CSL 301

OPERATING SYSTEMS

Lecture 3
Limited Direct Execution

Instructor
Dr. Dhiman Saha

Image Source: Google

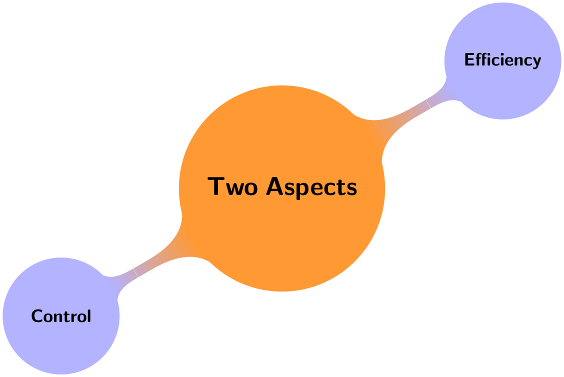
How to efficiently virtualize the CPU with control?

Two Aspects

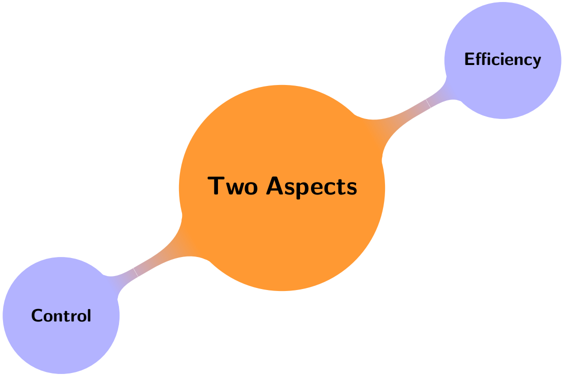
How to efficiently virtualize the CPU with control?

Efficiency
Two Aspects

How to efficiently virtualize the CPU with control?



How to efficiently virtualize the CPU with control?



Basic Idea

Direct Execution

Direct Execution Protocol (Without Limits)

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	

Problems!

- 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?
- When we are running a process, how does the operating system stop it from running and switch to another process?

"**Limited**" Direct Execution

Problem #1 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?

What if?

We let any process do whatever it wants in terms of I/O and other related operations.

The **user mode**

- A new processor mode

user mode

Code that runs in user mode is **restricted** in what it can do.

Example

- 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.

The **kernel mode**

- The OS (or kernel) run in this mode

In this mode, code that runs can do what it likes,

- including privileged operations
- issuing I/O requests
- executing all types of restricted instructions.

user mode $\leftrightarrow$ kernel mode

system call

What happens in a simple function call?

- function call $\implies$ a jump instruction
- stack $\leftarrow$ new stack frame pushed
- Stack Pointer (SP) $\leftarrow$ updated
- Old value of PC (return value) pushed to stack and PC updated
- Once function finishes, stack is popped and old PC value is retrieved effecting a return to the calling function

Stack frame

Contains return value, function arguments etc.

What changes in a system call?

- Switching from user to kernel mode
- The **trap** instruction (hidden from user)
- A separate **kernel stack** is used when in kernel mode. Why?
- Servicing the call. How?
- Switching from kernel to user mode
- The **return-from-trap** instruction

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

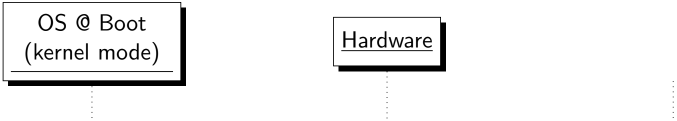
- Can/Should the calling process do this?
- The **trap table**
- Setup at **boot time** by the kernel

Limited Direct Execution Protocol

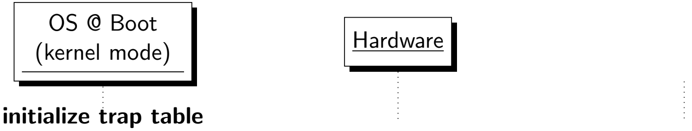
OS @ boot (kernel mode)	**Hardware**
initialize trap table	
	remember address of... syscall handler

