W4118: interrupts and system calls

Junfeng Yang

References: Modern Operating Systems (3rd edition), Operating Systems Concepts (8th edition), previous W4118, and OS at MIT, Stanford, and UWisc

Outline

- Dual mode of operation
- Interrupt
- System call

Need for protection

- Kernel privileged, cannot trust user processes
  - User processes may be malicious or buggy
- Must protect
  - User processes from one another
  - Kernel from user processes

Hardware mechanisms for protection

- Memory protection
  - Segmentation and paging
    - E.g., kernel sets segment/page table
- Timer interrupt
  - Kernel periodically gets back control
- Dual mode of operation
  - Privileged (+ non-privileged) operations in kernel mode
  - Non-privileged operations in user mode

What operations are privileged?

- Read raw keyboard input
- Call printf()
- Call write()
- Write global descriptor table
- Divide by 0
- Set timer interrupt handler
- Set segment registers
- Load cr3

x86 protection modes

- Four modes (0-3), but often only 0 & 3 used
  - Kernel mode: 0
  - User mode: 3
  - "Ring 0", "Ring 3"
- Segment has Descriptor Privilege Level (DPL)
  - DPL of kernel code and data segments: 0
  - DPL of user code and data segments: 3
- Current Privilege Level (CPL) = current code segment's DPL
  - Can only access data segments when CPL <= DPL

Outline

- Dual mode of operation
- Interrupt
- System call

OS: “event driven”

- Events causing mode switches
  - System calls: issued by user processes to request system services
  - Exceptions: illegal instructions (e.g., division by 0)
  - Interrupts: raised by devices to get OS attention

- Often handled using same hardware mechanism: interrupt
  - Also called trap

Interrupt view of CPU

```
while (fetch next instruction) {
    run instruction;
    if (there is an interrupt) {


        process interrupt


    }
}
```

x86 interrupt view

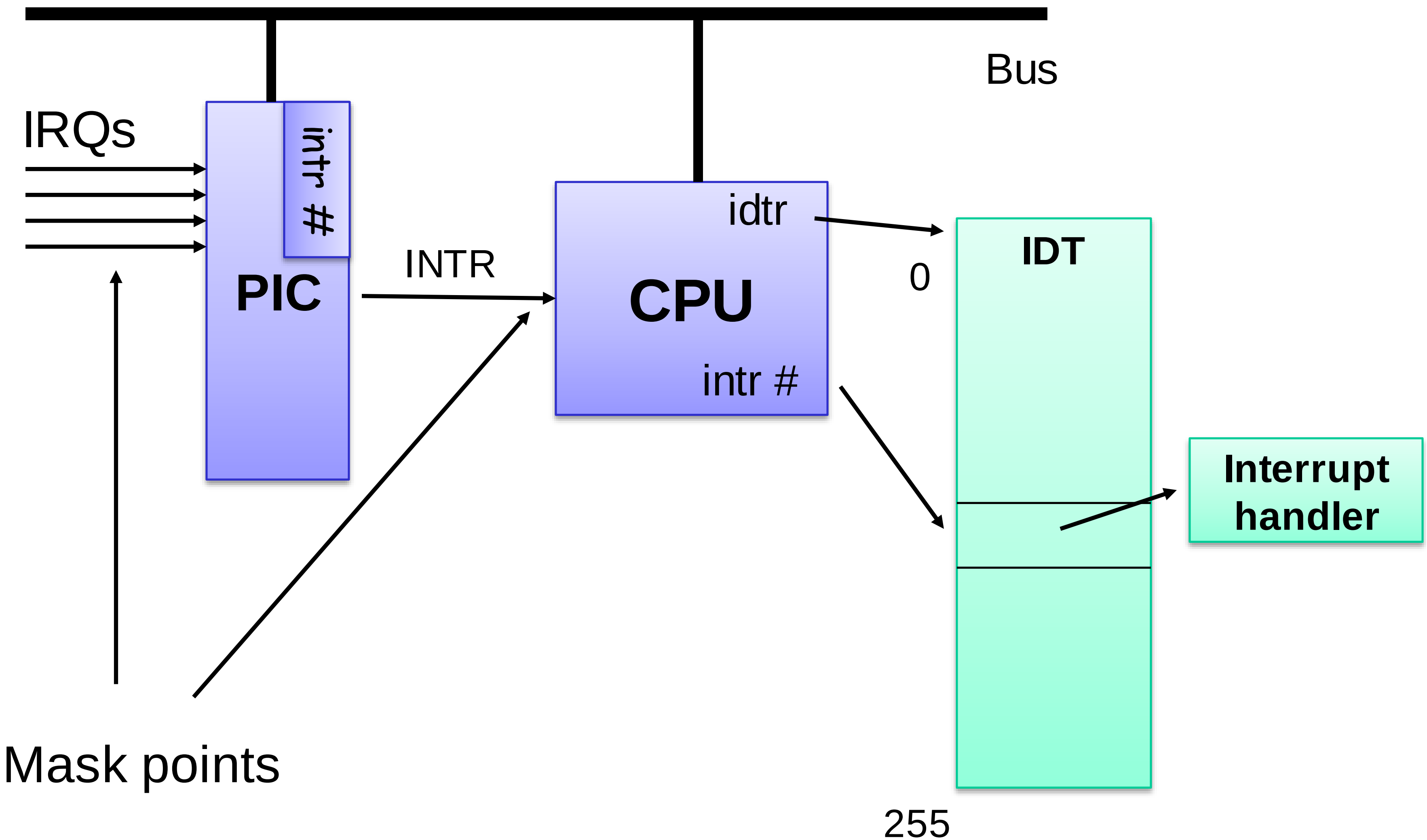
```
while (fetch next instruction) {
    run instruction;
    if (there is an interrupt) {
        switch to kernel stack if necessary
        save CPU context and error code if any
        find OS-provided interrupt handler
        jump to handler
        restore CPU context when handler returns
    }
}
```

