# 【OS】Day11

## 【Ch6】Limited Direct Execution(2)

### 6.3 Problem #2: Switching Between Processes

The next problem with direct execution is achieving a switch between processes.

*However, now we meet a crux: How can the operatin system regain control of the CPU so that it can switch between processes when a process is running on the CPU?*

#### A Coooperative Approach: Wait for System Calls

One approach that some systems have taken in the past is known as the cooperative approach.

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.

*Thus, we might think: How does a friendly process give up the CPU in this utopian world?*

Most processes 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.

Applications also transfer control to the OS when they do something illegal. For example, if an application divides by zero, or tries to access memory that it shouldn't be able to acess; it will enerate a tryp to the OS.

Thus, in a cooperative shceduling 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 happends if a process(maliciously or bugly) ends up in an infinite loop, and never makes a system call? What can the OS do then?*

## A Non-Cooperative Approach: The OS Takes Control

*The Crux: How to gain control without cooperation?*

The answer turns out to be simple: a timer interrupt.

A timer device can be programmed to raise an interrupt every so many milliseconds; when the interrupt is raised, the currently runnning 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.

As we discussed before with system calls, the OS must inform the hardware of which code to run when the timer interrupt occurs; thus, at boot time, the OS does exactly that.

Second, also during the boot sequence, the OS must start the timer, which is of course a privileged operation. The timer can also be turned off, but it is also a privleged operation, and will be discussed later.

## Saving and Restoring Context

Now the OS has regained control, a decision has to be made: *whether to continue running the currently-running process, or switch to a different one.*

This decision is made by a part of the operating system known as the scheduler.

If the decision is made to switch the OS then executes a low-level piece of code which we refer to as a context switch.

A context switch is conceptually simple: all the OS has to do is save a few register values for the currently-executing process(onto itskernel stack, for example) and restore a few for the soon-to-be executing process(from its kernel stack).

| OS @ boot (kernel mode) | Hardware | |
|---|---|---|
| **initialize trap table** | | |
| | remember addresses of... | |
| | syscall handler | |
| | timer handler | |
| **start interrupt timer** | | |
| | start timer | |
| | interrupt CPU in X ms | |

| OS @ run (kernel mode) | Hardware | Program (user mode) |
|---|---|---|
| | | Process A |
| | | ... |
| | **timer interrupt** | |
| | save regs(A) → k-stack(A) | |
| | move to kernel mode | |
| | jump to trap handler | |
| Handle the trap | | |
| Call `switch()` routine | | |
| save regs(A) → proc_t(A) | | |
| restore regs(B) ← proc_t(B) | | |
| switch to k-stack(B) | | |
| **return-from-trap (into B)** | | |
| | restore regs(B) ← k-stack(B) | |
| | move to user mode | |
| | jump to B's PC | |
| | | Process B |
| | | ... |

Figure 6.3: **Limited Direct Execution Protocol (Timer Interrupt)**

The figure above provides a timeline of the entire process.

In this example:

- Process A is running and then is interrupted by the timer interrupt. The hardware saves its registers(onto its kernel stack) and enters the kernel(switching to kernel mode)
- In the timer interrupt handerl, the OS decides to switch from running Process A to B.

- It calls the `switch()` routine, which carefully saves current register values(into the process structure of A), restores the registers of Process B(from its process structure entry), and then switches contexts, specifically by changing the stack pointer to use B's kernel stack.
- The OS returns from trap, which restores B's registers and starts running it.

There are two types of register saves/restores that happen during this protocol:

- The first is when the timer interrupt occurs; in thsi 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, but this time into memory in the process structure of the process.

```
1  # void swtch(struct context **old, struct context *new);
2  #
3  # Save current register context in old
4  # and then load register context from new.
5  .globl swtch
6  swtch:
7    # Save old registers
8    movl 4(%esp), %eax  # put old ptr into eax
9    popl 0(%eax)        # save the old IP
10   movl %esp, 4(%eax)  # and stack
11   movl %ebx, 8(%eax)  # and other registers
12   movl %ecx, 12(%eax)
13   movl %edx, 16(%eax)
14   movl %esi, 20(%eax)
15   movl %edi, 24(%eax)
16   movl %ebp, 28(%eax)
17
18   # Load new registers
19   movl 4(%esp), %eax  # put new ptr into eax
20   movl 28(%eax), %ebp # restore other registers
21   movl 24(%eax), %edi
22   movl 20(%eax), %esi
23   movl 16(%eax), %edx
24   movl 12(%eax), %ecx
25   movl 8(%eax), %ebx
26   movl 4(%eax), %esp  # stack is switched here
27   pushl 0(%eax)       # return addr put in place
28   ret                 # finally return into new ctxt
```

Figure 6.4: **The xv6 Context Switch Code**

*Summary*

ASIDE: KEY CPU VIRTUALIZATION TERMS (MECHANISMS)

- The CPU should support at least two modes of execution: a restricted **user mode** and a privileged (non-restricted) **kernel mode**.
- Typical user applications run in user mode, and use a **system call** to **trap** into the kernel to request operating system services.
- The trap instruction saves register state carefully, changes the hardware status to kernel mode, and jumps into the OS to a pre-specified destination: the **trap table**.
- When the OS finishes servicing a system call, it returns to the user program via another special **return-from-trap** instruction, which reduces privilege and returns control to the instruction after the trap that jumped into the OS.
- The trap tables must be set up by the OS at boot time, and make sure that they cannot be readily modified by user programs. All of this is part of the **limited direct execution** protocol which runs programs efficiently but without loss of OS control.
- Once a program is running, the OS must use hardware mechanisms to ensure the user program does not run forever, namely the **timer interrupt**. This approach is a **non-cooperative** approach to CPU scheduling.
- Sometimes the OS, during a timer interrupt or system call, might wish to switch from running the current process to a different one, a low-level technique known as a **context switch**.