Processes: OS Mechanisms

ICS332

Operating Systems

Henri Casanova (henric@hawaii.edu)

Direct Execution

- Figure 6.1 in OSTEP shows a simple 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?
  - This was a problem with Mac OS 9!!
- 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 implemented, 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 (modern) CPUs support (at least) two modes of execution:
  - The User Mode where protected instructions cannot be executed
  - The Kernel Mode where 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

User-Mode vs. Kernel-Mode

- In the Fetch-Decode-Execute cycle steps are added to the Decode stage:
  - Decode instruction
  - If the instruction is protected 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
- FYI:
  - There are actually multiple modes (multiple levels in the kernel, multiple levels in the CPU)
  - 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

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!

- This is exactly why the CPU has a special "system call" 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 (caused by a program's execution)

The Trap Table

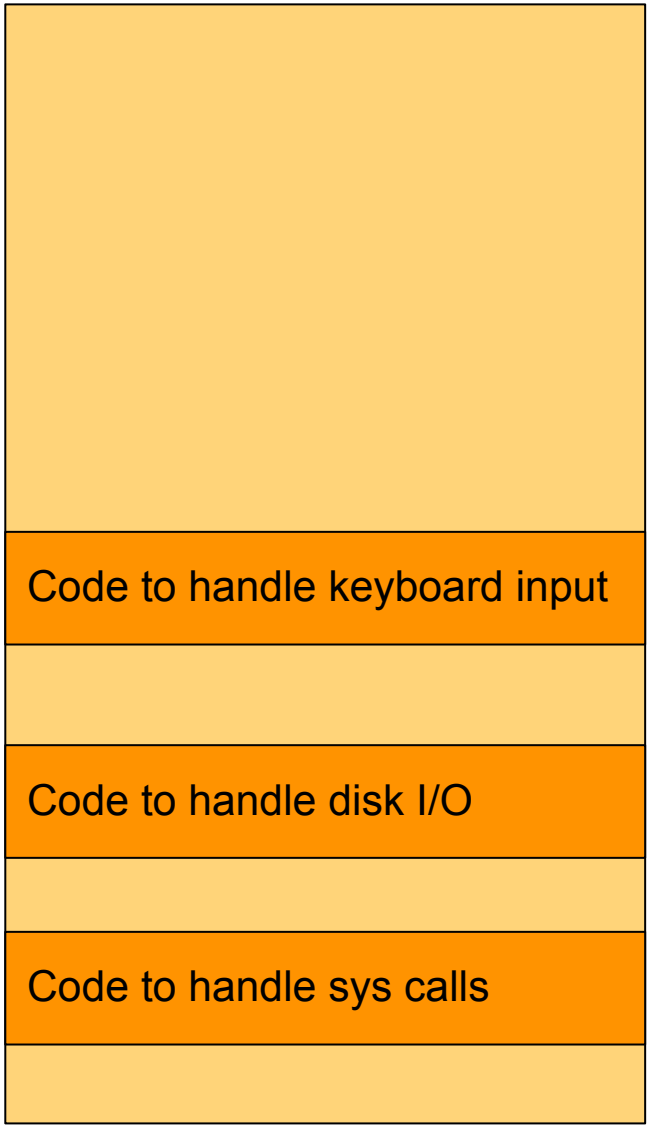
- At boot time, the OS initializes a Trap Table
  - On the x86 architecture, it's called the Interrupt Descriptor Table
- The Trap Table is stored in RAM, and the CPU has a register that points to it
- For each event type that the CPU could receive, this table indicates the address in the kernel of the 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

