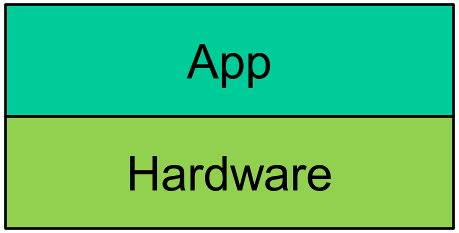
CSE 451: Operating Systems
Autumn 2019

Module 2
Architectural Support for Operating Systems

John Zahorjan

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

Privileged instructions

- 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 the same instructions that the OS does
- So how can this work?
  - some instructions must be “restricted to the OS”
  - known as privileged instructions

OS protection

- So how does the processor know whether to allow execution of a privileged instruction?
  - the architecture must support at least two “privilege levels”: kernel and user
    - x86 supports 4 privilege levels
  - current level is given by status bits in a protected processor register
    - user programs execute in user mode (3, in xk)
    - OS executes in kernel (privileged) mode (0, in xk)

- The hardware assures that privileged instructions can be executed only when the core is at kernel privilege
  - what happens if code running in user mode attempts to execute a privileged instruction?

Crossing protection boundaries

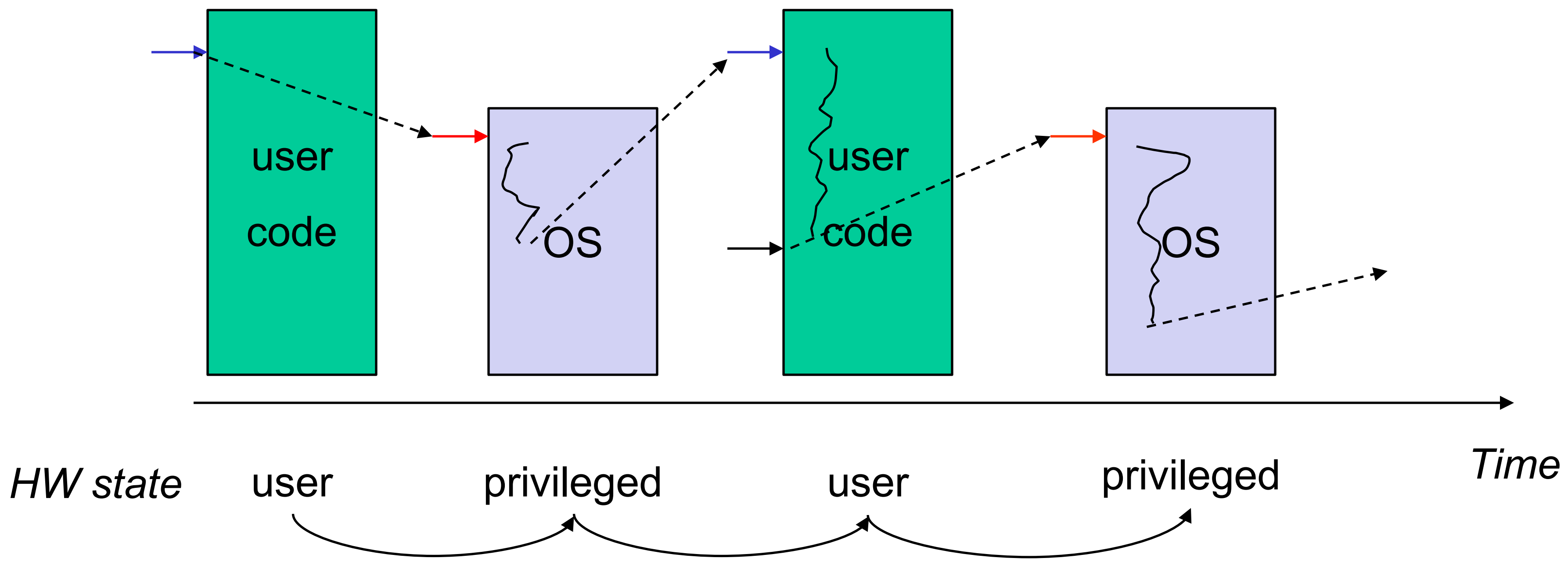
- Q: So how does code running at user level (apps) do something privileged?
  - e.g., how can it write to a disk if it can't execute the I/O instructions that are needed to do I/O?
- A: Ask code that can (the OS) to do it for you.

