CSE 451: Operating Systems

Spring 2021

Module 2

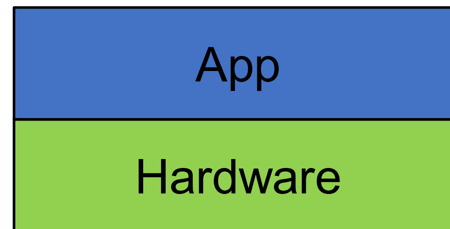
Architectural Support for Operating Systems

John Zahorjan

Lecture Questions

- What is the basic control flow of the system?
- Why do transitions from user code to the OS take place?
- Since they run on the same CPU, why can't applications do everything the OS can do?
- What happens on a transition from user code into the OS?
- On a transition from the OS to user code?
- What mechanisms does the hardware provide to help the OS keep control of the system?
- When the OS is running, what stack is it using (in xk)?
- How does xk use the segmented memory system provided by x86_64?
- How is memory protected?
- How are IO devices protected?

Low-level architecture affects the OS dramatically



Who's making sure the app behaves?

Who should get to define what "behaves" means?

*(Hardware provides **mechanism** and OS provides **policy.)***

Low-level architecture affects the OS dramatically

- The operating system supports sharing of hardware and protection of hardware
  - multiple applications can run concurrently, sharing resources
  - a buggy or malicious application can’t violate other applications or the system
- Those are high level goals
  - There are many mechanisms that can be used to achieve them
- The architecture determines which approaches are viable (reasonably efficient, or even possible)
  - includes instruction set (synchronization, I/O, …)
  - also hardware components like MMU or DMA controllers

Architectural features affecting OS's

- These hardware features were built primarily to support OS's:
  - timer (clock) operation
  - synchronization instructions (e.g., atomic test-and-set)
  - memory protection
  - I/O control operations
  - interrupts and exceptions
  - protected modes of execution (kernel vs. user)
  - privileged instructions
  - system calls (and software interrupts)
  - virtualization architectures

The OS Needs To Be Special

- Only the OS should be able to:
  - directly access I/O devices (disks, network cards)
    - why?
  - manipulate memory state management
    - page table pointers, TLB loads, etc.
    - why?
  - manipulate special 'mode bits'
    - interrupt priority level
    - why?
- But users can put any bit strings in memory they want
  - so they can execute any instructions that the OS uses to do those things
- So how can this work?

So How Can This Work?

- Some hardware resource must be available to the OS but not to applications
  - Could be instructions
  - Could be access to special registers and/or addresses
- Turns out it's both!

- The CPU hardware provides privileged instructions that "only the OS can execute"

- Some resources can be modified only by instructions that are privileged
  - E.g., information related to address translation

- The OS can use them to establish an execution environment that limits access (e.g., to memory)
  - The application cannot remove the restrictions because it must execute privileged instructions to do so

“... only the OS can execute”

- This is a policy goal
  - What mechanism(s) can be used to achieve it?
- Q1: How can the CPU hardware tell when the OS is running?
  - A1: It can’t. The OS is a concept, the hardware is a state machine.
- Q2: What should happen when something that isn’t the OS tries to execute a privileged instruction?
  - (Poor) A2: As a mechanism, the CPU could just consider it to be a NOP, say.
  - (Good) A2: Gee, what happens is really a policy decision. The OS should make it, not the hardware.

Policy/Mechanism

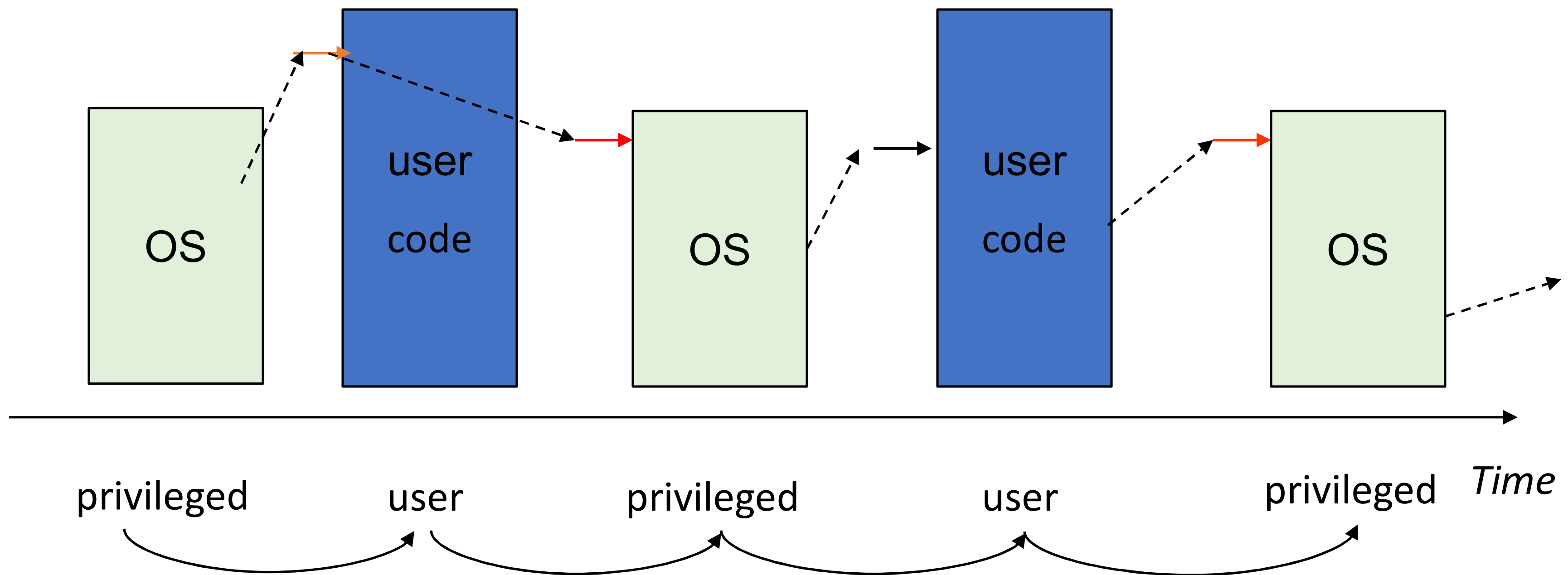
- Here's my lame analogy
  - Back when you could go out to a restaurant for dinner
    - You set the policy: what set of things to order, maybe what order they arrived in
    - The restaurant implemented the mechanism: a menu of things you could order, a stockpile of ingredients, pots/pans/stove, a chef, waitstaff
- You can order a bottle of wine and pancakes if you want
  - That's a positive
  - The restaurant is flexible enough to be attractive to all sorts of customers, even some who seem crazy
- What is the equivalent of pots and pans and stoves and chefs in the CPU?
  - What are the CPU mechanisms that allow the OS to realize its policies?