- Q1: how does hardware find OS-provided interrupt handler?
- Q2: why switch stack?
- Q3: what CPU context to save and restore?
- Q4: what does handler do?

Q1: how to find interrupt handler?

- Hardware maps interrupt type to interrupt number
- OS sets up Interrupt Descriptor Table (IDT) at boot
  - Also called interrupt vector
  - IDT is in memory
  - Each entry is an interrupt handler
  - OS lets hardware know IDT base
  - Defines all kernel entry points
- Hardware finds handler using interrupt number as index into IDT
  - handler = IDT[intr_number]

x86 interrupt hardware (legacy)



x86 interrupt numbers

- Total 256 number [0, 255]
- Intel reserved first 32, OS can use 224
  - 0: divide by 0
  - 1: debug (for single stepping)
  - 2: non-maskable interrupt
  - 3: breakpoint
  - 14: page fault
  - 64: system call in xv6
- xv6 traps.h

x86 interrupt gate descriptor

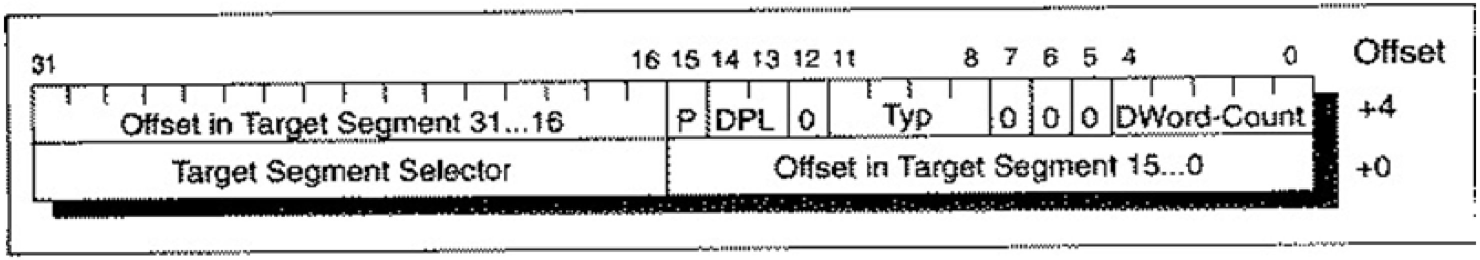
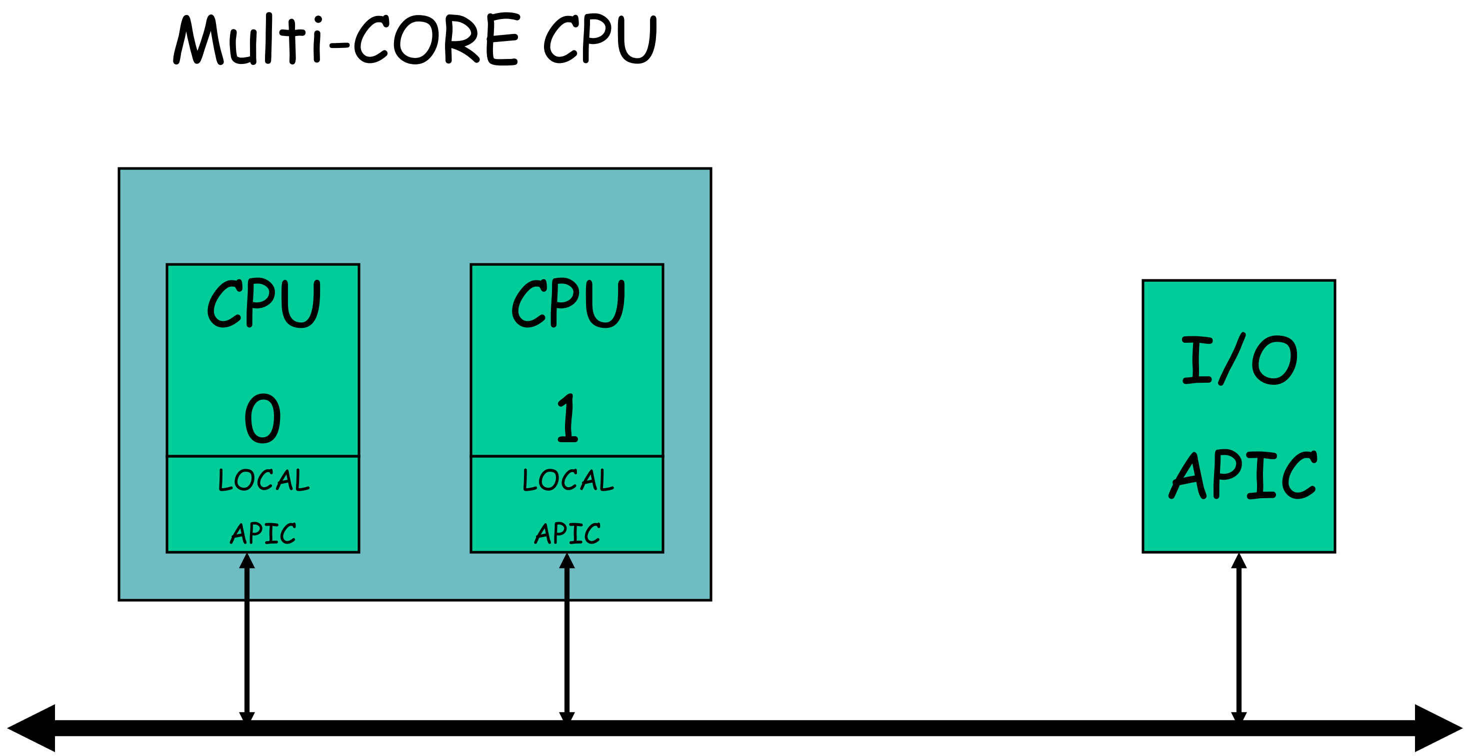


Figure 3.12: Format of an i386 gate descriptor.

- Interrupt gate descriptor
  - Code segment selector and offset of handler
  - Descriptor Privilege Level (DPL)
  - Trap or exception flag
- lidt instruction loads CPU with IDT base

- xv6
  - Handler entry points: vector.S
  - Interrupt gate format: SETGATE in mmu.h
  - IDT initialization: tvinit() & idtinit() in trap.c

Multiple Logical Processors



Advanced Programmable Interrupt Controller is needed to perform 'routing' of I/O requests from peripherals to CPUs

(The legacy PICs are masked when the APICs are enabled)

