

CS 318 Principles of Operating Systems

Fall 2021

Lecture 2: Architecture Support for OS

Prof. Ryan Huang



JOHNS HOPKINS

WHITING SCHOOL
of ENGINEERING

Administrivia

Lab 0

- Due next Thursday (09/09) noon EST, done **individually**
- Overview session this Friday 4-5 pm (encourage you to start now)

Project groups

- If you don't have a group, search for teammates on Campuswire

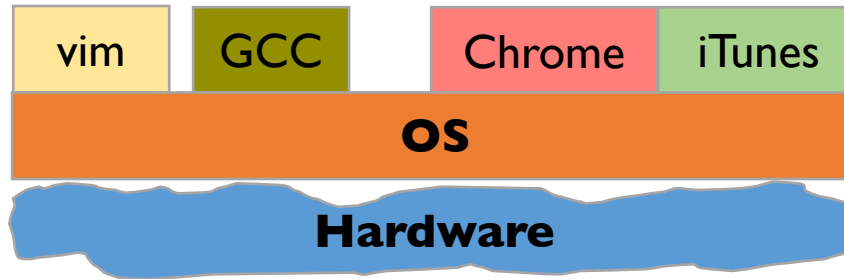
Office hours

- All Zoom links posted on Campuswire

Lecture questions

- Please feel free to interrupt and ask questions anytime during the lecture

Why Start With Hardware?



OS functionality fundamentally depends upon the architectural features

- 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

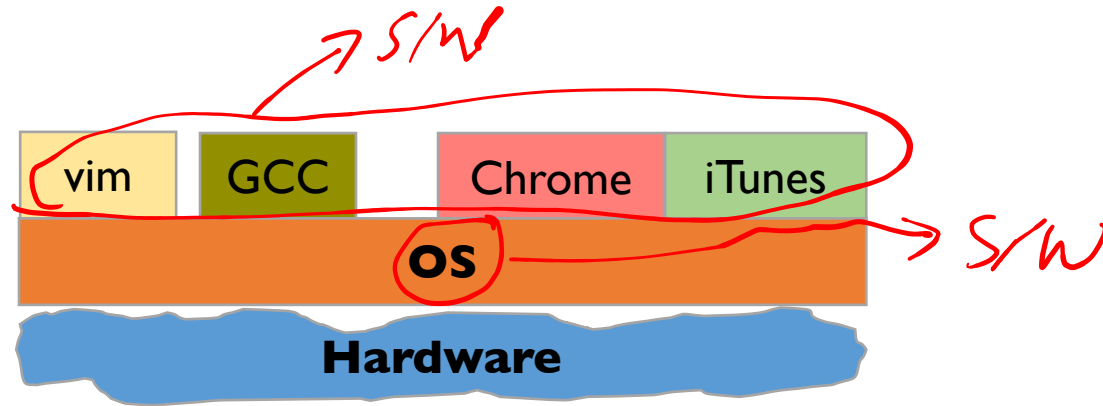
- Early DOS, MacOS lacked virtual memory in part because the h/w did not support it
- Early Sun I computers used two M68000 CPUs to implement virtual memory

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
- I/O control and operation
- Synchronization

