Operating Systems (A) (Honor Track)

Lecture 5: Hardware Features for OS

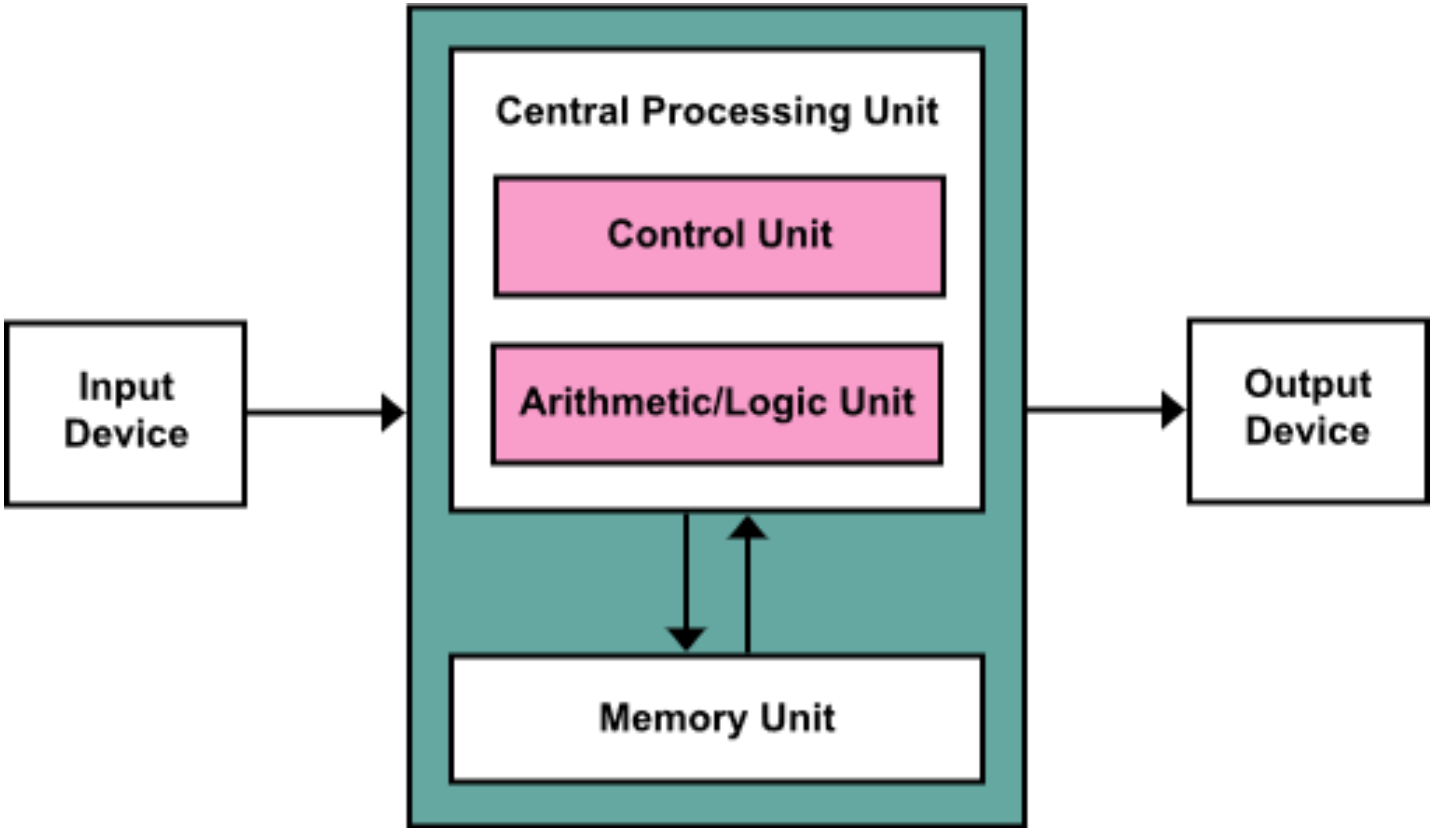
Yao Guo (郭耀)

