Step 5: Final Submission Document

Course Notes: Building a Minimal "Hello World" OS

Student: Scott Hakoda
Course: Operating Systems
Date: August 28, 2025

Part 1: Updated Lecture Outline

Project: Building a Minimal Booting Kernel from XV6

The primary goal is to understand the boot process of an operating system by stripping the existing XV6 kernel down to its essential components. The resulting kernel is a "minimal booting kernel" that runs only in supervisor mode on a single CPU core and does not support a user space. The "Hello World" name refers to its simplicity, not a literal output. The project is built for the RISC-V architecture and tested using the QEMU emulator.

---

1. Environment and Build Process Setup

- **Makefile Modifications**: The Makefile is adjusted to include only the necessary source files for compilation, removing those that are not needed.
- **Linker Script (.ld) Modifications**:
  - **Core Function**: The linker script is the master blueprint for the kernel's memory layout. Its role is critical because the kernel runs on bare metal and must establish its own structure. It ensures the .text (code) and .data sections are placed correctly.
  - **Known Entry Point**: It guarantees that the first instruction of the kernel (_entry) is at the precise physical memory address where the hardware expects to begin execution.
  - **Protection**: By separating code and data, it allows the kernel to later set memory page permissions (e.g., mark code as read-only), a crucial security measure.
  - **Trampoline Deletion**: The "trampoline" section, used for transitions between supervisor and user mode, is deleted as this kernel has no user space.
- **QEMU Configuration**: The QEMU build target is modified to simplify the virtual machine by disabling SMP (restricting to a single core) and removing the virtual hard drive, as there is no file system.

---

2. The Boot Sequence: From Power-On to Kernel Spin

The kernel execution flows through three distinct stages, transitioning from the all-powerful machine mode to the more restricted supervisor mode. This transition is a deliberate application of the **Principle of Least Privilege**, enhancing system stability by ensuring the main kernel code does not have permission to alter fundamental hardware configurations.

Phase A: entry.s - The First Instructions

- **Entry Point**: The linker is configured to start program execution at the _entry label.
- **Stack Setup**: It immediately sets up a system stack in a known, safe memory location so that C functions can be called. Incorrect stack placement could lead to overwriting kernel code or data, causing an immediate crash.
- **Transition to C**: After the stack is ready, it jumps to the start() function in start.c.

Phase B: start.c - Machine Mode Initialization

This code executes in the highly privileged **machine mode**. Its purpose is to perform the initial, critical hardware setup before the main kernel can run.

- Saves the hardware thread ID (hartid) for the current CPU.
- Delegates all interrupts and exceptions to be handled in supervisor mode.
- **Disables Paging**: Paging is temporarily turned off to create a simple, predictable environment where every memory address is a direct physical address. This is necessary to safely build the complex page table structures without paradoxes.
- Gives the supervisor mode access to all physical memory.
- **Prepares for Mode Switch**:
  - The mepc (Machine Exception Program Counter) register is set to the address of the main function.
  - The mstatus (Machine Status) register is configured to transition to supervisor mode upon return.
- **Executes mret**: This instruction causes the CPU to switch from machine mode to supervisor mode and begin executing at the address stored in mepc (i.e., the main function).

Phase C: main.c - Supervisor Mode and Kernel Initialization

This is the main entry point for the C kernel, running in **supervisor mode**.

- **Concurrency Code Removed**: Logic to manage the startup of multiple CPU cores is removed because our kernel is single-core.
- **Initialization Sequence**: It calls a series of functions to initialize core kernel subsystems:
  1. consoleinit(): Initializes the console and UART for printing messages.
  2. kinit(): Sets up the physical page allocator.
  3. kvminit(): Creates the initial kernel page table to map virtual addresses to physical addresses.
  4. kvminithart(): Turns on paging (virtual memory) for the CPU core.
  5. trapinit(): Initializes the trap vectors for handling exceptions.
  6. plicinit(): Sets up the Platform-Level Interrupt Controller (PLIC).
- **Kernel Idle State**: After all initialization is complete, the kernel enters an infinite while(1) loop and spins forever.

