CE 440 Introduction to Operating System

Lecture 2: Architectural Support
Fall 2025

Prof. Yigong Hu

BOSTON UNIVERSITY

Slides courtesy of Manuel Egele, Ryan Huang and Baris Kasikci

Administrivia

Lab 0

- Done individually, due next Friday (09/19) night
- Overview session on this Friday 2:30 - 4:00 PM, PHO305
- https://www.gradescope.com/courses/1115359

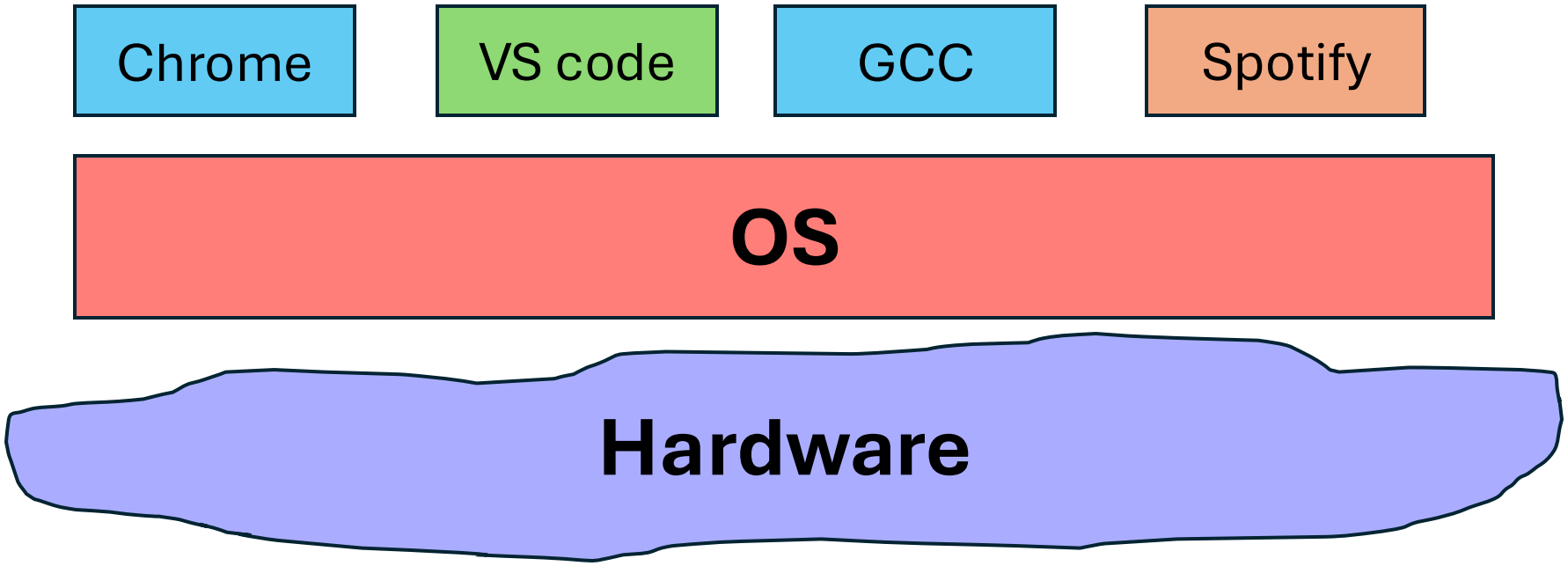
Project groups

- Talk with neighbors in class, a google doc in piazza

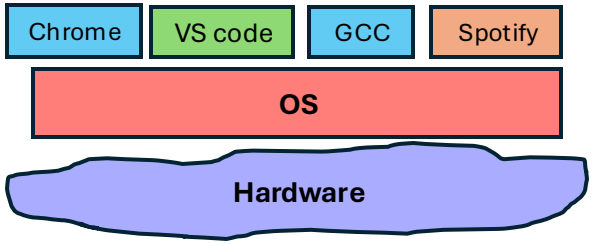
Recap: What is an Operating System

An operating system is

- A software layer between applications and hardware
- “all the code you didn’t write” to implement your application



Recap: OS and Applications



OS is main program

- Calls applications as subroutines
- Illusion: every app runs on its own computer

Provide protection

- Prevent one process messing other process

Provide sharing

- Concurrent execution of multiple programs
- Communication among multiple programs
- Shared implementations of common module like file system

How about 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 systems?

- Use banking application in this system
- Use social media application in this system

Challenge: protection

- How to execute a program with restricted privilege

A Solution

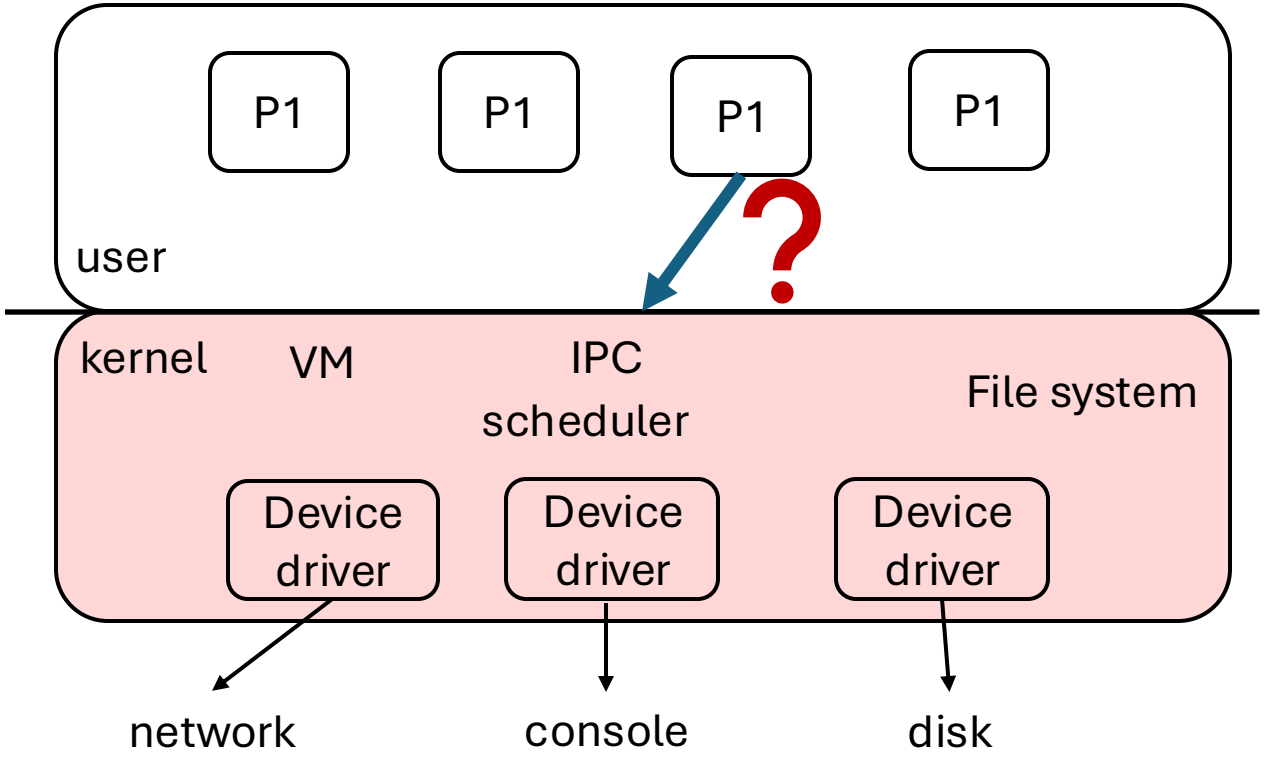
How about we implement execution with limited privilege?

