Join on Zoom if you are remote (see website for link).

377 Operating Systems

Lecture 04: Limited Direct Execution

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.

Basic Technique: Limited Direct Execution

To make a program run as fast as one might expect, not surprisingly OS developers came up with a technique, which we call:

limited direct execution.

Basic Technique: Limited Direct Execution

To make a program run as fast as one might expect, not surprisingly OS developers came up with a technique, which we call:

limited direct execution.

The direct execution part of the idea is simple: just run the program directly on the CPU.

Basic Technique: Limited Direct Execution

To make a program run as fast as one might expect, not surprisingly OS developers came up with a technique, which we call:

limited direct execution.

The direct execution part of the idea is simple: just run the program directly on the CPU.

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	

Basic Technique: Limited Direct Execution

To make a program run as fast as one might expect, not surprisingly OS developers came up with a technique, which we call:

limited direct execution.

The direct execution part of the idea is simple: just run the program directly on the CPU.

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	

Basic Technique: Limited Direct Execution

To make a program run as fast as one might expect, not surprisingly OS developers came up with a technique, which we call:

limited direct execution.

The direct execution part of the idea is simple: just run the program directly on the CPU.

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	

Basic Technique: Limited Direct Execution

To make a program run as fast as one might expect, not surprisingly OS developers came up with a technique, which we call:

limited direct execution.

The direct execution part of the idea is simple: just run the program directly on the CPU.

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	

Basic Technique: Limited Direct Execution

Pretty simple…

But, there are problems…

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	

Basic Technique: Limited Direct Execution

Problem #1

Restricted Operations

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?

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	

Basic Technique: Limited Direct Execution

Problem #2

Context Switching

When we are running a process, how does the operating system stop it from running and switch to another process, thus implementing the time sharing we require to virtualize the CPU?

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	

Problem #1: Restricted Operations

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?

Here is a bad solution

We could let a process do whatever it wants.

Here is a bad solution

We could let a process do whatever it wants.

But, this is likely to be a very bad idea.

It would prevent the construction of many kinds of systems that are desirable.

For example, we can't simply let any user issue I/Os to disk.

If we did, a process could simply read or write an entire disk and thus all protections would be lost.

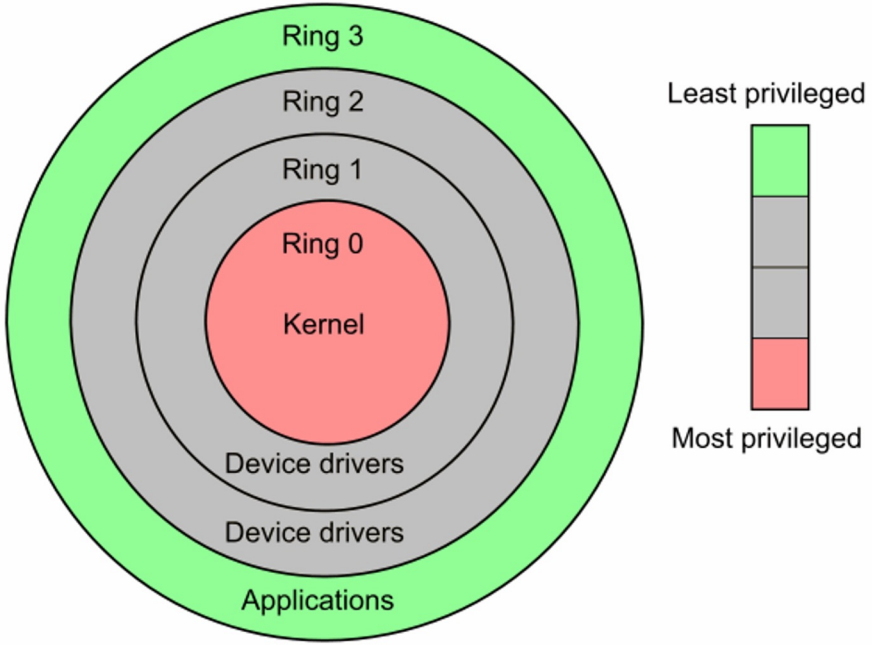
Introducing: User Mode

Thus, 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, no I/O requests!

