Operating Systems
CMPSCI 377
Spring 2020

Clicker Question

What is the best way to describe how user programs run?

(A) directly on the CPU

(B) on top of the Operating System, which runs on top of the CPU

(C) inside of the OS

(D) on the GPU

Answer on Next Slide

Today's Class

- 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.

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 <u>direct execution</u> 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.

Clicker Question

If there are no restrictions on the user process, what can it do?

(A) corrupt the memory of the kernel

(B) halt the machine

(C) run forever and starve other processes

(D) all of the above

Answer on Next Slide

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 process 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.

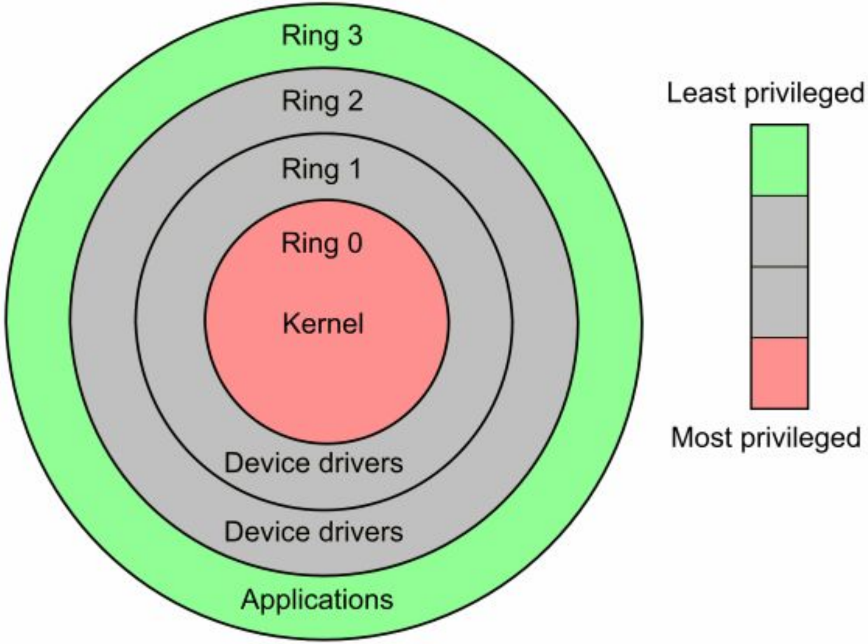
Introducing: User Mode

Thus, the approach we take is to introduce a new processor mode
known as <u>user mode</u>.

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

For example, no I/O requests!

If you try, then 



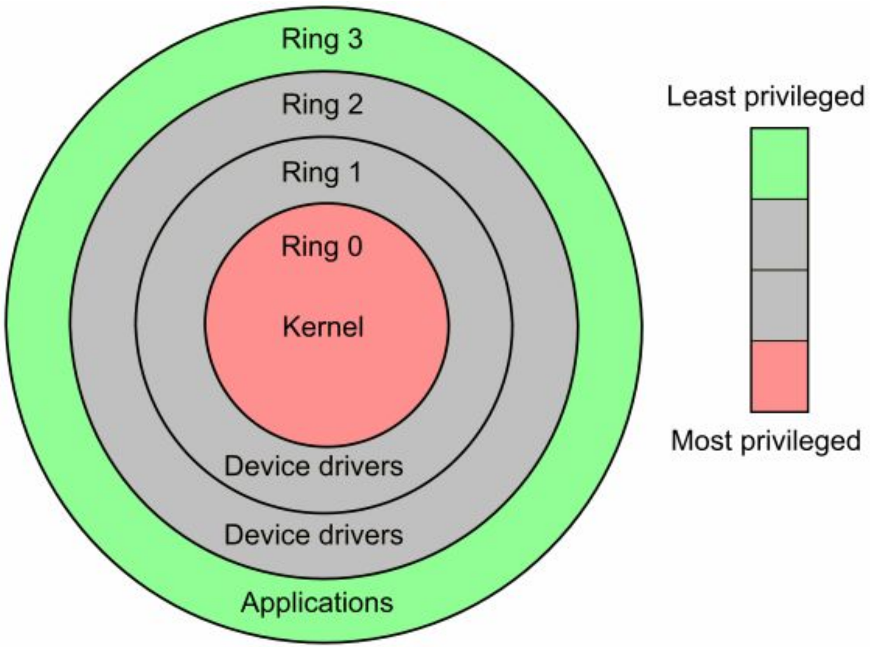
Introducing: Kernel Mode

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



Clicker Question

True or False: A user program can do something to change the machine to kernel mode.