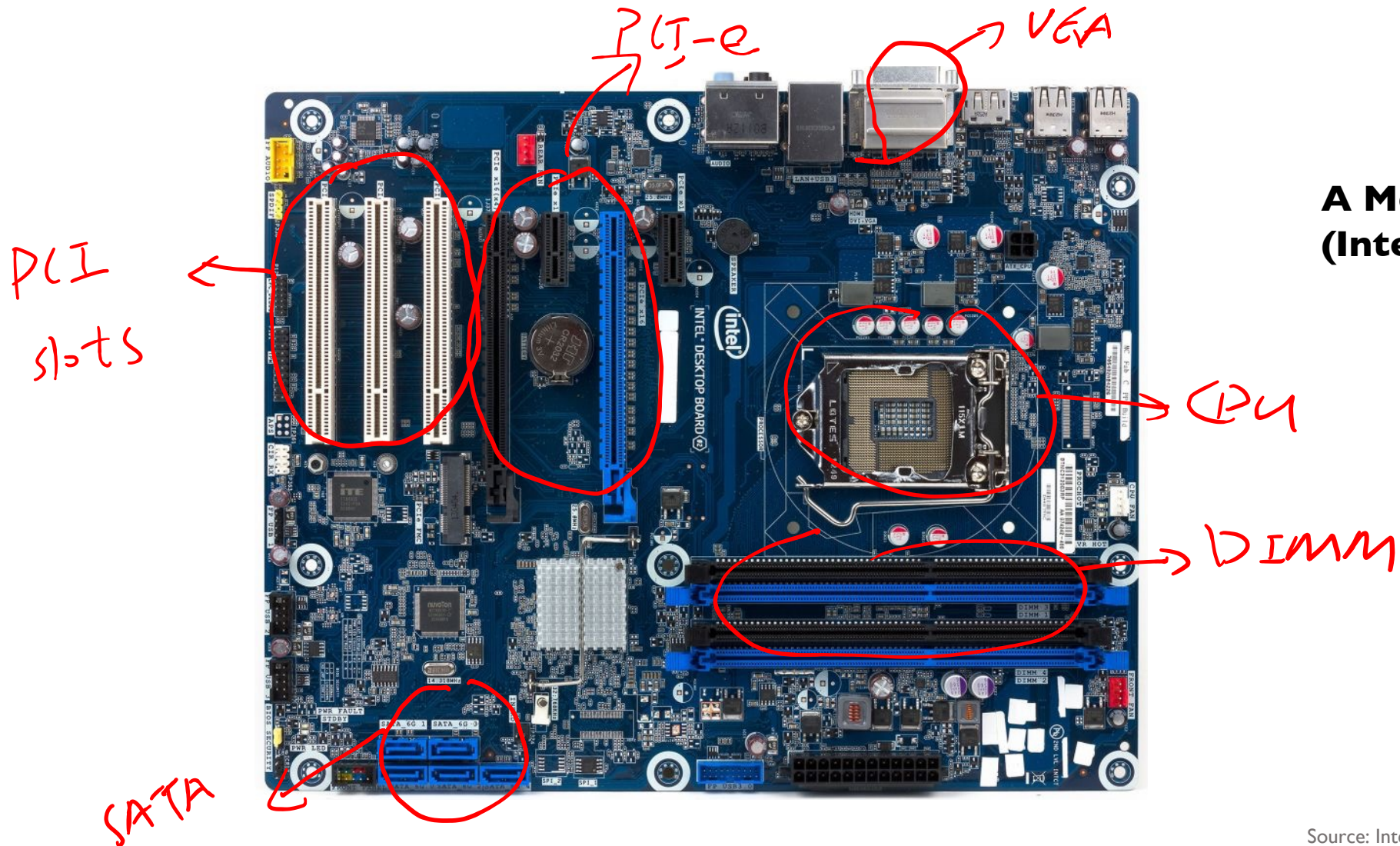
Types of Arch Support



I. Manipulating **privileged** machine state

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

What Is Inside A Computer?



**A Motherboard
(Intel DH87MC)**

Source: Intel® Desktop Board DH87MC
Technical Product Specification

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?

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 ¹

- Limited privileges
- Only those granted by the operating system kernel

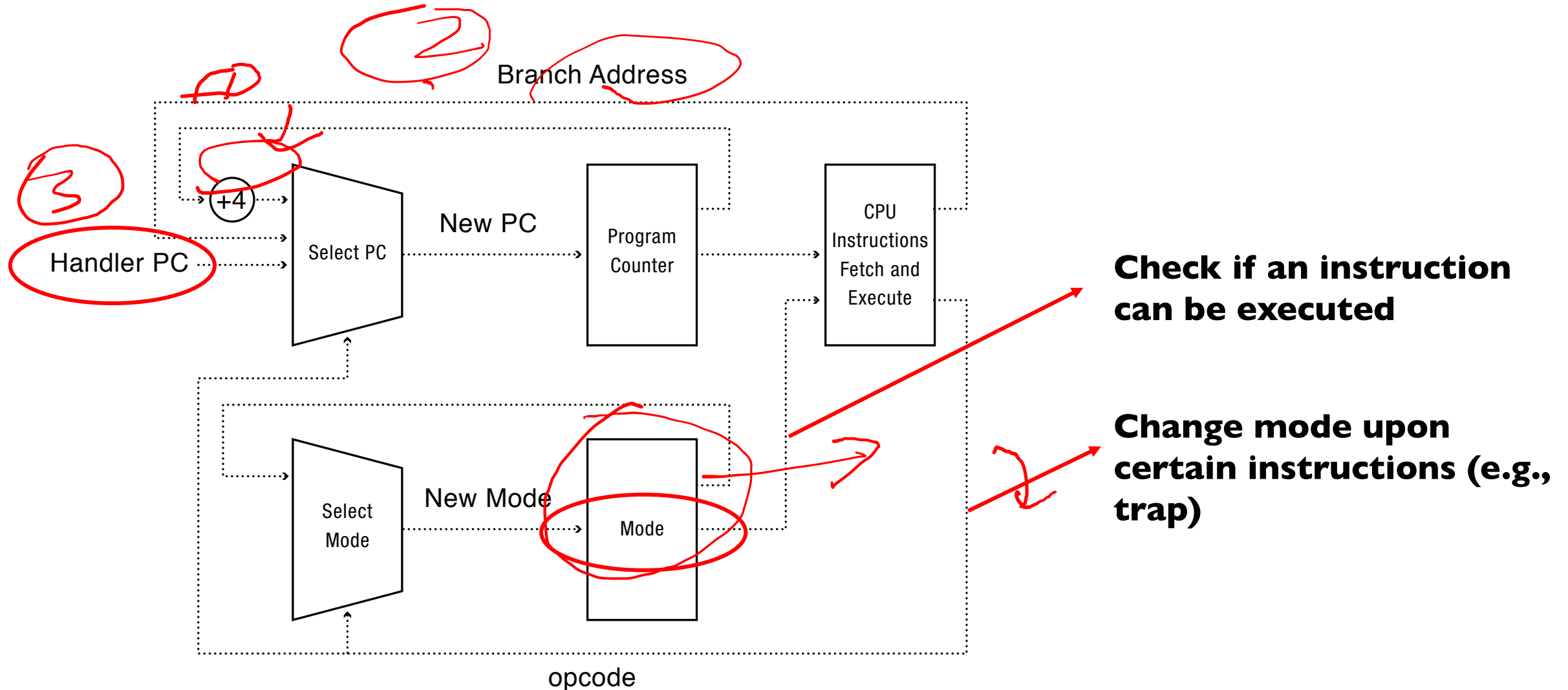
Kernel mode ²

- Execution with the full privileges of the hardware
- Read/write to any memory, access I/O device, read/write disk sector, send/read packet

