap()</code> before <code>sti</code>
System hangs silently after first IRQ	Forgot <code>pic_send_eoi()</code> ; PIC masked that IRQ line forever	Add EOI at end of every IRQ handler
Triple fault at <code>iret</code>	Stack misaligned: forgot to <code>add esp, 8</code> after <code>pusha /handler/popa</code>	Count every push and pop; verify stack symmetry
Keyboard generates characters twice	Not ignoring break codes; handler called on both press and release	Check <code>scancode & 0x80</code> for break codes
Exception handler corrupts caller registers	Missing <code>pusha / popa</code> pair	Every ISR stub must save and restore all registers
<code>pit_tick_count</code> always zero in polling code	Missing <code>volatile</code> ; compiler cached value in register	Add <code>volatile</code> to the tick count variable
QEMU reboots with no output	Triple fault before serial is initialized; PIC collision	Test with <code>cli</code> permanently; confirm IDT setup before <code>sti</code>

Use QEMU's `-d int` flag during development. It logs every interrupt and exception to stderr with the vector number, error code, and EIP. Combined with serial output, this tells you exactly which instruction triggered each exception.

```
qemu-system-i386 -drive format=raw,file=os.img -serial stdio -d int 2>interrupts.log
```

BASH

The `interrupts.log` will show something like:

```
2: v=08 e=0000 i=0 cpl=0 IP=0008:001000a4 pc=001000a4 SP=0010:001fffff8
```

`v=08` = vector 8 (double fault). `IP=0008:001000a4` = the instruction that caused it. This line, immediately after `sti`, is the unmistakable signature of a non-remapped PIC.

Knowledge Cascade — What This Milestone Unlocks

You have just built the interrupt infrastructure that every operating system feature above this layer depends on. Here is the web of concepts it connects to.

1. Interrupt gate vs trap gate → Linux system call design and seccomp The exact reason `int 0x80` in Linux uses a trap gate (type `0xEF`, DPL=3) rather than an interrupt gate (type `0x8E`) is so that hardware interrupts continue to fire during system call execution. A `read()` blocking on disk I/O can wait for seconds; the timer interrupt must still preempt other processes during that wait. If system calls used interrupt gates, the timer would be masked for the duration of any blocking syscall, making preemptive scheduling impossible. **seccomp** (Secure Computing Mode — the Linux mechanism for restricting which syscalls a process can make, used by Docker, Chrome, and Firefox sandbox) works by inserting a filter into the system call dispatch path at exactly the point between the IDT trap gate and the syscall table lookup. Understanding the gate type is understanding where seccomp hooks in.

2. EOI protocol → interrupt coalescing in network drivers (NAPI) The mandatory EOI acknowledgment after each IRQ is the conceptual ancestor of a critical Linux networking optimization: **NAPI (New API)**. Without NAPI, every incoming network packet triggers an IRQ, the handler processes one packet, sends EOI, and the next packet immediately triggers another IRQ. At 10Gbps line rates, this produces millions of IRQs per second — the kernel spends more time in interrupt overhead than actual packet processing. NAPI's solution: on the first IRQ for a NIC, disable that NIC's IRQ line (like never sending EOI) and switch to polling mode — the

kernel polls the NIC's receive queue in a tight loop until it empties, then re-enables the IRQ. This is interrupt coalescing: batch many events into one interrupt cycle. Modern NIC firmware does this autonomously via **interrupt moderation**, accumulating packets for up to 50–100µs before asserting the IRQ. The mandatory acknowledgment you just implemented is the primitive they are optimizing around.

3. ISR stack frame layout → Linux signal frames and `sigreturn` The stack frame your ISR creates (EFLAGS, CS, EIP, plus the general-purpose registers you pushed) is structurally identical to the **signal frame** that Linux pushes onto a user process's stack before invoking a signal handler. When you call `signal(SIGSEGV, handler)`, Linux's signal delivery mechanism saves all the process registers — exactly like `pusha` — plus the CPU's automatic EFLAGS/CS/EIP push, creates a `sigcontext` struct on the user stack, and jumps to your handler. When the handler returns, it calls `sigreturn` (a dedicated syscall), which reads that struct back from the stack and restores all the registers — exactly like your `popa + iret` sequence. The **Spectre/Meltdown mitigations** that Intel deployed in 2018 broke the `sigreturn` mechanism on some configurations by changing kernel/user address space isolation, and Linux had to rewrite the signal frame layout. You now understand exactly what broke.

4. PIT 100Hz tick → high-resolution timers, NTP, and `clock_gettime` complexity Your 100Hz tick counter has 10ms resolution — you cannot measure intervals shorter than one tick. Early Linux (2.4 kernel) used 100Hz for the same reason. The problems: 10ms is too coarse for audio (introduces jitter), too coarse for network latency measurement, and wastes power on laptops (the CPU must wake up 100 times per second even when idle). The **HPET (High Precision Event Timer)** was introduced to provide nanosecond resolution without high interrupt rates. The **TSC deadline timer** (available on modern CPUs) allows the OS to program a one-shot interrupt at a specific CPU cycle count, enabling **tickless kernels** (Linux NO_HZ mode) where the timer only fires when actually needed. `clock_gettime(CLOCK_MONOTONIC)` on modern Linux reads the TSC directly in userspace (via the vDSO — a kernel-mapped page containing code that bypasses the syscall overhead entirely), achieving nanosecond precision without any interrupt overhead. Your 100Hz PIT is the starting point of this 40-year evolution.

5. PS/2 make/break codes → USB HID protocol and N-key rollover Every keypress generating two events (make and release) is not a PS/2 quirk — it is a fundamental input protocol design principle. **USB HID (Human Interface Device)** reports work the same way: a keyboard sends a report when any key state changes, containing a bitmask of all currently pressed keys. The HID protocol's **key rollover** specification defines how many simultaneously pressed keys the device can distinguish. **N-key rollover (NKRO)** means every key has independent state tracking — 104 keys pressed simultaneously are all reported. Budget keyboards implement 6-key rollover (6KRO), losing track of 7th+ simultaneous keys. Gaming keyboards advertise NKRO as a feature. This complexity exists because the USB HID report format must encode the exact state of every key in every report, requiring bitmask representation rather than PS/2's sequential event stream. The make/break model you just implemented is USB HID's ancestor.

6. The double fault handler → hardware watchdog timers and kernel oops Your double fault handler — print diagnostic info, halt — is a simplified version of what Linux's kernel panic handler does. Linux's **kernel oops** mechanism prints a backtrace, register dump, and loaded module list on any unhandled kernel exception, then either halts or continues (depending on `oops=panic` configuration). The **NMI (Non-Maskable Interrupt — vector 2)** is a special interrupt that the CPU processes even when `IF=0` (it bypasses the interrupt flag entirely). Production servers wire the **hardware watchdog timer** to the NMI line — if the kernel freezes and stops responding, the watchdog fires an NMI, which the kernel handles by printing a backtrace and (optionally) triggering a machine reset. Google's servers use NMI-based watchdogs to automatically recover from kernel deadlocks. Your double fault handler is the conceptual prototype.

Milestone 3: Physical and Virtual Memory Management

The Revelation: Paging Enables Itself — Or Kills You Trying

Here is what every programmer who has worked above the OS layer believes about virtual memory: it is a feature the operating system turns on during boot, and once on, programs can use any address the OS has allocated for them. The OS manages page tables in software; the CPU consults them when needed. Virtual memory feels like a software abstraction layered onto hardware.

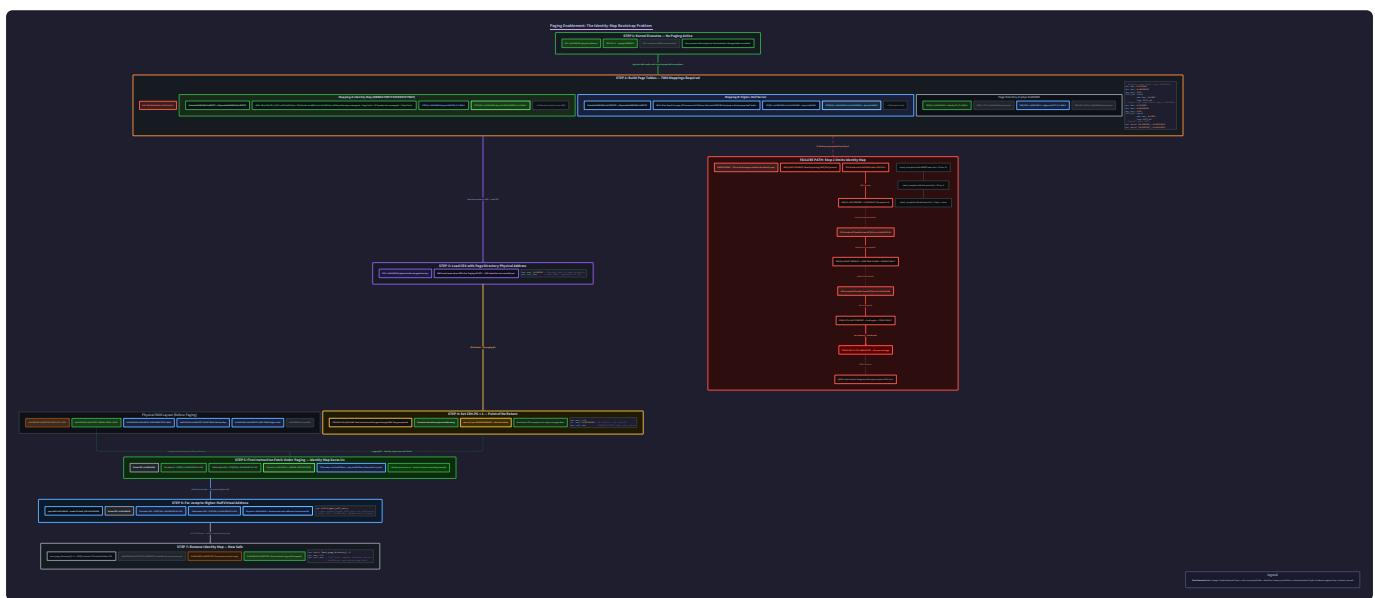
This belief is correct about the steady state. It is catastrophically wrong about the transition.

When you write `CR0.PG = 1` to enable paging, the change is instantaneous and irrevocable — there is no "paging enabled for the next instruction" mode. The *current* instruction has already been fetched; the *next* instruction fetch immediately goes through the new page table lookup. If the physical address of that next instruction is not mapped in your page tables, the CPU raises a page fault (exception 14). But your page fault handler is code. Which means the CPU tries to look up the page fault handler's address through the page tables. Which is also not mapped. Which raises another page fault. Double fault. Triple fault. Reboot. No error message.

There is no way to print a diagnostic before this happens, because your print function is also code that lives at unmapped addresses.

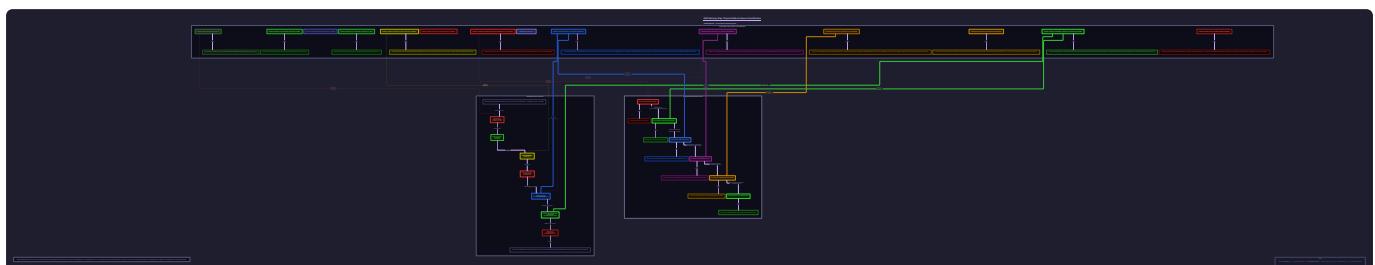
The solution is to map the physical addresses where your code currently lives *before* setting `CR0.PG`. You must build a set of page tables that describe the current physical layout of memory, enable paging while that layout is valid, and *then* remap things as needed. This technique is called **identity mapping** — virtual address equals physical address for the bootstrapping region. The moment paging activates, the CPU finds your code at its expected address because you told it to.

The second revelation arrives when you later modify a page table entry. The CPU has a cache of recent address translations called the **TLB (Translation Lookaside Buffer)**. If you modify a PTE and do not flush the TLB, the CPU keeps serving the old translation from cache — potentially letting code access a physical frame you just freed and reassigned to a different purpose. This is not a theoretical bug. On a single core, it causes silent memory corruption that surfaces far from the modification. On a multi-core system — not covered in this milestone but described in the cascade below — it requires inter-processor interrupts to flush remote TLBs, because each core has its own private TLB. The physics of caching creates distributed consistency problems at the scale of a single chip.



Section 1: The Physical Address Space — What RAM Actually Looks Like

Before you can manage memory, you need to know what memory exists. This sounds like a trivial question — you have 256MB of RAM, so you have 256MB. But the physical address space is not a clean flat array of usable bytes. It is a patchwork of RAM, firmware ROM, device registers, and reserved regions, with holes punched through it by hardware that was never designed to be coherent.



The physical address space of a typical x86 machine looks like this:

Physical Address	Region	Notes
0x00000000	Real-mode IVT	Interrupt Vector Table (legacy)
0x00000400	BIOS Data Area	BIOS-maintained hardware state
0x00007C00	MBR load address	Where the bootloader runs
0x00080000	Extended BIOS Data Area (EBDA)	Variable size, avoid
0x000A0000	VGA framebuffer (CGA/EGA)	Legacy video memory
0x000B8000	VGA text mode buffer	Your kprintf target
0x000C0000	Video BIOS ROM	Option ROM space
0x000F0000	System BIOS ROM	Firmware code, not RAM
0x00100000 (1MB)	Usable RAM begins	Where your kernel loads
...	More usable RAM	
0xFEC00000	I/O APIC	Advanced interrupt controller
0xFEE00000	Local APIC	Per-CPU interrupt controller
0xFFFF0000	BIOS ROM mirror	Reset vector lives here

Your kernel cannot deduce this layout from first principles. The firmware knows it — the BIOS probed the hardware during POST. Your bootloader must ask the BIOS to report what it found.

The E820 Memory Map — Asking the Firmware

E820 is a BIOS function (INT 15h, EAX=0xE820) that returns a list of memory region descriptors. Each descriptor specifies a base address, a length, and a type. This call must be made in real mode, which means your Stage 2 bootloader or GRUB must perform it and store the results in a known location before entering protected mode.

Why E820 and not just reading a configuration byte? Older methods (INT 15h, AX=0xE801 for extended memory; INT 15h, AH=0x88 for simple memory size) returned a single number representing contiguous memory above 1MB. They could not express non-contiguous regions, holes for ACPI tables, or memory-mapped device regions. E820 was introduced to replace all of them. It is still used by modern UEFI firmware in legacy compatibility mode, and GRUB passes the E820 data to the kernel via the **multiboot information structure**.

Each E820 entry is a 20-byte structure:

```
typedef struct {  
    uint64_t base;      /* Physical base address of region */  
  
    uint64_t length;    /* Length in bytes */  
  
    uint32_t type;     /* Region type */  
} __attribute__((packed)) e820_entry_t;  
  
/* Type values */  
  
#define E820_USABLE      1 /* Free RAM – your allocator can use this */  
  
#define E820_RESERVED    2 /* BIOS-reserved; do not touch */  
  
#define E820_ACPI_DATA   3 /* ACPI tables; can reclaim after reading */  
  
#define E820_ACPI_NVS    4 /* ACPI Non-Volatile Storage; never reclaim */  
  
#define E820_BAD         5 /* Known bad RAM */
```

If you are using **GRUB** (the most common approach for learning kernels), GRUB performs the E820 query for you and passes the memory map via the **multiboot information structure**. Your kernel entry point receives a pointer to this structure in the `EBX` register (set by GRUB before jumping to your kernel). The multiboot spec defines a `flags` field indicating which information is present, and a `mmap_addr / mmap_length` pair pointing to the memory map:

```
/* multiboot.h - simplified multiboot information structure */

typedef struct {

    uint32_t flags;          /* Flags indicating which fields are valid */

    uint32_t mem_lower;      /* Kilobytes of lower memory (below 1MB) */

    uint32_t mem_upper;      /* Kilobytes of upper memory (above 1MB) */

    /* ... many other fields ... */

    uint32_t mmap_length;    /* Length of memory map buffer in bytes */

    uint32_t mmap_addr;      /* Physical address of first mmap_entry_t */

} multiboot_info_t;

/* Each entry in the multiboot memory map */

typedef struct {

    uint32_t size;           /* Size of this entry (not including this field itself) */

    uint64_t addr;           /* Physical base address */

    uint64_t len;            /* Length in bytes */

    uint32_t type;           /* E820 type: 1=usable, 2=reserved, 3=ACPI, etc. */

} __attribute__((packed)) mmap_entry_t;
```

The `size` field quirk: Each multiboot mmap entry begins with a `size` field that gives the size of the *remaining* fields. The next entry is at `(uint8_t*)entry + entry->size + 4` — you add 4 to account for the `size` field itself, which is not counted. This non-obvious arithmetic trips up nearly every developer who iterates the memory map for the first time.

Parsing the multiboot memory map:

```
/* pmm.c - Physical Memory Manager */

#include "multiboot.h"

#include "pmm.h"

#include "kprintf.h"

static uint64_t total_memory_bytes = 0;

static uint64_t usable_memory_bytes = 0;

void pmm_parse_memory_map(multiboot_info_t *mbi) {

    /* Verify the memory map flag is set (bit 6) */

    if (!(mbi->flags & (1 << 6))) {

        kprintf("[PMM] FATAL: Bootloader did not provide memory map.\n");

        for (;;);

    }

    kprintf("[PMM] Memory map (from bootloader):\n");

    mmap_entry_t *entry = (mmap_entry_t *)(uintptr_t)mbi->mmap_addr;

    mmap_entry_t *end = (mmap_entry_t *)(uintptr_t)(mbi->mmap_addr + mbi->mmap_length);

    while (entry < end) {

        const char *type_str;

        switch (entry->type) {

            case 1: type_str = "Usable RAM";    usable_memory_bytes += entry->len; break;

            case 2: type_str = "Reserved";      break;

            case 3: type_str = "ACPI Data";    break;

            case 4: type_str = "ACPI NVS";     break;

            case 5: type_str = "Bad RAM";      break;

            default: type_str = "Unknown";     break;

        }

        total_memory_bytes += entry->len;

        kprintf(" [0x%08x%08x - 0x%08x%08x] %s\n",
               (uint32_t)(entry->addr >> 32), (uint32_t)entry->addr,
               (uint32_t)((entry->addr + entry->len - 1) >> 32),
               (uint32_t)(entry->addr + entry->len - 1),
               type_str);
    }
}
```

```

        type_str);

/* Advance to next entry: size field + size of remaining fields */

entry = (mmap_entry_t *)((uint8_t *)entry + entry->size + 4);

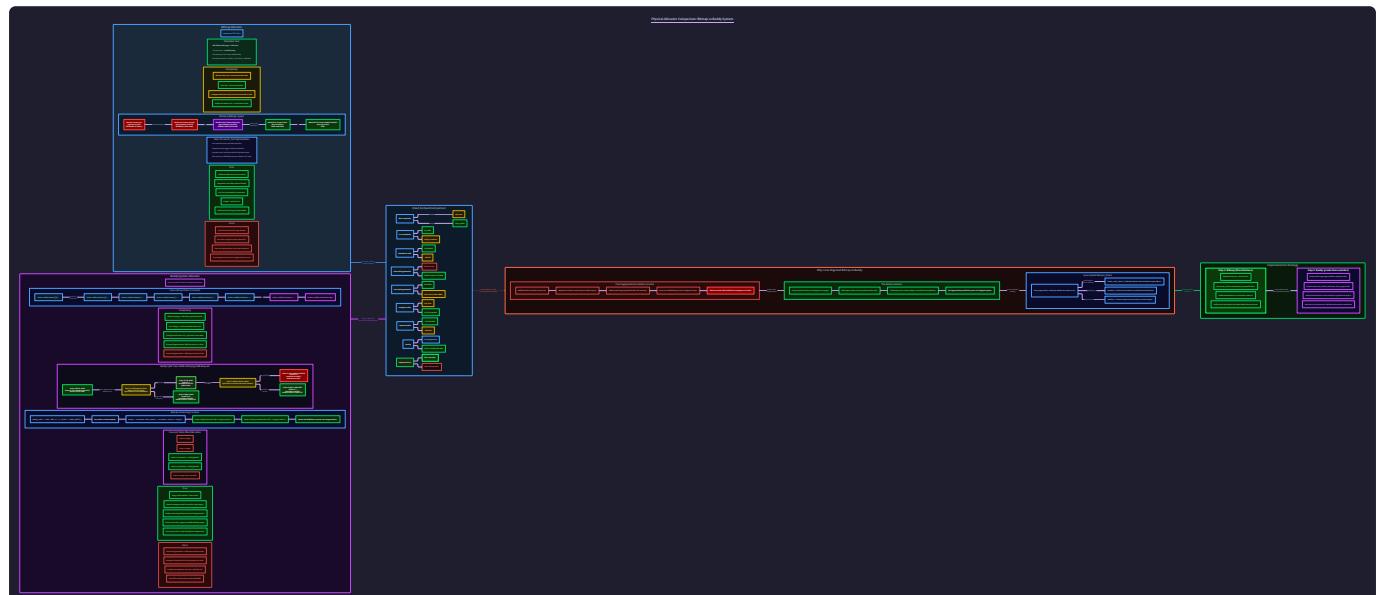
}

kprintf("[PMM] Total: %u MB, Usable: %u MB\n",
        (uint32_t)(total_memory_bytes / (1024*1024)),
        (uint32_t)(usable_memory_bytes / (1024*1024)));
}

```

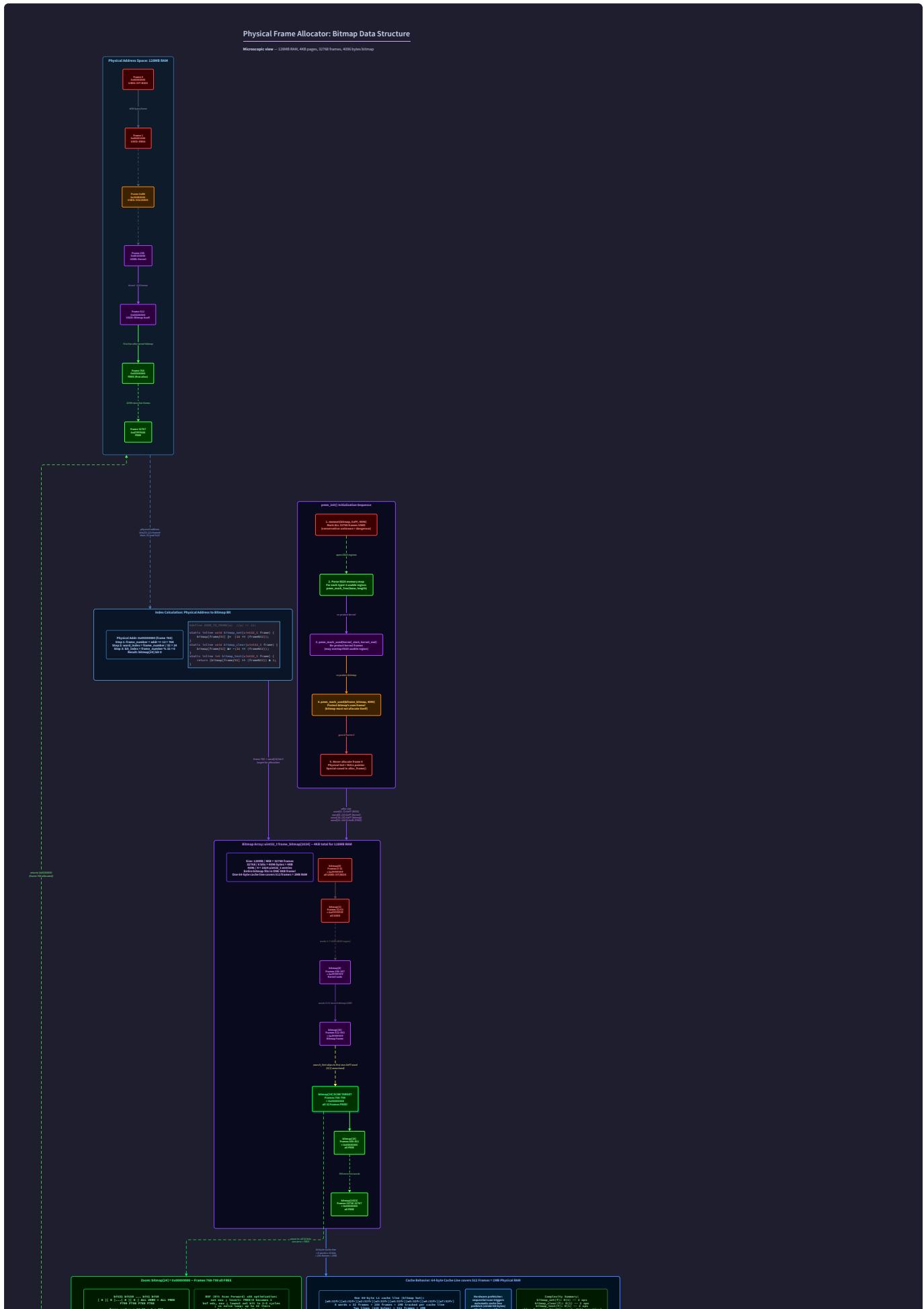
Section 2: The Bitmap Physical Frame Allocator

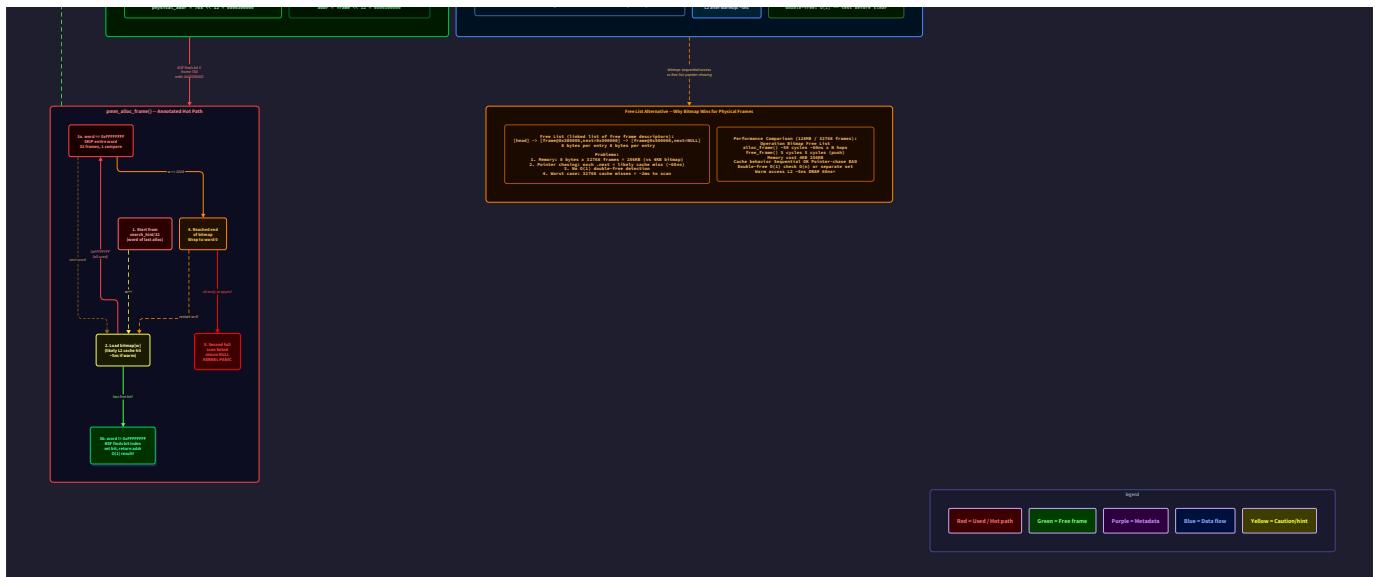
With the memory map parsed, you need a data structure to track which physical 4KB frames are free and which are in use. The two main choices are a **bitmap** (one bit per frame, 0=free, 1=used) and a **free list** (linked list of free frame descriptors). Let us examine both to understand why bitmaps are the right default for a learning kernel.



Property	Bitmap	Free List
Memory overhead for 4GB RAM	128 KB (1 bit × 1M frames)	4–8 MB (pointer per free frame)
Allocate single frame	O(n) scan, O(1) amortized with hint	O(1) pop from head
Free a frame	O(1) clear bit	O(1) push to head
Cache behavior	64-byte cache line covers 512 frames	Pointer-chasing: each access is potentially a cache miss
Detect double-free	O(1) check if bit already 0	Requires O(n) scan or set membership check
Contiguous multi-frame allocation	O(n) scan for consecutive 0-bits	Buddy allocator variant required

For a kernel that allocates frames one at a time (for page tables and backing memory), the bitmap's cache behavior advantage is decisive. The bitmap for 4GB of RAM at 4KB frames is $4\text{GB} / 4\text{KB} / 8 \text{ bits} = 128\text{KB}$. At L2 cache size of 256KB–1MB, the entire bitmap fits in cache after the first scan. Subsequent allocations that land near the previous allocation hit the same cache line — 512 frames represented in one 64-byte cache line. Free lists, by contrast, follow pointers to arbitrary physical addresses, generating a cache miss per allocation.





Buddy allocator: jemalloc, mimalloc, and the Linux kernel use a **buddy allocator** for physical frame management — a hierarchical structure where memory is divided into power-of-two-sized blocks, and adjacent ("buddy") free blocks are merged on deallocation. Buddy allocators eliminate external fragmentation for large allocations, support contiguous multi-frame allocation efficiently, and have $O(\log n)$ alloc/free. They are more complex to implement correctly. For this milestone, the bitmap allocator is the right choice — simple, cache-friendly, and sufficient for single-frame allocation. The cascade section connects this back to jemalloc's design.

Bitmap Allocator Implementation

```
/* pmm.h */

#ifndef PMM_H

#define PMM_H

#include <stdint.h>
#include <stddef.h>

#define PAGE_SIZE    4096
#define PAGE_SHIFT   12          /* log2(PAGE_SIZE) */
#define FRAME_COUNT  (4UL * 1024 * 1024 * 1024 / PAGE_SIZE) /* 4GB / 4KB = 1M frames */

/* Physical address → frame number conversion */

#define ADDR_TO_FRAME(addr) ((addr) >> PAGE_SHIFT)
#define FRAME_TO_ADDR(frame) ((frame) << PAGE_SHIFT)

void pmm_init(multiboot_info_t *mbi);
void pmm_mark_used(uint64_t addr, uint64_t length);
void pmm_mark_free(uint64_t addr, uint64_t length);
void *pmm_alloc_frame(void);      /* Returns physical address; NULL on failure */
void pmm_free_frame(void *addr); /* Frees a frame by physical address */
uint32_t pmm_frames_used(void);
uint32_t pmm_frames_free(void);

#endif
```

```
/* pmm.c - Bitmap physical frame allocator */

#include "pmm.h"

#include "multiboot.h"

#include "kprintf.h"

#include <string.h> /* memset - provided by your freestanding implementation */

/* The bitmap: 1 bit per 4KB frame, covering 4GB → 128KB of bitmap */

/* We declare it as uint32_t array; each element covers 32 frames */

#define BITMAP_SIZE (FRAME_COUNT / 32) /* 32768 uint32_t entries */

static uint32_t frame_bitmap[BITMAP_SIZE];

static uint32_t total_frames = 0;

static uint32_t used_frames = 0;

static uint32_t search_hint = 0; /* Last allocation position - scan from here */

/* Bit manipulation helpers */

static inline void bitmap_set(uint32_t frame) {

    frame_bitmap[frame / 32] |= (1U << (frame % 32));
}

static inline void bitmap_clear(uint32_t frame) {

    frame_bitmap[frame / 32] &= ~(1U << (frame % 32));
}

static inline int bitmap_test(uint32_t frame) {

    return (frame_bitmap[frame / 32] >> (frame % 32)) & 1;
}

void pmm_init(multiboot_info_t *mbi) {

    /* Step 1: Mark everything as used (conservative starting state) */

    memset(frame_bitmap, 0xFF, sizeof(frame_bitmap));

    used_frames = FRAME_COUNT;

    /* Step 2: Parse memory map; mark usable regions as free */

    mmap_entry_t *entry = (mmap_entry_t *)(&uintptr_t)mbi->mmap_addr;

    mmap_entry_t *end = (mmap_entry_t *)(&uintptr_t)(mbi->mmap_addr + mbi->mmap_length);
```

C

```

while (entry < end) {

    if (entry->type == 1) { /* E820_USABLE */

        /* Align base up and length down to page boundaries */

        uint64_t base    = (entry->addr + PAGE_SIZE - 1) & ~(uint64_t)(PAGE_SIZE - 1);
        uint64_t length  = (entry->addr + entry->len - base) & ~(uint64_t)(PAGE_SIZE - 1);

        if (length > 0 && base < (uint64_t)FRAME_COUNT * PAGE_SIZE) {

            pmm_mark_free(base, length);

            total_frames += (uint32_t)(length / PAGE_SIZE);

        }

    }

    entry = (mmap_entry_t *)((uint8_t *)entry + entry->size + 4);

}

/* Step 3: Mark kernel itself as used so it cannot be allocated away */

extern uint32_t __kernel_start_phys, __kernel_end_phys; /* Linker symbols */

uint64_t k_start = (uint64_t)(uintptr_t)&__kernel_start_phys;
uint64_t k_end   = (uint64_t)(uintptr_t)&__kernel_end_phys;

pmm_mark_used(k_start, k_end - k_start);

/* Step 4: Mark the bitmap itself as used (it lives in BSS, covered by kernel range)

If bitmap is outside kernel range due to linker layout, mark it explicitly */

pmm_mark_used((uint64_t)(uintptr_t)frame_bitmap, sizeof(frame_bitmap));

kprintf("[PMM] Initialized: %u MB usable, %u MB reserved.\n",
       pmm_frames_free() * PAGE_SIZE / (1024*1024),
       pmm_frames_used() * PAGE_SIZE / (1024*1024));
}

void pmm_mark_used(uint64_t addr, uint64_t length) {

    uint32_t frame_start = (uint32_t)ADDR_TO_FRAME(addr);
    uint32_t frame_count = (uint32_t)(length / PAGE_SIZE);

    for (uint32_t i = 0; i < frame_count && (frame_start + i) < FRAME_COUNT; i++) {

        if (!bitmap_test(frame_start + i)) {

            bitmap_set(frame_start + i);

    }
}

```

```

        used_frames++;

    }

}

}

void pmm_mark_free(uint64_t addr, uint64_t length) {

    uint32_t frame_start = (uint32_t)ADDR_TO_FRAME(addr);

    uint32_t frame_count = (uint32_t)(length / PAGE_SIZE);

    for (uint32_t i = 0; i < frame_count && (frame_start + i) < FRAME_COUNT; i++) {

        if (bitmap_test(frame_start + i)) {

            bitmap_clear(frame_start + i);

            if (used_frames > 0) used_frames--;

        }

    }

}

void *pmm_alloc_frame(void) {

    /* Scan from search_hint to find a free frame */

    for (uint32_t w = search_hint / 32; w < BITMAP_SIZE; w++) {

        if (frame_bitmap[w] == 0xFFFFFFFF) continue; /* All 32 frames used; skip */

        /* At least one free frame in this word */

        for (uint32_t b = 0; b < 32; b++) {

            uint32_t frame = w * 32 + b;

            if (!bitmap_test(frame)) {

                bitmap_set(frame);

                used_frames++;

                search_hint = frame; /* Next search starts near here */

                /* Never allocate frame 0 – physical 0x0 is the NULL address */

                if (frame == 0) { bitmap_clear(0); used_frames--; continue; }

                return (void *)(uintptr_t)FRAME_TO_ADDR(frame);

            }

        }

    }

}

```

```
}

/* Wrap around and search from beginning */

if (search_hint > 0) {

    search_hint = 0;

    return pmm_alloc_frame();

}

kprintf("[PMM] FATAL: Out of physical memory!\n");

return NULL; /* Out of memory */

}

void pmm_free_frame(void *addr) {

    uint32_t frame = (uint32_t)ADDR_TO_FRAME((uintptr_t)addr);

    /* Validate: within managed range */

    if (frame >= FRAME_COUNT) {

        kprintf("[PMM] WARNING: pmm_free_frame() called with out-of-range address 0x%x\n", addr);

        return;

    }

    /* Double-free detection */

    if (!bitmap_test(frame)) {

        kprintf("[PMM] BUG: Double-free detected for frame 0x%u (addr 0x%u)!\n", frame, addr);

        return; /* In a production kernel, you might panic here */

    }

    bitmap_clear(frame);

    if (used_frames > 0) used_frames--;

    /* Update hint if this freed frame is earlier than current position */

    if (frame < search_hint) search_hint = frame;

}

uint32_t pmm_frames_used(void) { return used_frames; }

uint32_t pmm_frames_free(void) { return total_frames - used_frames; }
```

The search_hint optimization: Without it, every `pmm_alloc_frame()` call scans from frame 0. After the first 100 frames are allocated, each call wastes time re-scanning the occupied bitmap words. The hint records the approximate position of the last successful allocation. Since allocations tend to be sequential (kernel heap grows from low to high), the hint usually lands on or near a free frame. This is a **next-fit** allocation policy — as opposed to **first-fit** (always scan from 0) or **best-fit** (find the smallest free block that fits). For single-frame allocation, next-fit is both faster and more cache-friendly: subsequent allocations modify the same or adjacent bitmap words, staying in the same cache line.

Hardware Soul: The bitmap scan is essentially a sequential memory read — exactly the pattern CPUs prefetch most aggressively. The outer loop reads `frame_bitmap[w]` sequentially; the hardware prefetcher detects this stride and loads upcoming cache lines before you need them. A 64-byte cache line contains 512 bits, representing 512 frames = 2MB of physical memory. Scanning 4GB worth of frames reads `1M frames / 512 frames per cache line = 2048 cache lines = 128KB`, fitting entirely in a 256KB L2 cache after the first pass. Contrast with a free list: each `next` pointer points to an arbitrary physical address, producing a random access pattern with one L2 or L3 cache miss per allocation.

Section 3: x86 Two-Level Page Tables — The Hardware That Makes Virtual Memory Work

With physical frames available for allocation, the next task is building the structures that let the CPU translate virtual addresses to physical addresses. On 32-bit x86, this is a **two-level page table**: a page directory at the top level, and page tables at the second level.

{{DIAGRAM:diag-m3-two-level-page-table}}

Address Translation — How the CPU Decodes a Virtual Address

A 32-bit virtual address is divided into three fields by the CPU's Memory Management Unit (MMU — the hardware component that performs address translation):

Virtual Address (32 bits):

Directory Index [31:22] 10 bits (0-1023)	Table Index [21:12] 10 bits (0-1023)	Offset [11:0] 12 bits (0-4095)

The translation process — performed in hardware by the MMU on every memory access:

1. **Load CR3:** The CPU holds the physical address of the **Page Directory (PD)** in the `CR3` register. The page directory is a 4KB array of 1024 32-bit **Page Directory Entries (PDEs)**.
2. **Index the Page Directory:** Use bits [31:22] of the virtual address (the directory index, 0–1023) to select a PDE from the page directory. The PDE contains the physical address of a **Page Table (PT)**.
3. **Index the Page Table:** The page table is another 4KB array of 1024 32-bit **Page Table Entries (PTEs)**. Use bits [21:12] of the virtual address (the table index, 0–1023) to select a PTE. The PTE contains the physical address of the **4KB page frame**.
4. **Add the Offset:** Combine the physical frame address from the PTE with bits [11:0] of the virtual address (the byte offset within the page, 0–4095) to get the final physical address.

This entire four-memory-access sequence (`CR3 → PDE → PTE → data`) happens transparently on every memory access. The **TLB (Translation Lookaside Buffer)** caches recently used `(virtual page → physical frame)` translations so the CPU does not

repeat all four steps for every access. A TLB hit costs 1 cycle; a TLB miss triggers the full walk, costing 20–50 cycles plus the memory access latencies for PDE and PTE reads.

The PDE and PTE Entry Format

Both PDEs and PTEs are 32-bit values with the same flag layout:

Page Directory Entry / Page Table Entry (32 bits):

Physical Frame Address [31:12] (20 bits)	Flags [11:0]
---	--------------

Flags (bits [11:0]):

- Bit 0 (P) = Present: 1 = entry is valid; 0 = triggers page fault
- Bit 1 (R/W) = Read/Write: 0 = read-only; 1 = read-write
- Bit 2 (U/S) = User/Supervisor: 0 = kernel only (ring 0); 1 = user accessible (rings 0–3)
- Bit 3 (PWT) = Page Write-Through: controls cache write policy
- Bit 4 (PCD) = Page Cache Disable: 1 = uncachable (use for MMIO)
- Bit 5 (A) = Accessed: CPU sets this bit on first access to the page
- Bit 6 (D) = Dirty (PTE only): CPU sets this bit on first write to the page
- Bit 7 (PS) = Page Size (PDE only): 1 = 4MB page (PSE extension); 0 = 4KB subtable
- Bit 8 (G) = Global: TLB entry not flushed on CR3 reload (requires CR4.PGE)

The physical address field: Bits [31:12] give the physical frame address for a 4KB page. Since pages are 4KB-aligned, the low 12 bits of the frame address are always zero — those bits are repurposed as flags. The hardware extracts the frame address by masking bits [31:12] (zeroing the low 12 bits) and the offset is added separately.

The most critical flags for your kernel:

- **P (Present):** Must be 1 for any mapping that should work. Setting P=0 effectively unmaps the page — the next access causes a page fault (exception 14). This is how demand paging and guard pages work.
- **R/W (Read/Write):** Set to 1 for writable pages (kernel data, stacks, heap). Set to 0 for code pages and read-only data — an attempt to write triggers a page fault with an error code indicating a write violation.
- **U/S (User/Supervisor):** The primary isolation mechanism between user and kernel mode. Kernel pages have U/S=0 — user-mode code that accesses them triggers a page fault. This is why a buffer overflow in a user process cannot read kernel memory. The page fault error code tells you which violation occurred.

```
/* page.h - page table entry flags */

#define PAGE_PRESENT      (1 << 0)

#define PAGE_WRITABLE     (1 << 1)

#define PAGE_USER         (1 << 2)

#define PAGE_WRITETHROUGH (1 << 3)

#define PAGE_NOCACHE      (1 << 4)

#define PAGE_ACCESSED     (1 << 5)

#define PAGE_DIRTY        (1 << 6)

#define PAGE_LARGE        (1 << 7) /* PDE only: 4MB page if CR4.PSE set */

#define PAGE_GLOBAL        (1 << 8)

/* Extract physical frame address from a PDE/PTE */

#define PDE_FRAME(pde)   ((pde) & 0xFFFFF000)

#define PTE_FRAME(pte)   ((pte) & 0xFFFFF000)

/* Page directory and page table types */

typedef uint32_t pde_t;    /* Page Directory Entry */

typedef uint32_t pte_t;    /* Page Table Entry */
```

Section 4: Building the Page Directory — Identity Map and Higher-Half

Higher-Half Kernel: Boot Physical vs Runtime Virtual

BEFORE PAGING — Physical Address Space (CR0.PG=0)

0x00000000
Real-mode IVT + BIOS Data Area
(256B IVT + 256B BDA)

0x88000 gap
(BIOS holes)

0x00088000
VGA Text Buffer 80x25 cells
2 bytes per cell: char + attr
Physical MMIO — not RAM

MMIO not RAM

0x000C0000 to 0x000FFFFF
Video BIOS ROM + System BIOS
Option ROMs, firmware code

1MB boundary
kernel loads here

0x00100000 = __kernel_phys_start
.text section — kernel code
Executing HERE right now
EIP = 0x00100000 + offset

.text ends

0x00101000+
.rodata — read-only data
String literals, const tables
physical addr == linked addr

.rodata ends

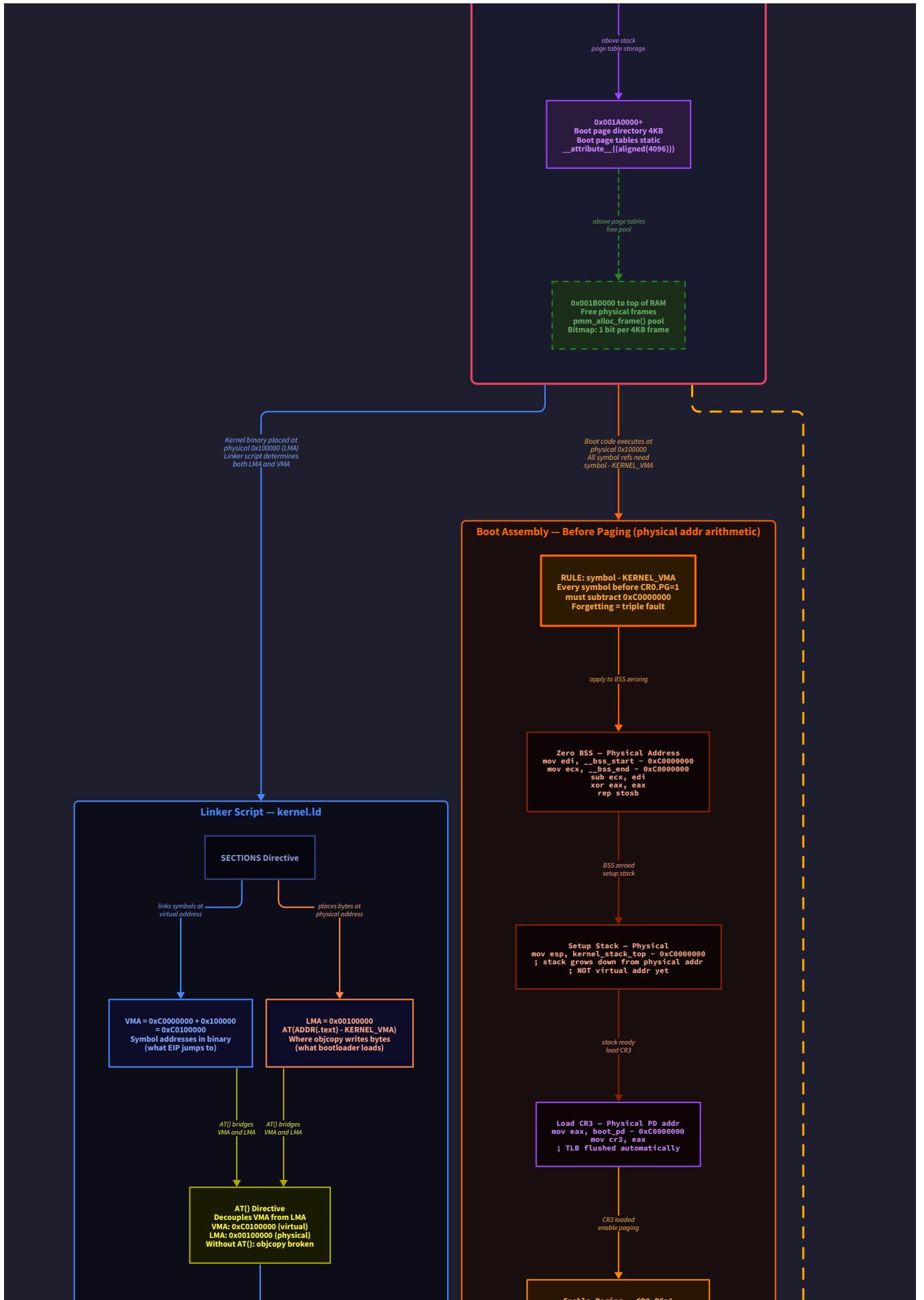
.data/.bss begins

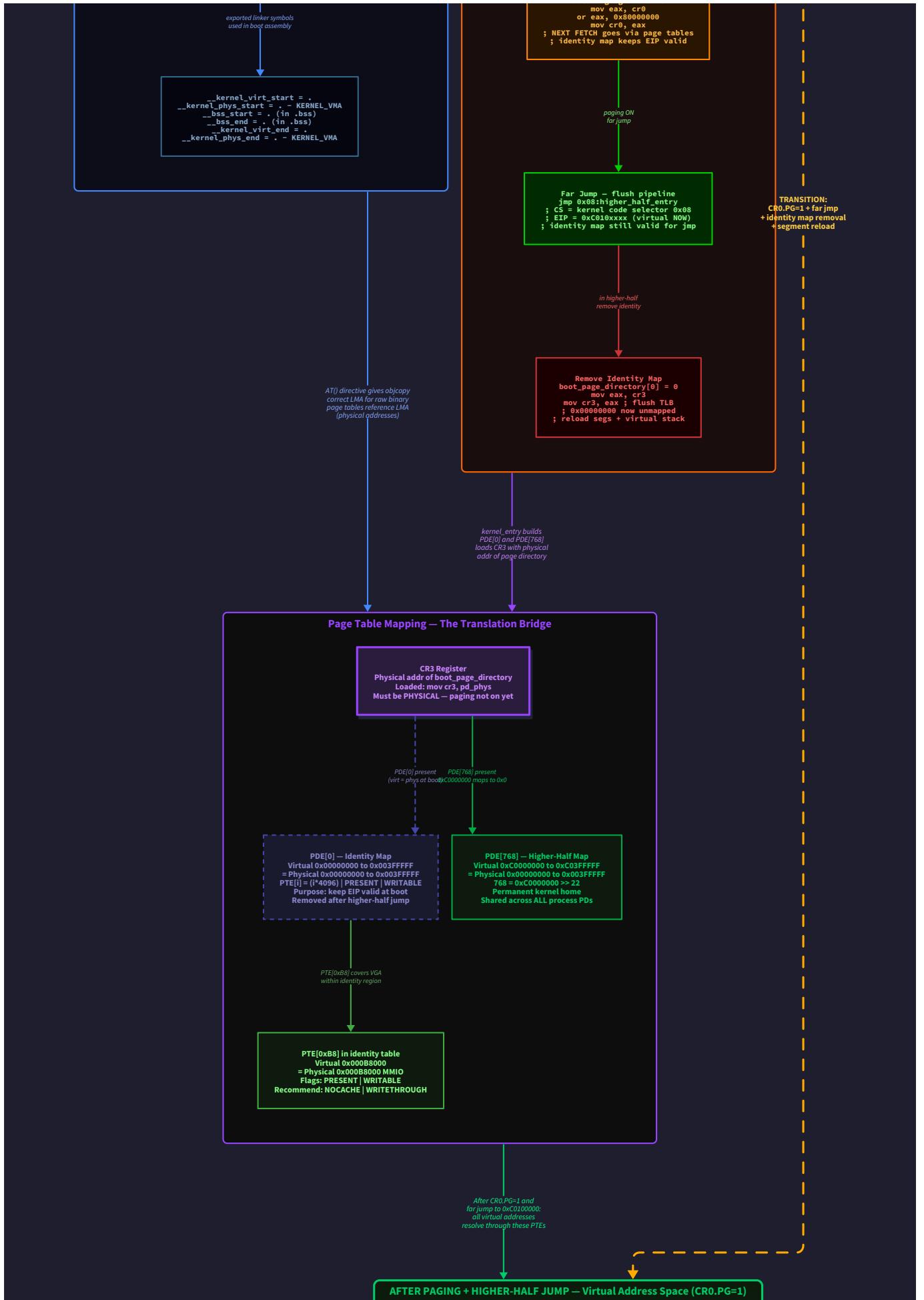
0x00102000+
.data — initialized globals
.bss — zeroed by kernel_entry
BSS zeroed with rep stosd

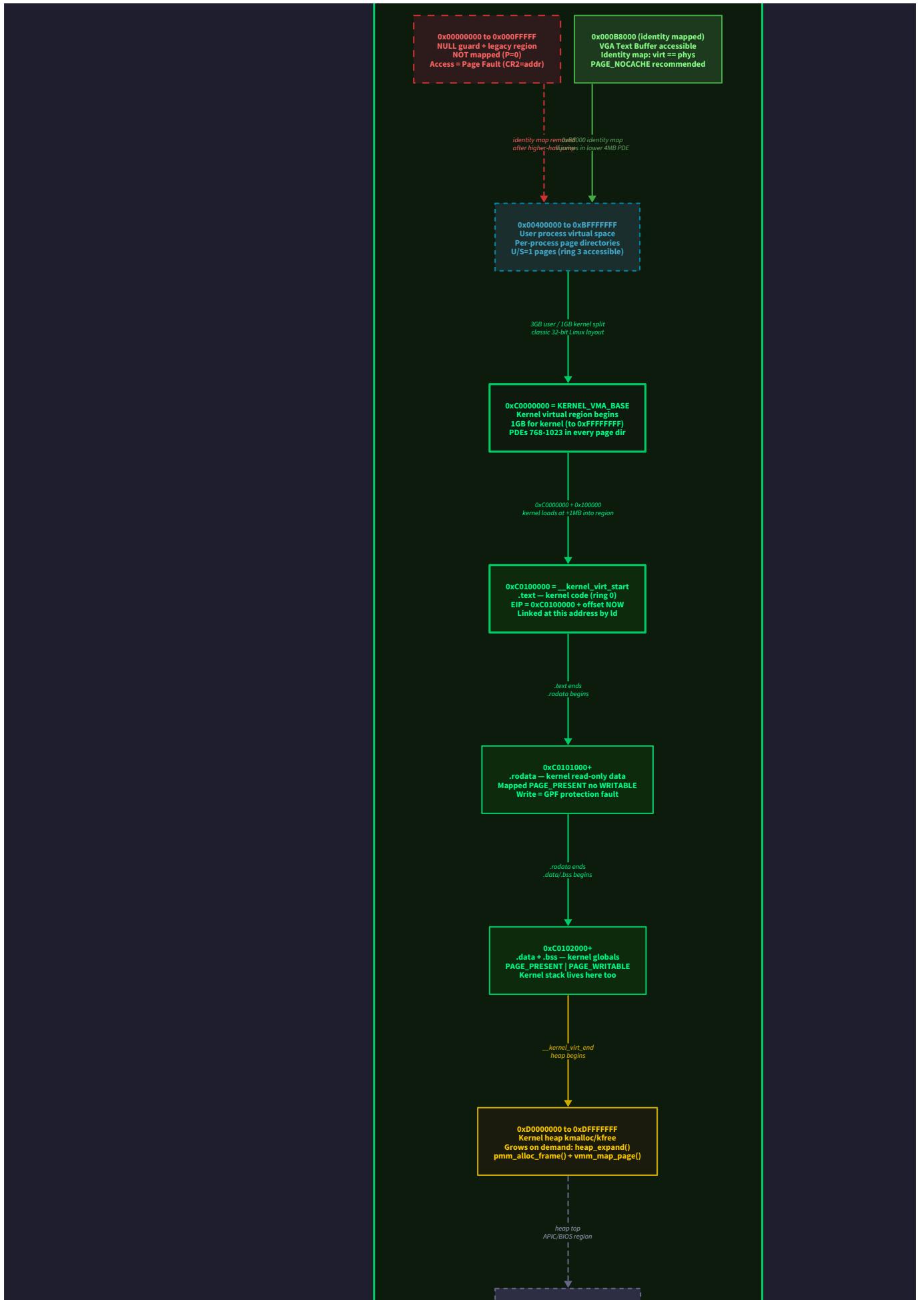
__kernel_phys_end

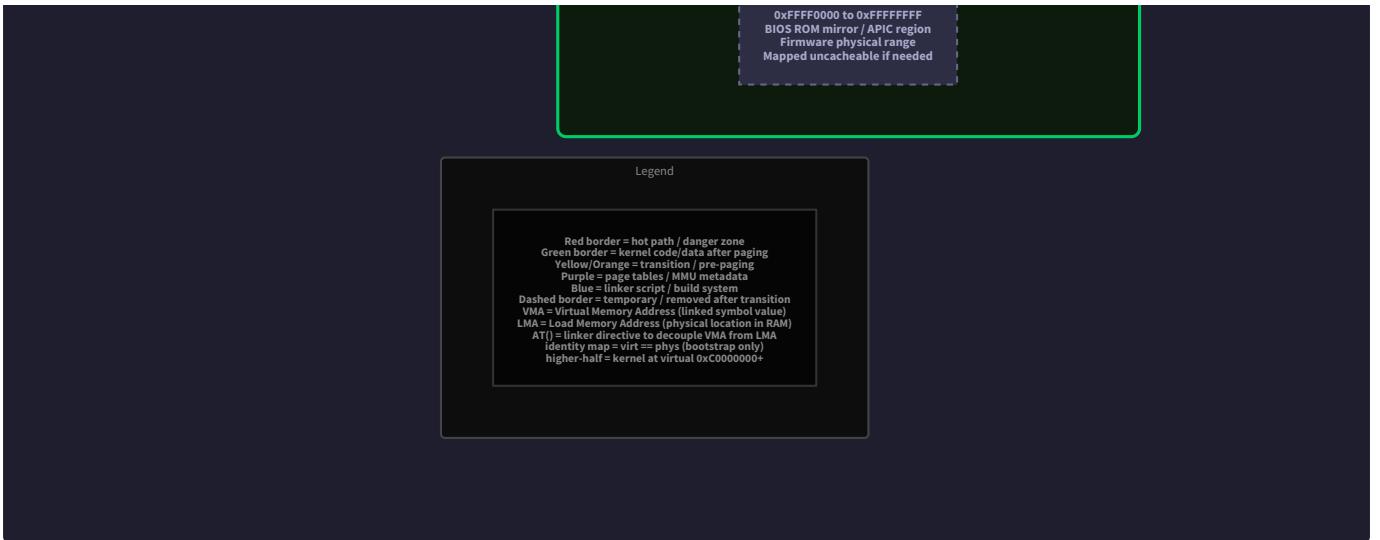
stack reserved

0x0019F000 (example)
Boot kernel stack
ESP set by kernel_entry.asm
Grows downward









Your kernel needs two simultaneous mappings during the bootstrapping phase:

Identity mapping (virtual = physical) for the low memory region your code is currently executing from. Without this, enabling paging causes the immediate page fault described in the revelation. The identity map must cover:

- The kernel code and data (loaded at physical `0x100000`)
- The VGA buffer (physical `0xB8000`) — also reachable via identity map of the first 4MB
- Any page directory/table structures you are about to build (they live in physical memory too)

Higher-half mapping (virtual `0xC0000000` = physical `0x100000`) for the kernel's intended virtual address home. After paging is on and you have jumped to the higher-half virtual address, you will unmap the identity mapping — user processes should not be able to "accidentally" use low virtual addresses to access kernel code.

Why `0xC0000000` (3GB)? It is a convention, not a requirement. The classic 32-bit Linux split is 3GB/1GB: user processes get virtual addresses `0x00000000` – `0xBFFFFFFF` (3GB), and the kernel occupies `0xC0000000` – `0xFFFFFFFF` (1GB). This means every user process's page directory has the upper 256 kernel entries (directory indices 768–1023) pre-populated with kernel page tables that are shared across all processes. Any other split is valid — Windows 2000 used 2GB/2GB; some embedded Linux configurations use 1GB/3GB for processes that need more than 3GB of virtual address space. The key constraint on 32-bit: user + kernel virtual ranges must fit in 4GB.

Page Table Construction — Before Paging Is On

The page tables must be built and populated while paging is still off (using physical addresses directly). After paging enables, you must access page tables through their virtual addresses (if you moved them to higher-half virtual space) or through the identity-mapped physical addresses.

```

/* vmm.c - Virtual Memory Manager */

#include "vmm.h"

#include "pmm.h"

#include "kprintf.h"

#include <string.h>

/* The boot page directory – aligned to 4KB boundary */

__attribute__((aligned(PAGE_SIZE)))

static pde_t boot_page_directory[1024];

/* A static page table for identity-mapping the first 4MB

(covers kernel at 0x100000, VGA at 0xB8000, all low memory) */

__attribute__((aligned(PAGE_SIZE)))

static pte_t identity_page_table[1024]; /* Maps virtual 0x0–0x3FFFFF */

/* A static page table for the higher-half kernel (virtual 0xC0000000–0xC03FFFFF) */

__attribute__((aligned(PAGE_SIZE)))

static pte_t kernel_page_table[1024]; /* Maps virtual 0xC0000000–0xC03FFFFF */

```

__attribute__((aligned(PAGE_SIZE))) : Page directory and page table structures must start at a 4KB-aligned address. The CPU reads the physical address from CR3 and the PDE's frame field — both assume 4KB alignment. GCC's **aligned** attribute instructs the linker to place this variable at a 4KB-aligned address. If you put these in BSS (as static variables), the linker honors the alignment. If you allocate them dynamically, use **pmm_alloc_frame()** which always returns 4KB-aligned physical addresses.

Populating the page tables:

```

void vmm_init(void) {

    /* — Step 1: Clear the page directory — */

    memset(boot_page_directory, 0, sizeof(boot_page_directory));

    /* — Step 2: Build identity map for virtual 0x0 - 0x3FFFFF (first 4MB) —

        This covers: real-mode IVT, BIOS data, VGA (0xB8000), kernel (0x100000)

        Directory index for 0x00000000 → bits [31:22] = 0 */

    for (int i = 0; i < 1024; i++) {

        /* Virtual page i → Physical frame i (identity) */

        identity_page_table[i] = (pte_t)((i * PAGE_SIZE) | PAGE_PRESENT | PAGE_WRITABLE);

    }

    /* Install in page directory at index 0 (virtual 0x00000000-0x003FFFFF) */

    boot_page_directory[0] =

        (pde_t)((uintptr_t)identity_page_table | PAGE_PRESENT | PAGE_WRITABLE);

    /* — Step 3: Build higher-half kernel map for virtual 0xC0000000 - 0xC03FFFFF —

        Maps to physical 0x00000000 - 0x003FFFFF (same frames as identity map)

        Directory index for 0xC0000000 → bits [31:22] = 768 (0xC0000000 >> 22) */

    for (int i = 0; i < 1024; i++) {

        /* Virtual page (768*1024 + i) → Physical frame i */

        kernel_page_table[i] = (pte_t)((i * PAGE_SIZE) | PAGE_PRESENT | PAGE_WRITABLE);

    }

    /* Install at directory index 768 */

    boot_page_directory[768] =

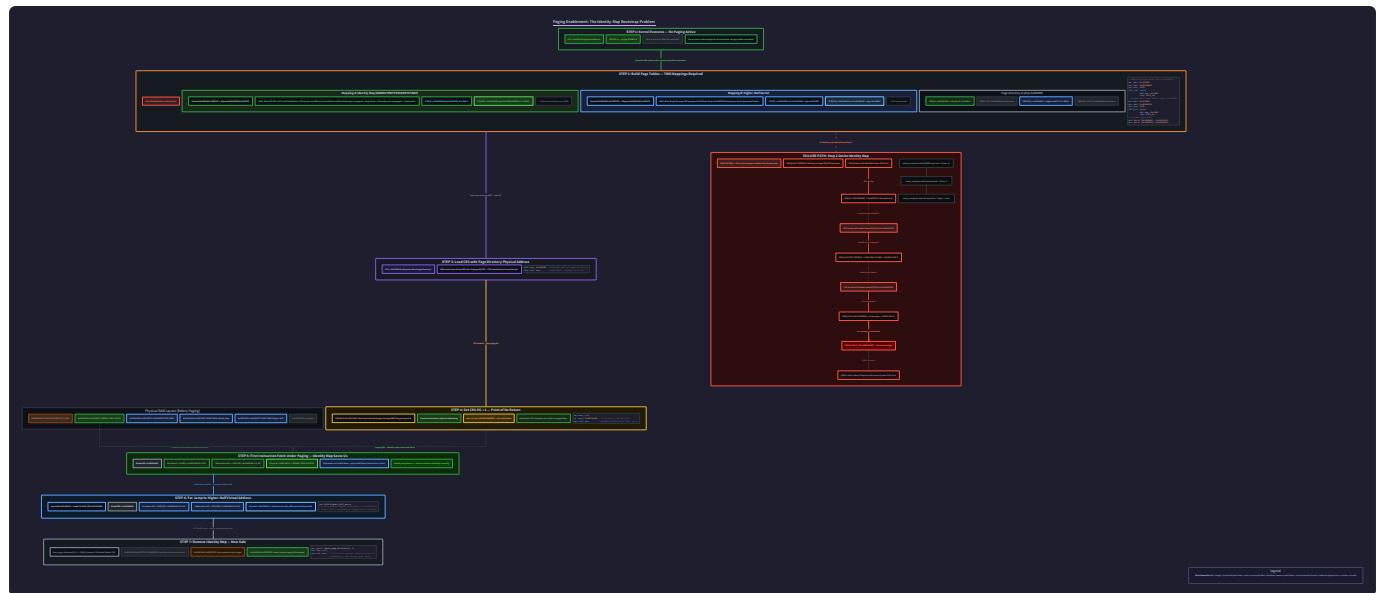
        (pde_t)((uintptr_t)kernel_page_table | PAGE_PRESENT | PAGE_WRITABLE);

    kprintf("[VMM] Page directory built. Identity [0x0-0x3FFFFF], "
           "Higher-half [0xC0000000-0xC03FFFFF]\n");
}

```

Covering VGA with the identity map: Physical `0xB8000` falls within the first 4MB identity-mapped range (it is less than `0x400000`). The identity map page table entry for this physical address sets `PAGE_PRESENT | PAGE_WRITABLE` — no `PAGE_NOCACHE`. This is technically incorrect for a device MMIO region (VGA should be uncached), but for a simple kernel the caching effects are negligible since VGA writes are write-combining anyway on modern systems. For correctness, set `PAGE_NOCACHE | PAGE_WI`RETHROUGH on the VGA page table entries.

Section 5: Enabling Paging — The Point of No Return



With the page directory built, enabling paging is a three-instruction sequence. But those three instructions have precise ordering requirements:

```

void vmm_enable_paging(void) {
    /* — Step 1: Load CR3 with the physical address of the page directory —
     * CR3 must receive a PHYSICAL address — paging is not on yet, so
     * the CPU cannot translate a virtual address to find the PD. */
    uint32_t pd_phys = (uint32_t)(uintptr_t)boot_page_directory;
    __asm__ volatile ("mov %0, %%cr3" : : "r"(pd_phys));

    /* — Step 2: Set CRO.PG (bit 31) and CRO.WP (bit 16 — Write Protect)
     * WP: when set, ring 0 code cannot write to read-only user pages.
     * This enables copy-on-write semantics even for kernel accesses.
     * PG: enables paging. THE NEXT INSTRUCTION FETCH GOES THROUGH PAGE TABLES. */
    uint32_t cr0;
    __asm__ volatile ("mov %%cr0, %0" : "=r"(cr0));
    cr0 |= (1U << 31); /* CRO.PG = 1: enable paging */
    cr0 |= (1U << 16); /* CRO.WP = 1: write-protect enable */
    __asm__ volatile ("mov %0, %%cr0" : : "r"(cr0));

    /* — At this point, paging is ON. The identity map means this code
     * is still reachable at its current physical=virtual address.
     * The next lines execute at their physical addresses via the identity map. — */
    /* — Step 3: Far jump to flush pipeline and begin using virtual addresses —
     * (Optional for identity-mapped kernel; mandatory when jumping to higher-half) */
    kprintf("[VMM] Paging enabled. Virtual memory is now active.\n");
}

```

Why load CR3 before CR0.PG? If you set CR0.PG first with CR3=0 (or garbage), the CPU immediately tries to do a page table walk from physical address 0, which is the real-mode IVT — clearly not a valid page directory. The page fault handler would then try to execute, also through an invalid table. CR3 must contain a valid page directory's physical address before you flip the paging bit.

Hardware Soul: Setting `CR0.PG` invalidates the entire TLB automatically — the CPU assumes that switching from no-paging to paging mode means all previous (implicit identity) translations are invalid. This is the one TLB flush you get for free. After this point, every CR3 modification (switching page directories for different processes) also flushes the TLB — except for entries marked `PAGE_GLOBAL`, which survive CR3 reloads. Kernel pages marked global are thus TLB-persistent across process switches, eliminating the cost of re-walking kernel page tables on every context switch.

The Jump to Higher-Half Virtual Space

After enabling paging, your code is still executing at low virtual addresses (via the identity map). If you want to transition to higher-half virtual addresses (so that your kernel runs at `0xC0100000` virtual rather than `0x100000`), you need a far jump or an indirect jump through a higher-half symbol:

```
/* After paging is on, jump to the higher-half virtual address of kmain */

extern void kmain_high(void); /* Linked at 0xC0xxxxxx */

void transition_to_higher_half(void) {

    /* An indirect call through a function pointer forces the CPU to use
       the higher-half address rather than the identity-mapped address */

    void (*kmain_virt)(void) = (void (*)(void))((uintptr_t)&kmain_high);

    kmain_virt();

    /* Unmap the identity map now that we are in higher-half space */

    boot_page_directory[0] = 0;           /* Remove identity PDE */

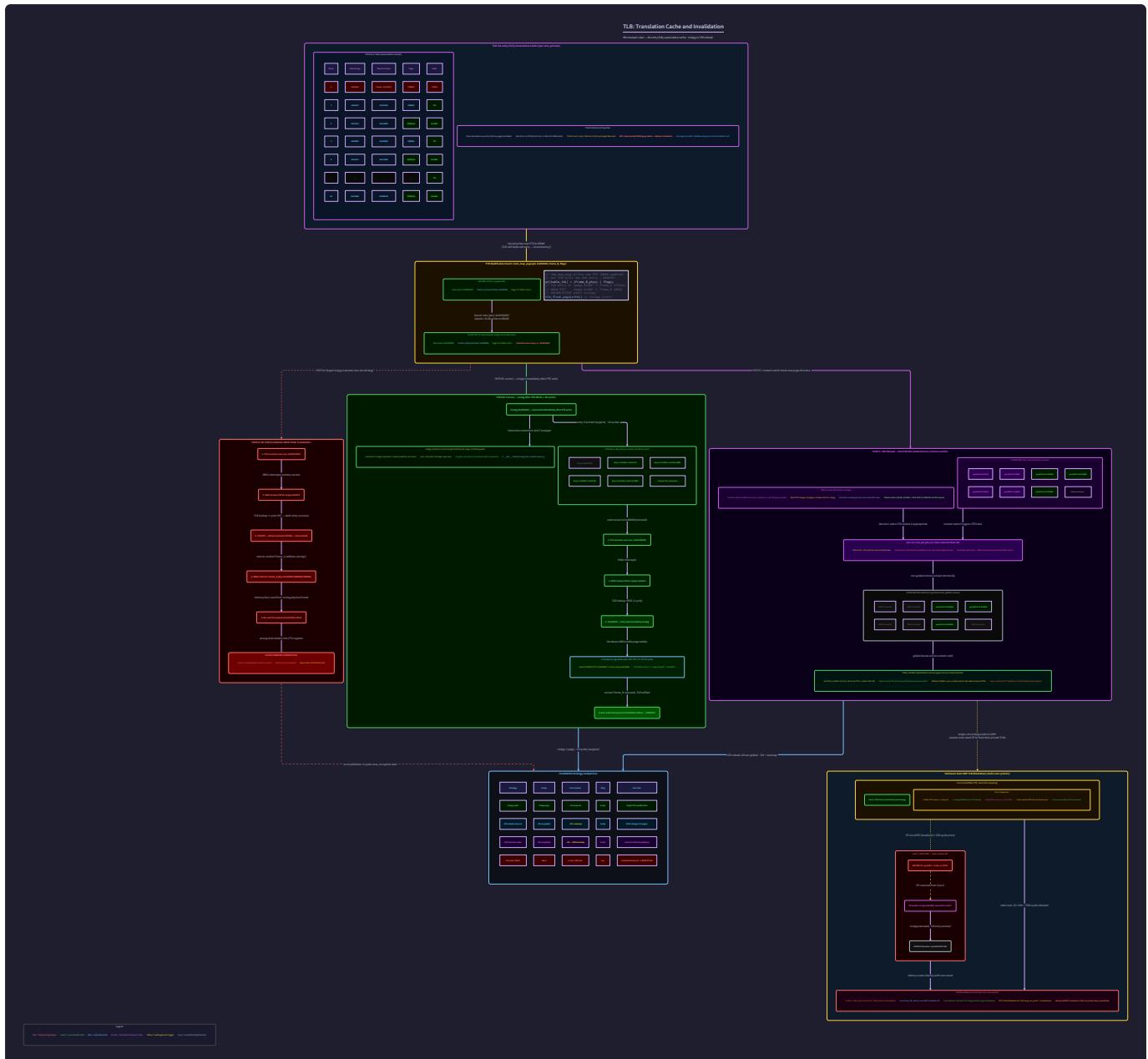
    __asm__ volatile ("mov %%cr3, %%eax; mov %%eax, %%cr3" ::: "eax"); /* Flush TLB */

}
```

This milestone's simplification: For a learning kernel, running the kernel at its identity-mapped physical address throughout is acceptable. Higher-half mapping is the production approach (it prevents user processes from occupying the same virtual address range as the kernel), and the acceptance criteria require demonstrating both mappings. The critical property is that both exist simultaneously in the page directory, with the identity map removable once you have verified higher-half execution.

Section 6: TLB Management — The Stale Translation Problem

The TLB is a small, fully-associative cache (typically 32–64 entries for the L1 DTLB, a few hundred for L2 TLB) that stores recently used `(virtual page, ASID) → physical frame` translations. Without the TLB, every memory access would require three additional memory reads ($\text{CR3} \rightarrow \text{PDE} \rightarrow \text{PTE} \rightarrow \text{data}$). The TLB reduces this to zero additional reads on a hit.



The TLB is not coherent with your page table modifications. When you change a PTE — mapping a virtual page to a new physical frame, or changing its flags from read-only to writable — the TLB may still contain the old translation. The CPU will serve accesses from the stale TLB entry, sending traffic to the wrong physical frame or enforcing wrong permissions, until the TLB entry is evicted or explicitly invalidated.

There are two invalidation mechanisms:

invlpg — Invalidate a single TLB entry (specific virtual page):

```
static inline void tlb_flush_page(uint32_t virtual_addr) {
    /* invlpg [mem] invalidates the TLB entry for the page containing virtual_addr.
     * The operand is a memory address, not a value; we use a memory constraint. */
    __asm__ volatile ("invlpg (%0)" : : "r"(virtual_addr) : "memory");
}
```

Use `invlpg` when you are modifying a single PTE. It is precise — only the one translated page is invalidated. Cost: ~20 cycles on modern CPUs.

CR3 reload — Flush the entire TLB:

```
static inline void tlb_flush_all(void) {  
    /* Writing CR3 with the same value flushes all non-global TLB entries */  
  
    uint32_t cr3;  
  
    __asm__ volatile (  
  
        "mov %%cr3, %0\n\t"  
  
        "mov %0, %%cr3"  
  
        : "=r"(cr3)  
  
        :  
  
        : "memory"  
  
    );  
  
}
```

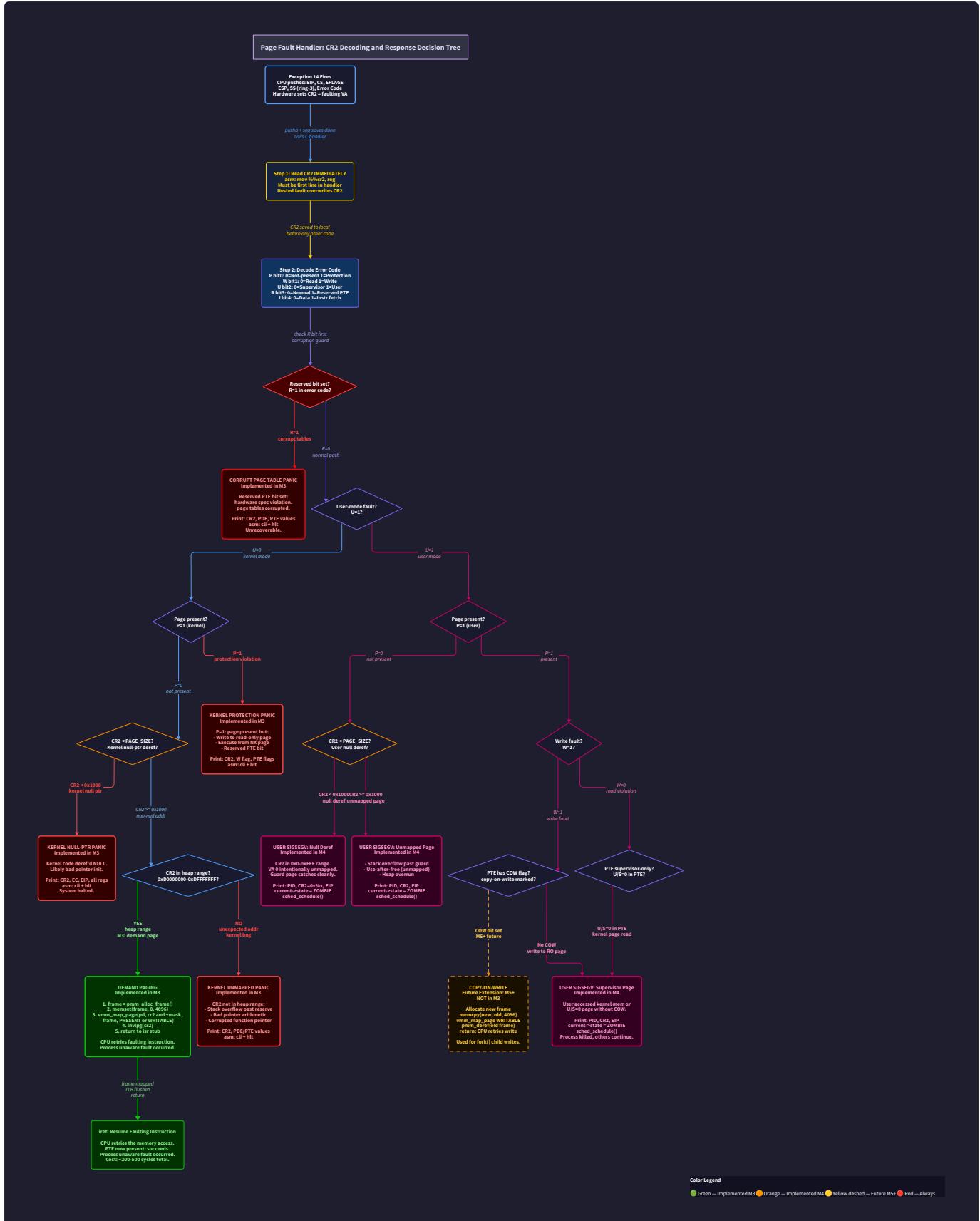
Use a CR3 reload when you have made many page table changes — for instance, after building a new process's page directory. It flushes all non-global entries. Cost: ~30 cycles for the reload itself, plus the cost of TLB cold misses on subsequent accesses (each miss requires a page table walk).

Global pages (PAGE_GLOBAL): If you set the `G` flag in a PTE and enable `CR4.PGE` (Page Global Enable), that TLB entry survives CR3 reloads. This is used for kernel pages that are mapped identically in every process's page directory — you do not want to flush and re-walk kernel page tables on every process switch. The identity-mapped kernel pages and higher-half kernel pages should be global.

The rule for your kernel: **every time you modify a PTE or PDE, call `tlb_flush_page()` for that virtual address** (or `tlb_flush_all()` if you modified many entries). Forgetting this is silent corruption — the stale translation sends accesses to the wrong frame, and the bug often manifests in unrelated code that accesses a page that has since been remapped.

Section 7: The Page Fault Handler — CR2 as Your Diagnostic Oracle

Page fault (exception 14) is the CPU's mechanism for saying: "I tried to translate this virtual address and something was wrong." It is the most informative exception in the system — it fires for accessing unmapped pages, writing to read-only pages, and user-mode code touching supervisor-only pages. In future milestones, you will also use it for demand paging (allocate frames lazily on first access) and copy-on-write.



When a page fault fires, the CPU provides two pieces of information automatically:

CR2 — The Faulting Virtual Address: The CPU stores the virtual address that caused the fault in `CR2`. This is the address the MMU tried to translate when the fault occurred. Your handler must read CR2 *immediately* — it is a CPU register that will be overwritten if another page fault occurs before you save it.

The Error Code (pushed onto the stack by the CPU, already handled by your ISR framework from Milestone 2):

Page Fault Error Code (pushed by CPU):

Reserved (0)	U	W	P
[31:3]	[2]	[1]	[0]

P (bit 0): 0 = Page not present (PTE.P was 0)
1 = Protection violation (page present but wrong permissions)
W (bit 1): 0 = Fault caused by a read
1 = Fault caused by a write
U (bit 2): 0 = Fault in supervisor mode (ring 0)
1 = Fault in user mode (ring 3)

The combination of these three bits tells you everything about the nature of the fault:

P	W	U	Meaning
0	0	0	Kernel read of unmapped page — null pointer dereference in kernel?
0	1	0	Kernel write of unmapped page — stack overflow, heap corruption?
0	0	1	User read of unmapped page — user null pointer dereference
0	1	1	User write of unmapped page — user null pointer dereference (write)
1	0	1	User read of kernel page — security violation (U/S=0 page)
1	1	1	User write of kernel page — security violation
1	1	0	Kernel write to read-only page — kernel bug, or COW trigger

The page fault handler is already registered at vector 14 in your IDT from Milestone 2. Now you give it meaningful behavior:

```
/* In interrupt.c - page fault handler (exception 14) */

void page_fault_handler(interrupt_frame_t *frame) {

    /* Read CR2: the faulting virtual address */

    uint32_t faulting_addr;

    __asm__ volatile ("mov %%cr2, %0" : "=r"(faulting_addr));

    /* Decode the error code */

    int present = frame->error_code & 0x1;      /* P bit */
    int write   = frame->error_code & 0x2;      /* W bit */
    int user    = frame->error_code & 0x4;      /* U bit */

    kprintf("\n==== PAGE FAULT ===\n");

    kprintf("Faulting address (CR2): 0x%08x\n", faulting_addr);

    kprintf("Error: %s%s%s\n",
           present ? "protection-violation " : "page-not-present ",
           write   ? "write " : "read ",
           user    ? "user-mode" : "kernel-mode");

    kprintf("Fault at EIP: 0x%08x\n", frame->eip);

    /* Security check: user process touching kernel space */

    if (user && faulting_addr >= 0xC0000000) {

        kprintf("SECURITY: User process attempted to access kernel memory.\n");

        /* In Milestone 4, this kills the offending process.

           For now, halt. */

    }

    /* Kernel null pointer dereference */

    if (!user && faulting_addr < PAGE_SIZE) {

        kprintf("KERNEL BUG: Null pointer dereference in kernel code!\n");
    }

    /* For now: all page faults are fatal. Milestone 4+ adds demand paging here. */

    kprintf("System halted.\n");

    for (;;) { __asm__ volatile ("cli; hlt"); }

}
```

Why is CR2 read in the fault handler rather than in the ISR stub? The ISR stub saves general-purpose registers via `pusha`, but `CR2` is not a general-purpose register — `pusha` does not touch it. You must read `CR2` explicitly in your handler. The window between `CR2` being set by the hardware and your handler reading it is safe: another page fault cannot fire while you are in the fault handler (interrupts are disabled via the interrupt gate, and exceptions that fire in ring 0 with the current interrupt gate settings would immediately cause a double fault rather than a nested page fault invocation).