- Execute each program instruction through a simulator(OS)
- If the instruction is permitted, do the instruction
- Otherwise, stop the process
- Slow: additional checking for each instruction

How do we go faster?

- Observation: most instruction are perfectly safe!
- Run the unprivileged code directly on the CPU
- Leave the privileged code to the OS

Typical OS Structure



- Most software runs as user-level processes
- OS kernel runs in privileged mode (shaded)
  - How does application communicate with OS

System Calls

System calls are the interface to operating system services

- Tell OS to do something
- interface of OS

Application can invoke kernel through systems calls

- Special instruction transfers control to kernel
- … which dispatches to one of few hundred syscall handlers

System Call: An Example

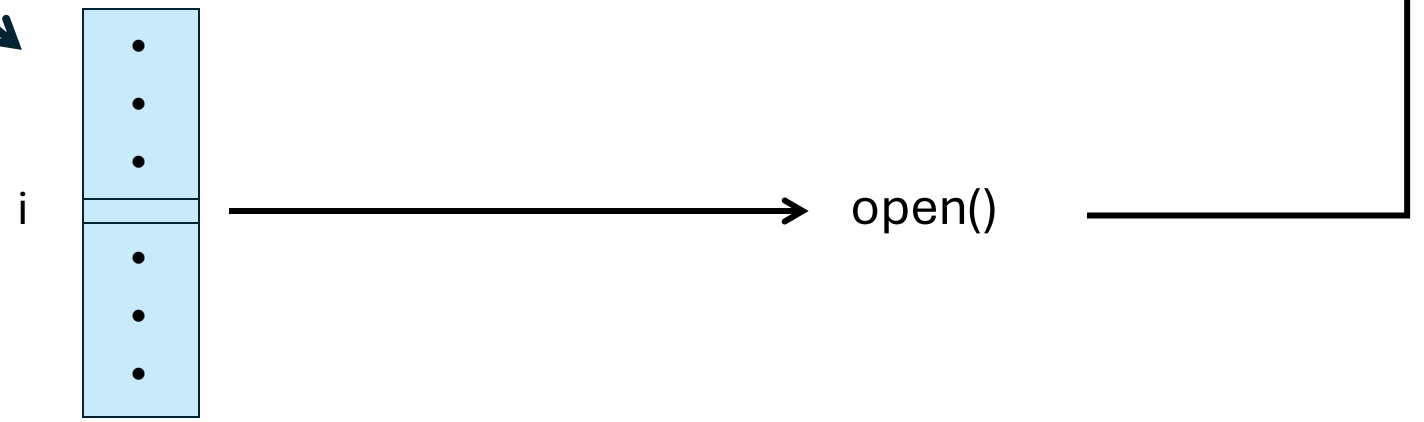
```c
#include <fcntl.h>
#include <unistd.h>

int main() {
    int fd = open("ec440.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd < 0) {
        write(2, "Failed to open EC440.txt\n", 25);
        exit(1);
    }
    write(fd, "Hello, OS!\n", 11);
    close(fd);
    return 0;
}
```

user application

User mode

Kernel mode



More about System Calls

The only way for an application to invoke OS services

Goal:

- Do things application can not do in unprivileged mode
- Like a library call, but into more privileged kernel code

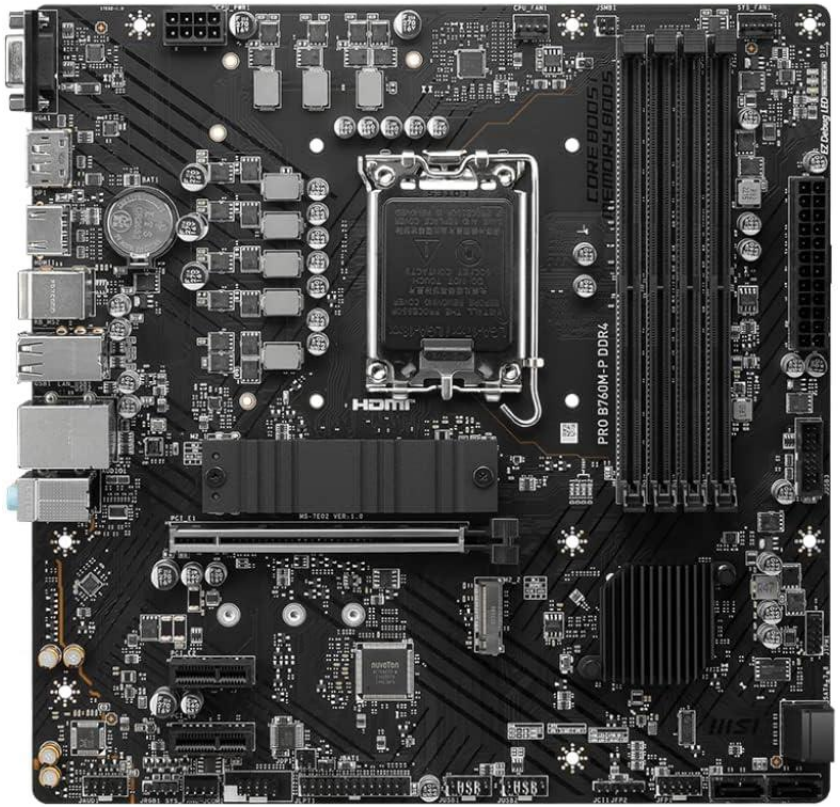
Kernel supplies well-defined system call interface

- Applications set up syscall arguments and trap to kernel
- Kernel performs operation and returns result

How to Manipulate Privileged Machine State?

