Introduction to LDR

Virtualization is achieved by the operating system must sharing the physical CPU’s time across many jobs running concurrently. The idea is to run one process, then another, and so on. This presents a few challenges concerning performance and control over the CPU.

The OS needs to efficiently virtualize the CPU while maintaining system control. So, this section is dedicated to helping us answer the following questions:

- **How can we virtualize without introducing too much overhead?**
- **How can we run processes effectively while maintaining control of the CPU?**

This requires support from both the hardware and operating system.

Basic Technique: Limited Direct Execution

To make a program run as fast as possible, OS developers developed a technique known as **limited direct execution**.

Direct execution involves running the program directly on the CPU. So, when the OS starts up the OS:
* Creates an entry for the program in a process list
* Allocates some memory to it
* Loads the program onto disk
* Locates its entry point (e.g., main() or something similar), and
* Starts running the program.

In the figure below, we can see this basic direct execution protocol (without any limitations), using a normal call and return to jump to the program's `main()` and then back into the kernel.

.guides/img/direct_exec

This approach presents a few problems in our attempt to virtualize the CPU.

1. If we just run a program, how does the OS make sure it isn't doing something it shouldn't be, while still effectively running?
2. How does the operating system switch from one process to another when we are running a process, using the time-sharing that we need for virtualization?

Through our exploration we'll see why the term “limited” makes a lot of sense. If the operating system had no limits, the OS would be just a library - a sad day for a hopeful operating system!

Drag the lines to the area below to describe the order of a basic Direct Execution.

Restricted Operations: User & Kernel Mode

Direct execution is fast since the program runs natively on the hardware CPU, but running on the CPU presents a problem: what if the process needs to perform a restricted operation, like writing to a disk or accessing extra CPU or memory?

A process needs the ability to perform I/O and other restricted operations, but not have complete system control.

How can the OS and hardware team up for this?

We could let each process do its own I/O and other tasks. But this would prevent the building of many useful systems. For example, if we want to build a file system that validates permissions before granting access to a file, we can't just let any user process read or write the disk, or we wouldn't have any protection!

To combat this, the processor should support two modes: restricted **user mode**, and privileged **kernel mode**.
* **User mode** - limits the capabilities of code running in it. In user mode, a process can't issue I/O requests. If it did, the CPU would raise an error, and the OS would likely kill the process.

- **Kernel mode** - where the operating system (or kernel) runs. This mode allows code to run and do what it likes, including privileged operations, like issuing I/O requests and executing all kinds of restricted instructions.

Fill in the blanks to complete the statements below.

System Calls

But the question remains: What should a user process do when it wants to perform privileged operations like reading from disk?

User applications run in user mode and use **system calls** to request privileged services from the kernel.

System calls let the kernel expose key pieces of functionality for programs to use, like:

- Accessing file systems
- Launching and killing processes
- Communicating with other processes, and
- Allocating more memory

A program must execute a special **trap** instruction that:
* Jumps into the kernel and jumps into the kernel, and
* Raises the privilege level to kernel mode

Once in the kernel, the system can perform privileged activities (if allowed) that it needs for the process.
When it's finished, the OS calls a special **return-from-trap** instruction, which:
Returns the calling user program to user mode, and
Reduces the privilege level back to user.

When executing a trap, the hardware has to save enough of the caller's registers to be able to return successfully when the OS issues the return-from-trap instruction.

How does the trap know which code to run inside the OS?

So, unlike a procedure call, the calling process cannot specify an address to jump to. This would allow applications to jump anywhere into the kernel. So, the kernel must carefully control what code execute when a trap occurs.

Enter, the **trap table**!

Which of the following is a way for a user program to request privileged services from the kernel?

Trap Tables

During boot, the kernel creates a **trap table**.

The machine starts in privileged (kernel) mode, allowing it to configure the machine hardware as needed. When certain unusual events occur, the OS tells the hardware what code to run. For example, **what code should run when a hard disk, keyboard, or system call occurs?**

The OS tells the hardware of these **trap handlers**' locations via special instructions.

So when system calls and other unusual events occur, the hardware knows what to do (which code to jump to) and where to look for handlers.

Each system call is assigned a **system-call number**. The user code must thus save the intended system-call number in a register or on the stack. The OS checks this number's legitimacy and, if it is, executes the matching code. The user code cannot specify a specific address to go to, but must request a service by number.

The ability to execute instructions that tell hardware where trap tables are located is very powerful. It is a privileged procedure, so the hardware won't let you execute this instruction in user mode, and you can probably predict what happens if it tried (hint: adios, offending program). Consider what you could do to a system if you had your own trap table. Could you take over?

The image to the left summarizes the protocol.

We assume each process has a kernel stack where registers are saved and restored (by hardware).

The Limited Direct Execution (LDE) protocol contains two steps.
1. Trap tables are initialized by the kernel at boot time, and the CPU remembers their location for later use. The kernel does so using privileged code (all privileged instructions are highlighted in bold).

2. The kernel allocates a node on the process list and allocates memory before using a return-from-trap instruction to start the process. In order to make a system call, a process traps back into the operating system, which handles it and once more returns control via a return-from-trap to the process.

When the process has completed its work, it returns to main(); this usually returns to some stub code that will properly exit the program (say, by calling the exit() system call, which traps into the OS). The OS now cleans up and we are done.

Fill in the blanks to complete the statements below.

Problem 2: Switching Between Processes

The next issue with direct execution is **process switching**. Changing procedures should be easy, right? The OS should simply decide to switch processes. But, if a process is running on the CPU, this means **the OS is not running**.

How can it do anything if it isn’t running?

The short answer: **It can’t**. So the question for us to answer is:

How can the OS regain control of the CPU to switch between processes?