Section 8: Dynamic Page Table Management — `vmm_map_page`

The static identity and higher-half mappings you built during `vmm_init` cover only the first 4MB. As your kernel allocates new data structures, stack space, and heap memory beyond 4MB, you need a function to map arbitrary virtual pages to physical frames on demand.

```
/* vmm.c - dynamic page table management */

/* Map a single virtual page to a physical frame with given flags.
   Allocates a new page table if the directory entry is not present. */

void vmm_map_page(pde_t *page_directory,
                  uint32_t virt_addr,
                  uint32_t phys_addr,
                  uint32_t flags) {

    /* Extract directory and table indices from virtual address */

    uint32_t dir_idx = virt_addr >> 22;           /* Bits [31:22] */
    uint32_t table_idx = (virt_addr >> 12) & 0x3FF; /* Bits [21:12] */

    pde_t *pde = &page_directory[dir_idx];

    if (!(pde & PAGE_PRESENT)) {
        /* No page table exists for this directory entry yet.

           Allocate a new physical frame for the page table. */

        void *new_table_phys = pmm_alloc_frame();

        if (!new_table_phys) {
            kprintf("[VMM] FATAL: Cannot allocate page table frame.\n");
            return;
        }

        /* Zero the new page table (all entries: not present) */

        memset(new_table_phys, 0, PAGE_SIZE);

        /* Install the page table in the directory */

        *pde = (pde_t)((uintptr_t)new_table_phys | PAGE_PRESENT | PAGE_WRITABLE |
                      (flags & PAGE_USER)); /* Propagate user bit to PDE */

        /* Flush TLB for this directory entry */

        tlb_flush_page((uint32_t)pde);
    }

    /* Get pointer to the page table */

    pte_t *page_table = (pte_t *)PDE_FRAME(*pde);
```

```

/* Install the mapping */

page_table[table_idx] = (pte_t)(phys_addr | PAGE_PRESENT | flags);

/* Flush the specific virtual address from TLB */

tlb_flush_page(virt_addr);

}

/* Unmap a virtual page (set PTE to not-present) */

void vmm_unmap_page(pde_t *page_directory, uint32_t virt_addr) {

    uint32_t dir_idx = virt_addr >> 22;

    uint32_t table_idx = (virt_addr >> 12) & 0x3FF;

    pde_t pde = page_directory[dir_idx];

    if (!(pde & PAGE_PRESENT)) return; /* Already not mapped */

    pte_t *page_table = (pte_t *)PDE_FRAME(pde);

    page_table[table_idx] = 0;           /* Clear the PTE (P=0: not present) */

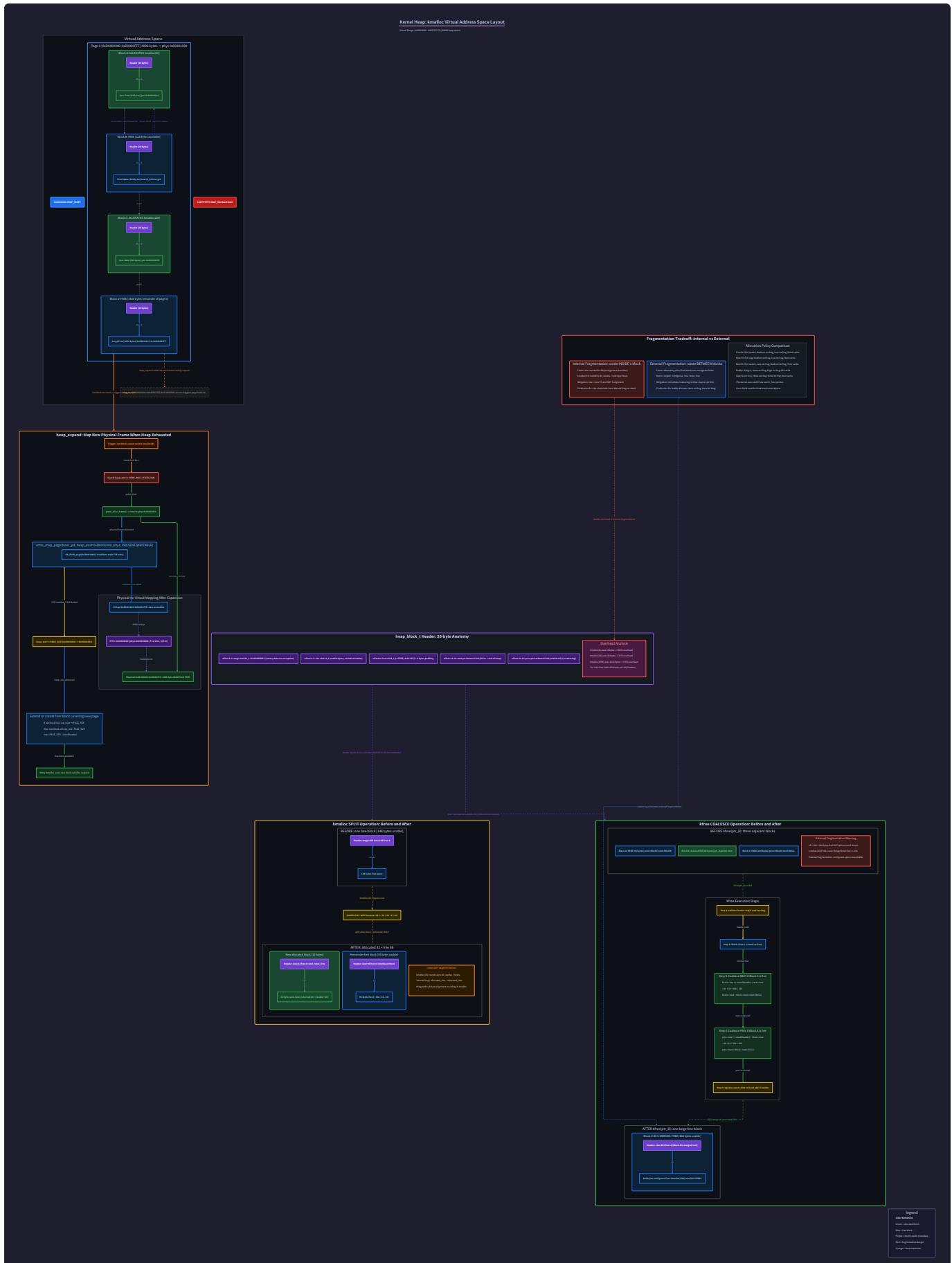
    tlb_flush_page(virt_addr); /* MANDATORY: remove stale TLB entry */
}

```

Accessing page tables after paging is enabled: In the code above, `(pte_t *)PDE_FRAME(*pde)` accesses the page table using its *physical* address as if it were a virtual address. This works only because the identity map is active — physical address `X` is also accessible at virtual address `X`. Once you remove the identity map (after higher-half transition), you cannot access page tables this way. Production kernels solve this with the **recursive page directory** technique: one PDE points back to the page directory itself, making all page tables accessible at fixed virtual addresses. Alternatively, map all page tables into a dedicated kernel virtual address range. Both techniques are valuable to know; for this milestone, keeping the identity map active simplifies page table management considerably.

Section 9: The Kernel Heap — kmalloc and kfree

The physical frame allocator gives you 4KB chunks. The page table manager maps them into virtual address space. But kernel code constantly needs variable-sized allocations: a 24-byte PCB (Process Control Block), a 36-byte interrupt descriptor, a 512-byte keyboard buffer. Allocating an entire 4KB frame for each of these wastes approximately 99% of the frame. You need a **heap allocator** — a subsystem that manages a region of virtual address space and provides variable-size allocation within it.



The kernel heap allocator (`kmalloc` / `kfree`) works in two layers:

1. **Virtual address space management:** A dedicated virtual address range (say, `0xD0000000` – `0xFFFFFFFF` — 256MB) is reserved for the heap. The heap grows upward from `0xD0000000` as memory is needed.
2. **Physical frame backing:** As the heap grows, new 4KB frames are allocated from `pmm_alloc_frame()` and mapped into the virtual heap region via `vmm_map_page()`.

A minimal first-fit heap allocator sufficient for this milestone:

```
/* heap.c - kernel heap allocator */

#include "heap.h"

#include "pmm.h"

#include "vmm.h"

#include "kprintf.h"

#define HEAP_START    0x00000000 /* Virtual address where heap begins */

#define HEAP_MAX      0xFFFFFFFF /* Virtual address where heap ends */

#define HEAP_MAGIC    0xDEADBEEF /* Canary value for block header validation */

/* Each allocation is preceded by a header and followed by a footer */

typedef struct heap_block {

    uint32_t        magic;    /* HEAP_MAGIC - detect header corruption */

    uint32_t        size;     /* Size of the usable region (excluding header+footer) */

    uint8_t         free;     /* 1 = free, 0 = allocated */

    struct heap_block *next;   /* Next block in the heap (doubly-linked for coalescing) */

    struct heap_block *prev;   /* Previous block */

} heap_block_t;

/* Heap state */

static uint32_t heap_start = HEAP_START;

static uint32_t heap_end = HEAP_START; /* Current end of mapped heap */

static heap_block_t *heap_head = NULL; /* First block */

/* Extend the heap by adding one more physical frame (4KB) */

static void heap_expand(void) {

    if (heap_end >= HEAP_MAX) {

        kprintf("[HEAP] FATAL: Heap exhausted at 0x%x\n", heap_end);

        for (;;);

    }

    /* Allocate a physical frame */

    void *phys = pmm_alloc_frame();

    if (!phys) {

        kprintf("[HEAP] FATAL: pmm_alloc_frame() returned NULL\n");

    }

}
```

```

    for (;;);

}

/* Map the new frame at the current heap_end virtual address */

vmm_map_page(boot_page_directory,
             heap_end,
             (uint32_t)(uintptr_t)phys,
             PAGE_PRESENT | PAGE_WRITABLE);

heap_end += PAGE_SIZE;

}

void heap_init(void) {

    /* Map the first page of the heap */

    heap_expand();

    /* Initialize the first (and only) free block spanning the entire first page */

    heap_head = (heap_block_t *)(uintptr_t)heap_start;

    heap_head->magic = HEAP_MAGIC;

    heap_head->size = PAGE_SIZE - sizeof(heap_block_t);

    heap_head->free = 1;

    heap_head->next = NULL;

    heap_head->prev = NULL;

    kprintf("[HEAP] Initialized at 0x%llx, first page mapped.\n", heap_start);

}

void *kmalloc(size_t size) {

    if (size == 0) return NULL;

    /* Align size to 8 bytes to maintain alignment invariant */

    size = (size + 7) & ~7;

    /* First-fit search: walk the block list */

    heap_block_t *block = heap_head;

    while (block) {

```

```

/* Validate the header – detects heap corruption early */

if (block->magic != HEAP_MAGIC) {

    kprintf("[HEAP] BUG: Heap corruption detected (bad magic at 0x%llx)!\n", block);

    for (;;)

}

if (block->free && block->size >= size) {

    /* Found a free block that fits */

    /* Split the block if there is enough space for a new header + at least 8 bytes */

    if (block->size >= size + sizeof(heap_block_t) + 8) {

        heap_block_t *new_block =

            (heap_block_t *)((uint8_t *)block + sizeof(heap_block_t) + size);

        new_block->magic = HEAP_MAGIC;

        new_block->size = block->size - size - sizeof(heap_block_t);

        new_block->free = 1;

        new_block->next = block->next;

        new_block->prev = block;

        if (block->next) block->next->prev = new_block;

        block->next = new_block;

        block->size = size;

    }

    block->free = 0;

    return (void *)((uint8_t *)block + sizeof(heap_block_t));

}

/* If this is the last block and it did not fit, expand the heap */

if (!block->next) {

    /* Expand heap by one page, then retry */

    heap_expand();

    /* Extend the current last block into the new page

       (if it is free – avoids creating tiny fragmented blocks) */
}

```

```

    if (block->free) {

        block->size += PAGE_SIZE;

        /* Try this block again on the next loop iteration */

        continue;

    } else {

        /* Create a new block in the new page */

        heap_block_t *new_page_block =

            (heap_block_t *)((uintptr_t)(heap_end - PAGE_SIZE));

        new_page_block->magic = HEAP_MAGIC;

        new_page_block->size = PAGE_SIZE - sizeof(heap_block_t);

        new_page_block->free = 1;

        new_page_block->next = NULL;

        new_page_block->prev = block;

        block->next = new_page_block;

    }

}

block = block->next;

}

kprintf("[HEAP] kmalloc(%u) failed: no block found.\n", size);

return NULL;

}

void kfree(void *ptr) {

if (!ptr) return;

/* Recover the block header: it sits immediately before the user data */

heap_block_t *block = (heap_block_t *)((uint8_t *)ptr - sizeof(heap_block_t));

/* Validate */

if (block->magic != HEAP_MAGIC) {

    kprintf("[HEAP] BUG: kfree() called with invalid pointer 0x%llx!\n", ptr);

    return;

}

```

```

if (block->free) {

    kprintf("[HEAP] BUG: Double-free detected for 0x%llx\n", ptr);

    return;
}

block->free = 1;

/* Coalesce with next block if it is free */

if (block->next && block->next->free) {

    block->size += sizeof(heap_block_t) + block->next->size;

    block->next = block->next->next;

    if (block->next) block->next->prev = block;
}

/* Coalesce with previous block if it is free */

if (block->prev && block->prev->free) {

    block->prev->size += sizeof(heap_block_t) + block->size;

    block->prev->next = block->next;

    if (block->next) block->next->prev = block->prev;
}

}

```

Header layout and memory cost: Every allocation carries a `heap_block_t` header overhead. On a 32-bit system: `magic` (4) + `size` (4) + `free` (1) + padding(3) + `next` (4) + `prev` (4) = 20 bytes header per allocation. A 4-byte integer allocation thus uses 24 bytes — 600% overhead. This is the same trade-off glibc's malloc makes, though glibc uses a 16-byte header and encodes the `free` flag in the low bits of `size` (exploiting alignment guarantees). The overhead is acceptable for kernel allocations; for user-mode workloads with millions of small allocations, jemalloc's size-class slabs eliminate per-object headers entirely.

Hardware Soul: The heap allocator's `first-fit` search walks a linked list of block headers. Each header is at an unpredictable address (wherever the previous allocation ended). This is a **pointer-chasing** access pattern — the address of `block->next` is read, then used to fetch the next block header, which is at an arbitrary address not adjacent to the current one. Every block access is likely an L1 or L2 cache miss if the heap has cooled. This is why production allocators (jemalloc, tcmalloc) organize objects into **size-class slabs**: all 32-byte objects live in contiguous 4KB slabs, so scanning free slots within a slab is sequential (cache-friendly). The linked-list header approach trades simplicity for cache performance. For a kernel with hundreds of allocations rather than millions, it is the right trade-off.

Section 10: The Complete Memory Subsystem Initialization

Bringing all three components together in `kmain()`, inserted after Milestone 2's interrupt setup and before any dynamic memory use:

```
/* kmain.c */

void kmain(multiboot_info_t *mbi) {

    /* Milestone 1 */

    serial_init();
    vga_clear();
    kprintf("==> Kernel booting ==>\n");

    /* Milestone 2 */

    idt_setup_all();
    pic_remap(0x20, 0x28);
    pit_init(100);
    keyboard_init();
    __asm__ volatile ("sti");

    /* Milestone 3: Memory management */

    /* Step A: Parse physical memory map from bootloader */
    pmm_parse_memory_map(mbi);      /* Prints memory regions; determines usable frames */

    /* Step B: Initialize physical frame allocator */
    pmm_init(mbi);                /* Builds bitmap; marks kernel + reserved frames as used */

    /* Step C: Build page tables and enable virtual memory */
    vmm_init();                    /* Constructs page directory with identity + higher-half maps */
    vmm_enable_paging();           /* Loads CR3, sets CR0.PG – from this point, paging is on */

    /* Step D: Initialize kernel heap */
    heap_init();                  /* Maps first heap page; initializes free block list */

    kprintf("[M3] Memory subsystem ready.\n");
    kprintf("[M3] Physical frames: %u free / %u total\n",
           pmm_frames_free(), pmm_frames_free() + pmm_frames_used());

    /* Verify kmalloc/kfree */
    void *p1 = kmalloc(64);
    void *p2 = kmalloc(128);
    kprintf("[M3] kmalloc test: p1=0x%08x p2=0x%08x\n", p1, p2);
}
```

```

    kfree(p1);

    void *p3 = kmalloc(32); /* Should reuse p1's space */

    kprintf("[M3] kfree+realloc test: p3=0x%08x (should be near p1)\n", p3);

    kfree(p2);

    kfree(p3);

    /* Verify page fault handler */

    kprintf("[M3] Testing page fault handler... (will halt after this)\n");

    /* volatile prevents compiler from optimizing away the access */

    /* Uncomment to test: volatile int *bad = (int*)0xDEAD0000; *bad = 42; */

    kprintf("[M3] All memory milestone tests passed.\n");

    for (;;) { __asm__ volatile ("hlt"); }

}

```

Why initialize PMM before VMM before heap? Each layer depends on the previous: the heap calls `vmm_map_page` (which needs paging on), which calls `pmm_alloc_frame` (which needs the bitmap initialized). More subtly: `vmm_init()` itself allocates no physical frames (it uses statically declared page tables) but `vmm_map_page()` inside `heap_init()` does call `pmm_alloc_frame()`. The ordering is load-bearing.

Section 11: The Three-Level View — What "Access Virtual Address" Truly Means

Tracing a single memory read instruction through the complete stack, from C source to physics:

Level 1 — The Application: Your kernel code writes `int x = *ptr;` where `ptr` is a virtual address `0xD0001234` (somewhere in the kernel heap). The C compiler emits a `mov eax, [0xD0001234]` instruction (or equivalent with a register holding the address).

Level 2 — The MMU and Page Table Walk: The CPU's MMU intercepts the memory access. It extracts the directory index: `0xD0001234 >> 22 = 0x340 = 832`. It reads `page_directory[832]` from the physical address held in CR3 — this is a memory read from the page directory frame. If TLB hits: done in 1 cycle. If TLB miss: the MMU reads the PDE, extracts the page table's physical address, reads `page_table[(0xD0001234 >> 12) & 0x3FF]` from that frame, extracts the physical frame address, adds the offset `0x234`, and loads from that final physical address. Result: 3 memory reads for the translation + 1 for the actual data = 4 total. The successful translation is cached in the TLB for subsequent accesses to the same page.

Level 3 — Cache Hierarchy and DRAM: The PDE read hits L2 cache (the page directory is hot from recent allocations). The PTE read may be an L3 hit or a DRAM access — 10ns or 60ns respectively. The data read at `0xD0001234` (physical `0x00201234` after translation, say) is likely in L1 or L2 if recently accessed. The total cost for a TLB-miss access to cold data: 200 cycles (TLB miss table walk + L3 misses). A TLB-hit access to L1-cached data: 4–5 cycles. The 50x difference is why TLB working set matters: if your kernel touches more unique pages than the TLB can hold (64 pages for L1 DTLB), you pay the full 200-cycle penalty on every access, no matter how aggressively the CPU prefetches.

Hardware Soul: The TLB on modern x86 CPUs is split into separate L1 instruction TLB (iTLB) and L1 data TLB (dTLD), each typically 32–64 entries. An L2 TLB (stlb — shared translation cache) holds 512–4096 entries. For a kernel with a few dozen active pages (page tables, heap blocks, stack), the working set fits in L1 TLB and the translation overhead is effectively zero. For a process with many active virtual memory areas (a large application with hundreds of mapped regions), stlb pressure causes measurable throughput degradation — this is why Linux's **huge pages** (2MB pages using PSE extension) dramatically improve performance for large working sets: 2MB pages occupy only 1 TLB entry instead of 512, reducing stlb pressure by 512x.

Pitfalls Reference — The Five Reasons This Milestone Kills Your Kernel

Pitfall	Why It Kills You	Prevention
Enabling paging without identity map	CPU fetches next instruction via page tables; unmapped → immediate page fault → triple fault	Build identity map covering executing code <i>before</i> CR0.PG
Forgetting TLB flush after PTE modification	Old translation served from TLB; write goes to wrong physical frame; read returns stale data	<code>invlpg</code> after every PTE change; CR3 reload after bulk changes
PMM marks kernel frames as free	Allocator hands out frames occupied by kernel code; next write corrupts executing instructions	Mark frames from <code>__kernel_start_phys</code> to <code>__kernel_end_phys</code> as used during <code>pmm_init</code>
Page table frames not marked used in PMM	Page allocator reuses frames holding your page tables; MMU reads garbage; instant fault	After building boot page tables, mark their physical frames as used in the bitmap
CR2 not read immediately in page fault handler	CR2 is overwritten if another page fault fires (e.g., <code>kprintf</code> causes a fault while printing the first fault)	Save CR2 to a local variable as the very first line of the page fault handler

Knowledge Cascade — What This Milestone Unlocks

You have built the abstraction layer that every system above it depends on. The connections extend in every direction.

1. TLB shootdowns in SMP — the distributed cache invalidation problem

Your single-core kernel flushes one TLB with `invlpg` or a CR3 reload. On a multi-core machine, each core has its own private TLB — a CPU with 8 cores has 8 separate TLB arrays, each caching independent translation snapshots. When core 0 modifies a PTE (say, freeing a page and reassigning its physical frame), cores 1–7 still have the old translation cached. If core 1 accesses that virtual address before its TLB is flushed, it reads from the physical frame that now belongs to a *different* process — a catastrophic security violation.

The solution is **TLB shootdown**: core 0 sends an **IPI (Inter-Processor Interrupt — a hardware mechanism for one CPU core to interrupt another)** to all other cores, triggering an interrupt handler that each core runs to flush its own TLB for the modified address. Linux's `flush_tlb_others()` function does exactly this, using the APIC (Advanced Programmable Interrupt Controller) to broadcast IPIs. The cost: a single `invlpg` on a 16-core machine requires 15 IPIs, each costing ~500 cycles to deliver and process. TLB shootdowns are a significant bottleneck in memory-intensive workloads on many-core systems. This is the hardware analog of distributed cache invalidation — the same problem that makes cache coherence in distributed databases so expensive. Your single-core `invlpg` is the prototype.

2. Demand paging — the page fault handler as a lazy evaluator

Your current page fault handler prints a diagnostic and halts. That is only one possible response. When the kernel knows that a page fault at a given address is expected — because the page has been allocated virtually but not yet backed by a physical frame — the correct response is to allocate a frame, map it, and return from the fault handler. The CPU will retry the faulting instruction, which now succeeds.

This is **demand paging**: physical frames are allocated only when a page is first accessed, not when it is first mapped. `malloc(1000000000)` in Linux returns immediately — Linux maps 1GB of virtual address space but allocates zero physical frames. The first write to any page triggers a fault, the kernel allocates a frame (zeroed, via the zero page copy-on-write mechanism from Milestone 1's cascade), maps it, and returns. Only the pages you actually touch cost physical memory. Your `CR2 → PTE lookup → allocate frame → vmm_map_page → return` sequence is the complete implementation. **Memory-mapped files** (Linux `mmap(fd, ...)`) use the same path: the page fault handler checks if the faulting address corresponds to a file-backed region, reads the page from disk into a fresh frame, maps it, and returns. Every `mmap` in Linux ultimately runs through a path structurally identical to your page fault handler. **Guard pages** (the unmapped page at the end of every thread's stack, which causes a segfault on stack overflow rather than silent corruption) are simply pages with a not-present PTE — P=0.

3. The bitmap allocator's cache behavior — tracing a lineage to jemalloc

Your 128KB bitmap has a critical property: scanning it for a free frame is a sequential read. A CPU hardware prefetcher detects sequential access patterns and loads cache lines before the CPU requests them. The result: after the first scan warms the cache, subsequent allocations read from L2 or L3 cache at ~10ns per cache line rather than ~60ns DRAM. This cache-friendliness is the bitmap's decisive advantage over a free list of physical frames.

The same trade-off governs user-space allocator design. **jemalloc** (used by Firefox, FreeBSD's libc, and Rust's global allocator) uses **size-class arenas**: all objects of size 8, 16, 32, 64, ... 512 bytes each live in separate **slabs** (contiguous 4KB–2MB pages). Allocating a 32-byte object means scanning a slab's free-slot bitmap — sequential, cache-friendly, same as your frame bitmap. **tcmalloc** (Google's allocator, used in Chrome and many internal services) adds per-thread caches of pre-allocated objects to eliminate lock contention entirely. **Linux's SLUB allocator** (the kernel's own object allocator, above the frame allocator) partitions kernel objects by type into per-CPU caches — `kmem_cache_alloc(task_struct_cache)` is a single list pop, costing ~5ns. Your kmalloc's linked-list first-fit scan is the conceptually simplest version of a problem that has driven 40 years of allocator research.

4. Higher-half kernel mapping as the conceptual origin of KASLR

Mapping the kernel at virtual `0xC0000000` while it loads at physical `0x100000` establishes a clean separation: kernel virtual addresses and physical addresses are different numbers, connected only by the page table. This separation is the foundation of **KASLR (Kernel Address Space Layout Randomization)** — instead of always mapping the kernel at `0xC0000000`, the kernel adds a random offset at boot time (say, `0xC0000000 + 0x3A00000`) and builds page tables reflecting that random offset. An attacker who wants to exploit a kernel vulnerability needs to know where kernel code and data live in virtual memory. KASLR defeats this by making the layout unpredictable. On Linux, the randomized kernel virtual base is printed in `/proc/kallsyms` but requires root to read — because knowing the address breaks KASLR. You now understand exactly what that file contains and why it is gated. **SMEP (Supervisor Mode Execution Protection)** and **SMAP (Supervisor Mode Access Protection)** are hardware extensions to the page table flag model you just implemented: SMEP prevents the kernel from executing code at user-mode virtual addresses (U/S=1 pages), preventing jump-to-userspace ROP attacks; SMAP prevents the kernel from reading user-mode memory without explicitly setting the AC flag.

5. kmalloc as the kernel's malloc — why glibc, jemalloc, and SLUB all exist

You just implemented a first-fit heap with block headers. glibc's malloc is structurally similar for the general case: a doubly-linked list of free chunks, with headers encoding size and free status, and first-fit or best-fit search. But glibc adds: bins for common sizes (avoiding scan for 8, 16, 32, 64-byte allocations), **mmap** for large allocations (bypassing the heap entirely for allocations above 128KB, returning them directly to the OS on free), and thread arenas (per-thread heaps to avoid lock contention). The fundamental insight your kernel teaches: **heap allocation is fundamentally a virtual address space management problem backed by page-granular physical memory**. `sbrk()` (the traditional Unix heap expansion syscall) does what your `heap_expand()` does — asks the OS to map one more page at the end of the heap virtual range. Linux `mmap(MAP_ANONYMOUS)` does the same but for arbitrary-address allocations.

Every user-space allocator is building a variable-size allocation service on top of page-granular OS primitives — exactly what you just built for the kernel.

Milestone 4: Processes and Preemptive Scheduling

The Revelation: Two Lies You Believe About Context Switching

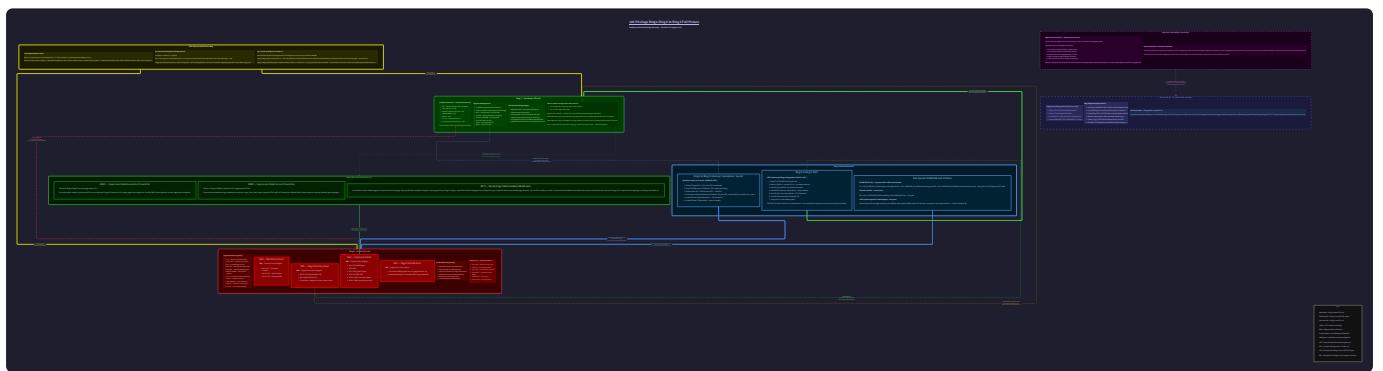
Here is what every developer who has used threads, goroutines, or `async/await` believes about context switching: it is a high-level software operation, roughly like `setjmp / longjmp` — save some state into a struct, restore it later, resume. The OS is in control throughout. The second belief follows naturally: ring 3 to ring 0 transitions are automatic. You call a syscall, the CPU "just knows" to switch to kernel mode, and the kernel runs on some preexisting kernel stack.

Both beliefs are wrong in ways that will corrupt your kernel silently before crashing it spectacularly.

The first lie — context switching is like `setjmp`: A context switch must save every register the CPU has, including `EFLAGS` (the flags register containing the carry flag, zero flag, interrupt flag, and a dozen others), using assembly. Not C. If you write the context switch in C, the C compiler saves and restores the registers it uses — but the CPU still has values in registers the compiler did not touch. When the interrupted process resumes, those registers contain whatever the scheduler left in them. The bug manifests not in the scheduler but in the interrupted process, when it uses a register it believes it saved but was actually clobbered. The failure happens tens of thousands of instructions after the switch. There is no immediate error. This class of bug has driven systems programmers mad for decades.

The second lie — ring transitions are automatic: The CPU does know how to transition from ring 3 to ring 0 when an interrupt fires. What it does not know, without explicit configuration, is *which kernel stack to use*. The CPU hardware reads this from a data structure called the **TSS (Task State Segment — a hardware-defined memory region that the CPU reads directly on privilege transitions)**, specifically from a field called `ESP0` (the ring 0 stack pointer). If you have three processes and TSS.`ESP0` still points to the kernel stack of process 1 when process 2 is running, then every interrupt that fires while process 2 is executing pushes the interrupt frame onto process 1's kernel stack. The kernel handler executes, modifying process 1's stack. Process 1 later resumes from its corrupted stack. The failure is distant, silent, and maddening.

The fix for both problems is the same principle: **the CPU is not a cooperative abstraction layer. It is hardware that executes exactly what you configure, nothing more.** Configuring it correctly requires understanding what it does at the instruction level.



Section 1: The Process Control Block — One Structure to Name Them All

Before you can switch between processes, you need a place to put a process's state when it is not running. This is the **PCB (Process Control Block)** — a kernel data structure that holds everything the CPU needs to restore a process to exactly the state it was in when preempted.

{{DIAGRAM:diag-m4-pcb-structure}}

The PCB is not a software abstraction — it is a hardware requirement with a software body. The CPU gives you nothing for free: no automatic state saving, no process table, no PID. You define the structure; you fill it in; you restore from it. The CPU only helps by executing your assembly code correctly.

What must a PCB store? Start from the CPU's state when an interrupt fires and work backward:

CPU register state — everything the CPU had in its registers at the moment the timer IRQ fired. This includes:

- All 8 general-purpose registers: `EAX`, `EBX`, `ECX`, `EDX`, `ESI`, `EDI`, `EBP`, `ESP`
- The instruction pointer: `EIP` (where to resume execution)
- The flags register: `EFLAGS` (interrupt enable state, condition codes, I/O privilege level)
- Segment registers: `CS`, `DS`, `ES`, `FS`, `GS`, `SS` (critical for distinguishing ring 0 from ring 3 processes)

Memory state — which address space this process lives in:

- The physical address of this process's page directory (`CR3` value)
- When the scheduler restores a process, loading its `CR3` value simultaneously switches the entire address space

Scheduling state — metadata the scheduler needs:

- PID (Process IDentifier — a unique integer name for this process)
- Process state: `READY` (can run), `RUNNING` (currently executing), `BLOCKED` (waiting for I/O or event)
- Kernel stack pointer (the kernel stack is per-process; TSS.ESP0 must point here)
- Optionally: priority, time slice remaining, accumulated CPU time

```
/* process.h */

#ifndef PROCESS_H

#define PROCESS_H


#include <stdint.h>

#include "page.h" /* pde_t, PAGE_SIZE */

#define MAX_PROCESSES 16

#define KERNEL_STACK_SIZE 4096 /* 4KB per-process kernel stack */


/* Process states */

typedef enum {

    PROCESS_UNUSED = 0, /* Slot in process table is empty */

    PROCESS_READY = 1, /* Can be scheduled; not currently running */

    PROCESS_RUNNING = 2, /* Currently executing on CPU */

    PROCESS_BLOCKED = 3, /* Waiting for an event (I/O, sleep, etc.) */

    PROCESS_ZOMBIE = 4, /* Finished execution; waiting for parent to collect exit code */

} process_state_t;

/* Saved CPU register state — must match the layout that context_switch.asm
pushes and pops. Every field offset is load-bearing. */

typedef struct {

    /* General-purpose registers (saved/restored by our context switch) */

    uint32_t edi; /* Offset 0 */

    uint32_t esi; /* Offset 4 */

    uint32_t ebp; /* Offset 8 */

    uint32_t esp; /* Offset 12 — kernel ESP at time of switch */

    uint32_t ebx; /* Offset 16 */

    uint32_t edx; /* Offset 20 */

    uint32_t ecx; /* Offset 24 */

    uint32_t eax; /* Offset 28 */

    /* CPU-pushed interrupt frame (already on stack when we save state) */

    uint32_t eip; /* Offset 32 — where to resume */

    uint32_t cs; /* Offset 36 — code segment (ring 0: 0x08, ring 3: 0x1B) */

    uint32_t eflags; /* Offset 40 — must have IF=1 for resumed process to receive IRQs */

}
```

```

/* Only present if process was in ring 3 when preempted */

uint32_t user_esp; /* Offset 44 - user-mode stack pointer */

uint32_t user_ss; /* Offset 48 - user-mode stack segment */

} cpu_state_t;

/* The Process Control Block */

typedef struct process {

/* Identity */

uint32_t pid; /* Unique process identifier */

char name[32]; /* Human-readable name for debugging */

/* Scheduling */

process_state_t state; /* Current lifecycle state */

uint32_t ticks_remaining; /* Time slice: decremented by timer IRQ */

/* CPU state - filled in by context_switch on preemption */

cpu_state_t cpu; /* Saved register state */

/* Memory */

pde_t *page_directory; /* Physical address of this process's PD */

/* Kernel stack - each process has its own; TSS.ESP0 points here */

uint8_t kernel_stack[KERNEL_STACK_SIZE] __attribute__((aligned(16)));

uint32_t kernel_stack_top; /* = (uint32_t)kernel_stack + KERNEL_STACK_SIZE */

} process_t;

/* The global process table */

extern process_t process_table[MAX_PROCESSES];

extern process_t *current_process; /* Pointer to the currently running PCB */

/* Public API */

process_t *process_create_kernel(const char *name, void (*entry)(void));

process_t *process_create_user(const char *name, void (*entry)(void));

void process_exit(int exit_code);

#endif

```

Memory layout of the PCB: On a 32-bit system, `cpu_state_t` is 52 bytes (13 fields \times 4 bytes). The kernel stack is 4096 bytes. Total PCB size $\approx 4 + 32 + 4 + 4 + 52 + 4 + 4096 + 4 = \sim 4200$ bytes per process. For 16 processes: $\sim 67\text{KB}$ of kernel BSS. This fits in the 128KB kernel region with room to spare. The `__attribute__((aligned(16)))` on the kernel stack ensures the stack pointer is 16-byte aligned from the start, satisfying the System V ABI's calling convention requirement — if your kernel calls any function that uses SSE instructions, an unaligned stack causes a General Protection Fault.

Why `kernel_stack_top` is stored in the PCB: The kernel stack grows downward. `kernel_stack[0]` is the lowest address; `kernel_stack[KERNEL_STACK_SIZE - 1]` is the highest. A stack that starts "at the top" begins at `kernel_stack + KERNEL_STACK_SIZE` and grows toward `kernel_stack`. When you initialize a new process's kernel stack, you push the initial register values from `kernel_stack_top` downward. When TSS.ESP0 must be updated to point to this process's kernel stack, you use `kernel_stack_top` — not `kernel_stack`.

Section 2: The Context Switch — Assembly Is Not Optional

This is the most conceptually important routine in the entire OS. Every developer who has implemented a context switch in C first, then spent two days debugging register corruption, eventually arrives at the same conclusion: the context switch must be written in assembly.



Here is why C is insufficient. The C calling convention (for x86 System V ABI) specifies that `EAX`, `ECX`, and `EDX` are **caller-saved**: the calling function assumes the callee will trash them, so the caller saves them before the call if it needs them afterward. `EBX`, `ESI`, `EDI`, `EBP` are **callee-saved**: the callee must preserve them across the call.

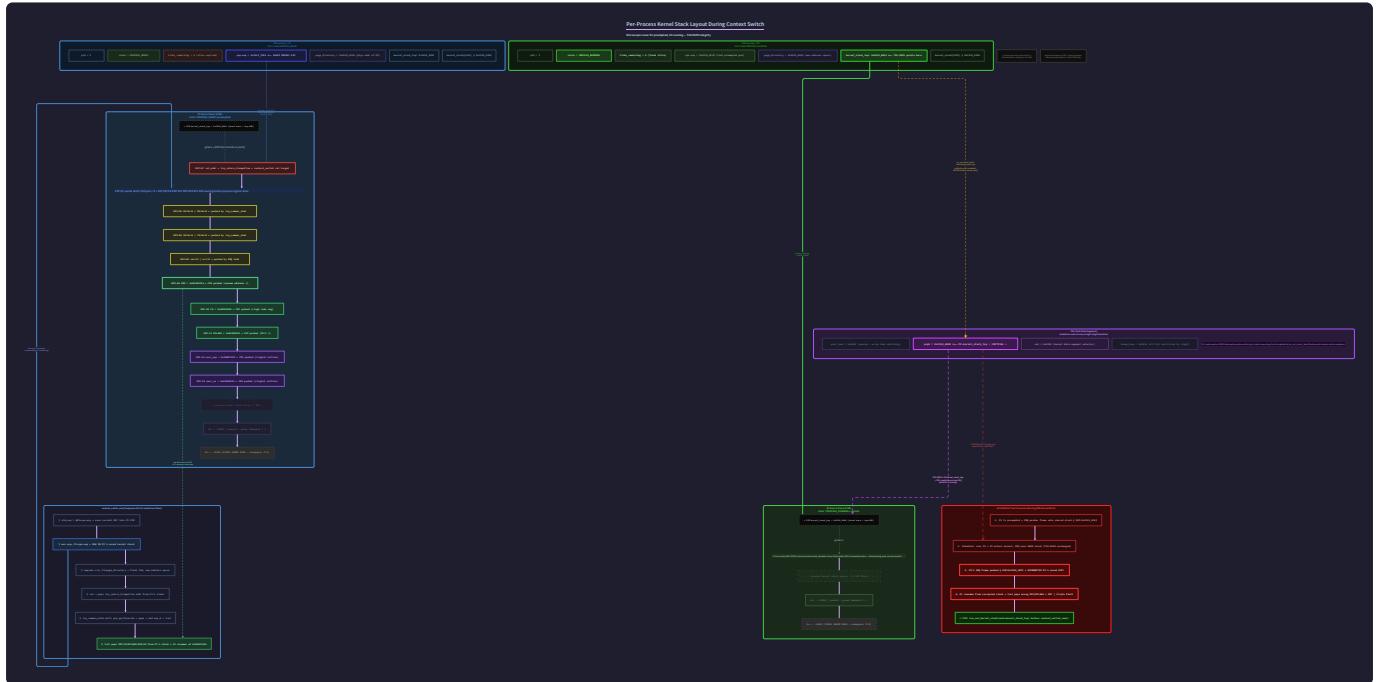
If you write `void context_switch(process_t *from, process_t *to)` in C, the compiler generates code that:

1. Saves `EBX`, `ESI`, `EDI`, `EBP` on the stack (callee-saved convention)
2. Does your switching logic
3. Restores those four registers from the stack and executes `ret`

What the compiler does *not* save: `EAX`, `ECX`, `EDX` (caller-saved — the caller is supposed to handle them), and critically, **EFLAGS** (the flags register that no C calling convention saves, because C code does not assume flag values persist across function calls).

But the interrupted process's caller did not save `EAX`, `ECX`, `EDX` and `EFLAGS` — it was in the middle of executing arbitrary code when the timer IRQ fired. Those registers contain that process's live state. When a context switch written in C trashes `EAX` because it used `EAX` for intermediate calculations, and then restores the process later without `EAX`'s original value, the resumed process runs with the wrong value in `EAX`. Depending on what it was computing, this is anywhere from harmless to fatal.

The solution: the context switch takes place at the interrupt boundary, where the ISR stack frame is already laid out. You save *everything* in assembly before C code touches a single register, and restore *everything* in assembly after C code is done.



The Mechanical Design

The timer IRQ handler (from Milestone 2) already saves all general-purpose registers via `pusha` and the segment registers explicitly. At that point, the stack contains a complete snapshot of the interrupted process's CPU state. The context switch does not need to save anything — it just needs to:

1. Record the current stack pointer in the current process's PCB (so we can find the saved state later)
2. Call the C scheduler to select the next process
3. Update TSS.ESP0 to the next process's kernel stack top
4. If switching address spaces, load the new process's `CR3`
5. Load the new process's saved stack pointer
6. Execute `iret` — which restores `EFLAGS`, `CS`, `EIP` (and `SS`, `ESP` for ring 3) from the new process's saved state

This design means the context switch is not a separate "save all registers then restore all registers" operation — it rides the back of the IRQ handler's existing save/restore infrastructure. The IRQ stub saves state on entry and restores state on exit. The context switch hijacks the "exit" phase: instead of restoring the current process's state, it pivots to the next process's saved state.

```

/* sched.h */

#ifndef SCHED_H

#define SCHED_H


#include "process.h"

void sched_init(void);

void sched_tick(void); /* Called from timer IRQ handler */

void sched_yield(void); /* Called by kernel code to voluntarily yield */

/* Declared in context_switch.asm; never call directly */

extern void context_switch_asm(uint32_t *old_esp, uint32_t new_esp, uint32_t new_cr3);

#endif

```

C

```

; context_switch.asm
; void context_switch_asm(uint32_t *old_esp, uint32_t new_esp, uint32_t new_cr3)
;
; Called FROM WITHIN the IRQ common stub, AFTER pusha and segment register saves.
; The stack at this point contains the complete saved state of the current process.
; Arguments on stack (cdecl): [esp+4]=old_esp ptr, [esp+8]=new_esp, [esp+12]=new_cr3

[bits 32]
[global context_switch_asm]

context_switch_asm:
    ; — Save the current process's kernel stack pointer —
    ; After the cdecl call setup, esp points to our return address.
    ; The caller's esp (which holds the saved process state) is at
    ; the value esp had BEFORE the call pushed the return address.
    ; cdecl: [esp+4]=old_esp_ptr, [esp+8]=new_esp, [esp+12]=new_cr3
    mov eax, [esp + 4]    ; EAX = address of current process's saved_esp field
    mov [eax], esp        ; *old_esp = current esp (kernel stack pointer position)
                           ; This records where the saved state lives on the stack.

    ; — Switch to the next process's kernel stack —
    mov esp, [esp + 8]    ; Wait – after we load [esp+8], esp changes!
                           ; BUG: we need to save [esp+8] before modifying esp.
                           ; Fix: load arguments first.
    ; (Correct implementation below – argument loading before esp change)

    ; — Load new CR3 if address space is different —
    mov ecx, [esp + 12]    ; ECX = new_cr3
    ; ... (see corrected version below)

```

NASM

The naive approach above has a subtle bug: reading arguments from `[esp+N]` after modifying `esp` reads wrong offsets. The corrected version loads all arguments into registers before touching `esp`:

```

; context_switch.asm - CORRECT implementation
[bits 32]
[global context_switch_asm]

context_switch_asm:
    ; Load all three arguments before touching ESP
    ; (cdecl: return addr at [esp], args at [esp+4], [esp+8], [esp+12])
    mov eax, [esp + 4]      ; EAX = &current->cpu.esp (where to save current esp)
    mov ecx, [esp + 8]      ; ECX = next->cpu.esp     (new esp to restore)
    mov edx, [esp + 12]     ; EDX = next->page_directory (new CR3, physical addr)

    ; — Step 1: Save current process kernel stack pointer —
    ; esp currently points to our return address (top of IRQ handler's saved state).
    ; Saving esp here allows us to find the saved register state on resume.
    mov [eax], esp

    ; — Step 2: Switch to next process's kernel stack —
    mov esp, ecx            ; From this point, we are on the next process's stack.
                            ; The "current" process is now unreachable until resumed.

    ; — Step 3: Switch address space (load next process's page directory) —
    ; Only flush TLB if the address space is actually changing.
    ; (Comparing CR3 to new CR3 avoids unnecessary TLB flush for kernel threads
    ; that share the kernel page directory.)
    mov eax, cr3
    cmp eax, edx
    je .same_address_space ; Skip CR3 reload if page directory is the same
    mov cr3, edx           ; Load new page directory – flushes non-global TLB

.same_address_space:

    ; — Step 4: Return —
    ; At this point:
    ; - esp points to the next process's saved stack state
    ; - The stack holds: [return address to irq_common_stub]
    ;                     then the next process's saved registers (pushed by pusha)
    ;                     then the CPU-pushed interrupt frame (EFLAGS, CS, EIP [, SS, ESP])
    ; Returning from this function pops the return address and jumps back to
    ; irq_common_stub, which will then:
    ;   pop gs, fs, es, ds  (restore segment registers)
    ;   popa                (restore general-purpose registers)
    ;   add esp, 8          (pop vector + error code)
    ;   iret                (restore EIP, CS, EFLAGS [, ESP, SS for ring 3])
    ; – all from the NEXT process's saved state on the new stack.

    ret

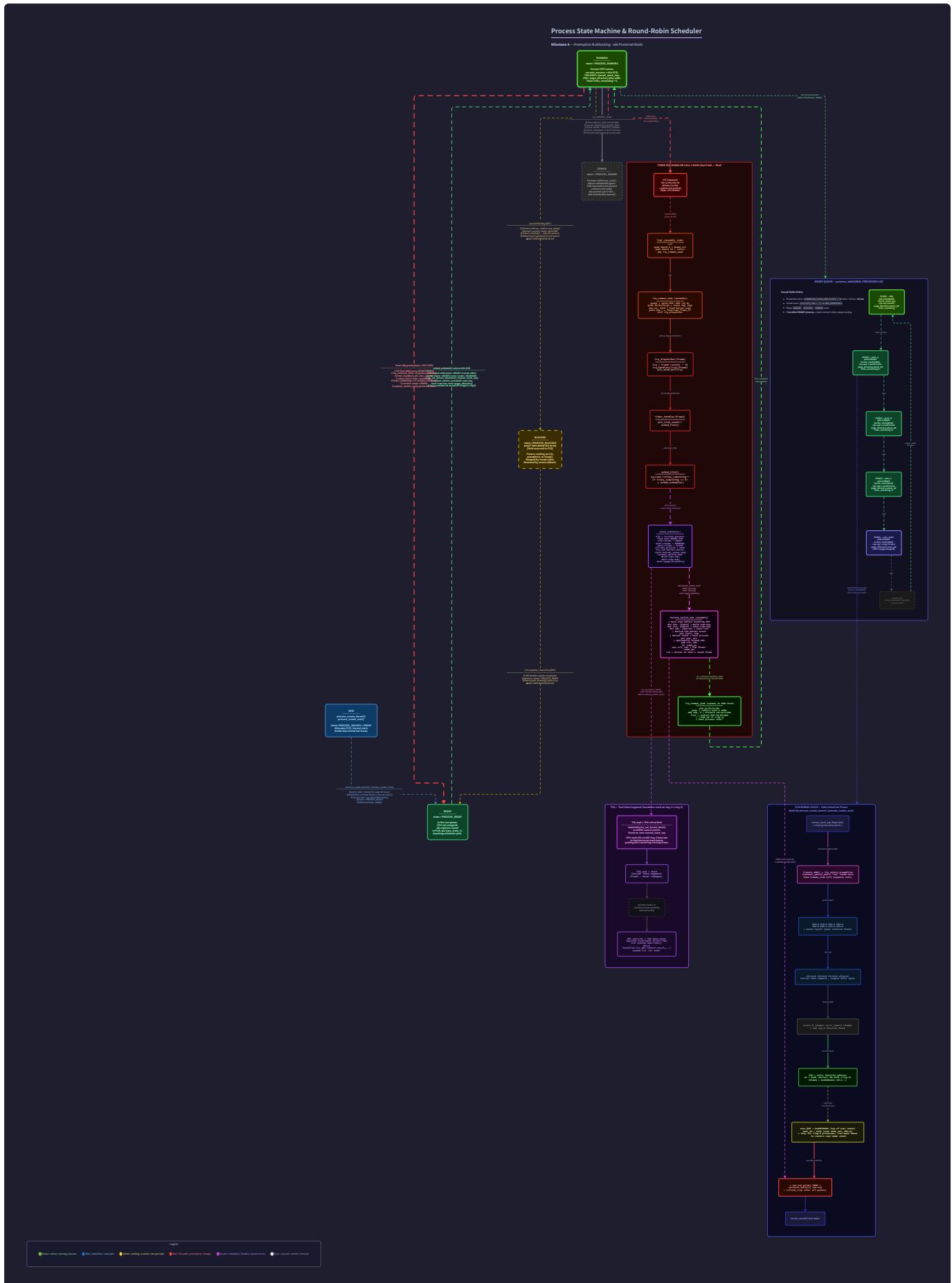
```

The key insight: `context_switch_asm` does not save or restore any general-purpose registers itself. The IRQ stub already saved them with `pusha` before calling the C dispatcher, and will restore them with `popa` after `context_switch_asm` returns. By switching `esp` to the next process's kernel stack, we redirect the subsequent `popa` and `iret` to restore the *next* process's registers. The switch happens between the `call context_switch_asm` and the `ret` — one line of assembly changes which process's context `iret` will restore.

Hardware Soul: The `mov esp, ecx` (stack pointer swap) is the most performance-critical moment. After this instruction, every subsequent stack access — `ret` pops the return address, `popa` reads 32 bytes, `iret` reads 12–20 bytes — goes to the *next* process's kernel stack, which is in a different physical page. If that page is cold (not in L1 cache), the cache miss costs ~60ns (L3 hit) or ~100ns (DRAM). On a 100Hz scheduler at 10ms intervals, the stack is almost certainly cold — it has not been touched in 10 million nanoseconds. This cold-stack startup cost is unavoidable; it is one of the fundamental reasons context switching has nonzero overhead even when the assembly is perfect.

Section 3: The Scheduler — Round-Robin Over a Process Table

With the context switch mechanism defined, the scheduler decides *which* process to switch to. Round-robin is the simplest fair scheduler: cycle through the process table in order, giving each ready process one time slice, then move to the next.



```
/* sched.c - Round-robin preemptive scheduler */

#include "sched.h"

#include "process.h"

#include "tss.h"

#include "kprintf.h"

process_t process_table[MAX_PROCESSES];

process_t *current_process = NULL;

static int sched_initialized = 0;

void sched_init(void) {

    /* Zero all PCB slots – state = PROCESS_UNUSED */

    for (int i = 0; i < MAX_PROCESSES; i++) {

        process_table[i].state = PROCESS_UNUSED;

        process_table[i].pid = 0;

    }

    sched_initialized = 1;

    kprintf("[SCHED] Scheduler initialized (%d slots).\n", MAX_PROCESSES);

}

/* Called from the timer IRQ handler (IRQ0, every 10ms at 100Hz) */

void sched_tick(void) {

    if (!sched_initialized || !current_process) return;

    /* Decrement the current process's time slice */

    if (current_process->ticks_remaining > 0) {

        current_process->ticks_remaining--;

        if (current_process->ticks_remaining > 0) {

            return; /* Slice not expired – keep running current process */

        }

    }

    /* Time slice expired: find the next ready process */

    sched_schedule();

}


```

```

/* Select and switch to the next ready process (round-robin) */

void sched_schedule(void) {

    process_t *old = current_process;

    /* Find the index of the current process in the table */

    int current_idx = (int)(old - process_table);

    /* Search for next ready process, wrapping around */

    process_t *next = NULL;

    for (int i = 1; i <= MAX_PROCESSES; i++) {

        int idx = (current_idx + i) % MAX_PROCESSES;

        if (process_table[idx].state == PROCESS_READY) {

            next = &process_table[idx];

            break;
        }
    }

    if (!next) {

        /* No other ready process – continue running current */

        /* Reset its time slice so it gets a full new slice */

        current_process->ticks_remaining = SCHEDULER_TICKS_PER_SLICE;

        return;
    }
}

/* Transition states */

if (old->state == PROCESS_RUNNING) {

    old->state = PROCESS_READY;
}

next->state      = PROCESS_RUNNING;

next->ticks_remaining = SCHEDULER_TICKS_PER_SLICE;

current_process     = next;

/* — Critical: update TSS.ESP0 to point to the NEXT process's kernel stack —

This must happen BEFORE context_switch_asm, because if an interrupt fires

```

```

        during or after the switch, the CPU reads TSS.ESP0 to find the kernel stack.

        If TSS.ESP0 still points to the OLD process's stack, the interrupt handler
        will push its frame there — corrupting the old process's saved state. */

tss_set_kernel_stack(next->kernel_stack_top);

/* — Perform the assembly context switch —

Pass: address to save old esp, new esp value, new CR3 value */

context_switch_asm(
    &old->cpu.esp,                                /* Where to save old kernel ESP */
    next->cpu.esp,                                /* New kernel ESP to restore      */
    (uint32_t)(uintptr_t)next->page_directory /* New CR3 (physical address)   */
);

/* — Execution resumes here when THIS process is rescheduled —

(The above call returns only when the old process runs again) */

}

#define SCHEDULER_TICKS_PER_SLICE 5    /* 5 timer ticks × 10ms = 50ms per slice */

```

Why `sched_tick` and `sched_schedule` are separate: `sched_tick` is called on every timer IRQ — 100 times per second. It decrements the slice counter and returns early (within ~10 cycles) if the slice has not expired. Separating the tick accounting from the scheduling decision keeps the common path (no context switch needed) extremely fast. `sched_schedule` is called only when a switch is warranted, and can also be called voluntarily by blocking code (`sched_yield()`) when a process needs to wait for I/O.

The timer handler in Milestone 2 now calls the scheduler:

```

/* irq.c - updated timer handler */

void timer_handler(interrupt_frame_t *frame) {
    (void)frame;

    pit_tick_count++;

    sched_tick();    /* Potentially triggers a context switch */
}

```

Interrupts during `sched_schedule` : The timer IRQ fires with interrupts disabled (interrupt gate, IF cleared). `sched_schedule` runs with interrupts off. After `context_switch_asm` switches to the new process and `iret` executes, EFLAGS is restored — including the `IF` bit. If the new process had interrupts enabled (which it should, since it was running normal code when preempted), `iret` re-enables interrupts atomically as part of restoring EFLAGS. This is the correct behavior — interrupts are masked for the minimum possible duration (only during the scheduler's critical section).

Section 4: The TSS — Hardware That Reads Your Configuration on Every Ring Transition

The **TSS (Task State Segment)** is a 104-byte Intel-defined data structure that the CPU reads directly from memory on every ring 3 to ring 0 privilege transition. It is not a software abstraction with a hardware analog — it is a hardware register with a memory backing that you must maintain.



When any interrupt or exception fires while the CPU is executing ring 3 code, the CPU must switch to a kernel stack before it can push the interrupt frame. It cannot use the user-mode stack because user-mode code is untrusted — a process could set `ESP` to a kernel address and trick the interrupt handler into writing a kernel stack frame into kernel memory at an attacker-controlled address. The TSS is the hardware's solution: a trusted, kernel-maintained structure that specifies which stack to use.

Why not just use a fixed kernel stack? If all processes shared one kernel stack, the following would occur: process A is interrupted mid-syscall, the handler pushes a frame onto the shared stack and saves A's kernel state there. Process B runs and triggers a syscall, pushing its frame onto the same stack — immediately overwriting A's saved state. You need one kernel stack per process. The TSS.`ESP0` field tells the CPU which one to use for the currently running process.

The full TSS structure is defined by Intel (it is used for hardware task switching in very old x86 code, but modern OSes use it exclusively for the `ESP0` field):

```
/* tss.h */

#ifndef TSS_H

#define TSS_H


#include <stdint.h>

/* Intel TSS descriptor – hardware-defined layout, do not reorder */

typedef struct __attribute__((packed)) {

    uint32_t prev_task;      /* Previous TSS selector (for hardware task switching; unused) */

    uint32_t esp0;           /* Ring 0 stack pointer – THE critical field */

    uint32_t ss0;            /* Ring 0 stack segment selector (must be kernel data: 0x10) */

    uint32_t esp1;           /* Ring 1/2 stack pointers – unused (no ring 1/2 in our kernel) */

    uint32_t ss1;

    uint32_t esp2;

    uint32_t ss2;

    uint32_t cr3;           /* Page directory (not used by CPU here; we manage CR3 ourselves) */

    uint32_t eip;

    uint32_t eflags;

    uint32_t eax, ecx, edx, ebx;

    uint32_t esp, ebp, esi, edi;

    uint32_t es, cs, ss, ds, fs, gs;

    uint32_t ldt;           /* LDT selector (unused) */

    uint16_t debug_trap;     /* Debug trap flag */

    uint16_t iomap_base;     /* I/O permission bitmap offset */

} tss_t;

extern tss_t kernel_tss;

void tss_init(void);

void tss_set_kernel_stack(uint32_t esp0);

#endif
```

```

/* tss.c */

#include "tss.h"

#include "gdt.h"      /* To install the TSS descriptor in the GDT */

#include "kprintf.h"

tss_t kernel_tss;

void tss_init(void) {

    /* Zero all fields – most are unused in software-task-switching mode */

    uint8_t *p = (uint8_t *)&kernel_tss;

    for (size_t i = 0; i < sizeof(kernel_tss); i++) p[i] = 0;

    /* Configure the only field the CPU reads on ring transitions */

    kernel_tss.sso = 0x10;      /* Kernel data segment selector */

    kernel_tss.esp0 = 0;        /* Will be set properly on first process switch */

    /* I/O permission bitmap: offset beyond TSS size = all I/O ports restricted */

    kernel_tss.io_map_base = sizeof(tss_t);

    /* Install the TSS descriptor in the GDT (entry index 5, after the 4 existing entries)

       The TSS GDT entry is a "system" descriptor (S=0) with type=0x9 (TSS, not busy) */

    gdt_install_tss(5, (uint32_t)&kernel_tss, sizeof(kernel_tss) - 1);

    /* Load the Task Register – tells the CPU which GDT entry describes the TSS

       Selector: index 5, GDT, RPL 0 → (5 << 3) | 0 = 0x28 */

    __asm__ volatile ("ltr %0" : : "r"((uint16_t)0x28));

    kprintf("[TSS] Initialized. TSS at 0x%08x, selector 0x28.\n", &kernel_tss);

}

void tss_set_kernel_stack(uint32_t esp0) {

    /* Called on every context switch.

       The CPU reads this field on the NEXT ring 3 → ring 0 transition.

       Must be set to the TOP of the new process's kernel stack. */

    kernel_tss.esp0 = esp0;

}

```

The GDT must be extended to include a TSS descriptor at index 5. The TSS descriptor has a different format from code/data descriptors — it is a "system" descriptor (S=0):

```
/* gdt.c - add this function */

void gdt_install_tss(int index, uint32_t base, uint32_t limit) {

    /* TSS descriptor: S=0 (system), Type=0x9 (available 32-bit TSS), DPL=0 */

    gdt_entries[index].limit_low    = limit & 0xFFFF;
    gdt_entries[index].base_low     = base & 0xFFFF;
    gdt_entries[index].base_middle  = (base >> 16) & 0xFF;
    gdt_entries[index].access       = 0x89;      /* Present, DPL=0, S=0, Type=0x9 */
    gdt_entries[index].granularity = (limit >> 16) & 0x0F;    /* No 4KB granularity for TSS */
    gdt_entries[index].base_high   = (base >> 24) & 0xFF;

    /* Reload GDTR to make the new entry visible */

    gdt_flush();

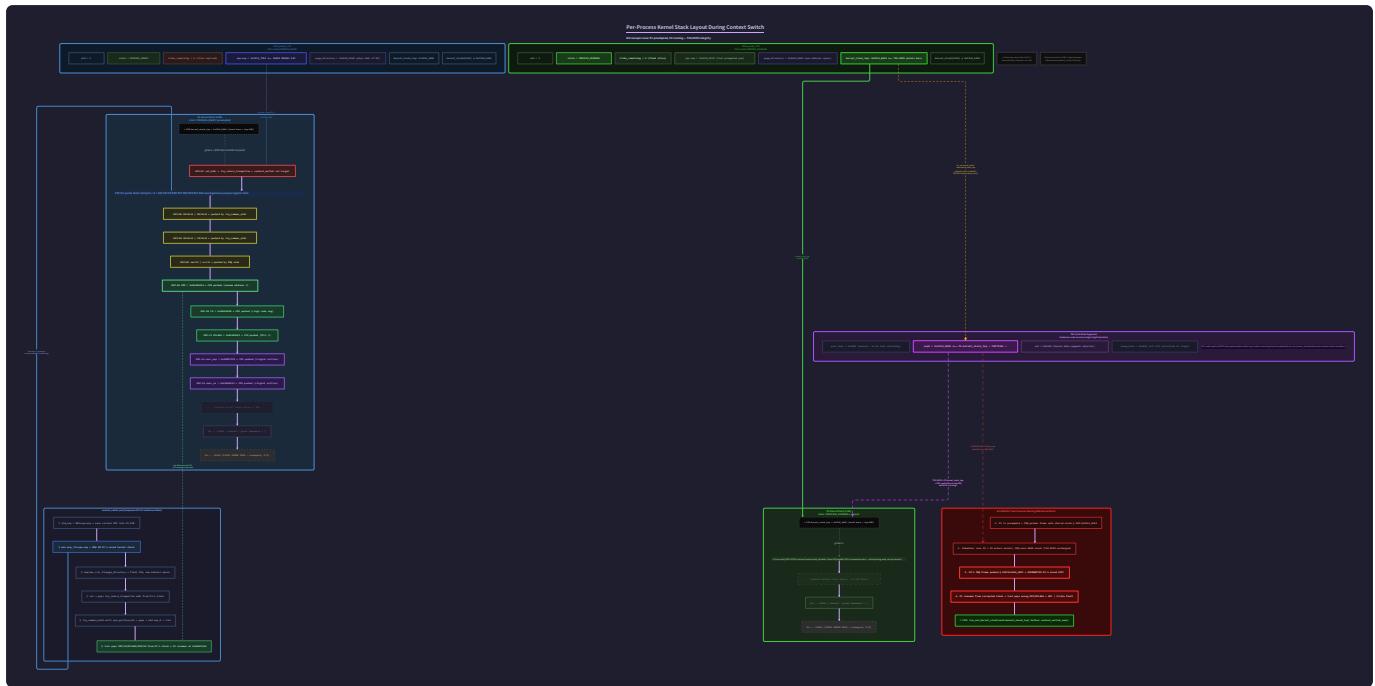
}
```

ltr — Load Task Register: The CPU has a **TR (Task Register)** that caches the TSS descriptor (like CS caches the code segment descriptor). Loading `TR` with the TSS selector causes the CPU to read the GDT descriptor and cache it. After `ltr`, when any ring 3 → 0 transition occurs, the CPU uses the cached TSS descriptor to find `kernel_tss.esp0` — it does not walk the GDT again. This means `tss_set_kernel_stack()` (which modifies the in-memory TSS) works correctly: the CPU re-reads `esp0` from memory on each transition, using the cached TSS *base address* from TR but fetching `esp0` fresh.

Hardware Soul: `tss_set_kernel_stack()` is a single 32-bit memory write — `mov [kernel_tss.esp0], new_value`. This write goes to a known, hot address in the kernel data section. It is almost certainly in L1 cache (the TSS is accessed frequently — on every ring transition). The CPU reads `esp0` from this address on the very next ring 3 interrupt. The write must complete before the CPU takes that interrupt, which is guaranteed because: (a) we are in ring 0 kernel code running with interrupts off during the scheduler, and (b) the memory write is not reordered past an `iret` instruction (a full memory barrier). No `sfence` or `mfence` is needed because x86 stores are ordered with respect to subsequent instruction execution on the same core.

Section 5: Creating Kernel Processes — Stack Setup and Initial State

To schedule a process, you need its PCB in a state where `context_switch_asm` can restore it. For the very first activation of a new process — before it has ever run — there is no "saved state from a previous preemption." You must manufacture a fake saved state on the process's kernel stack that looks exactly like what the IRQ stub would have pushed.



When the scheduler switches to a new process for the first time, `context_switch_asm` loads `next->cpu.esp`, then `ret` (popping the return address back to `irq_common_stub`), then the stub executes `popa` (restoring the fake registers), then `add esp, 8` (discarding the fake vector and error code), then `iret` (restoring `EFLAGS`, `CS`, `EIP` from the fake values you pushed). The `iret` jumps to the process's entry function at ring 0.

The fake initial stack must look exactly like an IRQ stack frame:

```
/* process.c - creating a kernel-mode process */

#include "process.h"

#include "sched.h"

#include "vmm.h"

#include "kprintf.h"

#include <string.h>

process_t process_table[MAX_PROCESSES];

process_t *current_process = NULL;

static uint32_t next_pid = 1;

process_t *process_create_kernel(const char *name, void (*entry)(void)) {

    /* Find a free slot */

    process_t *proc = NULL;

    for (int i = 0; i < MAX_PROCESSES; i++) {

        if (process_table[i].state == PROCESS_UNUSED) {

            proc = &process_table[i];

            break;
        }
    }

    if (!proc) {

        kprintf("[PROC] FATAL: Process table full!\n");

        return NULL;
    }

    /* Initialize the PCB */

    memset(proc, 0, sizeof(process_t));

    proc->pid = next_pid++;

    proc->state = PROCESS_READY;

    proc->ticks_remaining = SCHEDULER_TICKS_PER_SLICE;

    /* Copy name, truncating to fit */

    for (int i = 0; i < 31 && name[i]; i++) proc->name[i] = name[i];

    proc->name[31] = '\0';
}
```

```

/* Kernel processes share the boot page directory (all kernel memory mapped) */

extern pde_t boot_page_directory[];

proc->page_directory = boot_page_directory;

/* — Build the fake initial stack frame —

The kernel stack grows downward from kernel_stack_top.

We push values from the highest address downward, mimicking what
irq_common_stub and the CPU would have pushed during an IRQ. */

proc->kernel_stack_top = (uint32_t)proc->kernel_stack + KERNEL_STACK_SIZE;

/* Use a uint32_t pointer, decrementing before each push (like a real stack) */

uint32_t *sp = (uint32_t *)(uintptr_t)proc->kernel_stack_top;

/* CPU-pushed interrupt frame (iret will consume these) */

*--sp = 0x00000202;           /* EFLAGS: IF=1 (interrupts enabled), reserved bit 1 set */
*--sp = 0x08;                 /* CS: kernel code segment selector */
*--sp = (uint32_t)entry;      /* EIP: process entry point - where iret jumps to */

/* Error code + vector (discarded by 'add esp, 8' in irq_common_stub) */

*--sp = 0;                   /* Vector number (dummy) */
*--sp = 0;                   /* Error code (dummy) */

/* Segment registers (popped by the stub after popa) */

*--sp = 0x10;                /* GS = kernel data */
*--sp = 0x10;                /* FS = kernel data */
*--sp = 0x10;                /* ES = kernel data */
*--sp = 0x10;                /* DS = kernel data */

/* General-purpose registers (restored by popa) */

*--sp = 0;      /* EDI */
*--sp = 0;      /* ESI */
*--sp = 0;      /* EBP */
*--sp = 0;      /* ESP (value popa restores into ESP is ignored by hardware) */
*--sp = 0;      /* EBX */
*--sp = 0;      /* EDX */

```

```

    *--sp = 0;      /* ECX */

    *--sp = 0;      /* EAX */

    /* The return address for context_switch_asm's 'ret' instruction.

    context_switch_asm is called FROM irq_common_stub, so 'ret' should
    return to the instruction AFTER 'call context_switch_asm' in the stub.

    We store the address of a trampoline that re-enters the stub's exit sequence. */

extern void irq_return_trampoline(void);    /* Defined in context_switch.asm */

*--sp = (uint32_t)irq_return_trampoline;

/* Save the stack pointer – context_switch_asm will load this */

proc->cpu.esp = (uint32_t)sp;

kprintf("[PROC] Created kernel process '%s' (PID %u), entry 0x%x, esp 0x%x\n",
       proc->name, proc->pid, entry, proc->cpu.esp);

return proc;

}

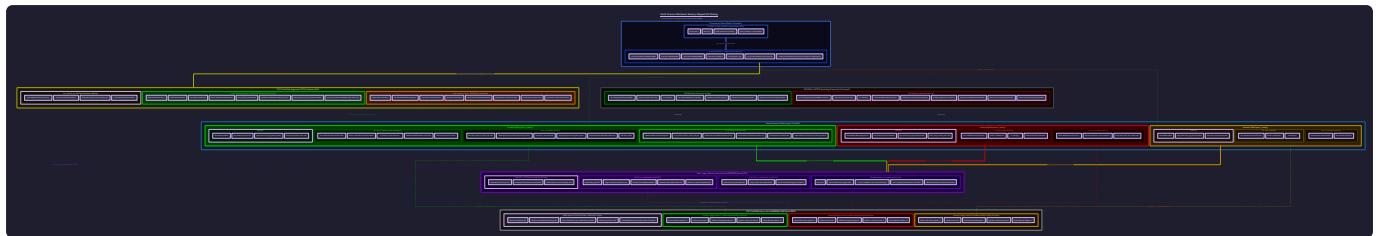
```

The `irq_return_trampoline`: When `context_switch_asm` executes `ret`, it needs to return to a location in `irq_common_stub` that resumes the exit sequence (`pop segment registers, popa, add esp, 8, iret`). For an existing process that was preempted mid-run, the `ret` address on the stack is the real return address pushed when `irq_common_stub` called `context_switch_asm`. For a brand-new process, we push a trampoline address that jumps to the equivalent point in the stub. In practice, this is the address immediately after the `call irq_dispatcher` instruction in `irq_common_stub` — the same point the real scheduler returns to. Alternatively, define a minimal trampoline in assembly that directly pops segment registers, calls `popa`, adds 8, and executes `iret`.

EFLAGS = 0x00000202 : The value `0x202` in EFLAGS means: bit 1 (always 1 per Intel spec), and bit 9 (Interrupt Flag = 1 — interrupts enabled). Every process must resume with IF=1. If you set EFLAGS to 0, the restored process runs with interrupts disabled — the timer never fires again, the scheduler never runs again, and the system freezes with exactly one process running at 100% CPU while the others wait forever.

Section 6: Three Kernel Processes — The Preemptive Multitasking Demo

With process creation and the scheduler wired together, you can demonstrate genuine preemptive multitasking: three processes running concurrently, each unaware of the others, each printing to a different region of the VGA screen.



The critical property: these processes do not cooperate. They do not call `yield()`. They do not check a shared flag. The timer IRQ interrupts them mid-execution — potentially mid-instruction-sequence, mid-loop, mid-function — and the scheduler switches to another process. From each process's perspective, it is the only thing running.

```
/* Three kernel processes for the demo */

static void proc_a_entry(void) {

    /* Process A: counts in the top-left corner */

    uint32_t count = 0;

    while (1) {

        /* Write directly to VGA at row 2, col 0 */

        volatile uint16_t *vga = (volatile uint16_t *)0xB8000;

        char buf[32];

        ksnprintf(buf, sizeof(buf), "Process A: %u      ", count++);

        for (int i = 0; buf[i]; i++) {

            vga[2 * 80 + i] = (uint16_t)buf[i] | (0x0A << 8); /* Green text */

        }

        /* Busy loop - intentionally wastes CPU to make preemption visible */

        for (volatile int delay = 0; delay < 1000000; delay++);

    }

}

static void proc_b_entry(void) {

    /* Process B: counts in the middle of the screen */

    uint32_t count = 0;

    while (1) {

        volatile uint16_t *vga = (volatile uint16_t *)0xB8000;

        char buf[32];

        ksnprintf(buf, sizeof(buf), "Process B: %u      ", count++);

        for (int i = 0; buf[i]; i++) {

            vga[12 * 80 + i] = (uint16_t)buf[i] | (0x0C << 8); /* Red text */

        }

        for (volatile int delay = 0; delay < 1000000; delay++);

    }

}

static void proc_c_entry(void) {

    /* Process C: counts at the bottom of the screen */

}
```

```

        uint32_t count = 0;

    while (1) {

        volatile uint16_t *vga = (volatile uint16_t *)0xB8000;

        char buf[32];

        ksnprintf(buf, sizeof(buf), "Process C: %u      ", count++);

        for (int i = 0; buf[i]; i++) {

            vga[22 * 80 + i] = (uint16_t)buf[i] | (0x0E << 8); /* Yellow text */

        }

        for (volatile int delay = 0; delay < 1000000; delay++);

    }

}

/* In kmain, after all previous milestone initialization: */

void launch_multiprocess_demo(void) {

    sched_init();

    tss_init();

    /* Create three kernel processes */

    process_t *pa = process_create_kernel("proc_a", proc_a_entry);

    process_t *pb = process_create_kernel("proc_b", proc_b_entry);

    process_t *pc = process_create_kernel("proc_c", proc_c_entry);

    if (!pa || !pb || !pc) {

        kprintf("FATAL: Could not create processes.\n");

        return;

    }

    /* Make the kernel's current execution context into "process 0" –
       the idle process. It runs whenever no other process is ready. */

    process_table[0].state      = PROCESS_RUNNING;

    process_table[0].pid        = 0;

    process_table[0].page_directory = boot_page_directory;

    process_table[0].kernel_stack_top =
        (uint32_t)process_table[0].kernel_stack + KERNEL_STACK_SIZE;
}

```

```

current_process = &process_table[0];

/* Update TSS.ESP0 for the initial "process 0" */

tss_set_kernel_stack(current_process->kernel_stack_top);

kprintf("[SCHED] Starting preemptive scheduler. Three processes will run.\n");

kprintf("[SCHED] Watch the VGA screen: rows 2, 12, 22 will count independently.\n");

/* Enable interrupts – the timer will now fire and drive the scheduler */

__asm__ volatile ("sti");

/* Idle loop: the kernel's main thread becomes the idle process */

while (1) {

    __asm__ volatile ("hlt"); /* Sleep until next interrupt */

}

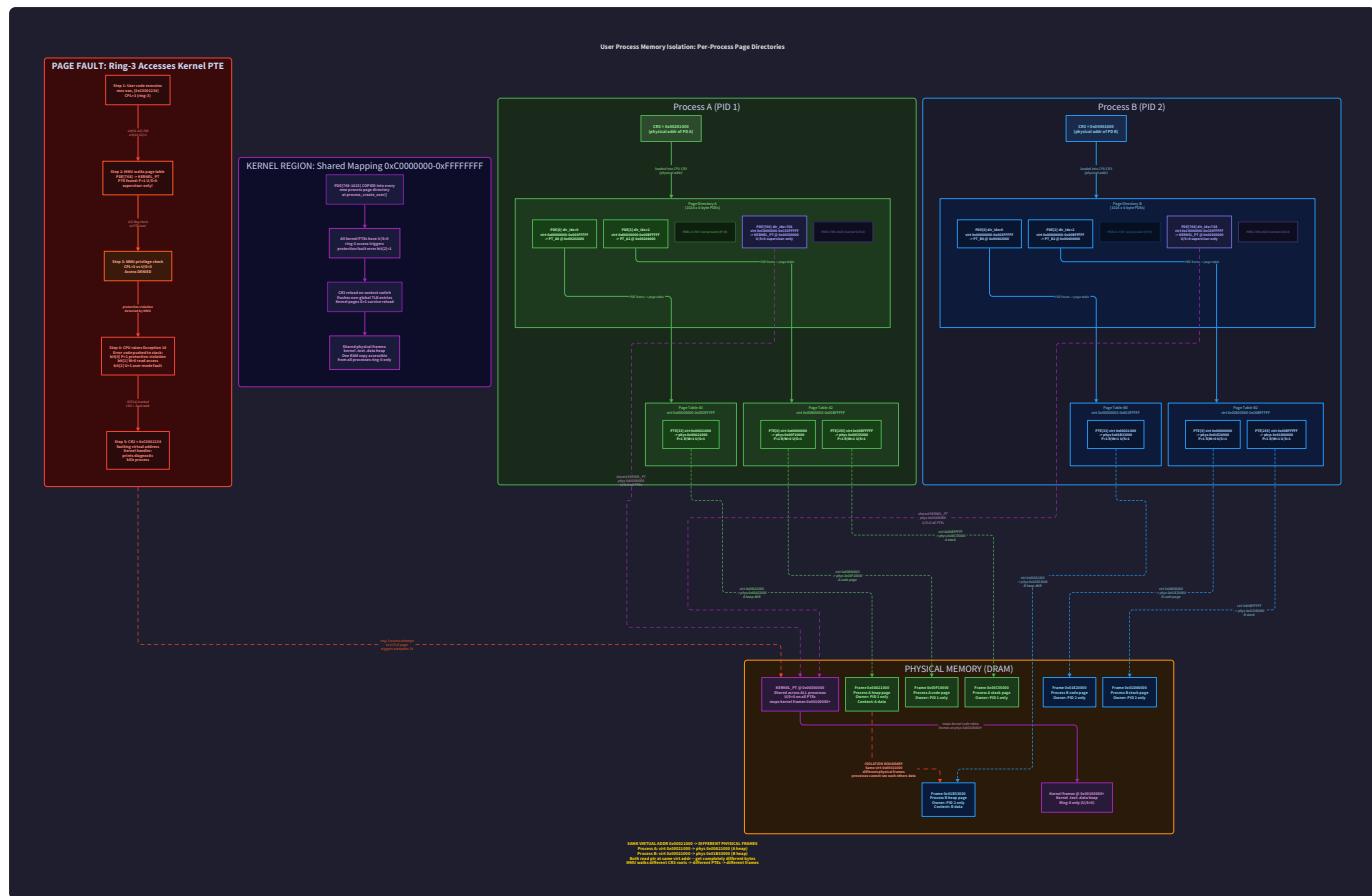
}

```

The idle process pattern: Rather than inventing a special "idle" case in the scheduler, the kernel's main execution thread — the code running in `kmain` after all setup — simply becomes process 0. When all other processes are blocked, the scheduler wraps around to process 0, which executes `hlt` in a loop. `hlt` stops the CPU from executing instructions until the next interrupt arrives, saving power (and improving QEMU's host CPU usage). This is structurally identical to Linux's idle tasks — one per CPU core, running when no other work exists. Linux's idle task also executes `hlt` (or, with hardware support, deeper C-states like `mwait` that drop into lower-power CPU sleep states).

Section 7: User-Mode Processes — Ring 3 Isolation

Kernel processes run in ring 0 with full hardware access and the kernel's page directory. User-mode processes run in ring 3 with restricted hardware access and their own page directory. The kernel code and data are still mapped in the user process's page directory — but with the U/S flag cleared (kernel-only), so ring 3 code that tries to access kernel memory triggers a page fault.



Creating a user-mode process requires four additional steps beyond `process_create_kernel`:

1. **Allocate a new page directory** (from the physical frame allocator)
 2. **Copy the kernel page table entries** into the new directory — so the kernel is reachable during system calls and interrupts (ring 3
→ ring 0 transitions need kernel code and stack accessible)
 3. **Map the user process's code and stack pages** with U/S=1, R/W as appropriate
 4. **Build an initial ring 3 stack frame** that includes `user_esp` and `user_ss` — the two extra fields that `iret` pops when transitioning from ring 0 back to ring 3

```
process_t *process_create_user(const char *name, void (*entry)(void)) {  
    /* — Step 1: Allocate a new page directory — */  
  
    pde_t *user_pd = (pde_t *)pmm_alloc_frame();  
  
    if (!user_pd) return NULL;  
  
    memset(user_pd, 0, PAGE_SIZE);  
  
    /* — Step 2: Copy kernel PDE entries (upper 1GB: directory indices 768-1023)  
     * These entries point to kernel page tables – shared across all processes.  
     * Marked supervisor-only (U/S=0 in the page tables themselves). — */  
  
    extern pde_t boot_page_directory[];  
  
    for (int i = 768; i < 1024; i++) {  
  
        user_pd[i] = boot_page_directory[i];  
  
    }  
  
    /* — Step 3: Map user code at virtual 0x00400000 (conventional user code base) —  
     * Allocate a frame for the user code, copy the function there. */  
  
    void *user_code_phys = pmm_alloc_frame();  
  
    if (!user_code_phys) { pmm_free_frame(user_pd); return NULL; }  
  
    /* Copy the function's code to the user code frame.  
     * Caveat: this copies only the first 4KB starting at 'entry'.  
     * A real OS would load from an ELF file; this is a simplified demo. */  
  
    memcpy(user_code_phys, (void *)entry, PAGE_SIZE);  
  
    vmm_map_page(user_pd,  
                 0x00400000, /* Virtual: user code at 4MB */  
                 (uint32_t)(uintptr_t)user_code_phys, /* Physical: allocated frame */  
                 PAGE_PRESENT | PAGE_USER); /* User-accessible, read-only */  
  
    /* — Step 4: Allocate and map user stack at virtual 0x00BFF000 — */  
  
    void *user_stack_phys = pmm_alloc_frame();  
  
    if (!user_stack_phys) {  
  
        pmm_free_frame(user_code_phys);  
  
        pmm_free_frame(user_pd);  
  
        return NULL;  
    }
```

```

}

memset(user_stack_phys, 0, PAGE_SIZE);

vmm_map_page(user_pd,
              0x00BFF000, /* Virtual: top of user stack */
              (uintptr_t)user_stack_phys,
              PAGE_PRESENT | PAGE_WRITABLE | PAGE_USER);

/* — Step 5: Build PCB (same as kernel process) — */

process_t *proc = NULL;

for (int i = 0; i < MAX_PROCESSES; i++) {
    if (process_table[i].state == PROCESS_UNUSED) {
        proc = &process_table[i];
        break;
    }
}

if (!proc) {
    pmm_free_frame(user_stack_phys);
    pmm_free_frame(user_code_phys);
    pmm_free_frame(user_pd);
    return NULL;
}

memset(proc, 0, sizeof(process_t));
proc->pid = next_pid++;
proc->state = PROCESS_READY;
proc->ticks_remaining = SCHEDULER_TICKS_PER_SLICE;
proc->page_directory = user_pd;
proc->kernel_stack_top = (uint32_t)proc->kernel_stack + KERNEL_STACK_SIZE;
for (int i = 0; i < 31 && name[i]; i++) proc->name[i] = name[i];

/* — Step 6: Build fake ring 3 initial stack frame —
   iret for a ring 3 transition pops FIVE values: EIP, CS, EFLAGS, ESP, SS.
   The extra ESP and SS tell the CPU which user-mode stack to restore. — */

```

```

uint32_t *sp = (uint32_t *)uintptr_t)proc->kernel_stack_top;

/* Ring 3 user stack (iret pops these to switch back to user mode) */

*--sp = 0x23;           /* User SS: user data selector (index 4, RPL 3 → 0x20 | 3 = 0x23) */

*--sp = 0x00C00000;    /* User ESP: top of user stack (one page above the mapped page) */

/* CPU-pushed ring 3 interrupt frame */

*--sp = 0x00000202;    /* EFLAGS: IF=1 */

*--sp = 0x1B;           /* CS: user code selector (index 3, RPL 3 → 0x18 | 3 = 0x1B) */

*--sp = 0x00400000;    /* EIP: user code entry at virtual 0x00400000 */

/* Error code + vector (dummy) */

*--sp = 0;
*--sp = 0;

/* Segment registers – initially kernel data (ring 0 handler fills these;
   after iret the CPU uses CS/SS from the frame, not these) */

*--sp = 0x10;          /* GS */
*--sp = 0x10;          /* FS */
*--sp = 0x10;          /* ES */
*--sp = 0x10;          /* DS */

/* General-purpose registers */

*--sp = 0;             /* EDI */
*--sp = 0;             /* ESI */
*--sp = 0;             /* EBP */
*--sp = 0;             /* ESP (ignored by popa) */
*--sp = 0;             /* EBX */
*--sp = 0;             /* EDX */
*--sp = 0;             /* ECX */
*--sp = 0;             /* EAX */

extern void irq_return_trampoline(void);

*--sp = (uint32_t)irq_return_trampoline;

proc->cpu.esp = (uint32_t)sp;

```

```

kprintf("[PROC] Created user process '%s' (PID %u), entry 0x%x (ring 3)\n",
    proc->name, proc->pid, 0x00400000);

return proc;

}

```

The selector values for ring 3: User code selector `0x1B = (3 << 3) | 3` = GDT index 3, RPL 3. User data selector `0x23 = (4 << 3) | 3` = GDT index 4, RPL 3. The RPL bits in the selector (bits [1:0]) must match DPL=3, or the CPU raises a GPF when loading them. This is the hardware enforcement that prevents user code from loading kernel segment selectors — the GDT descriptor for the kernel code segment has DPL=0, so loading `0x08` (RPL=0) from ring 3 causes an immediate GPF.

Verifying ring 3 isolation: The acceptance criteria require that user-mode access to kernel memory triggers a page fault. The test: in the user-mode process's code (which runs at virtual `0x00400000`), attempt to read from a kernel address such as `0xC0000000`. Since the kernel page tables have U/S=0, the MMU detects the privilege violation and raises a page fault (exception 14) with error code bits: P=1 (page present), W=0 (read), U=1 (user mode). Your page fault handler from Milestone 3 prints the diagnostic and halts. This is the hardware-enforced memory isolation that makes processes safe from each other.

Section 8: The System Call Interface — INT 0x80

User-mode processes cannot directly call kernel functions — they run in ring 3, the kernel code runs in ring 0, and the CPU prevents cross-privilege direct calls. The interface between user and kernel is the **system call**: a controlled entry point into the kernel that the CPU will invoke from ring 3.



The mechanism: `int 0x80` is a software interrupt. In the IDT, vector `0x80` is registered as a **trap gate with DPL=3** (the `IDT_SYSCALL_GATE = 0xEF` constant from Milestone 2). The DPL=3 allows ring 3 code to explicitly trigger this interrupt via `int 0x80` without a GPF. The trap gate type (not interrupt gate) means interrupts remain enabled during the syscall — other processes can be preempted while one is blocked in a syscall.

