**Chapter 6: Mechanism: Limited Direct Execution**

To virtualize the CPU, the OS needs to share the physical CPU among many jobs running seemingly at the same time. The basic idea is simple: run one process for a little while, then another, etc. By **time sharing** the CPU in this manner, virtualization is achieved.

Challenges to virtualization: **performance** and **control**

**6.1** **Basic Technique: Limited Direct Execution**

To make a program run as fast as we expect, OS developers come up with a technique called **Limited Direct Execution**. The “Direct Execution” means just run the program directly on the CPU. Thus, when the OS wishes to start a program running, it creates a process entry for it in a process list, allocates some memory for it, loads the program code into memory (from disk), locates its entry point (i.e., the main() routine or something similar), jumps to it, and starts running the user’s code. The following figure describe the protocol:

Text, letter

Description automatically generated

However, this raises some problems:

1. How can the OS make sure that it does not do anything we don’t want it to do, while running it efficiently?
2. How does the OS stop a process and switch to another process to fully implement the **time sharing** to **virtualize** the CPU?

**6.2 Problem #1: Restricted Operations:**

Direct execution is fast, but it prompts a problem: what if the process wishes to perform some kind of restricted operation, such as issuing an I/O request to a disk, or gaining access to more system resources such as CPU or memory?

We cannot let any process do whatever it wants in terms of I/O and other operations because protections would be lost. Thus, we introduce a new processor mode, known as **user mode**. Code that runs in user mode is restricted in what it can do. For example, when running in user mode, a process can’t issue I/O requests; doing so would result in the processor raising an exception; the OS would then likely kill the process.

In contrast, we have **kernel mode** which the OS (or kernel) runs in. In this model, code that runs can do what it likes.

However, what should a user process do when it wishes to perform some kind of privileged operation, such as reading from disk? To do so, virtually all modern hardware provides the ability for user programs to perform a system call. System calls allow the kernel to carefully expose certain key pieces of functionality to user programs.

To execute a system call, a program must execute a special trap instruction. This instruction simultaneously jumps into the kernel and raises the privilege level to kernel mode; once in the kernel, the system can now perform whatever privileged operations are needed (if allowed), and thus do the required work for the calling process. When finished, the OS calls a special return-from-trap instruction, which returns into the calling user program while simultaneously reducing the privileges level back to user mode.

The hardware needs to be a bit careful when executing a trap, in that it must make sure to save enough of the caller’s registers in order to be able to return correctly when the OS issues the return-from-trap instruction. On x86, the processor will push the program counter, flags, and a few other registers onto a per-process kernel stack. The return-from-trap will pop these values off the stack and resume execution of the user-mode program.

The kernel must carefully control what code executes upon a trap. The kernel does so by setting up a **trap table**. When the machine boots up, it does so in privileged (kernel) mode, and thus is free to configure machine hardware as need be. One of the first thing the OS does is to tell the hardware what code to run when exceptions occur. The OS tells the hardware the locations of these **trap handlers**. The hardware then remembers the location of these handlers until the machine next is rebooted. Therefore, the hardware knows what to do when system calls and other exceptional events take place.

To specify exact system call, **a system-call number** is usually assigned to each system call. The user code is responsible for placing the desired system-call number in the register or at a specified location on the stack. The OS, when handling the system call inside the trap handler, examines this number, ensures it is valid, and, if it is, executes the corresponding code. This level of indirection serves as a form of **protection**; user code cannot specify an exact address to jump to, but rather must request a particular service via number.

Being able to execute the instruction to tell the hardware where the trap tables are is a **privileged** operation.

Figure 6.2 summarizes the LDE protocols, including 2 phases.

**6.3 Problem #2: Switching Between Processes**

How can the operating system regain control of the CPU so that it can switch between processes?

1. A **Cooperative** Approach: Wait For System Calls:

In this style, the OS trusts the processes of the system to behave reasonably. Processes that run for too long are assumed to periodically give up the CPU so that the OS can decide to run some other task.

Most processes, as it turns out, transfer control of the CPU to the OS quite frequently by making system calls, for example, to open a file and subsequently read it, or to send a message to another machine, or to create a new process. Systems like this often include an explicit yield system call, which does nothing except to transfer control to the OS so it can run other processes.

Timeline

Description automatically generated with low confidence

Applications also transfer control to the OS when they do something illegal, such as division by zero.

Thus, in a cooperative scheduling system, the OS regains control of the CPU by waiting for a system call or an illegal operation of some kind to take place. However, what if a process ends up in an infinite loop, and never make a system call? (reboot the machine)

1. A **Non-Cooperative** Approach: The OS Takes Control

Without some additional help from the hardware, it turns out the OS can’t do much at all when a process refuses to make system calls and thus return control to the OS.

The answer to such problem is to develop a **timer interrupt**, which is a timer device can be programmed to raise an interrupt every so many milliseconds. When the interrupt is raised, the currently running process is halted, and a pre-configured interrupt handler in the OS runs. At this point, the OS has regained control of the CPU, and thus can do what it pleases: stop the current process and start a different one.

This timer can also be turned off.

1. Saving and Restoring context:

When OS regained control, a decision has to be made: whether to continue running the currently-running process, or switch to a different one. Such decision is made by a part of the OS called the **scheduler**.

When the decision is made to switch, the OS then executes a low-level piece of code called **context switch**. All the OS has to do is save a few register values for the currently-executing process and restore a few for the soon-to-be-executing process. By doing so, the OS thus ensures that when the return-from-trap instruction is finally executed, instead of returning to the process that was running, the system resumes execution of another process.

To save the context of the currently-running process, the OS will execute some low-level assembly code to save the general purpose registers, PC, and the kernel stack pointer of the currently-running process, and then restore said registers, PC, and switch to the kernel stack for the soon-to-be-executing process. By switching stacks, the kernel enters the call to the switch code in the context of one process (the one that was interrupted) and returns in the context of another (the soon-to-be-executing one). When the OS then finally executes a return-from-trap instruction, the soon-to-be-executing process becomes the currently-running process. And thus the context switch is complete. An example of the procedure is shown below:

Timeline

Description automatically generated with medium confidence

There are two types of register saves/restores that happen during this protocol. The first is when the timer interrupt occurs; in this case, the **user registers** of the running process are implicitly saved by the **hardware**, using the kernel stack of that process. The second is when the OS decides to switch from A to B; in this case, the **kernel registers** are explicitly saved by the **software** (the OS)

**6.4 Worried about Concurrency?**

The OS need to be concerned as to what happens if, during interrupt or trap handling, another interrupt occurs (concurrency).

One simple thing an OS might do is **disable interrupts** during interrupt processing. In other words, when one interrupt is being handled, no other one will be delivered to the CPU. However, disabling interrupts for too long could lead to lost interrupts.

Operating systems also have developed a number of sophisticated **locking** schemes to protect concurrent access to internal data structures.