Minimal Educational OS Kernel: Design Document

Overview

This document outlines the design for a minimal, educational operating system kernel built from scratch. The core architectural challenge is bootstrapping a system that transitions from raw hardware to a protected environment where user processes can run safely, requiring careful orchestration of CPU modes, memory mapping, and hardware abstraction.

> This guide is meant to help you understand the big picture before diving into each milestone. Refer back to it whenever you need context on how components connect.

1. Context and Problem Statement

> **Milestone(s):** This foundational section underpins all milestones, establishing the core problems the entire OS kernel must solve.

The Kernel as a Conductor and Guardian

Imagine an **orchestra** where each musician (hardware component) is a virtuoso: the CPU is a lightning-fast pianist, the memory is a vast choir, and the disk is a meticulous percussionist. Without coordination, their simultaneous, unconstrained efforts produce cacophony. The operating system kernel is the **conductor**. It doesn't play any instrument directly, but it dictates *when* the CPU executes instructions, *where* in memory data is placed, and *in what order* disk operations occur. It translates the high-level score of application programs into precise, timed instructions that each hardware component understands, ensuring the whole system performs in harmony.

Now, imagine a **large apartment building** (the computer) with many residents (user applications). Residents need shared resources: water (CPU time), electricity (memory), and the mailroom (I/O devices). If left unsupervised, a resident might monopolize the water supply or accidentally set the electrical wiring on fire. The kernel is the **building manager and security system**. It allocates resources fairly through scheduling and memory management, enforces rules to prevent accidents or malicious acts through privilege levels and memory protection, and provides secure, controlled interfaces (system calls) for residents to request services, like having a package delivered, without being allowed to drive the mail truck themselves.

These two analogies—the **conductor** and the **guardian**—capture the dual, essential roles of a kernel:

1. **Resource Manager & Abstraction Layer:** It virtualizes and multiplexes finite physical hardware (CPU, RAM, devices) among multiple, often competing, software processes. It provides higher-level, simpler abstractions (like files and processes) that hide the brutal, idiosyncratic complexity of the raw hardware.
2. **Protection & Isolation Enforcer:** It establishes and polices a strict boundary between "user mode" (where untrusted applications run) and "kernel mode" (where privileged control operations occur). This prevents a buggy or malicious program from crashing the entire system or accessing another program's private data.

Building a kernel from scratch, therefore, is not merely writing a large program. It is creating the **fundamental, privileged governance layer** that makes all other software possible. It is the first program the hardware loads, and it never relinquishes ultimate control.

The Core Problem: Bridging Hardware and Software

The fundamental technical challenge in kernel development is building a stable, secure, and efficient bridge between the abstract world of software and the concrete, often unforgiving, world of physical hardware. This bridge must be constructed in an environment with no pre-existing support—no standard library, no memory safety, and no error recovery beyond what you build. This challenge decomposes into three primary technical hurdles.

1. Bootstrapping: The Ultimate "Chicken and Egg" Problem A computer powers on into a primitive, minimal state. The CPU starts executing instructions from a predetermined, hardware-fixed memory address. There is no operating system in memory, no loader, no filesystem driver. The kernel itself is just bytes on a disk. The first hurdle is writing a **bootloader**—a tiny, special program that the BIOS or UEFI firmware can load using its rudimentary disk access routines. This bootloader must then perform the herculean task of loading the actual kernel binary into memory, configuring the CPU to enter a more powerful **protected mode** (which enables virtual memory and privilege separation), and finally jumping to the kernel's entry point. You are, in essence, writing the program that loads and sets the stage for *itself* to run.

> **Key Insight:** The boot process is a chain of increasingly sophisticated programs loading and initializing each other, each building upon the environment prepared by its predecessor until the full kernel is operational.

2. Hardware Diversity and Direct Manipulation Unlike application programming, where you rely on an OS to provide neat APIs, kernel development involves **direct hardware programming**. You will configure chipset registers, program interrupt controllers, and manipulate memory-mapped I/O regions. This requires deep, specification-level knowledge of:

- **CPU Modes:** Transitioning from 16-bit real mode to 32-bit protected mode, and later enabling paging.
- **Interrupt Descriptor Table (IDT) & Global Descriptor Table (GDT):** Data structures that tell the CPU where your interrupt handlers are and how memory segments are defined.

- **Programmable Interrupt Controller (PIC) / Advanced PIC (APIC):** Hardware chips that manage interrupts from devices like the keyboard and timer.
- **Memory Management Unit (MMU):** The hardware subsystem you program with page tables to implement virtual memory.

Each of these components has precise, often arcane, rules. Misprogramming them by even a single bit can lead to a **triple fault**—a CPU condition from which it cannot recover, causing an immediate system reset. There is no "debugger" in the conventional sense at this level; debugging often involves staring at hex dumps of memory and register states.

3. Resource Protection and Concurrency from the Ground Up Once the kernel is alive, it must immediately assume its role as guardian. This means:

- **Memory Isolation:** Ensuring Process A cannot read or write Process B's memory. This is achieved through **paging**, where you create separate, virtual address spaces for each process. The kernel must carefully manage the page tables that define these spaces.
- **Controlled Execution:** Preventing user programs from executing privileged CPU instructions (like `hlt` to halt the CPU or `in`/`out` to talk to devices directly). This is enforced by the CPU's **privilege rings** (Ring 0 for kernel, Ring 3 for user).
- **Fair Scheduling:** Sharing the single CPU core among multiple processes that all believe they have exclusive access. This requires implementing a **scheduler** that can transparently stop one process, save its entire execution state (registers, etc.), and resume another.
- **Safe Communication:** Providing a mechanism for user programs to request kernel services (like reading a keypress) without allowing them to jump arbitrarily into kernel code. This is the **system call interface**, typically implemented via a software interrupt (e.g., `int 0x80`) or a dedicated CPU instruction (e.g., `sysenter`).

The kernel must implement all these protection mechanisms while itself being fully trusted. A bug in the kernel's memory allocator or scheduler violates all security guarantees and typically crashes the entire system.

Existing Approaches and Our Path

The software architecture community has developed several broad models for structuring an operating system kernel, each with different trade-offs in complexity, performance, and reliability. For our educational kernel, we must choose a path that balances learning value with achievable complexity.

Kernel Model	Core Philosophy	Key Characteristics	Pros	Cons
Monolithic Kernel	The kernel is a single, large program running in a single, high-privilege address space.	All core services (scheduling, memory management, file systems, drivers) are part of the kernel binary and execute in kernel mode.	**Performance:** System calls and inter-component communication are fast (simple function calls). **Integrated Design:** Closely coupled components can be highly optimized.	**Reliability & Security:** A bug in any driver or subsystem can crash the entire kernel. **Maintainability:** The codebase becomes large and complex, making verification hard.
Microkernel	Minimize the kernel. Run as many services as possible (drivers, file systems) as separate, less-privileged "user-mode" processes.	The kernel provides only the bare minimum: low-level memory management, IPC (Inter-Process Communication), and scheduling.	**Reliability & Security:** Driver crashes do not take down the kernel. **Modularity:** Services can be developed, replaced, and restarted independently.	**Performance:** Communication between components requires costly IPC context switches. **Complexity:** Designing efficient, secure IPC is non-trivial.
Hybrid Kernel	Pragmatic blend. Combine aspects of both, often with a monolithic core but some modularity for drivers or certain subsystems.	Like a monolithic kernel but with some capacity for loadable modules and more structured internal boundaries.	**Balance:** Aims to get good performance while improving modularity over pure monolithic designs.	**Ambiguity:** The boundaries are not as clear-cut, which can lead to design compromises.

Decision: Monolithic Design for Educational Purposes

- **Context**: We are building a minimal, educational kernel from scratch with the primary goals of understanding fundamental OS concepts and seeing them work. We have limited development time and need to keep complexity manageable for a learner.
- **Options Considered**:
  1. **Pure Microkernel:** Start with a tiny kernel and build all services as separate processes.
  2. **Pure Monolithic Kernel:** Build all core services as part of the kernel binary.
- **Decision**: We will implement a **pure monolithic kernel**.
- **Rationale**:
  1. **Conceptual Simplicity:** A monolithic design has a linear progression: boot, initialize core subsystems, run processes. The control flow is primarily function calls within one address space, which is easier to trace and debug for a learner than complex IPC mechanisms.
  2. **Implementation Simplicity:** We avoid the significant upfront complexity of designing a secure, efficient IPC system and a driver process model. This allows us to focus on the core concepts of memory management, scheduling, and system calls sooner.
  3. **Performance Transparency:** The effects of our algorithms (scheduling, memory allocation) are more directly observable without the overhead of cross-process communication masking them.
  4. **Educational Alignment:** Classic OS teaching resources (like OSTEP) and historical kernels (like early UNIX or Linux) use monolithic designs, making it easier to map concepts from literature to our implementation.
- **Consequences**:
  - **Positive:** We can achieve a running system with processes and memory protection with less initial code and complexity.
  - **Negative:** Our device drivers (keyboard, timer) will run in kernel space. A bug in our keyboard driver will cause a full system crash, which is a valuable lesson in kernel programming rigor.
  - **Evolution Path:** The completed monolithic kernel serves as a perfect foundation for later exploration of microkernel concepts. One could, as a future extension, refactor a driver into a user-mode server to experience the trade-offs firsthand.

Our chosen path, therefore, is to construct a **minimal, monolithic, 32-bit protected mode kernel**. It will handle hardware abstraction, memory management, process isolation, and scheduling within a single privileged binary. This approach gives us the most direct route to experiencing the core challenges and triumphs of operating system development.

Key Takeaway: Building a kernel is the process of creating the foundational software layer that transforms inert hardware into a managed, multi-tasking computer. It requires solving the bootstrapping paradox, mastering low-level hardware programming, and implementing robust resource protection—all from a position of zero external support. Our monolithic design prioritizes educational clarity, allowing us to engage with these profound problems directly.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option (Recommended)	Advanced / Alternative Option
Boot Method	**Multiboot (GRUB)** - Use the Multiboot standard. This lets an existing, robust bootloader (GRUB) handle the messy details of loading your kernel, letting you start in 32-bit protected mode.	**Custom BIOS Bootloader** - Write your own bootloader in 16-bit assembly that loads your kernel from disk and switches to protected mode. More educational but more complex and error-prone.
Build System	**Make** - A simple `Makefile` to assemble `.asm` files, compile `.c` files, and link them into a kernel image.	**CMake or custom scripts** - More portable and powerful, but heavier for a small project.
Primary Language	**C** - The lingua franca of systems programming. Provides low-level control with enough abstraction to manage complexity.	**Rust** - Offers stronger memory safety guarantees, making some classes of kernel bugs (like buffer overflows) impossible. However, its learning curve and interaction with raw hardware/assembly can be steeper for an OS dev beginner.
Target Architecture	**32-bit x86 (i386)** - The most documented, tutorial-rich architecture for OS development. The concepts translate well to others.	**64-bit x86-64** or **ARM** - More modern but with increased initial complexity (e.g., Long Mode instead of Protected Mode).

B. Recommended File/Module Structure (Foundational) Even from the start, organizing your code is critical. Here is the suggested layout for the early milestones:

```
build-your-own-os/
├── Makefile                    # Build instructions
├── linker.ld                   # Linker script - defines kernel memory layout
├── boot/
│   └── multiboot_header.asm    # Multiboot header for GRUB
│   └── boot.asm                # Kernel entry point (if not using pure GRUB)
├── kernel/
│   ├── kernel.c                *Main kernel entry point (`kmain`) and core init*
│   ├── kernel.h                *Main header*
│   ├── drivers/
│   │   ├── vga.c               *VGA text mode driver (Milestone 1)*
│   │   ├── keyboard.c          *PS/2 Keyboard driver (Milestone 2)*
│   │   └── timer.c             *Programmable Interval Timer driver (future)*
│   ├── cpu/
│   │   ├── gdt.c               *Global Descriptor Table setup (Milestone 1)*
│   │   ├── idt.c               *Interrupt Descriptor Table setup (Milestone 2)*
│   │   ├── isr.asm             *Assembly interrupt service routine stubs*
│   │   └── interrupts.c        *C-level interrupt handler logic*
│   ├── mem/
│   │   ├── pmm.c               *Physical Memory Manager (Milestone 3)*
│   │   ├── vmm.c               *Virtual Memory Manager (Paging) (Milestone 3)*
│   │   └── heap.c              *Kernel heap allocator (Milestone 3)*
│   └── proc/                   *(For Milestone 4)*
│       ├── process.c           *Process Control Block (PCB) management*
│       ├── scheduler.c         *Round-robin scheduler*
│       └── switch.asm          *Context switch assembly routine*
└── lib/
    ├── string.c                *Basic memcpy, memset, strlen, etc.*
    ├── stdio.c                 *Formatted print (kprintf) for kernel logging*
    └── assert.h                *Kernel ASSERT macro for debugging*
```

C. Infrastructure Starter Code: The Multiboot Header and Kernel Entry To bypass the most finicky part of bootstrapping (writing a bootloader), we rely on the Multiboot standard. Place this in `boot/multiboot_header.asm`:

ASSEMBLY

```
; File: boot/multiboot_header.asm
; Declares a Multiboot-compliant header for GRUB.
; This must be within the first 8KB of the kernel binary.

section .multiboot_header
header_start:
    ; Magic number for Multiboot
    dd 0x1BADB002               ; magic
    ; Flags: align modules on page boundaries, provide memory map
    dd 0x00010003               ; flags (ALIGN + MEM_INFO)
    ; Checksum: magic + flags + checksum must equal 0
    dd -(0x1BADB002 + 0x00010003) ; checksum

    ; Optional fields (not used here, but space reserved as per spec)
    dd 0                        ; header_addr
    dd 0                        ; load_addr
    dd 0                        ; load_end_addr
    dd 0                        ; bss_end_addr
    dd 0                        ; entry_addr
    ; Graphics fields - set to 0 for text mode
    dd 0                        ; mode_type
    dd 0                        ; width
    dd 0                        ; height
    dd 0                        ; depth
header_end:
```

Now, we need a minimal kernel entry point in assembly that sets up a stack and calls our C main function. This goes in `boot/boot.asm`:

```assembly
; File: boot/boot.asm
; Kernel entry point. GRUB jumps here after loading the kernel.

section .text
global _start                   ; Entry symbol for linker
extern kmain                    ; Main C function
extern kernel_stack_top         ; Top of our dedicated kernel stack

_start:
    ; At this point, GRUB has loaded us into 32-bit protected mode.
    ; Interrupts are disabled. Paging is disabled.

    ; Set up a proper stack for the kernel.
    mov esp, kernel_stack_top   ; Point stack pointer to the top of the stack area

    ; Push the Multiboot information structure pointer (provided by GRUB in EBX)
    ; and magic number (in EAX) onto the stack as arguments for kmain.
    push ebx
    push eax

    ; Call the main kernel function written in C.
    call kmain

    ; If kmain ever returns (it shouldn't), hang the CPU.
    cli                         ; Disable interrupts
.hang:
    hlt                         ; Halt the CPU
    jmp .hang
```

D. Core Logic Skeleton: The Kernel Main Function This is the heart of your kernel's C code. It initializes all subsystems. The learner will fill in the calls to the initialization functions they build in each milestone.

```c
// File: kernel/kernel.c

#include "kernel.h"

#include "drivers/vga.h"

#include "cpu/gdt.h"

#include "cpu/idt.h"

#include "mem/pmm.h"

#include "mem/vmm.h"

#include "mem/heap.h"

#include "proc/scheduler.h"


// The main function called from boot.asm

void kmain(uint32_t mboot_magic, void* mboot_info) {

    // Milestone 1: Basic Setup & Output

    // TODO 1: Initialize the VGA text mode driver (clear screen, set default color)

    vga_init();


    // Print a welcome message to confirm we're alive.

    kprintf("Welcome to Build Your Own OS!\n");

    kprintf("Multiboot magic: 0x%x\n", mboot_magic);


    // TODO 2: Initialize the Global Descriptor Table (GDT) and load it.

    gdt_init();


    // Milestone 2: Interrupts

    // TODO 3: Initialize the Interrupt Descriptor Table (IDT) and load it.

    idt_init();

    // TODO 4: Remap the Programmable Interrupt Controller (PIC).

    pic_remap();

    // TODO 5: Initialize the keyboard driver (sets up interrupt handler).

    keyboard_init();

    // TODO 6: Enable interrupts (sti instruction) after everything is set up.

    asm volatile("sti");


    // Milestone 3: Memory Management

    // TODO 7: Parse the Multiboot memory map to initialize the Physical Memory Manager (PMM).

    pmm_init(mboot_info);

    // TODO 8: Initialize the Virtual Memory Manager (VMM) - set up page directories/tables.

    vmm_init();

    // TODO 9: Enable paging by setting the CR0 register.

    enable_paging();

    // TODO 10: Initialize the kernel heap allocator.

    heap_init();
```

```
    // Milestone 4: Process Management
    // TODO 11: Initialize the process manager and create the first kernel/idle process.
    // TODO 12: Initialize the scheduler.
    scheduler_init();
    // TODO 13: Create a simple user process for testing (load an ELF, set up its page table).
    // create_init_process();

    kprintf("Kernel initialization complete.\n");

    // Main kernel loop. The scheduler will take over from here.
    while (1) {
        // TODO 14: If no process is ready to run, halt the CPU (sti + hlt).
        asm volatile("hlt");
    }
}

// A simple kernel printf. You must implement vga_puts in your VGA driver.
void kprintf(const char* fmt, ...) {
    // TODO: Implement a basic variable argument formatter.
    // For now, just print the string directly.
    vga_puts(fmt);
}
```

E. Language-Specific Hints (C)

- **Inline Assembly:** Use `asm volatile()` for instructions that must not be moved by the compiler (like `cli`, `sti`, `outb`).
- **Pointers to Hardware:** Declare device registers as `volatile uint32_t*` to prevent the compiler from optimizing away or reordering accesses.
- **Pack Structures:** Use `__attribute__((packed))` for data structures that map directly to hardware (like GDT/IDT entries) to avoid unwanted padding inserted by the compiler.
- **Linker Script:** You will need a `linker.ld` file to place the `.multiboot_header` section at the very start of the binary and to define symbols like `kernel_stack_top`. This is crucial for the boot process.

F. Milestone 1 Checkpoint After implementing the boot logic and VGA driver from Milestone 1, you should be able to:

1. **Build:** Run `make` to produce a kernel binary (e.g., `kernel.bin`).
2. **Run in Emulator:** Run `qemu-system-i386 -kernel kernel.bin`.
3. **Expected Output:** The QEMU window should display a clean screen with the message "Welcome to Build Your Own OS!" followed by the Multiboot magic number (like `0x2BADB002`).
4. **Signs of Trouble:**
    - **QEMU window closes instantly/blank screen:** Likely a linker script issue or incorrect Multiboot header. Check that the header is at the very start of the binary.
    - **Garbled text or no text:** The VGA buffer address (0xB8000) is likely wrong, or you are writing data to the wrong location within the buffer.
    - **Triple fault/reset loop:** The CPU is crashing early. Check your GDT setup code—ensure the GDT pointer is correctly aligned and contains valid segment descriptors.

Next Step: With the kernel booting and displaying text, you have conquered the first major hurdle. You now have a "stage" upon which to build the rest of your operating system. The next section will delve into the design of the interrupt system, which is the kernel's mechanism for responding to external events.

2. Goals and Non-Goals

> **Milestone(s):** This section defines the foundational scope that guides the implementation of all four milestones.

This design document describes a minimal, educational operating system kernel. Its primary purpose is to serve as a **pedagogical tool**—a concrete artifact through which a developer can understand the foundational mechanisms that bridge hardware and software. Achieving this pedagogical goal requires a razor-sharp focus on core concepts while deliberately omitting features that, while essential for production systems, would distract from the learning objectives.

Think of this project as building a **functional scale model of a city**, not a livable metropolis. The model must have working infrastructure—roads, zoning, basic utilities—to demonstrate how a real city's systems interconnect and function. However, it does not need the full complexity of a sewer treatment plant, a financial district, or public transportation; those can be studied later once the core layout is understood. Similarly, our kernel will implement the essential machinery that makes an OS work, but will not attempt to be robust, feature-complete, or efficient enough for daily use.

The following goals and non-goals are defined to maintain this focus. Adherence to them ensures the project remains a manageable, enlightening journey into kernel development rather than an overwhelming attempt to rebuild Linux.

Must-Have Goals (Functional Requirements)

These are the **indispensable capabilities** the kernel must implement to fulfill its educational mission. Each capability corresponds directly to a project milestone and represents a foundational OS concept that a developer must understand to claim comprehension of how an operating system works at its core.

The table below maps each milestone to its primary functional requirements and the key concepts they demonstrate:

Milestone	Primary Functional Requirement	Key Demonstrated Concept	Expected Observable Outcome
1. Bootloader & Kernel Entry	The system must boot from a standard PC BIOS, transition the CPU to 32-bit protected mode, load the kernel binary into memory, and display text on screen.	Bootstrapping, CPU mode transitions, memory-mapped I/O.	When booted in an emulator (QEMU), the screen displays a colored welcome message (e.g., "MyOS Kernel v1.0") and a prompt.
2. Interrupts & Keyboard	The kernel must handle hardware interrupts (timer, keyboard) and CPU exceptions, providing a mechanism for asynchronous event handling and basic user input.	Interrupt Descriptor Table (IDT), Programmable Interrupt Controller (PIC), device driver patterns.	Pressing keys on the keyboard echoes the corresponding ASCII characters to the screen. A CPU exception (e.g., divide-by-zero) prints a diagnostic message instead of triple-faulting.
3. Memory Management	The kernel must manage physical memory, implement virtual memory via paging, and provide dynamic memory allocation for kernel data structures.	Physical/virtual address separation, page tables, heap allocation algorithms.	The kernel can allocate and free memory pages on demand. A user program (later) can be loaded at a arbitrary virtual address. Access to unmapped memory causes a page fault.
4. Process Management	The kernel must support multiple execution contexts (processes), switch between them via context switching, schedule CPU time, and provide a controlled system call interface.	Process isolation, context switching, scheduling, privilege levels (kernel vs. user mode).	Two or more simple user programs can run concurrently, taking turns on the CPU. A user program can invoke a system call (e.g., to print) without crashing the kernel.

Beyond this milestone mapping, the kernel must satisfy the following overarching functional requirements that span multiple components:

1. **Monolithic Kernel Architecture**: All core kernel services (memory manager, scheduler, device drivers) shall run in a single, privileged address space (`Ring 0`). This simplifies initial development and is the most straightforward model for understanding kernel internals.
2. **Bare-Metal Execution**: The kernel must run directly on the physical hardware (or a faithful emulator like QEMU) without reliance on a host operating system. It is the first software to run after the firmware (BIOS) hands off control.
3. **Deterministic Initialization**: The system must have a clear, sequential boot process where each subsystem is initialized in a correct order (e.g., GDT → IDT → memory manager → scheduler).
4. **Minimal Hardware Support**: The kernel will target a minimal, standardized hardware profile: legacy PC-compatible (i386/IA-32) architecture with VGA text mode, a PS/2 keyboard, and a PIT (Programmable Interval Timer). This maximizes compatibility with emulators and simplifies driver development.
5. **Diagnostic Output**: The kernel must provide a reliable, lowest-common-denominator method for reporting its state and debugging information. The primary method will be writing text to the VGA text buffer. A secondary method using the serial port is highly recommended for more advanced debugging.
6. **Graceful Failure Modes**: When the kernel encounters an unrecoverable error (e.g., a kernel panic), it must halt the CPU in a controlled manner, providing as much diagnostic information as possible on screen, rather than silently hanging or rebooting.

Explicit Non-Goals

To preserve educational focus, the following features are **explicitly excluded** from the scope of this minimal kernel. Each represents a significant area of complexity that is best tackled only after the fundamentals are solidly understood.

Non-Goal	Rationale for Exclusion	Potential for Future Extension
User-Friendly APIs or POSIX Compliance	The system call interface will be minimal and ad-hoc, designed only to demonstrate the mechanism of privilege transition. Implementing a full POSIX-like API (files, sockets, processes) is a massive undertaking that obscures the underlying hardware mechanisms.	A future project could implement a subset of POSIX system calls atop the minimal framework built here.
Advanced Filesystems	The kernel will not initially read from any disk or filesystem. The kernel binary is loaded in its entirety by the bootloader. File systems introduce complex data structures (inodes, directories), caching, and block device drivers that are a separate, advanced topic.	A simple read-only filesystem (e.g., initrd) could be added to load user programs, followed by a more complex read-write FS.
Networking (TCP/IP Stack)	Networking involves complex protocols, packet buffering, interrupt handling for network cards, and concurrency issues that are an order of magnitude more complex than basic keyboard handling.	A simple NE2000 driver and a minimal IP stack could be a capstone project after process isolation is stable.
Graphical User Interface (GUI)	Moving beyond VGA text mode requires managing video card hardware, graphics modes, framebuffers, and composition, which is a substantial project in itself (a "driver" project).	A basic linear framebuffer driver and a simple graphics library could be a separate milestone.
Power Management (ACPI/APM)	Properly shutting down or suspending a PC requires intricate coordination with system firmware and chipset-specific registers, which is not relevant to core OS concepts.	The kernel will simply halt the CPU (`hlt` instruction) when done.
Multiprocessor (SMP) Support	Synchronizing data structures and scheduling across multiple CPU cores introduces deep concurrency challenges (lock-free algorithms, IPIs) that are best studied after a single-COS kernel is complete.	The initial design should avoid global locks that would break with SMP, making future extension easier.
Real-Time or Fair Scheduling	The scheduler will use a simple, naive round-robin algorithm. Implementing priority-based, multi-level feedback queue, or real-time schedulers involves significant policy complexity.	The scheduler algorithm is modular; a more advanced one can be swapped in later.
Advanced Security (ASLR, Capabilities)	While the kernel establishes basic memory protection via paging, advanced exploit mitigations require sophisticated compiler support, randomization, and security models beyond educational scope.	Address Space Layout Randomization (ASLR) could be added as an extension to the page table allocation logic.
Dynamic Module Loading	All kernel code is linked into a single binary. Loadable kernel modules require careful versioning, symbol resolution, and runtime linking, adding substantial complexity.	The design could later be refactored to expose stable driver APIs to support modules.
Comprehensive Device Driver Suite	Only the essential devices for interaction (keyboard, timer) and output (VGA) will be supported. Supporting USB, sound cards, or even mice is a diversion.	The driver model established for the keyboard can serve as a template for adding more devices later.
Efficiency and Performance Optimizations	The kernel's primary metric is correctness and clarity, not speed. Techniques like slab allocators, copy-on-write, zero-copy I/O, or TLB shootdown optimization are out of scope.	Once the system works, profiling can identify bottlenecks to optimize in a later iteration.
Persistent Storage & Logging	The kernel will not write to disk. All state is lost on reboot. Implementing a logging subsystem or crash dumps adds persistence and recovery complexity.	A simple logging system to the serial port is a good intermediate step before tackling disk I/O.

> **Design Principle: Scope as a Learning Tool**
> The most critical skill taught by this project is not how to implement any specific feature, but how to make **informed scoping decisions**. By rigorously defining what we are *not* building, we free up intellectual bandwidth to deeply understand the components we *are* building. Each non-goal represents a conscious trade-off: complexity postponed for the sake of foundational comprehension.

---

3. High-Level Architecture

> **Milestone(s):** This architectural overview provides the structural blueprint for all four milestones, defining how the bootloader, core kernel, device drivers, memory manager, and process manager interact to form a complete system.

This section provides a bird's-eye view of the kernel's major components and their relationships. Before diving into implementation details, we establish a clear mental model of the system's organization—how responsibilities are divided, how components communicate, and how data flows between them. This architectural clarity is crucial for maintaining a clean separation of concerns in a codebase that must manage extreme hardware complexity.

System Component Overview

Think of the operating system as a **theater production company** putting on a play. The **bootloader** is the stage crew that prepares the theater before the audience arrives—setting up the stage, lighting, and seating. The **core kernel** is the director and stage manager, overseeing the entire production and making real-time decisions. **Device drivers** are the specialized technicians—lighting operators, sound engineers, and prop masters—who handle specific hardware equipment. The **memory manager** is the venue's floor plan architect and librarian, allocating spaces and keeping track of where everything is stored. Finally, the **process manager** is the casting director and timekeeper, managing which actors (processes) get to perform on stage and for how long.

Each component has clearly defined responsibilities and communication protocols, ensuring the system operates efficiently without components stepping on each other's toes.

Component Responsibilities and Dependencies

The following table details the major software components, their core responsibilities, what data they own, and how they interact with other components:

Component	Primary Responsibility	Key Data Owned	Key Dependencies	Communication Method
Bootloader	Load kernel binary from disk into memory at a predefined address, initialize CPU to 32-bit protected mode, set up minimal environment, and jump to kernel entry point.	Boot parameters, memory map from BIOS/UEFI, GDT (Global Descriptor Table) during transition.	BIOS/UEFI firmware, disk controller, memory.	Direct calls (kernel entry), CPU mode transitions, memory writes (loading kernel).
Core Kernel	Initialize all subsystems, provide foundational services (interrupt handling, system calls), coordinate between components, and maintain overall system stability.	Initial stack pointer, interrupt descriptor table (IDT), system call table, kernel panic state.	Bootloader (for initial parameters), all other kernel components.	Function calls, interrupt handlers, shared global state (minimal).
Device Drivers	Abstract hardware-specific details of peripheral devices, provide uniform interfaces to the kernel, handle device interrupts, and manage device state.	Device configuration registers, internal buffers, device state machines.	Core kernel (for interrupt registration, I/O port access), memory manager (for DMA buffers).	Interrupt Service Routines (ISRs), programmed I/O, memory-mapped I/O, kernel API calls.
Memory Manager	Manage physical and virtual memory resources: allocate/free physical frames, map virtual addresses to physical frames via page tables, provide kernel heap allocation.	Frame allocation bitmap/stack, page directory and tables, heap metadata structures.	Bootloader (for memory map), hardware (MMU, TLB).	Function calls (`pmm_alloc` , `vmm_map`), page fault exceptions, TLB flush instructions.
Process Manager	Create, schedule, and manage execution contexts (processes): maintain process state, perform context switches, enforce isolation, and provide system call interface.	Process Control Blocks (PCBs), scheduler queue, current process pointer.	Memory manager (for process address spaces), core kernel (for timer interrupts).	Timer interrupts (for preemption), system call interrupts (`int 0x80`), direct function calls for management.

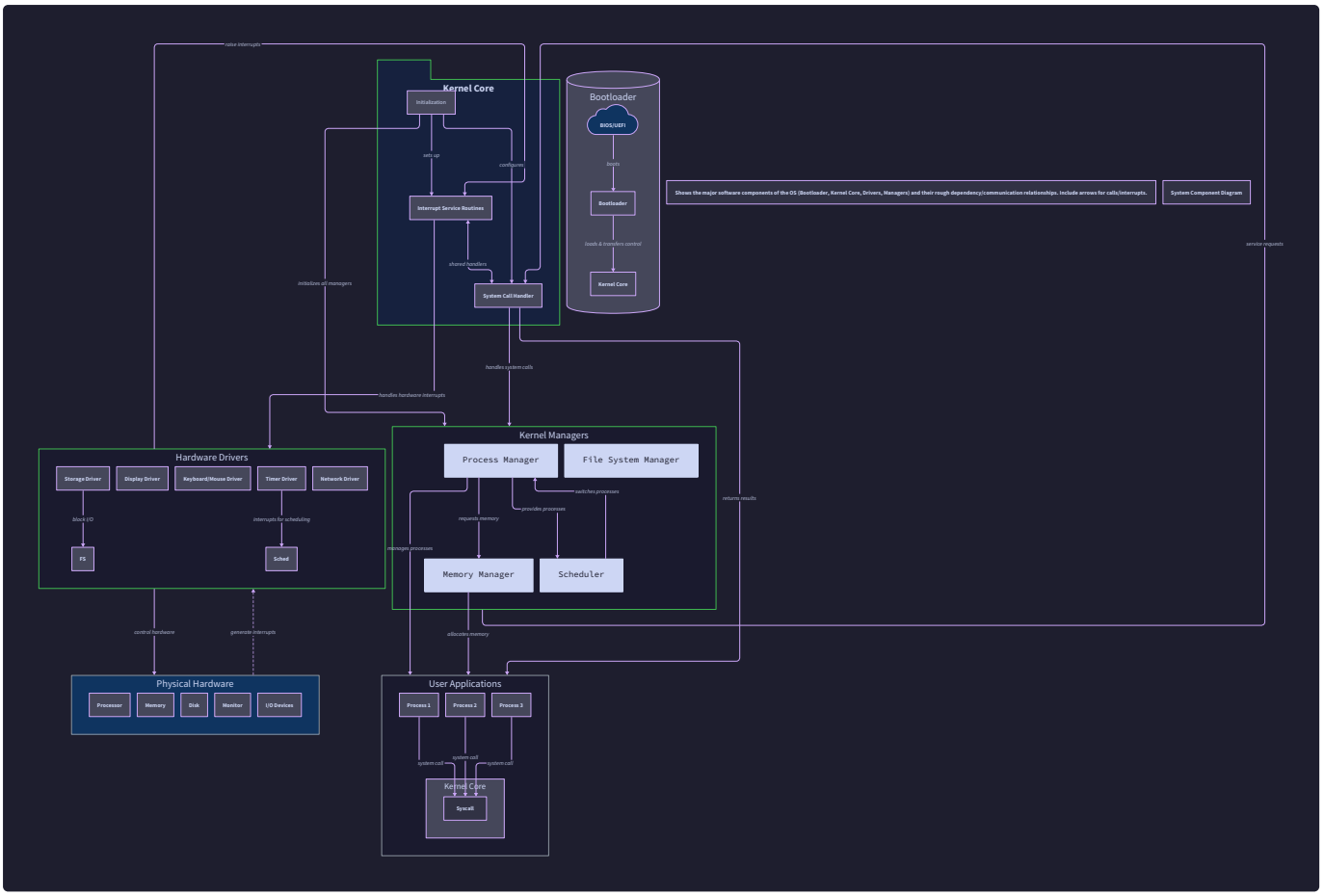
These components form a classic **monolithic kernel** architecture, where all core services run in a single, privileged address space. This design minimizes performance overhead from cross-domain communication (no user/kernel boundary crossings for internal services) at the cost of reduced fault isolation between subsystems. For our educational kernel, this trade-off is acceptable—simplicity and performance for learning outweigh the robustness benefits of a microkernel.

The component dependency graph shows a clear bootstrap hierarchy:

1. The **bootloader** initializes the minimal environment for the **core kernel**.
2. The **core kernel** initializes the **memory manager** (which needs to know available physical memory).
3. With memory management functional, the **core kernel** initializes **device drivers** (which may need DMA buffers from the memory manager).
4. Finally, the **process manager** is initialized, creating the first kernel process and enabling multitasking.

Communication flows in multiple directions:

- **Synchronous calls**: Higher-level components call lower-level ones (e.g., process manager calls memory manager to allocate process memory).
- **Asynchronous interrupts**: Hardware devices interrupt the CPU, triggering ISRs in device drivers, which may then notify the process manager (e.g., keyboard input wakes a waiting process).
- **System calls**: User processes trigger software interrupts that vector to the core kernel's system call dispatcher, which may call any component.



Architecture Decision Record: Monolithic vs. Microkernel Design

Decision: Adopt a Monolithic Kernel Architecture

- **Context**: We are building a minimal, educational OS kernel where the primary goals are understanding fundamental concepts and achieving functional milestones with manageable complexity. The kernel must handle hardware abstraction, memory management, and process scheduling within a limited codebase.
- **Options Considered**:
  1. **Pure Monolithic Kernel**: All core services (memory management, process management, device drivers, file systems) run in kernel space with full hardware privileges.
  2. **Microkernel**: Minimal kernel providing only process scheduling, inter-process communication (IPC), and basic memory management; all other services run as user-mode servers.
  3. **Hybrid (Linux-style)**: Monolithic core but with modules that can be loaded/unloaded dynamically, offering some isolation for driver development.
- **Decision**: We will implement a pure monolithic kernel for all core services.
- **Rationale**:
  - **Educational clarity**: A monolithic design allows us to trace the entire flow of operations—from system call to hardware manipulation—within a single codebase, making relationships between components more transparent.
  - **Performance**: Eliminates context switches and IPC overhead for kernel services, which is important for a minimal system with limited optimization opportunities.
  - **Implementation simplicity**: No need to design complex IPC mechanisms, capability systems, or user-mode server infrastructure for early milestones.
  - **Industry precedent**: Many successful educational and real-world kernels (Linux, FreeBSD, xv6) use monolithic designs, providing ample reference material.
- **Consequences**:
  - **Positive**: Simplified debugging (no cross-address-space issues), straightforward data sharing between components, easier reasoning about system state.
  - **Negative**: Reduced fault isolation—a buggy device driver can crash the entire system, and there's no memory protection between kernel components. This is acceptable for our educational context where reliability is secondary to understanding.
  - **Future implications**: Extending the kernel with additional services will add to the monolithic codebase; converting to a microkernel later would require significant architectural changes.

Option	Pros	Cons	Chosen?
Pure Monolithic	Simple to implement and debug; high performance; clear data flow for education.	No fault isolation between components; larger trusted computing base.	**Yes**
Microkernel	Excellent fault isolation; modularity; easier to extend and maintain long-term.	Complex IPC required; performance overhead from context switches; steeper learning curve.	No
Hybrid	Balance of performance and modularity; driver isolation possible via modules.	Increased complexity from module loading/versioning; still mostly monolithic.	No

Recommended File/Module Structure

A well-organized codebase is essential for managing complexity in a kernel project. Think of the file structure as the **organizational chart of a company**—each department (directory) has a specific focus, with clear reporting lines (header inclusions) and defined interfaces (APIs) between them. This structure enforces separation of concerns, making it easier to locate code, understand dependencies, and avoid circular references.

We adopt a **feature-based directory structure** where components are grouped by functional area rather than file type (e.g., not separating all header files into an `include/` directory). This approach keeps related code close together, which is beneficial for a monolithic kernel where components are tightly coupled but should maintain clean interfaces.

Directory Layout

```
build-your-own-os/
├── boot/                       # Bootloader code (x86 assembly)
│   ├── boot.asm                # Primary bootloader (BIOS MBR or UEFI stub)
│   ├── gdt.asm                 # GDT definition and loading routines
│   └── multiboot.asm           # Multiboot header for compatibility with GRUB
├── kernel/
│   ├── arch/x86/               # Architecture-specific code for x86
│   │   ├── gdt.c               # GDT initialization and descriptor management
│   │   ├── idt.c               # IDT setup and interrupt gate installation
│   │   ├── isr.asm             # Assembly ISR stubs (common entry points)
│   │   ├── isr.c               # C ISR handlers and dispatch logic
│   │   ├── pic.c               # PIC (8259) configuration and remapping
│   │   ├── paging.c            # Page table manipulation, CR3 loading, TLB flushes
│   │   └── syscall.asm         # System call entry point (int 0x80 handler)
│   ├── drivers/                # Hardware device drivers
│   │   ├── vga.c               # VGA text mode buffer driver
│   │   ├── keyboard.c          # PS/2 keyboard driver and scancode translation
│   │   ├── timer.c             # Programmable Interval Timer (PIT) driver
│   │   └── serial.c            # Serial port driver for debugging output
│   ├── mm/                     # Memory management subsystem
│   │   ├── pmm.c               # Physical Memory Manager (frame allocator)
│   │   ├── vmm.c               # Virtual Memory Manager (page tables)
│   │   └── heap.c              # Kernel heap allocator (malloc/free)
│   ├── process/                # Process management and scheduling
│   │   ├── pcb.c               # Process Control Block allocation/management
│   │   ├── scheduler.c         # Round-robin scheduler and queue management
│   │   ├── context_switch.asm  # Assembly routine for saving/restoring registers
│   │   └── syscall.c           # System call implementation and dispatch table
│   ├── lib/                    # Kernel utility libraries
│   │   ├── string.c            // memset, memcpy, strlen, etc.
│   │   ├── printf.c            // Formatted printing (supports %s, %d, %x)
│   │   └── debug.c             // Assertions, panic, stack trace
│   ├── include/                // Global kernel headers (shared across components)
│   │   ├── kernel.h            // Common types, constants, and global includes
│   │   ├── arch/x86/           // Architecture-specific headers
│   │   │   ├── gdt.h           // GDT entry structure, GDT pointer
│   │   │   ├── idt.h           // IDT entry structure, IDT pointer
│   │   │   ├── isr.h           // ISR function prototypes, exception numbers
│   │   │   └── paging.h        // Page table structures, flags
│   │   ├── drivers/            // Driver interfaces
│   │   │   ├── vga.h           // VGA constants, function prototypes
│   │   │   └── keyboard.h      // Scancode definitions, keyboard functions
│   │   ├── mm/                 // Memory management interfaces
│   │   │   ├── pmm.h           // Frame allocation prototypes
│   │   │   ├── vmm.h           // Virtual memory mapping prototypes
│   │   │   └── heap.h          // Heap allocation prototypes
│   │   └── process/            // Process management interfaces
│   │       ├── pcb.h           // PCB structure definition
│   │       ├── scheduler.h     // Scheduler function prototypes
│   │       └── syscall.h       // System call numbers and prototypes
│   ├── kernel.c                // Main kernel entry point (kmain)
│   └── linker.ld               // Linker script defining kernel memory layout
├── scripts/                    // Build and utility scripts
│   ├── build.sh                // Main build script (compile, assemble, link)
│   └── run-qemu.sh             // QEMU launch script with debug options
└── Makefile                    // Top-level build configuration
```

Key Structural Principles

1. **Architecture Isolation**: All x86-specific code lives under `kernel/arch/x86/`. This clear separation means porting to another architecture (e.g., ARM) would involve creating a new `arch/arm/` directory with implementations of the same interfaces (GDT, IDT, paging setup).
2. **Component Co-location**: Each major component (drivers, memory management, process management) has its own directory containing both implementation (.c files) and private headers. Public APIs are exposed via the central `include/` directory.
3. **Layered Dependencies**: Dependencies flow downward in the directory tree:
   - `kernel.c` (top-level) includes headers from `include/` and calls initialization functions from all components.
   - `process/` depends on `mm/` for memory allocation and `drivers/` for timer interrupts.
   - `mm/` depends on `arch/x86/` for page table manipulation but not on higher-level components.
   - `drivers/` depend only on `arch/x86/` for I/O port access and interrupt registration.
   - `lib/` utilities have no dependencies and can be used by any component.
4. **Header Discipline**: Every `.c` file includes its corresponding `.h` file first, followed by system headers. Header files use include guards (`#ifndef HEADER_NAME_H`) to prevent multiple inclusions. Public headers in `include/` should not include other public headers unless absolutely necessary to avoid

creating implicit dependency chains.

5. **Assembly Separation**: Assembly files are kept separate from C files and are placed in appropriate component directories. For instance, context switching assembly is in `process/context_switch.asm` because it's part of process management, while ISR stubs are in `arch/x86/isr.asm` because they're architecture-specific interrupt entry points.

Architecture Decision Record: Feature-Based vs. Type-Based Directory Structure

> **Decision: Use Feature-Based Directory Organization**
>
> - **Context**: We need to organize kernel source code in a way that minimizes cognitive load for learners, makes dependencies clear, and supports the monolithic architecture while allowing components to be developed and understood independently.
> - **Options Considered**:
>   1. **Feature-Based**: Directories organized by functional component (`mm/` , `process/` , `drivers/`), with each containing both implementation and internal headers.
>   2. **Type-Based**: Traditional separation of `.c` files in `src/` , `.h` files in `include/` , with subdirectories mirroring component names in both.
>   3. **Flat Structure**: All kernel `.c` and `.h` files in a single directory with naming conventions (`mm_paging.c` , `proc_scheduler.c`).
> - **Decision**: We will use a feature-based directory structure with a central `include/` directory for public interfaces.
> - **Rationale**:
>   - **Discoverability**: Learners working on memory management can find all related code in `mm/` without searching through unrelated files.
>   - **Encapsulation**: Components are naturally encapsulated—private helper functions and structures stay in the component directory, exposed only via public headers in `include/` .
>   - **Build simplicity**: Each directory can have its own compilation rules if needed, and dependencies are explicit through inclusion paths.
>   - **Educational alignment**: Mirrors how OS concepts are taught (as separate subsystems) and how most educational kernels (xv6, OSDev tutorials) are organized.
> - **Consequences**:
>   - **Positive**: Clear separation of concerns; easier to reason about component boundaries; simplifies testing of individual components in isolation.
>   - **Negative**: Some duplication of header files (component-private headers in component dirs, public ones in `include/`); slightly more complex include paths (`-I` flags).
>   - **Future implications**: Adding new components is straightforward—create a new directory and corresponding public headers. Converting to a microkernel would be easier as each directory could become a separate server.

Option	Pros	Cons	Chosen?
Feature-Based	Intuitive for learners; clear component boundaries; easy to navigate.	Header management slightly complex; some path configuration needed.	**Yes**
Type-Based	Traditional for C projects; clear separation of interface/implementation.	Forces jumping between directories for related code; less intuitive for learners.	No
Flat Structure	Extremely simple; no directory navigation needed.	Becomes unmanageable beyond ~20 files; no logical grouping.	No

Module Interface Definitions

To ensure clean separation between components, we define explicit interfaces for each major subsystem. These interfaces are declared in the public headers under `include/` and implemented in the corresponding component directories.

Memory Management Interface:

Method	Parameters	Returns	Description
`pmm_init`	`void* mboot_info`	`void`	Initializes physical frame allocator using memory map from Multiboot info.
`pmm_alloc_frame`	`void`	`uintptr_t`	Allocates a single 4KB physical frame; returns physical address or 0 on failure.
`pmm_free_frame`	`uintptr_t addr`	`void`	Marks the specified physical frame as free for reuse.
`vmm_init`	`void`	`void`	Sets up initial page tables, identity-maps kernel, enables paging.
`vmm_map_page`	`uintptr_t virt, uintptr_t phys, uint32_t flags`	`int`	Maps virtual address to physical address with specified flags (present, writable, etc.).
`vmm_unmap_page`	`uintptr_t virt`	`int`	Removes mapping for virtual address; frees physical frame if requested.
`heap_init`	`void`	`void`	Initializes kernel heap allocator using a region of virtual memory.
`kmalloc`	`size_t size`	`void*`	Allocates memory from kernel heap; returns pointer or NULL.
`kfree`	`void* ptr`	`void`	Frees previously allocated heap memory.

Process Management Interface:

Method	Parameters	Returns	Description
`scheduler_init`	`void`	`void`	Initializes scheduler with idle process and sets up timer tick.
`process_create`	`void (*entry)(void)`	`pid_t`	Creates new process with given entry point; returns PID or -1 on failure.
`process_exit`	`int exit_code`	`void`	Terminates current process, freeing resources and scheduling next.
`scheduler_yield`	`void`	`void`	Voluntarily yields CPU to next runnable process.
`scheduler_tick`	`void`	`void`	Called on each timer interrupt; updates timeslices and triggers reschedule.
`syscall_handler`	`int num, uint32_t arg1, uint32_t arg2, ...`	`uint32_t`	Dispatches system call based on number; returns result to user process.

Device Driver Interface (Example: Keyboard):

Method	Parameters	Returns	Description
`keyboard_init`	`void`	`void`	Initializes PS/2 controller, sets up interrupt handler, enables IRQ1.
`keyboard_get_scancode`	`void`	`uint8_t`	Reads raw scancode from keyboard controller (non-blocking).
`keyboard_getchar`	`void`	`char`	Blocks until a printable character is available; returns ASCII.
`keyboard_set_leds`	`bool scroll, bool num, bool caps`	`void`	Sets keyboard LED states (Scroll Lock, Num Lock, Caps Lock).