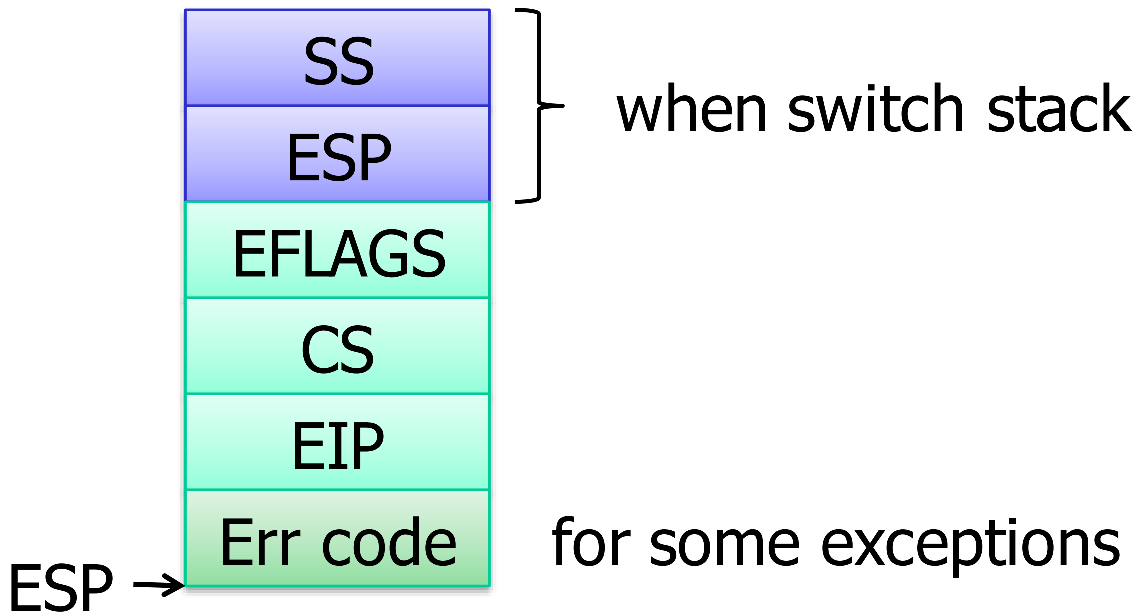
APIC, IO-APIC, LAPIC

- Advanced PIC (APIC) for SMP systems
  - Used in all modern systems
  - Interrupts “routed” to CPU over system bus
  - IPI: inter-processor interrupt
- Local APIC (LAPIC) versus “frontend” IO-APIC
  - Devices connect to front-end IO-APIC
  - IO-APIC communicates (over bus) with Local APIC
- Interrupt routing
  - Allows broadcast or selective routing of interrupts
  - Ability to distribute interrupt handling load
  - Routes to lowest priority process
    - Special register: Task Priority Register (TPR)
  - Arbitrates (round-robin) if equal priority

Q2: why switch stack?

- Cannot trust stack (SS, ESP) of user process!
- x86 hardware switches stack when interrupt handling requires user-kernel mode switch
- Where to find kernel stack?
  - Task gate descriptor has SS and ESP for interrupt
  - ltr loads CPU with task gate descriptor
- xv6 assigns each process a kernel stack, used in interrupt handling
  - switchuvm() in vm.c

Q3: what does hardware save?



