



Processes: OS Mechanisms

**ICS332
Operating Systems**

Direct Execution

- Figure 6.1 in OSTEP shows a simple possible timeline for an OS to run a program (slightly modified below):

os	Program
Create PCB and add it to the Process Table	
Allocate memory for the process	
Load the program into memory	
Set up the stack with argc/argv	
Clear the registers	
Starts the fetch-decode-execute (at 1st instruction in main)	Run main()
	Return from main
Free memory of the process	
Remove PCB from the Process Table (or keep as a zombie)	

Direct Execution: Not a Good Idea

- The approach on the previous slides has two big problems
- **Problem #1:** If the process needs to access hardware resources (e.g., to write to disk), then the only option is to give the process full access to the hardware
 - This was the case in the 60's, but it's WAY too dangerous
 - A bug in a user program could corrupt hardware status, bring the machine down, overwrite data, ...
- **Problem #2:** How do we kick a process out of the CPU and give the CPU to another process?
 - We can't just say "let's start the fetch-decode-execute cycle of a program and hope that it doesn't hog the CPU"
 - For that matter, what if a process goes into an infinite loop as a bug?
- We need to **limit** the way in which a process runs on the hardware
- In other words, we need mechanisms for virtualizing the CPU to solve both problems above

Limited Execution: Restricted Operations

- The OS cannot just be a “library” that a user program can call
 - Because then the program would have complete control over the system and do dangerous things and/or hog the CPU
- So when my program places a syscall like `read()` , what happens ***must be different*** from what happens when my program calls a regular function I have implemented myself, like `compute_stuff()`
- **This is done by building CPUs that have two kinds of instructions!**
- **Unprotected** instruction that a program can execute at any time
- **Protected** (or **Privileged**) instructions that do “special” things and that a program can’t just execute in normal operation

User-Mode vs. Kernel-Mode

- All (first in the 60s and mainstream since the late 70s) CPUs support (at least) two modes of execution:
 - **User Mode**: protected instructions cannot be executed
 - **Kernel Mode**: all instructions can be executed
- User code executes in user mode
- Kernel code executes in kernel mode
- The mode is indicated by a status bit (the **mode bit**) in a protected control register in the CPU
- The CPU checks the mode bit before executing a protected instruction
- MS-DOS had only one mode (because it was designed for the 8086 which had no kernel mode bit)
 - Which is very scary now, in hindsight

User-Mode vs. Kernel-Mode

- Steps are added to the Decode stage in the Fetch-Decode-Execute cycle:
 - Decode instruction and look up some internal hardware table of the opcodes that are protected?
 - If the instruction's opcode is in that table and the mode bit is **not** set to “Kernel mode”, abort and raise a **trap** (that the OS will answer by terminating the program saying something like “not allowed”)
 - Otherwise, execute the instruction
- In current CPUs there are actually multiple modes (multiple levels in the kernel, multiple levels in the CPU)
- Question: which instructions should be protected?

Which Instructions are Protected?

- The instruction to change the mode bit
 - Obviously :)
- Basically all instructions that directly control the hardware
 - Halt the CPU
 - Update the CPU's control registers (more later...)
 - Change the system clock
 - Read/Write to registers of I/O device controllers
- Therefore, all these operations can only happen in Kernel mode and only kernel code can use them
- Essentially, the kernel is the only trusted software component that is allowed to interact with hardware components directly
- Which is why we have syscalls to say “please execute to Kernel code on my behalf”

Syscalls: How do they work?