Mechanism: How Does CPU Decide Whether or Not to Execute a Privileged Instruction

- Privilege Level: There is at least one bit of data somewhere accessible to the CPU (e.g., in a special register)
  - When the bit == 1 we say we're executing in privileged mode, and the CPU is willing to execute privileged instructions
  - When the bit == 0 we say we're executing in unprivileged (or user) mode, and the CPU is unwilling to execute privileged instructions

- Exception Mechanism: What happens if the CPU fetches a privileged instruction while in unprivileged mode?
  - It invokes the OS, so that it can decide what to do
  - We'll see exactly how in just a bit

Making CPU Privilege Mode == Running OS



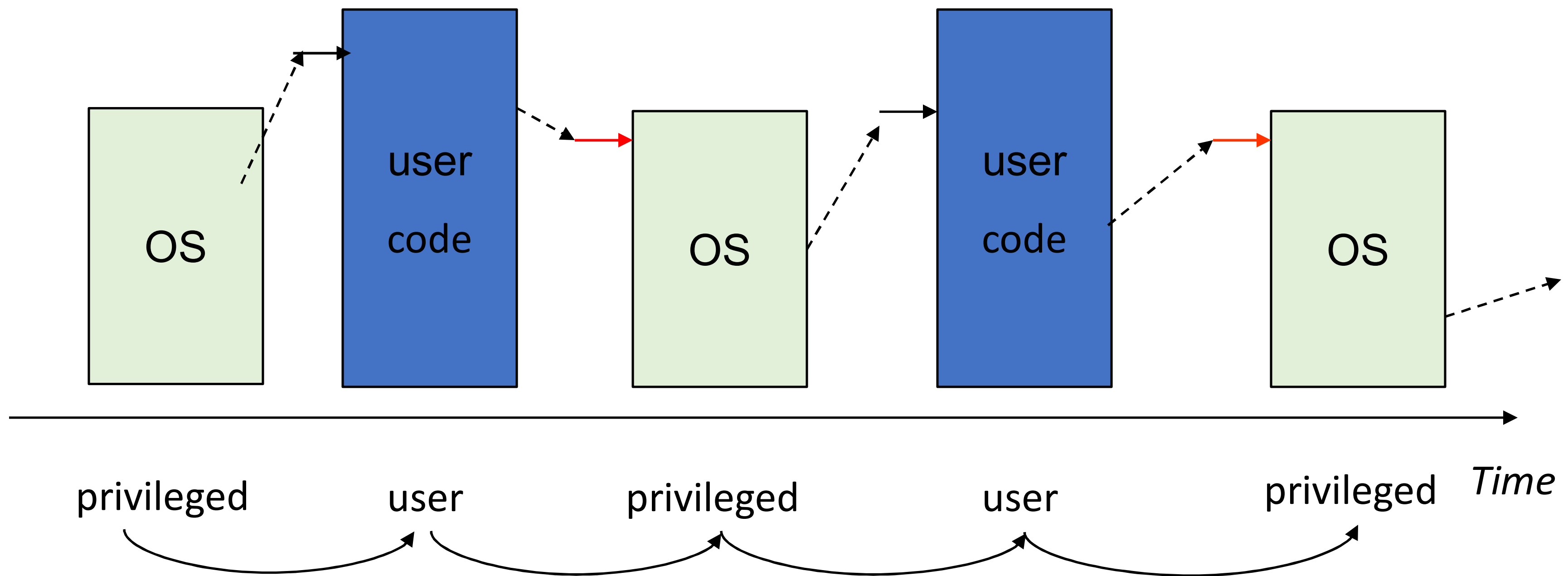
- OS runs first (boot)
  - CPU starts in privileged mode
- OS sets privilege mode to user before handing CPU over to user code
  - So far so good…
- Eventually we need to run the OS again…

Entering the OS: system calls

- Sometimes user code wants the OS to do something for it
  - E.g., read/write files, send/receive network data, start another program running, etc.
- In the abstract, it wants to do a procedure call, as though the OS were a library
  - Establish some arguments to be passed to the OS
  - Let the OS run for a bit and produce return values (and/or side effects)
  - Return to the statement following the call to the OS procedure
  - Find the return values produced by the OS

- CPU is at user privilege while executing user code
- CPU needs to be in privileged mode while executing the OS
- How can the user cause the CPU to transition from unprivileged to privileged?

Making CPU Privilege Mode == Running OS



- Each transition from user level code to OS code transfers control to the same place (the orange arrow)
- The user level code passes as an argument a "syscall number" identifying which operation it is asking for (as well as any further arguments needed for that system call)
- The OS runs at privileged level starting with lines of code it decided upon
- User level code can't both elevate CPU privilege level **and** define what instruction to execute next

System Calls

- User programs must cause execution of an OS
  - OS defines a set of system calls
  - App code places a bunch of arguments to the call somewhere the OS can a find them
    - e.g., on the stack or in registers
  - One of the arguments "names" the system call that is being requested
    - usually a syscall number
    - *when app code wants to call a subroutine in that app, how does it "name" which one it wants?*
  - App executes a syscall instruction
    - CPU privilege level is set to privileged
    - PC is set to the contents of a privileged register
    - during boot the OS set that register to point at the OS "trap handler" method
    - user code can't mess with it because modifying that register is a privileged operation