Hardware support

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



A motherboard
(Intel B760)

H/W Support: Dual-Model Operation in CPU

User mode:

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

Kernel mode:

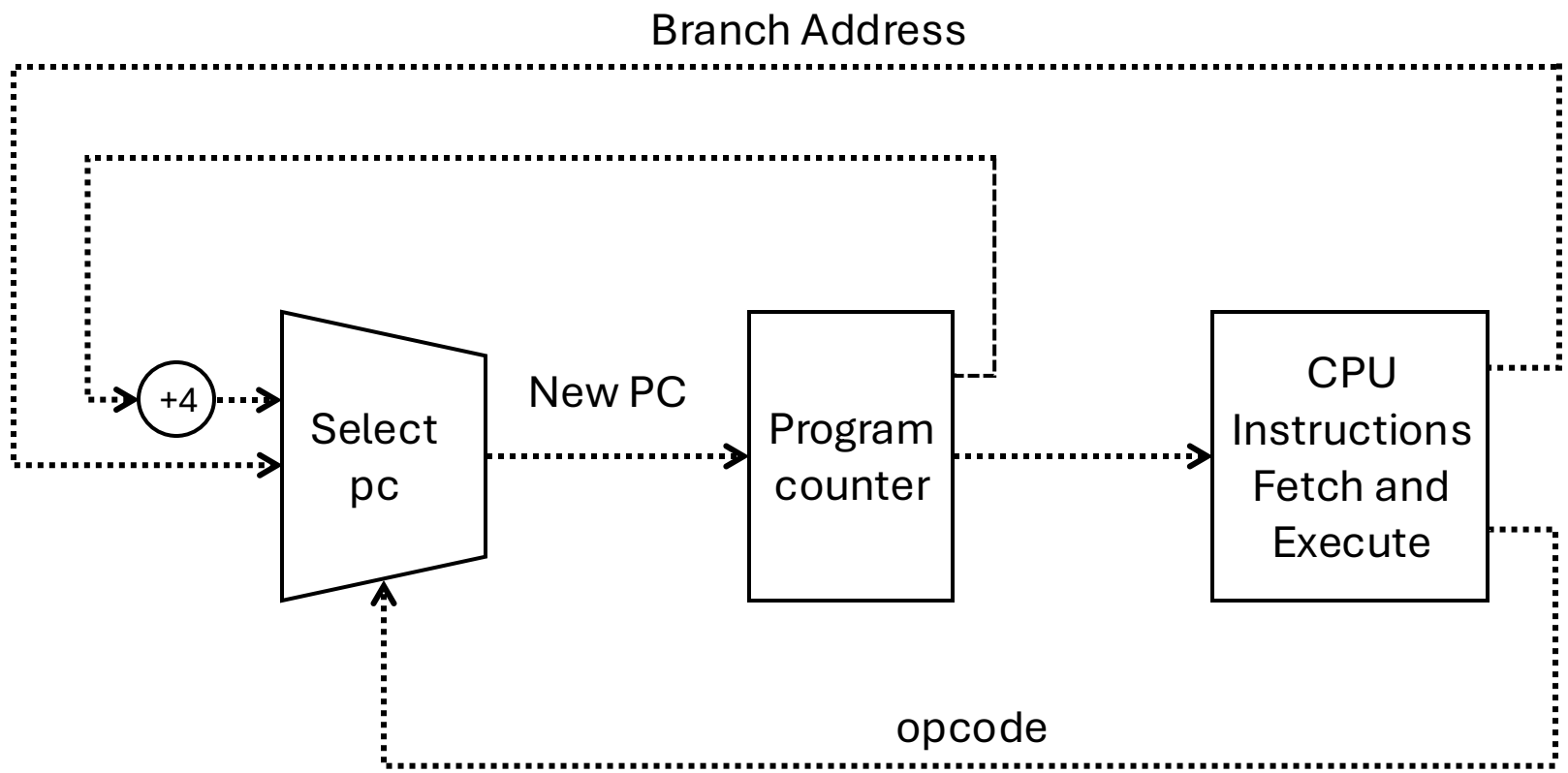
- Execution with the full privileges of the hardware
- Read/write to any memory, access I/O device, read/write disk, send/read packet

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

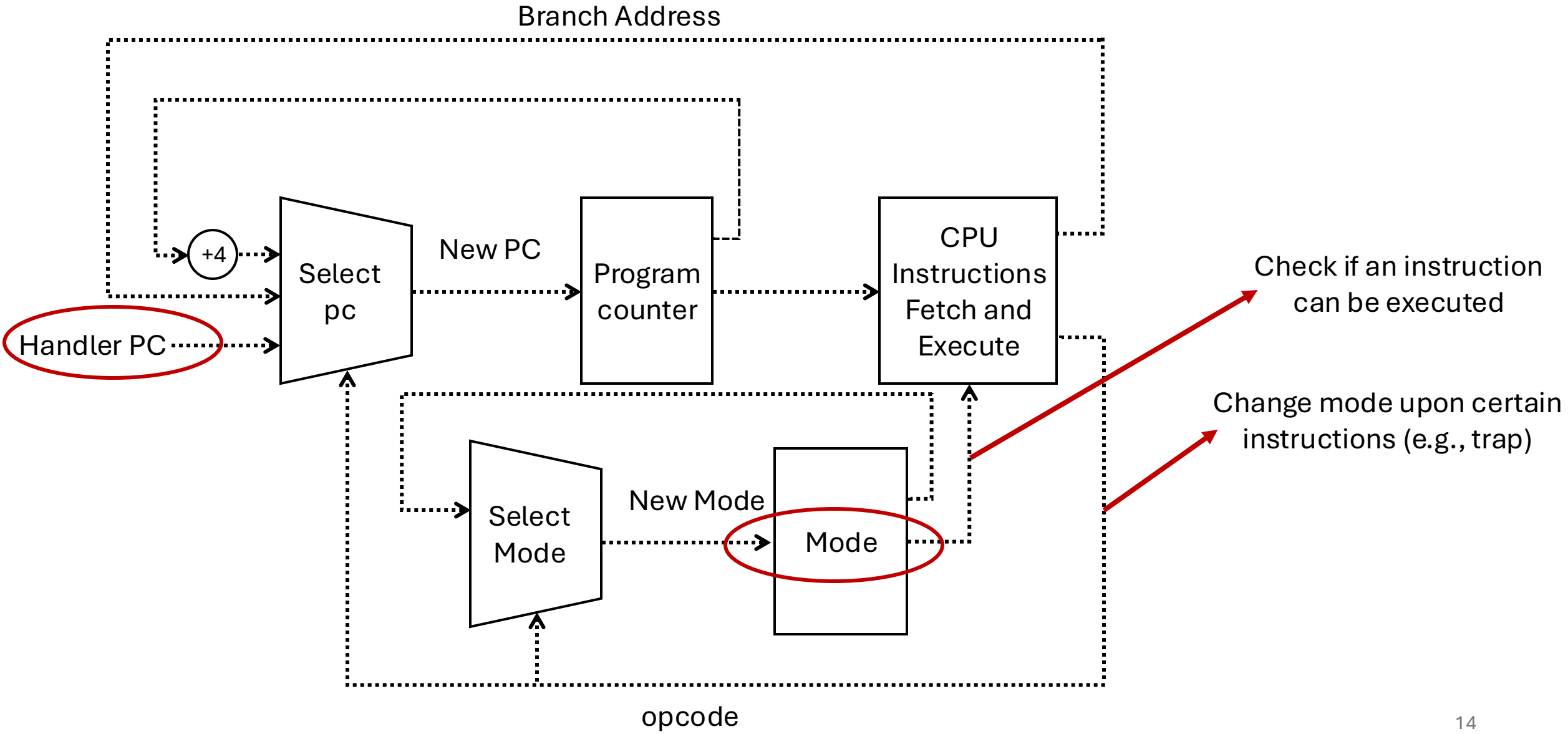
On the MIPS, the status register

A Simple Model of a CPU

Program counter	Instruction stream
4	load
8	add
12	sub
16	cmp
20	jne 36
24	store
28	mov
32	jmp 44
36	call
...	...