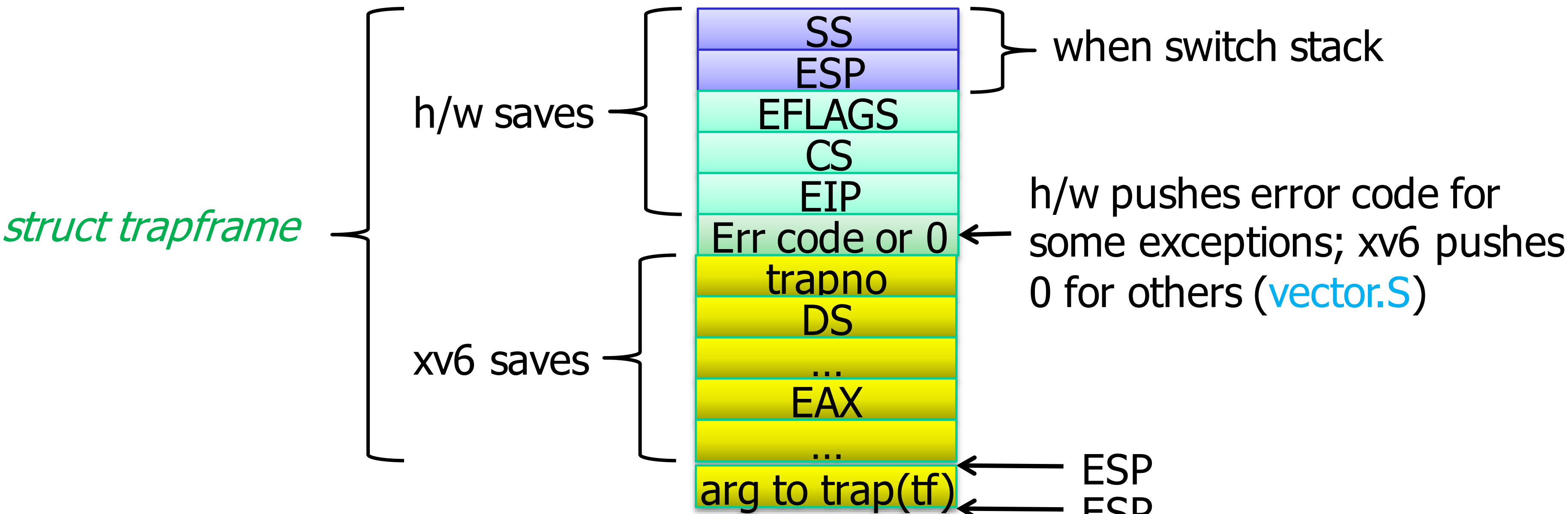
- x86 saves SS, ESP, EFLAGS, CS, EIP, Err code
- Restored by iret
- OS can save more context

Q4: what does interrupt handler do?

- Typical steps
  - Assembly to save additional CPU context
  - Invoke C handler to process interrupt
    - E.g., communicate with I/O devices
  - Invoke kernel scheduler
  - Assembly to restore CPU context and return

- xv6
  - Interrupt handler entries: vector.S
  - Saves & restore additional CPU context: trapasm.S
  - C handler: trap.c, *struct trapframe* in x86.h

xv6 kernel stack before calling trap(tf)



- xv6 saves all registers (user-mode CPU context)
- *struct trapframe* (x86.h) captures this layout
- "pushl %esp" pushes argument for trap(tf)

Issues with interrupts

- Interrupt dispatching has overhead
- Interrupt runs at the "highest priority"
  - Increases responsiveness, but …
- So, must be very careful
  - Can interrupt handler run for a very long time?
  - What if system cannot take more work?
  - Should we allow nested interrupts?
- Real-world: interrupt processing very complex (e.g., Linux)
- In general
  - Do as little as possible in the interrupt handler
  - Defer non-critical actions till later

Interrupt v.s. Polling

- Instead for device to interrupt CPU, CPU can poll the status of device
  - Intr: "I want to see a movie."
  - Poll: for(each week) {"Do you want to see a movie?"}

- Good or bad?
  - For mostly-idle device?
  - For busy device?
  - Responsiveness?
  - Overhead?

Outline

- Dual mode of operation
- Interrupt
- System call

System call

- User processes cannot perform privileged operations themselves
- Must request OS to do so on their behalf by issuing system calls
- OS must validate system call parameters