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		

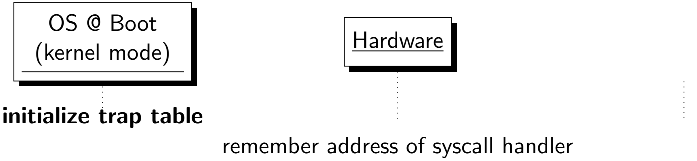
Limited Direct Execution Protocol



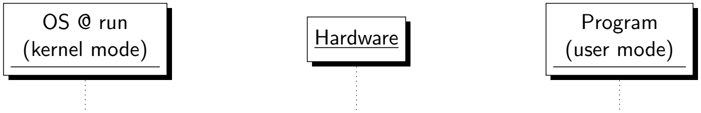
Limited Direct Execution Protocol



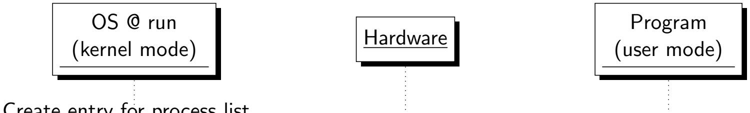
Limited Direct Execution Protocol


OS @ Boot
(kernel mode)
Hardware
initialize trap table
remember address of syscall handler

Limited Direct Execution Protocol



Limited Direct Execution Protocol



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

Limited Direct Execution Protocol

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	

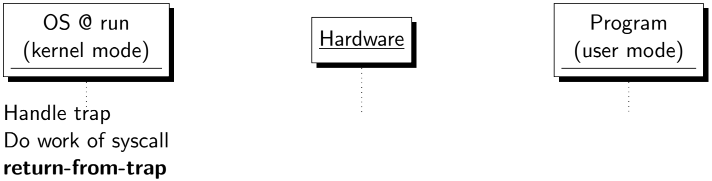
Limited Direct Execution Protocol

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

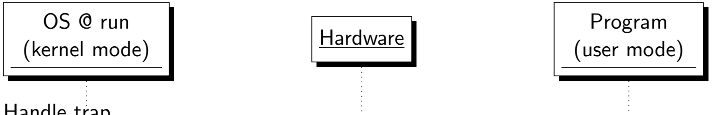
Limited Direct Execution Protocol

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	

Limited Direct Execution Protocol



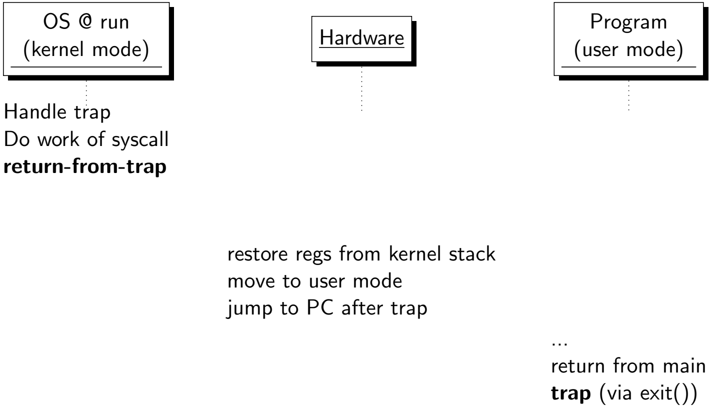
Limited Direct Execution Protocol



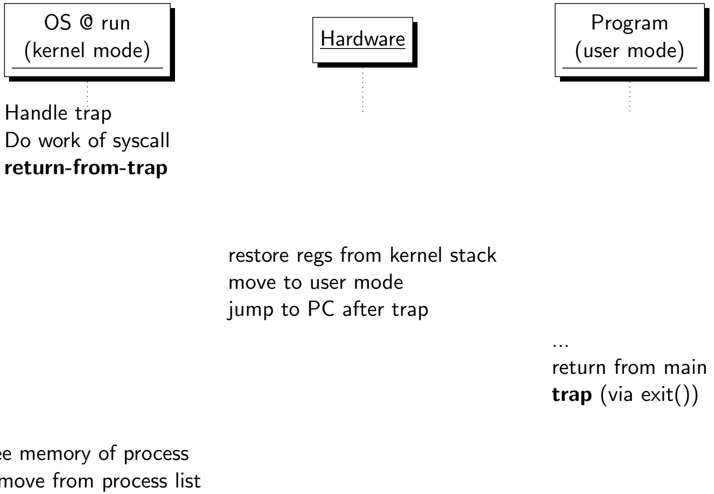
Handle trap
Do work of syscall
return-from-trap

restore regs from kernel stack
move to user mode
jump to PC after trap

Limited Direct Execution Protocol



Limited Direct Execution Protocol



system-call number

- 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 gives a level of protection. How?

Switching Between Processes

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

A Cooperative Approach

Wait For System Calls

- Switch contexts for `syscall` interrupt.
- Provide special **yield()** system call.
- Applications also transfer control to the OS when they do something illegal.

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.

A Non-Cooperative Approach:

The OS Takes Control

- How can the OS gain control of the CPU even if processes are not being cooperative?
- What can the OS do to ensure a rogue process does not take over the machine?

A timer interrupt

When the interrupt is raised, the currently running process is halted, and a pre-configured interrupt handler in the OS runs.