On the x86, the Current Privilege Level (CPL) in the CS register

On the MIPS, the status register

A CPU with Dual-Mode Operation



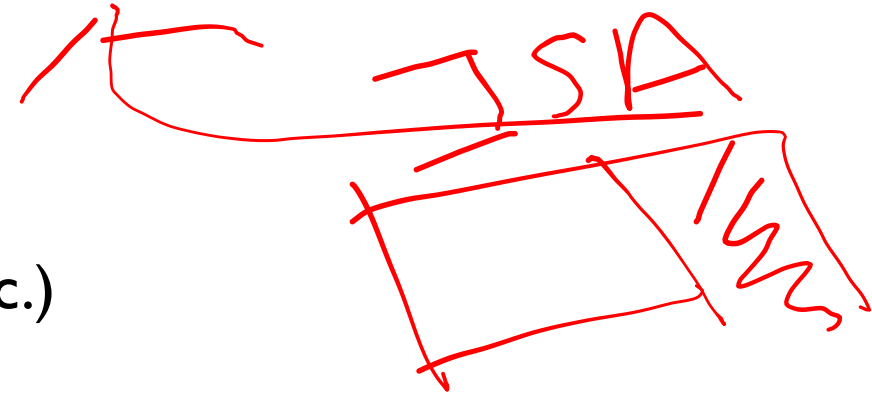
Protected Instructions

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 <i>m</i>	M	Valid	Valid	Invalidate TLB entries for page containing <i>m</i> .

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 (<i>r</i>)	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 invalidate 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.

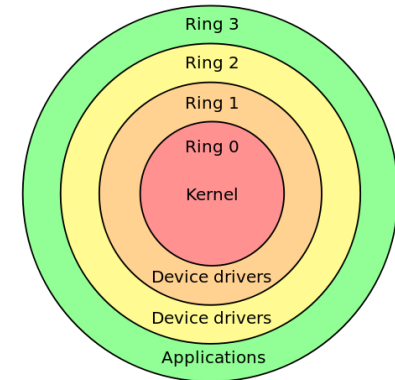
IA-32 Architecture Compatibility

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
- ARMv7 CPUs 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

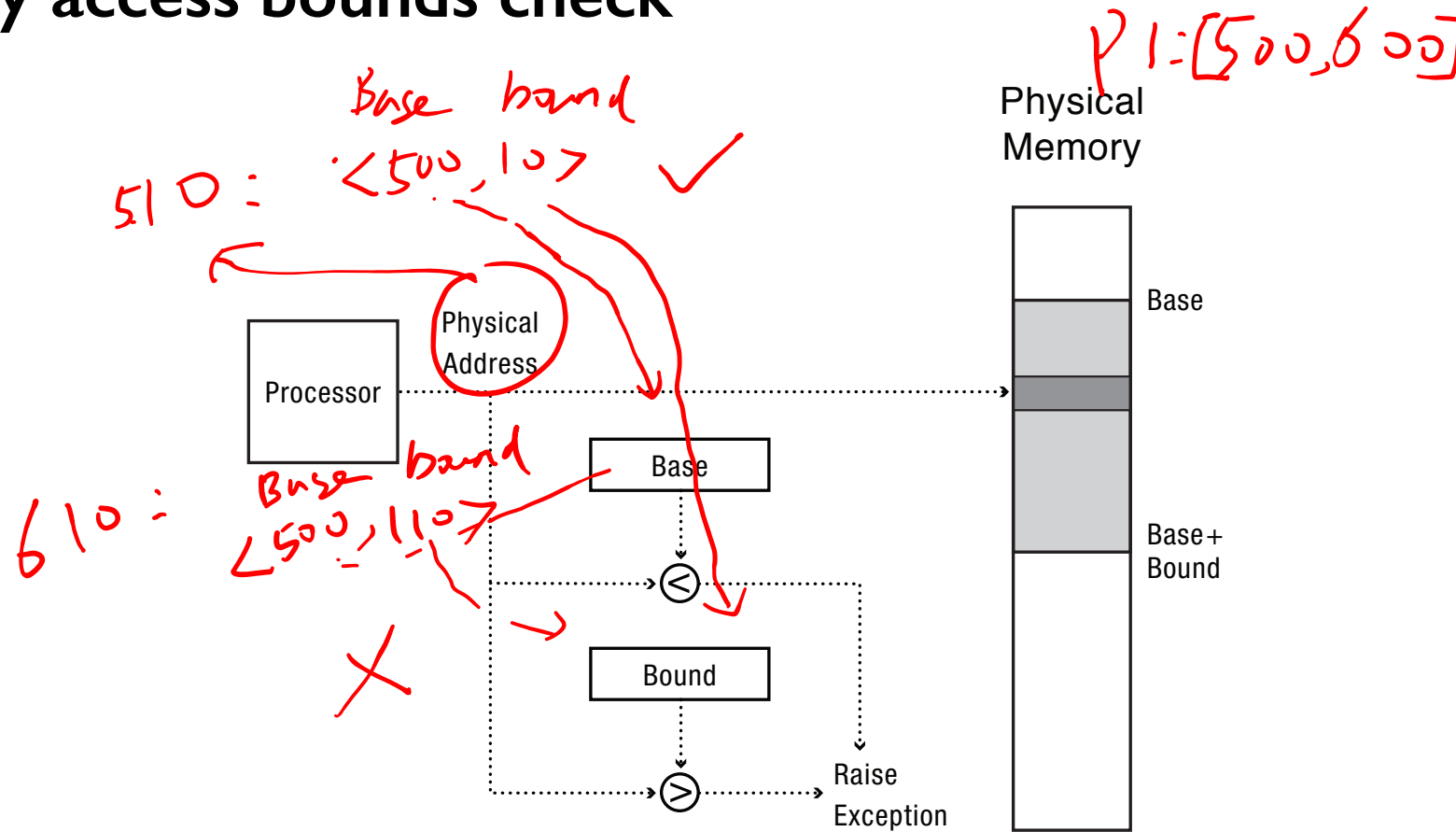
- Raises question of whether programs should trust the OS
- Untrusted operating systems? (Intel SGX)

