CS 318 Principles of Operating Systems

Fall 2018

Lecture 2: Architecture Support for OS

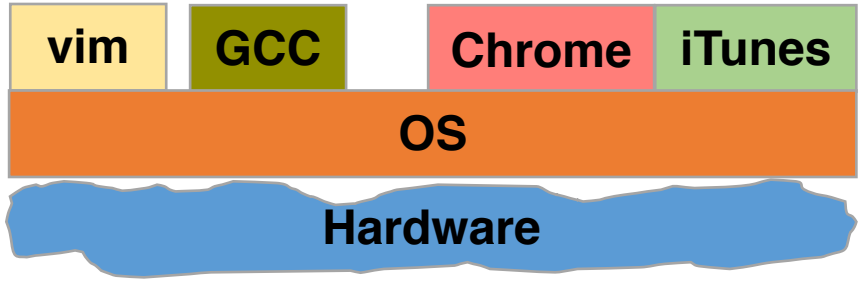
Ryan Huang

Slides partially adapted from Geoff Voelker's (UCSD) lectures

Administrivia

- **Lab 0**
  - Due next Thursday (09/13) midnight, done individually
  - Overview session this Friday 10am @ Malone G33/35
- **Project groups**
  - Fill out Google form (link in Piazza post)
  - Fill out even if you are working alone
  - If you don't have a group, talk with neighbors, search for teammates on Piazza

Why Start With Hardware?



- **OS functionality depends upon the architectural features of H/W**
  - Key goals of an OS are to enforce protection and resource sharing
  - If done well, applications can be oblivious to HW details
- **Architectural support can greatly simplify or complicate OS tasks**
  - Early PC operating systems (DOS, MacOS) lacked virtual memory in part because the architecture did not support it
  - Early Sun 1 computers used two M68000 CPUs to implement virtual memory (M68000 did not have VM hardware support)

Architectural Features for OS

- **Features that directly support the OS include**
  - Bootstrapping (Lab 0)
  - Protection (kernel/user mode)
  - Protected instructions
  - Memory protection
  - System calls
  - Interrupts and exceptions
  - Timer (clock)
  - I/O control and operation
  - Synchronization

Types of Arch Support

I. Manipulating privileged machine state

- Protected instructions
- Manipulate device registers, TLB entries, etc.

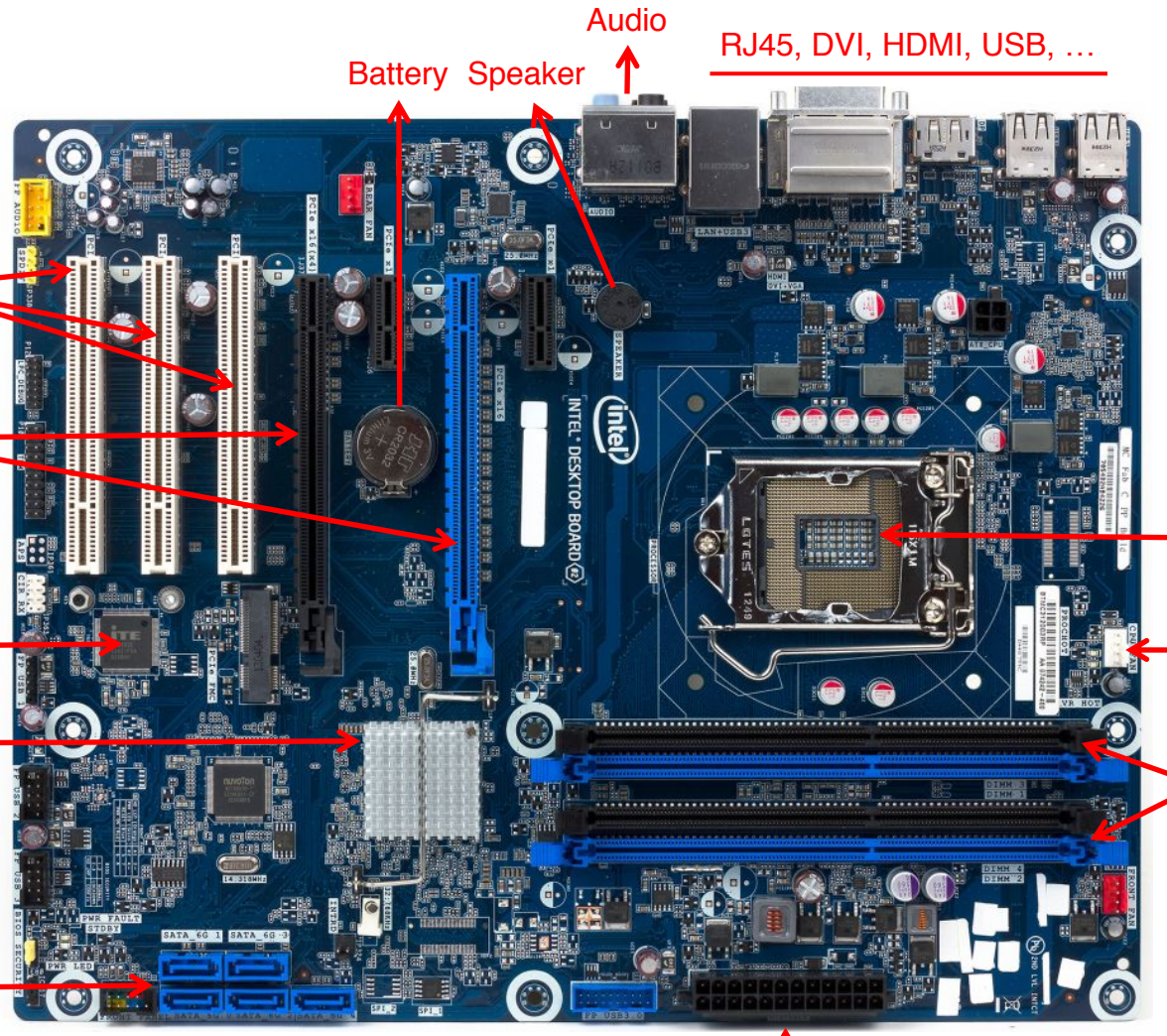
II. Generating and handling "events"

- Interrupts, exceptions, system calls, etc.
- Respond to external events
- CPU requires software intervention to handle fault or trap

III. Mechanisms to support synchronization

- Interrupt disabling/enabling, atomic instructions

What Is Inside A Computer?



A Motherboard (Intel DH87MC)