The Context Switch

Saving and Restoring Context

- All the OS has to do is save a few register values for the currently-executing process (onto its kernel stack)
- And restore a few for the soon-to-be-executing process (from its kernel stack).

The xv6 Context Switch Code

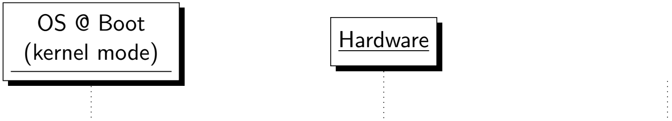
```
# void swtch(struct context **old, struct context *new);
#
# Save current register context in old
# and then load register context from new.
.globl swtch
swtch:
  # Save old registers
  movl 4(%esp), %eax  # put old ptr into eax
  popl 0(%eax)        # save the old IP
  movl %esp, 4(%eax)  # and stack
  movl %ebx, 8(%eax)  # and other registers
  movl %ecx, 12(%eax)
  movl %edx, 16(%eax)
  movl %esi, 20(%eax)
  movl %edi, 24(%eax)
  movl %ebp, 28(%eax)

  # Load new registers
  movl 4(%esp), %eax  # put new ptr into eax
  movl 28(%eax), %ebp # restore other registers
  movl 24(%eax), %edi
  movl 20(%eax), %esi
  movl 16(%eax), %edx
  movl 12(%eax), %ecx
  movl 8(%eax), %ebx
  movl 4(%eax), %esp  # stack is switched here
  pushl 0(%eax)       # return addr put in place
  ret                 # finally return into new ctxt
```

LDE Protocol (Timer Interrupt)

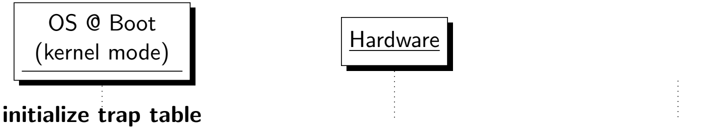
OS @ boot (kernel mode)	**Hardware**
initialize trap table	
	remember addresses of...
	syscall handler
	timer handler
start interrupt timer	
	start timer
	interrupt CPU in X ms

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
		...

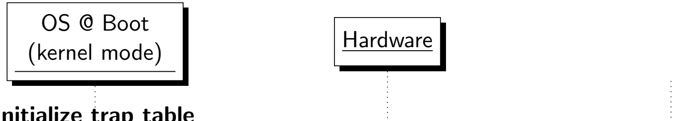
LDE Protocol with Timer Interrupt



LDE Protocol with Timer Interrupt



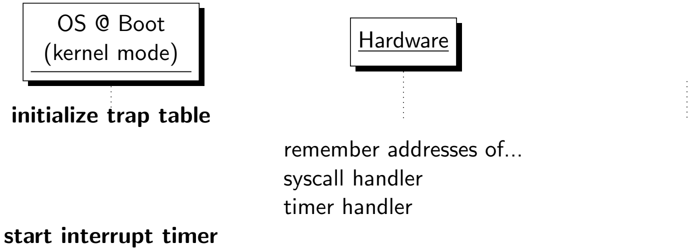
LDE Protocol with Timer Interrupt



initialize trap table

remember addresses of...
syscall handler
timer handler

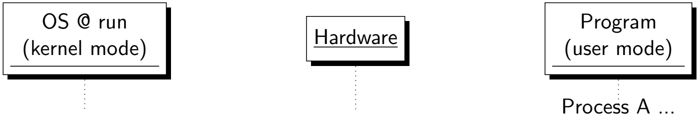
LDE Protocol with Timer Interrupt



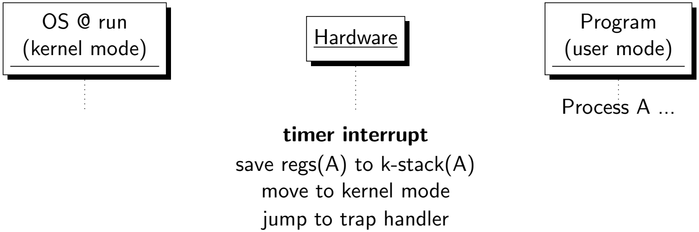
LDE Protocol with Timer Interrupt

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

LDE Protocol with Timer Interrupt



LDE Protocol with Timer Interrupt



LDE Protocol with Timer Interrupt

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)		

LDE Protocol with Timer Interrupt

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	

LDE Protocol with Timer Interrupt

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 ...

Dealing with concurrency

What happens if, during interrupt or trap handling, another interrupt occurs?

- Idea: Disable interrupts
- Locking mechanisms
- Will be discussed in lectures on concurrency

- 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**.