A CPU with Dual-Mode Operation



Protected Instruction

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(disk, printers, etc.)
  - Security, fairness(why?)
- Manipulate memory management state
  - Page table pointers, page protection, TBL management, etc.
- Manipulate protected control registers
  - Kernel mode, interrupt level
- Halt instruction (why?)

An Example of Protected Instruction

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.

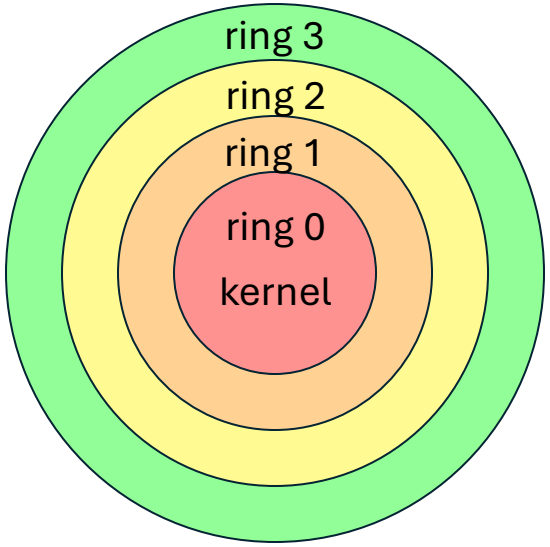
For more details on operations that flush the TLB, see "MOV—Move to/from Control Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B* and Section 4.10.4.1, "Operations that Invalidate TLBs and Paging-Structure Caches," in the *Intel® 64 and IA-32 Architectures 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.

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
- ARMv7 CPUs in modern smartphones have 8 level



Why?

- Protect the OS from itself (software engineering)
- Reserved for vendor, e.g. virtualization

Why Hardware Support?

OS functionality depends on the architectural features

- Key goal of OS: protection and resource sharing
- If done well, applications can be oblivious to HW details

Architectural support can greatly simplify OS tasks

- Early DOS/MacOS lacked virtual memory in part because the hardware did not support it
- Early Sun 1 computers used two M68000 CPUs to implement virtual memory

Architectural Feature for OS

What architectural feature that directly support the OS?

- Protection(kernel/user mode)
- Protected instructions
- Bootstrapping (Lab 0)
- Memory protection

Protection

- System calls
- Interrupts and exceptions
- Timer
- I/O control and operation

Handling ”events”

- Synchronization → **Concurrency**

Architectural Feature for OS

What architectural feature that directly support the OS?

- Protection(kernel/user mode)
- Protected instructions
- Bootstrapping (Lab 0)
- Memory protection

Protection

- System calls
- Interrupts and exceptions
- Timer
- I/O control and operation

Handling ”events”

- Synchronization → **Concurrency**

Memory Protection

What is Memory Protection?

- OS protect programs from each other
- OS protect itself from user programs

Memory management hardware (MMU) provides the mechanisms

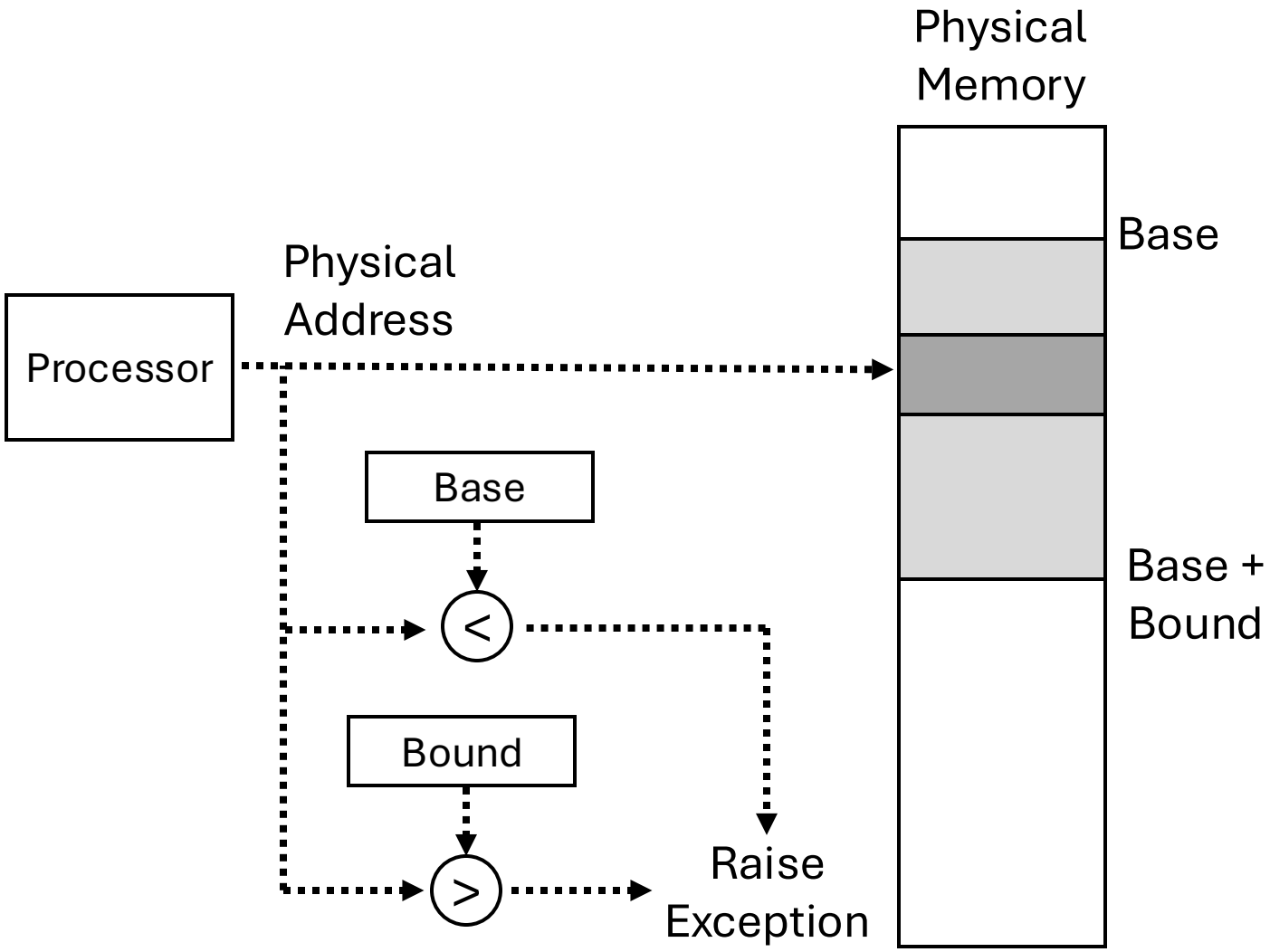
- Based and limit register
- Page table pointers, page protection, segmentation, TLB
- Manipulating the hardware uses protected (privileged) operations

Can we trust the OS?