Source: Intel® Desktop Board DH87MC Technical Product Specification

Typical PC System Architecture

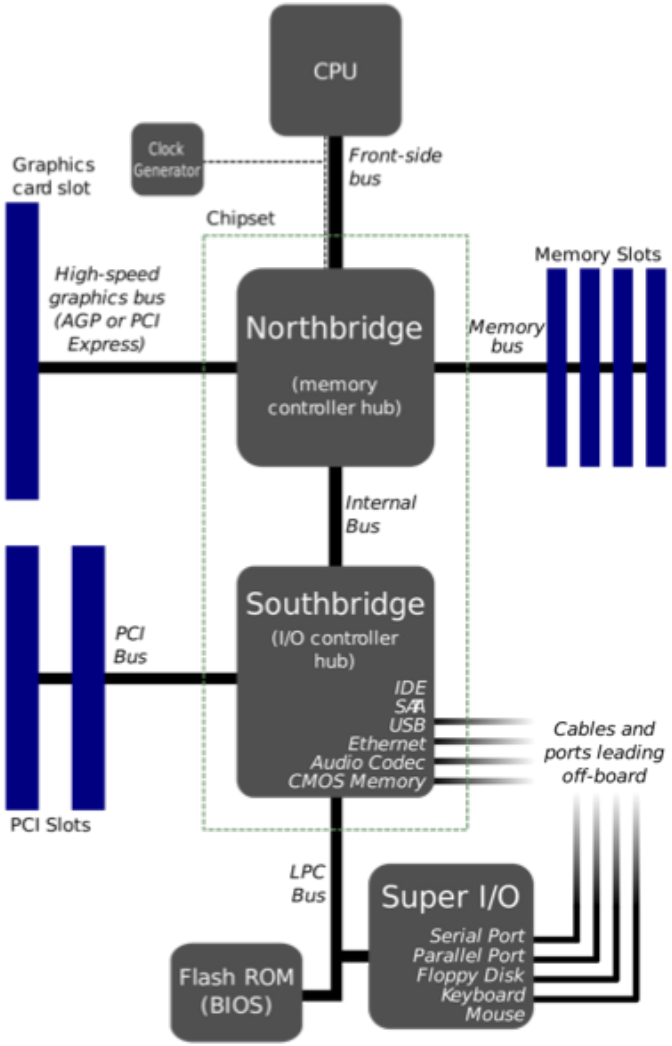


diagram source: wikipedia

Thought Experiment: A World of Anarchy

- **Any program in the system can…**
  - Directly access I/O devices
  - Write anywhere in memory
  - Read content from any memory address
  - Execute machine halt instruction

- **Do you trust such system?**
  - use Facebook app in this system
  - use Banking app in this system

- **Challenge: protection**
  - How to execute a program with restricted privilege?

Thought Experiment: A Solution

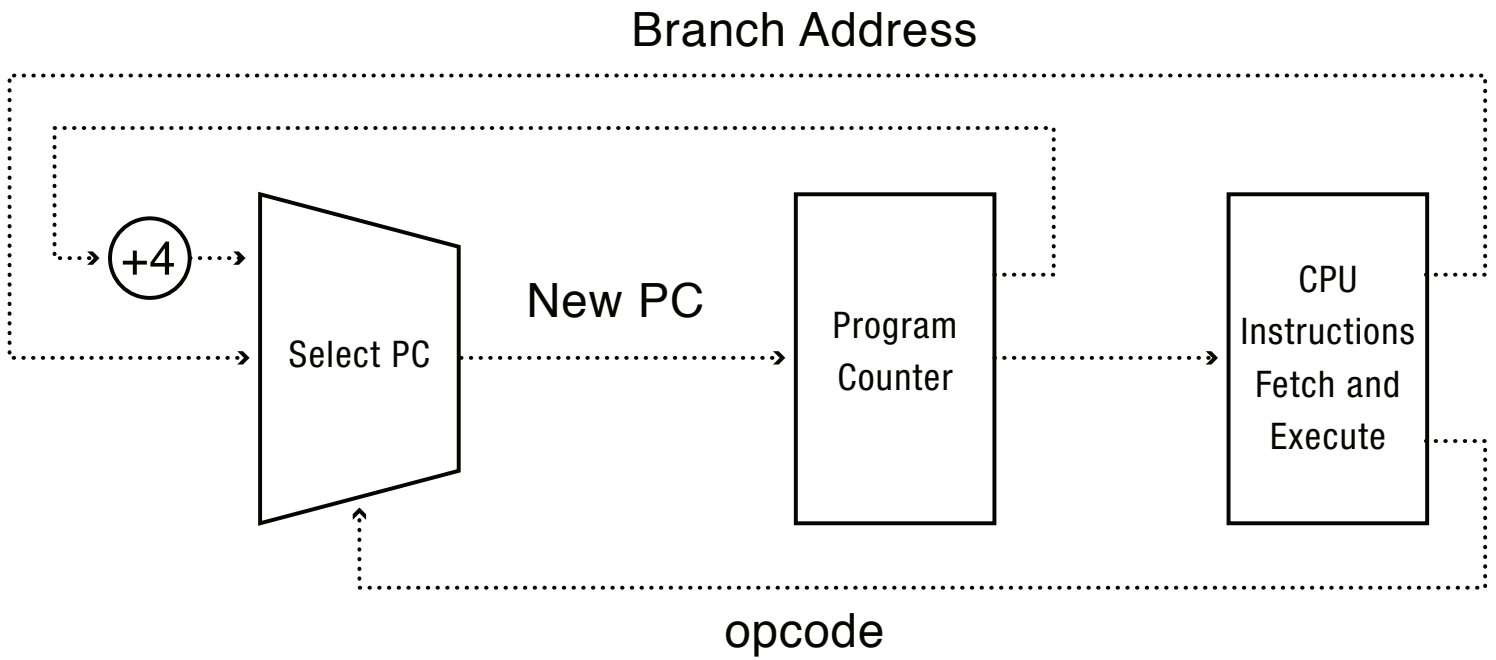
- **How can we implement execution with limited privilege?**
  - Execute each program instruction through a simulator (OS)
  - If the instruction is permitted, do the instruction
  - Otherwise, stop the process
  - Basic model in Javascript and other interpreted languages
- **How do we go faster?**
  - Observation: most instructions are perfectly safe!
  - Run the unprivileged code directly on the CPU
  - Do the check in h/w

H/W Support: Dual-Mode Operation in CPU

- **User mode**
  - Limited privileges
  - Only those granted by the operating system kernel
- **Kernel mode**
  - Execution with the full privileges of the hardware
  - Read/write to any memory, access any I/O device, read/write any disk sector, send/read any packet
- **On the x86, the Current Privilege Level (*CPL*) in the CS register**
- **On the MIPS, the status register**