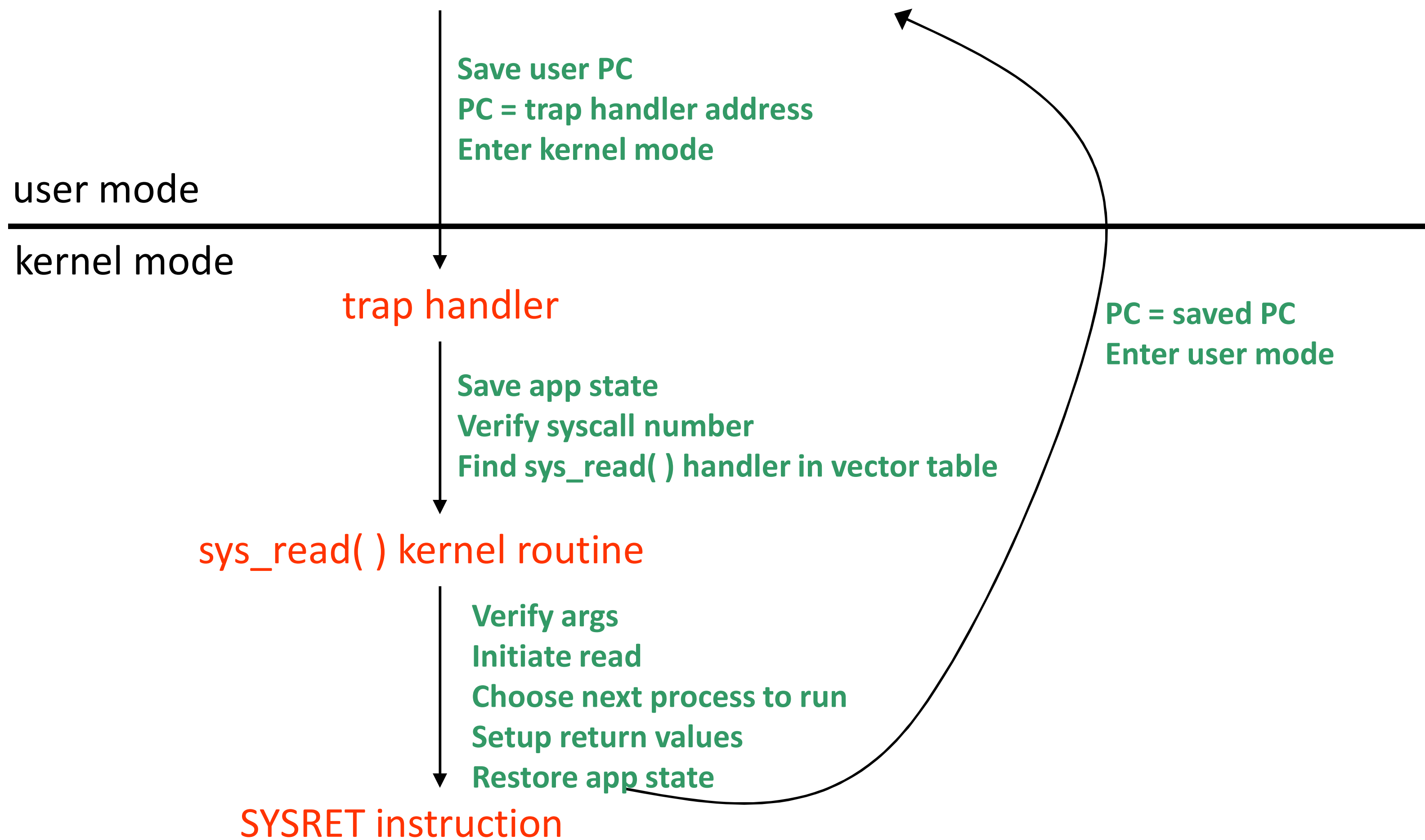
syscall/sysret instructions

- The syscall instruction atomically:
  - Sets the execution mode to privileged
  - Sets the PC to a handler address (that was established by the OS during boot)
  - Saves the current (user) PC
    - Why?
- The sysret instruction atomically:
  - Restores the previously saved user PC
  - Sets the execution mode to unprivileged

“Protected procedure call”

- Caller puts arguments in a place callee expects (registers or stack)
- Caller causes jump to OS by executing syscall instruction
  - **The OS determines what address to start executing at, not the caller**
  - One of the passed args is a syscall number, indicating which OS function to invoke
  - Some hardware state that can’t be saved if left to software (e.g., the user level PC of the instruction that follows the syscall instruction) is “pushed on the stack”
    - Which stack?
- Callee (OS) saves caller’s state (registers, other control state) so it can use the CPU
- OS function code runs
  - **OS must verify caller’s arguments** (e.g., pointers)
- OS (mostly) restores caller’s state
- OS returns by executing sysret instruction
  - Automatically sets execution mode to user and PC to return address previously saved on the stack

A kernel crossing illustrated



One More Issue: Stacks

- The kernel code is structured like user level code
  - It needs a stack
- The transition from user level to kernel level must involve a change in which stack is in use
  - A stack is just a region of memory used as the stack, so there can be any number of them in memory
- On some processors this transition is done in software
- On the x86 family it is done in hardware as part of the syscall instruction
  - On syscall, the user-level SP is saved to a temporary, the SP is set to an address in a privileged register previously initialized by the OS, and then the temporary is pushed onto that stack (along with the user-level PC)
  - On sysret, more or less the reverse is done

- Why can't the OS just use the user-level stack?

x86 Interrupt Stack (Mechanism)