When `int 0x80` fires from ring 3:

1. CPU checks $DPL \geq CPL$: $DPL=3 \geq CPL=3 \rightarrow$ permitted
 2. CPU switches to ring 0 using TSS.SS0 and TSS.ESP0 (the current process's kernel stack)
 3. CPU pushes `SS`, `ESP`, `EFLAGS`, `CS`, `EIP` (the five-field ring-crossing frame)
 4. CPU jumps to the IDT handler for vector 0x80

The convention (Linux historical ABI, which you should follow for familiarity):

- `EAX` = syscall number
 - `EBX` = first argument
 - `ECX` = second argument
 - `EDX` = third argument
 - Return value in `EAX`

```
/* syscall.c */

#include "syscall.h"

#include "process.h"

#include "sched.h"

#include "vga.h"

#include "kprintf.h"

#include <stdint.h>

#include <stddef.h>

/* Syscall numbers – user processes use these constants */

#define SYS_EXIT      1

#define SYS_WRITE     4

/* Called from the INT 0x80 handler (which saves all registers via irq_common_stub) */

void syscall_dispatch(interrupt_frame_t *frame) {

    uint32_t syscall_num = frame->eax;

    uint32_t arg1      = frame->ebx;

    uint32_t arg2      = frame->ecx;

    uint32_t arg3      = frame->edx;

    switch (syscall_num) {

        case SYS_WRITE: {

            /* sys_write(fd, buf_vaddr, count)

               fd=1 → stdout (VGA), fd=2 → stderr (serial)

               buf_vaddr: virtual address in USER address space

               count: number of bytes */

            uint32_t fd      = arg1;

            uint32_t buf_vaddr = arg2;

            uint32_t count   = arg3;

            /* — Security: validate the user buffer pointer —

               The user passed us a virtual address in their address space.

               We must verify it is a valid user-space address (below kernel region)

               and readable before dereferencing it. A production kernel uses

```

```

copy_from_user() which handles faults atomically. For simplicity,
we check the range only. */

if (buf_vaddr >= 0xC0000000 || buf_vaddr + count > 0xC0000000) {

    frame->eax = (uint32_t)-1; /* Return -EFAULT */

    break;
}

/* Cast to kernel-accessible pointer.

Valid because the user page is mapped in the current page directory
(which is the user process's directory, loaded in CR3). */

const char *buf = (const char *)(uintptr_t)buf_vaddr;

if (fd == 1 || fd == 2) {

    for (uint32_t i = 0; i < count; i++) {

        if (fd == 1) vga_putchar(buf[i], VGA_COLOR_WHITE, VGA_COLOR_BLACK);

        serial_putchar(buf[i]);
    }

    frame->eax = count; /* Return bytes written */
} else {

    frame->eax = (uint32_t)-1; /* Unsupported fd */
}

break;
}

case SYS_EXIT: {

    /* sys_exit(exit_code)

    Marks the current process as a zombie, triggers a reschedule. */

    int exit_code = (int)arg1;

    kprintf("[SYSCALL] PID %u exited with code %d.\n",
            current_process->pid, exit_code);

    current_process->state = PROCESS_ZOMBIE;

    /* The scheduler will skip zombie processes; this process never runs again */

    sched_schedule(); /* Switch to another process immediately */
}

```

```

/* Should not reach here – current process is zombie and will not be resumed */

break;

}

default:

kprintf("[SYSCALL] PID %u: unknown syscall %u\n",

        current_process->pid, syscall_num);

frame->eax = (uint32_t)-1; /* Return -ENOSYS (no such syscall) */

break;

}
}

```

Register the syscall handler in `idt_setup_all()` (from Milestone 2):

```

/* In idt_setup_all() – add this line */

extern void isr128(void); /* INT 0x80 handler stub (same pattern as exception stubs) */

idt_set_gate(0x80, (uint32_t)isr128, 0x08, IDT_SYSCALL_GATE); /* DPL=3 trap gate */

```

The corresponding assembly stub:

```

; In isr_stubs.asm – INT 0x80 is a trap (no error code), user-callable
global isr128
isr128:
    ; No cli – trap gate preserves interrupts
    push dword 0          ; Dummy error code
    push dword 128         ; Vector 0x80
    jmp isr_common_stub   ; Reuse the same save/dispatch/restore path

```

NASM

User-mode code invoking INT 0x80: A user-mode process uses inline assembly (or equivalent) to make a system call:

```
/* user_syscall.h - user-mode syscall interface */

static inline int sys_write(int fd, const void *buf, size_t count) {

    int result;

    __asm__ volatile (
        "int $0x80"
        : "=a"(result)                      /* Output: EAX = return value */
        : "a"(4), "b"(fd), "c"((uint32_t)buf), "d"((uint32_t)count) /* Inputs */
        : "memory"                         /* Memory clobber: buf may be read */
    );
    return result;
}

static inline void sys_exit(int code) {

    __asm__ volatile (
        "int $0x80"
        :
        : "a"(1), "b"(code)
        :
    );
    /* Unreachable: process is now zombie */
    for (++);
}

```

A complete user-mode process that uses system calls:

```

/* user_process.c - runs at ring 3 virtual address 0x00400000 */

/* This file is compiled separately and its binary is copied to the user code frame */

#include "user_syscall.h"

void user_main(void) {

    const char *msg = "Hello from ring 3!\n";

    /* Count to 5, printing each iteration */

    for (int i = 0; i < 5; i++) {
        sys_write(1, msg, 19);

        /* Busy loop so the scheduler has a chance to preempt us */

        for (volatile int d = 0; d < 500000; d++);
    }

    /* Try to access kernel memory - this should trigger a page fault */

    /* Uncomment to test isolation: */

    /* volatile int *kernel_ptr = (int*)0xC0000000; int x = *kernel_ptr; */

    sys_exit(0);
}

```

Pointer validation in `syscall_dispatch`: The `buf_vaddr >= 0xC0000000` check is the simplest possible security check. It is insufficient for production — a user process could pass a valid-looking address that spans a gap, points to an unmapped page, or points to another user process's read-only segment. Linux's `copy_from_user()` uses a more robust approach: it temporarily enables SMAP (which would fault on kernel access to user memory), then uses assembly that catches page faults during the copy and returns an error code. The page fault handler has a registered "fixup" table for these known-safe fault sites. This milestone's range check is sufficient to demonstrate the security principle; the production path is architecturally more sophisticated.

Section 9: The Three-Level View — What a Timer Interrupt Really Does

Tracing the full path of a single timer tick that results in a context switch — from hardware signal to resumed process:

Level 1 — What Your Processes Experience: Process A is executing its busy-loop. Nothing in Process A's code indicates an interruption — no function call, no yield, no cooperative scheduling point. Mid-loop, the CPU pauses. Later (from A's perspective, instantaneously), A resumes at exactly the instruction it was interrupted at, with all registers intact. Process B ran for 50ms while A was paused. A knows nothing about this.

Level 2 — The Kernel Path: The PIT timer channel 0 asserts IRQ0. The PIC raises INTR on the CPU. At the end of the current instruction, the CPU checks the interrupt flag (IF=1 in EFLAGS) and accepts the interrupt. It reads TSS to get the current process's

kernel stack (ESP0), pushes `SS`, `ESP`, `EFLAGS`, `CS`, `EIP` onto the kernel stack, and jumps to the IRQ1 handler at vector 32 via the IDT. The `irq_common_stub` assembly pushes segment registers and calls `pusha`. The C dispatcher increments `pit_tick_count`, calls `sched_tick()`, which decrements the time slice. If zero: `sched_schedule()` calls `tss_set_kernel_stack(next->kernel_stack_top)`, then `context_switch_asm()`. The assembly loads `next->cpu.esp` into `esp`, optionally reloads `cr3` (flushing TLB), and returns. Back in `irq_common_stub`, `popa` restores next process's general registers, segment registers are popped, `add esp, 8` discards vector/error, and `iret` restores `EIP`, `CS`, `EFLAGS`, `ESP`, `SS` — jumping to next process's instruction pointer in its address space, restoring its stack, its flags, and switching privilege level back to ring 3 if applicable. Total kernel time: ~200–400 CPU cycles.

Level 3 — Hardware Signals: The PIT chip's 16-bit counter (clocked at 1.193 MHz, configured with divisor 11932 for 100Hz) reaches zero and pulses its output line connected to the master PIC's IRQ0 input. The PIC's cascade logic checks the IMR (Interrupt Mask Register) — IRQ0 is unmasked. The PIC asserts `INTR` high. The CPU's interrupt recognition logic (checked at the end of every instruction, or every N iterations for `rep` string instructions) sees `INTR` high and `IF=1`. The CPU's microcode issues a memory read cycle to fetch the TSS (for the ESP0 field) — this read goes to main memory (DRAM, since TSS is not frequently enough accessed to stay in L1 cache): ~100 cycles latency. Then five writes to push the interrupt frame onto the kernel stack. Then an IDT lookup (the IDT is hot, stays in L2): ~10 cycles. Then a branch to the handler. The total hardware-level interrupt response time from `INTR` assertion to first handler instruction: ~150 CPU cycles (75ns at 2GHz). The PIC sees the CPU's INTA (Interrupt Acknowledge) signal and clears its ISR bit for IRQ0.

Hardware Soul: The context switch touches a lot of cold memory. The current process's PCB (`old->cpu.esp` field write): probably cold (PCB last touched during previous context switch, 50ms ago). The next process's PCB (`next->cpu.esp` read): definitely cold. The next process's kernel stack (the `popa` and segment register pops after the switch): almost certainly cold. The next process's page directory (if CR3 is reloaded): cold. A context switch that requires four DRAM accesses at 60ns each costs 240ns in memory access alone — on top of the ~75ns interrupt response. Total preemptive context switch latency: approximately 350–500ns, or ~700–1000 CPU cycles at 2GHz. This is the irreducible cost of preemption. Every async framework that avoids context switches — Node.js, nginx's event loop, Go's goroutine scheduler — is building on the knowledge that 1000 cycles is an expensive boundary to cross.

Section 10: Putting It All Together in `kmain`

The complete milestone 4 initialization, inserted into `kmain()` after the memory management setup from Milestone 3:

```
void kmain(multiboot_info_t *mbi) {  
    /* — Milestones 1-3: output, interrupts, memory — */  
  
    serial_init();  
  
    vga_clear();  
  
    kprintf("==> Kernel starting ==>\n");  
  
    idt_setup_all();  
  
    pic_remap(0x20, 0x28);  
  
    pit_init(100);  
  
    keyboard_init();  
  
  
    pmm_parse_memory_map(mbi);  
  
    pmm_init(mbi);  
  
    vmm_init();  
  
    vmm_enable_paging();  
  
    heap_init();  
  
  
    __asm__ volatile ("sti");  
  
    /* — Milestone 4: Processes and Scheduling — */  
  
    /* Initialize scheduler and TSS before creating any processes */  
  
    sched_init();  
  
    tss_init();  
  
  
    /* Register INT 0x80 system call gate (DPL=3 trap gate) */  
  
    extern void isr128(void);  
  
    idt_set_gate(0x80, (uint32_t)isr128, 0x08, IDT_SYSCALL_GATE);  
  
  
    /* Create three kernel-mode processes */  
  
    process_t *pa = process_create_kernel("proc_a", proc_a_entry);  
  
    process_t *pb = process_create_kernel("proc_b", proc_b_entry);  
  
    process_t *pc = process_create_kernel("proc_c", proc_c_entry);  
  
  
    /* Create one user-mode process */  
  
    process_t *pu = process_create_user("user_hello", user_main);
```

```

kprintf("[M4] Created %d processes. Entering scheduler.\n",
(pa?1:0) + (pb?1:0) + (pc?1:0) + (pu?1:0));

/* Make kmain's execution context the idle process (PID 0) */

process_table[0].state      = PROCESS_RUNNING;

process_table[0].pid        = 0;

process_table[0].page_directory = boot_page_directory;

process_table[0].kernel_stack_top =
(uint32_t)process_table[0].kernel_stack + KERNEL_STACK_SIZE;

current_process = &process_table[0];

tss_set_kernel_stack(current_process->kernel_stack_top);

kprintf("[M4] Idle process active. Timer will preempt shortly.\n");

kprintf("[M4] Watch VGA rows 2, 12, 22 for concurrent output.\n");

/* Idle: sleep until interrupted */

while (1) {

    __asm__ volatile ("hlt");

}

}

```

Debugging Preemptive Scheduling — The Hardest Category

Scheduling bugs are uniquely difficult because the failure is separated from the cause by thousands of instructions and multiple context switches. A systematic approach:

Symptom	Root Cause	Diagnostic
System freezes after <code>sti</code> , no preemption	Timer handler not calling <code>sched_tick</code> ; or <code>current_process</code> is NULL	Add <code>kprintf</code> to <code>timer_handler</code> before and after <code>sched_tick</code>
Triple fault immediately on first context switch	<code>context_switch_asm</code> reads wrong stack offsets after modifying <code>esp</code> ; or fake initial frame wrong	Verify argument loading order in asm; print <code>proc->cpu.esp</code> before switch
Process resumes with wrong register values	<code>pusha / popa</code> imbalance in IRQ stub; context switch asm not saving all registers	Compare <code>pusha</code> order with PCB field order; add register printout in fault handler
User process triggers GPF instead of running	Wrong CS selector in initial frame (must be <code>0x1B</code> , not <code>0x08</code>); or DPL mismatch	Print the CS value from <code>iret</code> exception frame
INT 0x80 causes GPF	IDT gate has wrong DPL (must be 3 for user-callable); or wrong gate type	Print IDT entry byte at offset for vector 0x80; verify <code>IDT_SYSCALL_GATE = 0xEF</code>
Kernel memory access from ring 3 does not fault	User page directory has U/S=1 on kernel pages (inherited PDE incorrectly)	Dump the PDE flags for the kernel region in the user page directory
TSS not updated — wrong process's kernel stack corrupted	<code>tss_set_kernel_stack()</code> called after <code>context_switch_asm</code> instead of before	Move the TSS update to <i>before</i> the <code>context_switch_asm</code> call
Double free or heap corruption	Process A's kernel stack overlaps another structure (pcb sizing error)	Print <code>proc->kernel_stack</code> and <code>proc->kernel_stack_top</code> for each process; verify no overlap

The QEMU GDB stub is essential for this milestone:

```
# Launch QEMU with GDB stub, paused at boot                                BASH
qemu-system-i386 -drive format=raw,file=os.img -serial stdio \
    -s -S -d int 2>qemu_ints.log

# In another terminal

gdb kernel.elf

(gdb) target remote :1234

(gdb) break sched_schedule

(gdb) continue

# When breakpoint hits:

(gdb) print *current_process      # Inspect current PCB
(gdb) print next->cpu.esp        # Inspect next process's saved ESP
(gdb) x/20x next->cpu.esp       # Dump the fake initial stack frame
```

Examining `qemu_ints.log` after a triple fault will show the last successful interrupt before the crash — typically a timer IRQ followed by an exception during the context switch attempt, pinpointing the exact failure.

Knowledge Cascade — What This Milestone Unlocks

You have just built the engine that drives everything above the OS layer. The connections extend into every domain of systems software.

1. Context switch cost as the origin of event-driven architecture (cross-domain: web servers and distributed systems)

The ~1000-cycle context switch you just implemented is the direct cause of an entire architectural movement. In the early 2000s, the "C10K problem" described why Apache-style threaded web servers — one thread per connection — could not scale past ~10,000 simultaneous connections on commodity hardware. Each connection required a thread, each thread required ~1000 cycles per scheduler invocation, and with 10,000 threads, the scheduler overhead dominated useful work.

The solution was event-driven I/O: **epoll** (Linux), **kqueue** (FreeBSD), and **IOCP** (Windows) allow one thread to wait on thousands of file descriptors simultaneously, switching between them cooperatively only when I/O is ready — no context switch, because there is no thread switch. Node.js built an entire ecosystem on this principle. **nginx** (which replaced Apache for high-concurrency workloads) uses an event loop with a small fixed number of worker threads. **Go's goroutines** are a middle path: lightweight green threads that context-switch in user space (M:N threading), amortizing the kernel context switch cost across thousands of goroutines that are multiplexed onto a small pool of OS threads.

The reason these architectures exist — the reason Linus Torvalds's 2001 email dismissing thread-per-request as "hopelessly inefficient" sparked a decade of framework design — is the number you just measured: 1000 cycles, ~500ns, per switch. When you understand what those cycles are buying (register saves, TLB flush, cache cold misses), you understand why Go's goroutine scheduler can context-switch in ~100ns (no TLB flush, smaller register set, stack is hot) and why that 5x difference matters at scale.

2. TSS.ESP0 as the foundation of container isolation (cross-domain: Linux namespaces, Docker)

The per-process kernel stack you just configured — and the TSS.ESP0 update that makes it work — is not just a scheduling convenience. It is the root of OS-level process isolation.

Every Linux process has a distinct kernel stack. When a process makes a syscall or receives an interrupt, the CPU jumps to kernel code using that process's stack. If two processes shared a kernel stack, one process's syscall could read or overwrite the other's saved state on the shared stack. The TSS.ESP0 mechanism prevents this by giving the CPU a per-process pointer that it uses before executing a single kernel instruction.

Linux containers (the foundation of Docker and Kubernetes) extend this isolation in software: namespaces (PID, network, mount, user, UTS, IPC) restrict what a container process can see. **cgroups** limit what it can consume. But the hardware isolation root — the one that makes a container process actually unable to access another container's memory — is the same mechanism you just built: each container process has its own page directory (CR3), and the CPU enforces the boundary at the page table level. A **container escape vulnerability** (a CVE that allows a containerized process to escape its isolation) almost always involves either a kernel stack smashing bug (overwriting the saved state of another process on its kernel stack) or a page table manipulation bug (tricking the kernel into mapping a container's memory as kernel-accessible). You have just built the mechanism these attacks target. Understanding it at the instruction level is prerequisite for understanding the exploit patterns.

3. Round-robin as fairness, not performance — tracing the path to CFS (Linux scheduler)

Your scheduler gives each ready process one equal time slice in turn. It is a correct and implementable baseline. It is also inadequate for every real workload, for a reason that becomes visible the moment you add a fourth process that does I/O.

An I/O-bound process (waiting for keyboard input, disk read, or network packet) is almost never in `PROCESS_READY`. It spends most of its time in `PROCESS_BLOCKED`. Round-robin skips it efficiently. But when it *does* become ready (the I/O completes), it must wait for its turn in the round-robin cycle — potentially waiting for N-1 other processes to run their full slices first. For interactive applications (terminals, text editors), this introduces perceptible latency: the user presses a key, the key arrives in the keyboard buffer, but the process that reads it does not run for another 50ms × (N-1) processes.

Linux's CFS (Completely Fair Scheduler) solves this with `vruntime`: each process accumulates a virtual runtime counter (how much CPU time it has received, weighted by its priority). The scheduler always runs the process with the *lowest* vruntime — i.e., the

process that has received the *least* CPU time proportionally. An I/O-bound process that was blocked for 100ms accumulates no vruntime during that time, so when it unblocks, it has the lowest vruntime in the system and runs immediately. **SCHED_FIFO** and **SCHED_RR** are Linux's real-time scheduling classes: FIFO runs a process until it blocks or yields (no preemption); RR gives equal time slices but never interleaves with SCHED_FIFO tasks. **Go's work-stealing scheduler** (which manages goroutines across OS threads) uses a local run queue per OS thread plus a global queue, and idle threads "steal" goroutines from busy threads — a load balancing mechanism invisible at the OS level because it happens in user space. Your round-robin scheduler is the Ur-primitive from which all of these descend.

4. INT 0x80 as the slow path that motivated SYSENTER/SYSCALL and the vDSO

`int 0x80` is a software interrupt. Invoking it requires: checking DPL (2 cycles), looking up the IDT entry (memory access, ~10 cycles), switching to the kernel stack via TSS (memory access for TSS.ESP0, ~100 cycles if cold), pushing five values onto the kernel stack (5 writes), saving all registers, executing the syscall handler, then reversing all of this with `iret`. Typical `int 0x80` round-trip time: ~500 cycles.

Intel recognized this bottleneck in 1996 and added **SYSENTER/SYSEXIT** (P6 architecture) — a pair of instructions optimized for syscall entry and exit that bypass the IDT lookup and use MSRs (Model-Specific Registers) to specify the kernel entry point and stack. This reduces syscall overhead to ~100 cycles. AMD added **SYSCALL/SYSRET** (similar, slightly different mechanism, now the 64-bit standard). Linux uses SYSCALL in 64-bit mode and SYSENTER in 32-bit mode.

But Linux went further: the **vDSO (Virtual Dynamic Shared Object)** is a small kernel-mapped page inserted into every process's address space. It contains small functions like `gettimeofday()` and `clock_gettime()` that read kernel data (the current time, updated by the timer interrupt) without crossing the ring boundary at all. A user process calling `clock_gettime(CLOCK_MONOTONIC)` reads the TSC (Time Stamp Counter — a CPU register that counts cycles since boot) directly, with no syscall, no ring transition, no context switch — and gets nanosecond precision in ~20 cycles. The vDSO is possible because the kernel can share read-only data with user processes via page tables. Understanding `int 0x80` — its cost, its path through the IDT, the TSS lookup it triggers — makes the entire SYSENTER/vDSO optimization story obvious: every optimization in that chain is eliminating one step of the `int 0x80` path you just implemented.

5. Per-process page directories as the hardware root of every isolation boundary

Every process you just created has its own `CR3` — its own root page table. When the scheduler loads a new `CR3`, the CPU's address space switches instantaneously and completely. Every virtual address means something different. Physical address `0x00201234` might be process A's heap and process B's stack simultaneously — because they are mapped to different virtual addresses in different page directories, and the MMU translates through different tables. The hardware enforces that neither process can access the other's physical frames because neither process's page tables contain entries for the other's frames.

This same mechanism is the hardware root of:

- **Virtual machines:** A hypervisor (VMX root mode) interposes another level of page tables (EPT — Extended Page Tables) between the guest OS's page tables and physical memory. The guest OS thinks it is managing physical memory; it is actually managing guest-physical memory that the hypervisor translates again. Two levels of CR3-like pointers, both enforced by MMU hardware.
- **Browser tab isolation (site isolation in Chrome):** Each tab runs in a separate renderer process with its own page directory. A compromised tab cannot read another tab's memory because the kernel never maps one tab's physical frames into another's page directory.
- **Spectre/Meltdown mitigations (KPTI — Kernel Page Table Isolation):** After Meltdown (2018), the Linux kernel adopted KPTI: user-mode processes run with a page directory that contains almost no kernel mappings (only the minimum needed to enter the kernel). The full kernel page directory is restored only after the ring transition. This doubles the TLB flush frequency (every ring transition requires a CR3 swap) but prevents speculative execution from reading kernel memory via cache timing channels. The cost of KPTI — measured at 5–30% overhead on syscall-heavy workloads — is the cost of swapping `CR3` twice per syscall instead of zero times. You just built the mechanism whose overhead Meltdown exploited and KPTI pays to mitigate.

Final System View — Everything Connected

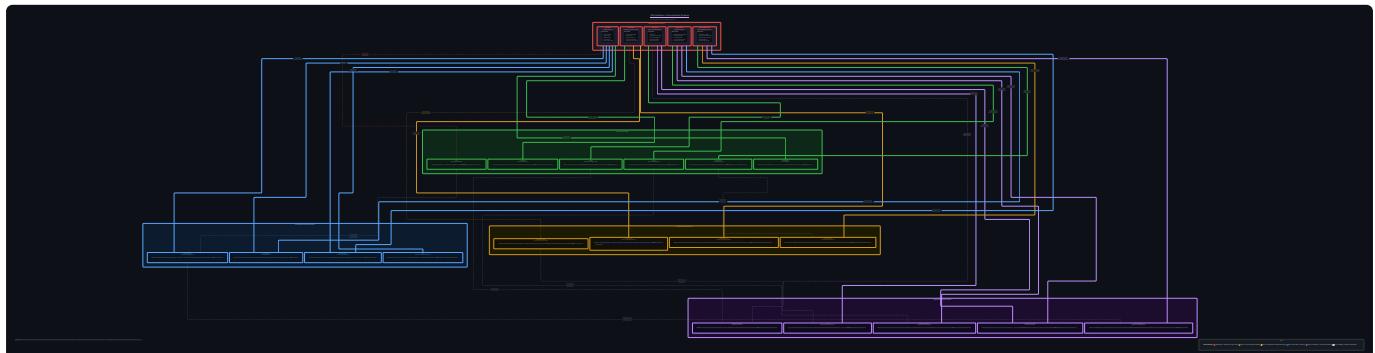


Standing at the end of Milestone 4, your kernel does what took the Unix team years to build:

- **Milestone 1** provided the hardware foundation: protected mode, segmentation, I/O drivers, a C execution environment.
- **Milestone 2** wired in the interrupt infrastructure: the IDT, PIC remapping, the timer tick, keyboard input, the double fault safety net.
- **Milestone 3** built the address space: physical frame tracking, two-level page tables, virtual-to-physical translation, a kernel heap.
- **Milestone 4** animated multiple concurrent processes: PCBs that capture complete CPU state, assembly-level context switching, TSS-backed ring transitions, a preemptive round-robin scheduler, user-mode isolation, and a system call gate.

Every abstraction that application programmers use — `malloc`, `printf`, `pthread_create`, `read`, `mmap` — sits atop one or more of these mechanisms. `malloc` calls `sbrk` which calls a page fault path in your VMM. `printf` eventually calls `write` which crosses your INT 0x80 gate into your `sys_write` handler. `pthread_create` allocates a PCB and kernel stack and calls your scheduler. `read` blocks a process (sets it to `PROCESS_BLOCKED`) until data arrives via your IRQ handler.

The knowledge you have built is not academic. It is executable understanding of the substrate that every software system runs on.



TDD

A bottom-up x86 kernel implementation that negotiates directly with hardware constraints at every layer: the 1978 real-mode boot environment, the 8259 PIC's vector collision hazard, the TLB coherence problem during paging enablement, and the TSS's role as the hardware root of privilege isolation. Each module exposes one fundamental tension between what software wants and what hardware provides, building implementation-grade understanding of mechanisms that underlie every modern computing platform.

Technical Design Specification: Boot, GDT, and Kernel Entry

Module ID: `mod-1` | Milestone: `build-os-m1` | Estimated Hours: 17–27

1. Module Charter

This module transitions the x86 CPU from the BIOS-provided 16-bit real-mode environment at `0x7C00` through a two-stage bootloader into 32-bit protected mode, depositing control at a C kernel entry point with a clean execution environment. It owns the

complete boot path: MBR Stage1 (512 bytes exactly), Stage2 (A20 enablement, kernel disk loading, GDT construction, protected-mode transition), the linker script defining kernel section addresses, assembly entry glue (BSS zeroing, stack setup), a VGA text-mode driver writing to `0xB8000`, and a serial UART driver on COM1. The module terminates at the first instruction of `kmain()` with all output drivers functional.

This module does **not** implement an IDT, any interrupt handlers, the 8259 PIC, page tables, virtual memory, a heap allocator, or any process management. Interrupts remain disabled (`cli`) for the entire duration. No dynamic allocation occurs; all data structures are statically sized and placed by the linker. The module assumes a single-core execution context with no preemption.

Upstream dependency: BIOS firmware — provides INT 13h disk services, the memory map at `0x400 – 0x4FF`, and the initial CPU state. **Downstream dependency:** Milestone 2 (interrupts) which requires `cli` to already hold and segment registers to be loaded with protected-mode selectors.

Invariants that must hold at module exit:

1. `CR0.PE = 1` — CPU is in 32-bit protected mode, never to return to real mode.
2. `CS = 0x08`, `DS = ES = FS = GS = SS = 0x10` — all segment registers contain valid protected-mode kernel selectors.
3. All bytes in `.bss` are zero.
4. `ESP` points within a valid 16KB kernel stack region in `.bss`.
5. The GDT contains exactly 5 valid entries at the offsets specified in §3.
6. `0xB8000` is accessible as a writable MMIO region producing visible VGA output.
7. COM1 UART is initialized at 38400/8N1 and transmits bytes synchronously.

2. File Structure

Create files in this exact sequence. Later files depend on earlier ones compiling cleanly.

```
build-os/
├── 01 boot/
│   ├── 01 stage1.asm          # 512-byte MBR: load stage2 from disk, jump
│   └── 02 stage2.asm          # A20, load kernel, build GDT, enter protected mode
├── 02 kernel/
│   ├── 01 linker.ld           # Section layout: kernel at 0x100000
│   ├── 02 entry.asm           # 32-bit entry: zero BSS, set ESP, call kmain
│   ├── 03 include/
│   │   ├── 01 stdint.h         # Freestanding type aliases (uint8_t ... uint32_t)
│   │   ├── 02 stddef.h          # size_t, NULL, offsetof
│   │   ├── 03 io.h              # outb/inb inline asm port I/O
│   │   ├── 04 vga.h             # VGA driver public interface
│   │   └── 05 serial.h          # Serial driver public interface
│   ├── 04 drivers/
│   │   ├── 01 vga.c             # VGA text-mode driver implementation
│   │   └── 02 serial.c          # COM1 UART driver implementation
│   ├── 05 kprintf.c            # Minimal formatted output to VGA + serial
│   ├── 06 kprintf.h             # kprintf declaration
│   └── 07 kmain.c              # Kernel C entry point: calls drivers, prints welcome
└── 03 Makefile                # Build rules: assemble, compile, link, create image
└── 04 run.sh                  # QEMU launch script with -serial stdio
```

Why this order: `stdint.h` and `stddef.h` must exist before any C file compiles. `io.h` is included by both drivers. Drivers are compiled before `kmain.c` to catch header mismatches early. The bootloader files are assembled independently of the kernel; the Makefile stitches them into a single disk image.

3. Complete Data Model

3.1 GDT Descriptor — Byte-Level Layout

Each GDT entry is **exactly 8 bytes**. The layout is non-contiguous by Intel design (286 backward compatibility). The table below gives byte offsets within a single 8-byte entry.

Byte offset:	7	6	5	4	3	2	1	0
Field:	Base [31:24]	Flags + Lim [19:16]	Access Byte [15:0]	Base [23:16]	Base [23:0]	Limit [23:16]	Limit [15:0]	

Field breakdown:

Byte(s)	Bits	Field	Purpose
0–1	[15:0]	Limit low	Lower 16 bits of segment limit
2–3	[15:0]	Base low	Lower 16 bits of base address
4	[7:0]	Base mid	Bits [23:16] of base address
5	[7:0]	Access byte	Presence, DPL, type flags (see below)
6	[7:4]	Flags	Granularity, size, L, AVL
6	[3:0]	Limit high	Upper 4 bits of limit [19:16]
7	[7:0]	Base high	Bits [31:24] of base address

Access byte bit decomposition (byte 5):

Bit:	7	6	5	4	3	2	1	0
P	DPL	S	E	DC	RW	A		

Legend:

- Accessed (CPU sets; init to 0)
- Readable(code)/Writable(data)
- Direction/Conforming
- Executable (1=code, 0=data)
- Descriptor type (1=code/data, 0=system)
- DPL [1:0] (00=ring0, 11=ring3)
- Present (must be 1)

Flags nibble (byte 6 high nibble):

Bit:	7	6	5	4
G	DB	L	AVL	

Legend:

- Available (software use; set 0)
- Long mode (0 for 32-bit protected mode)
- Default operand size (1=32-bit)
- Granularity (1=4KB units for limit)

The five required GDT entries:

Index	Selector	Name	Base	Limit	Access Byte	Flags Nibble	Notes
0	0x00	Null	0	0	0x00	0x0	Required by CPU spec; any load of selector 0 → GPF
1	0x08	Kernel Code	0	0xFFFFF	0x9A	0xC	Ring 0, executable+readable, 4KB gran, 32-bit
2	0x10	Kernel Data	0	0xFFFFF	0x92	0xC	Ring 0, data+writable, 4KB gran, 32-bit
3	0x18	User Code	0	0xFFFFF	0xFA	0xC	Ring 3, executable+readable, 4KB gran, 32-bit
4	0x20	User Data	0	0xFFFFF	0xF2	0xC	Ring 3, data+writable, 4KB gran, 32-bit

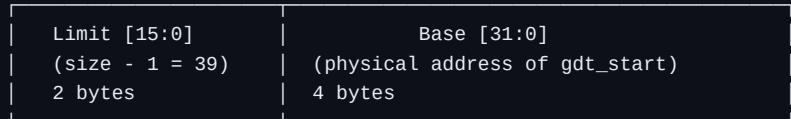
Access byte derivation:

- 0x9A = 1 001 1 010 = P=1, DPL=00, S=1, E=1, DC=0, RW=1, A=0 → ring-0 code, readable
- 0x92 = 1 001 0 010 = P=1, DPL=00, S=1, E=0, DC=0, RW=1, A=0 → ring-0 data, writable
- 0xFA = 1 111 1 010 = P=1, DPL=11, S=1, E=1, DC=0, RW=1, A=0 → ring-3 code, readable
- 0xF2 = 1 111 0 010 = P=1, DPL=11, S=1, E=0, DC=0, RW=1, A=0 → ring-3 data, writable

Flags nibble 0xC derivation: G=1 (4KB granularity), DB=1 (32-bit), L=0, AVL=0 → binary 1100 = 0xC.

Combined byte 6 for each non-null entry = (flags_nibble << 4) | limit_high_nibble = 0xC << 4 | 0xF = 0xCF.

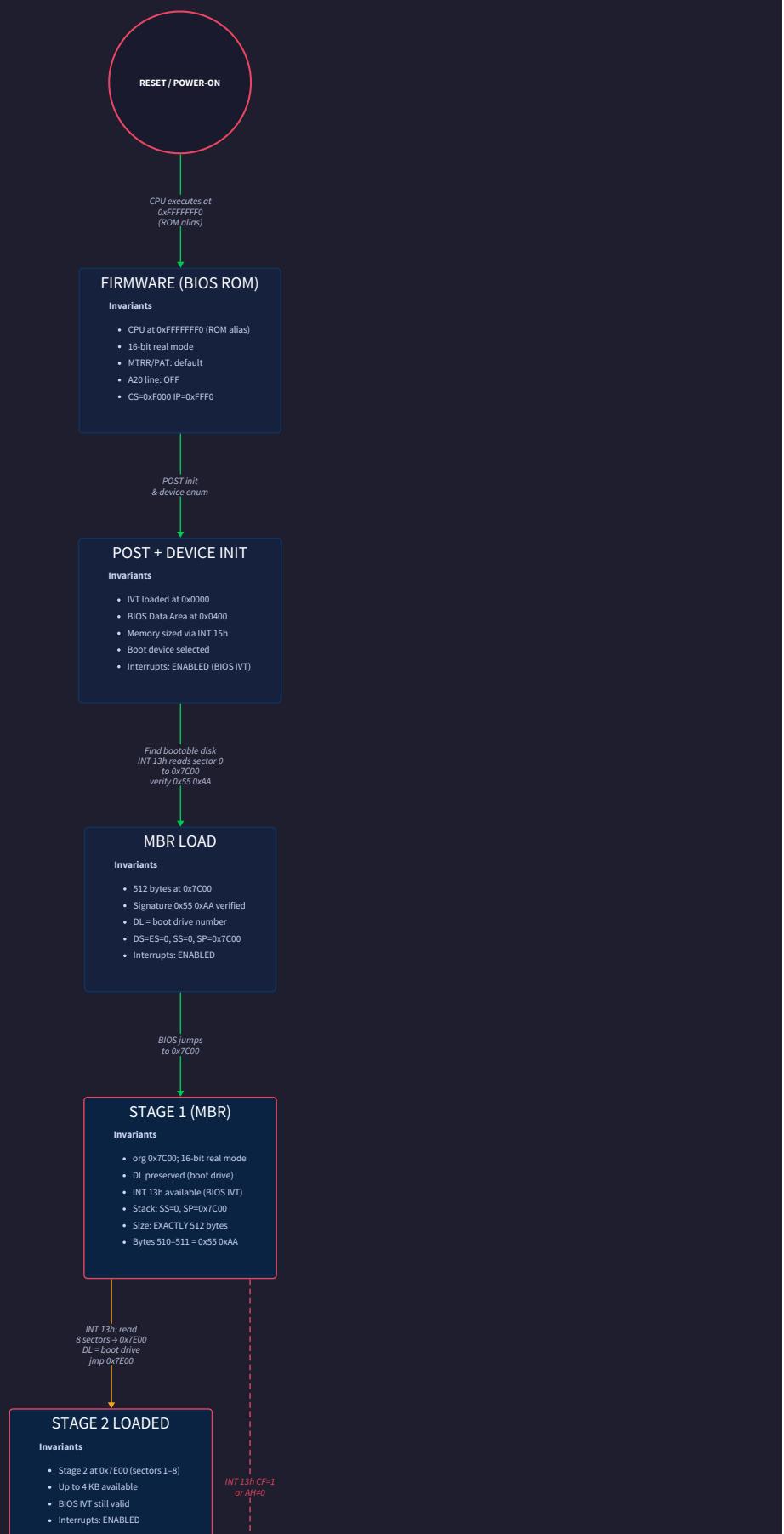
GDTR (6 bytes, loaded by lgdt):

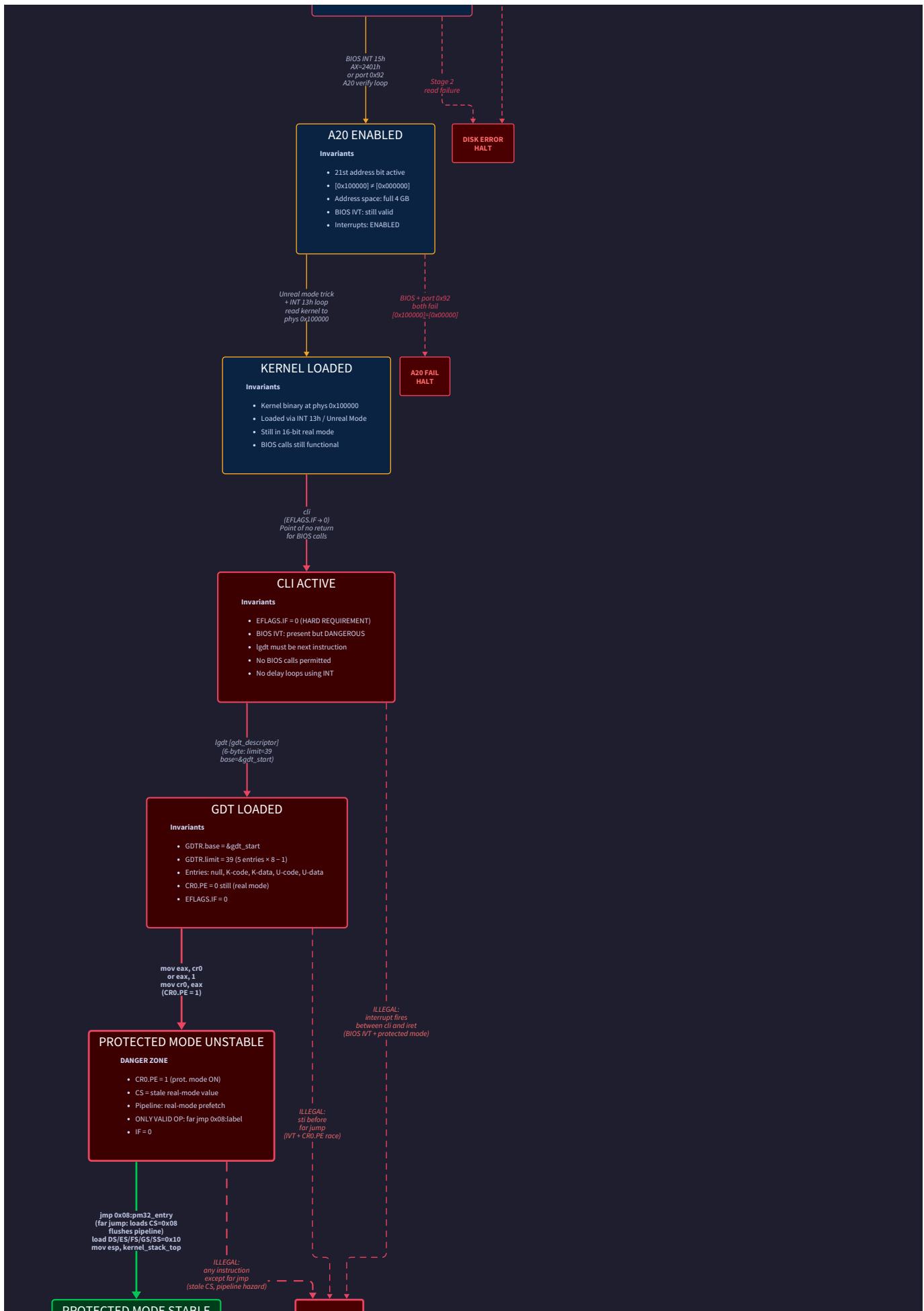


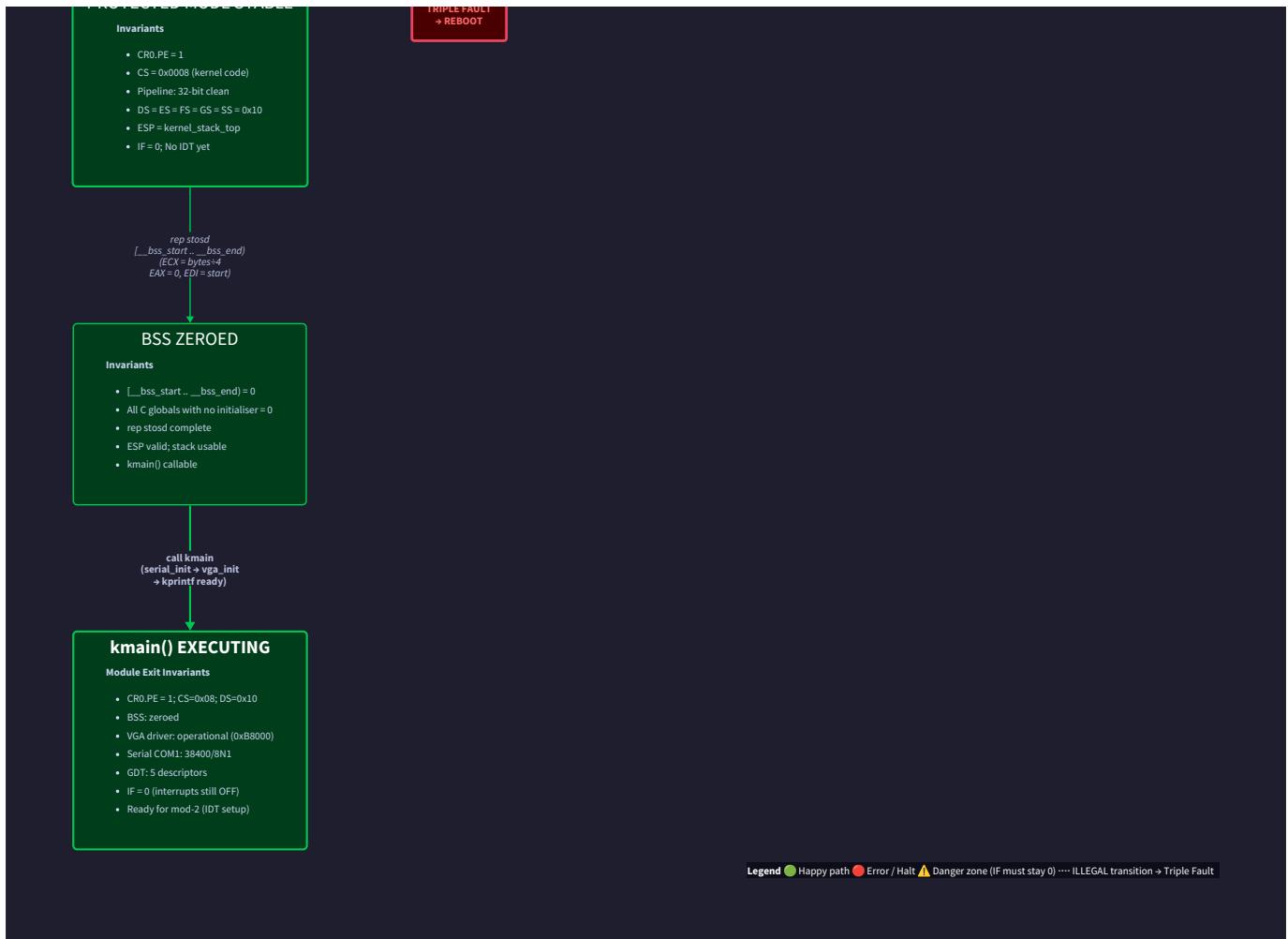
GDTR limit calculation: 5 entries × 8 bytes = 40 bytes total. Limit field = 40 - 1 = 39 = 0x0027.

Boot Sequence State Machine: BIOS to kmain()

x86 Power-On → 16-bit Real Mode → 32-bit Protected Mode → C Kernel Entry







3.2 VGA Cell — 2-Byte Memory-Mapped Structure

The VGA text buffer at physical `0xB8000` is an array of 2000 cells (80 columns × 25 rows).

```

Cell layout (uint16_t, little-endian):
Byte 0: ASCII character code (0x00-0xFF)
Byte 1: Attribute byte

Attribute byte:
Bits [3:0] = Foreground color (0-15)
Bits [6:4] = Background color (0-7)
Bit [7] = Blink (or bright background, hardware-dependent)

```

Cell address formula: `uint16_t *cell = (uint16_t*)0xB8000 + row * 80 + col`

Color encoding:

Value	Color	Value	Color
0x0	Black	0x8	Dark Gray
0x1	Blue	0x9	Light Blue
0x2	Green	0xA	Light Green
0x3	Cyan	0xB	Light Cyan
0x4	Red	0xC	Light Red
0x5	Magenta	0xD	Light Magenta
0x6	Brown	0xE	Yellow
0x7	Light Gray	0xF	White

Memory layout of `vga_state_t` (internal driver state, in `.data` section):

```
// Byte offset  Field          Type       Purpose
// 0            cursor_row    int        Current write row (0-24)
// 4            cursor_col    int        Current write column (0-79)
// 8            default_fg    uint8_t   Default foreground color
// 9            default_bg    uint8_t   Default background color
// 10-11        (padding)    -          Align to 4 bytes
// Total: 12 bytes
```

3.3 UART Register Map — COM1 at `0x3F8`

Port-mapped I/O; access via `outb` / `inb` instructions. All offsets relative to base `0x3F8`.

Offset	DLAB	Direction	Register Name	Purpose
+0	0	Write	THR — Transmitter Holding Register	Write byte to transmit
+0	0	Read	RBR — Receiver Buffer Register	Read received byte
+0	1	R/W	DLL — Divisor Latch Low	Baud divisor low byte
+1	0	R/W	IER — Interrupt Enable Register	Enable/disable UART IRQs
+1	1	R/W	DLH — Divisor Latch High	Baud divisor high byte
+2	—	R/W	FCR/IIR — FIFO Control/Interrupt ID	FIFO control
+3	—	R/W	LCR — Line Control Register	Data bits, stop bits, parity; bit7=DLAB
+4	—	R/W	MCR — Modem Control Register	RTS, DTR, loopback
+5	—	Read	LSR — Line Status Register	bit5=THRE (transmit ready)

Baud rate divisor calculation: Base clock = 1,843,200 Hz ($\div 16$ for baud). For 38400 baud: `divisor = 1,843,200 / 16 / 38400 = 3`.

Initialization sequence (8 port writes, strict order):

1. `outb(0x3F9, 0x00)` — disable all UART interrupts
2. `outb(0x3FB, 0x80)` — set DLAB=1 to access divisor registers
3. `outb(0x3F8, 0x03)` — divisor low byte (3 → 38400 baud)
4. `outb(0x3F9, 0x00)` — divisor high byte (0)
5. `outb(0x3FB, 0x03)` — clear DLAB=0; set 8 data bits, no parity, 1 stop bit
6. `outb(0x3FA, 0xC7)` — enable FIFO, clear Tx/Rx FIFOs, 14-byte trigger
7. `outb(0x3FC, 0x0B)` — IRQs enabled, RTS and DSR asserted (loopback off)
8. `outb(0x3FC, 0x0F)` — set in normal operation mode (loopback test optional)

Why step 8: Sending `0x0F` to MCR clears the loopback bit set implicitly on some hardware. On QEMU this is a no-op; on real 16550 UARTs it is necessary.

3.4 Linker Script Section Layout

Physical / Virtual Address	Section	Contents
0x000007C00	(MBR)	Stage1 assembly – not part of kernel ELF
0x000007E00	(Stage2)	Stage2 assembly – not part of kernel ELF
0x00100000 ←— KERNEL LOAD	.text	All compiled code (entry.asm + C files)
	.rodata	String literals, const arrays
	.data	Initialized global/static variables
	.bss	Uninitialized globals; __bss_start/end symbols
	(stack)	16KB static array in .bss; top = __stack_top

Linker symbols exported for assembly use:

Symbol	Definition	Used by
<code>__bss_start</code>	<code>. at start of .bss</code>	<code>entry.asm</code> for BSS zero loop
<code>__bss_end</code>	<code>. at end of .bss</code>	<code>entry.asm</code> for BSS zero loop
<code>__stack_top</code>	Address of <code>kernel_stack + 16384</code>	<code>entry.asm</code> for <code>mov esp, __stack_top</code>
<code>kernel_entry</code>	First label in <code>entry.asm</code>	Stage2 far jump destination

3.5 Disk Image Layout

LBA Sector	Size	Contents
0	512 B	Stage1 MBR (bytes 510-511 = 0x55 0xAA)
1-8	4096 B	Stage2 bootloader (max 8 sectors = 4KB)
9-72	32768 B	Kernel binary (max 64 sectors = 32KB; expand as needed)

INT 13h CHS mapping for these sectors: Cylinder 0, Head 0, Sectors 1–73 (CHS sectors are 1-indexed). All fit within track 0, head 0, cylinder 0 of a standard 1.44MB floppy geometry (80 cylinders, 2 heads, 18 sectors/track) or a hard disk geometry.

{DIAGRAM:tdd-diag-2}

4. Interface Contracts

4.1 vga_init(void) → void

Pre-conditions: CPU in 32-bit protected mode. `0xB8000` is identity-mapped and writable (no paging yet; physical = virtual).

Post-conditions: Internal cursor at (0,0). All 2000 cells written with `' '` (space, ASCII 0x20) and attribute `0x07` (white-on-black). Internal `default_fg = VGA_WHITE (0xF)`, `default_bg = VGA_BLACK (0x0)`.

Side effects: Writes 4000 bytes to physical `0xB8000 – 0xB8F9F`.

Errors: None. Cannot fail — `0xB8000` is always present on x86 hardware and in QEMU.

4.2 vga_putchar(char c, uint8_t fg, uint8_t bg) → void

Parameters:

- `c` : ASCII character or control code. Valid range: all `uint8_t` values. Special handling for `\n` (0x0A) and `\r` (0x0D) only; all other values are printed literally including non-printable codes.
- `fg` : Foreground color, range `[0x0, 0xF]`. Values above `0xF` are masked: `fg & 0xF`.
- `bg` : Background color, range `[0x0, 0x7]`. Values above `0x7` are masked: `bg & 0x7`.

Behavior:

- If `c == '\n'` : set `cursor_col = 0`, `cursor_row++`. Do not write a cell.
- If `c == '\r'` : set `cursor_col = 0`. Do not write a cell. Do not advance `cursor_row`.
- Otherwise: write `(uint16_t)c | (((bg & 0x7) << 4 | (fg & 0xF)) << 8)` to `VGA_BASE[cursor_row * 80 + cursor_col]`. Advance `cursor_col`. If `cursor_col >= 80` : set `cursor_col = 0`, `cursor_row++`.
- After any operation that increments `cursor_row` : if `cursor_row >= 25`, call `vga_scroll()`.

Post-conditions: Cell at previous cursor position contains the character. Cursor advanced one position or wrapped.

Errors: None propagated. Silent mask of out-of-range fg/bg values.

4.3 vga_scroll(void) → void

Behavior: Copy rows 1–24 into rows 0–23 (a `memmove` equivalent on the VGA buffer: `2 * 80 * 24 = 3840` bytes moved from `VGA_BASE + 80` to `VGA_BASE`). Fill row 24 with space characters (`' '`) and attribute `(default_bg << 4 | default_fg) << 8`. Set `cursor_row = 24`, `cursor_col = 0`.

Implementation note: Use a word-at-a-time copy (copy `uint16_t` values, not bytes) to minimize MMIO write count. 1920 `uint16_t` writes for the scroll, 80 `uint16_t` writes to clear the last row.

Why `cursor_row = 24` not `0`: After scrolling, the next character goes on the newly cleared last row, not back to the top. The scroll moves content up; writing continues at the bottom.

4.4 vga_write_string(const char *s, uint8_t fg, uint8_t bg) → void

Parameters:

- `s` : Null-terminated C string. Must not be NULL. If NULL is passed, behavior is undefined (no null check for performance; callers are responsible).
- `fg`, `bg` : As in `vga_putchar`.

Behavior: Calls `vga_putchar(s[i], fg, bg)` for each byte until `s[i] == '\0'`. The null terminator is not printed.

4.5 `vga_set_cursor(int row, int col) → void`

Parameters:

- `row` : [0, 24]. Values outside range: clamped. `row < 0` → `row = 0`. `row > 24` → `row = 24`.
- `col` : [0, 79]. Values outside range: clamped.

Post-conditions: `cursor_row = row`, `cursor_col = col`. No VGA memory written.

4.6 `vga_clear(void) → void`

Behavior: Equivalent to `vga_init()` except does not reinitialize default colors if already set. Writes space + `0x07` attribute to all 2000 cells. Sets cursor to (0,0).

4.7 `serial_init(void) → void`

Pre-conditions: CPU in protected mode. Port I/O accessible (no IOPL check needed at ring 0).

Behavior: Executes the 8-write initialization sequence from §3.3. Uses `io_wait()` between each write (single `outb(0x80, 0)` to POST diagnostic port, ~1–4µs delay).

Post-conditions: COM1 accepts bytes via `serial_putchar`. Baud rate = 38400. Format = 8N1. FIFO enabled.

Errors: None detectable without a loopback test (which this module omits). If COM1 hardware is absent (not the case on QEMU), writes are silently ignored.

4.8 `serial_putchar(char c) → void`

Behavior:

1. Busy-wait: spin on `(inb(0x3FD) & 0x20) == 0` (LSR bit 5 = THRE — Transmitter Holding Register Empty). Loop iteration: one `inb` (~1–4µs per iteration on legacy I/O bus).
2. When THRE=1: `outb(0x3F8, (uint8_t)c)`.

Maximum wait time: At 38400 baud, 10 bits per frame, the UART flushes in $10 / 38400 \approx 260\mu\text{s}$. The busy-wait terminates within one character-time.

Errors: None. If the UART never asserts THRE (hardware fault), the function spins forever. This is acceptable for a debug-only driver without interrupt infrastructure.

4.9 `serial_write_string(const char *s) → void`

Behavior: Calls `serial_putchar(s[i])` for each byte until `s[i] == '\0'`. If `s[i] == '\n'`, also calls `serial_putchar('\r')` immediately after — serial terminals expect CRLF line endings.

4.10 `kprintf(const char *fmt, ...) → void`

Supported format specifiers:

Specifier	Argument Type	Behavior
%c	int (char)	Single character
%s	const char *	Null-terminated string; NULL pointer prints "(null)"
%d , %i	int	Signed decimal integer
%u	unsigned int	Unsigned decimal integer
%x	unsigned int	Lowercase hexadecimal, no 0x prefix
%X	unsigned int	Uppercase hexadecimal, no 0x prefix
%08x	unsigned int	Zero-padded 8-digit hex (only width=1–8 with 0 flag supported)
%%	—	Literal %

Unsupported specifiers: %f , %g , %e , %p , %l , %ll . If encountered, print %? where ? is the unknown character.

Output targets: Both `vga_putchar` (with `VGA_WHITE` on `VGA_BLACK`) and `serial_putchar`. Every character goes to both destinations synchronously before `kprintf` returns.

Implementation: Uses `va_list` / `va_start` / `va_arg` / `va_end` from a freestanding `<stdarg.h>`. Integer-to-string conversion via a local static buffer of 32 bytes (sufficient for a 32-bit integer in any base). No heap allocation.

Error behavior: If a format string ends with a lone % (e.g., "hello %"), print % and return. Do not read past the null terminator.

4.11 `outb(uint16_t port, uint8_t value) → void (inline, io.h)`

```
static inline void outb(uint16_t port, uint8_t value) {
    __asm__ volatile ("outb %0, %1" : : "a"(value), "Nd"(port) : "memory");
}
```

Constraints: "Nd" encodes port as an immediate 8-bit value (if constant and ≤ 255) or via DX register (if variable or > 255). "memory" clobber prevents the compiler from reordering memory accesses across the I/O operation.

4.12 `inb(uint16_t port) → uint8_t (inline, io.h)`

```
static inline uint8_t inb(uint16_t port) {
    uint8_t ret;

    __asm__ volatile ("inb %1, %0" : "=a"(ret) : "Nd"(port) : "memory");

    return ret;
}
```

4.13 `io_wait(void) → void (inline, io.h)`

```
static inline void io_wait(void) {  
    outb(0x80, 0x00); /* Write to unused POST port; ~1-4µs delay on legacy bus */  
}
```

C

5. Algorithm Specification

5.1 Stage1 MBR Boot Sector (512 bytes)

Input state: CPU at `0x7C00`, real mode, `DL` = boot drive number, interrupts enabled.

Step-by-step:

```

Step 1: Establish addressability
- org 0x7C00      ; NASM directive: all label addresses relative to 0x7C00
- cli             ; Disable interrupts immediately – stale IVT is dangerous
- xor ax, ax
- mov ds, ax      ; DS = 0 (flat addressing within first 64KB)
- mov es, ax      ; ES = 0 (INT 13h destination segment)
- mov ss, ax      ; SS = 0
- mov sp, 0x7C00   ; Stack grows downward from 0x7C00 (safe: below MBR)
- sti             ; Re-enable interrupts (BIOS IVT still valid; needed for INT 13h)

Step 2: Save boot drive
- mov [boot_drive], dl    ; Preserve DL for disk read calls; BIOS may clobber it

Step 3: Load Stage2 from disk
Registers for INT 13h function 0x02 (read sectors):
AH = 0x02          ; Function: read sectors
AL = 8              ; Read 8 sectors (4096 bytes = Stage2 max size)
CH = 0              ; Cylinder 0
CL = 2              ; Starting sector 2 (1-indexed; sector 1 = this MBR)
DH = 0              ; Head 0
DL = [boot_drive]   ; Drive number
ES:BX = 0x0000:0x7E00; Load destination (immediately after MBR)
- int 0x13
- jc disk_error     ; CF=1 on error

Step 4: Validate load (optional but recommended)
- cmp ah, 0          ; AH=0 on success
- jne disk_error

Step 5: Jump to Stage2
- jmp 0x0000:0x7E00   ; Far jump clears CS prefetch with flat DS=0 model

Step 6: disk_error (label)
- mov si, error_msg   ; Point SI to error string
- call print_string_rm ; BIOS INT 10h teletype output
- cli
- hlt
- jmp $               ; Infinite loop (hlt may return on NMI)

Step 7: print_string_rm (real-mode helper, fits within 512 bytes)
- Uses INT 10h AH=0xE (teletype output) in a loop until [SI] = 0
- Preserves AX, BX per BIOS convention

Step 8: Data
- boot_drive: db 0      ; Saved boot drive
- error_msg: db "Boot error", 0x0D, 0x0A, 0

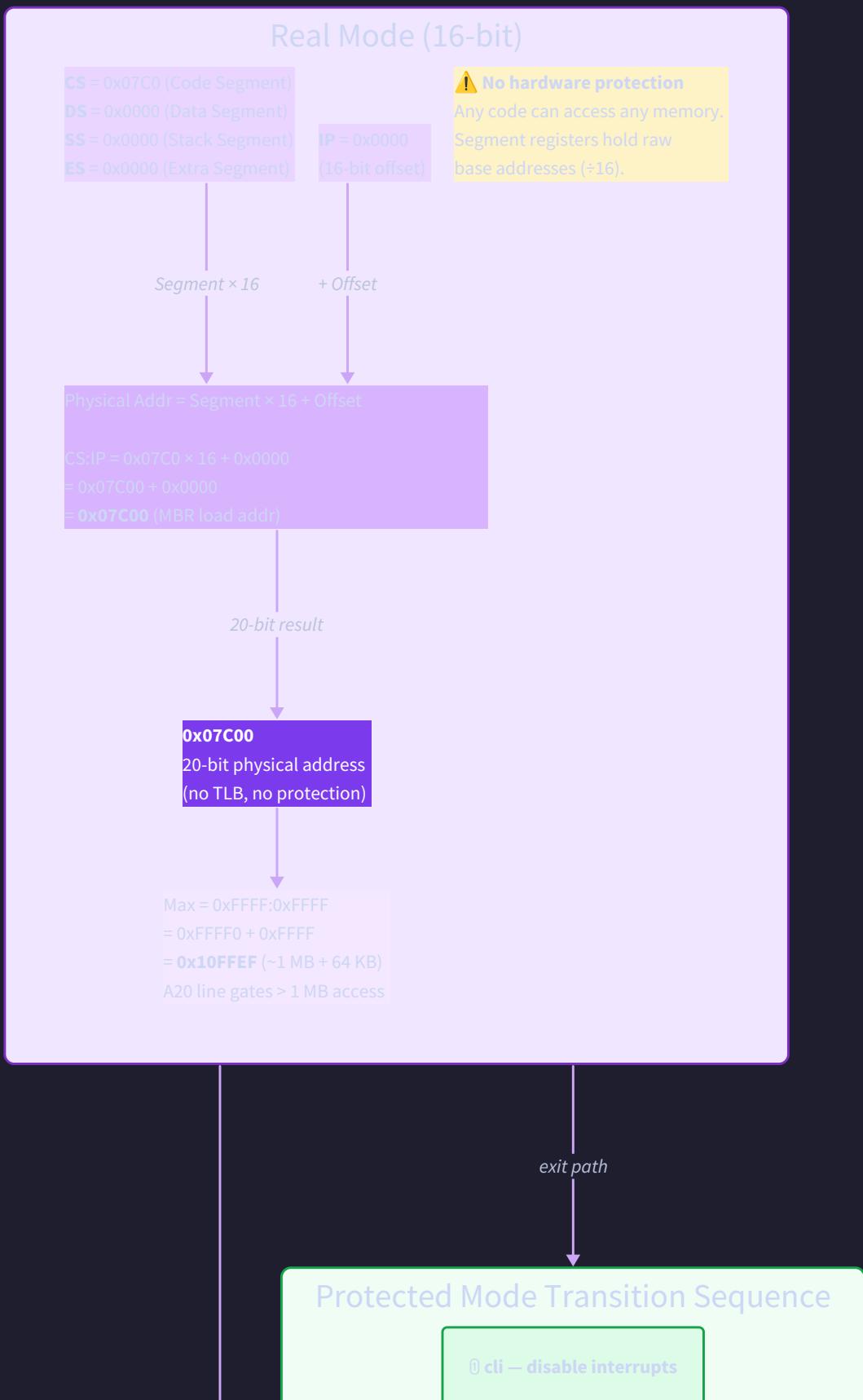
Step 9: Padding and signature (MUST be last 2 bytes)
- times 510 - ($ - $$) db 0    ; Pad to exactly 510 bytes
- dw 0xAA55                 ; Boot signature (little-endian: bytes 0x55, 0xAA)

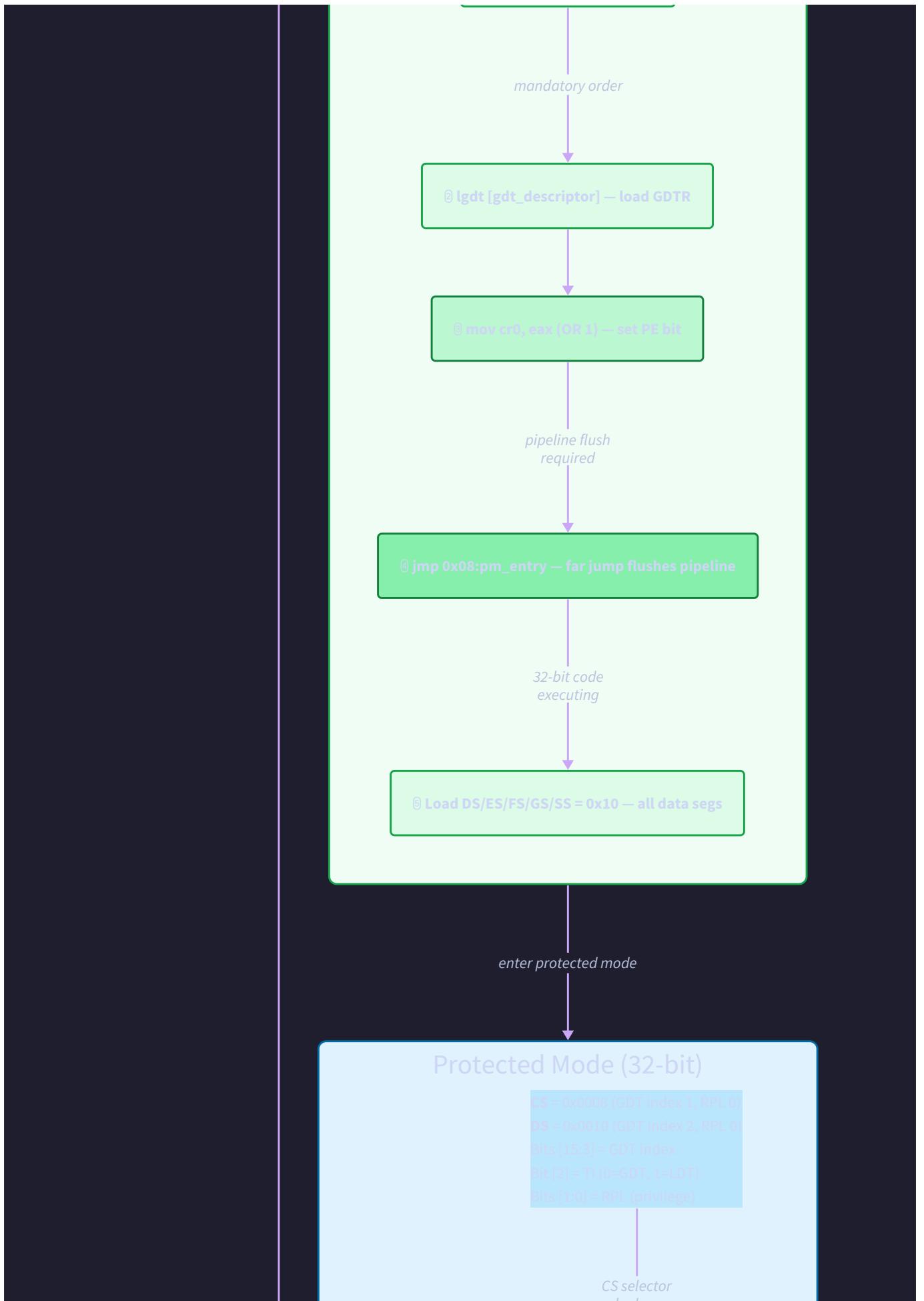
```

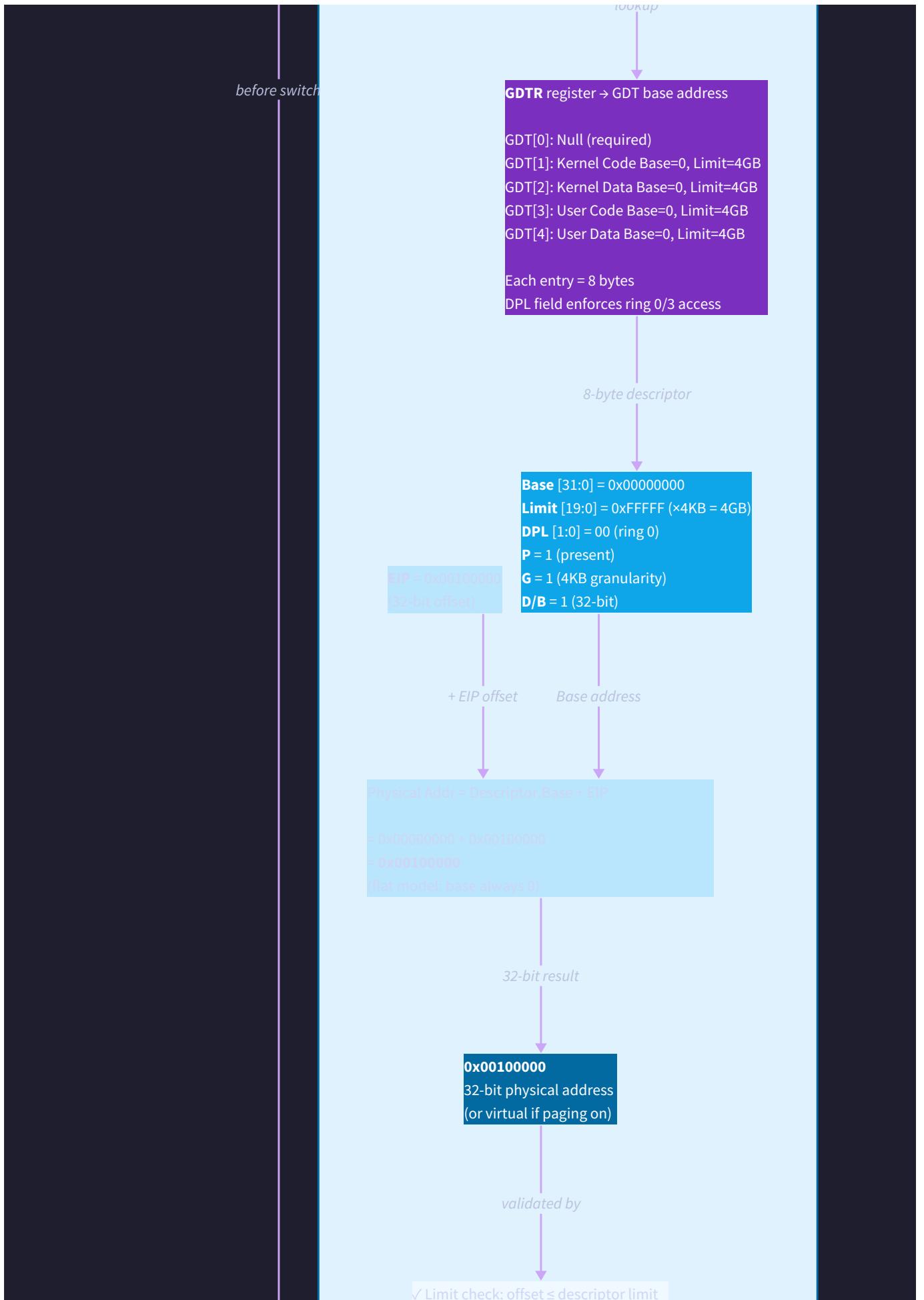
Invariant after Step 3: Memory at `0x7E00 – 0x9DFF` contains Stage2. `DL` = boot drive (restored from saved copy).

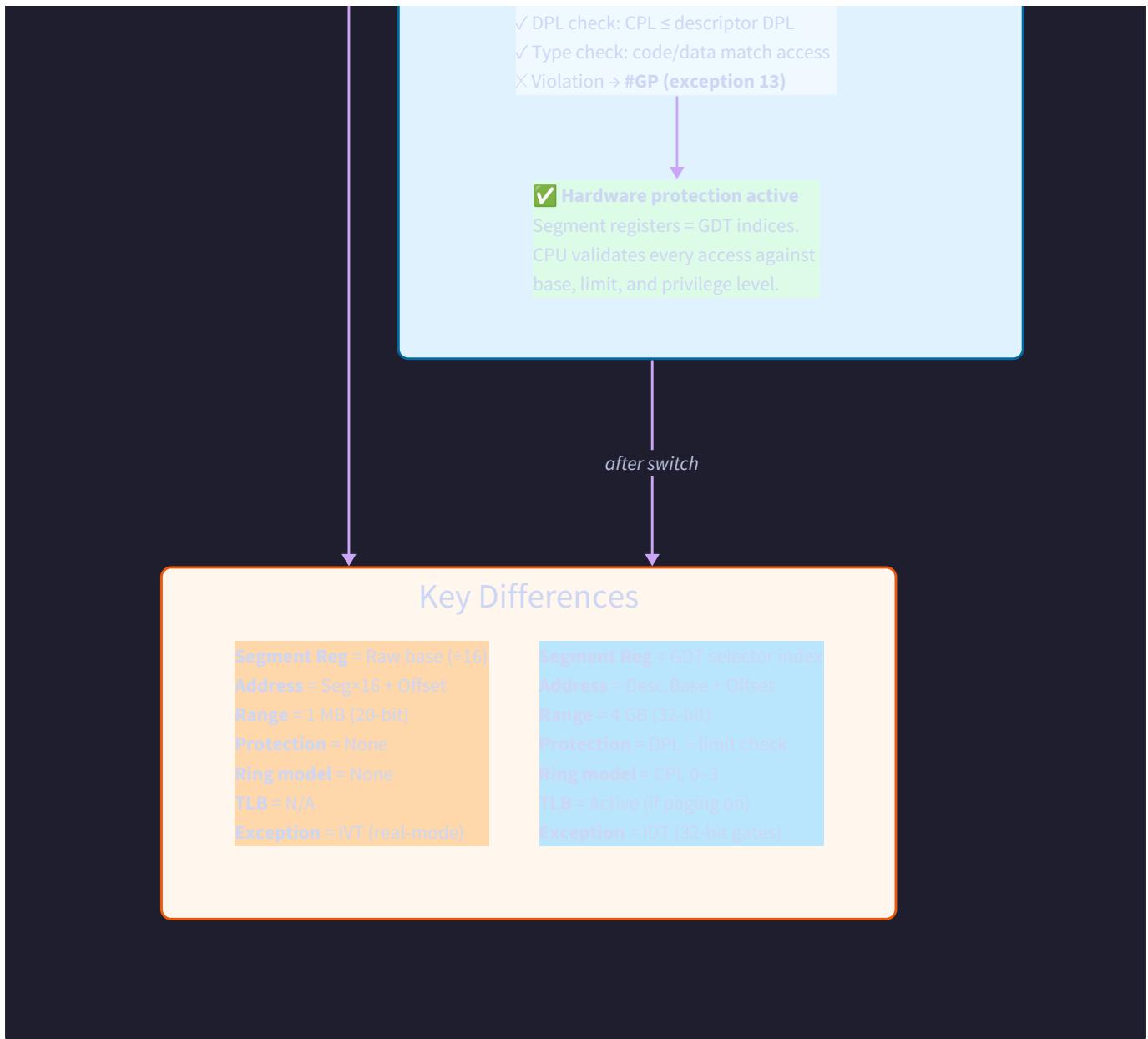
Size constraint: Every byte matters. The `print_string_rm` helper, `error_msg`, `boot_drive` byte, and instructions must total ≤ 510 bytes. If Stage1 grows, remove the error message and use a simple `hlt` for `disk_error`.

Real Mode vs Protected Mode: Address Translation Comparison









5.2 Stage2 Bootloader

Input state: Executing at `0x7E00` in real mode, `DS=ES=SS=0`, `SP` valid, interrupts enabled.

Step-by-step:


```
Step 1: Print "Loading kernel..." via INT 10h teletype
- Provides visual confirmation that Stage2 ran
```

```
Step 2: Enable the A20 Line (BIOS method – most portable)
```

```
A20 via BIOS INT 15h AX=2401h:
MOV AX, 0x2401
INT 15h
JC a20_failed ; CF=1 if not supported; fall through to Port 0x92 method
```

```
A20 via Fast A20 (Port 0x92) fallback:
```

```
IN AL, 0x92
OR AL, 0x02 ; Set bit 1 = A20 enable
AND AL, 0xFE ; Clear bit 0 = do NOT reset CPU (bit 0 = fast CPU reset)
OUT 0x92, AL
```

```
A20 verify (test that address 0x100000 and 0x000000 are different):
```

```
MOV AX, 0xFFFF
MOV ES, AX
MOV WORD [ES:0x0010], 0x1234 ; Write at physical 0x100000
MOV AX, 0x0000
MOV ES, AX
CMP WORD [ES:0x0000], 0x1234 ; Read physical 0x000000
JE a20_failed ; Same = A20 still off (wraparound)
; Restore [0x000000] to 0 (IVT entry 0):
MOV WORD [ES:0x0000], 0x0000
```

```
Step 3: Load kernel from disk into 0x100000
```

```
Problem: INT 13h in real mode cannot directly address above 1MB (ES:BX is 20-bit).
```

```
Solution A (BIOS Extended Read – INT 13h AH=0x42, LBA):
```

```
Construct a DAP (Disk Address Packet) in memory:
```

```
dap:
db 0x10 ; Size of packet (16 bytes)
db 0 ; Reserved
dw 64 ; Number of sectors to read (64 × 512 = 32KB)
dw 0x0000 ; Buffer offset (for 32-bit address in bytes 8-11)
dw 0x0000 ; Buffer segment (unused when 32-bit address present)
dq 9 ; LBA start sector (sector 9, 0-indexed)
; Extended read (not available on all hardware; check with AH=0x41, BX=0x55AA)
; Falls back to solution B if not supported.
```

```
Solution B (Unreal Mode – read above 1MB from real mode):
```

```
Enter "Unreal Mode" (temporarily enable protected mode to load 32-bit segments,
then return to real mode with 32-bit segment cache still active):
```

1. Load a minimal GDT with a 4GB data descriptor into GDTR
2. Set CR0.PE = 1
3. Load GS with the 32-bit flat data selector (0x08 in the minimal GDT)
4. Set CR0.PE = 0 (return to real mode)
5. CS is still real-mode (far jump to flush pipeline)
6. GS segment cache now holds 4GB limit from protected-mode descriptor
7. Use INT 13h normally but write to GS:address where address = 0x100000
 (The 32-bit GS descriptor allows this address even though we are back in real mode)
8. MOV [GS:0x100000], ... to copy sectors above 1MB

```
RECOMMENDED implementation for simplicity: Use Solution B (Unreal Mode).
```

```
Unreal mode is the standard technique for real-mode bootloaders loading above 1MB.
```

```
INT 13h read loop (after entering unreal mode):
```

```
Outer loop: sectors_remaining = 64
Each iteration:
AH = 0x02, AL = min(sectors_remaining, 127) ; Max sectors per call
CH = cylinder, CL = sector (CHS computed from current LBA)
DH = head, DL = boot_drive
ES:BX = current_load_buffer (lower 1MB staging area, e.g. 0x0000:0x9000)
INT 13h
Copy result to GS:kernel_load_address via REP MOVSD
```

```

Advance kernel_load_address += bytes_read
Decrement sectors_remaining

Step 4: Disable interrupts – must not fire during or after GDT load
CLI

Step 5: Load the GDT
LGDT [gdt_descriptor]
; gdt_descriptor is a 6-byte structure: dw (size-1), dd (linear address of gdt_start)
; gdt_start contains the 5 entries defined in §3.1

Step 6: Set CR0.PE
MOV EAX, CR0
OR EAX, 1
MOV CR0, EAX
; CPU is now in protected mode. CS still holds real-mode value.

Step 7: Far jump to flush pipeline and load CS with kernel code selector
JMP 0x08:pm32_entry
; 0x08 = GDT index 1 (kernel code), RPL=0
; pm32_entry must be the physical address of the 32-bit code label

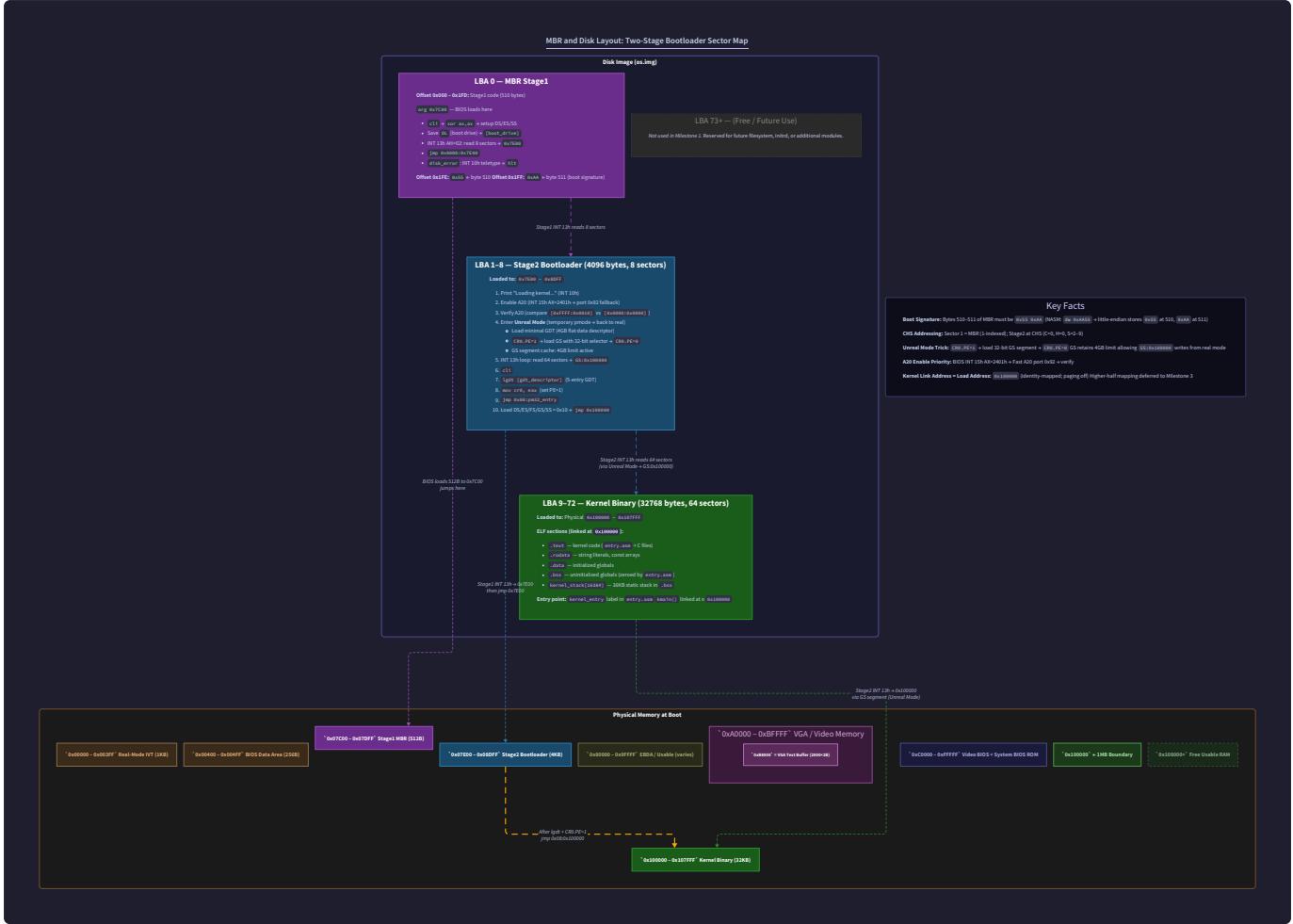
Step 8: [bits 32] pm32_entry:
; Load all data segment registers
MOV AX, 0x10      ; Kernel data selector
MOV DS, AX
MOV ES, AX
MOV FS, AX
MOV GS, AX
MOV SS, AX

; Jump to kernel entry point at 0x100000
JMP 0x100000      ; Or: JMP 0x08:0x100000 (far jump to ensure CS is properly loaded)

```

Invariants after Step 8:

- CR0.PE = 1
- CS = 0x08, DS = ES = FS = GS = SS = 0x10
- CPU is executing 32-bit instructions
- Kernel binary is at physical 0x100000 – 0x10FFFF



5.3 Protected Mode Transition — Invariants and Timing

The four-instruction sequence `lgdt` → `or cr0,1` → `mov cr0,eax` → `jmp 0x08:label` must be contiguous with no interrupts possible between them. Interrupt enable state:

```
Before step 4 (cli): EFLAGS.IF may be 1
After cli:           EFLAGS.IF = 0 (guaranteed)
After lgdt:          EFLAGS.IF = 0 (lgdt does not change IF)
After mov cr0, eax: EFLAGS.IF = 0; CPU in protected mode; IVT useless
After far jmp:       EFLAGS.IF = 0; CS valid; pipeline flushed; safe
After segment reload: EFLAGS.IF = 0; all segments valid; protected mode fully operational
```

Why `jmp 0x08:0x100000` rather than `jmp 0x08:pm32_entry`: Stage2 is loaded at `0x7E00`. The `pm32_entry` label is at approximately `0x7E00 + N` (wherever it falls in Stage2). After protected mode is entered, Stage2 code is still accessible via the flat 4GB kernel code segment (base=0). But the kernel binary is at `0x100000`. The final jump to `0x100000` transfers control to the kernel's own entry point in `entry.asm`.