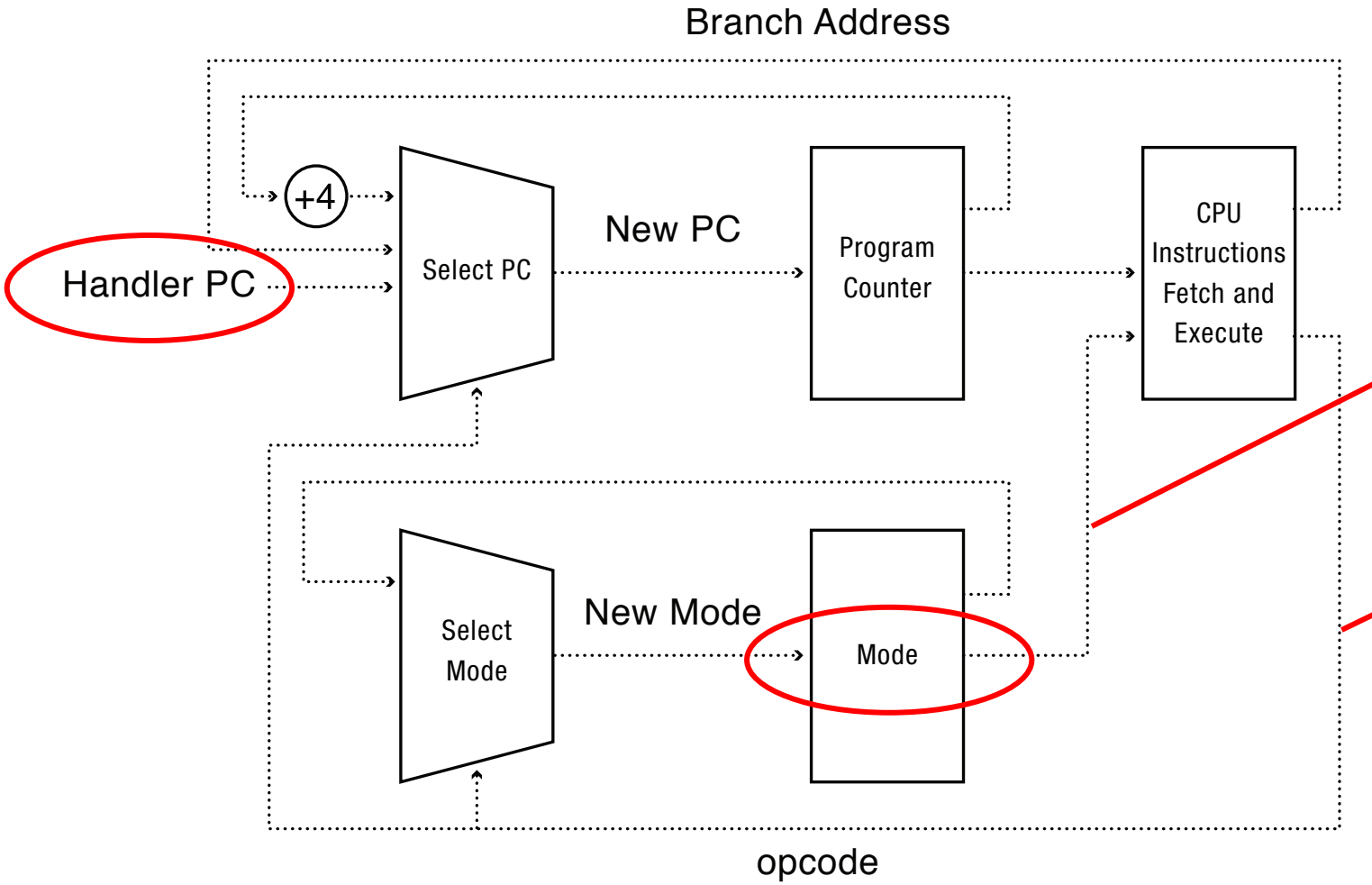
A Simple Model of a CPU

- **Basic operation**
  - Fetch next instruction → Decode instruction → Execute



add/subtract,
read/write memory,
call function, branch…

A CPU with Dual-Mode Operation



Check if a instruction is allowed to be executed

Change mode upon certain instructions (e.g., trap)

Protected Instructions

- **A subset of instructions restricted to use only by the OS**
  - Known as protected (privileged) instructions
- **Only the operating system can …**
  - Directly access I/O devices (disks, printers, etc.)
    - Security, fairness (why?)
  - Manipulate memory management state
    - Page table pointers, page protection, TLB management, etc.
  - Manipulate protected control registers
    - Kernel mode, interrupt level
  - Halt instruction (why?)

INVLPG—Invalidate TLB Entries

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01/7	INVLPG *m*	M	Valid	Valid	Invalidate TLB entries for page containing *m*.

NOTES:

* See the IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

Description

Invalidates any translation lookaside buffer (TLB) entries specified with the source operand. The source operand is a memory address. The processor determines the page that contains that address and flushes all TLB entries for that page.[1]

The INVLPG instruction is a privileged instruction. When the processor is running in protected mode, the CPL must be 0 to execute this instruction.

The INVLPG instruction normally flushes TLB entries only for the specified page; however, in some cases, it may flush more entries, even the entire TLB. The instruction is guaranteed to invalidates only TLB entries associated with the current PCID. (If PCIDs are disabled — CR4.PCIDE = 0 — the current PCID is 000H.) The instruction also invalidates any global TLB entries for the specified page, regardless of PCID.

For more details on operations that flush the TLB, see "MOV—Move to/from Control Registers" and Section 4.10.4.1, "Operations that Invalidate TLBs and Paging-Structure Caches," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

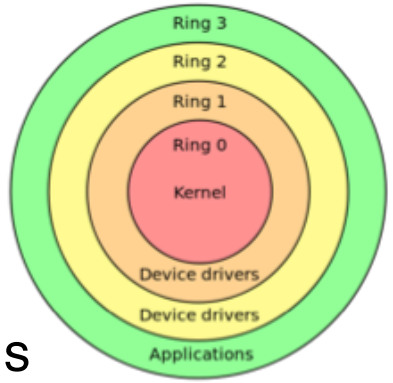
This instruction's operation is the same in all non-64-bit modes. It also operates the same in 64-bit mode, except if the memory address is in non-canonical form. In this case, INVLPG is the same as a NOP.

IA-32 Architecture Compatibility

The INVLPG instruction is implementation dependent, and its function may be implemented differently on different

Beyond Dual-Mode Operations

- **(Modern) CPU may provide more than 2 privilege levels**
  - Called hierarchical protection domains or protection rings
  - x86 supports **four** levels:
    - bottom 2 bits (CPL) of the CS register indicate execution privilege
      - ring 0 (CPL=00) is kernel mode, ring 3 (CPL=11) is user mode
  - Multics provides 8 levels of privilege
  - Even seen in mobile devices
    - ARMv7 processors in modern smartphones have 8 different protection levels



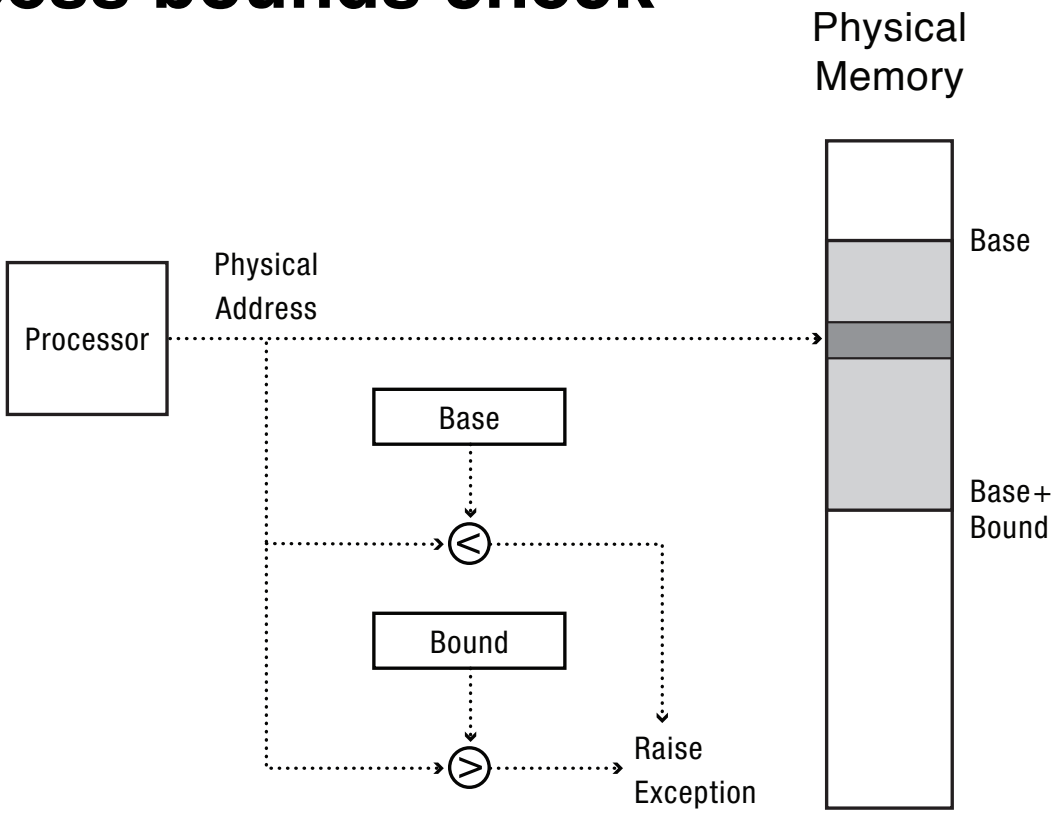
- **Why?**
  - Protect the OS from itself (software engineering)
  - reserved for vendor, e.g., virtualization