- User programs must cause execution of an OS
  - OS defines a set of system calls
  - App code leaves 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 is a name for which system call is being requested
    - usually a syscall number
  - App somehow causes processor to elevate its privilege level to 0

Elevating the CPU privilege level

- Syscall instruction
  - Like a <u>protected</u> procedure call
  - What's protected?
    - The entry point
  - What about the arguments?
    - Are they valid?
      - Would assuming they are potentially cause an execution error while running the OS?

Dynamic View



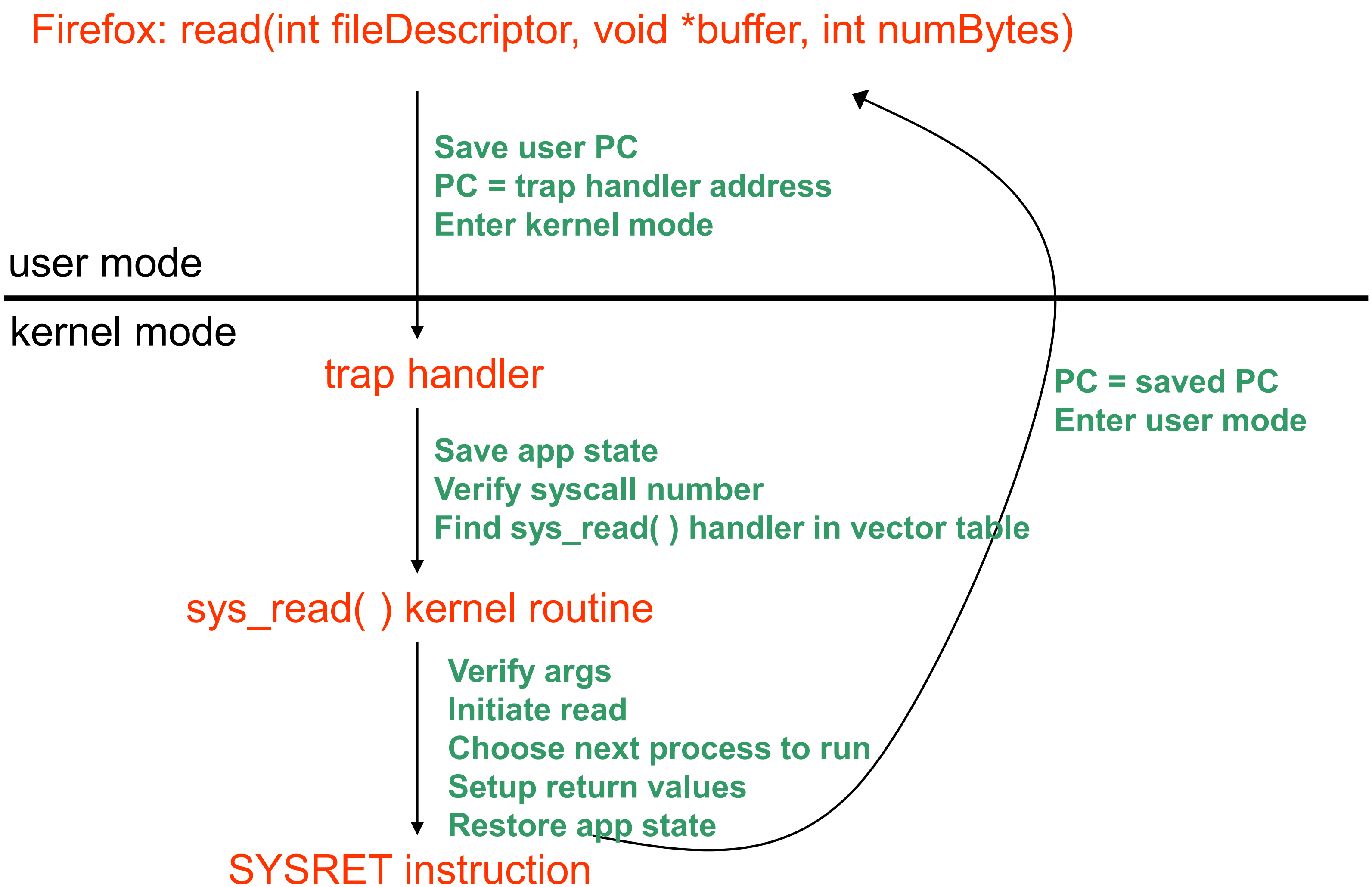
syscall/sysret instructions

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

"Protected procedure call"

- Similar to local 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
  - 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 PC to return address and sets execution mode to user

A kernel crossing illustrated



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 shown
  - Acquiring privileged mode and branching to the trap handler are inseparable
- Terminology:
  - **Interrupt**: asynchronous; caused by an external device
  - **Exception**: synchronous; unexpected problem with instruction
  - **Trap**: synchronous; 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, …