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

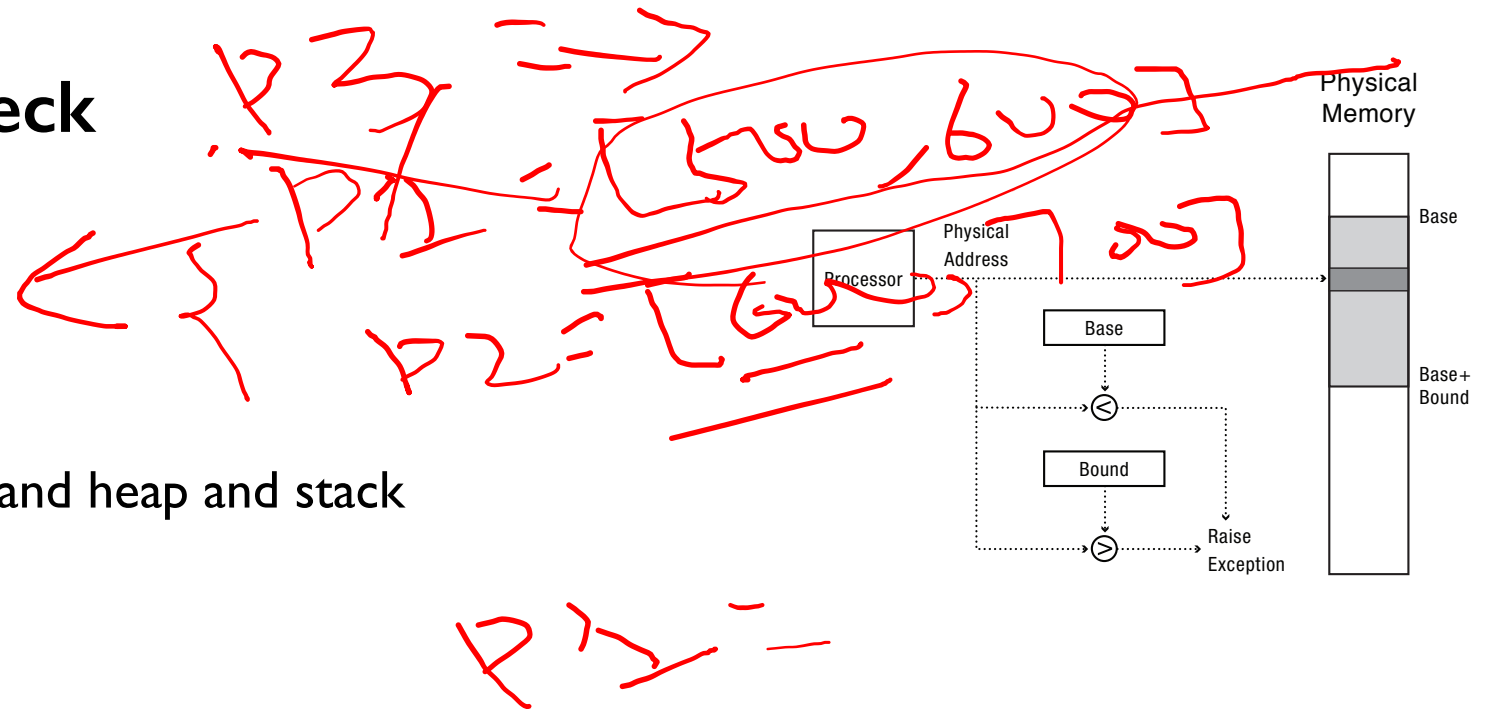


Simple Memory Protection

Memory access bounds check

Problem

- Inflexible:
 - fixed allocation, difficult to expand heap and stack
- 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

[0, 4GB)

The virtual address is translated to physical address

- upon each memory access
- done in hardware (MMU) 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