Memory Protection

- **Why?**
  - OS must be able to protect programs from each other
  - OS must protect itself from user programs
- **May or may not protect user programs from OS**
  - Raises question of whether programs should trust the OS
  - Untrusted operating systems? (Intel SGX)
- **Memory management hardware (MMU) provides the mechanisms**
  - Base and limit registers
  - Page table pointers, page protection, segmentation, TLB
  - Manipulating the hardware uses protected (privileged) operations

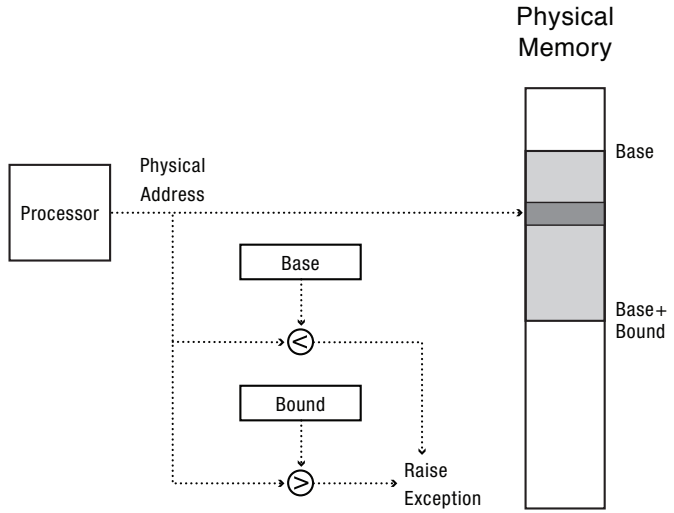
Simple Memory Protection

- **Memory access bounds check**



Simple Memory Protection

- **Memory access bounds check**
- **Problem**
  - Inflexible:
    - fix allocation, difficult to expand head and stack
  - Memory sharing
  - Memory fragmentation
  - Physical memory
    - require changes to mem instructions each time the program is loaded

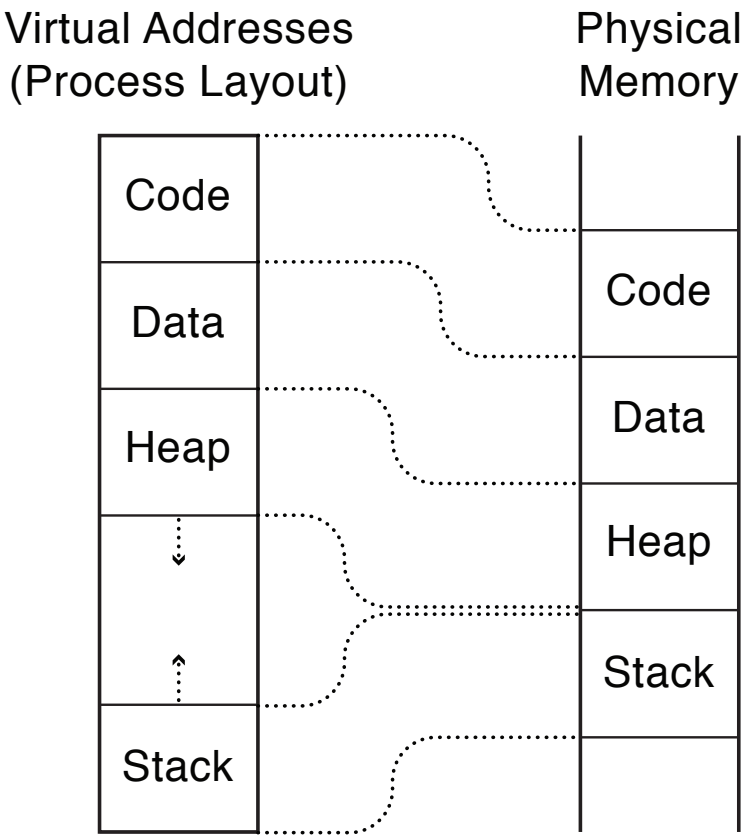


Idea: Virtual Address

- **Programs refer to memory by virtual addresses**
  - start from 0
  - illusion of “owning” the entire memory address space
- **The virtual address is translated to physical address**
  - upon each memory access
  - done in hardware using a table
  - table setup by the OS

Idea: Virtual Address

- **Programs refer to memory by virtual addresses**
  - start from 0
  - illusion of “owning” the entire memory address space
- **The virtual address is translated to physical address**
  - upon each memory access
  - done in hardware using a table
  - table setup by the OS



Types of Arch Support

I. **Manipulating privileged machine state**
   - Protected instructions
   - Manipulate device registers, TLB entries, etc.

II. **Generating and handling "events"**
   - Interrupts, exceptions, system calls, etc.
   - Respond to external events
   - CPU requires software intervention to handle fault or trap

III. **Mechanisms to support synchronization**
   - Interrupt disabling/enabling, atomic instructions

OS Control Flow

- **After OS booting, all entry to kernel is a result of some event**
  - event immediately stops current execution
  - changes mode to kernel mode
  - invoke a piece of code to handle event (event handler)
- **When the processor receives an event of a given type, it**
  - transfers control to handler within the OS
  - handler saves program state (PC, regs, etc.)
  - handler executes core functionality
    - e.g., writing data to disk
  - handler restores program state, returns to program

Events

- **An event is an "unnatural" change in control flow**
  - Events immediately stop current execution
  - Changes mode, context (machine state), or both
- **The kernel defines a handler for each event type**
  - The specific types of events are defined by the architecture
    - e.g., timer event, I/O interrupt, system call trap
  - In effect, the operating system is one big event handler

Event: Interrupt vs. Exceptions

- **Two kinds of events, interrupts and exceptions**
- **Interrupts are caused by an external event (asynchronous)**
  - Device finishes I/O, timer expires, etc.
- **Exceptions are caused by executing instructions (synchronous)**
  - x86 `int` instruction, page fault, divide by zero, etc.

Interrupts

- **Interrupts signal asynchronous events**
  - Indicates some device needs services
  - I/O hardware interrupts
  - Software and hardware timers
- **Why?**
  - A computer is more than CPU
    - keyboard, disk, printer, camera, etc.
  - These devices occasionally need attention
    - but cannot predict when

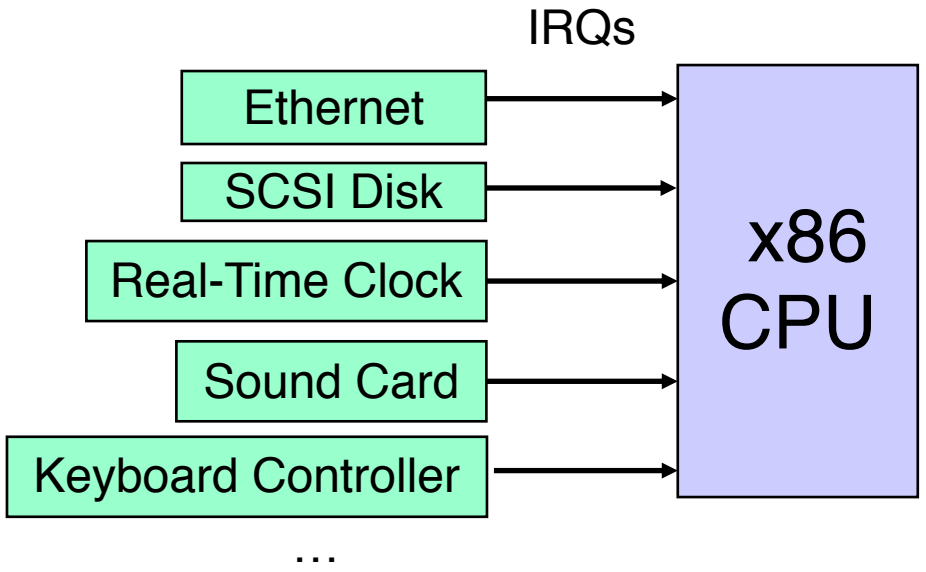
One Solution: Polling

- **CPU periodically checks if each device needs service**
  - takes CPU time when there are no events pending
  - reduce checking frequency → longer response time
  + can be efficient if events arrive rapidly

" Polling is like picking up your phone every few seconds to see if you have a call…"

" Interrupts are like waiting for the phone to ring."