Waiting for System Calls: A Cooperative Approach

The **cooperative approach** trusts the system processes to behave appropriately and periodically give up the CPU to conduct other activities.

When applications do anything illegal, **they give the OS control**. It will generate a trap if, for example, an application divides by zero or tries to access memory that it should not be able to access.

The OS regains control of the CPU (and likely terminate the offending process).

With cooperative scheduling, the OS regains control of the CPU by waiting for a system call or some illegal operation.

But what happens if a process ends up in an infinite loop and never makes a system call?

The OS takes over: a non-cooperative approach

The OS can only do very little without the help of the hardware when a process refuses to make system calls (or makes mistakes), returning control to the OS.

How can the OS control the CPU when processes refuse to cooperate?

What can the OS do to prevent a renegade process from taking over?

The answer is to use a **timer interrupt**. This is a timer that can be programmed to raise an interrupt every so many milliseconds to stop the current process and starting an interrupt handler in the OS. When an interrupt happens, the current process stops and the OS launches a pre-configured interrupt handler.

Now that the OS has recovered control of the CPU, it can halt the current process and start a new one.

As with system calls, the OS must:

1. Instruct the hardware which code to run when a timer interrupt occurs, and it does so at boot time.

2. Start the timer during the boot sequence, which is a privileged activity.

Once the timer starts, the OS knows that it will regain control and so be free to run user programs.

When an interrupt occurs, the hardware must save enough of the program’s state to allow a future return-from-trap command to appropriately continue the running program.

As in an explicit system-call trap inside the kernel, certain registers are saved (e.g., onto a kernel stack) and recovered by the return-from-trap instruction.

Which of the following describes a timer interrupt?

Saving and Restoring Context

Context Switch

If the OS decides to switch, it performs a **context switch**.

A **context switch** is where the OS just saves some register values for the currently running process (onto its kernel stack) and restores a few values for the next process (from its kernel stack).

This is how the OS makes sure that when the return-from-trap command is finally executed, the system continues running another process instead of the one that was previously running.

To save the context of the current process, the OS will:
* Execute low-level assembly code to save general purpose registers, PC, and the kernel stack pointer of the current process,

- Then restore all of those and switch to the kernel stack for the next process to execute.

By switching stacks, the kernel enters the call to the switch code in the context of the interrupted process and returns in the context of the soon-to-be-executing process.

The soon-to-be-executing process becomes the currently-running one once the OS finally executes a return-from-trap instruction.

In this case:

- Process A is interrupted by the timer.
- The hardware stores its registers and enters the kernel (switching to kernel mode).
- The OS switches from Process A to Process B in the timer interrupt handler.
- The next step is to call the `switch()` routine.
    - This saves all incoming register values (into the process structure of A), restores the registers of Process B (from its process structure entry), and switches context by switching between B and A stacks (and not vice versa).
- Finally, the OS exits the trap, restoring B’s registers and launching it.

During this protocol, two forms of register save/restore occur.

- When a timer interrupt occurs, the hardware implicitly saves the running process's user registers using the kernel stack.
- And when the OS decides to go from A to B, the software (the OS) explicitly saves the kernel registers into memory in the process structure.

This action changes the system's behavior from A to B, as if it were simply trapped into the kernel from A.

Concurrency Concerns

What happens when a timer interrupts a system call? What happens when you're dealing with one interrupt, and another comes?

Isn't that difficult in the kernel?

Yes, the OS should be worried about what happens if another interrupt occurs during interrupt or trap handling. There is an entire course on concurrency where we will explore this in detail.

As a brief introduction, let's sketch the OS's basic handling of these challenging scenarios.

- An OS could disable interrupts during interrupt processing to make sure that no other interrupts are transmitted to the CPU.
    - The OS must be cautious; deactivating interrupts for too long may result in lost interrupts, which is problematic.
- Operating systems have also evolved sophisticated locking methods to prevent simultaneous access to internal data.
    - This allows many kernel activities to run simultaneously, which is useful on multiprocessors.

However, we'll see in our **concurrency** course that such locking can be complex and lead to unique and hard-to-find bugs.

Summary

We’ve detailed a few low-level strategies for implementing CPU virtualization, which we call **limited direct execution**. The idea is to run the application you want on the CPU, but first set up the hardware to limit what the process can do without OS help.

It’s a little like baby-proofing: locking cabinets containing dangerous items and covering electrical outlets.

As soon as the room is proofed, the baby can roam freely, because you know you’ve removed all of the danger.

The OS also “baby-proofs” the CPU by:
* Establishing the trap handlers (during boot time) * Starting the interrupt timer, and then
* Only running processes in a restricted mode.

So the OS can be confident that processes can run efficiently, only requiring OS intervention for privileged operations or when they have monopolized the CPU for too long and need to be switched out.

- At least two modes of execution should be supported by the CPU: a restricted **user mode** and a privileged (non-restricted) **kernel mode**.
- User applications run in user mode and use a **system call** to **trap** into the kernel to request services.
- The trap instruction saves register state, switches hardware to kernel mode, and jumps into the OS to the **trap table**.
- When the OS is finished with a system call, special return-from-trap instruction reduces privilege and returns control to the instruction after the trap jumped into the OS.
- The OS must set up the trap tables at boot time, and user programs must not be allowed to modify them. Using the limited direct execution protocol, programs are efficiently run without compromising OS control.
- Once a program is running, the OS must use hardware means to stop it, mainly the timer interrupt. This is a non-cooperative CPU scheduling method.
- During a timer interrupt or system call, the OS may want to change the running process, a low-level mechanism known as a context switch.

A fundamental question remains unanswered: **What process should we run at a given time?** This is a question for the scheduler to answer.