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

OS Control Flow

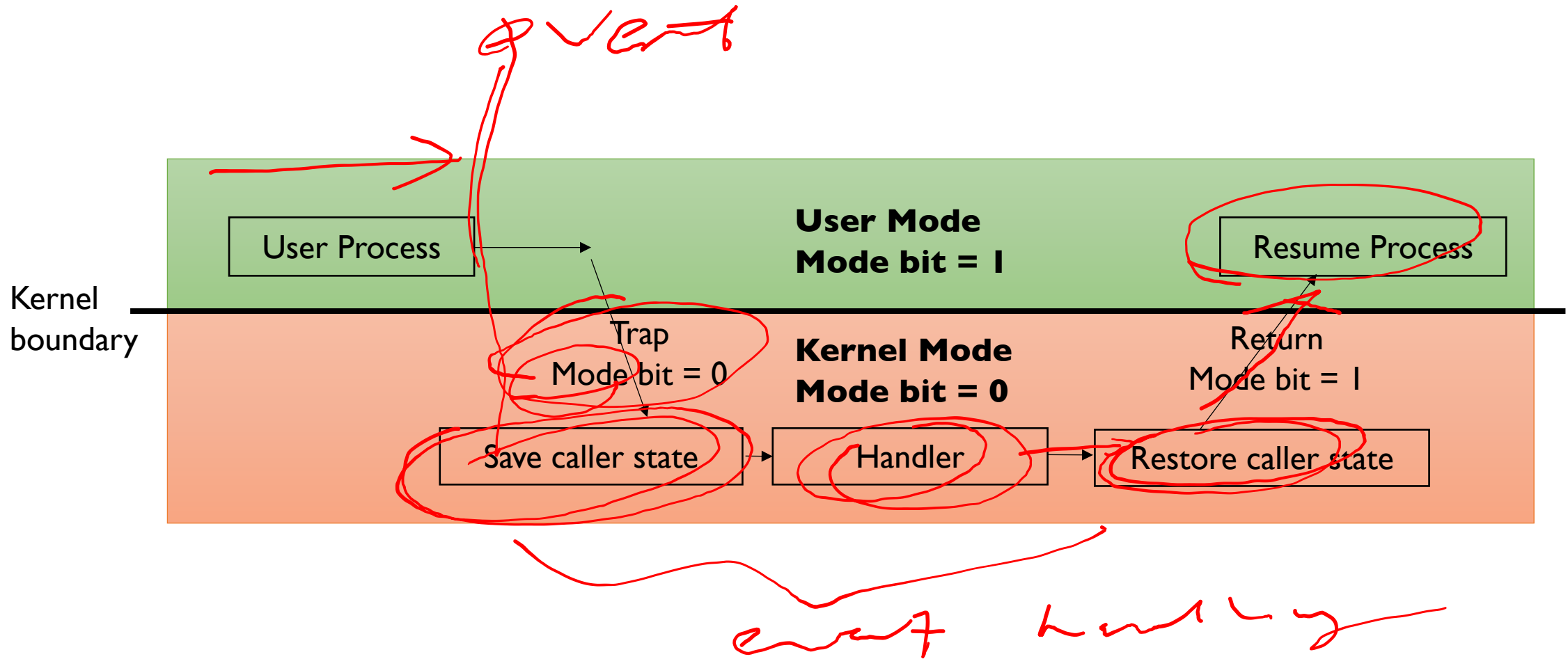
After OS booting, all entry to kernel is a result of some event

- 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 OS functionality, e.g., writing data to disk
- handler **restores program state**, returns to program

Illustration of OS Control Flow



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

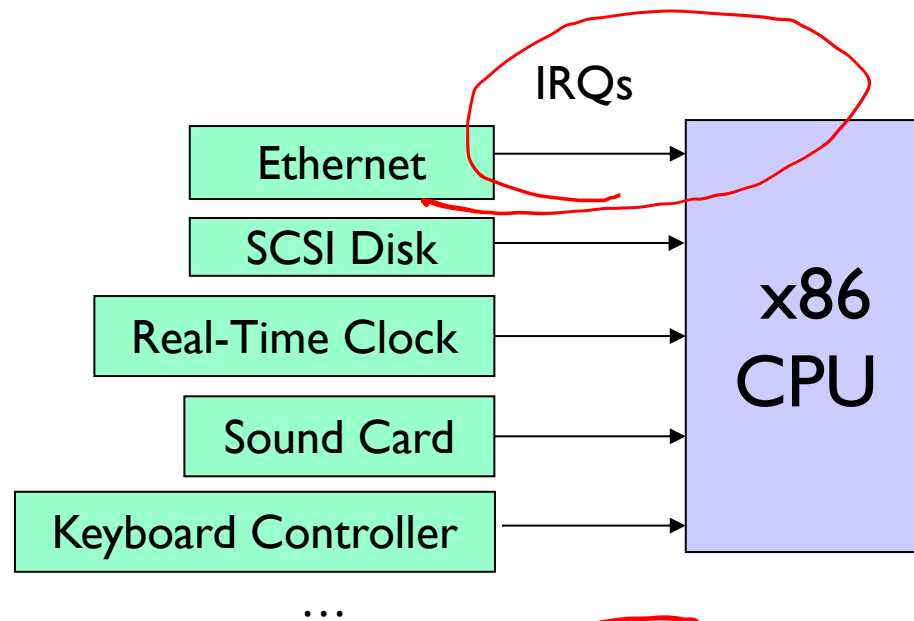
CPU periodically checks if each device needs service

Problem?

