Chapter 6: Mechanism: Limited Direct Execution

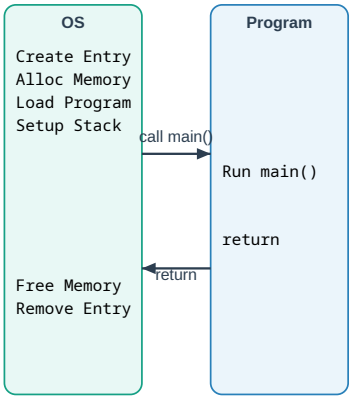
This chapter introduces the fundamental mechanism for CPU virtualization: **Limited Direct Execution (LDE)**. The goal is to run user programs nearly as fast as they would run on bare metal hardware, while ensuring the Operating System retains control to manage resources and protect the system.

> **The Crux of the Problem: How to Efficiently Virtualize the CPU with Control?**
>
> The OS must virtualize the CPU in an efficient manner while retaining control over the system. To do so, both hardware and operating-system support will be required.

1. Basic Technique: Limited Direct Execution

The core idea of LDE is simple: let the user program run directly on the CPU for maximum performance. However, this "direct execution" must be "limited" to prevent the program from taking over the machine or performing malicious actions.

- **Direct Execution Protocol (Initial Idea):**
  1. OS creates a process entry and allocates memory.
  2. OS loads the program from disk into memory.
  3. OS sets up the stack (e.g., with `argc`, `argv`).
  4. OS jumps to the program's `main()` entry point.
  5. The program runs directly on the hardware.
  6. When `main()` returns, control goes back to the OS.
  7. OS frees the process's memory.



This simple protocol raises two critical problems:

1. **Restricted Operations:** How can a program perform privileged actions (like I/O) without having full control of the hardware?
2. **Regaining Control:** How can the OS stop one process and switch to another if the process itself is currently running on the CPU?

---

2. Problem #1: Restricted Operations

A running program needs to perform tasks like reading files or accessing the network, but giving it direct hardware access would break all security and protection models.

> **The Crux of the Problem: How to Perform Restricted Operations?**
>
> A process must be able to perform I/O and some other restricted operations, but without giving the process complete control over the system. How can the OS and hardware work together to do so?

The solution is a hardware-supported mechanism involving different **processor modes**:

- **User Mode:** A restricted mode where applications run. In this mode, privileged instructions (like direct I/O or modifying system settings) are forbidden. Attempting to execute one will cause a hardware **exception**, trapping control back to the OS.
- **Kernel Mode:** A privileged mode where the OS kernel runs. In this mode, all instructions are allowed, giving the OS full control over the hardware.

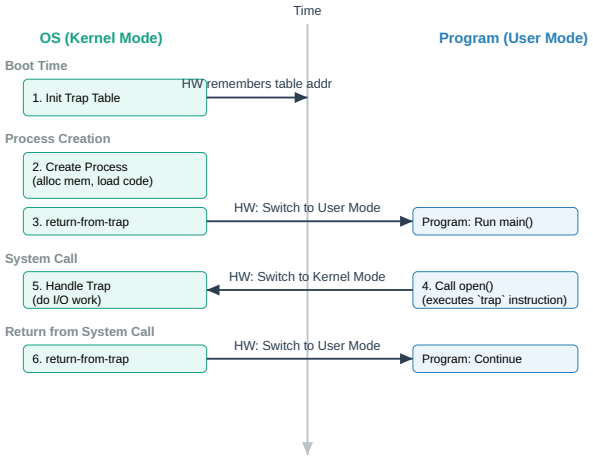
System Calls: The Bridge Between Modes

To allow user programs to request privileged services, the OS provides **system calls**.

- **Mechanism:**
  1. A user program wanting to perform a privileged action (e.g., `open()`) calls a library wrapper function.
  2. This library function sets up arguments and a specific **system call number**.
  3. It then executes a special `trap` instruction.
  4. The `trap` instruction does two things simultaneously:
     - It raises the CPU privilege level to **kernel mode**.
     - It jumps to a pre-determined location in the OS kernel code, known as a **trap handler**.
  5. The OS's trap handler reads the system call number, validates arguments, and performs the requested service.
  6. Once finished, the OS executes a `return-from-trap` instruction, which lowers the privilege level back to **user mode** and returns

control to the user program.