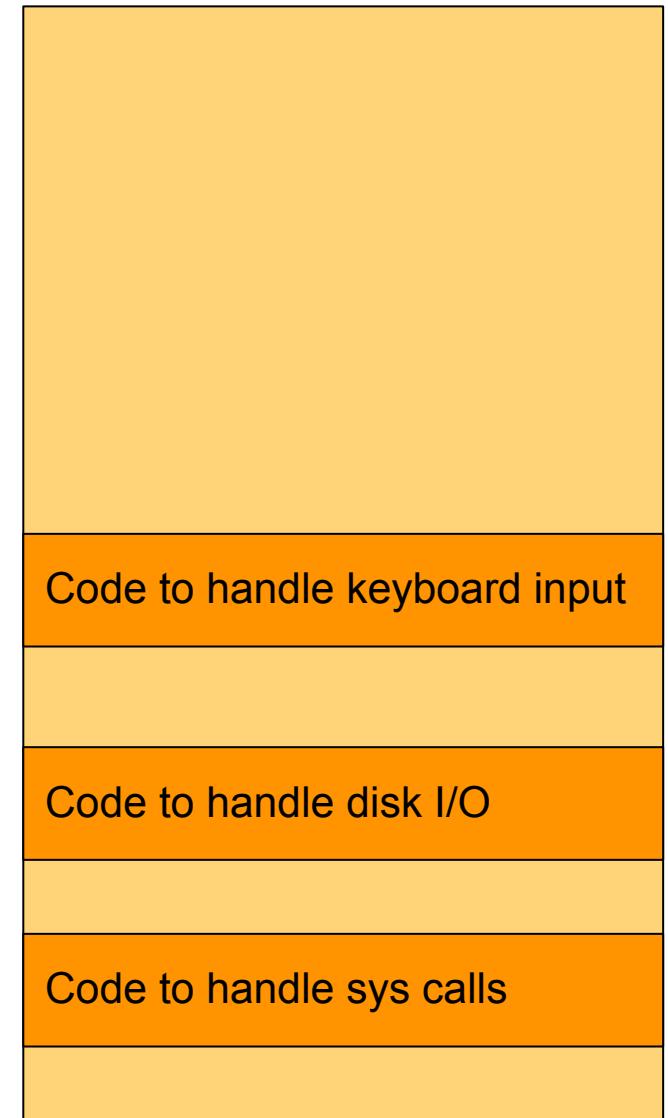
- The user code runs in user mode
- The kernel code runs in kernel mode
- So the mode bit must change for doing a syscall!
- This is exactly why the CPU has a special “syscall” instruction
- This instruction is a trap to which the Kernel must react
 - Remember that the Kernel is basically a big event handler, and that a trap is an event (cause by a program’s execution)

The Trap Table

- At boot time, the OS initializes a **Trap Table**
 - On the x86 architecture, it's called the Interrupt Descriptor Table
- The Trap Table is stored in RAM, and the CPU has a register that points to it
- For each event type that the CPU could receive, this table indicates the address in the kernel of the code that should be run to react to the event
- Whenever an event occurs the CPU can just do:
 - Look at the Trap Table in RAM
 - Lookup the entry in the Trap Table for the event and find the kernel handler's address
 - Set the mode bit to "Kernel"
 - Jump to the kernel handler and fetch-decode-execute it
- Let's look at this on a picture...

The Trap Table

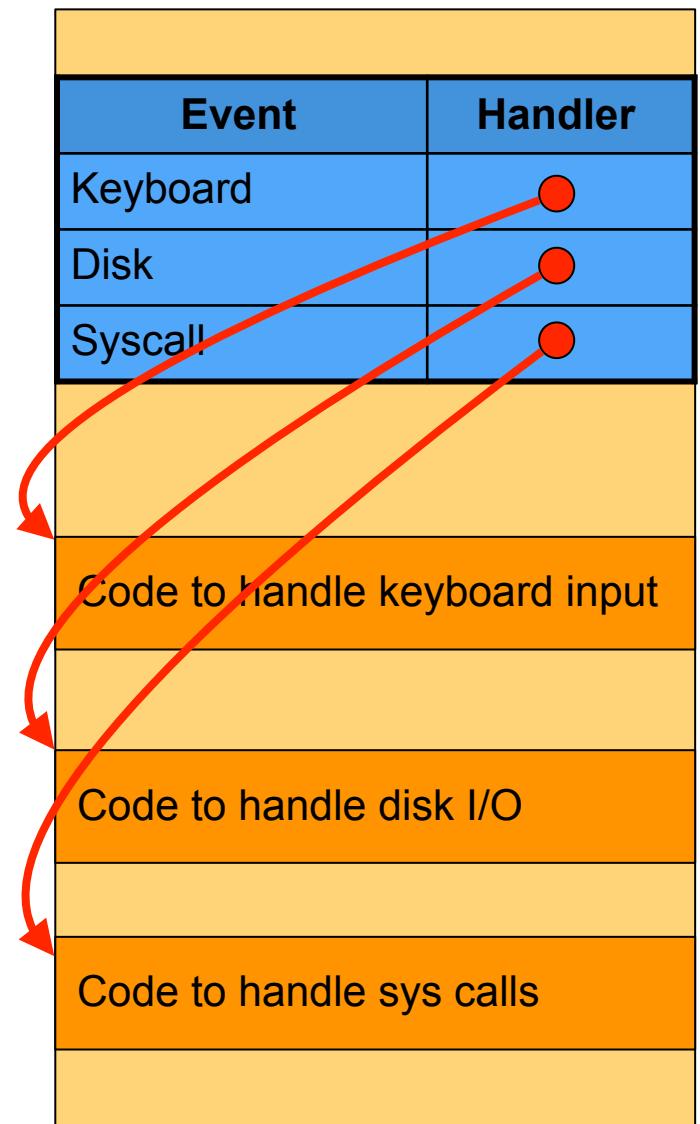
- At boot time, the kernel is loaded into RAM
- The kernel code includes handlers, i.e., pieces of code that should execute to answer particular events
- In this example, we consider
 - a “keyboard event” handler
 - a “disk I/O event” handler
 - a “syscall event” handler