If you try, then



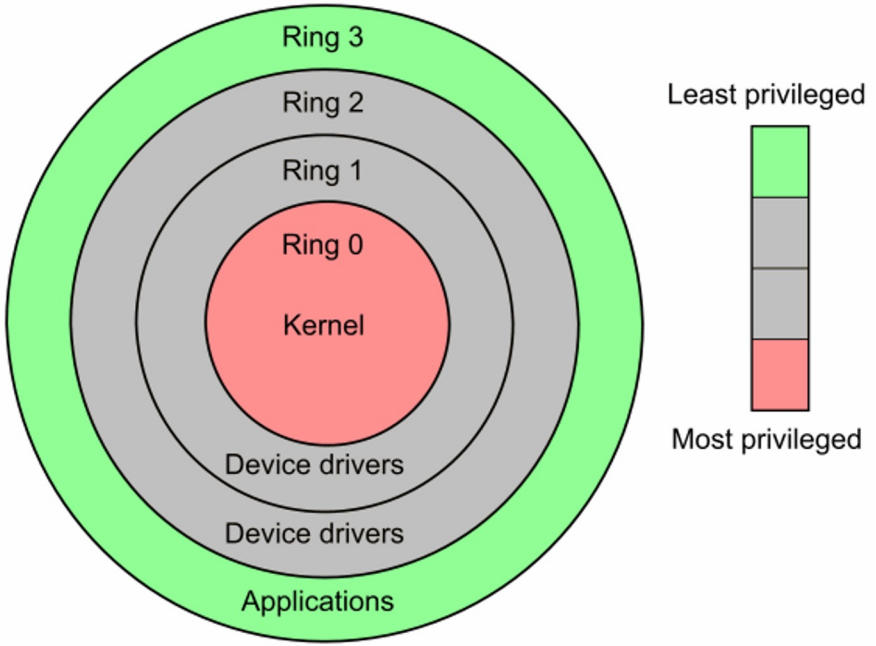
Introducing: Kernel Mode

And the other processor mode is known as kernel mode.

Code that runs in kernel mode can do whatever it wants.

For example, I/O requests!

If you try, then



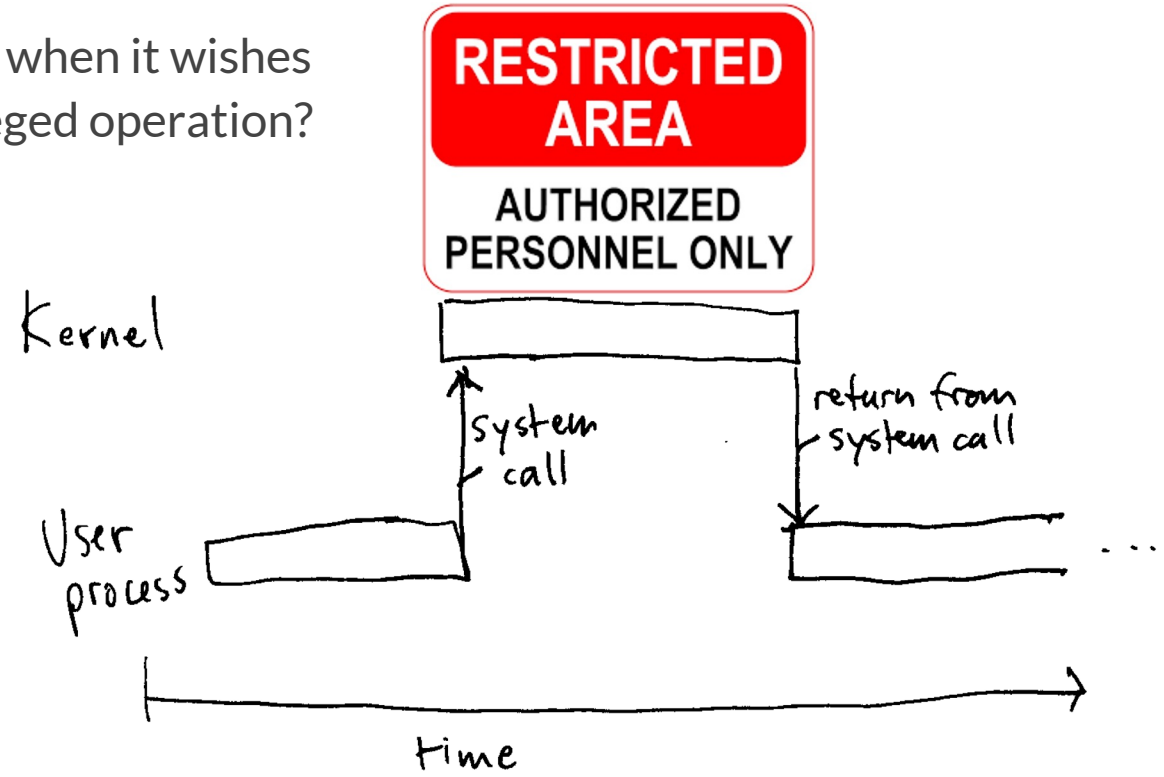
Question?