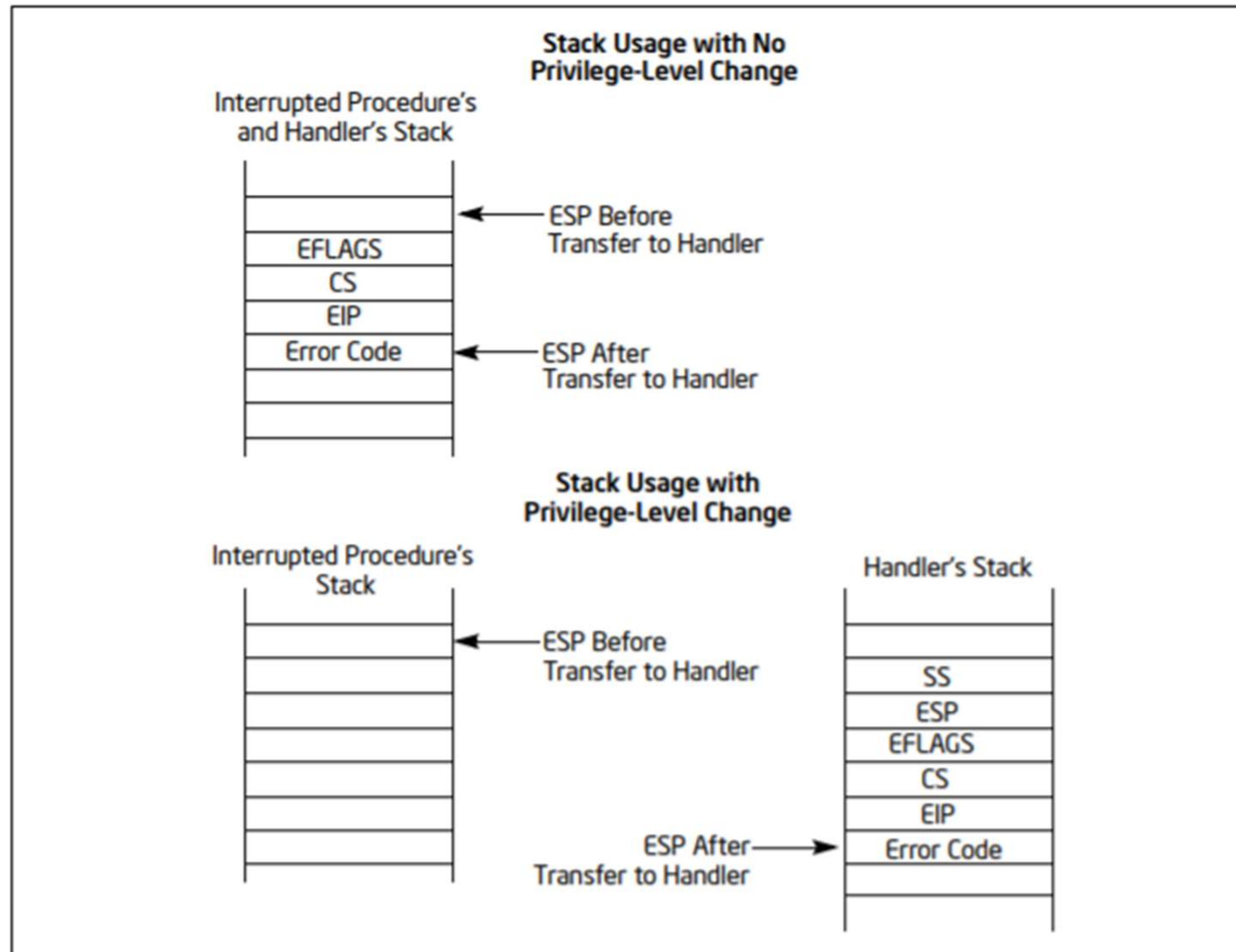


Figure 6-5. Stack Usage on Transfers to Interrupt and Exception Handling Routines

System call issues

- What would be wrong if a syscall worked like a regular subroutine call, with the caller specifying the next PC?
- What would happen if kernel didn’t save state?
- Why must the kernel verify arguments?
- How can you reference kernel objects as arguments to or results from system calls?
  - What does that question mean?!

Exception Handling and Protection

- *All* entries to the OS occur via the mechanism just described
  - Acquiring privileged mode and branching to the trap handler are inseparable
- Terminology:
  - **Interrupt**: asynchronous event; caused by an external device
  - **Exception**: synchronous event; unexpected problem with instruction
  - **Trap**: synchronous event; intended transition to OS due to an instruction
- Privileged instructions and resources are the basis for most everything: memory protection, protected I/O, limiting user resource consumption, …

Some Details

- The architecture defines the trap handling mechanism
- Exactly what's done in hardware and what in software differs across architectures
  - So, what I described isn't "the way it's done" it's more the idea of the way it's done
- For example, x86 trap handling doesn't have a register that gives the single entry point into the OS, it has something more complicated
  - You can think of it as a privileged register that points to an array of entry addresses
  - On trap/exception/interrupt, the hardware uses the trap/exception/hardware type (a number, called the "vector") to index the table and set the PC
- In general, x86 does a lot of complicated things in hardware, and RISC-like processors try to push as much as possible to software

x86 Interrupt/Trap Handling: Interrupt vectors

Vector	Mne-monic	Description	Type	Error Code	Source
0	#DE	Divide Error	Fault	No	DIV and IDIV instructions.
1	#DB	Debug Exception	Fault/ Trap	No	Instruction, data, and I/O breakpoints; single-step; and others.
2	—	NMI Interrupt	Interrupt	No	Nonmaskable external interrupt.
3	#BP	Breakpoint	Trap	No	INT 3 instruction.
4	#OF	Overflow	Trap	No	INTO instruction.
5	#BR	BOUND Range Exceeded	Fault	No	BOUND instruction.
6	#UD	Invalid Opcode (Undefined Opcode)	Fault	No	UD2 instruction or reserved opcode.[1]
7	#NM	Device Not Available (No Math Coprocessor)	Fault	No	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Abort	Yes (zero)	Any instruction that can generate an exception, an NMI, or an INTR.
9		Coprocessor Segment Overrun (reserved)	Fault	No	Floating-point instruction.[2]
10	#TS	Invalid TSS	Fault	Yes	Task switch or TSS access.
11	#NP	Segment Not Present	Fault	Yes	Loading segment registers or accessing system segments.
12	#SS	Stack-Segment Fault	Fault	Yes	Stack operations and SS register loads.
13	#GP	General Protection	Fault	Yes	Any memory reference and other protection checks.
14	#PF	Page Fault	Fault	Yes	Any memory reference.