---

3. Key Subsystems and Utility Files

- **Console I/O (uart.c, console.c, printf.c)**: Provides the functions necessary for low-level serial output and formatted printing to the console. Spinlocks and interrupt handling are removed for simplicity.
- **String Utilities (string.c)**: Provides standard C string manipulation functions (memcpy, memset, etc.), which are essential as there is no standard C library to link against.

Part 2: Q&A Transcript

Question 1:

Why does a RISC-V processor have these different privilege modes in the first place? What's the point of running the main part of the kernel in a less-privileged "supervisor" mode instead

of just staying in machine mode?

Answer:
The short answer is: Protection and Stability. The different modes are there to create protective barriers, ensuring that a bug in one part of the system doesn't destroy the entire thing. This is guided by a core security concept called the Principle of Least Privilege.

- **Machine Mode (M-mode)** is "God Mode," with unrestricted access to all hardware. It's used for the initial bootloader and firmware to perform critical hardware setup.
- **Supervisor Mode (S-mode)** is for the OS kernel. It can manage memory and handle interrupts but is blocked from accessing the most critical low-level hardware configuration that M-mode controls.

By switching to Supervisor Mode, the kernel intentionally gives up some power. This creates a protective barrier between the main OS logic and the fundamental machine hardware. A bug in a device driver might crash the kernel, but it can't easily corrupt the underlying hardware configuration, making the entire system far more **robust and stable**.

Question 2:

Paging is disabled in start.c and then the kernel page table is set up and paging is re-enabled in main.c. Why turn it off just to turn it back on again? What problems might we run into if we tried to enable virtual memory while still in machine mode?

Answer:
You turn paging off to create a simple, predictable environment where you can build the complex machinery of virtual memory.

1. **Work with a Simple System First (start.c)**: With paging off, every memory address is a direct physical address. This provides a stable, easy-to-understand foundation.
2. **Build the Page Tables (kvminit)**: While working with physical addresses, the kernel creates the page tables, which are the "index cards" that map virtual addresses to physical addresses.
3. **Activate the New System (kvminithart)**: Only after the page tables are fully built does the kernel enable paging, telling the CPU's Memory Management Unit (MMU) to start using them for all memory access.

Trying to build the page tables while they are already in use would create a paradox. Furthermore, the RISC-V architecture is designed to separate concerns: machine mode deals with physical hardware, while supervisor mode uses the virtual memory system. The settings for paging are configured *in* M-mode, but they only take effect once the CPU transitions *to* S-mode. If the page tables were faulty upon this transition, the CPU would immediately

page-fault on the very first instruction and crash.

Question 3:

The notes say the linker script's job is to pull things in and "make sure in right order". Why is the exact memory layout and order of code/data so critical when building a kernel? What could go wrong if the linker script placed the stack, for example, in the wrong memory location?

Answer:
The exact memory layout is critical because a kernel runs on bare metal and must create all memory structure from scratch. The linker script is the master blueprint that imposes this structure.

1. **A Known Entry Point**: The hardware starts executing at a fixed physical address. The linker must place the kernel's first instruction at that exact address.
2. **Separating Code from Data for Protection**: The script organizes the kernel into sections like .text (code) and .data (variables). This allows the kernel to later set memory page permissions (e.g., mark code as read-only but data as read/write), a critical security feature.
3. **Knowing Where Things Are**: The linker provides symbols that tell the kernel where its own code ends, so functions like the memory allocator (kinit) know where free RAM begins.

If the stack were placed in the wrong location, disaster would be immediate. If it overlapped the **code section**, function calls would overwrite kernel instructions, leading to a crash. If it overlapped the **data section**, it would corrupt global variables. If it was placed in a memory region used for **hardware devices**, it would cause a hardware fault and freeze the system.