- May or may not protect user program from OS
- Untrusted operating systems? (Intel SGX)

Simple Memory Protection

Memory access bounds check



Problems?

- Inflexible
  - Fix allocation, difficult to expand heap and stack
- Inconvenient
  - Require changes to mem instruction each time the program is loaded
- Fragmentation
  - Many “ holes” of memory that are free but cannot be used

Solution: 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(MMU) using a table
- Table setup by the OS

Types of Arch Support

What architectural feature that directly support the OS?

- Protection(kernel/user mode)
- Protected instructions
- Bootstrapping (Lab 0)
- Memory protection

Protection

- System calls
- Interrupts and exceptions
- Timer
- I/O control and operation

Handling ”events”

- Synchronization

Events

An Event is an “unnatural” change in OS execution

- Event immediately stop current execution
- Changes mode, context (machine state)

The kernel (OS) 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 on big event handler

OS Execution

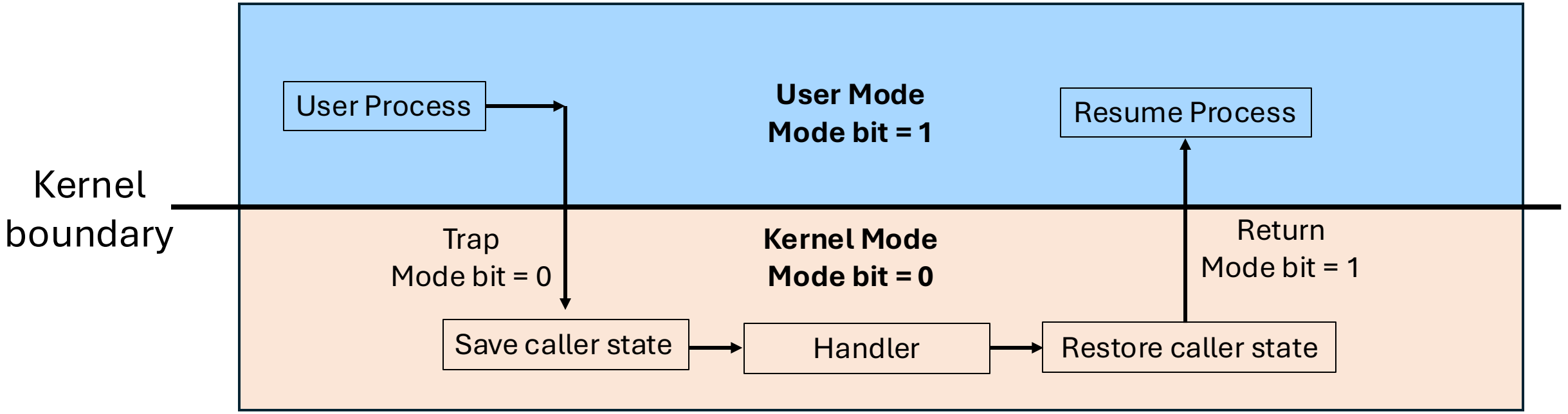
After OS booting, all entry to kernel is a result of some event

- Event immediately stops current execution
- Changes mode to kernel mode
- Invoke a piece

Architectural support can greatly simplify OS tasks

- Early DOS/MacOS lacked virtual memory in part because the hardware did not support it
- Early Sun 1 computers used two M68000 CPUs to implement virtual memory

Workflow of OS Execution 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 event

- Indicates some device needs services
- I/O hardware interrupts
- Software and hardware timers

Challenges of realizing interrupts

- A computer is more than CPU
  - Keyboard, disk, printer, camera, etc.
- OS can not predict when the signal will be sent by these devices

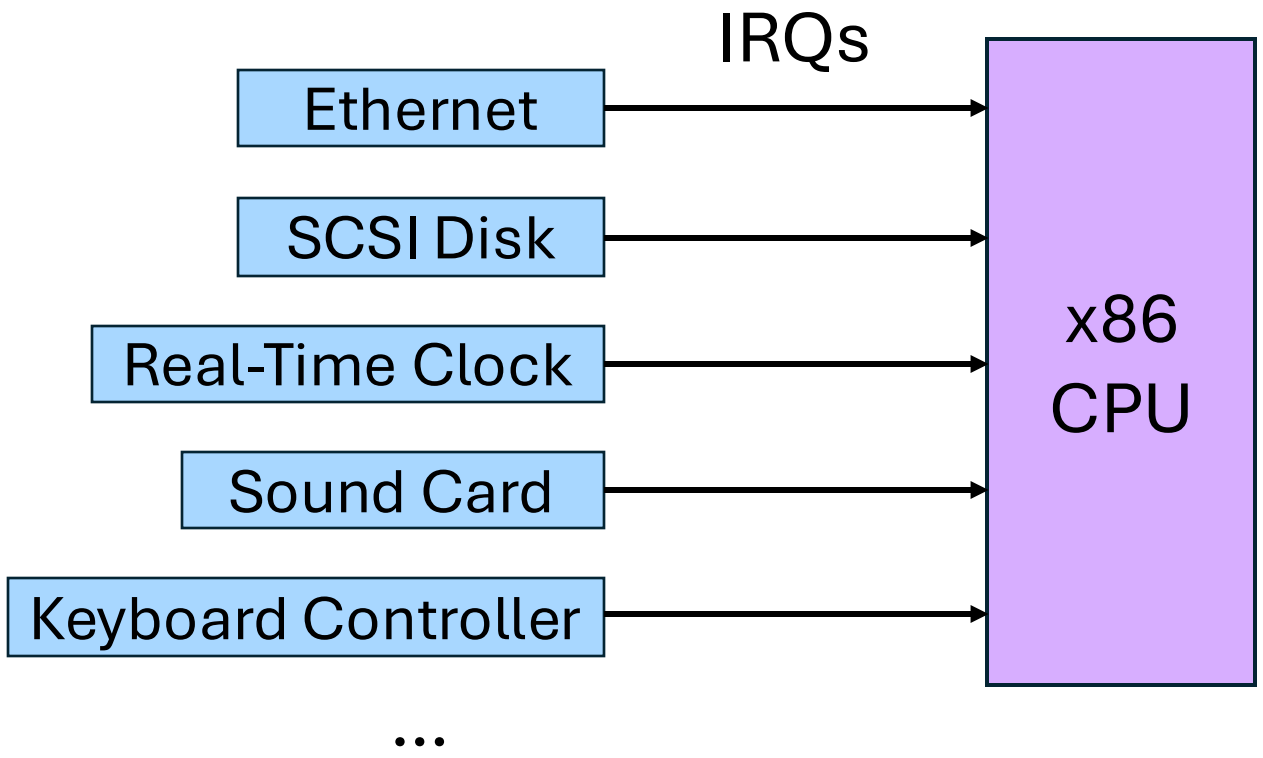
How about Polling?

