Chapter 6 Mechanism: Limited Direct Execution

Recall: **separate mechanisms and policy**…

- Mechanism: how to virtualize cpu
- Policies: Schedule

Virtualize the CPU

Simple idea: Time sharing

- run one process for a little while, then run another process for a little while, and so on…

Challenges:

- Performance: How to avoid excessive overhead
- Control: How can we run processes efficiently while retaining control over the CPU
  - Without control:
    - A process could run forever — taking over the machine
    - Access Information it shouldn't be allowed to access
- Require both OS and hardware support for efficiency

6.1 Basic Technique: Limited Direct Execution

Direct execution: Just run the program directly on the CPU

Three problems:

1. What if the proc wants to do something restricted?
2. What if the OS wants to stop P1 and run P2 instead?
3. What if running P does something slow, like I/O?

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)**

6.2 Problem #1: Restricted Operations

Ex. I/O request, gain access to more resources (CPU/memory)

Ex. Want a file system to check permissions before granting access to a file.

Hardware support for processor modes (**privilege levels**)

- **User mode**: restricted (what a process can do)
  - Can't issue IO requests directly but must go through kernel mode.
- **Kernel mode**: (OS runs in kernel mode)
  - Code can do anything
    - I/O requests
    - Restricted instructions

Need a way to change between user and kernel mode: to allow user procs to access I/O and other restricted ops:

- syscall (or trap instruction)

Trap instruction:

- Change privilege level to kernel mode
- Run "trap handler" (OS code for handling the trap)
- Save state of the running process
- OS can then do privileged instruction or I/O op if user is allowed

Return-from-trap instruction

- Restore proc state
- Change privilege level back to user mode

Ex. open(), fork(), exec() calls that you use from standard library

Implemented in the C library as hand-coded assembly following conventions

- Process arguments and return values correctly
- Execute trap instruction

Saving/Restoring state:

- Trap: CPU pushes PC, flags, registers onto a per-process kernel stack
- Return-from-trap: CPU pops these values off the stack and resume

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

Trap table is set up at boot time.

- OS informs the hardware of locations of trap handlers
- Using a privileged instruction
  - From user mode: trying to change the trap table — you will be killed by the OS

To call a system call

- The user code must place the desired system-call number in a register (or specified location on the stack)
- The OS, when handling the system call inside the trap handler
  - Check this number is valid
  - Execute code for the handler

- Indirection through system-call numbers is protection
  - User code cannot jump into arbitrary location in kernel code
  - Must specify a service via the system call number

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**

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) to k-stack(A)	
	move to kernel mode	
	jump to trap handler	
Handle the trap		
Call `switch()` routine		
save regs(A) to proc-struct(A)		
restore regs(B) from proc-struct(B)		
switch to k-stack(B)		
return-from-trap (into B)		
	restore regs(B) from k-stack(B)	
	move to user mode	
	jump to B's PC	
		Process B
		...

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

6.3 Problem #2: Switching between Processes

Q: How can the OS regain control of the CPU so that it can switch between processes?

A Cooperative Approach: Wait for System Call

- Early versions of MacOS and other OSes
- OS trusts processes to relinquish CPU periodically
  - So that the OS can run other tasks
- Turns out: most processes transfer control to the OS
  - Make system calls to open / read files, send messages…
  - Processes can also call yield() system call
- Not such a good idea: what can the OS do if a process
  - Is malicious or buggy: (ends up in an) infinite loop without any system calls
  - Only recourse: **reboot the machine**.

A Non-Cooperative Approach: The OS Takes Control

Q: What can the OS do to ensure a rogue process does not take over the machine?

Hardware comes to the rescue:

Timer interrupts

- Interrupt the CPU every X ms
  - When interrupted process is halted
  - OS interrupt handler runs (OS regains control and can what it pleases)
  - Save state

```
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**

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**.

At boot time (privileged instruction)

- set interrupt handler and start timer

The timer interrupt must perform similar actions to
save/restore state as for the trap instructions used for explicit system calls.

Saving and Restoring of Context

- Scheduler decide which process to run next (policy: topic of the next few chapters)

Scheduler decide to switch process:

- OS low-level mechanism (**context switch**):
  - Save registers, PC, kernel SP of currently-running process onto its kernel stack
  - Restore registers, PC, and switch to the kernel of the process-to-be-executed next.

————————————————-

`SYSCALL` is an x86-64 assembly language instruction used in modern 64-bit operating systems, including Linux, to perform a system call. It provides a faster and more efficient interface for making system calls compared to the older `INT 0x80` mechanism used in 32-bit x86 systems.

In the x86-64 architecture, system calls are made using the `SYSCALL` instruction as follows:

- Set Up Registers:
- Before invoking the system call, the user-level program sets up the system call number in the `RAX` register and any required arguments in `RDI`, `RSI`, `RDX`, `R10`, `R8`, and `R9` registers, following the calling convention.
- Execute SYSCALL:
- The `SYSCALL` instruction is then executed, which causes a transition from user mode to kernel mode, and the processor switches to the operating system's privileged execution context.
- System Call Handling:
- In the kernel, the operating system's system call handler examines the value in the `RAX` register to determine which system call the user program wants to invoke.
- Execute the System Call:
- The kernel then performs the requested operation or service on behalf of the user-level program using the provided arguments.
- Result Return:
- After completing the system call, the result (if any) is stored in the `RAX` register, which the user program can access upon returning from the system call.
- Return to User Mode:

- Finally, the operating system executes a special instruction to return from the system call, switching the processor back to user mode, and the user program continues its execution with the result of the system call.

The `SYSCALL` instruction offers several advantages over the older `INT 0x80` mechanism, including reduced overhead and improved performance, making it the preferred way to make system calls on 64-bit x86 systems running modern operating systems. However, it's important to note that the specific register usage and conventions may vary depending on the operating system and calling conventions used by the platform.