These interfaces define the **contracts** between components. For example, the process manager calls `vmm_map_page` through the memory management interface to set up process address spaces, but it doesn't need to know the internal implementation of page tables. Similarly, device drivers register their interrupt handlers with the core kernel's IDT management functions but handle device-specific details internally.

Common Pitfalls: Architecture and Structure

⚠️ Pitfall: Circular Dependencies Between Components

- **Description**: Component A includes header from component B, which includes header from component A, creating a compile-time circular dependency.
- **Why it's wrong**: The compiler cannot resolve the dependencies, leading to "incomplete type" errors or infinite inclusion loops. This breaks the separation of concerns and creates tightly coupled code that's hard to maintain.
- **How to fix**: Restructure headers to use forward declarations where possible. For example, if `pcb.h` needs a pointer to a `page_table_t` defined in `paging.h`, use `typedef struct page_table page_table_t;` instead of including the full header. Alternatively, create a third, shared header with only the minimal type definitions needed by both components.

⚠️ Pitfall: Global State Pollution

- **Description**: Creating numerous global variables accessible across components (e.g., `extern struct process* current_process;` used everywhere).

- **Why it's wrong**: Makes code hard to reason about, creates hidden dependencies, and complicates testing. In a monolithic kernel, some global state is unavoidable (like `current_process`), but excessive globals undermine modularity.
- **How to fix**: Limit global variables to truly system-wide singletons. For component-specific state, use static variables within the component's implementation files and provide accessor functions if needed from other components. For example, instead of exposing the scheduler queue globally, provide `scheduler_get_next_process()` function.

⚠️ **Pitfall: Architecture-Specific Code Leakage**

- **Description**: Putting x86-specific code (like inline assembly for port I/O) directly in driver implementations instead of abstracting through architecture interfaces.
- **Why it's wrong**: Makes porting to another architecture extremely difficult, as x86 assumptions are scattered throughout the codebase.
- **How to fix**: Isolate all architecture-dependent operations in `arch/x86/` directory. Create abstract interfaces for common operations (I/O ports, interrupt control) that have architecture-specific implementations. For example, instead of `outb(0x60, data)` in keyboard driver, call `arch_port_write_byte(0x60, data)` which is implemented in `arch/x86/io.c`.

Implementation Guidance

This section provides concrete guidance for implementing the high-level architecture, focusing on the recommended file structure and essential infrastructure code.

A. Technology Recommendations Table:

Component	Simple Option	Advanced Option
Boot Method	BIOS + Multiboot (using GRUB as bootloader)	UEFI with custom bootloader
Build System	Makefile with explicit rules for each file type	CMake with cross-compiler toolchain
Debugging	QEMU with serial output and GDB stub	Bochs with internal debugger and memory watchpoints
Testing	Manual verification via screen output and keyboard input	Automated QEMU tests with expect scripts
Code Quality	Manual code review and adherence to style guide	Clang static analyzer, custom linting scripts

B. Recommended File/Module Structure Starter:

Create the following directory structure as the foundation for your kernel:

```bash
mkdir -p build-your-own-os/{boot,kernel/{arch/x86,drivers,mm,process,lib,include/{arch/x86,drivers,mm,process}}}
```

Create a minimal `kernel/include/kernel.h` to establish common types and macros:

```c
#ifndef KERNEL_H

#define KERNEL_H


// Standard integer types

typedef unsigned char uint8_t;

typedef unsigned short uint16_t;

typedef unsigned int uint32_t;

typedef unsigned long long uint64_t;

typedef char int8_t;

typedef short int16_t;

typedef int int32_t;

typedef long long int64_t;


// Type for physical/virtual addresses

typedef uint32_t uintptr_t;


// Boolean type

typedef enum { false, true } bool;


// NULL pointer

#define NULL ((void*)0)


// Useful macros

#define ALIGN_UP(addr, align) (((addr) + (align) - 1) & ~((align) - 1))

#define ALIGN_DOWN(addr, align) ((addr) & ~((align) - 1))


// Kernel panic - halt the system

#define PANIC(msg) panic(__FILE__, __LINE__, msg)

void panic(const char* file, int line, const char* msg);


#endif // KERNEL_H
```

Create a simple `Makefile` to build the project:

```makefile
# Toolchain
CC = gcc
AS = nasm
LD = ld
OBJCOPY = objcopy

# Flags
CFLAGS = -m32 -ffreestanding -nostdlib -fno-builtin -fno-stack-protector -Wall -Wextra -Ikernel/include
ASFLAGS = -f elf32
LDFLAGS = -m elf_i386 -T linker.ld -nostdlib

# Output files
KERNEL = kernel.bin
ISO = os.iso

# Source directories
SRCDIR = kernel
ARCHDIR = $(SRCDIR)/arch/x86
DRVDIR = $(SRCDIR)/drivers
MMDIR = $(SRCDIR)/mm
PROCDIR = $(SRCDIR)/process
LIBDIR = $(SRCDIR)/lib

# Find source files
CSOURCES = $(shell find $(SRCDIR) -name "*.c" -not -path "*/include/*")
ASMSOURCES = $(shell find $(SRCDIR) -name "*.asm")
BOOTSOURCES = boot/multiboot.asm boot/boot.asm

# Object files
COBJS = $(CSOURCES:.c=.o)
ASMOBJS = $(ASMSOURCES:.asm=.o)
BOOTOBJS = $(BOOTSOURCES:.asm=.o)
OBJS = $(BOOTOBJS) $(ASMOBJS) $(COBJS)

# Build rules
all: $(ISO)

$(ISO): $(KERNEL)
        mkdir -p isodir/boot/grub
        cp $(KERNEL) isodir/boot/
        cp grub.cfg isodir/boot/grub/
        grub-mkrescue -o $(ISO) isodir

$(KERNEL): $(OBJS)
        $(LD) $(LDFLAGS) -o $@ $^

%.o: %.c
        $(CC) $(CFLAGS) -c $< -o $@

%.o: %.asm
        $(AS) $(ASMSOURCES) $< -o $@

clean:
        rm -f $(OBJS) $(KERNEL) $(ISO)
        rm -rf isodir

run: $(ISO)
        qemu-system-x86_64 -cdrom $(ISO) -serial stdio

debug: $(ISO)
        qemu-system-x86_64 -cdrom $(ISO) -serial stdio -s -S

.PHONY: all clean run debug
```

C. Infrastructure Starter Code:

Create a minimal linker script `linker.ld` to define kernel memory layout:

```ld
/* Kernel linker script for 32-bit protected mode */
ENTRY(_start)               /* Entry point symbol (defined in boot.asm) */

SECTIONS
{
    /* Load kernel at 1MB physical address (conventional for x86) */
    . = 1M;

    /* Multiboot header must be within first 8KB */
    .multiboot : {
        *(.multiboot)
    }

    /* Text section (code) */
    .text : ALIGN(4K) {
        *(.text)
    }

    /* Read-only data */
    .rodata : ALIGN(4K) {
        *(.rodata)
    }

    /* Read-write data (initialized) */
    .data : ALIGN(4K) {
        *(.data)
    }

    /* Read-write data (uninitialized) and stack */
    .bss : ALIGN(4K) {
        *(COMMON)
        *(.bss)
        /* Reserve space for kernel stack (16KB) */
        . = ALIGN(16);
        kernel_stack_start = .;
        . += 16K;
        kernel_stack_end = .;
    }

    /* End of kernel image */
    end = .;
}
```

Create a basic GRUB configuration file `grub.cfg` for booting:

```grub
menuentry "Build Your Own OS" {
    multiboot /boot/kernel.bin
    boot
}
```

D. Core Logic Skeleton Code:

Create the main kernel entry point `kernel/kernel.c` with TODOs for initialization:

```c
#include <kernel.h>

#include <arch/x86/gdt.h>

#include <arch/x86/idt.h>

#include <drivers/vga.h>

#include <drivers/keyboard.h>

#include <mm/pmm.h>

#include <mm/vmm.h>

#include <process/scheduler.h>


// Main kernel entry point - called from bootloader assembly

void kmain(uint32_t mboot_magic, void* mboot_info) {

    // TODO 1: Verify we were loaded by a Multiboot-compliant bootloader

    // Check if mboot_magic equals MULTIBOOT_MAGIC (0x2BADB002)

    // If not, panic with appropriate error message


    // TODO 2: Initialize basic debugging/output

    // Call vga_init() to clear screen and set up text mode

    // Call kprintf() to print welcome message


    // TODO 3: Set up CPU environment

    // Call gdt_init() to load Global Descriptor Table

    // Call idt_init() to set up Interrupt Descriptor Table


    // TODO 4: Initialize hardware interrupt controller

    // Call pic_remap() to remap PIC IRQs to vectors 32-47

    // Call keyboard_init() to set up PS/2 keyboard driver

    // Call timer_init() to set up Programmable Interval Timer


    // TODO 5: Initialize memory management

    // Call pmm_init(mboot_info) to read memory map and set up frame allocator

    // Call vmm_init() to create initial page tables and enable paging

    // Call heap_init() to set up kernel heap allocator


    // TODO 6: Initialize process management

    // Call scheduler_init() to set up idle process and ready queue

    // Create first user process (optional for later milestone)


    // TODO 7: Enable interrupts and start multitasking

    // Use inline assembly to execute 'sti' instruction

    // Call scheduler_start() to begin round-robin scheduling
```

```
    // TODO 8: Main kernel idle loop
    // If no processes are ready, halt the CPU (hlt instruction)
    // Otherwise, this should never return
}
```

E. Language-Specific Hints (C):

- Use `__attribute__((packed))` for hardware structures (GDT, IDT entries) to prevent compiler padding that would break alignment expectations of the CPU.
- For inline assembly, use `asm volatile` with memory clobbers to ensure the compiler doesn't reorder or optimize away critical operations like port I/O.
- Mark functions that should only be called from kernel mode with `__attribute__((no_caller_saved_registers))` to prevent unexpected register corruption.
- Use `static inline` for small, frequently-called helper functions in headers to avoid function call overhead.
- When accessing memory-mapped I/O regions, use `volatile` pointers to prevent compiler optimizations from removing or reordering accesses.

F. Milestone Checkpoint: After setting up the high-level architecture and file structure, you should be able to:

1. Run `make` successfully without errors, producing `os.iso`
2. Launch QEMU with `make run` and see a blank screen (or GRUB menu if using CD-ROM boot)
3. Add a simple test in `kmain()` that prints "Hello, kernel world!" to verify the build chain works
4. Check that all expected directories exist and are properly organized

Signs something is wrong:

- **Linker errors about undefined symbols**: Check that all function implementations exist and are being compiled (included in `CSOURCES`).
- **QEMU shows "Booting from CD-ROM" then nothing**: The kernel isn't being loaded or is crashing immediately. Check Multiboot header alignment and magic numbers.
- **Make reports "no rule to make target"**: Ensure file extensions match (.c vs .asm) and paths in Makefile are correct.

4. Data Model

> **Milestone(s):** This section defines the foundational data structures used across all four milestones, forming the concrete skeleton upon which the entire kernel implementation is built.

The data model represents the kernel's internal memory blueprint—the structures that organize bytes in memory into meaningful representations of hardware state, memory regions, and running processes. Unlike application programming where data structures primarily model business logic, kernel data structures must precisely mirror hardware specifications while enabling efficient software abstraction.

CPU and Hardware Structures

Think of the CPU as an **industrial machine with programmable control panels**. Each panel (a data structure) configures a specific aspect of the machine's operation: which instructions it can execute (GDT), how it responds to emergencies (IDT), and how it translates virtual addresses to physical ones (page tables). These structures aren't just software conveniences—they must match exact binary formats that the CPU hardware expects when specific control registers are loaded.

Global Descriptor Table (GDT) Structures

The GDT defines memory segments in **protected mode**, creating the fundamental isolation between kernel and (future) user code. Each entry describes a memory segment's base address, limit (size), and access permissions.

Decision: Packed GDT Entry Structure

- **Context**: The x86 CPU expects GDT entries in a specific 8-byte packed format. Compiler padding would break this layout.
- **Options Considered**:
  1. Individual bitfield operations with shift/mask
  2. Packed structure with compiler attributes
  3. Byte array with manual encoding functions
- **Decision**: Use a packed structure with compiler attributes (`__attribute__((packed))` in GCC/clang, `#pragma pack(1)` in MSVC)
- **Rationale**: Provides both type safety and correct memory layout. Direct field access is clearer than manual bit manipulation, while the packed attribute ensures no compiler padding.
- **Consequences**: Must be careful with alignment when loading the GDT register; some architectures may have performance penalties for unaligned accesses, but this occurs only during boot.

Option	Pros	Cons	Chosen?
Bitfield operations	Full control, portable	Verbose, error-prone	No
Packed structure	Clean syntax, type-safe	Compiler-specific attributes	**Yes**
Byte array	Completely portable	No type checking, manual encoding	No

GDT Entry Fields:

Field Name	Type	Description
`limit_low`	`uint16_t`	Lower 16 bits of the segment limit (size-1)
`base_low`	`uint16_t`	Lower 16 bits of the segment's base address
`base_mid`	`uint8_t`	Middle 8 bits of the base address
`access`	`uint8_t`	Access byte: Present (bit 7), DPL (bits 5-6), Type (bits 0-3)
`granularity`	`uint8_t`	Flags: Granularity (bit 7, 0=byte, 1=4KB), Size (bit 6, 0=16-bit, 1=32-bit)
`base_high`	`uint8_t`	High 8 bits of the base address

The GDT itself is simply an array of these 8-byte entries. A special 6-byte `gdtr` structure (limit: `uint16_t`, base: `uint32_t*`) is loaded via the `lgdt` instruction.

Interrupt Descriptor Table (IDT) Structures

The IDT is the **emergency contact list** for the CPU—each entry tells the processor exactly which function to call when a specific interrupt number occurs, along with privilege level requirements and gate type.

IDT Entry Fields:

Field Name	Type	Description
`offset_low`	`uint16_t`	Lower 16 bits of the interrupt handler's address
`selector`	`uint16_t`	Code segment selector from GDT (e.g., 0x08 for kernel code)
`zero`	`uint8_t`	Must be zero (reserved)
`type_attr`	`uint8_t`	Type attributes: Present (bit 7), DPL (bits 5-6), Gate Type (bits 0-3)
`offset_high`	`uint16_t`	Upper 16 bits of the interrupt handler's address

Similar to the GDT, a `gdtr`-like `idtr` structure (limit: `uint16_t`, base: `uint32_t*`) is loaded with `lidt`.

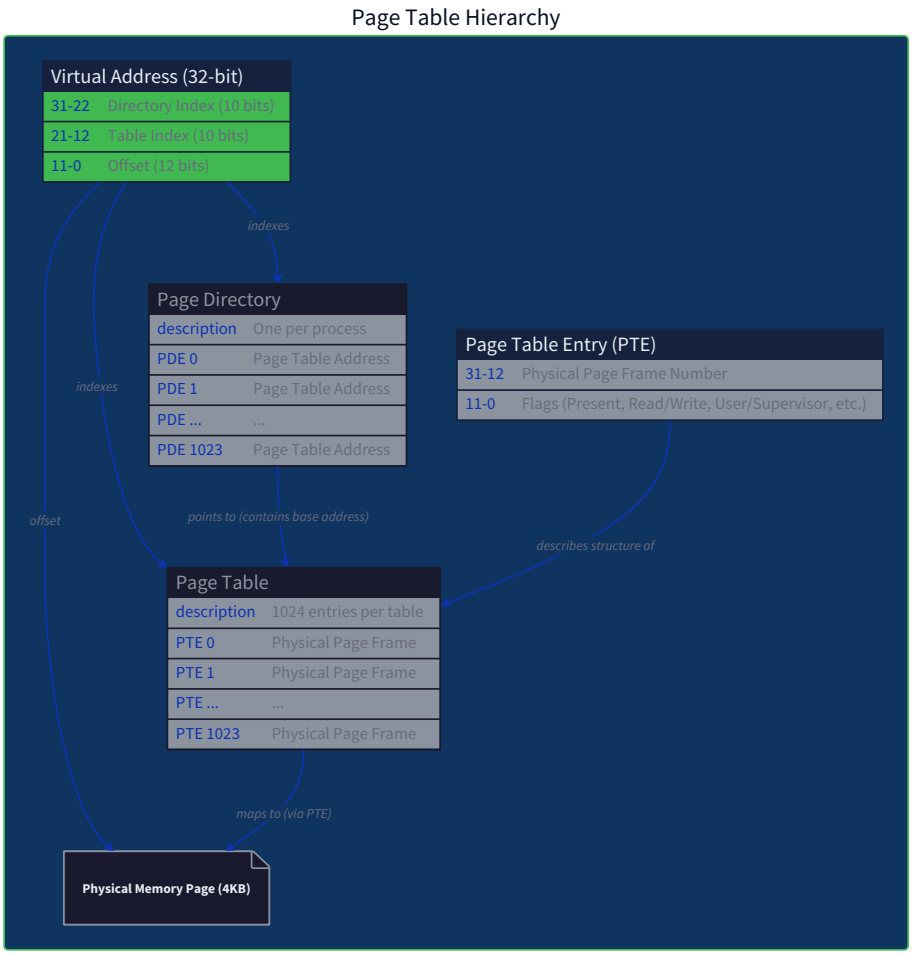
Decision: Interrupt Handler Function Signature

- **Context**: The CPU pushes specific registers onto the stack when an interrupt occurs, and the handler must clean up properly.
- **Options Considered**:
  1. Naked functions with inline assembly prologue/epilogue
  2. Regular functions with `__attribute__((interrupt))`
  3. Assembly stubs that call C functions
- **Decision**: Use assembly stubs that call C functions with a standardized `struct registers` pointer
- **Rationale**: Provides maximum control over stack handling and register saving while allowing handlers to be written in C. The assembly stub handles the CPU-mandated prologue and calls a C function with a clean context structure.
- **Consequences**: Requires maintaining assembly stubs for each interrupt vector, but these are highly regular and can be generated with macros.

Page Table Structures

Page tables form a **multi-level translation directory**, similar to a corporate phone book: the page directory (top-level) points to page tables (middle-level), which point to actual physical pages (employee offices). Each entry contains not just the physical address but also protection flags.

Page Table Structure Diagram



Page Table Entry (PTE) Fields:

Field Name	Type	Description
`present`	`uint32_t : 1`	1=Page is in physical memory, 0=Not present (triggers page fault)
`rw`	`uint32_t : 1`	0=Read-only, 1=Read-write
`us`	`uint32_t : 1`	0=Supervisor (kernel), 1=User
`pwt`	`uint32_t : 1`	Page Write-Through cache policy
`pcd`	`uint32_t : 1`	Page Cache Disable (1=uncacheable for MMIO)
`accessed`	`uint32_t : 1`	Set by CPU when page is accessed
`dirty`	`uint32_t : 1`	Set by CPU when page is written to
`pat`	`uint32_t : 1`	Page Attribute Table (advanced caching)
`global`	`uint32_t : 1`	Global page (TLB not flushed on CR3 write)
`available`	`uint32_t : 3`	Available for OS use (we store swapping info here)
`frame`	`uint32_t : 20`	Physical frame number (shifted right 12 bits)

Both page directory entries (PDEs) and page table entries (PTEs) share this format, though PDEs use the `frame` field to point to a page table instead of a data page.

> **Design Insight:** The `available` bits (9-11) are explicitly reserved for OS use. We can use these to track page allocation metadata without needing separate data structures—for example, marking pages as "kernel owned" vs "user owned" or storing swap disk locations.

Memory Management Structures

Think of memory management as a **library system**: the physical frame allocator is the librarian tracking which books (4KB frames) are checked out; the page tables are the catalog system mapping requested titles (virtual addresses) to actual book locations; the heap allocator is the inter-library loan service that can split and combine book chapters (smaller allocations) within a single book.

Physical Memory Manager (PMM) Structures

The PMM tracks which 4KB physical frames are free or allocated. It needs to be space-efficient (stored in memory it manages) and fast for allocation/deallocation.

Decision: Bitmap vs Stack Allocator

- **Context**: We need to track thousands of 4KB frames with minimal overhead and fast allocation.
- **Options Considered**:
  1. Bitmap: One bit per frame, simple but linear scan can be slow
  2. Stack/Free list: Push freed frames, pop for allocation, O(1) but needs storage outside managed memory
  3. Buddy allocator: Fast coalescing but more complex
- **Decision**: Bitmap allocator with optimization for finding contiguous frames
- **Rationale**: Simplicity and minimal external dependencies. The bitmap can be stored within the managed memory itself (we reserve space during initialization). While finding free frames requires scanning, we can optimize by tracking the last allocated index and scanning forward.
- **Consequences**: Allocation time grows with memory size, but for educational purposes and typical VM sizes (<1GB), this is acceptable. The bitmap consumes 1 bit per 4KB = 1/32768 of total memory.

Physical Frame Bitmap Structure:

Field Name	Type	Description
`bitmap`	`uint32_t*`	Pointer to array of bits (1=allocated, 0=free)
`total_frames`	`size_t`	Total number of 4KB frames in system
`free_frames`	`size_t`	Current count of free frames
`last_alloc_index`	`size_t`	Optimization: index where last search ended

Memory Map Entry (from Multiboot):

Field Name	Type	Description
`base_addr`	`uint64_t`	Starting physical address
`length`	`uint64_t`	Size of region in bytes
`type`	`uint32_t`	1=Available RAM, 2=Reserved, 3=ACPI reclaimable, 4=ACPI NVS
`reserved`	`uint32_t`	Must be 0

The PMM initializes by walking the Multiboot memory map, marking available regions as free, and reserving regions containing the kernel, multiboot info, and the bitmap itself.

Virtual Memory Manager (VMM) Structures

The VMM maintains the page directory and page tables that define the virtual-to-physical mapping for the entire system. Each process will have its own page directory, but the kernel portion is shared via recursive mapping.

Page Directory Structure:

Field Name	Type	Description
`entries`	`page_table_entry_t[1024]`	1024 page directory entries (PDEs)
`physical_addr`	`uint32_t`	Physical address of this directory (for loading into CR3)

Decision: Recursive Page Table Mapping

- **Context**: The kernel needs to modify page tables, but page tables themselves are mapped through... page tables. We need a way to access them.
- **Options Considered**:

1. Temporary mapping: Map page tables temporarily when needed, unmap after
2. Identity mapping: Map all physical memory 1:1 at high virtual addresses
3. Recursive mapping: Map the page directory into itself at a fixed slot

- **Decision**: Recursive page table mapping with the last PDE pointing to the directory itself
- **Rationale**: Simplifies page table manipulation—any page table entry can be accessed through predictable virtual addresses. The recursive mapping creates a consistent virtual address space for page table structures.
- **Consequences**: Consumes one PDE slot (1/1024 of kernel address space), but provides elegant access to all page tables.

With recursive mapping at PDE 1023:

- Page directory accessible at virtual address `0xFFFFF000`
- Page table `n` accessible at `0xFFC00000 + (n * 4096)`
- Page directory entry `m` accessible at `0xFFFFF000 + (m * 4)`

Kernel Heap Allocator Structures

The heap allocator manages dynamic memory allocation within the kernel's virtual address space. It uses freed memory to track itself through embedded linked list nodes.

Heap Block Header Structure:

Field Name	Type	Description
`size`	`size_t`	Size of this block (including header, excluding footer)
`free`	`bool`	1=Block is free, 0=Allocated
`magic`	`uint32_t`	Magic number `0xDEADBEEF` for integrity checking
`next`	`heap_block_header_t*`	Pointer to next block (only valid when free)
`prev`	`heap_block_header_t*`	Pointer to previous block (only valid when free)

Heap Footer Structure (for coalescing):

Field Name	Type	Description
`header`	`heap_block_header_t*`	Pointer back to block header
`magic`	`uint32_t`	Magic number `0xF00DF00D`

> **Design Insight:** The heap allocator uses boundary tags (headers and footers) to enable bidirectional coalescing of free blocks. While this increases overhead (8-16 bytes per allocation), it prevents fragmentation—a critical concern in kernel space where memory is limited and allocation patterns are unpredictable.

Heap Instance Structure:

Field Name	Type	Description
`start_addr`	`void*`	Starting virtual address of heap region
`end_addr`	`void*`	Current end of heap (grows via sbrk)
`max_addr`	`void*`	Maximum address heap can grow to
`free_list`	`heap_block_header_t*`	Head of linked list of free blocks
`lock`	`spinlock_t`	Spinlock for thread-safe operations

Process Management Structures

Think of process management as a **chess tournament organization**: each player has a scoresheet (PCB) tracking their current position and state; the tournament director (scheduler) maintains a pairing chart (ready queue) deciding who plays next; when switching players, the director carefully records the exact board position (register state) before letting the next player take their turn.

Process Control Block (PCB)

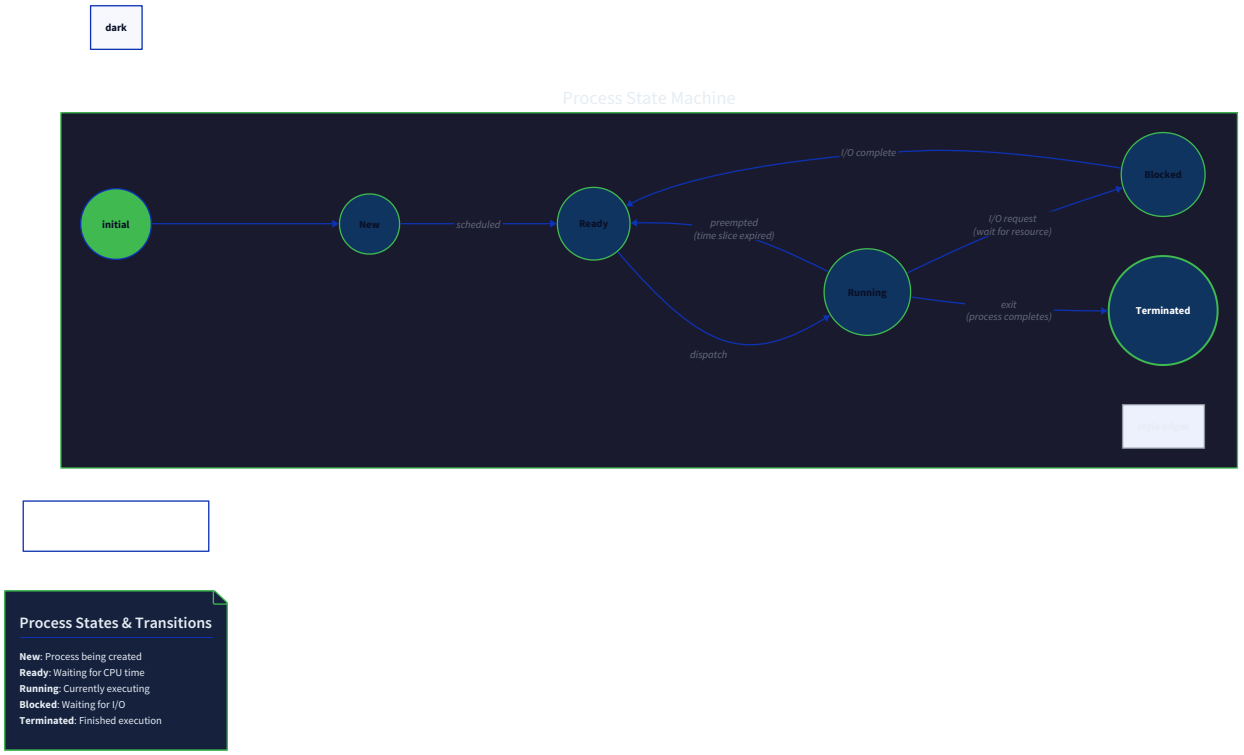
The PCB is the kernel's complete record of a process—everything needed to stop it, store it, and restart it later. It's the **process's medical chart**, containing diagnostic information, current vital signs (registers), and treatment history.

Process Control Block (PCB) Structure:

Field Name	Type	Description
`pid`	`pid_t` (int)	Process ID (unique identifier)
`parent_pid`	`pid_t`	PID of parent process (for process tree)
`state`	`process_state_t`	Current state: NEW, READY, RUNNING, BLOCKED, TERMINATED
`exit_code`	`int`	Exit code when terminated
`priority`	`uint8_t`	Scheduling priority (0=highest)
`time_used`	`uint64_t`	CPU time used so far (in ticks)
`registers`	`cpu_registers_t`	Saved CPU registers during context switch
`kernel_stack`	`void*`	Pointer to top of kernel stack
`kernel_stack_base`	`void*`	Base (bottom) of kernel stack
`page_directory`	`page_directory_t*`	Process's page directory (physical address)
`brk`	`void*`	Current program break (heap end)
`image_start`	`void*`	Start of process image in memory
`image_size`	`size_t`	Size of process image
`name`	`char[16]`	Process name (for debugging)
`waiting_for`	`resource_id_t`	Resource PID is waiting for (e.g., another PID)
`next`	`pcb_t*`	Next PCB in scheduler list

CPU Registers Structure (saved during context switch):

Field Name	Type	Description
`eax`, `ebx`, `ecx`, `edx`	`uint32_t`	General purpose registers
`esi`, `edi`	`uint32_t`	Source/Destination index registers
`ebp`	`uint32_t`	Base pointer (stack frame)
`esp`	`uint32_t`	Stack pointer (saved separately for kernel stack)
`eip`	`uint32_t`	Instruction pointer
`eflags`	`uint32_t`	Flags register
`cs`, `ds`, `es`, `fs`, `gs`, `ss`	`uint32_t`	Segment registers



Decision: Process State Model

- **Context**: We need to model the lifecycle of a process with clear transitions between states.
- **Options Considered**:
  1. Five-state model (New, Ready, Running, Blocked, Terminated)
  2. Seven-state model (add Suspend states for swapping)
  3. Three-state model (Ready, Running, Blocked) for simplicity
- **Decision**: Five-state model with explicit NEW and TERMINATED states
- **Rationale**: The five-state model clearly distinguishes between creation (NEW), admission to scheduler (READY), actual execution (RUNNING), waiting for I/O (BLOCKED), and cleanup (TERMINATED). This matches educational materials and provides clear hooks for initialization and cleanup logic.
- **Consequences**: Slightly more complex state transitions but clearer implementation.

Process State Transitions:

Current State	Event	Next State	Actions Taken
NEW	`scheduler_admit()`	READY	PCB added to ready queue, memory allocated
READY	`scheduler_dispatch()`	RUNNING	Registers loaded, page tables switched
RUNNING	Timer interrupt	READY	Registers saved, moved to ready queue end
RUNNING	I/O request	BLOCKED	Registers saved, moved to blocked list
RUNNING	Process exit	TERMINATED	Resources marked for reclamation
BLOCKED	I/O complete	READY	Moved from blocked to ready queue
TERMINATED	`scheduler_cleanup()`	(removed)	Memory freed, PCB removed from all lists

Scheduler Structures

The scheduler implements the **round-robin tournament pairing system**, maintaining queues of processes in each state and selecting the next one to run based on simple rules.

Ready Queue Structure (circular linked list):

Field Name	Type	Description
`head`	`pcb_t*`	Pointer to first ready process
`tail`	`pcb_t*`	Pointer to last ready process (for O(1) append)
`count`	`size_t`	Number of processes in ready state
`lock`	`spinlock_t`	Spinlock for thread-safe operations

Blocked List Structure (singly linked list by resource):

Field Name	Type	Description
`resource_id`	`resource_id_t`	What resource processes are waiting for
`head`	`pcb_t*`	First process waiting for this resource
`next_blocked_list`	`blocked_list_t*`	Next blocked list for different resource

Scheduler Instance Structure:

Field Name	Type	Description
`ready_queue`	`ready_queue_t`	Queue of processes ready to run
`blocked_lists`	`blocked_list_t*`	Array of lists for different resources
`current_process`	`pcb_t*`	Currently running process (NULL if idle)
`idle_process`	`pcb_t*`	Special idle process to run when none ready
`pid_counter`	`pid_t`	Counter for assigning new PIDs
`total_processes`	`size_t`	Total processes (all states)

Decision: Round-Robin vs Priority Scheduler

- **Context**: The educational kernel needs a simple, predictable scheduler that's easy to implement and debug.
- **Options Considered**:
  1. Round-robin: Equal time slices, simple FIFO queue
  2. Priority-based: Different priorities, more complex but realistic
  3. Multilevel feedback queue: Adaptive, prevents starvation but complex
- **Decision**: Simple round-robin with fixed time quantum
- **Rationale**: Implementation simplicity and deterministic behavior aid debugging. Round-robin ensures no starvation and provides clear, predictable scheduling that's easy to reason about. Priority scheduling can be added later as an extension.
- **Consequences**: All processes treated equally regardless of importance, but acceptable for educational purposes where we typically run only a few test processes.

System Call Structures

System calls follow a **restaurant ordering system**: the customer (user process) fills out an order form (system call number and arguments), rings a bell (`int 0x80`), and the kitchen staff (kernel) processes the order, returning the result.

System Call Table Entry:

Field Name	Type	Description
`handler`	`syscall_handler_t`	Function pointer to system call implementation
`num_args`	`uint8_t`	Number of arguments expected (0-6)
`requires_auth`	`bool`	Whether caller must be privileged

System Call Arguments Structure (as passed from user mode):

Field Name	Type	Description
`eax`	`uint32_t`	System call number (return value also here)
`ebx` , `ecx` , `edx` , `esi` , `edi` , `ebp`	`uint32_t`	Up to 6 arguments (following Linux convention)

Decision: System Call Convention

- **Context**: Need a standard way for user processes to pass arguments to system calls and receive results.
- **Options Considered**:
  1. Register-based (Linux style): Arguments in registers `ebx`, `ecx`, etc.
  2. Stack-based: Arguments pushed on user stack
  3. Memory-based: Pointer to argument block in register
- **Decision**: Register-based convention matching Linux i386 ABI
- **Rationale**: Simpler for assembly stub implementation, faster (no memory access for arguments), and follows common practice making it familiar to those with Linux experience.
- **Consequences**: Limited to 6 arguments (plus syscall number), but sufficient for educational system calls. Must carefully validate all arguments from user space.

Common Pitfalls: Data Structure Implementation

⚠️ Pitfall: Structure Padding Breaking Hardware Compatibility

- **Description**: Forgetting to use packed attributes on GDT/IDT structures, causing compiler padding that misaligns fields relative to CPU expectations.
- **Why it's wrong**: The CPU reads these structures as raw bytes. Extra padding bytes shift field positions, causing misinterpretation (e.g., wrong segment limits, handler addresses).
- **Fix**: Always use `__attribute__((packed))` or `#pragma pack(1)` for hardware-defined structures. Verify structure size with `sizeof()` equals expected byte count.

⚠️ Pitfall: Forgetting Magic Numbers in Heap Headers

- **Description**: Not including or checking magic numbers in heap block headers/footers.
- **Why it's wrong**: Heap corruption (buffer overflows, double frees) goes undetected, causing mysterious crashes later. The magic number acts as a canary that reveals when a block's metadata has been overwritten.
- **Fix**: Include distinct magic numbers in headers (`0xDEADBEEF`) and footers (`0xF00DF00D`). Check them on every allocation and free operation, calling `panic()` on mismatch.

⚠️ Pitfall: Not Invalidating TLB After Page Table Modification

- **Description**: Modifying page table entries but forgetting to flush the TLB, causing stale translations.
- **Why it's wrong**: The CPU caches translations in its TLB. Changes to page tables aren't visible until TLB entries are invalidated, leading to inconsistencies between intended and actual mappings.
- **Fix**: After modifying any page table entry, either:
  1. Use `invlpg` instruction for single page invalidation
  2. Reload CR3 (expensive but flushes entire TLB)
  3. Use global pages (G-bit) for kernel mappings that shouldn't be flushed

⚠️ Pitfall: PCB Stack Pointer Saved Incorrectly

- **Description**: Saving the wrong stack pointer value in the PCB during context switch.
- **Why it's wrong**: When restoring, the CPU resumes with wrong stack, causing immediate crash or corruption. Particularly tricky because we switch from process kernel stack to scheduler stack.
- **Fix**: In assembly context switch, save ESP *after* switching to a safe kernel stack. The sequence should be: 1) Save general registers, 2) Save current ESP to PCB, 3) Switch to scheduler stack, 4) Do scheduling logic, 5) Load new PCB's ESP, 6) Restore general registers, 7) Return.

Implementation Guidance

A. Technology Recommendations Table:

Component	Simple Option	Advanced Option
Hardware Structures	Packed structs with compiler attributes	Generated from specifications (e.g., from CPU manuals)
Memory Allocator	Bitmap PMM + boundary-tag heap	Buddy allocator + slab allocator
Process Management	Linked list queues	Balanced trees for priority queues

B. Recommended File/Module Structure:

```
build-your-own-os/
├── boot/                      # Bootloader code
│   ├── boot.asm               # Stage 1 bootloader
│   └── multiboot_header.asm   # Multiboot header
├── kernel/
│   ├── arch/
│   │   ├── x86/
│   │   │   ├── gdt.c          # GDT structures and initialization
│   │   │   ├── gdt.h          # GDT structure definitions
│   │   │   ├── idt.c          # IDT structures and initialization
│   │   │   ├── idt.h          # IDT structure definitions
│   │   │   ├── isr.asm        # ISR assembly stubs
│   │   │   └── paging.h       # Page table structure definitions
│   │   └── io.h               # I/O port operations
│   ├── mm/
│   │   ├── pmm.c              # Physical memory manager
│   │   ├── pmm.h              # PMM structures
│   │   ├── vmm.c              # Virtual memory manager
│   │   ├── vmm.h              # VMM structures
│   │   ├── heap.c             # Kernel heap allocator
│   │   └── heap.h             # Heap structures
│   ├── proc/
│   │   ├── process.c          # Process creation/destruction
│   │   ├── process.h          # PCB structure definition
│   │   ├── scheduler.c        # Scheduler implementation
│   │   ├── scheduler.h        # Scheduler structures
│   │   ├── switch.asm         # Context switch assembly
│   │   └── syscall.c          # System call dispatch
│   └── drivers/
│       ├── vga.c              # VGA text driver
│       ├── keyboard.c         # Keyboard driver
│       └── pic.c              # PIC controller
├── include/                    # Common headers
│   ├── types.h                # Standard types (uint32_t, etc.)
│   ├── common.h               # Common macros and constants
│   └── panic.h                # Panic function declaration
└── kernel.c                   # Main kernel entry point
```

C. Infrastructure Starter Code (Hardware Structures):

```c
/* kernel/arch/x86/gdt.h */
#ifndef KERNEL_ARCH_X86_GDT_H
#define KERNEL_ARCH_X86_GDT_H

#include <stdint.h>

/* GDT Entry Structure - MUST be packed */
struct gdt_entry {
    uint16_t limit_low;
    uint16_t base_low;
    uint8_t base_mid;
    uint8_t access;
    uint8_t granularity;
    uint8_t base_high;
} __attribute__((packed));

/* GDTR structure for lgdt instruction */
struct gdtr {
    uint16_t limit;
    uint32_t base;
} __attribute__((packed));

/* Segment selectors (index << 3) */
#define KERNEL_CS 0x08
#define KERNEL_DS 0x10

void gdt_init(void);
void gdt_load(void);

#endif /* KERNEL_ARCH_X86_GDT_H */
```

```c
/* kernel/arch/x86/idt.h */

#ifndef KERNEL_ARCH_X86_IDT_H

#define KERNEL_ARCH_X86_IDT_H


#include <stdint.h>


/* IDT Entry Structure - MUST be packed */

struct idt_entry {

    uint16_t offset_low;

    uint16_t selector;

    uint8_t zero;

    uint8_t type_attr;

    uint16_t offset_high;

} __attribute__((packed));


/* IDTR structure for lidt instruction */

struct idtr {

    uint16_t limit;

    uint32_t base;

} __attribute__((packed));


/* Type attribute bits */

#define IDT_PRESENT  (1 << 7)

#define IDT_DPL_0    (0 << 5)

#define IDT_DPL_3    (3 << 5)

#define IDT_GATE_32  (0xE)  /* 32-bit interrupt gate */


/* CPU Register state saved by ISR stub */

struct registers {

    uint32_t ds;

    uint32_t edi, esi, ebp, esp, ebx, edx, ecx, eax;

    uint32_t int_no, err_code;

    uint32_t eip, cs, eflags, user_esp, ss;

};


void idt_init(void);

void idt_set_gate(uint8_t num, uint32_t base, uint16_t sel, uint8_t flags);

void isr_handler(struct registers *regs);


#endif /* KERNEL_ARCH_X86_IDT_H */
```

D. Core Logic Skeleton Code:

```c
/* kernel/mm/pmm.h - Physical Memory Manager Structures */

#ifndef KERNEL_MM_PMM_H

#define KERNEL_MM_PMM_H


#include <stdint.h>

#include <stddef.h>

#include <stdbool.h>


/* Memory map entry from Multiboot */

struct multiboot_mmap_entry {

    uint32_t size;           /* Size of this entry (including this field) */

    uint64_t base_addr;

    uint64_t length;

    uint32_t type;

    uint32_t reserved;

} __attribute__((packed));


/* Physical Memory Manager state */

struct pmm_state {

    uint32_t *bitmap;        /* One bit per 4KB frame: 1=allocated, 0=free */

    size_t total_frames;     /* Total number of frames in system */

    size_t free_frames;      /* Current number of free frames */

    size_t last_alloc_index; /* Optimization: where we last searched */

    uint32_t bitmap_frames;  /* Number of frames the bitmap itself occupies */

};


/* TODO: In pmm.c, implement:

   1. pmm_init(struct multiboot_info *mboot_info) - Parse memory map, setup bitmap

   2. pmm_alloc_frame() - Find and mark free frame as allocated

   3. pmm_free_frame(uint32_t frame) - Mark frame as free

   4. pmm_get_free_count() - Return free_frames

   5. Helper: find_free_contiguous_frames(size_t count) - For multi-page allocations

*/


#endif /* KERNEL_MM_PMM_H */
```

```c
/* kernel/arch/x86/paging.h - Page Table Structures */

#ifndef KERNEL_ARCH_X86_PAGING_H

#define KERNEL_ARCH_X86_PAGING_H


#include <stdint.h>

#include <stdbool.h>


/* Page Table Entry (also works for Page Directory Entry) */

typedef uint32_t page_table_entry_t;


/* Macros for accessing PTE fields */

#define PTE_PRESENT    (1 << 0)

#define PTE_WRITABLE   (1 << 1)

#define PTE_USER       (1 << 2)

#define PTE_WRITETHROUGH (1 << 3)

#define PTE_CACHE_DISABLE (1 << 4)

#define PTE_ACCESSED   (1 << 5)

#define PTE_DIRTY      (1 << 6)

#define PTE_PAT        (1 << 7)  /* Page Attribute Table */

#define PTE_GLOBAL     (1 << 8)

#define PTE_FRAME_MASK 0xFFFFF000


/* Functions for manipulating PTEs */

static inline uint32_t pte_make_frame(uint32_t phys_addr) {

    return phys_addr & PTE_FRAME_MASK;

}


static inline uint32_t pte_get_frame(page_table_entry_t pte) {

    return pte & PTE_FRAME_MASK;

}


static inline bool pte_is_present(page_table_entry_t pte) {

    return pte & PTE_PRESENT;

}


/* TODO: In vmm.c, implement:

   1. vmm_init() - Create kernel page directory with recursive mapping

   2. vmm_map_page(uint32_t virt, uint32_t phys, uint32_t flags) - Map single page

   3. vmm_unmap_page(uint32_t virt) - Remove mapping

   4. vmm_get_physical(uint32_t virt) - Translate virtual to physical

   5. vmm_switch_directory(uint32_t phys_dir) - Load new page directory (for context switch)

*/


#endif /* KERNEL_ARCH_X86_PAGING_H */
```

```c
/* kernel/proc/process.h - Process Control Block */

#ifndef KERNEL_PROC_PROCESS_H

#define KERNEL_PROC_PROCESS_H


#include <stdint.h>

#include <stdbool.h>


/* Process states */

typedef enum {

    PROC_STATE_NEW,        /* Being created */

    PROC_STATE_READY,      /* Ready to run, in scheduler queue */

    PROC_STATE_RUNNING,    /* Currently executing on CPU */

    PROC_STATE_BLOCKED,    /* Waiting for I/O or event */

    PROC_STATE_TERMINATED  /* Finished, awaiting cleanup */

} process_state_t;


/* Saved CPU registers for context switch */

struct cpu_registers {

    uint32_t eax, ebx, ecx, edx;

    uint32_t esi, edi;

    uint32_t ebp;

    uint32_t esp;    /* Saved separately for kernel stack */

    uint32_t eip;

    uint32_t eflags;

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

};


/* Process Control Block */

struct pcb {

    int pid;                    /* Process ID */

    int parent_pid;             /* Parent PID */

    process_state_t state;      /* Current state */

    int exit_code;              /* Exit code when terminated */



    struct cpu_registers regs;  /* Saved registers */

    void *kernel_stack;         /* Top of kernel stack */

    void *kernel_stack_base;    /* Base of kernel stack */



    uint32_t *page_directory;   /* Physical address of page directory */

    void *brk;                  /* Program break (heap end) */

    void *image_start;          /* Start of process image */

    size_t image_size;          /* Size of process image */
```

```
    char name[16];              /* Process name */
    uint64_t time_used;         /* CPU time used (ticks) */
    uint8_t priority;           /* Scheduling priority */

    int waiting_for;            /* Resource PID is waiting for */
    struct pcb *next;           /* Next in scheduler list */
};

/* TODO: In process.c, implement:
   1. process_create(const char *name, void *entry_point) - Create new PCB
   2. process_destroy(int pid) - Clean up terminated process
   3. process_get_current() - Return pointer to currently running PCB
   4. process_block(int pid, int resource) - Move process to blocked state
   5. process_unblock(int pid) - Move from blocked to ready
*/

#endif /* KERNEL_PROC_PROCESS_H */
```

E. Language-Specific Hints:

- **Packed Structures**: In C, use `__attribute__((packed))` for GCC/clang or `#pragma pack(1)` for MSVC compatibility. Always verify with `static_assert(sizeof(struct) == expected_size, "Structure packing incorrect")` if your compiler supports it.
- **Bit Manipulation**: For page table entries and other bitfield-heavy structures, use helper functions or macros rather than direct bit manipulation in multiple places. This centralizes the logic and prevents errors.
- **Type Safety**: Use `typedef` for distinct types (e.g., `pid_t`, `page_table_entry_t`) even if they're just integers. This makes function signatures clearer and helps catch type mismatches.
- **Forward Declarations**: Use forward declarations in headers to break circular dependencies (e.g., `typedef struct pcb pcb_t;` before using `pcb_t*` in other structures).

F. Milestone Checkpoint:

After implementing the data structures for a milestone, verify them by:

1. **Size Verification**: Write a test that uses `sizeof()` to verify each hardware structure matches the expected size (8 bytes for GDT entry, etc.)
2. **Memory Layout**: For packed structures, write a function that prints the byte offset of each field to ensure no unintended padding.
3. **Smoke Test**: Create instances of each structure, populate them with test values, and verify the values can be read back correctly.

Example verification for GDT:

```c
/* In a test function */
struct gdt_entry entry = {0};
assert(sizeof(entry) == 8);  /* Must be exactly 8 bytes */
```

G. Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
Triple fault immediately after loading GDT/IDT	Structure padding misaligning fields	Print raw bytes of structure, compare to expected layout	Add packed attribute, verify sizeof()
Heap corruption causing random crashes	Missing magic numbers or boundary tags	Add heap integrity check that walks all blocks verifying magic numbers	Include and check magic numbers in all allocations
Page faults when accessing valid memory	TLB not flushed after page table update	Add logging before/after TLB flush operations	Call `invlpg` or reload CR3 after modifying PTEs
Context switch loses register values	Incorrect save/restore order in assembly	Single-step through context switch in debugger, check saved values match	Follow precise save/restore sequence in switch.asm

5.1 Component: Bootloader & Kernel Entry (Milestone 1)

> **Milestone(s):** Milestone 1: Bootloader & Kernel Entry

The bootloader and kernel entry code perform the most fundamental act in an operating system's life: **bootstrapping**. This is the process by which a collection of bytes on a storage device is transformed into a running, protected-mode kernel with control over the hardware. It's a precise, unforgiving sequence where a single misstep—an incorrect address, a missing CPU flag—results in total system failure (a hang, reboot, or "triple fault"). This component's design ensures the processor transitions from its primitive, 16-bit real-mode state into the modern, protected 32-bit environment where our kernel's `main` function can safely execute.