How to Implement Interrupts in Hardware?

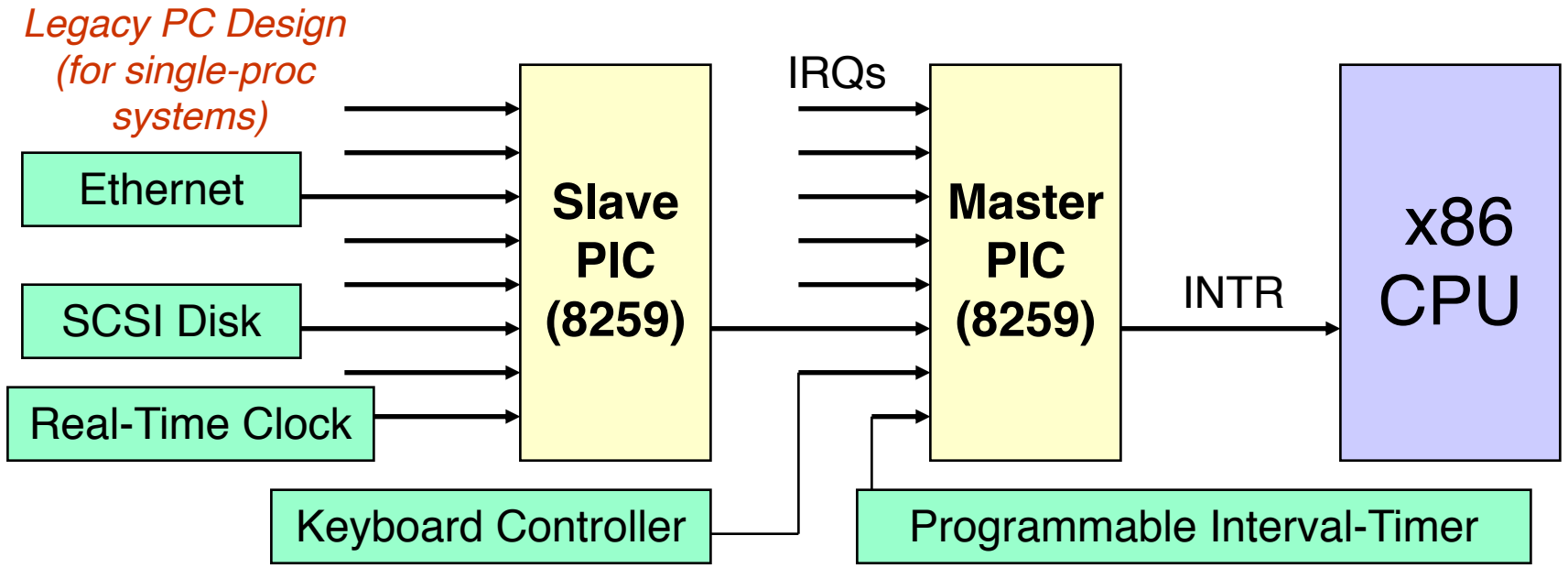


Problem?

- CPU might get interrupted non-stop
- Some device may overwhelm CPU
- Critical interrupts delayed
- Interrupts handling inflexible ("hard-coded")

- **I/O devices wired with *Interrupt Request Lines* (IRQs)**

Improvement: Introducing Controllers



- **I/O devices have (unique or shared) *Interrupt Request Lines* (IRQs)**
- **IRQs are mapped by special hardware to *interrupt vectors*, and passed to the CPU**
- **This hardware is called a *Programmable Interrupt Controller* (PIC)**

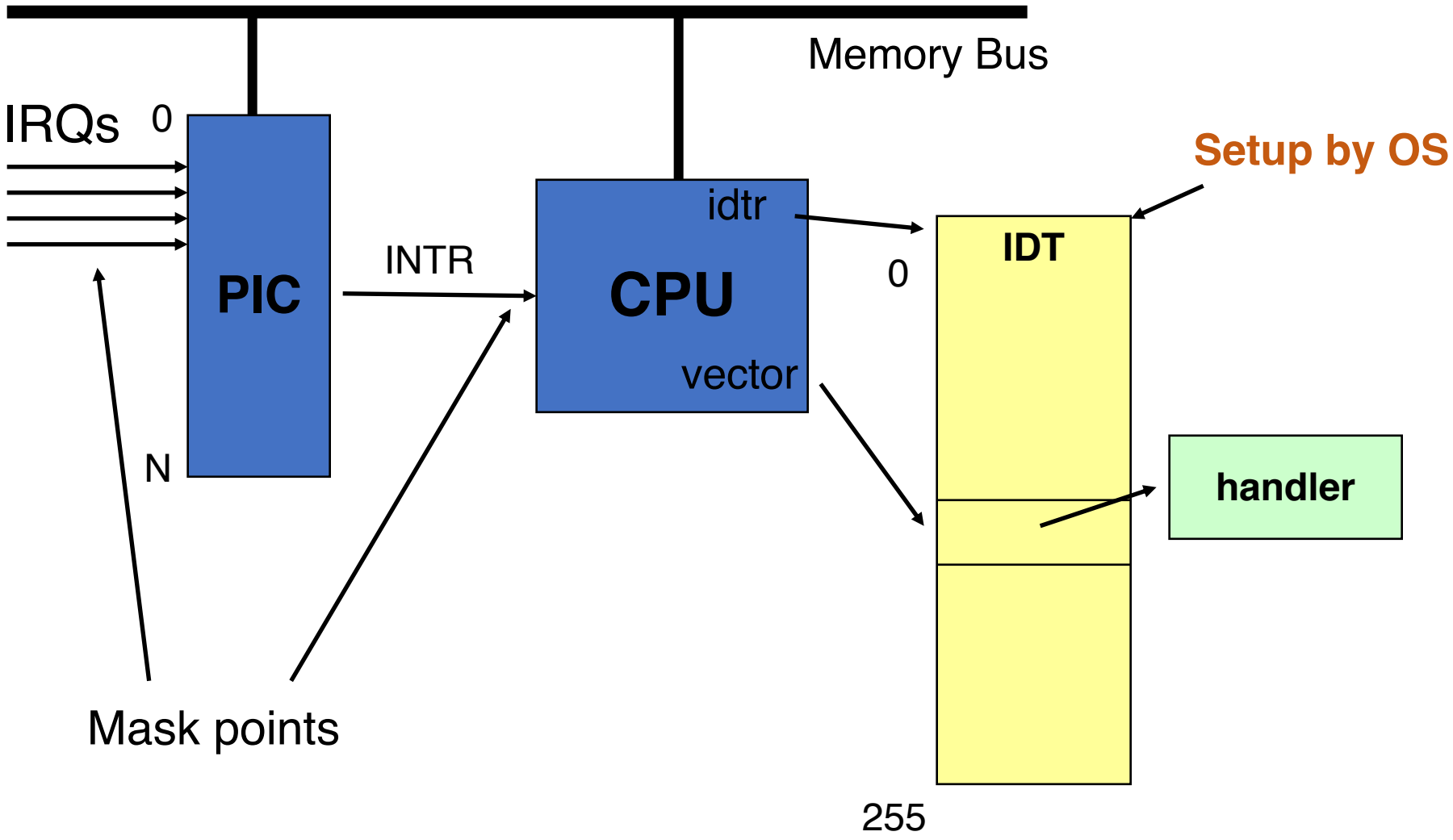
The “Interrupt Controller”