- 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...”

Idea: Give Each Device a Wire (Interrupt Line)



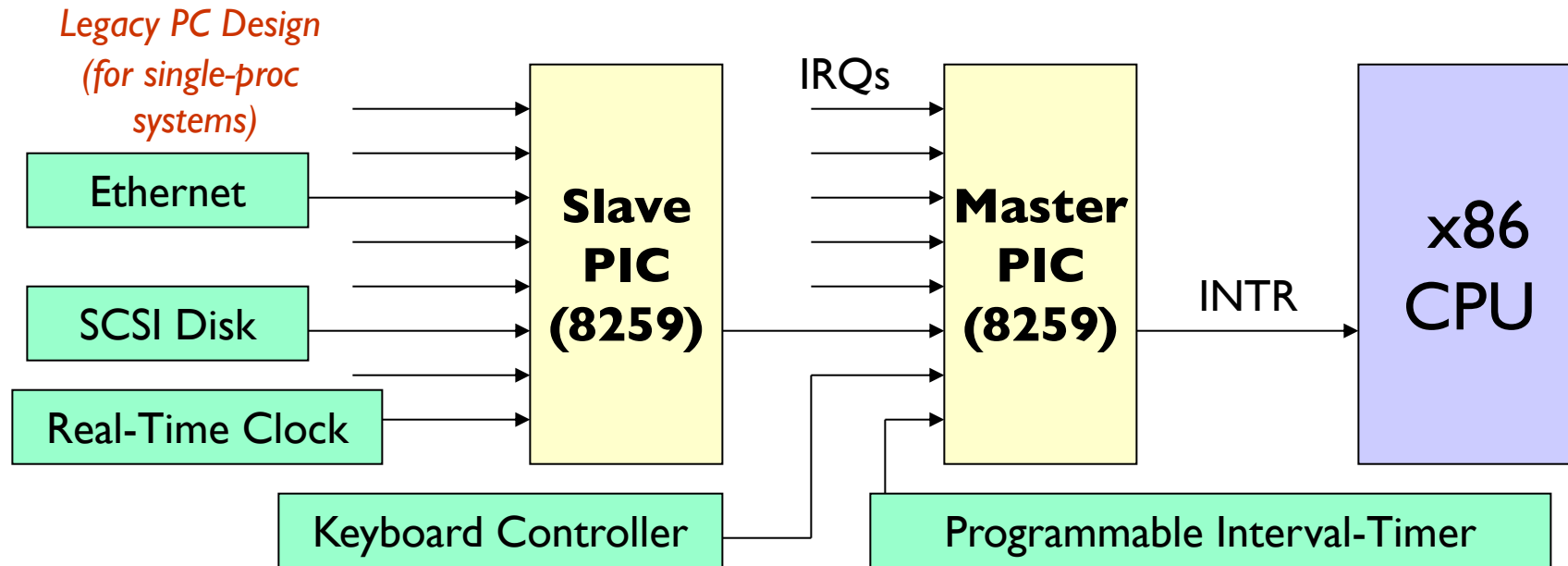
Problem?

- 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)

“Interrupts are like waiting for the phone to ring.”

Improvement: Interrupt Controller



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)**

slide source: Erich Nahum

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 delivered to vector 32...47 ([src/threads/interrupt.c](#))

Interrupts can have varying priorities

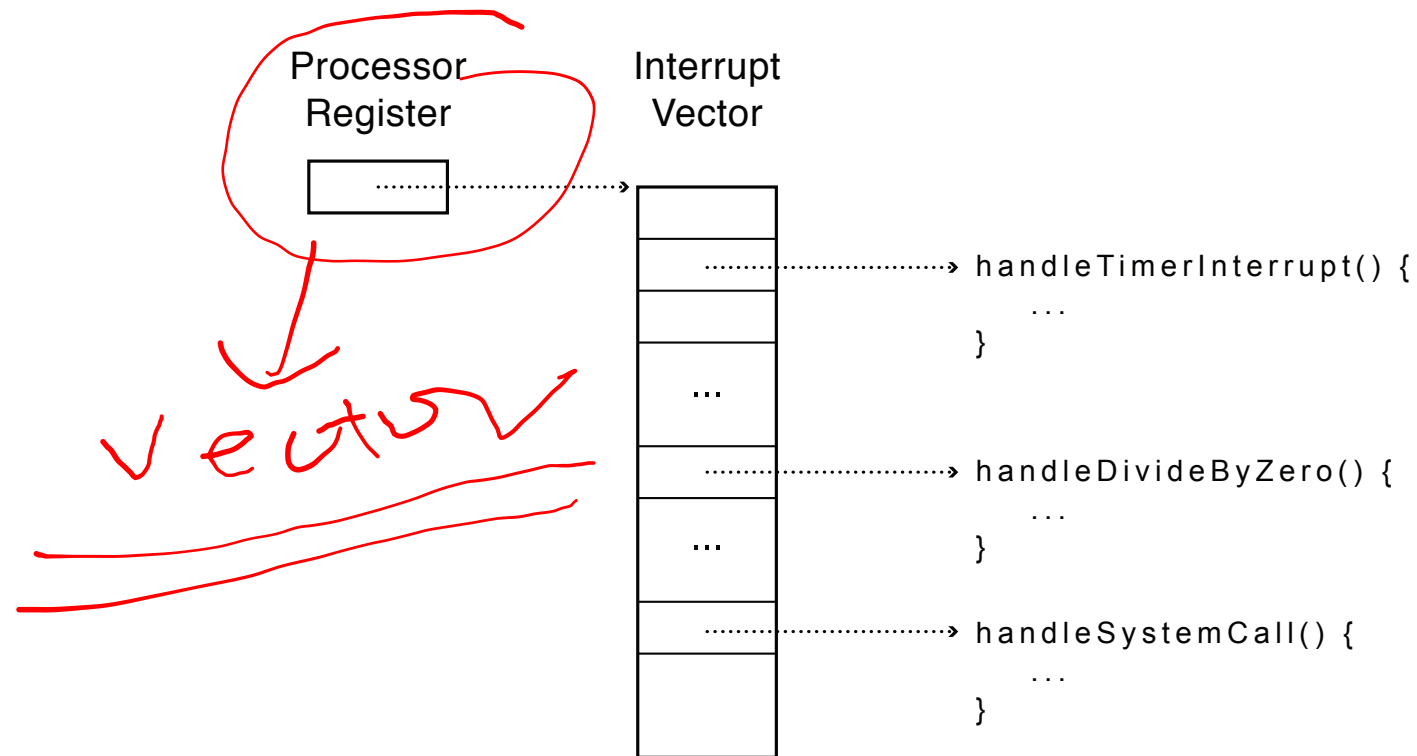
- PIC also needs to prioritize multiple requests