CPU periodically checks if each device needs service

- \+ Easy to implement
- \+ Can be efficient if events arrive rapidly
- \- Takes CPU time when there are no events pending
- \- Reduce checking frequency → longer response time

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

Give Each Device a Wire



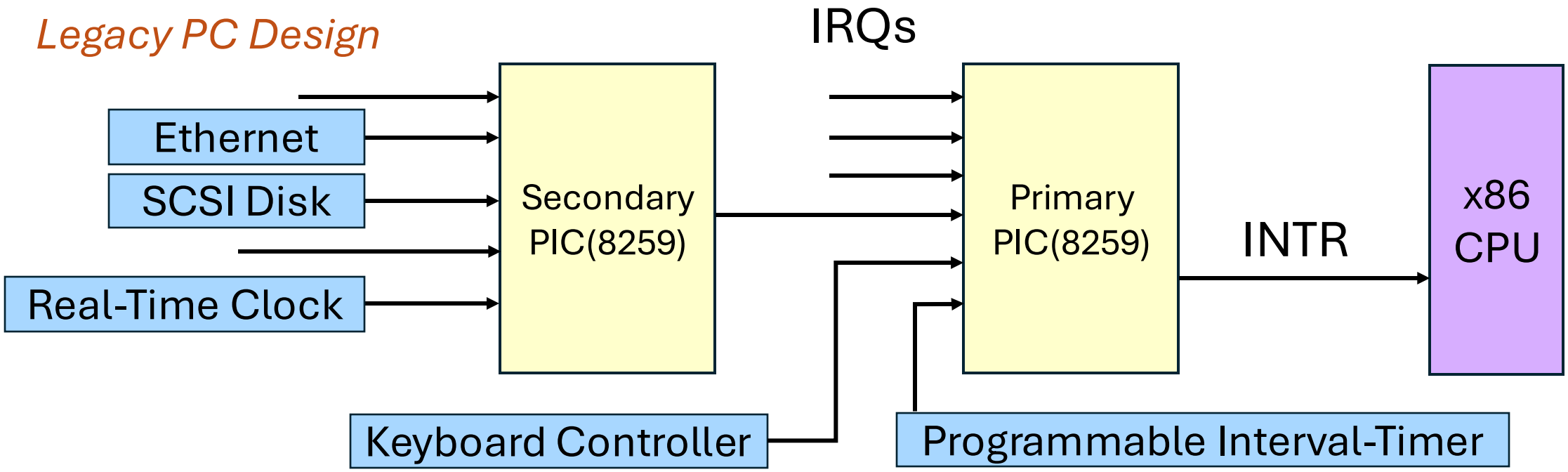
Problems?

- CPU might get interrupted no-stop
- Some device may overwhelm CPU
- Critical interrupt delayed
- Interrupts handling inflexible ("hard-coded")

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

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

A Better Solution: Interrupt Controller



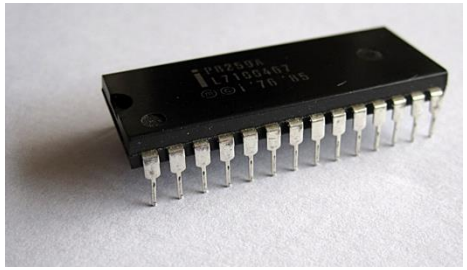
This hardware is called a *Programmable Interrupt Controller* (PIC)

- 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

The Interrupt Controller

PIC: Programmable Interrupt Controller (8259A)

- 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 delivers to vector 32-47 (src/threads/interrupt.c)

Interrupts can have varying priorities

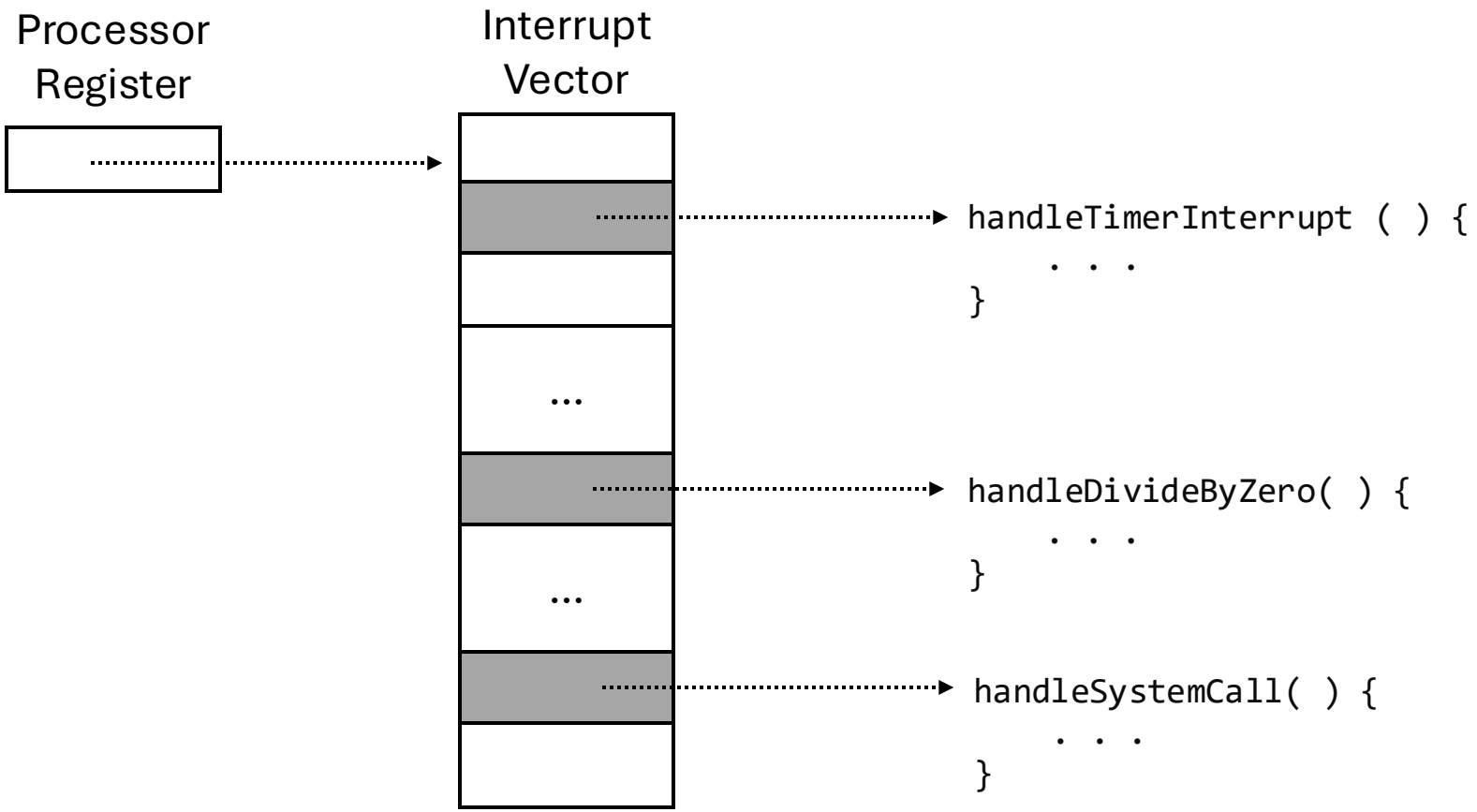
- PIC also needs to prioritize multiple requests