- **PIC: Programmable Interrupt Controller (8259A)**
  - Responsible for telling the CPU when and which device wishes to ‘interrupt’
  - Has 16 wires to devices (IRQ0 – IRQ15)
- **PIC translates IRQs to CPU interrupt *vector* number**
  - Vector number is signaled over INTR line
  - In Pintos:
    - IRQ0...15 are delivered to interrupt vectors 32...47 (`src/threads/interrupt.c`)
- **Interrupts can have varying priorities**
  - PIC also needs to prioritize multiple requests
- **Possible to “mask” (disable) interrupts at PIC or CPU**

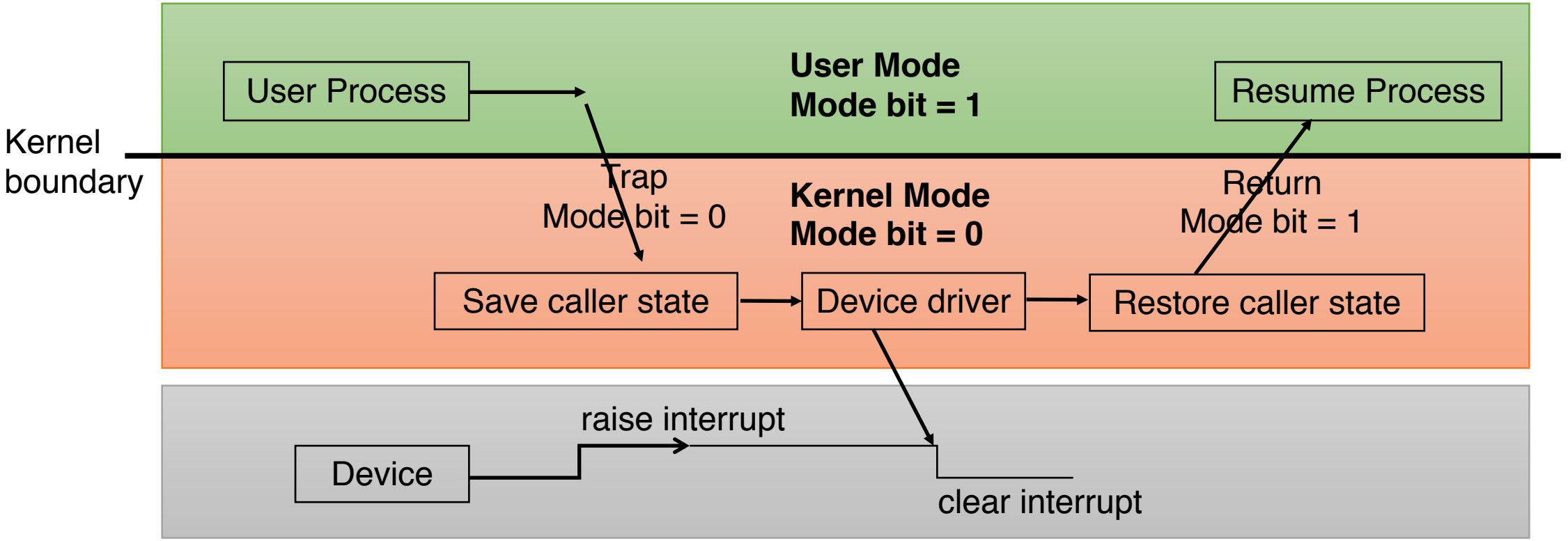
Example: Keyboard & Interrupt

- **When a key is pressed…**
  - Keyboard controller tells PIC to cause an interrupt (how?)
  - Keyboard controller is wired to PIC with IRQ #1 (see figure in slide 27)
  - So IRQ 1 is sent to PIC, which decides if CPU should be notified
  - If so, IRQ 1 will be translated into a vector number to index into CPU's interrupt table
- **What should the OS do?**
  - Setup the CPU interrupt table properly
  - When the interrupt is signaled to CPU, the corresponding handler will be invoked
    - OS talks to the keyboard via IN and OUT instructions
    - OS asks what key was pressed
    - OS does something about it, e.g., prints the key on screen, notifies applications

Putting It All Together

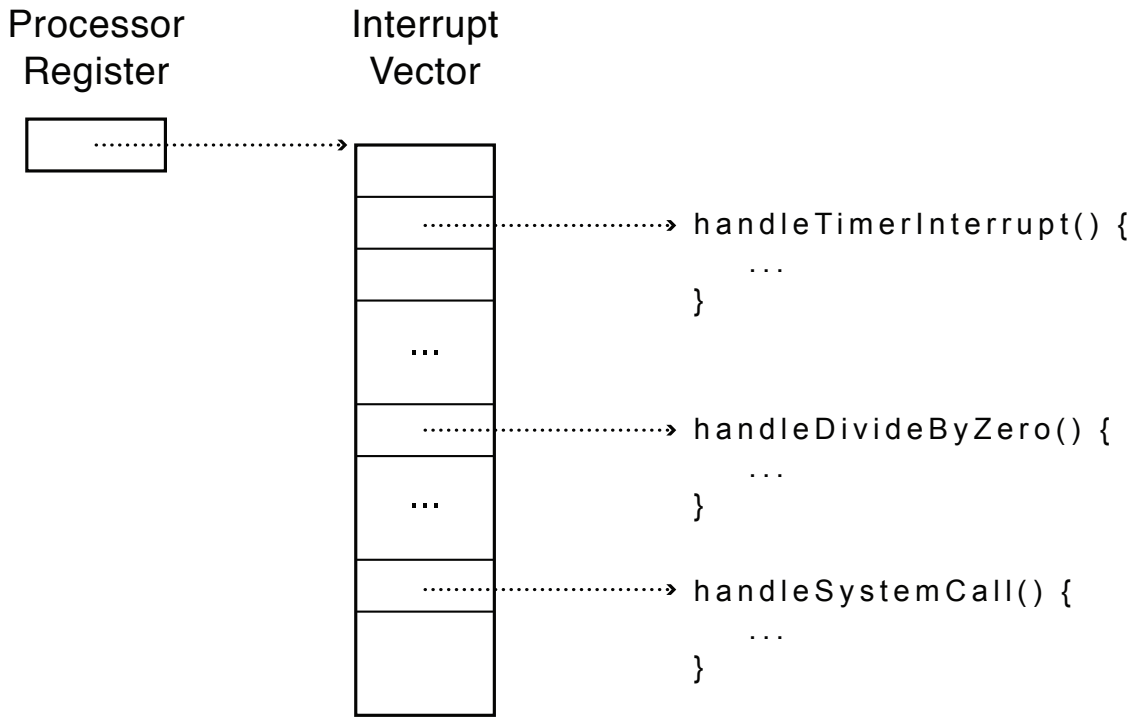


An Interrupt Illustrated



Software Interface: Interrupt Vector Table

- **A data structure to associate interrupt requests with handlers**
  - each entry is called an interrupt vector (specifies the address of the handler)
  - architecture-specific implementation



