【OS】Day10

【Ch6】Direct Execution

In order to virtualize the CPU, the operating system needs to somehow share the physical CPU among many jobs running seemingly at the same time.

The basic idea is: run one process for a little while, then run another one, and so forth.

By time sharing the CPU in this manner, virtualization is achieved.

However, there are a few challenges in building such virtualization machinery:

- The first is performance: how can we implement virtualization without adding excessive overhead to the system?
- The second is control: how can we run processes efficiently while retaining control over the CPU?

Obtaining high performance while maintaining control is thus one of the central challenges in building an operating system.

6.1 Basic Technique: Limited Direct Execution

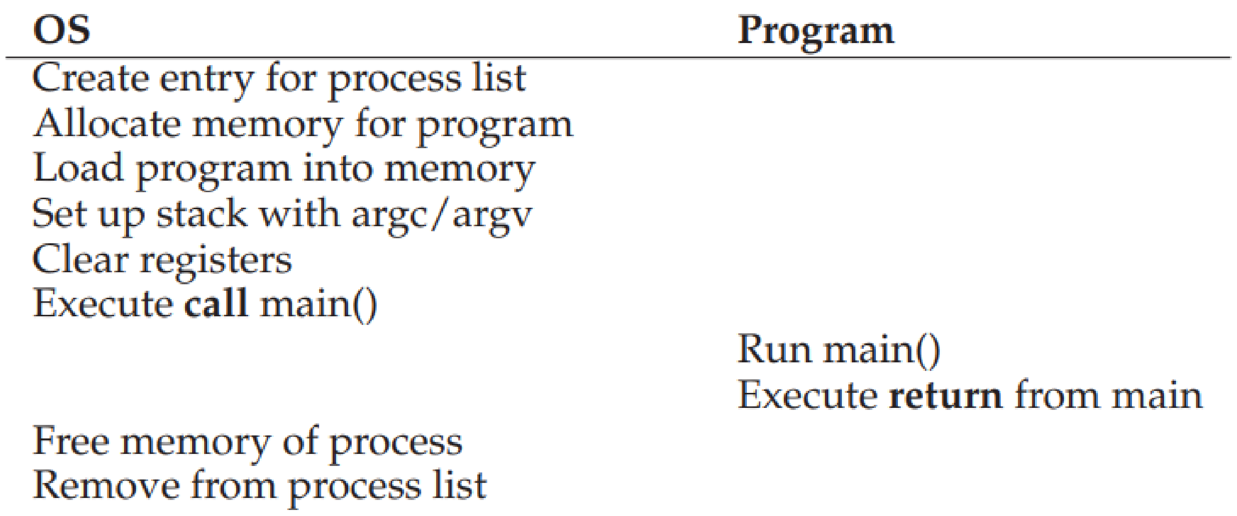
To make a program run as fast as one might expect, OS developers came up with a technique that we call limited direct execution.

The "direct execution" part of the idea is simple: 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), jumps to it

- Starts running the user’s code.



OS	Program
Create entry for process list	
Allocate memory for program	
Load program into memory	
Set up stack with argc/argv	
Clear registers	
Execute **call** main()	
	Run main()
	Execute **return** from main
Free memory of process	
Remove from process list	

Figure 6.1: **Direct Execution Protocol (Without Limits)**

The above figure shows this basic direct execution protocol(without any limits yet) using a normal `call` and return to jump to the program’s `main()` and later back into the kernel.

This approach leads to a few questions:

1. First, if we just run a program, how can the OS make sure the program doesn’t do anything that we don’t want it to do, while still running it efficiently?
2. Second, when we are running a process, how does the operating system stop it from running and switch to another process?

6.2 Problem #1: Restricted Operations

Direct execution enables the program runs natively on the hardware CPU and thus executes as quickly as one would expect.