(A) True

(B) False

Answer on Next Slide

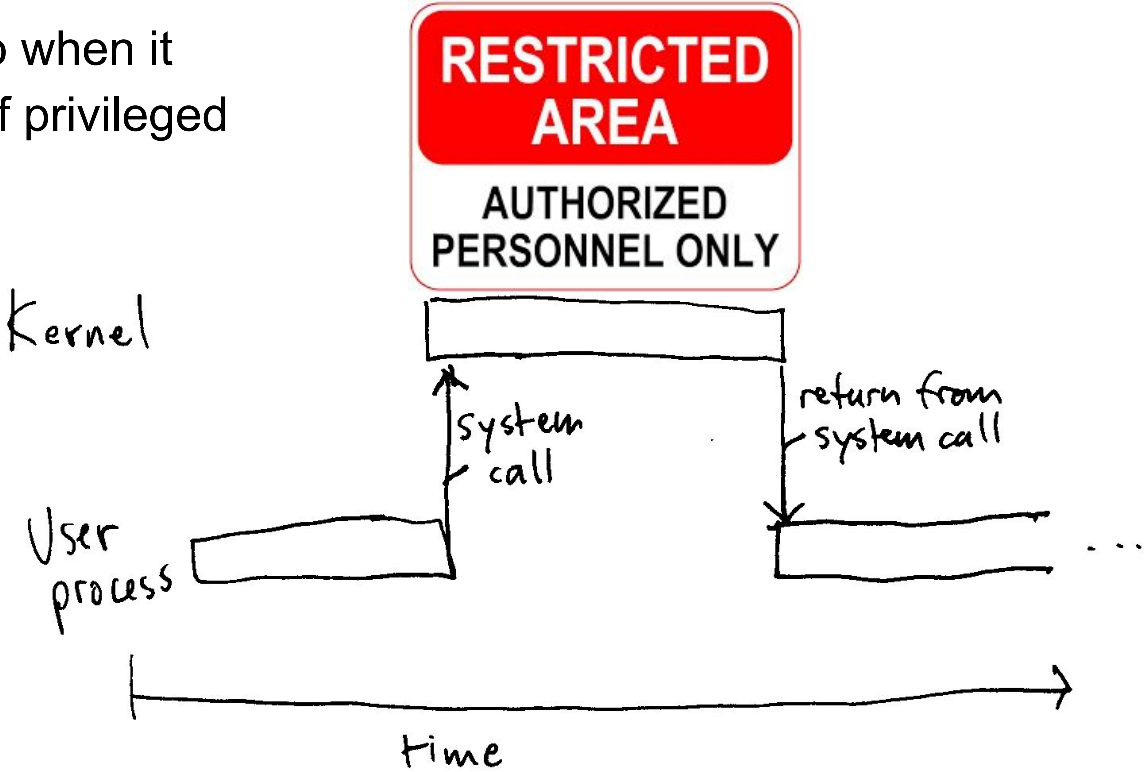
Question?

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 <u>system call</u>.



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 <u>trap</u> instruction
  - Jumps into the kernel and raises the privilege level <u>at the same time</u>
  - Privileged operations **can be** performed