What should a user process do when it wishes to perform some kind of privileged operation?

Question?

What should a user process do when it wishes to perform some kind of privileged operation?

To enable this, virtually all modern hardware provides the ability for user programs to perform a system call.



System Calls

System calls expose key pieces of functionality to user programs

- Accessing the file system
- Creating and destroying processes
- Communicating with other processes
- Allocating more memory

Most operating systems provide a few hundred system calls

- POSIX
- Early Unix systems had a more concise set of 20

Question?

If the kernel allows privileged functionality, how does a system call get us into the kernel?

How are system calls implemented?

We will describe this process in a “general” way.

- There are a lot of specifics and differences between x86 and x86-64
- Read: https://blog.packagecloud.io/eng/2016/04/05/the-definitive-guide-to-linux-system-calls/

Executing a System Call

To execute a system call:

- **A program executes a special trap instruction**
  - Jumps into the kernel and raises the privilege level at the same time
  - Privileged operations **can be** performed

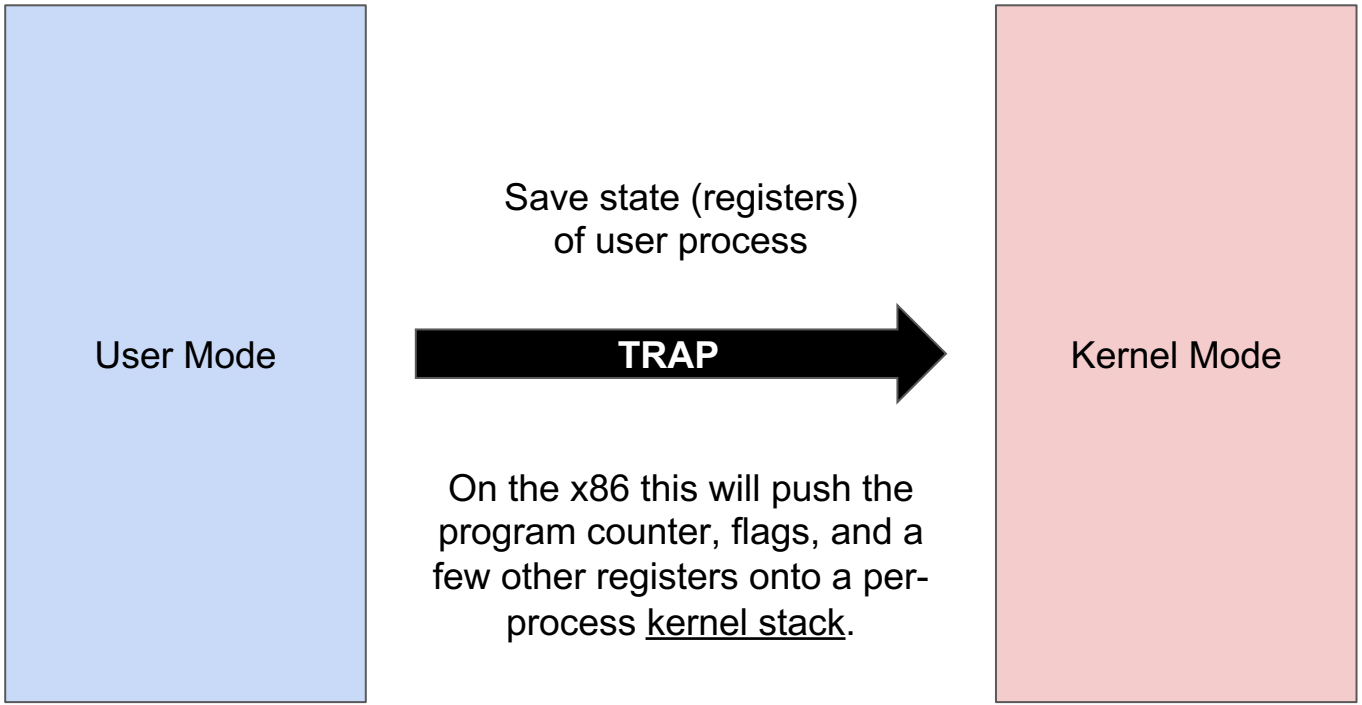
Executing a System Call

To execute a system call:

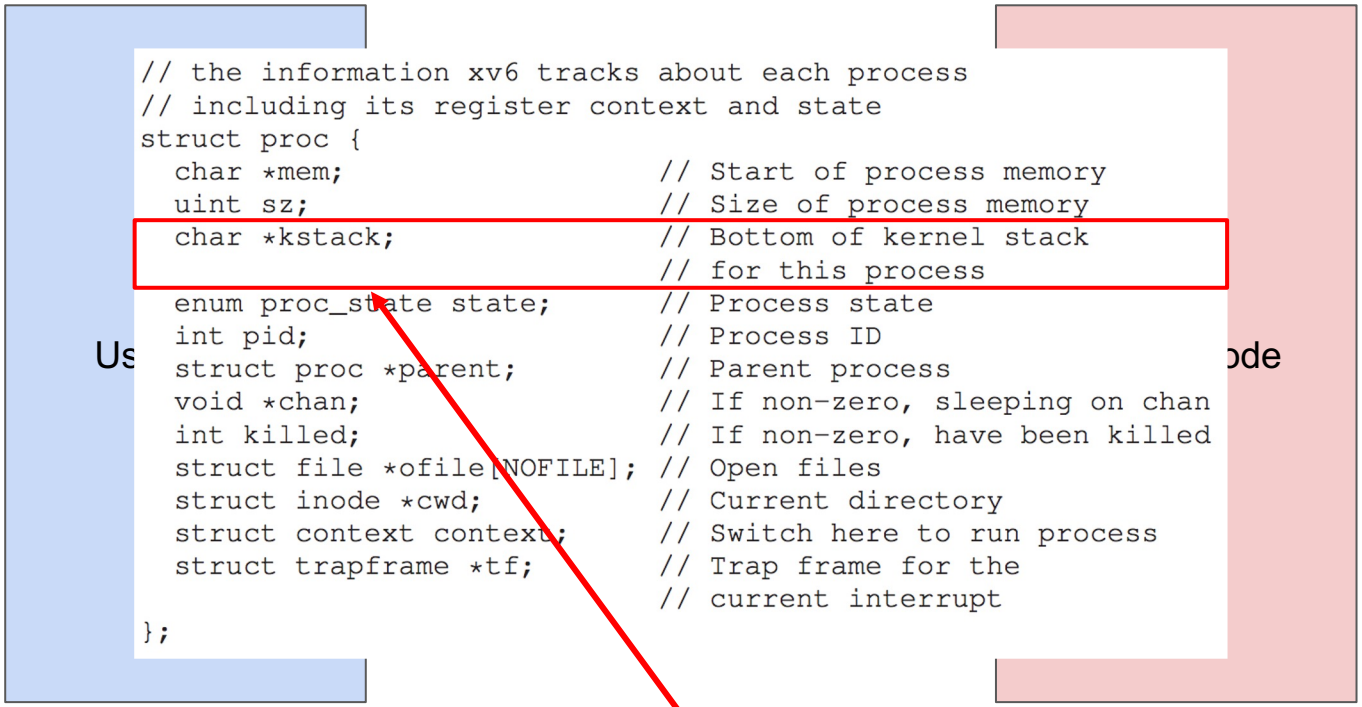
- **A program executes a special trap instruction**
  - Jumps into the kernel and raises the privilege level at the same time
  - Privileged operations **can be** performed

- **When finished, the OS calls a special return-from-trap instruction**
  - Returns to the calling program and lowers the privilege level at the same time
  - Privileged operations **can not be** performed