Software Interface: Interrupt Vector Table

- **A data structure to associate interrupt requests with handlers**
  - each entry is called an interrupt vector (specifies the address of the handler)
  - architecture-specific implementation
- **In x86 called Interrupt Descriptor Table (IDT)**
  - supports 256 interrupts, so the IDT contains 256 entries
  - each entry specifies the address of the handler plus some flags
  - programmed by the OS
    - In Pintos: `make_intr_gate (src/threads/interrupt.c)`

Interrupt Usage Scenario 1: I/O Control

- **I/O issues**
  - Initiating an I/O
  - Completing an I/O
- **Initiating an I/O**
  - Special instructions
  - Memory-mapped I/O
    - Device registers mapped into address space
    - Writing to address sends data to I/O device

I/O Completion

- **Interrupts are the basis for asynchronous I/O**
  - OS initiates I/O
  - Device operates independently of rest of machine
  - Device sends an interrupt signal to CPU when done
  - OS maintains a vector table containing a list of addresses of kernel routines to handle various events
  - CPU looks up kernel address indexed by interrupt number, context switches to routine

I/O Example

1. Ethernet receives packet, writes packet into memory
2. Ethernet signals an interrupt
3. CPU stops current operation, switches to kernel mode, saves machine state (PC, mode, etc.) on kernel stack
4. CPU reads address from vector table indexed by interrupt number, branches to address (Ethernet device driver)
5. Ethernet device driver processes packet (reads descriptors to find packet in memory)
6. Upon completion, restores saved state from stack

Interrupt Usage Scenario 2: Timer

- **The timer is critical for an operating system**
- **It is the fallback mechanism for OS to reclaim control over the machine**
  - Timer is set to generate an interrupt after a period of time
    - Setting timer is a privileged instruction
  - When timer expires, generates an interrupt
  - Handled by kernel, which controls resumption context
    - Basis for OS scheduler *(more later…)*
- **Prevents infinite loops**
  - OS can always regain control from erroneous or malicious programs that try to hog CPU
- **Also used for time-based functions (e.g., *sleep()*)**

Event: Interrupt vs. Exceptions

- **Two kinds of events, interrupts and exceptions**
- **Interrupts are caused by an external event (asynchronous)**
  - Device finishes I/O, timer expires, etc.
- **Exceptions are caused by executing instructions (synchronous)**
  - x86 `int` instruction, page fault, divide by zero, etc.
  - a deliberate exception is a "**trap**", unexpected exception is a "**fault**"
  - CPU requires software intervention to handle a fault or trap

Deliberate Exception: Trap

- **A trap is an intentional software-generated exception**
  - the main mechanism for programs to interact with the OS
  - On x86, programs use the `int` instruction to cause a trap
  - On ARM, `SVC` instruction
- **Handler for trap is defined in interrupt vector table**
  - Kernel chooses one vector for representing system call trap
  - e.g., `int $0x80` is used to in Linux to make system calls
  - Pintos uses `int $0x30` for system call trap

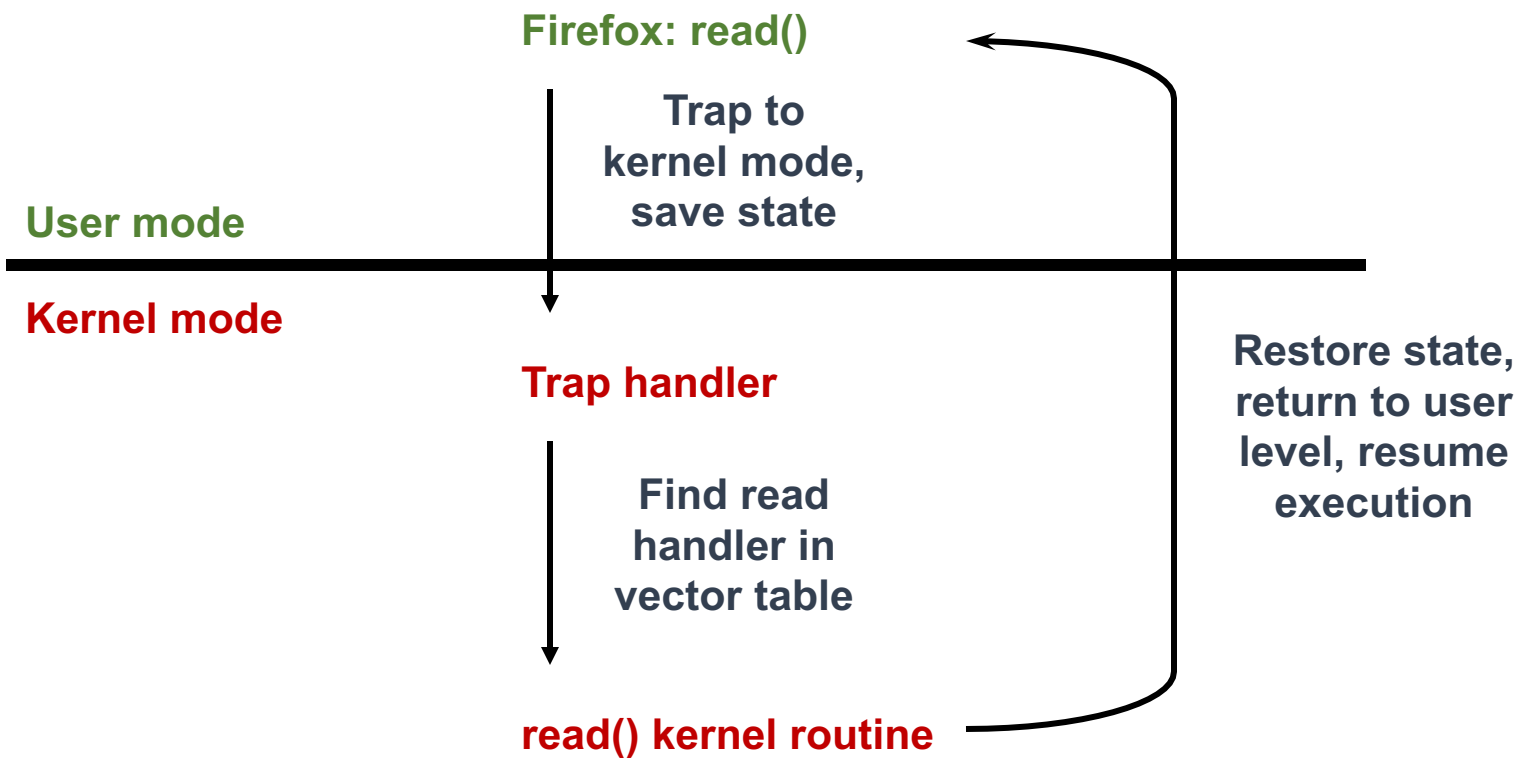
System Call Trap

- **For a user program to "call" OS service**
  - Known as crossing the protection boundary, or protected control transfer
- **The system call instruction**
  - Causes an exception, which vectors to a kernel handler
  - Passes a parameter determining the system routine to call

```
movl $20, %eax # Get PID of current process
int $0x80 # Invoke system call!
# Now %eax holds the PID of the current process
```

  - Saves caller state (PC, regs, mode) so it can be restored
  - Returning from system call restores this state
- **Requires architectural support to:**
  - Restore saved state, reset mode, resume execution

System Call


Firefox: read()
Trap to kernel mode, save state
User mode
Kernel mode
Trap handler
Find read handler in vector table
read() kernel routine
Restore state, return to user level, resume execution

LINUX System Call Quick Reference

Jialong He
Jialong_he@bigfoot.com
http://www.bigfoot.com/~jialong_he

Introduction

System call is the services provided by Linux kernel. In C programming, it often uses functions defined in **libc** which provides a wrapper for many system calls. Manual page section 2 provides more information about system calls. To get an overview, use "man 2 intro" in a command shell.