x86 Interrupt/Trap Handling: Interrupt vector

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

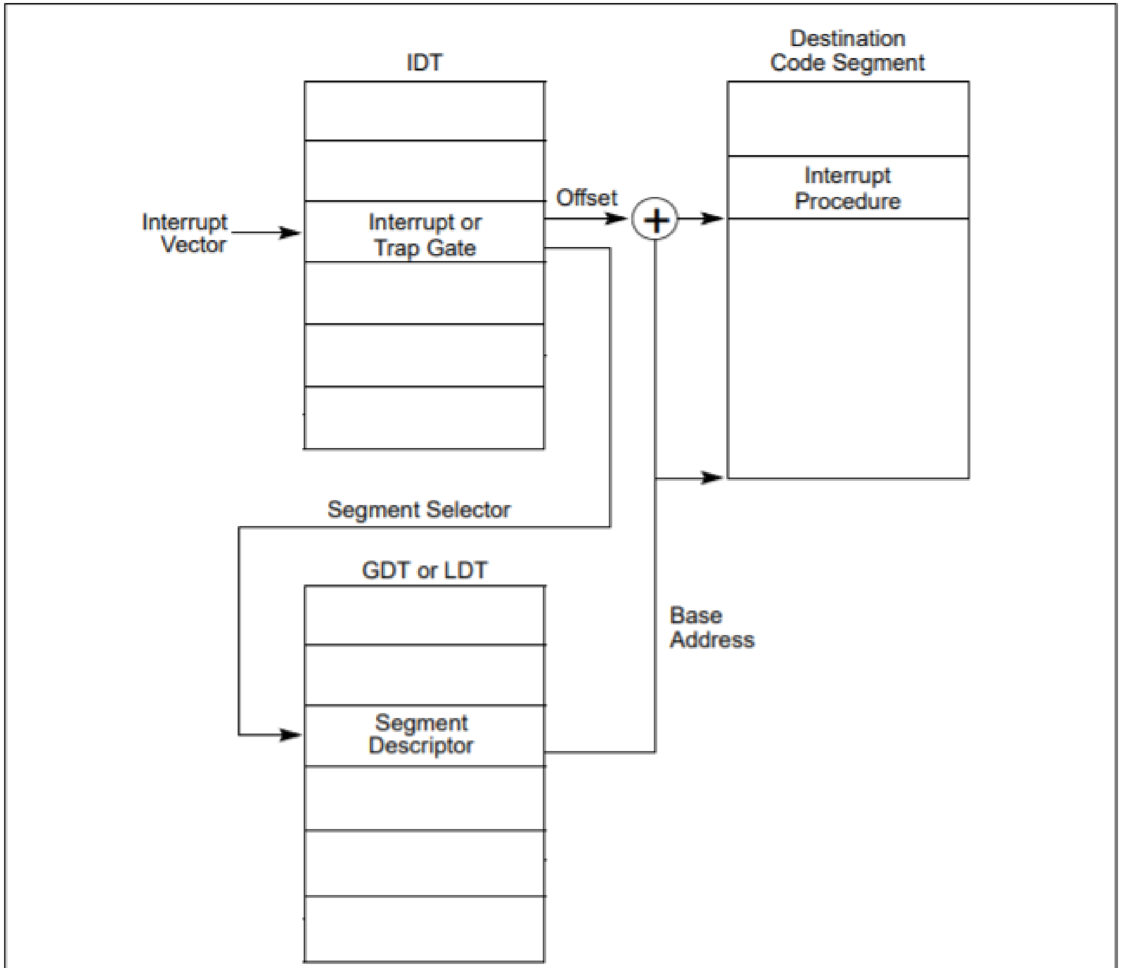