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 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

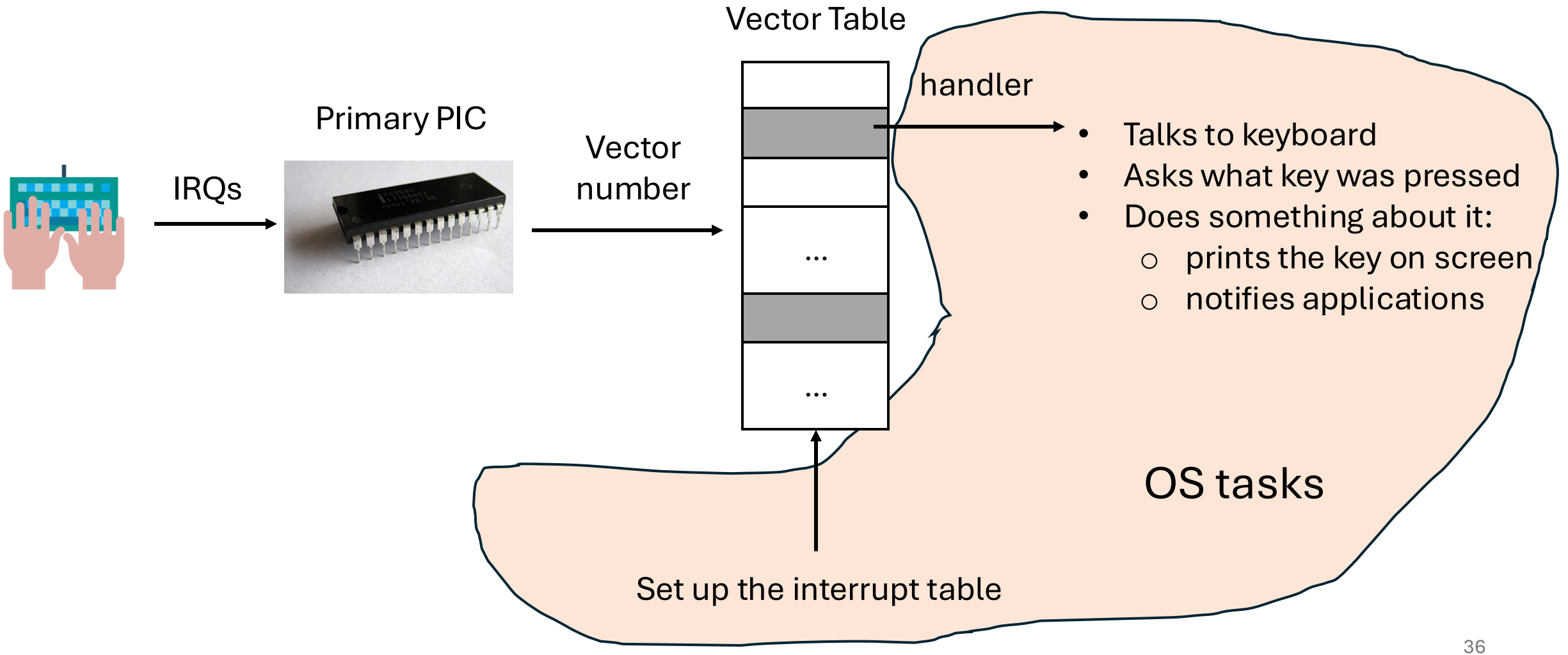
- Each entry is an interrupt vector (specifies the address of the handler)
- Architecture-specific implementation

In x86 called Interrupt Descriptor Table (IDT)

- Support 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)

Example: Press a Keyboard

When a key is pressed..



Interrupt Use Case 1: Timers

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, generate an interrupt
- Handled by kernel, which controls resumption context
  - Basis for OS scheduler (more later ..)

Prevents infinite loops

- OS can always regain control from buggy or malicious program that try to hog CPU

Also Used for time-based functions (e.g. sleep())