It is also possible to invoke **syscall()** function directly. Each system call has a function number defined in **<syscall.h>** or **<unistd.h>**. Internally, system call is invokded by software interrupt 0x80 to transfer control to the kernel. System call table is defined in Linux kernel source file "**arch/i386/kernel/entry.S**".

System Call Example

```
#include <syscall.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>

int main(void) {

        long ID1, ID2;
        /*----------------------------*/
        /* direct system call         */
        /* SYS_getpid (func no. is 20) */
        /*----------------------------*/
        ID1 = syscall(SYS_getpid);
        printf ("syscall(SYS_getpid)=%ld\n", ID1);

        /*----------------------------*/
        /* "libc" wrapped system call  */
        /* SYS_getpid (Func No. is 20) */
        /*----------------------------*/
        ID2 = getpid();
        printf ("getpid()=%ld\n", ID2);

        return(0);
}
```

System Call Quick Reference

No	Func Name	Description	Source
1	exit	terminate the current process	*kernel/exit.c*
2	fork	create a child process	*arch/i386/kernel/process.c*
3	read	read from a file descriptor	*fs/read_write.c*
4	write	write to a file descriptor	*fs/read_write.c*
5	open	open a file or device	*fs/open.c*
6	close	close a file descriptor	*fs/open.c*
7	waitpid	wait for process termination	*kernel/exit.c*
8	creat	create a file or device ("man 2 open" for information)	*fs/open.c*
9	link	make a new name for a file	*fs/namei.c*
10	unlink	delete a name and possibly the file it refers to	*fs/namei.c*
11	execve	execute program	*arch/i386/kernel/process.c*
12	chdir	change working directory	*fs/open.c*
13	time	get time in seconds	*kernel/time.c*
14	mknod	create a special or ordinary file	*fs/namei.c*
15	chmod	change permissions of a file	*fs/open.c*
16	lchown	change ownership of a file	*fs/open.c*
18	stat	get file status	*fs/stat.c*
19	lseek	reposition read/write file offset	*fs/read_write.c*
20	getpid	get process identification	*kernel/sched.c*
21	mount	mount filesystems	*fs/super.c*
22	umount	unmount filesystems	*fs/super.c*
23	setuid	set real user ID	*kernel/sys.c*
24	getuid	get real user ID	*kernel/sched.c*
25	stime	set system time and date	*kernel/time.c*
26	ptrace	allows a parent process to control the execution of a child process	*arch/i386/kernel/ptrace.c*
27	alarm	set an alarm clock for delivery of a signal	*kernel/sched.c*
28	fstat	get file status	*fs/stat.c*
29	pause	suspend process until signal	*arch/i386/kernel/sys_i386.c*
30	utime	set file access and modification times	*fs/open.c*
33	access	check user's permissions for a file	*fs/open.c*
34	nice	change process priority	*kernel/sched.c*
36	sync	update the super block	*fs/buffer.c*
37	kill	send signal to a process	*kernel/signal.c*
38	rename	change the name or location of a file	*fs/namei.c*
39	mkdir	create a directory	*fs/namei.c*
40	rmdir	remove a directory	*fs/namei.c*
41	dup	duplicate an open file descriptor	*fs/fcntl.c*
42	pipe	create an interprocess channel	*arch/i386/kernel/sys_i386.c*
43	times	get process times	*kernel/sys.c*
45	brk	change the amount of space allocated for the calling process's data segment	*mm/mmap.c*
46	setgid	set real group ID	*kernel/sys.c*
47	getgid	get real group ID	*kernel/sched.c*
48	sys_signal	ANSI C signal handling	*kernel/signal.c*
49	geteuid	get effective user ID	*kernel/sched.c*
50	getegid	get effective group ID	*kernel/sched.c*

System Call Questions

- **What would happen if the kernel did not save state?**
- **What if the *kernel* executes a system call?**
- **What if a user program returns from a system call?**
- **How to reference kernel objects as arguments or results to/from syscalls?**
  - A naming issue
  - Use integer object handles or descriptors
    - E.g., Unix file descriptors, Windows HANDLEs
    - Only meaningful as parameters to other system calls
  - Also called capabilities (more later when we cover protection)
  - Why not use kernel addresses to name kernel objects?

Unexpected Exception: Faults

- **Hardware detects and reports "exceptional" conditions**
  - Page fault, unaligned access, divide by zero
- **Upon exception, hardware "faults" (verb)**
  - Must save state (PC, regs, mode, etc.) so that the faulting process can be restarted
- **Modern OSes use VM faults for many functions**
  - Debugging, end-of-stack, garbage collection, copy-on-write
- **Fault exceptions are a performance optimization**
  - Could detect faults by inserting extra instructions into code (at a significant performance penalty)

Handling Faults

- **Some faults are handled by "fixing"…**
  - "Fix" the exceptional condition and return to the faulting context
  - Page faults cause the OS to place the missing page into memory
  - Fault handler resets PC of faulting context to re-execute instruction that caused the page fault
- **Some faults are handled by notifying the process**
  - Fault handler changes the saved context to transfer control to a user-mode handler on return from fault
  - Handler must be registered with OS
  - Unix signals or Win user-mode Async Procedure Calls (APCs)
    - SIGALRM, SIGHUP, SIGTERM, SIGSEGV, etc.

Handling Faults (2)

- **Kernel may handle unrecoverable faults by killing the process**
  - Program fault with no registered handler
  - Halt process, write process state to file, destroy process
  - In Unix, the default action for many signals (e.g., SIGSEGV)
- **What about faults in the kernel?**
  - Dereference NULL, divide by zero, undefined instruction
  - These faults considered fatal, operating system crashes
  - Unix panic, Windows “Blue screen of death”
    - Kernel is halted, state dumped to a core file, machine locked up

Types of Arch Support

I. **Manipulating privileged machine state**
   - Protected instructions
   - Manipulate device registers, TLB entries, etc.

II. **Generating and handling "events"**
   - Interrupts, exceptions, system calls, etc.
   - Respond to external events
   - CPU requires software intervention to handle fault or trap

III. **Mechanisms to support synchronization**
   - Interrupt disabling/enabling, atomic instructions

Synchronization

- **Interrupts cause difficult problems**
  - An interrupt can occur at any time
  - A handler can execute that interferes with code that was interrupted
- **OS must be able to synchronize concurrent execution**
- **Need to guarantee that short instruction sequences execute atomically**
  - Disable interrupts – turn off interrupts before sequence, execute sequence, turn interrupts back on
  - Special atomic instructions – read/modify/write a memory address, test and conditionally set a bit based upon previous value
    - `xchg` instruction on x86

Summary

- **Protection**
  - User/kernel modes
  - Protected instructions
- **Interrupts**
  - Timer, I/O

	Unexpected	Deliberate
Exceptions (sync)	fault	syscall trap
Interrupts (async)	interrupt	software interrupt

- **System calls**
  - Used by user-level processes to access OS functions
  - Access what is “in” the OS
- **Exceptions**
  - Unexpected event during execution (e.g., divide by zero)

Next Time…

- **Read Chapters 4-6 (Processes)**
- **Homework #1**
- **Lab 0**