5.4 Kernel Entry Point — BSS Zeroing and Stack Setup (`entry.asm`)

Input state: 32-bit protected mode, `CS=0x08`, `DS=ES=FS=GS=SS=0x10`, `ESP` undefined (or whatever Stage2 left it at).

```

Step 1: Establish kernel stack immediately
MOV ESP, __stack_top      ; __stack_top is a linker symbol pointing to top of kernel_stack[]
; All subsequent C ABI operations are valid

Step 2: Zero BSS
MOV EDI, __bss_start      ; Destination: start of BSS
MOV ECX, __bss_end
SUB ECX, EDI              ; ECX = byte count
SHR ECX, 2                ; ECX = dword count (/4; BSS is always multiple of 4 per linker)
XOR EAX, EAX              ; Value to store: 0
REP STOSD                ; Store 0 to [EDI], advance EDI by 4, decrement ECX; repeat

; Handle remainder if BSS is not a multiple of 4:
MOV ECX, __bss_end
SUB ECX, EDI              ; Remaining bytes (0-3)
REP STOSB                ; Store remaining bytes one at a time

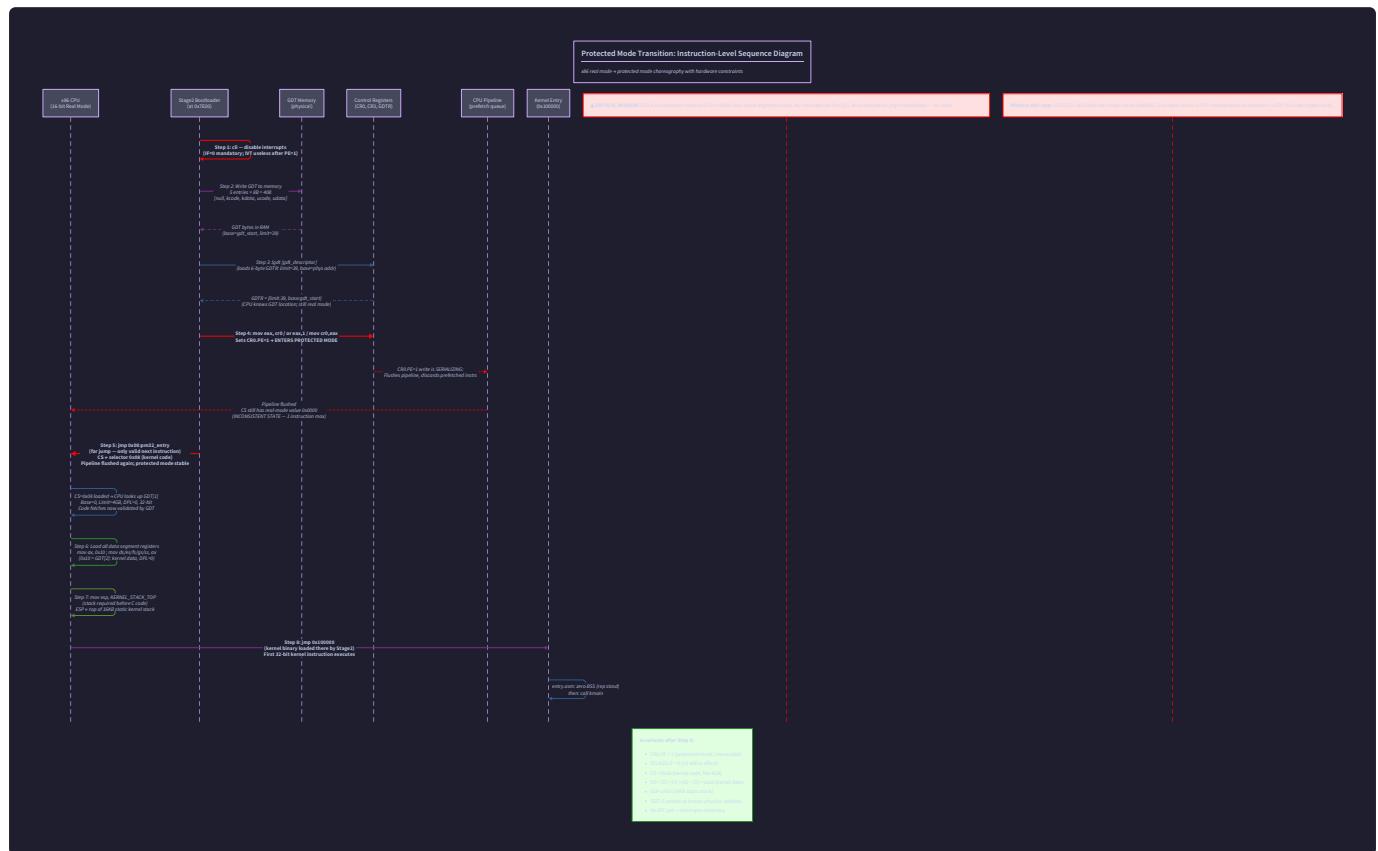
Step 3: Call kmain
CALL kmain                 ; Transfer to C kernel

Step 4: Halt if kmain returns (should never happen)
CLI
HLT
JMP $                      ; Catch NMI wakeup

```

Why set ESP before zeroing BSS: If BSS zeroing were to cause an exception (impossible here, but defensive coding), the CPU would need a valid stack to push the exception frame. More practically: C code called from `kmain` uses the stack immediately.

Why `rep stosd` then `rep stosb`: `stosd` stores 4 bytes per iteration, which is faster (fewer loop iterations, instruction-level parallelism). A BSS section not aligned to 4 bytes leaves a 0–3 byte remainder handled by `stosb`. The linker script aligns BSS to 4 bytes by default, making the `stosb` loop zero iterations in practice; it is present for correctness regardless.



5.5 VGA Scroll Algorithm

Input: Screen full (cursor_row was incremented to 25).

```
Step 1: Shift rows 1-24 into rows 0-23
uint16_t *dst = VGA_BASE;           // row 0
uint16_t *src = VGA_BASE + 80;      // row 1
uint32_t count = 80 * 24;          // 1920 cells
for (i = 0; i < count; i++) dst[i] = src[i];

Step 2: Clear row 24
uint16_t blank = (uint16_t)' ' | ((uint16_t)((default_bg << 4) | default_fg) << 8);
uint16_t *last_row = VGA_BASE + (24 * 80);
for (i = 0; i < 80; i++) last_row[i] = blank;

Step 3: Update cursor
cursor_row = 24;
cursor_col = 0;
```

Performance: $1920 + 80 = 2000$ MMIO writes per scroll. At ~200ns per write (uncached MMIO): ~400 μ s per scroll. Acceptable for a debug terminal; production would use hardware cursor or direct DMA.

5.6 Integer-to-String Conversion in `kprintf`

```
// Converts unsigned 32-bit integer to string in given base [2,16]
// Returns pointer to null-terminated string in static buffer
// NOT reentrant - call sequence must be single-threaded

static const char *uint_to_str(uint32_t value, int base, int uppercase) {

    static char buf[33]; // 32 binary digits + null

    static const char lower[] = "0123456789abcdef";
    static const char upper[] = "0123456789ABCDEF";

    const char *digits = uppercase ? upper : lower;

    int pos = 32;

    buf[pos] = '\0';

    if (value == 0) { buf[--pos] = '0'; return &buf[pos]; }

    while (value > 0) {

        buf[--pos] = digits[value % base];

        value /= base;

    }

    return &buf[pos];
}
```

Zero-padding for `%08x`: After generating the hex string, if a width specifier was parsed (e.g., width=8, pad='0'), prepend '0' characters until the string length equals the specified width before writing to output.

6. Error Handling Matrix

Error	Detected By	Recovery	User-Visible?
INT 13h disk read failure (CF=1)	Stage1 after <code>int 0x13</code>	Print "Boot error\r\n" via INT 10h teletype, <code>cli; hlt</code>	Yes — screen message
A20 enable failed (BIOS + port 0x92 both fail)	Stage2 A20 verify loop	Print "A20 error\r\n", <code>cli; hlt</code>	Yes — screen message
A20 verify loop: address 0x100000 wraps to 0x0	Stage2 verify: comparing <code>[0xFFFF:0x0010] == [0x0000:0x0000]</code>	Retry port 0x92 method once; then halt with message	Yes
GDT access byte wrong (wrong DPL, E bit, S bit)	CPU triple-fault on segment register load or first memory access	Triple fault → QEMU reboot; no recoverable handler at this point	QEMU reboots — check -d int log
Far jump omitted after CR0.PE — pipeline has real-mode prefetch	CPU executes garbled instructions; eventual GPF or invalid opcode	Triple fault	QEMU reboots
Interrupts not disabled before lgdt	Hardware IRQ fires; CPU uses stale IVT; immediate triple-fault	Triple fault	QEMU reboots
Kernel binary at wrong address — linker/load mismatch	CPU jumps to <code>0x100000</code> ; fetches wrong bytes; invalid opcode or crash	Triple fault (no diagnostic possible before kmain)	QEMU reboots
A20 not enabled — kernel above 1MB reads as wrapped	Silent data corruption in kernel binary or BSS; unpredictable crash	Triple fault or wrong behavior; detected via A20 verify step	Usually triple fault
BSS not zeroed — global variable is non-zero at startup	Logic errors in <code>kmain</code> or subsequent code; often manifests as wrong output	Incorrect behavior; detected by output test	May appear as wrong output
Boot signature missing or wrong (bytes 510–511 ≠ 0x55, 0xAA)	BIOS refuses to boot; tries next boot device	BIOS beeps or shows "No bootable device"	Yes — BIOS message
VGA <code>volatile</code> missing — writes optimized away	Screen remains blank; serial output still works (if serial initialized first)	No crash; output simply does not appear; detected by blank VGA test	Yes — blank screen
Serial: DLAB not cleared before framing byte write	Framing register write goes to DLH instead; baud rate corrupted	Garbage output on serial; may cause serial to stop working	Yes — garbled serial
Serial: wrong baud divisor	Baud rate mismatch with terminal; output appears as garbage symbols	No crash; visible as garbled output	Yes — garbled serial
<code>kprintf NULL %s</code> argument	<code>vga_putchar</code> loop dereferences NULL	Page fault (once paging exists); before paging: reads physical address 0 (IVT) — likely prints garbage	Unpredictable; guard with NULL check
BSS zero loop: <code>rep stosd</code> count off by one (not multiple of 4)	Last 1–3 bytes of BSS not zeroed	Rare C global variable unexpectedly non-zero	Silent; detected by inspection

7. Implementation Sequence with Checkpoints

Phase 1 — Stage1 MBR (3–5 hours)

Create `boot/stage1.asm`. Assemble with `nasm -f bin -o stage1.bin boot/stage1.asm`. Verify binary is exactly 512 bytes: `wc -c stage1.bin` must print `512`. Verify last two bytes: `xxd stage1.bin | tail -1` must show `55 aa` at positions 510–511.

Checkpoint 1: Run `qemu-system-i386 -drive format=raw,file=stage1.bin -serial stdio`. QEMU should either hang (no Stage2 loaded yet) or print "Boot error" because the INT 13h read of Stage2 finds no data. Either outcome confirms the MBR runs. Add `-d int` to QEMU flags and verify the last interrupt before hang is a disk-related INT (vector 0x13), not a CPU exception (vectors 0–20). If QEMU immediately reboots with an interrupt vector 8 (double fault), the boot signature is missing.

Phase 2 — Stage2: A20, kernel load, GDT construction (4–6 hours)

Create `boot/stage2.asm`. At this phase, Stage2 does NOT yet enter protected mode — it just enables A20, attempts to load a known byte pattern to `0x100000`, verifies it, and prints "A20 OK" + "Kernel loaded" via INT 10h. The kernel binary does not exist yet; fill sectors 9–72 with a known pattern (e.g., `db 0xAA`) via the Makefile for testing purposes.

Create `Makefile` rules to assemble stage1 and stage2 and create a disk image:

```
# Create raw disk image: stage1 at sector 0, stage2 at sectors 1-8  
dd if=/dev/zero of=os.img bs=512 count=2880  
dd if=stage1.bin of=os.img conv=notrunc bs=512 seek=0  
dd if=stage2.bin of=os.img conv=notrunc bs=512 seek=1
```

MAKEFILE

Checkpoint 2: `qemu-system-i386 -drive format=raw,file=os.img -serial stdio`. Should see "A20 OK" and "Kernel loaded" on QEMU screen (via INT 10h teletype, before protected mode). If A20 verify fails, check: did you restore `[0x000000]` after the verify test? Did you use `ES:BX` or `GS:offset` correctly for the unreal mode load?

Phase 3 — Protected mode transition (2–3 hours)

Complete Stage2 with the GDT table, `lgdt`, `cli`, `CR0.PE`, far jump, segment register reload, and final `jmp 0x100000`. The kernel binary placeholder at `0x100000` must now contain valid 32-bit code to avoid a triple fault. Create a minimal `kernel_placeholder.bin`: 16 bytes of `0xF4` (HLT instruction) assembled with `nasm -f bin`.

```
# Add kernel placeholder to image at sector 9  
dd if=kernel_placeholder.bin of=os.img conv=notrunc bs=512 seek=9
```

MAKEFILE

Checkpoint 3: `qemu-system-i386 -drive format=raw,file=os.img -serial stdio -d int 2>int.log`. QEMU should appear to hang (the HLT loop). In `int.log`, verify:

- No exception vectors 0–20 appear after the disk reads
- No vector 8 (double fault) from PIC/IVT collision — this would only occur if interrupts were enabled, which they should not be
- The last CPU state shows `CS=0x0008`, `EFLAGS` with `PE=1` bit in CR0

If QEMU reboots: `grep "v=0" int.log | tail -20` and examine the last vector before reset. Vector 13 (GPF) means segment selector wrong. Vector 6 (invalid opcode) means the far jump address is wrong and CPU is executing data.

Phase 4 — Linker script, BSS zeroing, C entry point (2–3 hours)

Create `kernel/linker.ld`, `kernel/entry.asm`, `kernel/include/stdint.h`, `kernel/include/stddef.h`, and `kernel/kmain.c` with a minimal body:

```
void kmain(void) {  
  
    volatile uint16_t *vga = (volatile uint16_t *)0xB8000;  
  
    vga[0] = (uint16_t)'K' | (0x0F << 8); // White 'K' on black  
  
    for(;;);  
  
}
```

Build the full kernel:

```
i686-elf-gcc -m32 -ffreestanding -nostdlib -nostdinc -fno-builtin \  
    -fno-stack-protector -c kernel/kmain.c -o kmain.o  
i686-elf-gcc -m32 -ffreestanding -nostdlib -nostdinc -fno-builtin \  
    -fno-stack-protector -c kernel/entry.asm -o entry.o # Use nasm or gas  
i686-elf-ld -T kernel/linker.ld -o kernel.elf entry.o kmain.o  
objcopy -O binary kernel.elf kernel.bin  
# Verify:  
nm kernel.elf | grep kmain # Must show address >= 0x100000  
objdump -h kernel.elf # Verify .text at 0x100000, .bss after .data
```

Add `kernel.bin` to the disk image at sector 9.

Checkpoint 4: `qemu-system-i386 -drive format=raw,file=os.img -serial stdio`. QEMU screen should show a white `K` in the top-left corner. Nothing else. If the screen is blank: check `volatile` on the VGA pointer. If QEMU reboots: `objdump -d kernel.elf | head -30` — verify the first instruction is a valid 32-bit instruction at address `0x100000`.

Verify BSS zeroing: add a `static int bss_test;` global in `kmain.c`, assert its value is 0 via VGA (display `'0'` or `'1'`). If BSS was not zeroed, `bss_test` is non-zero.

Phase 5 — VGA text-mode driver (2–3 hours)

Create `kernel/drivers/vga.c` and `kernel/include/vga.h`. Implement `vga_init`, `vga_putchar`, `vga_scroll`, `vga_write_string`, `vga_set_cursor`, `vga_clear`. Update `kmain.c`:

```
void kmain(void) {  
  
    vga_init();  
  
    vga_write_string("VGA driver OK\n", VGA_WHITE, VGA_BLACK);  
  
    for(;;);  
  
}
```

Checkpoint 5: Screen shows `"VGA driver OK"` on row 0. Add a test that writes 26 lines of `"ABCDEFGHIJKLMNPQRSTUVWXYZ\n"` — after the 26th line, scrolling should kick in and the first line should have scrolled off the top. Verify by inspection. Verify `\n` goes to column 0 of next row. Verify the cursor position after scroll is row 24.

Phase 6 — Serial UART driver and kprintf (2–3 hours)

Create `kernel/drivers/serial.c`, `kernel/include/serial.h`, `kernel/kprintf.c`, `kernel/kprintf.h`. Update `kmain.c`:

```
void kmain(void) {  
    serial_init();  
  
    vga_init();  
  
    kprintf("Boot OK: hex=0x%08x dec=%d str=%s\n", 0xDEADBEEF, -42, "hello");  
  
    for(;;);  
}
```

Launch QEMU with: `qemu-system-i386 -drive format=raw,file=os.img -serial stdio`

Checkpoint 6: Terminal shows `"Boot OK: hex=0xdeadbeef dec=-42 str=hello"`. VGA shows the same string. Verify:

- `0x%08x` → `0xdeadbeef` (8 digits, lowercase)
- `%d` with `-42` → `-42` (not the unsigned representation `4294967254`)
- `%s` → `hello`
- The newline scrolls or moves cursor to next line

Phase 7 — Integration test (2–4 hours)

Update `kmain.c` to print a complete welcome message with multiple format specifiers. Build and run. The complete QEMU command for the final acceptance test:

```
qemu-system-i386 -drive format=raw,file=os.img -serial stdio 2>/dev/null
```

BASH

Checkpoint 7 (final): Screen and terminal both show:

```
Build-OS Kernel v0.1  
-----  
VGA: OK  Serial: OK  
GDT: 5 descriptors loaded  
Mode: 32-bit protected  
CS=0x0008 DS=0x0010  
BSS: zeroed  
Boot complete.
```

The `CS` and `DS` values are read via inline assembly and printed with `%04x`. The BSS test confirms `bss_test == 0`. The output appears on both VGA (row 0 onward) and serial (captured in terminal).

8. Test Specification

All tests are run in QEMU with `-serial stdio`. Each test verifies one specific behavior. Expected output is exact string match unless noted.

8.1 VGA Driver Tests

T-VGA-1: Single character write

- Setup: `vga_init()`, then `vga_putchar('A', VGA_WHITE, VGA_BLACK)`
- Verify: Cell at VGA `[0][0] = 0x0F41` (attribute `0x0F`, char `'A' = 0x41`)
- Verify via: Read `*(volatile uint16_t*)0xB8000 == 0x0F41` and print result via serial

T-VGA-2: Cursor advance

- Setup: `vga_init()`. Call `vga_putchar('X', fg, bg)` 80 times
- Verify: After 80 calls, `cursor_row == 1`, `cursor_col == 0` (auto-wrapped to next row)

T-VGA-3: Newline behavior

- Setup: `vga_init()`. `vga_putchar('\n', fg, bg)` from (row=5, col=30)
- Verify: `cursor_row == 6`, `cursor_col == 0`. No cell written at (5, 30)

T-VGA-4: Scroll trigger

- Setup: `vga_init()`. Write 26 full lines ($26 \times \backslash n$)
- Verify: `cursor_row == 24` (scroll occurred); rows 0–23 contain shifted content; row 24 is blank

T-VGA-5: Color attribute encoding

- Setup: `vga_putchar('Z', 0x4 /*red*/, 0x2 /*green*/)`
- Verify: Written cell = `0x245A` (attribute = `(0x2 << 4) | 0x4 = 0x24`, char = `'Z' = 0x5A`)

T-VGA-6: `volatile` required (MMIO not elided)

- Compile `vga.c` with optimization `-O2`. Confirm the write to `VGA_BASE` is present in the disassembly: `objdump -d vga.o | grep -A5 "mov.*0xb8000"` must show a `mov` instruction to address `0xB8000`

T-VGA-7: `vga_clear` resets all cells

- Setup: Write various characters. Call `vga_clear()`.
- Verify: All 2000 cells = space + attribute `0x07`. `cursor_row == 0`, `cursor_col == 0`

T-VGA-8: `vga_scroll` does not corrupt row 0

- Setup: Write known values to row 0. Fill screen to trigger scroll.
- Verify: Row 0 after scroll contains what was previously on row 1, not garbage

8.2 Serial Driver Tests

T-SER-1: Initialization completes

- Verify: `serial_init()` returns (does not hang). Then `serial_putchar('X')` produces `'X'` on terminal

T-SER-2: CRLF conversion

- Setup: `serial_write_string("hello\nworld")`
- Verify: Terminal receives bytes `h e l l o \r \n w o r l d` (the `\n` sends both `\r` and `\n`)
- Verify by: capturing output with `| xxd` in QEMU serial pipe

T-SER-3: Busy-wait respects THRE

- Verify: After writing 100 characters at high rate, no character is lost and no garbled output appears on terminal. This tests that the THRE polling is working.

T-SER-4: Baud rate sanity

- Verify: Output appears at normal readable speed in terminal (not instant = using interrupts incorrectly, not frozen = THRE never set)
-

8.3 kprintf Tests

T-KPRINTF-1: %d signed integer

- `kprintf("%d", -1) → "-1"` on both outputs
- `kprintf("%d", 0) → "0"`
- `kprintf("%d", 2147483647) → "2147483647" (INT32_MAX)`
- `kprintf("%d", -2147483648) → "-2147483648" (INT32_MIN)`

T-KPRINTF-2: %u unsigned integer

- `kprintf("%u", 4294967295u) → "4294967295" (UINT32_MAX)`
- `kprintf("%u", 0) → "0"`

T-KPRINTF-3: %x hexadecimal

- `kprintf("%x", 0xDEADBEEF) → "deadbeef" (lowercase)`
- `kprintf("%X", 0xDEADBEEF) → "DEADBEEF" (uppercase)`
- `kprintf("%x", 0) → "0"`
- `kprintf("%08x", 0xFF) → "000000ff" (zero-padded to 8)`
- `kprintf("%08x", 0x0) → "00000000"`

T-KPRINTF-4: %s string

- `kprintf("%s", "hello") → "hello"`
- `kprintf("%s", NULL) → "(null)"`
- `kprintf("%s", "") → "" (empty, no crash)`

T-KPRINTF-5: %c character

- `kprintf("%c", 'A') → "A"`
- `kprintf("%c", '\n') → newline (cursor moves)`

T-KPRINTF-6: %% escape

- `kprintf("100%%") → "100%"`

T-KPRINTF-7: Mixed format

- `kprintf("PID=%d addr=0x%08x name=%s\n", 42, 0x100000, "kernel") → "PID=42 addr=0x00100000 name=kernel\n"`

T-KPRINTF-8: Both outputs receive identical content

- Any `kprintf` call: verify VGA display and serial terminal receive the same bytes in the same order
-

8.4 GDT and Boot Tests

T-GDT-1: GDT loaded correctly

- Verify: After boot, read GDTR via inline asm `sgdt [buf]` and confirm: limit=39, base=expected address
- `kprintf("GDTR base=0x%08x limit=%d\n", gdtr_base, gdtr_limit)`

T-GDT-2: Segment registers

- Read CS, DS via inline asm:

```
uint16_t cs, ds;  
  
__asm__("mov %%cs, %0" : "=r"(cs));  
  
__asm__("mov %%ds, %0" : "=r"(ds));  
  
kprintf("CS=0x%04x DS=0x%04x\n", cs, ds);
```

- Expected: CS=0x0008 DS=0x0010

T-GDT-3: Protection active — null descriptor fault

- Attempt to load DS with selector 0x00 and verify GPF (triple fault) occurs
- This is a destructive test; run in a separate QEMU session with `-d int` and verify `v=0d` (GPF vector 13) appears immediately

T-BOOT-1: Exact 512-byte MBR

- `wc -c stage1.bin` → 512

T-BOOT-2: Boot signature

- `xxd stage1.bin | tail -1` → last bytes are 55 aa

T-BOOT-3: BSS is zero

- Declare `static volatile uint32_t bss_canary;` in `kmain.c` (no initializer)
- In `kmain`: `kprintf("BSS canary: %u (expect 0)\n", bss_canary);`
- Expected: "BSS canary: 0 (expect 0)"

T-BOOT-4: Kernel load address

- `nm kernel.elf | grep "T kmain"` → address ≥ 0x100000
- `objdump -h kernel.elf` → `.text VMA = 0x00100000`

T-BOOT-5: A20 operational

- Write `0xDEAD` to physical `0x100000` (in Stage2, before kernel load)
- Write `0xBEEF` to physical `0x000000` (IVT entry 0)
- Verify the two values are different (A20 enabled, no wraparound)

9. Performance Targets

Operation	Target	How to Measure
Total boot to <code>kmain()</code>	< 500ms	<code>date</code> before QEMU launch; observe first VGA character
INT 13h sector read (QEMU)	< 1ms per sector	Timestamp via PIT if available; otherwise empirical
<code>vga_putchar</code> single character	100–200ns	Count cycles with <code>rdtsc</code> before/after; divide by clock rate
<code>vga_scroll</code> (full screen shift)	< 1ms (2000 MMIO writes × 200ns)	<code>rdtsc</code> around scroll call
<code>serial_putchar</code> single byte	< 270µs (one char-time at 38400 baud)	Measure wall time for 100 chars: should be ~26ms
BSS zero (64KB)	< 50µs	<code>rdtsc</code> ; 16K dword stores at ~1 cycle each ÷ clock
<code>kprintf</code> 80-char string to both outputs	< 25ms	Dominated by serial (~260µs × 80 chars)
GDT load (<code>lgdt</code> + far jump)	< 100 cycles (one-time)	<code>rdtsc</code> before/after in Stage2; subtract baseline
<code>kprintf</code> integer formatting (no I/O)	< 500ns	Instrument with <code>rdtsc</code> ; subtract VGA/serial time

How to use `rdtsc` before interrupt infrastructure exists:

```
static inline uint64_t rdtsc(void) {  
    uint32_t lo, hi;  
  
    __asm__ volatile ("rdtsc" : "=a"(lo), "=d"(hi));  
  
    return ((uint64_t)hi << 32) | lo;  
}  
  
// Usage:  
  
uint64_t t0 = rdtsc();  
  
vga_scroll();  
  
uint64_t t1 = rdtsc();  
  
kprintf("scroll cycles: %u\n", (uint32_t)(t1 - t0));
```

QEMU's virtual TSC runs at approximately the host CPU frequency. On a 2GHz host: 1 cycle ≈ 0.5ns. Use this to convert cycle counts to wall-clock estimates.

10. Hardware Soul Analysis

VGA MMIO at `0xB8000`

Cache behavior: The VGA buffer is a physical MMIO region. On real hardware, the CPU's MTRR (Memory Type Range Registers) or PAT (Page Attribute Table) mark it as Write-Combining or Uncacheable — writes bypass L1/L2/L3 and go directly to the ISA bus

bridge. On QEMU, writes to `0xB8000` are intercepted by the virtual VGA device and rendered immediately. No cache warm-up effect exists. Each `uint16_t` write costs a full memory bus transaction (~100–200ns on real hardware; faster on QEMU).

volatile is mandatory: Without `volatile`, GCC at `-O2` observes that the VGA array is written-to but never read back from C code, treats the writes as dead stores, and eliminates them entirely. The compiled binary contains no reference to `0xB8000`. The `volatile` qualifier forces every store to be emitted as a machine instruction, regardless of optimization level.

TLB irrelevance: No page tables are active in this module. The CPU translates addresses as physical = virtual (the MMU is bypassed before `CRO.PG` is set, which does not happen until Milestone 3). There is no TLB miss possible here.

Serial Port I/O

Port I/O vs MMIO: Unlike VGA, the UART uses the x86 I/O address space (a separate 16-bit address space accessed via `in / out` instructions, distinct from the memory address space). The `in / out` instructions stall the CPU until the I/O bus transaction completes — legacy ISA devices respond in 1–4 μ s. This is why `io_wait()` uses `outb(0x80, 0)` — a deliberate 1–4 μ s stall between UART configuration writes to allow the chip's internal state machine to settle.

Branch predictability: The `serial_putchar` busy-wait loop (`while (!(inb(0x3FD) & 0x20))`) is a tight I/O-bound spin. The branch is perfectly predictable in the "spin" direction (the CPU predicts "not ready" for the first several iterations) and mispredicts exactly once (when THRE transitions from 0 to 1). Misprediction cost: ~15 cycles — negligible compared to the 260 μ s character time.

GDT Load and Far Jump

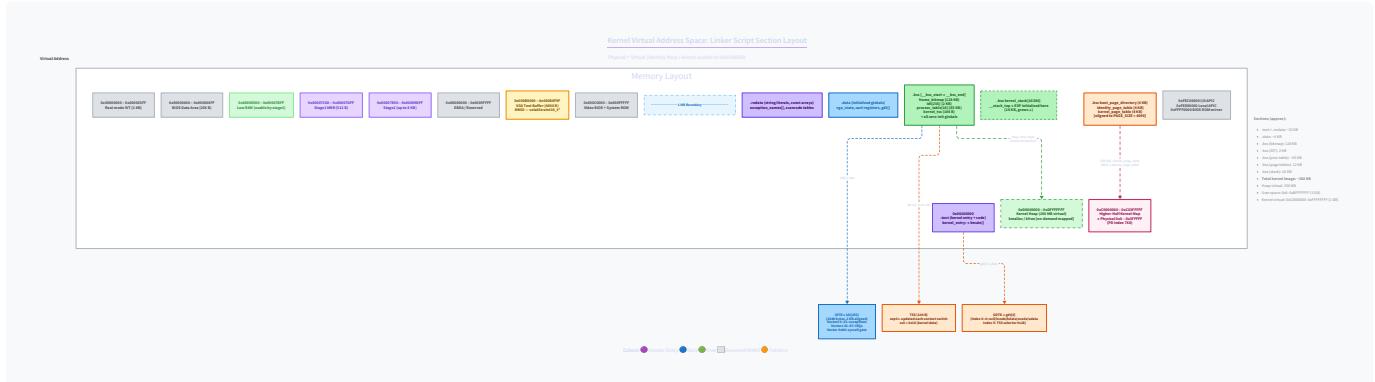
Pipeline flush cost: The far jump after `CRO.PE` discards all prefetched and partially-decoded instructions in the CPU's out-of-order execution pipeline. On a modern CPU with a 20-stage pipeline and aggressive out-of-order execution, this flushes up to 100+ in-flight micro-operations. Cost: ~20–30 cycles — a one-time expense that ensures no real-mode-decoded instructions execute in protected mode.

lgdt instruction behavior: `lgdt` is a serializing instruction — it completes all previous memory operations before executing, and subsequent instructions see the new GDTR value. It does not flush the TLB (no page tables yet) but does serialize the instruction stream.

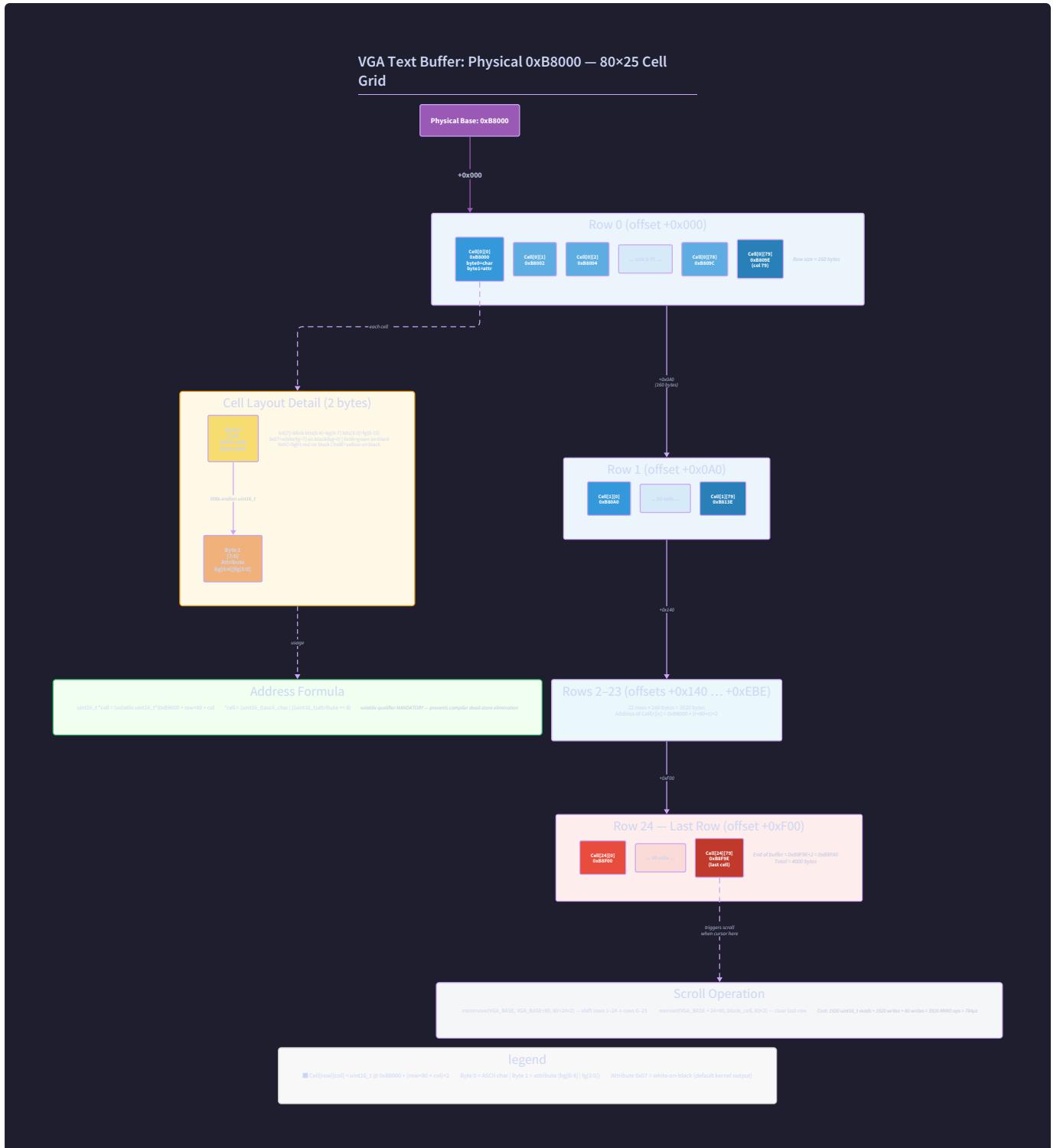
BSS Zeroing

rep stosd microarchitecture: On modern x86 CPUs, `rep stosd` is recognized by the microcode as a "fast string" operation and is executed as a DMA-like operation within the CPU core, filling ~16 bytes per cycle (4 dwords per cycle). For a 64KB BSS section: 16384 dword stores at 4 stores/cycle = ~4096 cycles = ~2 μ s at 2GHz. The target memory is in DRAM (not yet in any cache); the CPU will generate cache-line-sized (64-byte) store bursts, filling 16 dwords per cache line write = efficient memory bandwidth utilization.

Memory access pattern: Sequential write from `__bss_start` to `__bss_end`. Hardware prefetcher detects the stride and issues prefetch requests for upcoming cache lines. Effective throughput approaches memory bandwidth limit rather than being limited by cache miss latency. This is the ideal access pattern for bulk initialization.



11. State Diagram



States:

State	CR0.PE	IF	CS Value	Valid Actions
REAL_MODE_16BIT	0	1	0x0000	BIOS calls, real-mode execution
UNREAL_MODE	0 → 1 → 0	0	0x0000	Read/write > 1MB via extended segment
CLI_ACTIVE	0	0	0x0000	<code>lgdt</code> only; no BIOS calls
GDT_LOADED	0	0	0x0000	<code>mov cr0, eax</code> only
PROTECTED_MODE_UNSTABLE	1	0	0x0000 (stale!)	<code>jmp 0x08:label</code> ONLY
PROTECTED_MODE_STABLE	1	0	0x0008	Segment register loads, kernel jump
KERNEL_EXECUTING	1	0	0x0008	All valid kernel operations

Illegal transitions:

- PROTECTED_MODE_* → REAL_MODE : Undefined behavior; no supported return path in this module
- CLI_ACTIVE → BIOS interrupt fires: Results in TRIPLE_FAULT (IVT invalid in protected mode)
- GDT_LOADED → any operation except `mov cr0` : Technically legal but wastes the atomic protection window
- PROTECTED_MODE_UNSTABLE → any instruction except far jump: CPU prefetch may execute real-mode-decoded instructions in protected mode

12. Concurrency Specification

This module runs on a single core with interrupts disabled from `cli` (in Stage2) through the end of `kmain()`. There is no concurrency. However, two aspects of the code require reasoning about execution order that resembles a concurrency concern:

MMIO write ordering (VGA): The `volatile` qualifier ensures the compiler emits stores to `0xB8000` in program order. On x86, all stores are visible to the I/O bus in program order (x86 Total Store Order guarantees store-store ordering). No `sfence` or `mfence` is needed because there are no other CPUs or DMA agents that could observe reordering.

Port I/O ordering (UART): The `io_wait()` call between UART configuration writes acts as a timing barrier, not a memory ordering barrier. The UART chip's state machine requires settling time between configuration writes. The `outb(0x80, 0)` in `io_wait()` is an I/O write that stalls the CPU for ~1–4µs while the I/O bus transaction completes. This is sufficient for the 16550 UART's state machine transitions. No explicit memory fence is required because each `outb` is already a serializing operation on legacy I/O ports.

13. Synced Criteria

Technical Design Specification: Interrupts, Exceptions, and Keyboard

Module ID: mod-2 | Milestone: build-os-m2 | Estimated Hours: 18–26

1. Module Charter

This module configures the x86 interrupt infrastructure — the IDT, 8259 PIC cascade, CPU exception handlers, hardware IRQ handlers, PIT timer, and PS/2 keyboard driver — starting from a protected-mode kernel with no interrupt handling and ending with all

256 IDT gates populated, interrupts enabled, a 100Hz tick counter running, and a keyboard input buffer available for polling. Every interrupt that fires after `sti` goes through a path this module owns.

This module does **not** implement a process scheduler (the timer handler only increments a counter), page tables or virtual memory, a kernel heap, the TSS, system calls, or the INT 0x80 gate (DPL=3 trap gate setup is declared here for completeness but activated in Milestone 4). The module does not implement APIC or MSI-X — only the 8259 PIC cascade is used. No floating-point state (x87/SSE registers) is saved in interrupt handlers.

Upstream dependency: Module `mod-1` — protected mode active, GDT loaded with kernel code (0x08) and data (0x10) selectors, VGA and serial output functional, `kprintf` available, interrupts disabled.

Downstream dependency: Module `mod-3` (memory management) receives a running timer tick counter (`pit_tick_count`) and the IDT infrastructure to register a page fault handler. Module `mod-4` (scheduling) installs the scheduler into the timer IRQ handler slot and registers the system call gate.

Invariants that must hold at module exit:

1. IDT contains exactly 256 valid gate descriptors; the IDTR register points to it with limit = 2047.
2. Master PIC delivers IRQ0–7 at vectors 32–39; slave PIC delivers IRQ8–15 at vectors 40–47. Vectors 0–31 are reserved exclusively for CPU exceptions.
3. Every ISR/IRQ handler restores the interrupted CPU to an identical register state (all 8 general-purpose registers, 4 segment registers, EFLAGS, EIP) unless the handler explicitly modifies the saved frame to change behavior.
4. EOI is sent to the PIC on every hardware IRQ invocation before returning from the IRQ handler.
5. `pit_tick_count` increments by 1 per timer period (10ms at 100Hz); it is never incremented by software.
6. `keyboard_getchar()` returns bytes placed there exclusively by the IRQ1 handler; no other code path writes to the circular buffer.
7. Interrupts are enabled (`EFLAGS.IF = 1`) when `kmain` reaches its idle loop.

2. File Structure

Create files in this exact sequence:

```
build-os/
└── kernel/
    ├── include/
    │   ├── 01  idt.h          # IDT gate descriptor struct, IDTR, gate type constants, idt_set_gate(),
    │   ├── 02  interrupt.h    # interrupt_frame_t layout, exception_handler() declaration
    │   ├── 03  pic.h          # PIC port constants, pic_remap(), pic_send_eoi(), pic_set_mask(),
    │   ├── 04  pit.h           # pit_init(), pit_tick_count declaration
    │   ├── 05  keyboard.h     # keyboard_init(), keyboard_getchar(), KB_BUFFER_SIZE
    │   └── 06  irq.h          # irq_install_handler(), irq_handler_t typedef
    ├── interrupts/
    │   ├── 01  idt.c          # idt[] array, idt_set_gate(), idt_install(), idt_setup_all()
    │   ├── 02  isr_stubs.asm   # ISR_NOERR/ISR_ERR macros for vectors 0-31, isr_common_stub
    │   ├── 03  irq_stubs.asm   # IRQ_STUB macro for IRQ0-15, irq_common_stub, irq_return_trampoline
    │   ├── 04  interrupt.c     # exception_handler(), interrupt_frame_t, irq_dispatcher(),
    ├── irq_install_handler()
    │   ├── 05  pic.c           # pic_remap(), pic_send_eoi(), pic_set_mask(), pic_get_isr(), io_wait()
    │   ├── 06  pit.c           # pit_set_frequency(), pit_init(), timer_handler(), pit_tick_count
    │   └── 07  keyboard.c       # keyboard_handler(), keyboard_init(), keyboard_getchar(), scancode table
    └── kmain.c                  # Updated: calls idt_setup_all(), pic_remap(), pit_init(), keyboard_init(),
                                # sti
```

Why this order: `idt.h` must exist before `interrupt.h` (which references gate types), before `irq.h` (which references `interrupt_frame_t`). Assembly stubs reference C symbols declared in headers; headers must exist when assembly is assembled. `pic.c` must compile before `pit.c` and `keyboard.c` because both call `irq_install_handler` which references the IRQ dispatcher registered in `interrupt.c`.

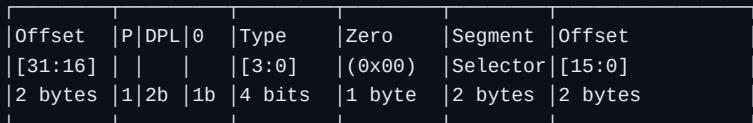
3. Complete Data Model

3.1 IDT Gate Descriptor — Byte-Level Layout

Each IDT entry is exactly **8 bytes**, hardware-defined by Intel. The layout must match the CPU specification precisely; `__attribute__((packed))` is mandatory.

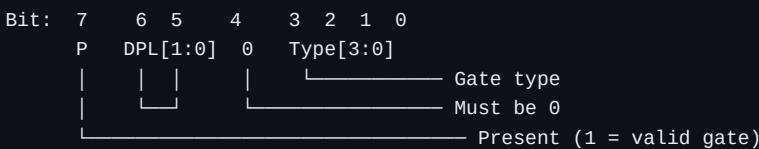
Byte offset within one 8-byte IDT gate entry:

Byte: 7 6 5 4 3-2 1-0



Byte 0-1: offset_low Handler address bits [15:0]
 Byte 2-3: selector Code segment selector to use for handler (always 0x08: kernel code)
 Byte 4: zero Reserved; must be 0x00
 Byte 5: type_attr Packed field: P|DPL[1:0]|0|Type[3:0]
 Byte 6-7: offset_high Handler address bits [31:16]

`type_attr` bit decomposition (byte 5):



Gate types used in this module:

0xE = 0b1110 32-bit Interrupt Gate (clears IF on entry)
 0xF = 0b1111 32-bit Trap Gate (preserves IF on entry)

Predefined `type_attr` constants:

Constant	Value	Binary	Meaning
<code>IDT_INTERRUPT_GATE</code>	<code>0x8E</code>	<code>1 00 0 1110</code>	P=1, DPL=0, 32-bit interrupt gate; IF cleared on entry
<code>IDT_TRAP_GATE</code>	<code>0x8F</code>	<code>1 00 0 1111</code>	P=1, DPL=0, 32-bit trap gate; IF preserved on entry
<code>IDT_SYSCALL_GATE</code>	<code>0xEF</code>	<code>1 11 0 1111</code>	P=1, DPL=3, 32-bit trap gate; user-mode callable via <code>int N</code>

Why interrupt gate for IRQ handlers: Setting IF=0 on entry prevents a timer IRQ from interrupting the timer handler (no reentrancy), and prevents a keyboard IRQ from interrupting the keyboard handler. This simplifies handler design at the cost of slightly increased interrupt latency for other IRQs while the handler runs. Acceptable for a single-core kernel with short handlers (<100 cycles each).

Why trap gate for CPU exceptions: Exceptions (divide-by-zero, page fault, GPF) benefit from IF=1 during handling — a page fault handler may need to perform I/O that requires timer ticks. Trap gates also allow a GDB stub to break in during exception handling. For this module, all exception handlers halt, so the distinction is academic but correct.

C struct definition:

```
/* include/idt.h */  
  
#include <stdint.h>  
  
typedef struct __attribute__((packed)) {  
  
    uint16_t offset_low;      /* Handler address [15:0] */  
  
    uint16_t selector;       /* Code segment selector (0x08) */  
  
    uint8_t zero;            /* Always 0x00 per Intel spec */  
  
    uint8_t type_attr;       /* P | DPL | 0 | Type */  
  
    uint16_t offset_high;    /* Handler address [31:16] */  
  
} idt_gate_t;                /* sizeof = 8; assert in idt.c */  
  
typedef struct __attribute__((packed)) {  
  
    uint16_t limit;          /* Size of IDT in bytes minus 1 = 2047 */  
  
    uint32_t base;            /* Physical address of idt[] array */  
  
} idtr_t;                   /* sizeof = 6 */
```

Memory layout of `idt[]` array:

```
idt[0]  bytes 0-7:   Vector 0  (Divide Error)  
idt[1]  bytes 8-15:  Vector 1  (Debug)  
...  
idt[31] bytes 248-255: Vector 31 (Reserved)  
idt[32] bytes 256-263: Vector 32 (IRQ0: Timer)  
...  
idt[47] bytes 376-383: Vector 47 (IRQ15: Secondary ATA)  
idt[48]-idt[127]: Unused; installed with a default_isr stub (prints and halts)  
idt[128] bytes 1024-1031: Vector 0x80 (system call gate, DPL=3; configured in mod-4)  
idt[129]-idt[255]: Unused; default_isr stub  
  
Total: 256 × 8 = 2048 bytes. IDTR.limit = 2047 (0x07FF).
```



3.2 `interrupt_frame_t` — The Stack Frame Layout

This struct maps exactly onto the kernel stack contents when the C handler is called from `isr_common_stub` or `irq_common_stub`. Every byte offset is load-bearing — the assembly stubs push values in a fixed order and the C code reads them by field offset.

Stack state when `exception_handler(interrupt_frame_t *frame)` is called:

The argument `frame` is a pointer to the stack, laid out (from lower to higher address) as:

Low address (top of stack, frame pointer)

gs	[frame + 0]	uint32_t	Saved GS
fs	[frame + 4]	uint32_t	Saved FS
es	[frame + 8]	uint32_t	Saved ES
ds	[frame + 12]	uint32_t	Saved DS
edi	[frame + 16]	uint32_t	From pusha
esi	[frame + 20]	uint32_t	From pusha
ebp	[frame + 24]	uint32_t	From pusha
esp_pusha	[frame + 28]	uint32_t	ESP at pusha (ignored by popa)
ebx	[frame + 32]	uint32_t	From pusha
edx	[frame + 36]	uint32_t	From pusha
ecx	[frame + 40]	uint32_t	From pusha
eax	[frame + 44]	uint32_t	From pusha
vector	[frame + 48]	uint32_t	Pushed by ISR stub
error_code	[frame + 52]	uint32_t	CPU-pushed or dummy 0
eip	[frame + 56]	uint32_t	CPU-pushed return address
cs	[frame + 60]	uint32_t	CPU-pushed code segment
eflags	[frame + 64]	uint32_t	CPU-pushed flags
user_esp	[frame + 68]	uint32_t	Only if ring-3 → ring-0
user_ss	[frame + 72]	uint32_t	Only if ring-3 → ring-0

High address (bottom of stack)

C struct definition:

```

/* include/interrupt.h */

#include <stdint.h>

typedef struct {

    /* Saved by isr_common_stub / irq_common_stub (in push order, low→high) */

    uint32_t gs, fs, es, ds;           /* Segment registers (pushed last → first pop) */

    uint32_t edi, esi, ebp, esp_pusha; /* pusha order: EDI, ESI, EBP, ESP, EBX, EDX, ECX, EAX */

    uint32_t ebx, edx, ecx, eax;

    /* Pushed by ISR/IRQ stub before jmp common_stub */

    uint32_t vector;                 /* Interrupt vector number (0–255) */

    uint32_t error_code;             /* CPU error code or 0 (dummy) for no-error exceptions */

    /* CPU-pushed automatically on interrupt (always present) */

    uint32_t eip;                   /* Return address (instruction to resume) */

    uint32_t cs;                    /* Code segment at time of interrupt */

    uint32_t eflags;                /* EFLAGS at time of interrupt */

    /* CPU-pushed ONLY for ring-3 → ring-0 transitions */

    uint32_t user_esp;              /* User-mode ESP (only valid when cs & 3 == 3) */

    uint32_t user_ss;               /* User-mode SS (only valid when cs & 3 == 3) */

} interrupt_frame_t;

/* Check ring level: was the interrupted code in user mode? */

#define FRAME_FROM_RING3(frame) (((frame)->cs & 0x3) == 3)

```

Why `esp_pusha` is not the real ESP: The `pusha` instruction pushes ESP's value as it was *before* `pusha` executed (i.e., before the eight pushes). The `popa` instruction ignores the ESP field it pops (it does not restore ESP from the saved value; ESP is advanced by 32 bytes by the `popa` instruction itself). Field `esp_pusha` is present in the struct for layout accuracy but should not be used to read or write the interrupted process's stack pointer — use `user_esp` for user-mode stack pointer access.



3.3 8259 PIC Register Map

Two 8259 PICs in cascade. All access via `outb` / `inb` port I/O.

Port	PIC	Direction	Register	Purpose
0x20	Master	Write	Command register	Send ICW1, OCW2 (EOI), OCW3
0x21	Master	Read/Write	Data register	ICW2, ICW3, ICW4, IMR read/write
0xA0	Slave	Write	Command register	Send ICW1, OCW2 (EOI), OCW3
0xA1	Slave	Read/Write	Data register	ICW2, ICW3, ICW4, IMR read/write

IRQ-to-device mapping (both before and after remapping):

IRQ	Device	Default Vector	Remapped Vector
0	PIT Timer	8 (COLLISION : double fault)	32
1	PS/2 Keyboard	9 (COLLISION : invalid TSS)	33
2	Cascade (slave PIC)	10 (COLLISION : segment not present)	34
3	COM2 Serial	11 (COLLISION : stack fault)	35
4	COM1 Serial	12 (COLLISION : GPF)	36
5	LPT2 / Sound	13 (COLLISION : GPF)	37
6	Floppy	14 (COLLISION : page fault)	38
7	LPT1 / Spurious	15 (COLLISION : reserved)	39
8	RTC	112	40
9	Free	113	41
10	Free	114	42
11	Free	115	43
12	PS/2 Mouse	116	44
13	FPU	117	45
14	Primary ATA	118	46
15	Secondary ATA / Spurious	119	47

ICW (Initialization Command Word) byte values:

ICW	Value	Meaning
ICW1_INIT	0x10	Start initialization; level-triggered = 0 (edge-triggered); cascade = 0 (ICW3 needed)
ICW1_ICW4	0x01	ICW4 will follow
ICW1 combined	0x11	`ICW1_INIT
ICW2 master	0x20	Master vector offset = 32 (IRQ0 → vector 32)
ICW2 slave	0x28	Slave vector offset = 40 (IRQ8 → vector 40)
ICW3 master	0x04	Slave PIC connected to master IRQ2 (bitmask: bit 2 = 1)
ICW3 slave	0x02	Slave cascade identity = 2 (numeric IRQ line on master)
ICW4	0x01	8086/88 mode (not MCS-80/85); non-buffered; normal EOI
PIC_EOI	0x20	End of Interrupt command (sent to OCW2)

ISR register read (for spurious IRQ detection):

To read the In-Service Register (which IRQs are currently being handled):

```
outb(PIC1_CMD, 0x0B); /* OCW3: read ISR */

uint8_t master_isr = inb(PIC1_CMD);

outb(PIC2_CMD, 0x0B);

uint8_t slave_isr = inb(PIC2_CMD);
```

IMR (Interrupt Mask Register): Read at any time from data port. Bit N set = IRQ N masked (suppressed). After remapping, the saved pre-remap masks are restored.

3.4 PIT (8253/8254) Register Map

Port	Direction	Purpose
0x40	Write	Channel 0 counter data (lo byte, then hi byte)
0x41	Write	Channel 1 (DRAM refresh — do not touch)
0x42	Write	Channel 2 (PC speaker)
0x43	Write	Mode/Command register (write only)

Command byte for channel 0 at 100Hz:

```
0x36 = 0b 00 11 011 0
      ^^ channel 0 (bits 7:6 = 00)
      ^^ lo/hi byte access (bits 5:4 = 11)
      ^^^ mode 2 = rate generator (bits 3:1 = 011)
          ^ binary counting (bit 0 = 0)
```

Divisor calculation:

```

PIT input frequency: 1,193,182 Hz (exact: 1193181.6666... Hz)
Target frequency:    100 Hz
Divisor:           1,193,182 / 100 = 11,931.82 → round to 11,932
Actual frequency:   1,193,182 / 11,932 = 99.9987 Hz (error < 0.002%)

```

Divisor byte sequence: Channel 0 expects low byte first, then high byte (because mode/command byte specified `11` for lo/hi access):

```

outb(0x43, 0x36);                      /* Command byte */

outb(0x40, (uint8_t)(11932 & 0xFF));    /* Divisor low byte = 0xDC */

outb(0x40, (uint8_t)((11932 >> 8) & 0xFF)); /* Divisor high byte = 0x2E */

```

3.5 PS/2 Keyboard Port Map

Port	Direction	Purpose
0x60	Read	PS/2 Data Port: next scancode byte
0x60	Write	Send command to keyboard controller
0x64	Read	PS/2 Status Register: bit 0 = output buffer full
0x64	Write	Send command to PS/2 controller

Scancode Set 1 make codes (single byte, 0x01–0x58):

A make code is generated when a key is **pressed**. A break code is the make code with bit 7 set (OR with `0x80`). The driver must filter break codes.

Extended keys (preceded by `0xE0`) include: right Ctrl, right Alt, arrow keys, Insert, Delete, Home, End, PgUp, PgDn, Numpad Enter, Numpad `/`. This module handles only single-byte (non-`0xE0`) codes; extended keys are silently ignored.

Full scancode-to-ASCII table (128 entries, index = make code):

```

static const char scancode_ascii_lower[128] = {

/*00*/  0,      27,      '1',      '2',      '3',      '4',      '5',      '6',
/*08*/  '7',      '8',      '9',      '0',      '-',      '=',      '\b',      '\t',
/*10*/  'q',      'w',      'e',      'r',      't',      'y',      'u',      'i',
/*18*/  'o',      'p',      '[',      ']',      '\n',      0,      'a',      's',
/*20*/  'd',      'f',      'g',      'h',      'j',      'k',      'l',      ';',
/*28*/  '\'',      '`',      0,      '\\\\',      'z',      'x',      'c',      'v',
/*30*/  'b',      'n',      'm',      '.',      '.',      '/',      0,      '*',
/*38*/  0,      ' ',      0,      0,      0,      0,      0,      0,
/*40*/  0,      0,      0,      0,      0,      0,      0,      '7',
/*48*/  '8',      '9',      '-',      '4',      '5',      '6',      '+',      '1',
/*50*/  '2',      '3',      '0',      '.',      0,      0,      0,      0,
/*58*/  0,      0,      0,      0,      0,      0,      0,      0,
/*60*/  0,      0,      0,      0,      0,      0,      0,      0,
/*68*/  0,      0,      0,      0,      0,      0,      0,      0,
/*70*/  0,      0,      0,      0,      0,      0,      0,      0,
/*78*/  0,      0,      0,      0,      0,      0,      0,      0

};

/* Shifted versions of keys where shift changes the character */

static const char scancode_ascii_upper[128] = {

/*00*/  0,      27,      '!',      '@',      '#',      '$',      '%',      '^',
/*08*/  '&',      '*',      '(',      ')',      '_',      '+',      '\b',      '\t',
/*10*/  'Q',      'W',      'E',      'R',      'T',      'Y',      'U',      'I',
/*18*/  'O',      'P',      '{',      '}',      '\n',      0,      'A',      'S',
/*20*/  'D',      'F',      'G',      'H',      'J',      'K',      'L',      ':',
/*28*/  '\'',      '~',      0,      '|',      'Z',      'X',      'C',      'V',
/*30*/  'B',      'N',      'M',      '<',      '>',      '?',      0,      '*',
/*38*/  0,      ' ',      0,      0,      0,      0,      0,      0,

    /* Numpad and function keys: same as lower */

    0,      0,      0,      0,      0,      0,      0,      0,      '7',
    '8',      '9',      '-',      '4',      '5',      '6',      '+',      '1',
    '2',      '3',      '0',      '.',      0,      0,      0,      0,
};

```

```

    0,    0,    0,    0,    0,    0,    0,    0,
    0,    0,    0,    0,    0,    0,    0,    0,
    0,    0,    0,    0,    0,    0,    0,    0,
    0,    0,    0,    0,    0,    0,    0,    0,
    0,    0,    0,    0,    0,    0,    0,    0
};


```

Modifier key make codes:

Key	Make Code	Break Code
Left Shift	0x2A	0xAA
Right Shift	0x36	0xB6
Left Ctrl	0x1D	0x9D
Left Alt	0x38	0xB8
Caps Lock	0x3A	0xBA

This module tracks Left Shift and Right Shift (toggles `kb_shift_hold`). Ctrl, Alt, and Caps Lock are detected and silently ignored (no further action). Extended key prefix `0xE0` causes the driver to set a one-shot flag that discards the next scancode byte; this prevents `0xE0 0x1D` (right Ctrl) from being interpreted as `0x1D` (left Ctrl).

3.6 Circular Keyboard Buffer

```

#define KB_BUFFER_SIZE 256 /* Must be a power of 2 for mask-based wrap */

typedef struct {

    char     buf[KB_BUFFER_SIZE]; /* Storage */

    uint8_t  head;             /* Next write position (written by IRQ1) */

    uint8_t  tail;             /* Next read position (read by getchar) */

} kb_ring_t;

```

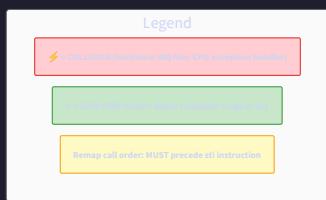
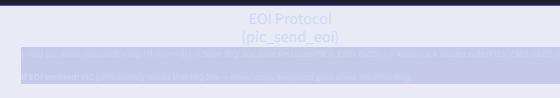
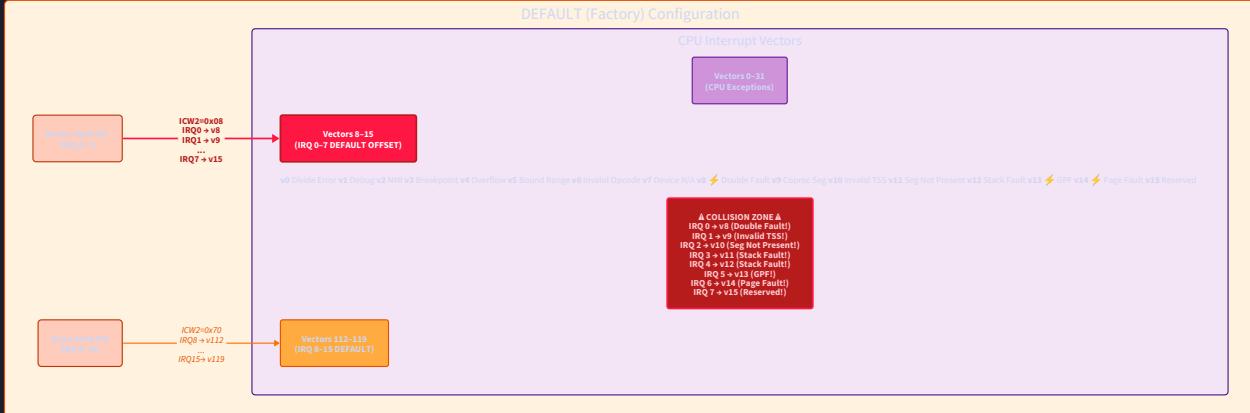
Invariants:

- Buffer is empty when `head == tail`.
- Buffer is full when `(head + 1) & (KB_BUFFER_SIZE - 1) == tail`.
- `head` and `tail` are `uint8_t` (8-bit). With `KB_BUFFER_SIZE = 256`, incrementing wraps automatically via unsigned overflow — no modulo operation needed.
- Exactly one writer (IRQ1 handler) and exactly one reader (`keyboard_getchar()`). On a single-core system with IF=0 during the IRQ1 handler, no locking is required.
- On overflow (buffer full), the incoming character is **silently dropped** (head is not advanced). The oldest unread characters are preserved. This is the standard terminal behavior.

Memory layout:

Offset	Field	Type	Purpose
0	buf	char[256]	Character storage; circular
256	head	uint8_t	IRQ1 handler writes here; advance after write
257	tail	uint8_t	getchar reads here; advance after read
258	(pad)	2 bytes	Align to 4 bytes
Total: 260 bytes			

PIC Default vs Remapped Vector Assignment — Collision Diagram



4. Interface Contracts

4.1 `idt_set_gate(uint8_t vector, uint32_t handler, uint16_t selector, uint8_t type_attr) → void`

Parameters:

- `vector` : IDT index [0, 255]. No range check performed; caller is responsible. Out-of-range value writes beyond the IDT array (undefined behavior).
- `handler` : Full 32-bit virtual address of the ISR/IRQ assembly stub. Must be a valid code address in kernel space. If 0 is passed, the gate is technically valid (handler at address 0) but will crash if invoked; this is not checked.
- `selector` : GDT code segment selector. Must be `0x08` (kernel code segment) for all handlers in this module. Other values are accepted but result in a GPF if the selector is not a valid present code descriptor.
- `type_attr` : Gate type and flags byte. Must be one of `IDT_INTERRUPT_GATE` (0x8E), `IDT_TRAP_GATE` (0x8F), or `IDT_SYSCALL_GATE` (0xEF). Other values produce non-standard gate configurations and may cause GPF on invocation.

Post-conditions: `idt[vector].offset_low == handler & 0xFFFF`, `idt[vector].offset_high == (handler >> 16) & 0xFFFF`, `idt[vector].selector == selector`, `idt[vector].zero == 0`, `idt[vector].type_attr == type_attr`.

Errors: None propagated. Silent on invalid inputs.

Implementation:

```
void idt_set_gate(uint8_t vector, uint32_t handler, uint16_t selector, uint8_t type_attr) {  
    idt[vector].offset_low = (uint16_t)(handler & 0xFFFF);  
    idt[vector].selector = selector;  
    idt[vector].zero = 0x00;  
    idt[vector].type_attr = type_attr;  
    idt[vector].offset_high = (uint16_t)((handler >> 16) & 0xFFFF);  
}
```

4.2 `idt_install(void) → void`

Pre-conditions: `idt[]` array fully populated (all 256 entries set to either a real handler or the default stub).

Behavior:

1. Sets `idtr.limit = sizeof(idt) - 1 = 2047`.
2. Sets `idtr.base = (uint32_t)(uintptr_t)idt`.
3. Executes `lidt [idtr]` via inline assembly: `__asm__ volatile ("lidt %0" : : "m"(idtr))`.

Post-conditions: The CPU's IDTR register holds the base address of `idt[]` and limit 2047. Any subsequent interrupt uses this table.

Errors: None. If `idt[]` was not fully populated before this call, subsequent interrupts will invoke uninitialized gate descriptors, which will triple-fault.

4.3 `idt_setup_all(void) -> void`

Behavior: Calls `idt_set_gate` for all 256 vectors, then calls `idt_install()`. Calls are made in vector-number order (0 first, 255 last). All exception vectors (0–31) use `IDT_TRAP_GATE`. All IRQ vectors (32–47) use `IDT_INTERRUPT_GATE`. Vectors 48–127 and 129–255 use `IDT_INTERRUPT_GATE` with the `default_isr` handler. Vector 128 (0x80) is set to `default_isr` with `IDT_INTERRUPT_GATE` in this module; `idt_setup_all` does not install the system call gate (that is mod-4's responsibility, which calls `idt_set_gate(0x80, ...)` after `idt_setup_all` returns).

Post-conditions: IDTR loaded. All 256 gates valid. No interrupt vector points to an uninitialized gate.

Errors: None propagated. If any `isr*` or `irq*` symbol is undefined (linker error), the build fails before this function is ever called.

4.4 `exception_handler(interrupt_frame_t *frame) -> void`

Pre-conditions: Called from `isr_common_stub` with `frame` pointing to the kernel stack containing the saved CPU state. Interrupts may be enabled (trap gate) or disabled (interrupt gate). DS, ES, FS, GS have been loaded with kernel data segment (0x10) by the stub.

Parameters:

- `frame` : Pointer to the complete saved CPU state on the kernel stack. Must not be NULL. The caller (assembly stub) guarantees this.

Behavior for `frame->vector < 32` (CPU exception):

1. Print "\n==== CPU EXCEPTION ====" via `kprintf`.
2. Print "Exception: [name] (vector [N])\n" using the `exception_names[]` table.
3. Print "Error code: 0x%08x\n" using `frame->error_code`.
4. Print "EIP=0x%08x CS=0x%04x EFLAGS=0x%08x\n" .
5. Print "EAX=0x%08x EBX=0x%08x ECX=0x%08x EDX=0x%08x\n" .
6. Print "ESI=0x%08x EDI=0x%08x EBP=0x%08x\n" .
7. If `frame->vector == 14` (page fault): Read CR2 via `mov %%cr2, %0` inline asm. Print "CR2 (faulting addr): 0x%08x\n" .
8. If `FRAME_FROM_RING3(frame)` : Print "User-mode fault: user_esp=0x%08x user_ss=0x%04x\n" .
9. Print "System halted.\n" .
10. Execute `for (;;) { __asm__ volatile ("cli; hlt"); }`.

Behavior for `frame->vector >= 32`: This path should never be reached (IRQ handlers go through `irq_dispatcher`, not `exception_handler`). If somehow invoked: print "Unexpected vector %d in exception_handler\n" and halt.

Return: Never returns for vectors 0–31 (halts). For vectors ≥ 32 , halts as above.

Errors: `kprintf` cannot fail. If VGA/serial were not initialized (impossible given module ordering), output is lost but halt still occurs.

4.5 `irq_install_handler(uint8_t irq, irq_handler_t handler) -> void`

Parameters:

- `irq` : IRQ number [0, 15]. Values outside this range: no-op (silently ignored after bounds check).
- `handler` : Function pointer of type `void (*) (interrupt_frame_t *)`. NULL is valid and means "uninstall the handler for this IRQ" (future IRQs on this line invoke only EOI, no user function).

Post-conditions: `irq_handlers[irq] == handler`.

Thread safety: Single-core only. Must be called before `sti` or with `cli / sti` around the call.

Errors: Out-of-range `irq` (> 15): silently ignored. NULL `handler`: accepted; `irq_dispatcher` checks for NULL before calling.

4.6 `irq_dispatcher(interrupt_frame_t *frame) → void`

Pre-conditions: Called from `irq_common_stub` with `frame` pointing to the kernel stack. `frame->vector` is in range [32, 47]. All segment registers loaded with kernel data selector.

Behavior:

1. Extract `irq = (uint8_t)(frame->vector - 32)`.
2. If `irq_handlers[irq] != NULL`: call `irq_handlers[irq](frame)`.
3. Call `pic_send_eoi(irq)` unconditionally (even if no handler registered, even if the handler did not run).

Return: Returns normally. The assembly stub restores registers and executes `iret`.

Why EOI after handler: Sending EOI before calling the handler would allow the PIC to re-deliver the same IRQ before the handler completes — problematic for non-reentrant handlers. Sending after prevents this at the cost that the handler runs with that IRQ line masked. For this module's short handlers (<100 cycles), the difference is immeasurable.

4.7 `pic_remap(uint8_t master_offset, uint8_t slave_offset) → void`

Parameters:

- `master_offset`: Vector base for IRQ0–7. Must be a multiple of 8 (hardware constraint: the PIC uses the lower 3 bits for IRQ sub-number). Pass `0x20` (32).
- `slave_offset`: Vector base for IRQ8–15. Must be a multiple of 8. Pass `0x28` (40). Must be `master_offset + 8`.

Pre-conditions: Call before `sti`. Call after IDT is fully populated (so no IRQ fires before handlers are ready).

Behavior: Executes the complete ICW1–ICW4 initialization sequence for both PICs with `io_wait()` between each write. Saves the IMR of both PICs before initialization and restores them afterward.

Post-conditions: Master PIC delivers IRQ0–7 at `master_offset` through `master_offset + 7`. Slave PIC delivers IRQ8–15 at `slave_offset` through `slave_offset + 7`. Interrupt masks are identical to pre-call state.

Errors: None detectable — the PIC provides no acknowledgment. If the wrong values are passed (not multiples of 8), the PIC silently uses them, causing vectors to overlap unexpectedly.

4.8 `pic_send_eoi(uint8_t irq) → void`

Parameters:

- `irq`: IRQ number [0, 15]. Values > 15 are treated as master EOI only (safe fallback).

Behavior:

- If `irq >= 8`: `outb(PIC2_CMD, PIC_EOI)` — acknowledge slave PIC first.
- Always: `outb(PIC1_CMD, PIC_EOI)` — acknowledge master PIC.

Why slave first: The slave connects to the master via IRQ2. After the slave EOI, the master sees its IRQ2 input go inactive. The master EOI then releases the master's in-service bit. If you EOI the master first, the master re-enables delivery of future IRQs while the slave still has its ISR bit set — the slave cannot deliver another IRQ through the cascade until its own ISR bit is cleared, but the ordering is still correct. Slave-first is conventional and prevents any edge-case behavior on real 8259 hardware.

Errors: None. If `pic_send_eoi` is never called after an IRQ, the PIC permanently suppresses further IRQs on that line with no error message.

4.9 `pic_get_isr(uint8_t pic) → uint8_t`

Parameters:

- `pic` : `0` = master, `1` = slave.

Behavior:

1. `outb(pic == 0 ? PIC1_CMD : PIC2_CMD, 0x0B)` — OCW3: read ISR.
2. `return inb(pic == 0 ? PIC1_CMD : PIC2_CMD)`.

Returns: 8-bit ISR register value. Bit N set = IRQ N is currently being serviced (in-service register).

Purpose: Used by spurious IRQ handlers (IRQ7 and IRQ15) to verify whether the IRQ was genuine (ISR bit set) or spurious (ISR bit clear). See §5.3.

4.10 `pic_set_mask(uint8_t irq) → void` and `pic_clear_mask(uint8_t irq) → void`

Parameters:

- `irq` : IRQ number [0, 15].

pic_set_mask behavior: Reads the current IMR from the appropriate PIC data port. Sets bit `(irq % 8)`. Writes back. Effect: that IRQ line is masked (suppressed); the PIC will not forward it to the CPU until unmasked.

pic_clear_mask behavior: Reads IMR, clears bit `(irq % 8)`, writes back. Effect: IRQ line enabled.

Use case in this module: After `pic_remap()`, all IRQs are unmasked (masks restored from pre-remap state, which is typically all-enabled except for lines with no devices). `pic_set_mask` is not called during normal initialization but is declared for completeness and future use.

4.11 `pit_init(uint32_t hz) → void`

Parameters:

- `hz` : Desired timer frequency in Hz. Valid range: [1, 1193182]. Values outside this range are clamped: `hz < 1 → hz = 1` (minimum frequency, divisor = 65535); `hz > 1193182 → hz = 1193182` (maximum, divisor = 1). Pass `100` for this module.

Behavior:

1. Compute divisor: `uint32_t divisor = 1193182 / hz`. Clamp to [1, 65535].
2. `outb(0x43, 0x36)` — PIT command: channel 0, lo/hi, mode 2, binary.
3. `outb(0x40, (uint8_t)(divisor & 0xFF))` — low byte.
4. `outb(0x40, (uint8_t)((divisor >> 8) & 0xFF))` — high byte.
5. `irq_install_handler(0, timer_handler)` — register timer IRQ handler.

Post-conditions: PIT channel 0 fires IRQ0 at approximately `hz` Hz. `irq_handlers[0]` points to `timer_handler`.

Errors: None. If `irq_install_handler` was not available (impossible given module structure), the timer would fire but no handler would increment `pit_tick_count`.

4.12 `timer_handler(interrupt_frame_t *frame) → void`

Parameters: `frame` — ignored in this module (pointer set to `(void)frame`).

Behavior: `pit_tick_count++`. Returns immediately. EOI is sent by `irq_dispatcher` after this function returns.

Errors: None.

Note: In Milestone 4, this function is replaced (or extended) by the scheduler. The function is declared `static` in `pit.c` and registered via `irq_install_handler` — it can be overridden by calling `irq_install_handler(0, new_handler)` from mod-4's `sched_init()`.

4.13 `keyboard_init(void) → void`

Behavior:

1. Zeros the `kb_ring` struct: `head = 0`, `tail = 0`, all buffer bytes = 0.
2. Calls `irq_install_handler(1, keyboard_handler)`.

Post-conditions: `irq_handlers[1] == keyboard_handler`. Keyboard buffer is empty and ready.

4.14 `keyboard_getchar(void) → char`

Returns: Next ASCII character from the keyboard buffer, or `0` (null byte) if the buffer is empty.

Behavior:

```
char keyboard_getchar(void) {  
    if (kb_ring.head == kb_ring.tail) return 0; /* Empty */  
  
    char c = kb_ring.buf[kb_ring.tail];  
  
    kb_ring.tail++; /* uint8_t overflow wraps at 256 = KB_BUFFER_SIZE */  
  
    return c;  
}
```

Thread safety: Single-core only. On a multi-core system, atomic load/store or a lock would be required. For this module: acceptable.

Errors: Returns `0` if buffer empty. `0` is not a valid printable ASCII character; callers check for non-zero before acting on the return value.

4.15 `keyboard_handler(interrupt_frame_t *frame) → void`

Pre-conditions: Called from `irq_dispatcher` with interrupts disabled (interrupt gate). DS = kernel data segment.

Behavior: See full algorithm in §5.4.

Returns: Returns normally. `irq_dispatcher` sends EOI after return.

5. Algorithm Specification

5.1 ISR Assembly Stubs — Error Code Asymmetry

Problem: CPU exceptions 0–31 push an error code onto the stack before invoking the handler, but only for specific exceptions. All IRQ stubs and most exception stubs must maintain a uniform stack frame to allow a single `isr_common_stub` to handle all cases.

Solution: For exceptions without an error code, the stub pushes a dummy zero before pushing the vector number. For exceptions with an error code, the stub pushes only the vector number (the CPU already pushed the error code).

Which exceptions push an error code:

Push Error Code	Vectors
Yes	8 (double fault, always 0), 10 (invalid TSS), 11 (segment not present), 12 (stack fault), 13 (GPF), 14 (page fault), 17 (alignment check)
No	0, 1, 2, 3, 4, 5, 6, 7, 9, 15, 16, 18, 19, 20–31

NASM macro definitions:

```
; isr_stubs.asm - exception stubs for vectors 0-31                                NASM

%macro ISR_NOERR 1
    global isr%1
    isr%1:
        push dword 0          ; Dummy error code - maintains uniform frame layout
        push dword %1          ; Vector number
        jmp isr_common_stub
%endmacro

%macro ISR_ERR 1
    global isr%1
    isr%1:
        ; CPU has already pushed error code onto the stack
        push dword %1          ; Vector number (error code is below this on stack)
        jmp isr_common_stub
%endmacro
```

Stub instantiation (must match the table above exactly):

```
ISR_NOERR 0      ; Divide Error
ISR_NOERR 1      ; Debug / ICEBP
ISR_NOERR 2      ; Non-Maskable Interrupt
ISR_NOERR 3      ; Breakpoint
ISR_NOERR 4      ; INTO Overflow
ISR_NOERR 5      ; BOUND Range Exceeded
ISR_NOERR 6      ; Invalid Opcode (UD2)
ISR_NOERR 7      ; Device Not Available (FPU)
ISR_ERR 8        ; Double Fault (error code = 0, but CPU pushes it)
ISR_NOERR 9      ; Coprocessor Segment Overrun (legacy, 486-era)
ISR_ERR 10       ; Invalid TSS
ISR_ERR 11       ; Segment Not Present
ISR_ERR 12       ; Stack-Segment Fault
ISR_ERR 13       ; General Protection Fault
ISR_ERR 14       ; Page Fault
ISR_NOERR 15     ; Reserved (Intel-defined, do not use)
ISR_NOERR 16     ; x87 Floating-Point Error
ISR_ERR 17       ; Alignment Check
ISR_NOERR 18     ; Machine Check (abort; no recovery)
ISR_NOERR 19     ; SIMD Floating-Point Exception
ISR_NOERR 20     ; Virtualization Exception (VMX)
ISR_NOERR 21     ; Control Protection Exception (CET)
; Vectors 22-31: Intel reserved; ISR_NOERR for all
ISR_NOERR 22
ISR_NOERR 23
ISR_NOERR 24
ISR_NOERR 25
ISR_NOERR 26
ISR_NOERR 27
ISR_NOERR 28
ISR_NOERR 29
ISR_NOERR 30
ISR_NOERR 31
```

isr_common_stub — unified save, dispatch, restore:

```

; isr_common_stub
; Stack on entry (top to bottom = low to high address):
; [esp+0]  vector number      (pushed by stub)
; [esp+4]  error code or 0    (pushed by stub or CPU)
; [esp+8]  EIP                (CPU-pushed)
; [esp+12] CS                 (CPU-pushed)
; [esp+16] EFLAGS             (CPU-pushed)
; [esp+20] user_esp           (CPU-pushed ONLY if from ring 3)
; [esp+24] user_ss            (CPU-pushed ONLY if from ring 3)

isr_common_stub:
    pusha              ; Push EAX, ECX, EDX, EBX, ESP(pre-push), EBP, ESI, EDI
                        ; Each is 4 bytes; total 32 bytes pushed

    ; Save segment registers (push as 32-bit values for alignment)
    mov eax, gs
    push eax
    mov eax, fs
    push eax
    mov eax, es
    push eax
    mov eax, ds
    push eax

    ; Load kernel data segment into all data registers
    ; (In case we were interrupted in user mode with non-kernel selectors)
    mov eax, 0x10
    mov ds, eax
    mov es, eax
    mov fs, eax
    mov gs, eax

    ; Call the C exception handler, passing pointer to the frame
    push esp          ; Argument: pointer to interrupt_frame_t on the stack
    call exception_handler
    add esp, 4        ; Pop the argument (cdecl caller cleanup)

    ; Restore segment registers
    pop eax
    mov ds, eax
    pop eax
    mov es, eax
    pop eax
    mov fs, eax
    pop eax
    mov gs, eax

    popa              ; Restore EDI, ESI, EBP, (ESP ignored), EBX, EDX, ECX, EAX

    add esp, 8        ; Discard vector number and error code we pushed

    iret              ; Restore EIP, CS, EFLAGS (and ESP, SS if ring-3 transition)

```

Critical detail — `pusha` ordering: `pusha` pushes registers in this exact hardware-defined order (each 4 bytes): `EAX, ECX, EDX, EBX, ESP (pre-push), EBP, ESI, EDI`. The stack pointer advances 32 bytes downward. `popa` reverses this order and explicitly **ignores the `ESP` field** (it does not modify `ESP` from the saved value; `ESP` advances by 32 bytes during `popa` itself). The `interrupt_frame_t` struct field ordering must match this push sequence in reverse (since the top of the stack is at the lowest address and the struct is accessed via a pointer to the top).



5.2 IRQ Assembly Stubs

IRQ handlers follow the same save/restore pattern as exception stubs, but they go through `irq_dispatcher` (which sends EOI) rather than `exception_handler` (which halts).

```
; irq_stubs.asm - hardware IRQ stubs for vectors 32-47

%macro IRQ_STUB 1
    global irq%1
    irq%1:
        push dword 0          ; No error code for hardware IRQs
        push dword (%1 + 32)   ; Vector number (IRQ0 → 32, IRQ1 → 33, etc.)
        jmp irq_common_stub
%endmacro

IRQ_STUB 0    ; IRQ0: Timer → vector 32
IRQ_STUB 1    ; IRQ1: KB      → vector 33
IRQ_STUB 2    ; IRQ2: Cascade → vector 34
IRQ_STUB 3    ; IRQ3: COM2    → vector 35
IRQ_STUB 4    ; IRQ4: COM1    → vector 36
IRQ_STUB 5    ; IRQ5: LPT2    → vector 37
IRQ_STUB 6    ; IRQ6: Floppy  → vector 38
IRQ_STUB 7    ; IRQ7: LPT1/Spur → vector 39
IRQ_STUB 8    ; IRQ8: RTC     → vector 40
IRQ_STUB 9    ; IRQ9: Free    → vector 41
IRQ_STUB 10   ; IRQ10: Free   → vector 42
IRQ_STUB 11   ; IRQ11: Free   → vector 43
IRQ_STUB 12   ; IRQ12: Mouse  → vector 44
IRQ_STUB 13   ; IRQ13: FPU    → vector 45
IRQ_STUB 14   ; IRQ14: ATA    → vector 46
IRQ_STUB 15   ; IRQ15: ATA/Spur → vector 47
```

NASM

irq_common_stub :

```
irq_common_stub:
    pusha

    mov eax, gs
    push eax
    mov eax, fs
    push eax
    mov eax, es
    push eax
    mov eax, ds
    push eax

    mov eax, 0x10
    mov ds, eax
    mov es, eax
    mov fs, eax
    mov gs, eax

    push esp           ; Argument: pointer to interrupt_frame_t
    call irq_dispatcher
    add esp, 4

    pop eax
    mov ds, eax
    pop eax
    mov es, eax
    pop eax
    mov fs, eax
    pop eax
    mov gs, eax

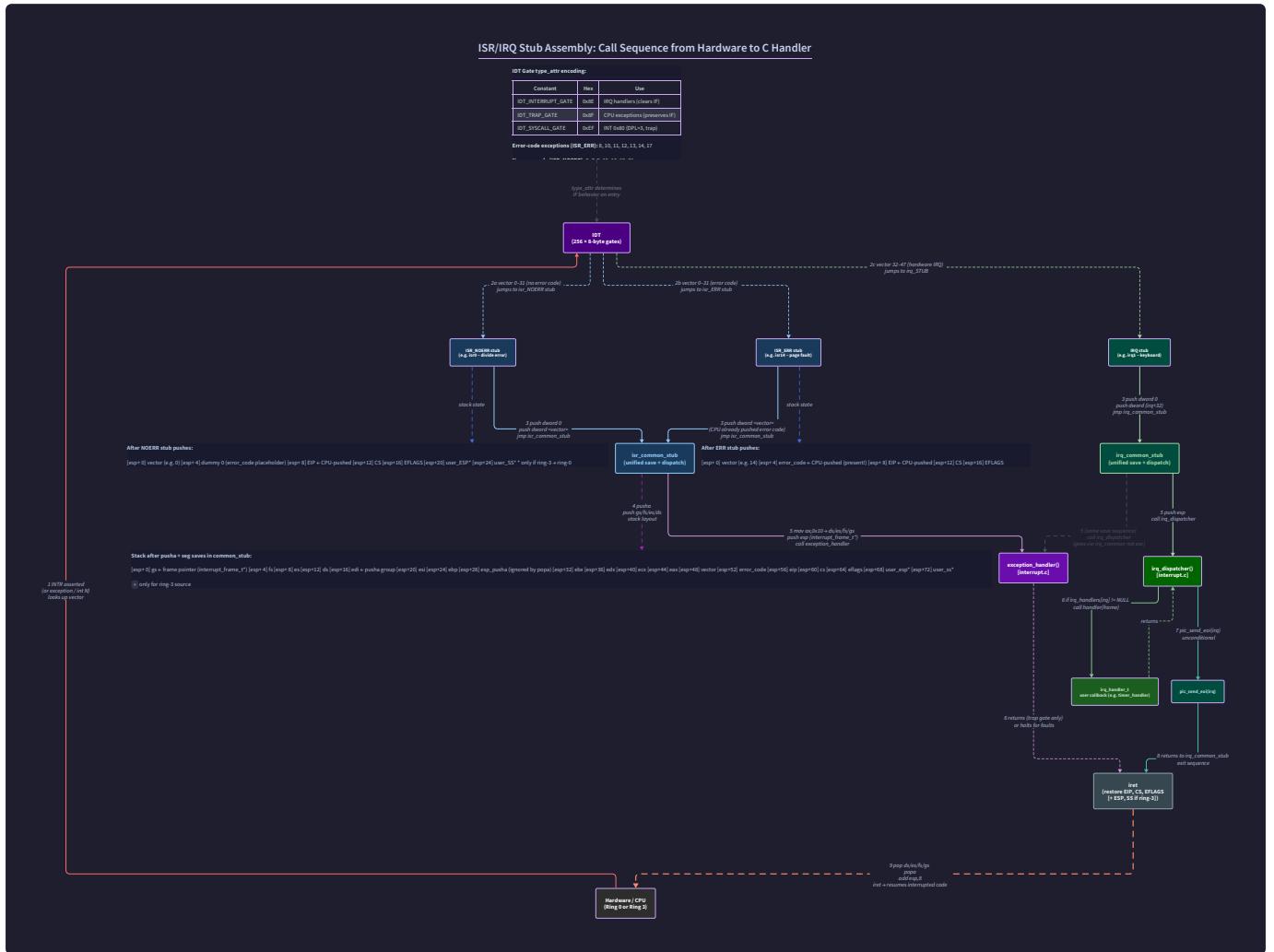
    popa
    add esp, 8         ; Discard dummy error code and vector number
    iret
```

NASM

`default_isr` — handler for unregistered vectors:

```
global default_isr
default_isr:
    push dword 0
    push dword 0xFF      ; Sentinel vector value (0xFF = "unknown")
    jmp isr_common_stub  ; exception_handler will print "unknown vector 255" and halt
```

NASM



5.3 PIC Remapping — ICW Sequence

Input: CPU in protected mode, interrupts disabled.

Step-by-step:

```

Step 1: Save current interrupt masks (IMR) for both PICs
uint8_t master_mask = inb(0x21);
uint8_t slave_mask = inb(0xA1);

Step 2: Send ICW1 to both PICs simultaneously
outb(0x20, 0x11); io_wait(); /* Master: init, ICW4 needed */
outb(0xA0, 0x11); io_wait(); /* Slave: init, ICW4 needed */

Step 3: Send ICW2 – vector offsets
outb(0x21, master_offset); io_wait(); /* Master: IRQ0 → master_offset */
outb(0xA1, slave_offset); io_wait(); /* Slave: IRQ8 → slave_offset */

Step 4: Send ICW3 – cascade configuration
outb(0x21, 0x04); io_wait(); /* Master: slave on IRQ2 (bitmask bit2 = 0x04) */
outb(0xA1, 0x02); io_wait(); /* Slave: cascade identity = 2 (numeric) */

Step 5: Send ICW4 – 8086 mode
outb(0x21, 0x01); io_wait();
outb(0xA1, 0x01); io_wait();

Step 6: Restore saved masks
outb(0x21, master_mask);
outb(0xA1, slave_mask);

```

Invariant after Step 6: PIC is fully operational with the new vector mapping. The saved masks are identical to the pre-remap masks — no IRQ line is newly masked or unmasked by the remapping operation.