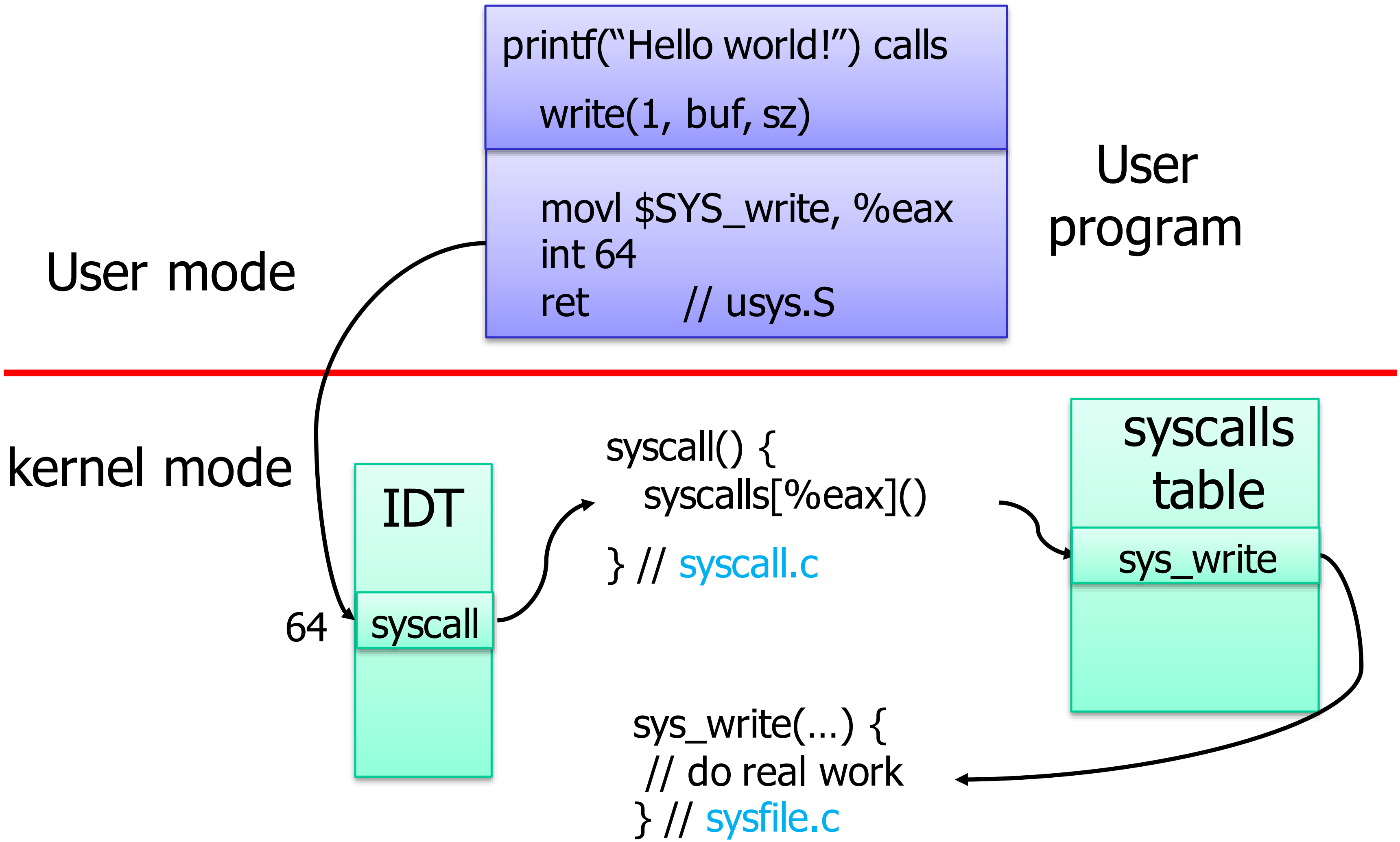
Examples of Windows and Unix System Calls

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

System call dispatch

1. Kernel assigns system call type a system call number
2. Kernel initializes system call table, mapping system call number to function implementing the system call
   - Also called system call vector
3. User process sets up system call number and arguments
4. User process runs int X
5. Hardware switches to kernel mode and invokes kernel's interrupt handler for X (interrupt dispatch)
6. Kernel looks up syscall table using system call number
7. Kernel invokes the corresponding function
8. Kernel returns by running iret (interrupt return)

xv6 system call dispatch



System call parameter passing

- Typical methods
  - Pass via registers (e.g., Linux)
  - Pass via user-mode stack (e.g., xv6)
  - Pass via designated memory region
- xv6 system call parameter passing
  - Arguments pushed onto user stack based on gcc calling convention
  - Kernel function uses special routines to fetch these arguments
    - syscall.c
    - Why?

xv6 system call naming convention

- Usually the user-mode wrapper foo() (usys.S) traps into kernel, which calls sys_foo()
  - sys_foo() implemented in sys*.c
  - Often wrappers to foo() in kernel
- System call number for foo() is SYS_foo
  - syscalls.h
- All system calls begin with sys_

Linux system call naming convention

- Usually the user-mode wrapper foo() traps into kernel, which calls sys_foo()
  - sys_foo is defined by DEFINEx(foo, …)
  - Expands to "asmlinkage long sys_foo(void)"
  - Where x specifies the number of parameters to syscall
  - Often wrappers to foo() in kernel
- System call number for foo() is __NR_foo
  - arch/x86/include/asm/unistd_32.h
  - Architecture specific
- All system calls begin with sys_