Kernel in RAM

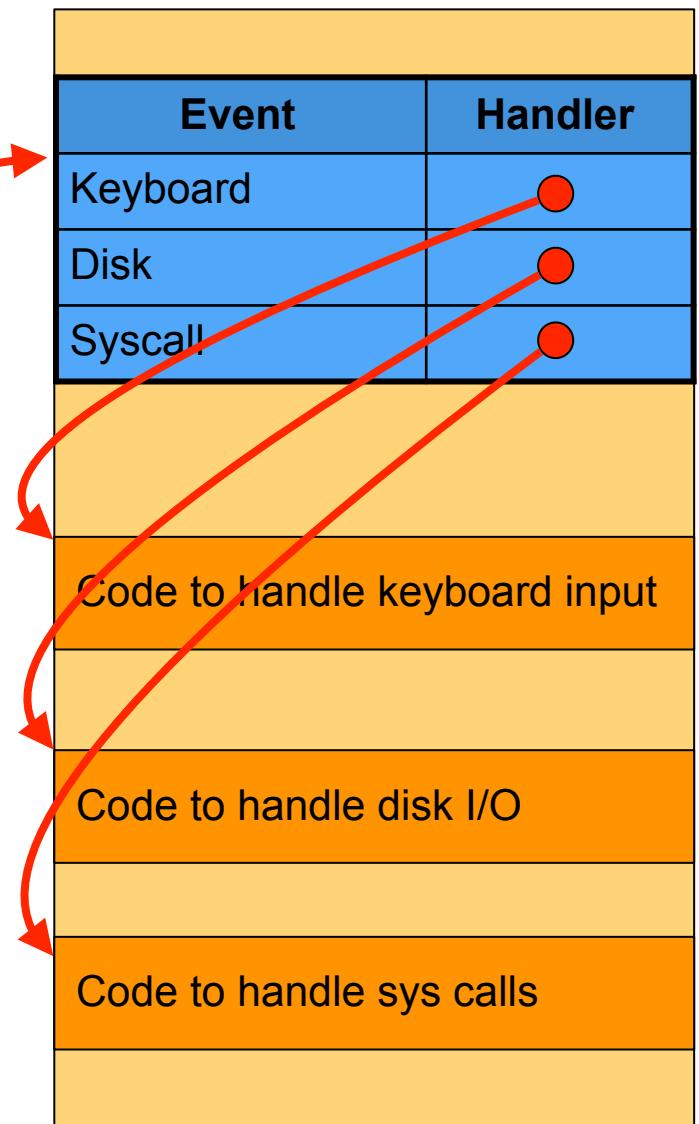
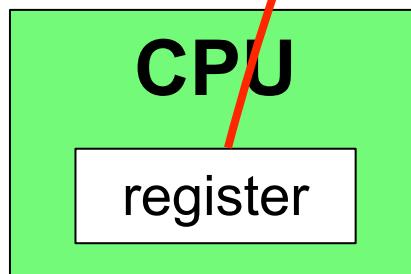
The Trap Table