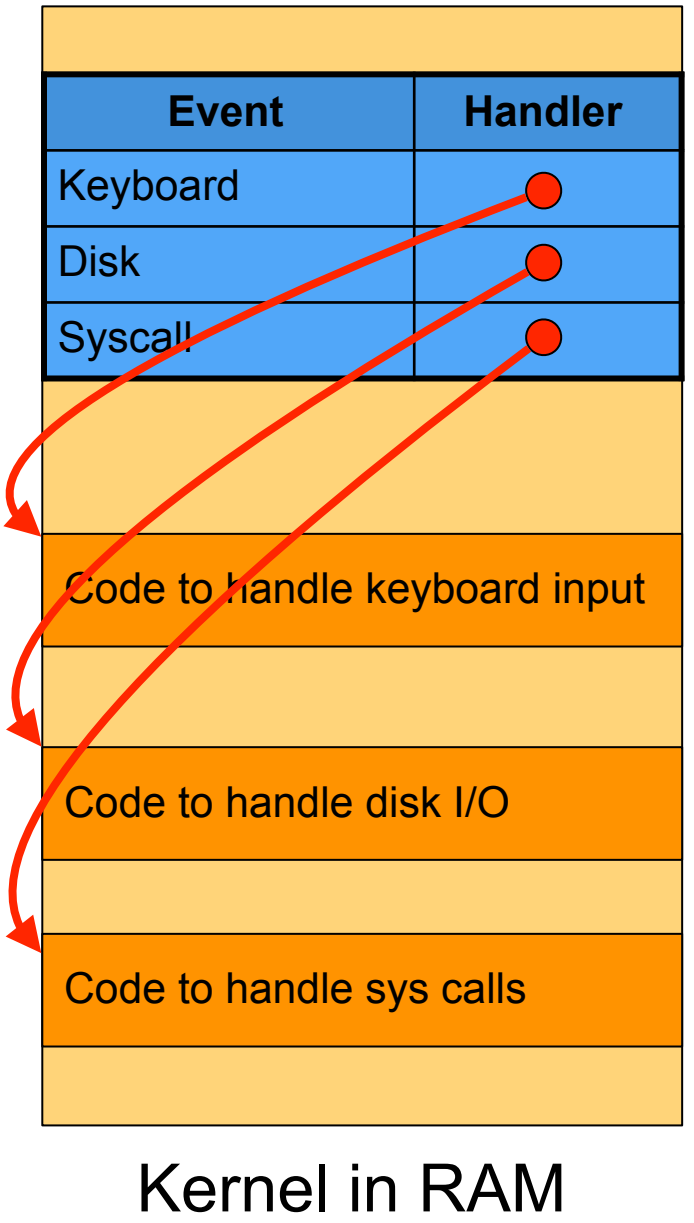
- At boot time, the kernel is loaded into RAM
- The kernel code includes handlers, i.e., pieces of code that should execute to answer particular events

- In this example, we consider
  - a “keyboard event” handler
  - a “disk I/O event” handler
  - a “syscall event” handler