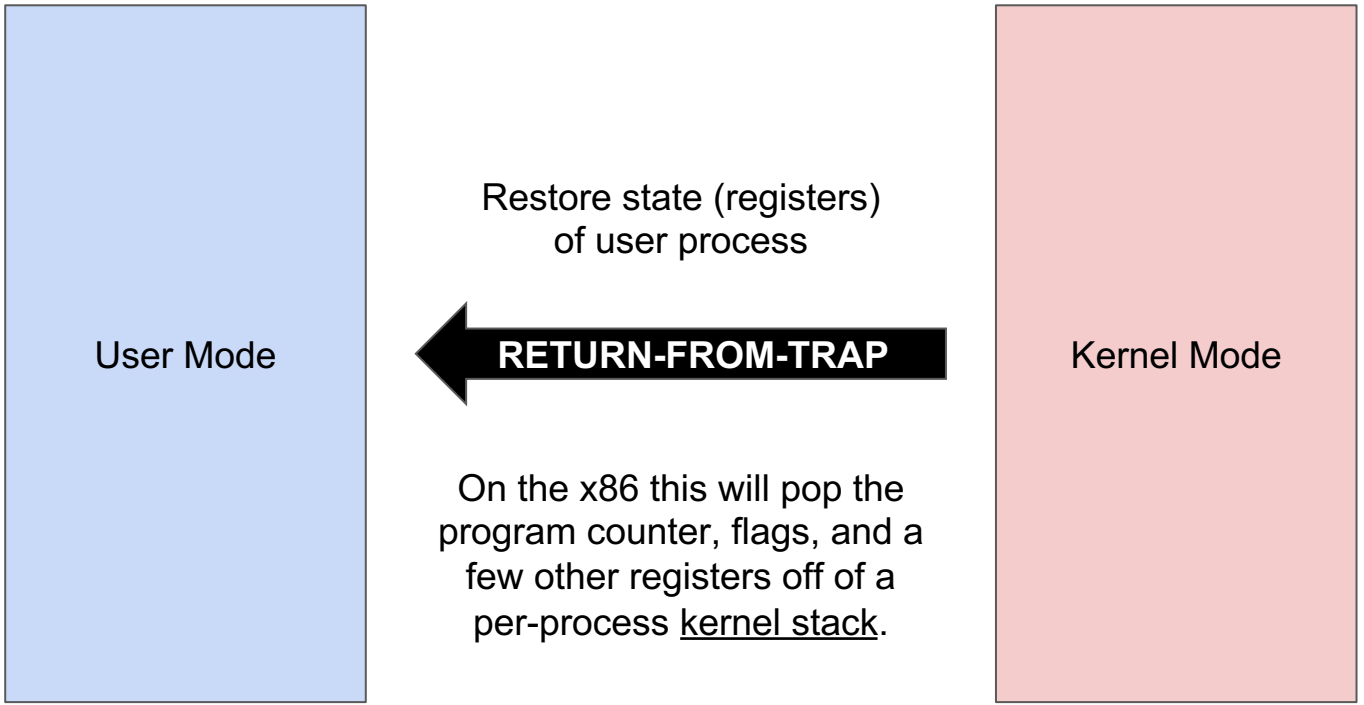
Preparing for a System Call: Saving State



Preparing for a System Call: Saving State



Completing a System Call: Restoring State



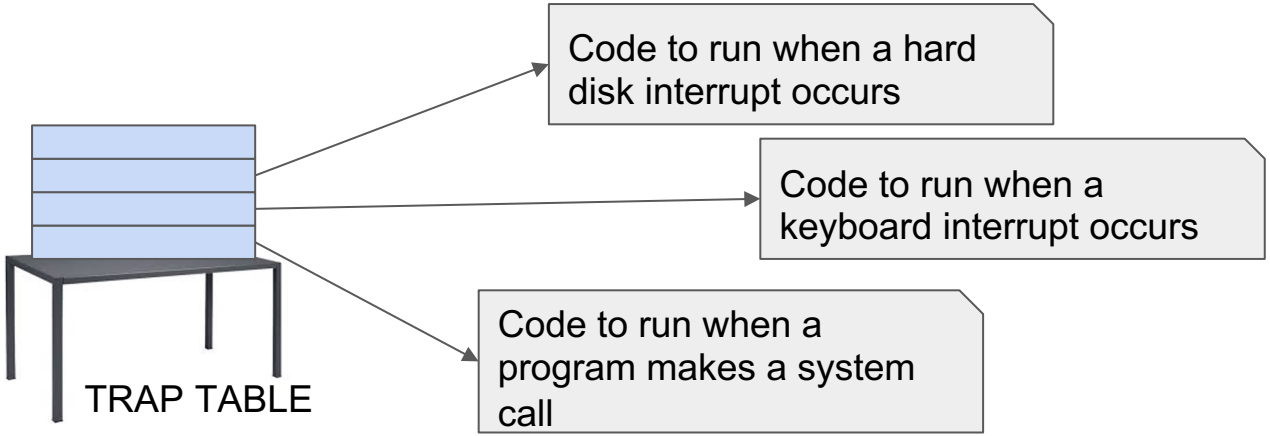
How does the trap know which code to run in the OS?

Can we simply use an address?

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.

