ECE391 Computer System Engineering Lecture 5

University of Illinois at Urbana- Champaign

Fall 2022

Lecture Topics

- Role of system software
- System calls, exceptions, & interrupts
- Processor support for interrupts

MP1 Handin and Demo Schedule

- Code must be committed to master/main branch on GitLab by
 - 5:59 PM on Monday 9/12

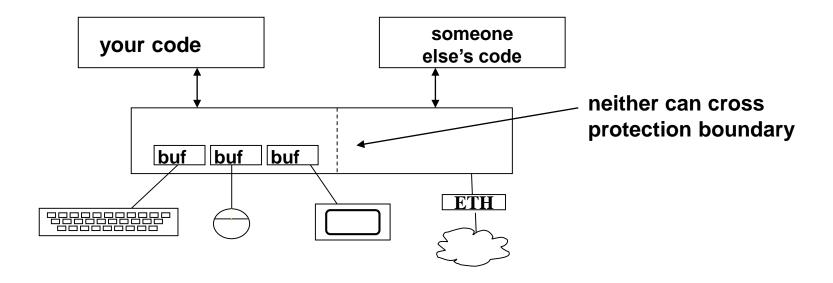
Role of System Software (1)

- System software serves three purposes
 - virtualization
 - protection
 - abstraction (particularly hiding asynchrony)
- virtualization:
 - the illusion of multiple/practically unlimited resources
- protection:
 - reduce/eliminate the chance of accidental and/or malicious destruction of data/results by another program

Role of System Software (2)

abstraction:

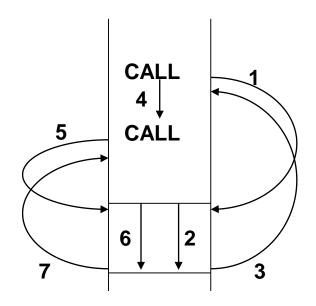
- hide fundamentally asynchronous nature of processor/device interaction
- provide simpler and more powerful interfaces (integrated w/protection)



System Calls, Interrupts, and Exceptions (1)

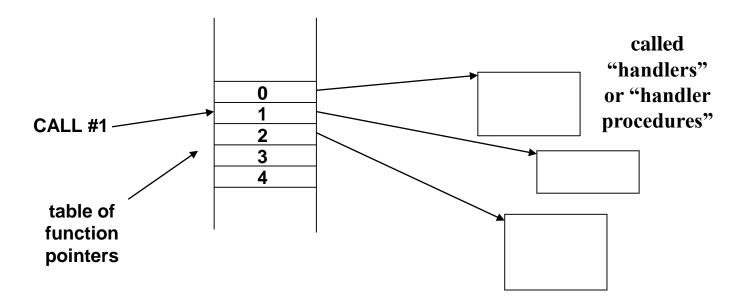
 recall that subroutines allow a programmer to encapsulate common operations

- the operating system
 - want to provide an interface including common operations
 - BUT don't want to re-link programs
 - NOR rely on everyone having exactly the same OS version



System Calls, Interrupts, and Exceptions (2)

- solution
 - add a level of indirection!
- with indirection
 - to rewrite OS, just change the table
 - application code does not change



System Calls, Interrupts, and Exceptions (3)

• in LC-3, we used the TRAP instruction; in x86, it's the INT instruction:

INT 8-bit imm. # (PUSH EIP), EIP ← table[imm8]

- the RTL is actually a little more complicated, as you'll see later in course
- called a trap (after instruction, or trap door through protection boundary)
- also called a system call (for operating system)

System Calls, Interrupts, and Exceptions (4)

- vector tables/jump tables
 - i.e., tables of function pointers
 - convenient abstraction for many procedure-like activities
- Question:
 - What happens if software does something wrong, e.g.,
 - accesses a non-existent memory location?
 - issues an illegal/undefined instruction? divides by 0?
- What do we do to handle problems?
 - state machine that you design for processor may have don't cares
 - state machine that you build will do something (may be unknown)
 - so just let it run! (e.g., 6502 did so... and programmers used!)

System Calls, Interrupts, and Exceptions (5)

- a better solution: exceptions!
 - processor maps each problem to a vector #
 - calls procedure in vector table by #
- Where else might we use vector tables?

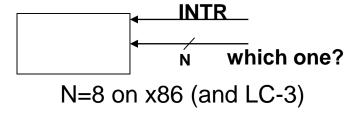
- Consider processor interactions with devices
 - a disk access takes about 10 milliseconds
 - new machines in lab: 10 ms = 32 million cycles
- should processor sit around asking, "Are my data here yet?"

System Calls, Interrupts, and Exceptions (6)