Mental Model: The Stage Manager

Imagine preparing for a grand theatrical performance. The theater (the computer) is dark and empty when the audience (the user) arrives. Before the main show (the kernel) can begin, a **Stage Manager** (the bootloader) must work behind the scenes:

1. **Reads the Script:** The Stage Manager consults the playbill (the disk's boot sector or Multiboot header) to learn what set pieces (kernel code) and props (data) are needed and where to find them backstage (on disk).
2. **Sets the Stage:** They move all the necessary set pieces from storage into their exact, pre-determined positions on the stage (loading the kernel binary into memory at the correct physical address).
3. **Configures the Theater:** They adjust the theater's lighting rig (the Global Descriptor Table - GDT) to create distinct, isolated areas for the actors (code) and stagehands (data). They lock the doors to backstage (disable interrupts) to prevent interruptions during the final preparations.
4. **Transforms the Space:** With a cue, they fundamentally change the nature of the theater itself, switching from a simple, open floor plan (16-bit real mode) to a complex, multi-level stage with safety railings (32-bit protected mode). This transformation allows for sophisticated special effects (virtual memory, process isolation) later in the show.
5. **Hands Over Control:** Finally, the Stage Manager points a spotlight at the lead actor's marked position on the stage (the kernel's entry point), gives a silent count, and exits, leaving the actors (the kernel code) to begin their performance.

This mental model emphasizes the bootloader's role as a **transient, preparatory agent**. Its job is not to run the OS, but to meticulously arrange all the conditions necessary for the kernel to start running *safely* in a more capable CPU mode. Once its work is done, it is overwritten and never used again.

ADRs: BIOS vs. UEFI, 16-bit vs. 32-bit Boot

Several foundational choices determine how the kernel is loaded and initialized. These decisions are recorded as Architecture Decision Records (ADRs).

Decision: Use BIOS + Multiboot over UEFI + PE for Bootloading

- **Context:** The CPU starts in 16-bit real mode after power-on, with only the Basic Input/Output System (BIOS) or Unified Extensible Firmware Interface (UEFI) firmware available to load our initial code from disk. We need a reliable, standardized method to get our kernel image loaded and to pass it important information (like a memory map) without hardcoding hardware-specific assumptions.
- **Options Considered:**
  1. **Write a custom BIOS bootloader** (handling disk I/O via BIOS interrupts, parsing filesystems).
  2. **Use the Multiboot specification** (rely on a compliant bootloader like GRUB to load the kernel).
  3. **Implement UEFI booting** (create a UEFI application in PE format).
- **Decision:** Use the **Multiboot 1 specification** with a pre-existing bootloader (GRUB).
- **Rationale:**
  - **Focus:** Writing a production-quality bootloader from scratch is a massive project involving disk drivers, filesystem parsers, and complex memory management. Using Multiboot offloads this complexity, allowing us to focus our educational effort on the kernel itself.
  - **Standardization:** Multiboot provides a well-defined contract. The bootloader places the CPU in a known 32-bit protected mode state, loads kernel sections to defined addresses, and passes a `multiboot_info` structure pointer containing vital data (memory map, command line, module locations).
  - **Tooling & Debugging:** GRUB is robust, widely available in emulators (QEMU) and on real hardware, and simplifies kernel loading. The Multiboot magic number (`0x2BADB002`) provides a reliable check that we were loaded correctly.
- **Consequences:**
  - **Enables:** Rapid development; access to a rich memory map without probing hardware; support for loading additional modules (e.g., an initial ramdisk).
  - **Trade-offs:** Introduces a dependency on an external bootloader; the kernel must be linked at a high memory address (e.g., `0xC0000000` + 1MB) if we want to enable paging with higher-half kernel addressing later. The initial CPU state (GDT, IDT) is undefined, so the kernel must set up its own immediately.

Option	Pros	Cons	Chosen?
Custom BIOS Bootloader	Full control, minimal external dependencies, deep learning of low-level disk/hardware.	Extremely time-consuming, error-prone, duplicates significant non-kernel work.	❌
Multiboot (GRUB)	Standardized, simple kernel entry point, provides memory map and other info, widely supported.	External dependency, imposes specific linking model, less control over very early boot.	✅
UEFI Boot	Modern, more powerful services, supports larger disks, secure boot potential.	More complex specification, different development model (PE executables, protocols).	❌

Decision: Target 32-bit Protected Mode, Not 64-bit Long Mode

- **Context:** After being loaded, we must decide the primary operating mode for our CPU. Modern x86 CPUs support 64-bit "Long Mode," but starting there is not directly possible from the 16-bit real mode the BIOS leaves us in.
- **Options Considered:**
  1. **Boot directly to 32-bit Protected Mode** (via Multiboot or our own bootloader).
  2. **Implement a chain: 16-bit Real Mode → 32-bit Protected Mode → 64-bit Long Mode**.
- **Decision:** Target **32-bit Protected Mode** as our kernel's primary execution environment.
- **Rationale:**
  - **Simplicity & Pedagogy:** 32-bit paging (with a two-level Page Directory and Page Table) is significantly simpler to understand and implement than the four-level (or five-level) paging required for 64-bit mode. The core concepts of segmentation (via GDT) and paging are more cleanly separated in 32-bit mode.
  - **Sufficiency:** 32-bit mode provides all the core mechanisms we need to teach: protected memory, virtual address spaces, privilege levels (rings), and hardware task switching. The 4GB address space is ample for an educational kernel.
  - **Ecosystem:** Most educational OS resources and tutorials focus on the 32-bit to 64-bit transition, making 32-bit a better starting point. The Multiboot standard specifically expects a 32-bit kernel.
- **Consequences:**
  - **Enables:** Straightforward GDT setup, simpler page table structures, compatibility with a vast body of educational material.
  - **Trade-offs:** Cannot natively access more than 4GB of RAM or use 64-bit CPU features. A future extension to 64-bit would require a significant rewrite of low-level assembly and memory management code.

Option	Pros	Cons	Chosen?
32-bit Protected Mode	Simpler paging, well-documented, aligns with Multiboot, sufficient for core concepts.	4GB address limit, cannot use 64-bit instructions or registers.	✅
64-bit Long Mode	Modern, access to more RAM and registers, more efficient ABI.	Complex paging, requires intermediate 32-bit setup anyway, less beginner-friendly.	❌

Common Pitfalls: Bootloader & GDT

This stage is riddled with subtle traps that cause opaque failures. Understanding these *before* writing code is crucial.

⚠️ Pitfall: Incorrect Load Address / Linker Script

- **Description:** The kernel's `.text` (code), `.data` (initialized data), and `.bss` (zero-initialized data) sections must be placed at specific physical addresses in memory. If the bootloader loads the kernel to the wrong place, or the linker script tells the code it's located somewhere it's not, the CPU will fetch garbage instructions and crash.
- **Why it's wrong:** The addresses in the binary (for function calls, data references) are absolute. If the code is not at the expected location, these addresses point to random memory.
- **How to fix:**
  1. **Use a correct linker script.** This file (often `linker.ld`) dictates the memory layout. For a simple higher-half kernel loaded at 1MB physical, the `.text` section should start at virtual address `0xC0100000` (3GB + 1MB) but have a load address (`AT`) of `0x00100000` (1MB physical).
  2. **Verify with `objdump`.** Use `objdump -h kernel.bin` to inspect section addresses and ensure the VMA (Virtual Memory Address) and LMA (Load Memory Address) match your design.
  3. **Pass correct addresses to GRUB.** The Multiboot header's `load_addr` and `load_end_addr` fields must accurately reflect your kernel's load segment.

⚠️ Pitfall: GDT Misalignment and Incorrect Entries

- **Description:** The Global Descriptor Table (GDT) is an array of 8-byte entries describing memory segments. The CPU's GDTR register must point to the *physical address* of this table. Each entry has a complex bit-packed structure for base, limit, and access flags (present, privilege level, type). A single mis-set bit can prevent the CPU from transitioning to protected mode or cause immediate crashes upon loading segment registers.
- **Why it's wrong:** The CPU uses the GDT to translate logical addresses and enforce protection. A bad GDT can make all memory inaccessible or violate CPU expectations, leading to a general protection fault or triple fault.
- **How to fix:**
  1. **Use packed structures.** Define `gdt_entry` and `gdtr` with `__attribute__((packed))` (GCC) to prevent compiler alignment from creating gaps the CPU doesn't expect.
  2. **Create helper functions.** Write a `gdt_set_gate` function that takes human-readable parameters (base, limit, access, flags) and correctly packs them into the `gdt_entry` structure.
  3. **Start with a minimal GDT.** Implement just three entries: a NULL descriptor (index 0, required), a code segment descriptor (index 1, `KERNEL_CS`), and a data segment descriptor (index 2, `KERNEL_DS`). Both kernel segments should have base=0, limit=0xFFFFF (covering 4GB), and appropriate access bytes (code: execute/read, data: read/write).
  4. **Load the GDTR carefully.** Use an inline assembly statement to execute the `lgdt` instruction with the address of your `gdtr` structure.

⚠️ Pitfall: Forgetting to Disable/Re-enable Interrupts

- **Description:** During the critical transition from real mode to protected mode, and while setting up the GDT and IDT, hardware interrupts (from the timer, keyboard) must be disabled. If an interrupt occurs while the CPU is in an inconsistent state (e.g., segment registers contain real-mode values but the CPU is in protected mode), it will look up the interrupt vector in the real-mode interrupt vector table (IVT) at the wrong memory location, leading to a crash.
- **Why it's wrong:** The CPU's mechanism for handling interrupts changes completely between real mode and protected mode. Leaving interrupts enabled during the transition guarantees unpredictable behavior.
- **How to fix:**
  1. **Disable interrupts early.** Use `cli` (clear interrupt flag) in assembly before any mode-changing operations.
  2. **Only re-enable after IDT is ready.** Interrupts should remain disabled until *after* the Interrupt Descriptor Table (IDT) is fully populated and loaded with the `lidt` instruction. In our milestone sequence, this happens in Milestone 2. The `kmain` function should not enable interrupts until `idt_init()` and `pic_remap()` are complete.
  3. **Use `sti` carefully.** The instruction to set interrupts (`sti`) should be called from C only when the full interrupt-handling infrastructure is in place.

Implementation Guidance

This section provides concrete, compilable code to establish the bootstrapping foundation. The starter code handles the boilerplate, while the TODOs guide you through the core logic of milestone 1.

A. Technology Recommendations Table

Component	Simple Option (Recommended)	Advanced Option
Boot Protocol	Multiboot 1 with GRUB	Custom BIOS bootloader or UEFI
Assembly	Inline Assembly in C (`asm volatile`)	Pure NASM/GAS assembly files
Linker	Custom linker script (`linker.ld`)	Default linking with precise section attributes
Early Output	Writing directly to VGA text buffer (`0xB8000`)	Serial port debugging (`COM1`)

B. Recommended File/Module Structure

Organize the boot and early kernel code as follows. This separates concerns and makes the build process clear.

```
project-root/
├── Makefile                    # Build rules, links kernel, creates ISO
├── boot/                       # Bootloader-related files (if writing custom one)
│   └── boot.asm
├── kernel/                     # Main kernel source
│   ├── arch/i386/              # Architecture-specific 32-bit x86 code
│   │   ├── boot.asm            # *NEW*: Early assembly entry point (multiboot header, entry)
│   │   ├── gdt.c               # GDT initialization and structures
│   │   ├── gdt.asm             # *NEW*: GDT loading assembly (lgdt)
│   │   └── linker.ld           # Linker script for kernel binary
│   ├── drivers/                # Hardware drivers
│   │   └── vga.c               # VGA text mode driver (Milestone 1)
│   ├── include/                # Kernel headers
│   │   ├── kernel/             # Public kernel API
│   │   │   └── vga.h
│   │   └── arch/               # Private arch headers
│   │       ├── multiboot.h     # Multiboot structures
│   │       ├── gdt.h           # GDT structures and constants
│   │       └── io.h            `arch_port_write_byte`, etc.
│   ├── kernel.c                # Main kernel entry point (`kmain`)
│   └── lib/                    # Kernel libraries (string, memory functions)
│       └── mem.c               `memset`, `memcpy` for clearing BSS
└── build/                      # Build output directory (created by Makefile)
```

C. Infrastructure Starter Code

The following code provides complete, working implementations for the foundational pieces you need before writing the core boot logic. Copy these verbatim.

File: `kernel/arch/i386/multiboot.h`

```c
#ifndef ARCH_MULTIBOOT_H

#define ARCH_MULTIBOOT_H


#include <stdint.h>


/* The Multiboot header structure. Must be in the first 8KB of the kernel. */

struct multiboot_header {

    uint32_t magic;

    uint32_t flags;

    uint32_t checksum;

    uint32_t header_addr;

    uint32_t load_addr;

    uint32_t load_end_addr;

    uint32_t bss_end_addr;

    uint32_t entry_addr;

};


/* The symbol name for the Multiboot header in the linker script. */

extern struct multiboot_header mb_header;


/* The Multiboot information structure passed by the bootloader. */

struct multiboot_info {

    uint32_t flags;

    uint32_t mem_lower;

    uint32_t mem_upper;

    uint32_t boot_device;

    uint32_t cmdline;

    uint32_t mods_count;

    uint32_t mods_addr;

    uint32_t syms[4];

    uint32_t mmap_length;

    uint32_t mmap_addr;

    uint32_t drives_length;

    uint32_t drives_addr;

    uint32_t config_table;

    uint32_t boot_loader_name;

    uint32_t apm_table;

    uint32_t vbe_control_info;

    uint32_t vbe_mode_info;

    uint16_t vbe_mode;

    uint16_t vbe_interface_seg;

    uint16_t vbe_interface_off;
```

```
    uint16_t vbe_interface_len;
};

/* Memory map entry structure. */
struct multiboot_mmap_entry {
    uint32_t size;
    uint64_t base_addr;
    uint64_t length;
    uint32_t type;
    uint32_t reserved;
};

#define MULTIBOOT_MAGIC 0x2BADB002
#define MULTIBOOT_MEMORY_AVAILABLE 1

#endif // ARCH_MULTIBOOT_H
```

File: `kernel/drivers/vga.c`

```c
#include <stdint.h>

#include <stddef.h>

#include <string.h>

#include <kernel/vga.h>


#define VGA_BUFFER_ADDR 0xB8000


static uint16_t* vga_buffer = (uint16_t*) VGA_BUFFER_ADDR;

static size_t vga_row = 0;

static size_t vga_column = 0;

static uint8_t vga_color = 0x0F; // White on black


void vga_init(void) {

    // Clear the screen

    for (size_t y = 0; y < VGA_HEIGHT; y++) {

        for (size_t x = 0; x < VGA_WIDTH; x++) {

            const size_t index = y * VGA_WIDTH + x;

            vga_buffer[index] = vga_entry(' ', vga_color);

        }

    }

    vga_row = 0;

    vga_column = 0;

}


void vga_setcolor(uint8_t color) {

    vga_color = color;

}


void vga_putentryat(unsigned char c, uint8_t color, size_t x, size_t y) {

    const size_t index = y * VGA_WIDTH + x;

    vga_buffer[index] = vga_entry(c, color);

}


void vga_putchar(char c) {

    if (c == '\n') {

        vga_column = 0;

        vga_row++;

    } else if (c == '\r') {

        vga_column = 0;

    } else {

        vga_putentryat(c, vga_color, vga_column, vga_row);

        if (++vga_column == VGA_WIDTH) {

            vga_column = 0;
```

```
            vga_row++;
        }
    }
    // Scroll if needed
    if (vga_row >= VGA_HEIGHT) {
        // Move all lines up one
        for (size_t y = 1; y < VGA_HEIGHT; y++) {
            memcpy(&vga_buffer[(y-1) * VGA_WIDTH],
                   &vga_buffer[y * VGA_WIDTH],
                   VGA_WIDTH * sizeof(uint16_t));
        }
        // Clear the last line
        for (size_t x = 0; x < VGA_WIDTH; x++) {
            vga_buffer[(VGA_HEIGHT-1) * VGA_WIDTH + x] = vga_entry(' ', vga_color);
        }
        vga_row = VGA_HEIGHT - 1;
    }
}

void vga_write(const char* data, size_t size) {
    for (size_t i = 0; i < size; i++) {
        vga_putchar(data[i]);
    }
}
```

D. Core Logic Skeleton Code

Now, implement the core boot sequence by filling in the TODOs in these skeleton files.

File: `kernel/arch/i386/boot.asm` (Assembly Entry Point)

```assembly
; Declare constants for the Multiboot header
MB_ALIGN   equ 1 << 0            ; align loaded modules on page boundaries
MB_MEMINFO equ 1 << 1            ; provide memory map
MB_FLAGS   equ MB_ALIGN | MB_MEMINFO
MB_MAGIC   equ 0x1BADB002
MB_CHECKSUM equ -(MB_MAGIC + MB_FLAGS)

; Declare a multiboot header that marks the program as a kernel.
section .multiboot
align 4
    dd MB_MAGIC
    dd MB_FLAGS
    dd MB_CHECKSUM

; The initial stack grows downwards. Reserve 16KB for it.
section .bootstrap_stack
align 4
stack_bottom:
    times 16384 db 0
stack_top:

; The linker script specifies _start as the entry point to the kernel.
section .text
global _start
_start:
    ; TODO 1: Set up the stack pointer.
    ;         Load the address of `stack_top` into the ESP register.
    ;         Hint: The stack grows downwards, so the top is the highest address.

    ; TODO 2: Push the Multiboot information pointer to the stack.
    ;         The bootloader passes this in EBX. Push it as an argument for kmain.
    ;         Hint: Push EBX before pushing the magic number.

    ; TODO 3: Push the Multiboot magic number to the stack.
    ;         The bootloader passes this in EAX. Push it as the first argument for kmain.

    ; TODO 4: Transition to protected mode.
    ;         a. Disable interrupts using the `cli` instruction.
    ;         b. Load the Global Descriptor Table (GDT) using `lgdt [gdtr]`.
    ;         c. Set the Protection Enable (PE) bit in the CR0 register.
    ;         d. Execute a far jump to flush the CPU pipeline (e.g., `jmp 0x08:.flush`).
    ;         Label the flush section appropriately.

    ; TODO 5: Reload segment registers.
    ;         After entering protected mode, load the data segment selector (0x10) into
    ;         DS, ES, FS, GS, and SS registers.

    ; TODO 6: Call the kernel's main C function, `kmain`.
    ;         The arguments (magic, info) are already on the stack in the correct order.

    ; TODO 7: If kmain returns (it shouldn't), disable interrupts and halt.
    ;         Use `cli` and `hlt` in a loop.

; GDT structure definition and loading
global gdt_flush
extern gp
gdt_flush:
    lgdt [gp]
    mov ax, 0x10      ; 0x10 is the offset in the GDT to our data segment
    mov ds, ax
    mov es, ax
    mov fs, ax
    mov gs, ax
    mov ss, ax
    jmp 0x08:.flush   ; 0x08 is the offset to our code segment
.flush:
    ret

section .data
; TODO 8: Define the GDT pointer (`gdtr`) and the GDT entries.
;         Create at minimum: a null descriptor (0x00), a code segment (0x08), and a data segment (0x10).
;         Use the `gdt_entry` structure pattern from `gdt.h`.
;         The GDT pointer (`gdtr`) must contain the limit (size-1) and base address of the table.
```

File: `kernel/arch/i386/gdt.c`

```c
#include <arch/gdt.h>

// Define the GDT entries and the GDT pointer
struct gdt_entry gdt[3];
struct gdtr gp;

// Helper function to set a GDT entry
void gdt_set_gate(int num, uint32_t base, uint32_t limit, uint8_t access, uint8_t gran) {
    // TODO 1: Set the base address fields.
    //         Split the 32-bit base into low (16 bits), mid (8 bits), and high (8 bits) parts.

    // TODO 2: Set the limit fields.
    //         Split the 20-bit limit into low (16 bits) and high (4 bits) parts.
    //         The high 4 bits of limit go into the granularity byte.

    // TODO 3: Set the access byte.
    //         This includes Present bit, Descriptor Privilege Level (DPL 0), and Type flags.

    // TODO 4: Set the granularity byte.
    //         This includes the Granularity bit (1 for 4KB pages), Size bit (1 for 32-bit), and limit high bits.
}

void gdt_init(void) {
    // TODO 5: Set up the GDT pointer.
    //         Calculate the limit (size of GDT - 1) and set the base address.

    // TODO 6: Set up the three required GDT entries:
    //         - Null descriptor (index 0)
    //         - Code segment descriptor (index 1, selector 0x08)
    //         - Data segment descriptor (index 2, selector 0x10)
    //         For both segments: base = 0, limit = 0xFFFFF, granularity = 0xCF (4KB, 32-bit)
    //         Access: Code = 0x9A (Present, DPL 0, Execute/Read), Data = 0x92 (Present, DPL 0, Read/Write)

    // TODO 7: Call the assembly function `gdt_flush` to load the new GDT.
}
```

File: `kernel/kernel.c` (Main Kernel Entry Point)

```c
#include <kernel/vga.h>

#include <arch/multiboot.h>


// Forward declaration of the Multiboot header from the linker script

extern struct multiboot_header mb_header;


// The main kernel entry point

void kmain(uint32_t mboot_magic, void* mboot_info) {

    // TODO 1: Initialize the VGA text mode driver.

    //         Call `vga_init()` to clear the screen.


    // TODO 2: Verify the Multiboot magic number.

    //         Compare `mboot_magic` against `MULTIBOOT_MAGIC` (0x2BADB002).

    //         If it doesn't match, print an error message (e.g., "Invalid magic!") and halt.


    // TODO 3: Clear the kernel's .bss section.

    //         The Multiboot header provides `bss_end_addr`. Use a simple `memset` to zero

    //         from `&_bss_start` to `&_bss_end`. You'll need to define these symbols in your linker script.


    // TODO 4: Initialize the Global Descriptor Table.

    //         Call `gdt_init()` to set up and load the GDT.


    // TODO 5: Print a success message.

    //         Use `vga_write("Kernel booted successfully!\n", ...)` to confirm basic functionality.


    // TODO 6: For Milestone 1, halt here.

    //         Later milestones will add interrupt setup, memory management, etc.

    //         Use `for (;;) __asm__ volatile ("hlt");` to halt the CPU.

}
```

E. Language-Specific Hints (C/Assembly)

- **Inline Assembly:** Use `__asm__ volatile` with extended asm syntax for precise control. For example, to output to a port: `__asm__ volatile ("outb %0, %1" : : "a"(value), "Nd"(port));`.
- **Linker Script Variables:** In C, you can declare external symbols for addresses defined in the linker script: `extern uint32_t _bss_start;`. The linker will resolve these.
- **Packed Structures:** Always use `__attribute__((packed))` for hardware structures like `gdt_entry` and `multiboot_header` to prevent the compiler from inserting padding bytes.
- **Volatile for Hardware Memory:** Pointers to memory-mapped hardware (like the VGA buffer at `0xB8000`) should be declared as `volatile uint16_t*` to prevent the compiler from optimizing away reads/writes.

F. Milestone Checkpoint

After implementing the code above and building with your Makefile (which should produce a `kernel.bin` and a bootable ISO), you can test in QEMU:

1. **Expected Behavior:**
    - Run `qemu-system-i386 -cdrom os.iso -serial stdio`.
    - The screen should clear to black.
    - The message **"Kernel booted successfully!"** should appear in white text at the top-left corner.
    - The system should then halt (no further output, but the emulator window remains open).
2. **Verification Steps:**
    - **If the screen is blank:** Check that `vga_init()` is called, that the VGA buffer address is correct, and that `vga_write` is being called.

- **If you see corrupt characters or the system reboots:** The most likely cause is an incorrect GDT or a mistake in the protected mode transition in `boot.asm`. Double-check the GDT entries, the `lgdt` instruction, and the far jump.
- **If QEMU reports "No bootable device":** Your ISO is not bootable. Verify that the Multiboot header is in the first 8KB of the kernel binary and that GRUB is correctly installed to the ISO.

3. **Debugging Command:** Add `-d cpu_reset -no-reboot` to QEMU flags. If a triple fault occurs, QEMU will dump CPU state instead of rebooting, which can provide crucial clues about the faulting instruction.

5.2 Component: Interrupts & Keyboard Driver (Milestone 2)

> **Milestone(s):** Milestone 2: Interrupts & Keyboard

This component transforms the kernel from a passive program into a reactive system that can respond to external events in real time. It establishes the foundational infrastructure for handling both exceptional conditions (CPU exceptions) and asynchronous hardware events (IRQs), with the keyboard driver serving as the first concrete example of device interaction.

Mental Model: The Receptionist and Intercom

Imagine the kernel as the administrative office of a large building. Most of the time, office workers (kernel code) are busy at their desks performing scheduled tasks. However, the office needs to handle urgent, unpredictable events:

1. **The Receptionist (Interrupt Descriptor Table - IDT)**: A well-organized directory at the front desk lists every type of emergency phone call the office might receive. Each entry in this directory specifies exactly which specialist (Interrupt Service Routine - ISR) should handle that call and under what security protocols. When a call comes in, the receptionist immediately looks up the number in this directory to route it correctly.

2. **Urgent Phone Lines (Interrupt Requests - IRQs)**: Hardware devices are like external entities with dedicated "hotlines" into the office. When a key is pressed on the keyboard (IRQ1), the phone rings with a specific ringtone (interrupt vector number). The receptionist answers, consults the directory, and patches the call through to the keyboard specialist.

3. **The Intercom System (Programmable Interrupt Controller - PIC)**: Since there are many phone lines (16 IRQs) but the building's main phone switchboard (CPU) only has a single emergency input, the office uses an intercom system (PIC) to manage all these lines. The PIC consolidates multiple lines, prioritizes them (IRQ0 highest), and forwards the highest-priority active call to the main switchboard. It also ensures that once a call is handled, the line is cleared (End of Interrupt - EOI) so future calls can get through.

4. **Specialists on Call (Interrupt Service Routines - ISRs)**: Each specialist is a highly trained expert who:
   - **Drops everything immediately** when their phone rings (the CPU automatically saves the instruction pointer and flags).
   - **Takes notes** about the exact state of their work (saves CPU registers) before handling the emergency.
   - **Resolves the issue quickly** (minimal processing in interrupt context).
   - **Restores their desk exactly** (restores registers) before returning to their interrupted work.
   - **Always informs the intercom system** that they're done (sends EOI) so other calls can proceed.

This mental model emphasizes the **indirection** (IDT maps vectors to handlers), **prioritization** (PIC manages IRQ priorities), **atomicity** (ISRs save/restore state), and **asynchronicity** (interrupts can occur at any time) that characterize interrupt handling.

ADRs: Interrupt Controller and Stack Management

Decision: Programmable Interrupt Controller (PIC) over Advanced Programmable Interrupt Controller (APIC)

> **Decision: Use legacy 8259 PIC for initial simplicity**
>
> - **Context**: The kernel must handle hardware interrupts from devices like the keyboard and timer. Modern x86 systems support both legacy PICs and more advanced APICs, with APIC being required for multiprocessor support and offering more flexible interrupt routing.
> - **Options Considered**:
>   1. **Legacy 8259 PIC**: Two cascaded chips providing 15 maskable IRQ lines (IRQ0-15), widely emulated even in virtual machines, simpler programming model.
>   2. **APIC/X2APIC**: Modern interrupt controller supporting multiple CPUs, more interrupt vectors, and message-signaled interrupts (MSI), but requiring significantly more complex initialization and configuration.
>   3. **Hybrid approach**: Detect APIC presence and use it if available, fall back to PIC otherwise.
> - **Decision**: Implement the legacy 8259 PIC as the initial interrupt controller.
> - **Rationale**:
>   - **Educational simplicity**: The PIC's straightforward port I/O programming (8 registers per chip) makes the concepts of IRQ masking, prioritization, and EOI more accessible than APIC's memory-mapped registers and complex routing tables.
>   - **Universal emulation**: All x86 emulators (QEMU, Bochs) and real hardware boot in legacy PIC mode by default, ensuring the kernel works everywhere without complex detection logic.
>   - **Progressive enhancement**: The PIC implementation serves as a foundation; APIC support can be added later (see Future Extensions) without breaking existing functionality.
> - **Consequences**:
>   - **Limited to 15 IRQs**: Cannot handle systems with many modern devices without interrupt sharing.
>   - **Single-processor only**: Cannot support SMP without transitioning to APIC later.
>   - **Portable but legacy**: Works everywhere but doesn't leverage modern hardware capabilities.
>   - **IRQ conflict management**: Must remap PIC vectors (from default 0x08-0x0F and 0x70-0x77) to avoid overlapping CPU exception vectors (0x00-0x1F).

Option	Pros	Cons	Chosen?
Legacy 8259 PIC	Simple programming model (8 registers), universally available, easier debugging	Limited to 15 IRQs, no SMP support, requires vector remapping	**Yes** - for initial educational implementation
APIC/X2APIC	SMP support, more interrupt vectors, flexible routing, MSI support	Complex initialization, memory-mapped registers, detection required	No - too complex for initial milestone
Hybrid approach	Best of both worlds, works on all hardware	Significant complexity, doubles testing surface	No - defer to future enhancement

Decision: Interrupt Stack Switching with Dedicated Kernel Stack

Decision: Automatically switch to dedicated kernel stack on interrupt entry

- **Context**: When an interrupt occurs while the CPU is executing user-mode code (ring 3), the current stack pointer points to user-space memory, which may be invalid or malicious. Even in kernel mode, using the interrupted thread's stack risks corruption if nested interrupts occur.
- **Options Considered**:
  1. **No stack switching**: Use whatever stack is currently active (user or kernel). Simplest but dangerous.
  2. **Task gates**: Use x86 task switching feature to automatically load a new stack via TSS. Hardware-supported but complex and slow.
  3. **Software stack switching with TSS backup**: Use the Interrupt Stack Table (IST) feature or manually load a kernel stack pointer from the Task State Segment (TSS) in the interrupt handler prologue.
  4. **Separate stacks per privilege level**: Configure the CPU to automatically load a kernel stack from TSS when transitioning from ring 3 to ring 0 (but not for ring 0→ring 0 interrupts).
- **Decision**: Use x86's built-in privilege-level-based stack switching by setting up the Task State Segment (TSS) with valid kernel stack pointers.
- **Rationale**:
  - **Hardware-enforced safety**: The CPU automatically loads the `ss0:esp0` from the TSS when transitioning from user to kernel mode, preventing user-space from corrupting kernel stacks.
  - **Minimal assembly**: Requires only TSS initialization once at boot, not per-interrupt stack switching code.
  - **Nested interrupt protection**: Kernel-mode interrupts continue using the current kernel stack (which is fine), while user-mode interrupts get a fresh, known-good stack.
  - **Standard practice**: Matches how Linux and other OSes handle kernel stack management.
- **Consequences**:
  - **Requires TSS setup**: Must initialize a minimal Task State Segment and load it with `ltr` instruction.
  - **Single kernel stack per CPU**: In this uniprocessor design, all user-mode interrupts share the same kernel stack. For SMP, each CPU would need its own TSS entry.
  - **Stack overflow possible**: If interrupts nest deeply or handlers use excessive stack, the kernel stack could overflow. This is mitigated by keeping ISRs minimal.

Option	Pros	Cons	Chosen?
No stack switching	No setup required	User stack may be invalid, security risk, nested interrupts problematic	No - too dangerous
Task gates	Hardware isolation, full context switch	Complex, slow, deprecated in 64-bit	No - overly complex for needs
Software switching	Full control, flexible	Manual assembly in each ISR, error-prone	No - more code than necessary
Privilege-level switching (via TSS)	Hardware-supported, automatic for user→kernel, minimal code	Requires TSS setup, single stack per privilege level	**Yes** - balances safety and simplicity

Common Pitfalls: Interrupt Handling Implementation Errors

⚠️ Pitfall: Forgetting to Send End-of-Interrupt (EOI) to the PIC

- **Description**: After handling an interrupt, failing to send the EOI command (0x20 to PIC command ports) tells the PIC that the interrupt is still being serviced, blocking all lower-priority interrupts of the same or higher level.
- **Why it's wrong**: The PIC uses a fixed priority scheme (IRQ0 highest, IRQ7 lowest on PIC1, IRQ8-15 on PIC2). Without EOI, the PIC won't send more interrupts at that priority level or lower. This manifests as:
  - Timer interrupts (IRQ0) stop, freezing the system's sense of time.
  - Keyboard stops working after first keypress (IRQ1).
  - Eventually all hardware interrupts stop.
- **How to fix**: Always send EOI at the end of your interrupt handler. For PIC1 interrupts (IRQ0-7): `outb(PIC1_COMMAND, 0x20)`. For PIC2 interrupts (IRQ8-15): send EOI to both PICs: `outb(PIC2_COMMAND, 0x20); outb(PIC1_COMMAND, 0x20);`

⚠️ Pitfall: Incorrect PIC Remapping

- **Description**: The BIOS initializes the PIC to use interrupt vectors 0x08-0x0F for IRQ0-7 and 0x70-0x77 for IRQ8-15, which conflict with CPU exception vectors (0x00-0x1F are reserved for exceptions).
- **Why it's wrong**: If a keyboard interrupt (IRQ1) fires and the PIC hasn't been remapped, it will trigger vector 0x09, which is the "Coprocessor Segment Overrun" CPU exception. The CPU will try to execute the wrong handler, likely causing a cascade of exceptions ending in a triple fault.

- **How to fix**: Remap the PIC during kernel initialization before enabling interrupts:
  1. Send ICW1 (initialization command word) to both PICs to start initialization sequence.
  2. Send new vector offsets (e.g., 0x20 for PIC1, 0x28 for PIC2) as ICW2.
  3. Configure cascade wiring (ICW3).
  4. Set additional mode information (ICW4).
  5. Mask all interrupts initially, then unmask only those you're ready to handle.

⚠️ **Pitfall: Stack Corruption in Interrupt Handlers**

- **Description**: ISRs that modify critical registers without saving them, use excessive stack space, or have mismatched stack operations (push/pop imbalance) corrupt the execution context.
- **Why it's wrong**: When the IRET instruction executes, it expects the stack to contain exactly: `[esp] = EIP, [esp+4] = CS, [esp+8] = EFLAGS`. If additional values remain or registers are changed, the CPU returns to the wrong address, with wrong flags, causing unpredictable crashes.
- **How to fix**:
  - **Save all registers** in assembly stub before calling C handler: pushad (all general-purpose), push ds/es/fs/gs.
  - **Maintain stack alignment**: Ensure an even number of pushes before calling C functions (which expect 16-byte stack alignment).
  - **Match pushes with pops**: Before IRET, pop in reverse order.
  - **Minimize stack usage**: Keep ISRs lean; defer processing to non-interrupt context when possible.
  - **Use compiler directives**: Mark C handler with `__attribute__((interrupt))` if compiler supports it (though we use assembly stubs for full control).

⚠️ **Pitfall: Enabling Interrupts Too Early**

- **Description**: Calling `sti` to enable interrupts before the IDT is fully populated or the PIC is remapped.
- **Why it's wrong**: Any hardware interrupt (timer, keyboard) or software interrupt (INT instruction) will cause the CPU to look up a handler in the IDT. If entries are zero or point to invalid code, a General Protection Fault or Triple Fault will reset the system immediately.
- **How to fix**: Follow strict initialization order:
  1. Disable interrupts with `cli`.
  2. Remap PIC (while interrupts disabled).
  3. Populate IDT with valid handler addresses.
  4. Load IDT with `lidt`.
  5. Configure TSS for stack switching.
  6. Unmask specific IRQs in PIC.
  7. Finally enable interrupts with `sti`.

Data Structures: Interrupt Descriptor Table and Handler State

The IDT is the central data structure for interrupt handling, defining the mapping between interrupt vectors and their handlers.

IDT Entry Structure (`idt_entry`)

Each 8-byte entry in the IDT describes one interrupt gate, trap gate, or task gate. For our 32-bit interrupt gates:

Field	Type	Description
`offset_low`	`uint16_t`	Lower 16 bits of the interrupt handler's address (where the ISR code begins).
`selector`	`uint16_t`	Code segment selector in GDT (should be `KERNEL_CS = 0x08` for kernel handlers).
`zero`	`uint8_t`	Always zero (reserved).
`type_attr`	`uint8_t`	Type and attributes bitfield (see detailed breakdown below).
`offset_high`	`uint16_t`	Upper 16 bits of the interrupt handler's address.

The `type_attr` byte has the following bit layout:

Bits	Field	Value	Meaning
0-3	Gate Type	`0xE` (1110)	32-bit interrupt gate (automatically clears IF flag, preventing nested interrupts)
4	Storage Segment	`0`	Always 0 for interrupt/trap gates (1 for task gates)
5-6	Descriptor Privilege Level (DPL)	`0` or `3`	Minimum privilege required to call via INT instruction. `IDT_DPL_0` (0) for kernel-only, `IDT_DPL_3` (3<<5) for user-accessible system calls.
7	Present	`1`	`IDT_PRESENT` (1<<7) - entry is valid

IDT Register (`idtr`)

The 6-byte structure loaded by the `lidt` instruction:

Field	Type	Description
`limit`	`uint16_t`	Size of IDT in bytes minus 1 (e.g., 256 entries × 8 bytes - 1 = 2047).
`base`	`uint32_t*`	Linear address of the first byte of the IDT.

Saved Register State (`registers`)

When an interrupt occurs, the assembly stub pushes register values onto the stack in this order before calling the C handler:

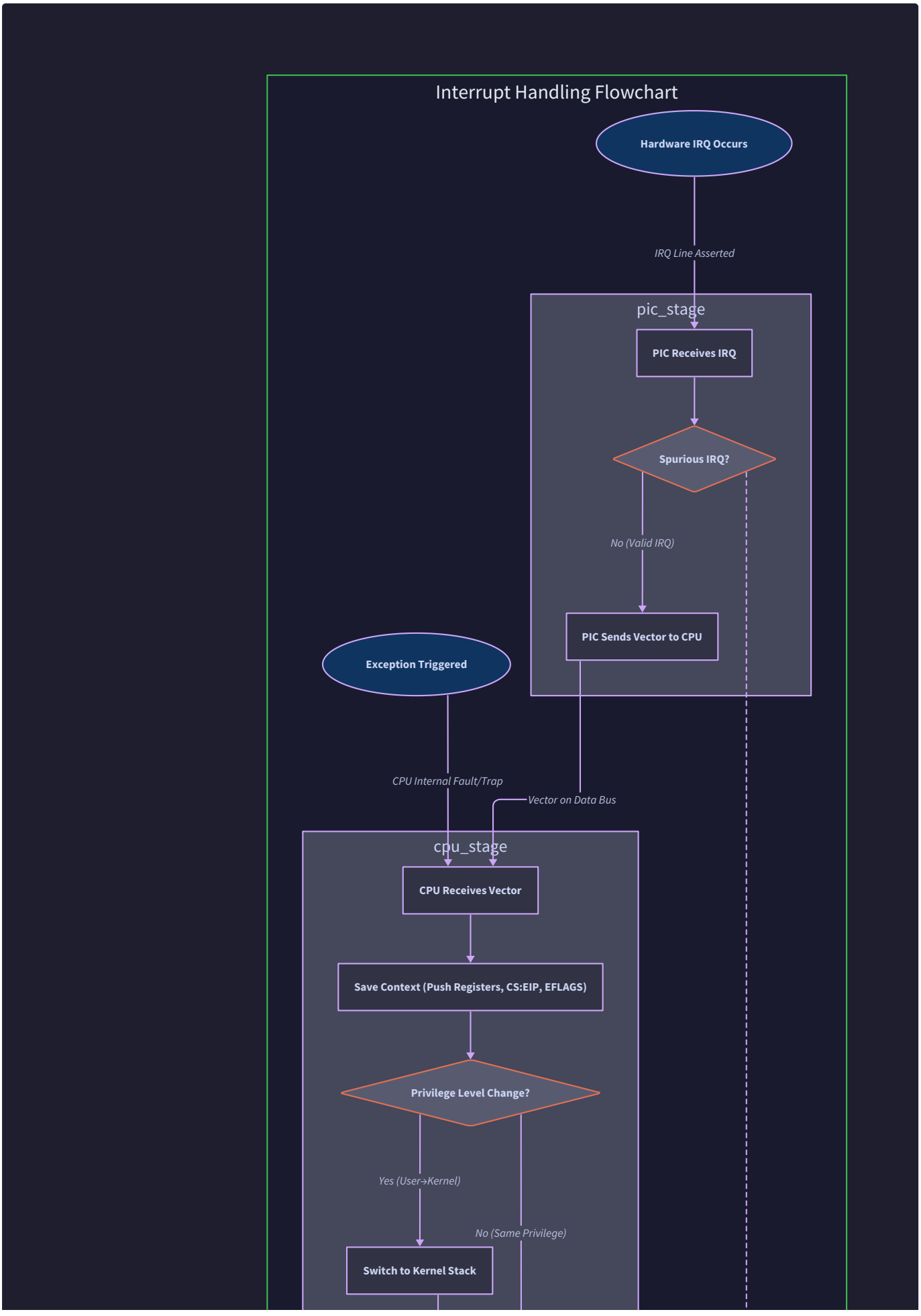
Field	Type	Description
`ds`	`uint32_t`	Data segment selector (pushed manually).
`edi` , `esi` , `ebp` , `esp` , `ebx` , `edx` , `ecx` , `eax`	`uint32_t`	General-purpose registers (pushed by `pushad`).
`int_no`	`uint32_t`	Interrupt vector number (pushed manually).
`err_code`	`uint32_t`	Error code (pushed by CPU for some exceptions, otherwise we push 0).
`eip`	`uint32_t`	Instruction pointer at time of interrupt (pushed by CPU).
`cs`	`uint32_t`	Code segment selector (pushed by CPU).
`eflags`	`uint32_t`	EFLAGS register (pushed by CPU).
`user_esp`	`uint32_t`	User stack pointer (only present if interrupt came from user mode, from TSS).
`ss`	`uint32_t`	Stack segment selector (only present if interrupt came from user mode).

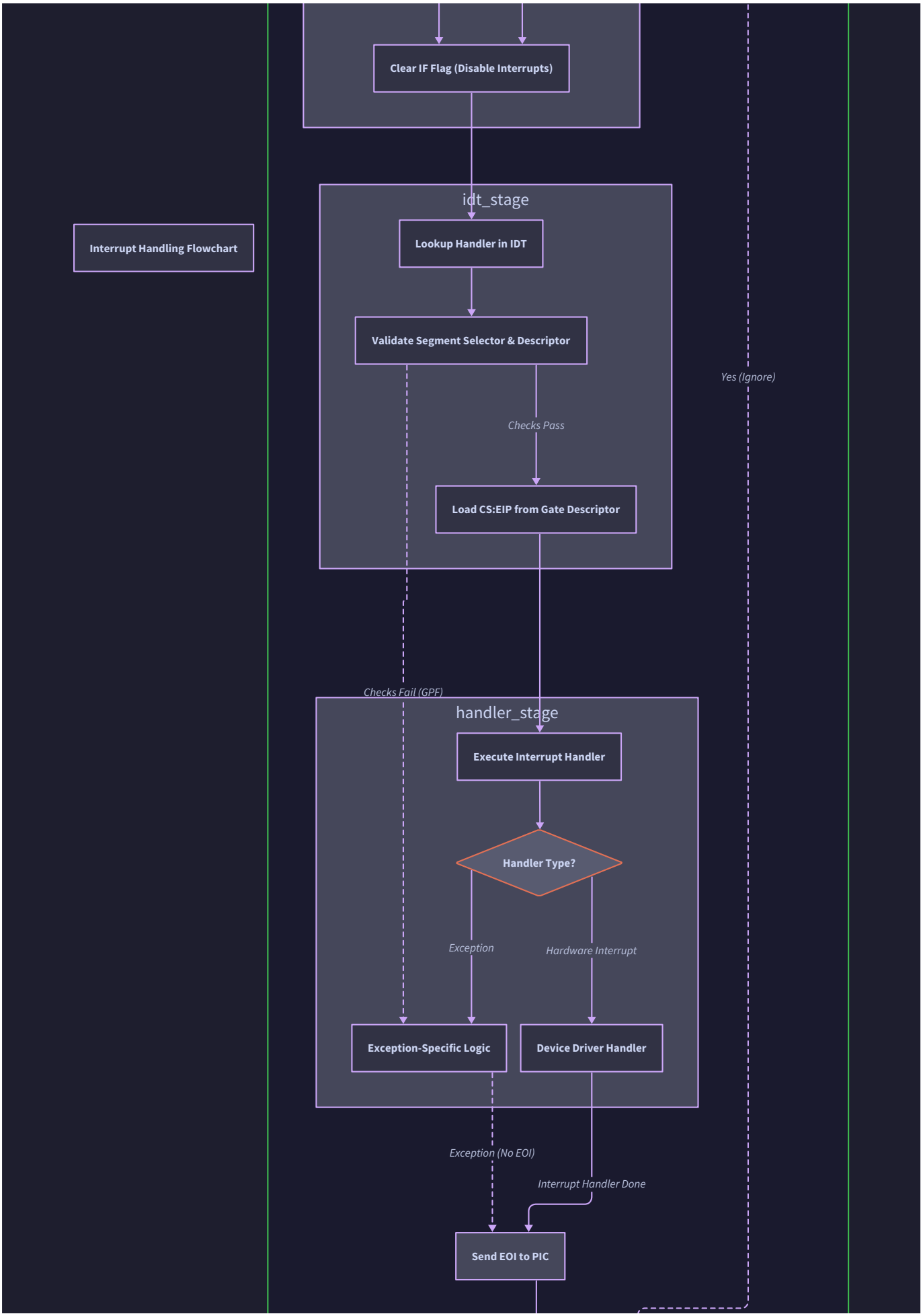
Interfaces: Interrupt Management API

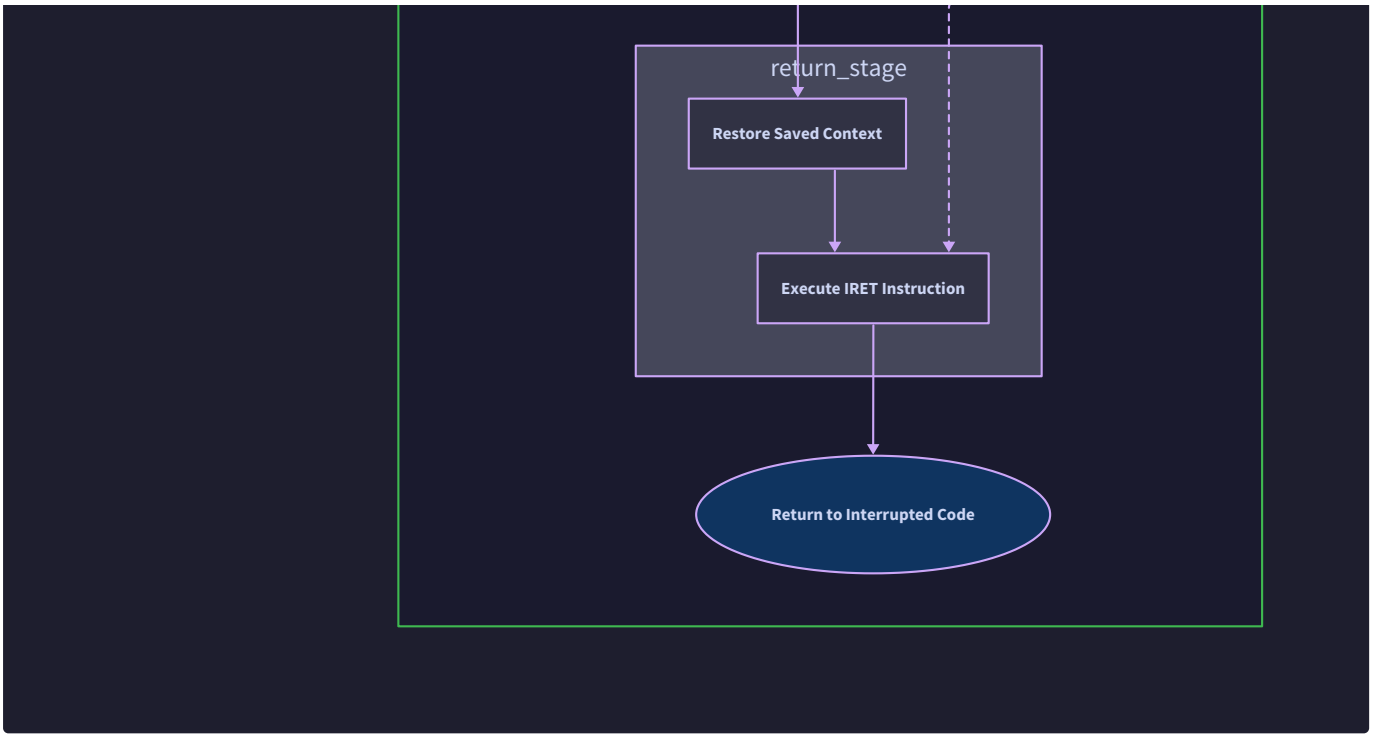
Method	Parameters	Returns	Description
`idt_init()`	`void`	`void`	Initializes the IDT: sets all 256 entries, loads with `lidt` .
`idt_set_gate(uint8_t num, uint32_t base, uint16_t sel, uint8_t flags)`	`num` : interrupt vector (0-255) `base` : handler address `sel` : code segment selector `flags` : type attributes	`void`	Sets a single IDT entry with the given handler address and attributes.
`isr_handler(struct registers *regs)`	`regs` : pointer to saved register structure	`void`	Common C-level dispatcher called from assembly stubs; routes to specific handler based on `regs->int_no` .
`pic_remap()`	`void`	`void`	Remaps PIC IRQs to vectors 0x20-0x2F, configures cascade, and masks all interrupts.
`keyboard_init()`	`void`	`void`	Initializes PS/2 keyboard: sets up interrupt handler, enables IRQ1, configures scancode set.
`irq_enable(uint8_t irq)`	`irq` : IRQ number (0-15)	`void`	Unmasks the specified IRQ in the PIC mask register.
`irq_disable(uint8_t irq)`	`irq` : IRQ number (0-15)	`void`	Masks (disables) the specified IRQ in the PIC mask register.