Possible to “mask” (disable) interrupts at PIC or CPU

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 Exception*

↓
INT

Interrupt Usage Scenario : Timer

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()*)

Timer in Pintos

Needed in Pintos Lab I's [Alarm Clock](#) exercise

```
/* Sets up the timer to interrupt TIMER_FREQ times per second,  
and registers the corresponding interrupt. */
```

```
void  
timer_init (void)  
{  
    pit_configure_channel (0, 2, TIMER_FREQ);  
    intr_register_ext (0x20, timer_interrupt, "8254 Timer");  
}
```

```
/* Timer interrupt handler. */
```

```
static void timer_interrupt (struct intr_frame *args UNUSED)  
{  
    ticks++;  
    thread_tick ();  
}
```

```
/* Called by the timer interrupt  
handler at each timer tick. */
```

```
void thread_tick (void)  
{  
    struct thread *t = thread_current ();  
    /* Update statistics. */  
    if (t == idle_thread)  
        idle_ticks++;  
    else  
        kernel_ticks++;  
    /* Enforce preemption. */  
    if (++thread_ticks >= TIME_SLICE)  
        intr_yield_on_return ();  
}
```


Interrupt Usage Scenario 2: I/O Control

I/O issues

- Initiating an I/O
- Completing an I/O

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 an interrupt vector table (IVT)
- CPU looks up IVT by interrupt number, context switches to routine

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

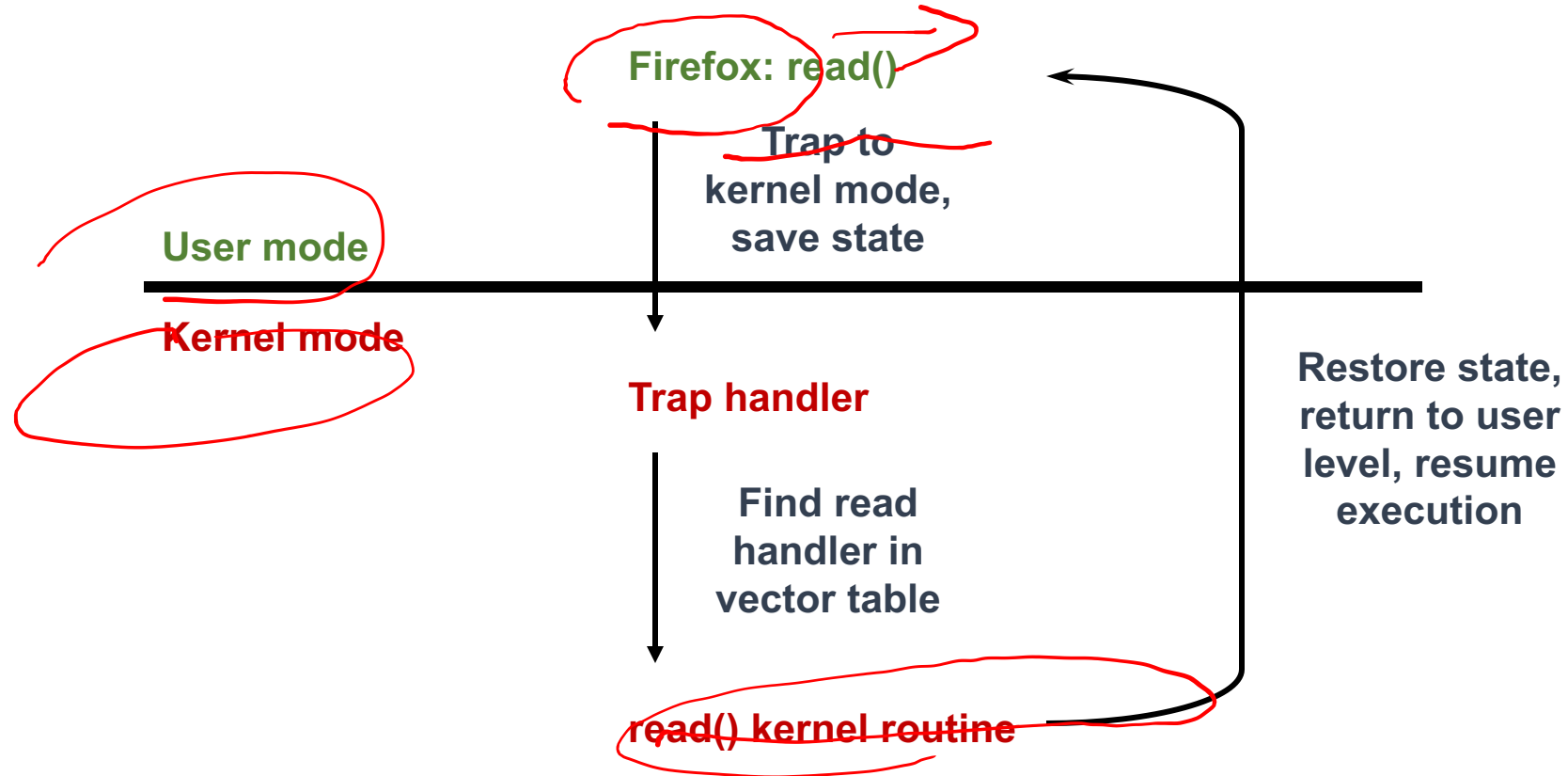
```
movl $20, %eax # Get PID of current process
int $0x80 # Invoke system call!
# Now %eax holds the PID of the current process
```