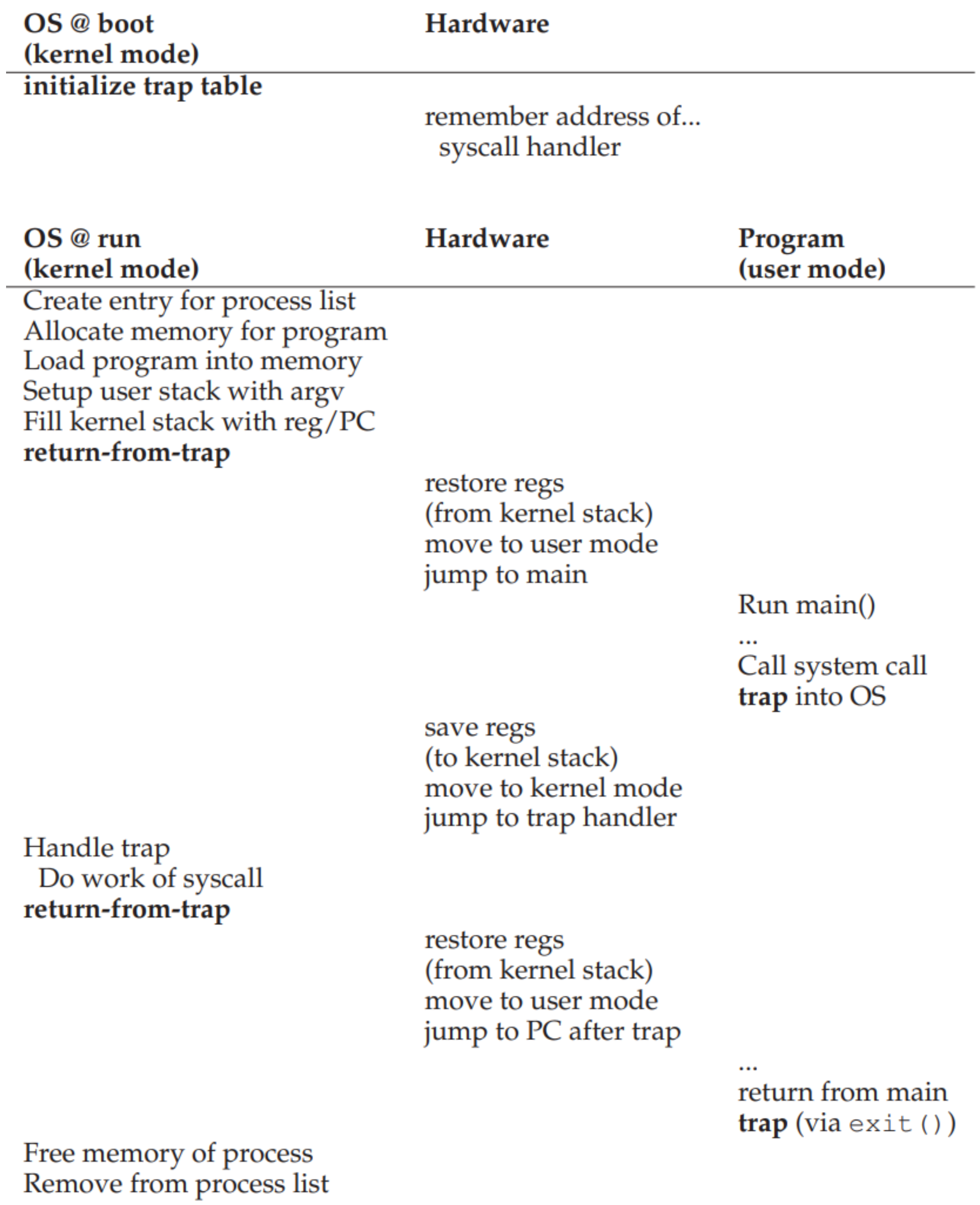
But running on the CPU introduces a problem: *what if the process wishes to perform some kind of restricted operation*, such as issuing an I/O request to a disk, or gaining access to more system resources such as CPU or memory?

The approach we take is to 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 to user mode is kernel mode, which the operating system(or kernel) runs in.

In this mode, code that runs can do what it likes, including privileged operations such as issuing I/O requests and executing all types of restricted instructions.



OS @ boot (kernel mode)	**Hardware**
initialize trap table	
	remember address of... syscall handler

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
Create entry for process list Allocate memory for program Load program into memory Setup user stack with argv Fill kernel stack with reg/PC **return-from-trap**		
	restore regs (from kernel stack) move to user mode jump to main	
		Run main() ... Call system call **trap** into OS
	save regs (to kernel stack) move to kernel mode jump to trap handler	
Handle trap Do work of syscall **return-from-trap**		
	restore regs (from kernel stack) move to user mode jump to PC after trap	
		... return from main **trap** (via `exit()`)
Free memory of process Remove from process list		

Figure 6.2: **Limited Direct Execution Protocol**

What should a user process do when it wishes to perform some kind of privileged operations, such as reading from disk?

To enable this, 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, such as accessing the file system, creating and destroying processes, communicating with other processes, and allocating more memory.

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.

When finished, the OS calls a special return-from-trap instruction, which as you might expect, returns into the calling user program while simultaneously reducing the privilege level back to user mode.

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.

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

The kernel sets up a trap table at boot time. 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 things the OS thus does is to tell the hardware what code to run when certain exceptional events occur.

For example, what code should run when a hard-disk interrupts takes place, when a keyboard interrupt occurs, or when a program makes a system call?

The OS informs the hardware locations of these trap handlers, usually with some kind of special instruction. Thus, the hardware knows what to do when system calls and other exceptional events take place.

To specify the exact system call, a system-call number is usually assigned to each system call. The user code is thus responsible for placing the desired system-call number in a 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.

There are two phases in the limited direct execution protocol:

1. In the first(at boot time), the kernel initializes the trap table, and the CPU remembers its location for subsequent use.
2. In the second(when running a process), the kernel sets up a few things before using a return-from-trap instruction to start the execution of the process; this switches the CPU to user mode and begins running the process. When the process wishes to issue a system call, it traps back into the OS, which handles it and once again returns control via return-from-trap to the process.

   The process then completes its work, and returns from `main()`; this usually will return into some stub code which will properly exit the program(say, by calling the `exit()` system call, which traps into the OS).

   At this point, the OS cleans up and we are done.