Peking University

Fall 2021

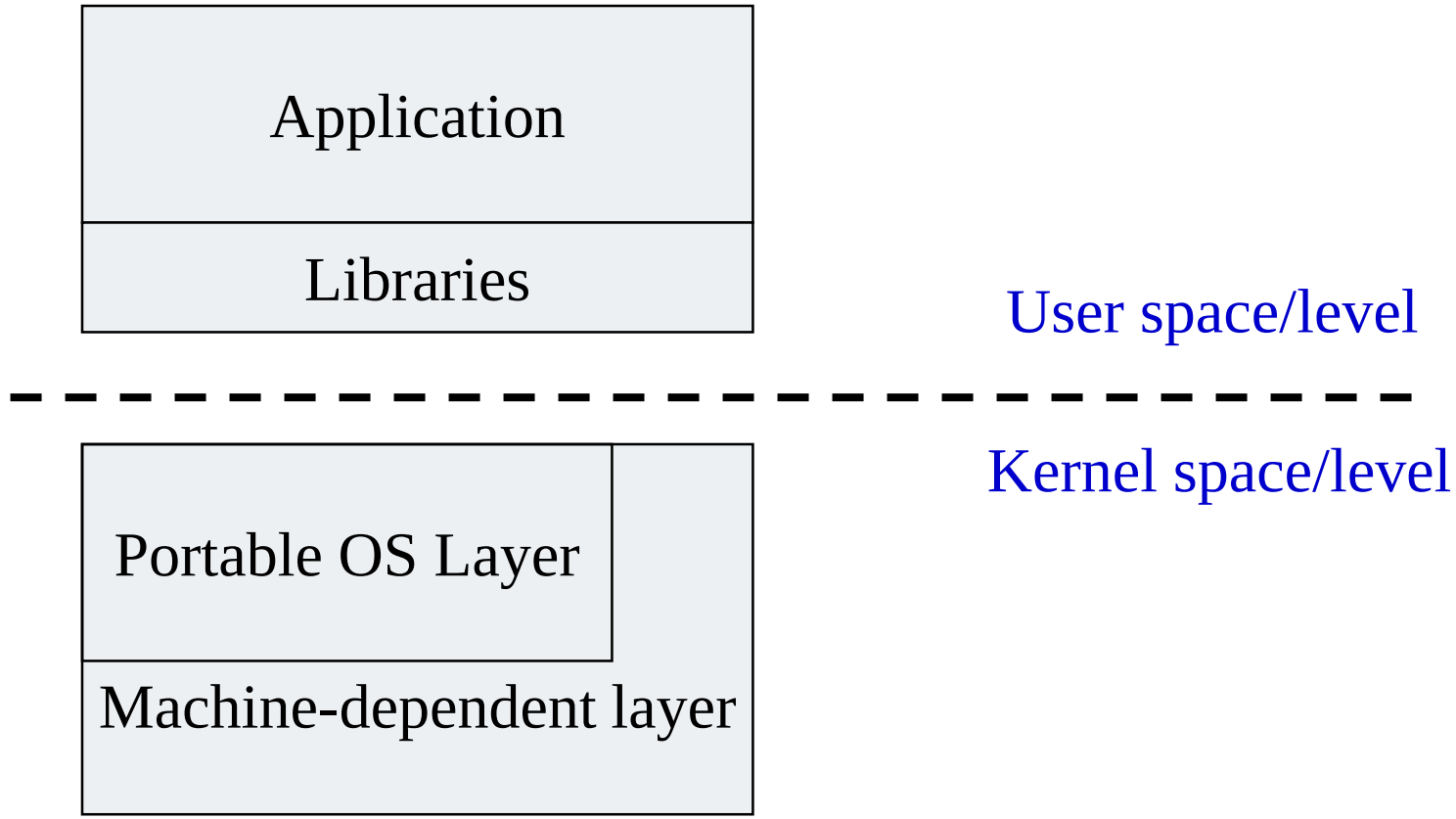
Acknowledgements: Prof. Xiangqun Chen & Tao Wang at PKU and Prof. Yuanyuan Zhou at UCSD

Review: Von Neumann Architecture



Von Neumann Bottleneck

A Peek into Unix/Linux



This Lecture

Hardware Features for OS

- Protection
- Events

Buzz Words

- Protection
- Protected/privileged instruction
- Event
- Fault
- System call
- Interrupt

Question to Ponder

- What are the necessary hardware features for OS?
  - Protected instructions
  - Kernel/user mode
  - Memory protection
  - Faults
  - System calls
  - Interrupts (timer, I/O, etc.)
  - Synchronization

- **Manipulating privileged machine state**

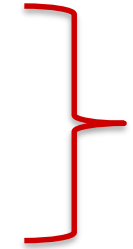
- **Generating and handling "events"**

- **Mechanisms to handle concurrency**

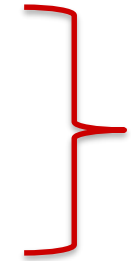
- **And how do they interfere with OS design?**

Hardware Features for OS

- What are the necessary hardware features for OS?
  - Protected instructions
  - Kernel/user mode
  - Memory protection
  - Faults
  - System calls
  - Interrupts (timer, I/O, etc.)
  - Synchronization



- **Manipulating privileged machine state**



- **Generating and handling "events"**

- **Mechanisms to handle concurrency**

Now Close Your Eyes

- Imagine a world that any programs can
  - Directly access I/O devices
  - Write anywhere in memory
  - Execute machine halt instruction
  - …
- What would happen?
  - Do you trust such systems to manage
    - Your Email/WeChat account?
    - Your banking account?
- Advanced question:
  - But some machines allow user-level programs to do some of the above, why?