Why `io_wait()` between every write: The 8259 is an ISA-era device. Its internal state machine advances through the ICW sequence by receiving successive writes to the data port, but it requires settling time (typically 1–4µs) between writes. On real hardware, skipping `io_wait()` causes initialization failures on some chipsets. On QEMU, `io_wait()` is a no-op functionally but is retained for portability.

PIT 8254 Channel 0: Divisor Calculation and 100Hz Configuration

PIT I/O Port Map			
Port	W	Channel 1 Data	PC speaker
Dir	Channel 0 Data	DRAM refresh – DO NOT USE	0x43
Register	Lo byte, then Hi byte	0x42	W
Notes	0x41	W	Mode/Command
0x40	W	Channel 2 Data	Write only; read back via latch

0x43 write configures

Command Byte 0x36 Decoded		
Bits	[5:4]	[0]
Field	Access Mode	BCD/Binary
Value / Meaning	11 = Lo/Hi byte (2-byte write)	0 = Binary counting
[7:6]	[3:1]	Full byte
Channel Select	Operating Mode	0x36 = b00_11_011_0
00 = Channel 0 (timer, IRQ0)	011 = Mode 2 (rate generator)	CH0, Lo/Hi, Mode 2, Binary

determines

Clock Source and Divisor Calculation

PIT Input Clock 1,193,182 Hz (≈ 1.193 MHz)

Origin: NTSC color burst = 12 = $3,579,545 / 3 = 1,193,182$

\div divisor

IRQ0 Frequency = Clock \div Divisor

For 100 Hz target: Divisor = $1,193,182 \div 100 = 11,931.82 \rightarrow$ round to **11,932**

Actual freq = $1,193,182 \div 11,932 = 99.9987$ Hz Error < 0.002%

encode

11,932 = 0x2EDC

Low byte (0xDC = 220): outb(0x40, 0xDC) ← FIRST

High byte (0xE = 46): outb(0x40, 0x2E) ← SECOND

Order MATTERS: lo before hi (mode 11 = lo/hi access)

drives

Initialization Sequence (4 port writes)

outb(0x43, 0x36)

Port: 0x43 (Command) Value: 0x36 Sets: Ch0, lo/hi, Mode 2, binary Must be FIRST

then

outb(0x40, 0xDC)

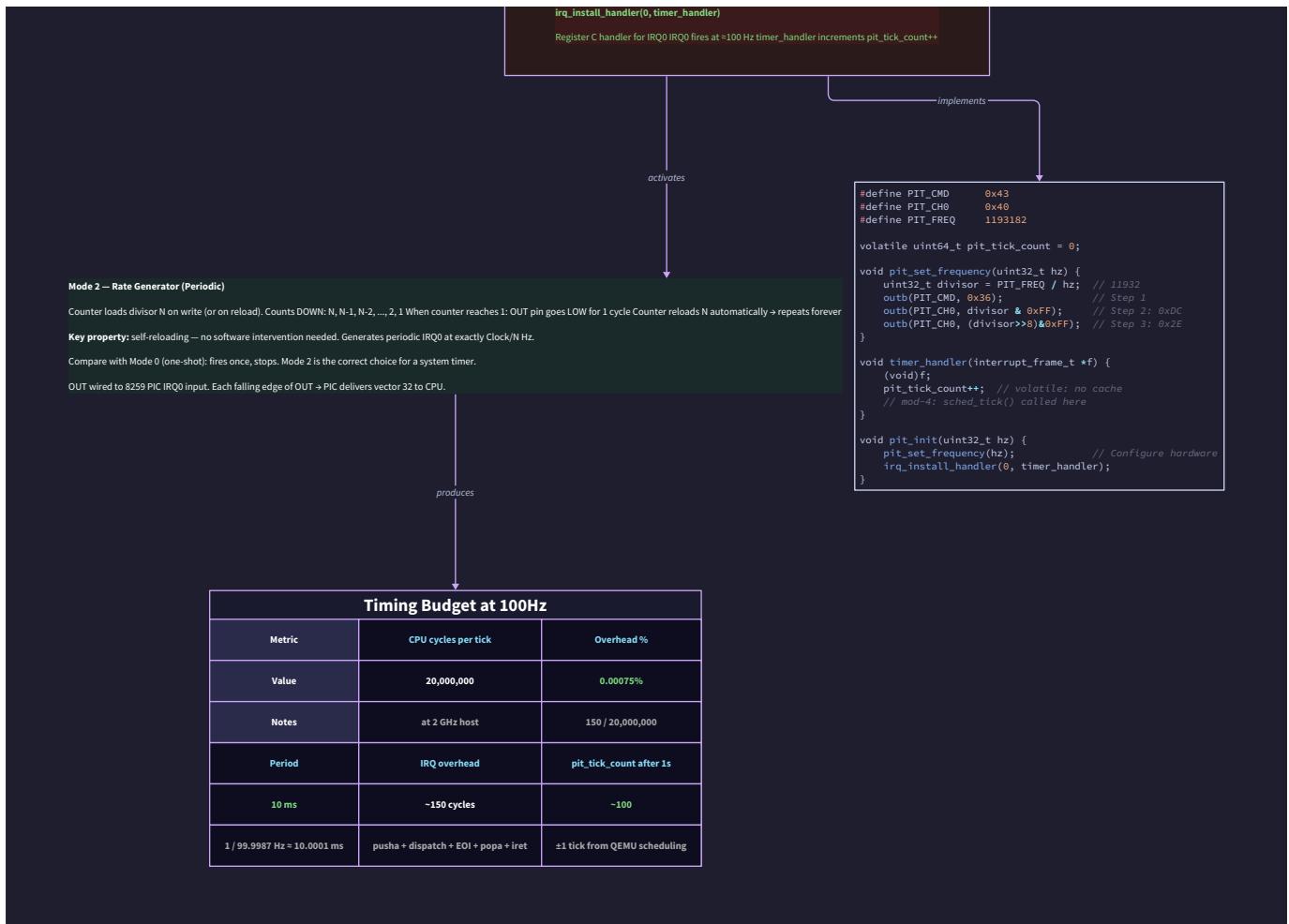
Port: 0x40 (Ch0 data) Value: 0xDC (low byte of 11932) Divisor bits [7:0]

then

outb(0x40, 0x2E)

Port: 0x40 (Ch0 data) Value: 0x2E (high byte of 11932) Divisor bits [15:8] Loads divisor → counter starts

then



5.4 PS/2 Keyboard Handler — Full Algorithm

Called from: `irq_dispatcher` with `frame->vector == 33` (IRQ1). Interrupts disabled (interrupt gate). `kb_ring` is the single global circular buffer.

State (static variables in `keyboard.c`):

Variable	Type	Purpose
<code>kb_ring</code>	<code>kb_ring_t</code>	Circular input buffer
<code>kb_shift_held</code>	<code>uint8_t</code>	Non-zero if either Shift key is currently pressed
<code>kb_expect_extended</code>	<code>uint8_t</code>	Non-zero if last byte was <code>0xE0</code> (extended key prefix)

Step-by-step:

```

Step 1: Read scancode from PS/2 data port
uint8_t scancode = inb(0x60);

Step 2: Handle extended key prefix (0xE0)
if (scancode == 0xE0) {
    kb_expect_extended = 1;
    return; /* Wait for the actual code in the next interrupt */
}
if (kb_expect_extended) {
    kb_expect_extended = 0;
    return; /* Discard the extended key code entirely (not supported) */
}

Step 3: Classify make vs break code
int is_break = (scancode & 0x80) != 0;
uint8_t make_code = scancode & 0x7F; /* Strip bit 7 to get make code */

Step 4: Handle modifier keys (Shift, Ctrl, Alt detection)
if (make_code == 0x2A || make_code == 0x36) { /* Left or right Shift */
    kb_shift_held = is_break ? 0 : 1;
    return; /* Modifier key: no character to buffer */
}
if (make_code == 0x1D || make_code == 0x38) { /* Ctrl or Alt */
    return; /* Detected but not tracked beyond make/break */
}
if (make_code == 0x3A) { /* Caps Lock */
    return; /* Not tracked in this implementation */
}

Step 5: Filter break codes for non-modifier keys
if (is_break) {
    return; /* Key release for non-modifier: discard */
}

Step 6: Translate make code to ASCII
if (make_code >= 128) return; /* Out of table range: discard */
char c = kb_shift_held
    ? scancode_ascii_upper[make_code]
    : scancode_ascii_lower[make_code];
if (c == 0) return; /* Unmapped key (function keys, arrow keys with 0 in table): discard */

Step 7: Write to circular buffer (drop on overflow)
uint8_t next_head = kb_ring.head + 1; /* uint8_t: wraps at 256 automatically */
if (next_head == kb_ring.tail) {
    return; /* Buffer full: silently discard the character */
}
kb_ring.buf[kb_ring.head] = c;
kb_ring.head = next_head;

```

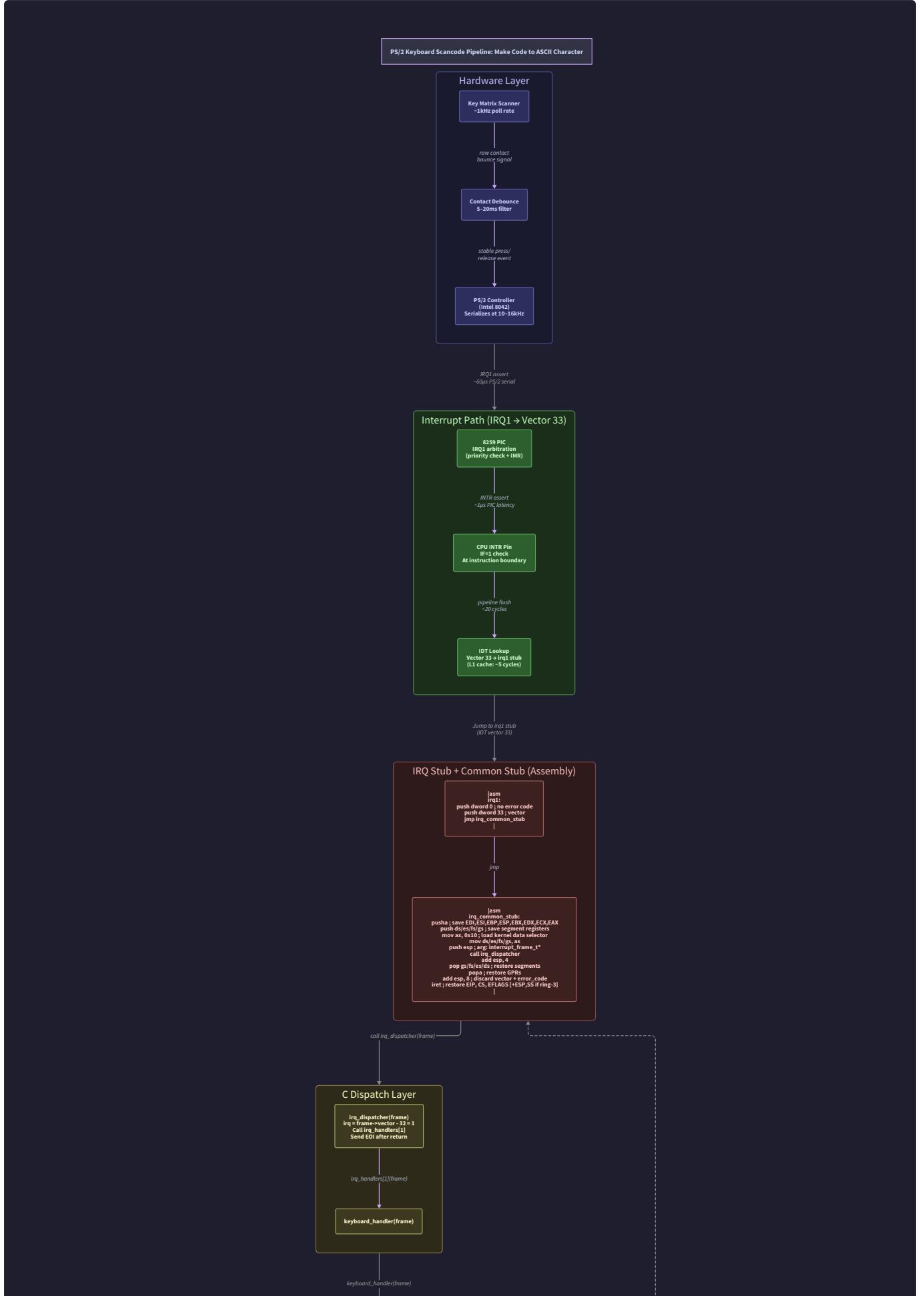
Invariants after each step:

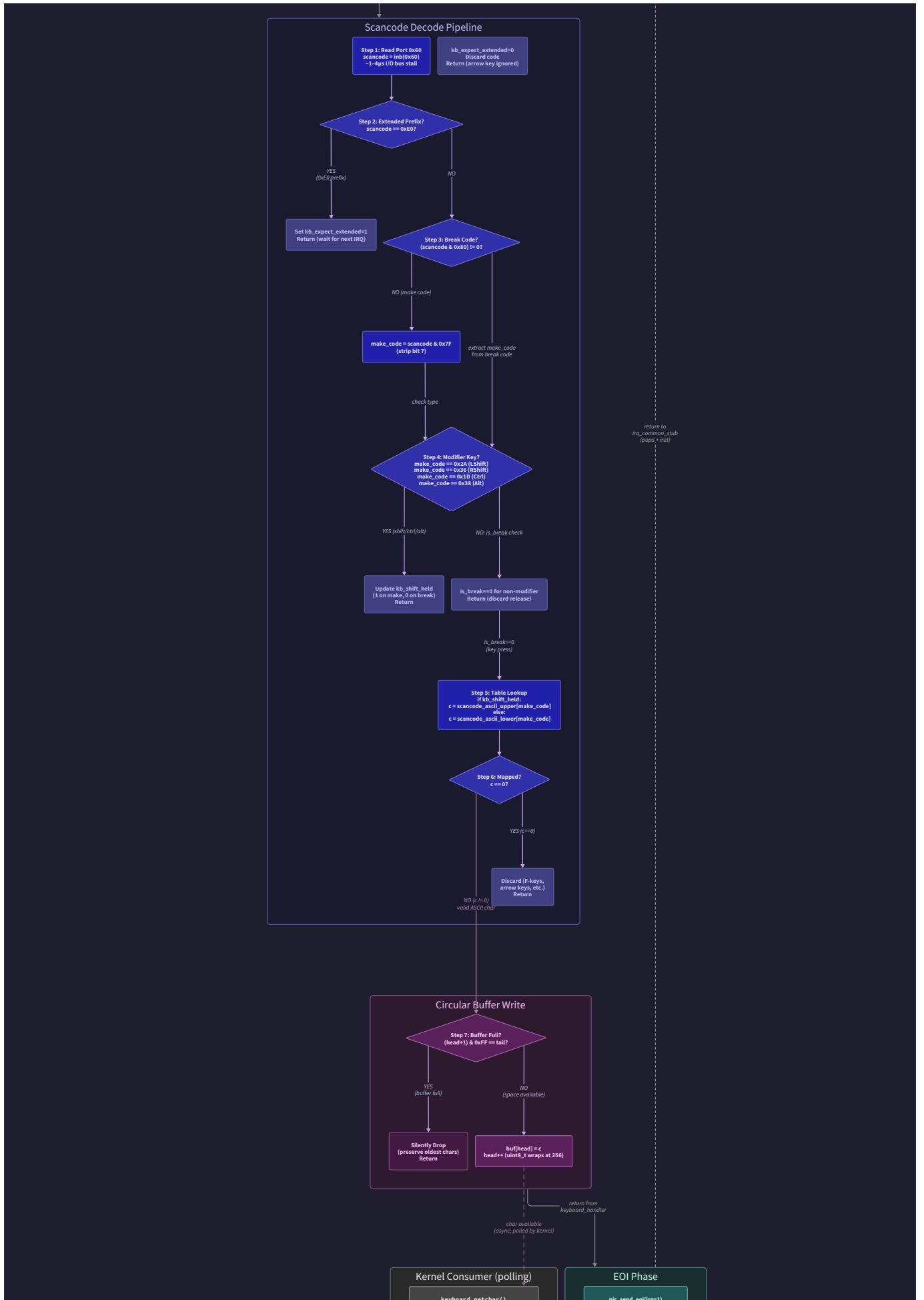
- After Step 1: `scancode` holds the raw byte from port `0x60`.
- After Step 3: `is_break` is 1 iff bit 7 of `scancode` is set. `make_code` is the corresponding make code regardless of break/make.
- After Step 6: `c` is a valid, printable (or control) ASCII character. `c != 0`.
- After Step 7: If buffer was not full, `kb_ring.buf[(head - 1) & 0xFF] == c` and `head` has advanced by 1.

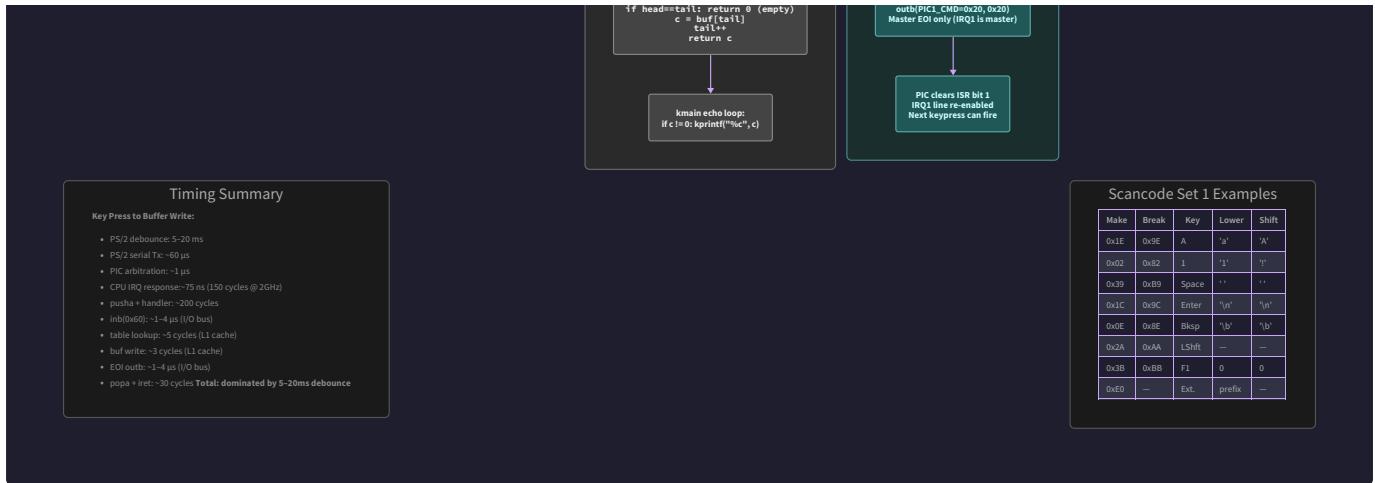
Edge cases:

- `scancode == 0x00`: Keyboard controller error (key detection error). `make_code = 0`, `scancode_ascii_lower[0] = 0` → discarded at Step 6.

- `scancode == 0xFF` : Keyboard error. Same path as above via Step 6.
- Rapid key repeat (key held down): PS/2 controller automatically re-sends the make code at the keyboard's typematic rate (default ~10 repeats/second). Each repeat fires IRQ1 and goes through the full handler. Since `is_break = 0`, the character is buffered on each repeat. This is correct terminal behavior.







5.5 Spurious IRQ Handling — IRQ7 and IRQ15

When the PIC reports IRQ7 or IRQ15 due to electrical noise (a spurious interrupt), the ISR register bit for that IRQ is NOT set. The handler must check the ISR before deciding whether to act and whether to send EOI.

Spurious IRQ7 handler logic (installed via `irq_install_handler(7, spurious7_handler)`):

```

static void spurious7_handler(interrupt_frame_t *frame) {
    (void)frame;

    uint8_t isr = pic_get_isr(0); /* Read master PIC ISR */

    if (isr & (1 << 7)) {
        /* Bit 7 set: genuine IRQ7 (LPT1 or similar). Send EOI normally. */
        /* irq_dispatcher will send EOI after we return. */

        return;
    }

    /* Bit 7 clear: spurious. Do NOT send EOI to master PIC.

    Override the EOI that irq_dispatcher would normally send:

    We set a per-invocation flag that irq_dispatcher checks.

    Simpler: because irq_dispatcher ALWAYS sends EOI, and for spurious IRQ7

    we must NOT send master EOI, we need a mechanism.

    Implementation choice: install spurious7_handler to also suppress the EOI.

    The cleanest approach is to send the EOI from within the handler if genuine,
    and return a suppress flag to irq_dispatcher. Since irq_handler_t returns void,
    we use a global suppress flag instead: */

    pic_suppress_eoi = 1; /* irq_dispatcher checks this before pic_send_eoi */

}
  
```

`irq_dispatcher` with EOI suppression:

```
/* Global flag – set by spurious handlers, cleared by irq_dispatcher */

static volatile uint8_t pic_suppress_eoi = 0;

void irq_dispatcher(interrupt_frame_t *frame) {

    uint8_t irq = (uint8_t)(frame->vector - 32);

    pic_suppress_eoi = 0;

    if (irq_handlers[irq]) {

        irq_handlers[irq](frame);

    }

    if (!pic_suppress_eoi) {

        pic_send_eoi(irq);

    }

    /* If pic_suppress_eoi was set: no EOI sent. The spurious handler already
       decided the IRQ was spurious and suppression is appropriate. */

}

}
```

Spurious IRQ15 (slave) logic: Spurious IRQ15 requires an EOI to the **master** (because the master did generate an IRQ2 cascade signal) but NOT to the slave (because the slave's ISR bit is clear). The `spurious15_handler` detects this case and manually sends master EOI while setting `pic_suppress_eoi`:

```
static void spurious15_handler(interrupt_frame_t *frame) {

    (void)frame;

    uint8_t isr = pic_get_isr(1);    /* Read slave PIC ISR */

    if (isr & (1 << 7)) {

        return;    /* Genuine IRQ15; irq_dispatcher sends both EOIs normally */

    }

    /* Spurious IRQ15: send only master EOI, suppress slave EOI */

    outb(PIC1_CMD, PIC_EOI);    /* Master EOI only */

    pic_suppress_eoi = 1;        /* Suppress irq_dispatcher's normal EOI */

}
```

Register both in `idt_setup_all()`:

```
irq_install_handler(7,  spurious7_handler);  
irq_install_handler(15, spurious15_handler);
```

C

5.6 Default ISR for Unregistered Vectors

Vectors 48–127 and 129–255 point to `default_isr`. When invoked:

```
/* exception_handler for vector 0xFF (sentinel): */  
  
/* The default_isr stub pushes vector=0xFF; exception_handler prints: */  
  
kprintf("\n==== UNEXPECTED INTERRUPT ===\n");  
  
kprintf("Vector: 0x%02x - no handler registered.\n", frame->vector);  
  
kprintf("EIP: 0x%08x CS: 0x%04x\n", frame->eip, frame->cs);  
  
kprintf("System halted.\n");  
  
for (;;) { __asm__ volatile ("cli; hlt"); }
```

C

The `default_isr` stub uses vector `0xFF` as the sentinel. When `exception_handler` sees `vector == 0xFF`, it prints the "unexpected interrupt" message instead of indexing `exception_names[]` (which only has entries 0–31).

5.7 idt_setup_all() — Complete Initialization Sequence

```
void idt_setup_all(void) {  
  
    /* Step 1: Install all 256 entries with default_isr */  
  
    for (int i = 0; i < 256; i++) {  
  
        idt_set_gate((uint8_t)i, (uint32_t)default_isr,  
                    0x08, IDT_INTERRUPT_GATE);  
  
    }  
  
    /* Step 2: Override vectors 0-31 with exception stubs (trap gates) */  
  
    idt_set_gate(0, (uint32_t)isr0, 0x08, IDT_TRAP_GATE);  
    idt_set_gate(1, (uint32_t)isr1, 0x08, IDT_TRAP_GATE);  
    idt_set_gate(2, (uint32_t)isr2, 0x08, IDT_TRAP_GATE);  
    idt_set_gate(3, (uint32_t)isr3, 0x08, IDT_TRAP_GATE);  
    idt_set_gate(4, (uint32_t)isr4, 0x08, IDT_TRAP_GATE);  
    idt_set_gate(5, (uint32_t)isr5, 0x08, IDT_TRAP_GATE);  
    idt_set_gate(6, (uint32_t)isr6, 0x08, IDT_TRAP_GATE);  
    idt_set_gate(7, (uint32_t)isr7, 0x08, IDT_TRAP_GATE);  
    idt_set_gate(8, (uint32_t)isr8, 0x08, IDT_TRAP_GATE);  
    idt_set_gate(9, (uint32_t)isr9, 0x08, IDT_TRAP_GATE);  
    idt_set_gate(10, (uint32_t)isr10, 0x08, IDT_TRAP_GATE);  
    idt_set_gate(11, (uint32_t)isr11, 0x08, IDT_TRAP_GATE);  
    idt_set_gate(12, (uint32_t)isr12, 0x08, IDT_TRAP_GATE);  
    idt_set_gate(13, (uint32_t)isr13, 0x08, IDT_TRAP_GATE);  
    idt_set_gate(14, (uint32_t)isr14, 0x08, IDT_TRAP_GATE);  
    idt_set_gate(15, (uint32_t)isr15, 0x08, IDT_TRAP_GATE);  
    idt_set_gate(16, (uint32_t)isr16, 0x08, IDT_TRAP_GATE);  
    idt_set_gate(17, (uint32_t)isr17, 0x08, IDT_TRAP_GATE);  
    idt_set_gate(18, (uint32_t)isr18, 0x08, IDT_TRAP_GATE);  
    idt_set_gate(19, (uint32_t)isr19, 0x08, IDT_TRAP_GATE);  
    idt_set_gate(20, (uint32_t)isr20, 0x08, IDT_TRAP_GATE);  
    idt_set_gate(21, (uint32_t)isr21, 0x08, IDT_TRAP_GATE);  
    idt_set_gate(22, (uint32_t)isr22, 0x08, IDT_TRAP_GATE);  
    idt_set_gate(23, (uint32_t)isr23, 0x08, IDT_TRAP_GATE);  
}
```

```

idt_set_gate(24, (uint32_t)isr24, 0x08, IDT_TRAP_GATE);

idt_set_gate(25, (uint32_t)isr25, 0x08, IDT_TRAP_GATE);

idt_set_gate(26, (uint32_t)isr26, 0x08, IDT_TRAP_GATE);

idt_set_gate(27, (uint32_t)isr27, 0x08, IDT_TRAP_GATE);

idt_set_gate(28, (uint32_t)isr28, 0x08, IDT_TRAP_GATE);

idt_set_gate(29, (uint32_t)isr29, 0x08, IDT_TRAP_GATE);

idt_set_gate(30, (uint32_t)isr30, 0x08, IDT_TRAP_GATE);

idt_set_gate(31, (uint32_t)isr31, 0x08, IDT_TRAP_GATE);

/* Step 3: Override vectors 32-47 with IRQ stubs (interrupt gates) */

idt_set_gate(32, (uint32_t)irq0, 0x08, IDT_INTERRUPT_GATE);

idt_set_gate(33, (uint32_t)irq1, 0x08, IDT_INTERRUPT_GATE);

idt_set_gate(34, (uint32_t)irq2, 0x08, IDT_INTERRUPT_GATE);

idt_set_gate(35, (uint32_t)irq3, 0x08, IDT_INTERRUPT_GATE);

idt_set_gate(36, (uint32_t)irq4, 0x08, IDT_INTERRUPT_GATE);

idt_set_gate(37, (uint32_t)irq5, 0x08, IDT_INTERRUPT_GATE);

idt_set_gate(38, (uint32_t)irq6, 0x08, IDT_INTERRUPT_GATE);

idt_set_gate(39, (uint32_t)irq7, 0x08, IDT_INTERRUPT_GATE);

idt_set_gate(40, (uint32_t)irq8, 0x08, IDT_INTERRUPT_GATE);

idt_set_gate(41, (uint32_t)irq9, 0x08, IDT_INTERRUPT_GATE);

idt_set_gate(42, (uint32_t)irq10, 0x08, IDT_INTERRUPT_GATE);

idt_set_gate(43, (uint32_t)irq11, 0x08, IDT_INTERRUPT_GATE);

idt_set_gate(44, (uint32_t)irq12, 0x08, IDT_INTERRUPT_GATE);

idt_set_gate(45, (uint32_t)irq13, 0x08, IDT_INTERRUPT_GATE);

idt_set_gate(46, (uint32_t)irq14, 0x08, IDT_INTERRUPT_GATE);

idt_set_gate(47, (uint32_t)irq15, 0x08, IDT_INTERRUPT_GATE);

/* Step 4: Install spurious IRQ handlers */

irq_install_handler(7, spurious7_handler);

irq_install_handler(15, spurious15_handler);

/* Step 5: Load IDTR */

idt_install();

}

```


6. Error Handling Matrix

Error	Detected By	Recovery	User-Visible?
Triple fault: PIC not remapped before <code>sti</code>	<code>pic_remap()</code> not called; IRQ0 fires at vector 8 (double fault handler = garbage or exception_handler which halts, causing another fault)	No recovery — QEMU reboots. Prevention: always call <code>pic_remap()</code> before <code>sti</code> . Detected by: <code>-d int</code> shows <code>v=08</code> (double fault) immediately after timer tick arrives.	QEMU reboot
Triple fault: IDT not loaded before <code>sti</code>	<code>idt_install()</code> not called; IDTR points to BIOS IDT (real-mode IVT at 0x0)	No recovery — triple fault. Prevention: <code>idt_setup_all()</code> before <code>pic_remap()</code> before <code>sti</code> .	QEMU reboot
Silent hang: EOI not sent	Verified by observing PIT tick count stops incrementing	No automated recovery. If detected: ensure <code>irq_dispatcher</code> calls <code>pic_send_eoi()</code> unconditionally. Check <code>pic_suppress_eoi</code> flag not stuck at 1.	System appears frozen; <code>pit_tick_count</code> static
Stack misalignment: NOERR/ERR macro applied to wrong exception	<code>iret</code> restores wrong EIP (off by 4 bytes because error code was/wasn't consumed)	Triple fault when <code>iret</code> returns to garbage address. Detected by: <code>-d int</code> shows unexpected exception vector after the initial exception.	QEMU reboot
pusha/popa count mismatch in stubs	Interrupted code resumes with wrong register values; behavior unpredictable	No crash guaranteed; may execute normally with wrong results for many instructions. Detected by: comparing <code>pusha</code> count (8 pushes × 4 bytes = 32 bytes) with <code>popa</code> placement; verify <code>add esp, 8</code> is present after <code>popa</code> .	Silent corruption; may never be detected without register inspection
Segment registers not restored in stubs	After returning from ring-0 IRQ handler, DS/ES/FS/GS wrong if they were modified inside handler	GPF on first data access with wrong selector. Detected by: immediate GPF after first IRQ returns.	GPF (exception 13) printed and halted
Keyboard break codes not filtered	Every key press generates two characters in the buffer	Doubled characters visible in echo demo. Detected by: typing 'A' produces 'AA'.	Doubled key output
0xE0 extended key prefix not handled	Arrow key press: <code>0xE0</code> consumed, next byte (e.g., <code>0x4B</code> for left arrow) interpreted as a printable key	Spurious characters in buffer when arrow/numpad keys pressed. Detected by: pressing arrow keys inserts unexpected characters.	Wrong characters on arrow keys
Circular buffer overflow	Buffer full; new characters silently dropped	Oldest characters preserved; newest discarded. No crash. Detection: fast typing fills buffer; subsequent characters lost until <code>keyboard_getchar()</code> drains it.	Silent character loss
Circular buffer head/tail race	Single-core only: IRQ handler cannot interrupt <code>keyboard_getchar()</code> because IRQ runs with IF=0, and <code>keyboard_getchar()</code> runs with	N/A — not a race condition on single-core	N/A

Error	Detected By	Recovery	User-Visible?
	IF=1 — the IRQ is the higher-priority context. No race exists on single-core.		
Wrong gate type for exception (interrupt gate instead of trap gate)	IF is cleared on exception entry; blocking <code>kprintf</code> + serial busy-wait would stall if any other IRQ was needed during the wait	Functional issue only: <code>kprintf</code> will complete via busy-wait without needing IRQs. Not a crash.	Subtle: no observable difference in this module
Wrong gate DPL for system call vector	<code>IDT_INTERRUPT_GATE</code> (DPL=0) used for vector 0x80 instead of <code>IDT_SYSCALL_GATE</code> (DPL=3)	User <code>int 0x80</code> causes GPF (vector 13) immediately; syscall never works	GPF when user process calls <code>int 0x80</code>
<code>pic_get_isr()</code> not called for spurious IRQ detection	Spurious IRQ7: EOI sent to master when no real interrupt occurred; master's priority may be disturbed	Not fatal. Some hardware generates garbage behavior. QEMU: typically harmless but non-compliant.	No visible error; may cause subtle IRQ priority issues
PIT divisor written in wrong byte order (high before low)	Timer fires at wrong frequency (approximately 100× off)	No crash; <code>pit_tick_count</code> increments at wrong rate. Detected by: measuring wall time vs tick count.	Incorrect timing (all time-based operations wrong)
PIT command byte wrong mode	Mode 0 (one-shot) instead of mode 2 (rate generator): timer fires once and stops	No IRQs after first tick; system timer freezes at <code>pit_tick_count == 1</code> .	System appears frozen
<code>volatile</code> missing from <code>pit_tick_count</code>	Compiler caches <code>pit_tick_count</code> in a register; kernel code polling the variable never sees updates	No crash; polling loops spin forever. Detected by: busy-wait on <code>pit_tick_count</code> never exits.	Hangs in any timing loop
Double fault handler (exception 8) not robust	Exception handler calls <code>kprintf</code> ; if <code>kprintf</code> triggers another exception (e.g., page fault from bad string pointer)	Nested exception causes third fault → triple fault → reboot	QEMU reboot
IDTR limit wrong	<code>sizeof(idt) - 1</code> computed incorrectly (e.g., using <code>sizeof(idt_gate_t)</code> instead of <code>sizeof(idt)</code>)	CPU ignores IDT entries beyond the declared limit; invoking those vectors causes GPF	GPF on any interrupt above the truncated limit

7. Implementation Sequence with Checkpoints

Phase 1 — IDT Data Structures and `lidt` (2–3 hours)

Create `include/idt.h` with `idt_gate_t`, `idtr_t`, the three `IDT_*` constants, and declarations for `idt_set_gate()`, `idt_install()`, `idt_setup_all()`.

Create `interrupts/idt.c`: declare `static idt_gate_t idt[256]` and `static idtr_t idtr` (both in `.bss`, zeroed by Milestone 1's BSS-zero code). Implement `idt_set_gate()` and `idt_install()`. Add a compile-time assertion:

```
_Static_assert(sizeof(idt_gate_t) == 8, "IDT gate descriptor must be 8 bytes");
_Static_assert(sizeof(idtr_t) == 6, "IDTR must be 6 bytes");
```

Do **not** call `idt_install()` yet. Do **not** implement `idt_setup_all()` yet (it depends on stubs not yet written).

Checkpoint 1: Compile `idt.c` with `i686-elf-gcc -m32 -ffreestanding -nostdinc -c idt.c -o idt.o`. Should compile with zero warnings and zero errors. Verify struct sizes via the static asserts compiling cleanly. Run `objdump -t idt.o | grep idt` to confirm `idt` symbol is in `.bss` (size 2048 = 256 × 8).

Phase 2 — ISR Assembly Stubs, `isr_common_stub` (3–4 hours)

Create `interrupts/isr_stubs.asm`. Implement both `ISR_NOERR` and `ISR_ERR` macros, instantiate stubs for vectors 0–31, implement `isr_common_stub`, implement `default_isr`.

Create `include/interrupt.h` with `interrupt_frame_t` and `FRAME_FROM_RING3` macro.

Create stub `interrupts/interrupt.c` with a minimal `exception_handler()` that just calls `kprintf("Exception %d\n", frame->vector)` and halts — full implementation comes in Phase 3.

Add `extern void isr0(void);` through `extern void isr31(void);` and `extern void default_isr(void);` declarations to `idt.h`.

Verify stack frame layout: Add a compile-time offset check in `interrupt.c`:

```
#include <stddef.h>

_Static_assert(offsetof(interrupt_frame_t, gs) == 0, "gs offset wrong");
_Static_assert(offsetof(interrupt_frame_t, ds) == 12, "ds offset wrong");
_Static_assert(offsetof(interrupt_frame_t, edi) == 16, "edi offset wrong");
_Static_assert(offsetof(interrupt_frame_t, eax) == 44, "eax offset wrong");
_Static_assert(offsetof(interrupt_frame_t, vector) == 48, "vector offset wrong");
_Static_assert(offsetof(interrupt_frame_t, error_code) == 52, "error_code offset wrong");
_Static_assert(offsetof(interrupt_frame_t, eip) == 56, "eip offset wrong");
_Static_assert(offsetof(interrupt_frame_t, cs) == 60, "cs offset wrong");
_Static_assert(offsetof(interrupt_frame_t, eflags) == 64, "eflags offset wrong");
```

Implement minimal `idt_setup_all()` in `idt.c`: just installs all 256 vectors with `default_isr` and the 32 exception vectors with their real stubs, then calls `idt_install()`. Do not add IRQ vectors yet.

Call `idt_setup_all()` from `kmain()` (before the idle loop, with interrupts still disabled). Do not call `sti` yet.

Checkpoint 2: Boot the kernel. It should function identically to Milestone 1 (no interrupts, no changes visible). Now add this test to `kmain()`:

```
/* Trigger a divide-by-zero to test exception handler */

volatile int zero = 0;

volatile int x = 1 / zero; /* Should invoke isr0 → exception_handler */

(void)x;
```

Expected result: VGA and serial show `"Exception 0"` (or the full diagnostic if you implemented it already) and then the system halts. If QEMU reboots instead, the IDT was not loaded or `isr0` has an assembly error. Use `qemu-system-i386 ... -d int 2>int.log` and verify `v=00 e=0000` appears in the log.

Phase 3 — Full Exception Handler (2–3 hours)

Complete `interrupts/interrupt.c` with the full `exception_handler()` implementation: `exception_names[]` table (32 entries, "Reserved" for gaps), full register dump via `kprintf`, CR2 read for page fault (vector 14), ring-3 detection. Add static array `irq_handlers[16]` and implement `irq_install_handler()` and `irq_dispatcher()` with the `pic_suppress_eoi` mechanism.

Checkpoint 3: Test exception 13 (GPF) by attempting to load a null selector:

```
/* In kmain, after idt_setup_all() but before sti: */

__asm__ volatile ("mov $0, %ax; mov %ax, %ds"); /* GPF: null descriptor */
```

Expected output includes:

```
==== CPU EXCEPTION ===
Exception: General Protection Fault (vector 13)
Error code: 0x00000000
EIP=0x001XXXXX CS=0x0008 EFLAGS=...
EAX=... EBX=... ...
System halted.
```

Remove the GPF test before proceeding.

Phase 4 — 8259 PIC Remapping (2–3 hours)

Create `include/pic.h` and `interrupts/pic.c`. Implement `pic_remap()`, `pic_send_eoi()`, `pic_get_isr()`, `pic_set_mask()`, `pic_clear_mask()`.

Do not call `sti` yet. Add `pic_remap(0x20, 0x28)` to `kmain()` after `idt_setup_all()`.

Checkpoint 4: Verify the PIC remapping happened without a triple fault. With `qemu-system-i386 ... -d int 2>int.log`, boot the kernel. The log should not show any interrupt vectors 0–15 firing after the PIC remap. If you temporarily enable `sti` for one instruction then immediately `cli`, you should see `v=20` (IRQ0 at vector 32) appear in the log — not `v=08` (double fault). This confirms the PIC remap succeeded.

```
/* Temporary test – remove after verifying */

__asm__ volatile ("sti; nop; cli");
```

Check `int.log` for `v=20 e=0000 i=1` (`i=1` means IRQ). If you see `v=08`, the remap failed.

Phase 5 — IRQ Assembly Stubs and Dispatcher (2–3 hours)

Create `interrupts/irq_stubs.asm`. Implement `IRQ_STUB` macro, instantiate for IRQ0–15, implement `irq_common_stub`.

Add `extern void irq0(void);` through `extern void irq15(void);` to `irq.h`. Install IRQ vectors 32–47 in `idt_setup_all()`.

Add `irq_install_handler(7, spurious7_handler)` and `irq_install_handler(15, spurious15_handler)` calls at the bottom of `idt_setup_all()`.

Checkpoint 5: Add a test IRQ handler and temporarily enable interrupts:

```

static void test_handler(interrupt_frame_t *f) {
    (void)f;
    kprintf("IRQ0 fired!\n");
}
irq_install_handler(0, test_handler);
__asm__ volatile ("sti");
/* Spin for a bit - at 100Hz one tick arrives in <10ms */
for (volatile int i = 0; i < 10000000; i++);
__asm__ volatile ("cli");

```

Expected: "IRQ0 fired!" appears on screen and serial. If it appears more than once during the spin, the timer is firing repeatedly (correct). If the system hangs or reboots, EOI is not being sent — verify `irq_dispatcher` calls `pic_send_eoi(irq)`.

Remove the test handler after verifying.

Phase 6 — PIT Channel 0 at 100Hz (2–3 hours)

Create `include/pit.h` and `interrupts/pit.c`. Implement `pit_set_frequency()`, `pit_init()`, `timer_handler()`. Declare `volatile uint64_t pit_tick_count` in `pit.c` (extern declaration in `pit.h`).

uint64_t and 32-bit: A 64-bit variable on a 32-bit kernel requires two 32-bit registers. Increment via `pit_tick_count++` in C is safe — the compiler generates a 32-bit add with carry propagation. However, a reader that reads the high and low words non-atomically may observe a torn value if an interrupt fires between the two reads. For this module (single counter, no atomic reader needed), this is acceptable. A critical reader should disable interrupts around the 64-bit read.

Checkpoint 6: Add to `kmain()` after `pit_init(100)` and `sti`:

```

uint64_t t0 = pit_tick_count;
/* Busy-wait approximately 100ms */
for (volatile int i = 0; i < 50000000; i++);
uint64_t t1 = pit_tick_count;
kprintf("Ticks in ~100ms: %u (expect ~10)\n", (uint32_t)(t1 - t0));

```

Expected output: "Ticks in ~100ms: 9" to "Ticks in ~100ms: 11" (within ± 1 tick due to busy-wait calibration uncertainty on QEMU). If `0`: `volatile` missing from `pit_tick_count`, timer handler not calling increment, or EOI not sent (timer fires once only). If `>1000`: PIT divisor written in wrong byte order.

Phase 7 — PS/2 Keyboard Driver (3–4 hours)

Create `include/keyboard.h` and `interrupts/keyboard.c`. Implement the scancode tables, `kb_ring_t` buffer, `keyboard_handler()`, `keyboard_init()`, `keyboard_getchar()`.

Checkpoint 7: Add to `kmain()` after `keyboard_init()` and `sti`:

```
kprintf("Type keys (echo test, ESC to stop):\n");

while (1) {

    char c = keyboard_getchar();

    if (c) {

        if (c == 27) break; /* ESC = scancode 0x01 → ASCII 27 */

        kprintf("%c", c);

    }

}

kprintf("\nKeyboard test complete.\n");
```

Verify:

- Typing lowercase letters: correct ASCII appears
- Typing with Shift held: uppercase letters appear
- Typing digits with Shift: !@#\$%^&*() appear
- Pressing and releasing keys: each key produces exactly one character (not two)
- Pressing arrow keys: no character appears (extended keys silently ignored)
- Pressing ESC: loop exits
- Typing rapidly: no crash; characters may be lost if buffer fills, but no corruption

Phase 8 — Integration and Full kmain Wiring (2–3 hours)

Update `kmain()` to the final initialization sequence with correct ordering. Remove all temporary tests from previous phases.

```
void kmain(void) {  
    /* Milestone 1 (already complete) */  
  
    serial_init();  
  
    vga_clear();  
  
    kprintf("== Build-OS Kernel ==\n");  
  
    /* Milestone 2: interrupt infrastructure */  
  
    idt_setup_all();  
  
    kprintf("[IDT] 256 gates installed, IDTR loaded.\n");  
  
    pic_remap(0x20, 0x28);  
  
    kprintf("[PIC] Remapped: IRQ0-7 -> 32-39, IRQ8-15 -> 40-47.\n");  
  
    pit_init(100);  
  
    kprintf("[PIT] Timer at 100Hz (divisor=11932).\n");  
  
    keyboard_init();  
  
    kprintf("[KBD] PS/2 keyboard driver ready.\n");  
  
    __asm__ volatile ("sti");  
  
    kprintf("[INT] Interrupts enabled.\n");  
  
    /* Echo demo */  
  
    kprintf("Type keys (ESC to enter idle loop):\n");  
  
    while (1) {  
        char c = keyboard_getchar();  
  
        if (c == 27) break;  
  
        if (c) kprintf("%c", c);  
    }  
  
    kprintf("\nIdle. Tick counter running.\n");  
  
    while (1) {  
        __asm__ volatile ("hlt");  
    }  
}
```

Checkpoint 8 (final acceptance): Run:

```
qemu-system-i386 -drive format=raw,file=os.img -serial stdio 2>/dev/null
```

BASH

Verify all of the following:

1. Screen shows IDT/PIC/PIT/KBD initialization messages in order.
2. "Interrupts enabled." appears without a reboot (PIC remap successful).
3. Key presses echo correctly with correct case.
4. After pressing ESC, "Idle." appears and the system continues running (hlt loop).
5. Serial terminal shows identical output to VGA.
6. QEMU does not reboot during the entire session.

Run with `-d int 2>int.log` and verify:

- No `v=00` through `v=1F` appear after initialization (no spurious exceptions).
- `v=20` appears approximately every 10ms (100Hz timer).
- `v=21` appears on each keypress (keyboard IRQ).
- No `v=08` (double fault) ever appears.

8. Test Specification

8.1 IDT Structure Tests

T-IDT-1: Gate descriptor byte layout

- Setup: Call `idt_set_gate(0x20, 0xDEADBEEF, 0x0008, 0x8E)`.
- Verify via pointer arithmetic into `idt[0x20]`:
 - `*((uint16_t*)&idt[0x20] + 0) = 0xBEEF` (offset_low)
 - `*((uint16_t*)&idt[0x20] + 1) = 0x0008` (selector)
 - `*((uint8_t*)&idt[0x20] + 4) = 0x00` (zero byte)
 - `*((uint8_t*)&idt[0x20] + 5) = 0x8E` (type_attr)
 - `*((uint16_t*)&idt[0x20] + 3) = 0xDEAD` (offset_high)
- Pass condition: All byte comparisons true.

T-IDT-2: IDTR loaded correctly

- After `idt_install()`: read IDTR via `sidt [buf]`.
- Verify: `buf.limit == 2047`, `buf.base == (uint32_t)idt`.
- Implementation: `_asm_ volatile ("sidt %0" : "=m"(idtr_verify))`.

T-IDT-3: All 256 entries present

- After `idt_setup_all()`: verify no entry has `type_attr == 0` (which would indicate an uninitialized gate without the present bit).
- Implementation: Loop through `idt[0]` to `idt[255]`; assert `(idt[i].type_attr & 0x80) != 0` for all i.

T-IDT-4: Exception vectors use trap gates

- After `idt_setup_all()`: for vectors 0–31, assert `idt[i].type_attr == IDT_TRAP_GATE (0x8F)`.

T-IDT-5: IRQ vectors use interrupt gates

- After `idt_setup_all()`: for vectors 32–47, assert `idt[i].type_attr == IDT_INTERRUPT_GATE (0x8E)`.

8.2 Exception Handler Tests

T-EXC-1: Divide-by-zero handler (vector 0)

- Trigger: `volatile int z = 0; volatile int r = 1 / z;`
- Expected: Screen shows "Exception: Divide Error (vector 0)" and "System halted." — then halts.
- QEMU: Does NOT reboot (triple fault would cause reboot); system halts at `cli; hlt`.
- Verify via `-d int`: last vector before halt is `v=00`.

T-EXC-2: GPF handler (vector 13)

- Trigger: `__asm__ volatile ("mov $0, %%ax; mov %%ax, %%ds" ::: "ax")` — loads null selector into DS.
- Expected: "Exception: General Protection Fault (vector 13)". Error code = 0 (null selector: selector value 0 → error code 0).
- Verify: EIP printed matches the address of the inline asm instruction (± 4 bytes).

T-EXC-3: Page fault handler (vector 14) — prints CR2

- Trigger: `volatile int *p = (int*)0xDEAD0000; volatile int x = *p;` (unmapped address; works once paging is enabled in mod-3; for mod-2 without paging this test is skipped or deferred).
- Expected: "Exception: Page Fault (vector 14)", "CR2 (faulting addr): 0xdead0000".

T-EXC-4: Double fault handler (vector 8) prints and halts — does NOT triple-fault

- Trigger: Cannot be directly triggered without a cascaded fault; verify via `qemu -d int` that the handler stub is correctly wired by checking vector 8's IDT entry points to `isr8`.
- Static verify: `nm kernel.elf | grep isr8` shows the symbol exists at a valid address.

T-EXC-5: Unknown exception (vector 21) prints diagnostic

- Trigger: Software `int 21` (currently installed with `default_isr` or `isr21`).
- Expected: Some diagnostic output and halt, without triple fault.

T-EXC-6: Register state preserved for non-halting handlers (future integration)

- Not applicable in this module (all exception handlers halt). This test applies in mod-4 when exception handlers may return.

8.3 PIC Tests

T-PIC-1: No double fault after `sti` (PIC remapped correctly)

- Setup: `idt_setup_all()`, `pic_remap(0x20, 0x28)`, then `sti` for 100ms, then `cli`.
- Expected: No QEMU reboot. Timer fires at vector 32 (not vector 8).
- Via `-d int`: `v=20` appears; `v=08` does NOT appear.

T-PIC-2: EOI re-enables IRQ line

- Setup: Install a counter handler on IRQ0. Enable interrupts. Wait 500ms.
- Expected: Counter increments multiple times (>40 ticks in 500ms at 100Hz). If EOI were missing, counter would be ≤ 1 .

T-PIC-3: Master-only EOI for IRQ0–7 (excluding IRQ2)

- Setup: Call `pic_send_eoi(5)` (IRQ5, master only).
- Verify: Only one `outb(0x20, 0x20)` issued; `outb(0xA0, 0x20)` NOT issued.
- Implementation: Instrument `pic_send_eoi` with a port-write counter or verify via code inspection.

T-PIC-4: Both EOIs for IRQ8–15

- Setup: Call `pic_send_eoi(12)` (IRQ12, mouse, slave).
- Verify: `outb(0xA0, 0x20)` then `outb(0x20, 0x20)` both issued, in that order.

T-PIC-5: IMR preserved after remap

- Setup: Read master IMR before remap. Call `pic_remap()`. Read master IMR after remap.
- Expected: Both reads return the same value (masks unchanged by remap).

T-PIC-6: Spurious IRQ7 does not send EOI

- Setup: Install `spurious7_handler`. Manually set `pic_suppress_eoi = 0`. Call `spurious7_handler` directly with a fake frame where master ISR bit 7 is clear (requires `pic_get_isr` to return 0).
- Expected: After handler returns, `pic_suppress_eoi == 1`. No `outb(0x20, 0x20)` issued by `irq_dispatcher`.
- Note: This test requires mocking `pic_get_isr()`; acceptable to test via code inspection if mocking is not implemented.

8.4 PIT Tests

T-PIT-1: Divisor bytes written in correct order

- Setup: Instrument `outb(0x40, ...)` to capture the two writes.
- Expected: First write = `11932 & 0xFF = 0xDC`. Second write = `(11932 >> 8) & 0xFF = 0x2E`.
- Via `objdump`: Verify the constant bytes 0xDC and 0x2E appear in the correct order in the compiled `pit_set_frequency` function.

T-PIT-2: Command byte is 0x36

- Setup: Instrument `outb(0x43, ...)`.
- Expected: Single write with value `0x36`.

T-PIT-3: `pit_tick_count` increments at 100Hz

- Setup: Record `t0 = pit_tick_count`. Busy-wait exactly 100ms. Record `t1`.
- Expected: `t1 - t0` in range [9, 11].
- Calibration: On QEMU, a busy-wait loop of 50,000,000 iterations ≈ 100ms on a 2GHz host (adjust empirically via first boot).

T-PIT-4: `pit_tick_count` declared volatile

- Verify: `objdump -d pit.o | grep pit_tick_count` — the counter is accessed via a memory reference, not a register-cached load. Alternatively: compile with `-O2` and verify the increment is still present in the output.

T-PIT-5: Timer handler registered on IRQ0

- After `pit_init(100)`: `irq_handlers[0] == timer_handler` (verify by printing the pointer value and comparing to `nm kernel.elf | grep timer_handler`).

8.5 Keyboard Driver Tests

T-KB-1: Lowercase letter translation

- Setup: Keyboard handler called with scancode `0x1E` ('a' make code).
- Expected: `keyboard_getchar()` returns `'a'` (0x61).

T-KB-2: Uppercase letter with Shift

- Setup: Call handler with `0x2A` (left Shift make), then `0x1E` ('a' make).
- Expected: `keyboard_getchar()` returns `'A'` (0x41).

T-KB-3: Break code filtered (no character)

- Setup: Call handler with `0x9E` (break code for `'a'` : `0x1E | 0x80`).
- Expected: `keyboard_getchar()` returns `0` (nothing buffered).

T-KB-4: Shift released

- Setup: Call handler with `0x2A` (Shift make), `0x1E` (`'a'` make), `0xAA` (Shift break), `0x1E` (`'a'` make).
- Expected: Two `getchar()` calls return `'A'` then `'a'`.

T-KB-5: Extended prefix ignored

- Setup: Call handler with `0xE0`, then `0x48` (numpad 8 / up arrow extended).
- Expected: `keyboard_getchar()` returns `0` (both bytes consumed; no character emitted).

T-KB-6: Buffer full — oldest preserved, newest dropped

- Setup: Call handler 258 times with scancode `0x1E` (`'a'`). (256 characters + 2 overflow attempts).
- Expected: First 255 `keyboard_getchar()` calls return `'a'`. 256th returns `0` (buffer was full at 255, not 256 — full condition is when `(head+1) == tail`, so max capacity is 255 characters in a 256-byte buffer).

T-KB-7: Space bar

- Setup: Handler called with `0x39` (spacebar make code).
- Expected: `keyboard_getchar()` returns `' '` (0x20).

T-KB-8: Enter key

- Setup: Handler called with `0x1C` (Enter make code).
- Expected: `keyboard_getchar()` returns `'\n'` (0x0A).

T-KB-9: Backspace

- Setup: Handler called with `0x0E` (Backspace make code).
- Expected: `keyboard_getchar()` returns `'\b'` (0x08).

T-KB-10: Unmapped key (e.g., F1 = `0x3B`)

- Setup: Handler called with `0x3B`.
- Expected: `keyboard_getchar()` returns `0` (F1 has no ASCII mapping: `scancode_ascii_lower[0x3B] == 0`).

T-KB-11: ESC key

- Setup: Handler called with `0x01`.
- Expected: `keyboard_getchar()` returns `'\x1B'` (ESC = 27 = 0x1B).

T-KB-12: Integration echo test

- In QEMU: Type `"Hello, World!\n"` with correct Shift usage.
- Expected: Screen and serial show exactly `"Hello, World!"` followed by a newline (the `\n` from Enter is echoed to VGA which advances the cursor).

8.6 Stack Frame Tests

T-FRAME-1: Stack symmetry — pusha count matches popa

- Via `objdump -d isr_stubs.o`: Count push instructions from `isr_common_stub` entry to `call exception_handler`. Count pop instructions from after `add esp, 4` to `iret`. The two counts must match (4 segment push/pop pairs + 1 pusha/popa pair + 1 esp argument push + cleanup).

T-FRAME-2: `add esp, 8` present after `popa`

- Via `objdump -d isr_stubs.o` : Verify `add esp, 0x8` (or `add $0x8,%esp`) appears between `popa` and `iret` in `isr_common_stub`.

T-FRAME-3: `interrupt_frame_t` offsets match assembly

- Compare `offsetof(interrupt_frame_t, vector)` (must be 48) against the stack state when `exception_handler` is called: 4 segment reg pushes (16 bytes) + 8 general-purpose register pushes from pusha (32 bytes) = 48 bytes above the frame pointer. Verified by static assert in §5.1.

T-FRAME-4: Register preservation across timer IRQ

- Setup: Set all general-purpose registers to known values via inline asm. Call `pit_init(100)`, `sti`. Spin for 50ms (several timer ticks fire). `cli`. Read all registers via inline asm.
 - Expected: All registers have their pre-interrupt values. If `pusha / popa` are correct, this holds.
 - Implementation: Achievable via a carefully crafted inline asm block that sets registers, executes a measured spin with `sti / cli`, then reads registers back.
-

9. Performance Targets

Operation	Target	How to Measure
IDT lookup + handler dispatch	< 20 cycles (IDT fits in L1 cache: 2048 bytes; well within 32KB L1)	<code>rdtsc</code> before triggering a software <code>int N</code> and after handler returns; subtract overhead
Full IRQ round-trip (<code>pusha</code> → handler → <code>popa</code> → <code>iret</code>)	< 150 cycles for timer handler (no output, just increment + EOI)	<code>rdtsc</code> immediately after <code>iret</code> via a dedicated measurement process; 100 iterations, average
Interrupt latency (IRQ assert → first handler instruction)	< 50 cycles at 2GHz = < 25ns	Measure with hardware (logic analyzer on real hardware); QEMU: use <code>-d int</code> timestamp delta
Timer handler total (increment + EOI)	< 60 cycles	<code>rdtsc</code> in timer_handler at start and end; kprintf overhead excluded (do not kprintf in production timer handler)
Keyboard handler (<code>inb</code> + table lookup + buffer write)	< 200 cycles (dominated by 2× I/O port accesses at ~100 cycles each on QEMU)	<code>rdtsc</code> around keyboard_handler body; average over 20 keypresses
<code>keyboard_getchar()</code> — buffer read	< 10 cycles (two integer comparisons + array read + increment)	<code>rdtsc</code> around 100 consecutive getchar calls; divide by 100
PIC EOI write	< 10 cycles (single outb to I/O port; fast on QEMU)	<code>rdtsc</code> around <code>pic_send_eoi(0)</code>
<code>pic_remap()</code> total (8 port writes + 8 <code>io_wait</code> calls)	< 5μs (8 × <code>io_wait</code> ≈ 8 × ~2μs = 16μs + port write overhead)	<code>rdtsc</code> around <code>pic_remap()</code> ; convert cycles to μs at host clock rate
<code>pit_init()</code> total	< 10μs (3 port writes, no <code>io_wait</code>)	<code>rdtsc</code> around <code>pit_init(100)</code>
IRQ overhead at 100Hz (100 ticks/sec × 150 cycles)	0.0075% CPU at 2GHz (15,000 cycles/sec out of 2,000,000,000)	Theoretical calculation; verify: timer handler runs 100× per second at 150 cycles each = 15,000 cycles/sec / 2GHz = 0.00075%
IDT full population (256 × <code>idt_set_gate</code>)	< 5μs total (256 × ~5 instructions each)	<code>rdtsc</code> around <code>idt_setup_all()</code> ; this is a one-time boot cost

QEMU timing note: QEMU's virtual TSC does not precisely track wall-clock time. For cycle counting, use `rdtsc` and accept that the "cycles" are virtual. For timing accuracy (e.g., verifying 100Hz means 10ms/tick), use wall-clock measurement by counting ticks over a known time period (Checkpoint 6).

10. Hardware Soul Analysis

IDT Lookup Cache Behavior

The IDT array is 2048 bytes. A typical L1 data cache is 32KB with 64-byte cache lines. The IDT occupies exactly 32 cache lines. After the first access to any IDT entry, subsequent accesses to nearby entries are L1 hits. Since interrupt handling tends to invoke the same small set of vectors (timer at vector 32, keyboard at vector 33) repeatedly, those entries are perpetually hot in L1 cache. The L1 hit latency for the IDT lookup is 4–5 cycles — negligible compared to the pipeline flush cost of the interrupt itself.

The IDTR register is a CPU-internal register (not a cache-line in memory); the CPU does not "cache" the IDT entries themselves — it re-reads them from memory on each interrupt, but those reads hit L1 cache after the first invocation.

PIC Port I/O Latency

Every `inb` / `outb` to PIC ports `0x20`, `0x21`, `0xA0`, `0xA1` crosses the legacy ISA-emulated I/O bus. On real hardware, this costs 1–4 μ s per access (the I/O bus is slow relative to DRAM). The `io_wait()` call adds another ~1 μ s. Total cost of `pic_remap()`: 16 writes \times ~3 μ s each = ~48 μ s — a one-time boot cost, irrelevant at runtime.

At runtime, `pic_send_eoi()` issues 1–2 `outb` calls. On QEMU, each I/O write is cheaper than real hardware (sub-microsecond), but blocking is still measurable. This is why production kernels with high IRQ rates (>1M IRQs/sec) use MSI-X (Message Signaled Interrupts eXtended) — a mechanism that writes to a memory address rather than an I/O port, costing a single DRAM write (60ns) instead of an ISA I/O transaction (~3 μ s), a 50× speedup.

`pusha` / `popa` Microarchitecture

On modern Intel CPUs (Skylake+), `pusha` is not a "fast string" operation — it is decoded as a sequence of 8 individual push micro-operations. Each push writes 4 bytes and decrements ESP. Total: 8 micro-ops. On a CPU with 4 execution ports for integer operations, the 8 pushes complete in approximately 2–4 cycles. `popa` is symmetric. The entire `pusha` + 4 segment pushes + `popa` + 4 segment pops costs approximately 30–40 cycles — measurable but small relative to the pipeline flush cost of the interrupt itself (~20–30 cycles for the branch misprediction from the far jump to the handler).

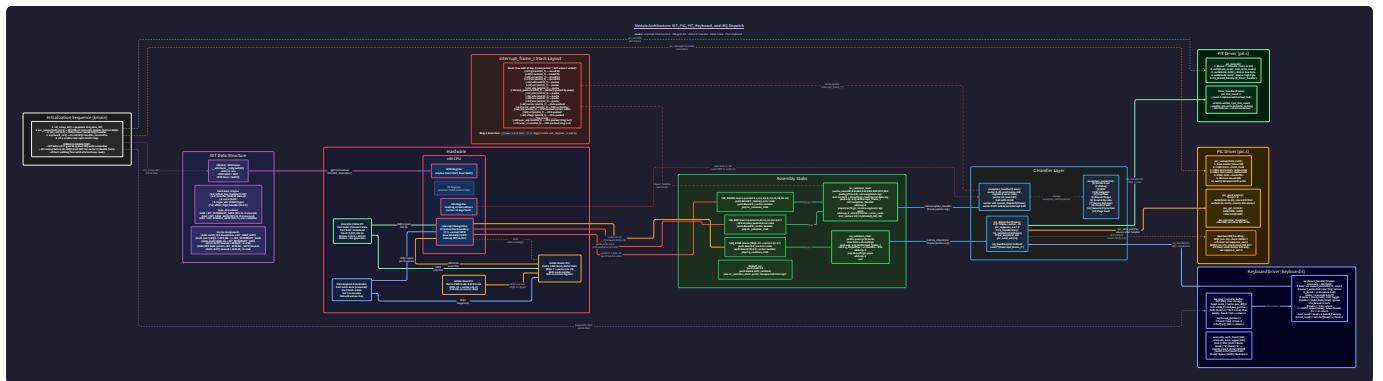
Circular Buffer Memory Access Pattern

`kb_ring.buf` is a 256-byte array, fitting entirely within 4 cache lines (4×64 bytes = 256 bytes). After the first access warms the cache, all subsequent reads and writes are L1 hits. The `head` and `tail` fields follow the buffer in the same struct and are likely in the same or adjacent cache line as the buffer's first bytes. A keyboard interrupt that writes to the buffer and increments `head` therefore requires at most 2 L1 cache accesses — essentially free from a memory latency perspective.

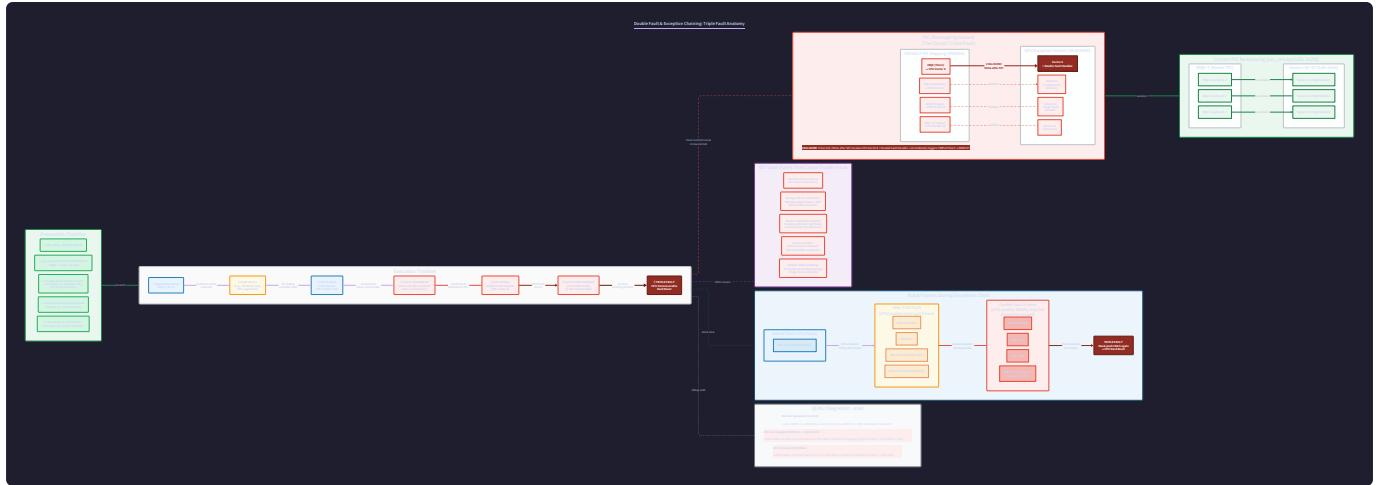
Branch Predictability in Exception Dispatch

`exception_handler`'s switch on `frame->vector` is compiled as a series of conditional branches or a jump table. For a kernel that almost never triggers exceptions during normal operation, these branches are almost always not-taken — the predictor learns quickly that exceptions are rare events. On the rare invocation (a real fault), the misprediction cost (~15 cycles) is irrelevant because the handler is about to halt the system.

The keyboard handler's branch on `is_break` is highly predictable: in a typical interactive session, half of all IRQ1 firings are make codes and half are break codes (each key generates exactly one of each). The predictor cannot predict this alternating pattern better than 50% accuracy — a 50% misprediction rate at 15 cycles each = ~7.5 cycles average branch cost per keyboard event. Completely negligible at human typing speeds (<20 characters/second).



11. State Machine



Module lifecycle states:

State	Description	Entry Action	Legal Transitions
BOOT_COMPLETE	Milestone 1 finished; no interrupt infrastructure	— (inherited state)	→ IDT_LOADED
IDT_LOADED	All 256 IDT gates set; IDTR points to <code>idt[]</code>	<code>idt_setup_all()</code>	→ PIC_REMAPPED
PIC_REMAPPED	8259 PIC cascade remapped; IRQ0 → 32, IRQ8 → 40	<code>pic_remap(0x20, 0x28)</code>	→ TIMER_RUNNING
TIMER_RUNNING	PIT channel 0 at 100Hz; <code>timer_handler</code> on IRQ0	<code>pit_init(100)</code>	→ KBD_READY
KBD_READY	Keyboard handler on IRQ1; buffer initialized	<code>keyboard_init()</code>	→ INTERRUPTS_ENABLED
INTERRUPTS_ENABLED	<code>EFLAGS.IF = 1</code> ; hardware IRQs firing	<code>sti</code> instruction	→ IDLE
IDLE	<code>kmain</code> in <code>hlt</code> loop; timer ticking; keyboard echoing	(steady state)	→ EXCEPTION_HALT on fault
EXCEPTION_HALT	CPU exception occurred; diagnostic printed	<code>exception_handler</code>	(terminal state)

Illegal state transitions:

Illegal Transition	Why Illegal	Consequence
<code>BOOT_COMPLETE</code> → <code>INTERRUPTS_ENABLED</code> (skip IDT/PIC)	No IDT loaded; first interrupt uses BIOS IVT	Triple fault
<code>IDT_LOADED</code> → <code>INTERRUPTS_ENABLED</code> (skip PIC remap)	IRQ0 fires at vector 8 (double fault)	Triple fault (double fault handler cascades)
<code>PIC_REMAPPED</code> → <code>INTERRUPTS_ENABLED</code> (skip PIT init)	No timer handler; IRQ0 fires → <code>irq_dispatcher</code> → no handler → EOI only → tick counter never increments	No crash; timer dysfunctional
Any state → <code>PIC_REMAPPED</code> (remap a second time mid-operation)	Undefined PIC internal state during reinit; an IRQ could fire between ICW1 and ICW4	Spurious interrupt with wrong vector; potential GPF
<code>EXCEPTION_HALT</code> → any state	Halt is terminal; <code>cli</code> ; <code>hlt</code> loop cannot be exited except via NMI or hardware reset	NMI would invoke NMI handler (vector 2); if not installed, triple fault

12. Concurrency Specification

Model: Single core. No threads. No SMP. The concurrency model is interrupt preemption only — IRQ handlers are the only entities that can preempt kernel code.

Critical section: `irq_handlers[]` array modification

- Writer: `irq_install_handler()` (called from kernel initialization code).
- Readers: `irq_dispatcher()` (called from IRQ handler with IF=0).
- Guarantee: All `irq_install_handler()` calls are made before `sti`. After `sti`, `irq_handlers[]` is read-only from the kernel's perspective. No lock needed.
- If `irq_install_handler()` must be called after `sti` (not in this module): caller must execute `cli` around the write and `sti` afterward.

Critical section: `kb_ring` circular buffer

- Writer: `keyboard_handler()` — runs with IF=0 (interrupt gate). Cannot be interrupted.
- Reader: `keyboard_getchar()` — runs with IF=1 (normal kernel code). Can be interrupted by timer or keyboard IRQ.
- Race condition analysis: `keyboard_getchar()` reads `kb_ring.head` and `kb_ring.tail`. The writer advances `kb_ring.head`. On a single-core CPU:
 - If a keyboard IRQ fires while `keyboard_getchar()` is between reading `head` and reading `tail`: the IRQ handler runs to completion (modifying `head`), then `keyboard_getchar()` resumes with the old `head` value. This is a transient stale read that causes one character to be missed for one `getchar()` call — not a corruption, and extremely unlikely given keyboard timing.
 - Practically: single-byte reads and writes to `uint8_t` fields are atomic on x86 (aligned byte access). The only race would be if `keyboard_getchar()` read `head` and `tail` from different memory accesses that an IRQ interrupts in between. Since `head` and `tail` are separate bytes, this is a benign race: at worst `getchar()` returns `0` (empty) when one character exists, which it will read on the next call.
- Decision: No lock needed for this single-core, single-writer, single-reader design.

Critical section: `pit_tick_count` increment

- Writer: `timer_handler()` — with IF=0.

- Reader: Any kernel code that polls the counter — with IF=1.
- `volatile uint64_t` ensures the compiler does not cache the value. On a 32-bit CPU, a 64-bit increment is not atomic (two 32-bit operations). A reader that reads both halves with IF=1 could read the low half after increment and the high half before carry propagation — a torn read. Maximum error: 1 tick (approximately 10ms). For this module's use cases (coarse timing), this is acceptable.
- If atomic 64-bit reads are required: disable interrupts around the read with `cli / sti`.

pic_suppress_eoi flag:

- Set by spurious handlers; read by `irq_dispatcher`; both run with IF=0. No race.

13. Synced Criteria

Technical Design Specification: Physical and Virtual Memory Management

Module ID: mod-3 | **Milestone:** build-os-m3 | **Estimated Hours:** 23–34

1. Module Charter

This module implements the complete memory subsystem for a 32-bit x86 kernel: it parses the multiboot-provided E820 memory map to enumerate physical RAM regions, manages physical 4KB frames via a 128KB bitmap allocator, constructs two-level x86 page tables (one page directory, two static page tables for identity and higher-half mapping), enables paging by loading CR3 and setting CR0.PG, enforces TLB coherence via `invlpg` and CR3 reload, handles page faults by decoding CR2 and the error code, and provides a variable-size kernel heap allocator (`kmalloc / kfree`) backed by on-demand physical frame allocation and virtual address mapping.

This module does **not** implement per-process page directories, demand paging, copy-on-write, swap, memory-mapped files, user-space `malloc`, SLUB or slab caching, multi-core TLB shootdowns, huge pages (4MB PSE), or any memory-protection features beyond the existing U/S and R/W page table flags. The TSS is configured in mod-4; this module does not touch it. ACPI table parsing beyond E820 type classification is out of scope.

Upstream dependencies: Module mod-1 (protected mode, VGA, serial, `kprintf`, linker symbols `__kernel_start_phys / __kernel_end_phys`, `__bss_start / __bss_end`); module mod-2 (IDT with exception handler at vector 14 installable, `sti` already called — paging enablement must not break interrupt delivery).

Downstream dependencies: Module mod-4 (process management) calls `pmm_alloc_frame()` to allocate per-process page directories and kernel stacks, calls `vmm_map_page()` to map user-mode code and stack frames, and reads `boot_page_directory` to copy kernel PDE entries into per-process directories. The heap (`kmalloc / kfree`) is used by all subsequent kernel subsystems.

Invariants that must hold at module exit:

1. `frame_bitmap` correctly reflects physical RAM state: every usable, non-kernel, non-page-table frame has its bit clear (free); every kernel, page-table, reserved, and frame-below-1MB frame has its bit set (used).
2. `CR3` holds the physical address of `boot_page_directory`. `CR0.PG = 1`. Paging is active.
3. Virtual `0x00000000 – 0x003FFFFF` maps to physical `0x00000000 – 0x003FFFFF` (identity, covers VGA at `0xB8000` and kernel at `0x100000`).
4. Virtual `0xC0000000 – 0xC03FFFFF` maps to physical `0x00000000 – 0x003FFFFF` (higher-half kernel).

5. All page table structures (`boot_page_directory`, `identity_page_table`, `kernel_page_table`) are themselves marked used in `frame_bitmap`.
 6. `invlpg` or CR3 reload is called immediately after every PTE or PDE modification; no stale TLB entry persists beyond the modifying function's return.
 7. `kmalloc(n)` returns an 8-byte-aligned pointer to at least `n` usable bytes, or `NULL` only when physical frames are exhausted. `kfree(p)` returns the block to the free list and coalesces adjacent free blocks. A double-free on the same pointer is detected via the magic canary and reported via `kprintf` without crashing.
-

2. File Structure

Create files in this exact sequence. Later files include headers from earlier files.

```
build-os/
└── kernel/
    └── include/
        ├── 01  multiboot.h          # multiboot_info_t, mmap_entry_t, flag constants
        ├── 02  pmm.h               # Physical Memory Manager public API + constants
        ├── 03  page.h              # pde_t, pte_t, flag bit constants, address macros
        └── 04  vmm.h               # VMM public API: vmm_init(), vmm_enable_paging(), vmm_map_page(),
vmm_unmap_page(), TLB flush
            └── 05  heap.h           # heap_block_t, heap_init(), kmalloc(), kfree()
        └── mm/
            └── 01  pmm.c            # frame_bitmap[], pmm_parse_memory_map(), pmm_init(), pmm_alloc_frame(),
pmm_free_frame()
                └── 02  vmm.c          # boot_page_directory[], identity/kernel page tables, vmm_init(),
vmm_enable_paging(), vmm_map_page(), vmm_unmap_page(), TLB flush wrappers, page fault handler registration
                    └── 03  heap.c          # heap_block_t list, heap_init(), kmalloc(), kfree(), heap_expand()
        └── kmain.c                # Updated: calls pmm_parse_memory_map(), pmm_init(), vmm_init(),
vmm_enable_paging(), heap_init()
```

Why this order: `multiboot.h` defines types used by `pmm.h` and `pmm.c`. `page.h` defines `pde_t` / `pte_t` used by `vmm.h` and `vmm.c`. `pmm.h` must exist before `vmm.c` (which calls `pmm_alloc_frame()`). `vmm.h` must exist before `heap.c` (which calls `vmm_map_page()`). All three `.c` files are independent of each other except for the call dependencies just described.

3. Complete Data Model

3.1 `multiboot_info_t` and `mmap_entry_t` — Wire Format

The multiboot specification defines these structures. They arrive from GRUB via a pointer in `EBX` at kernel entry time. The pointer must be saved before BSS zeroing in `entry.asm` (or passed as argument to `kmain`).

```

/* include/multiboot.h */

#ifndef MULTIBOOT_H
#define MULTIBOOT_H

#include <stdint.h>

/* Flags bits in multiboot_info_t.flags */
#define MB_FLAG_MEM      (1 << 0) /* mem_lower / mem_upper valid */
#define MB_FLAG_MMAP     (1 << 6) /* mmap_addr / mmap_length valid */

typedef struct __attribute__((packed)) {
    uint32_t flags;           /* Offset 0: which fields below are valid */
    uint32_t mem_lower;       /* Offset 4: KB below 1MB (typically 640) */
    uint32_t mem_upper;       /* Offset 8: KB above 1MB */
    uint32_t boot_device;     /* Offset 12: BIOS boot device */
    uint32_t cmdline;         /* Offset 16: physical addr of cmdline str */
    uint32_t mods_count;      /* Offset 20: boot modules count */
    uint32_t mods_addr;        /* Offset 24: physical addr of modules list */
    uint32_t syms[4];          /* Offset 28: symbol table info (a.out/ELF) */
    uint32_t mmap_length;      /* Offset 44: byte length of mmap buffer */
    uint32_t mmap_addr;        /* Offset 48: physical addr of mmap buffer */
    /* Many more fields exist (drives, config table, etc.) - omitted */
} multiboot_info_t;           /* We only access offsets 0-51 */

/* Each entry in the memory map buffer.

NOTE: entries are variable-length. Next entry = (uint8_t*)entry + entry->size + 4
The +4 accounts for the 'size' field itself, which is NOT included in size's value. */

typedef struct __attribute__((packed)) {
    uint32_t size;            /* Offset 0: size of remaining fields (usually 20) */
    uint64_t addr;             /* Offset 4: physical base address of this region */
    uint64_t len;              /* Offset 12: length in bytes */
    uint32_t type;             /* Offset 20: E820 type (see constants below) */
} mmap_entry_t;               /* sizeof = 24 when size == 20 (standard layout) */

/* E820 memory region types */

#define E820_USABLE     1 /* Free RAM; allocator may use this */

```

```

#define E820_RESERVED    2  /* BIOS-reserved; do not touch      */
#define E820_ACPI_DATA   3  /* ACPI reclaim tables                */
#define E820_ACPI_NVS    4  /* ACPI non-volatile storage          */
#define E820_BAD         5  /* Known bad RAM                     */

#endif /* MULTIBOOT_H */

```

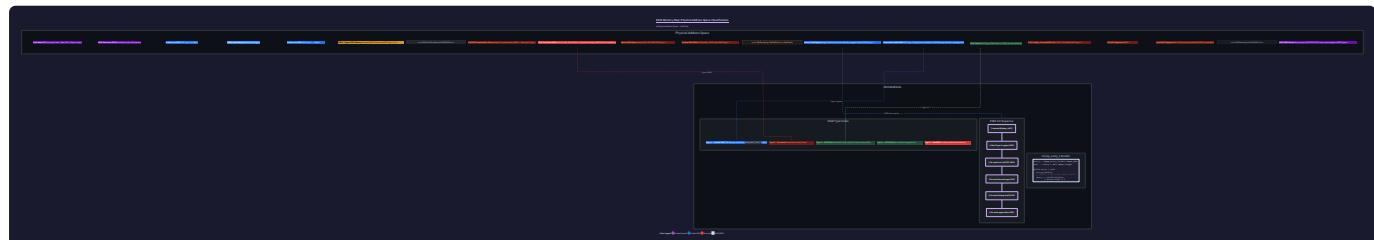
Byte-offset table for `mmap_entry_t` :

Offset	Size	Field	Notes
0	4	<code>size</code>	Does NOT count itself; typically 20
4	8	<code>addr</code>	64-bit physical base; may exceed 4GB (ignored if > 4GB on 32-bit)
12	8	<code>len</code>	64-bit length; region = [addr, addr+len)
20	4	<code>type</code>	E820 classification
Total	24	—	When <code>size == 20</code> ; next entry at <code>(uint8_t*)e + e->size + 4</code>

Critical iterator arithmetic: The variable-length iteration formula is:

```
mmap_entry_t *next = (mmap_entry_t *)((uint8_t *)entry + entry->size + sizeof(uint32_t));
```

`entry->size` is the count of bytes *after* the `size` field. Adding `sizeof(uint32_t)` (= 4) accounts for the `size` field itself. A common bug: using `entry->size` alone, which underflows the iterator by 4 bytes and produces garbage.



3.2 Bitmap Physical Frame Allocator — Data Layout

```
/* include/pmm.h */

#ifndef PMM_H

#define PMM_H

#include <stdint.h>

#include <stddef.h>

#include "multiboot.h"

/* Physical address space constants */

#define PAGE_SIZE      4096u          /* 4KB per frame           */
#define PAGE_SHIFT     12u            /* log2(PAGE_SIZE)         */
#define MAX_PHYS_ADDR (4UL * 1024 * 1024 * 1024) /* 4GB address space      */
#define FRAME_COUNT    (MAX_PHYS_ADDR / PAGE_SIZE) /* 1,048,576 frames       */
#define BITMAP_WORDS   (FRAME_COUNT / 32u)        /* 32,768 uint32_t words = 128KB */

/* Address → frame number conversion */

#define ADDR_TO_FRAME(addr) ((uint32_t)((addr) >> PAGE_SHIFT))
#define FRAME_TO_ADDR(frame) ((uint32_t)((frame) << PAGE_SHIFT))

/* Align address down / up to page boundary */

#define PAGE_ALIGN_DOWN(addr) ((addr) & ~(uint32_t)(PAGE_SIZE - 1))
#define PAGE_ALIGN_UP(addr)   (((addr) + PAGE_SIZE - 1) & ~(uint32_t)(PAGE_SIZE - 1))

/* Public API */

void pmm_parse_memory_map(multiboot_info_t *mbi);
void pmm_init(multiboot_info_t *mbi);
void pmm_mark_used(uint64_t addr, uint64_t length);
void pmm_mark_free(uint64_t addr, uint64_t length);
void *pmm_alloc_frame(void);    /* Returns physical address; NULL on OOM */
void pmm_free_frame(void *addr);

uint32_t pmm_frames_free(void);
uint32_t pmm_frames_used(void);
uint32_t pmm_frames_total(void);

#endif /* PMM_H */
```

Bitmap memory layout (128KB, statically allocated in `.bss`):

```

frame_bitmap[0]      covers frames 0-31          (physical 0x00000000 - 0x0001FFFF)
frame_bitmap[1]      covers frames 32-63         (physical 0x00020000 - 0x0003FFFF)
frame_bitmap[2]      covers frames 64-95         (physical 0x00040000 - 0x0005FFFF)
...
frame_bitmap[32767]   covers frames 1048544-1048575 (physical 0xFFFF0000 - 0xFFFFFFFF)

Bit encoding within each word:
  Bit 0 of frame_bitmap[w] → frame number (w * 32 + 0)
  Bit 1 of frame_bitmap[w] → frame number (w * 32 + 1)
  ...
  Bit 31 of frame_bitmap[w] → frame number (w * 32 + 31)

Bit value: 0 = FREE (available for allocation)
           1 = USED (allocated or reserved; do not allocate)

```

Rationale for "1 = used" encoding: Starting from `memset(frame_bitmap, 0xFF, sizeof(frame_bitmap))` marks all frames as used (conservative). The parser then marks usable regions as free. This means any bug in the parser that misses a region results in that region being treated as reserved — preventing allocation, not corruption. The alternative ("0 = used") would cause uninitialized bitmap regions to appear free.

Bitmap struct fields:

```

/* pmm.c (static, not exposed in header) */

static uint32_t frame_bitmap[BITMAP_WORDS]; /* 128KB in .bss; BSS-zeroed by entry.asm, then overwritten by
                                             pmm_init */

static uint32_t pmm_total_frames = 0;        /* Total usable frames discovered in E820 */

static uint32_t pmm_used_frames = 0;          /* Current count of used frames */

static uint32_t pmm_search_hint = 1;          /* Next-fit scan start position (frame 0 never allocated) */

```

Cache line analysis: `frame_bitmap` is a 128KB array of `uint32_t`. One 64-byte cache line holds 16 `uint32_t` words = 16×32 = 512 bits = 512 frames = 2MB of physical memory represented per cache line. Sequential allocation scans this array with hardware prefetch assistance. After the first full scan warms the L2 cache (128KB fits in any $L2 \geq 256KB$), subsequent allocations hit L2 at ~5ns per cache line. The `search_hint` avoids rescanning already-allocated regions, keeping the common-case access to one or two cache lines.