Figure 6-3. Interrupt Procedure Call

x86 Interrupt/Trap Handling: Finding the IDT

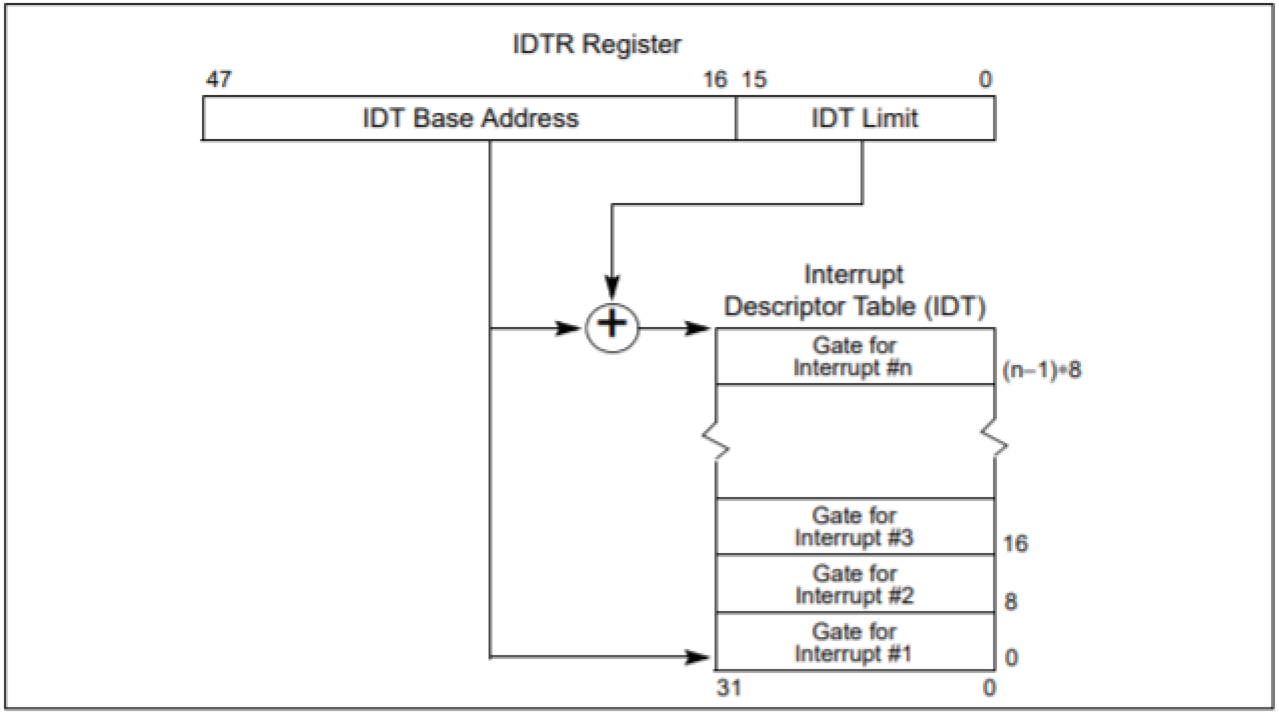


Figure 6-1. Relationship of the IDTR and IDT