Interrupt Handling Algorithm

The complete flow from interrupt occurrence to handler return follows these steps (reference:


Interrupt Handling Flowchart
Hardware IRQ Occurs
IRQ Line Asserted
pic_stage
PIC Receives IRQ
Spurious IRQ?
No (Valid IRQ)
PIC Sends Vector to CPU
Exception Triggered
CPU Internal Fault/Trap
Vector on Data Bus
cpu_stage
CPU Receives Vector
Save Context (Push Registers, CS:EIP, EFLAGS)
Privilege Level Change?
Yes (User→Kernel)
No (Same Privilege)
Switch to Kernel Stack


Interrupt Handling Flowchart
Clear IF Flag (Disable Interrupts)
idt_stage
Lookup Handler in IDT
Validate Segment Selector & Descriptor
Checks Pass
Load CS:EIP from Gate Descriptor
Yes (Ignore)
Checks Fail (GPF)
handler_stage
Execute Interrupt Handler
Handler Type?
Exception
Hardware Interrupt
Exception-Specific Logic
Device Driver Handler
Exception (No EOI)
Interrupt Handler Done
Send EOI to PIC



):

1. **Hardware Event**: Device (e.g., keyboard controller) signals interrupt request line (IRQ1).
2. **PIC Processing**:
   - PIC evaluates priority (IRQ0 highest, IRQ7 lowest on master, IRQ8-15 on slave).
   - If interrupt is unmasked and higher priority than any in-service interrupt, PIC sends INT signal to CPU.
   - PIC sets corresponding bit in In-Service Register (ISR) to mark interrupt as being serviced.
3. **CPU Response**:
   - CPU finishes current instruction (unless it's a locked instruction like `rep movsb`).
   - Checks IF flag in EFLAGS (if interrupts are enabled).
   - Saves minimal state onto stack: EFLAGS, CS, EIP (and error code for some exceptions).
   - Clears IF flag (if interrupt gate) to prevent nested interrupts.
   - Loads new CS:EIP from IDT entry corresponding to interrupt vector.
4. **Assembly Stub Execution**:
   - Push error code (if not already pushed by CPU) to standardize stack layout.
   - Push interrupt vector number.
   - Save all general-purpose registers with `pushad`.
   - Save segment registers (ds, es, fs, gs).
   - Load kernel data segments (mov ax, 0x10; mov ds, ax; etc.).
   - Call C handler `isr_handler` with stack pointer as argument.
5. **C Handler Dispatch**:
   - Cast stack pointer to `struct registers*` to access saved state.
   - For exceptions (0-31): potentially panic with error message.
   - For IRQs (32-47): call specific device handler (e.g., `keyboard_handler` for IRQ1).
   - Send EOI to PIC for handled IRQs.
6. **Return Path**:
   - Restore segment registers (pop gs, fs, es, ds).
   - Restore general-purpose registers (`popad`).
   - Clean up error code and interrupt number from stack.
   - Execute `iret` to restore CS:EIP, EFLAGS, and potentially SS:ESP if returning to user mode.

Keyboard Driver Design

The keyboard driver translates raw PS/2 scancodes into ASCII characters, handling key press/release events and modifier keys (Shift, Ctrl, Alt, Caps Lock).

Scancode Set 1 Mapping

PS/2 keyboards typically use one of three scancode sets; most emulate set 1. Key events generate:

- **Make code**: Byte sent when key is pressed.
- **Break code**: `0xF0` followed by make code when key is released.

The driver must:

1. **Read from keyboard controller**: Read data port (0x60) when IRQ1 fires.
2. **Handle special bytes**: `0xE0` indicates extended key, `0xF0` indicates break code.
3. **Track state**: Maintain shift/ctrl/alt/caps lock state in a modifier bitmap.
4. **Convert to ASCII**: Use lookup tables (normal and shifted) based on modifier state.
5. **Buffer characters**: Store typed characters in a circular buffer for reading by kernel or processes.

Keyboard State Machine

Current State	Event (Byte Read)	Next State	Action
Normal	`0xE0`	Extended	Set extended flag, await next byte
Normal	`0xF0`	Break	Set break flag, await make code
Normal	other byte (make code)	Normal	Process key press: update modifiers, convert to ASCII if printable, add to buffer
Extended	any byte	Normal	Process extended key (e.g., arrow keys), clear extended flag
Break	any byte (make code)	Normal	Process key release: update modifiers if modifier key, clear break flag

ASCII Conversion Tables

Two 128-byte lookup tables map scancodes to ASCII:

- `scancode_to_ascii_normal`: For unshifted keys (a-z → lowercase, numbers, symbols).
- `scancode_to_ascii_shifted`: For shifted keys (A-Z, upper symbols).

Special keys (arrows, function keys, Ctrl/Alt) don't produce ASCII but generate special keycodes or modify state.

Implementation Guidance: IDT & Keyboard ISR

Technology Recommendations Table:

Component	Simple Option	Advanced Option
Interrupt Handling	Assembly stubs + C dispatcher	Compiler `__attribute__((interrupt))` with careful stack management
PIC Programming	Direct port I/O with inline assembly	Abstracted PIC driver with configuration structures
Keyboard Decoding	Lookup tables for scancode set 1	Auto-detection of scancode set, support for all sets
Buffer Management	Simple circular buffer	Lock-free ring buffer for SMP

Recommended File/Module Structure:

```
src/
├── boot/                # Bootloader (from Milestone 1)
├── kernel/
│   ├── arch/
│   │   ├── x86/
│   │   │   ├── idt.c     # IDT initialization and management
│   │   │   ├── idt.asm   # Assembly interrupt stubs
│   │   │   ├── isr.c     # C-level interrupt dispatcher
│   │   │   ├── pic.c     # PIC remapping and IRQ masking
│   │   │   └── tss.c     # Task State Segment setup
│   │   └── ports.c       # I/O port read/write utilities
│   ├── drivers/
│   │   ├── keyboard.c    # PS/2 keyboard driver
│   │   └── vga.c         # VGA driver (from Milestone 1)
│   ├── include/          # Headers
│   │   ├── kernel/
│   │   │   ├── idt.h
│   │   │   ├── isr.h
│   │   │   ├── keyboard.h
│   │   │   └── ports.h
│   │   └── arch/
│   │       └── x86/
│   │           ├── registers.h
│   │           └── tss.h
│   └── main.c            # Kernel main()
└── scripts/
    └── linker.ld         # Linker script
```

Infrastructure Starter Code (Complete):

```c
/* kernel/arch/x86/ports.c - I/O port utilities */

#include <kernel/arch/ports.h>


/* Write a byte to an I/O port */

void arch_port_write_byte(uint16_t port, uint8_t value) {

    asm volatile ("outb %0, %1" : : "a"(value), "Nd"(port));

}


/* Read a byte from an I/O port */

uint8_t arch_port_read_byte(uint16_t port) {

    uint8_t result;

    asm volatile ("inb %1, %0" : "=a"(result) : "Nd"(port));

    return result;

}


/* kernel/include/kernel/arch/registers.h - Saved register structure */

#ifndef REGISTERS_H

#define REGISTERS_H


#include <stdint.h>


/* Matches the order pushed by our assembly stubs */

struct registers {

    uint32_t ds;                                  /* Data segment selector */

    uint32_t edi, esi, ebp, esp, ebx, edx, ecx, eax; /* pushad */

    uint32_t int_no, err_code;                    /* Interrupt number and error code */

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

};


#endif
```

Core Logic Skeleton Code:

```c
/* kernel/arch/x86/idt.c - IDT Management */

#include <kernel/arch/idt.h>

#include <kernel/arch/ports.h>

#include <kernel/panic.h>


#define IDT_ENTRIES 256

static struct idt_entry idt[IDT_ENTRIES];

static struct idtr idt_ptr;


/* Set an IDT gate */

void idt_set_gate(uint8_t num, uint32_t base, uint16_t sel, uint8_t flags) {

    // TODO 1: Set offset_low to lower 16 bits of base

    // TODO 2: Set selector to sel (kernel code segment)

    // TODO 3: Set zero byte to 0

    // TODO 4: Set type_attr to flags | 0x60 (DPL=0) | 0x80 (present)

    // TODO 5: Set offset_high to upper 16 bits of base

}


/* Initialize and load IDT */

void idt_init(void) {

    idt_ptr.limit = sizeof(struct idt_entry) * IDT_ENTRIES - 1;

    idt_ptr.base = (uint32_t)&idt;



    // TODO 1: Zero out all IDT entries first



    // TODO 2: Set up exception handlers (0-31)

    //   Use idt_set_gate for each, pointing to assembly labels:

    //   extern void isr0(), isr1(), ... isr31();



    // TODO 3: Set up IRQ handlers (32-47)

    //   extern void irq0(), irq1(), ... irq15();



    // TODO 4: Load IDT with lidt instruction

    //   asm volatile("lidt (%0)" : : "r"(&idt_ptr));



    kprintf("IDT initialized at 0x%x\n", idt_ptr.base);

}


/* kernel/arch/x86/isr.c - C-level Interrupt Dispatcher */

#include <kernel/arch/isr.h>

#include <kernel/arch/registers.h>

#include <kernel/arch/pic.h>
```

```c
#include <kernel/panic.h>
#include <kernel/kprintf.h>

/* Array of function pointers for IRQ handlers */
static void (*irq_handlers[16])(struct registers*);

/* Register an IRQ handler */
void irq_register_handler(int irq, void (*handler)(struct registers*)) {
    // TODO 1: Validate irq is 0-15
    // TODO 2: Assign handler to irq_handlers[irq]
}

/* Main C handler called from assembly stubs */
void isr_handler(struct registers *regs) {
    // TODO 1: Check if interrupt is exception (0-31)
    //   If yes: panic with error message showing int_no and registers

    // TODO 2: Check if interrupt is IRQ (32-47)
    //   Calculate irq = regs->int_no - 32

    // TODO 3: If handler exists for this IRQ, call it
    //   if (irq_handlers[irq]) irq_handlers[irq](regs);

    // TODO 4: Send EOI to PIC
    //   if (irq >= 8) outb(PIC2_COMMAND, 0x20);
    //   outb(PIC1_COMMAND, 0x20);
}

/* kernel/arch/x86/pic.c - PIC Remapping */
#include <kernel/arch/pic.h>
#include <kernel/arch/ports.h>

#define PIC1_CMD  PIC1_COMMAND
#define PIC1_DATA PIC1_DATA
#define PIC2_CMD  PIC2_COMMAND
#define PIC2_DATA PIC2_DATA

void pic_remap(void) {
    // Save current masks
    uint8_t mask1 = arch_port_read_byte(PIC1_DATA);
    uint8_t mask2 = arch_port_read_byte(PIC2_DATA);

    // TODO 1: Send ICW1 (initialization) to both PICs
```

```
    //   arch_port_write_byte(PIC1_CMD, 0x11); // 0x11 = ICW1 + ICW4 needed
    //   arch_port_write_byte(PIC2_CMD, 0x11);

    // TODO 2: Send ICW2 (vector offsets)
    //   arch_port_write_byte(PIC1_DATA, 0x20); // IRQ0-7 -> 0x20-0x27
    //   arch_port_write_byte(PIC2_DATA, 0x28); // IRQ8-15 -> 0x28-0x2F

    // TODO 3: Send ICW3 (cascade wiring)
    //   arch_port_write_byte(PIC1_DATA, 0x04); // PIC1 has slave at IRQ2
    //   arch_port_write_byte(PIC2_DATA, 0x02); // PIC2 cascade identity

    // TODO 4: Send ICW4 (additional mode info)
    //   arch_port_write_byte(PIC1_DATA, 0x01); // 8086 mode
    //   arch_port_write_byte(PIC2_DATA, 0x01);

    // Restore masks
    arch_port_write_byte(PIC1_DATA, mask1);
    arch_port_write_byte(PIC2_DATA, mask2);

    kprintf("PIC remapped to vectors 0x20-0x2F\n");
}

void irq_enable(uint8_t irq) {
    uint16_t port;
    uint8_t value;

    // TODO 1: Determine which PIC (master or slave) handles this IRQ
    // TODO 2: Read current mask from appropriate PIC data port
    // TODO 3: Clear the bit for this IRQ (0=enable, 1=disable)
    // TODO 4: Write new mask back to PIC
}

/* kernel/drivers/keyboard.c - PS/2 Keyboard Driver */
#include <kernel/drivers/keyboard.h>
#include <kernel/arch/ports.h>
#include <kernel/arch/isr.h>
#include <kernel/kprintf.h>

#define KEYBOARD_DATA_PORT 0x60
#define KEYBOARD_STATUS_PORT 0x64

static uint8_t shift_down = 0;
```

```c
static uint8_t ctrl_down = 0;
static uint8_t alt_down = 0;
static uint8_t caps_lock = 0;
static uint8_t extended = 0;
static uint8_t break_code = 0;

/* Circular buffer for typed characters */
#define KEYBOARD_BUFFER_SIZE 256
static char key_buffer[KEYBOARD_BUFFER_SIZE];
static uint32_t buf_read_pos = 0;
static uint32_t buf_write_pos = 0;

/* Lookup tables (simplified - needs complete implementation) */
static const char scancode_to_ascii_normal[128] = {
    0, 0, '1', '2', '3', '4', '5', '6', '7', '8', '9', '0', '-', '=', '\b',
    '\t', 'q', 'w', 'e', 'r', 't', 'y', 'u', 'i', 'o', 'p', '[', ']', '\n',
    0, 'a', 's', 'd', 'f', 'g', 'h', 'j', 'k', 'l', ';', '\'', '`', 0, '\\',
    'z', 'x', 'c', 'v', 'b', 'n', 'm', ',', '.', '/', 0, '*', 0, ' ', 0,
    /* ... remaining entries */
};

static void keyboard_handler(struct registers *regs) {
    (void)regs; // Unused parameter


    uint8_t scancode = arch_port_read_byte(KEYBOARD_DATA_PORT);


    // TODO 1: Handle special bytes (0xE0 = extended, 0xF0 = break)
    //   Set extended or break flags and return early


    // TODO 2: Handle modifier keys (Shift, Ctrl, Alt, Caps Lock)
    //   Update state variables on press/release


    // TODO 3: For printable keys, convert scancode to ASCII
    //   Use appropriate lookup table based on shift/caps state


    // TODO 4: Add character to circular buffer if printable
    //   key_buffer[buf_write_pos] = ascii_char;
    //   buf_write_pos = (buf_write_pos + 1) % KEYBOARD_BUFFER_SIZE;


    // TODO 5: Echo character to screen (for testing)
    //   if (ascii_char) kprintf("%c", ascii_char);
}
```

```
void keyboard_init(void) {
    // TODO 1: Register keyboard handler for IRQ1
    //   irq_register_handler(1, keyboard_handler);

    // TODO 2: Enable IRQ1 (keyboard) in PIC mask
    //   irq_enable(1);

    kprintf("Keyboard driver initialized\n");
}
```

Assembly Stubs (Critical Infrastructure):

```assembly
; kernel/arch/x86/idt.asm - Assembly interrupt stubs
section .text
global isr0, isr1, isr2, ... isr31  ; Exception handlers
global irq0, irq1, irq2, ... irq15  ; IRQ handlers

; Common stub macro for all interrupts
%macro ISR_NOERRCODE 1  ; Interrupt without error code
global isr%1
isr%1:
    push 0              ; Push dummy error code
    push %1             ; Push interrupt number
    jmp isr_common_stub
%endmacro

%macro ISR_ERRCODE 1    ; Interrupt that pushes error code
global isr%1
isr%1:
    push %1             ; Push interrupt number (error code already on stack)
    jmp isr_common_stub
%endmacro

; TODO: Define all 32 exception handlers using appropriate macro
; ISR_NOERRCODE 0   ; Divide by zero
; ISR_NOERRCODE 1   ; Debug
; ISR_NOERRCODE 2   ; Non-maskable interrupt
; ... etc.

; IRQ handlers (all no error code)
%macro IRQ 2
global irq%1
irq%1:
    push 0
    push %2             ; Interrupt number (32 + IRQ)
    jmp irq_common_stub
%endmacro

; TODO: Define all 16 IRQ handlers
; IRQ 0, 32   ; Timer
; IRQ 1, 33   ; Keyboard
; ... etc.

extern isr_handler

; Common stub for exceptions
isr_common_stub:
    ; TODO 1: Save all registers (pushad)
    ; TODO 2: Save segment registers (push ds, es, fs, gs)
    ; TODO 3: Load kernel data segments (mov ax, 0x10; mov ds, ax; etc.)
    ; TODO 4: Call C handler (push esp; call isr_handler; add esp, 4)
    ; TODO 5: Restore segment registers (pop gs, fs, es, ds)
    ; TODO 6: Restore general registers (popad)
    ; TODO 7: Clean up error code and interrupt number (add esp, 8)
    ; TODO 8: Return from interrupt (iret)

irq_common_stub:
    ; Same as isr_common_stub but different handler
    ; TODO: Same steps as above, but call isr_handler
    ; (The C handler will distinguish exceptions from IRQs)
```

Language-Specific Hints:

- **Inline Assembly in C**: Use `asm volatile` with proper constraints for port I/O. The `"a"` constraint means value in EAX register, `"Nd"` means constant in DX register (or small immediate).
- **Packed Structures**: Ensure `struct idt_entry` is packed (no compiler padding) with `__attribute__((packed))` to match CPU expectations.
- **Interrupt Context**: Never call blocking functions or use floating point in ISRs. Keep handlers minimal and fast.
- **Volatile Variables**: Keyboard buffer indices (`buf_read_pos` , `buf_write_pos`) should be `volatile` if accessed from both interrupt and main context.

Milestone Checkpoint:

After implementing this component, test with QEMU:

1. **Build and run**: `make run`
2. **Expected behavior**:
   - Kernel boots with "IDT initialized", "PIC remapped", "Keyboard driver initialized" messages.
   - System doesn't crash immediately (no triple fault from unhandled interrupts).
   - Typing on keyboard echoes characters to screen.

- Pressing `Shift` + letter produces uppercase.
- Backspace (`\b`) moves cursor back.

3. **Verification commands**:
   - Press keys: should see characters appear.
   - Hold multiple keys: should handle basic rollover.
   - Check that timer interrupts are working (system clock advancing if implemented).
4. **Signs of trouble**:
   - No keyboard response: Check PIC remapping, IRQ1 unmasking, EOI sending.
   - Characters wrong: Scancode lookup table incomplete.
   - System freezes on keypress: Stack corruption in ISR; check push/pop balance.
   - Double characters: Not handling break codes correctly.

Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
Triple fault immediately after boot	IDT not initialized or contains NULL entries	Use QEMU debugger (`-s -S`), break at kernel entry, examine IDT memory	Ensure `idt_init()` called before `sti` , all 256 entries populated
Keyboard works once then stops	Missing EOI to PIC	Add debug print in keyboard handler showing scancode and whether EOI sent	Send EOI at end of IRQ handler: `outb(0x20, 0x20)` for IRQ1
Wrong characters appear	Incorrect scancode table	Print scancode hex values in handler, compare with PS/2 set 1 reference	Complete the lookup tables with all scancodes
System hangs on keypress	Stack misalignment or corruption	Add stack canary values, check push/pop count in assembly stub	Ensure even number of pushes before C call, match pops before `iret`
No interrupts fire at all	PIC not remapped or IRQs masked	Read PIC mask registers (ports 0x21, 0xA1), should be 0xFC (mask all but IRQ0/1)	Call `pic_remap()` , unmask IRQ1 with `irq_enable(1)`
Nested interrupts cause crash	Interrupts re-enabled in handler	Check IF flag not set in handler (interrupt gates auto-clear it)	Don't call `sti` in ISR; use interrupt gates (type 0xE) not trap gates (0xF)

5.3 Component: Memory Management (Milestone 3)

> **Milestone(s):** Milestone 3: Memory Management

This component transforms the kernel's view of memory from a flat, physical address space into a structured, protected virtual environment. It provides three foundational services: tracking and allocating physical RAM frames, establishing virtual-to-physical address mappings through paging, and managing dynamic memory allocation for kernel data structures. Success here creates the isolation and abstraction necessary for running multiple user processes safely.

Mental Model: The City Planner and Librarian

Imagine managing the memory of a computer as akin to organizing a growing city and its library system. This mental model breaks down the three core responsibilities:

1. **The City Planner (Physical Frame Allocator):** The city planner oversees all physical land (RAM). They maintain a master map (`pmm_state`) showing which plots (4KB frames) are vacant, occupied by buildings, or reserved for special purposes (like parks or government buildings which represent memory-mapped I/O regions). When the kernel needs space for a new structure (like a page table), it requests a plot from the planner. The planner finds a suitable vacant frame, marks it as occupied on the map, and returns its physical address. When a structure is demolished (memory is freed), the planner updates the map to mark that frame as vacant again.
2. **The Address Translator / Postal Service (Paging System):** The city uses a complex addressing system where every building has a "virtual address" for mail delivery, which is different from its physical plot location. The page tables act as the central postal directory. When a program sends mail to virtual address `0x401000` , the CPU's Memory Management Unit (MMU) consults the directory (page directory and page table) to translate this into the actual physical plot number. The kernel, acting as the postal service administrator, sets up and maintains these translation directories, ensuring the VGA buffer's virtual address always delivers to the physical screen memory at `0xB8000` , and that each process's view of address `0x400000` points to its own private physical data.
3. **The Librarian (Kernel Heap Allocator):** Inside a large public building (a single 4KB frame allocated by the city planner), the librarian manages a collection of bookshelves (the kernel heap). Programs request books (small, variable-sized memory blocks) by calling `malloc` . The librarian finds a suitable empty spot on a shelf, marks it as checked out in their ledger (heap metadata), and returns a pointer to that spot. When a book is returned (`free`), the librarian marks the spot as available. The librarian's key challenge is minimizing wasted space between books (fragmentation) and efficiently merging adjacent free spots when books are returned. If the entire building fills up, the librarian requests a new building (another frame) from the city planner to expand the library.

This separation of concerns is critical: the city planner worries about *physical scarcity*, the postal service about *translation and isolation*, and the librarian about *convenient, variable-sized allocations* within the large chunks provided by the planner.

ADRs: Physical Allocator Bitmap vs. Stack, Paging Strategy

Decision: Physical Memory Tracking with a Bitmap

- **Context**: After obtaining the memory map from the bootloader (`multiboot_info`), the kernel must track which 4KB physical frames are free and which are in use by the kernel, devices, or future processes. We need a data structure that is memory-efficient, provides O(n) search in the worst case but O(1) allocation in common cases, and allows us to mark arbitrary frames as used or free (e.g., for memory-mapped I/O).
- **Options Considered**:
  1. **Bitmap**: An array of bits, where each bit represents the status (1=used, 0=free) of one physical frame. The bitmap itself is stored in a reserved, known region of memory.
  2. **Stack/Free List**: Maintain a stack (or linked list) of addresses of free frames. Allocating pops an address; freeing pushes it back.
  3. **Buddy Allocator**: A more complex system that can allocate blocks in power-of-two sizes, reducing external fragmentation but increasing internal fragmentation and complexity.
- **Decision**: Use a **bitmap allocator**.
- **Rationale**:
  1. **Memory Overhead is Predictable and Small**: The bitmap size is fixed at ~~1 bit per 4KB frame (~~3.2 KB for 128MB RAM). The stack approach has similar overhead but can become fragmented in metadata.
  2. **Flexibility for Marking Arbitrary Ranges**: It's straightforward to mark any contiguous range of frames as used (e.g., for the kernel image, multiboot modules, or MMIO regions) by setting a sequence of bits. A stack would require iterating to remove specific addresses.
  3. **Good Cache Locality for Sequential Allocation**: A simple "first-fit" or "next-fit" search through the bitmap has decent cache performance. We can optimize by remembering the last allocated index (`last_alloc_index`).
  4. **Simplicity for Educational Purposes**: The bitmap's operation is visually and conceptually simple to understand and debug (you can literally print the bitmap).
- **Consequences**:
  - Allocation time is O(n) in the worst case (if memory is very fragmented), but with a next-fit approach and typical early-boot fragmentation patterns, it approaches O(1).
  - The kernel must permanently reserve memory for the bitmap itself. We allocate this space early, before the allocator is fully initialized, by marking those frames as used in the bitmap.
  - Freeing a frame is a constant-time bit clear operation.

Option	Pros	Cons	Chosen?
Bitmap	Predictable, small overhead (~1/32768 of RAM). Easy to mark arbitrary ranges. Simple to implement and debug.	Worst-case allocation is O(frames). Requires scanning for free frames.	**Yes**
Stack/Free List	O(1) allocation and free. Very fast.	Requires storing addresses (4 bytes per free frame). Harder to initially populate and mark arbitrary used ranges. Can be corrupted by bugs.	No
Buddy Allocator	Efficient for power-of-two allocations, reduces external fragmentation.	Significant internal fragmentation for non-power-of-two requests. Complex implementation. Overkill for our simple frame allocator.	No

Decision: Higher-Half Kernel with 3GB/1GB Split and Recursive Page Table Mapping

- **Context**: We must decide how to lay out the virtual address space: where the kernel code/data lives versus user space, and how the kernel will efficiently manipulate page tables (which are themselves stored in memory and subject to paging).
- **Options Considered**:
  1. **Lower-Half Kernel**: Kernel occupies the lower part of virtual addresses (e.g., `0x100000`). User processes share the same space with different page tables.
  2. **Higher-Half Kernel (3GB/1GB Split)**: Kernel occupies the upper 1GB of a 4GB space (virtual addresses `0xC0000000` and above). User processes use the lower 3GB.
  3. **Higher-Half Kernel (2GB/2GB Split)**: Kernel and user each get 2GB.
- **Decision**: Use a **Higher-Half Kernel with a 3GB/1GB split** (`0xC0000000` as the kernel base) and implement **recursive page table mapping** for kernel-side page table management.
- **Rationale**:
  1. **Simplifies User Process Creation**: Every user process can have its own complete 3GB address space starting at `0x0`, with no need to avoid kernel addresses. The kernel is always mapped at the same high virtual address in every address space.
  2. **Protection**: User code, which runs in lower privilege levels (ring 3), cannot accidentally or maliciously access kernel memory because those addresses (`>=0xC0000000`) will be mapped with `PTE_USER=0` (supervisor-only).
  3. **Recursive Mapping Solves a Chicken-and-Egg Problem**: To modify a page table entry (PTE), you need its virtual address. But the PTE's page table might not be mapped. Recursive mapping maps the entire page directory into itself at a known virtual address (e.g., `0xFFC00000`), providing direct, uniform access to all PTEs.
  4. **Established Convention**: This is a standard layout used by Linux, BSD, and many educational OSes, making resources and explanations more applicable.
- **Consequences**:
  - The bootloader must load the kernel at a *physical* address that can be identity-mapped initially and later mapped to `0xC0000000`. This requires careful linker script adjustments (setting `-Ttext=0x100000` but using addresses with `0xC0000000` offset in C code).
  - The initial page table setup (in `vmm_init`) is more complex, as it must create mappings for both the early boot identity mapping and the final higher-half mapping.
  - Recursive mapping consumes one entry in the page directory (typically the last one), reducing the directly addressable kernel space by 4MB. This is an acceptable trade-off.

Option	Pros	Cons	Chosen?
Lower-Half Kernel	Simpler initial bootstrapping. No address translation complications in early boot.	User process address space is limited and must avoid kernel addresses. Less conventional.	No
Higher-Half (3GB/1GB)	Clean separation, user space starts at 0. Strong isolation. Industry standard.	More complex bootstrapping. Requires careful linker script.	**Yes**
Higher-Half (2GB/2GB)	More kernel address space.	Less user address space. Less common.	No

Core Data Structures

The memory manager is built upon several key data structures that track physical frames, define virtual mappings, and manage the heap.

Physical Memory Manager State (`pmm_state`)

This structure holds the global state of the physical frame allocator. It is initialized by `pmm_init` using the memory map from the bootloader.

Field Name	Type	Description
`bitmap`	`uint32_t*`	Pointer to the bitmap array. Each bit represents one 4KB physical frame. Bit=1 means the frame is in use (allocated or reserved). The bitmap is stored in a reserved memory area.
`total_frames`	`size_t`	Total number of 4KB frames in the system (calculated from the highest usable memory address).
`free_frames`	`size_t`	Current count of free (available) frames. Used for quick out-of-memory checks.
`last_alloc_index`	`size_t`	The index in the bitmap where the last search for a free frame ended. Used to implement a "next-fit" allocation strategy to avoid scanning from the start every time.
`bitmap_frames`	`uint32_t`	The number of 4KB frames occupied by the bitmap itself. Stored so these frames can be explicitly marked as "used" and never allocated.

Page Directory and Table Entries (`page_table_entry_t`)

A page table entry (PTE) and a page directory entry (PDE) share the same 32-bit format. The CPU interprets them based on context: a PDE points to a page table, a PTE points to a physical frame.

We define `page_table_entry_t` as a `uint32_t` for direct bit manipulation. The following table describes the bit layout and flags. The constants (e.g., `PTE_PRESENT`) are defined as macros or enums.

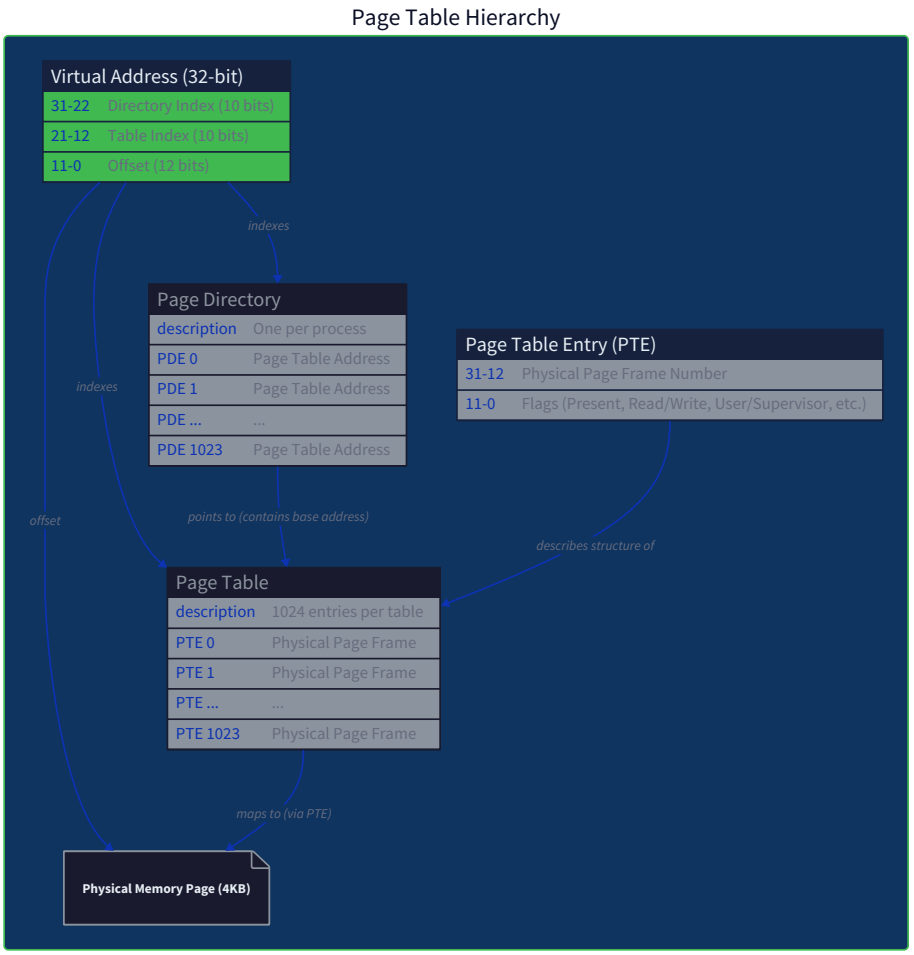
Bits (31-12)	Bits (11-9)	Bits (8-0)
Frame Address (20 bits)	**Available for OS**	**Flags**

Flag Field Details:

Flag Constant	Bit Position	Description
`PTE_PRESENT`	0	Set if the entry is valid (the page/frame is in memory). If 0, any access causes a page fault.
`PTE_WRITABLE`	1	If set, the page is writable. If clear, writes cause a page fault.
`PTE_USER`	2	If set, the page is accessible from user mode (ring 3). If clear, only supervisor (ring 0) can access it.
`PTE_WRITETHROUGH`	3	Controls write-through caching policy for this page.
`PTE_CACHE_DISABLE`	4	If set, disables caching for this page. Crucial for memory-mapped I/O regions.
`PTE_ACCESSED`	5	Set by CPU automatically when the page is read from or written to. Can be cleared by the OS to track page usage.
`PTE_DIRTY`	6	(For PTEs only) Set by CPU automatically when the page is written to. Used by the OS for page-out decisions.
`PTE_PAT`	7	Used in conjunction with PAT (Page Attribute Table) for advanced cache control (we ignore this).
`PTE_GLOBAL`	8	If set, the translation is not flushed from the TLB on a CR3 reload (useful for kernel pages).

The **Frame Address** field stores the **physical address of the target frame** shifted right by 12 bits (divided by 4096). This is because the address is 4KB-aligned, so its lower 12 bits are always zero.

Page Table Structure Diagram



Kernel Heap Block Header

The heap allocator (`heap_init` , `malloc` , `free`) requires metadata for each allocated block to track its size and free/used status. We use a common "boundary tag" approach with a header and footer.

Structure Name	Field Name	Type	Description
`heap_block_header`	`magic`	`uint32_t`	A magic number (e.g., `0xDEADBEEF`) used to detect heap corruption. If this value is corrupted, `free` can panic.
	`size`	`size_t`	The total size of the *user data area* in bytes (excluding header and footer).
	`is_free`	`bool`	A flag indicating whether this block is currently free.
	`next`	`struct heap_block_header*`	Pointer to the next block header in the heap's implicit free/used list.
`heap_block_footer`	`magic`	`uint32_t`	Same magic number as the header, placed at the end of the block for backward traversal and corruption checking.
	`header`	`struct heap_block_header*`	Pointer back to the block's header. Used when coalescing with the previous free block.

Algorithms and Workflows

1. Physical Frame Allocation (`pmm_alloc_frame`)

This algorithm is called when the kernel or `vmm_init` needs a fresh 4KB frame (e.g., for a new page table or user process data).

1. **Check Availability**: If `pmm_state.free_frames` is zero, return a `NULL` pointer (or trigger a kernel panic due to out-of-memory).
2. **Search Bitmap**: Starting from `pmm_state.last_alloc_index`, scan the bitmap array for the first 32-bit (`uint32_t`) element that is not equal to `0xFFFFFFFF` (i.e., it has at least one free bit).
3. **Find Free Bit**: Within that element, find the index of the first `0` bit using a helper like `__builtin_ffs(~bitmap_word)` or a loop.
4. **Calculate Frame Index**: The global frame index is: `(word_index * 32) + bit_index`.
5. **Mark Frame Used**: Set the corresponding bit in the bitmap to `1`. Decrement `pmm_state.free_frames`. Update `pmm_state.last_alloc_index` to the frame index just allocated (or the next one).
6. **Return Physical Address**: Convert the frame index to a physical address: `frame_index * 4096`. Return this address.
7. **Wrap Around**: If the scan reaches the end of the bitmap, wrap around to index 0 and continue. If a full loop finds no free frames, panic (this should have been caught in step 1).

2. Page Table Manipulation with Recursive Mapping

With recursive mapping, the last entry of the page directory (PDE 1023) points to the page directory itself. This creates a virtual address formula to access any PTE or PDE.

- **Accessing a Page Directory Entry (PDE) for a given virtual address `vaddr`:**
  - Let `dir_index = (vaddr >> 22) & 0x3FF` (bits 31-22).
  - The virtual address of that PDE is: `0xFFC00000 + (dir_index * 4)`.
  - Example: The PDE controlling virtual addresses starting at `0xC0000000` (kernel base) is at `0xFFC00000 + (0x300 * 4) = 0xFFC00C00`.
- **Accessing a Page Table Entry (PTE) for a given virtual address `vaddr`:**
  - Let `dir_index = (vaddr >> 22) & 0x3FF`.
  - Let `table_index = (vaddr >> 12) & 0x3FF` (bits 21-12).
  - The virtual address of that PTE is: `0xFFC00000 + (0x400 * dir_index) + (table_index * 4)`.
  - Explanation: `0xFFC00000` maps the entire 4MB region of the page directory. `0x400 * dir_index` skips to the virtual page that corresponds to the `dir_index`-th page table. Since each PTE is 4 bytes, `table_index * 4` gives the offset within that page.

Procedure to Map a Virtual Page to a Physical Frame:

1. Get the PTE virtual address for the target `vaddr` using the formula above.
2. Ensure the corresponding PDE (and thus the page table) exists and is present. If not, allocate a new physical frame for the page table, clear it to zero, and create a PDE pointing to it with flags `PTE_PRESENT | PTE_WRITABLE`.
3. Write the PTE: Set its value to `(phys_frame_addr & PTE_FRAME_MASK) | flags`. Flags should include at least `PTE_PRESENT`.
4. **Invalidate TLB Entry**: Use the `invlpg` instruction on the virtual address `vaddr` to flush any stale translation from the CPU's TLB.

3. Kernel Heap Allocation (`malloc`)

The heap is initialized by `heap_init`, which requests a large physical frame (or multiple) from the PMM and sets up the initial free block covering the entire frame.

1. **Find a Suitable Free Block (`malloc`)**:
   - Traverse the linked list of heap blocks (starting from a global `heap_start` pointer).
   - For each block where `is_free` is true and `size >= requested_size`, consider it a candidate.
   - Use a **first-fit** strategy: choose the first candidate block encountered.
   - If no block is large enough, call `heap_expand` to request another frame from the PMM, add it to the heap as a new large free block, and retry.
2. **Split the Block (If Necessary)**:
   - If the candidate free block is significantly larger than the requested size (e.g., `block->size >= requested_size + sizeof(header) + sizeof(footer) + MIN_BLOCK_SIZE`), split it.
   - Calculate the address for the new block header inside the existing free block.
   - Adjust the original block's size to the requested size. Update its footer.
   - Create a new free block header and footer in the remaining space, setting its size and linking it into the list.
   - This action helps reduce internal fragmentation.
3. **Mark Block as Used and Return Pointer**:
   - Set the chosen block's `is_free` to false.
   - Return a pointer to the user data area, which is `(char*)block_header + sizeof(heap_block_header)`.
4. **Freeing a Block (`free`)**:
   - Given the user pointer, calculate the block header address: `block = (heap_block_header*)((char*)ptr - sizeof(heap_block_header))`.
   - **Sanity Check**: Verify `block->magic == HEAP_MAGIC`. If not, call `panic` (heap corruption detected).
   - Mark the block as free: `block->is_free = true`.

- **Coalesce with Next Block (if free)**: Check the block immediately following this one in memory (using `block->size`). If that block's header exists and is free, merge them: add the next block's size (plus its metadata) to this block's size, update this block's footer, and adjust the linked list pointers.
- **Coalesce with Previous Block (if free)**: Use the footer of the block immediately before this one to find its header. If that block is free, merge them (the previous block absorbs the current block). This requires updating the previous block's size and footer, and skipping the current block in the list.

Common Pitfalls

⚠️ Pitfall: Forgetting to Invalidate the TLB After Modifying a Page Table Entry

- **Description**: The CPU caches address translations in the Translation Lookaside Buffer (TLB). If you modify a PTE/PDE in memory but don't tell the CPU, it may continue using the old, cached translation.
- **Why it's Wrong**: This leads to unpredictable behavior: writes may go to the wrong physical address, or a page might seem to retain old data. This is especially dangerous when unmapping pages or changing permissions.
- **How to Fix**: After any page table update that changes the mapping or flags for a specific virtual page, execute `asm volatile("invlpg (%0)" : : "r" (vaddr) : "memory")`. When changing the entire address space (e.g., during a context switch by writing to CR3), a full TLB flush occurs automatically.

⚠️ Pitfall: Incorrectly Setting Up Recursive Mapping Leading to Infinite Recursion or Page Faults

- **Description**: When setting the last PDE to point to the page directory's physical address, you must ensure the flags are correct (`PTE_PRESENT | PTE_WRITABLE`) and that you use the *physical* address of the page directory, not its virtual address. Also, you must not attempt to access the recursive mapping area *before* enabling paging.
- **Why it's Wrong**: Using the virtual address creates a circular dependency (the CPU needs the mapping to translate the address you're using to set up the mapping). This causes a triple fault or silent failure.
- **How to Fix**: In `vmm_init`, before calling `enable_paging`, store the *physical* address of the page directory in a variable. Use this physical address to set the recursive PDE. Only after paging is enabled should you calculate and use the virtual addresses (`0xFFC00000`, etc.) to access page tables.

⚠️ Pitfall: Heap Fragmentation and Corruption Due to Missing Boundary Tags or Magic Numbers

- **Description**: Implementing `malloc` / `free` without footers or magic numbers makes coalescing adjacent free blocks impossible (leading to fragmentation) and provides no defense against buffer overflows or double frees.
- **Why it's Wrong**: Fragmentation wastes memory; the heap may have enough total free memory but no single contiguous block large enough for a request. Without magic numbers, a simple buffer overflow can corrupt the `next` pointer or `size` field, causing the allocator to traverse into invalid memory and crash.
- **How to Fix**: Use the boundary tag structure with both header and footer. Include a magic number in both. On every `malloc` and `free`, verify the magic numbers. Always attempt to coalesce with both immediate neighbors when freeing a block.

⚠️ Pitfall: Not Marking Memory-Mapped I/O Regions as Uncacheable

- **Description**: Mapping device registers (like the VGA buffer at `0xB8000`) into the virtual address space with the default cacheable setting.
- **Why it's Wrong**: The CPU may cache reads and writes to these addresses. A write to the VGA buffer might be held in the cache and not reach the physical screen memory immediately, causing display artifacts or no output at all. Reads from device status registers might return stale, cached values instead of the current device state.
- **How to Fix**: When creating the PTE for a memory-mapped I/O region, set the `PTE_CACHE_DISABLE` and `PTE_WRITETHROUGH` flags (e.g., `PTE_PRESENT | PTE_WRITABLE | PTE_CACHE_DISABLE`). This ensures all accesses go directly to the bus.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Physical Allocator	Bitmap with next-fit search. Store bitmap in a reserved, statically-known memory region.	Multi-level bitmap or buddy system for faster allocation in larger memory.
Virtual Memory	Recursive page tables for simplicity. Identity map the first 4MB for bootstrapping.	Per-process page directories with a shared kernel mapping. Lazy allocation for user space.
Kernel Heap	Implicit free list with boundary tags and first-fit allocation. Coalesce on free.	Segregated free lists or slab allocator for common object sizes (like PCBs).