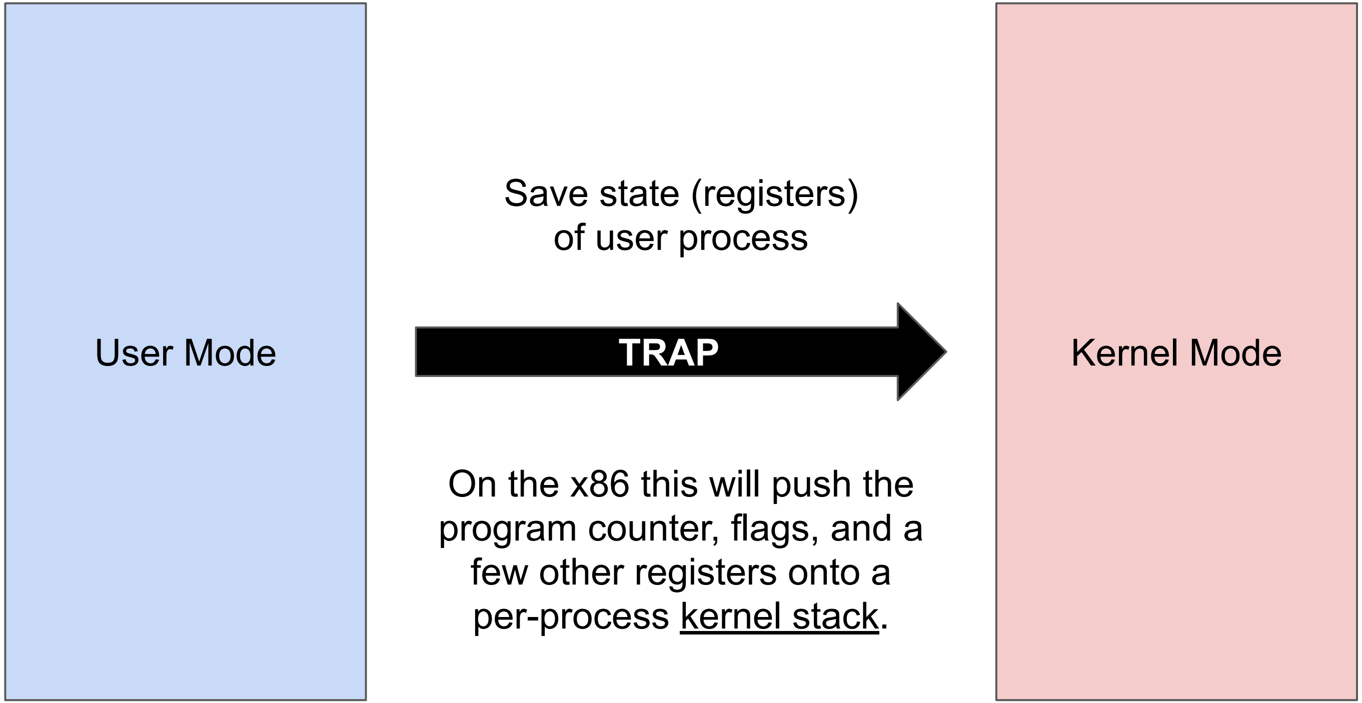
Executing a System Call

To execute a system call:

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

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

Preparing for a System Call: Saving State

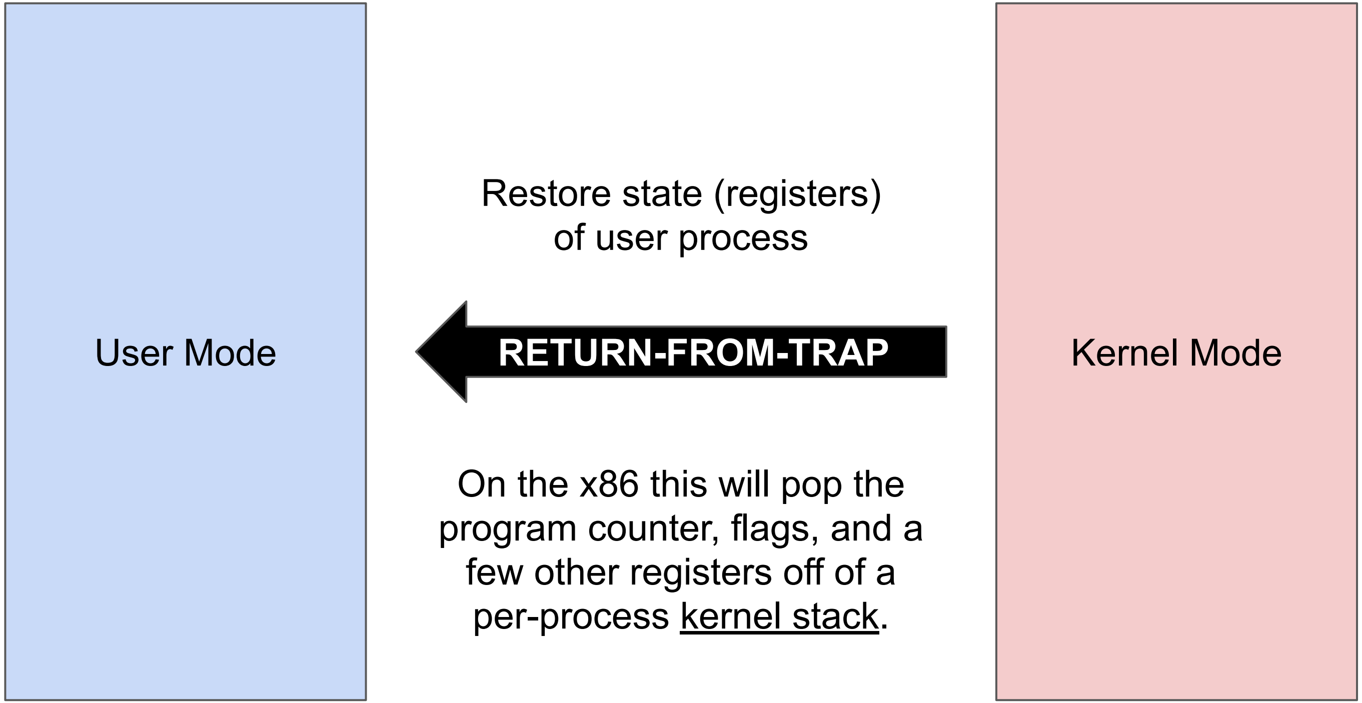


Preparing for a System Call: Saving State

```
// the information xv6 tracks about each process
// including its register context and state
struct proc {
  char *mem;                   // Start of process memory
  uint sz;                     // Size of process memory
  char *kstack;                // Bottom of kernel stack
                               // for this process
  enum proc_state state;       // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE]; // Open files
  struct inode *cwd;           // Current directory
  struct context context;      // Switch here to run process
  struct trapframe *tf;        // Trap frame for the
                               // current interrupt
};
```

Remember this from last time?

Completing a System Call: Restoring State



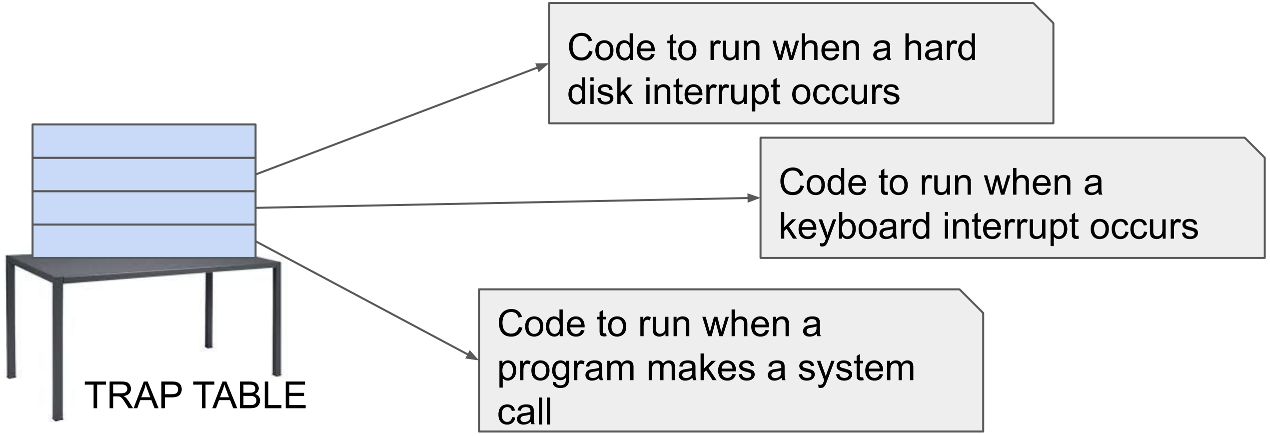
How does the trap know which code to run in the OS?

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.

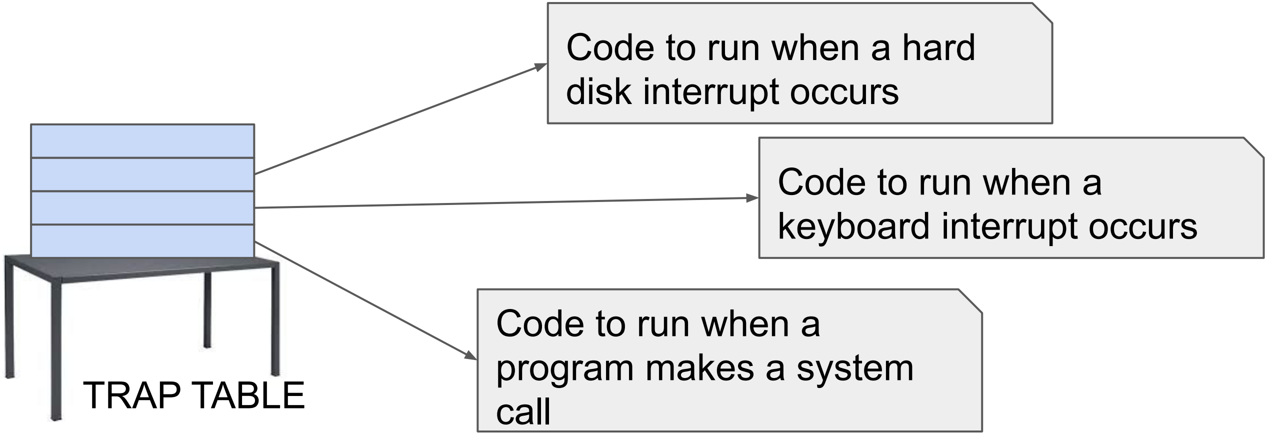
One of the first things the OS thus does is to tell the hardware what code to run when certain exceptional events occur (special instruction).