- At boot time, in RAM a Trap Table is created as an array of consecutive bytes
- Entry each event type is set to the address of the first instruction of the corresponding kernel even handler code
 - Of course, each event is described as an integer, which is simply an index into the Trap Table, which is just an array of addresses



The Trap Table

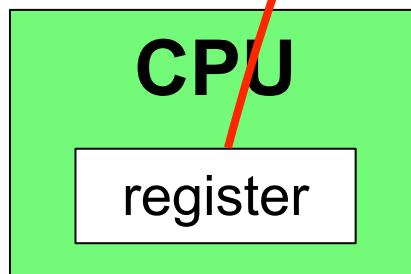
- A special register on the CPU is initialized with the address of the first byte of the Trap Table



Kernel in RAM

The Trap Table

- This is how the Kernel is able to react to all event (Is everything in this course about indirection?)



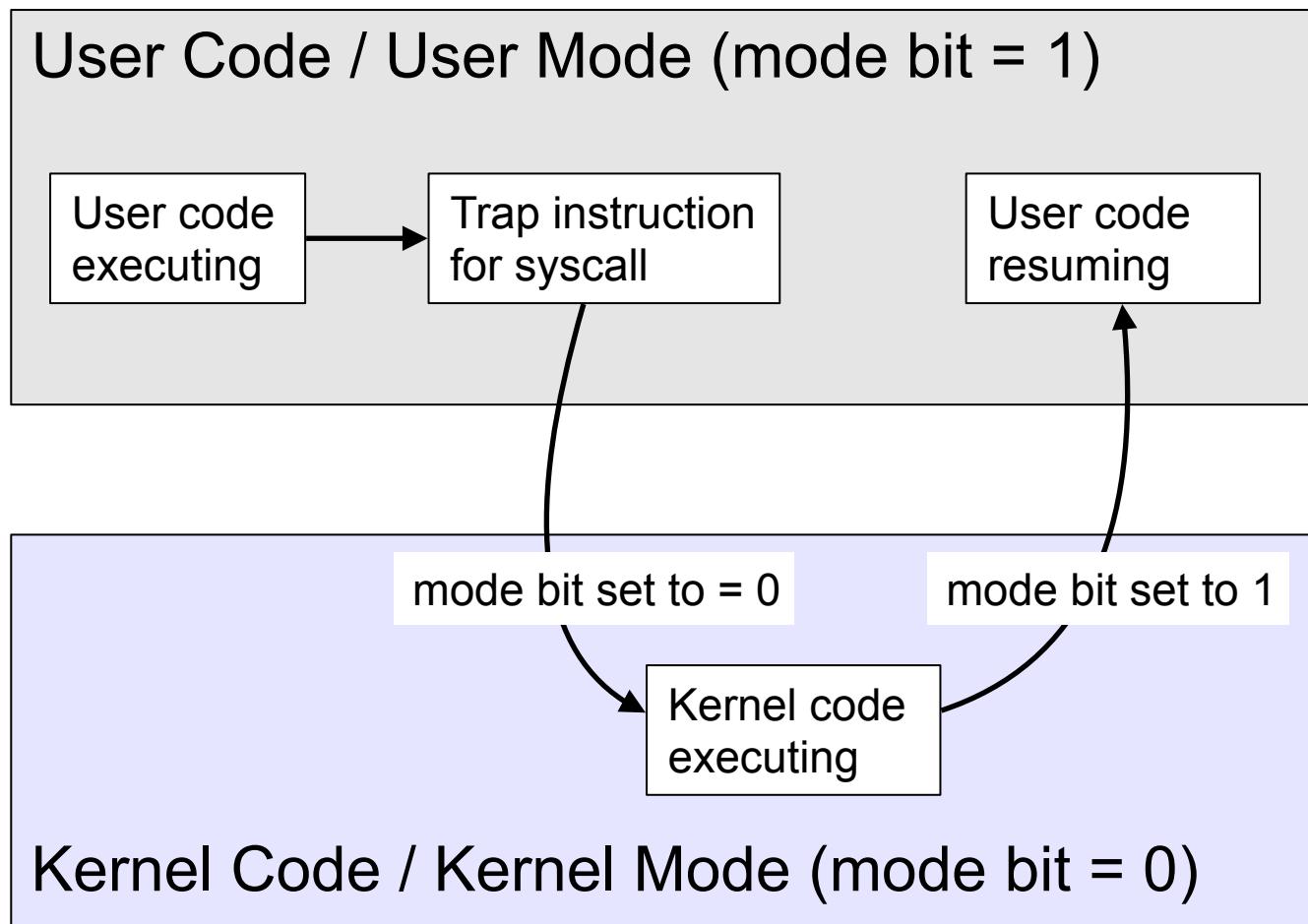
Event	Handler
Keyboard	Code to handle keyboard input
Disk	Code to handle disk I/O
Syscall	Code to handle sys calls