- **The Trap Table:** To ensure user programs can't jump to arbitrary locations in the kernel, the OS sets up a **trap table** in memory during boot time (a privileged operation). This table tells the hardware the exact address of the handler for each type of trap (system calls, interrupts, etc.).



---

3. Problem #2: Switching Between Processes

If a process is running directly on the CPU, the OS is not. How can the OS interrupt a running process to switch to another one, enabling time sharing?

> **The Crux of the Problem: How to Regain Control of the CPU?**
>
> How can the operating system regain control of the CPU so that it can switch between processes?

The Cooperative Approach (Legacy)

- **Idea:** The OS trusts processes to give up the CPU periodically.
- **Mechanism:** A process voluntarily yields control by:
  - Making a system call (e.g., for I/O).
  - Calling an explicit `yield()` system call.
  - Performing an illegal operation (e.g., divide by zero), which traps to the OS.
- **Fatal Flaw:** A buggy or malicious process in an infinite loop will never yield, effectively hijacking the entire machine. The only solution is to reboot.

The Non-Cooperative Approach (Modern)

Modern systems cannot rely on cooperation. They need a way to forcibly regain control.

> **The Crux of the Problem: How to Gain Control Without Cooperation?**
>
> How can the OS gain control of the CPU even if processes are not being cooperative?

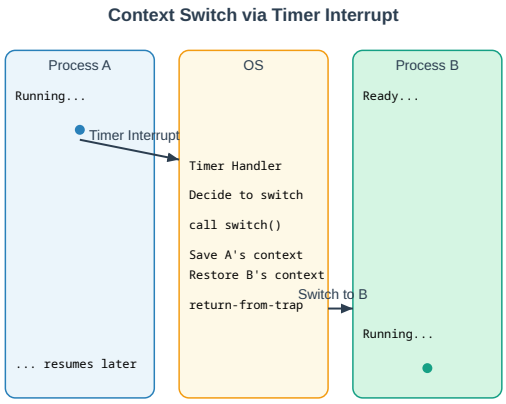
The solution is the **timer interrupt**.

- **Mechanism:**
  1. During boot, the OS programs a hardware timer to generate an interrupt after a specific interval (e.g., every few milliseconds). This is a privileged operation.
  2. The OS then starts a user process.
  3. When the timer goes off, it generates an interrupt, which forcibly stops the current user process.
  4. The hardware saves the state of the user process and jumps to the OS's pre-configured **timer interrupt handler**.
  5. The OS now has control of the CPU and can decide whether to resume the current process or switch to a different one.

The Context Switch

When the OS decides to switch from Process A to Process B, it performs a **context switch**.

- **Action:** The OS saves the context of the current process (A) and restores the context of the next process (B).
- **Steps:**
  1. The timer interrupt occurs while Process A is running. The hardware automatically saves A's user-level registers (PC, etc.) onto A's **kernel stack**.
  2. The OS's timer interrupt handler runs. It decides to switch to Process B.
  3. The OS calls its `switch()` routine. This routine explicitly saves the rest of A's kernel-level context (kernel registers) into A's **Process Control Block (PCB)**.
  4. The `switch()` routine then restores B's kernel-level context from B's PCB. A key part of this is changing the stack pointer to point to B's kernel stack.
  5. The OS executes `return-from-trap`. The hardware now restores B's user-level registers from B's kernel stack and jumps to B's program counter. Process B is now running.



4. Concurrency Concerns

- **The Problem:** What happens if a timer interrupt occurs while the OS is already in the middle of handling a system call or another interrupt?
- **Simple Solution:** The OS can temporarily **disable interrupts** while it is handling a trap or interrupt. This must be done for short periods to avoid missing other important events.
- **Advanced Solution:** Modern OSes use sophisticated **locking schemes** to protect internal data structures, allowing multiple activities to occur within the kernel simultaneously, which is especially important on multi-processor systems.

---

5. Summary

Limited Direct Execution is the core mechanism for CPU virtualization. The OS achieves this by:

1. **Setting up the Hardware:** At boot time, the OS configures the trap table and starts the timer interrupt. This is the "limited" part.
2. **Running Programs in User Mode:** Programs execute directly on the CPU for efficiency but are restricted from performing privileged operations.
3. **Using System Calls:** Programs trap into the kernel to request OS services.
4. **Regaining Control:** The OS forcibly regains control via the timer interrupt, allowing it to implement time sharing by performing context switches between processes.

This combination of hardware support (modes, traps, timer) and OS software (handlers, schedulers, context switch routines) allows for both high performance and robust control.