x86 Interrupt/Trap Handling: Overview

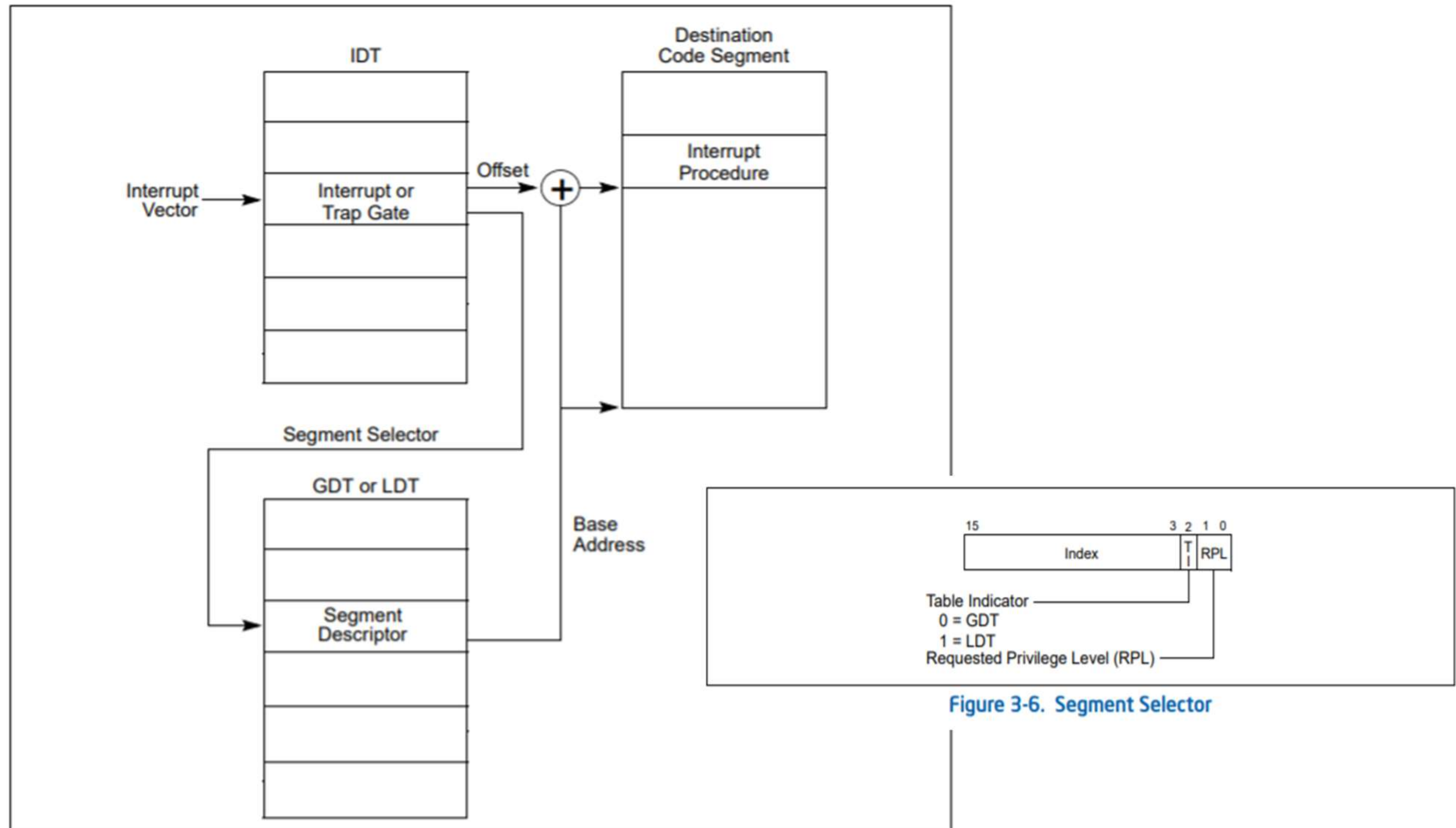


Figure 3-6. Segment Selector

Figure 6-3. Interrupt Procedure Call

x86 Interrupt/Trap Handling: Finding the IDT

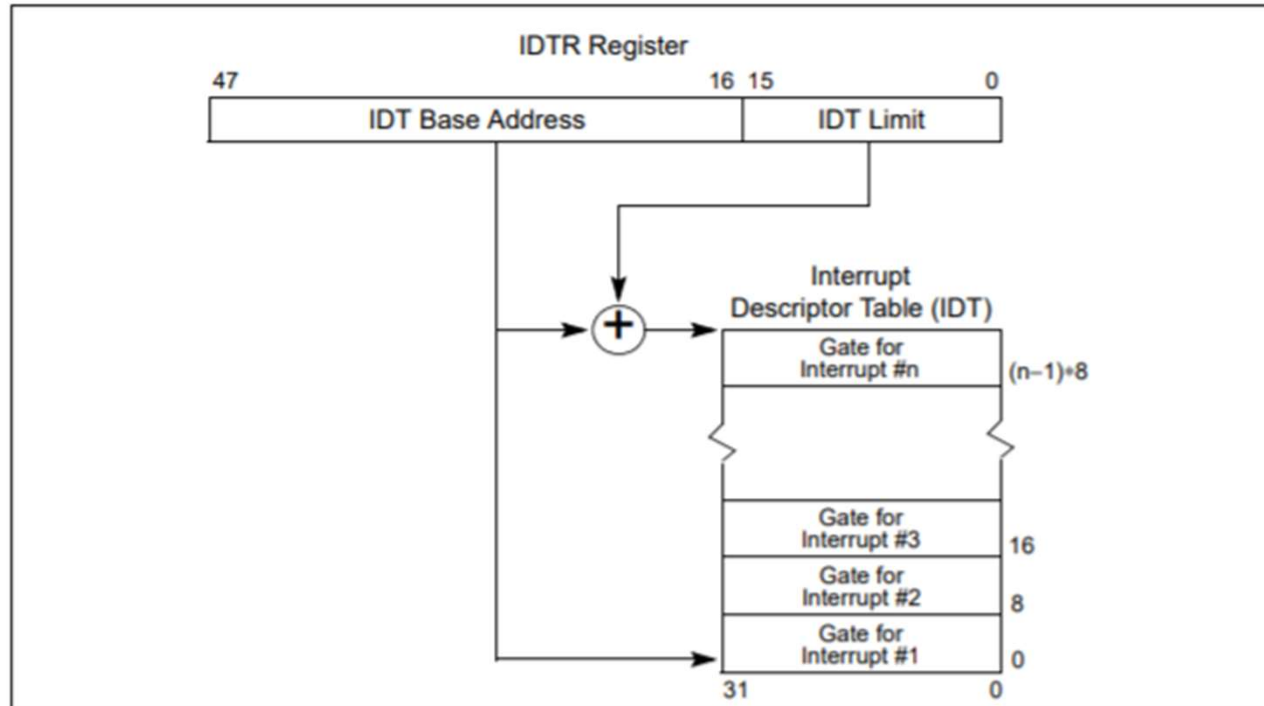


Figure 6-1. Relationship of the IDTR and IDT

x86 Interrupt/Trap Handling: IDT entries

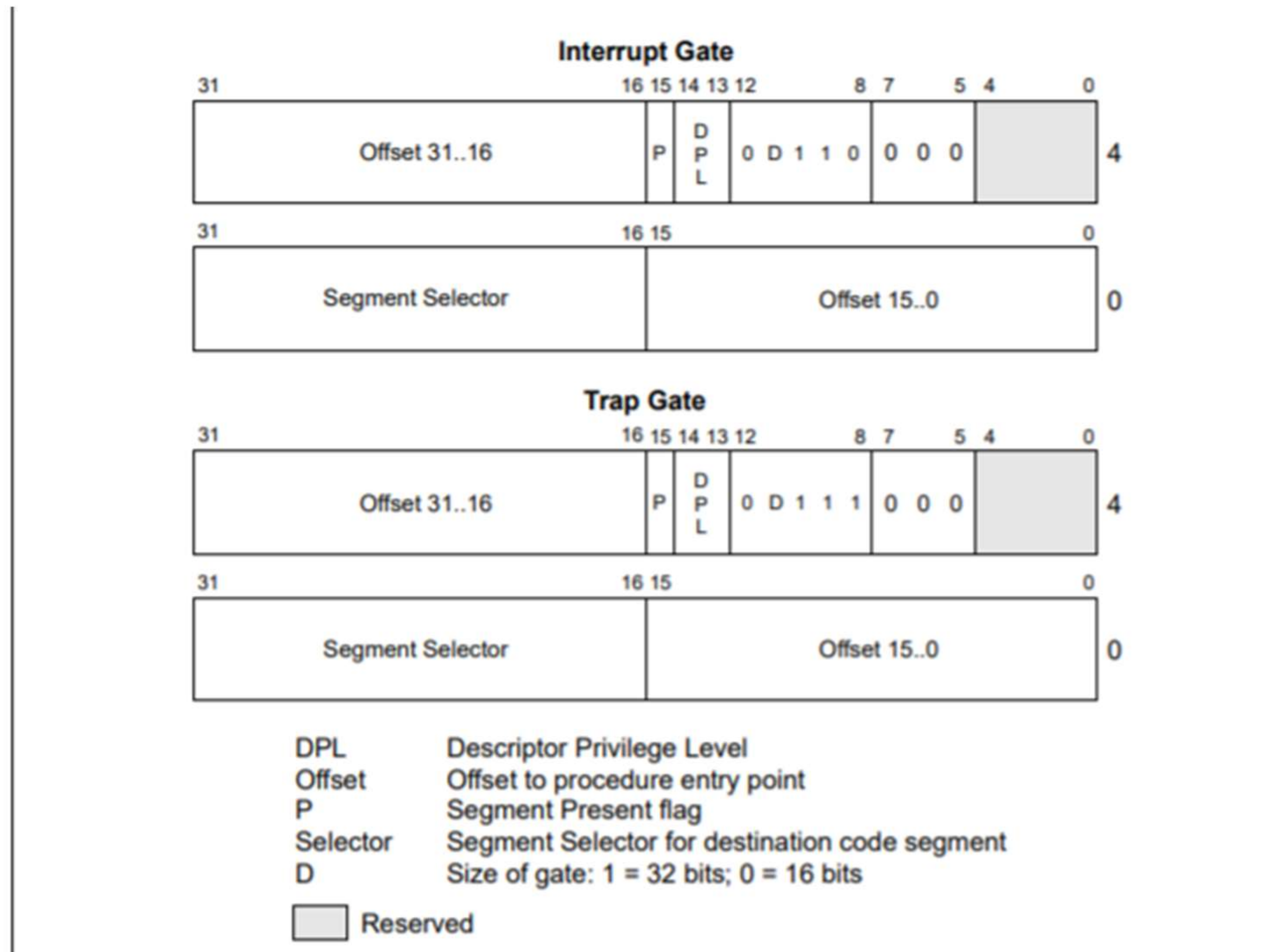


Figure 6-2. IDT Gate Descriptors

x86 Interrupt/Trap Handling: Segment Descriptors

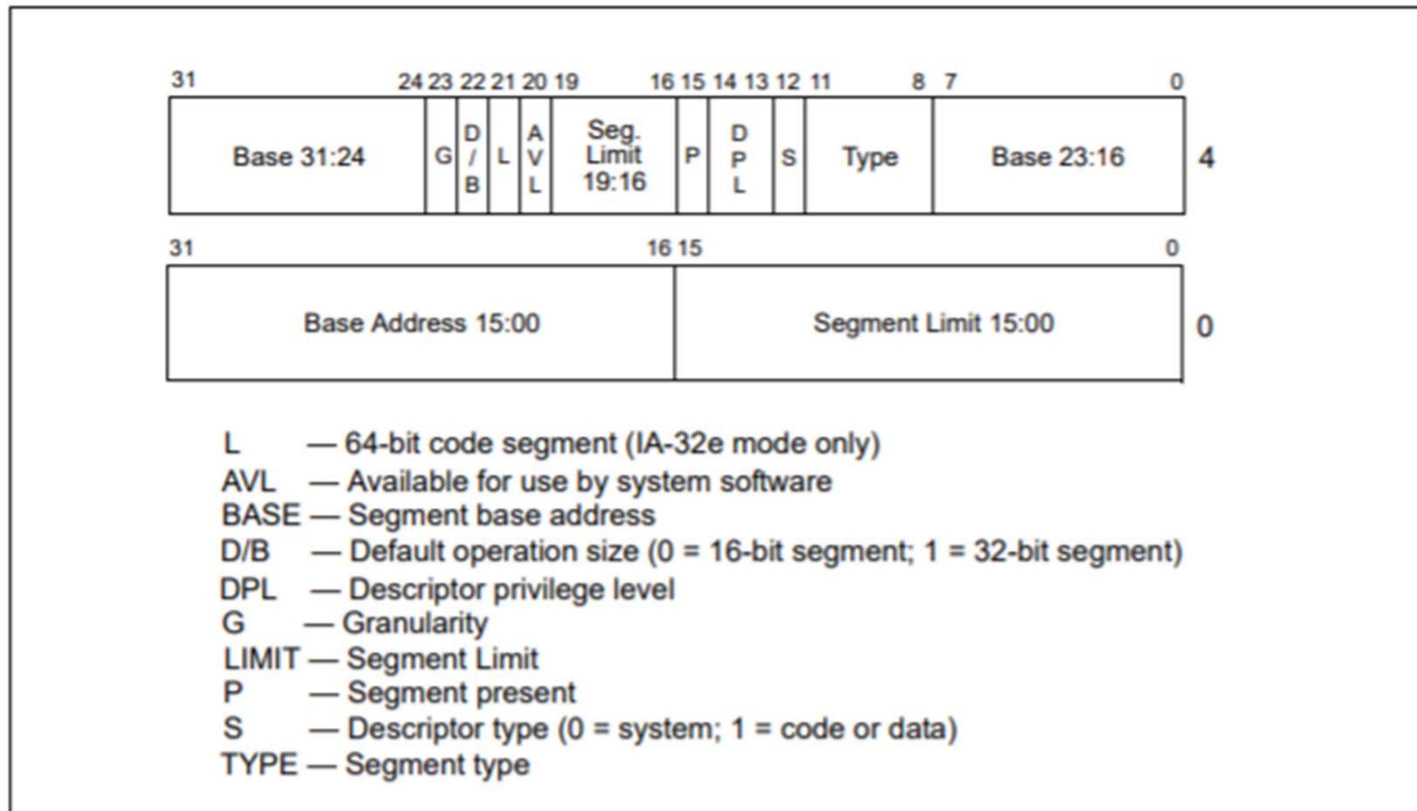


Figure 3-8. Segment Descriptor

x86 Interrupt/Trap Handling: Stacks

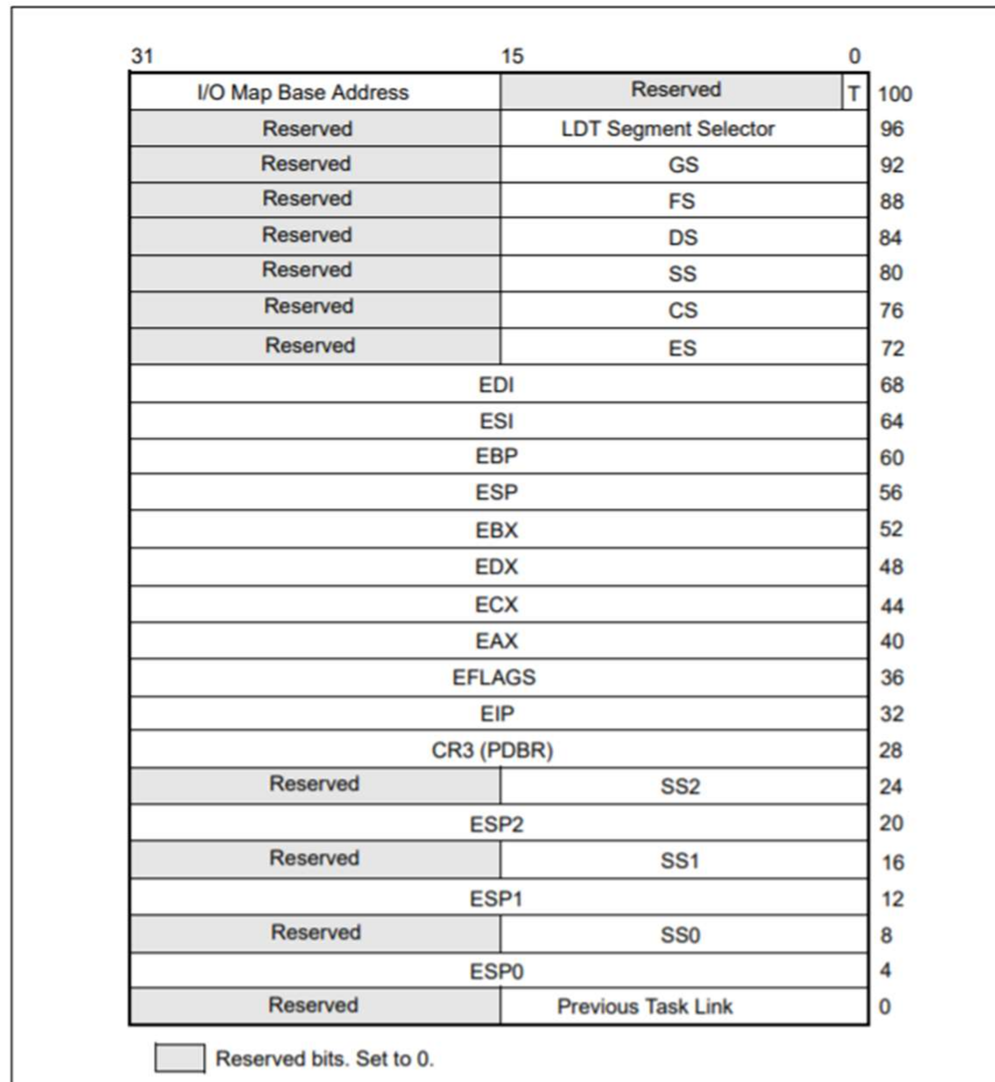