Protected Instructions

- A subset of instructions of every CPU is 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
  - Manipulate memory management state
    - Page table pointers, page protection, TLB management, etc.
  - Manipulate protected control registers
    - Kernel mode, interrupt level
  - Execute machine halt instruction

OS Protection (1/2)

- How do we know when it can execute a protected instruction?
  - An easy answer: when it runs the OS code.
- But how does it know that it runs in the OS code?
  - Architecture must support (at least) two modes of operation: kernel mode and user mode
  - User programs execute in user mode
  - OS executes in kernel mode (OS == “kernel”)
  - Mode is indicated by a status bit in a protected control register

OS Protection (2/2)

- Protected instructions only execute in kernel mode
  - CPU checks mode bit when protected instruction executes
  - Setting mode bit must be a protected instruction, why?
  - Attempts to execute in user mode are detected and prevented

- How to switch from user mode to kernel mode?
  - A special system call → exception (will be discussed in the next lecture)

Memory Protection

- Why?
  - OS must be able to protect programs from each other
  - OS must protect itself from user programs
  - ➔ all these need OS control on memory usage
- Memory management hardware (called MMU) provides memory protection mechanisms
- Manipulating MMU uses protected (privileged) operations

This Lecture

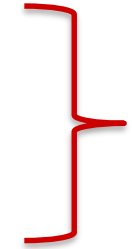
Hardware Features for OS

- Protection

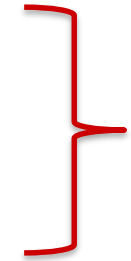
- Events

Hardware Features for OS

- What are the necessary hardware features for OS?
  - Protected instructions
  - Kernel/user mode
  - Memory protection
  - Faults
  - System calls
  - Interrupts (timer, I/O, etc.)
  - Synchronization



- **Manipulating privileged machine state**



- **Generating and handling "events"**

- **Mechanisms to handle concurrency**

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
  - Event handlers always execute in kernel mode
  - The specific types of events are defined by the machine
- Once the system is booted, all entry to the kernel occurs as the result of an event
  - In effect, the operating system is one big event handler
  - OS only executes in reaction of events

Categorizing Events

- Two kinds of events, interrupts and exceptions
- Exceptions are caused by executing instructions
  - CPU requires software intervention to handle a fault or trap
- Interrupts are caused by an external event
  - Device finishes I/O, timer expires, etc.

- Two reasons for events, unexpected and deliberate
- Unexpected events are, well, unexpected
  - What is an example?
- Deliberate events are scheduled by OS or application
  - Why would this be useful?

Categorizing Events (2)

- This gives us a convenient table:

	Unexpected	Deliberate
Exceptions (sync)	fault	system call
Interrupts (async)	interrupt	software interrupt

  - Terms may be used slightly differently by various OSes, CPU architectures…

Events

	Unexpected	Deliberate
Exceptions (sync)	fault	system call
Interrupts (async)	interrupt	software interrupt

Faults


Explorer.EXE - Application Error
The exception Floating-point division by zero.
(0xc000008e) occurred in the application at location 0x66e9433c.
Click on OK to terminate the program
OK

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 OSs use VM faults for many functions
  - Debugging, end-of-stack, garbage collection, copy-on-write
- Fault exceptions can be regarded as a performance optimization
  - Could detect faults by inserting extra instructions into code (at a significant performance penalty)

Handling Faults (1)

- Some faults are handled by “fixing” the exceptional condition and returning to the faulting context
  - Page faults cause the OS to place the missing page into memory
- Some faults are handled by notifying the process
  - Fault handler changes the saved context (e.g. PC, in kernel) to transfer control to a user-mode handler on return from fault
  - Handler must be registered with OS
  - Unix signals or NT user-mode Async Procedure Calls (APCs)
    - SIGALRM, SIGHUP, SIGTERM, etc.

Handling Faults (2)

- The kernel may handle unrecoverable faults by killing the user 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

Events

	Unexpected	Deliberate
Exceptions (sync)	fault	system call
Interrupts (async)	interrupt	software interrupt

System Calls

- For a user program to do something "privileged" (e.g., I/O), it must call an OS procedure
  - Known as crossing the protection boundary, or a protected procedure call
- CPU ISA provides a system call instruction that:
  - Causes an exception, which vectors to a kernel handler
  - Passes a parameter determining the system routine to call
  - Saves caller state (PC, regs, mode) so it can be restored
  - Returning from system call restores this state
- Requires architectural support to:
  - Verify input parameters (e.g., valid addresses for buffers)
  - Restore saved state, reset mode, resume execution (iret)

System Call Functions

- Process control
  - Create process, allocate memory
- File management
  - Create, read, delete file
- Device management
  - Open device, read/write device, mount device
- Information maintenance
  - Get time, get system data/parameters
- Communications
  - Create/delete channel, send/receive message
- Programmers generally do **not** use system calls directly
  - They use runtime libraries (e.g. Java, C)
  - Why?

