Chapter 6: Limited Direct Execution

Adam Disney

THE UNIVERSITY OF
TENNESSEE
KNOXVILLE

How To Efficiently Virtualize The CPU w/ Control

THE CRUX:
HOW TO EFFICIENTLY VIRTUALIZE THE CPU WITH CONTROL

The OS must virtualize the CPU in an efficient manner while retaining control over the system. To do so, both hardware and operating-system support will be required. The OS will often use a judicious bit of hardware support in order to accomplish its work effectively.

Performance – Direct Execution

- Let the program run directly on the CPU!
- GGEZ

Direct Execution Issues

- How do we know the program will obey the rules?
- How do we stop the program so we can time share the CPU?

Restricted Operations

- We need hardware support for a "user mode"
- The unrestricted mode is AKA "kernel mode" or "privileged mode".
- Now the process can't break the rules but how do we allow it to run privileged instructions? (e.g. read/write disk)

THE CRUX: HOW TO PERFORM RESTRICTED OPERATIONS

A process must be able to perform I/O and some other restricted operations, but without giving the process complete control over the system. How can the OS and hardware work together to do so?

System Calls

- The hardware supports this with the "trap" instruction.
- This instruction allows the process to jump into the kernel while switching to kernel mode.
- "Return-from-trap" instruction returns to the user process while switching to user mode.
- How does it know where to jump into the kernel?
  - We can't allow the user process to jump to an arbitrary location in the kernel!

Trap Table

- During boot up, the OS sets up a trap table.
- This is essentially where the OS defines the system calls it provides.
- The OS simply assigns an integer to each system call and that index in the table has a function pointer.
- Now the user can't jump to arbitrary locations in the kernel.

Retaining Control

If the user process is running, then the OS is NOT running...so how can it do anything at all?

Cooperative Approach

- Let's assume processes will behave reasonably.
- Wait for the process to make a system call to get into the OS.
- We might have a "yield" system call in this approach.
- Illegal operations will also give the OS control.
- If the user process doesn't behave, our only retort is a reboot.

Non-Cooperative Approach

- We need hardware support for this to work.
- Timer interrupts
  - The OS sets up a handler for when a timer interrupt is received.
  - The hardware must save enough state to resume the user process.

Non-Cooperative Approach

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

Scheduling

- Now that we're in the kernel via cooperative means or non-cooperative means, the scheduler must decide if it should context switch.
- We'll go into details in the followiung chapters.

Concurrency?

- What if the OS receives an interrupt while handling an interrupt?
- What if the OS receives a timer interrupt during a system call?
- Perhaps we disable interrupts while handling an interrupt.
- Use locks to handle manipulating data structures in the OS.
- There's many chapters ahead on concurrency where we'll get into the details.