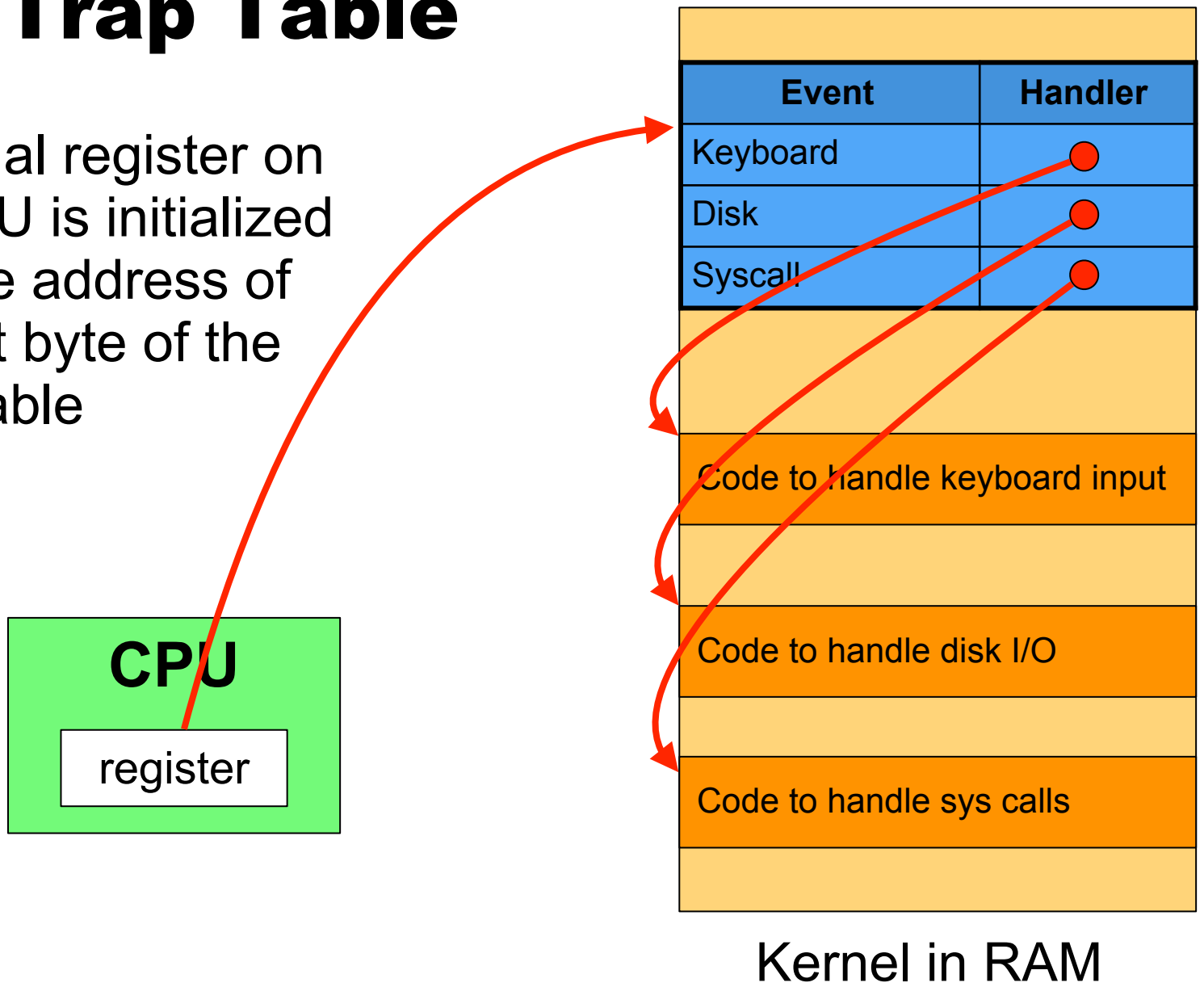
The Trap Table

- At boot time, in RAM a Trap Table is created as an array of consecutive bytes
- Each event type is set to the address of the first instruction of the corresponding kernel event 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