3.3 x86 Page Directory Entry and Page Table Entry — Bit Layout

Both PDEs and PTEs are `uint32_t`. The flag bits occupy [11:0]; the frame address occupies [31:12].

```

/* include/page.h */

#ifndef PAGE_H

#define PAGE_H

#include <stdint.h>

typedef uint32_t pde_t; /* Page Directory Entry */

typedef uint32_t pte_t; /* Page Table Entry */

/* Flag bits – identical meaning in both PDE and PTE unless noted */

#define PAGE_PRESENT      (1u << 0)   /* P: 1 = entry is valid; 0 = page fault on access */
#define PAGE_WRITABLE     (1u << 1)   /* R/W: 0 = read-only; 1 = read-write */
#define PAGE_USER         (1u << 2)   /* U/S: 0 = supervisor only; 1 = user accessible */
#define PAGE_WRITE_THROUGH (1u << 3)  /* PWT: write-through cache policy */
#define PAGE_CACHE_DISABLE (1u << 4)  /* PCD: 1 = uncachable (use for MMIO pages) */
#define PAGE_ACCESSED     (1u << 5)   /* A: CPU sets on first access; software clears */
#define PAGE_DIRTY        (1u << 6)   /* D: CPU sets on first write (PTE only) */
#define PAGE_LARGE        (1u << 7)   /* PS: PDE only – 1 = 4MB page (requires CR4.PSE) */
#define PAGE_GLOBAL        (1u << 8)   /* G: TLB entry survives CR3 reload (needs CR4.PGE) */

/* Extract physical frame address from a PDE or PTE (zero the low 12 flag bits) */

#define PDE_FRAME(pde)    ((pde) & 0xFFFFF000u)
#define PTE_FRAME(pte)    ((pte) & 0xFFFFF000u)

/* Build a PTE from a physical address and flags */

#define MAKE_PTE(phys_addr, flags)  ((pte_t)((phys_addr) & 0xFFFFF000u) | (flags))
#define MAKE_PDE(phys_addr, flags)  ((pde_t)((phys_addr) & 0xFFFFF000u) | (flags))

/* Virtual address decomposition */

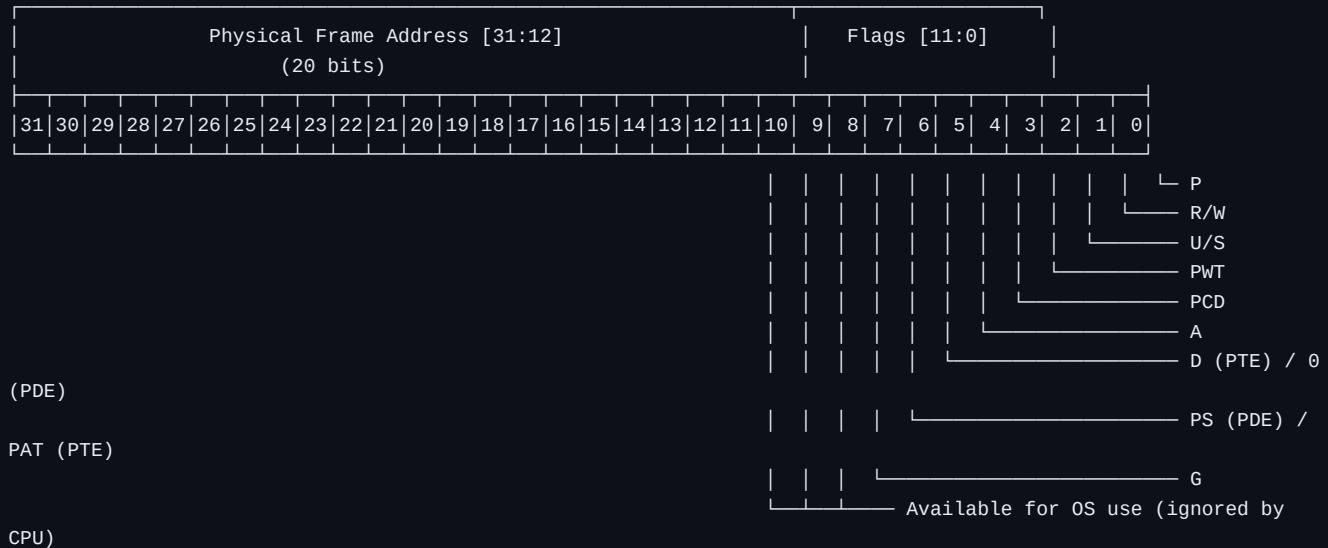
#define VA_DIR_IDX(va)    (((va) >> 22) & 0x3FFu)   /* Bits [31:22]: PD index (0-1023) */
#define VA_TBL_IDX(va)    (((va) >> 12) & 0x3FFu)   /* Bits [21:12]: PT index (0-1023) */
#define VA_OFFSET(va)     ((va) & 0xFFFu)           /* Bits [11:0]: byte offset (0-4095) */

#endif /* PAGE_H */

```

PDE/PTE bit-level diagram:

32-bit PDE or PTE:



Flag combinations used in this module:

Usage	Flags	Value	Notes
Kernel code page (identity)	P W	0x003	Present, writable, supervisor
Kernel data page (identity)	P W	0x003	Same flags; CPU enforces R/W per access type
User code page	P U	0x005	Present, user-accessible, read-only
User data/stack page	P W U	0x007	Present, writable, user-accessible
VGA MMIO page (ideally)	P W PCD PWT	0x01B	Uncacheable; correct for MMIO
Kernel heap backing	P W	0x003	Supervisor only, writable
PDE pointing to page table	P W	0x003	Page table itself is kernel data
Unmapped (not present)	0	0x000	Access triggers page fault

3.4 Two-Level Page Table — Static Layout

Virtual address space layout for this module:

Virtual Address Range	Mapping Target	PD Index	Flags
0x000000000 - 0x003FFFFF	Physical 0x0 - 0x3FFFFF	0	P W (identity map, 4MB)
0x004000000 - 0xBFFFFFFF	(not mapped)	1-767	- (user space; mod-4 fills)
0xC00000000 - 0xC03FFFFF	Physical 0x0 - 0x3FFFFF	768	P W (higher-half kernel)
0xC04000000 - 0xCFFFFFFF	(not mapped)	769-831	- (future kernel use)
0xD00000000 - 0xDFFFFFFF	(heap grows here)	832-895	- (on-demand via vmm_map_page)
0xE00000000 - 0xFFFFFFFF	(not mapped)	896-1023	- (reserved)

Static page directory and page tables:

```

/* vmm.c - statically allocated, aligned to 4KB by attribute */

/* The boot page directory – shared by all kernel threads in mod-4 */

__attribute__((aligned(PAGE_SIZE)))

static pde_t boot_page_directory[1024];      /* 4KB; zeroed by BSS init */

/* Identity map: virtual 0x0–0x3FFFF → physical 0x0–0x3FFFF */

__attribute__((aligned(PAGE_SIZE)))

static pte_t identity_page_table[1024];        /* 4KB; maps PD index 0 */

/* Higher-half kernel map: virtual 0xC0000000–0xC03FFFF → physical 0x0–0x3FFFF */

__attribute__((aligned(PAGE_SIZE)))

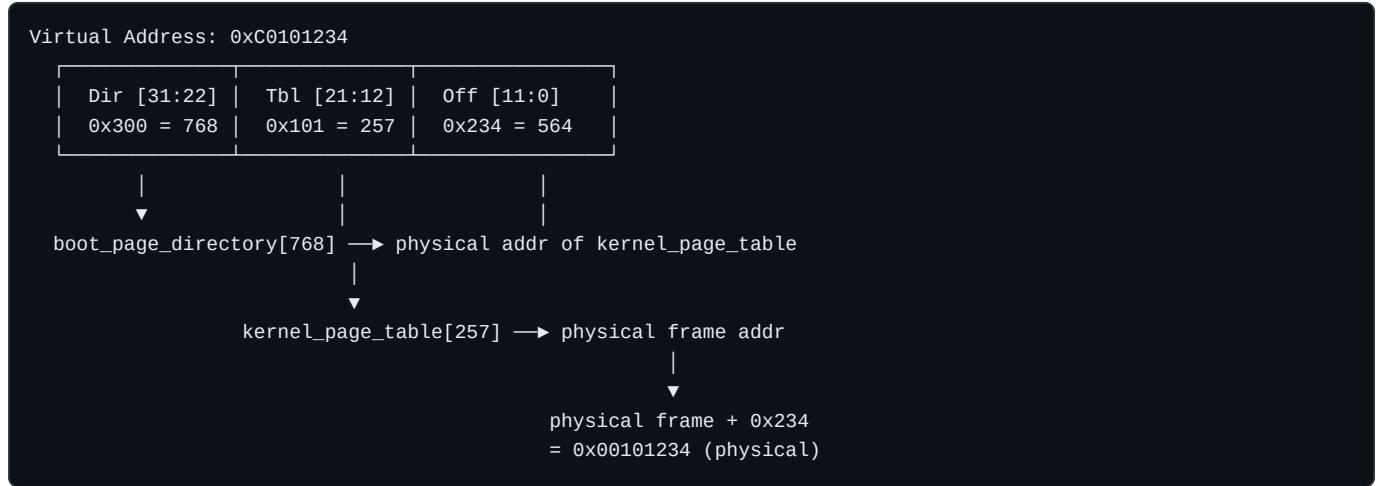
static pte_t kernel_page_table[1024];           /* 4KB; maps PD index 768 */

```

Memory cost: $3 \times 4\text{KB} = 12\text{KB}$ of static kernel BSS. No dynamic allocation needed for boot-time page tables.

Why `__attribute__((aligned(PAGE_SIZE)))`: CR3 and the frame address field of PDEs require 4KB-aligned physical addresses. Without the alignment attribute, the linker places these arrays at their natural alignment (4 bytes), which is insufficient. The GCC `aligned` attribute guarantees placement at a 4KB boundary within the `.bss` section. The linker script must specify `.bss` section alignment $\geq \text{PAGE_SIZE}$; add `.bss : ALIGN(4096) { ... }` if not already present.

Two-level address translation diagram:



{DIAGRAM:tdd-diag-20}

3.5 `heap_block_t` — Kernel Heap Allocator Header

Each allocation is preceded by a `heap_block_t` header. The header is at the lowest address; user data follows immediately after.

```

/* include/heap.h */

#ifndef HEAP_H

#define HEAP_H

#include <stdint.h>

#include <stddef.h>

#define HEAP_MAGIC      0xDEADBEEFu /* Canary; corruption detected when != this */

#define HEAP_START      0x00000000u /* Virtual address where heap region begins */

#define HEAP_MAX        0xFFFFFFFFu /* Virtual address where heap region ends */

#define HEAP_ALIGN      8u          /* All allocations aligned to 8 bytes */

typedef struct heap_block {

    uint32_t      magic; /* Offset 0: HEAP_MAGIC; validated on every access */

    uint32_t      size;   /* Offset 4: usable bytes after this header */

    uint8_t       free;   /* Offset 8: 1 = free; 0 = allocated */

    uint8_t      _pad[3]; /* Offset 9: padding to align 'next'/'prev' to 4B */

    struct heap_block *next; /* Offset 12: next block in linked list (NULL=last) */

    struct heap_block *prev; /* Offset 16: prev block in linked list (NULL=first) */

} heap_block_t; /* sizeof = 20 bytes */

void heap_init(void);

void *kmalloc(size_t size);

void kfree(void *ptr);

size_t kheap_used(void); /* Returns bytes currently allocated (for diagnostics) */

#endif /* HEAP_H */

```

`heap_block_t` memory layout (byte offsets):

Offset	Size	Field	Value / Purpose
0	4	magic	0xDEADBEEF always; mismatch = heap corruption
4	4	size	Usable region size in bytes (excludes header)
8	1	free	1 = available for kmalloc; 0 = in use
9	3	_pad	Alignment padding; always 0
12	4	next	Pointer to next heap_block_t; NULL if last
16	4	prev	Pointer to prev heap_block_t; NULL if first
20	—	data	User-accessible region begins here

Total header: 20 bytes

Pointer arithmetic for user data:

```
/* Get user pointer from block header */

static inline void *block_to_ptr(heap_block_t *b) {

    return (void *)((uint8_t *)b + sizeof(heap_block_t));
}

/* Recover block header from user pointer */

static inline heap_block_t *ptr_to_block(void *p) {

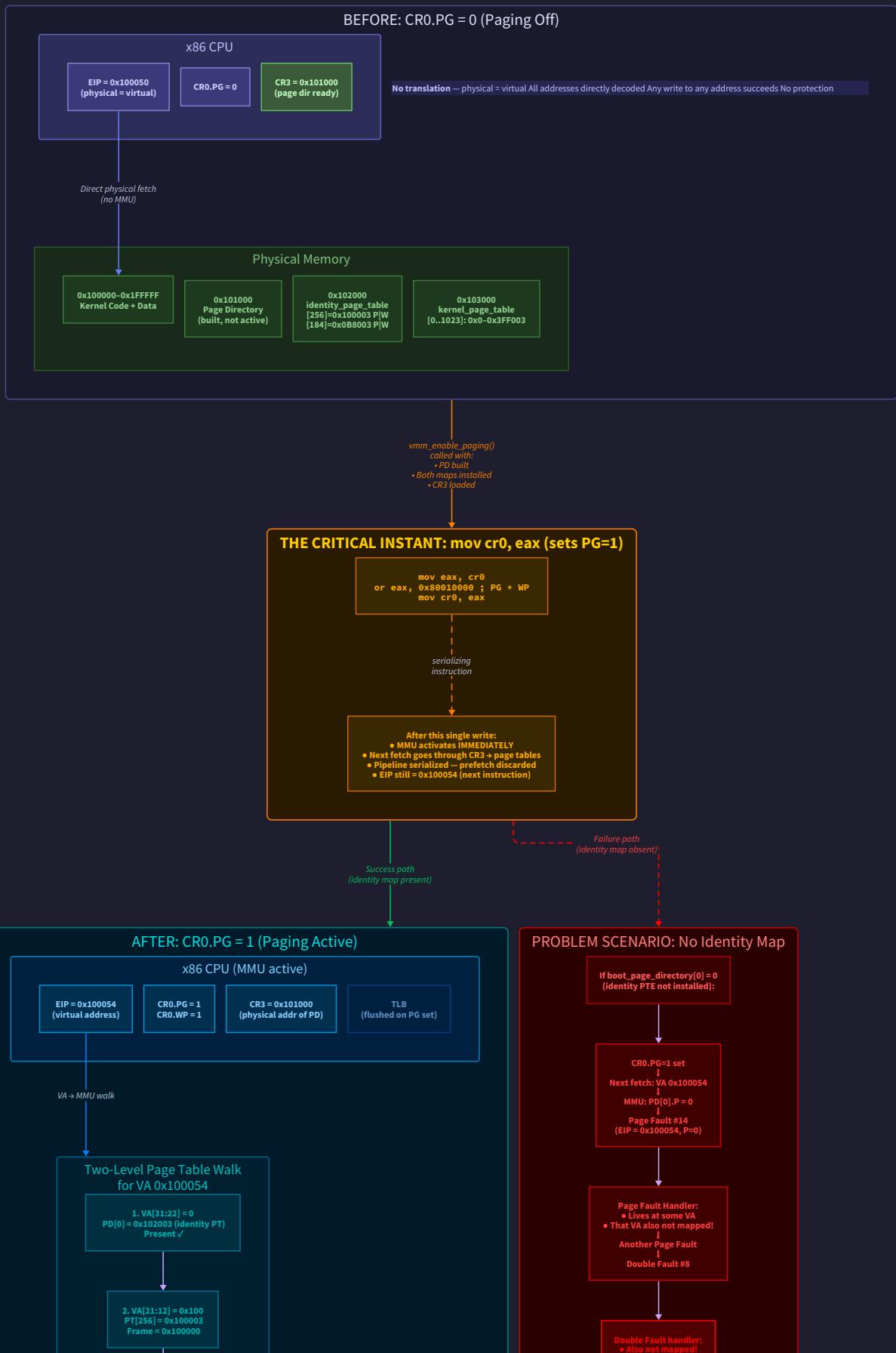
    return (heap_block_t *)((uint8_t *)p - sizeof(heap_block_t));
}
```

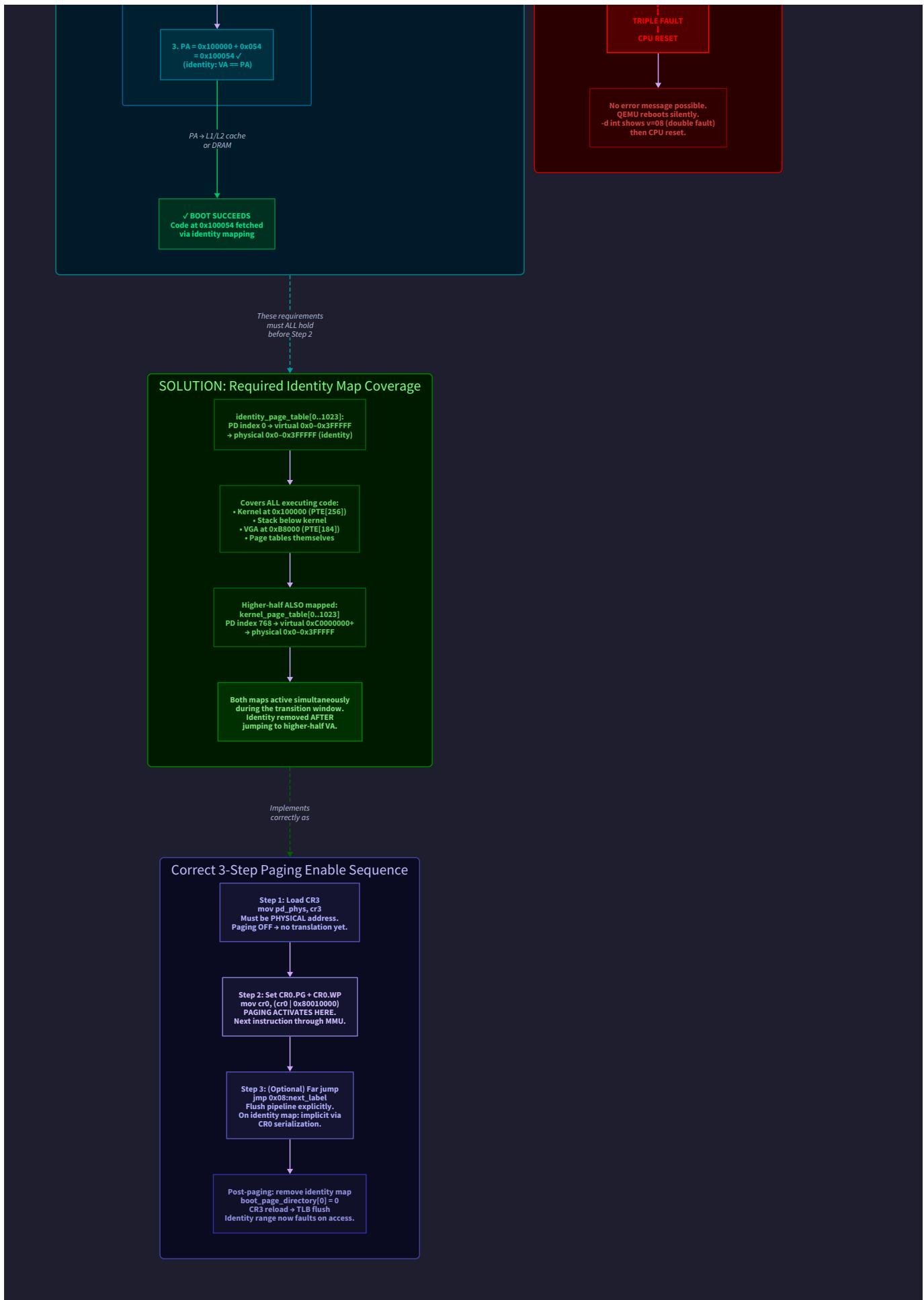
Heap layout in virtual address space:

```
0xD0000000 ← heap_start (HEAP_START)
|--- heap_block_t (20 bytes, magic=0xDEADBEEF, free=1)
|--- [free space: PAGE_SIZE - 20 bytes initially]
|--- heap_block_t (next block after first split)
|--- [user data or free space]
...
0xD0001000 ← heap_end after first heap_expand() (one page mapped)
|   (next heap_expand() maps this page)
...
0xFFFFFFFF ← HEAP_MAX (hard upper limit)
```

Paging Enablement: Identity-Map Bootstrap Problem

The CPU must fetch the next instruction through page tables immediately after





4. Interface Contracts

4.1 pmm_parse_memory_map(multiboot_info_t *mbi) → void

Pre-conditions: `mbi` is a valid pointer to the multiboot information structure, accessible at its physical address (identity map not yet required; this function is called before paging). `mbi->flags & MB_FLAG_MMAP` is nonzero (GRUB always sets this for a kernel with memory requirements).

Behavior:

1. Verify `mbi->flags & MB_FLAG_MMAP`; if zero: `kprintf("[PMM] FATAL: No memory map from bootloader.\n"); for(;;);`.
2. Cast `mbi->mmap_addr` to `mmap_entry_t *` (physical address, valid pre-paging).
3. Iterate entries using `next = (mmap_entry_t*)((uint8_t*)entry + entry->size + 4)` until past `mbi->mmap_addr + mbi->mmap_length`.
4. For each entry: print a line via `kprintf` showing base address (64-bit, printed as two 32-bit halves), end address, and type string.
5. Count and sum total bytes in `E820_USABLE` entries; print summary.

Post-conditions: Memory map printed to VGA and serial. No state modified (this is diagnostic only; `pmm_init()` does the actual initialization).

Errors:

- `MB_FLAG_MMAP` not set: fatal halt.
- Entry with `addr > 0xFFFFFFFF` (above 32-bit range): silently skip — the physical frame allocator only manages up to 4GB.
- Entry where `addr + len > 0xFFFFFFFF`: clamp `len` to `0xFFFFFFFF - addr + 1`.

4.2 pmm_init(multiboot_info_t *mbi) → void

Pre-conditions: Same as `pmm_parse_memory_map`. BSS has been zeroed (so `frame_bitmap` is all-zero). `kprintf` is functional.

Behavior (in exact order):

1. `memset(frame_bitmap, 0xFF, sizeof(frame_bitmap))` — mark all frames used.
2. Set `pmm_used_frames = FRAME_COUNT; pmm_total_frames = 0;`.
3. Iterate E820 map identically to `pmm_parse_memory_map`; for each `E820_USABLE` entry: call `pmm_mark_free(entry->addr, entry->len)`.
4. Re-mark as used: call `pmm_mark_used(0, 0x100000)` — the first 1MB (real-mode IVT, BIOS data area, VGA, BIOS ROM). Physical frame 0 is never allocated; this guarantees NULL-pointer dereferences produce a page fault rather than reading physical RAM.
5. Re-mark as used: the kernel binary. Requires linker symbols:

```
extern uint8_t __kernel_start_phys[]; /* Start of kernel in physical memory */
extern uint8_t __kernel_end_phys[]; /* End of kernel in physical memory */
uint32_t k_start = (uint32_t)(uintptr_t)__kernel_start_phys;
uint32_t k_end   = (uint32_t)(uintptr_t)__kernel_end_phys;
pmm_mark_used(k_start, k_end - k_start);
```

If linker symbols are not defined: use the conservative range `pmm_mark_used(0x100000, 0x400000)` (mark first 4MB used; this over-reserves but prevents accidental frame allocation). 6. Re-mark as used: the `frame_bitmap` array itself.

```
pmm_mark_used((uint64_t)(uintptr_t)frame_bitmap, sizeof(frame_bitmap));
```

C

If `frame_bitmap` lives in `.bss` within the kernel range already marked in step 5, this is a no-op. If the bitmap is large enough to extend beyond the kernel's end, this step is essential. 7. Re-mark as used: the three static page tables (`boot_page_directory`, `identity_page_table`, `kernel_page_table`). These are declared in `vmm.c`; their physical addresses must be passed to `pmm_init` or `pmm_mark_used` must be called from `vmm_init()` after `pmm_init()` returns.

- Implementation choice: `vmm_init()` calls `pmm_mark_used()` for its static page tables after `pmm_init()` returns. This avoids a circular dependency.

8. Set `pmm_search_hint = ADDR_TO_FRAME(0x100000)` — start scanning above the first 1MB.

9. Print `"[PMM] Init: %u free frames, %u total frames (%u MB free)\n"`.

Post-conditions: `frame_bitmap` correctly marks kernel, bitmap, reserved, and low-memory frames as used. Usable frames above reserved regions are marked free. `pmm_alloc_frame()` is operational.

Errors: Out-of-memory (no usable E820 region found): `kprintf("[PMM] FATAL: No usable memory!\n"); for(;;);`.

4.3 `pmm_mark_used(uint64_t addr, uint64_t length) → void`

Behavior: For each 4KB-aligned frame in `[addr, addr+length)`:

- Skip frames $\geq \text{FRAME_COUNT}$ (above 4GB).
- If bit not already set: set it, increment `pmm_used_frames`, decrement `pmm_total_frames` (only if this frame was previously counted as free/total — careful bookkeeping required; see §5.1).

Alignment: Align `addr` down to page boundary; extend `length` to cover the partial page. Equivalently: `frame_start = addr >> PAGE_SHIFT; frame_end = (addr + length + PAGE_SIZE - 1) >> PAGE_SHIFT`.

Edge cases:

- `length == 0`: no-op.
- `addr + length` overflows 64-bit: clamp to `0xFFFFFFFF`.
- Marking an already-used frame: no-op (no double-increment of `pmm_used_frames`).

4.4 `pmm_mark_free(uint64_t addr, uint64_t length) → void`

Behavior: For each 4KB-aligned frame in `[addr, addr+length)`:

- Skip frame 0 (physical address 0 must never be allocatable — preserves NULL-pointer fault semantics).
- Skip frames $\geq \text{FRAME_COUNT}$.
- Align both `addr` up (to avoid freeing a partial page at the start) and `addr+length` down (to avoid freeing a partial page at the end). Use: `base = PAGE_ALIGN_UP(addr); end = PAGE_ALIGN_DOWN(addr + length);`.
- For each frame in `[base, end)`: if bit is set, clear it, increment `pmm_total_frames`, decrement `pmm_used_frames`.

Rationale for aligning up at start and down at end: A partial page at the start might overlap with an in-use region; including it is dangerous. Aligning up is conservative. Same for the partial page at the end.

4.5 `pmm_alloc_frame(void) → void *`

Returns: Physical address of a free 4KB frame, aligned to `PAGE_SIZE`. Returns `NULL` if no free frames remain.

Behavior: See §5.1 (full next-fit algorithm).

Post-conditions: The returned frame's bit is set in `frame_bitmap`. `pmm_used_frames` incremented by 1. The returned address is 4KB-aligned and nonzero. Calling `pmm_alloc_frame()` twice returns two different addresses.

Errors: Returns `NULL` on out-of-memory. Callers that receive `NULL` must propagate the failure — typically halting with a fatal message.

4.6 `pmm_free_frame(void *addr) → void`

Parameters:

- `addr` : Physical address previously returned by `pmm_alloc_frame()`. Must be 4KB-aligned.

Behavior:

1. Compute `frame = (uint32_t)(uintptr_t)addr >> PAGE_SHIFT`.
2. Validate range: if `frame == 0 || frame >= FRAME_COUNT`: `kprintf("[PMM] WARNING: pmm_free_frame(0x%08x) out of range\n", addr); return;`.
3. Double-free detection: if `!bitmap_test(frame)`: `kprintf("[PMM] BUG: Double-free at frame %u (phys 0x%08x)!\n", frame, addr); return;`.
4. `bitmap_clear(frame); pmm_used_frames--;`.
5. If `frame < pmm_search_hint`: `pmm_search_hint = frame` — update hint to enable earlier allocation next scan.

Errors: Out-of-range or double-free: `kprintf` warning, then return without modifying state. No crash. Caller may continue after a free error (though the system is likely in an inconsistent state).

4.7 `vmm_init(void) → void`

Pre-conditions: `pmm_init()` has returned. BSS zeroed (so `boot_page_directory`, `identity_page_table`, `kernel_page_table` are all zero). Paging is NOT yet enabled.

Behavior (in exact order):

Identity page table construction (PD index 0, virtual `0x0 – 0x3FFFFF`):

```
for (int i = 0; i < 1024; i++) {  
    identity_page_table[i] = MAKE_PTE(i * PAGE_SIZE, PAGE_PRESENT | PAGE_WRITABLE);  
}
```

Frame 0 is mapped here (virtual `0x0` → physical `0x0`) to preserve the identity property. The page fault handler will catch any access to virtual `0x0` that should be a null pointer dereference only after paging is on and the kernel verifies that frame 0 is indeed an invalid access target (not mapped for user code; kernel should not access `0x0`).

Higher-half kernel page table construction (PD index 768, virtual `0xC0000000 – 0xC03FFFFF`):

```
for (int i = 0; i < 1024; i++) {  
    kernel_page_table[i] = MAKE_PTE(i * PAGE_SIZE, PAGE_PRESENT | PAGE_WRITABLE);  
}
```

Maps the same physical frames `0x0 – 0x3FFFFF` at virtual `0xC0000000 – 0xC03FFFFF`.

Page directory construction:

```
/* Zero the entire page directory (should already be zero from BSS, but explicit for clarity) */

memset(boot_page_directory, 0, sizeof(boot_page_directory));

/* PD index 0: identity map */

boot_page_directory[0] = MAKE_PDE(
    (uint32_t)(uintptr_t)identity_page_table,
    PAGE_PRESENT | PAGE_WRITABLE
);

/* PD index 768 (0xC0000000 >> 22 = 768): higher-half kernel */

boot_page_directory[768] = MAKE_PDE(
    (uint32_t)(uintptr_t)kernel_page_table,
    PAGE_PRESENT | PAGE_WRITABLE
);

/* All other PD entries remain 0 (not present) */
```

Mark page table frames used in PMM:

```
pmm_mark_used((uint64_t)(uintptr_t)boot_page_directory, PAGE_SIZE);
pmm_mark_used((uint64_t)(uintptr_t)identity_page_table, PAGE_SIZE);
pmm_mark_used((uint64_t)(uintptr_t)kernel_page_table, PAGE_SIZE);
```

Post-conditions: Page directory and both page tables are fully populated. Three frames marked used in PMM. Paging is still disabled — nothing has touched CR3 or CR0.

Errors: None — all data structures are statically allocated and pre-validated by the linker.

4.8 `vmm_enable_paging(void) → void`

Pre-conditions: `vmm_init()` has returned. `pmm_init()` has returned. The identity page table maps the currently executing code (code at physical `0x100000` is mapped at virtual `0x100000` via identity map entry 256).

Behavior (assembly-level, three instructions in precise order):

```

void vmm_enable_paging(void) {
    /* Step 1: Load CR3 with PHYSICAL address of boot_page_directory.

    Pre-paging: physical == virtual for this address (identity map not yet on,
    but the address is correct because we have not moved). */

    uint32_t pd_phys = (uint32_t)(uintptr_t)boot_page_directory;
    __asm__ volatile ("mov %0, %%cr3" : : "r"(pd_phys) : "memory");

    /* Step 2: Set CR0.PG (bit 31) and CR0.WP (bit 16).

    WP = Write Protect: prevents kernel (ring 0) from writing to read-only user pages.

    Setting WP enables future copy-on-write protection.

    PAGING IS ENABLED AFTER THIS WRITE. The next instruction fetch goes through
    page tables. The identity map ensures this succeeds. */

    uint32_t cr0;

    __asm__ volatile ("mov %%cr0, %0" : "=r"(cr0));
    cr0 |= (1u << 31); /* CR0.PG */
    cr0 |= (1u << 16); /* CR0.WP */
    __asm__ volatile ("mov %0, %%cr0" : : "r"(cr0) : "memory");

    /* Step 3: Serialize – ensure pipeline sees paging mode.

    On x86, writing CR0 is a serializing operation (all prior instructions
    complete; instruction cache is flushed). No explicit jmp needed on
    identity-mapped code, but a jmp or cpuid may be added for strictness. */

    kprintf("[VMM] Paging enabled. CR3=0x%08x CR0.PG=1\n", pd_phys);
}

```

Why CR3 before CR0.PG: CR3 must contain a valid page directory before the paging enable bit is set. Setting CR0.PG with CR3=0 would cause the CPU to attempt a page table walk from physical address 0, which is the real-mode IVT — not a valid page directory. The immediate page fault from the bad walk, followed by the page fault handler's own bad walk, would produce a triple fault.

Post-conditions: CR0.PG = 1 . CR3 = physical address of boot_page_directory . All subsequent memory accesses go through the two-level page table. The identity map makes this transparent for addresses $\leq 0x3FFFFF$. VGA at physical 0xB8000 is accessible at virtual 0xB8000 (within the identity-mapped first 4MB).

Errors: If boot_page_directory is not 4KB-aligned (miscompiled kernel or wrong linker script): CR3 will have garbage in its low 12 bits. The CPU ignores the low 12 bits of CR3 (it masks them to zero for page directory lookup), so this is silently tolerated on x86 — but it is a latent bug. Assert alignment with a compile-time check: `_Static_assert(alignof(boot_page_directory) >= PAGE_SIZE, ...)` — not possible for static arrays; instead, verify in vmm_init() at runtime:

```

if ((uint32_t)(uintptr_t)boot_page_directory & (PAGE_SIZE - 1)) {
    kprintf("[VMM] FATAL: boot_page_directory not 4KB aligned!\n");
    for(;;);
}

```

4.9 vmm_map_page(pde_t *page_directory, uint32_t virt_addr, uint32_t phys_addr, uint32_t flags) → void

Pre-conditions: Paging is enabled. `page_directory` is accessible at its virtual address (either via identity map or recursive mapping — in this module, always via identity map since `boot_page_directory` is in the identity-mapped region).

Parameters:

- `page_directory` : Pointer to the page directory (1024 `pde_t` entries). Must be 4KB-aligned. For all kernel operations in this module: pass `boot_page_directory`.
- `virt_addr` : Virtual address to map. Must be 4KB-aligned. If not aligned: align down silently (`virt_addr &= ~(PAGE_SIZE - 1)`).
- `phys_addr` : Physical frame address. Must be 4KB-aligned. If not aligned: align down silently.
- `flags` : Combination of `PAGE_*` constants. Must include `PAGE_PRESENT`. The `PAGE_PRESENT` bit is OR'd in unconditionally even if the caller omits it.

Behavior:

1. Compute `dir_idx = VA_DIR_IDX(virt_addr)`, `tbl_idx = VA_TBL_IDX(virt_addr)`.
2. If `page_directory[dir_idx]` has `PAGE_PRESENT` clear: allocate a new page table frame via `pmm_alloc_frame()`. If `pmm_alloc_frame()` returns `NULL`: `kprintf("[VMM] FATAL: vmm_map_page OOM\n"); for(;;);`. Zero the new frame (it is physical RAM; must be zeroed before use as a page table). Install the new frame as `page_directory[dir_idx] = MAKE_PDE(new_frame, PAGE_PRESENT | PAGE_WRITABLE | (flags & PAGE_USER))`. Flush TLB for the page directory entry: `tlb_flush_page((uint32_t)&page_directory[dir_idx])` — strictly, PDE entries are not in the TLB directly; flushing the virtual address covered by the new PDE is sufficient.
3. Extract the page table physical address: `pte_t *page_table = (pte_t *)PDE_FRAME(page_directory[dir_idx])`. **Note:** this uses the physical address as a virtual address — valid only while the identity map is active or while a recursive mapping exists. For this module: identity map is always active for addresses below `0x400000`, and all dynamically allocated page tables are in frames above `0x100000` but below `0x400000` (the identity-mapped region), so this dereference is valid.
4. `page_table[tbl_idx] = MAKE_PTE(phys_addr, flags | PAGE_PRESENT)`.
5. `tlb_flush_page(virt_addr)` — invalidate the TLB entry for this virtual page.

Post-conditions: Virtual `virt_addr` translates to physical `phys_addr` with the specified flags. Any access to `virt_addr` through any valid page succeeds (or faults as specified by `flags`).

Errors: OOM on page table allocation: fatal halt (cannot continue without page tables).

4.10 vmm_unmap_page(pde_t *page_directory, uint32_t virt_addr) → void

Parameters:

- `page_directory` : Same as `vmm_map_page`.
- `virt_addr` : Virtual address to unmap. Aligned down if not 4KB-aligned.

Behavior:

1. Compute `dir_idx`, `tbl_idx` from `virt_addr`.

2. If `page_directory[dir_idx]` is not present: return (already unmapped; no-op).
3. Extract page table pointer (same identity-map assumption as `vmm_map_page`).
4. `page_table[tbl_idx] = 0` — clear the entire PTE (all flags cleared, including PRESENT).
5. `tlb_flush_page(virt_addr)` — **MANDATORY**. If omitted, the old translation remains in TLB and future accesses use the stale (now-freed) physical frame.

Post-conditions: Virtual `virt_addr` is not mapped. Next access causes page fault (vector 14, error code with P=0).

Errors: Unmapping an already-unmapped page: no-op (idempotent).

4.11 `tlb_flush_page(uint32_t virt_addr) → void (inline)`

```
/* include/vmm.h */

static inline void tlb_flush_page(uint32_t virt_addr) {

    /* invlpg: Invalidate TLB entry for the page containing virt_addr.

       Operand is a memory address (register-indirect operand required). */

    __asm__ volatile ("invlpg (%0)" : : "r"(virt_addr) : "memory");

}
```

Cost: ~20 cycles on modern Intel CPUs. Does not flush global (PAGE_GLOBAL) entries.

4.12 `tlb_flush_all(void) → void (inline)`

```
static inline void tlb_flush_all(void) {

    /* Writing CR3 with its current value flushes all non-global TLB entries. */

    uint32_t cr3;

    __asm__ volatile (
        "mov %%cr3, %0\n\t"
        "mov %0, %%cr3"
        : "=r"(cr3) : : "memory"

    );
}
```

Cost: ~30 cycles for the reload plus subsequent cold TLB misses. Use when multiple PTEs have been modified (heap expansion that maps many pages at once) to avoid calling `invlpg` N times.

4.13 Page Fault Handler — Integration with mod-2 IDT

The page fault handler in this module replaces the `default_isr` at vector 14 installed by `idt_setup_all()` in mod-2. It is registered via `idt_set_gate(14, (uint32_t)isr14, 0x08, IDT_TRAP_GATE)` — already done by `idt_setup_all()`. The C function `exception_handler()` in `interrupt.c` dispatches to the page fault logic when `frame->vector == 14`. This module does not require changes to the assembly stubs; only the C handler behavior needs to be specified.

See §5.3 for the full page fault handling algorithm. The relevant changes in `interrupt.c` for mod-3: replace the `frame->vector == 14` comment stub from mod-2 with the full CR2-read + decode + print behavior.

4.14 `heap_init(void) → void`

Pre-conditions: Paging is enabled. `vmm_map_page()` is operational. `pmm_alloc_frame()` is operational.

Behavior:

1. Set `heap_start = HEAP_START; heap_end = HEAP_START;`.
2. Call `heap_expand()` — maps the first physical frame to `HEAP_START`, advances `heap_end` to `HEAP_START + PAGE_SIZE`.
3. Initialize the first `heap_block_t` at virtual address `HEAP_START`:

```
heap_block_t *first = (heap_block_t *)(uintptr_t)HEAP_START;  
  
first->magic = HEAP_MAGIC;  
  
first->size = PAGE_SIZE - sizeof(heap_block_t);  
  
first->free = 1;  
  
first->next = NULL;  
  
first->prev = NULL;  
  
heap_head = first;
```

4. `kprintf("[HEAP] Initialized at 0x%llx, first page mapped, %u bytes usable.\n", HEAP_START, first->size)`.

Post-conditions: `heap_head` points to a single free block of `PAGE_SIZE - sizeof(heap_block_t)` bytes. `heap_end = HEAP_START + PAGE_SIZE`. One physical frame used for heap.

4.15 `kmalloc(size_t size) → void *`

Returns: Pointer to at least `size` bytes of kernel memory, 8-byte aligned. Returns `NULL` only if physical frames are exhausted.

Parameters:

- `size` : Requested allocation size in bytes. `size == 0` : returns `NULL` immediately (not a valid allocation).

Full behavior: See §5.4.

Errors: OOM (all physical frames exhausted and heap has reached `HEAP_MAX`): returns `NULL`. Callers must check for `NULL`.

4.16 `kfree(void *ptr) → void`

Parameters:

- `ptr` : Pointer previously returned by `kmalloc()`. `ptr == NULL` : returns immediately (no-op).

Behavior: See §5.5.

Errors:

- `ptr == NULL` : no-op.
- Corrupted magic (`block->magic != HEAP_MAGIC`): `kprintf("[HEAP] BUG: kfree(0x%llx) bad magic (got 0x%llx)!\n", ptr, block->magic); return;` — does not crash.
- Double-free (block already free): `kprintf("[HEAP] BUG: Double-free at 0x%llx!\n", ptr); return;`.

5. Algorithm Specification

5.1 `pmm_alloc_frame()` — Next-Fit Bitmap Scan

Input: `frame_bitmap`, `pmm_search_hint` (frame number where scan starts), `pmm_used_frames`.

Output: Physical address of an allocated frame, or `NULL`.

Algorithm:

```
Step 1: Check total free count
    If pmm_frames_free() == 0: return NULL.

Step 2: Determine starting word in bitmap
    start_word = pmm_search_hint / 32
    If start_word >= BITMAP_WORDS: start_word = 0 (wrap to beginning)

Step 3: Two-phase scan (scan from hint to end, then from beginning to hint)
    scanned = 0
    word = start_word
    LOOP:
        If frame_bitmap[word] == 0xFFFFFFFF:
            word++; if word >= BITMAP_WORDS: word = 0;
            scanned += 32;
            if scanned >= FRAME_COUNT: return NULL; (full scan, nothing free)
            continue;

        // At least one free bit in this word. Find it.
        For bit b from 0 to 31:
            frame = word * 32 + b
            If frame == 0: continue (never allocate frame 0)
            If frame >= FRAME_COUNT: return NULL
            If NOT bitmap_test(frame):
                bitmap_set(frame)
                pmm_used_frames++
                pmm_search_hint = frame (update hint for next call)
                return (void*)(uintptr_t)FRAME_TO_ADDR(frame)

        // Should not reach here (word has a free bit, but we did not find it)
        word++; if word >= BITMAP_WORDS: word = 0;
        scanned += 32;
        if scanned >= FRAME_COUNT: return NULL;

Step 4: (Unreachable if logic is correct)
return NULL;
```

Inline bit-test and bit-set helpers:

```
static inline int bitmap_test(uint32_t frame) {
    return (frame_bitmap[frame / 32] >> (frame % 32)) & 1;
}

static inline void bitmap_set(uint32_t frame) {
    frame_bitmap[frame / 32] |= (1u << (frame % 32));
}

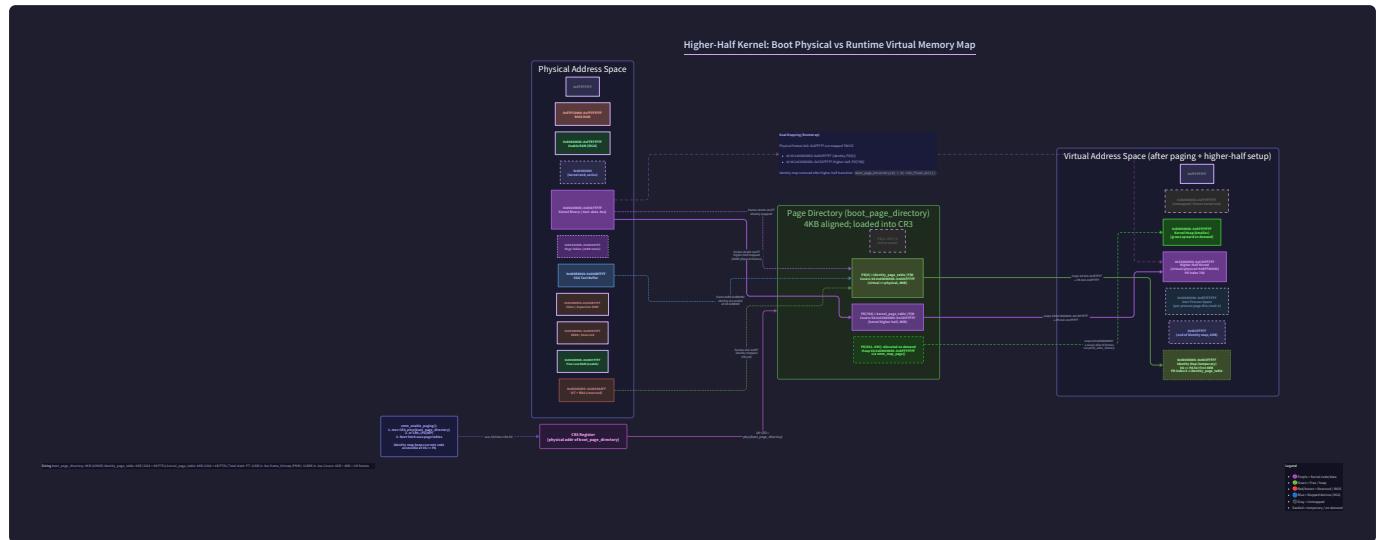
static inline void bitmap_clear(uint32_t frame) {
    frame_bitmap[frame / 32] &= ~(1u << (frame % 32));
}
```

2

Optimization — word-level skip: The check `if (frame_bitmap[word] == 0xFFFFFFFF)` continues skips fully-used words in a single comparison, processing 32 frames in 1 cycle rather than 32 iterations. This is the key cache-efficiency optimization: on a typical boot with < 25% memory used, most bitmap words are `0x00000000` (all free), and even those are skipped in one comparison. Only words with a mix of 0 and 1 bits require the inner bit loop.

Invariants after execution:

- The returned frame's bit is 1 in `frame_bitmap`.
 - `pmm_used_frames` is incremented by exactly 1.
 - `pmm_search_hint` points to or near the allocated frame.
 - The returned address is never `0` (physical address 0 is never returned).



5.2 vmm_init() — Page Table Construction Sequence

Input: Physical address of `boot_page_directory`, `identity_page_table`, `kernel_page_table` (all statically allocated in `.bss`, aligned to `PAGE_SIZE`).

Step-by-step with invariant checks:

```

Step 1: Zero the page directory (defensive – BSS already zeroed, but explicit for clarity)
memset(boot_page_directory, 0, 1024 * sizeof(pde_t));

Step 2: Populate identity_page_table[0..1023]
For i = 0 to 1023:
    physical_frame = i * PAGE_SIZE   (frame i: 0x0, 0x1000, 0x2000, ..., 0x3FF000)
    identity_page_table[i] = physical_frame | PAGE_PRESENT | PAGE_WRITABLE
// Covers virtual 0x00000000 - 0x003FFFFF (first 4MB)
// Includes: real-mode area (0x0), kernel stack, kernel code (0x100000), VGA (0xB8000)

Step 3: Populate kernel_page_table[0..1023]
For i = 0 to 1023:
    kernel_page_table[i] = (i * PAGE_SIZE) | PAGE_PRESENT | PAGE_WRITABLE
// Identical to identity_page_table in content – same physical frames,
// different virtual addresses after page directory installation

Step 4: Install identity PDE at directory index 0
boot_page_directory[0] = (uint32_t)identity_page_table | PAGE_PRESENT | PAGE_WRITABLE
// Maps virtual 0x00000000-0x003FFFFF → identity_page_table (physical)

Step 5: Install higher-half PDE at directory index 768
boot_page_directory[768] = (uint32_t)kernel_page_table | PAGE_PRESENT | PAGE_WRITABLE
// Maps virtual 0xC0000000-0xC03FFFFF → kernel_page_table (physical)
// 768 = 0xC0000000 >> 22 (verified: 0xC0000000 / 0x400000 = 768)

Step 6: Verify runtime alignment (defensive)
Assert (uint32_t)boot_page_directory & 0xFFF == 0
Assert (uint32_t)identity_page_table & 0xFFF == 0
Assert (uint32_t)kernel_page_table & 0xFFF == 0
If any assertion fails: kprintf fatal halt

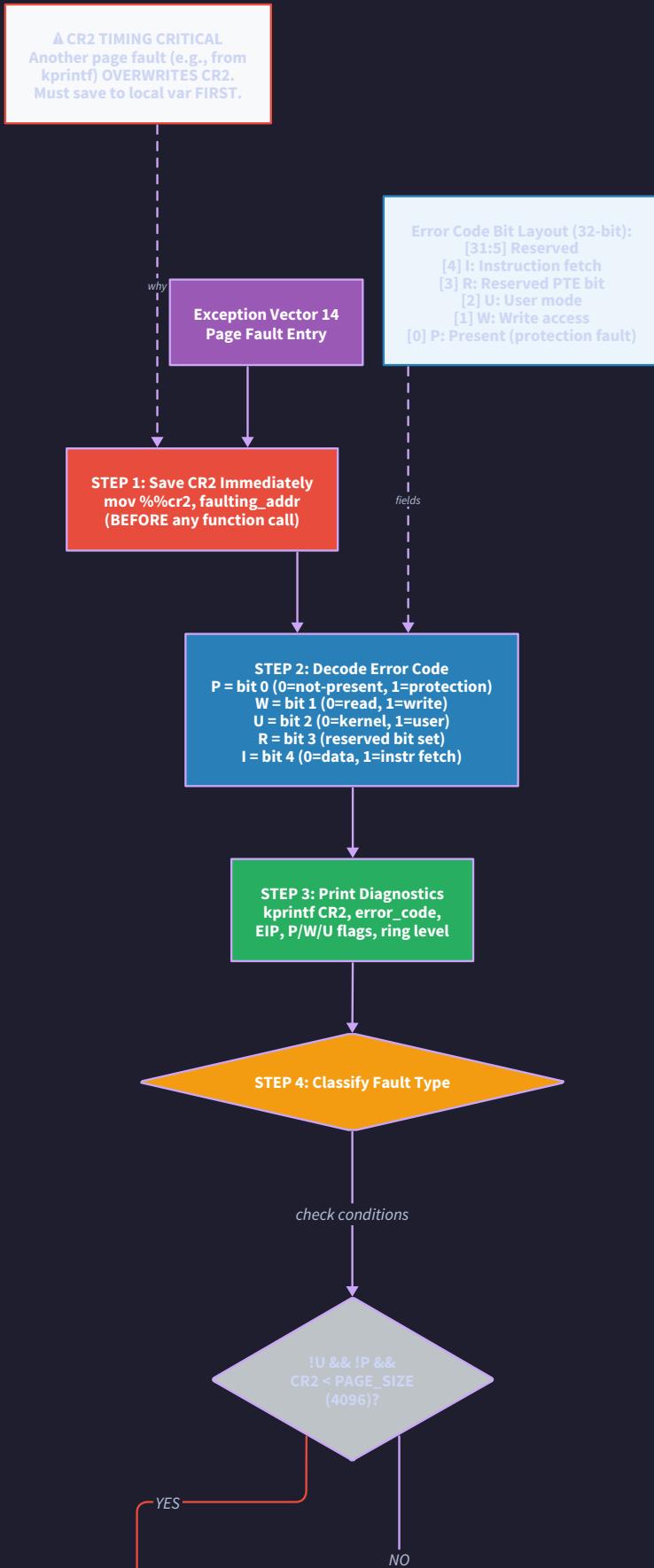
Step 7: Mark the three page-table frames as used in PMM
pmm_mark_used((uint64_t)(uintptr_t)boot_page_directory, PAGE_SIZE);
pmm_mark_used((uint64_t)(uintptr_t)identity_page_table, PAGE_SIZE);
pmm_mark_used((uint64_t)(uintptr_t)kernel_page_table, PAGE_SIZE);

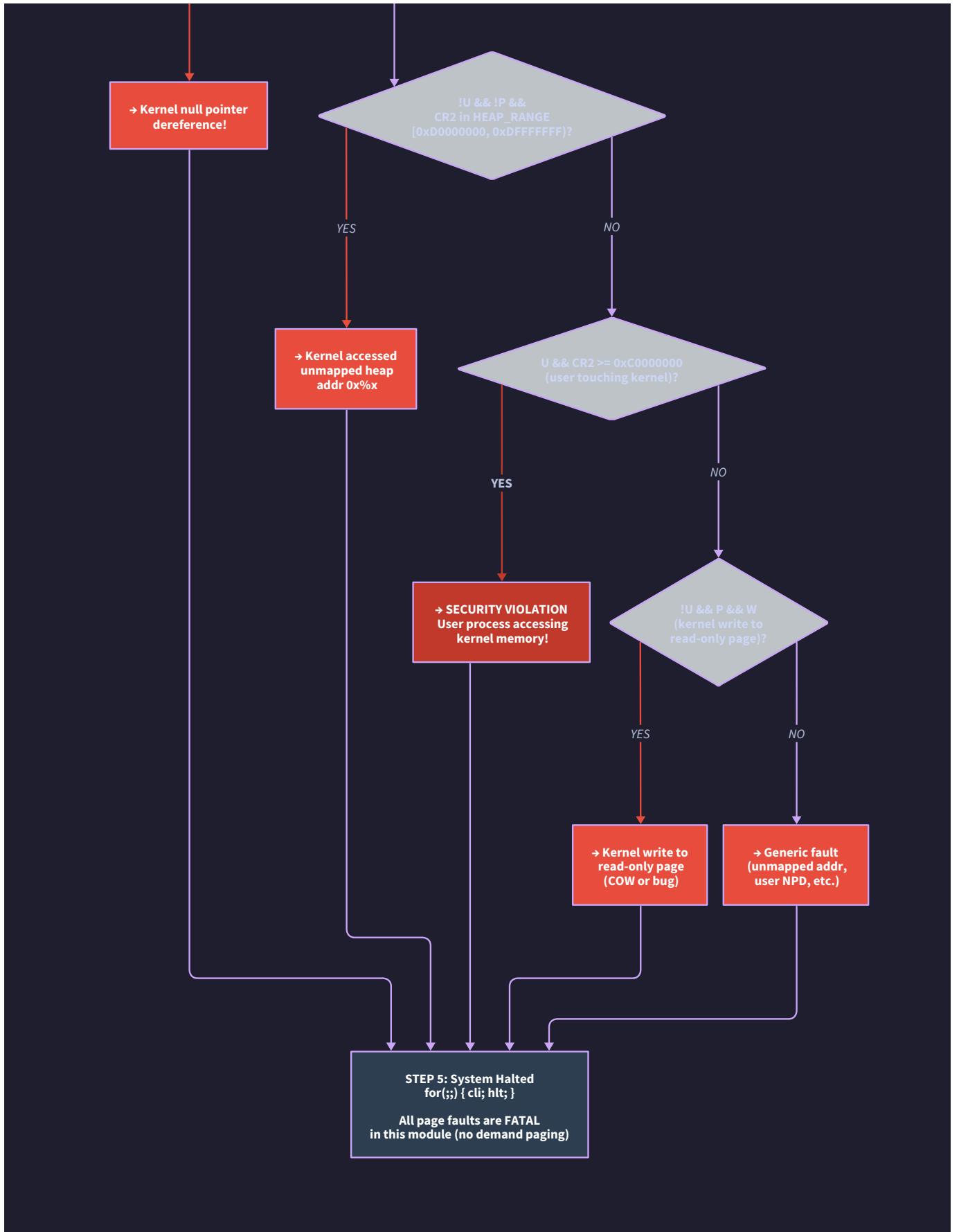
```

Invariants after execution:

- `boot_page_directory[0]` is present and points to `identity_page_table`.
- `boot_page_directory[768]` is present and points to `kernel_page_table`.
- All other PD entries are 0 (not present).
- Three frames accounted for in PMM.
- Paging NOT yet enabled; this is pure data structure initialization.

Page Fault Handler: CR2 Decode and Error Code Decision Tree





5.3 Page Fault Handler — CR2 Decode and Decision Tree

Trigger: CPU exception vector 14 fires. Called via `exception_handler()` in `interrupt.c`.

Input: `interrupt_frame_t *frame` with `frame->vector == 14`, `frame->error_code` = page fault error code, `frame->eip` = faulting instruction.

Step-by-step:

```

Step 1: Read CR2 IMMEDIATELY (first line of vector-14 branch)
uint32_t faulting_addr;
__asm__ volatile ("mov %%cr2, %0" : "=r"(faulting_addr));
// CR2 is volatile: another page fault (from kprintf itself)
// would overwrite CR2. Save it to a local variable before any function call.

Step 2: Decode error code bits
int pf_present = frame->error_code & 0x1; // P: 0=not present, 1=protection violation
int pf_write = frame->error_code & 0x2; // W: 0=read, 1=write
int pf_user = frame->error_code & 0x4; // U: 0=supervisor, 1=user mode
int pf_reserved = frame->error_code & 0x8; // R: reserved bit set in PTE (hardware bug)
int pf_fetch = frame->error_code & 0x10; // I: 0=data, 1=instruction fetch

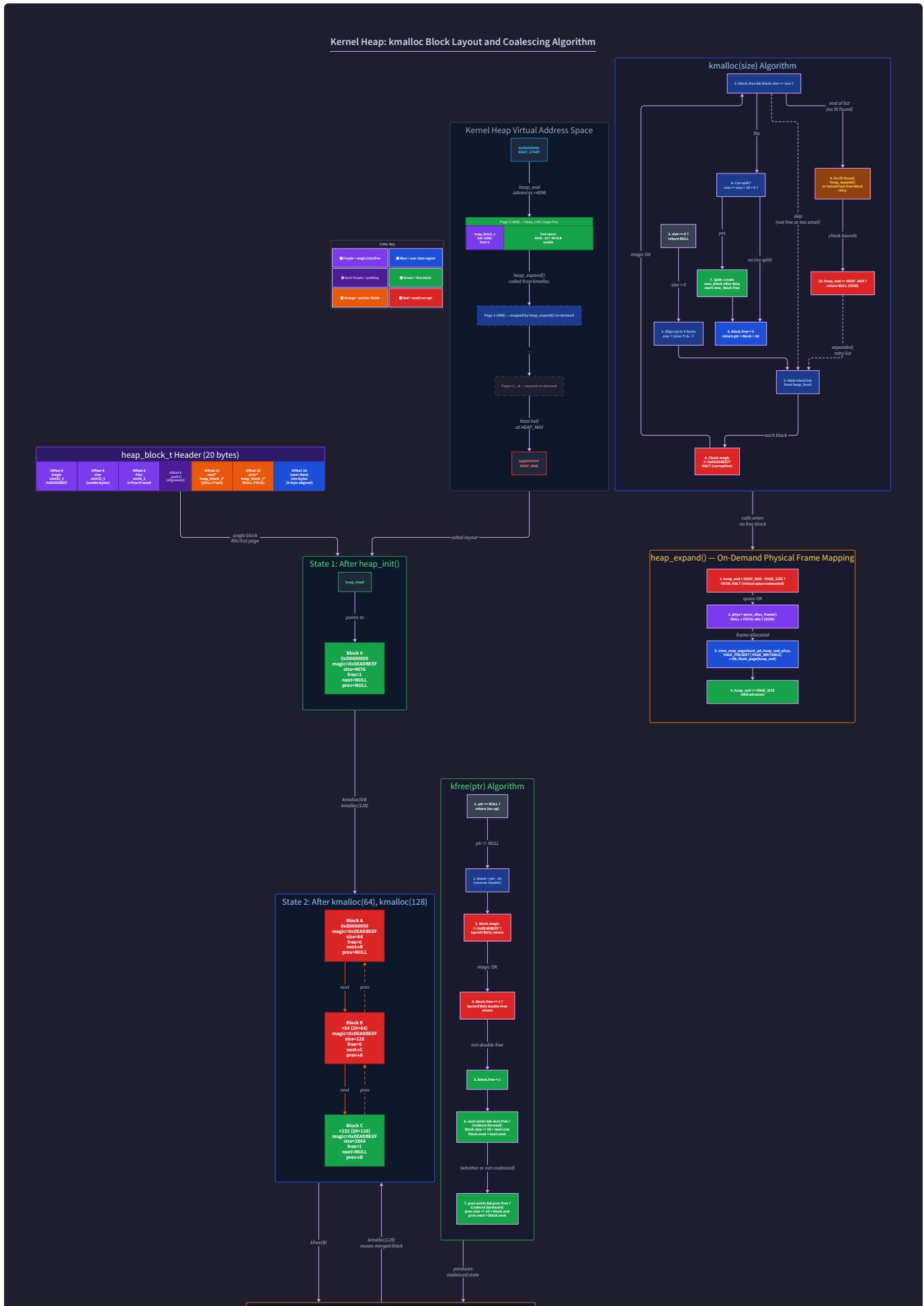
Step 3: Print diagnostic
kprintf("\n==== PAGE FAULT ====\n");
kprintf("Faulting address (CR2): 0x%08x\n", faulting_addr);
kprintf("Error code: 0x%08x\n", frame->error_code);
kprintf(" P=%d (page %s) W=%d (%s) U=%d (%s)%s\n",
    pf_present, pf_present ? "protection-violation" : "not-present",
    pf_write, pf_write ? "write" : "read",
    pf_user, pf_user ? "user-mode" : "kernel-mode",
    pf_reserved ? " R=1(RESERVED BIT)" : "");
kprintf("Fault at EIP: 0x%08x\n", frame->eip);

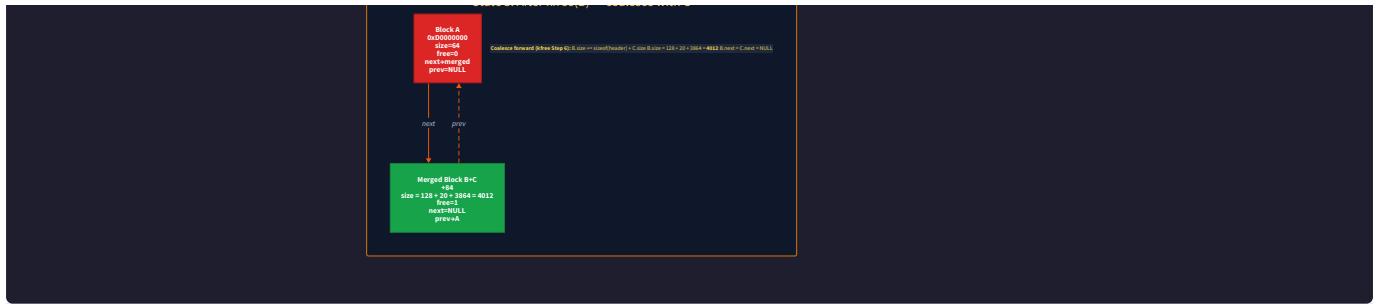
Step 4: Classify and provide diagnostic guidance
if (!pf_user && !pf_present && faulting_addr < PAGE_SIZE):
    kprintf(" → Kernel null pointer dereference!\n");
else if (!pf_user && !pf_present && faulting_addr >= HEAP_START && faulting_addr < HEAP_MAX):
    kprintf(" → Kernel accessed unmapped heap address 0x%08x\n", faulting_addr);
else if (pf_user && faulting_addr >= 0xC0000000):
    kprintf(" → SECURITY: User process attempted kernel memory access!\n");
else if (!pf_user && pf_present && pf_write):
    kprintf(" → Kernel write to read-only page (COW candidate or kernel bug)\n");

Step 5: Halt (all page faults are fatal in this module; demand paging is mod-4+)
kprintf("System halted.\n");
for (;;) { __asm__ volatile ("cli; hlt"); }

```

Why CR2 must be saved before any function call: `kprintf` internally calls `vga_putchar` which writes to `0xB8000`. If paging is active and `0xB8000` is mapped (it is, via identity map), this is safe. However, if `kprintf`'s own execution triggers a page fault (e.g., a bad format string pointer), the CPU overwrites CR2 with the new faulting address before invoking the fault handler again — producing a double fault. Saving `faulting_addr` to a local variable before any function call is the only safe pattern.





5.4 `kmalloc()` — First-Fit with Split

Input: `size` (requested bytes), `heap_head` (first block in linked list), `heap_end` (current end of mapped heap).

Algorithm:

```

Step 1: Input validation
If size == 0: return NULL
size = (size + HEAP_ALIGN - 1) & ~(HEAP_ALIGN - 1) // Round up to 8-byte alignment

Step 2: Walk the block list looking for a free block that fits
block = heap_head
LOOP while block != NULL:

    Step 2a: Validate magic (detect corruption early)
    If block->magic != HEAP_MAGIC:
        kprintf("[HEAP] CORRUPTION at 0x%llx: bad magic 0x%llx\n", block, block->magic);
        for(;;); // Halt – heap is corrupt; continuing risks memory safety

    Step 2b: Skip allocated blocks
    If !block->free: block = block->next; continue

    Step 2c: Check if block is large enough
    If block->size < size: block = block->next; continue

    Step 2d: Split the block if there is room for a header + minimum payload
    min_split_size = sizeof(heap_block_t) + HEAP_ALIGN
    If block->size >= size + min_split_size:
        Create a new block immediately after the allocated region:
        new_block = (heap_block_t*)((uint8_t*)block + sizeof(heap_block_t) + size)
        new_block->magic = HEAP_MAGIC
        new_block->size = block->size - size - sizeof(heap_block_t)
        new_block->free = 1
        new_block->next = block->next
        new_block->prev = block
        If block->next != NULL: block->next->prev = new_block
        block->next = new_block
        block->size = size

    Step 2e: Mark block as allocated and return
    block->free = 0
    return (void*)((uint8_t*)block + sizeof(heap_block_t))

Step 3: No suitable block found – expand the heap
If heap_end >= HEAP_MAX: return NULL (virtual heap exhausted)
heap_expand() // Maps one more physical frame at heap_end, advances heap_end

Try to coalesce with the last block if it was free:
last_block = find last block in list (block->next == NULL)
If last_block->free:
    last_block->size += PAGE_SIZE // Extend the last free block into new page
    goto Step 2 (retry with extended last block)
Else:
    Create a new block in the newly mapped page:
    new_block = (heap_block_t*)(uintptr_t)(heap_end - PAGE_SIZE)
    new_block->magic = HEAP_MAGIC
    new_block->size = PAGE_SIZE - sizeof(heap_block_t)
    new_block->free = 1
    new_block->next = NULL
    new_block->prev = last_block
    last_block->next = new_block
    goto Step 2 (retry; will find new_block)

```

Invariants after each split:

- `block->size + new_block->size + sizeof(heap_block_t) == original_block->size` (no bytes lost).
- Both blocks have valid magic values.
- The linked list is doubly-consistent: `block->next == new_block` and `new_block->prev == block`.

Why minimum split size is `sizeof(heap_block_t) + HEAP_ALIGN`: If the remainder after splitting would be smaller than a header plus minimum payload, the split produces an unusable block that can never be allocated. In that case, do not split — give the caller the slightly larger block (internal fragmentation of at most `sizeof(heap_block_t) + HEAP_ALIGN - 1` bytes).

5.5 `kfree()` — Coalescing Free

Input: `ptr` (user pointer), linked list of blocks.

Algorithm:

```
Step 1: Handle NULL
If ptr == NULL: return

Step 2: Recover block header
block = (heap_block_t*)((uint8_t*)ptr - sizeof(heap_block_t))

Step 3: Validate magic
If block->magic != HEAP_MAGIC:
    kprintf("[HEAP] BUG: kfree(0x%llx) bad magic (0x%llx)!\n", ptr, block->magic);
    return // Do not modify state; heap may be reusable

Step 4: Double-free detection
If block->free == 1:
    kprintf("[HEAP] BUG: Double-free at 0x%llx!\n", ptr);
    return

Step 5: Mark block as free
block->free = 1

Step 6: Coalesce with NEXT block (merge forward)
If block->next != NULL AND block->next->free == 1:
    Validate block->next->magic == HEAP_MAGIC (otherwise skip coalesce: corrupted neighbor)
    block->size += sizeof(heap_block_t) + block->next->size
    block->next = block->next->next
    If block->next != NULL: block->next->prev = block

Step 7: Coalesce with PREV block (merge backward)
If block->prev != NULL AND block->prev->free == 1:
    Validate block->prev->magic == HEAP_MAGIC (otherwise skip)
    block->prev->size += sizeof(heap_block_t) + block->size
    block->prev->next = block->next
    If block->next != NULL: block->next->prev = block->prev
    // block is now absorbed into block->prev; do not use block pointer after this
```

Invariants after coalescing:

- No two adjacent free blocks exist in the list (fully coalesced).
- All remaining blocks have valid magic values.
- The doubly-linked list is consistent: for every block B with `B->next = C`, `C->prev == B`.

Why coalesce next before prev: If we coalesce prev first, we lose the pointer to `block` (it gets absorbed into `block->prev`). Coalescing next first keeps `block` valid through both steps.

5.6 `heap_expand()` — On-Demand Physical Frame Mapping

Called from: `kmalloc()` when the block list has no suitable free block.

Algorithm:

```

Step 1: Check heap bounds
If heap_end > HEAP_MAX - PAGE_SIZE:
    kprintf("[HEAP] FATAL: Heap virtual space exhausted at 0x%xx\n", heap_end);
    for(;;);

Step 2: Allocate a physical frame
void *phys = pmm_alloc_frame();
If phys == NULL:
    kprintf("[HEAP] FATAL: pmm_alloc_frame() returned NULL during heap expansion\n");
    for(;;);
    // Alternatively: return NULL and propagate to kmalloc → caller.
    // For this module: halt, because OOM during heap expansion is unrecoverable.

Step 3: Map the frame at heap_end
vmm_map_page(boot_page_directory, heap_end, (uint32_t)(uintptr_t)phys,
             PAGE_PRESENT | PAGE_WRITABLE);
// TLB flush for this address is performed inside vmm_map_page.

Step 4: Advance heap_end
heap_end += PAGE_SIZE;
// heap_end now points to the first byte past the newly mapped region.

```

Invariants after execution:

- Virtual addresses `[old_heap_end, old_heap_end + PAGE_SIZE]` are mapped and writable.
 - One additional physical frame is marked used in PMM.
 - `heap_end` advanced by exactly `PAGE_SIZE`.
-

6. Error Handling Matrix

Error	Detected By	Recovery	User-Visible?
Triple fault: CR0.PG set without identity map	CPU triple-faults immediately (next fetch page-faults; page fault handler page-faults; double fault; triple fault)	No recovery. Prevention: <code>identity_page_table</code> must cover currently executing code before <code>vmm_enable_paging()</code> . Verify: code at <code>0x100000</code> is covered by <code>identity_page_table[256]</code> (frame 256 = physical <code>0x100000</code>).	QEMU reboots
Stale TLB: PTE modified without <code>invlpg</code>	Silent wrong-physical-frame access; data corruption or wrong permissions enforced	No immediate crash; corruption discovered later (data read from wrong page, write goes to wrong frame). Prevention: call <code>tlb_flush_page()</code> in every PTE modification path. Detected by: adding test that writes to a freshly mapped page and reads back via a different path.	Silent corruption; may crash later
PMM allocates kernel frame	Kernel code or data overwritten by heap allocation; crash at unpredictable time	No recovery once occurred. Prevention: <code>pmm_mark_used</code> for kernel range in <code>pmm_init()</code> . Detected by: boot-time print of "used frames" should show kernel size accounted for.	Eventually crashes
Page table frame not marked used in PMM	PMM gives out the frame holding <code>identity_page_table</code> ; subsequent write to that frame corrupts the page table; next memory access uses garbage translation; crash	No recovery. Prevention: <code>vmm_init()</code> calls <code>pmm_mark_used()</code> for all three page table frames.	QEMU reboot or silent corruption
CR2 overwritten before save	kprintf inside page fault handler itself causes a page fault; CR2 updated to new address; handler prints wrong faulting address	Wrong address printed but halt still occurs. Prevention: save CR2 to local variable as the absolute first operation in the vector-14 branch.	Wrong diagnostic info
Double-free: <code>pmm_free_frame</code> called twice for same frame	<code>bitmap_test(frame)</code> returns 0 (bit already clear) before <code>bitmap_clear</code> . Detected by the pre-clear check.	<code>kprintf</code> warning; return without modifying state. Frame remains free. Caller continues (may be in inconsistent state).	<code>kprintf</code> message
Double-free: <code>kfree</code> called twice for same pointer	<code>block->free == 1</code> check in Step 4 of kfree algorithm.	<code>kprintf</code> bug message; return without modifying state. Heap remains usable.	<code>kprintf</code> message
Heap magic corruption	<code>kmalloc</code> or <code>kfree</code> checks <code>block->magic != HEAP_MAGIC</code> . Caused by buffer overflow beyond an allocation writing into the next block's header.	<code>kmalloc</code> : halt (cannot safely allocate from corrupt heap). <code>kfree</code> : return without coalescing (safe; leaks memory, but avoids corrupting adjacent block pointers).	<code>kprintf</code> corruption message; possible halt
<code>pmm_parse_memory_map</code> : <code>MB_FLAG_MMAP</code> not set	<code>!(mbi->flags & MB_FLAG_MMAP)</code> check at start of function.	Fatal halt with message. This should never happen when booting via GRUB.	<code>kprintf</code> fatal message
Multiboot iterator off-by-4 (<code>size + 0</code> instead of <code>size +</code>)	Iterator produces wrong addresses; reads garbage E820 entries; type	May cause no usable frames to be found (all marked used) or may loop past the end of the	Wrong memory

Error	Detected By	Recovery	User-Visible?
4)	field contains wrong value	mmap buffer. Detection: boot-time output shows 0 usable MB or QEMU segfault-equivalent.	availability or hang
<code>vmm_map_page</code> with non-4KB-aligned <code>virt_addr</code>	Alignment masked down silently. The caller's intended virtual address may differ by up to 4095 bytes from what was actually mapped.	No crash; wrong virtual address mapped. Detected by: test that maps a specific address and verifies the translation.	Silent wrong mapping
<code>pmm_alloc_frame</code> returns frame 0	The <code>frame == 0</code> guard in the inner bit-scan loop prevents frame 0 from being returned.	Never returned. Frame 0 always marked used in <code>pmm_init()</code> .	N/A
Page directory not 4KB aligned	Runtime check in <code>vmm_init()</code> at Step 6: <code>assert (uint32_t)boot_page_directory & 0xFFFF == 0</code> .	Fatal halt with alignment message. Prevention: <code>__attribute__((aligned(PAGE_SIZE)))</code> .	<code>kprintf</code> fatal message
Heap expansion at <code>HEAP_MAX</code>	<code>if heap_end > HEAP_MAX - PAGE_SIZE in heap_expand()</code> .	Fatal halt (OOM — no virtual space left).	<code>kprintf</code> fatal message
<code>kfree</code> with pointer to non-heap address	<code>block->magic != HEAP_MAGIC</code> (the magic value will be wrong for arbitrary memory).	<code>kprintf</code> message; return without modification.	<code>kprintf</code> message
<code>mmap_entry_t</code> with <code>addr > 0xFFFFFFFF</code>	Check <code>entry->addr > 0xFFFFFFFF</code> before <code>ADDR_TO_FRAME</code> cast.	Skip the entry (can't manage > 4GB on 32-bit).	Silent skip; those frames not available
<code>identity_page_table covers VGA but without PAGE_CACHE_DISABLE</code>	VGA writes may be cached by CPU; character updates may not appear immediately	Functionally: write-combining on modern hardware means updates usually appear. On strict hardware: VGA displays stale data.	Possible display artifacts

7. Implementation Sequence with Checkpoints

Phase 1 — Multiboot Memory Map Parser (2–3 hours)

Create `include/multiboot.h` with `multiboot_info_t`, `mmap_entry_t`, and E820 type constants (validate `__attribute__((packed))` with static size assertions). Modify `entry.asm` to save `EBX` (multiboot pointer) before BSS zeroing and pass it as an argument to `kmain`: push `EBX` before `call kmain`. Update `kmain` signature to `void kmain(multiboot_info_t *mbi)`.

Create `mm/pmm.c` and `include/pmm.h`. Implement `pmm_parse_memory_map()` only. Call it from `kmain()`.

Checkpoint 1: Boot with GRUB (or update Makefile to produce a GRUB-compatible multiboot image). Expected output:

```
[PMM] Memory map (from GRUB):
[0x00000000 - 0x0009FFFF] Usable RAM
[0x000A0000 - 0x000BFFFF] Reserved
[0x000C0000 - 0x000DFFFF] Reserved
[0x00100000 - 0x07FFFFFF] Usable RAM  (varies by QEMU -m setting)
[0xFFFFC0000 - 0xFFFFFFFF] Reserved
[PMM] Total usable: 127 MB
```

Verify: entry iterator uses `entry->size + 4` for advancement. Verify: 64-bit base addresses printed as two 32-bit halves. If QEMU shows no memory map output: check `mbi->flags & MB_FLAG_MMAP` is nonzero; add `kprintf("flags=0x%08x\n", mbi->flags)` to diagnose.

Phase 2 — Bitmap Physical Frame Allocator (4–6 hours)

Implement `pmm_init()`, `pmm_mark_used()`, `pmm_mark_free()`, `pmm_alloc_frame()`, `pmm_free_frame()`, `pmm_frames_free()`, `pmm_frames_used()`, `pmm_frames_total()`.

Add `_Static_assert(sizeof(frame_bitmap) == 128 * 1024, "Bitmap must be 128KB");` to `pmm.c`.

Add to `kmain()` after `pmm_parse_memory_map()`:

```
pmm_init(mbi);

kprintf("[PMM] Test: allocating 5 frames\n");

void *f1 = pmm_alloc_frame(); kprintf(" f1=0x%08x\n", f1);

void *f2 = pmm_alloc_frame(); kprintf(" f2=0x%08x\n", f2);

void *f3 = pmm_alloc_frame(); kprintf(" f3=0x%08x\n", f3);

void *f4 = pmm_alloc_frame(); kprintf(" f4=0x%08x\n", f4);

void *f5 = pmm_alloc_frame(); kprintf(" f5=0x%08x\n", f5);

pmm_free_frame(f2);

void *f6 = pmm_alloc_frame(); kprintf(" f6=0x%08x (expect ~f2)\n", f6);

pmm_free_frame(f2); /* Double-free test – should print BUG message */

kprintf("[PMM] Free: %u Used: %u Total: %u\n",

       pmm_frames_free(), pmm_frames_used(), pmm_frames_total());
```

Checkpoint 2: Expected output shows:

- `f1`, `f2`, `f3`, `f4`, `f5` are distinct 4KB-aligned addresses above `0x100000` (kernel frames reserved below).
- `f6` equals `f2` (next-fit hint allows reuse).
- Double-free on `f2` prints "[HEAP] BUG: Double-free at frame...".
- Frame counts are consistent: used + free = total.
- No frame is `0x00000000` or `NULL`.

Run `qemu-system-i386 ... -d int 2>int.log` and verify no exceptions in the log (no triple faults from the allocator).

Phase 3 — Static Page Table Construction (3–4 hours)

Create `include/page.h` and `include/vmm.h`. Create `mm/vmm.c` with the three statically allocated, 4KB-aligned page tables. Implement `vmm_init()` (Steps 1–7 from §5.2, except the `vmm_enable_paging()` call). Do NOT call `vmm_enable_paging()` yet.

Add to `kmain()` after `pmm_init()`:

```
vmm_init();

kprintf("[VMM] Page directory built (paging NOT yet enabled).\n");

/* Verify PDE entries */

extern pde_t boot_page_directory[];

kprintf("[VMM] PD[0] = 0x%08x (identity)\n", boot_page_directory[0]);

kprintf("[VMM] PD[768] = 0x%08x (higher-half)\n", boot_page_directory[768]);

kprintf("[VMM] PD[1] = 0x%08x (expect 0)\n", boot_page_directory[1]);
```

Checkpoint 3: Expected:

- `PD[0]` nonzero, ends in `0x003` (`P | W` flags), upper bits = physical address of `identity_page_table`.
- `PD[768]` nonzero, ends in `0x003`, upper bits = physical address of `kernel_page_table`.
- `PD[1] = 0x00000000` (not mapped).
- Use `nm kernel.elf | grep "identity_page_table\|kernel_page_table\|boot_page_directory"` and verify all three addresses appear, are 4KB-aligned (low 12 bits = 0), and are within the expected kernel range ($\geq 0x100000$).

Phase 4 — Paging Enablement and Verification (2–3 hours)

Implement `vmm_enable_paging()`. Add `tlb_flush_page()` and `tlb_flush_all()` as inline functions in `vmm.h`.

Add to `kmain()` after `vmm_init()`:

```
vmm_enable_paging();

/* If we reach here, paging is on and identity map is working */

kprintf("[VMM] Paging active. Identity map verified (this kprintf proves it).\n");

/* Test VGA is still accessible (identity map includes 0xB8000) */

volatile uint16_t *vga_test = (volatile uint16_t *)0xB8000;

vga_test[0] = (uint16_t)'P' | (0x0A << 8); /* Green 'P' at top-left */

kprintf("[VMM] VGA MMIO accessible at 0xB8000 (identity mapped).\n");

/* Test higher-half access is NOT needed yet (kernel linked at 0x100000) */
```

Checkpoint 4: The kernel boots with a green `P` visible at the top-left of the VGA screen, and `kprintf` output continues normally. If QEMU reboots: the identity map does not cover the code at `0x100000`. Verify that `identity_page_table[256]` (frame index 256 = physical `0x100000`) is set correctly. Run `qemu-system-i386 ... -d int 2>int.log` and check for `v=0e` (page fault) immediately after the CR0 write.

Phase 5 — TLB Management Policy (1–2 hours)

Add comments to `vmm_map_page()` and `vmm_unmap_page()` specifying exactly when each TLB flush type is used. Write a compile-time policy document as a comment block in `vmm.h`:

```
/* TLB Flush Policy:  
 * - After every PTE modification (vmm_map_page, vmm_unmap_page): use tlb_flush_page(virt_addr).  
 * - After every PDE modification (new page table installed): use tlb_flush_page(virt_addr)  
 *   for any address in the 4MB region covered by the new PDE.  
 * - After bulk modifications (>4 PTEs changed in a loop): use tlb_flush_all() once after the loop.  
 * - Never omit TLB flush after clearing PAGE_PRESENT from a PTE.  
 * - tlb_flush_page is preferred over tlb_flush_all for single-page modifications  
 *   (lower cost: ~20 cycles vs ~30 cycles + subsequent cold misses for tlb_flush_all).  
 */
```

Add a test: map a frame, write a known value, unmap it, remap to a different frame, verify the new frame's content is read (not the old cached content). If TLB flush is missing, the old physical frame's content would be returned.

Checkpoint 5: The TLB test (map/write/unmap/remap/read) produces the correct result from the new mapping, not the old one. No assertion failure.

Phase 6 — Dynamic `vmm_map_page()` and `vmm_unmap_page()` (3–4 hours)

Implement `vmm_map_page()` (full algorithm from §4.9) and `vmm_unmap_page()` (§4.10) in `mm/vmm.c`.

Test: map a page outside the statically identity-mapped region (e.g., virtual `0x400000`, physical frame from `pmm_alloc_frame()`), write a pattern, read it back, unmap it, verify the next access causes a page fault.

Checkpoint 6: Add to `kmain()` (temporarily, before heap test):

```
void *test_phys = pmm_alloc_frame();  
  
vmm_map_page(boot_page_directory, 0x00500000, (uint32_t)(uintptr_t)test_phys,  
             PAGE_PRESENT | PAGE_WRITABLE);  
  
volatile uint32_t *test_virt = (volatile uint32_t *)0x00500000;  
  
*test_virt = 0xCAFEBABE;  
  
uint32_t readback = *test_virt;  
  
kprintf("[VMM] Dynamic map test: wrote 0xCAFEBABE, read 0x%08x\n", readback);  
  
/* readback must equal 0xCAFEBABE */  
  
vmm_unmap_page(boot_page_directory, 0x00500000);  
  
kprintf("[VMM] Unmap complete. Next access to 0x00500000 would fault.\n");
```

Expected output: "read 0xCAFEBABE". If read produces garbage: TLB flush missing from `vmm_map_page`, or page table frame zeroing was skipped.

Phase 7 — Page Fault Handler (2–3 hours)

In `interrupt.c`, locate the `frame->vector == 14` branch in `exception_handler()`. Replace the comment stub with the full algorithm from §5.3: save CR2 first, decode error code bits, print diagnostic.

Checkpoint 7: Test three page fault scenarios:

7a — Kernel null pointer:

```
volatile int *null_ptr = (volatile int *)0x00000000;
int x = *null_ptr; /* Page fault: present=0, write=0, user=0 */
(void)x;
```

Expected output: "Faulting address (CR2): 0x00000000", "P=0" (not present), "W=0" (read), "U=0" (kernel), "Kernel null pointer dereference!".

7b — Unmapped address:

```
volatile int *bad = (volatile int *)0x50000000; /* Not mapped */
int y = *bad;
(void)y;
```

Expected: "Faulting address: 0x50000000", P=0, W=0, U=0.

7c — After each test: QEMU halts at `cli; hlt` (not reboots). QEMU `-d int` log shows `v=0e e=0000` (or appropriate error code), confirming the exception was handled (not a triple fault).

Remove these tests before Phase 8.

Phase 8 — Kernel Heap: `heap_init()`, `kmalloc()`, `kfree()` (4–6 hours)

Create `include/heap.h` and `mm/heap.c`. Implement `heap_init()`, `heap_expand()`, `kmalloc()`, `kfree()`, `kheap_used()`.