Timer in Pintos

Needed in Pintos Lab1's Alarm Clock exercise

```c
/* 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 ();
}
```

```c
/* 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 Use Case 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 for rest of machine
- Device sends an interrupt signal to CPU when done
- OS maintains an interrupt vector tables (IVT)
- CPU looks up IVT by interrupt number, context switches to rountine

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”, while 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

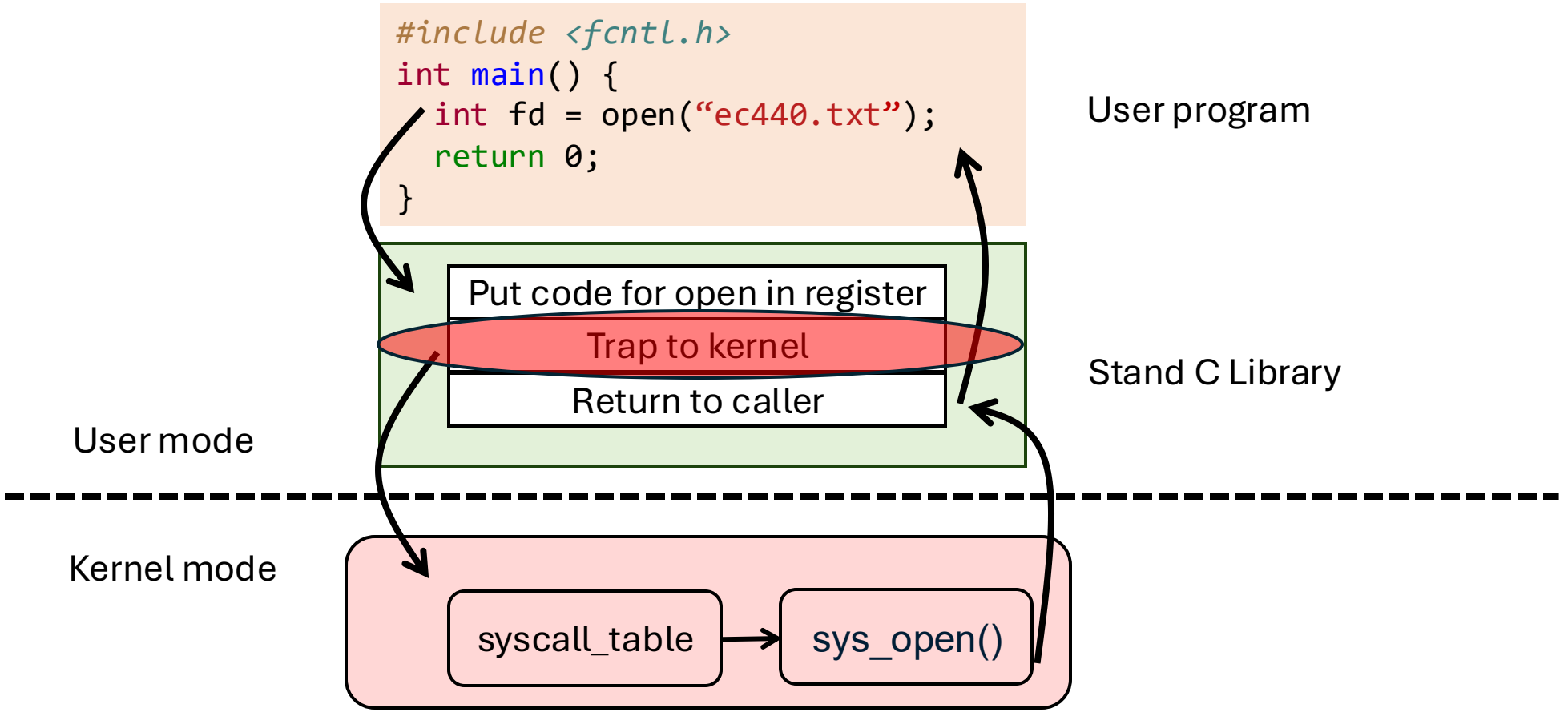
```
movl $20, %eax # Get PID of current process
int $0x80 # Invoke system call!
# Now %eax holds the PID of the current process
```

- Saves caller state(PC, regs, mode) so it can be restored
- Returning from system call restores this state

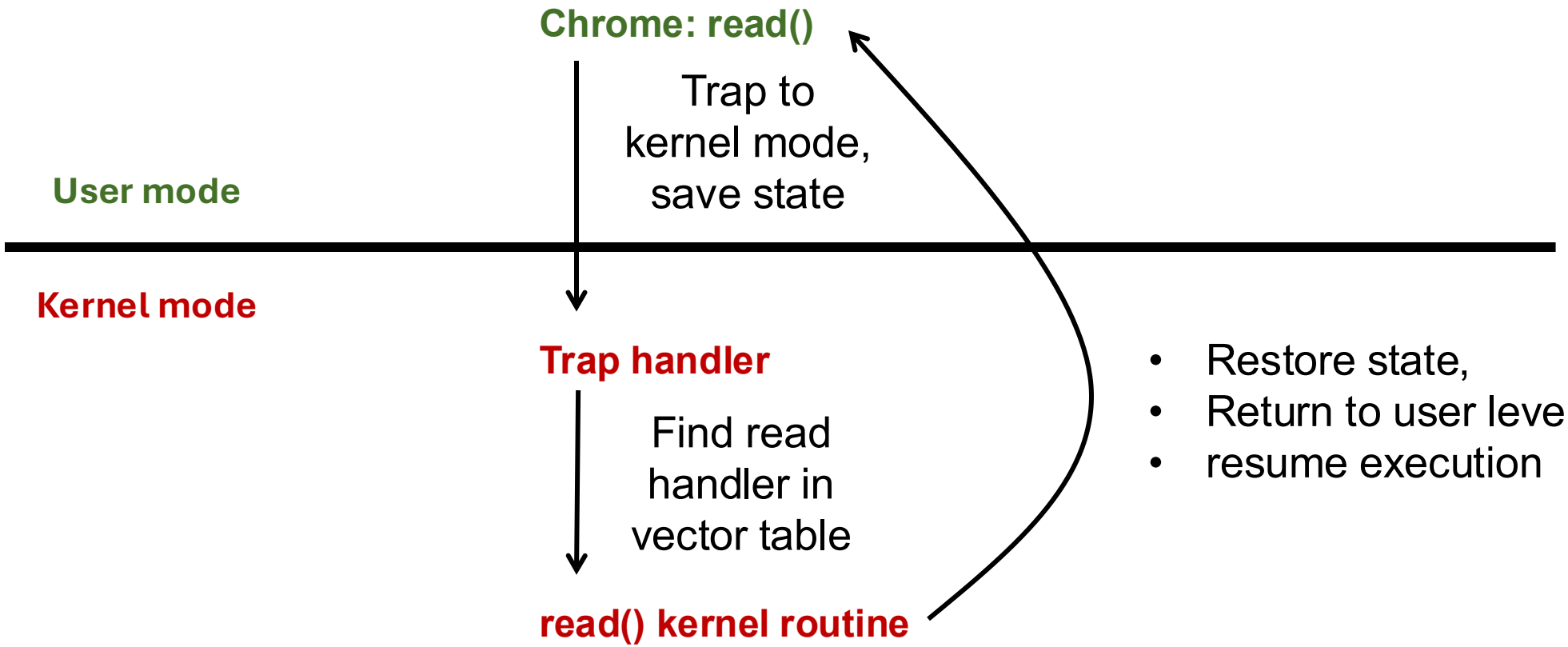
Requires architectural support to:

- Restore saved state, reset mode, resume execution

System Call: Workflow



System Call


Chrome: 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

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*

Any Questions about System Call

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

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 uses virtual memory faults for many function
  - Debugging, end-of-stack, garbage collection, copy-on-write

Fault exceptions are essentially a performance optimization

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 fault

Some fault are handled by notifying the process

- Fault handler changes the saved context to transfer control to a user model handler
- Handler must be registered with OS
- Unix signals or Win user-mode Async Procedure Calls (APCs)

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

What architectural feature that directly support the OS?

- Protection(kernel/user mode)
- Protected instructions
- Bootstrapping (Lab 0)
- Memory protection

Protection

- System calls
- Interrupts and exceptions
- Timer
- I/O control and operation

Handling ”events”

- 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

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

Exceptions

- Unexpected event during execution

Next Time..

Read Chapters 4-6 (Processes)

Lab0