Kernel in RAM

The “trap” Instruction

- A CPU has an instruction to trigger the “I want to do a system call” event, often called the “trap instruction”
 - On the x86 architecture the instruction is called `int` (short for interrupt)
 - Nothing to do with an integer!
- The trap instruction does:
 - Set the mode bit to “kernel”
 - Jump to the “handle system call” kernel code
 - Set the mode bit to “user”
 - Jump back to user code
- There are many syscalls, but a single syscall handler
- Therefore, the user must specify which syscall to run as a **syscall number**
- The handler checks that the syscall number is valid, and then jumps to the corresponding kernel code
- Yes, there is a table that says for each syscall number what the address in the kernel of the code for that syscall is (`/usr/src/linux-headers-*/include/uapi/asm-generic/unistd.h`)

On a Picture



The New York Times

The software patches could slow the performance of affected machines by 20 to 30 percent, said Andres Freund, an independent software developer who has tested the new Linux code. The researchers who discovered the flaws voiced similar concerns



**News headlines
on 1/3/2018
“Intel chip bug”**

What's a kernel?

PCWorld
FROM IDG

The kernel inside a chip is basically an invisible process that facilitates the way apps and functions work on your computer. It has complete control over your operating system. Your PC needs to switch between user mode and kernel mode thousands of times a day, making sure instructions and data flow seamlessly and instantaneously. Here's how [The Register](#) puts it: "Think

The Register®
Biting the hand that feeds IT

Think of the kernel as God sitting on a cloud, looking down on Earth. It's there, and no normal being can see it, yet they can pray to it.

These KPTI patches move the kernel into a completely separate address space, so it's not just invisible to a running process, it's not even there at all. Really, this shouldn't be needed, but clearly there is a flaw in Intel's silicon that allows kernel access protections to be bypassed in some way.

THE VERGE
THURSDAY, JANUARY 4, 2018 | CHIPCALYPSE NOW

**FLAW IS RELATED TO KERNEL
MEMORY ACCESS**

The exact bug is related to the way that regular apps and programs can discover the contents of protect kernel memory areas. Kernels in operating systems have complete control over the entire system, and connect applications to the processor, memory, and other hardware inside a computer. There appears to be a flaw in Intel's processors that lets attackers bypass kernel access protections so that regular apps can read the contents of kernel memory. To protect against this, Linux programmers have been separating the kernel's memory away from user processes in what's being called "[Kernel Page Table Isolation](#)".

The New York Times

The software patches could slow the performance of affected machines by 20 to 30 percent, said Andres Freund, an independent software developer who has tested the new Linux code. The researchers who discovered the flaws voiced similar concerns



News headlines
on 1/2/2019
“Intel chip bug”

What's a kernel?

PCWorld
FROM IDG

The kernel inside a chip is basically an invisible process that facilitates the way apps and functions work on your computer. It has complete control over your operating system. Your PC needs to switch between user mode and kernel mode thousands of times a day, making sure instructions and data flow seamlessly and instantaneously. Here's how [The Register](#) puts it: "Think

The Register®
Biting the hand that feeds IT

Think of the kernel as God sitting on a cloud, looking down on Earth. It's there, and no normal being can see it, yet they can pray to it.