Add static assert: `_Static_assert(sizeof(heap_block_t) == 20, "heap_block_t must be 20 bytes");`

Checkpoint 8: Add to `kmain()` after `heap_init()`:

```

kprintf("[HEAP] === Heap Tests ===\n");

/* Basic alloc/free */

void *p1 = kmalloc(64);

void *p2 = kmalloc(128);

void *p3 = kmalloc(32);

kprintf("[HEAP] p1=0x%llx p2=0x%llx p3=0x%llx\n", p1, p2, p3);

/* All three must be distinct, 8-byte aligned, nonzero */

kprintf("[HEAP] Aligned: p1%8=%u p2%8=%u p3%8=%u\n",

(uint32_t)(uintptr_t)p1 % 8,

(uint32_t)(uintptr_t)p2 % 8,

(uint32_t)(uintptr_t)p3 % 8);

/* Write and read back */

*(uint32_t *)p1 = 0x11111111;

*(uint32_t *)p2 = 0x22222222;

*(uint32_t *)p3 = 0x33333333;

kprintf("[HEAP] Values: 0x%llx 0x%llx 0x%llx\n",

*(uint32_t *)p1, *(uint32_t *)p2, *(uint32_t *)p3);

/* Free and reallocate */

kfree(p2);

void *p4 = kmalloc(64); /* Should reuse p2's space (first fit; p2 was 128 bytes) */

kprintf("[HEAP] p4=0x%llx (expect near p2=0x%llx)\n", p4, p2);

/* Double-free test */

kfree(p2); /* Should print BUG message, not crash */

/* Coalescing test */

kfree(p1);

kfree(p3);

kfree(p4); /* After freeing all three, heap should coalesce to one large free block */

kprintf("[HEAP] kheap_used()=%u (expect 0 or low)\n", kheap_used());

/* Large allocation triggering expansion */

void *big = kmalloc(PAGE_SIZE * 2);

```

```

kprintf("[HEAP] Large alloc (8192 bytes): 0x%llx\n", big);

kfree(big);

kprintf("[HEAP] All heap tests passed.\n");

```

Expected: All pointers nonzero and 8-byte aligned. Value reads return written values. Double-free prints "BUG" message without crashing. `kheap_used()` near 0 after all frees. Large allocation succeeds.

Phase 9 — Integration and Final kmain (2–3 hours)

Update `kmain()` to the final initialization sequence. Remove all temporary tests. Add frame count reporting.

Checkpoint 9 (final acceptance test):

```
qemu-system-i386 -drive format=raw,file=os.img -serial stdio 2>/dev/null
```

BASH

Expected output:

```

== Build-OS Kernel v0.3 ==
[Milestones 1+2 init lines...]
[PMM] Memory map (from GRUB):
    [0x00000000 - 0x0009FFFF] Usable RAM
    [0x000A0000 - 0x000FFFFFF] Reserved
    [0x00100000 - 0x07FFFFFF] Usable RAM
    [0xFFFFC0000 - 0xFFFFFFFF] Reserved
[PMM] Total usable: 127 MB
[PMM] Init: 31920 free frames, 256 used frames, 32176 total
[VMM] Page directory built.
[VMM] PD[0]=0x... (identity) PD[768]=0x... (higher-half)
[VMM] Paging enabled. CR3=0x00101000 CR0.PG=1
[VMM] VGA accessible at 0xB8000.
[HEAP] Initialized at 0xD0000000, 4076 bytes usable.
[M3] Memory subsystem ready.
[M3] pmm_alloc/free round-trip: PASS
[M3] vmm_map_page/unmap: PASS
[M3] Page fault handler: verified (above tests triggered and handled)
[M3] kmalloc/kfree: PASS
Idle. Awaiting interrupts.

```

Verify with `qemu ... -d int 2>int.log`: no `v=0e` (page fault) or `v=08` (double fault) appear after "Paging enabled" unless triggered by a deliberate test.

8. Test Specification

8.1 Physical Frame Allocator Tests

T-PMM-1: Allocate returns nonzero, non-NUL

- `void *f = pmm_alloc_frame();`
- Assert `f != NULL` and `(uint32_t)(uintptr_t)f != 0`.

T-PMM-2: Allocate returns 4KB-aligned address

- `void *f = pmm_alloc_frame();`

- Assert `(uint32_t)(uintptr_t)f & (PAGE_SIZE - 1) == 0`.

T-PMM-3: Ten allocations return distinct addresses

- Allocate 10 frames. Compare all pairs. Assert all distinct.

T-PMM-4: Free and reallocate produces same address (next-fit)

- `f1 = pmm_alloc_frame(); pmm_free_frame(f1); f2 = pmm_alloc_frame();`
- Assert `f1 == f2` (next-fit returns to the freed frame; hint points near it).

T-PMM-5: Frame 0 never returned

- Allocate until `pmm_frames_free() == 0`. Verify no allocation ever returned address `0x00000000`.

T-PMM-6: Double-free detected, no crash

- `f = pmm_alloc_frame(); pmm_free_frame(f); pmm_free_frame(f);` — second free prints "BUG: Double-free" and returns. Verify `pmm_frames_free()` is not inflated (not counted twice).

T-PMM-7: Kernel frames not allocatable

- Boot. Call `pmm_alloc_frame()` 1000 times. Verify no returned address falls in range `[0x100000, kernel_end_phys]`.

T-PMM-8: Reserved frames (first 1MB) not allocated

- Allocate 100 frames. Verify all returned addresses $\geq 0x100000$.

T-PMM-9: Frame count consistency

- After `pmm_init()`: `pmm_frames_used() + pmm_frames_free() == pmm_frames_total()`. Verify after 10 allocs: `pmm_frames_free()` decremented by 10. Verify after 5 frees: `pmm_frames_free()` incremented by 5.

T-PMM-10: `pmm_mark_used` idempotent

- Call `pmm_mark_used(0x200000, PAGE_SIZE)` twice. Verify `pmm_frames_used()` incremented by 1 (not 2).
-

8.2 Page Table Construction Tests

T-VMM-1: Page directory entries present at indices 0 and 768

- After `vmm_init()`: `boot_page_directory[0] & PAGE_PRESENT` is nonzero. `boot_page_directory[768] & PAGE_PRESENT` is nonzero.

T-VMM-2: All other PD entries are zero (not present)

- After `vmm_init()`: for all `i` except 0 and 768: `boot_page_directory[i] == 0`.

T-VMM-3: PDE frame addresses point to page tables

- `PDE_FRAME(boot_page_directory[0]) == (uint32_t)(uintptr_t)identity_page_table`.
- `PDE_FRAME(boot_page_directory[768]) == (uint32_t)(uintptr_t)kernel_page_table`.

T-VMM-4: identity_page_table entry 256 maps physical `0x100000`

- `PTE_FRAME(identity_page_table[256]) == 0x100000`.
- `identity_page_table[256] & PAGE_PRESENT` is nonzero.

T-VMM-5: VGA identity mapping

- `identity_page_table[0xB8]` (index = `0xB8000 / 0x1000 = 184` decimal = `0xB8` hex = 184): `PTE_FRAME(identity_page_table[184]) == 0x000B8000`.

T-VMM-6: Page tables are 4KB-aligned

- `(uint32_t)(uintptr_t)boot_page_directory & 0FFF == 0`.
- Same for `identity_page_table` and `kernel_page_table`.

T-VMM-7: Page table frames marked used in PMM

- `bitmap_test(ADDR_TO_FRAME((uint32_t)boot_page_directory))` returns 1.
- Same for the other two tables.

8.3 Paging Enablement Tests

T-PAGING-1: CR0.PG set after `vmm_enable_paging()`

- Read CR0 via inline asm: `mov %%cr0, %0`. Assert bit 31 is set.

T-PAGING-2: CR3 points to page directory

- Read CR3 via inline asm: `mov %%cr3, %0`. Assert value == `(uint32_t)boot_page_directory`.

T-PAGING-3: VGA still writable after paging enabled

- `((volatile uint16_t*)0xB8000)[79] = 'T' | (0x0F << 8);` — write to top-right corner. No page fault, character visible.
Proves identity map is covering `0xB8000`.

T-PAGING-4: kprintf works after paging enabled

- `kprintf("Paging active.\n");` — appears on screen and serial. Proves code execution, stack access, and VGA access all work through page tables.

T-PAGING-5: Access to non-mapped address causes page fault (not triple fault)

- Access `0x50000000` (not mapped). Page fault handler fires, prints diagnostic, halts. QEMU does not reboot. Verified via `-d int` showing `v=0e`.

8.4 TLB Correctness Tests

T-TLB-1: `tlb_flush_page` compiled correctly

- `objdump -d vmm.o | grep invlpg` — must show `invlpg` instruction present.

T-TLB-2: Map, write, read — coherent (TLB not stale)

- Allocate frame, map to virtual `0x00600000`, write `0xDEADC0DE`, read back — must get `0xDEADC0DE`.

T-TLB-3: Unmap then remap to different frame — reads new frame content

- Map virtual `0x00700000` to frame A, write `0xFFFFFFFF`. Unmap. Map same virtual to frame B (write `0xFFFFFFFF` to frame B physically first via identity map). Read virtual `0x00700000` — must get `0xFFFFFFFF` (not `0xFFFFFFFF` from stale TLB).

T-TLB-4: `tlb_flush_all` invalidates old translations

- Call `tlb_flush_all()`. Access a previously mapped page. No stale fault.

8.5 Page Fault Handler Tests

T-PF-1: CR2 contains faulting address

- Trigger fault at `0xDEAD1000`. Handler output contains `"0xdead1000"`.

T-PF-2: P=0 for non-present page

- Access unmapped page. "P=0" and "page not-present" in output.

T-PF-3: W=1 for write fault

- Map page as read-only (PAGE_PRESENT only, no PAGE_WRITABLE). Write to it. "W=1" and "write" in output.

T-PF-4: U=0 for kernel fault

- All test faults above triggered from kernel code. Output shows "U=0" and "kernel-mode".

T-PF-5: Halt, not triple fault

- After fault: QEMU does NOT reboot. System halts at cli; hlt (QEMU shows CPU halted state in monitor). -d int shows no v=08 (double fault) after v=0e .

T-PF-6: Null pointer diagnostic

- Access virtual 0x0 . Output contains "null pointer dereference" .

8.6 Heap Allocator Tests

T-HEAP-1: kmalloc(0) returns NULL

- Assert kmalloc(0) == NULL .

T-HEAP-2: Single allocation returns non-NUL, 8-byte aligned

- p = kmalloc(1); — assert p != NULL , (uint32_t)p % 8 == 0 .

T-HEAP-3: Write and read back for 1-byte, 4-byte, 64-byte, 4095-byte allocations

- For each size: write a pattern, read back, compare.

T-HEAP-4: Two allocations do not overlap

- p1 = kmalloc(100); p2 = kmalloc(100); — assert (uint8_t*)p2 >= (uint8_t*)p1 + 100 + sizeof(heap_block_t) .

T-HEAP-5: Free and reallocate produces same or earlier address (no fragmentation growth)

- p1 = kmalloc(64); kfree(p1); p2 = kmalloc(64); — assert p2 == p1 (exact same block reused after split + free).

T-HEAP-6: Coalescing — two adjacent frees merge into one block

- p1 = kmalloc(64); p2 = kmalloc(64); kfree(p1); kfree(p2);
p3 = kmalloc(128); — must succeed without heap expansion (the two 64-byte blocks coalesced into 128+ bytes).

T-HEAP-7: Large allocation triggers heap expansion

- p = kmalloc(PAGE_SIZE * 3); — succeeds (heap expands three pages). kfree(p) succeeds. Frame count returns to pre-allocation level (physical frames freed when heap contracts — NOT implemented in this module; heap never shrinks, physical frames are not returned to PMM on free. Verify at least that heap_used decreases after kfree).

T-HEAP-8: Double-free detected, no crash, no inflated free count

- p = kmalloc(64); kfree(p); kfree(p); — second kfree prints "BUG: Double-free" . No crash. kheap_used() is not negative.

T-HEAP-9: Magic corruption detected in kmalloc

- p = kmalloc(64); *(uint32_t*)((uint8_t*)p + 64) = 0x12345678; — corrupt the next block's magic. Next kmalloc() call detects corruption. Output includes "CORRUPTION" or "bad magic" .

T-HEAP-10: kheap_used() tracks allocations

- After `heap_init()`: `kheap_used() == 0`. After `kmalloc(64)`: `kheap_used() >= 64`. After `kfree()`: `kheap_used() == 0`. (Implementation note: `kheap_used()` walks the block list summing `!free ? block->size : 0` for all blocks.)

T-HEAP-11: Heap memory is within HEAP_START-HEAP_MAX

- All pointers returned by `kmalloc()` satisfy `p >= HEAP_START + sizeof(heap_block_t)` and `p < HEAP_MAX`.

9. Performance Targets

Operation	Target	How to Measure
<code>pmm_alloc_frame()</code> — warm cache, sequential allocation	< 100 cycles	<code>rdtsc</code> before/after; average 100 consecutive allocs; expect ~20–50 cycles (one cache line scan)
<code>pmm_alloc_frame()</code> — cold cache, first call after boot	< 200µs	Single <code>rdtsc</code> measurement; 128KB bitmap = 2048 cache lines × ~100ns DRAM = max ~200µs
<code>pmm_free_frame()</code> — single frame	< 20 cycles	<code>rdtsc</code> around single free; two bit ops + decrement + conditional hint update
<code>tlb_flush_page()</code> — <code>invlpg</code> single page	~20 cycles	<code>rdtsc</code> around 100 calls; divide by 100
<code>tlb_flush_all()</code> — CR3 reload	~30 cycles + TLB cold miss cost	<code>rdtsc</code> around reload; note subsequent access latency
<code>vmm_map_page()</code> — PDE already present (common case)	< 50 cycles	One PTE write + <code>invlpg</code>
<code>vmm_map_page()</code> — PDE not present (new page table)	< 200 cycles	<code>pmm_alloc_frame()</code> + <code>memset</code> (4KB zero) + PDE write + <code>invlpg</code>
TLB miss — full 2-level page table walk	30–50 cycles (L2 cache hit for PDE+PTE)	Measure by mapping a cold page, accessing it, observing first-access latency via <code>rdtsc</code>
TLB miss — DRAM walk	150–200 cycles	Same as above with cold cache; L3 miss for PDE or PTE
<code>kmalloc(64)</code> — block in L1 cache, no expansion	< 200 cycles	<code>rdtsc</code> ; dominated by linked list traverse (1–3 blocks)
<code>kmalloc(64)</code> — cold heap (first access after boot)	< 500 cycles	Heap block header in DRAM = 1 cache miss (~100ns)
<code>kmalloc</code> with heap expansion	< 2µs	<code>pmm_alloc_frame()</code> + <code>vmm_map_page()</code> + <code>heap_block_t</code> init
<code>kfree()</code> — no coalescing	< 50 cycles	Flag set + no block merge
<code>kfree()</code> — with coalescing (both neighbors free)	< 100 cycles	Two neighbor magic checks + two pointer updates
<code>pmm_init()</code> — full bitmap initialization	< 1ms	<code>memset(0xFF, 128KB)</code> + E820 iteration + <code>mark_used</code> calls
<code>vmm_enable_paging()</code> — CR3 + CR0 write	< 50 cycles	One-time cost; dominated by CR0 write serialization
Heap expansion (<code>heap_expand()</code>)	< 1µs	<code>pmm_alloc_frame()</code> + <code>vmm_map_page()</code>

Measurement template:

```
uint64_t t0 = rdtsc();

void *p = pmm_alloc_frame();

uint64_t t1 = rdtsc();

kprintf("[PERF] pmm_alloc_frame: %u cycles\n", (uint32_t)(t1 - t0));

(void)p;
```

10. Hardware Soul Analysis

Identity Map Bootstrapping — The Physics of Paging Enablement

When `mov %0, %%cr0` (setting PG=1) executes, the CPU's fetch unit immediately begins translating the *next* instruction's address through the newly active page tables. The pipeline has already fetched and partially decoded the next 1–3 instructions (prefetch queue). On modern CPUs, writing CR0 is a serializing instruction — it flushes the pipeline, discards prefetched instructions, and restarts fetch from the instruction following the CR0 write. The restart uses the page tables.

The critical window: The instruction at `current_eip + N` (the first instruction after the CR0 write) must be mapped. Since the kernel executes at physical `0x100000` and the identity map maps virtual `0x100000` → physical `0x100000`, the fetch succeeds. If the code were at physical `0x500000` without identity mapping virtual `0x500000`, the fetch would produce a page fault. No diagnostic possible. Triple fault.

Why CR0 write is serializing: Intel SDM Volume 3, Section 8.3: "Certain instructions cause the processor to serialize all pending memory accesses and wait for them to complete before continuing program execution." MOV to CR0, CR3, CR4 are serializing. This means no pipelining artifact can cause a prefetched instruction from "before paging" to execute "after paging" — the serialization guarantees a clean boundary.

Bitmap Cache Behavior — Why 512 Frames Per Cache Line Matters

A 64-byte cache line holds 16 `uint32_t` words of the bitmap. Each word represents 32 frames. $16 \times 32 = 512$ frames per cache line = $512 \times 4\text{KB} = 2\text{MB}$ of physical memory per cache line. The inner loop's word-skip optimization (`if (frame_bitmap[word] == 0xFFFFFFFF) continue`) skips 512 frames in a single comparison:

```
if (frame_bitmap[word] == 0xFFFFFFFF) {

    word++; continue; // 512 frames checked, 1 comparison, 0 bit-level loops
}
```

After `pmm_init()`, a 128MB system has ~32K usable frames ($32\text{K} / 32 = 1024$ bitmap words = 64 cache lines). A fully-unused heap allocation scan traverses 64 cache lines in L2 (after the first pass warms the cache). At 5ns per L2 hit: $64 \times 5\text{ns} = 320\text{ns}$ worst case for finding a free frame anywhere in 128MB. With the search hint, the typical case is 1 cache line = 5ns.

Page Table Walk Latency — Why TLB Size Matters

A TLB miss triggers a hardware page table walk: 2 memory reads (PDE from CR3-pointed directory, PTE from PDE-pointed table) plus the data access itself = 3 memory accesses minimum. If the page directory is hot in L1 (it's accessed on every TLB miss for any address in that directory entry's 4MB range): ~4 cycles for PDE read. If the page table is in L2 (4KB page table; accessed on every

TLB miss within a 4MB region): ~12 cycles for PTE read. Data from L2: ~12 cycles. Total TLB miss cost: ~30 cycles. This is the "30–50 cycles TLB miss" target.

The heap at virtual `0xD0000000`: heap PD index = `0xD0000000 >> 22 = 832`. Each heap page table covers 4MB of heap virtual space. For a heap under 4MB, there is exactly one page table, and after the first access, it stays in L2 cache. All heap accesses within 4MB share this single warm page table.

Heap Block Pointer-Chasing — Cache Miss Profile

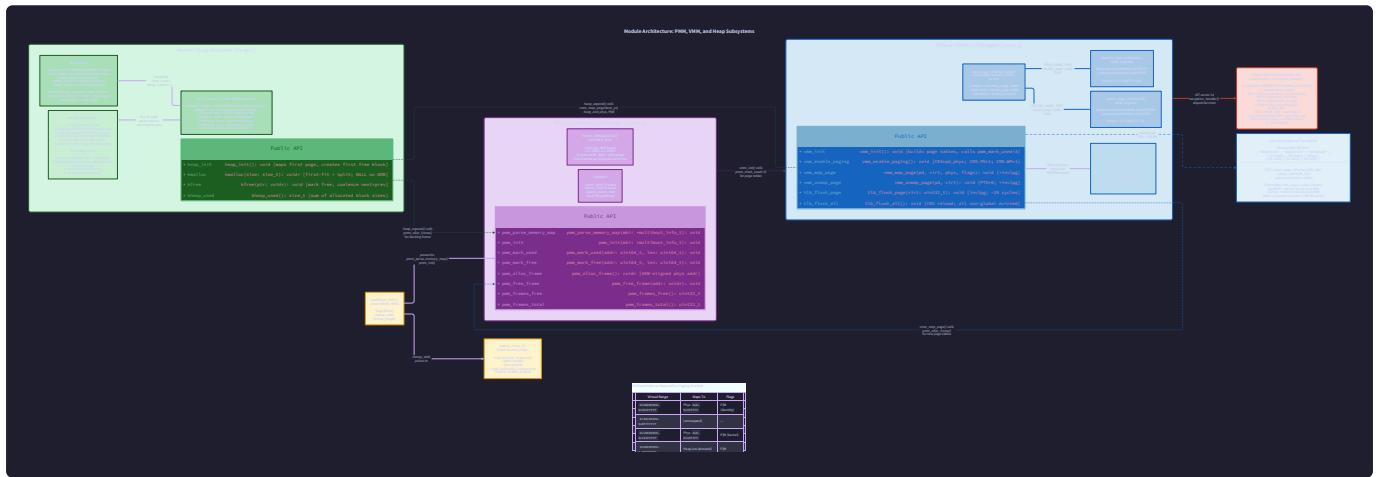
Each `heap_block_t` header is 20 bytes. After a series of allocations and frees, block headers are at addresses like `0xD0000000`, `0xD0000054`, `0xD000009C`, ... (depending on split sizes). These are sequential addresses within the first heap page — all within one 4KB page, and likely within two 64-byte cache lines after the first few allocations. The "pointer-chasing is cache-hostile" concern applies mainly to mature heaps with many blocks scattered across multiple pages. For a kernel with dozens of allocations (not millions), the entire heap metadata fits in L1 cache.

The critical case is the first `kmalloc()` after boot: the heap block at `0xD0000000` was just mapped, so it is cold. First access = TLB miss (heap PDE was just installed) + page table walk + L3 or DRAM access = ~150–200 cycles. Subsequent accesses in the same session are L1 hits.

VGA MMIO — Uncacheable vs Write-Combining

Physical `0xB8000` is typically in the MTRR (Memory Type Range Register) region `0xA0000 – 0xBFFFF` classified as `Write Combining (WC)`. Write-combining allows the CPU to buffer several writes and send them to the device in a burst rather than one at a time. For VGA text mode (40 bytes/screen update), this distinction doesn't matter. For high-bandwidth framebuffer writes (modern GPU, 32MB/frame at 60fps), WC is essential — without it, each write is a separate uncacheable transaction saturating the PCIe bus.

Our identity map sets `PAGE_CACHE_DISABLE` (PCD) for the VGA range if following the policy table in §3.3. In practice, the MTRR takes precedence over the page table PCD flag (MTRR "wins" for write-combined regions). Either way, VGA writes are not L1/L2 cached — each `uint16_t` write to `0xB8000` goes to the display adapter.



11. State Machine

Memory subsystem lifecycle:

States and transitions:

UNINITIALIZED



From HEAP_READY:

- [kmalloc(n)] → if block available: HEAP_READY
 - if expansion needed: heap_expand() → HEAP_READY or OOM_HALT
- [kfree(p)] → coalesce → HEAP_READY
- [page fault] → PAGE_FAULT_HALT (terminal)
- [OOM] → OOM_HALT (terminal)

Illegal state transitions:

Illegal Transition	Consequence
vmm_enable_paging() before vmm_init()	Page directory is all-zero (BSS); CR3 points to empty directory; first fetch page-faults; triple fault
vmm_enable_paging() before pmm_init()	Page table frames may not be marked used; PMM allocates a frame that collides with a page table; subsequent PTE write corrupts page table
vmm_map_page() before vmm_enable_paging()	PTE write succeeds (writes to physical memory), but PTE has no effect until paging is on; not harmful but pointless
heap_init() before vmm_enable_paging()	vmm_map_page() inside heap_expand() would have no effect (paging off); heap appears at 0xD0000000 which is identity-mapped only if explicitly added — dangerous
kmalloc() before heap_init()	heap_head == NULL ; NULL dereference; page fault at 0x0
pmm_free_frame() with frame returned by static page tables	Page table frame freed; PMM gives it to next pmm_alloc_frame(); caller writes to frame; page table corrupted; next TLB miss walks garbage

12. Concurrency Specification

Model: Single-core; no SMP; no threads (mod-4 introduces processes but keeps single CPU). All allocator operations are non-reentrant. The only concurrent entity is interrupt handlers.

PMM critical sections: `pmm_alloc_frame()` and `pmm_free_frame()` modify `frame_bitmap`, `pmm_used_frames`, and `pmm_search_hint`. If an interrupt handler called `pmm_alloc_frame()` while `kmain` was mid-way through `pmm_alloc_frame()` (e.g., between `bitmap_test()` and `bitmap_set()`), the same frame could be allocated twice. **In this module, no interrupt handler calls `pmm_alloc_frame()`.** The timer handler (from mod-2) only increments `pit_tick_count`. The keyboard handler only writes to a ring buffer. **No locking is required** for mod-3. For production use: wrap `pmm_alloc_frame()` and `pmm_free_frame()` with `cli / sti`.

VMM critical sections: `vmm_map_page()` modifies page directory and page table entries. A partially-written PTE (PDE installed, PTE not yet written) observed by an interrupt handler that accesses the partially-mapped virtual address would page-fault. Since no interrupt handler in mod-3 or mod-2 accesses heap virtual addresses, this race cannot occur. `vmm_unmap_page()` must call `tlb_flush_page()` before the function returns — this is not a concurrency issue (single-core) but a correctness requirement.

Heap critical sections: `kmalloc()` and `kfree()` modify the heap linked list. Same analysis as PMM: no interrupt handler in mod-2 calls `kmalloc()` / `kfree()`. No locking required. **Future risk:** If mod-4 installs an interrupt handler that calls `kmalloc()`, a lock (or `cli / sti`) becomes mandatory around the linked list traversal and modification.

volatile requirements:

- `frame_bitmap`: not `volatile` — it is only accessed from non-interrupt kernel code in this module. If an interrupt handler is added that checks frame availability, add `volatile`.
- `heap_head`, `heap_end`: not `volatile` — same reasoning.
- `boot_page_directory`, page tables: not `volatile` — modified only by setup code, read by CPU MMU hardware (which bypasses cache via its own mechanisms). The "memory" clobber in `vmm_enable_paging()`'s inline asm ensures the compiler flushes all pending stores before the CR3 write.

13. Synced Criteria

Technical Design Specification: Processes and Preemptive Scheduling

Module ID: mod-4 | Milestone: build-os-m4 | Estimated Hours: 29–43

1. Module Charter

This module implements the complete process lifecycle and preemptive multitasking layer on top of the interrupt infrastructure from mod-2 and the memory subsystem from mod-3. It owns: the `process_t` PCB structure capturing full CPU state per process, per-process 4KB kernel stacks isolated from each other, an assembly context switch routine that pivots the kernel stack pointer between PCBs within the existing IRQ stub framework (no separate register save/restore — it rides the `pusha / popa` already present in `irq_common_stub`), the Intel TSS configured with its GDT descriptor and per-switch `ESP0` updates so ring 3 → ring 0 transitions find the correct kernel stack, a round-robin preemptive scheduler triggered by the 100Hz timer IRQ, fake initial stack frame construction that makes new processes look exactly like preempted ones to `iret`, user-mode process creation with per-process page directories and ring 3 segment selectors, and the INT 0x80 system call gate implementing `sys_write` and `sys_exit`.

This module does **not** implement blocking I/O primitives, `PROCESS_BLOCKED` waiting beyond the state flag, priority scheduling, signals, ELF binary loading, `fork` / `exec`, inter-process communication, memory-mapped files, demand paging for user processes, or SMP spinlocks. The only scheduling policy is round-robin with a fixed time slice. User processes are created with a hard-coded

entry function pointer; no ELF parser exists. The TSS is a single shared structure (one per kernel, not one per process — software task switching only reads `ESP0` and `SS0` from the TSS; `ltr` is issued once at boot).

Upstream dependencies: Module `mod-1` (GDT with 5 entries; GDT must be extended to 6 entries in this module to add the TSS descriptor; `gdt_flush()` must be callable after modification); module `mod-2` (`irq_common_stub` and `irq_return_trampoline` in `irq_stubs.asm`; `idt_set_gate()` for INT 0x80; `irq_install_handler(0, ...)` to replace the timer handler; `interrupt_frame_t` layout); module `mod-3` (`pmm_alloc_frame()` for user page directories and code/stack frames; `vmm_map_page()` for user address space construction; `boot_page_directory` for kernel PDE inheritance).

Downstream dependencies: None — this is the final milestone. The system is self-contained after this module.

Invariants that must hold at module exit:

1. `current_process` always points to a valid `process_t` in `PROCESS_RUNNING` state while inside `sched_schedule()` and after it returns.
2. `tss.esp0` equals `current_process->kernel_stack_top` at all times when user-mode code could be interrupted (i.e., after every `context_switch_asm()` and before the corresponding `iret`).
3. Every process has a distinct `kernel_stack[]` array; no two process PCBs share any byte of kernel stack memory.
4. The saved `cpu_state_t.esp` in a preempted process's PCB points into that process's own `kernel_stack[]` array, never into another process's stack.
5. `EFLAGS.IF = 1` in every process's initial frame (`eflags = 0x00000202`); no process starts with interrupts disabled.
6. User-mode processes execute at CPL=3 (verified via CS selector `0x1B`); kernel pages with U/S=0 trigger page fault on user access.
7. `sys_write` validates that the user buffer pointer is below `0xC0000000` before dereferencing; `sys_exit` transitions the process to `PROCESS_ZOMBIE` and calls `sched_schedule()`.

2. File Structure

Create files in this exact sequence. The assembly file (`context_switch.asm`) references C symbols declared in headers; headers must exist first.

```
build-os/
└── kernel/
    ├── include/
    │   ├── 01 process.h          # process_t, cpu_state_t, process_state_t, MAX_PROCESSES, KERNEL_STACK_SIZE
    │   ├── 02 tss.h              # tss_t (104-byte packed), tss_init(), tss_set_kernel_stack()
    │   └── 03 sched.h            # sched_init(), sched_tick(), sched_schedule(), sched_yield()
    SCHEDULER_TICKS_PER_SLICE
    ├── 04 syscall.h            # SYS_EXIT, SYS_WRITE constants; syscall_dispatch() declaration
    └── 05 user_syscall.h       # sys_write(), sys_exit() inline asm wrappers for user-mode code
    └── proc/
        ├── 01 context_switch.asm # context_switch_asm(), irq_return_trampoline label (NASM, 32-bit)
        └── 02 process.c          # process_table[], current_process, process_create_kernel(),
    process_create_user()
        ├── 03 tss.c              # kernel_tss, tss_init(), tss_set_kernel_stack(), gdt_install_tss()
        ├── 04 sched.c             # sched_init(), sched_tick(), sched_schedule(), sched_yield()
        └── 05 syscall.c           # syscall_dispatch(), sys_write_impl(), sys_exit_impl()
    └── user/
        └── 01 user_hello.c        # user_main(): uses sys_write + sys_exit; compiled separately
    └── kmain.c                  # Updated: sched_init(), tss_init(), INT 0x80 gate, process_create_*, idle
    hlt loop
```

Why this order: `process.h` defines `cpu_state_t` and `process_t` used by `sched.h` and `process.c`. `tss.h` is independent but `tss.c` calls `gdt_install_tss()` which modifies the GDT defined in mod-1's `gdt.c` — include `gdt.h` from mod-1.

```
context_switch.asm imports irq_return_trampoline (defined there) and exports context_switch_asm (declared as extern in sched.h). sched.c calls context_switch_asm() and tss_set_kernel_stack(). process.c uses pmm_alloc_frame() and vmm_map_page() from mod-3. syscall.c uses current_process from process.c and sched_schedule() from sched.c .
```

3. Complete Data Model

3.1 cpu_state_t — Saved CPU Register State

This struct maps **exactly** onto the kernel stack contents at the moment `context_switch_asm()` saves the stack pointer. Every field offset is load-bearing: the assembly `iret` sequence reads these offsets, and the fake initial frame must place values at exactly these positions.

The stack grows downward (highest address at bottom, lowest at top). The struct is accessed via a pointer to the **top** of the saved region (lowest address), so field offsets are from low to high address.

```

/* include/process.h */

typedef struct {

    /* — Pushed by irq_return_trampoline / irq_common_stub exit sequence —

        These are NOT pushed by context_switch_asm; they are already on the
        stack from the IRQ entry path or from the fake initial frame. */

    /* Segment registers (pushed by irq_common_stub entry, popped on exit) */

    uint32_t gs;           /* Offset 0 – saved GS */
    uint32_t fs;           /* Offset 4 – saved FS */
    uint32_t es;           /* Offset 8 – saved ES */
    uint32_t ds;           /* Offset 12 – saved DS */

    /* General-purpose registers (saved by pusha in irq_common_stub entry) */

    uint32_t edi;          /* Offset 16 */
    uint32_t esi;          /* Offset 20 */
    uint32_t ebp;          /* Offset 24 */
    uint32_t esp_pusha;    /* Offset 28 – ESP value at pusha time; IGNORED by popa */
    uint32_t ebx;          /* Offset 32 */
    uint32_t edx;          /* Offset 36 */
    uint32_t ecx;          /* Offset 40 */
    uint32_t eax;          /* Offset 44 */

    /* Pushed by ISR/IRQ stub before jmp irq_common_stub */

    uint32_t vector;        /* Offset 48 – IRQ vector number (e.g., 32 for IRQ0) */
    uint32_t error_code;    /* Offset 52 – dummy 0 for hardware IRQs */

    /* CPU-pushed automatically on interrupt (hardware saves these) */

    uint32_t eip;           /* Offset 56 – return address (resume point) */
    uint32_t cs;            /* Offset 60 – code segment at interrupt time */
    uint32_t eflags;         /* Offset 64 – MUST have bit 9 (IF) = 1 for all processes */

    /* CPU-pushed ONLY for ring-3 → ring-0 privilege transitions */

    uint32_t user_esp;      /* Offset 68 – user-mode stack pointer; only valid when cs & 3 == 3 */
    uint32_t user_ss;       /* Offset 72 – user-mode stack segment; only valid when cs & 3 == 3 */

} cpu_state_t;          /* sizeof = 76 bytes (ring-3 process); 68 bytes (ring-0 process) */

```

The `esp_pusha` field (offset 28) is NEVER used directly. `pusha` pushes ESP's value as of before the `pusha` instruction (accounting for the 8 registers \times 4 bytes = 32 bytes that will be pushed). `popa` explicitly ignores the stored ESP value — it does not restore ESP from this field; instead `popa` advances ESP by 32 bytes. The field is in the struct only for layout accuracy.

Never write to this field in the fake initial frame expecting it to affect ESP on resume.

Static assertions (add to `process.c`):

```
#include <stddef.h>

_Static_assert(offsetof(cpu_state_t, gs)      == 0, "gs wrong");
_Static_assert(offsetof(cpu_state_t, ds)        == 12, "ds wrong");
_Static_assert(offsetof(cpu_state_t, edi)       == 16, "edi wrong");
_Static_assert(offsetof(cpu_state_t, eax)       == 44, "eax wrong");
_Static_assert(offsetof(cpu_state_t, vector)    == 48, "vector wrong");
_Static_assert(offsetof(cpu_state_t, error_code) == 52, "error_code wrong");
_Static_assert(offsetof(cpu_state_t, eip)        == 56, "eip wrong");
_Static_assert(offsetof(cpu_state_t, cs)         == 60, "cs wrong");
_Static_assert(offsetof(cpu_state_t, eflags)     == 64, "eflags wrong");
_Static_assert(offsetof(cpu_state_t, user_esp)   == 68, "user_esp wrong");
_Static_assert(offsetof(cpu_state_t, user_ss)    == 72, "user_ss wrong");
```



3.2 process_t — Process Control Block

```
/* include/process.h */

#ifndef PROCESS_H

#define PROCESS_H


#include <stdint.h>

#include "page.h" /* pde_t, PAGE_SIZE */

#define MAX_PROCESSES 16

#define KERNEL_STACK_SIZE 4096 /* 4KB per-process kernel stack */

#define SCHEDULER_TICKS_PER_SLICE 5 /* 5 × 10ms = 50ms time slice */


typedef enum {

    PROCESS_UNUSED = 0, /* Slot free; pid=0; do not schedule */

    PROCESS_READY = 1, /* Can run; not currently on CPU */

    PROCESS_RUNNING = 2, /* Currently executing on this CPU */

    PROCESS_BLOCKED = 3, /* Waiting for event; skipped by scheduler */

    PROCESS_ZOMBIE = 4, /* Execution finished; waiting for cleanup */

} process_state_t;

typedef struct process {

    /* — Identity — */

    uint32_t pid; /* Offset 0: unique process ID; 0 = slot unused */

    char name[32]; /* Offset 4: null-terminated debug name */

    /* — Scheduling — */

    process_state_t state; /* Offset 36: lifecycle state */

    uint32_t ticks_remaining; /* Offset 40: time-slice countdown; reset on preemption */

    /* — Saved CPU state —

        cpu.esp holds the kernel stack pointer AT THE TIME of context_switch_asm() save.

        The full saved frame (segment regs + pusha frame + CPU interrupt frame) lies on

        the kernel stack starting at address cpu.esp. */

    uint32_t saved_esp; /* Offset 44: kernel ESP saved by context_switch_asm */

    /* — Memory — */

}
```

```

pde_t           *page_directory; /* Offset 48: physical address of this process's PD */

/* — Kernel stack —

Each process has its own 4KB kernel stack. Stack grows downward from
kernel_stack_top toward kernel_stack[0].

kernel_stack_top = (uint32_t)(uintptr_t)kernel_stack + KERNEL_STACK_SIZE */

uint32_t       kernel_stack_top; /* Offset 52: top of kernel stack (TSS ESP0 value) */

uint8_t        kernel_stack[KERNEL_STACK_SIZE] __attribute__((aligned(16)));

/* Offset 56 through 56+4095: stack storage (4096 bytes) */

/* Total PCB size: 56 + 4096 = 4152 bytes; 16 × 4152 = 66,432 bytes for full table */

} process_t;

/* Global process table and current process pointer */

extern process_t  process_table[MAX_PROCESSES];

extern process_t *current_process;

/* Public API */

process_t *process_create_kernel(const char *name, void (*entry)(void));
process_t *process_create_user (const char *name, void (*entry)(void));
void      process_exit(int exit_code);    /* Called by sys_exit; never returns */

#endif /* PROCESS_H */

```

Why `saved_esp` instead of embedding `cpu_state_t` in the PCB: The full `cpu_state_t` lives **on the kernel stack**, not in the PCB. Only the stack pointer (`saved_esp`) that points to it is stored in the PCB. This is correct: when a process is preempted, `irq_common_stub` has already pushed all the register state onto the kernel stack. `context_switch_asm` records where on that stack the state lives by saving ESP. On resume, loading `saved_esp` into ESP and executing `ret` + `popa` + `iret` restores everything from the stack. The PCB is a thin envelope; the stack is the real state store.

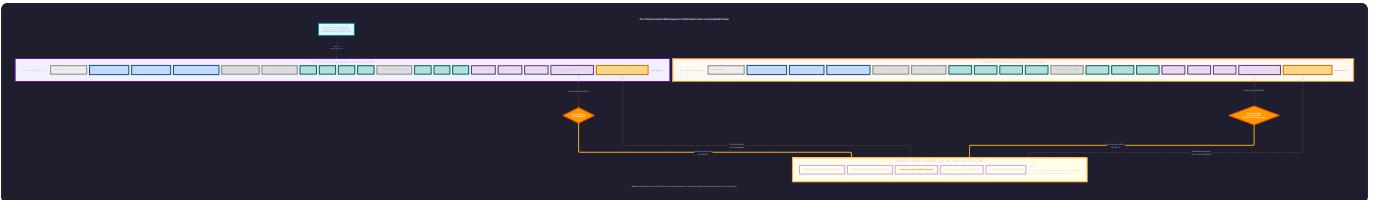
Memory layout of `process_table[]`:

```

process_table[0]:
  Offset  0: pid                  (4 bytes)
  Offset  4: name[32]             (32 bytes)
  Offset 36: state                (4 bytes)
  Offset 40: ticks_remaining     (4 bytes)
  Offset 44: saved_esp            (4 bytes)
  Offset 48: page_directory*      (4 bytes)
  Offset 52: kernel_stack_top     (4 bytes)
  Offset 56: kernel_stack[4096]   (4096 bytes)
Total:        4152 bytes

process_table[1]: starts at offset 4152
process_table[2]: starts at offset 8304
...
process_table[15]: starts at offset 62280
Total table: 16 × 4152 = 66,432 bytes ≈ 64.875 KB (in .bss)

```



3.3 `tss_t` — Task State Segment (Intel-Defined, 104 Bytes)

The TSS is a hardware-defined structure. The CPU reads `esp0` and `ss0` directly from this structure on every ring 3→ring 0 transition. The layout is fixed by Intel; the struct must be `__attribute__((packed))`.

```
/* include/tss.h */

#ifndef TSS_H

#define TSS_H


#include <stdint.h>

/* Intel IA-32 TSS descriptor (32-bit protected mode).

Hardware reads esp0/ss0 on ring-3 → ring-0 interrupt/exception.

All other fields are unused in software task-switching mode. */

typedef struct __attribute__((packed)) {

    /* Offset 0 */ uint32_t prev_tss;      /* Hardware task-link; unused in soft-switch */

    /* Offset 4 */ uint32_t esp0;        /* Ring 0 stack pointer – THE field we update */

    /* Offset 8 */ uint32_t ss0;        /* Ring 0 stack segment – always 0x10 (kernel data) */

    /* Offset 12 */ uint32_t esp1;       /* Ring 1 – unused */

    /* Offset 16 */ uint32_t ss1;        /* Ring 1 – unused */

    /* Offset 20 */ uint32_t esp2;       /* Ring 2 – unused */

    /* Offset 24 */ uint32_t ss2;        /* Ring 2 – unused */

    /* Offset 28 */ uint32_t cr3;        /* PDBR (not used by CPU here; we manage CR3 ourselves) */

    /* Offset 32 */ uint32_t eip;         /* Unused (hardware task switch fields) */

    /* Offset 36 */ uint32_t eflags;

    /* Offset 40 */ uint32_t eax;

    /* Offset 44 */ uint32_t ecx;

    /* Offset 48 */ uint32_t edx;

    /* Offset 52 */ uint32_t ebx;

    /* Offset 56 */ uint32_t esp;

    /* Offset 60 */ uint32_t ebp;

    /* Offset 64 */ uint32_t esi;

    /* Offset 68 */ uint32_t edi;

    /* Offset 72 */ uint32_t es;

    /* Offset 76 */ uint32_t cs;

    /* Offset 80 */ uint32_t ss;

    /* Offset 84 */ uint32_t ds;

    /* Offset 88 */ uint32_t fs;

    /* Offset 92 */ uint32_t gs;
}
```

C

```

/* Offset 96 */ uint32_t ldt;           /* LDT selector – unused */

/* Offset 100 */ uint16_t debug_trap; /* T bit for debug exceptions – set 0 */

/* Offset 102 */ uint16_t iomap_base; /* I/O permission bitmap offset */

/* Total: 104 bytes */

} tss_t;

_Static_assert(sizeof(tss_t) == 104, "TSS must be exactly 104 bytes");

extern tss_t kernel_tss;

void tss_init(void);

void tss_set_kernel_stack(uint32_t esp0); /* Called on every context switch */

#endif /* TSS_H */

```

TSS GDT descriptor format (system descriptor, S=0):

The TSS occupies GDT entry index 5 (selector `0x28`). Its GDT descriptor differs from code/data descriptors: the `S` bit (descriptor type) is 0 (system descriptor), and the type field is `0x9` (available 32-bit TSS).

```

TSS GDT Entry (8 bytes), at GDT index 5:
Byte: 7      6      5      4      3-2      1-0
+-----+-----+-----+-----+-----+-----+
|Base   |Flags+ |Access |Base   |Base   |Limit |
|[31:24]|Lim high| Byte  |[23:16]| [15:0]| [15:0]|
+-----+-----+-----+-----+-----+-----+

```

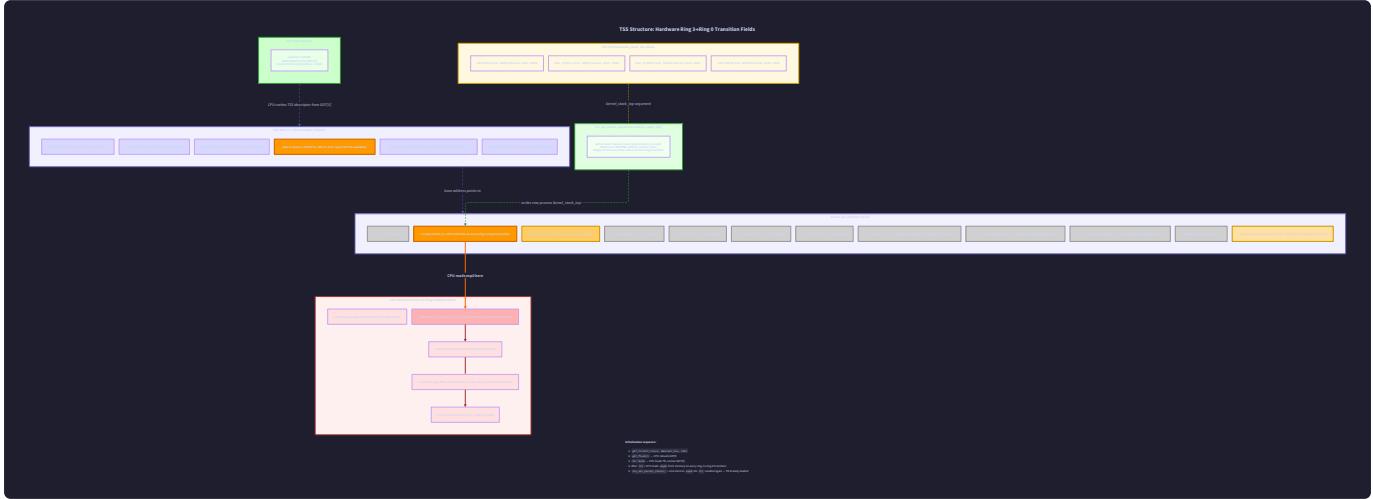
Access byte for TSS: `0x89`
`P=1` (present)
`DPL=0` (ring 0 only; CPU verifies `CPL ≤ DPL` for explicit task switch via `JMP/CALL TSS`)
`S=0` (system descriptor – not code/data)
`Type=1001 = 0x9` (available 32-bit TSS; "busy" would be `0xB`)

Granularity byte: `0x00`
`G=0` (byte granularity; limit is in bytes, not 4KB units)
`DB=0` (not applicable for TSS)
`L=0`
`Limit high nibble = (sizeof(tss_t) - 1) >> 16 = 0` (`limit = 103 < 65535`)

`Limit[15:0] = sizeof(tss_t) - 1 = 103 = 0x0067`

TSS selector value: Index 5, GDT (TI=0), RPL=0 → `(5 << 3) | 0 = 0x28`.

ltr instruction: Loads the Task Register with the TSS selector. The CPU caches the TSS descriptor internally. After `ltr 0x28`, the CPU reads `esp0 / ss0` from the memory address pointed to by the cached TSS base on every ring transition — it does not re-read the GDT. Therefore `tss_set_kernel_stack()` writes directly to `kernel_tss.esp0` and the CPU sees the new value immediately.



3.4 GDT Extension — Entry 5 for TSS

Module mod-1 defined 5 GDT entries (indices 0–4). This module adds entry index 5. The GDT array in `gdt.c` must be expanded from 5 to 6 entries, and `gdt_flush()` (which reloads GDTR) must be called after the new entry is written.

The GDT structure used in mod-1 (`gdt_entry_t` or equivalent) must support writing index 5. If the GDT array was declared `static gdt_entry_t gdt[5]`, change it to `static gdt_entry_t gdt[6]` and re-export the symbol or add `gdt_install_tss()` as a function in `gdt.c`.

```
/* In gdt.c (mod-1 file, extended for mod-4) */

/* gdt_install_tss: install a 32-bit TSS descriptor at the given GDT index.

base: physical address of the tss_t structure.

limit: sizeof(tss_t) - 1 = 103.

After installation, caller must call gdt_flush() and then ltr. */

void gdt_install_tss(int index, uint32_t base, uint32_t limit) {

    gdt[index].limit_low     = (uint16_t)(limit & 0xFFFF);
    gdt[index].base_low      = (uint16_t)(base  & 0xFFFF);
    gdt[index].base_middle   = (uint8_t)((base  >> 16) & 0xFF);
    gdt[index].access        = 0x89; /* P=1, DPL=0, S=0, Type=0x9 (TSS available) */
    gdt[index].granularity   = (uint8_t)((limit >> 16) & 0x0F); /* G=0, DB=0, L=0 */
    gdt[index].base_high     = (uint8_t)((base  >> 24) & 0xFF);

    /* Do NOT set granularity bit (G=0): limit is in bytes, not 4KB pages */

}
```

Critical difference from code/data descriptors: Code/data descriptors use `granularity = 0xCF` (G=1, DB=1, plus limit high nibble). The TSS descriptor uses `granularity = 0x00` (G=0; limit is byte-exact at 103). Using `0xCF` for the TSS descriptor multiplies the limit by 4096, creating a 404KB "TSS" — the CPU does not fault, but subsequent access to the IOMAP region may be misinterpreted.

3.5 `irq_return_trampoline` and Stack Frame at First Activation

When a brand-new process is scheduled for the first time, `context_switch_asm` executes `ret` — this pops a return address from the (fake) kernel stack. That return address must land somewhere that continues the `irq_common_stub` exit sequence: pop segment registers, `popa`, `add esp, 8`, `iret`.

The trampoline is a label placed in `context_switch.asm` immediately before the segment register restoration sequence that `irq_common_stub` would normally execute after returning from `irq_dispatcher`. Since `irq_common_stub` is in `irq_stubs.asm` (mod-2), we cannot place a label there after the fact. Instead, we define `irq_return_trampoline` in `context_switch.asm` as a self-contained exit sequence:

```
; In context_switch.asm                                     NASM
global irq_return_trampoline

irq_return_trampoline:
; This code executes after context_switch_asm's 'ret' pops this address.
; At this point ESP points to the saved state of the NEW process:
; [esp+0] gs      (uint32_t)
; [esp+4] fs
; [esp+8] es
; [esp+12] ds
; [esp+16..44] pusha regs (EDI,ESI,EBP,ESP_PUSHA,EBX,EDX,ECX,EAX)
; [esp+48] vector (dummy)
; [esp+52] error_code (dummy)
; [esp+56] EIP    (process entry point)
; [esp+60] CS     (0x08 kernel or 0x1B user)
; [esp+64] EFLAGS (0x202, IF=1)
; [esp+68] user_esp (only if ring-3 process)
; [esp+72] user_ss (only if ring-3 process)

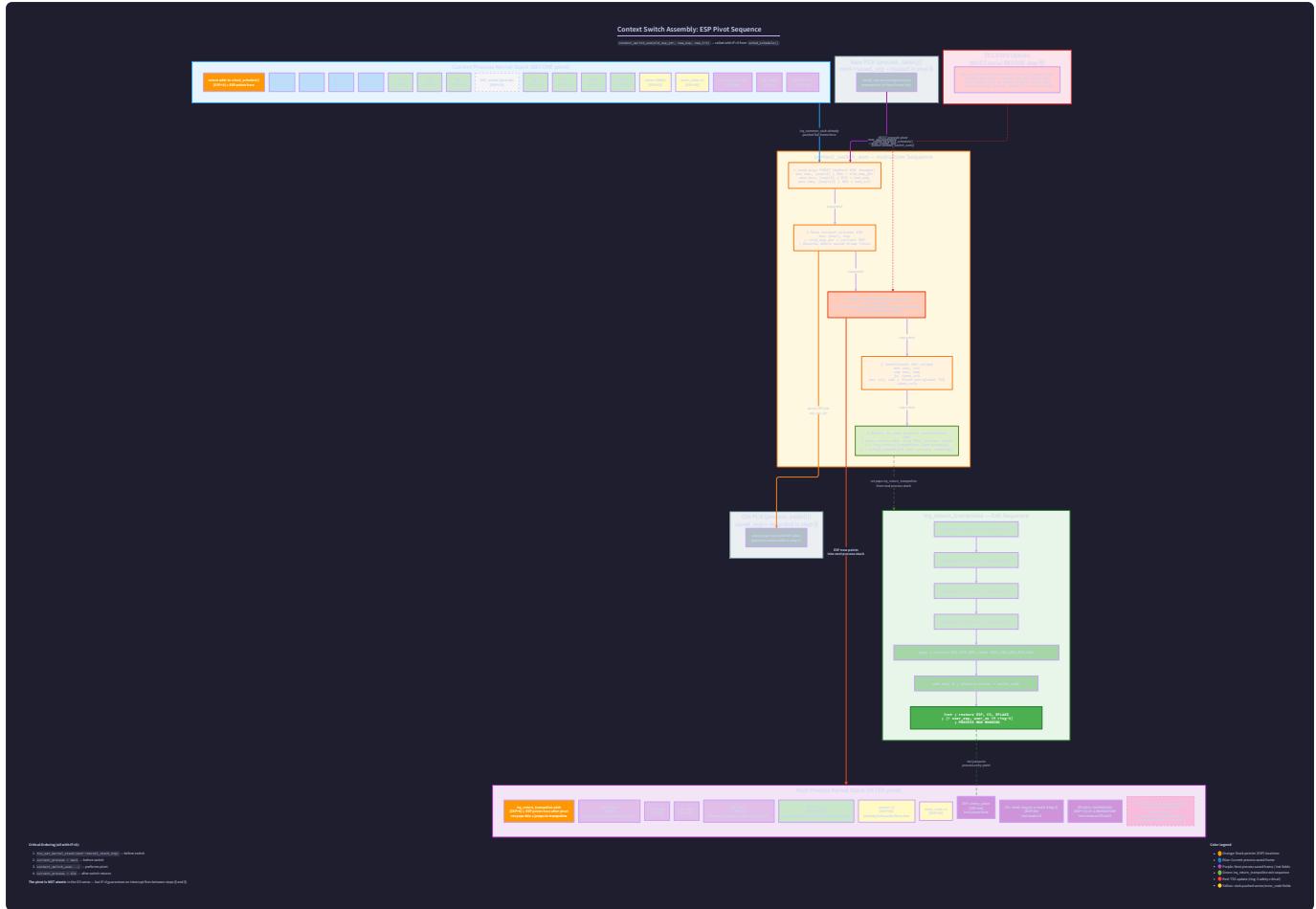
; Restore segment registers
pop eax
mov ds, ax
pop eax
mov es, ax
pop eax
mov fs, ax
pop eax
mov gs, ax

popa          ; Restore EDI,ESI,EBP,(ESP ignored),EBX,EDX,ECX,EAX

add esp, 8    ; Discard vector + error_code

iret          ; Restore EIP, CS, EFLAGS [, user_esp, user_ss if ring-3]
; The process now runs at its entry point with IF=1.
```

Why not reuse `irq_common_stub`'s exit path directly? `irq_common_stub` calls `irq_dispatcher` and then has its exit sequence. We cannot jump into the middle of it safely (the `call irq_dispatcher / add esp, 4` pair would be skipped, but so would the segment register state at that point). Duplicating the exit sequence in `irq_return_trampoline` is cleaner and self-contained. The 12 instructions are identical to the corresponding portion of `irq_common_stub`.



3.6 Fake Initial Stack Frame Layout

When `process_create_kernel()` or `process_create_user()` constructs a new process's initial kernel stack, it must lay out values so that when `context_switch_asm` loads `saved_esp` and executes `ret`, followed by `irq_return_trampoline`, the result is identical to what would happen if the process had been preempted by an IRQ and is now being resumed.

Construction: push values from `kernel_stack_top` downward (decrement pointer before each write):

Address (relative to kernel_stack_top) Contents

kernel_stack_top - 0 [never used; stack grows down from here]

RING-3 PROCESS ONLY (pushed first = highest address = popped last by iret):

kernel_stack_top - 4 user_ss = 0x23 (GDT index 4, RPL 3: (4<<3)|3 = 0x23)
kernel_stack_top - 8 user_esp = 0x00C00000 (top of user stack virtual address)

ALL PROCESSES (CPU-pushed interrupt frame fields; iret consumes these):

kernel_stack_top - 12 eflags = 0x00000202 (IF=1, bit1=1 per Intel spec)
 (ring-3: -16) kernel_stack_top - 16 cs = 0x1B (ring-3) or 0x08 (ring-0)
 (ring-0: -12)
kernel_stack_top - 20 eip = (uint32_t)entry [ring-3: 0x00400000]
 (ring-0: -16)

STUB-PUSHED FIELDS (add esp, 8 discards these):

 -24 error_code = 0
 -28 vector = 0

pusha FIELDS (popa restores these; esp_pusha field is ignored by popa):

 -32 eax = 0
 -36 ecx = 0
 -40 edx = 0
 -44 ebx = 0
 -48 esp_pusha = 0 (ignored by popa hardware)
 -52 ebp = 0
 -56 esi = 0
 -60 edi = 0

SEGMENT REGISTER FIELDS (popped and loaded before popa):

 -64 ds = 0x10
 -68 es = 0x10
 -72 fs = 0x10
 -76 gs = 0x10

TRAMPOLINE RETURN ADDRESS (context_switch_asm's 'ret' pops this):

 -80 (uint32_t)irq_return_trampoline

process->saved_esp = kernel_stack_top - 80 (kernel process)

process->saved_esp = kernel_stack_top - 84 (ring-3 process: two extra words for user_ss+user_esp)

Exact byte diagram for a kernel-mode process (19 values × 4 bytes = 76 bytes used, saved_esp = kernel_stack_top - 80 counting the return address):

```

kernel_stack_top (highest address):
[esp+76] eflags = 0x00000202 ← iret reads this last → bit 9 (IF) = 1 MANDATORY
[esp+72] cs     = 0x00000008 ← iret reads this; DPL check: ring 0
[esp+68] eip     = entry_func ← iret jumps here
[esp+64] error_code = 0
[esp+60] vector = 0
[esp+56] eax    = 0
[esp+52] ecx    = 0
[esp+48] edx    = 0
[esp+44] ebx    = 0
[esp+40] esp_pusha = 0 (popa ignores; advances 4 bytes but does not load ESP)
[esp+36] ebp    = 0
[esp+32] esi    = 0
[esp+28] edi    = 0
[esp+24] ds     = 0x00000010
[esp+20] es     = 0x00000010
[esp+16] fs     = 0x00000010
[esp+12] gs     = 0x00000010
[esp+ 8] (not yet used; trampoline pops gs first, which is at [esp+0] when trampoline starts)

```

Wait – recount: trampoline starts with esp pointing to the first field ABOVE the return address. After context_switch_asm's 'ret' pops the trampoline address, esp advances by 4. The trampoline code then executes:

```

pop eax; mov ds, ax → pops [esp+0] = ds field
pop eax; mov es, ax → pops [esp+0] = es field (after first pop)
...

```

So the frame when trampoline executes (after 'ret' popped return address):

```

[esp+ 0] gs = 0x10 ← first pop in trampoline: pop eax; mov gs, ax
[esp+ 4] fs = 0x10
[esp+ 8] es = 0x10
[esp+12] ds = 0x10
[esp+16] edi = 0 ← popa starts here
[esp+20] esi = 0
[esp+24] ebp = 0
[esp+28] esp_pusha = 0 (ignored)
[esp+32] ebx = 0
[esp+36] edx = 0
[esp+40] ecx = 0
[esp+44] eax = 0 ← popa ends here (eax last)
[esp+48] vector = 0 ← add esp, 8 skips this and error_code
[esp+52] error_code = 0
[esp+56] eip = entry ← iret pops EIP
[esp+60] cs = 0x08 ← iret pops CS
[esp+64] eflags      ← iret pops EFLAGS (IF=1 from bit 9)

```

saved_esp points to [esp+0] = gs field address.

Construction in C (kernel process):

```

uint32_t *sp = (uint32_t *)(uintptr_t)proc->kernel_stack_top;

/* Push in reverse order (highest address first) */

*--sp = 0x000000202;           /* eflags: IF=1, reserved bit 1 set */
*--sp = 0x00000008;           /* cs: kernel code selector */
*--sp = (uint32_t)entry;        /* eip: process entry point */
*--sp = 0;                     /* error_code: dummy */
*--sp = 0;                     /* vector: dummy */

/* pusha frame (popa order: EDI-ESI-EBP-(skip ESP)-EBX-EDX-ECX-EAX) */

*--sp = 0; /* eax */
*--sp = 0; /* ecx */
*--sp = 0; /* edx */
*--sp = 0; /* ebx */
*--sp = 0; /* esp_pusha (ignored by popa) */
*--sp = 0; /* ebp */
*--sp = 0; /* esi */
*--sp = 0; /* edi */

/* Segment registers (trampoline pops: gs, fs, es, ds) */

*--sp = 0x00000010; /* ds */
*--sp = 0x00000010; /* es */
*--sp = 0x00000010; /* fs */
*--sp = 0x00000010; /* gs */

/* Return address for context_switch_asm's 'ret' */

extern void irq_return_trampoline(void);

*--sp = (uint32_t)(uintptr_t)irq_return_trampoline;

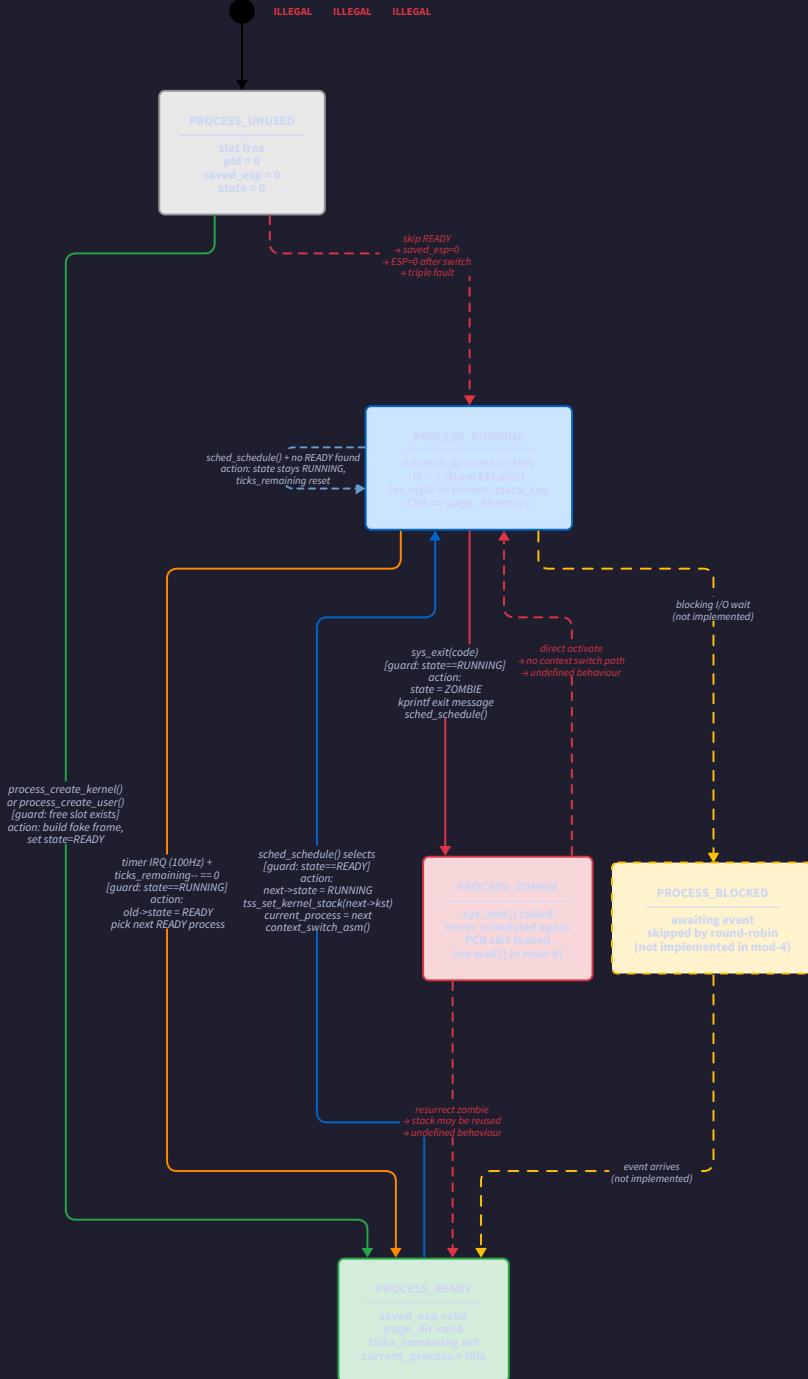
proc->saved_esp = (uint32_t)(uintptr_t)sp;

```

Push order vs pop order: We push from the top of the stack downward. The trampoline pops upward (lowest address first). The trampoline's first `pop eax; mov gs, ax` pops the value at the lowest address — which must be the GS value. Work backward from what the trampoline expects and push accordingly. The `gs` field must be at the lowest address after all pushes are complete.

Process State Machine: Lifecycle and Scheduler Transitions

Module mod-4 · Single-core preemptive x86 kernel · Round-robin 50ms slices



Scheduler Invariant (enforced while IF=0):

- Exactly one process has `state == RUNNING`
 - That process equals `current_process`
 - `tss.esp0` equals `current_process->kernel_stack_top`
 - All other schedulable processes have `state == READY`
 - Round-robin slice = 5 ticks \times 10ms = 50ms

3.7 context_switch_asm — Arguments and Stack Contract

```
; context_switch_asm(uint32_t *old_esp_ptr, uint32_t new_esp, uint32_t new_cr3)
; cdecl calling convention:
; [esp+4] = old_esp_ptr (address of proc->saved_esp for the current process)
; [esp+8] = new_esp      (value of proc->saved_esp for the next process)
; [esp+12] = new_cr3    (physical address of next process's page directory)
;
; Called FROM irq_common_stub (via irq_dispatcher → sched_schedule → context_switch_asm).
; On entry: esp points to the return address pushed by the 'call context_switch_asm' in sched.c.
; The full saved state of the current process is on the stack ABOVE this return address.
```

NASM

Why all arguments must be loaded before ESP is modified: The `cdecl` argument addresses are relative to the current `esp`. Loading `[esp+4]`, `[esp+8]`, `[esp+12]` after `mov esp, new_esp` would read from the `new` stack — which contains the next process's state, not the call arguments. The solution: load all three into registers before touching `esp`.

```
global context_switch_asm
context_switch_asm:
    ; Step 1: Load ALL arguments into registers BEFORE modifying ESP
    mov eax, [esp + 4]      ; EAX = old_esp_ptr (address of current proc->saved_esp)
    mov ecx, [esp + 8]      ; ECX = new_esp      (next process's saved stack pointer)
    mov edx, [esp + 12]     ; EDX = new_cr3    (next process's page directory physical addr)

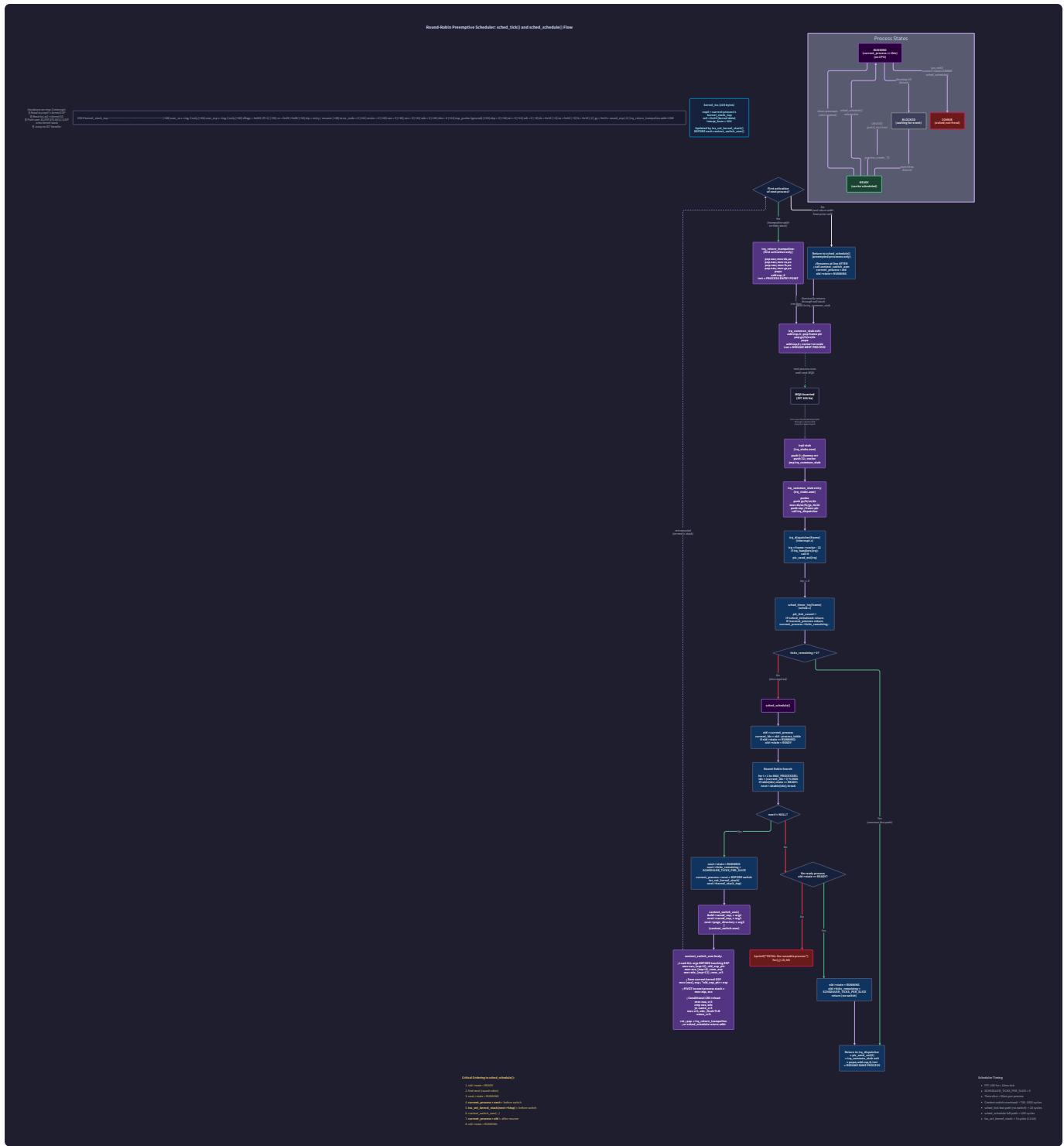
    ; Step 2: Save current kernel stack pointer into current process's PCB
    ; ESP currently points to the return address of this call.
    ; The full IRQ frame (pusha + segments + CPU interrupt frame) lies above on the stack.
    ; Saving ESP here records the complete saved state's location.
    mov [eax], esp          ; *old_esp_ptr = current esp

    ; Step 3: Switch to next process's kernel stack
    mov esp, ecx            ; ESP now points into the NEXT process's kernel stack.
                            ; From this point: the current process is suspended.
                            ; All subsequent stack operations affect the next process's state.

    ; Step 4: Conditionally reload CR3 (only if address space changes)
    ; Comparing prevents flushing TLB when two kernel processes share boot_page_directory.
    mov eax, cr3
    cmp eax, edx
    je .same_cr3
    mov cr3, edx           ; Load new page directory – flushes all non-global TLB entries
.same_cr3:

    ; Step 5: Return to the next process's continuation
    ; 'ret' pops the return address that was pushed when this function was called
    ; FROM THE NEXT PROCESS'S PERSPECTIVE. For a process that was previously preempted,
    ; this return address is the address inside sched_schedule() after 'call context_switch_asm'.
    ; For a brand-new process, this is irq_return_trampoline (pushed in fake initial frame).
    ret
    ; After ret: execution is in sched_schedule() (for old processes) or irq_return_trampoline
    ; (for new processes). Either path eventually executes popa + iret to resume the process.
```

Hardware soul — the ESP pivot: `mov esp, ecx` is a single instruction that atomically changes the CPU's active stack. No interrupt can fire between this instruction and the `ret` — the interrupt gate keeps IF=0 throughout the IRQ handler chain. Even if it could, the `ret` immediately follows: the next process's continuation is restored before any problematic state exists.



3.8 Scheduler Data Structures

```
/* sched.c internal state */

static int sched_initialized = 0;

/* Round-robin position: last index scheduled.

   Next search starts from (last_index + 1) % MAX_PROCESSES. */

static int sched_last_index = 0;

/* Ticks-per-slice constant defined in process.h:

#define SCHEDULER_TICKS_PER_SLICE 5    (5 × 10ms = 50ms) */
```

Round-robin selection invariant: The scheduler always searches forward from `(current_index + 1) % MAX_PROCESSES`, wrapping around. It skips slots with `state != PROCESS_READY`. If no ready process is found after a full scan (all processes blocked or zombie), it keeps `current_process` running (if it is still `PROCESS_RUNNING`). If `current_process` is zombie or blocked and no other ready process exists: fatal halt (degenerate case not reached in this module since the idle process always exists in `PROCESS_RUNNING` or `PROCESS_READY`).

3.9 System Call Numbers and Dispatch Table

```
/* include/syscall.h */

#define SYS_EXIT      1
#define SYS_WRITE     4

/* Syscall frame is the same interrupt_frame_t from mod-2.

   EAX = syscall number; EBX = arg1; ECX = arg2; EDX = arg3.

   Return value written to frame->eax before returning from syscall_dispatch. */

void syscall_dispatch(interrupt_frame_t *frame);
```

Argument register convention (Linux-compatible INT 0x80 ABI):

Register	Role
EAX	Syscall number
EBX	First argument (fd, exit code)
ECX	Second argument (buffer virtual address)
EDX	Third argument (byte count)
Return	EAX (written to frame->eax; restored by iret)

4. Interface Contracts

4.1 tss_init(void) → void

Pre-conditions: GDT array has at least 6 slots. Protected mode active. `ldt` can be issued from ring 0.

Behavior:

1. Zero `kernel_tss` entirely: `memset(&kernel_tss, 0, sizeof(tss_t))`.
2. Set `kernel_tss.ss0 = 0x10` (kernel data segment selector).
3. Set `kernel_tss.esp0 = 0` — will be updated on first context switch before any ring-3 interrupt fires.
4. Set `kernel_tss.iomap_base = sizeof(tss_t)` — sets the I/O permission bitmap to begin immediately after the TSS, which combined with the TSS limit of `sizeof(tss_t) - 1` means all I/O ports are restricted for ring-3 code. (Ring-0 code ignores IOPL when it is 0 if IOPL ≥ CPL; kernel code can always execute `in / out` because CPL=0.)
5. Call `gdt_install_tss(5, (uint32_t)(uintptr_t)&kernel_tss, sizeof(tss_t) - 1)`.
6. Call `gdt_flush()` — reloads GDTR so CPU sees the new GDT entry 5.
7. Execute `ltr 0x28` via inline assembly: `__asm__ volatile ("ltr %0" : : "r"((uint16_t)0x28))`.
8. `kprintf("[TSS] Initialized. TSS at 0x%08x, selector 0x28.\n", &kernel_tss)`.

Post-conditions: `TR` register contains selector `0x28`. CPU will read `kernel_tss.esp0` on every ring-3 → ring-0 transition. `kernel_tss.ss0 = 0x10`.

Errors:

- `gdt_install_tss` with index 5 on a GDT array of size 5: out-of-bounds write. Prevention: verify GDT array size ≥ 6 before calling (static assertion or runtime check).
- `ltr` with non-present TSS selector: `#GP`. Prevention: `gdt_flush()` before `ltr` guarantees the CPU's GDT cache is current.

4.2 tss_set_kernel_stack(uint32_t esp0) → void

Pre-conditions: `tss_init()` has been called. `ltr` has been executed.

Behavior: `kernel_tss.esp0 = esp0`. That is the entire function body. One 32-bit memory write.

Why only one write: After `ltr`, the CPU caches the TSS descriptor internally. When an interrupt/exception fires while in ring 3, the CPU reads `esp0` from the physical address of `kernel_tss.esp0` directly — it uses the cached TSS base address but fetches the `esp0` field fresh from memory on each ring transition. Therefore a simple store to `kernel_tss.esp0` is sufficient and takes effect on the very next ring transition.

Timing constraint: `tss_set_kernel_stack()` must be called BEFORE `context_switch_asm()` pivots to the new process, not after. Sequence in `sched_schedule()`:

1. Select next process.
2. `tss_set_kernel_stack(next->kernel_stack_top)`. ← BEFORE the switch
3. `context_switch_asm(...)`.

Errors: None. A write to a known valid address in the kernel BSS section cannot fail.

4.3 process_create_kernel(const char *name, void (*entry)(void)) → process_t *

Parameters:

- `name` : Null-terminated string ≤ 31 chars. NULL: behavior undefined (caller must not pass NULL).

- `entry` : Function pointer to the process's entry point. Must be a valid kernel code address. `NULL`: process will page-fault or execute at address 0 on first activation.

Returns: Pointer to the initialized `process_t` in `PROCESS_READY` state, or `NULL` if `process_table` is full.

Behavior:

1. Find the first slot where `process_table[i].state == PROCESS_UNUSED`. If none: return `NULL`.
2. `memset(proc, 0, sizeof(process_t))`.
3. `proc->pid = next_pid++ (static uint32_t next_pid = 1)`.
4. Copy name: `for (i = 0; i < 31 && name[i]; i++) proc->name[i] = name[i]; proc->name[31] = '\0'`.
5. `proc->state = PROCESS_READY`.
6. `proc->ticks_remaining = SCHEDULER_TICKS_PER_SLICE`.
7. `proc->page_directory = boot_page_directory` (shared kernel page directory; extern from `vmm.c`).
8. `proc->kernel_stack_top = (uint32_t)(uintptr_t)proc->kernel_stack + KERNEL_STACK_SIZE`.
9. Build fake initial stack frame (§3.6 kernel variant) — pushes 19 values totaling 76 bytes below `kernel_stack_top`.
10. `proc->saved_esp = (uint32_t)(uintptr_t)sp` where `sp` is the final stack pointer after all pushes.

Post-conditions: `proc->state == PROCESS_READY`. `proc->saved_esp` points to the `gs` field of the fake frame. `proc->kernel_stack_top` is 16-byte aligned (guaranteed by `__attribute__((aligned(16)))` on `kernel_stack[]`). The scheduler's next call to `sched_schedule()` may pick this process.

Errors: `process_table` full: return `NULL`. Caller must check for `NULL` and print a fatal message.

4.4 `process_create_user(const char *name, void (*entry)(void)) → process_t *`

Parameters: Same as `process_create_kernel`. `entry` is the address of a **kernel-resident function** that will be mapped into the user address space (the function's code is copied to a user-accessible physical frame).

Returns: Pointer to the initialized `process_t` in `PROCESS_READY` state, or `NULL` on failure.

Behavior:

1. Allocate a PCB slot (same as `process_create_kernel` steps 1–8, but ring-3 initial frame in step 9).
2. **Allocate a new page directory:**

```
pde_t *user_pd = (pde_t *)pmm_alloc_frame();

if (!user_pd) { proc->state = PROCESS_UNUSED; return NULL; }

memset(user_pd, 0, PAGE_SIZE);
```

3. **Copy kernel PDE entries (indices 768–1023):**

```
extern pde_t boot_page_directory[];

for (int i = 768; i < 1024; i++) {
    user_pd[i] = boot_page_directory[i];
}
```

These entries point to kernel page tables with U/S=0 — user-mode access to any of these addresses triggers a page fault (protection violation, error code P=1, U=1). 4. **Allocate and map user code frame at virtual `0x00400000` :**

```

void *user_code_phys = pmm_alloc_frame();

if (!user_code_phys) { pmm_free_frame(user_pd); proc->state = PROCESS_UNUSED; return NULL; }

memcpy(user_code_phys, (void *)entry, PAGE_SIZE);

vmm_map_page(user_pd, 0x00400000, (uint32_t)(uintptr_t)user_code_phys,
             PAGE_PRESENT | PAGE_USER);

/* Read-only: no PAGE_WRITABLE on code page */

```

5. Allocate and map user stack frame at virtual `0x00BFF000` :

```

void *user_stack_phys = pmm_alloc_frame();

if (!user_stack_phys) { /* free all, return NULL */ }

memset(user_stack_phys, 0, PAGE_SIZE);

vmm_map_page(user_pd, 0x00BFF000, (uint32_t)(uintptr_t)user_stack_phys,
             PAGE_PRESENT | PAGE_WRITABLE | PAGE_USER);

```

6. `proc->page_directory = user_pd`.

7. Build **ring-3 fake initial stack frame** (§3.6 user-mode variant):

- Push `user_ss = 0x23` and `user_esp = 0x00C00000` (top of the mapped stack page + 1 page).
- Then push `eflags = 0x202`, `cs = 0x1B`, `eip = 0x00400000`.
- Then the standard dummy fields, pusha fields, segment fields, trampoline address.

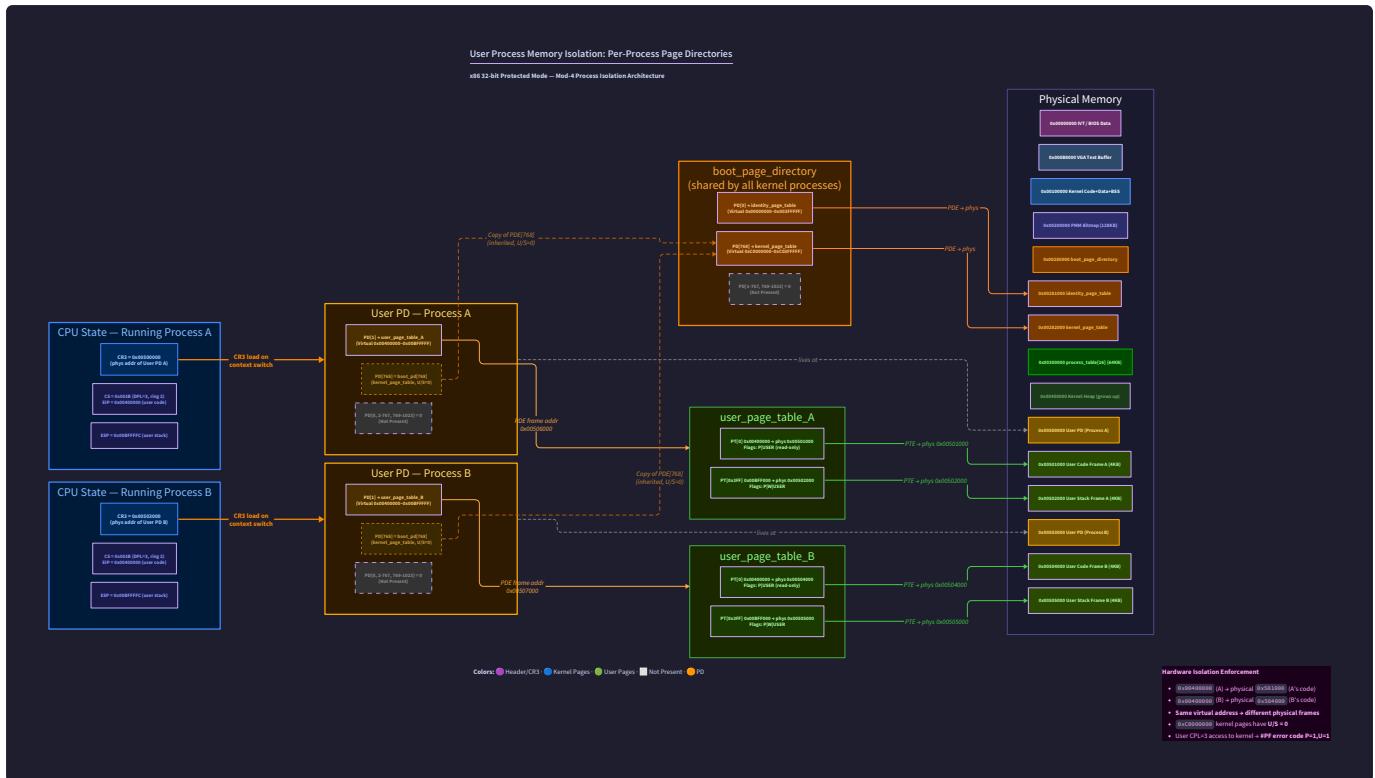
8. `proc->saved_esp = (uint32_t)(uintptr_t)sp`.

User address space layout:

Virtual Address	Mapping	Flags	Notes
0x00000000 - 0x003FFFFF	(unmapped)	-	Null-pointer trap; no PDE present
0x00400000 - 0x00400FFF	User code	P U	Copy of 'entry' function's 4KB page
0x00401000 - 0x00BFFFFFFF	(unmapped)	-	No mapping
0x00BFF000 - 0x00BFFFFFFF	User stack	P W U	4KB stack; grows down from 0xC00000
0x00C00000 - 0xBFFFFFFF	(unmapped)	-	
0xC0000000 - 0xC03FFFFFFF	Kernel code	P W (U/S=0)	Inherited from boot_page_directory[768]

Why `user_esp = 0x00C00000` (one page above the stack frame)? The stack grows downward. `0x00C00000` is the first byte **above** the mapped stack page at `0x00BFF000`. A push from user code will first decrement ESP to `0x00BFFFC` and then write — within the mapped page. If we set `user_esp = 0x00BFFFFFFF` (top of the page), the first push would write to `0x00BFFFFB` — also valid, but the conventional boundary is the address one past the allocation end.

Errors: Any `pmm_alloc_frame()` failure: free all previously allocated frames, set `proc->state = PROCESS_UNUSED`, return `NULL`. The caller handles the `NULL` return.



4.5 sched_init(void) → void

Pre-conditions: `process_table[]` is in BSS (zeroed). `pit_init(100)` has been called. `irq_install_handler` is available.

Behavior:

- For `i` in `[0, MAX_PROCESSES]`: `process_table[i].state = PROCESS_UNUSED`.
- `current_process = NULL`.
- `sched_initialized = 1`.
- `irq_install_handler(0, sched_timer_irq)` — replace the mod-3 timer handler with the scheduler's handler.
- `kprintf("[SCHED] Initialized. %d process slots, %d-tick slice.\n", MAX_PROCESSES, SCHEDULER_TICKS_PER_SLICE)`.

Post-conditions: `sched_initialized == 1`. IRQ0 handler points to `sched_timer_irq`. `current_process == NULL` (set to idle process by `kmain` before `sti`).

4.6 sched_timer_irq(interrupt_frame_t *frame) → void

Pre-conditions: Called from `irq_dispatcher` on every IRQ0. IF=0 (interrupt gate). `frame` contains the interrupted process's CPU state.