System Call

```
open:
        push    dword mode
        push    dword flags
        push    dword path
        mov     eax, 5
        push    eax
        int     80h
        add     esp, byte 16
```

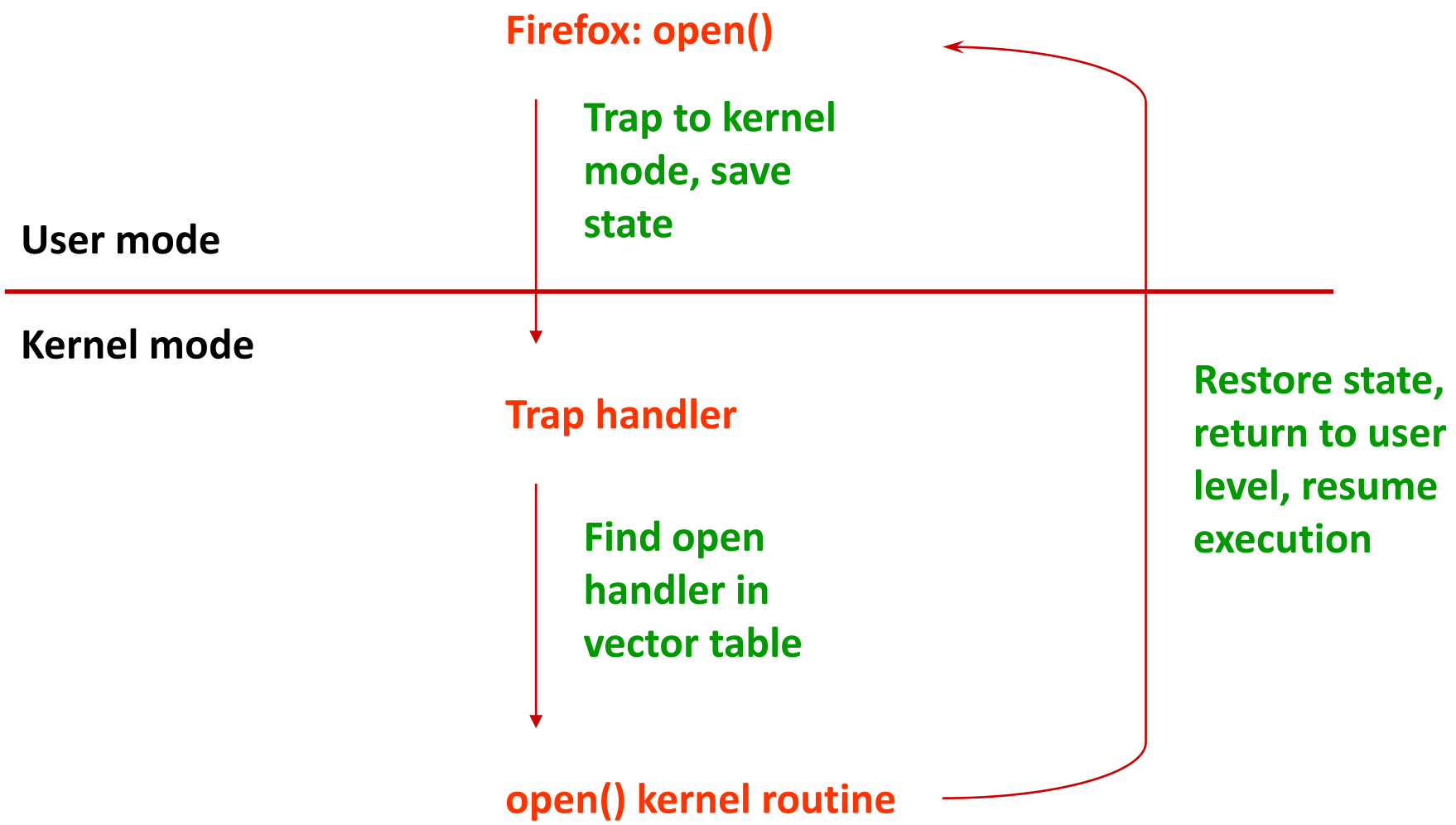
Differences with a function call?

```
main:    movl $10, %eax
         call foo
         …

foo:     addl $5, %eax
         ret
```

How much more expensive is a system call than a function call? (homework)

More information can be found at http://www.int80h.org/

System Call: Overall Process



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 sys**call**() 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**".