B. Recommended File/Module Structure

```
project-root/
  boot/                # Bootloader code (from Milestone 1)
  kernel/
    arch/i386/         # Architecture-specific code
      paging.c         # Page table setup, mapping functions
      pmm.c            # Physical Memory Manager (bitmap)
    kernel/            # Core kernel
      heap.c           # Kernel heap allocator (malloc/free)
      kprintf.c        # Formatted output (needs heap for `%s`?)
    include/           # Header files
      kernel/
        pmm.h          # Declares pmm_state, pmm_init, pmm_alloc_frame
        paging.h       # Declares vmm_init, map_page, recursive mapping macros
        heap.h         # Declares heap_init, kmalloc, kfree
        types.h        # Common types (size_t, uint32_t, etc.)
    linker.ld          # Linker script placing .text at 0x100000 but using high addresses
```

C. Infrastructure Starter Code

`include/kernel/paging.h` (Partial)

```c
#ifndef KERNEL_PAGING_H

#define KERNEL_PAGING_H


#include <stdint.h>

#include <stdbool.h>


typedef uint32_t page_table_entry_t;


// Page table entry flags (from Naming Conventions)

#define PTE_PRESENT         (1 << 0)

#define PTE_WRITABLE        (1 << 1)

#define PTE_USER            (1 << 2)

#define PTE_WRITETHROUGH    (1 << 3)

#define PTE_CACHE_DISABLE   (1 << 4)

#define PTE_ACCESSED        (1 << 5)

#define PTE_DIRTY           (1 << 6)

#define PTE_GLOBAL          (1 << 8)

#define PTE_FRAME_MASK      0xFFFFF000


// Recursive mapping constants for a 3GB/1GB split

#define PAGE_DIRECTORY_INDEX(x) (((x) >> 22) & 0x3FF)

#define PAGE_TABLE_INDEX(x) (((x) >> 12) & 0x3FF)

#define PAGE_GET_PHYS_ADDR(x) ((x) & ~0xFFF)


// Virtual addresses for recursive access

#define RECURSIVE_PD_BASE   0xFFC00000

#define RECURSIVE_PT_BASE   0xFFF00000 // Alternative: single-page mapping


// Convert a PDE/PTE index to its virtual address via recursive mapping

#define PDE_VIRT_ADDR(index) ((page_table_entry_t*)(RECURSIVE_PD_BASE + (index) * 4))

#define PTE_VIRT_ADDR(dir_idx, tbl_idx) ((page_table_entry_t*)(RECURSIVE_PD_BASE + 0x400 * (dir_idx) + (tbl_idx) * 4))


// Function declarations

void vmm_init(void);

void map_page(uint32_t virt_addr, uint32_t phys_addr, uint32_t flags);

void unmap_page(uint32_t virt_addr);

uint32_t get_phys_addr(uint32_t virt_addr);

void flush_tlb_single(uint32_t virt_addr);


#endif // KERNEL_PAGING_H
```

`kernel/arch/i386/pmm.c` **(Starter)**

```c
#include <kernel/pmm.h>

#include <kernel/kprintf.h>

#include <kernel/panic.h>

#include <stddef.h>

#include <stdint.h>


// Global PMM state

struct pmm_state pmm_state;


void pmm_init(struct multiboot_info *mboot_info) {

    // TODO 1: Validate the multiboot info structure has a memory map (check flags & MB_MEMINFO)

    // TODO 2: Iterate through the memory map entries (mboot_info->mmap_addr, mboot_info->mmap_length)

    // TODO 3: Find the highest usable physical address to calculate total_frames

    // TODO 4: Calculate the size (in bytes) needed for the bitmap: (total_frames + 7) / 8

    // TODO 5: Find a suitable, aligned, contiguous region in the memory map to store the bitmap (preferably in "available" RAM).

    // TODO 6: Reserve that region (mark frames as used). Set pmm_state.bitmap to the virtual address of this region.

    // TODO 7: Initialize the entire bitmap to 0xFF (all frames marked used initially).

    // TODO 8: Re-iterate the memory map. For each "available" type region, mark the corresponding frames as free (0) in the bitmap.

    // TODO 9: Mark the frames occupied by the kernel image, multiboot modules, and the bitmap itself as used (1).

    // TODO 10: Count the number of free frames and set pmm_state.free_frames.

    // TODO 11: Set pmm_state.last_alloc_index to 0.

    kprintf("PMM initialized: %d frames total, %d free.\n", pmm_state.total_frames, pmm_state.free_frames);

}


void* pmm_alloc_frame(void) {

    // TODO 1: Check if free_frames == 0. If so, return NULL (or panic).

    // TODO 2: Starting from last_alloc_index, search the bitmap for a free frame (bit == 0).

    // TODO 3: Use a nested loop: outer over bitmap words, inner over bits in a word.

    // TODO 4: When a free bit is found, set it to 1.

    // TODO 5: Decrement free_frames.

    // TODO 6: Update last_alloc_index to the index of the allocated frame (or the next one).

    // TODO 7: Calculate the physical address: frame_index * 4096.

    // TODO 8: Return the physical address as a void pointer.

    // TODO 9: If you reach the end of the bitmap, wrap around to the beginning.

    return (void*)0; // Placeholder

}


void pmm_free_frame(void *phys_addr) {

    // TODO 1: Calculate the frame index from the physical address: ((uint32_t)phys_addr) / 4096.

    // TODO 2: Ensure the frame index is valid (less than total_frames).

    // TODO 3: Ensure the frame is currently marked as used (bit == 1). If not, panic (double free).

    // TODO 4: Clear the bit to 0.
```

```
    // TODO 5: Increment free_frames.

}
```

D. Core Logic Skeleton Code

`kernel/arch/i386/paging.c` (Core Logic Skeleton)

```c
#include <kernel/paging.h>

#include <kernel/pmm.h>

#include <stdint.h>

#include <stdbool.h>


// The kernel's page directory (must be 4KB-aligned)

static page_table_entry_t kernel_page_directory[1024] __attribute__((aligned(4096)));


void vmm_init(void) {

    // TODO 1: Allocate a physical frame for the kernel page directory (or use a statically allocated one, as above).

    // TODO 2: Clear the entire page directory (set all entries to 0).

    // TODO 3: Set up identity mapping for the first 4MB (or the kernel's load size). This allows code to continue running after paging is enabled.

    //         - Create a page table for the first directory entry (index 0).

    //         - Map virtual addresses 0x0..0x3FFFFF to the same physical addresses.

    // TODO 4: Set up the higher-half kernel mapping (e.g., virtual 0xC0000000 maps to physical 0x100000).

    //         - Use directory index corresponding to 0xC0000000 (which is 0x300).

    //         - Create a page table for that directory entry.

    //         - Map the kernel's virtual addresses to their physical locations.

    // TODO 5: Set up recursive mapping in the last directory entry (index 1023).

    //         - Set PDE[1023] to the PHYSICAL address of kernel_page_directory, with flags PRESENT | WRITABLE.

    // TODO 6: Load the physical address of kernel_page_directory into CR3.

    // TODO 7: Enable paging by setting the PG bit in CR0 (use assembly: 'mov %eax, cr0; or $0x80000000, %eax; mov %eax, cr0').

    // TODO 8: After paging is enabled, you can now use the recursive mapping virtual addresses (0xFFC00000) to modify page tables.

    // TODO 9: Optionally, remove the identity mapping for the first 4MB to free up the page table and reclaim that virtual address range.

}


void map_page(uint32_t virt_addr, uint32_t phys_addr, uint32_t flags) {

    // TODO 1: Calculate the directory index and table index from virt_addr.

    // TODO 2: Use PDE_VIRT_ADDR(dir_index) to get a pointer to the PDE.

    // TODO 3: Check if the PDE is present. If not:

    //         a. Allocate a new physical frame for a page table using pmm_alloc_frame.

    //         b. Zero out the new page table.

    //         c. Set the PDE to point to the new table's physical address, with flags PRESENT | WRITABLE | USER (if needed).

    // TODO 4: Use PTE_VIRT_ADDR(dir_index, table_index) to get a pointer to the PTE.

    // TODO 5: Set the PTE: (*pte) = (phys_addr & PTE_FRAME_MASK) | (flags & ~PTE_FRAME_MASK).

    // TODO 6: Call flush_tlb_single(virt_addr) to invalidate the old TLB entry.

}
```

`kernel/heap.c` (Core Logic Skeleton)

```c
#include <kernel/heap.h>

#include <kernel/pmm.h>

#include <stddef.h>

#include <stdint.h>

#include <stdbool.h>


#define HEAP_MAGIC 0xDEADBEEF

#define MIN_BLOCK_SIZE 16 // Smallest allocation size (including metadata)


struct heap_block_header {

    uint32_t magic;

    size_t size;

    bool is_free;

    struct heap_block_header *next;

};


struct heap_block_footer {

    uint32_t magic;

    struct heap_block_header *header;

};


static struct heap_block_header *heap_start = NULL;


void heap_init(void) {

    // TODO 1: Allocate the initial heap frame (4KB) using pmm_alloc_frame.

    // TODO 2: Set heap_start to the virtual address of the allocated frame.

    // TODO 3: Create the initial free block covering the entire frame minus metadata.

    //         - Calculate user data size: 4096 - sizeof(header) - sizeof(footer).

    //         - Set header: magic=HEAP_MAGIC, size=user_data_size, is_free=true, next=NULL.

    //         - Set footer at the end of the block with magic and pointer back to header.

    kprintf("Heap initialized at %p\n", heap_start);

}


void* kmalloc(size_t size) {

    // TODO 1: Align the requested size to 8 bytes (or another alignment) to prevent misaligned access.

    // TODO 2: Traverse the linked list starting at heap_start.

    // TODO 3: For each block that is free (is_free == true) and has size >= requested_size, stop (first-fit).

    // TODO 4: If no suitable block found, call heap_expand() to add more memory and retry (or return NULL).

    // TODO 5: If the found block is large enough to split (see algorithm step 2 above), split it into two blocks.

    // TODO 6: Mark the chosen block as not free (is_free = false).

    // TODO 7: Return a pointer to the user data area (header + 1).

    return NULL;

}
```

```
void kfree(void *ptr) {

    if (!ptr) return;

    // TODO 1: Get the block header from the user pointer: header = (struct heap_block_header*)((char*)ptr - sizeof(struct heap_block_header));

    // TODO 2: Verify the header's magic number equals HEAP_MAGIC. If not, panic("heap corruption").

    // TODO 3: Mark the block as free (is_free = true).

    // TODO 4: Coalesce with next block if free:

    //         a. Calculate address of next block: next = (struct heap_block_header*)((char*)header + sizeof(header) + header->size + sizeof(footer));

    //         b. Check if next is within heap bounds and next->magic == HEAP_MAGIC and next->is_free.

    //         c. If so, merge: increase current header's size by (sizeof(next header) + next->size + sizeof(next footer)).

    //         d. Update the new footer at the end of the merged block.

    //         e. Adjust linked list pointers: current->next = next->next.

    // TODO 5: Coalesce with previous block if free (using footer):

    //         a. Find previous block's footer: prev_footer = (struct heap_block_footer*)((char*)header - sizeof(struct heap_block_footer));

    //         b. Check if prev_footer->magic == HEAP_MAGIC.

    //         c. Get previous header: prev_header = prev_footer->header.

    //         d. If prev_header->is_free, merge previous into current: prev_header->size += (sizeof(current header) + current->size + sizeof(current footer)).

    //         e. Update footer at the end of the merged block (which is now the old current footer).

    //         f. Adjust linked list pointers: prev_header->next = current->next.

}
```

E. Language-Specific Hints (C)

- **Bit Manipulation**: Use `uint32_t` for bitmap words. To find the first free bit in a word `w`, you can use `__builtin_ffs(~w)` on GCC/Clang to get the 1-based index of the first `1` bit in the complement. For portability, implement a loop or a small lookup table.
- **Alignment**: Use `__attribute__((aligned(4096)))` or `_Alignas(4096)` to ensure page directories and tables are page-aligned. The linker script must also enforce this.
- **Volatile for MMIO**: When accessing memory-mapped I/O regions through their virtual addresses, declare pointers as `volatile` (e.g., `volatile uint16_t* vga_buffer = (volatile uint16_t*)0xC00B8000;`) to prevent the compiler from optimizing away or reordering accesses.
- **Inline Assembly for TLB Flush**: Use `asm volatile("invlpg (%0)" : : "r" (addr) : "memory");` for a single TLB flush. To reload CR3 (full flush), `asm volatile("mov %0, %%cr3" : : "r" (pd_phys_addr));`.

F. Milestone Checkpoint (Memory Management)

After implementing the PMM, paging, and heap:

1. **Boot Test**: Boot the kernel in QEMU. It should reach `kmain` and print the PMM initialization message (e.g., "PMM initialized: 32768 frames total, 32000 free.").
2. **Page Table Test**: In `kmain`, after `vmm_init`, write a test that maps a new virtual page (e.g., `0xD0000000`) to a newly allocated physical frame, writes a known pattern to that virtual address, then reads it back via the recursive mapping to verify the PTE was set correctly. Use `kprintf` to report success.
3. **Heap Test**: Call `heap_init`. Then perform a sequence of allocations and frees:

```c
    void *a = kmalloc(100);

    void *b = kmalloc(200);

    kfree(a);

    void *c = kmalloc(150); // Should reuse a's block (possibly split)

    void *d = kmalloc(5000); // Should trigger heap expansion
```

`Print the pointers returned. Verify that `c` is within the same region as `a` (indicating reuse). Check that no panic occurs.`

4. **Expected Output**: The kernel should boot, show PMM stats, successfully complete the page mapping test, and allocate/free heap blocks without crashing. The screen should display relevant success messages.
5. **Signs of Trouble**:
   - **Triple fault immediately after enabling paging**: Likely an error in the initial page table setup (e.g., identity map missing or wrong). Use QEMU's `-d int` flag to log interrupts/exceptions.
   - **Heap allocations return NULL or corrupt data**: Check the heap initialization size calculation and the block splitting/coalescing logic. Add many `kprintf` statements in `kmalloc` / `kfree` to trace block states.
   - **Writing to a mapped address causes a page fault**: Forgot to set `PTE_WRITABLE` flag, or TLB not flushed after mapping. Double-check flags and call `flush_tlb_single`.

5.4 Component: Process Management & Scheduler (Milestone 4)

> **Milestone(s):** Milestone 4: Process Management

This component completes the transformation of our kernel from a simple, reactive program into a true, multi-tasking operating system. It introduces the fundamental abstraction of a **process**—an isolated instance of a program in execution—and the mechanisms to create, manage, and switch between these processes. The core challenge is to orchestrate the illusion of concurrent execution on a single CPU while maintaining strict isolation and control, all while efficiently managing the processor's time and state.

Mental Model: The Chess Tournament Director

Imagine you are the director of a large, simultaneous chess tournament. You have many players (processes), each seated at their own board (their isolated memory and state), all competing for the attention of a single, master chess clock (the CPU). Your job is to ensure fair play, manage time, and handle any special requests.

- **The Player Dossier (PCB):** For each player, you maintain a detailed dossier—the **Process Control Block (PCB)**. This dossier contains everything about that player's current game: the position of every piece (register state), a photo of the board (memory map/page tables), how much time they've used, whether they are currently thinking, waiting for an opponent's move, or have finished their game (process state). Without this dossier, you could not pause one player's game and later resume it exactly where it left off.
- **The Master Clock and Round Robin (Scheduler):** The tournament uses a **round-robin** format. Each player gets exactly 5 minutes on the clock (a time slice) before you must interrupt them and move the clock to the next player in line. You, as the director, are the **scheduler**. You maintain a list (the ready queue) of all players who are ready and waiting for their turn. When a player's time is up (triggered by the timer interrupt), you note down their exact board state in their dossier, look at your list to choose the next player, and then set the clock in front of that player, handing them their dossier so they can resume their game from the saved position. This rapid, fair switching creates the illusion that all games are progressing simultaneously.
- **Enforcing the Rules (Isolation & System Calls):** Players are not allowed to directly touch the master clock or another player's board. If a player needs more time, wants to report a result, or needs a new board (makes a **system call**), they must raise their hand and summon you, the director in charge (the kernel). You then pause their game, evaluate their request according to the tournament rules (kernel system call handler), perform the action on their behalf (e.g., allocating memory), and then return them to their game. This mechanism protects the integrity of the tournament (system stability) and ensures no player can cheat by interfering with another.

This model captures the essence of process management: maintaining detailed per-process state (PCB), making scheduling decisions, performing the mechanical work of switching contexts, and providing a controlled interface (system calls) for processes to request kernel services.

ADRs: Process State Model, Scheduler Algorithm

Decision: Simple Five-State Process Lifecycle Model

- **Context:** The kernel must track the status of each process to make correct scheduling decisions (e.g., don't schedule a process waiting for keyboard input). We need a finite state machine that is simple enough to implement and reason about for an educational kernel, yet captures the essential lifecycle stages.
- **Options Considered:**
  1. **Three-State (Ready, Running, Blocked):** The most minimal model. "New" and "Terminated" are considered transient.
  2. **Five-State (New, Ready, Running, Blocked, Terminated):** Explicit states for creation and termination, simplifying process table management.
  3. **Seven-State (with Suspend states):** Adds "Ready/Suspend" and "Blocked/Suspend" to model processes swapped out to disk, which is beyond our scope.
- **Decision:** Adopt the **Five-State Model**.
- **Rationale:** The five-state model provides clear, distinct phases for process creation (`PROC_STATE_NEW`) and cleanup (`PROC_STATE_TERMINATED`). This separation makes the logic in `process_create` and the scheduler's reaping of dead processes more explicit and less error-prone than overloading the `READY` or `BLOCKED` states. It matches the level of complexity appropriate for our educational goals.
- **Consequences:** We must define and handle transitions between all five states. The process table will contain processes in the `TERMINATED` state until their resources (PCB, memory) are reclaimed by a parent process or an init process (which we simplify to immediate cleanup in our first iteration).

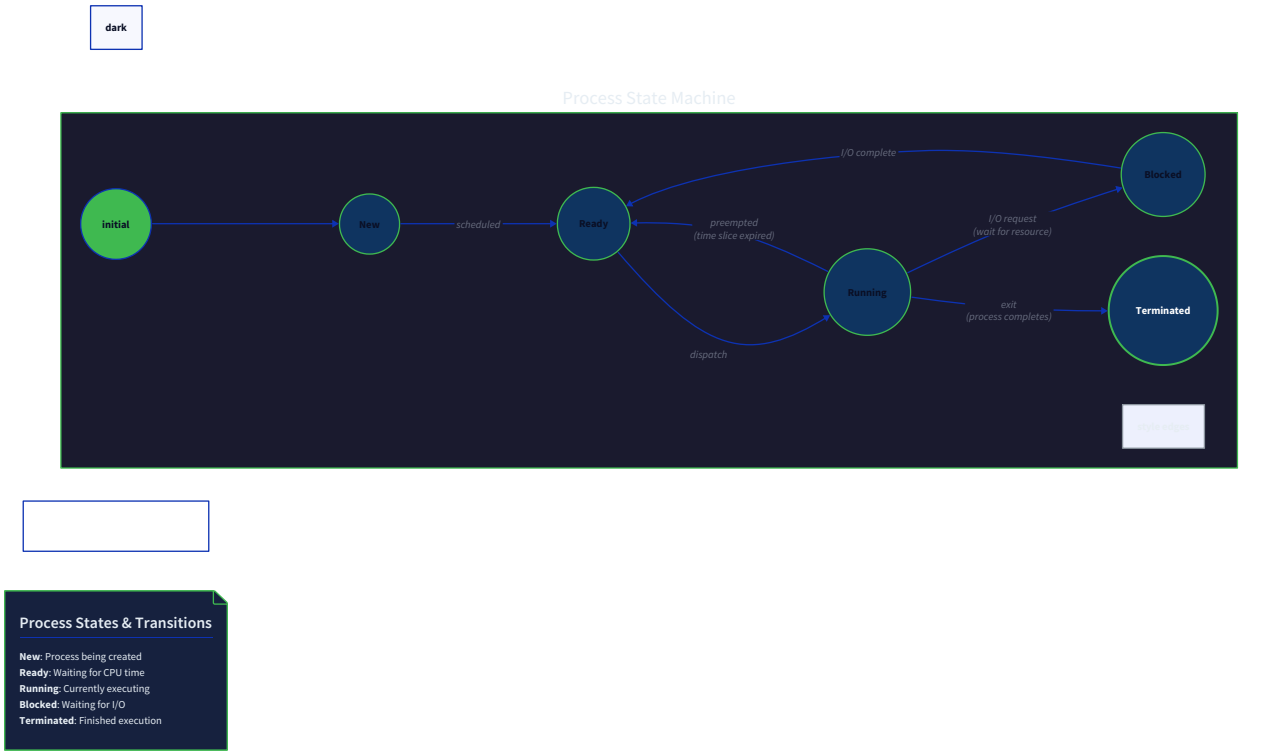
Option	Pros	Cons	Chosen?
Three-State	Extremely simple, fewer transitions to implement.	Blurs creation/termination logic, making cleanup and initialization code harder to structure.	No
Five-State	Clear separation of concerns, aligns well with typical OS textbooks, easy to reason about.	Slightly more states and transitions to manage.	**Yes**
Seven-State	Models real-world swapping for memory pressure.	Significant extra complexity (disk I/O, swap manager) far beyond our minimal scope.	No

Decision: Round-Robin Scheduling with Fixed Time Quantum

- **Context:** With multiple processes in the `READY` state, the scheduler must decide which one to run next. We need an algorithm that is simple to implement, understand, and debug, while providing basic fairness and the illusion of concurrency.
- **Options Considered:**
  1. **First-Come, First-Served (FCFS):** Run each process to completion. Very simple, but no fairness; a long process can monopolize the CPU.
  2. **Round-Robin:** Each process gets a fixed time slice (quantum); preempt if it exceeds it. Provides fairness and responsiveness.
  3. **Priority-Based:** Assign static/dynamic priorities; always run the highest-priority ready process. More realistic but requires defining a priority mechanism and handling starvation.
- **Decision:** Implement **Round-Robin (RR) Scheduling** with a fixed time quantum (e.g., 10-100ms as simulated by timer ticks).
- **Rationale:** Round-robin is the canonical introductory scheduling algorithm. It's trivial to implement with a simple linked list or queue of PCBs, and its behavior is easy to predict and observe (e.g., by printing which PID is running). The preemptive nature forces us to correctly implement timer interrupt handling and context switching, which are core learning objectives. Priority scheduling introduces additional complexity (priority inversion, aging) that distracts from the foundational concepts.
- **Consequences:** All processes are treated equally, which may not be optimal for mixes of I/O-bound and CPU-bound processes but is perfectly acceptable for our goals. The length of the time quantum is a tunable parameter that affects perceived responsiveness and switching overhead.

Option	Pros	Cons	Chosen?
FCFS	Trivial to implement (a single queue).	Causes "convoy effect", poor responsiveness, no preemption.	No
Round-Robin	Fair, simple, preemptive, excellent teaching tool.	All processes treated equally, quantum size is a tuning parameter.	**Yes**
Priority-Based	More realistic, can favor important processes.	Complex, can starve low-priority processes, requires more state.	No

The state machine defined by these decisions can be visualized as follows, showing the lifecycle of a process from birth to death:



Common Pitfalls: Context Switch and System Calls

Implementing process isolation and switching is notoriously tricky. Small mistakes lead to silent corruption, mysterious crashes, or total system locks. Here are the most common pitfalls and how to avoid them.

⚠️ Pitfall: Incomplete Register Save/Restore in Context Switch

- **Description:** The assembly context switch function (`switch_context`) saves/restores only the general-purpose registers (`eax` , `ebx` , etc.) but forgets critical control registers like the stack pointer (`esp`), instruction pointer (saved implicitly by `call` / `ret`), or segment registers.
- **Why it's Wrong:** If `esp` is not switched, the new process will use the old process's kernel stack, leading to catastrophic stack corruption. Segment registers must be consistent with the process's privilege level and memory view.
- **The Fix:** Meticulously define a `cpu_registers` structure that includes **all** registers that can vary between processes. In your switch assembly, follow a standard pattern: push all registers (including `ebp` , `esp`) onto the current kernel stack, save this stack pointer in the current PCB, load the next PCB's stack pointer, then pop all registers for the new process. Use the `iret` path for first-time switches to user mode.

⚠️ Pitfall: Incorrect Kernel Stack Management on First Switch

- **Description:** When switching to a user process for the very first time (from `NEW` to `RUNNING`), there is no previous context to "restore." Trying to use the standard context switch routine, which pops a saved state, will fail because the new process's kernel stack is empty.
- **Why it's Wrong:** The standard context switch assumes a process was previously running and had its state saved on its kernel stack. A new process has never run, so its stack doesn't contain a saved register frame.
- **The Fix:** Implement a separate initialization path for new processes. Set up the process's kernel stack to look **as if** it had been interrupted just before entering user mode. This involves pushing a fake interrupt stack frame (SS, ESP, EFLAGS, CS, EIP) and possibly general-purpose registers onto the process's kernel stack. Then, when the scheduler selects this process, the normal context switch restore and subsequent `iret` will "return" to the user-mode entry point correctly.

⚠️ Pitfall: Forgetting to Switch Page Directories

- **Description:** The context switch code switches all CPU registers but forgets to update the `cr3` register with the new process's page directory physical address.
- **Why it's Wrong:** The CPU's Memory Management Unit (MMU) uses `cr3` to locate the current page tables. If not updated, the new process will continue using the old process's virtual memory map, violating isolation and likely causing a page fault or silent data corruption.
- **The Fix:** Immediately after switching stacks in your context switch routine (but before popping the new process's registers), load the new process's `page_directory` physical address into `cr3` . Remember to also issue a `invlpg` instruction or reload `cr3` in a way that flushes the TLB for non-global mappings.

⚠️ Pitfall: Unsafe System Call Argument Validation

- **Description:** The system call handler directly uses pointer arguments passed from user space (`void* buf, size_t len`) to read or write kernel memory, without validating that the pointers point to valid, user-accessible memory.
- **Why it's Wrong:** A malicious or buggy user process can pass a pointer to kernel memory (`0xC0000000`), causing the kernel to overwrite its own data structures. This destroys security and stability.
- **The Fix: Never trust user input.** Before dereferencing any user-provided pointer, validate that the entire memory region it references (from `buf` to `buf+len-1`) lies within the user portion of the process's address space (typically below `0xC0000000`) and that the pages are present and user-accessible. This requires walking the process's page tables or using helper functions like `copy_from_user` and `copy_to_user` that perform the check and copy in one safe operation.

⚠️ **Pitfall: Not Saving Floating Point/SSE State**

- **Description:** The context switch ignores the FPU and SIMD registers (`ST0`-`ST7`, `XMM0`-`XMM7`, `MXCSR`).
- **Why it's Wrong:** If one process uses floating-point math and another later uses the FPU, the old state will be corrupted, leading to incorrect calculations. This is less critical in a minimal OS but is a realistic detail.
- **The Fix (Advanced):** Use `fxsave`/`fxrstor` instructions to save/restore the extended FPU state. For simplicity in an educational kernel, you can avoid the issue by setting the `TS` (Task Switched) bit in `cr0` and handling the `#NM` exception lazily, but a full save/restore is more predictable.

Implementation Guidance: PCB & Scheduler

This section provides the concrete code structure and skeletons to implement process management. The primary language is C, with critical portions in x86 Assembly.

A. Technology Recommendations Table

Component	Simple Option (Recommended)	Advanced Option (For Later)
Process State Tracking	Five-state enum in PCB, updated by scheduler/kernel.	Extended state with exit signals, resource usage statistics.
Scheduler Algorithm	Round-robin using a simple linked list of PCBs.	Multi-level Feedback Queue (MLFQ) or Completely Fair Scheduler (CFS) inspired.
System Call Interface	Single interrupt gate (`int 0x80`) with syscall number in `eax`.	`sysenter`/`sysexit` or `syscall`/`sysret` for faster transitions.
Inter-Process Communication	None initially.	Shared memory, pipes, or message queues.

B. Recommended File/Module Structure

Place process management code in a dedicated directory to separate concerns from the core kernel and drivers.

```
src/
├── kernel/
│   ├── kernel.c          # Contains `kmain`, high-level initialization
│   ├── panic.c           # `panic` function
│   └── kprintf.c         # Formatted kernel output
├── arch/i386/
│   ├── boot.asm          # Bootloader and kernel entry
│   ├── gdt.asm / gdt.c   # GDT setup
│   ├── idt.asm / idt.c   # IDT setup
│   ├── isr.asm           # ISR assembly stubs
│   ├── irq.c             # IRQ handlers (PIC, timer, keyboard)
│   ├── paging.c          # Page table manipulation
│   └── context_switch.asm # The core `switch_context` assembly routine
├── mm/
│   ├── pmm.c             # Physical Memory Manager
│   ├── vmm.c             # Virtual Memory Manager
│   └── heap.c            # Kernel heap allocator (`kmalloc`, `kfree`)
├── proc/                 # NEW: Process Management Module
│   ├── proc.c            # Process lifecycle: `process_create`, `process_destroy`
│   ├── sched.c           # Scheduler: `scheduler_init`, `scheduler_get_next_process`, `schedule`
│   ├── syscall.c         # System call dispatcher and handlers
│   └── pcb.h             # Definition of `struct pcb`, `enum process_state_t`, etc.
└── drivers/
    ├── vga.c             # VGA text mode driver
    └── keyboard.c        # Keyboard driver
```

C. Infrastructure Starter Code (PCB Structure and System Call Table)

These are foundational definitions that should be stable. Place them in `proc/pcb.h`.

```c
// proc/pcb.h

#ifndef KERNEL_PCB_H

#define KERNEL_PCB_H


#include <stdint.h>

#include <stdbool.h>

#include <stddef.h>


// Process States (Five-State Model)

typedef enum {

    PROC_STATE_NEW = 0,      // Being created, not yet ready

    PROC_STATE_READY,        // Ready to run, waiting for CPU

    PROC_STATE_RUNNING,      // Currently executing on CPU

    PROC_STATE_BLOCKED,      // Waiting for event (I/O, signal)

    PROC_STATE_TERMINATED    // Finished, awaiting cleanup

} process_state_t;


// CPU Register State - Must match order pushed/popped in context_switch.asm

typedef struct cpu_registers {

    // General Purpose Registers (pushed by common ISR stub)

    uint32_t eax, ebx, ecx, edx, esi, edi;


    // Stack Frame Registers

    uint32_t ebp;

    uint32_t esp; // Kernel stack pointer at time of switch


    // Control Registers (pushed by CPU on interrupt)

    uint32_t eip;

    uint32_t eflags;

    uint32_t cs;

    uint32_t ds;

    uint32_t es;

    uint32_t fs;

    uint32_t gs;

    uint32_t ss; // User stack segment (if switching to/from user mode)

} __attribute__((packed)) cpu_registers_t;


// Process Control Block (PCB) - The "Player Dossier"

typedef struct pcb {

    // Process Identification

    int pid;

    int parent_pid;

    char name[16];
```

```
    // State and Execution
    process_state_t state;
    cpu_registers_t regs;          // Saved CPU context when not running
    void* kernel_stack;            // Current top of kernel stack
    void* kernel_stack_base;       // Base (bottom) of kernel stack allocation

    // Memory Management
    uint32_t* page_directory;      // Physical address of the page directory
    void* brk;                     // Current heap break (for sbrk syscall)
    void* image_start;             // Start of process image in memory
    size_t image_size;             // Size of process image

    // Scheduling and Accounting
    uint64_t time_used;            // Ticks of CPU time consumed
    uint8_t priority;              // Simple priority (for future use)

    // Waiting/Blocking
    int waiting_for;               // PID or resource being waited on

    // Linked List for Scheduler Queues
    struct pcb* next;
} pcb_t;

// System Call Numbers
#define SYS_EXIT    0
#define SYS_WRITE   1
#define SYS_READ    2
#define SYS_FORK    3 // For future extension
#define SYS_GETPID  4

#endif // KERNEL_PCB_H
```

```c
// proc/syscall.c - System Call Dispatch Table Skeleton

#include "proc/pcb.h"

#include "drivers/vga.h"  // For console output

#include <stddef.h>


// System Call Handler Function Prototype

typedef int (*syscall_handler_t)(uint32_t, uint32_t, uint32_t, uint32_t, uint32_t);


// Forward declarations of individual syscall handlers

static int sys_exit(int exit_code);

static int sys_write(int fd, const char* buf, size_t count);

static int sys_read(int fd, char* buf, size_t count);

static int sys_getpid(void);


// System Call Dispatch Table

static const syscall_handler_t syscall_table[] = {

    [SYS_EXIT]   = (syscall_handler_t)sys_exit,

    [SYS_WRITE]  = (syscall_handler_t)sys_write,

    [SYS_READ]   = (syscall_handler_t)sys_read,

    [SYS_GETPID] = (syscall_handler_t)sys_getpid,

};

static const size_t MAX_SYSCALL = sizeof(syscall_table)/sizeof(syscall_table[0]) - 1;


// Central System Call Dispatcher (called from isr_handler for int 0x80)

int syscall_dispatcher(int syscall_num, uint32_t arg1, uint32_t arg2, uint32_t arg3, uint32_t arg4, uint32_t arg5) {

    if (syscall_num < 0 || syscall_num > MAX_SYSCALL || syscall_table[syscall_num] == NULL) {

        return -1; // Invalid syscall number

    }

    // TODO 6: Add safety checks for user-provided pointers in relevant syscalls.

    return syscall_table[syscall_num](arg1, arg2, arg3, arg4, arg5);

}


// Individual Syscall Handlers (stubs)

static int sys_exit(int exit_code) {

    // TODO 7: Mark current process as TERMINATED, possibly store exit_code.

    // TODO 8: Call scheduler to switch to another process immediately.

    return 0; // Never actually returns to caller

}


static int sys_write(int fd, const char* buf, size_t count) {

    // For now, only support writing to stdout (fd=1)

    if (fd != 1) {

        return -1;
```

```
    }
    // TODO 9: Validate `buf` points to user memory range (0x0 - 0xBFFFFFFF).
    // TODO 10: For each character in buf[0..count-1], call vga_putchar.
    // This is a blocking, synchronous write.
    return count; // Return number of bytes written
}

static int sys_read(int fd, char* buf, size_t count) {
    // TODO 11: For future extension (keyboard input).
    return -1; // Not implemented yet
}

static int sys_getpid(void) {
    // TODO 12: Return the `pid` of the currently running process (current_pcb->pid).
    return 0;
}
```

D. Core Logic Skeleton Code

Here are the key functions to implement, with TODOs mapping to the conceptual steps.

```c
// proc/proc.c - Process Lifecycle

#include "proc/pcb.h"

#include "mm/pmm.h"

#include "mm/vmm.h"

#include "mm/heap.h"

#include "panic.h"

#include <string.h>


static int next_pid = 1;

pcb_t* current_pcb = NULL; // Global pointer to the currently running process


pcb_t* process_create(const char *name, void *entry_point) {

    // TODO 1: Allocate a new PCB using kmalloc.

    // TODO 2: Allocate a kernel stack (e.g., 2 pages = 8KB) using kmalloc or pmm_alloc_frame.

    // TODO 3: Allocate a page directory for the process by copying the kernel portion of the current directory.

    // TODO 4: Initialize the PCB fields:

    //   - Assign a unique PID (next_pid++).

    //   - Set state to PROC_STATE_NEW.

    //   - Copy the process name.

    //   - Set kernel_stack_base and kernel_stack (point to top of stack area).

    //   - Store the page_directory physical address.

    // TODO 5: Set up the process's initial kernel stack to mimic an interrupt frame:

    //   - Push values for SS, USER_ESP, EFLAGS, CS, EIP (the entry_point).

    //   - This allows the first context switch to this process to "return" to user mode via iret.

    // TODO 6: Add the new PCB to the scheduler's ready queue.

    // TODO 7: Return the pointer to the new PCB.

    return NULL;

}


void process_destroy(int pid) {

    // TODO 8: Find the PCB with the given PID (likely in a terminated state).

    // TODO 9: Free all resources associated with the process:

    //   - Free user memory pages (walk page tables and free frames).

    //   - Free the page directory and page table structures themselves.

    //   - Free the kernel stack.

    //   - Free the PCB structure.

    // TODO 10: Remove the PCB from any scheduler queues.

}
```

```c
// proc/sched.c - Scheduler

#include "proc/pcb.h"

#include "arch/irq.h" // For timer tick

#include "panic.h"


static pcb_t* ready_queue = NULL;

static pcb_t* blocked_queue = NULL;


void scheduler_init(void) {

    // TODO 1: Initialize the ready_queue and blocked_queue to NULL.

    // TODO 2: Create the initial "kernel idle process" or set current_pcb to a dummy PCB.

    // TODO 3: Call timer_init (from irq.c) to start the timer interrupt for preemption.

}


// Internal helper: adds a PCB to the tail of the ready queue

static void enqueue_ready(pcb_t* pcb) {

    // TODO 4: If ready_queue is NULL, set it to pcb. Otherwise, traverse to the end and append.

    // Ensure pcb->next is set to NULL.

}


pcb_t* scheduler_get_next_process(void) {

    // Simple Round-Robin: pick the head of the ready queue

    if (ready_queue == NULL) {

        return NULL; // No processes ready (should have at least an idle task)

    }

    // TODO 5: Remove the head of the ready queue (to be returned).

    // TODO 6: If the process is still READY (not terminated), re-add it to the *tail* of the ready queue.

    // This implements the round-robin circular behavior.

    // TODO 7: Return the PCB that was at the head.

    return ready_queue;

}


// The main scheduling function, called from timer interrupt handler

void schedule(void) {

    pcb_t* next = scheduler_get_next_process();

    if (next == NULL || next == current_pcb) {

        return; // No switch needed

    }

    pcb_t* prev = current_pcb;

    current_pcb = next;

    current_pcb->state = PROC_STATE_RUNNING;


    // TODO 8: Perform the actual context switch.
```

```
    // This will call the assembly routine switch_context(&prev->regs, &next->regs);

    // The function does not return until this process is switched back in.

}
```

ASSEMBLY

```
; arch/i386/context_switch.asm - Core Assembly Routine
; C signature: void switch_context(cpu_registers_t** old_sp, cpu_registers_t* new_sp);
; Saves the current register state to the location pointed to by `old_sp`,
; then restores the state from `new_sp`.

global switch_context
switch_context:
    ; Function prologue
    push ebp
    mov ebp, esp

    ; Save the current (old) context pointer location (first argument) to ecx
    mov ecx, [ebp + 8]   ; ecx = &old_sp

    ; TODO 1: Save ALL current register state.
    ; Push registers in the exact order defined by `cpu_registers_t` in pcb.h.
    ; e.g., push eax, push ebx, ... push gs
    ; The current stack pointer (esp) after all pushes points to the saved state.

    ; TODO 2: Save the current stack pointer (the location of the saved state) into [ecx].
    ; This updates the old PCB's `regs` pointer.

    ; Now switch to the new stack and restore its state.
    mov eax, [ebp + 12]  ; eax = new_sp (pointer to new saved registers)

    ; TODO 3: Load the new stack pointer from the new context structure.
    ; The new_sp points to a `cpu_registers_t`, which is the *top* of the saved register stack.
    ; mov esp, eax

    ; TODO 4: Before popping registers, switch the page directory (cr3) if necessary.
    ; This might require passing the new page directory physical address as another argument.
    ; mov cr3, [eax + offset_of_page_directory_in_pcb] ; Example

    ; TODO 5: Restore the new process's register state by popping in reverse order.
    ; pop gs, pop fs, ... pop eax

    ; Function epilogue - we are now on the new process's kernel stack.
    mov esp, ebp
    pop ebp
    ret ; Returns into the new process's kernel context
```

E. Language-Specific Hints (C/Assembly)

- **Assembly-C Interface:** The `switch_context` function is called from C but implemented in assembly. Ensure the calling convention (`cdecl`) is respected: arguments are pushed right-to-left, caller cleans the stack. Preserve the registers `ebp` , `esi` , `edi` , `ebx` as required by the ABI.
- **Pack the Structures:** Use `__attribute__((packed))` for `cpu_registers_t` to prevent the compiler from inserting padding between fields, ensuring the assembly code's hardcoded offsets match.
- **Volatile in ASM:** Use `asm volatile` when writing inline assembly that has side effects (like loading `cr3`) to prevent the compiler from reordering or optimizing it away.
- **System Call Convention:** Adopt a simple convention: syscall number in `eax` , arguments in `ebx` , `ecx` , `edx` , `esi` , `edi` . The return value is placed in `eax` . Document this clearly.

F. Milestone Checkpoint After implementing this component, you should be able to:

1. **Create Processes:** In your `kmain` , after initializing the scheduler, create two dummy user processes with different entry points (simple C functions that loop and print their PID). Observe via `kprintf` that both process IDs are created successfully.
2. **Observe Scheduling:** The timer interrupt should fire periodically. You should see log messages showing the scheduler switching between the two processes (e.g., "Switching from PID 1 to PID 2"). This demonstrates preemptive multitasking.
3. **Test System Calls:** Write a simple test where a user process (a function you created) makes the `SYS_WRITE` system call (`int 0x80` with `eax=1`) to print "Hello from PID X!". You should see this message appear on the VGA screen, proving the user-to-kernel transition works.
4. **Verify Isolation (Advanced):** Attempt to have one process write to a memory location and have another read it. If isolation is correctly implemented via separate page directories, they should not see each other's changes.

Expected Output (Example VGA screen):

```
Kernel initialized. Scheduler started.
Created process PID 1: 'test1'
Created process PID 2: 'test2'
[Timer] Switching to PID 1
Hello from PID 1!
[Timer] Switching to PID 2
Hello from PID 2!
[Timer] Switching to PID 1
Hello from PID 1!
...
```

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Triple fault immediately after first context switch to user process.	Kernel stack for new process not set up correctly; `iret` tries to load invalid CS/EIP/EFLAGS.	Use Bochs debugger to single-step through `switch_context` and examine the stack contents before `iret`. Check the values of the pushed segment selectors and `eip`.	Ensure you push a valid user-mode CS (e.g., `0x18`
Processes run but corrupt each other's memory (e.g., variables change unexpectedly).	Page directory (`cr3`) not being switched during context switch, or kernel space not identically mapped in all processes.	Add debug prints showing the value of `cr3` before and after a switch. Check that kernel virtual addresses (e.g., `0xC0000000`) map to the same physical frame in both processes' page tables.	Update `cr3` in `switch_context`. Ensure your `vmm_map_page` function, when copying kernel mappings, uses the same physical frame for kernel virtual addresses.
System call works once then crashes on subsequent calls.	System call handler corrupts the user process's stack or saved register state.	Compare the `cpu_registers_t` saved on the process's kernel stack before and after the system call. Look for registers that changed unexpectedly.	Ensure your ISR stub and system call dispatcher save/restore **all** registers. The dispatcher must not clobber `ebx`, `ecx`, etc., that belong to the user.
Scheduler only runs the first process, never switches.	Timer interrupt handler not calling `schedule()`, or `schedule()` always returns the same process.	Verify the timer ISR is being triggered (add a tick counter print). Check the logic in `scheduler_get_next_process` – is it correctly moving the head process to the tail?	Implement the round-robin queue correctly. The timer ISR should call `schedule()` after handling the tick.

6. Interactions and Data Flow

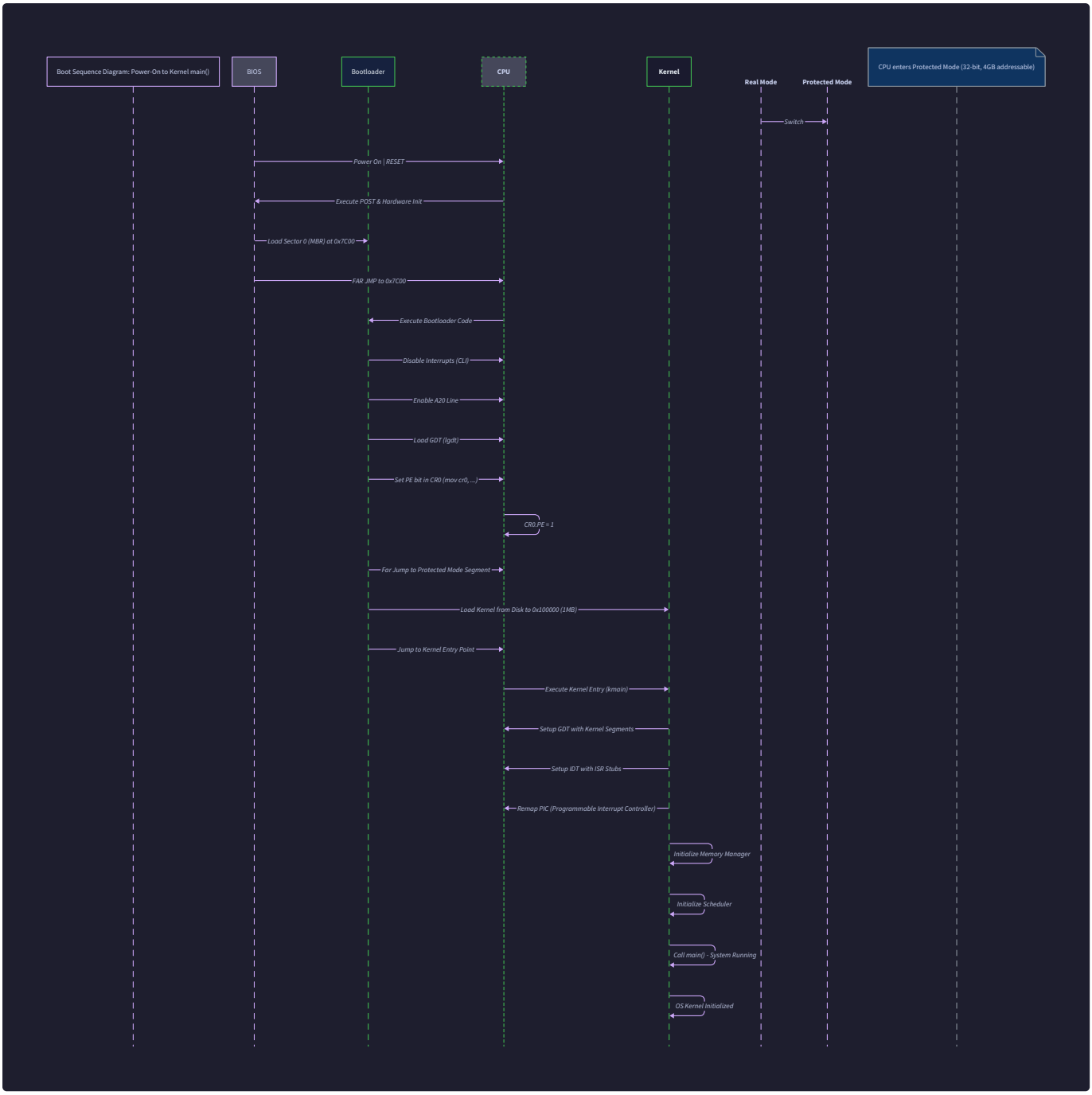
> **Milestone(s):** This section synthesizes the operation of components defined in Milestones 1, 2, 3, and 4. It traces the critical runtime workflows that span multiple subsystems, illustrating how the kernel transforms from a static binary into a dynamic, multitasking system.

Understanding a system's static structure—its data models and components—is only half the picture. The true essence of an operating system kernel emerges in its dynamic behavior: the orchestrated sequence of operations that occur when the machine powers on, when a key is pressed, or when a process yields the CPU. This section traces the flow of control and data for the kernel's most fundamental workflows. These narratives connect the architectural dots, showing how the `multiboot_header`, `pcb`, `idt_entry`, and `page_table_entry_t` structures come alive during execution. By walking through these concrete sequences, you will see the kernel's state machine in action, transforming hardware events into software abstractions.

Boot Sequence Flow

Mental Model: The Theater Opening Night. Imagine a grand theater's opening night. The **BIOS/UEFI** is the stagehand who flips the main power switch and performs a basic safety check of the building (POST). The **bootloader** (like GRUB) is the stage manager who ensures the lead actor—the kernel binary—is in the correct dressing room (memory address), that the stage lights (CPU mode) are set correctly, and who gives the "places everyone" call. Finally, the **kernel entry point** is the lead actor stepping onto the stage, clearing their throat (zeroing BSS), and delivering the opening monologue (`kmain`). This flow is the irreversible transition from inert hardware to a running, thinking software system.

The boot sequence is a carefully choreographed handoff of control from firmware to bootloader to kernel, accompanied by a fundamental shift in the CPU's operating paradigm. The following numbered steps detail this process, which is also visualized in the sequence diagram.



1. **Power-On and Firmware Initialization:** When the physical power is applied, the CPU begins executing code from a hardwired memory address in the system's ROM. The **BIOS** (or **UEFI**) firmware initializes critical hardware components: conducting the Power-On Self-Test (POST), enumerating devices, and building a basic system configuration table. Its final task is to locate a bootable device by reading the first sector (512 bytes) of each disk in a configured order, searching for the **boot signature** `0x55AA` at bytes 511-512.
2. **Bootloader Stage 1 (BIOS MBR):** Upon finding a valid boot signature, the BIOS loads the entire first sector—the Master Boot Record (MBR)—into memory at address `0x7C00` and jumps to it. This tiny 512-byte program is the *first-stage bootloader*. Its sole responsibility is to load a larger, more capable *second-stage bootloader* from disk into memory. Due to severe size constraints, it typically contains only basic disk I/O routines using BIOS interrupts (`int 0x13`).
3. **Bootloader Stage 2 (GRUB-like):** The second-stage bootloader, now free from the 512-byte limit, provides a rich interface: it can read files from a filesystem, parse a configuration file, and present a menu to the user. For our kernel, its critical job is to locate the kernel's ELF binary on disk, parse its program headers, and load the necessary segments (`.text` , `.data`) into memory at the addresses specified by the linker script. Crucially, it must also locate the `multiboot_header` structure within the kernel image to verify compatibility.
4. **Transition to Protected Mode:** Before handing control to the kernel, the bootloader must prepare the CPU environment. It switches the CPU from 16-bit real mode to **32-bit protected mode**. This involves:
    - Disabling interrupts (`cli`).