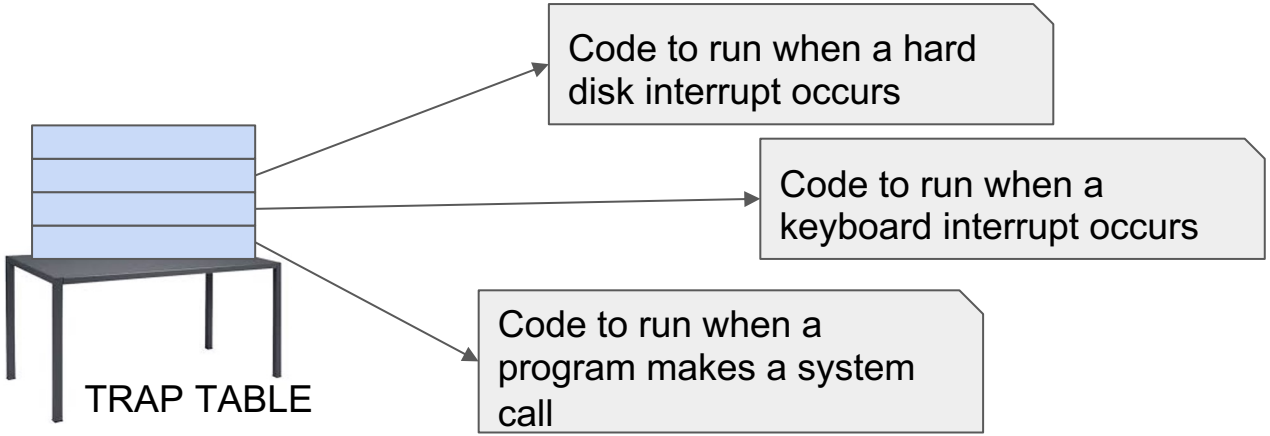
One of the first things the OS thus does is to tell the hardware what code to run when certain exceptional events occur (special instruction).



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 (special instruction).



Once the hardware is informed, it remembers the location of these handlers until the machine is next rebooted, and thus the hardware knows what to do (i.e., what code to jump to) when system calls and other exceptional events take place.

System Call Numbers

"All problems in computer science can be solved by another level of indirection"

- David Wheeler

To specify the exact system call, a system call number is usually assigned to each system call.

System Call Numbers

"All problems in computer science can be solved by another level of indirection"

- David Wheeler

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.

System Call Numbers

"All problems in computer science can be solved by another level of indirection"

- David Wheeler

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.

System Call Numbers

"All problems in computer science can be solved by another level of indirection"

- David Wheeler

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.

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

OS @ run (kernel mode)	Hardware	Program (user mode)

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		

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	

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

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	

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		

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	

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	
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()`)

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	
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

Problem #2: Switching Between Processes

THE CRUX: HOW TO REGAIN CONTROL OF THE CPU

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

A Cooperative Approach: Wait for System Calls

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: OS Takes Control

THE CRUX: HOW TO GAIN CONTROL WITHOUT COOPERATION

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 Non-Cooperative Approach: OS Takes Control

THE CRUX: HOW TO GAIN CONTROL WITHOUT COOPERATION

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?

TIP: USE THE TIMER INTERRUPT TO REGAIN CONTROL

The addition of a **timer interrupt** gives the OS the ability to run again on a CPU even if processes act in a non-cooperative fashion. Thus, this hardware feature is essential in helping the OS maintain control of the machine.

Saving and Restoring Context

Now that the OS has regained control, whether cooperatively via a system call, or more forcefully via a timer interrupt, a decision has to be made:

- **whether to continue running the currently-running process**
- **or switch to a different one.**

Saving and Restoring Context

Now that the OS has regained control, whether cooperatively via a system call, or more forcefully via a timer interrupt, 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

we will discuss scheduling policies in detail later.

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.

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

Limited Direct Execution Protocol (timer interrupt)

OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A
		...

Limited Direct Execution Protocol (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	

Limited Direct Execution Protocol (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)		

Limited Direct Execution Protocol (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	

Limited Direct Execution Protocol (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
		...

Limited Direct Execution Protocol (timer interrupt)

Hmm... what happens when, during a system call, a timer interrupt occurs?" or "What happens when you're handling one interrupt and another one happens?

Hmm... what happens when, during a system call, a timer interrupt occurs?" or "What happens when you're handling one interrupt and another one happens?

Disable Interrupts

Hmm... what happens when, during a system call, a timer interrupt occurs?" or "What happens when you're handling one interrupt and another one happens?

Disable Interrupts

Locking schemes to protect concurrent access to OS data structures


The End