Figure 7-2. 32-Bit Task-State Segment (TSS)

Exception Summary

- Basically all protection provided by the OS relies in some way on the exception mechanism
  - Performance requires application code to run directly on the CPU and memory hardware
  - That leaves it to the hardware to intercept unsafe/illegal activity
  - Separation of policy and mechanism means that when the hardware notices something wrong, it should invoke OS code to decide what to do in response
- The same mechanism is used whenever the hardware wants to "upcall" to the OS, even when nothing has gone wrong
  - Interrupts: some IO device wants attention
  - Traps: user level code wants to do a syscall

Exception Generalization

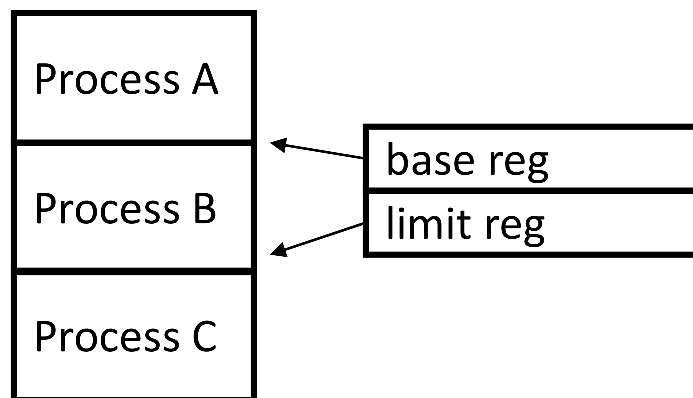
- To think about:
  - Let’s move up a level, from hw/sw to os/user-level code.

  - Might there be situations in which it makes sense for the OS to provide mechanism and the application to provide policy
    - The OS mechanism would be the detection of some event
    - The application policy would be the steps it wants to take in response to that event
    - What might be an example of an OS-level “event”?