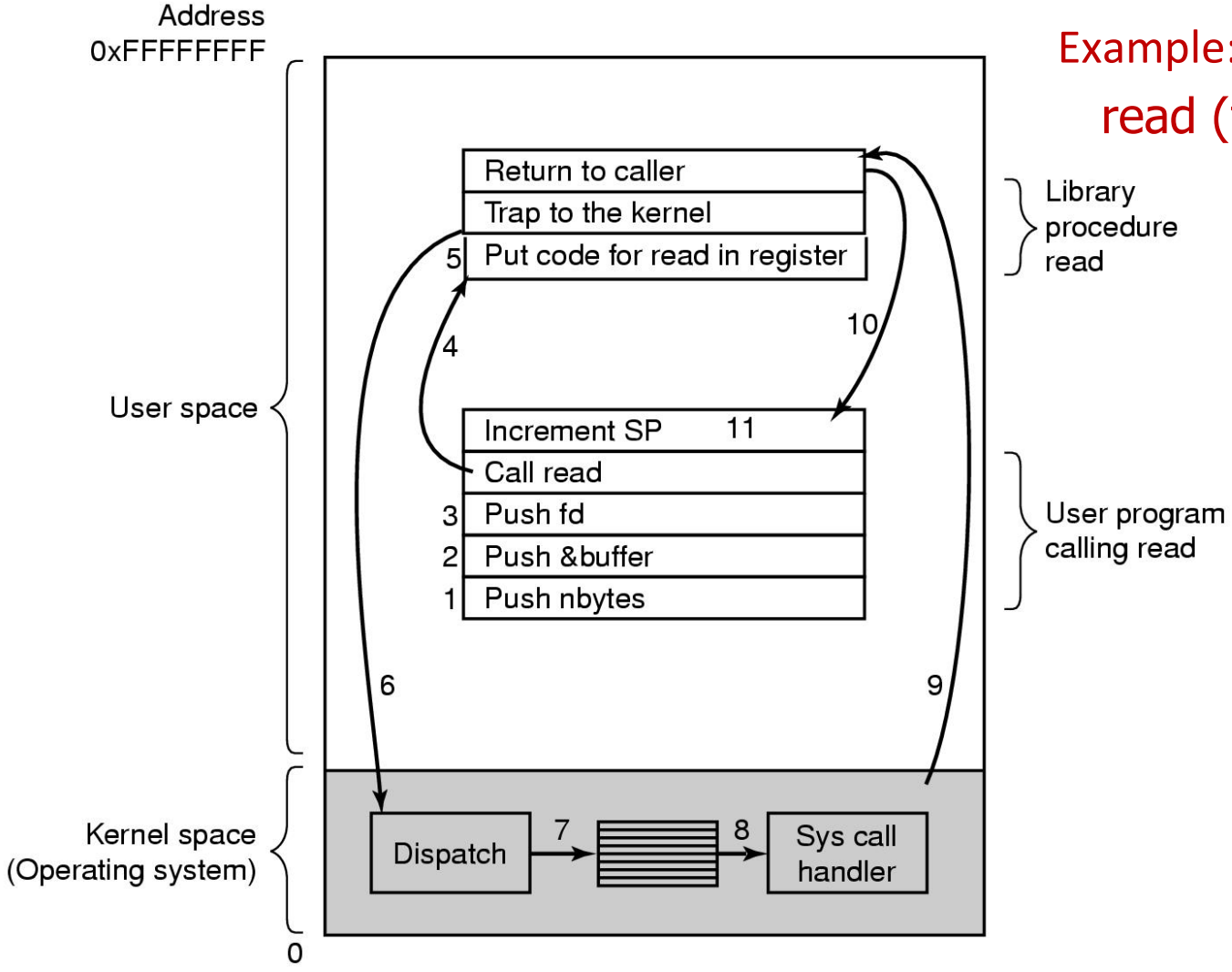
System Call Example

```
#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*

Steps in Making a System Call



Example:

read (fd, buffer,nbytes)

System Call Questions

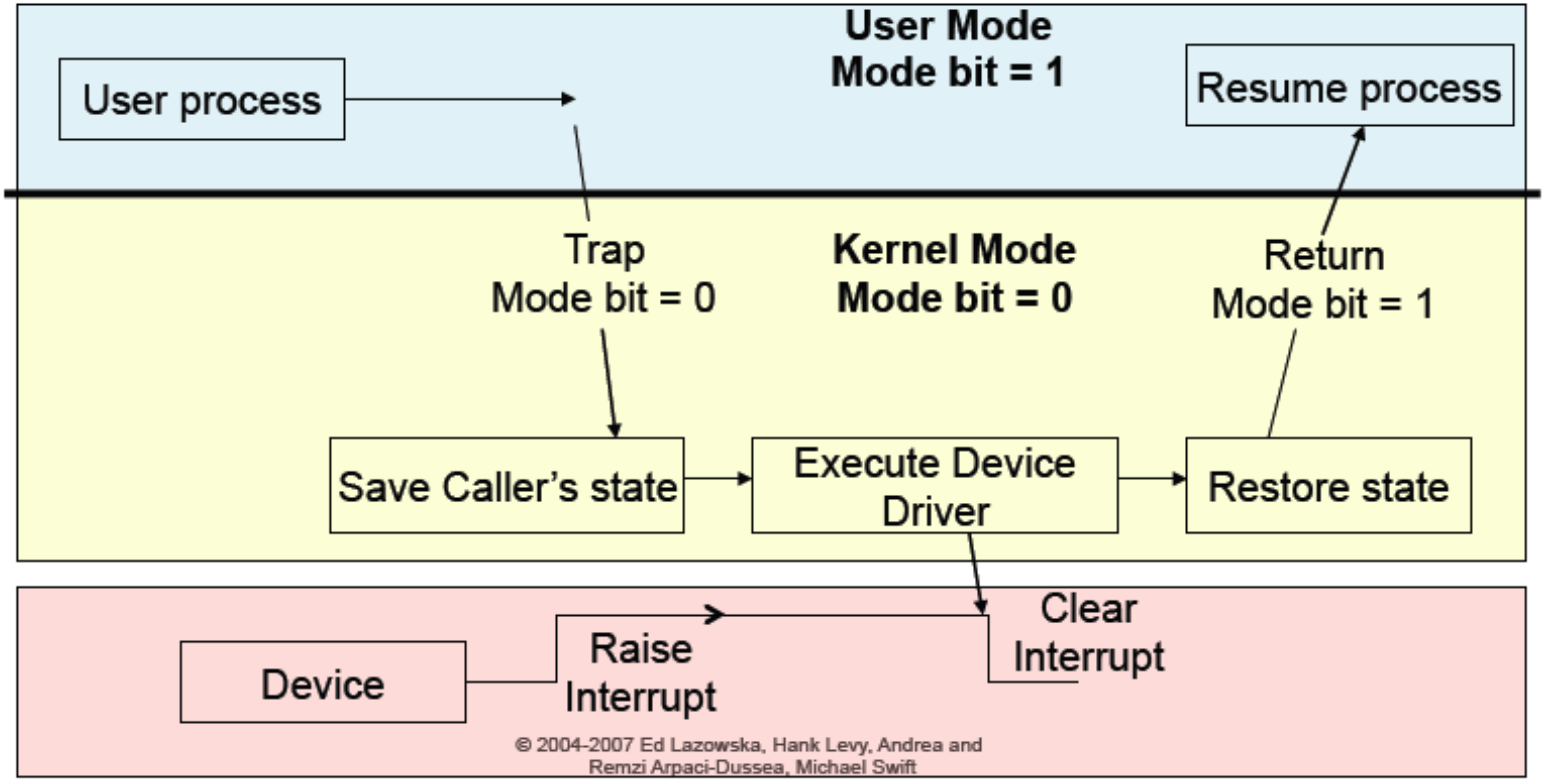
- What would happen if kernel did not save state before a system call?
- Why must the kernel verify arguments?
- Why is a table of system calls in the kernel necessary?

Events

	Unexpected	Deliberate
Exceptions (sync)	fault	system call
Interrupts (async)	interrupt	software interrupt

Interrupts

- Interrupts signal asynchronous events
  - I/O hardware interrupts
  - Software and hardware timers



Two Flavors of Interrupts

- Two flavors of interrupts
  - Precise: CPU transfers control only on instruction boundaries (prev insts committed, curr/next insts no impact)
  - Imprecise: CPU transfers control in the middle of instruction execution
    - What the heck does that mean?
  - OS designers like precise interrupts, CPU designers like imprecise interrupts
    - Why?

Timer

- The timer is critical for an operating system
- It is the fallback mechanism by which the OS reclaims 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()*)

I/O Control

- Initiating an I/O
  - Special instructions
- Completing an I/O – interrupt (asynchronous I/O)
  - Device operates independently of rest of machine
  - Device sends an interrupt signal to CPU when done
  - OS handles the interrupt
- IRQ: Interrupt request

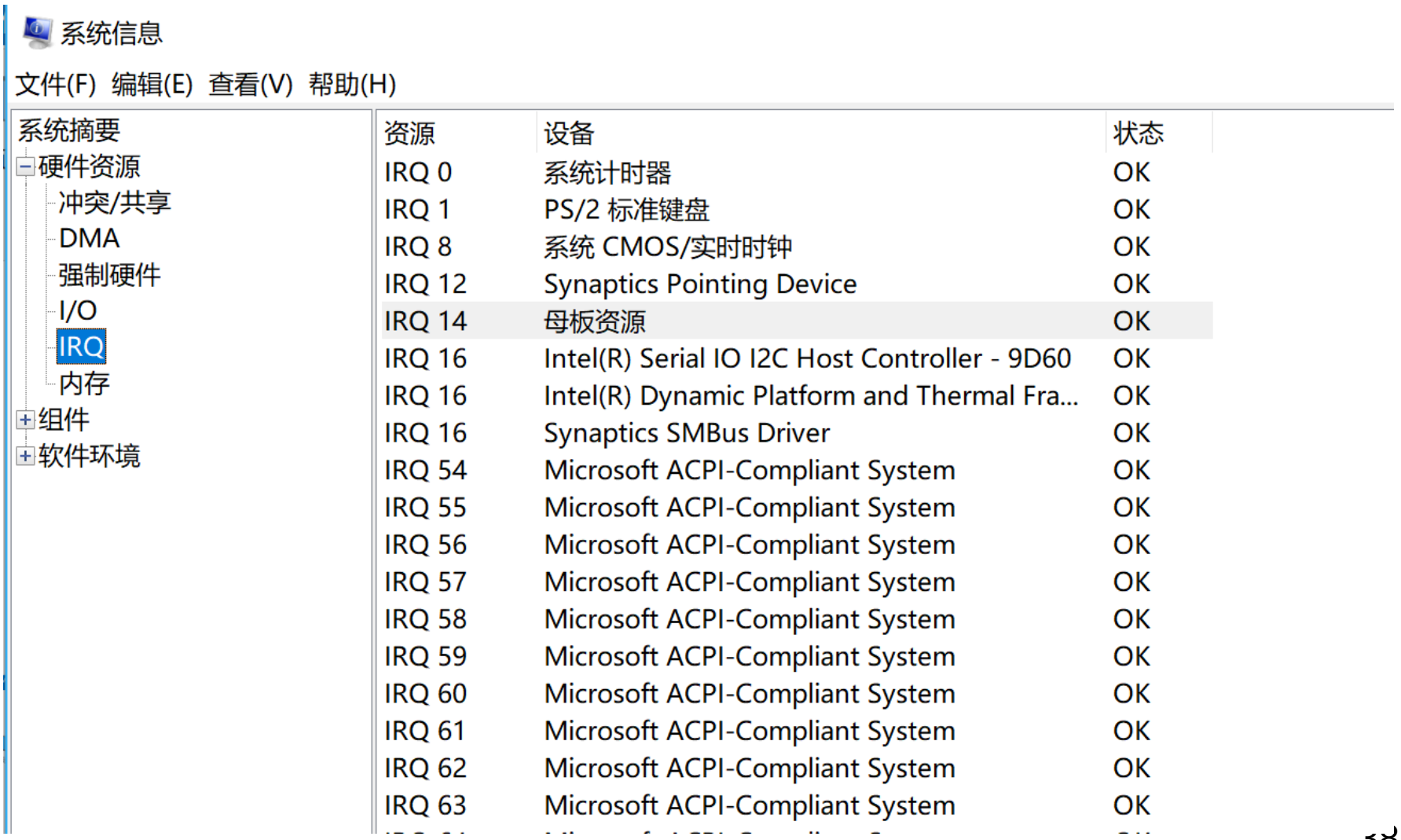
x86 IRQs

- Managed by two 8259 PICs: master PIC and slave PIC
- Master PIC
  - IRQ 0 – system timer (cannot be changed)
  - IRQ 1 – keyboard controller (cannot be changed)
  - IRQ 2 – cascaded signals from IRQs 8–15 (any devices configured to use IRQ 2 will actually be using IRQ 9)
  - IRQ 3 – serial port controller for serial port 2 (shared with serial port 4, if present)