These KPTI patches move the kernel into a completely separate address space, so it's not just invisible to a running process, it's not even there at all. But the chips have a bug that makes it possible to peek at the kernel, which is a flaw in Intel's chips in some way.

Think of the kernel as God sitting on a cloud, looking down on Earth. It's there, and no normal being can see it, yet they can pray to it

The exact bug is related to the way that

regular apps and programs can discover the contents of protected kernel memory areas.

Kernels in operating systems have complete control over the entire system, and connect applications to the processor, memory, and

FLAW IS RELATED TO KERNEL MEMORY ACCESS

processors that lets them read the contents of kernel memory. To protect against this, Linux programmers have been separating the kernel's memory away from user processes in what's being called "[Kernel Page Table Isolation](#)."

The software patches could slow the

The kernel inside a chip is basically
way apps and functions work on yo

“Whenever a running program needs to do anything useful – such as write to a file or open a network connection – it has to temporarily hand control of the processor to the kernel to carry out the job. To make the transition from user mode to kernel mode and back to user mode as fast and efficient as possible, the kernel is present in all processes' virtual memory address spaces, although it is invisible to these programs. When the kernel is needed, the program makes a system call, the processor switches to kernel mode and enters the kernel. When it is done, the CPU is told to switch back to user mode, and reenter the process. While in user mode, the kernel's code and data remains out of sight but present in the process's page tables.

Think of the kernel as God sitting on a cloud, looking down on Earth. It's there, and no normal being can see it, yet they can pray to it.”

Inter chip bug

attackers bypass kernel access protections so that regular apps can read the contents of kernel memory. To protect against this, Linux programmers have been separating the kernel's memory away from user processes in what's being called “[Kernel Page Table Isolation](#).”

The New York Times

The software patches could slow the performance of affected machines by 20 to 30 percent, said Andres Freund, an independent software developer who has tested the new Linux code. The researchers who discovered the flaws voice

We cannot really understand the rest of the article because we need to know more about virtual memory, which is later this semester

It actually will be difficult to understand the whole bug and solution because it goes beyond course material, but we'll do what we can...

Part of the bug is due to speculative execution (show of hand?)

**News headlines
on 1/3/2018
“Intel chip bug”**

What's a kernel?

PCWorld
FROM IDG

The kernel inside a chip is basically an invisible process that facilitates the way apps and functions work on your computer. It has complete control over your operating system. Your PC needs to switch between user mode and kernel mode thousands of times a day, making sure instructions and data flow seamlessly and instantaneously. Here's how [The Register](#) puts it: "Think

The exact bug is related to the way that regular apps and programs can discover the contents of protect kernel memory areas. Kernels in operating systems have complete control over the entire system, and connect applications to the processor, memory, and other hardware inside a computer. There appears to be a flaw in Intel's processors that lets attackers bypass kernel access protections so that regular apps can read the contents of kernel memory. To protect against this, Linux programmers have been separating the kernel's memory away from user processes in what's being called "[Kernel Page Table Isolation](#)."

FLAW IS RELATED TO KERNEL MEMORY ACCESS

The New

The software performance of percent, said A software devel code. The rese voiced similar

THE MELTDOWN AND SPECTRE EXPLOITS USE "SPECULATIVE EXECUTION?" WHAT'S THAT?

YOU KNOW THE TROLLEY PROBLEM? WELL, FOR A WHILE NOW, CPUs HAVE BASICALLY BEEN SENDING TROLLEYS DOWN BOTH PATHS, QUANTUM-STYLE, WHILE AWAITING YOUR CHOICE. THEN THE UNNEEDED "PHANTOM" TROLLEY DISAPPEARS.

THAT SOUNDS BAD.

HONESTLY, I'VE BEEN ASSUMING WE WERE DOOMED EVER SINCE I LEARNED ABOUT ROWHAMMER.

WHAT'S THAT?

IF YOU TOGGLE A ROW OF MEMORY CELLS ON AND OFF REALLY FAST, YOU CAN USE ELECTRICAL INTERFERENCE TO FLIP NEARBY BITS AND—
DO WE JUST SUCK AT...COMPUTERS?