  - What would the mechanism to “upcall” from OS to app need to do?
    - Invoke a handler method in the app, implying
    - Finding a thread of execution to execute the handler (a stack)
    - How would it know the location of the handler?

Issue: Memory protection

- OS must protect user programs from each other
  - malice, bugs
- OS must also protect itself from user programs
  - integrity and security
  - what about protecting user programs from OS?
- Simplest scheme: base and limit registers
  - (Hey, segments!)
  - are these protected?



base and limit registers are loaded by OS before starting program

More sophisticated memory protection

- Paging, segmentation, virtual memory
  - page tables, page table pointers
  - translation lookaside buffers (TLBs)
  - page fault handling
  - isolation via naming
- Coming later in the course
  - also coming earlier in your course sequence!
  - so we won’t spend much time on these in 451

Issue: I/O control

- Issues:
  - how does the OS start an I/O?
    - special I/O instructions
    - memory-mapped I/O
      - special addresses, not special instructions
  - how does the OS notice when something interesting has happened (e.g., an I/O has finished or a network packet has arrived)?
    - polling
    - Interrupts
  - how does the OS exchange data with an I/O device?
    - Programmed I/O (PIO)
    - Direct Memory Access (DMA)

Asynchronous I/O

- what does the “asynchronous” part mean?
  - device performs an operation asynchronously to CPU

- Interrupts are the basis for asynchronous I/O
  - device sends an interrupt signal on bus when done
  - in memory, a vector table contains list of addresses of kernel routines to handle various interrupt types
    - who populates the vector table, and when?
  - CPU switches to address indicated by vector index specified by interrupt signal and the stack registered for that handler

- What’s the advantage of asynchronous I/O?
  - Is this an advantage only to the OS? Is there a reason for an individual app to want to use asynchronous I/O? What would be required to allow it to do so?

Issue: Taking the CPU Back from Apps

- Q: How can the OS prevent runaway user programs from hogging the CPU (infinite loops?)
- A: Use a hardware timer that generates a periodic interrupt
  - before it transfers to a user program, the OS loads the timer with a time to interrupt
    - "quantum" – how big should it be set?
  - when timer fires, an interrupt transfers control back to OS
    - at which point OS must decide which program to schedule next
    - very interesting policy question: we'll dedicate a class to it
- Should access to the timer be privileged?
  - for reading or for writing?

Issue: Synchronization

- Interrupts cause a wrinkle:
  - may occur any time, causing code to execute that interferes with code that was interrupted
  - OS must be able to synchronize concurrent processes
- Synchronization:
  - guarantee that short instruction sequences (e.g., read-modify-write) execute atomically
  - one method: turn off interrupts before the sequence, execute it, then re-enable interrupts
    - architecture must support disabling interrupts
      - Privileged???
    - does this method work?
  - another method: have special complex atomic instructions
    - read-modify-write
    - test-and-set
    - load-linked store-conditional

“Concurrent programming”

- Management of concurrency and asynchronous events is an important difference between “systems programming” and “traditional application programming”
  - “event-driven” application programming is a middle ground
  - And in a multi-core world, more and more apps have internal concurrency and more and more languages acknowledge and support it
  - And in a networked world more and more apps engage in asynchronous I/O (network communication)
- Arises from the architecture
  - Can be sugar-coated, but cannot be totally abstracted away
- Serious intellectual challenge
  - Unlike vulnerabilities due to buffer overruns, which are just sloppy programming

Architectures are still evolving

- New features are still being introduced to meet modern demands
  - Support for virtual machine monitors
  - Hardware transaction support (to simplify parallel programming)
  - Support for security (encryption, trusted modes)
  - Increasingly sophisticated video / graphics
  - Other stuff that hasn't been invented yet…
- In current technology transistors are free – CPU makers are looking for new ways to use transistors to make their chips more desirable

Some questions

- Why wouldn’t you want a user program to be able to access an I/O device (e.g., the disk) directly?
  - Why would you?!
- OK, so what keeps this from happening? What prevents user programs from directly accessing the disk?
- How then does a user program cause disk I/O to occur?

Some questions

- What prevents a user program from writing the memory of another user program?
  - Why might you want to allow it to?
- What prevents a user program from writing the memory of the operating system?
- What prevents a user program from over-writing its own instructions?
  - Why do you want to prevent that?
  - Why do you want to allow it?!
- Is there any reason to support preventing an application from over-writing any of its own data?
  - Is there a use for read-only data memory?
- What prevents a user program from doing a denial of service attack on the CPU simply by going into an infinite loop?

Lecture Question Answers

- What is the basic control flow of the system?
  - The CPU switches between running OS code and application code
- Why do transitions from user code to the OS take place?
  - Interrupts – some IO device (typically) needs attention
  - Exceptions – the CPU has detected something problematic in completing execution of an instruction
  - Trap – the purpose of the instruction being executed is to transition into the OS (syscall)
- Since they run on the same CPU, why can’t applications do everything the OS can do?
  - The hardware has two or more privilege levels
  - Some instructions are privileged – require a sufficiently high privilege level – for the CPU to be willing to execute them
- What happens on a transition from user code into the OS?
  - Some registers that execution of the OS is about wipe out are saved by hardware, e.g., the user code PC at which the switch is occurring
  - The PC is set to the address previously set by the OS. The address is a safe entry point into the OS.
  - The privilege level of the CPU is elevated so that it can execute privileged instructions
  - The hw or sw saves all registers so that execution of the user code can eventually be resumed
- On a transition from the OS to user code?
  - The previously saved registers (including the PC) are restored on t he CPU
  - The privilege level is lowered to user level

Lecture Question Answers

- What mechanisms does the hardware provide to help the OS keep control of the system?
  - CPU privilege level + privileged instructions
  - memory access limitations, e.g., virtual memory
  - the exception mechanism – detecting when something that needs OS attention has happened and causing a switch into the OS
- When the OS is running, what stack is it using (in xk)?
  - A per-process kernel stack
- How does xk use the segmented memory system provided by x86_64?
  - It basically renders it moot by mapping every hardware segment to the full linear address space (i.e., base 0 and length 4GB)

Lecture Question Answers

- How is memory protected?
  - On modern system, virtual memory
  - The OS sets a privileged CPU register to point to address mapping structures for the address space the CPU should be using (e.g., when it dispatches a user process)
- How are IO devices protected?
  - Depending on the architecture or even system, it could be privileged instructions are required to communicate with the IO devices, or it could be that protected addresses must be read/written to communicate with them