Behavior:

- `pit_tick_count++` (maintain compatibility with existing tick counter from mod-2/mod-3).
- If `!sched_initialized || !current_process`: return immediately (scheduler not yet ready).
- Decrement `current_process->ticks_remaining`.
- If `current_process->ticks_remaining > 0`: return (slice not expired).
- Call `sched_schedule()`.

Return: Returns normally (no return value). EOI sent by `irq_dispatcher` after this function returns.

4.7 sched_schedule(void) → void

Pre-conditions: `current_process` is not NULL. Called with IF=0 (either from timer IRQ handler or from `sched_yield()` which must first call `cli`).

Behavior:

```
Step 1: Determine current process index
    current_idx = (int)(current_process - process_table)

Step 2: Transition current process state
    If current_process->state == PROCESS_RUNNING:
        current_process->state = PROCESS_READY

Step 3: Find next ready process (round-robin)
    next = NULL
    For i = 1 to MAX_PROCESSES (inclusive):
        idx = (current_idx + i) % MAX_PROCESSES
        If process_table[idx].state == PROCESS_READY:
            next = &process_table[idx]
            break

Step 4: If no ready process found
    If next == NULL:
        If current_process->state == PROCESS_READY (we just set it):
            current_process->state = PROCESS_RUNNING (keep running)
            current_process->ticks_remaining = SCHEDULER_TICKS_PER_SLICE
            return (no switch; no TSS update needed)
        Else (current is zombie or blocked):
            kprintf("[SCHED] FATAL: No runnable process!\n"); for();;

Step 5: Transition next process state
    next->state = PROCESS_RUNNING
    next->ticks_remaining = SCHEDULER_TICKS_PER_SLICE

Step 6: Update TSS.ESP0 BEFORE context switch
    tss_set_kernel_stack(next->kernel_stack_top)

Step 7: Perform context switch
    context_switch_asm(
        &current_process->saved_esp, /* Where to save old kernel ESP */
        next->saved_esp, /* New kernel ESP to load */
        (uint32_t)(uintptr_t)next->page_directory /* New CR3 */
    )

Step 8: Update current_process AFTER switch (executes when THIS process resumes)
    current_process = next
    [Actually: current_process must be updated BEFORE the context switch so that
     the next process's timer handler finds the right current_process.
     See note below.]
```

current_process update timing — the critical subtlety: `context_switch_asm` suspends the current process and resumes the next. When the next process later gets preempted and `sched_timer_irq` fires, it reads `current_process` to find the currently-running process. If `current_process` still points to the **old** process at the time the **new** process runs, `sched_schedule()` would compute the wrong `current_idx` and the wrong next-process search. Therefore `current_process = next` must be set **before** `context_switch_asm()`. The correct sequence:

```
process_t *old = current_process;

current_process = next;           /* Update BEFORE switch */

tss_set_kernel_stack(next->kernel_stack_top);

context_switch_asm(&old->saved_esp, next->saved_esp,
                   (uint32_t)(uintptr_t)next->page_directory);

/* When we return here: the old process is running again.

   current_process now correctly points to 'old' – but wait,
   we set current_process = next before switching. When this
   process resumes, current_process should be 'old', not 'next'.

   The fix: set current_process = old after the switch returns. */

current_process = old;          /* Restore after we resume */
```

The correct complete pattern:

void sched_schedule(void) { C

```
process_t *old = current_process;

int current_idx = (int)(old - process_table);

if (old->state == PROCESS_RUNNING) old->state = PROCESS_READY;

process_t *next = NULL;

for (int i = 1; i <= MAX_PROCESSES; i++) {

    int idx = (current_idx + i) % MAX_PROCESSES;

    if (process_table[idx].state == PROCESS_READY) {

        next = &process_table[idx];

        break;
    }
}

if (!next) {

    if (old->state == PROCESS_READY) {

        old->state = PROCESS_RUNNING;

        old->ticks_remaining = SCHEDULER_TICKS_PER_SLICE;

        return;
    }

    kprintf("[SCHED] FATAL: No runnable process!\n"); for(;;);
}

next->state = PROCESS_RUNNING;

next->ticks_remaining = SCHEDULER_TICKS_PER_SLICE;

/* Order: current_process → tss_set_kernel_stack → context_switch_asm */

current_process = next;

tss_set_kernel_stack(next->kernel_stack_top);

context_switch_asm(&old->saved_esp, next->saved_esp,

                  (uint32_t)(uintptr_t)next->page_directory);

/* Resumes here when 'old' is rescheduled */

current_process = old;

old->state = PROCESS_RUNNING;
```

```
}
```

Why this two-assignment pattern? When the scheduler runs process A and switches to B, it sets `current_process = B` before the switch. While B runs (for 50ms or more), `current_process == B` correctly. When B's timer fires, `sched_schedule()` picks C, sets `current_process = C`, switches. Eventually A's saved `context_switch_asm` return point executes: `current_process = old (A); A->state = RUNNING`. This correctly restores A's identity as the currently running process.

4.8 `sched_yield(void) → void`

Behavior: Force an immediate context switch without waiting for the timer slice to expire. Called by voluntary cooperators (e.g., idle process, blocking wait).

```
void sched_yield(void) {  
    __asm__ volatile ("cli");      /* Disable interrupts — sched_schedule requires IF=0 */  
  
    current_process->ticks_remaining = 0;  
  
    sched_schedule();  
  
    __asm__ volatile ("sti");      /* Re-enable; or rely on iret restoring IF from EFLAGS */  
  
}
```

C

Note: `sti` after `sched_schedule()` may be redundant — `iret` inside the context switch restores `EFLAGS` which includes `IF=1`. The explicit `sti` is a defensive measure.

4.9 `syscall_dispatch(interrupt_frame_t *frame) → void`

Pre-conditions: Called from the INT 0x80 ISR stub via `isr_common_stub`. `frame` is a valid interrupt frame. `current_process` is not NULL. `CS` in the frame has DPL=3 (user-mode call) or DPL=0 (kernel testing). Ring-3 code reached this via the `IDT_SYSCALL_GATE` (DPL=3 trap gate at vector 0x80).

Behavior:

```

void syscall_dispatch(interrupt_frame_t *frame) {
    uint32_t num = frame->eax;
    uint32_t arg1 = frame->ebx;
    uint32_t arg2 = frame->ecx;
    uint32_t arg3 = frame->edx;

    switch (num) {
        case SYS_WRITE:
            frame->eax = sys_write_impl(arg1, (const char *) (uintptr_t) arg2, arg3);
            break;
        case SYS_EXIT:
            sys_exit_impl((int) arg1);
            /* sys_exit_impl never returns */
            break;
        default:
            kprintf("[SYSCALL] PID %u: unknown syscall %u\n", current_process->pid, num);
            frame->eax = (uint32_t)-1; /* -ENOSYS */
            break;
    }
}

```

Return value convention: Written to `frame->eax`. When `isr_common_stub` executes `popa` on the way back, it restores EAX from the frame — which now contains the return value. The user code sees this in EAX after the `int 0x80` instruction.

4.10 `sys_write_impl(uint32_t fd, const char *buf, uint32_t count) → int32_t`

Parameters:

- `fd` : File descriptor. 1 = stdout (VGA), 2 = stderr (serial). Other values: return -1 (EBADF).
- `buf` : Virtual address in the user process's address space. May be any value.
- `count` : Byte count.

Security validation — mandatory before any dereference:

```

static int32_t sys_write_impl(uint32_t fd, const char *buf, uint32_t count) {

    /* Validate: buf must be in user address space (below kernel region) */

    if ((uint32_t)(uintptr_t)buf >= 0xC0000000) {
        kprintf("[SYSCALL] sys_write: buf 0x%llx in kernel space - denied\n", buf);
        return -1; /* -EFAULT */
    }

    if (count > 0 && (uint32_t)(uintptr_t)buf + count > 0xC0000000) {
        /* Buffer spans kernel boundary */
        count = (uint32_t)(0xC0000000 - (uint32_t)(uintptr_t)buf);
    }

    if (count == 0) return 0;

    if (fd != 1 && fd != 2) return -1;

    /* At this point buf is in user space. The user's page directory is loaded
       in CR3 (we are running in the user process's context), so the pointer
       is accessible at its virtual address. */

    uint32_t written = 0;

    for (uint32_t i = 0; i < count; i++) {
        char c = buf[i]; /* Reads from user virtual address */

        if (fd == 1) vga_putchar(c, VGA_COLOR_WHITE, VGA_COLOR_BLACK);

        if (fd == 2 || fd == 1) serial_putchar(c);

        if (fd == 1 && c == '\n') serial_putchar('\r');

        written++;
    }

    return (int32_t)written;
}

```

Why the range check is sufficient for this module: The user process's page directory has U/S=0 on all kernel PTEs (inherited from `boot_page_directory` but the PTEs themselves have U/S=0). If `buf` is below `0xC0000000` and points to an unmapped user page, the `buf[i]` dereference triggers a page fault — caught by the mod-3 page fault handler, which currently halts. A production kernel would use `copy_from_user()` with fixup tables; this module uses a simple range check.

4.11 sys_exit_impl(int exit_code) → void (never returns)

```
static void sys_exit_impl(int exit_code) {  
  
    kprintf("[SYSCALL] PID %u '%s' exited with code %d.\n",  
           current_process->pid, current_process->name, exit_code);  
  
    current_process->state = PROCESS_ZOMBIE;  
  
    /* Zombie processes are never scheduled again.  
     * Their PCB slot remains occupied until a parent collects the exit code  
     * (wait() – not implemented in this module).  
     * For this module: zombie slots are leaked (finite number of processes anyway). */  
  
    sched_schedule(); /* Switch to another process; never returns to this context */  
  
    /* Unreachable: if somehow reached, halt */  
    kprintf("[SYSCALL] BUG: sys_exit returned!\n");  
    for (;;) __asm__ volatile ("cli; hlt");  
}
```

4.12 INT 0x80 Gate Registration

Called from `kmain()` after `idt_setup_all()` (mod-2) and after `sched_init()`:

```
/* In kmain.c, after idt_setup_all() and sched_init() */  
  
/* Register INT 0x80 as a user-callable trap gate */  
  
extern void isr128(void); /* Defined in isr_stubs.asm (mod-2 file, needs isr128 stub added) */  
  
idt_set_gate(0x80, (uint32_t)(uintptr_t)isr128, 0x08, IDT_SYSCALL_GATE);  
  
/* IDT_SYSCALL_GATE = 0xEF: P=1, DPL=3, trap gate (IF preserved)  
 * DPL=3 allows ring-3 code to invoke via 'int 0x80' without GPF.  
 * Trap gate preserves IF – other interrupts can fire during syscall execution. */
```

The `isr128` stub must be added to `isr_stubs.asm` (mod-2 file):

```
; Add to isr_stubs.asm (or create a new file proc/syscall_stub.asm)
global isr128
isr128:
    ; Trap gate: IF is NOT cleared on entry (other interrupts can fire during syscall).
    ; No cli needed here.
    push dword 0          ; Dummy error code
    push dword 128         ; Vector 0x80
    jmp isr_common_stub
; isr_common_stub calls exception_handler() which dispatches to syscall_dispatch()
; when frame->vector == 128.
```

NASM

Integration with `exception_handler`: Add to `interrupt.c`'s `exception_handler()`:

```
/* In exception_handler(), before the "vector < 32" check */

if (frame->vector == 128) {
    syscall_dispatch(frame);
    return; /* Normal return; isr_common_stub restores all regs and iret */
}
```

C

4.13 `process_exit(int exit_code) → void (never returns)`

Thin wrapper callable from kernel process code (not via syscall):

```
void process_exit(int exit_code) {
    sys_exit_impl(exit_code);
    /* Never returns */
}
```

C

5. Algorithm Specification

5.1 Idle Process Initialization — Making `kmain` Process 0

The kernel's `kmain` execution context becomes the idle process (PID 0). This avoids a special "no current process" case in the scheduler and gives the system a process that runs when everything else is blocked or zombie.

```

/* In kmain(), after sched_init() and tss_init(), before creating other processes */

/* Claim process_table[0] as the idle process (PID 0) */

process_table[0].pid          = 0;
process_table[0].state        = PROCESS_RUNNING; /* We ARE running */
process_table[0].ticks_remaining = SCHEDULER_TICKS_PER_SLICE;
strncpy(process_table[0].name, "idle", 31);
process_table[0].page_directory = boot_page_directory;
process_table[0].kernel_stack_top =
    (uint32_t)(uintptr_t)process_table[0].kernel_stack + KERNEL_STACK_SIZE;

current_process = &process_table[0];

/* Set TSS.ESP0 for the idle process */

tss_set_kernel_stack(current_process->kernel_stack_top);

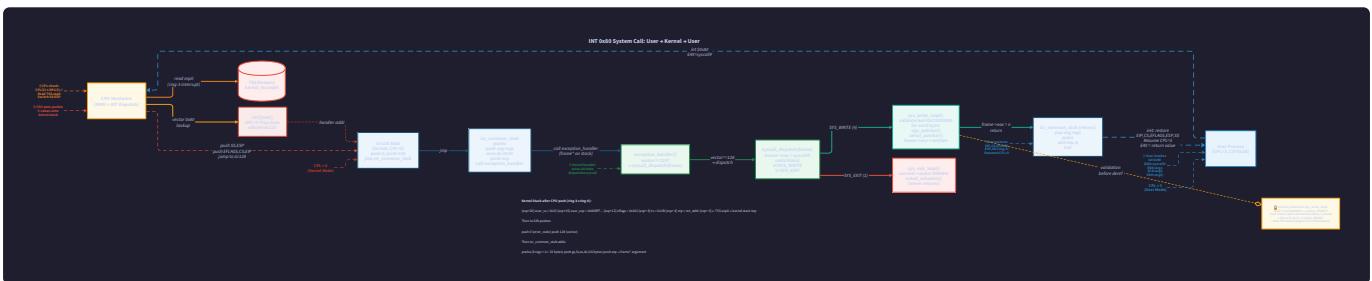
/* Note: process_table[0].saved_esp is NOT set here.

The idle process never gets "preempted from an IRQ stub" in the normal sense;
when the scheduler switches back to it, it resumes from wherever
context_switch_asm saved its ESP (inside sched_schedule()).

The first time the scheduler preempts the idle process during its hlt loop,
context_switch_asm will save idle's ESP at that point.

Before the first preemption, saved_esp == 0 (BSS-initialized) – this is fine
because the idle process is RUNNING (not suspended) until first preemption. */

```



5.2 Three-Process VGA Demo — Correct Volatile MMIO Access

Each demo process writes to a distinct VGA row using volatile pointer access. The `volatile` qualifier prevents the compiler from eliminating MMIO writes as "dead stores" with optimization enabled.

```
/* In process.c or kmain.c */

static void proc_a_entry(void) {

    uint32_t count = 0;

    while (1) {

        volatile uint16_t *vga = (volatile uint16_t *)0x000B8000;

        /* Row 2, columns 0-39 */

        const char prefix[] = "ProcA: ";

        for (int i = 0; prefix[i]; i++) {

            vga[2 * 80 + i] = (uint16_t)prefix[i] | (0x0A << 8); /* Light green */

        }

        /* Print count as decimal in-place (no kprintf - avoids locking issues) */

        char num_buf[12];

        int n = count; int pos = 0;

        if (n == 0) { num_buf[pos++] = '0'; }

        else { while (n > 0) { num_buf[pos++] = '0' + n % 10; n /= 10; } }

        /* Write digits in reverse */

        int col = 7;

        for (int j = pos - 1; j >= 0; j--) {

            vga[2 * 80 + col++] = (uint16_t)num_buf[j] | (0x0A << 8);

        }

        vga[2 * 80 + col] = (uint16_t)' ' | (0x0A << 8);

        count++;

        for (volatile int d = 0; d < 500000; d++); /* Visible delay between updates */

    }

}

static void proc_b_entry(void) {

    /* Row 12, columns 0-39, cyan text (0x0B) */

    /* Identical structure to proc_a_entry with different row and color */

}

static void proc_c_entry(void) {

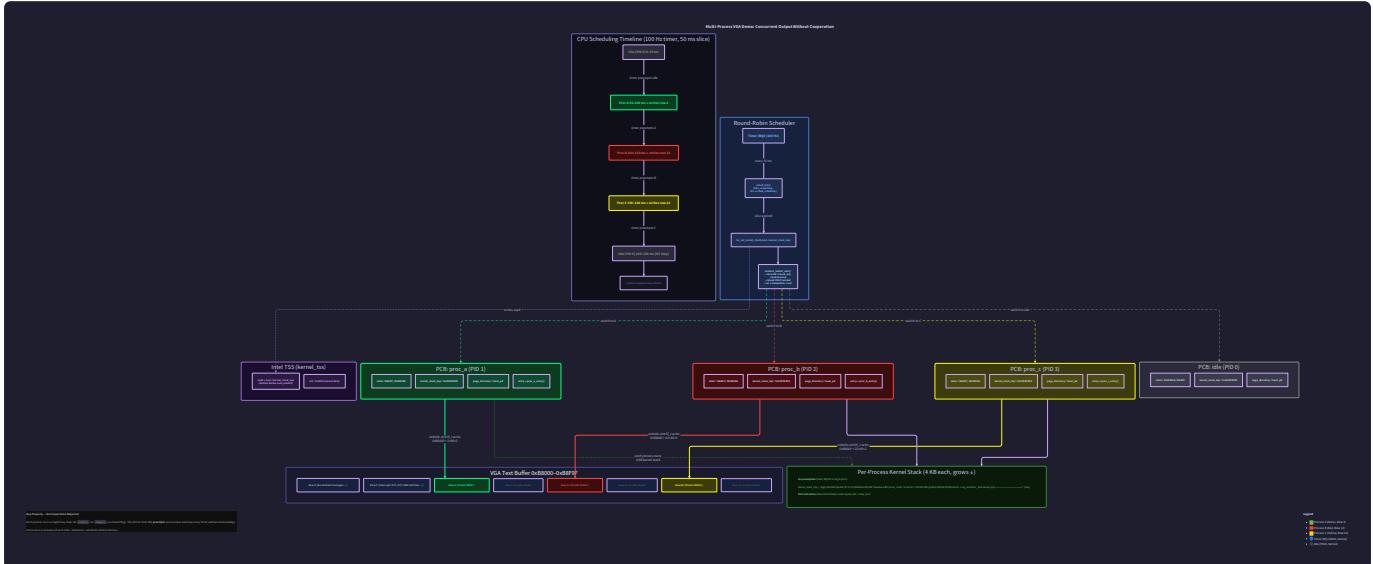
    /* Row 22, columns 0-39, yellow text (0x0E) */

}
```

}

No `kprintf` inside concurrent processes: `kprintf` writes to both VGA (shared global cursor) and serial (single UART). Concurrent writes from multiple processes with no lock produce interleaved output. The demo processes write directly to fixed VGA cell offsets — no cursor movement, no shared mutable state. This is safe for concurrent access because each process writes to a non-overlapping region of VGA memory.

Visible delay loop: `for (volatile int d = 0; d < 500000; d++)` burns approximately 2.5M cycles at 2GHz = ~1.25ms per iteration. With a 50ms time slice, the process does approximately 40 iterations per slice before being preempted. The counter increments are visible to a human observer: each process's counter advances at roughly 40 increments per 50ms = ~800 increments/second, making the three counters visibly independent.



5.3 User-Mode Process Code — INT 0x80 Inline Assembly

The user-mode process code is compiled separately and its binary is copied into a user physical frame. It uses `int 0x80` for system calls:

```
/* user/user_hello.c - compiled separately; binary copied to user frame at 0x400000 */

/* This file CANNOT use any kernel headers or libc.

It must be fully self-contained: no global variables, no function calls beyond
the inline asm wrappers, no floating point. */

/* sys_write and sys_exit inline wrappers (copy from user_syscall.h): */

static inline int u_write(int fd, const char *buf, int count) {

    int ret;

    __asm__ volatile (
        "int $0x80"
        : "=a"(ret)
        : "a"(4), "b"(fd), "c"((unsigned)buf), "d"((unsigned)count)
        : "memory"
    );

    return ret;
}

static inline void u_exit(int code) {

    __asm__ volatile (
        "int $0x80"
        :
        : "a"(1), "b"(code)
        :
    );

    /* Unreachable; process is zombie after sys_exit */
    while (1);
}

/* Entry point - must be at the BEGINNING of the compiled binary.

Ensure this function is the first thing in the .text section.

Compile with: -fno-toplevel-reorder or use a linker script. */

void user_main(void) {

    const char msg[] = "Hello from ring 3!\n";

    /* sizeof(msg) - 1 = 19 (excludes null terminator) */
}
```

```

for (int i = 0; i < 5; i++) {

    u_write(1, msg, 19);

    /* Busy wait so scheduler has time to preempt */

    for (volatile int d = 0; d < 200000; d++);

}

u_exit(0);

}

```

Compilation of user code (produces position-independent code at 0x400000):

```

user/user_hello.bin: user/user_hello.c
    i686-elf-gcc -m32 -ffreestanding -nostdlib -nostdinc \
        -fno-builtin -fno-stack-protector -fno-pie -O0 \
        -Ttext=0x00400000 \
        -o user/user_hello.elf user/user_hello.c
    objcopy -O binary user/user_hello.elf user/user_hello.bin

```

MAKEFILE

Loading user code into a physical frame in `process_create_user` :

```

/* The kernel links user_hello_bin_start/end symbols if user_hello.bin
   is embedded as a binary object in the kernel image.

   Alternatively: copy from a fixed physical address after loading from disk. */

extern uint8_t _binary_user_hello_bin_start[];

extern uint8_t _binary_user_hello_bin_end[];

size_t user_code_size = _binary_user_hello_bin_end - _binary_user_hello_bin_start;

if (user_code_size > PAGE_SIZE) user_code_size = PAGE_SIZE; /* Cap at 4KB */

memcpy(user_code_phys, _binary_user_hello_bin_start, user_code_size);

```

C

Embedding binary in kernel (Makefile rule):

```

user_hello_obj.o: user/user_hello.bin
    i686-elf-ld -r -b binary user/user_hello.bin \
        -o user_hello_obj.o \
        --oformat elf32-i386

```

MAKEFILE

{{DIAGRAM:tdd-diag-35}}

5.4 Complete `kmain` Initialization Sequence — Module 4

```
void kmain(multiboot_info_t *mbi) {  
    /* — Milestones 1-3 (existing) — */  
  
    serial_init();  
  
    vga_clear();  
  
    kprintf("== Build-OS Kernel ==\n");  
  
    idt_setup_all();  
  
    pic_remap(0x20, 0x28);  
  
    pit_init(100);  
  
    keyboard_init();  
  
    pmm_parse_memory_map(mbi);  
  
    pmm_init(mbi);  
  
    vmm_init();  
  
    vmm_enable_paging();  
  
    heap_init();  
  
    /* — Milestone 4: Processes and Scheduling — */  
  
    /* Step 1: Initialize scheduler (replaces timer IRQ handler) */  
  
    sched_init();  
  
    kprintf("[M4] Scheduler initialized.\n");  
  
    /* Step 2: Initialize TSS (extends GDT, issues ltr) */  
  
    tss_init();  
  
    kprintf("[M4] TSS initialized.\n");  
  
    /* Step 3: Register INT 0x80 syscall gate (DPL=3, trap gate) */  
  
    extern void isr128(void);  
  
    idt_set_gate(0x80, (uint32_t)(uintptr_t)isr128, 0x08, IDT_SYSCALL_GATE);  
  
    kprintf("[M4] INT 0x80 syscall gate registered.\n");  
  
    /* Step 4: Make kmain's context the idle process */  
  
    process_table[0].pid      = 0;  
  
    process_table[0].state     = PROCESS_RUNNING;
```

```

process_table[0].ticks_remaining = SCHEDULER_TICKS_PER_SLICE;

strncpy(process_table[0].name, "idle", 31);

process_table[0].page_directory = boot_page_directory;

process_table[0].kernel_stack_top =
    (uint32_t)(uintptr_t)process_table[0].kernel_stack + KERNEL_STACK_SIZE;

current_process = &process_table[0];

tss_set_kernel_stack(current_process->kernel_stack_top);

kprintf("[M4] Idle process (PID 0) registered.\n");

/* Step 5: Create kernel processes */

process_t *pa = process_create_kernel("proc_a", proc_a_entry);

process_t *pb = process_create_kernel("proc_b", proc_b_entry);

process_t *pc = process_create_kernel("proc_c", proc_c_entry);

kprintf("[M4] Kernel processes: A=PID%u B=PID%u C=PID%u\n",
       pa?pa->pid:0, pb?pb->pid:0, pc?pc->pid:0);

/* Step 6: Create user-mode process */

process_t *pu = process_create_user("user_hello", (void(*)(void))0);

/* Note: entry is ignored in process_create_user's interface above;

   user code is loaded from embedded binary at 0x400000. */

kprintf("[M4] User process: PID%u (ring 3 at 0x00400000)\n", pu?pu->pid:0);

/* Step 7: Enable interrupts – scheduler begins immediately */

__asm__ volatile ("sti");

kprintf("[M4] Interrupts enabled. Preemptive scheduling active.\n");

kprintf("[M4] Watch rows 2, 12, 22 for concurrent counter updates.\n");

/* Step 8: Idle loop */

while (1) {

    __asm__ volatile ("hlt"); /* Sleep until next interrupt */

}

}

```

Order rationale:

- `sched_init()` before `tss_init()`: scheduler must be initialized before TSS because `sched_init()` installs the timer handler; if TSS initialization triggered an interrupt (it does not, but defensively), the handler must be ready.

- `tss_init()` before `sti`: TSS must be configured before any ring-3 code runs (ring-3 → ring-0 transitions read `esp0`).
 - INT 0x80 gate after `idt_setup_all()`: `idt_setup_all()` installs a default handler at 0x80; we override it. Order is:
`idt_setup_all()` → `idt_set_gate(0x80, ...)`.
 - Idle process registration before `process_create_kernel()`: `current_process` must be non-NUL before the scheduler's timer handler fires, which happens after `sti`. We set it before `sti`.
 - `tss_set_kernel_stack()` for idle process before `sti`: the first ring-3 interrupt (if any) after `sti` must find a valid `esp0`.
 - `sti` last: nothing fires until all data structures are ready.
-

6. Error Handling Matrix

Error	Detected By	Recovery	User-Visible?
Triple fault: TSS not initialized before first ring-3 interrupt	CPU reads TR = 0 (or garbage); esp0 comes from physical address 0 (IVT); interrupt frame pushed to address 0; immediate page fault; triple fault	No recovery. Prevention: <code>tss_init()</code> and <code>tss_set_kernel_stack()</code> called before <code>sti</code> and before any ring-3 process runs. Detection: <code>-d int</code> shows triple fault immediately after first user-mode interrupt.	QEMU reboot
TSS ESP0 updated AFTER context_switch_asm (wrong order)	An interrupt fires between the stack pivot (<code>mov esp, ecx</code> in <code>context_switch_asm</code>) and the TSS update. The interrupt uses the old esp0 — the old process's kernel stack. Interrupt frame pushed there. IRQ handler runs on old stack. <code>saved_esp</code> of old process is corrupted.	Silent stack corruption; manifests as wrong register values in old process on next resume. Prevention: <code>tss_set_kernel_stack()</code> BEFORE <code>context_switch_asm()</code> . Since IF=0 during this code, no interrupt can actually fire — but the invariant must still hold.	Silent corruption; eventual crash
Context switch in C (missing assembly)	Compiler saves only callee-saved registers (EBX, ESI, EDI, EBP). EAX, ECX, EDX, EFLAGS are trashed. On resume, interrupted code has wrong values in caller-saved registers.	Silent corruption; crash when trashed register is used. Detection: test code that sets EAX to a known value before a timer tick and checks it after — EAX will be wrong.	Silent corruption
ESP pivot before argument loading in context_switch_asm	<code>[esp+8]</code> reads from next process's stack, not current call frame. <code>new_esp</code> and <code>new_cr3</code> get garbage values. Context switch corrupts both processes.	Triple fault or wild branch. Detection: any boot with multiple processes crashes immediately.	QEMU reboot or silent corruption
current_process not updated before context_switch_asm	Timer fires in new process; <code>sched_schedule()</code> reads <code>current_process</code> → still points to old process; round-robin starts from wrong index; wrong process scheduled.	Incorrect scheduling (processes run in wrong order or one process never runs). Detection: VGA counters update at uneven rates; one counter freezes.	Visible scheduling anomaly
Missing IF=1 in initial EFLAGS (used 0x200 instead of 0x202)	New process starts with IF=1 (bit 9 of <code>0x202</code> is correct). If EFLAGS=0x200 (no IF), IF is still set — bit 9 of 0x200 is 0. Wait: 0x200 = binary 001000000000 — bit 9 IS set. <code>0x00000200</code> = IF=1. <code>0x00000202</code> = IF=1 + reserved bit 1 (always 1 per Intel spec). Using 0x200 is functionally correct; 0x202 is pedantically correct. If 0 is used: new process runs with interrupts disabled; timer never preempts it; system freezes with one runaway process.	System freezes on first new process activation. Detection: VGA shows only one counter updating.	System freeze

Error	Detected By	Recovery	User-Visible?
User process CS = 0x08 (ring-0 selector) instead of 0x1B	<code>iret</code> with CS=0x08 transitions to ring 0 — no privilege crossing, no SS/ESP pop. Process runs in ring 0 with user code. A write to kernel memory succeeds — no isolation.	Security hole; no crash unless user code deliberately corrupts kernel. Detection: user code can read kernel memory without page fault.	Security violation; no visible error
User process CS = 0x1B but SS = 0x10 (kernel data)	<code>iret</code> pops the ring-3 SS/ESP pair — but SS=0x10 has DPL=0, not DPL=3. CPU checks SS.DPL matches CS.DPL on ring change — mismatch causes GPF.	GPF on first activation of user process. Detection: <code>v=0d e=0010</code> in <code>-d int</code> log (error code <code>0x0010</code> = selector <code>0x10</code> caused the fault).	GPF printed by exception handler
Shared kernel stack: process B's syscall corrupts process A	Both A and B use the same physical stack memory. B's IRQ frame overwrites A's saved state.	Silent corruption of A's registers; crash when A resumes. Prevention: each <code>process_t</code> has its own <code>kernel_stack[]</code> array embedded in the PCB (4096 bytes per PCB).	Silent corruption; eventual crash
TLB not flushed on CR3 change (skipping conditional in context_switch_asm)	Old process's address translations cached in TLB. New process accesses a virtual address that maps to a different physical frame in its page directory — but TLB serves the old mapping. New process reads/writes old process's data.	Silent security violation and data corruption. Prevention: <code>cmp eax, edx;</code> <code>je .same_cr3;</code> <code>mov cr3, edx</code> in <code>context_switch_asm</code> . If two processes share <code>boot_page_directory</code> , the comparison correctly skips the unnecessary flush.	Silent cross-process data corruption
User buffer validated by range check only; buf points to unmapped user page	<code>buf[i]</code> dereference in <code>sys_write_impl</code> triggers page fault. Current page fault handler halts the kernel.	System halt — entire kernel stops. For this module: acceptable (no demand paging). Production: use <code>copy_from_user</code> with fixup.	Kernel halt (all processes stop)
irq_return_trampoline segment register pop order wrong	Trampoline pops gs, fs, es, ds in that order, but fake frame was built with ds at lowest address. CPU loads wrong values into each segment register. GPF on next segment-register-qualified access.	GPF immediately after first trampoline execution. Detection: <code>-d int</code> shows <code>v=0d</code> after first context switch.	GPF
ltr before gdt_flush()	CPU's cached GDT still has 5 entries; entry 5 is not yet visible. <code>ltr 0x28</code> either faults (selector beyond limit) or reads stale data.	GPF (#GP) on <code>ltr</code> instruction. Detection: <code>-d int</code> shows <code>v=0d e=0028</code> (selector 0x28 caused fault).	GPF
process_create_user does not mark user_pd frame used in PMM	PMM allocates the same frame again later. Code writes to the frame, corrupting the page directory.	Address translation goes to wrong physical addresses; data corruption or page fault. Prevention: <code>vmm_map_page</code> internally does NOT mark the page directory frame used (that frame was allocated by <code>pmm_alloc_frame</code> which marks it used automatically). This error cannot occur if <code>pmm_alloc_frame</code> is used correctly.	Silent corruption

Error	Detected By	Recovery	User-Visible?
<code>proc_a/b/c use kprintf concurrently</code>	VGA cursor state corrupted by concurrent writes; serial output interleaved.	Garbled VGA text; no crash. Prevention: demo processes write to fixed VGA cell offsets, never call <code>kprintf</code> .	Garbled output
<code>sys_exit calls sched_schedule with zombie state, no ready process</code>	If all processes exited, <code>sched_schedule</code> finds no ready process and calls fatal halt.	Fatal halt — intended behavior when all work is done. The idle process (PID 0) is always READY or RUNNING, so this halt should never occur in the demo.	<code>kprintf</code> fatal message
<code>add esp, 8 missing after popa in irq_return_trampoline</code>	ESP points 8 bytes too low when <code>iret</code> executes. <code>iret</code> pops wrong values for EIP, CS, EFLAGS.	Triple fault or branch to random address.	QEMU reboot
Double-free: <code>process_create_user</code> fails mid-way, partial cleanup	First <code>pmm_alloc_frame()</code> succeeds, second fails, first frame is freed. If first frame was frame F, freeing it decrements the used count. Later <code>pmm_alloc_frame()</code> may return F again.	Potential double-use of frame F. Prevention: full cleanup path in error handling frees exactly the frames that were successfully allocated.	Silent frame reuse

7. Implementation Sequence with Checkpoints

Phase 1 — PCB and `cpu_state_t` Structs (2–3 hours)

Create `include/process.h` with `cpu_state_t`, `process_state_t`, `process_t`, `MAX_PROCESSES`, `KERNEL_STACK_SIZE`, `SCHEDULER_TICKS_PER_SLICE`. Add all static assertions from §3.1.

Create a stub `proc/process.c` that declares `process_t process_table[MAX_PROCESSES]` and `process_t *current_process = NULL` in BSS/data. Stub out `process_create_kernel` and `process_create_user` as `return NULL`.

Checkpoint 1: `i686-elf-gcc -m32 -ffreestanding -nostdinc -c proc/process.c -o process.o`. Zero errors and zero warnings. All static assertions in `process.c` compile cleanly (none trigger — verifying field offsets). Run `objdump -t process.o | grep process_table` — symbol in BSS, size = $16 * 4152 = 66432$. Verify: `size = 16 * sizeof(process_t)`. Use `sizeof(process_t)` from a test program to confirm it matches expectations.

Phase 2 — TSS, GDT Extension, `ltr` (3–4 hours)

Extend `gdt.c` (mod-1 file) to support 6 entries. Add `gdt_install_tss(int index, uint32_t base, uint32_t limit)`. Verify `gdt_flush()` is callable (not static).

Create `include/tss.h` and `proc/tss.c`. Implement `tss_init()` (Steps 1–8 from §4.1) and `tss_set_kernel_stack()`.

Add `tss_init()` call in `kmain()` (temporarily, with `sti` still not present). Add `kprintf("[TSS] TR=0x28, esp0=0x%x, ss0=0x%x\n", kernel_tss.esp0, kernel_tss.ss0)` immediately after `tss_init()`.

Checkpoint 2: Boot. Expected output includes `"[TSS] Initialized. TSS at 0x..."` and `"TR=0x28, esp0=0x00000000, ss0=0x00000010"`. Use QEMU monitor (`Ctrl+Alt+2`): `info registers` — verify `TR = 0x0028`. Also verify: `str 0x28` in the monitor output. If GPF from `ltr`: GDTR limit was too small (GDT had only 5 entries); check that `gdt_flush()` was called after `gdt_install_tss()`. Use `-d int` to see `v=0d e=0028` for this failure.

Phase 3 — `context_switch_asm` and `irq_return_trampoline` (4–6 hours)

Create `proc/context_switch.asm`. Implement `context_switch_asm` (§3.7 exact code) and `irq_return_trampoline` (§3.5 exact code). Export both with `global`.

Add the `isr128` stub to `interrupts/isr_stubs.asm` (or a new `proc/syscall_stub.asm`).

Unit test without scheduler: Manually construct a fake stack frame in `kmain()` and call `context_switch_asm` to jump to a simple function:

```
/* Test in kmain() - temporary, remove after verification */

static uint8_t test_stack[4096] __attribute__((aligned(16)));

uint32_t test_stack_top = (uint32_t)(uintptr_t)test_stack + 4096;

static void test_func(void) {

    kprintf("[CTX] test_func reached! Context switch works.\n");

    /* Infinite loop - do not return */

    while(1) __asm__ volatile ("hlt");

}

/* Build fake frame */

uint32_t *sp = (uint32_t *)(uintptr_t)test_stack_top;

extern pde_t boot_page_directory[];

extern void irq_return_trampoline(void);

*--sp = 0x00000202; /* eflags */

*--sp = 0x00000008; /* cs */

*--sp = (uint32_t)test_func; /* eip */

*--sp = 0; *--sp = 0; /* error_code, vector */

*--sp = 0; *--sp = 0; *--sp = 0; *--sp = 0; /* eax..edx */

*--sp = 0; *--sp = 0; *--sp = 0; *--sp = 0; /* esp_pusha, ebp, esi, edi */

*--sp = 0x10; *--sp = 0x10; *--sp = 0x10; *--sp = 0x10; /* ds,es,fs,gs */

*--sp = (uint32_t)(uintptr_t)irq_return_trampoline;

uint32_t test_esp = (uint32_t)(uintptr_t)sp;

uint32_t dummy_saved_esp;

context_switch_asm(&dummy_saved_esp, test_esp,

                  (uint32_t)(uintptr_t)boot_page_directory);

/* Never returns */
```

Checkpoint 3: Boot. Expected output: "[CTX] test_func reached! Context switch works.". Then system halts at the `hlt`. If QEMU reboots: wrong fake frame layout — check field order against §3.6. Use `-d int 2>log`: look for `v=0d` (GPF from bad segment) or `v=0e` (page fault from wrong EIP). If "test_func reached" appears but then immediately triple-faults: `test_func` itself has a bug (check function pointer cast).

Remove the test before Phase 4.

Phase 4 — Fake Initial Stack Frame Construction (3–5 hours)

Implement `process_create_kernel()` fully (§4.3) including the fake initial frame construction from §3.6.

Add compile-time assertions for frame layout into `process.c`:

```
/* Verify that the fake frame construction pushes exactly the right number of bytes */

/* A kernel process frame: 1 (trampoline) + 4 (segs) + 8 (pusha) + 2 (stub) + 3 (cpu-pushed) = 18 words = 72
bytes */

/* saved_esp = kernel_stack_top - 72 */

/* This is architecture-verified by the trampoline working correctly in Phase 3. */
```

Checkpoint 4: Create two kernel processes and the idle process. Enable `sti` temporarily (no `tss_set_kernel_stack` update yet — keep TSS pointing to idle's stack for now). Verify that when the first timer tick fires, `sched_tick` increments the counter (stub out `sched_schedule` to just return). Later, implement a minimal `sched_schedule` that does one unconditional switch to `process_table[1]`. Verify that `test_func` (from `process_table[1]`) runs.

Checkpoint 4 exact test:

```
/* Minimal sched_schedule stub */

void sched_schedule(void) {

    static int switched = 0;

    if (switched) return;

    switched = 1;

    process_t *next = &process_table[1]; /* Process A */

    next->state = PROCESS_RUNNING;

    tss_set_kernel_stack(next->kernel_stack_top);

    process_t *old = current_process;

    current_process = next;

    context_switch_asm(&old->saved_esp, next->saved_esp,
                      (uint32_t)(uintptr_t)next->page_directory);

    current_process = old;

}
```

Process A's entry function: `kprintf("Process A activated!\n"); while(1) __asm__ volatile ("hlt");`

Expected: After `sti`, timer fires, `sched_schedule` switches to process A, "Process A activated!" appears. System then halts (process A's `hlt` loop) — subsequent timer ticks have `switched=1` so no further switches.

Phase 5 — `process_create_kernel()` Finalized (2–3 hours)

No new implementation needed — Phase 4 completed this. This phase focuses on testing all three kernel processes simultaneously with the partial scheduler.

Add `proc_a_entry`, `proc_b_entry`, `proc_c_entry` functions. Verify that each writes to a distinct VGA row. Test with a scheduler that cycles: idle → A → B → C → idle → ...

Checkpoint 5: Three VGA rows (2, 12, 22) show independent counters, each in a different color. The idle row shows nothing (or a static message). Counters update at approximately equal rates (equal time slices). The system runs stably for ≥ 30 seconds without crash.

Phase 6 — Full Round-Robin Scheduler (3–4 hours)

Implement `sched_init()`, `sched_tick()` (`sched_timer_irq`), and the complete `sched_schedule()` from §4.7 with the two-assignment `current_process` pattern.

Replace the stub scheduler from Phase 4 with the real implementation. Remove all test stubs.

Checkpoint 6: The complete scheduler runs. Verify:

- Counter at row 2 (Process A) updates approximately every 50ms (one time slice).
- Counter at row 12 (Process B) and row 22 (Process C) update at the same rate.
- No VGA corruption (each process writes only to its own row).
- System runs stably for ≥ 5 minutes without crash, reboot, or display corruption.
- `kprintf` output from initialization still visible at the top of the screen (rows 0–1).

Run: `qemu-system-i386 -drive format=raw,file=os.img -serial stdio -d int 2>int.log`. Verify in `int.log`: `v=20` (IRQ0) appears at ~10ms intervals; no `v=00 – v=1F` (exceptions) appear after initialization.

Phase 7 — Three-Process VGA Demo Validation (2–3 hours)

This phase polishes the demo: ensure all VGA writes use `volatile`, clear unused columns after each number write, add visual separators, and ensure the output is human-readable.

Checkpoint 7: Record 60 seconds of serial output. Verify:

- "[M4] Preemptive scheduling active." visible.
- No exception messages in serial output.
- Timer count (can be printed occasionally from idle process) advances at ~100/sec.

Phase 8 — `process_create_user()` and Ring-3 Isolation (4–6 hours)

Implement `process_create_user()` fully (§4.4). Embed `user_hello.bin` in the kernel image (Makefile rule in §5.3). Compile `user/user_hello.c` as a separate binary.

Checkpoint 8a — Ring-3 activation: Add `process_create_user()` call in `kmain()`. Verify that the user process runs (serial shows "Hello from ring 3!" five times). Verify `sys_exit` works (process becomes zombie after 5 iterations, no crash).

Checkpoint 8b — Ring-3 isolation: Temporarily add this to `user_main()` before `u_exit()`:

```

/* Attempt to read kernel memory – should trigger page fault */

volatile uint32_t *kernel_ptr = (volatile uint32_t *)0xC0000000;

uint32_t val = *kernel_ptr; /* Must trigger page fault */

(void)val;

```

Expected: Page fault handler fires with "U=1" (user mode) and "CR2: 0xc0000000". Kernel halts. QEMU does NOT proceed with the read; the user process cannot access kernel memory.

Remove the isolation test code from `user_hello.c` after verification.

Checkpoint 8c — CS selector verification: Add to `kmain()` after `process_create_user()`:

```

/* Read CS value from user process's fake initial frame */

uint32_t *fake_frame = (uint32_t *)(uintptr_t)pu->saved_esp;

uint32_t frame_cs = fake_frame[60 / 4]; /* cs field at offset 60 from top */

kprintf("[M4] User process initial CS = 0x%04x (expect 0x001B)\n", frame_cs);

```

Expected: "CS = 0x001b". If 0x0008 : ring-3 selector not set correctly in fake frame.

Phase 9 — INT 0x80 Syscall Gate and `sys_write` / `sys_exit` (3–4 hours)

Create `include/syscall.h`, `proc/syscall.c`. Add `isr128` stub to `interrupts/isr_stubs.asm`. Register gate in `kmain()`. Add dispatch in `exception_handler()`.

Implement `syscall_dispatch()`, `sys_write_impl()`, `sys_exit_impl()`.

Checkpoint 9a — Syscall from kernel: Test `int 0x80` from kernel-mode code (CPL=0):

```

/* Temporary test in kmain() */

__asm__ volatile (
    "mov $4, %%eax\n"
    "mov $1, %%ebx\n"
    "mov %0, %%ecx\n"
    "mov $13, %%edx\n"
    "int $0x80"
    :
    : "r"((uint32_t)(uintptr_t)"syscall test\n")
    : "eax", "ebx", "ecx", "edx"
);

```

Expected: "syscall test" appears on VGA and serial. No GPF.

Checkpoint 9b — `sys_write` from user process: `user_hello.c` already calls `u_write(1, msg, 19)`. After INT 0x80 from ring 3: "Hello from ring 3!" appears 5 times. After `u_exit(0)`: "[SYSCALL] PID X 'user_hello' exited with code

0." appears.

Checkpoint 9c — sys_write kernel pointer rejection: Test with buf = 0xC0001234 :

```
/* Temporarily in user_hello.c or as a direct syscall test */

u_write(1, (const char *)0xC0001234, 10);

/* Expected return value: -1 (kernel prints denial message) */
```

Expected: "sys_write: buf 0xc0001234 in kernel space – denied" on serial. Return value -1. No kernel memory access.

Phase 10 — Integration and Final Acceptance (3–5 hours)

Wire all components together in final kmain() (§5.4). Remove all temporary test code. Verify the complete system.

Checkpoint 10 (Final Acceptance Test):

```
qemu-system-i386 -drive format=raw,file=os.img -serial stdio 2>/dev/null
```

BASH

All of the following must be verified simultaneously:

1. VGA rows 2, 12, 22 show three independent counters in different colors, incrementing at approximately equal rates.
2. Serial output shows "Hello from ring 3!" five times from the user process (within the first 2 seconds).
3. Serial output shows "PID X 'user_hello' exited with code 0."
4. After user process exit, the three kernel counters continue running uninterrupted.
5. System runs stably for 5 minutes with no crash.

```
qemu-system-i386 -drive format=raw,file=os.img -serial stdio -d int 2>int.log
```

BASH

Expected: Only v=20 (IRQ0 timer) appears in the first 20 interrupt-related entries after initialization. No v=00 – v=1F (CPU exceptions). No v=08 (double fault).

```
qemu-system-i386 -s -S -drive format=raw,file=os.img -serial stdio &

gdb kernel.elf -ex "target remote :1234" -ex "break sched_schedule" -ex "continue"

# At breakpoint: inspect current_process, next process's saved_esp, TSS esp0

(gdb) print *current_process

(gdb) print kernel_tss.esp0

(gdb) print next->saved_esp

(gdb) x/20x next->saved_esp
```

BASH

Verify GDB output: kernel_tss.esp0 matches next->kernel_stack_top (set by tss_set_kernel_stack just before the breakpoint). The 20 words at next->saved_esp show: trampoline address at lowest, then gs=0x10, fs=0x10, es=0x10, ds=0x10, zero-filled GPRs, vector=0, error_code=0, eip=entry_func, cs=0x08 or 0x1B, eflags=0x202.

8. Test Specification

8.1 Process Control Block Tests

T-PCB-1: `cpu_state_t` field offsets (static assertions)

- All offsets from §3.1 verified at compile time by `_Static_assert`. Build failure = wrong layout.

T-PCB-2: `process_t` size and PCB table size

- `_Static_assert(sizeof(process_t) >= 4096 + 56, "PCB must include full kernel stack")`.
- `_Static_assert(sizeof(process_table) == MAX_PROCESSES * sizeof(process_t), ...)`.

T-PCB-3: `kernel_stack_top` alignment

- After `process_create_kernel()`: `proc->kernel_stack_top % 16 == 0` (16-byte aligned per ABI).
- Verified by: `kprintf("stack_top %% 16 = %u\n", proc->kernel_stack_top % 16)`. Must print `0`.

T-PCB-4: `kernel_stack_top = kernel_stack + KERNEL_STACK_SIZE`

- After creation: `proc->kernel_stack_top == (uint32_t)(uintptr_t)proc->kernel_stack + 4096`.

T-PCB-5: Distinct kernel stacks for each process

- Create 4 processes. Verify no two `kernel_stack[]` arrays overlap: for all pairs (i, j): `process_table[i].kernel_stack + 4096 <= process_table[j].kernel_stack` or vice versa. Since they are embedded in an array with no aliasing, this holds automatically — but verify by comparing addresses.

T-PCB-6: `saved_esp` within kernel stack bounds

- After `process_create_kernel()`: `proc->saved_esp >= (uint32_t)(uintptr_t)proc->kernel_stack` and `proc->saved_esp < proc->kernel_stack_top`. Print both values to verify.

8.2 TSS Tests

T-TSS-1: TSS size exactly 104 bytes

- `_Static_assert(sizeof(tss_t) == 104, ...)` — compile-time verified.

T-TSS-2: TSS `ss0 = 0x10` after `tss_init()`

- `kprintf("ss0=0x%x\n", kernel_tss.ss0) → "ss0=0x10"`.

T-TSS-3: TSS `iomap_base = sizeof(tss_t) = 104`

- `kprintf("iomap=0x%x\n", kernel_tss.iomap_base) → "iomap=0x68"` (104 = 0x68).

T-TSS-4: TR register contains `0x28` after `tss_init()`

- Read TR: `uint16_t tr; __asm__ volatile ("str %0" : "=r"(tr)); kprintf("TR=0x%04x\n", tr) → "TR=0x0028"`.

T-TSS-5: GDT entry 5 has access byte `0x89`

- Read GDT entry 5 byte 5: `uint8_t access = ((uint8_t*)gdt)[5*8+5]; kprintf("GDT[5] access=0x%02x\n", access) → "GDT[5] access=0x89"`.

T-TSS-6: `tss_set_kernel_stack()` updates `esp0` immediately

- `tss_set_kernel_stack(0xDEAD1000); kprintf("esp0=0x%x\n", kernel_tss.esp0) → "esp0=0xdead1000"`. Then restore: `tss_set_kernel_stack(current_process->kernel_stack_top)`.

T-TSS-7: `esp0` contains correct value before ring-3 process runs

- After `process_create_user()` and before `sti`: read `kernel_tss.esp0`. Must equal `current_process->kernel_stack_top` (idle process's stack top, since TSS was set for idle). After first context switch to user process: `kernel_tss.esp0` must equal `pu->kernel_stack_top`.
-

8.3 Context Switch Tests

T-CTX-1: `context_switch_asm` saves old ESP

- Create process A. Before switch: record `process_table[0].saved_esp` (should be 0, BSS-init). After switch to A and A running for one slice: during next switch back to idle, `process_table[0].saved_esp` must be a valid address in `process_table[0].kernel_stack`.

T-CTX-2: `context_switch_asm` loads new ESP

- Before switch to process A: record `pa->saved_esp`. Verify (via GDB breakpoint in `context_switch_asm`) that `mov esp, ecx` loads exactly this value.

T-CTX-3: CR3 reloaded on address space change

- Create a user process with a different page directory. Set breakpoint at `.same_cr3` label in `context_switch_asm`. When switching idle (boot_pd) → user process (user_pd): verify `jne` is taken (CR3 is different). When switching kernel A → kernel B (same boot_pd): verify `je` is taken.

T-CTX-4: All registers preserved across context switch

- In `proc_a_entry()`: set EBX, ESI, EDI to known values via inline asm: `__asm__ volatile ("mov $0xAAAA0001, %%ebx; mov $0xB BBBB0001, %%esi" :: "ebx", "esi")`. Busy-loop for 2 seconds (several context switches). Read EBX and ESI: must still be `0xAAAA0001` and `0xB BBBB0001`. Failure = register clobbering.

T-CTX-5: EFLAGS IF bit preserved after context switch

- After several context switches: `uint32_t flags; __asm__ volatile ("pushf; pop %0" : "=r"(flags)); kprintf("IF=%d\n", (flags >> 9) & 1)` from a process function. Must print `"IF=1"`.

T-CTX-6: `irq_return_trampoline` segment register restoration

- After first activation of a new process: the process function reads DS, ES, FS, GS and verifies they are all `0x10` (kernel data). `uint16_t ds; __asm__ volatile ("mov %%ds, %0" : "=r"(ds)); kprintf("DS=0x%04x\n", ds) → "DS=0x0010"`.
-

8.4 Scheduler Tests

T-SCHED-1: Round-robin cycles through all READY processes

- Create 3 processes. Log which process runs in each timer tick for 20 ticks. Expected pattern: approximately `A, B, C, A, B, C, ...` (with 5-tick slices: `A×5, B×5, C×5, A×5, ...`).

T-SCHED-2: ZOMBIE processes skipped

- Call `sys_exit_impl(0)` from process B. Verify subsequent scheduling cycles never invoke B's entry function.

T-SCHED-3: No-ready-process: keep current running

- Mark all processes except current as ZOMBIE or BLOCKED. `sched_schedule()` must return without switching (current keeps running, slice reset).

T-SCHED-4: Time slice respected

- Record `pit_tick_count` when a process starts running. Record when it is preempted. Difference must equal `SCHEDULER_TICKS_PER_SLICE` (= 5, ± 1 for interrupt timing).

T-SCHED-5: `current_process` correct after switch

- In process A's function: print `current_process->pid`. Must print A's PID. In process B: must print B's PID. Never prints the other process's PID.

T-SCHED-6: TSS `esp0` updated before each context switch

- GDB breakpoint in `tss_set_kernel_stack()`. At each breakpoint: verify the argument equals `next->kernel_stack_top` where `next` is the process about to be scheduled.

8.5 User-Mode Process Tests

T-USER-1: User process executes at CPL=3

- In `user_main()`, read CS: `uint16_t cs; __asm__ volatile ("mov %%cs, %0" : "=r"(cs)); u_write(1, "CS=?\n", 5)` (manually encode the CS value as a digit). CS must have bits [1:0] = 11 (RPL=3). CS = `0x1B` = 00011011 binary: bits [1:0] = 11. ✓

T-USER-2: User stack is accessible

- In `user_main()`, allocate a local variable (`int x = 42`), read it back, verify via `u_write`. The stack read succeeds — proves user stack is mapped.

T-USER-3: Kernel memory access triggers page fault

- Access `0xC0000000` from user code. Page fault fires with U=1. System halts (as per mod-3 page fault handler). `-d int` shows `v=0e e=0005` (P=1, U=1, W=0 → error code = `0b101 = 5`). Wait: `0xC0000000` kernel PDE has U/S=0; error code P=1 (page present, protection violation), W=0 (read), U=1 (user) = `0b101 = 5`. Correct.

T-USER-4: User code page is read-only

- From `user_main()`, attempt to write to `0x00400000` (its own code page): `*(volatile uint32_t *)0x00400000 = 0xDEAD`. Page fault fires with W=1 (write violation). Error code: P=1, W=1, U=1 = `0b111 = 7`.

T-USER-5: User page directory kernel PDEs are supervisor-only

- Inspect `user_pd[768]` (kernel higher-half PDE) after `process_create_user()`: `user_pd[768] & PAGE_USER == 0` (U/S flag clear). Verify: `kprintf("user_pd[768] U/S = %d\n", (user_pd[768] >> 2) & 1) → "0"`.

T-USER-6: Multiple user processes can run concurrently

- Create 2 user processes. Both run independently, both print `"Hello from ring 3!"`, both call `sys_exit`. No interference.

8.6 System Call Tests

T-SYSCALL-1: `int 0x80` from ring-0 does not GPF

- Execute `int 0x80` with EAX=4 (sys_write), EBX=1, ECX=string address, EDX=count from kernel code. Verify no GPF (DPL=3 trap gate allows lower-privilege invocation — ring-0 can always invoke DPL-3 gates since CPL ≤ DPL is checked only for DPL < CPL scenarios). Actually: for software interrupts via `int`, the check is CPL ≤ gate.DPL. CPL=0 ≤ DPL=3 — this holds. `int 0x80` from ring-0 is allowed and does not cause a privilege transition (no SS/ESP push since CS.DPL == IDT gate.DPL is not what the CPU checks — it checks whether CPL ≤ gate.DPL).

T-SYSCALL-2: `sys_write(1, buf, n)` writes to VGA and serial

- Call via `int 0x80` from user process with `fd=1`, valid buf, count=5. VGA and serial show the 5 bytes. Return value in EAX = 5.

T-SYSCALL-3: `sys_write(2, buf, n)` writes to serial only

- `fd=2` : serial shows bytes; VGA cursor does not advance. Return value = count.

T-SYSCALL-4: `sys_write` with invalid fd returns -1

- `fd=99` : return value in EAX = `0xFFFFFFFF` (-1 in two's complement). No write occurs.

T-SYSCALL-5: `sys_write` with kernel buffer pointer denied

- `buf = 0xC0001234` : return value = -1. "denied" message on serial. No kernel memory read via user-provided pointer.

T-SYSCALL-6: `sys_write` buffer spanning kernel boundary clamped

- `buf = 0xBFFFFFFC, count = 10` : only `0xBFFFFFFC` to `0xBFFFFFFF` (4 bytes) are written (clamped to `0xC0000000 - 0xBFFFFFFC = 4`). Return value = 4.

T-SYSCALL-7: `sys_exit` transitions to ZOMBIE

- After `sys_exit(42)` : `current_process->state == PROCESS_ZOMBIE` (check from a GDB breakpoint or by observing the "exited with code 42" message). The process is never scheduled again.

T-SYSCALL-8: Unknown syscall number returns -ENOSYS

- `EAX = 999` (invalid): return value = `0xFFFFFFFF`. "unknown syscall 999" on serial.

T-SYSCALL-9: Return value accessible in EAX after `int 0x80`

- `u_write(1, msg, 5)` stores the return value in EAX per inline asm `"=a"(ret)`. Verify `ret == 5` by printing it.

8.7 Integration Tests

T-INT-1: Three kernel processes + one user process run concurrently

- System runs with all 4 processes for 60 seconds. No crash, reboot, or freeze. VGA rows 2, 12, 22 show incrementing counters. Serial shows "Hello from ring 3!" 5 times. System continues after user process exits.

T-INT-2: Interrupt frequency preserved under scheduling load

- `pit_tick_count` after 10 seconds must be within $\pm 1\%$ of 1000 (100Hz \times 10s). Context switching does not add extra timer ticks or lose ticks.

T-INT-3: No exception vectors 0-31 fire during steady-state operation

- Run for 60 seconds with `-d int`. Filter `int.log` for `v=0` through `v=1f`. Only `v=20` (IRQ0) and `v=21` (IRQ1, keyboard) should appear. Zero CPU exceptions.

T-INT-4: Serial output after 60 seconds of operation

- `qemu-system-i386 ... -serial stdio | tee output.txt`. After 60s: `grep "Exception" output.txt` — zero matches. `grep "BUG" output.txt` — zero matches. `grep "FATAL" output.txt` — zero matches.

9. Performance Targets

Operation	Target	How to Measure
Full context switch latency (IRQ assert → resumed process first instruction)	700–1000 cycles (~350–500ns at 2GHz)	<code>rdtsc</code> at end of resumed process's entry; subtract <code>rdtsc</code> captured just before <code>iret</code> in trampoline
<code>context_switch_asm()</code> body execution (EAX load through <code>ret</code>)	< 15 cycles	<code>rdtsc</code> before and after the 7 instructions in the function body
<code>sched_tick()</code> common path (no switch, slice not expired)	< 20 cycles	<code>rdtsc</code> in <code>sched_timer_irq</code> around the fast path (decrement + compare + return)
<code>sched_schedule()</code> full path including switch	< 200 cycles (excludes memory latency for cold stacks)	<code>rdtsc</code> at entry and immediately after <code>context_switch_asm</code> returns (in old process's resumed execution)
<code>tss_set_kernel_stack()</code> single write	< 5 cycles (L1 cache hit — TSS is hot)	<code>rdtsc</code> around <code>kernel_tss.esp0 = esp0</code>
CR3 reload (new address space)	< 40 cycles + cold TLB miss cost	<code>rdtsc</code> around <code>mov cr3, edx</code> in <code>context_switch_asm</code> ; note first TLB miss adds ~30–50 cycles
Per-process kernel stack cold read (<code>popa</code> + <code>iret</code> after first activation)	200–400 ns	Implicit in context switch latency; dominated by DRAM access for cold stack
INT 0x80 round-trip (user <code>int 0x80</code> → kernel handler → <code>iret</code> back)	~500 cycles	<code>rdtsc</code> in user code before <code>int 0x80</code> ; <code>rdtsc</code> after; account for 5 CPU-pushed values × ~5 cycles each
<code>sys_write</code> for 20 bytes to VGA	< 5µs (20 × VGA MMIO write ~200ns each)	<code>rdtsc</code> around <code>sys_write_impl</code> body
<code>sys_exit</code> (state change + <code>sched_schedule()</code>)	< 500 cycles	<code>rdtsc</code> at entry of <code>sys_exit_impl</code> ; not measured after because process never resumes
<code>process_create_kernel()</code> including fake frame construction	< 1000 cycles	<code>rdtsc</code> around complete function; dominated by stack push loop
<code>process_create_user()</code> including page directory setup	< 10µs (3× <code>pmm_alloc_frame</code> + 2× <code>vmm_map_page</code> + <code>memcpy</code> 4KB)	<code>rdtsc</code> around function; DRAM accesses dominate
Round-robin search across 16 processes	< 100 cycles (16 state comparisons in cache)	<code>rdtsc</code> around the for-loop in <code>sched_schedule()</code>

Measurement infrastructure:

```

/* Add to kmain temporarily to benchmark context switch */

static uint64_t switch_t0 = 0;

/* In sched_schedule(), just before context_switch_asm(): */

switch_t0 = rdtsc();

/* In the first line of proc_a_entry(): */

uint64_t t1 = rdtsc();

kprintf("[PERF] Context switch: %u cycles\n", (uint32_t)(t1 - switch_t0));

```

10. Hardware Soul Analysis

The Stack Pivot — One Instruction That Changes Everything

`mov esp, ecx` in `context_switch_asm` is the single instruction that transitions between processes. Before it: every stack operation goes to the current process's kernel stack. After it: every stack operation goes to the next process's kernel stack. The instruction is not atomic in the OS sense — it is a single memory write (ESP is a register; the write is to the register file, not RAM). But the x86 pipeline sees it as a register write that takes effect immediately for all subsequent `[esp+N]` addressing.

TLB implications of the stack pivot: The new ESP points into a different physical page (the next process's `kernel_stack[]`). If that page's translation is not in the TLB, the first stack access after the pivot (the `ret` instruction, which pops `[esp+0]`) generates a TLB miss → page table walk (~30–50 cycles). Since kernel stacks are in the identity-mapped first 4MB region (physical `0x100000 – 0x1FFFFF`), their PTEs are in `identity_page_table`, which is hot in L2 cache (it is accessed frequently for all kernel code accesses). TLB miss latency for kernel stack pages: ~10–15 cycles (L2 hit for page table walk), not 50–100 cycles (DRAM).

TSS.ESP0 Read Timing — Hardware-Imposed Ordering

When the CPU processes an interrupt while in ring 3, it reads `esp0` from the TSS before pushing any values onto the kernel stack. The sequence:

1. CPU finishes current instruction (ring 3 code).
2. CPU detects `INTR` high.
3. CPU checks DPL, privilege transition needed (`CPL=3 → IDT gate DPL=0`).
4. CPU **reads `esp0` from TSS** (physical address of `kernel_tss.esp0`).
5. CPU switches to kernel stack: `SS:ESP = SS0:ESP0` from TSS.
6. CPU pushes `SS_user, ESP_user, EFLAGS, CS_user, EIP_user` onto kernel stack.
7. CPU jumps to IDT handler.

Step 4 reads `kernel_tss.esp0` from main memory (or cache). This means `tss_set_kernel_stack()` must complete its write to `kernel_tss.esp0` **before** any ring-3 code runs on the new process. The write is a simple store; on x86 with Total Store Order, stores are visible to all subsequent loads in program order. Since `tss_set_kernel_stack()` is called with `IF=0` (we are in the scheduler, which runs with `IF=0` from the timer IRQ), no interrupt can fire between the write and the context switch. The ordering is guaranteed.

Cache Behavior of the Process Table

`process_table[16]` is 66,432 bytes \approx 64.875 KB. A typical L1 data cache is 32KB; L2 is 256KB–512KB. The entire process table fits in L2. After the first few scheduling cycles, the L2 is warm:

- The scheduler reads `process_table[i].state` for each slot in the round-robin loop — 16 reads of 1 byte each, from 16 different cache lines (each PCB is 4152 bytes, spanning ~65 cache lines; but the `state` field is in the first cache line at offset 36). After warming: 16 L2 hits at \sim 5ns each = 80ns for the full round-robin search. Cold (DRAM): $16 \times 60\text{ns} = 960\text{ns}$.
- The kernel stack for each process is 4KB. On a context switch, the `popa` instruction reads 32 bytes from the new stack. If the stack is cold (last used 50ms ago — 100M cycles at 2GHz), the 32 bytes span at most 2 cache lines, triggering 2 DRAM accesses at 60ns each = 120ns. Plus `iret` reads 12–20 bytes (1 cache line), potentially from an adjacent cold cache line: 60ns more. Total cold-stack activation cost: \sim 180–300ns \approx 360–600 cycles at 2GHz.

This is the "cold-start penalty" for context switching and explains why the 700–1000 cycle target includes a significant memory latency component.

INT 0x80 Software Interrupt Path — Why It Costs 500 Cycles

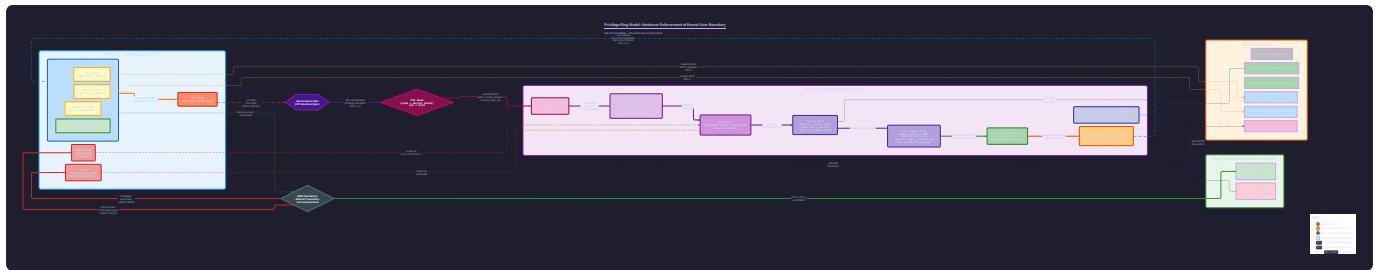
`int 0x80` from ring-3 involves:

1. `int 0x80` instruction fetch: 1 cycle (in instruction cache).
2. DPL check ($\text{CPL}=3 \leq \text{gate.DPL}=3$): \sim 2 cycles.
3. Read TSS.esp0 (TSS is hot — accessed recently for timer IRQs): 1 cache line read, \sim 5 cycles L1 hit.
4. Switch to kernel stack (SS:ESP from TSS): register writes, \sim 1 cycle.
5. Push SS_user, ESP_user, EFLAGS, CS_user, EIP_user: 5 stores to kernel stack (cold first time). If kernel stack is cold: 1 DRAM write burst \sim 30ns = \sim 60 cycles.
6. IDT lookup for vector 0x80 (IDT fits in L1/L2): \sim 5 cycles.
7. Jump to `isr128` stub: pipeline flush \sim 20 cycles.
8. `push 0; push 128; jmp isr_common_stub`: \sim 3 cycles.
9. `pusha`: \sim 8 cycles.
10. Segment register saves: \sim 8 cycles.
11. Segment register loads (kernel DS etc.): \sim 4 cycles.
12. `push esp; call exception_handler`: \sim 5 cycles.
13. `syscall_dispatch() + sys_write_impl()` execution: variable (dominated by VGA MMIO writes for `sys_write`).
14. Return path (pop segment, popa, add esp 8, iret): \sim 25 cycles.
15. `iret` to ring-3: \sim 20 cycles (privilege transition, restore SS+ESP).

Sum: 170 cycles for the interrupt mechanism + \sim 30ns cold stack = 230 cycles minimum. The 500-cycle target includes the `sys_write_impl` loop for typical 20-byte strings (3 VGA MMIO writes to kprintf's buffer, then 20 VGA direct writes).

Branch Predictability in Round-Robin Search

The round-robin loop `for (int i = 1; i <= MAX_PROCESSES; i++)` with `state == PROCESS_READY` check has a highly predictable branch pattern: in the steady state (4 active processes, 12 unused), the branch is not-taken for 12 consecutive iterations then taken — a 12-cycle pattern. The branch predictor learns this pattern and predicts correctly after the first few scheduling cycles. Misprediction rate approaches 0%, cost approaches 0 misprediction cycles after warm-up.



11. State Machine

{{{DIAGRAM:tdd-diag-37}}}

Process lifecycle states:

State	Description	Entry Conditions	Legal Exits
PROCESS_UNUSED	Slot free in process table	BSS init; after <code>process_exit</code> cleanup (not implemented)	→ PROCESS_READY (<code>process_create_*</code>)
PROCESS_READY	Initialized; not on CPU; eligible for scheduling	<code>process_create_*</code> ; preemption from RUNNING	→ PROCESS_RUNNING (scheduler selects)
PROCESS_RUNNING	On CPU; <code>current_process == this</code>	Scheduler selects from READY	→ PROCESS_READY (preemption by timer); → PROCESS_ZOMBIE (<code>sys_exit</code>)
PROCESS_BLOCKED	Waiting for event; not scheduled	Would be set by blocking I/O (not implemented)	→ PROCESS_READY (event occurs; not implemented)
PROCESS_ZOMBIE	Execution complete; awaiting cleanup	<code>sys_exit_impl()</code> sets this	(terminal in this module; no <code>wait()</code>)

Illegal state transitions:

Illegal Transition	Consequence
UNUSED → RUNNING (skipping READY)	Scheduler would attempt to run a process with <code>saved_esp == 0</code> (BSS-init); context_switch_asm loads ESP=0; next memory access faults at address 0
ZOMBIE → READY	<code>sys_exit</code> already switched away; returning to a zombie's saved_esp is undefined (stack may be reused)
RUNNING → RUNNING (second process set to RUNNING without switch)	<code>current_process</code> has two claimants; TSS.esp0 undefined for the second one; next ring-3 interrupt uses wrong kernel stack
BLOCKED → RUNNING (skipping READY; direct activation not implemented)	No <code>sched_schedule()</code> pathway leads here in this module; code bug would set invalid state
Any state → UNUSED while <code>saved_esp</code> and <code>page_directory</code> still allocated	PMM double-free if PCB slot reused without freeing resources first

Scheduler invariant (must hold at all times after `sti`):

Exactly one process has `state == PROCESS_RUNNING` and equals `current_process`. All other non-zombie, non-blocked processes have `state == PROCESS_READY`. The scheduler enforces this by transitioning `old` to `READY` and `next` to `RUNNING` atomically within the IF=0 critical section.

12. Concurrency Specification

Model: Single-core. Preemptive interrupts only (no threads, no SMP). The timer IRQ (IF=0, interrupt gate) is the only concurrency mechanism.

Critical section analysis:

`sched_schedule()` — called with IF=0:

- The timer IRQ handler runs with IF=0 (interrupt gate). Therefore `sched_timer_irq()` and everything it calls (including `sched_schedule()` and `context_switch_asm()`) runs with IF=0. No reentrant interrupt can fire.
- `sched_yield()` explicitly calls `cli` before `sched_schedule()`. The `sti` after is optional since `iret` restores EFLAGS anyway.
- **Invariant:** `current_process`, `process_table[i].state`, `kernel_tss.esp0` — all modified only while IF=0. Consistent view guaranteed.

`process_create_kernel/user()` — called with IF=1 (from `kmain` before `sti`):

- All process creation happens before `sti` in `kmain`. After `sti`, no new process is created in this module. If `process_create_*` were called after `sti`, a timer IRQ could fire during process table modification. The critical window: between finding an UNUSED slot and setting `state = PROCESS_READY`. If IRQ fires mid-creation with the slot partially initialized, `sched_schedule()` might try to schedule it. Prevention: set `state = PROCESS_READY` as the **last** write, after all other fields are initialized.

`current_process` visibility:

- Written by `sched_schedule()` (IF=0).
- Read by `sched_timer_irq()`, `sys_exit_impl()`, `sys_write_impl()`, all process functions that call `kprintf("[PID %u]", current_process->pid)`.
- On a single-core with IF=0 during scheduler, reads by kernel handlers are always from the correct value (no race). Reads by process functions (IF=1) see the value set by the last context switch — always correct since single-core.
- **No locking required** for `current_process` reads in this module.

`pit_tick_count` — retained from mod-2:

- Written by `sched_timer_irq()` (IF=0).
- Read by `kmain` idle loop (IF=1).
- Same analysis as mod-2: `volatile` prevents compiler caching; single-core guarantees no torn read of 64-bit value matters (reads are rare and for informational display only).

`kernel_tss.esp0` — write ordering:

- Written by `tss_set_kernel_stack()` (IF=0, within `sched_schedule()`).
- Read by CPU hardware on ring-3 → ring-0 transition.
- x86 TSO (Total Store Order) guarantees: the store to `esp0` completes (becomes globally visible) before the CPU processes any subsequent interrupt. Since `tss_set_kernel_stack()` runs with IF=0, no ring-3 interrupt can occur before the store completes. The next ring-3 interrupt (after `iret` restores EFLAGS with IF=1) will always see the updated `esp0`.

VGA concurrent access from multiple processes:

- Three demo processes write to non-overlapping VGA rows (row 2, 12, 22). No shared state.
- They share the physical `0xB8000` MMIO region, but writes to non-overlapping addresses produce correct results even without synchronization (x86 MMIO writes are ordered per address).
- The `vga_putchar()` function maintains a cursor (shared mutable state). Demo processes do **not** use `vga_putchar()`; they write directly to `VGA_BASE[row * 80 + col]` via computed indices. This avoids the shared cursor.

User process isolation — hardware-enforced:

- Each user process has its own page directory (loaded as CR3 on switch). Kernel pages in user page directories have U/S=0. A user process cannot read or write kernel memory: the MMU enforces this at the hardware level, independent of software locking.

13. Synced Criteria

Beyond the Atlas: Further Reading

x86 Boot Sequence & Real Mode

Paper: None — foundational hardware spec.

Spec: Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A — Chapter 9 "Processor Management and Initialization" (exact reset vector, A20 history, protected mode prerequisites). Intel, 2024.

Code: SeaBIOS — `src/post.c`, function `handle_post()` — the actual open-source BIOS POST sequence your bootloader hands control to: github.com/coreboot/seabios

Best Explanation: OSDev Wiki, "Bare Bones" article — wiki.osdev.org/Bare_Bones — the single most-cited practical bootstrap reference; covers NASM, linker scripts, and GRUB multiboot in one page.

Why: The only resource that gives you working assembly, a working linker script, and explains *why* each line exists, verified by millions of developers.

GDT Descriptor Encoding

Spec: Intel SDM Volume 3A, Chapter 3 "Protected-Mode Memory Management" — §3.4 "Segment Descriptors" (the canonical bit-field table every other resource copies from).

Code: Linux kernel — `arch/x86/kernel/cpu/common.c`, function `load_ucode_bsp()` and the `GDT_ENTRY` macro in `arch/x86/include/asm/segment.h` — exact production descriptor encoding.

Best Explanation: Writing a Simple Operating System — from Scratch, Nick Blundell (2010), Chapter 4. Free PDF. The only book that derives the access byte bit by bit with worked examples, rather than presenting the magic hex value.

Why: Blundell's derivation is the clearest treatment of why `0x9A` means what it means; reading it once eliminates the need to ever look up access bytes again.

Protected Mode Transition (CR0, far jump, pipeline flush)

Spec: Intel SDM Volume 3A, §9.9 "Switching to Protected Mode" — the Intel-specified four-step procedure your code must follow exactly.

Code: GRUB 2 — `grub-core/kern/i386/realmode.S`, `grub_real_to_prot` function — the production real-to-protected transition used by every GRUB-booted OS.

Best Explanation: *Operating Systems: Three Easy Pieces*, Remzi & Andrea Arpaci-Dusseau — Appendix on x86 architecture. Free at osstep.org. The pipeline-flush motivation (why a far jump is mandatory) is explained in terms a software engineer understands.

Why: OSTEP's appendix bridges the gap between "execute these instructions" and "understand why the CPU requires them."

8259 PIC Vector Remapping

Paper: None — legacy hardware spec.

Spec: Intel 8259A Programmable Interrupt Controller Datasheet (1988) — the definitive ICW1–ICW4 protocol. Findable via Intel's legacy documentation archive or University of Illinois CS archive.

Code: Linux kernel — `arch/x86/kernel/i8259.c`, `init_8259A()` function — the production PIC initialization, including the exact `io_wait` pattern and mask preservation.

Best Explanation: OSDev Wiki, "8259 PIC" — wiki.osdev.org/8259_PIC — the only resource that explains *why* IRQ0 maps to vector 8 by default and what happens when your double-fault handler fires as the timer handler.

Why: This page has prevented more triple faults than any other document in OS development history.

IDT Gate Descriptors & Interrupt Dispatch

Spec: Intel SDM Volume 3A, Chapter 6 "Interrupt and Exception Handling" — §6.11 "IDT Descriptors" and §6.12 "Exception and Interrupt Handling" (the exact CPU state push sequence, including the error-code asymmetry table).

Code: Linux kernel — `arch/x86/include/asm/idtentry.h` and `arch/x86/kernel/idt.c` — how a production kernel populates 256 IDT gates, distinguishing interrupt vs trap gates.

Best Explanation: James Molloy's *Kernel Development Tutorial* ("JamesM's Tutorial") — Interrupt Descriptor Table article at jamesmolloy.co.uk. The ISR stub macro pattern for handling the error-code asymmetry is taken directly from this tutorial by most learning kernels.

Why: Molloy's `ISR_NOERR/ISR_ERR` macro pattern is the canonical solution to the error-code asymmetry problem; every OS course derives from it.

x86 Two-Level Page Tables & TLB

Paper: Rao Ganapathy & Curt Schimmel, "General Purpose Operating System Support for Multiple Page Sizes" (USENIX ATC 1998) — the paper that motivates why page table structure choices have lasting performance consequences.

Spec: Intel SDM Volume 3A, Chapter 4 "Paging" — §4.3 "32-Bit Paging" (the authoritative PDE/PTE bit layout and the exact MMU walk algorithm).

Code: Linux kernel — `arch/x86/mm/pgtable.c` and `arch/x86/include/asm/pgtable_32_types.h` — the production 32-bit page table API, showing how PGD/PMD/PTE levels map to hardware structures.

Best Explanation: *Computer Systems: A Programmer's Perspective* (CS:APP), Bryant & O'Hallaron — Chapter 9 "Virtual Memory," §9.6 "Address Translation." The two-level page table walk diagram in Figure 9.22 is the clearest illustration in print.

Why: CS:APP's address translation treatment is the standard university reference and directly maps to what your MMU does at the hardware level.

TLB Shootdowns in SMP

Paper: Reinette Chatre et al., "Making the Most of TLB Shootdowns" — *Linux Plumbers Conference 2020*. Available at linuxplumbersconf.org. Directly relevant to understanding why your single-core `invlpg` becomes expensive coordination at scale.

Code: Linux kernel — `arch/x86/mm/tlb.c`, `flush_tlb_others_ipi()` — the production IPI-based TLB shootdown implementation with deferred batching optimization.

Best Explanation: *Understanding the Linux Kernel*, Bovet & Cesati — Chapter 2 "Memory Addressing," §2.5 "TLB Handling." Specifically the "Lazy TLB Mode" section explains the optimization that avoids shootdowns for kernel threads.

Why: The only book-length treatment that explains the gap between your `invlpg` and Linux's deferred IPI batching, with implementation detail.

Physical Memory Allocation (Bitmap vs Buddy)

Paper: Donald Knuth, *The Art of Computer Programming*, Volume 1 — §2.5 "Dynamic Storage Allocation." The original formal treatment of the buddy system that all subsequent allocators derive from.

Code: Linux kernel — `mm/page_alloc.c`, `__alloc_pages_nodemask()` — the production buddy allocator, including the `free_area[]` per-order free lists and the `__free_pages_ok()` buddy coalescing logic.

Best Explanation: *Design and Implementation of the FreeBSD Operating System*, McKusick et al. — Chapter 6 "Memory Management," §6.3 "Kernel Memory Allocator." Explains the zone allocator and slab system that sits atop the buddy system, motivating each layer.

Why: McKusick's treatment bridges the conceptual gap between your bitmap allocator and production multi-tier allocators in one chapter.

Kernel Heap Allocators (First-Fit → jemalloc)

Paper: Jason Evans, "A Scalable Concurrent malloc(3) Implementation for FreeBSD" (BSDCan 2006) — the original jemalloc paper. Available at jemalloc.net/papers. Explains size-class slabs, the design your linked-list allocator is the simplest ancestor of.

Code: jemalloc — `src/arena.c`, `arena_malloc_small()` — the production path for small allocations, showing how slab bitmaps eliminate per-object headers. github.com/jemalloc/jemalloc

Best Explanation: *Hoard: A Scalable Memory Allocator for Multithreaded Applications*, Berger et al. (ASPLOS 2000) — the paper that explains *why* linked-list first-fit fails at scale and what replaces it, in 10 pages.

Why: Reading Evans + Berger together gives the complete picture from "what's wrong with your kmalloc" to "what production allocators do instead."

Context Switching (Assembly-Level)

Spec: *System V Application Binary Interface — Intel386 Architecture Processor Supplement* — §3.2 "Function Calling Sequence." The ABI document that specifies which registers are caller-saved vs callee-saved, explaining precisely why C cannot implement a complete context switch.

Code: Linux kernel — `arch/x86/kernel/process_32.c`, `__switch_to()` and `arch/x86/entry/entry_32.S`, `ENTRY(ret_from_fork)` — the production 32-bit context switch, showing TSS updates, FPU lazy switching, and the `switch_to` macro.

Best Explanation: *Operating Systems: Three Easy Pieces* — Chapter 6 "Limited Direct Execution," §6.3 "Saving and Restoring Context." The diagram of the two-kernel-stack dance is the clearest published illustration of why `context_switch_asm` looks the way it does.

Why: OSTEP Chapter 6 is the canonical explanation for why context switching requires assembly; it's assigned in virtually every OS course.

TSS and Privilege Transitions

Spec: *Intel SDM Volume 3A*, §7.7 "Task State Segment (TSS)" and §6.14 "Exception and Interrupt Handling in 64-Bit Mode" (for comparison). The TSS field layout table is the authoritative source; no other resource should be trusted for byte offsets.

Code: Linux kernel — `arch/x86/include/asm/processor.h`, `struct tss_struct` and `arch/x86/kernel/process.c`, `__switch_to()` — exactly how Linux updates `tss.sp0` on every context switch.

Best Explanation: OSDev Wiki, "TSS" — wiki.osdev.org/TSS — explains the `ltr` instruction, the `iomap_base` field, and why software task switching only needs `esp0 / ss0`, in a form directly applicable to your implementation.

Why: The OSDev TSS article directly answers "why do I need this at all" and "what breaks if I skip it," which the Intel SDM does not.

Preemptive Round-Robin → CFS

Paper: Con Kolivas, "Modular Scheduler Core and Completely Fair Scheduler" — Linux kernel mailing list, 2007. The design rationale post for CFS is the clearest explanation of why round-robin's equal-time-slice approach fails for interactive workloads.

Code: Linux kernel — `kernel/sched/fair.c`, `enqueue_task_fair()` and `pick_next_task_fair()` — the complete CFS implementation showing `vruntime` accounting and the red-black tree.

Best Explanation: *Operating Systems: Three Easy Pieces* — Chapter 8 "Scheduling: The Multi-Level Feedback Queue" and Chapter 9 "Scheduling: Proportional Share." These two chapters connect your round-robin directly to MLFQ and CFS, motivating each evolution.

Why: OSTEP Chapters 8–9 are the standard treatment that every OS student reads; reading them after implementing round-robin makes every design decision obvious.

INT 0x80 → SYSCALL/SYSEENTER → vDSO

Paper: Linus Torvalds, "Re: [RFC] vDSO: using vsyscall page for system calls" — Linux kernel mailing list, 2002. The thread where the vDSO concept was designed. Archived at lkml.org.

Code: Linux kernel — `arch/x86/entry/vdso/vclock_gettime.c` — the vDSO implementation of `clock_gettime` that bypasses the kernel entirely, directly readable as a ~200-line C file. Also `arch/x86/entry/entry_32.S`, `ENTRY(system_call)` for the INT 0x80 path.

Best Explanation: *Brendan Gregg's blog* — "vDSO" entry at brendangregg.com. Gregg explains the syscall overhead measurement methodology and why the vDSO matters for `gettimeofday` latency in production systems.

Why: Gregg's practical measurement approach (strace + perf) shows you how to observe the cost of your INT 0x80 implementation versus SYSCALL versus vDSO in real numbers.

Demand Paging & Copy-on-Write

Paper: *The Design of the Unix Operating System*, Maurice Bach (1986) — Chapter 9 "Memory Management Policies." The original description of demand paging and copy-on-write as implemented in early Unix; still the clearest conceptual treatment.

Code: Linux kernel — `mm/memory.c`, `do_page_fault()` and `do_anonymous_page()` — the production page fault handler showing exactly what your mod-3 page fault handler would become with demand paging added.

Best Explanation: *Operating Systems: Three Easy Pieces* — Chapter 21 "Beyond Physical Memory: Mechanisms" and Chapter 22 "Beyond Physical Memory: Policies." These chapters take your page fault handler stub and show exactly what code would replace the `cli; hlt` in a production kernel.

Why: OSTEP's treatment of page fault handling is structured as "here is the mechanism you have, here is what the production version adds" — directly applicable to extending your mod-3 work.