- 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



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 invoked 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	<code>kernel/exit.c</code>
2	fork	create a child process	<code>arch/i386/kernel/process.c</code>
3	read	read from a file descriptor	<code>fs/read_write.c</code>
4	write	write to a file descriptor	<code>fs/read_write.c</code>
5	open	open a file or device	<code>fs/open.c</code>
6	close	close a file descriptor	<code>fs/open.c</code>
7	waitpid	wait for process termination	<code>kernel/exit.c</code>

8	creat	create a file or device ("man 2 open" for information)	<code>fs/open.c</code>
9	link	make a new name for a file	<code>fs/namei.c</code>
10	unlink	delete a name and possibly the file it refers to	<code>fs/namei.c</code>
11	execve	execute program	<code>arch/i386/kernel/process.c</code>
12	chdir	change working directory	<code>fs/open.c</code>
13	time	get time in seconds	<code>kernel/time.c</code>
14	mknod	create a special or ordinary file	<code>fs/namei.c</code>
15	chmod	change permissions of a file	<code>fs/open.c</code>
16	lchown	change ownership of a file	<code>fs/open.c</code>
18	stat	get file status	<code>fs/stat.c</code>
19	lseek	reposition read/write file offset	<code>fs/read_write.c</code>
20	getpid	get process identification	<code>kernel/sched.c</code>
21	mount	mount filesystems	<code>fs/super.c</code>
22	umount	unmount filesystems	<code>fs/super.c</code>
23	setuid	set real user ID	<code>kernel/sys.c</code>
24	getuid	get real user ID	<code>kernel/sched.c</code>
25	stime	set system time and date	<code>kernel/time.c</code>
26	ptrace	allows a parent process to control the execution of a child process	<code>arch/i386/kernel/ptrace.c</code>
27	alarm	set an alarm clock for delivery of a signal	<code>kernel/sched.c</code>
28	fstat	get file status	<code>fs/stat.c</code>
29	pause	suspend process until signal	<code>arch/i386/kernel/sys_i386.c</code>
30	utime	set file access and modification times	<code>fs/open.c</code>
33	access	check user's permissions for a file	<code>fs/open.c</code>
34	nice	change process priority	<code>kernel/sched.c</code>
36	sync	update the super block	<code>fs/buffer.c</code>
37	kill	send signal to a process	<code>kernel/signal.c</code>
38	rename	change the name or location of a file	<code>fs/namei.c</code>
39	mkdir	create a directory	<code>fs/namei.c</code>
40	rmdir	remove a directory	<code>fs/namei.c</code>
41	dup	duplicate an open file descriptor	<code>fs/fcntl.c</code>
42	pipe	create an interprocess channel	<code>arch/i386/kernel/sys_i386.c</code>
43	times	get process times	<code>kernel/sys.c</code>
45	brk	change the amount of space allocated for the calling process's data segment	<code>mm/mmap.c</code>
46	setgid	set real group ID	<code>kernel/sys.c</code>
47	getgid	get real group ID	<code>kernel/sched.c</code>
48	sys_signal	ANSI C signal handling	<code>kernel/signal.c</code>
49	geteuid	get effective user ID	<code>kernel/sched.c</code>
50	getegid	get effective group ID	<code>kernel/sched.c</code>

System Call Questions

What would happen if the kernel did not save state?

What if the *kernel* executes 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

Faults are not necessarily “bad”

- Modern OSes use virtual memory faults for many functions
 - debugging, end-of-stack, garbage collection, copy-on-write

Fault exceptions are essentially 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 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
- 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)

Lab 0