x86 Interrupt/Trap Handling: IDT entries

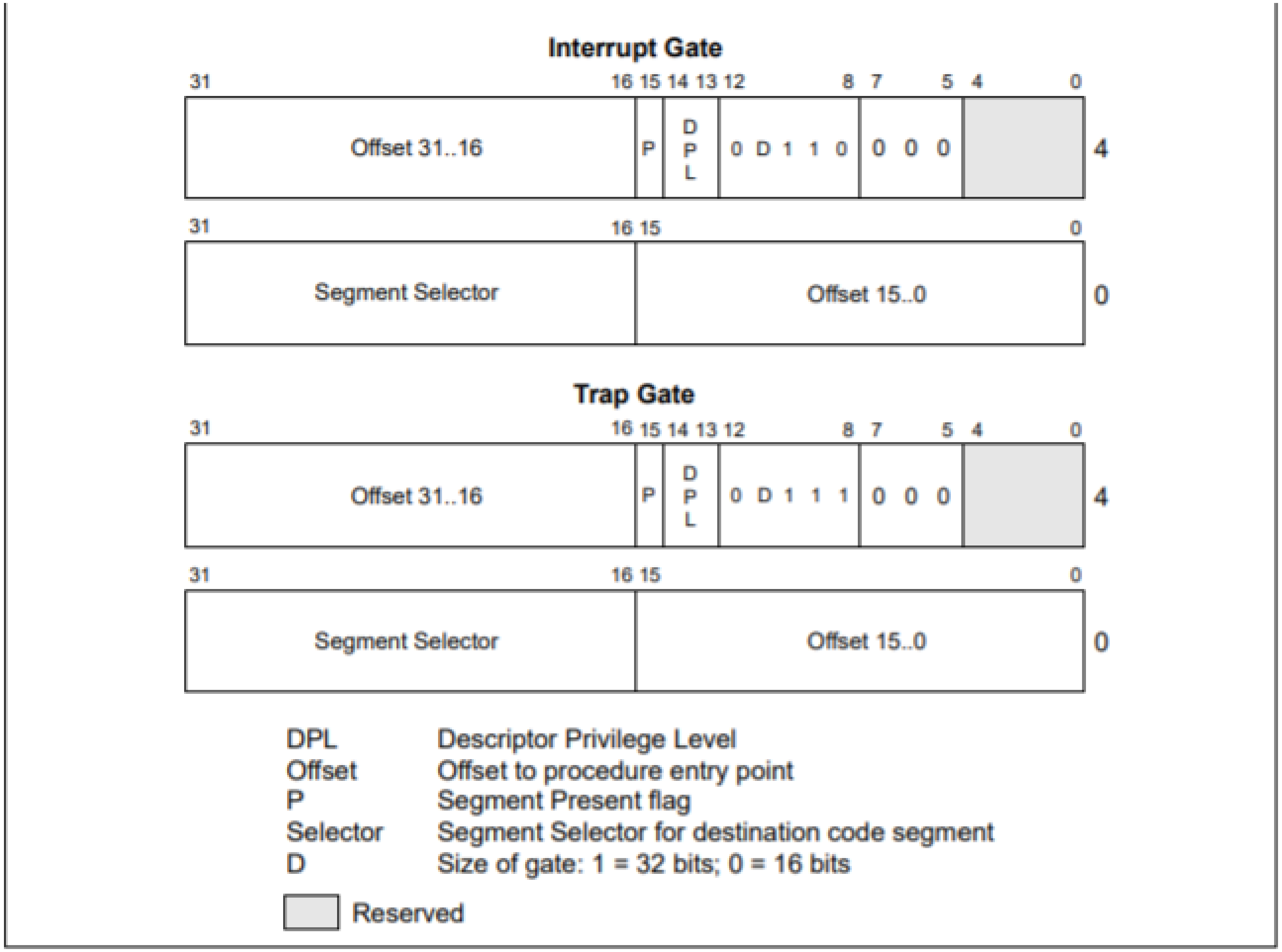
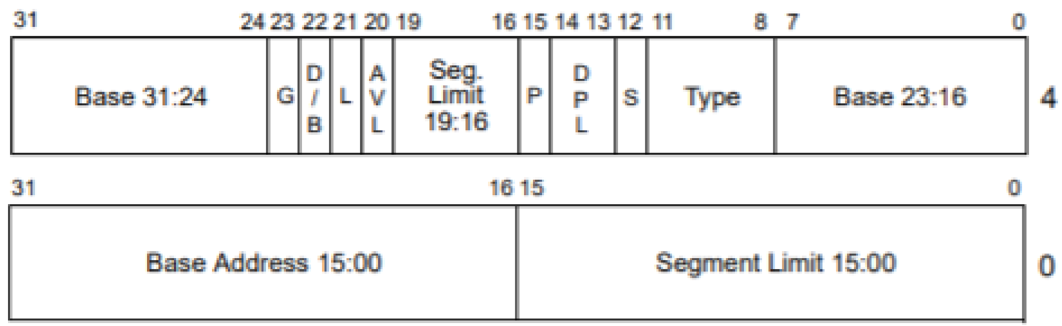