System Call from Userspace

- Generic syscall stub provided in libc
  - _syscalln
  - Where n is the number of parameters
- Example
  - To implement:

```
ssize_t write(int fd, const void *buf, size_t count);
```

  - Declare:

```
#define __NR_write 4 /* Syscall number */
_syscall3(ssize_t, write, int, fd, const void*, buf, size_t count)
```

- Usually done in libc for standard syscalls

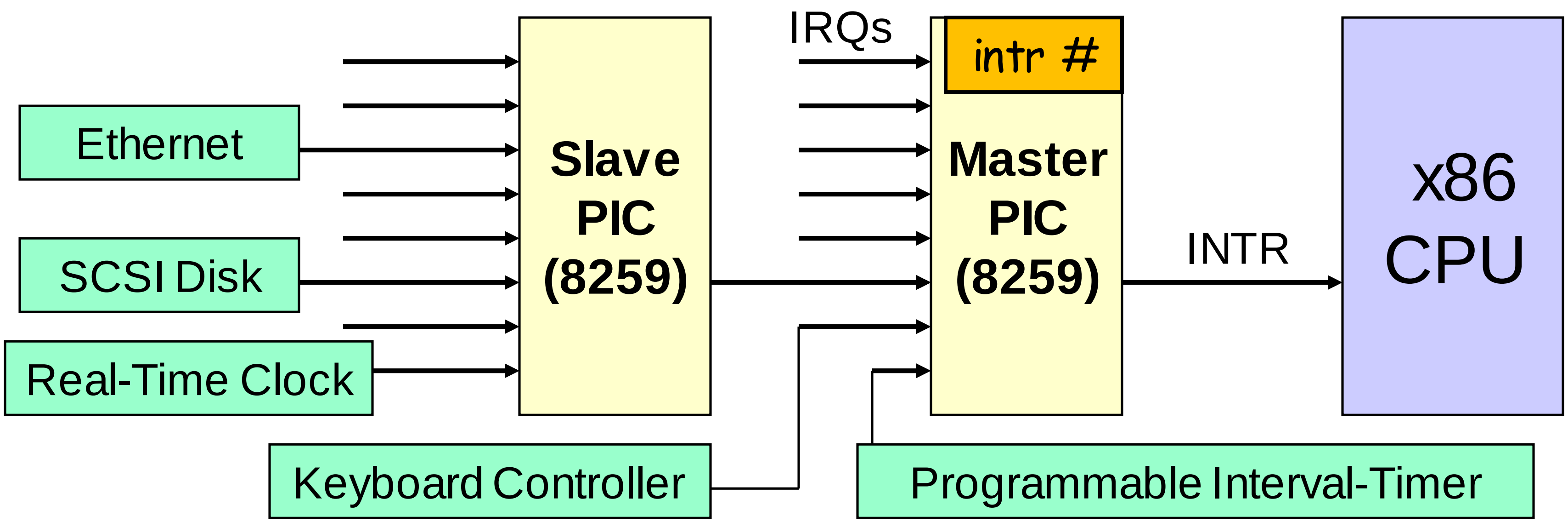
Tracing system calls in Linux

- Use the "strace" command (man strace for info)
- Linux has a powerful mechanism for tracing system call execution for a compiled application
- Output is printed for each system call as it is executed, including parameters and return codes
- ptrace() system call is used to implement strace
  - *Also used by debuggers (breakpoint, singlestep, etc)*
- Use the "ltrace" command to trace dynamically loaded library calls

System Call Tracing Demo

- ssh clic-lab.cs.columbia.edu
- pwd
- ltrace pwd
  - Library calls
  - setlocale, getcwd, puts: makes sense
- strace pwd
  - System calls
  - execve, open, fstat, mmap, brk: what are these?
  - getcwd, write

x86 interrupt hardware (legacy)



- I/O devices raise Interrupt Request lines (IRQ)
- Programmable Interrupt controller (PIC) maps IRQ to Interrupt Numbers
- PIC raises INTR line to interrupt CPU
- Nest PIC for more devices