  - IRQ 4 – serial port controller for serial port 1 (shared with serial port 3, if present)
  - IRQ 5 – parallel port 2 and 3 or sound card
  - IRQ 6 – floppy disk controller
  - IRQ 7 – parallel port 1. It is used for printers or for any parallel port if a printer is not present.

x86 IRQs (2)

- Slave PIC
  - IRQ 8 – real-time clock (RTC)
  - IRQ 9 – Advanced Configuration and Power Interface (ACPI) system control interrupt on Intel chipsets
  - IRQ 10 – The Interrupt is left open for the use of peripherals (open interrupt/available, SCSI or NIC)
  - IRQ 11 – The Interrupt is left open for the use of peripherals (open interrupt/available, SCSI or NIC)
  - IRQ 12 – mouse on PS/2 connector
  - IRQ 13 – CPU co-processor or integrated floating point unit or inter-processor interrupt (use depends on OS)
  - IRQ 14 – primary ATA channel (ATA interface usually serves hard disk drives and CD drives)
  - IRQ 15 – secondary ATA channel

Modern IRQs


系统信息
文件(F) 编辑(E) 查看(V) 帮助(H)
系统摘要
硬件资源
冲突/共享
DMA
强制硬件
I/O
IRQ
内存
组件
软件环境
资源 设备 状态
IRQ 0 系统计时器 OK
IRQ 1 PS/2 标准键盘 OK
IRQ 8 系统 CMOS/实时时钟 OK
IRQ 12 Synaptics Pointing Device OK
IRQ 14 母板资源 OK
IRQ 16 Intel(R) Serial IO I2C Host Controller - 9D60 OK
IRQ 16 Intel(R) Dynamic Platform and Thermal Fra... OK
IRQ 16 Synaptics SMBus Driver OK
IRQ 54 Microsoft ACPI-Compliant System OK
IRQ 55 Microsoft ACPI-Compliant System OK
IRQ 56 Microsoft ACPI-Compliant System OK
IRQ 57 Microsoft ACPI-Compliant System OK
IRQ 58 Microsoft ACPI-Compliant System OK
IRQ 59 Microsoft ACPI-Compliant System OK
IRQ 60 Microsoft ACPI-Compliant System OK
IRQ 61 Microsoft ACPI-Compliant System OK
IRQ 62 Microsoft ACPI-Compliant System OK
IRQ 63 Microsoft ACPI-Compliant System OK

I/O Example

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

Interrupt Questions