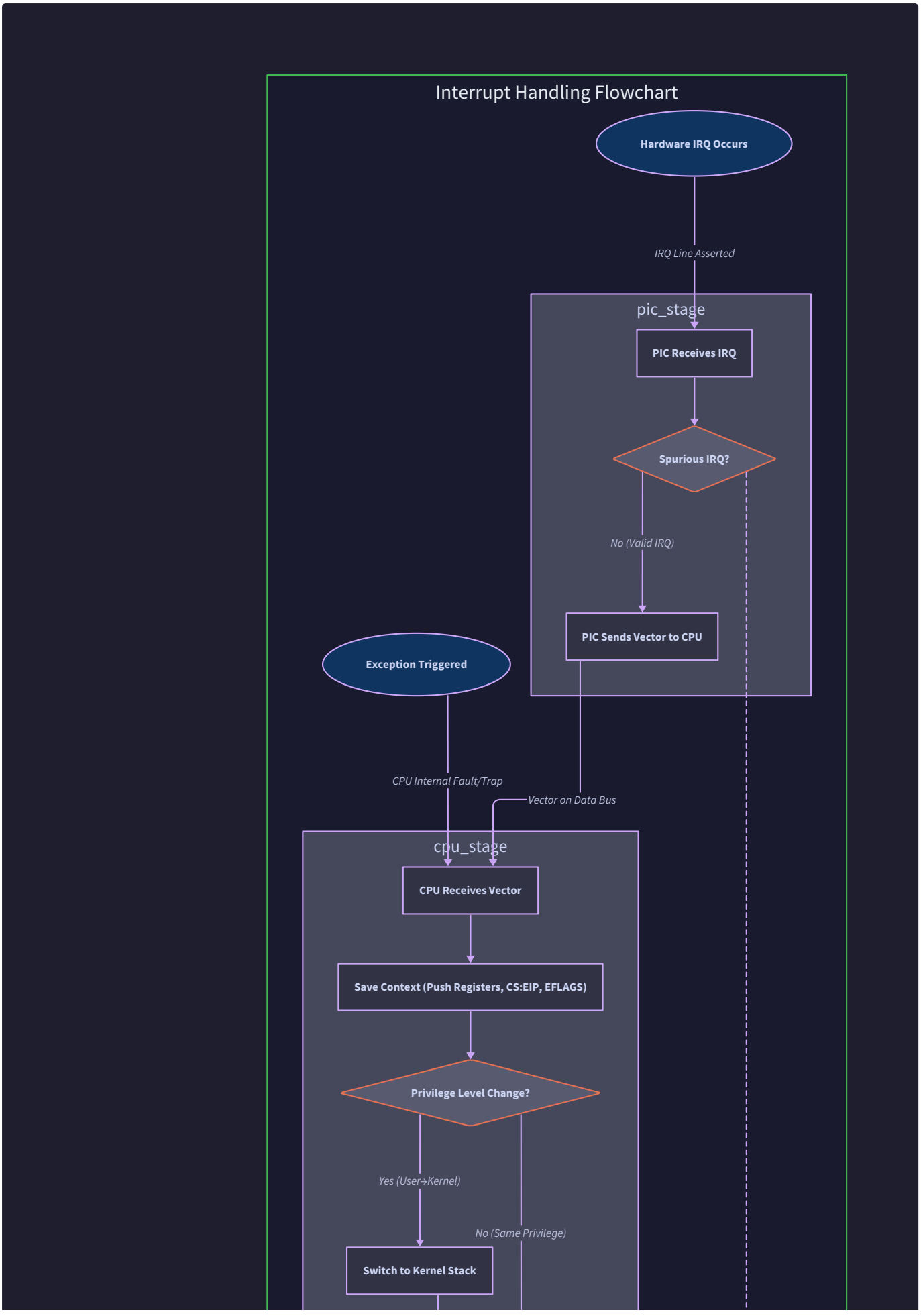
- Loading a **Global Descriptor Table (GDT)** that defines flat code and data segments for protected mode. The `gdtr` structure is filled and loaded using the `lgdt` instruction.
- Setting the Protection Enable (PE) bit in the `CR0` control register.
- Performing a far jump (`jmp <CODE_SEGMENT>:<protected_mode_entry>`) to flush the CPU's instruction pipeline and begin executing 32-bit code.

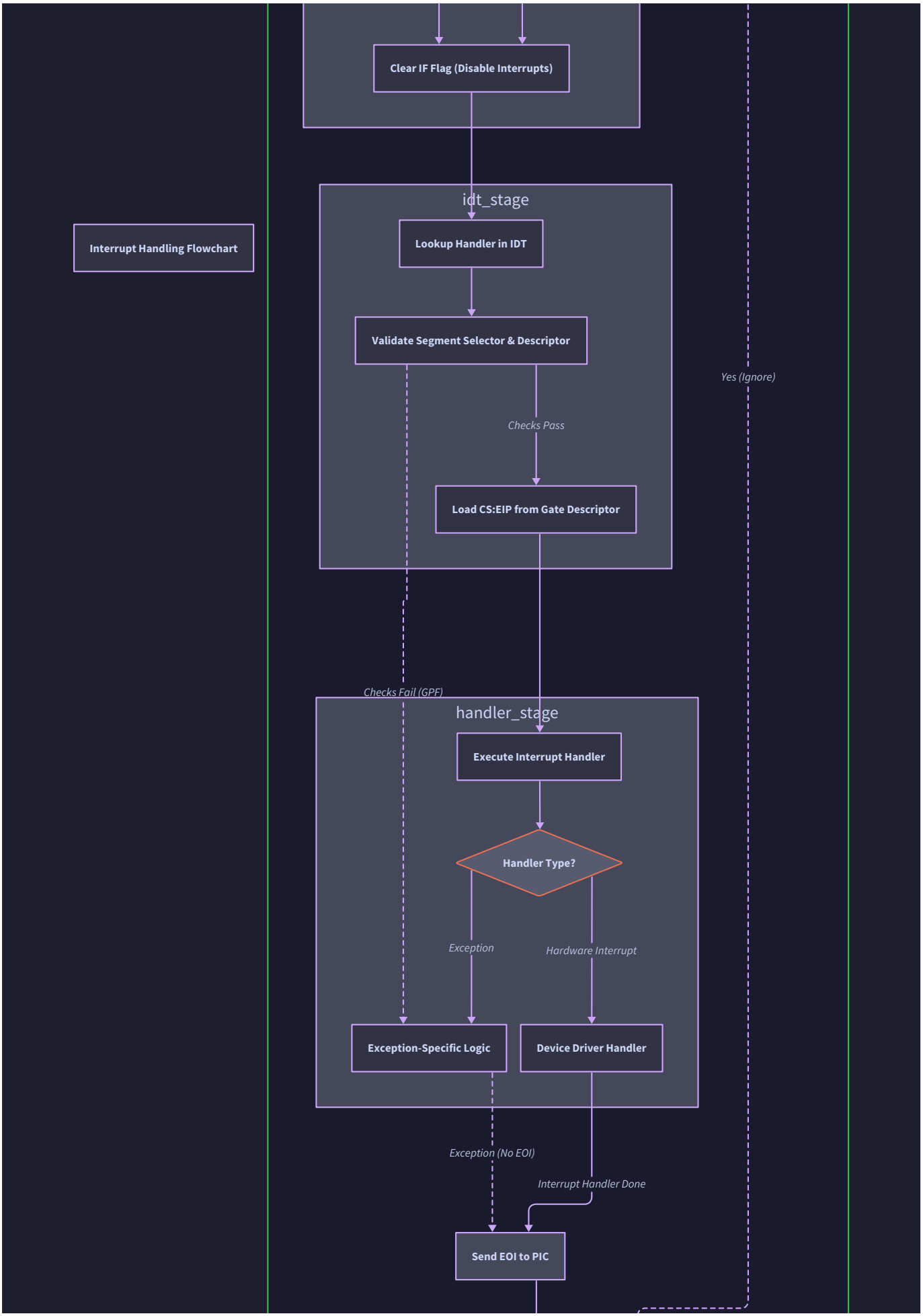
5. **Kernel Entry and `kmain` Invocation:** The bootloader's final act is to transfer control to the kernel's entry point. For a Multiboot-compliant kernel, it does so by calling (or jumping to) the address specified in the `multiboot_header` 's `entry_addr` field. Before the jump, it ensures the CPU registers are in a defined state: `EAX` must contain the magic value `MULTIBOOT_MAGIC` (`0x2BADB002`), and `EBX` must hold a physical pointer to the `multiboot_info` structure. The kernel's entry code (often written in assembly) immediately performs two vital tasks: a. **Clear the BSS section:** It iterates from `_bss_start` to `_bss_end` , setting all memory to zero. This ensures all uninitialized static/global variables start with a value of zero. b. **Set up the stack pointer:** It initializes the `ESP` register to point to the top of a pre-allocated kernel stack (e.g., `_kernel_stack + KERNEL_STACK_SIZE`). Finally, it calls the high-level `kmain(uint32_t mboot_magic, void* mboot_info)` function in C, passing the values from `EAX` and `EBX` .
6. **Kernel Early Initialization:** Inside `kmain` , the kernel takes over the "stage." Its first actions are to initialize core subsystems that everything else depends on:
   - `vga_init()` : Prepares the video buffer at `VGA_BUFFER_ADDR` for text output, allowing `kprintf` to work.
   - `gdt_init()` : May reload or adjust the GDT set up by the bootloader, ensuring it's precisely configured for the kernel's needs.
   - `pmm_init(mboot_info)` : Initializes the physical memory manager by parsing the memory map provided in the `multiboot_info` structure (accessed via the `multiboot_mmap_entry` chain). This builds the `pmm_state` bitmap of free and used physical frames. At this point, the kernel is alive, can print to the screen, and knows what physical memory it can use. The subsequent flows describe how it becomes interactive and capable of multitasking.

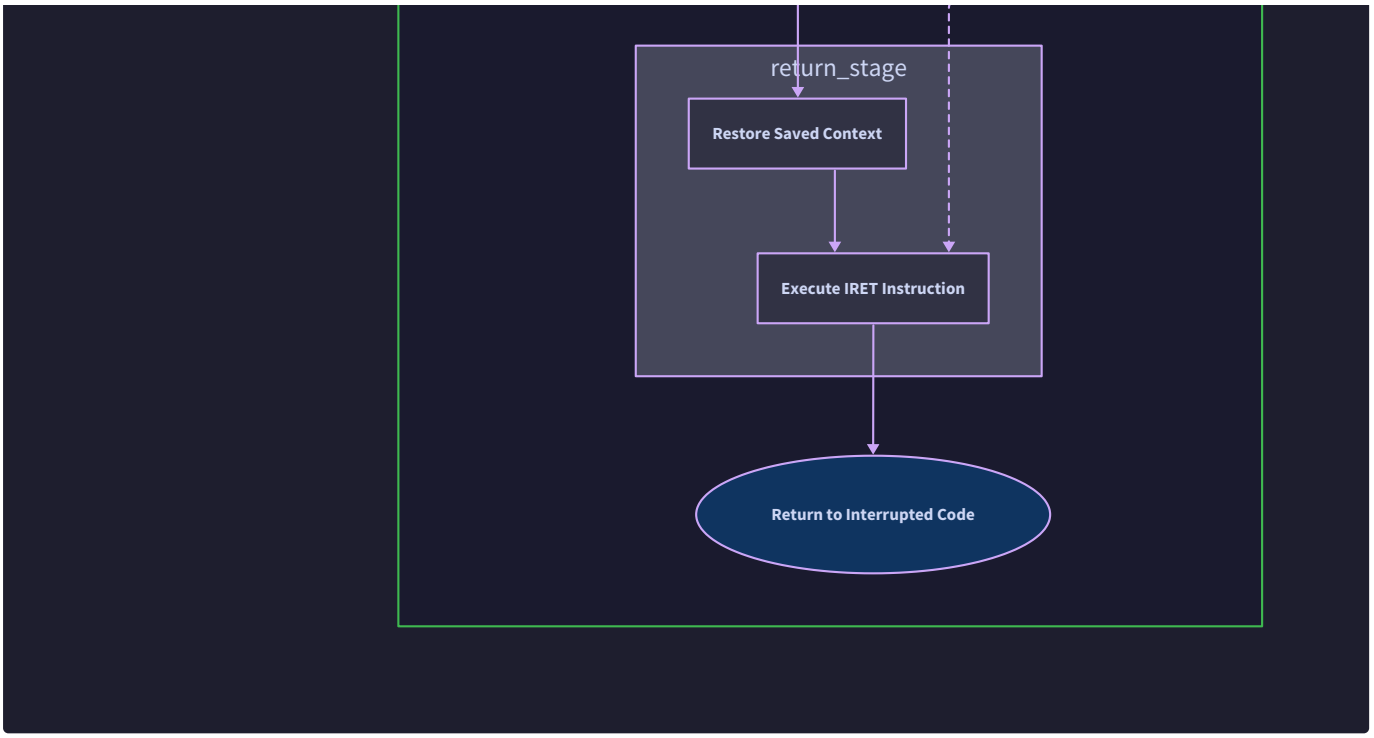
Interrupt Handling Flow

Mental Model: The Office Receptionist and Priority Intercom System. Imagine an office where employees (the CPU) are focused on their core work. A wall of intercom lines (**IRQ** lines 0-15) connects to various departments (hardware devices). A **receptionist (PIC/APIC)** monitors these lines. When a department calls (e.g., the keyboard department on line IRQ1), the receptionist checks if the call is allowed (not masked), decides its priority, and then buzzes the employee's desk with a specific code (**interrupt vector**, e.g., `0x21`). The employee immediately stops their work, notes exactly where they left off (saving registers on the **stack**), and handles the call by looking up the code in a procedure manual (**IDT**) to find the correct handler routine (**ISR**). After handling the call, they tell the receptionist they're done (**EOI**) and return to their original task, restoring their notes. This mechanism allows urgent, external events to get immediate attention without the CPU constantly polling.

Interrupts are the primary mechanism for hardware to communicate with the CPU asynchronously. The flow from a hardware signal to a software handler and back is a delicate dance involving the interrupt controller, CPU hardware, and kernel software. The following steps detail this flow, visualized in the accompanying flowchart.


Interrupt Handling Flowchart
Hardware IRQ Occurs
IRQ Line Asserted
pic_stage
PIC Receives IRQ
Spurious IRQ?
No (Valid IRQ)
PIC Sends Vector to CPU
Exception Triggered
CPU Internal Fault/Trap
Vector on Data Bus
cpu_stage
CPU Receives Vector
Save Context (Push Registers, CS:EIP, EFLAGS)
Privilege Level Change?
Yes (User→Kernel)
No (Same Privilege)
Switch to Kernel Stack


Interrupt Handling Flowchart
Clear IF Flag (Disable Interrupts)
idt_stage
Lookup Handler in IDT
Validate Segment Selector & Descriptor
Checks Pass
Load CS:EIP from Gate Descriptor
Yes (Ignore)
Checks Fail (GPF)
handler_stage
Execute Interrupt Handler
Handler Type?
Exception
Hardware Interrupt
Exception-Specific Logic
Device Driver Handler
Exception (No EOI)
Interrupt Handler Done
Send EOI to PIC



1. **Hardware Device Signals an Interrupt:** A hardware device (e.g., keyboard controller, timer chip) needs attention. It raises an electrical signal on its dedicated **Interrupt Request (IRQ)** line. For the classic PC architecture, this line is connected to one of two cascaded **Programmable Interrupt Controllers (PICs)**.
2. **PIC Evaluates and Signals the CPU:** The PIC receives the IRQ signal. It checks its internal mask register to see if that specific IRQ line is enabled. If unmasked, the PIC determines the highest-priority pending interrupt. It then asserts the **INTR** (interrupt request) pin on the CPU. The PIC also places the corresponding **interrupt vector number** (which the kernel configured during `pic_remap()`, e.g., IRQ0 -> vector `0x20`) onto the data bus.
3. **CPU Acknowledges and Looks Up Handler:** The CPU, after completing its current instruction, checks the INTR pin and, if interrupts are enabled (`IF` flag in `EFLAGS` is set), begins interrupt processing. It reads the vector number from the PIC. The CPU then uses this number as an index into the **Interrupt Descriptor Table (IDT)**, which was loaded via `lidtr` during `idt_init()`. It fetches the corresponding `idt_entry` descriptor.
4. **CPU Validates and Switches Stacks (Privilege Change):** The CPU performs critical validation checks on the `idt_entry`: it must be **Present** (`IDT_PRESENT` flag), and the current privilege level (CPL, from `CS`) must be at least as privileged as the descriptor's **DPL** (`IDT_DPL_0` for kernel handlers). If coming from user mode (CPL=3) to a kernel handler (DPL=0), the CPU automatically switches to the kernel stack. It does this by loading the `SS:ESP` values from the **Task State Segment (TSS)**. This prevents user-space stack corruption from compromising the kernel.
5. **CPU Saves State and Transfers Control:** Before jumping to the handler, the CPU saves the essential machine state onto the (now kernel) stack to allow a seamless return. This **saved context** typically includes: `EFLAGS`, `CS`, `EIP` (the return address), and possibly an `error_code` for CPU exceptions. It then clears the `IF` flag (disabling further maskable interrupts), loads `CS:EIP` from the `idt_entry` (the handler's address), and begins executing the **Interrupt Service Routine (ISR)**.
6. **Assembly ISR Stub and C Handler:** The first code executed is a small, assembly-language **ISR stub**. Its jobs are: a. Save all general-purpose registers (`EAX`, `ECX`, ...) and segment registers (`DS`, `ES`, ...) onto the stack, creating a complete `registers` structure. b. Load the kernel data segment selector (`KERNEL_DS`) into segment registers. c. Call the high-level C function `isr_handler(struct registers *regs)`, passing a pointer to the saved registers. Inside `isr_handler`, the kernel code can safely examine the `registers` structure (including the `int_no` field) to determine the cause and handle it (e.g., read a scancode from the keyboard's I/O port `0x60`).
7. **Sending End-of-Interrupt (EOI):** After the device-specific handling is complete, but before returning, the kernel **must** signal the PIC that the interrupt is finished. For the PIC, this is done by sending a non-specific EOI command (`0x20`) to the `PIC1_COMMAND` port (`0x20`). If the interrupt came from the second PIC (IRQ8-15), an EOI must also be sent to `PIC2_COMMAND` (`0xA0`). Failing to send the EOI will "mask" that IRQ line, preventing any future interrupts from that device.
8. **Restoring State and Returning (`iret`):** The `isr_handler` returns to the assembly stub. The stub then: a. Restores the saved segment registers. b. Restores all general-purpose registers from the stack. c. Executes the `iret` (interrupt return) instruction. This instruction atomically pops `EIP`, `CS`, and `EFLAGS` from the stack, effectively resuming the interrupted code. If a stack switch occurred, `iret` also pops `SS:ESP`, returning to the user stack.

System Call Flow

Mental Model: The Customer Service Desk. A user process is like a customer in a corporate building. They can work independently in their office (user space) but need to request services from corporate headquarters (the kernel), such as printing a document or accessing a secure file. They cannot enter the secure headquarters themselves. Instead, they use a dedicated, secure phone line (**software interrupt** `int 0x80`) that rings at a specific desk. The customer states their request number (**system call number**, e.g., `SYS_WRITE`) and provides arguments (file descriptor, buffer, count). The corporate employee (**kernel dispatcher**) looks

up the request in a manual (**system call table**), verifies the arguments are safe, performs the privileged operation, and returns the result via the same phone line. This controlled interface is the only way for user processes to access kernel functionality, enforcing security and abstraction.

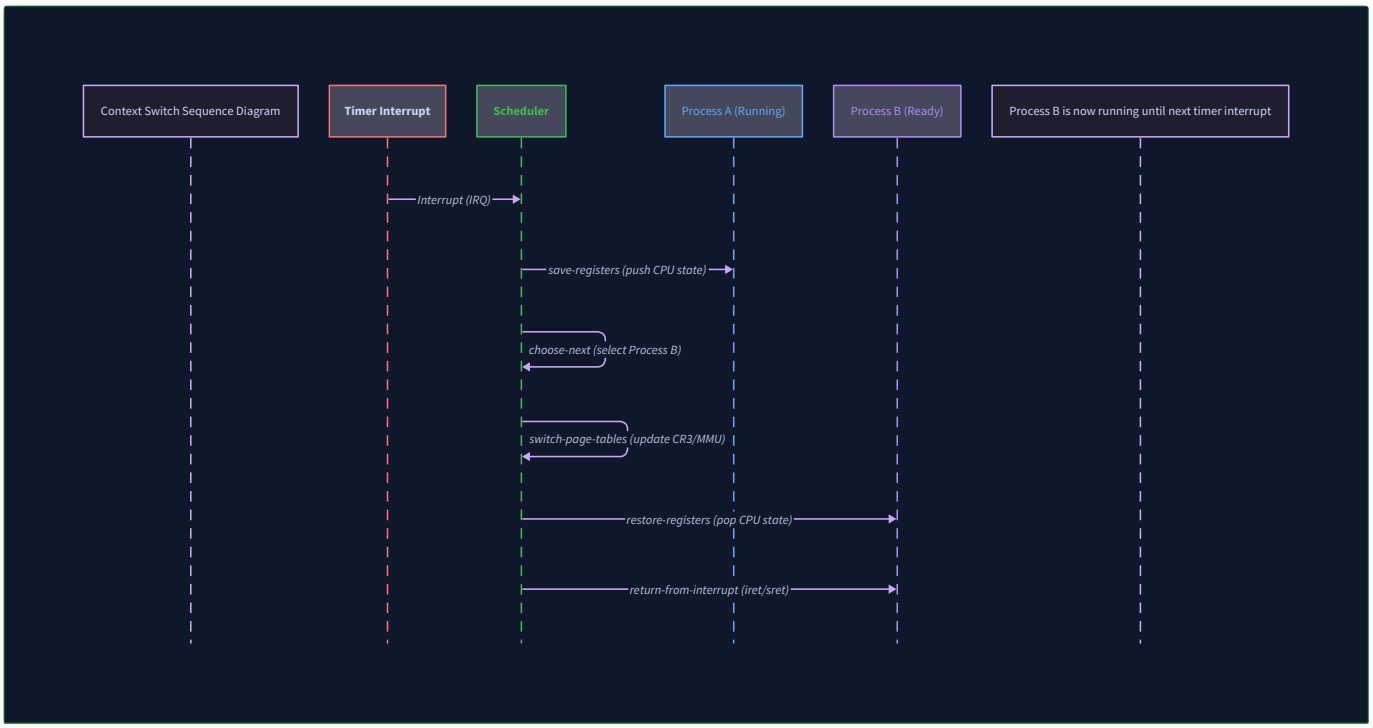
System calls are the fundamental API of the operating system, allowing user-mode processes to request services from the kernel. They are a special case of the interrupt flow, triggered intentionally by software (`int` instruction) rather than by hardware. The sequence highlights the transition between privilege levels and the marshaling of arguments.

1. **User Process Prepares and Executes `int 0x80` :** A user-mode process needs a kernel service (e.g., write to the screen). It follows the **system call convention**, placing the system call number in `EAX` (e.g., `SYS_WRITE` = 1) and its arguments in the other registers (e.g., `EBX` =file descriptor, `ECX` =buffer pointer, `EDX` =count). It then executes the assembly instruction `int 0x80` . This instruction is a software-generated interrupt and triggers the same CPU mechanism as a hardware interrupt.
2. **CPU Interrupt Processing (Vector 0x80):** The CPU looks up vector `0x80` in the IDT. The kernel must have configured this entry during initialization (`idt_set_gate`), setting its **DPL** to `IDT_DPL_3` to allow it to be invoked from user mode. The CPU switches to the kernel stack (as described in the interrupt flow), saves the user state (`SS:ESP` , `EFLAGS` , `CS:EIP`), and jumps to the handler address specified in the IDT entry for vector `0x80` .
3. **System Call Entry Assembly Stub:** The assembly stub for `int 0x80` is similar to other ISR stubs but has a critical additional role: it must carefully save the user-space stack pointer (`ESP`) and the contents of the data segment registers. It then calls the central **C dispatcher**: `syscall_dispatcher(int syscall_num, uint32_t arg1, uint32_t arg2, uint32_t arg3, uint32_t arg4, uint32_t arg5)` . The stub extracts `syscall_num` from the saved `registers.eax` field and the arguments from `registers.ebx` , `registers.ecx` , etc., passing them as function parameters.
4. **Dispatcher Validation and Routing:** Inside `syscall_dispatcher` , the kernel performs safety and sanity checks before acting. It validates that the `syscall_num` is within the bounds of the implemented system call table. More importantly, for any system call involving pointers (like `SYS_WRITE` with a buffer pointer in `ECX`), the kernel **must** verify that the pointer points to memory that the user process is allowed to access and that the buffer lies entirely within the user's address space. This prevents malicious or buggy processes from reading/writing kernel memory. After validation, the dispatcher uses the `syscall_num` as an index into a function pointer table to call the appropriate kernel service routine (e.g., `sys_write`).
5. **Kernel Service Execution:** The kernel service routine (`sys_write`) now executes in kernel mode, with full privilege. It performs the requested operation—copying data from the (now-verified) user buffer to the VGA memory or a file descriptor. It may block the calling process if necessary (e.g., waiting for disk I/O). When finished, it places its return value (e.g., number of bytes written, or `-1` on error) into a designated location, typically the `eax` field of the saved `registers` structure.
6. **Return to User Mode:** Control returns to `syscall_dispatcher` , then to the assembly stub. The stub restores the general-purpose registers, but crucially, it loads the saved `eax` with the system call's return value. Finally, it executes `iret` , which restores the user-mode `CS:EIP` (pointing to the instruction after the `int 0x80`), `EFLAGS` , and `SS:ESP` . The user process resumes, finding the result of its system call in the `EAX` register, following standard calling conventions.

Context Switch Flow

Mental Model: The Chess Tournament Director. Imagine a round-robin chess tournament with multiple games happening on a single board. The **tournament director (scheduler)** controls a timer. When the timer rings, the director says "stop!" to the current player (**Process A**). That player must carefully note the exact position of every piece on the board (**saves CPU register state**). The director then consults the roster and declares it's the next player's (**Process B**) turn. That player approaches the board, sets up the pieces exactly as they were in their last game (**restores their saved register state**), and the timer is reset. The director's job is to ensure fair time slices and a smooth transition between players, creating the illusion that each has their own dedicated board. The **timer interrupt** is the director's bell.

Context switching is the act of suspending the execution of one process and resuming another. It is the cornerstone of preemptive multitasking. This flow is typically initiated by the timer interrupt (IRQ0), transforming a hardware timer tick into a software decision to change the running task. The following sequence, also depicted in the diagram, shows the collaboration between the interrupt handler, scheduler, and low-level assembly routine.



1. **Timer Interrupt Fires (IRQ0):** The programmable interval timer (PIT) or APIC timer raises IRQ0 at a fixed frequency (e.g., 100 Hz). This triggers the **Interrupt Handling Flow** described earlier. The CPU saves the state of the currently running **Process A** and enters the timer's ISR.
2. **Timer ISR and Scheduler Hook:** The timer's interrupt handler, after performing device-specific actions (e.g., updating a kernel tick counter), calls a scheduler-specific function. Often, this is integrated into the `isr_handler` logic when `int_no` corresponds to the timer IRQ. The key call is to `schedule()`.
3. **Scheduler Decision (`schedule()`):** The `schedule()` function is the heart of preemption. It: a. **Saves the current process context:** It accesses the currently running `pcb` (e.g., via a global `current_process` pointer). It needs to save the CPU state *after the interrupt saved the minimal state*. The `registers` structure on the kernel stack, saved by the ISR stub, contains the state of Process A as it was *at the moment of the interrupt*. `schedule()` ensures this state is stored in `current_process->regs`. b. **Chooses the next process:** It calls `scheduler_get_next_process()`, which implements the **round-robin** algorithm. This function selects the next `pcb` from the `PROC_STATE_READY` queue, often by rotating a linked list. The selected process becomes `next_process`. c. **Updates Process States:** It changes the state of Process A from `PROC_STATE_RUNNING` to `PROC_STATE_READY` (if it's still alive) and places it back in the ready queue. It changes the state of `next_process` (Process B) from `PROC_STATE_READY` to `PROC_STATE_RUNNING`. d. **Updates the current pointer:** It sets the global `current_process` pointer to `next_process`.
4. **Invoke Low-Level Context Switch (`switch_context`):** With the decision made, `schedule()` calls the architecture-dependent `switch_context(cpu_registers_t** old_sp, cpu_registers_t* new_sp)` function. It passes the address where to *save* the current kernel stack pointer (which points to Process A's saved registers) and the kernel stack pointer from Process B's `pcb` (which points to Process B's previously saved registers).
5. **Assembly Context Switch (`switch_context`):** This assembly routine performs the actual "swapping of worlds." a. **Save current stack pointer:** It stores the current value of `ESP` (which points to Process A's saved state on its kernel stack) into the memory location specified by `old_sp`. b. **Switch stack:** It loads `ESP` with the value of `new_sp` (Process B's kernel stack pointer). c. **Switch page directory (CR3):** It loads the `CR3` register with the physical address of Process B's page directory (`next_process->page_directory`). This changes the virtual address space. The instruction immediately following this load must be in an identity-mapped or kernel-global page to avoid crashing. d. **Flush TLB:** Changing `CR3` automatically flushes the entire TLB, ensuring no stale translations from Process A's address space are used. e. **Restore new process's state:** The new stack (`ESP`) now points to Process B's saved `registers` structure. The assembly code restores all general-purpose and segment registers from this structure, precisely as an ISR stub would. f. **Return (`iret`):** Finally, it executes an `iret`. This instruction pops `EIP`, `CS`, and `EFLAGS` from Process B's kernel stack. Crucially, if Process B was in user mode when it was preempted, the saved `CS` will have a DPL of 3, and `iret` will also pop `SS:ESP`, switching the CPU back to Process B's user-mode stack. Execution thus resumes inside Process B at the exact instruction where its last timer interrupt (or system call) occurred.
6. **Return from Interrupt (to Process B):** The `iret` in `switch_context` does not return to the `schedule()` function in Process A's context. Instead, it jumps directly into the middle of Process B's previously interrupted execution. The timer ISR's final `iret` (which would have returned to Process A) is never executed from Process A's perspective. The flow of control has been surgically redirected. The timer interrupt that began this sequence is now seen as finishing in Process B's timeline. The EOI for the timer interrupt was likely sent early in the ISR, so no further device cleanup is needed.

Implementation Guidance

This guidance translates the high-level flows into concrete implementation steps, focusing on the critical junctions where components interact.

Technology Recommendations Table:

Component	Simple Option	Advanced Option
Bootloader	Multiboot-compliant (GRUB)	Custom bootloader in assembly
Interrupt Handling	8259 PIC	Local APIC & I/O APIC
Context Switch	`jmp`-based assembly with manual register save/restore	`fxsave` / `fxrstor` for FPU state, TSS for stack switching
System Call Entry	Single `int 0x80` gate	`sysenter` / `sysexit` or `syscall` / `sysret` (AMD/Intel)

Recommended File/Module Structure:

```
src/
├── boot/                # Bootloader and early entry (Milestone 1)
│   ├── boot.asm         # First-stage bootloader (optional)
│   ├── multiboot.asm    # Multiboot header + early entry point
│   └── gdt.asm          # GDT loading and protected mode switch
├── kernel/              # Core kernel
│   ├── main.c           # kmain(), high-level init sequence
│   ├── panic.c          # panic() function
│   └── kprintf.c        # kprintf() and VGA driver (vga_init)
├── arch/i386/           # Architecture-specific code
│   ├── isr.asm          # ISR assembly stubs and irq_install
│   ├── idt.c            # idt_init(), idt_set_gate()
│   ├── pic.c            # pic_remap(), irq_enable/disable
│   ├── ports.asm        # arch_port_read/write_byte
│   └── context_switch.asm # switch_context()
├── drivers/             # Hardware drivers
│   └── keyboard.c       # keyboard_init(), scancode mapping
├── mm/                  # Memory management (Milestone 3)
│   ├── pmm.c            # pmm_init(), pmm_alloc/free_frame()
│   ├── vmm.c            # vmm_init(), map_page(), page table ops
│   └── heap.c           # heap_init(), kmalloc(), kfree()
└── proc/                # Process management (Milestone 4)
    ├── process.c        # process_create(), process_destroy(), PCB
    ├── scheduler.c      # scheduler_init(), schedule(), get_next_process
    └── syscall.c        # syscall_dispatcher(), system call handlers
```

Core Logic Skeleton Code:

```c
// File: src/kernel/main.c

// This function orchestrates the high-level boot flow described in Section 6.1.

void kmain(uint32_t mboot_magic, void* mboot_info_ptr) {

    // Sanity check: Did we get booted by a Multiboot-compliant loader?

    // TODO 1: If mboot_magic != MULTIBOOT_MAGIC, call panic().


    // Initialize fundamental output first, so we can see messages.

    vga_init();

    kprintf("Kernel booting...\n");


    // Set up the CPU environment for protected mode operation.

    // TODO 2: Call gdt_init() to (re)load the Global Descriptor Table.


    // Initialize the Physical Memory Manager using the bootloader's memory map.

    // TODO 3: Cast mboot_info_ptr to (struct multiboot_info*), then call pmm_init() with it.


    // Set up the Interrupt Descriptor Table and hardware interrupt controller.

    // TODO 4: Call idt_init() to populate the IDT with default handlers.

    // TODO 5: Call pic_remap() to move IRQs 0-15 to vectors 0x20-0x2F.

    // TODO 6: Call keyboard_init() to set up the keyboard IRQ handler (IRQ1).


    // Enable virtual memory. This is a point of no return.

    // TODO 7: Call vmm_init() to set up initial kernel page tables (identity map + higher half).

    // TODO 8: Call enable_paging() to set the PG bit in CR0. Ensure code is position-independent or identity-mapped.


    // Initialize the kernel heap, which depends on virtual memory being active.

    // TODO 9: Call heap_init().


    // Initialize the process scheduler. This sets up the timer interrupt for preemption.

    // TODO 10: Call scheduler_init().


    // Create the first user process (e.g., a simple test program).

    // TODO 11: Call process_create("init", &some_entry_point) to create an initial process.


    kprintf("Kernel initialization complete. Entering idle loop.\n");


    // The kernel's main work is now done. It becomes an idle loop,

    // with all further activity driven by interrupts (timer, keyboard, syscalls).

    // TODO 12: Enable interrupts with the 'sti' instruction (asm volatile).

    // TODO 13: Enter an infinite loop: while (1) { asm volatile("hlt"); }

}
```

```asm
; File: src/arch/i386/context_switch.asm
; This assembly routine implements the low-level context switch described in Section 6.4.
; It is called from schedule() in C.
global switch_context
switch_context:
    ; Function signature: void switch_context(cpu_registers_t** old_sp, cpu_registers_t* new_sp);
    ; old_sp: pointer to where to save the current ESP
    ; new_sp: the new kernel stack pointer to load

    ; TODO 1: Save the current (Process A) kernel stack pointer.
    ; The first parameter ([esp+4]) is the address of 'old_sp'.
    ; Move the current value of ESP into the location pointed to by that address.

    ; TODO 2: Switch to the new (Process B) kernel stack.
    ; The second parameter ([esp+8]) is the value 'new_sp'.
    ; Move this value into the ESP register.

    ; TODO 3: Switch the page directory (CR3) if needed.
    ; The new process's PCB should contain its page directory physical address.
    ; This step is highly architecture-specific and may be done in C before calling switch_context.
    ; If done here, assume the address is passed as a third argument or is at a known offset on the new stack.

    ; TODO 4: Restore the general-purpose registers for Process B.
    ; The new stack pointer (ESP) now points to a saved cpu_registers_t structure.
    ; Pop (or mov) the values for EDI, ESI, EBP, EBX, EDX, ECX, EAX from this structure.
    ; Note: The order must match how they were saved in the ISR stub.

    ; TODO 5: Restore the segment registers for Process B (if they were saved).
    ; Pop (or mov) DS, ES, FS, GS. Ensure you load valid segment selectors.

    ; TODO 6: Return using IRET to restore EIP, CS, EFLAGS, and possibly user SS:ESP.
    ; The stack should now be pointing at the saved EIP for Process B.
    ; Use the 'iret' instruction. This does not return to the caller (schedule()).
    ; Execution will resume in Process B.
    iret
```

Language-Specific Hints (C/ASM):

- **Inline Assembly:** Use `asm volatile` for instructions that must not be moved by the optimizer (e.g., `cli`, `sti`, `outb`).
- **Calling Convention:** The `switch_context` assembly function uses the cdecl calling convention; arguments are on the stack in reverse order.
- **Register Clobbering:** In your ISR assembly stubs, carefully list clobbered registers for the compiler if using GCC's extended asm in C.
- **Memory Barriers:** After writing to the `CR3` register, consider a serializing instruction like `jmp` to a known identity-mapped address to ensure the pipeline is flushed in the new address space.
- **Stack Alignment:** Ensure the stack is 16-byte aligned before making C calls from assembly, as required by some ABIs, to prevent subtle crashes.

Debugging Tips for Data Flows:

Symptom	Likely Cause	How to Diagnose	Fix
Triple fault immediately after enabling paging.	Page tables incorrectly set up, or kernel code not identity-mapped after `CR0.PG` is set.	Use QEMU's `-d cpu_reset` flag. Check the `CR2` register value after the fault (it holds the offending linear address).	Ensure the code that enables paging and the code immediately after it are mapped at the same virtual and physical address (identity mapped).
Keyboard interrupts work once, then stop.	Missing End-of-Interrupt (EOI) signal to the PIC.	Add a `kprintf` inside the keyboard ISR. If it prints only once, EOI is missing.	Ensure you send `0x20` to `PIC1_COMMAND` (and `PIC2_COMMAND` if IRQ >= 8) at the **end** of the ISR, before `iret`.
System call returns garbage values or crashes.	Incorrect argument marshaling between user registers and the C dispatcher.	Log all arguments inside `syscall_dispatcher`. Check they match what the user process set.	Ensure your assembly stub correctly extracts `regs->eax`, `regs->ebx`, etc., and passes them in the correct order to the C function.
After a context switch, the new process starts from its beginning, not where it left off.	The saved `EIP` on the process's kernel stack is wrong (likely the process's start address, not the interrupted address).	Examine the saved `registers` structure on the process's kernel stack. The `eip` field should point into the middle of the process's code.	Ensure the timer ISR stub saves the correct `EIP` (pushed by the CPU on interrupt). In `process_create`, initialize the saved `EIP` to the process's entry point **and** set up a fake stack frame so the first `iret` works.
Switching to a process causes an immediate page fault.	The process's page directory (`CR3`) is invalid, or its kernel stack is not mapped.	Check the physical address loaded into `CR3`. Use QEMU's `info tab` command to examine page tables.	Ensure `process_create` allocates and sets up a valid page directory that includes the kernel's code/data mapping (shared higher-half) and a mapped user stack.

7. Error Handling and Edge Cases

> **Milestone(s):** Milestone 2 (Interrupts & Keyboard), Milestone 3 (Memory Management), Milestone 4 (Process Management)

An operating system kernel operates at the boundary between hardware and software, where failures can have catastrophic consequences. Unlike user-space applications that can simply crash and be restarted, kernel errors can corrupt system state, cause data loss, or lead to complete system hangs. This section outlines our strategies for detecting, handling, and recovering from the various failure modes that our educational kernel will encounter. The core philosophy is **defensive programming at every layer** — we assume hardware can misbehave, software can contain bugs, and resources are always finite.

7.1 Hardware and Low-Level Errors

> **Mental Model:** The **Emergency Response Team** at a nuclear power plant. When sensors detect abnormal conditions (radiation spikes, pressure changes), predefined emergency protocols activate immediately. Some conditions require automatic containment (closing valves), others require operator intervention (manual overrides), and catastrophic failures trigger complete shutdowns to prevent wider damage. Similarly, the kernel must respond to hardware anomalies with appropriate severity—logging, containment, or controlled shutdown.

Hardware errors originate from the CPU itself (exceptions), peripheral devices (device failures), or invalid hardware states (triple faults). These errors occur asynchronously and often at the most inconvenient times, requiring careful handling to maintain system integrity.

7.1.1 CPU Exception Handling Strategy

CPU exceptions are synchronous errors that occur when the processor encounters an illegal condition while executing an instruction (e.g., dividing by zero, accessing invalid memory). Our exception handling strategy follows a tiered approach based on severity:

Decision: CPU Exception Classification and Response

- **Context:** The CPU generates 20+ predefined exception types with varying severity. We must decide how to handle each: some are recoverable (page faults), some indicate programming errors (general protection faults), and some are catastrophic (double faults).
- **Options Considered:**
  1. **Panic on all exceptions:** Simple to implement but prevents legitimate uses like demand paging or copy-on-write.
  2. **Attempt recovery for all exceptions:** Complex and risky—some exceptions indicate unrecoverable corruption.
  3. **Tiered classification:** Categorize exceptions by severity and handle appropriately.
- **Decision:** Implement tiered classification with three response levels: Recoverable, Kill Process, and Panic.
- **Rationale:** This matches real-world OS practice: page faults are normal for virtual memory; protection faults in user mode kill the offending process; kernel-mode exceptions indicate kernel bugs and require panic.
- **Consequences:** Requires careful exception handler design, accurate fault information decoding, and process termination mechanisms before they're fully available.

The following table defines our classification and default handling for each CPU exception:

Exception Number	Exception Name	Severity Class	Default Action	Recovery Strategy
0	Divide Error	Kill Process	Terminate process with exit code 128+0	Not recoverable—invalid arithmetic
1	Debug	Special	Continue or invoke debugger	Used for breakpoints in debugging
2	NMI Interrupt	Panic	Halt system	Non-maskable interrupt—hardware failure
3	Breakpoint	Kill Process	Terminate process	Debugging trap—not for production
4	Overflow	Kill Process	Terminate process	INTO instruction overflow
5	Bound Range Exceeded	Kill Process	Terminate process	BOUND instruction violation
6	Invalid Opcode	Kill Process	Terminate process	Malformed or privileged instruction
7	Device Not Available	Kill Process	Terminate process	FPU/MMX instruction without FPU
8	Double Fault	Panic	Halt with error message	Exception while handling exception
9	Coprocessor Segment Overrun	Kill Process	Terminate process	Obsolete (x87 FPU error)
10	Invalid TSS	Kill Process	Terminate process	Bad Task State Segment
11	Segment Not Present	Kill Process	Terminate process	Missing segment descriptor
12	Stack-Segment Fault	Kill Process	Terminate process	Stack overflow or bad SS
13	General Protection Fault	Mixed	Kernel: Panic, User: Kill Process	Invalid memory access or privilege violation
14	Page Fault	Recoverable	Invoke page fault handler	Handle missing page, protection, or reserved bits
15	(Reserved)	—	—	—
16	x87 Floating-Point Exception	Kill Process	Terminate process	FPU calculation error
17	Alignment Check	Kill Process	Terminate process	Unaligned memory access (if enabled)
18	Machine Check	Panic	Halt with error message	Hardware failure (CPU, memory, bus)
19	SIMD Floating-Point Exception	Kill Process	Terminate process	SSE/AVX calculation error
20-31	(Reserved/Vendor)	Panic	Halt system	Unknown/processor-specific

Implementation Approach: Each exception handler in `isr_handler` will examine the `int_no` field in the `registers` structure, classify the exception, and take appropriate action. For page faults (exception 14), the handler will:

1. Extract faulting address from CR2 register
2. Check error code bits in `registers->err_code` to determine cause
3. If cause is "page not present" and address is valid, allocate frame and map it
4. If cause is protection violation or invalid address, kill the process

7.1.2 Device Failure Handling

Peripheral devices (keyboard, timer, disk) can fail, timeout, or return invalid data. Our strategy focuses on **graceful degradation**—when a device fails, we disable it and continue operating without that functionality when possible.

Decision: Device Error Response Strategy

- **Context:** Device drivers communicate with hardware via I/O ports or memory-mapped I/O. These operations can fail due to missing hardware, electrical issues, or firmware bugs.
- **Options Considered:**
  1. **Ignore and retry:** Simple but can hang indefinitely on truly failed hardware.
  2. **Panic on device failure:** Too drastic—keyboard failure shouldn't crash entire system.
  3. **Disable failing device and continue:** Allows system to operate in degraded mode.
- **Decision:** Implement retry-with-timeout for transient errors, disable device after persistent failures.
- **Rationale:** This matches real-world resilience: transient errors (electrical noise) recover with retries; permanent failures (disconnected device) are isolated.
- **Consequences:** Requires device state tracking, timeout mechanisms, and ability to disable IRQs for failed devices.

Device failure handling follows this flow:

1. **Detection:** Timeout on I/O operation, invalid status bits, or checksum failure
2. **Retry:** Attempt operation N times (configurable per device)
3. **Escalation:** If retries fail, log error, disable device IRQ, mark device unavailable

4. **Recovery:** Some devices support reset commands; attempt if available

For the PS/2 keyboard driver:

- **Scancode corruption:** If keyboard controller returns invalid scancodes (not in mapping table), ignore them
- **Controller timeout:** If keyboard doesn't respond to commands after 3 attempts, disable keyboard IRQ and log error
- **Buffer overflow:** If scancode buffer fills, discard oldest entries and increment error counter

7.1.3 Triple Fault and Catastrophic Failure Protocol

A triple fault occurs when the CPU encounters an exception while trying to handle a previous exception (e.g., page fault handler causes another page fault). This is unrecoverable and triggers a CPU reset.

Our strategy: While we cannot prevent all triple faults (they indicate serious bugs), we can maximize diagnostic information before reset:

1. **Double fault handler (exception 8)** saves critical registers to a reserved memory area
2. **Triple fault** triggers CPU reset via hardware
3. **Bootloader** checks for "last crash" signature in reserved memory
4. If signature exists, bootloader displays crash info before loading kernel

This requires cooperation between bootloader and kernel:

- Reserve 1KB of memory at known physical address (e.g., 0x1000) for crash dump
- Double fault handler writes `registers` structure and error info to this area
- Set magic number (e.g., 0xDEADFA11) to indicate valid crash data
- Bootloader checks for magic number and prints contents if present

7.2 Resource Management Edge Cases

> **Mental Model:** A **National Park with Limited Campgrounds**. The park has fixed resources (campsites, water, parking). Rangers must track reservations and current usage. When demand exceeds capacity, they implement policies: waitlist for campsites, water rationing during drought, or closing the park entirely when conditions become unsafe. Similarly, the kernel must manage finite memory and process slots with clear policies for exhaustion scenarios.

Resource management components (physical memory allocator, heap allocator, process table) operate under constant pressure. Well-behaved processes eventually release resources, but bugs, attacks, or legitimate high demand can exhaust any finite resource.

7.2.1 Out-of-Memory (OOM) Conditions

Memory exhaustion can occur at three levels: physical frames, virtual address space, or kernel heap. Each requires different handling strategies.

Decision: Physical Memory Exhaustion Policy

- **Context:** The physical frame allocator (`pmm_alloc_frame`) tracks free 4KB frames. When all frames are allocated, subsequent allocation requests cannot be satisfied.
- **Options Considered:**
  1. **Panic immediately:** Simple but unrealistic—real OSes attempt recovery.
  2. **Return NULL and let caller handle:** Places burden on all allocation sites.
  3. **Invoke OOM killer:** Select and terminate processes to free memory.
- **Decision:** For educational kernel: return NULL from `pmm_alloc_frame` , panic in kernel allocations, kill process in user allocations.
- **Rationale:** This teaches the distinction between kernel and user memory pressure. Kernel memory exhaustion is fatal (can't trust kernel heap). User memory exhaustion can be resolved by terminating the offending process.
- **Consequences:** Requires careful checking of `pmm_alloc_frame` return values, especially in page fault handler.