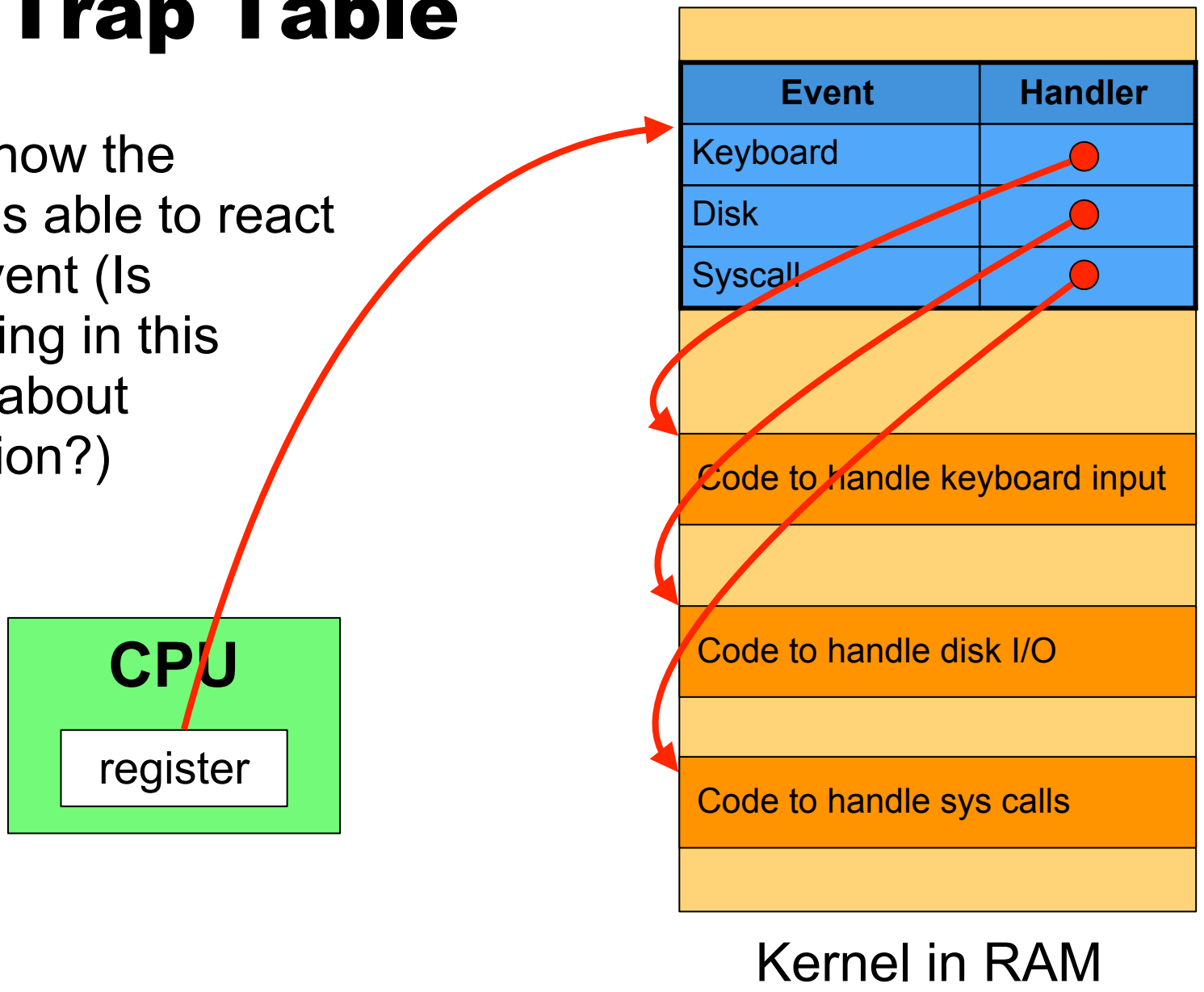
The Trap Table

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



The Trap Table

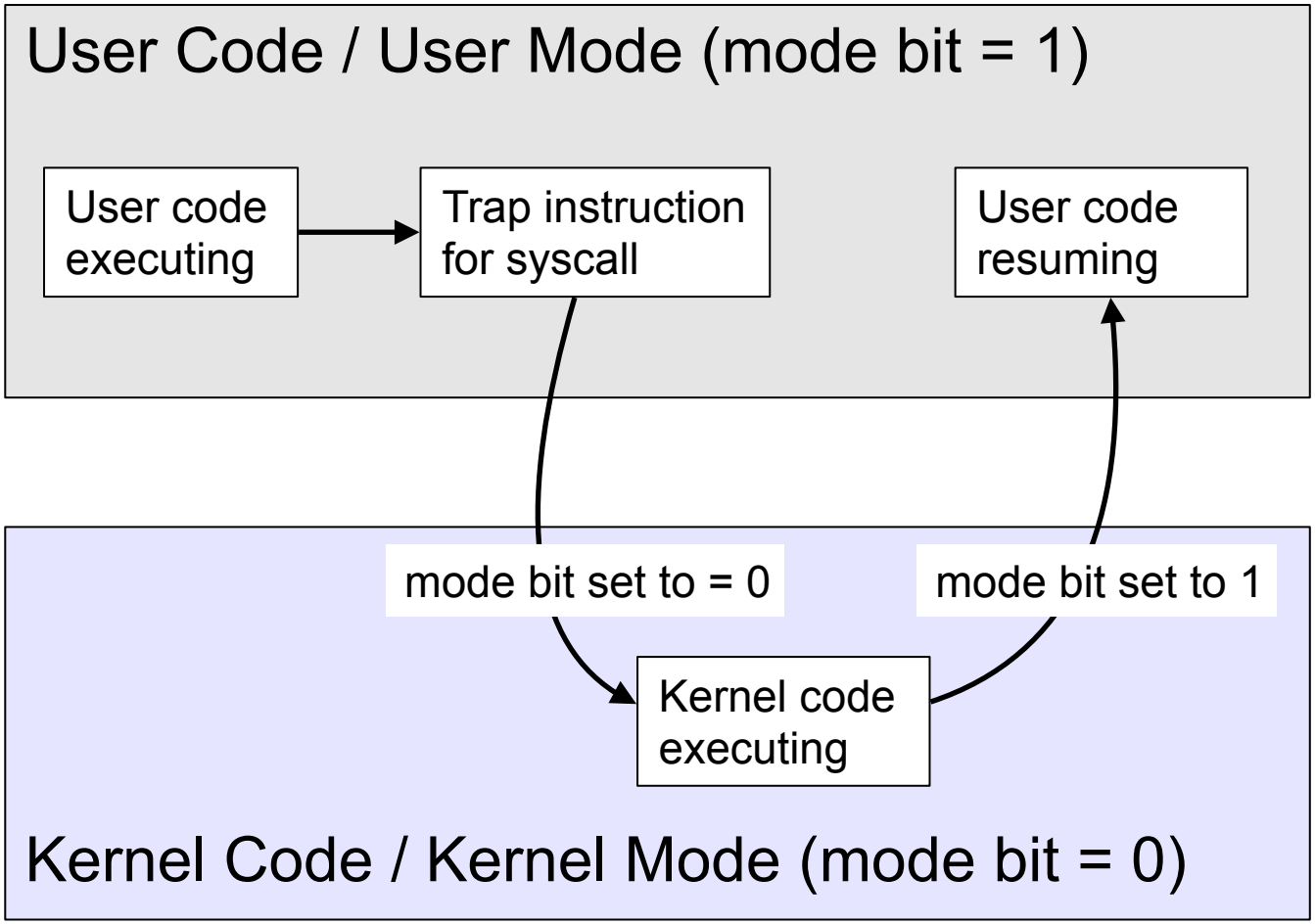
- This is how the Kernel is able to react to all event (Is everything in this course about indirection?)



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 if 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


User Code / User Mode (mode bit = 1)
User code executing
Trap instruction for syscall
User code resuming
mode bit set to = 0
mode bit set to 1
Kernel code executing
Kernel Code / Kernel Mode (mode bit = 0)

Limited Execution: Whole Story

- You write your user program, which calls a standard library function that 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

Limited Execution: Restricted Time

- Remember the two problems we identified at the beginning:
- **Problem #1:** How do we prevent user programs from getting full control/access to the hardware?
- **Problem #2:** How do we kick a process out of the CPU and give the CPU to another process?

- We've just dealt with Problem #1
  - Mode bit, trap instruction, sys calls

- 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 to just give up the CPU
  - 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, a trap is generated, so that the CPU will stop what it's doing and notify the OS
- The kernel has a handler for this trap (pointed to by an entry in the Trap Table, as we've seen)
- 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
- **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 (i.e., all register values)
  - Change its state from Running to Ready
  - Restore, from the PCB in RAM, the context of another Ready process (i.e., 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 1µs
- Context switch is a mechanism, and deciding when to context switch (i.e., picking 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 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 to go 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	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	-	Ready
-	5	Ready	Restore state from PCB #2	Ready
-	6	Ready	-	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	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	-	Ready
-	5	Ready	Restore state from PCB #2	Ready
-	6	Ready	-	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

Conclusion

- OSTEP makes a good “baby proofing” analogy
- The idea is that you can think of the mechanisms we’ve talked about as the OS “baby proofing” the CPU
  - Make sure processes don’t do anything dangerous (privileged instructions they’re not allow 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) too long (via a timer interrupt)
- Chapter 6 in OSTEP finishes by saying ”now let’s talk about scheduling”
- But before we get there, let’s talk about IPCs (in this module)
- And then we’ll talk about threads (in the next module)
- And then we’ll talk about scheduling…