YUP. ESPECIALLY SHARED ONES.

THE PHANTOM TROLLEY ISN'T SUPPOSED TO TOUCH ANYONE. BUT IT TURNS OUT YOU CAN STILL USE IT TO DO STUFF.
AND IT CAN DRIVE THROUGH WALLS.



News
on
"Intel"



DEVELOPING ST
EXPERTS: ALMOST

SO YOU'RE SAYING THE CLOUD IS FULL OF PHANTOM TROLLEYS ARMED WITH HAMMERS.

...YES. THAT IS EXACTLY RIGHT.

OKAY. I'LL, UH...
INSTALL UPDATES?

GOOD IDEA.

tes the
ntrol over
e and
er
ds IT
link

ster
hat feeds IT

Earth. It's

e address
n there at
n Intel's
some way.

GE

INTEL

y, and
s that lets
ntents of
the
able

Limited Execution: Whole Story

- You write your user program, which calls a standard library function, which places a system call, e.g., `write()`
- The trap instruction is executed, the CPU sets the mode bit to kernel, figures out this is a “syscall” event, looks up the Trap Table, finds out in it the address of the handler for that event in the kernel code, and jumps to that code
- The handler code looks at the system call number passed to the trap instruction, looks up its table of syscall, finds the address of the code for that particular system call, and jumps to that code
- The syscall code is executed
- The syscall code returns to the system call handler, which sets the mode bit to “user” and returns to your program

One Problem down, one to go

- **Problem #1:** How do we prevent user programs from getting full control/access to the hardware?
 - Mode bit, trap instruction, syscalls, **SOLVED!**
- **Problem #2:** How do we kick a process out of the CPU and give the CPU to another process?
- Let's now deal with Problem #2
- The main idea is to **switch between processes**

It's all about Regaining Control

- Switching between processes should be simple
- The OS should just decide to stop one process and start another
- But it's not so easy: if a process is running on the CPU, by definition the OS is not running!
 - Meaning, Kernel code is not running
- So then how can the OS do anything???
- The question is: **How can the OS regain control of the CPU?**

The Cooperative Approach

- From the title, you already know it's not going to work ;)
- In the **cooperative approach**, you just assume processes are **nice** and willingly give up the CPU frequently
- For instance, each time a process places a syscall, then by definition Kernel code is running, and then the OS can take whatever action (like kicking the process off the CPU)
 - There could be a `yield()` syscall that does nothing and to just give up the CPU so that Kernel code gets a chance to run
 - We'll see that there is something like this for threads!
- The old MacOS 9 is a famous example that used this approach
 - Yes, on an old Mac, a `while(1) {}` program will lock up the machine and you'll need to reboot!
 - The easiest malware ever?
- How can we avoid this?
- Answer: **with a timer**

The Timer Interrupt

- To deal with non-cooperative processes, whenever the OS starts the fetch-decode-execute cycle of a process it sets a **timer**
- When the timer goes off, an interrupt is generated, so that the CPU will stop what it's doing and notify the OS
- The kernel has a handler for this interrupt
- This handler is the way in which the OS regains control
 - And can say “you've have enough CPU, let me kick you off the CPU and pick somebody else to run”
- Setting and enabling/disabling the timer are privileged instructions
 - Otherwise a user program could set the timer to 10 hours and hog the CPU as much as it wants
- **So now, we have the mechanism to regain control**
- Next up: **how to switch between processes?**

Context Switching

- The mechanism to kick a process off the CPU and give the CPU to another process is called a **context switch**:
 - Save the context of the running process to the PCB in RAM (which includes all register values)
 - Change its state from Running to Ready
 - Restore, from the PCB in RAM, the context of another Ready process (which includes all register values)
 - Make the state of this process Running
 - Restart its fetch-decode-execute cycle
- The context switch code is in assembly (Figure 6.4 in OSTEP)
- It should be as fast as possible because it is pure overhead
 - Nothing “useful to users” happens during a context switch
 - Nowadays it’s under 1 μ s
- Context switch is a **mechanism**, and deciding when to context switch (i.e., picking good timer values) and which Ready process to pick is a **policy**, which is called **scheduling**