The following table outlines our memory exhaustion handling at each allocation layer:

Memory Layer	Exhaustion Detection	Primary Response	Fallback Response
Physical Frames	`pmm_alloc_frame()` returns NULL	Kernel: Call `panic()` , User: Kill process	Attempt to free cached/buffered pages (when implemented)
Virtual Address Space	`map_page()` finds no free page directory/table entries	Kill current process	Expand page tables if possible
Kernel Heap	`kmalloc()` finds no suitable free block	Call `panic()`	Attempt heap compaction (defragmentation)
Process Memory	`process_create()` cannot allocate stack/page tables	Return NULL to caller	Wait for memory to become available (block)

Implementation Details for Physical Memory Exhaustion:

1. **In page fault handler:** When `pmm_alloc_frame()` returns NULL during page fault handling:
   - If fault occurred in kernel mode → `panic("Out of physical memory in kernel")`
   - If fault occurred in user mode → Set process exit code to "OUT_OF_MEMORY" and terminate
2. **In `kmalloc()`:** When heap cannot satisfy request:
   - Attempt coalescence of adjacent free blocks
   - If still insufficient → `panic("Kernel heap exhausted")`
3. **Future enhancement:** Implement OOM killer that:
   - Scores processes by memory usage, runtime, priority
   - Selects victim process to terminate
   - Frees all its memory
   - Retries allocation

7.2.2 Process Table Exhaustion

The process table has a fixed maximum number of entries (e.g., 64 processes). When all slots are occupied, `process_create` cannot create new processes.

Decision: Process Limit Enforcement

- **Context:** The `pcb` array has fixed size `MAX_PROCESSES`. Each process occupies one slot until termination and cleanup.
- **Options Considered:**
  1. **Return error to caller:** Simple but may not be appropriate for all system calls (e.g., `fork()`).
  2. **Block caller until slot free:** More realistic but requires complex waiting mechanism.
  3. **Dynamically resize table:** Complex and introduces allocation during process creation.
- **Decision:** Return error (NULL) from `process_create` when table is full.
- **Rationale:** Keeps implementation simple for educational purposes. Real OSes would block or dynamically resize.
- **Consequences:** System calls like `fork()` must handle NULL return and set appropriate error code in user process.

Process table management edge cases:

Scenario	Detection	Handling Strategy
Table full	`process_create()` finds all `pcb.state != PROC_STATE_TERMINATED`	Return NULL, system call returns -EAGAIN
PID wraparound	PID counter reaches maximum (e.g., 32767)	Reset to 1, skip active PIDs
Orphaned process	Parent terminates before child	Reassign parent to init (PID 1) process
Zombie accumulation	Processes terminated but not reaped	Regular cleanup in scheduler or explicit wait system call

Zombie Process Prevention: To prevent terminated processes from permanently occupying `pcb` slots:

1. `process_destroy()` doesn't immediately free the slot
2. Instead, marks process as `PROC_STATE_TERMINATED` (zombie)
3. Parent process must call `wait()` system call to collect exit status
4. If parent terminates first, orphaned zombie is automatically reaped by init process
5. Scheduler's `get_next_process()` skips zombie processes

7.2.3 Deadlock Detection and Prevention

While our simple round-robin scheduler avoids many deadlock scenarios, resource deadlocks can still occur when processes compete for multiple resources.

Common Deadlock Scenarios:

1. **Memory deadlock:** Process A holds memory needed by B, B holds memory needed by A
2. **Sleep/wakeup race:** Process sleeps waiting for condition that never occurs
3. **Spinlock misuse:** Interrupt handler tries to acquire lock held by interrupted code

Our Prevention Strategy:

- **For memory:** Implement allocation order (always allocate frames in increasing address order)
- **For locks:** Never sleep while holding spinlock; disable interrupts when acquiring kernel locks
- **For sleep/wakeup:** Use condition variables with careful predicate checking

Decision: Lock Acquisition Ordering

- **Context:** Multiple kernel components may need to acquire multiple locks (e.g., process table lock and scheduler lock).

- **Options Considered:**
  1. **No ordering:** Risk of deadlock if two threads acquire same locks in different order.
  2. **Global lock ordering:** Define hierarchy and always acquire in same order.
- **Decision:** Implement simple lock hierarchy: Process table → Scheduler → Memory manager.
- **Rationale:** Prevents circular wait condition—most common deadlock cause.
- **Consequences:** Requires developers to know and follow the hierarchy.

7.2.4 Stack Overflow Protection

Kernel stack overflow corrupts adjacent memory and causes unpredictable crashes. We implement two protection mechanisms:

1. **Stack Canaries:** Place magic value (e.g., 0xCAFEBABE) at bottom of each kernel stack. Check during context switch—if corrupted, `panic("Kernel stack overflow")`.
2. **Stack Boundary Checking:** In interrupt handlers, compare `ESP` with stack limits before using stack. If `ESP` is outside valid range, trigger exception.

For User Processes: Each process gets guard page below stack (marked not present). Stack growth into guard page triggers page fault, allowing controlled stack expansion or overflow detection.

7.3 Error Recovery and System Stability

The ultimate goal of error handling is to maintain system stability despite failures. Our recovery hierarchy, from most to least desirable:

1. **Transparent Recovery:** Error corrected without user/process awareness (e.g., page fault allocating frame)
2. **Localized Failure:** Contain error to single process (e.g., kill process with segmentation fault)
3. **Degraded Operation:** Disable failing component, continue with reduced functionality (e.g., disable keyboard)
4. **Graceful Shutdown:** Save state, log errors, halt system cleanly
5. **Emergency Shutdown:** Immediate halt to prevent data corruption

Critical Rule: Never trust user-space pointers. All system calls must validate every pointer argument before dereferencing:

- Check pointer is in user-space range (below 0xC0000000 in higher-half kernel)
- Check pointer+length doesn't wrap around
- For write operations, verify page is writable

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Exception Handling	Fixed handler per exception	Pluggable handler registry with priority
OOM Handling	Panic on kernel OOM, kill process on user OOM	OOM killer with process scoring
Crash Dumping	Print registers to screen	Save to reserved memory area for bootloader display
Stack Protection	Stack canaries	Guard pages with page fault detection
Device Failure	Disable device after N failures	Device reset and reinitialization

B. Recommended File/Module Structure

```
src/
├── kernel/
│   ├── error/
│   │   ├── panic.c          # panic() implementation
│   │   ├── exceptions.c     # CPU exception handlers
│   │   ├── oom.c            # Out-of-memory handling
│   │   └── checks.c         # Validation functions (pointer checks, etc.)
│   ├── arch/x86/
│   │   └── double_fault_asm.s # Double fault handler with crash dump
│   └── boot/
│       └── crash_info.c     # Bootloader crash info display
├── include/kernel/
│   └── error.h              # Error codes, panic prototype, validation macros
└── bootloader/
    └── crash_display.asm    # Display crash dump from reserved memory
```

C. Infrastructure Starter Code

Complete panic() implementation with register dumping:

```c
/* kernel/error/panic.c */

#include <kernel/error.h>

#include <kernel/vga.h>

#include <kernel/io.h>


/* Reserve memory for crash dump at known physical address */

#define CRASH_DUMP_ADDR 0x1000

#define CRASH_MAGIC 0xDEADFA11


struct crash_dump {

    uint32_t magic;

    uint32_t error_code;

    uint32_t eip, eax, ebx, ecx, edx, esi, edi, esp, ebp;

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

    uint32_t cr0, cr2, cr3, cr4;

    char message[64];

} __attribute__((packed));


static void dump_registers(const char *file, int line, const char *msg) {

    /* Disable interrupts */

    asm volatile("cli");



    /* Save registers to crash dump area */

    struct crash_dump *dump = (struct crash_dump*)CRASH_DUMP_ADDR;

    dump->magic = CRASH_MAGIC;



    /* Get registers via inline assembly */

    asm volatile(

        "movl %%eax, %0\n"

        "movl %%ebx, %1\n"

        "movl %%ecx, %2\n"

        "movl %%edx, %3\n"

        : "=m"(dump->eax), "=m"(dump->ebx),

          "=m"(dump->ecx), "=m"(dump->edx)

    );



    /* Copy error message (truncate if too long) */

    int i;

    for (i = 0; msg[i] && i < 63; i++) {

        dump->message[i] = msg[i];

    }

    dump->message[i] = '\0';
```

```
}

void panic(const char *file, int line, const char *msg) {
    dump_registers(file, line, msg);

    /* Display panic message on screen */
    vga_set_color(VGA_COLOR_RED, VGA_COLOR_WHITE);
    vga_clear_screen();

    kprintf("KERNEL PANIC at %s:%d\n", file, line);
    kprintf("Reason: %s\n\n", msg);
    kprintf("System halted. Restart computer.\n");

    /* Halt CPU forever */
    asm volatile("cli\nhlt");
    for(;;);
}

/* Helper for checking user pointers */
bool is_user_pointer(const void *ptr, size_t len) {
    uint32_t addr = (uint32_t)ptr;

    /* In higher-half kernel, user space is below 0xC0000000 */
    if (addr >= 0xC0000000) {
        return false;  /* Kernel pointer */
    }

    /* Check for overflow/wraparound */
    if (addr + len < addr) {
        return false;  /* Would wrap around */
    }

    /* Check if address+length is still in user space */
    if (addr + len >= 0xC0000000) {
        return false;  /* Crosses into kernel space */
    }

    return true;
}
```

D. Core Logic Skeleton Code

TODO-based exception handler with classification:

```c
/* kernel/error/exceptions.c */

#include <kernel/error.h>

#include <kernel/process.h>

#include <kernel/scheduler.h>


/* Array of exception messages */

static const char *exception_messages[32] = {

    "Division By Zero",

    "Debug",

    "Non Maskable Interrupt",

    "Breakpoint",

    "Into Detected Overflow",

    "Out of Bounds",

    "Invalid Opcode",

    "No Coprocessor",

    "Double Fault",

    "Coprocessor Segment Overrun",

    "Bad TSS",

    "Segment Not Present",

    "Stack Fault",

    "General Protection Fault",

    "Page Fault",

    "Unknown Interrupt",

    "Coprocessor Fault",

    "Alignment Check",

    "Machine Check",

    "Reserved",

    "Reserved",

    "Reserved",

    "Reserved",

    "Reserved",

    "Reserved",

    "Reserved",

    "Reserved",

    "Reserved",

    "Reserved",

    "Reserved",

    "Reserved",

    "Reserved"

};
```

```
void handle_exception(struct registers *regs) {
    uint32_t int_no = regs->int_no;

    /* TODO 1: Check if exception number is within bounds (0-31) */
    /* If not, set int_no to 15 ("Unknown Interrupt") for handling */

    /* TODO 2: Check if we're in kernel mode when exception occurred */
    /* Hint: Check CS segment in regs->cs - kernel CS is 0x08 */
    bool in_kernel = (regs->cs == 0x08);

    /* TODO 3: Handle page fault specially (exception 14) */
    if (int_no == 14) {
        /* TODO 3.1: Read faulting address from CR2 register */
        /* uint32_t fault_address = read_cr2(); */

        /* TODO 3.2: Check error code bits in regs->err_code */
        /* Bit 0: Present (1=protection violation, 0=not present) */
        /* Bit 1: Write (1=write, 0=read) */
        /* Bit 2: User (1=user mode, 0=kernel mode) */

        /* TODO 3.3: If page not present and address is valid user address */
        /*   - Call pmm_alloc_frame() to allocate physical frame */
        /*   - Call map_page() to map virtual to physical */
        /*   - Return from interrupt (page fault resolved) */

        /* TODO 3.4: If protection violation or invalid address */
        /*   - If in kernel mode: panic("Page fault in kernel") */
        /*   - If in user mode: kill current process with SEGFAULT */
        /*   - process_destroy(current_process->pid) */
    }

    /* TODO 4: Handle double fault (exception 8) - always panic */
    if (int_no == 8) {
        /* TODO 4.1: Save critical registers to crash dump area */
        /* TODO 4.2: Call panic() with double fault message */
    }

    /* TODO 5: Handle other exceptions based on severity class */
    /* Use the classification table from section 7.1.1 */
```

```
    /* For "Kill Process" severity (exceptions 0, 3, 4, 5, 6, 7, etc.) */
    if (!in_kernel) {
        /* TODO 5.1: If in user mode, kill the offending process */
        /* kprintf("Process %d: %s\n", current_pid, exception_messages[int_no]); */
        /* process_destroy(current_pid); */
        /* scheduler_get_next_process(); */
    } else {
        /* TODO 5.2: If in kernel mode, panic (kernel bug) */
        /* panic(__FILE__, __LINE__, exception_messages[int_no]); */
    }

    /* TODO 6: For "Panic" severity (exceptions 2, 18, etc.): always panic */
    /* TODO 7: For "Recoverable" severity: implement appropriate recovery */
    /* TODO 8: For "Special" severity (Debug, Breakpoint): implement if needed */
}

/* Physical memory allocator with OOM handling */
void *pmm_alloc_frame(void) {
    /* TODO 1: Find free frame in bitmap */
    /* TODO 2: If found, mark as used, return physical address */

    /* TODO 3: If no free frame (OOM condition) */
    /*   - Check if called from kernel context (CS == 0x08) */
    /*   - If kernel context: panic("Out of physical memory") */
    /*   - If user context: return NULL (let page fault handler kill process) */

    /* TODO 4: Update free frame count in pmm_state */
    /* TODO 5: Return physical address of allocated frame */
    return NULL; /* Placeholder */
}
```

E. Language-Specific Hints (C)

1. **Use** `__attribute__((packed))` for structures shared with assembly (like `crash_dump`) to prevent alignment padding that could confuse the bootloader.
2. **Inline assembly for special registers:** Use `asm volatile("movl %%cr2, %0" : "=r"(value))` to read CR2 (page fault address).
3. **Volatile for hardware access:** Always use `volatile` when reading/writing hardware registers or memory-mapped I/O to prevent compiler optimization removal.
4. **Noreturn attribute:** Mark `panic()` with `__attribute__((noreturn))` to inform compiler it never returns.

F. Milestone Checkpoint for Error Handling

After implementing exception handlers (Milestone 2):

- **Test:** Trigger divide by zero in a test process
- **Expected:** Process terminates, system continues running
- **Command:** Run kernel in QEMU, type `calc 1/0` in test shell
- **Verification:** See "Process X: Division By Zero" message, process exits

After implementing OOM handling (Milestone 3):

- **Test:** Allocate memory until exhaustion
- **Expected:** User process killed, kernel continues
- **Command:** Run memory-hog test program
- **Verification:** See "Out of memory" message, hog process exits

After implementing pointer validation (Milestone 4):

- **Test:** Pass kernel pointer to `write()` system call
- **Expected:** System call returns error, process continues
- **Command:** Run test program with `write(1, (void*)0xC0000000, 10)`
- **Verification:** System call returns -EFAULT, no crash

G. Debugging Tips for Error Handling Issues

Symptom	Likely Cause	How to Diagnose	Fix
Triple fault immediately after boot	Exception handler causing another exception	Check IDT entries point to valid handlers	Ensure handlers preserve all registers, don't cause page faults
System hangs on first page fault	Page fault handler in infinite loop	Add debug print at start of handler	Check CR2 reading, ensure valid virtual address
"Out of memory" but memory available	Bitmap corruption or wrong memory map	Print bitmap and memory map at initialization	Verify multiboot memory map parsing, bitmap initialization
Process killed but becomes zombie	`process_destroy` not freeing slot	Check process state after termination	Ensure parent calls `wait()` or init reaps orphans
Kernel panic on valid user pointer	Pointer check too restrictive	Print pointer value and check logic	Fix `is_user_pointer()` bounds calculation
Keyboard stops working after error	Keyboard IRQ disabled on error	Check keyboard status register	Re-enable keyboard IRQ after transient error

8. Testing Strategy

> **Milestone(s):** This section provides the verification framework for all four milestones. It describes how to confirm the kernel is functioning correctly at each stage of development, using observable outputs and systematic testing techniques.

Testing an operating system kernel presents unique challenges. Unlike a user-space application, a kernel cannot rely on an underlying OS for debugging facilities or graceful failure recovery. A single bug can cause a complete system lockup, a **Triple Fault**, or silent corruption. Therefore, a strategic, incremental, and observable testing approach is paramount. The core philosophy is to **make the kernel's internal state externally visible** and to **test one major component at a time** before integrating them.

The strategy is divided into two parts. First, **Milestone Verification Checkpoints** provide concrete, observable outcomes for each development milestone—what you should see on screen or log when a component works. Second, **Testing Techniques for an OS Kernel** outlines the tools and methods to probe and validate the system, even when it appears to have crashed completely.

Milestone Verification Checkpoints

Each milestone culminates in specific, demonstrable capabilities. The tables below map each milestone's acceptance criteria to a verification method, expected output, and a concrete test procedure you can perform.

Milestone 1: Bootloader & Kernel Entry

The goal is to see the kernel take control from the firmware, initialize a clean C environment, and produce visible output.

Acceptance Criteria	Verification Method & Expected Output	Test Procedure
Bootloader loads kernel binary into memory.	The bootloader (or GRUB) prints a loading message, then the kernel's first print appears. No immediate reboot or blank screen.	1. Build the kernel and bootloader (or Multiboot-compliant kernel). 2. Boot in QEMU. 3. Observe the transition from bootloader messages (e.g., "Loading kernel...") to the kernel's first `kprintf` output (e.g., "Kernel entered.").
GDT configures 32-bit protected mode.	Kernel prints a success message from `gdt_init()` and begins executing 32-bit C code (evidenced by `kprintf` working). No triple fault.	1. Add a `kprintf("GDT loaded.\n");` call after `gdt_flush()`. 2. Boot and verify this message appears. 3. Further evidence: The kernel uses 32-bit pointers and the `kprintf` function works, which requires correct segment register reloads.
Kernel entry zeroes BSS and calls `kmain`.	A global variable in the BSS section (e.g., `int my_bss_var;`) is observed to be zero when printed.	1. In `kmain`, declare an external `extern int my_bss_var;` and print its value: `kprintf("BSS var: %d\n", my_bss_var);`. 2. Verify the output is `0`. If it's a random large number, BSS was not cleared.
VGA driver prints text with color.	Colored text appears at the top-left of the screen. Changing the color attribute (e.g., to red on black) produces a visible change.	1. In `kmain`, call `vga_init()` then `kprintf("Hello, Kernel World!\n");`. 2. Observe the white-on-black text. 3. Modify the default color in `vga_init` to `VGA_COLOR_RED` and rebuild. The text should now be red.

> **Key Insight:** For Milestone 1, the most critical test is **visual confirmation**. The screen is your primary debug interface. If the screen is blank after the bootloader, the CPU has likely triple-faulted and reset. Use QEMU's `-d cpu_reset` flag to confirm a reset occurred, then check your GDT loading and first `jmp` to the kernel entry point.

Milestone 2: Interrupts & Keyboard

The system must transition from a linear program to a reactive one, handling hardware events on demand.

Acceptance Criteria	Verification Method & Expected Output	Test Procedure
IDT contains valid entries for exceptions 0-31.	Triggering a controlled exception (e.g., divide-by-zero) prints a diagnostic message and halts gracefully, rather than triple-faulting.	1. In `kmain`, after `idt_init()`, intentionally execute `int 0;` (divide-by-zero) or `asm volatile("div %0" : : "r"(0));`. 2. The registered `isr_handler` should print a message like "Exception 0: Divide Error". The system should halt in a controlled `panic` state.
Interrupt handlers save/restore registers and return with `iret`.	After handling an interrupt (e.g., timer), the kernel continues normal execution without corruption. A periodic timer tick message appears.	1. Configure the Programmable Interval Timer (PIT) to generate IRQ0 interrupts at a slow rate (e.g., 10 Hz). 2. In the timer ISR, increment a global counter and print it every 10 ticks. 3. Observe the counter incrementing on screen. If the system locks up or the counter jumps erratically, register saving/restoring or the `iret` stack frame is incorrect.
PIC remapping to non-conflicting vectors.	Both CPU exceptions (0-31) and hardware IRQs (32-47) are handled correctly without conflict. Keyboard presses are received.	1. After remapping, verify the keyboard works (see below). 2. If the keyboard generates spurious characters or exceptions fire when you press keys, the PIC vector offsets likely overlap the exception range.
Keyboard driver converts scancodes to ASCII.	Pressing keys on the (emulated) keyboard echoes the correct character to the screen.	1. After `keyboard_init()`, focus the QEMU window and press keys like 'A', '1', and Enter. 2. The corresponding characters should appear on screen. Test lowercase (a) and uppercase (Shift+A).

> **Key Insight:** The **timer interrupt** is your best friend for testing the interrupt infrastructure. It's periodic, non-destructive, and provides visual feedback. Start by getting the timer working before moving to the keyboard, as it validates the entire PIC remapping and ISR dispatch pipeline.

Milestone 3: Memory Management

The kernel must manage memory dynamically and create a virtual address space, all while preserving its own ability to execute.

Acceptance Criteria	Verification Method & Expected Output	Test Procedure
Physical frame allocator allocates/frees 4KB frames without leaks.	The allocator can service a series of allocation and free requests. The count of free frames decreases and increases accordingly.	1. After `pmm_init`, print the initial `free_frames` count. 2. Call `pmm_alloc_frame()` in a loop 10 times, printing the returned address. 3. Call `pmm_free_frame()` on half of them. 4. Print the `free_frames` count again; it should equal `initial - 10 + 5`. The allocated addresses should be page-aligned (low 12 bits = 0).
Page tables correctly map virtual to physical addresses.	After enabling paging, the kernel's own code continues to run (`kprintf` still works). A newly mapped page can be written to and read from.	1. **The "Higher-Half Kernel" Smoketest:** If using a higher-half layout, the moment `enable_paging()` is called, all kernel code and data references become virtual. If the kernel doesn't immediately crash, your identity mapping or higher-half mapping is likely correct. 2. **Explicit Mapping Test:** Use `map_page` to map a new virtual address (e.g., `0x1000000`) to a freshly allocated physical frame. Write a known pattern (e.g., `0xDEADBEEF`) to that virtual address. Read it back and verify.
Kernel heap allocator handles `kmalloc` and `kfree`.	Dynamic allocation for varying sizes works. Repeated allocation/free cycles do not corrupt metadata or cause infinite loops.	1. Write a test in `kmain`: allocate four blocks of different sizes (e.g., 16, 128, 512 bytes). Store unique integers in each. 2. Free the second and fourth blocks. 3. Allocate a new block of size 200; it should reuse one of the freed spaces (you can print the returned pointer to see). 4. Verify the integers in the first and third blocks are unchanged.
MMIO regions marked uncacheable.	Writing to a memory-mapped device register (e.g., VGA buffer) produces an immediate visible effect.	1. This is inherently tested by your VGA driver: if the text appears instantly, the mapping is likely correct. For an explicit test, map the VGA buffer address `VGA_BUFFER_ADDR` with the `PTE_CACHE_DISABLE` and `PTE_WRITETHROUGH` flags. Text output should still work.

> **Key Insight: Test paging incrementally.** First, set up page tables but don't enable paging (`CR0.PG` bit). Walk the page tables in software to verify they look correct. Then, enable paging with a minimal identity map of the first 4MB. Finally, implement your full virtual memory layout. At each step, have a way to print diagnostic info *before* the risky operation.

Milestone 4: Process Management & Scheduler

The kernel must multitask between multiple processes, isolating them and providing a system call interface.

Acceptance Criteria	Verification Method & Expected Output	Test Procedure
Process Control Block (PCB) stores process state.	A dummy process can be created, and its `pcb` fields (PID, state, page table) are populated correctly.	1. Call `process_create("test_proc", dummy_entry_point)`. 2. In the scheduler's ready queue, traverse and print each PCB's `pid`, `name`, and `state`. The new process should appear with state `PROC_STATE_READY`.
Context switch saves/restores register state.	Two processes alternately execute, each maintaining its own independent counter or message loop.	1. Create two simple processes that each run an infinite loop: Process A prints "A" and increments a *local* counter, Process B prints "B". 2. Enable the timer interrupt and `schedule()` call. 3. Observe "A" and "B" interleaving on screen. If one process's counter resets or the output stops, registers (especially `ESP`, `EIP`) are not being saved/restored properly.
Round-robin scheduler with time slicing.	Processes yield execution at regular, timed intervals, not just when they voluntarily give up control.	1. Using the two processes from above, configure the PIT timer to interrupt at 100 Hz and call `schedule()` on each tick. 2. Observe the "A" and "B" output. You should see a sequence like "AAAABBBB" where the number of consecutive letters is determined by the time slice. If you only see one letter forever, the timer interrupt is not triggering a context switch.
System call interface via interrupt.	A user-mode process (or a kernel test) can invoke a system call (e.g., `SYS_WRITE`) and have it serviced by the kernel.	1. Implement a minimal `sys_write` that calls `kprintf`. 2. Create a process whose entry point is a function that uses `asm volatile("int $0x80")` with `SYS_WRITE` in `EAX`. 3. When scheduled, this process should trigger the system call and print a message. This proves the transition from user-mode (or simulated user-mode) to kernel-mode and back.

> **Key Insight: Start with a single process and a voluntary yield.** Implement `process_create` and a `yield()` system call that manually triggers the scheduler. Get one process to run, yield, and another to start. This validates your PCB and basic context switch without the complexity of preemptive timer interrupts. Then add the timer to make it truly round-robin.

Testing Techniques for an OS Kernel

Given the kernel's privileged position and fragility, traditional debugging methods (e.g., attaching GDB to a live process) are either impossible or require special setup. The following techniques form the essential toolkit for OS development.

1. Emulator-Based Debugging with QEMU and Bochs

Emulators are the safest and most powerful testing environment. They allow you to pause, inspect, and modify the entire machine state.

- **QEMU** is fast and feature-rich. Key testing flags:

- `-d cpu_reset,guest_errors,in_asm` logs CPU resets (triple faults), guest software errors, and executed assembly. This is the first tool to reach for when the screen goes black.
- `-s` opens a GDB server on port 1234. You can connect with `gdb` and set breakpoints, step through kernel code, and examine memory. This is invaluable for tracing complex bugs.
- `-S` freezes the CPU at startup, allowing you to connect GDB before any instruction executes.
- `-serial stdio` redirects the serial port output to the console, which is crucial for logging (see below).

- **Bochs** is slower but provides extremely detailed, pedagogical logging of every CPU instruction, memory access, and interrupt. Its built-in debugger is text-based and powerful for stepping through boot code. Use Bochs when you need to understand *exactly* what the CPU is doing cycle-by-cycle, especially during early boot and mode transitions.

Technique: When a test fails, run QEMU with `-d cpu_reset`. If you see a "Triple Fault" message followed by "CPU Reset", you know the failure caused a cascade of exceptions. The last logged instruction before the reset is the likely culprit.

2. Serial Port Logging (`kprintf` to Console)

The VGA screen is limited in space and can be corrupted by bugs in memory or video drivers. The **PC serial port (UART)** provides a robust, stable logging channel that survives most kernel panics.

1. **Implement a `serial_write` function** that writes characters to I/O ports `0x3F8` (COM1).
2. **Create a `klog` function** that formats strings and sends them via `serial_write`, in addition to or instead of VGA.
3. **Configure QEMU** with `-serial stdio`. Now, all `klog` output appears in your terminal, separate from the VGA display.

> This technique is so critical it deserves emphasis: **From Milestone 2 onward, all diagnostic prints should go to the serial port.** This allows you to debug issues that corrupt the VGA buffer or happen during context switches where the screen output is chaotic.

Example Logging Strategy:

```
[INFO] kmain: Kernel entry. Multiboot magic: 0x2BADB002
[INFO] pmm_init: Memory map parsed. 32768 frames available.
[ERROR] page_fault_handler: Page fault at 0xC0001234, error code 0x7
[PANIC] Assertion failed: pte != NULL (file: vmm.c, line: 142)
```
PLAINTEXT

3. Controlled Fault Injection and Stress Testing

Proactively test error paths and robustness by simulating bad conditions.

- **Memory Allocator Stress:** In a test loop, allocate random-sized blocks until `kmalloc` returns `NULL` (out of memory). Then free a random subset and continue. Monitor for corruption using **magic numbers** (`HEAP_MAGIC`) in block headers/footers. This exposes fragmentation bugs and coalescing errors.
- **Page Fault Injection:** Use the `invlpg` instruction or modify page table entries to invalidate a mapping while a process is using it. Verify your `page_fault_handler` correctly identifies the faulting address, allocates a frame, and resumes the process.
- **Process Table Exhaustion:** Create processes in a loop until `process_create` fails (e.g., `MAX_PROCESSES` reached). Ensure the kernel handles this gracefully, perhaps returning an error to the caller, rather than crashing.

4. "Canary" Values and Integrity Checks

Embed known values in data structures to detect corruption early.

- **Stack Canary:** Place a magic value (`0xCAFEBABE`) at the top of each process's kernel stack during creation. In the timer interrupt or context switch, check this value. If it's changed, a stack overflow has occurred.
- **Heap Magic Numbers:** As mentioned, every `heap_block_header` and `heap_block_footer` should contain a `HEAP_MAGIC` constant. `kfree` should verify these before proceeding. If they mismatch, print a diagnostic and panic—this catches buffer overflows and double-frees.
- **PCB Sanity Fields:** Include a `magic` field in the `pcb` set to a unique value (e.g., `0xPROCB0CK`). Any function that receives a `pcb*` should assert this magic is correct, catching wild pointers.

5. Systematic State Dumping on Panic

When a critical error occurs (`panic` is called), don't just halt. Dump as much state as possible to the serial port.

- **CPU Registers:** Save `CR0-CR4` , `EAX` , `EBX` , etc., into a `crash_dump` structure.
- **Fault Information:** Include the `EIP` where the panic occurred, an error message, and for page faults, the contents of `CR2` .
- **Stack Trace:** For 32-bit x86, you can walk the `EBP` chain to print a rudimentary backtrace (if the stack hasn't been corrupted).
- **Write this structure** to a known physical memory location (e.g., `CRASH_DUMP_ADDR = 0x1000`) so it survives a reboot. Configure QEMU to dump memory (`-d guest_errors,file=memdump.bin`) or use Bochs's internal dump capability to analyze it post-mortem.

Testing Workflow Summary:

1. **Develop & Test in QEMU** with serial logging enabled.

2. **On failure**, check the serial log first.
3. **If the log is silent or ends abruptly**, run QEMU with `-d cpu_reset` to confirm a triple fault.
4. **For intricate bugs**, use QEMU's GDB server (`-s -S`) to step through the code, setting breakpoints before the crash.
5. **For boot-time or hardware initialization bugs**, use Bochs for its superior logging of low-level CPU state.
6. **Regularly run stress tests** for memory and processes to uncover hidden bugs.

Implementation Guidance

This section provides practical code and commands to set up the testing environment described above.

A. Technology Recommendations Table

Component	Simple Option (Recommended)	Advanced Option
Emulator	QEMU (User Mode) with `-serial stdio`. Easy to use, fast, good debugging.	Bochs with detailed configuration file. Slower but more detailed logs.
Debugging	QEMU's GDB stub (`-s -S`) with GNU GDB.	Integrated Bochs internal debugger.
Logging	Serial port (UART) to console. Simple, reliable.	In-memory ring buffer dumped on panic. More complex but no I/O dependency.
Assertions	Custom `ASSERT(cond)` macro that calls `panic` on failure.	Compile-time static assertions for invariants where possible.

B. Recommended File/Module Structure

Add testing and debugging modules to your existing structure.

```
project-root/
  boot/               # Bootloader ASM files
  kernel/
    arch/i386/        # Architecture-specific
      serial.c        # <-- NEW: UART serial driver for logging
      debug.c         # <-- NEW: panic(), stack trace, register dump
    drivers/
      vga.c
      keyboard.c
    mm/
      pmm.c
      vmm.c
      heap.c
    proc/
      scheduler.c
      syscall.c
    kernel.c          # kmain()
    printk.c          # kprintf, klog (extends to serial)
    tests/            # <-- NEW: Test suites (optional)
      test_mm.c
      test_proc.c
  scripts/
    run_qemu.sh       # <-- NEW: Script to launch QEMU with correct flags
    run_bochs.sh      # <-- NEW: Script to launch Bochs
```

C. Infrastructure Starter Code (Serial Port Logging)

Here is a complete, simple UART driver for COM1. Place this in `kernel/arch/i386/serial.c`.

```c
// kernel/arch/i386/serial.c

#include <stdint.h>

#include <stdbool.h>


#define PORT_COM1 0x3F8


// Initialize the serial port (COM1) at 9600 baud, 8N1.

void serial_init() {

    arch_port_write_byte(PORT_COM1 + 1, 0x00); // Disable interrupts

    arch_port_write_byte(PORT_COM1 + 3, 0x80); // Enable DLAB (set baud rate)

    arch_port_write_byte(PORT_COM1 + 0, 0x03); // Set divisor low byte (115200 / 9600 = 12)

    arch_port_write_byte(PORT_COM1 + 1, 0x00); // Set divisor high byte

    arch_port_write_byte(PORT_COM1 + 3, 0x03); // 8 bits, no parity, one stop bit

    arch_port_write_byte(PORT_COM1 + 2, 0xC7); // Enable FIFO, clear, 14-byte threshold

}


// Check if the transmit buffer is empty.

static bool serial_is_transmit_empty() {

    return arch_port_read_byte(PORT_COM1 + 5) & 0x20;

}


// Write a single character to the serial port.

void serial_putc(char c) {

    while (!serial_is_transmit_empty()) {

        // Wait for the buffer to be ready.

    }

    arch_port_write_byte(PORT_COM1, c);

}


// Write a null-terminated string to the serial port.

void serial_puts(const char* str) {

    while (*str) {

        serial_putc(*str++);

    }

}
```

Now, modify your `kernel/printk.c`'s `kprintf` to also output to the serial port.

D. Core Logic Skeleton Code (Testing Helper)

Here is a skeleton for a memory allocator stress test. Place it in `kernel/tests/test_mm.c` (optional, but recommended).

```c
// kernel/tests/test_mm.c

#include <stdint.h>

#include <stddef.h>

#include "printk.h"

#include "heap.h"


#define NUM_ALLOCS 100

#define MAX_ALLOC_SIZE 2048


void test_heap_stress() {

    void* pointers[NUM_ALLOCS] = {0};

    size_t sizes[NUM_ALLOCS] = {0};


    klog("[TEST] Starting heap stress test.\n");


    // Phase 1: Allocate many blocks of random size.

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

        // TODO 1: Generate a pseudo-random size between 1 and MAX_ALLOC_SIZE.

        // Hint: Use a simple linear congruential generator or a fixed pattern.

        sizes[i] = (i * 73 + 123) % MAX_ALLOC_SIZE + 1;


        // TODO 2: Call kmalloc with this size, store pointer in pointers[i].

        // TODO 3: If kmalloc returns NULL, log "OOM at iteration i" and break.

        // TODO 4: Write a known pattern (e.g., 0xAA) into the allocated block.

    }


    // Phase 2: Randomly free half of the blocks.

    // TODO 5: For j = 0 to NUM_ALLOCS/2, pick a random index that is still allocated.

    // TODO 6: Call kfree on that pointer and mark it as freed (set pointers[idx] = NULL).

    // TODO 7: Verify the pattern in the block before freeing? (Be careful, it may be coalesced).


    // Phase 3: Allocate more blocks, checking for reuse.

    // TODO 8: Perform another series of allocations.

    // TODO 9: Print the pointers; some should fall within the gaps freed in Phase 2.


    klog("[TEST] Heap stress test completed.\n");

}
```

E. Language-Specific Hints (C)

- **Use** `__attribute__((packed))` for all structures that map directly to hardware (GDT, IDT, Multiboot structures) to prevent the compiler from inserting padding.
- **Inline Assembly for Critical Operations:** Use `asm volatile` for operations that must not be reordered or optimized away, like `cli` / `sti`, `invlpg`, `iret`.
- **Macros for Assertions:** Define a macro in a header: `#define ASSERT(cond) ((cond) ? (void)0 : panic(__FILE__, __LINE__, "Assertion failed: " #cond))`.

F. Milestone Checkpoint Commands

For each milestone, after building your kernel image (e.g., `kernel.bin`), run these commands to test.

- **Milestone 1 & 2 (Basic Boot & Interrupts):**

```bash
# Run with logging and CPU reset logging
qemu-system-i386 -kernel kernel.bin -serial stdio -d cpu_reset 2>&1 | tee qemu.log
```

Expected: Boot messages, "GDT loaded", "IDT loaded", and then periodic timer ticks or response to keyboard input. No "CPU reset"

- **Milestone 3 (Memory Management):**

```bash
# Run with GDB server for debugging potential page faults
qemu-system-i386 -kernel kernel.bin -serial stdio -s -S &
gdb kernel.elf -ex "target remote localhost:1234" -ex "break panic" -ex "c"
```

Expected: Kernel boots, prints memory statistics (free frames), and passes the heap stress test. If it crashes, GDB will break at

- **Milestone 4 (Processes):**

```bash
qemu-system-i386 -kernel kernel.bin -serial stdio
```

Expected: Kernel boots, creates initial processes, and you see interleaved output from multiple processes (e.g., "A1", "B1", "A2"

G. Debugging Tips Table

Symptom	Likely Cause	How to Diagnose	Fix
Screen is blank after bootloader.	Triple fault during early kernel init (GDT, IDT, first C code).	Run QEMU with `-d cpu_reset`. If you see a reset log, the last instruction before it is the culprit. Add serial logs right before `gdt_flush()` and `idt_init()`.	Check GDT descriptor limits and base addresses. Ensure the first GDT entry is NULL. Verify the `jmp` to protected mode is correct.
Keyboard presses produce wrong or no characters.	PIC not remapped correctly, keyboard ISR not registered, or scancode map wrong.	1. Check serial log for "PIC remapped" message. 2. In keyboard ISR, log the raw scancode port read. 3. Verify the ISR sends EOI to *both* PIC1 and PIC2 if IRQ >= 8.	Ensure `pic_remap()` moves IRQs to start at vector 0x20. Map scancode set 1 (XT) to ASCII correctly. Don't forget `irq_enable(1)` for keyboard.
`kmalloc` returns same pointer twice or corrupts data.	Heap metadata corruption (buffer overflow, double free).	Enable `HEAP_MAGIC` checks in `kmalloc` / `kfree`. Before each operation, traverse the heap list and print each block's magic, size, and free status.	Look for code writing past allocated boundaries. Ensure `kfree` only frees pointers returned by `kmalloc`. Implement boundary tag coalescing carefully.
Context switch crashes, jumping to random code.	`switch_context` saves/restores incomplete register set or wrong stack pointer.	In `switch_context` assembly, after saving registers, write them to a known location and print them. Compare saved vs. restored values.	Ensure you save/restore all general-purpose registers, `EIP`, and segment registers if changing privilege level. The `ESP` saved must point to the top of the saved context frame.
Page fault in kernel code after enabling paging.	Kernel code/data not mapped in virtual address space after switching page tables.	Before enabling paging, print the page directory entries for the kernel's virtual addresses. Ensure they map to correct physical frames with `PTE_PRESENT	PTE_WRITABLE`.

9. Debugging Guide

> **Milestone(s):** This section provides essential debugging techniques and tools applicable to all milestones, from the initial boot process to complex process management.

Kernel development is a unique and challenging form of debugging, where traditional tools like source-level debuggers may be unavailable or limited. When the system fails, it often fails catastrophically—resulting in a frozen screen, a **Triple Fault**-induced reset, or complete silence. This guide provides a structured, symptom-based approach to diagnosing and fixing the most common bugs encountered when building a minimal OS kernel. The goal is to transform the opaque "black screen of death" into a series of testable hypotheses you can systematically verify.

> **Debugging Mindset:** Think of kernel debugging as being a detective at a crime scene where the only witnesses are the CPU registers and the contents of memory. Your job is to reconstruct the series of events that led to the failure by examining the physical evidence left behind.

Common Bug Symptom → Cause → Fix Table

The following table organizes common failure symptoms by the milestone in which they most frequently appear. It links each symptom to its likely root cause and provides a concrete diagnostic and fix strategy. Use this as your first stop when your kernel behaves unexpectedly.

Symptom	Likely Milestone	Probable Cause	Diagnostic Steps	Fix
Screen remains blank after boot; no text appears.	1	1. VGA buffer is being written to the wrong memory address. 2. Bootloader failed to load the kernel, so execution never reaches `kmain`. 3. CPU is stuck in an endless loop or halted due to an early fault.	1. Use QEMU's `-d cpu_reset` and `info registers` to see if `kmain` is reached. Check `EIP`. 2. Use `objdump -d` on your kernel binary to verify the expected code at the entry point. 3. Add a `serial_puts("A")` at the very start of `kmain` to test if code runs.	1. Ensure `vga_write` uses `VGA_BUFFER_ADDR` (0xB8000). 2. Verify the bootloader loads the kernel at the address specified in your linker script. 3. Check for a missing `cli` (disable interrupts) before setting up protected mode.
System immediately reboots (Triple Fault) after boot.	1, 2, 3	1. **GDT misconfiguration:** Segment selectors point to invalid memory or privilege levels. 2. **IDT misconfiguration:** An exception (e.g., #GP, #PF) occurs but the IDT entry is invalid, causing a double fault, then a triple fault. 3. **Stack pointer (`ESP`) is invalid** when entering protected mode or calling a function.	1. Check the GDT limit and base in `gdtr`. Use QEMU's `info gdt`. 2. Use QEMU's `-d int` to log every interrupt/exception. The last logged exception before the reset is the culprit. 3. Print or inspect the `ESP` value right after the `lgdt` instruction and before the far jump.	1. Ensure GDT entries have correct base (0), limit (0xFFFFF), and granularity (4KB) for a flat model. 2. Fill all 256 IDT entries, even if with a generic handler. Ensure `type_attr` includes `IDT_PRESENT` and `IDT_GATE_32`. 3. Set `ESP` to a known valid address (e.g., end of a stack array) before any `call` or `push`.
Characters appear on screen, but are gibberish or at wrong positions.	1	1. **VGA character/attribute byte order is reversed.** The VGA buffer expects character byte then attribute byte for each cell. 2. **Newline (`\n`) handling is missing** in `kprintf`, causing characters to wrap incorrectly. 3. Writing beyond the VGA buffer bounds, corrupting other memory.	1. Write a simple test that prints a single 'A' with attribute 0x0F (white on black) and inspect memory at 0xB8000. 2. Print a string containing a newline and observe if the cursor moves to the next line.	1. Ensure buffer indexing: `buffer[index] = character; buffer[index+1] = attribute;`. 2. Implement newline handling: move cursor to beginning of next line (`pos += VGA_WIDTH*2 - (pos % (VGA_WIDTH*2))`). 3. Add bounds checking: `if (pos >= VGA_BUFFER_SIZE) { scroll_screen(); }`.
Keyboard presses are not detected or produce wrong characters.	2	1. **PIC not remapped** or **EOI not sent** in keyboard ISR, blocking further interrupts. 2. Keyboard ISR not reading the scancode from port `0x60`. 3. Scancode to ASCII mapping is incorrect (e.g., not handling key release scancodes).	1. Check if other interrupts (e.g., timer) also don't work. If so, PIC is likely the issue. 2. In the ISR, read `arch_port_read_byte(0x60)` and log it via serial. Verify you see scancodes on press (e.g., 0x1E for 'A'). 3. Check if scancodes with bit 7 set (release) are being incorrectly mapped.	1. Call `pic_remap()` before enabling interrupts. In the ISR, send `EOI` to both PIC1 and PIC2 if IRQ >= 8. 2. Ensure the ISR reads the data port, even if just to discard it. 3. Mask the release bit: `scancode & 0x7F`. Use a lookup table for set 1 scancodes.
Timer interrupt fires, but context switches don't happen.	4	1. **Scheduler not initialized** or ready queue is empty. 2. **Context switch assembly (`switch_context`) does not save/restore all critical registers** (including `EFLAGS`, segment registers). 3. **Page tables not switched** during context switch, so new process runs in old process's address space.	1. Log when `schedule()` is called. Check if `scheduler_get_next_process()` returns a valid `pcb`. 2. Single-step through `switch_context` in Bochs, comparing register values before and after the switch. 3. Check `CR3` before and after the switch; it should change to the new process's `page_directory`.	1. Ensure `process_create` adds the new PCB to the scheduler's ready queue and that `scheduler_init` is called. 2. Verify `switch_context` saves/restores at least: `EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI, EFLAGS, EIP, CS, SS, DS, ES, FS, GS`. 3. In `switch_context`, load the new process's `page_directory` into `CR3` and perform a `jmp` to the new `EIP`.
Kernel heap allocations fail or cause corruption.	3	1. **Heap not initialized** (`heap_init` not called) or initialization incorrectly calculates available memory. 2. **Heap block metadata (`heap_block_header/footer`) corrupted** by buffer overflows in user code.	1. After `heap_init`, log the heap start and end addresses. Verify they are within mapped kernel memory. 2. Add `HEAP_MAGIC` values to headers/footers and check them in `kmalloc` / `kfree`. Log any corruption.	1. Call `heap_init` after `pmm_init` and `vmm_init`. Ensure the heap virtual region is mapped to physical frames. 2. Use **boundary tags** and validate the magic numbers on every

Symptom	Likely Milestone	Probable Cause	Diagnostic Steps	Fix
		3. **Fragmentation or coalescing bug** causes `kmalloc` to return overlapping or invalid addresses.	3. Write a test (`test_heap_stress`) that performs many random allocations and frees, checking for overlaps and leaks.	operation. Add **guard pages** around heap blocks in debug builds. 3. Implement **splitting** of large free blocks and **coalescing** of adjacent free blocks when `kfree` is called.
Page fault exception (#PF) occurs, often in an unexpected location.	3, 4	1. **Page table entry missing `PTE_PRESENT` flag** for a required kernel or user address. 2. **Accessed a user-mode pointer from kernel mode** without proper validation or temporary mapping. 3. **Did not flush the TLB** after modifying a page table entry, causing the CPU to use a stale translation.	1. In the page fault handler (`handle_exception`), examine `CR2` (faulting address) and the error code. Log them. 2. Check if the faulting address is in user space (>=0x40000000) while executing in kernel mode (CPL=0). 3. After any `map_page` or page flag change, call `flush_tlb_single` or reload `CR3` .	1. Ensure all kernel code/data is mapped. For a **Higher-Half Kernel**, the linker script must place kernel at virtual addresses >= 0xC0000000, and early boot must map them. 2. Use `is_user_pointer` to validate user pointers before dereferencing. For system calls, copy data from user to kernel buffer. 3. Invalidate TLB entries after modifying page tables. Use `invlpg` instruction for single addresses or reload `CR3` for full flush.
System calls from user processes crash or return wrong results.	4	1. **Incorrect system call convention:** User process does not set up registers properly before `int 0x80` . 2. **Kernel system call dispatcher does not correctly extract arguments** from the user stack or registers. 3. **Kernel accesses user memory directly** without validation, causing a page fault.	1. In the `syscall_dispatcher` , log the `syscall_num` and arguments received. Compare with what the user intended. 2. Single-step the `int 0x80` handler in Bochs to see how the saved registers are placed on the kernel stack. 3. Check if the page fault occurs inside a system call handler when trying to read `arg1` as a pointer.	1. Define a convention: e.g., system call number in `EAX` , args in `EBX, ECX, EDX, ESI, EDI` . Document it and ensure user libc follows it. 2. In the ISR stub, push all registers to create a `registers` struct. The dispatcher can then access `regs->eax` , `regs->ebx` , etc. 3. Use `is_user_pointer` to check user pointers, then use `copy_from_user` (which handles page faults gracefully) to copy data to kernel buffers.
Only one process runs; scheduler never preempts it.	4	1. **Timer interrupt is disabled** (`cli` or masked in PIC) after the first process starts. 2. **Process never yields** and timer interrupt handler doesn't call `schedule()` . 3. **Scheduler's ready queue is corrupted** (e.g., `pcb->next` pointer is invalid).	1. Verify timer interrupts are firing by incrementing a counter in the timer ISR and logging it periodically. 2. Check if the timer ISR calls `schedule()` and that `schedule` performs a context switch. 3. Traverse the ready queue from `scheduler_get_next_process` and log each PID. Check for infinite loops.	1. Ensure `sti` is executed after initializing the scheduler and launching the first process. Do not leave interrupts disabled. 2. In the timer ISR, after sending EOI, call `schedule()` to potentially switch processes. 3. Use a **linked list** with a sentinel node for the ready queue. Validate `next` pointers when adding/removing processes.