- Interrupts halt the execution of a process and transfer control (execution) to the operating system
  - Can the OS be interrupted? (Consider why there might be different IRQ levels)
- Interrupts are used by devices to have the OS do stuff
  - What is an alternative approach to using interrupts?
  - What are the drawbacks of that approach?

Events

	Unexpected	Deliberate
Exceptions (sync)	fault	system call
Interrupts (async)	interrupt	software interrupt

Software Interrupt

- Software interrupt – a.k.a. async system trap (AST), async or deferred procedure call (APC or DPC)
  - Is software interrupt a hardware feature?

➔ Learn software interrupt by yourself

Synchronization

- Interrupts cause difficult problems
  - An interrupt can occur at any time
  - A handler may interfere 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 atomically
    - XCHG instruction in x86

XCHG

Exchange Register/Memory with Register

Opcode	Mnemonic	Description
90+rw	XCHG AX, 16	Exchange r16 with AX.
90+rw	XCHG r16, X	Exchange AX with r16.
90+rd	XCHG EAX, r32	Exchange r32 with EAX.
90+rd	XCHG r32, EAX	Exchange EAX with r32.
86 /r	XCHG r/m8, r8	Exchange r8 (byte register) with byte from r/m8.
86 /r	XCHG r8, r/m8	Exchange byte from r/m8 with r8 (byte register).
87 /r	XCHG r/m16, r16	Exchange r16 with word from r/m16.
87 /r	XCHG r16, r/m16	Exchange word from r/m16 with r16.
87 /r	XCHG r/m32, r32	Exchange r32 with doubleword from r/m32.
87 /r	XCHG r32, r/m32	Exchange doubleword from r/m32 with r32.

Exchanges the contents of the destination (first) and source (second) operands. The operands can be two general-purpose registers or a register and a memory location. If a memory operand is referenced, the processor's locking protocol is automatically implemented for the duration of the exchange operation, regardless of the presence or absence of the LOCK prefix or of