Disclaimer about the Next Slide

- The next slide makes simplifying assumptions:
 - We assume a single CPU / single core system
 - We won't talk about threads, scheduling, and other concepts
 - We'll see those later, and we want to keep things simple for now
 - We assume that we have only **two** processes in memory
 - We also assume that they never go to the Waiting state (e.g., performing some I/O) and that they never go to the Terminated state (i.e., they run forever)
- Therefore with the above assumptions: At any given time, one process is in the Running state and the other is in the Ready state

Context Switching

Event	Time	Process #1	OS	Process #2
-	1	Running	-	Ready
Timer!	-	Running	-	Ready
-	2	Ready	(Context switch begins)	Ready
-	3	Ready	Save state in PCB #1	Ready
-	4	Ready	Save state in PCB #1	Ready
-	5	Ready	Restore state from PCB #2	Ready
-	6	Ready	Restore state from PCB #2	Ready
-	7	Ready	(Context switch ends)	Running
-	8	Ready	-	Running
-	9	Ready	-	Running
-
-	30	Ready	-	Running
Timer!	31	Ready	-	Running
-	32	Ready	(Context switch begins)	Ready
-

Context Switching

Event	Time	Process #1	OS	Process #2
-	1	Running	-	Ready
Timer!	-	Running	-	Ready
-	2	Ready	(Context switch begins)	Ready
-	3	Ready	Save state in PCB #1	Ready
-	4	Ready	Save state in PCB #1	Ready
-	5	Ready	Restore state from PCB #2	Ready
-	6	Ready	Restore state from PCB #2	Ready
-	7	Ready	(Context switch ends)	Running
-	8	Ready	-	Running
-	9	Ready	-	Running
-
-	30	Ready	-	Running
Timer!	31	Ready	-	Running
-	32	Ready	(Context switch begins)	Ready
-

Context switching overhead

Throwback to previous Module

- We've now said several time that the Kernel is an event handler
- Remember that in the previous module we showed “unrealistic pseudo-code” for the kernel?
- We've now talked about several new events, so let's add to what we had said back then...

The Kernel's (unrealistic) pseudo-

Event handling code

```
class Kernel {  
    [ . . . ]  
  
    method processEvent(Event event) {  
        switch (event.type) {  
            case MOUSE_CLICK:  
                Kernel.MouseManager.handleClick(event.mouse_position); break;  
            case NETWORK_COMMUNICATION:  
                Kernel.NetworkManager.handleConnection(event.network_interface); break;  
            case DIVISION_BY_ZERO:  
                Kernel.ProcessManager.terminateProgram("Can't divide by zero"); break;  
  
            case INVALID_MODE:  
                Kernel.ProcessManager.terminateProcess("Forbidden instruction"); break;  
  
            case TRAP_INSTRUCTION:  
                Kernel.doSystemCall(event.syscall_number); break;  
  
            case TIMER_INTERRUPT:  
                Kernel.ContextSwitchToAnotherProcess(); break;  
        }  
        return;  
    }  
}
```

Main Takeaways

- Letting programs use all CPU instructions directly is a BAD idea
 - User mode and kernel mode execution is a much better idea
 - Some (dangerous) instructions are protected
 - The Trap Table and the Trap (syscall) instruction
 - Programs keep switching mode
- Letting programs hog the CPU is a BAD idea
 - Setting a timer and having the kernel regaining control when the time goes off is a much better idea
 - Context switching

Conclusion

- OSTEP makes a good “the OS is baby proofing the CPU” analogy:
 - Make sure processes don’t do anything dangerous (privileged instructions they’re not allowed to execute)
 - But they can ask permission for an adult (the kernel) to do something dangerous on their behalf (via system calls)
 - Make sure they don’t hog shared toys (the CPU) for too long (via a timer interrupt)
- **Quiz next week on this Module!**
- Chapter 6 in OSTEP finishes by saying “now let’s talk about scheduling”
- But first we’ll talk about **threads** (in the next module)
- And then we’ll talk about **scheduling**...