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 <u>system calls</u> and other <u>exceptional events</u> take place.

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

- David Wheeler

System Call Numbers

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

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

- David Wheeler

System Call Numbers

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.

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

- David Wheeler

System Call Numbers

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.

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

- David Wheeler

System Call Numbers

To specify the exact system call, a <u>system call number</u> 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

Limited Direct Execution Protocol

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		

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

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		

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	

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

Kernel Stack vs User Space Stack?

https://stackoverflow.com/questions/12911841/kernel-stack-and-user-space-stack

142

> 1. *What's the difference between kernel stack and user stack ?*

In short, nothing - apart from using a different location in memory (and hence a different value for the stackpointer register), and usually different memory access protections. I.e. when executing in user mode, kernel memory (part of which is the kernel stack) will not be accessible even if mapped. Vice versa, without explicitly being requested by the kernel code (in Linux, through functions like `copy_from_user()`), user memory (including the user stack) is not usually directly accessible.

> 2. *Why is [a separate] kernel stack used ?*

Separation of privileges and security. For one, userspace programs can make their stack(pointer) anything they want, and there is usually no architectural requirement to even have a valid one. The kernel therefore cannot *trust* the userspace stackpointer to be valid nor usable, and therefore will require one set under its own control. Different CPU architectures implement this in different ways; x86 CPUs automatically switch stackpointers when privilege mode switches occur, and the values to be used for different privilege levels are configurable - by privileged code (i.e. only the kernel).

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 next week.

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	

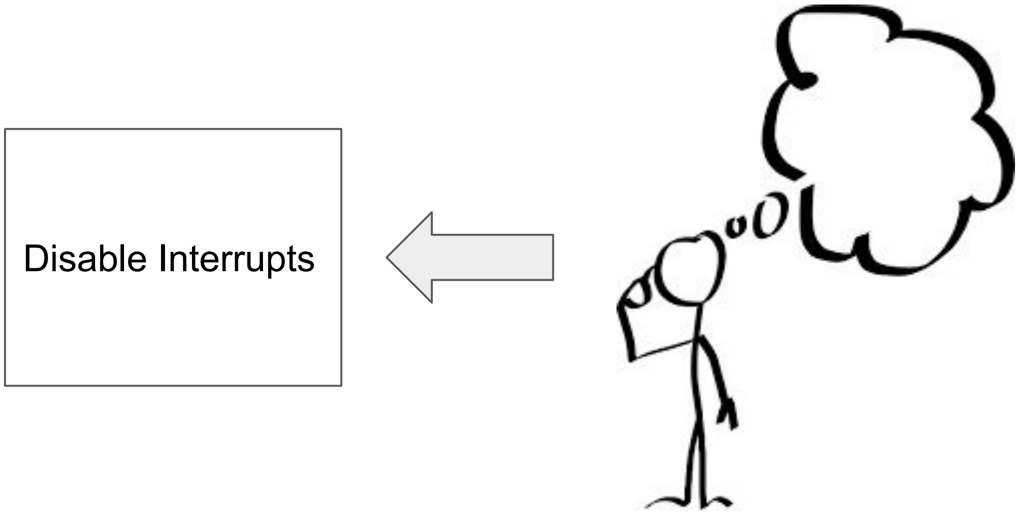
Handle the trap
Call `switch()` routine
  save regs(A) to proc-struct(A)
  restore regs(B) from proc-struct(B)
  switch to k-stack(B)
return-from-trap (into B)		

	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?



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

Clicker Question

When a timer interrupt occurs causing a context switch from process A to B, what is missing here?

Use Process A running -> Timer interrupt occurs -> CPU save user registers in kernel stack -> Kernel Code -> Kernel saves kernel registers -> Kernel switches to another kernel thread -> <MISSING> -> Process B runs

(A) CPU saves registers to kernel stack

(B) Kernel invokes return-from-trap

(C) Process B calls a trap instruction

(D) The CPU halts

Answer on Next Slide

Next Time

CPU Scheduling