the value of the IOPL. (See the LOCK prefix description in this chapter for more information on the locking protocol.) This instruction is useful for implementing semaphores or similar data structures for process synchronization. The XCHG instruction can also be used instead of the BSWAP instruction for 16-bit operands.

Summary of Events

- After the OS has booted, all entry to the kernel happens as the result of an event
  - event immediately stops current execution
  - changes mode to kernel mode, event handler is called
- Kernel defines handlers for each event type
  - specific types are defined by the architecture
    - e.g.: timer event, I/O interrupt, system call trap
- When the processor receives an event, it
  - transfers control to handler within the OS
  - handler saves program state (PC, regs, etc.)
  - handler functionality is invoked
  - handler restores program state, returns to program (if OS wants to)

Architecture Trends Impact OS Design

- Human: computer ratio
  - Batch - time sharing - PCs - embedded/pervasive computing
  - Single job - time shared - internetworked
- Programmer: processor cost ratio
  - assembly to C to Java to Python languages
  - command line to GUI to pen / voice interfaces
- Networking
  - Isolation to dialup to LAN to WAN to wireless
    - OS must devote more effort to communications
    - OS must provide more security / protection
- Mobile/battery-operated
  - OS must pay attention to energy consumption

Summary

- Protection
  - Protected instructions
  - Protection of kernel/user mode
  - Memory protection
  - Further readings
    - TPM: Trusted Platform Module
    - ARM Trustzone: trusted world vs. normal world
- Events
  - Faults
  - System calls
  - Interrupts (timer, I/O, etc.)
  - Software interrupts
- Architecture trends impact OS design
- Next lecture: Processes and threads