Tools and Techniques: QEMU, Bochs, and Logging

When physical hardware is unresponsive, emulators become your primary window into the kernel's soul. Coupled with strategic logging, they allow you to observe execution, inspect CPU state, and trace memory without affecting the system's operation.

Emulator Debuggers: Your Microscope

QEMU is fast and convenient for quick iteration, while **Bochs** is slower but provides extremely detailed, deterministic simulation and a more intuitive built-in debugger. Use both: QEMU for rapid development and Bochs for deep, complex bugs.

QEMU Command-Line Debugging: Start QEMU with debug options: `qemu-system-i386 -kernel myos.kernel -serial stdio -d cpu_reset,int,guest_errors -no-reboot -s -S`

- `-s` : Shorthand for `-gdb tcp::1234` , waits for a GDB connection.
- `-S` : Freeze CPU at startup, allowing you to attach a debugger before any code runs.

- `-d` : Log specific events. `cpu_reset` shows each CPU reset, `int` logs every interrupt/exception, `guest_errors` logs CPU faults.

Using GDB with QEMU:

1. In one terminal: `qemu-system-i386 -kernel myos.kernel -s -S`
2. In another: `gdb`

```GDB
(gdb) target remote localhost:1234
(gdb) file myos.kernel          # Load your kernel's symbol file
(gdb) break kmain               # Set breakpoint at kernel main
(gdb) continue                  # Start execution
```

You can now step through assembly (`stepi`), inspect registers (`info registers`), and examine memory (`x/10x 0xB8000`). This is invaluable for watching the exact moment a register takes a wrong value.

Bochs Built-in Debugger: Bochs includes a powerful console debugger. Configure it in your `bochsrc.txt` :

```
magic_break: enabled=1
debug_symbols: file=myos.sym
```

When Bochs starts, press `Ctrl+C` to break into the debugger. Useful commands:

- `s` or `step` : Execute a single instruction.
- `c` : Continue execution.
- `break 0x100000` : Set a breakpoint at a physical address.
- `info gdt` , `info idt` , `info tab` : Display CPU tables.
- `x /10 0xB8000` : Examine 10 words at VGA buffer.
- `trace-reg on` : Log all register changes, then `trace-reg off` and `show trace` to see history.

Bochs's **Magic Breakpoint** is particularly useful: insert `xchg bx, bx` (opcode `0x87 0xDB`) into your assembly code. When Bochs encounters this instruction, it automatically breaks into the debugger, acting as a software breakpoint that requires no debugger attachment.

Strategic Logging: The Kernel's Black Box

When a debugger is too intrusive or you need to understand a sequence of events, logging is your best friend. Since the screen may be unreliable, the **serial port (UART)** provides a robust, out-of-band channel to emit diagnostic messages.

Setting Up Serial Logging:

1. Early in `kmain` , call `serial_init()` to configure COM1 at 9600 baud.
2. Implement `serial_puts` to send strings.
3. Create a `klog` function that formats a message and sends it via serial (and optionally to screen).

You can then capture this output:

- With QEMU: `qemu-system-i386 -kernel myos.kernel -serial file:serial.log`
- With Bochs: Configure `com1: enabled=1, mode=file, dev=serial.log` in `bochsrc` .

What to Log:

- **Function Entry/Exit:** `klog("ENTER: kmain")` and `klog("EXIT: kmain")` .
- **Critical Values:** `klog("CR3=%x", cr3_value)` when setting up paging.
- **Interrupts:** `klog("IRQ %d fired", irq_num)` in your ISR.
- **Heap Operations:** `klog("kmalloc(%d) -> %p", size, ptr)` .
- **Scheduler Decisions:** `klog("Schedule: switching from pid %d to %d", old_pid, new_pid)` .

By strategically placing these logs, you can reconstruct the exact path of execution leading to a crash. If the system triple faults, the last message in the serial log is your most important clue.

Post-Mortem Debugging with a Crash Dump: For unrecoverable errors, implement a `panic` function that:

1. Disables interrupts.
2. Saves all CPU registers and critical control registers (`CR0-CR4`) into a `crash_dump` structure at a known physical address (e.g., `CRASH_DUMP_ADDR = 0x1000`).
3. Prints the dump to serial and the screen.
4. Halts the CPU (`cli; hlt;`).

You can then examine this dump from the emulator after the crash by inspecting memory at the dump address. This gives you a snapshot of the system state at the moment of failure.

Implementation Guidance

This section provides concrete code to establish a robust debugging infrastructure for your kernel.

Technology Recommendations Table:

Component	Simple Option	Advanced Option
Logging	Serial port (COM1) output to a file.	In-memory ring buffer that persists across panics, dumped on next boot.
Debugger Interface	QEMU + GDB remote protocol.	Integrated monitor commands over serial port (e.g., examine memory, set breakpoints).
Crash Analysis	`panic()` with register dump to screen and serial.	Kernel core dump written to a reserved disk sector for offline analysis.

Recommended File/Module Structure:

```
src/
  kernel/
    main.c                # kmain, panic
    debug/                # Debugging utilities
      serial.c           # serial_init, serial_putc, serial_puts
      logging.c          # klog, kprintf (if not in main)
      panic.c            # panic, crash_dump structure
    arch/i386/           # Architecture-specific debug
      gdb_stub.c         # Optional GDB stub implementation
```

Infrastructure Starter Code (Serial Port Driver): The following is a complete, working serial port driver for COM1. Place it in `debug/serial.c`.

```c
// debug/serial.c

#include <stdint.h>

#include "debug/serial.h"


#define PORT_COM1 0x3F8


void serial_init() {

    // Disable interrupts

    arch_port_write_byte(PORT_COM1 + 1, 0x00);

    // Enable DLAB (set baud rate divisor)

    arch_port_write_byte(PORT_COM1 + 3, 0x80);

    // Set divisor low byte (115200 / 9600 = 12)

    arch_port_write_byte(PORT_COM1 + 0, 0x0C);

    arch_port_write_byte(PORT_COM1 + 1, 0x00);

    // 8 bits, no parity, one stop bit

    arch_port_write_byte(PORT_COM1 + 3, 0x03);

    // Enable FIFO, clear them, with 14-byte threshold

    arch_port_write_byte(PORT_COM1 + 2, 0xC7);

    // Enable interrupts (optional) - leave disabled for simple logging

    // arch_port_write_byte(PORT_COM1 + 4, 0x0B);

}


int serial_is_transmit_fifo_empty() {

    return arch_port_read_byte(PORT_COM1 + 5) & 0x20;

}


void serial_putc(char c) {

    while (!serial_is_transmit_fifo_empty());

    arch_port_write_byte(PORT_COM1, c);

}


void serial_puts(const char* str) {

    while (*str) {

        serial_putc(*str++);

    }

}
```

Core Logic Skeleton Code (Panic and Logging): The following skeleton provides the structure for a `panic` function and a simple logging utility. Fill in the TODOs.

```c
// debug/panic.c

#include <stdint.h>

#include "debug/panic.h"

#include "debug/serial.h"

#include "drivers/vga.h"


// The crash dump structure at a known physical address

volatile struct crash_dump* system_crash_dump = (struct crash_dump*)CRASH_DUMP_ADDR;


void panic(const char* file, int line, const char* msg) {

    // TODO 1: Disable interrupts immediately to stop any further activity

    // Use `__asm__ volatile("cli");`


    // TODO 2: Save CPU registers into the crash_dump structure

    // You will need assembly to capture registers. Consider writing a small

    // assembly function `capture_registers(struct crash_dump *dump)` that

    // saves EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP, EIP, EFLAGS, CS, DS, ES, FS, GS, SS.

    // Also save control registers CR0, CR2, CR3, CR4.

    // Set dump->magic = CRASH_MAGIC.


    // TODO 3: Print panic message to serial and VGA screen

    // Use `serial_puts("PANIC: ")` and `vga_puts` (or direct VGA write).

    // Include file, line, and msg.


    // TODO 4: Dump the saved registers to serial port

    // Iterate through the crash_dump and print each register value in hex.


    // TODO 5: Halt the CPU indefinitely

    // Use `__asm__ volatile("hlt");` in a loop.

}


// debug/logging.c

#include "debug/logging.h"

#include "debug/serial.h"

#include <stdarg.h>


void klog(const char* fmt, ...) {

    // TODO 1: Optionally add a timestamp (if timer is available)


    // TODO 2: Use va_list, vsprintf (you need to implement a simple vsprintf) to format the string into a buffer


    // TODO 3: Send the formatted string to serial_puts


    // TODO 4: Optionally also print to VGA screen for immediate visibility

}
```

Language-Specific Hints (C):

- Use `__attribute__((packed))` for the `crash_dump` and `registers` structures to ensure the layout in memory matches your assembly expectations.
- Inline assembly is compiler-specific. For GCC, use `__asm__ volatile()` syntax. Capture registers by listing them as outputs (e.g., `"=a"(regs->eax)`).
- Implement a simple `vsprintf` for `klog` that handles `%d`, `%x`, `%s`, and `%c`. Avoid floating point.

Milestone Checkpoint for Debugging Infrastructure: After implementing serial logging and `panic`:

1. **Run:** `make run` (which should start QEMU with `-serial file:serial.log`).
2. **Expected Output:** In `serial.log`, you should see any `klog` messages from your kernel's initialization.
3. **Test Panic:** Intentionally cause a panic early in `kmain` (e.g., `panic(__FILE__, __LINE__, "Test panic");`).
4. **Verify:** The serial log should contain the panic message and a register dump. The emulator should halt (not reboot).
5. **Signs of Trouble:**
   - No serial output: Check that `serial_init` is called and that the I/O ports are correct for your emulator's COM1 emulation.
   - Garbage in serial log: Baud rate mismatch. Ensure divisor set correctly for 9600 baud.
   - Panic doesn't halt: Ensure interrupts are disabled and the `hlt` loop is entered.

Debugging Tips for the Debugging Infrastructure:

Symptom	Likely Cause	How to Diagnose	Fix
Serial log remains empty.	1. Serial port not initialized. 2. QEMU not capturing serial output.	Add a `serial_puts("HELLO")` as the very first line in `kmain`. Check QEMU command line for `-serial`.	Ensure `serial_init` is called before any log. Use `-serial stdio` to see output directly in terminal.
Panic register dump shows zeros.	The `capture_registers` assembly does not correctly save values to memory.	Single-step the `capture_registers` function in Bochs. Check that the memory address (`CRASH_DUMP_ADDR`) is being written to.	Verify the assembly saves to the correct offset within the struct. Use `objdump` to see the generated instructions.
Using `klog` causes a page fault.	The `klog` function or `vsprintf` uses an invalid pointer or exceeds a local buffer.	Comment out `klog` calls one by one to isolate the faulty call. Check that format string and arguments are valid.	Ensure `vsprintf` does not write beyond its buffer. Use a static buffer of fixed size (e.g., 256 chars).

10. Future Extensions

> **Milestone(s):** This section looks beyond the four core milestones, exploring how the minimal kernel can be extended with additional features and architectural changes to become a more complete operating system. The completed kernel provides the essential substrate—processes, memory, and hardware abstraction—upon which many traditional operating system features can be built.

Completing the four milestones yields a functional but minimal kernel capable of running multiple isolated processes with basic console I/O. This foundation, while educationally complete, represents just the beginning of what a modern operating system can offer. This section explores two categories of future work: **Feature Additions** that extend functionality within the existing monolithic architecture, and **Architectural Evolutions** that fundamentally change the kernel's design philosophy or capabilities. Each extension presents its own set of design trade-offs and implementation challenges, serving as excellent independent study projects for deepening one's understanding of operating systems.

Feature Additions

Once the core kernel is stable, the most natural next step is to add user-facing features and system services that transform the bare-bones kernel into a usable platform. Think of the current kernel as a house with a solid foundation, walls, and roof, but lacking furniture, plumbing, and electrical outlets. Feature additions install these utilities, making the space livable and functional.

Shell and User Interface

Mental Model: The Hotel Concierge A shell acts as the concierge for your operating system. Guests (users) arrive at the lobby (console) and tell the concierge what they need—run a program, list files, or manage processes. The concierge interprets these requests, dispatches the appropriate services (kernel system calls), and presents the results back to the guest. A graphical user interface (GUI) would be like adding a touch-screen kiosk next to the concierge, providing a more intuitive but resource-intensive way to interact with the same underlying services.

The first step is implementing a basic command-line shell. This requires extending the system call interface with file operations and process control, then building a user-mode program that reads commands, parses them, and invokes the appropriate system calls.

Decision: Shell Design Philosophy

- **Context**: We need an interactive interface for users to launch programs and manage the system. The kernel currently only supports loading static processes at boot via `process_create`.
- **Options Considered**:
  1. **Built-in Kernel Shell**: Implement command parsing and execution directly in the kernel, accessible via the console.
  2. **User-Mode Shell Process**: Create a shell as a regular user process that uses system calls to perform operations.
  3. **Hybrid Approach**: Kernel provides minimal built-in commands (like `help`), with extensibility via user-mode programs.
- **Decision**: User-Mode Shell Process.
- **Rationale**: A user-mode shell reinforces the process isolation model and keeps the kernel minimal. It treats the shell as just another process, which can be killed and restarted without crashing the system. This also allows for multiple different shells to coexist.
- **Consequences**: Requires a more complete system call interface (e.g., `exec`, `waitpid`), a filesystem to store shell binaries, and a way to load programs from disk dynamically. The kernel becomes more complex, but the architecture remains cleanly layered.

Option	Pros	Cons	Viability
Built-in Kernel Shell	Simple to implement; direct kernel access	Violates process isolation; shell bugs crash kernel; not extensible	Poor—contradicts educational goals
User-Mode Shell Process	Reinforces kernel/process separation; allows multiple shells; fault-tolerant	Requires dynamic program loading and richer system calls	**Chosen**—aligns with architecture
Hybrid Approach	Some commands always available	Inconsistent design; still requires user-mode shell for extensibility	Medium—adds complexity without clear benefit

To implement a shell, you would first need to add system calls for process creation with argument passing (`sys_execve`), process waiting (`sys_waitpid`), and possibly terminal control (`sys_ttyctl`). The shell process itself would be a simple loop that:

1. Prints a prompt using `kprintf` (or a user-mode `write` to stdout).
2. Reads a line of input from the keyboard (via existing driver).
3. Parses the line into command and arguments.
4. Uses `sys_fork` to create a child process.
5. The child uses `sys_execve` to replace its image with the requested program.
6. The parent uses `sys_waitpid` to wait for the child to finish.
7. Repeat.

A minimal shell might support commands like `ls`, `cat`, and `echo`, each implemented as separate user-mode programs. This naturally leads to the next major feature: a filesystem.

Filesystem

Mental Model: The Library Catalog System A filesystem is like a library's catalog system. Books (files) are stored on shelves (disk blocks). The catalog (directory structure and metadata) records each book's title, author, location, and checkout status. When you request a book, the librarian (filesystem driver) consults the catalog, navigates to the correct shelf, retrieves the book, and hands it to you. The kernel's role is to provide the librarian with secure access to the library and ensure two patrons don't try to modify the same catalog entry simultaneously.

Adding a filesystem requires several new components: a disk driver (ATA or AHCI), a filesystem driver (e.g., for a simple custom FS or FAT32), and corresponding system calls (`open`, `read`, `write`, `close`). The virtual filesystem (VFS) layer is a common abstraction that allows multiple filesystem types to coexist.

Decision: Initial Filesystem Choice

- **Context**: We need persistent storage for programs and data. The kernel currently has no disk I/O capability.
- **Options Considered**:
  1. **Custom Simple Filesystem**: Design a minimalist filesystem (e.g., a flat directory with fixed-size files) tailored for simplicity.
  2. **FAT32**: Implement the widely-documented FAT32 specification, enabling interoperability with existing tools.
  3. **In-Memory RAM Disk**: Start with a non-persistent filesystem in RAM to defer hardware complexity.
- **Decision**: Start with a RAM disk and a custom simple filesystem, then add FAT32 later.
- **Rationale**: A RAM disk avoids the initial complexity of disk drivers and DMA, allowing the filesystem logic (directory structures, block allocation, caching) to be developed and tested in isolation. A custom simple design keeps the initial implementation manageable. FAT32 can be added as a second driver once the VFS layer is stable.
- **Consequences**: Data is lost on reboot, but development is faster. The VFS interface must be designed to accommodate multiple filesystem types from the start.

Option	Pros	Cons	Viability
Custom Simple FS	Complete control; can be optimized for kernel needs	Non-standard; requires custom tools to create disk images	**Chosen (initial)**—educational value
FAT32	Standard format; tools exist; can read/write from host OS	Complex specification; patents/licensing considerations	**Chosen (later)**—for interoperability
RAM Disk	No hardware dependencies; fast	Volatile storage; limited capacity	**Chosen (scaffolding)**—for early development

The implementation sequence would be:

1. Implement a RAM disk driver that allocates a contiguous memory region.
2. Design a simple filesystem layout: a superblock, a block bitmap, an inode table, and data blocks.
3. Implement the core VFS operations: `vfs_open`, `vfs_read`, `vfs_write`, `vfs_close`.
4. Create a filesystem driver that implements these operations for your simple FS.
5. Add system calls that wrap VFS operations.
6. Later, write an ATA driver using PIO (Programmed I/O) mode, then DMA.
7. Implement the FAT32 driver, registering it with the VFS.

Networking (TCP/IP Stack)

Mental Model: The Postal Service A networking stack is like an international postal service. An application writes a letter (data packet) with an address (IP and port). The kernel's TCP/IP stack acts as the postal service, adding envelopes (headers), routing the letter through sorting facilities (routing tables), and ensuring delivery with acknowledgments (TCP ACKs). The network interface card (NIC) is the local post office that physically dispatches and receives mail. Just as postal services handle lost letters and out-of-order delivery, the networking stack must manage packet loss, retransmission, and congestion.

Adding networking transforms the kernel from a standalone system into one that can communicate with the outside world. This is a substantial project involving multiple layers: NIC driver (e.g., for Intel E1000), Ethernet frame handling, IP packet routing, TCP connection management, and sockets API.

> **Decision: Network Stack Scope**
>
> - **Context**: We want to enable network communication for applications (e.g., a web server, remote shell).
> - **Options Considered**:
>   1. **Minimal UDP/IP Only**: Implement just Ethernet, ARP, IP, and UDP. Avoid the complexity of TCP.
>   2. **Full TCP/IP Stack**: Implement a complete TCP/IP stack including TCP with congestion control.
>   3. **LWIP Integration**: Port the lightweight IP (LWIP) stack, a mature open-source implementation.
> - **Decision**: Start with a minimal UDP/IP stack, then incrementally add TCP.
> - **Rationale**: UDP is connectionless and stateless, making it far simpler to implement correctly. It provides immediate utility for simple services (DNS, custom protocols) and allows testing of the lower layers (driver, IP). TCP can be added later as a major sub-project. Porting LWIP is viable but reduces the educational value of implementing the protocols oneself.
> - **Consequences**: Applications requiring reliable streams (like HTTP) won't work until TCP is complete. However, the core networking infrastructure (driver, buffer management, IP routing) will be battle-tested by UDP.

Option	Pros	Cons	Viability
Minimal UDP/IP	Achievable scope; tests core infrastructure	No reliable streams; limited application use	**Chosen**—foundational step
Full TCP/IP	Full functionality for most applications	Very complex; high risk of subtle bugs	Long-term goal
LWIP Integration	Robust, feature-complete; faster development	Less educational value; potential licensing issues	Good for production-oriented projects

The implementation would proceed layer-by-layer:

1. Write a NIC driver for your emulated hardware (e.g., QEMU's `e1000` or `rtl8139`).
2. Implement Ethernet frame reception/transmission with DMA ring buffers.
3. Add ARP (Address Resolution Protocol) to map IP addresses to MAC addresses.
4. Implement IP packet fragmentation/reassembly and a simple routing table.
5. Build UDP sockets, including a system call interface (`socket`, `sendto`, `recvfrom`).
6. (Later) Implement TCP with finite state machine, sequence numbers, sliding window, and basic congestion control (Tahoe or Reno).

Advanced Scheduling

Mental Model: The Air Traffic Control Tower Advanced scheduling is like an air traffic control tower managing arrivals and departures. Simple round-robin treats every plane equally, giving each a turn on the runway. Advanced schedulers act like controllers prioritizing emergency flights (high priority), allowing long-haul flights

to stay in the air longer (longer time slices for CPU-bound tasks), and keeping short regional flights moving quickly (short time slices for I/O-bound tasks). They also dynamically adjust based on weather (system load) and fuel levels (process aging).

The current round-robin scheduler is fair but naive. Real-world schedulers optimize for responsiveness (low latency for interactive tasks) and throughput (efficient use of CPU for batch tasks). Implementing an advanced scheduler is a pure-kernel project that doesn't require new hardware drivers.

> **Decision: Scheduler Algorithm Upgrade**
>
> - **Context**: The simple round-robin scheduler treats all processes equally, which is suboptimal for mixed workloads.
> - **Options Considered**:
>   1. **Priority-Based Round-Robin**: Assign static or dynamic priorities; higher priority processes run more frequently.
>   2. **Multilevel Feedback Queue (MLFQ)**: The classic interactive scheduler that automatically classifies processes as I/O-bound or CPU-bound.
>   3. **Completely Fair Scheduler (CFS) Inspired**: Use a virtual runtime (vruntime) model to approximate ideal fairness.
> - **Decision**: Implement a Multilevel Feedback Queue (MLFQ).
> - **Rationale**: MLFQ is well-documented in operating systems literature (see OSTEP), provides excellent interactivity for I/O-bound processes, and automatically adapts to process behavior without requiring user-specified priorities. It's a significant step up from round-robin while being more approachable than CFS.
> - **Consequences**: Requires maintaining multiple ready queues, implementing aging to prevent starvation, and carefully tuning time slice lengths and priority boost intervals. The `pcb` structure needs new fields for tracking time used and current priority level.

Option	Pros	Cons	Viability
Priority Round-Robin	Simple to implement; intuitive	Requires manual priority assignment; can starve low-priority tasks	Moderate—limited improvement
Multilevel Feedback Queue	Self-tuning; good for interactive tasks; classic algorithm	More complex; requires careful parameter tuning	**Chosen**—great educational value
CFS-Inspired	Very fair; Linux-proven	Complex data structures (red-black trees); subtle vruntime accounting	Advanced—significant challenge

To implement MLFQ, you would:

1. Define 3-5 priority levels, each with its own ready queue and time quantum (higher priority = shorter quantum).
2. Modify the `pcb` to include `priority_level`, `time_used_current_slice`, and `total_cpu_time`.
3. On each timer interrupt, increment `time_used_current_slice`. If it exceeds the quantum for that level, move the process to a lower priority queue and reset the timer.
4. Periodically (e.g., every second) boost all processes to the highest priority queue to prevent starvation (aging).
5. Modify `scheduler_get_next_process` to always check the highest non-empty queue first.
6. I/O-bound processes that voluntarily yield (e.g., blocking on `read`) could retain their high priority.

Architectural Evolutions

Beyond adding features, you might reconsider the fundamental architecture of the kernel itself. Architectural changes are like renovating the house's floor plan—moving walls, adding staircases, or even converting a single-family home into apartment units (microkernel). These changes are profound and affect every component.

Evolving Towards a Microkernel

Mental Model: From Monolithic Castle to Microkernel City-State Our current monolithic kernel is like a medieval castle: all services (kitchen, armory, barracks) are inside the same fortified walls. This is efficient (short distances) but a single fire (bug) can engulf the entire structure. A microkernel is like a modern city-state: the central government (microkernel) provides only the most essential services (process scheduling, IPC, basic memory management), while other services (filesystem, networking, device drivers) run as separate user-mode "municipalities." Communication happens via formal messages (IPC). This improves fault isolation—a crashing filesystem doesn't take down the network—but adds overhead (message passing).

Converting a monolithic kernel to a microkernel is a major redesign. It involves extracting core services into separate server processes and replacing direct function calls with inter-process communication (IPC).

Decision: Microkernel Migration Strategy

- **Context**: We want to improve fault isolation and modularity by moving services out of kernel space.
- **Options Considered**:
  1. **Big-Bang Rewrite**: Discard the monolithic kernel and implement a new microkernel from scratch.
  2. **Incremental Extraction**: Gradually move subsystems (first filesystem, then network) into user-mode servers, replacing internal calls with IPC.
  3. **Hybrid Kernel**: Keep some subsystems in kernel for performance (e.g., network driver) while moving others to user space.
- **Decision**: Incremental Extraction.
- **Rationale**: A full rewrite loses the working foundation and is high-risk. Incremental extraction allows testing at each step and preserves the ability to boot and run. It also provides a clear learning path: first implement a robust IPC mechanism, then move one subsystem at a time.
- **Consequences**: The kernel will temporarily become a hybrid during transition, which can be messy. Performance will degrade due to IPC overhead, but reliability should improve. The `syscall_dispatcher` will need to route some system calls to external servers via IPC.

Option	Pros	Cons	Viability
Big-Bang Rewrite	Clean design; no legacy constraints	Massive effort; lose existing functionality; high risk	Poor for a learning project
Incremental Extraction	Leverages existing code; testable steps; educational	Temporary hybrid state; complex transition	**Chosen**—pragmatic and educational
Hybrid Kernel	Balance of performance and isolation	Inconsistent design; hardest to reason about	Common in real-world (Windows, XNU)

The migration would follow these high-level steps:

1. **Design and Implement IPC**: Create a fast, secure mechanism for processes to exchange messages. This likely involves shared memory regions for bulk data and system calls for synchronization (`send` , `receive` , `notify`).
2. **Extract the Filesystem**:
   - Modify the VFS layer to become an IPC client.
   - Create a filesystem server process that holds the actual FS driver.
   - Redirect `sys_open` , `sys_read` etc. to send IPC requests to the server.
   - The server runs with elevated capabilities but in user mode.
3. **Extract the Network Stack**: Similarly, turn the network stack into a server. The NIC driver remains in kernel (or could be a separate driver server).
4. **Extract Device Drivers**: Move keyboard, VGA, and disk drivers into separate driver servers.
5. **The Microkernel Core**: Eventually, the kernel is left with only scheduler, memory manager, IPC, and basic hardware abstraction (interrupt handling). All other services are external.

This evolution teaches the principles of message-passing, capability-based security, and system design for reliability.

Adding Symmetric Multiprocessing (SMP) Support

Mental Model: From Single Chef to Kitchen Brigade A uniprocessor kernel is like a solo chef preparing all dishes sequentially. SMP support turns the kitchen into a brigade system: multiple chefs (CPU cores) work simultaneously, each with their own station (local cache), coordinated by a head chef (kernel). They share access to pantries (memory) and must avoid collisions (two chefs grabbing the same ingredient). Synchronization primitives (locks) are like tickets for exclusive use of a resource.

Adding SMP support allows the kernel to schedule processes on multiple CPU cores concurrently, dramatically increasing potential throughput. This requires fundamental changes to almost every kernel subsystem to handle concurrency correctly.

Decision: SMP Initialization and Locking Strategy

- **Context**: We want to support multi-core CPUs to run multiple processes truly in parallel.
- **Options Considered**:
  1. **BSP Boots APs Symmetrically**: The Bootstrap Processor (BSP) activates Application Processors (APs) and gives them identical kernel code.
  2. **Asymmetric Multiprocessing**: Designate one core for kernel tasks, others for user processes only.
  3. **Big Kernel Lock (BKL)**: Use a single global lock to protect all kernel data structures—simple but limits parallelism.
  4. **Fine-Grained Locking**: Protect each major data structure (e.g., process table, heap) with its own lock.
- **Decision**: Start with BKL for simplicity, then evolve to fine-grained locking.
- **Rationale**: The BKL is a pragmatic first step: it ensures correctness on SMP by allowing only one core in kernel mode at a time. This allows the AP startup and inter-processor interrupt (IPI) mechanisms to be implemented and tested without the complexity of concurrent data structure access. Once SMP boots stably, fine-grained locks can be introduced incrementally.
- **Consequences**: Performance will not scale with cores initially (Amdahl's law). However, the system will be functionally correct and provide a foundation for later optimization. The locking must be carefully placed around any shared kernel data.

Option	Pros	Cons	Viability
BKL	Simple to implement; guarantees correctness	Severe contention; poor scalability	**Chosen (initial)**—safety first
Fine-Grained Locking	Better parallelism; modern approach	Complex; deadlock risk; difficult to debug	**Chosen (evolution)**—long-term goal
Asymmetric MP	Simplifies kernel concurrency	Wastes cores; not true SMP	Niche use (real-time systems)

The implementation sequence:

1. **Detect and Initialize APs**:
   - Use the ACPI or MP tables to discover CPU cores.
   - The BSP sends Startup IPIs (Inter-Processor Interrupts) to APs, directing them to a startup trampoline code in low memory.
   - APs initialize their local APIC, load the GDT/IDT, enable paging, and jump into a common kernel entry point.
2. **Implement BKL**: A global spinlock (`kernel_lock`) acquired on entry to any kernel code (system calls, interrupts). Use `atomic` operations to test-and-set.
3. **Per-CPU Data Structures**:
   - Create a per-CPU structure holding the current `pcb` , kernel stack, and idle thread.
   - Modify `scheduler_get_next_process` to have a per-CPU ready queue or a shared queue with locking.
4. **IPI for Scheduling**: When a core makes a process ready, it may need to send an IPI to another core to trigger a reschedule.
5. **Locking Evolution**: Gradually replace BKL with finer locks: a lock for the process table, a lock for the heap, a lock for each device driver, etc. This requires careful analysis of data access patterns.

SMP support is one of the most challenging but rewarding extensions, touching deep architectural aspects of concurrency and hardware coordination.

Implementation Guidance

Since this section discusses future extensions rather than core implementation, we provide high-level guidance and starter concepts rather than complete code.

Technology Recommendations Table

Component	Simple Option	Advanced Option
Shell	Built-in command loop in kernel (for quick testing)	User-mode shell with full argument parsing and job control
Filesystem	RAM disk with flat directory (no subdirectories)	FAT32 driver with VFS layer and disk caching
Networking	SLIP (Serial Line IP) over COM1 (avoids NIC driver)	Ethernet (E1000) with UDP/IP stack
Advanced Scheduling	Priority-based round-robin (static priorities)	Multilevel Feedback Queue (MLFQ) with aging
Microkernel IPC	Synchronous message passing via shared memory and system calls	Asynchronous capability-based IPC with kernel-mediated grants
SMP Support	Big Kernel Lock (BKL) with AP startup	Fine-grained locking with per-CPU runqueues and lock-free data structures

Recommended File/Module Structure for Extensions

```
project-root/
  kernel/
    fs/                    # Filesystem layer
      vfs.c                # Virtual Filesystem interface
      ramdisk.c            # RAM disk driver and simple FS
      fat32.c              # FAT32 driver (future)
    net/                   # Networking stack
      nic/                 # Network interface drivers
        e1000.c
      net.c                # Core networking (Ethernet, IP, UDP)
      tcp.c                # TCP implementation (future)
    servers/               # User-mode servers (microkernel)
      fs_server.c          # Filesystem server
      net_server.c         # Network server
    ipc/                   # Inter-Process Communication
      message.c            # Message passing primitives
      shared_mem.c         # Shared memory regions
    arch/x86/smp.c         # SMP startup and IPI handling
    include/kernel/        # Headers for new subsystems
      fs.h
      net.h
      ipc.h
      smp.h
  user/                    # User-mode programs
    shell/                 # Shell implementation
      shell.c
    commands/              # Basic utilities (ls, cat, echo)
      ls.c
      cat.c
      echo.c
```

Infrastructure Starter Code: Simple IPC Message Structure

For a microkernel extension, a simple synchronous IPC mechanism can start with this message structure:

```c
// ipc.h
#ifndef KERNEL_IPC_H
#define KERNEL_IPC_H

#include <stdint.h>

#define IPC_MAX_DATA_SIZE  256

typedef enum {
    IPC_MSG_OPEN,
    IPC_MSG_READ,
    IPC_MSG_WRITE,
    IPC_MSG_CLOSE,
    IPC_MSG_REPLY,
    // ... other message types
} ipc_msg_type_t;

typedef struct {
    uint32_t src_pid;          // Sender's PID
    uint32_t dst_pid;          // Destination PID (or 0 for kernel)
    ipc_msg_type_t type;
    uint32_t transaction_id;   // For matching request/reply
    uint32_t data_len;
    uint8_t data[IPC_MAX_DATA_SIZE];
} ipc_message_t;

// System calls for IPC
void ipc_send(uint32_t dst_pid, ipc_message_t* msg);
void ipc_receive(uint32_t* src_pid, ipc_message_t* msg);
void ipc_reply(uint32_t dst_pid, ipc_message_t* msg);

#endif // KERNEL_IPC_H
```

Core Logic Skeleton: Multilevel Feedback Queue Scheduler

```c
// scheduler.c (extension)
#define NUM_PRIORITY_LEVELS  5
#define TIME_QUANTUM_LEVEL0  10  // ticks for highest priority
#define TIME_QUANTUM_LEVEL1  20
#define TIME_QUANTUM_LEVEL2  40
#define TIME_QUANTUM_LEVEL3  80
#define TIME_QUANTUM_LEVEL4  160 // lowest priority

static struct pcb* ready_queues[NUM_PRIORITY_LEVELS];

void scheduler_init_mlfq(void) {
    for (int i = 0; i < NUM_PRIORITY_LEVELS; i++) {
        ready_queues[i] = NULL;
    }
    // TODO 1: Initialize all priority queues as empty linked lists
    // TODO 2: Set up a periodic timer (e.g., every 1000 ticks) for priority boost
}

struct pcb* scheduler_get_next_process_mlfq(void) {
    // TODO 1: Iterate from highest priority (0) to lowest (NUM_PRIORITY_LEVELS-1)
    // TODO 2: For each non-empty queue, remove the front process and return it
    // TODO 3: If all queues empty, return the idle process
    return NULL; // placeholder
}

void scheduler_enqueue_mlfq(struct pcb* proc) {
    // TODO 1: Validate proc->priority_level is within bounds
    // TODO 2: Add process to the tail of the appropriate ready_queues[priority]
    // TODO 3: Ensure process state is set to PROC_STATE_READY
}

void on_timer_tick_mlfq(struct pcb* current) {
    if (current == idle_process) return;

    // TODO 1: Increment current->time_used_current_slice
    // TODO 2: If time_used_current_slice >= quantum for its priority level:
    //   a) Move process to next lower priority queue (if not already lowest)
    //   b) Reset time_used_current_slice to 0
    //   c) Call scheduler_enqueue_mlfq to requeue it
    //   d) Set need_reschedule flag to trigger a context switch
    // TODO 3: Else, continue running current process
}
```

```
void priority_boost(void) {
    // Called periodically (e.g., every second)
    // TODO 1: Iterate through all processes in all queues
    // TODO 2: Move each process to the highest priority queue
    // TODO 3: Reset their time_used_current_slice to 0
    // This prevents starvation of CPU-bound processes
}
```

Language-Specific Hints (C)

- **Atomic Operations**: For SMP locks, use GCC built-ins like `__sync_lock_test_and_set` for spinlocks, or implement with inline assembly for x86 `lock cmpxchg`.
- **Alignment**: Ensure shared memory structures for IPC are cache-line aligned (`__attribute__((aligned(64)))`) to avoid false sharing on SMP.
- **Volatile**: Use `volatile` for memory-mapped I/O registers and variables accessed by multiple CPUs without explicit locking.
- **Packaging**: Consider compiling user-mode servers as standalone ELF binaries that the kernel loads like any other process.

Milestone Checkpoint for Shell

- **Test Command**: After implementing `sys_execve` and `sys_waitpid`, create a simple user program that prints "Hello from user mode!".
- **Expected Behavior**: From the kernel console, you should be able to type the program name and see its output appear on the next line, then return to the shell prompt.
- **Sign of Success**: The shell process survives the child's execution and can run another program.
- **Common Failure**: If the child process crashes, ensure you are properly switching page directories during `exec` and that user-mode stack is set up correctly.

Debugging Tips for Networking

Symptom	Likely Cause	How to Diagnose	Fix
No packets received	NIC not initialized or IRQ not enabled	Use QEMU monitor `info mem` to check MMIO region; check PIC/APIC masking	Verify BAR setup and enable IRQ in NIC config
Received packets but IP checksum fails	Byte order mismatch (endianness)	Dump raw Ethernet frame bytes via serial log; compare with Wireshark capture	Convert network byte order (ntohs/htons) for all multi-byte header fields
UDP packet sent but no reply	ARP failure (no MAC address for destination)	Check ARP cache; send a ping (ICMP echo) to trigger ARP	Implement ARP request on first packet; wait for reply before sending
TCP connection stuck in SYN_SENT	Firewall or no listening socket	Use `kprintf` to log TCP state transitions; check remote host is listening	Ensure your TCP SYN-ACK response has correct sequence numbers and flags

11. Glossary

> **Milestone(s):** This reference section provides definitions for key technical terms, acronyms, and concepts used throughout all four milestones, serving as a quick lookup for learners encountering unfamiliar terminology.

This glossary provides clear, concise definitions for the specialized terms, acronyms, and concepts used throughout this design document. Understanding this terminology is essential for comprehending the kernel's architecture and implementation details.

Terms and Acronyms

Term/Acronym	Definition
Application Processor (AP)	In a symmetric multiprocessing (SMP) system, any CPU core other than the Bootstrap Processor (BSP) that is started and initialized after the primary boot process.
BSS Section	The Block Started by Symbol section of an executable file containing statically-allocated variables that are initialized to zero by the program loader or kernel at runtime. The kernel entry point must explicitly zero this region.
Big Kernel Lock (BKL)	A single global spinlock that serializes all kernel-mode execution in early SMP systems, providing simple thread safety at the cost of limiting parallelism. Often replaced with finer-grained locking in modern systems.
Bitmap	A data structure consisting of an array of bits (0/1 values) used to efficiently track the allocation status of resources, most commonly physical memory frames where each bit represents whether a 4KB frame is free (0) or allocated (1).
Bootstrapping	The multi-stage process of loading and starting an operating system kernel from a powered-off state, typically involving BIOS/UEFI firmware, a bootloader, and the kernel's own initialization code.
Boundary Tags	Metadata headers and footers placed around heap-allocated memory blocks that contain information about block size, allocation status, and links to neighboring blocks, enabling efficient coalescing of free blocks.
Bootstrap Processor (BSP)	In symmetric multiprocessing (SMP) systems, the primary CPU core that executes the bootloader, initializes system hardware, starts other application processors, and runs the kernel's main initialization sequence.
Canary Value	A known, randomly generated value placed in memory (typically at the end of a stack frame) that is checked before function return to detect stack buffer overflow attacks or corruption.
Context Switch	The procedure performed by the kernel to save the complete CPU state (registers, program counter, stack pointer) of the currently running process and restore the saved state of the next process to run, effectively transferring execution between processes.
DMA	Direct Memory Access, a hardware mechanism that allows peripheral devices to read from and write to system memory without involving the CPU for every data transfer, improving I/O performance.
EOI	End of Interrupt, a specific command sent to the Programmable Interrupt Controller (PIC) or Advanced Programmable Interrupt Controller (APIC) to signal that interrupt handling for a specific IRQ has completed and the controller can issue further interrupts.
Fault Injection	A testing technique that deliberately introduces errors (memory allocation failures, invalid pointers, device errors) into a running system to evaluate its robustness, error handling, and recovery mechanisms.
GDT	Global Descriptor Table, a processor data structure in x86 architecture that defines memory segments and their attributes (base address, limit, access permissions) used in protected mode operation.
Graceful Degradation	A design principle where a system continues to operate with reduced functionality when some components fail, rather than completely crashing. In the kernel context, this might involve disabling specific hardware features while maintaining core operations.
Guard Page	An intentionally unmapped memory page placed immediately below a stack's allocated region that triggers a page fault (and potentially kernel intervention) if the stack grows beyond its allocated bounds, protecting against stack overflow corruption.
Higher-Half Kernel	A kernel design approach where the kernel code and data reside in the upper half of the virtual address space (typically above `0xC0000000`), leaving the lower half entirely for user processes and simplifying address space isolation.
IDT	Interrupt Descriptor Table, a processor data structure in x86 architecture that maps interrupt vectors (0-255) to the memory addresses of their corresponding Interrupt Service Routines (ISRs) along with privilege and type information.
Inter-Process Communication (IPC)	A set of mechanisms that allow processes to exchange data, synchronize execution, and coordinate activities. Common IPC methods include message passing, shared memory, and pipes.
Inter-Processor Interrupt (IPI)	A hardware interrupt generated by one processor core to another in a multiprocessor system, used for coordination tasks like scheduling, TLB flushing, or waking sleeping processors.
IRQ	Interrupt Request, a hardware signal line from a peripheral device to the interrupt controller (PIC/APIC) indicating that the device requires attention. Each device is typically assigned a specific IRQ number.
ISR	Interrupt Service Routine, the handler function that executes in response to a specific hardware or software interrupt. ISRs must be fast, save/restore processor state, and often acknowledge the interrupt to the controller.
Linker Script	A configuration file that directs the linker program on how to combine sections from object files into the final executable, specifying memory addresses, section ordering, and symbol definitions critical for kernel placement.
Magic Breakpoint	The x86 instruction `xchg bx, bx` (opcode `0x87DB`), which acts as a software breakpoint in the Bochs emulator, causing execution to pause and enter the debugger when encountered.
Magic Number	A distinctive constant value placed within data structures (like `HEAP_MAGIC: 0xDEADBEEF` in heap block headers) or files used to verify integrity, detect corruption, or identify specific formats during debugging.

Term/Acronym	Definition
Microkernel	A kernel architecture design that minimizes the amount of code running in privileged kernel mode, implementing many traditional kernel services (file systems, drivers) as separate user-mode processes that communicate via message passing.
Monolithic Kernel	A kernel architecture where all core operating system services (memory management, process scheduling, device drivers, file systems) run in a single, privileged address space, communicating via function calls rather than IPC.
Multiboot	An open standard specifying an interface between a bootloader (like GRUB) and an operating system kernel, ensuring the kernel is loaded in a consistent, well-defined state with memory map information passed to the kernel.
Multilevel Feedback Queue (MLFQ)	An advanced scheduling algorithm that maintains multiple priority queues, dynamically adjusting process priorities based on their observed behavior (CPU-bound vs. I/O-bound) to optimize response time and fairness.
OOM Killer	An Out-Of-Memory manager component that selects processes to terminate when the system exhausts available memory, aiming to reclaim memory while minimizing disruption to critical services.
Packed Structure	A data structure definition (using compiler attributes like `__attribute__((packed))` in GCC) that instructs the compiler not to insert padding bytes between fields, ensuring an exact, predictable byte layout essential for hardware register maps and binary formats.
Paging	The virtual memory technique used by modern x86 processors where memory is divided into fixed-size pages (typically 4KB), and page tables translate virtual addresses to physical addresses, enabling memory protection, isolation, and demand paging.
PCB	Process Control Block, the central kernel data structure (typically `struct pcb`) that stores all information about a process's state, identity, and resources, including PID, register state, page tables, scheduling information, and open file handles.
PIC	Programmable Interrupt Controller, a hardware chip (8259A) that manages hardware interrupt requests (IRQs) from peripheral devices, prioritizes them, and signals the CPU via a single interrupt line. Modern systems use the APIC.
Post-mortem Debugging	Analyzing system state after a crash has occurred, typically by examining saved register dumps, stack traces, and log messages written to persistent storage or serial console before system halt.
Process Control Block (PCB)	(See PCB above) The definitive data structure representing a process within the kernel, containing all metadata needed to manage, schedule, and context switch between processes.
Process State Model	A finite state machine describing the lifecycle of a process through states like New, Ready, Running, Blocked, and Terminated, with defined transitions between them triggered by scheduler decisions and I/O events.
Protected Mode	The primary 32-bit operating mode of x86 CPUs that enables memory protection, virtual memory via paging, privilege levels (ring 0-3), and multi-tasking capabilities, required for modern operating systems.
Recursive Mapping	A page table technique where the page directory is mapped into itself at a specific virtual address, allowing the kernel to access and modify page table structures using regular memory accesses rather than complex physical address calculations.
Round-Robin	A simple, preemptive scheduling algorithm where each process in the ready queue runs for a fixed time slice (quantum) before being preempted and placed at the back of the queue, ensuring fair CPU time distribution.
Scancode	The raw byte value generated by a PS/2 keyboard controller when a key is pressed or released, representing the physical key position rather than the character meaning. The keyboard driver converts scancodes to ASCII or Unicode characters.
Serial Port (UART)	Universal Asynchronous Receiver/Transmitter, a hardware interface for serial communication that transmits data one bit at a time. Used in kernel development for debugging output when the screen is unavailable.
Stack Canary	(See Canary Value above) A security mechanism that places a random value before the return address on the stack and checks it before function return to detect stack-based buffer overflow attacks.
Stress Testing	A testing approach that subjects a system to extreme or maximum operational load (high memory allocation rates, many concurrent processes, rapid interrupt firing) to identify breaking points, resource leaks, and performance bottlenecks.
Symmetric Multiprocessing (SMP)	A multiprocessor computer architecture where two or more identical processor cores share the same main memory and are controlled by a single operating system instance that can schedule tasks on any processor.
System Call Convention	The agreed-upon method for user processes to invoke kernel services, specifying how system call numbers and arguments are passed (typically via registers or stack) and how results are returned, usually triggered by a software interrupt (`int 0x80`) or dedicated instruction (`syscall`).
TLB	Translation Lookaside Buffer, a processor cache that stores recently used virtual-to-physical address translations from page tables, speeding up memory accesses. The TLB must be flushed after page table modifications.
TLB Flush	The operation of invalidating entries in the Translation Lookaside Buffer, required after modifying page table entries to ensure the processor uses the new translations rather than stale cached ones. Can be global or address-specific.

Term/Acronym	Definition
Triple Fault	A catastrophic CPU condition on x86 architecture that occurs when an exception happens while the CPU is trying to invoke an exception handler (double fault), and that double fault handler itself causes another exception, forcing an immediate system reset. Often indicates severe kernel bugs.
TSS	Task State Segment, an x86 data structure that holds information needed for task switching and privilege level changes, most importantly the kernel stack pointer for each privilege level, used when handling interrupts from user mode.
VGA	Video Graphics Array, a display hardware standard with a text mode where character data is written to a memory-mapped buffer at physical address `0xB8000`, with each character cell consisting of an ASCII byte and an attribute byte for colors.
Virtual Filesystem (VFS)	An abstraction layer that provides a uniform, filesystem-agnostic interface to file operations (open, read, write, close), allowing the kernel to support multiple concrete filesystem implementations (FAT, ext2, etc.) through a common API.
Zombie Process	A process that has terminated (entered the Terminated state) but whose process control block and resources have not yet been fully cleaned up because its parent process has not yet called a `wait()`-style system call to collect its exit status.