Figure 6-2. IDT Gate Descriptors

x86 Interrupt/Trap Handling: Segment Descriptors



L — 64-bit code segment (IA-32e mode only)
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

Figure 3-8. Segment Descriptor

x86 Interrupt/Trap Handling: Stacks

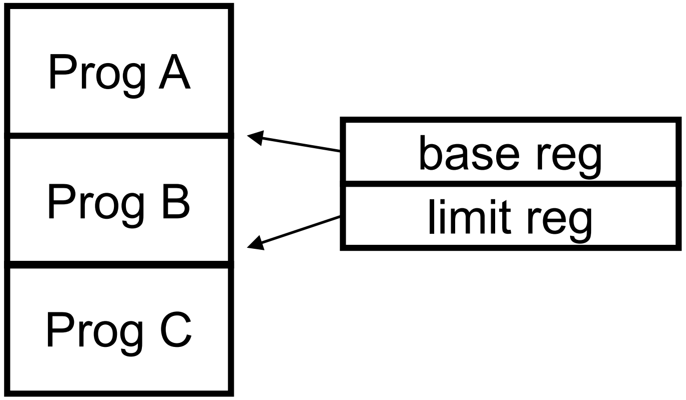
31	15	0	
I/O Map Base Address	Reserved	T	100
Reserved	LDT Segment Selector		96
Reserved	GS		92
Reserved	FS		88
Reserved	DS		84
Reserved	SS		80
Reserved	CS		76
Reserved	ES		72
EDI			68
ESI			64
EBP			60
ESP			56
EBX			52
EDX			48
ECX			44
EAX			40
EFLAGS			36
EIP			32
CR3 (PDBR)			28
Reserved	SS2		24
ESP2			20
Reserved	SS1		16
ESP1			12
Reserved	SS0		8
ESP0			4
Reserved	Previous Task Link		0

Reserved bits. Set to 0.

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

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

- coming later in the course
  - also coming earlier in your course sequence!
- paging, segmentation, virtual memory
  - page tables, page table pointers
  - translation lookaside buffers (TLBs)
  - page fault handling

I/O control

- Issues:
  - how does the OS start an I/O?
    - special I/O instructions
    - memory-mapped I/O
  - how does the OS notice an I/O has finished?
    - polling
    - Interrupts
  - how does the OS exchange data with an I/O device?
    - Programmed I/O (PIO)
    - Direct Memory Access (DMA)

Asynchronous I/O

- Interrupts are the basis for asynchronous I/O
  - device performs an operation asynchronously to CPU
  - 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
- What’s the advantage of asynchronous I/O?

Timers

- How can the OS prevent runaway user programs from hogging the CPU (infinite loops?)
  - 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?

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???
  - 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 biggest difference between "systems programming" and "traditional application programming"
  - modern "event-oriented" application programming is a middle ground
  - And in a multi-core world, more and more apps have internal concurrency
- Arises from the architecture
  - Can be sugar-coated, but cannot be totally abstracted away
- Huge 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

- Intel's big challenge: finding applications that require new hardware support, so that you will want to upgrade to a new computer to run them

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 scribbling on the memory of another user program?
  - Why might you want to allow it to?!
- What prevents a user program from scribbling on 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?!
- What prevents a user program from running away with the CPU?