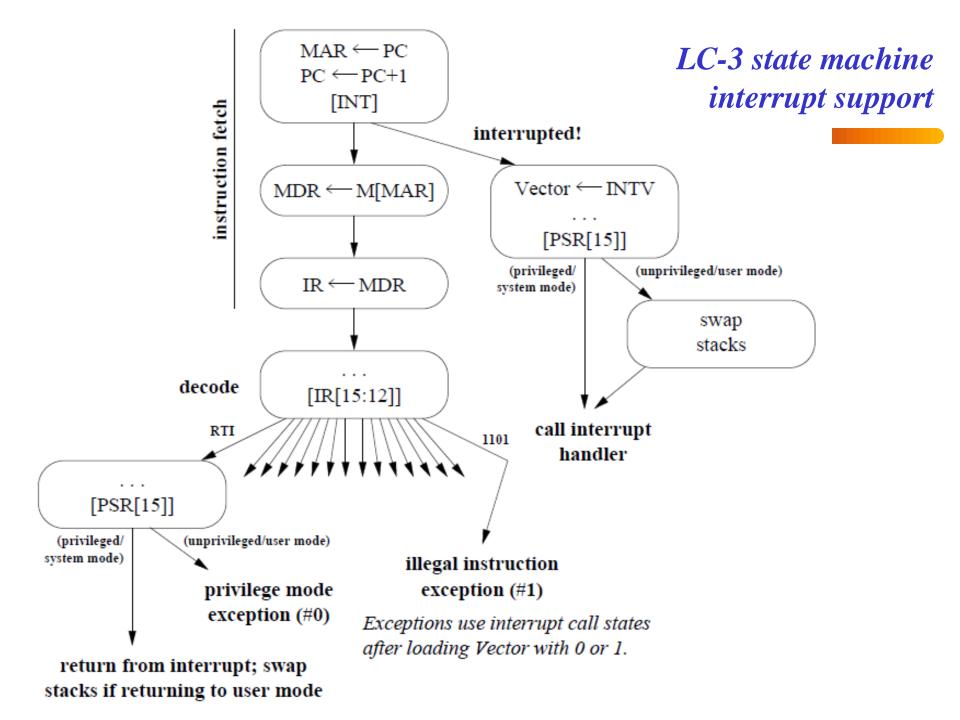
- analogous to posting a letter to a friend in Europe
- and checking your mailbox every minute for a reply
- instead, have your mail carrier ring your doorbell when it arrives
- in a processor, we call that an interrupt
- How can we use a vector table for interrupts?
 - each device has a vector #
 - call corresponding procedure in vector table when device generates a request for service
- x86 ISA
 - uses one table for all three kinds
 - called the Interrupt Descriptor Table (IDT)

Processor Support for Interrupts (1)

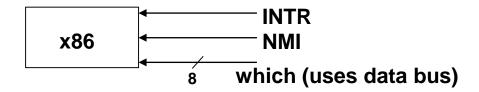
- How does a processor support interrupts?
- Logically...



 How should we change a processor's state machine to incorporate interrupts?



Processor Support for Interrupts (2)



- x86 allows software to block interrupts with a status flag (in EFLAGS)
- normal interrupts occur only when the interrupt enable flag (IF) is set
- some interrupts are too important
 - e.g., memory errors, power warnings, etc.
 - these are NOT maskable, and use a separate input to processor
 - called non-maskable interrupts (NMI)

Interrupt Descriptor Table

- as mentioned earlier
 - x86 uses a single vector table
 - the Interrupt Descriptor Table (IDT)
 - hold vectors for interrupts, exceptions, and system calls
- note that this picture is partly OS-specific
 - the exception vector numbers are specified by Intel Why?
 - A: generated directly by processor's state machine
 - programmable interrupt controller (PIC) will be discussed later;
 - range of vectors generated is programmable
 - note that a single entry is used for all system calls in Linux

	0x00	division error		
	:			
	0x02	NMI (non-maskable interrupt)		
	0x03	breakpoint (used by KGDB)		
0x00-0x1F	0x04	overflow		
0.000-0.111	0.704	Overnow		
defined	0x0B	segment not present		
by Intel	0x0C	stack segment fault		
	0x0D	general protection fault		
	0x0E	page fault		
	:			
		IDOO diamandia		
	0x20	IRQ0 — timer chip		
0.00.0.07	0x21	IRQ1 — keyboard		
0x20-0x27	0x22	IRQ2 — (cascade to secondary)		
	0x23	IRQ3		
primary	0x24	IRQ4 — serial port (KGDB)		
8259 PIC	0x25	IRQ5		
	0x26	IRQ6	example	
	0x27	IRQ7	of	
	0x28	IRQ8 — real time clock	possible	
	0x29	IRQ9	settings	
0x28-0x2F	0x2A			
	0x2B	,		
secondary	0x2C	IRQ12 — PS/2 mouse		
8259 PIC	0x2D	IRQ13		
	0x2E	IRQ14 — ide0 (hard drive)		
	0x2F	IRQ15		
0x30-0x7F	:	APIC vectors available to device drivers		
0x80	0x80	system call vector (INT 0x80)		
0x81-0xEE	:	more APIC vectors available to device drivers		
0xEF	0xEF	local APIC timer		
0xF0-0xFF	:	symmetric multiprocessor (SMP) communication vectors		

Interrupt Descriptor Table (IDT)

- x86 uses a single vector table
 Interrupt Descriptor Table
 - hold vectors for interrupts, exceptions, and system calls

Interrupts, System calls, and Exceptions

Characteristics of *interrupts, system calls, and exceptions* with respect to a program being executed

Туре	Generated by	Asynchronous	Unexpected
Interrupts	external device	YES	YES
Exceptions	invalid opcode or operand	NO	YES
System calls	deliberate, via INT instruction	NO	NO

Device I/O

- How does a processor communicate with devices?
- Two possibilities
 - independent I/O use special instructions and a separate I/O port address space
 - memory-mapped I/O use loads/stores
 and dedicate part of the memory address space to I/O
- x86 originally used only independent I/O
 - but when used in PC, needed a good interface to video memory
 - solution? put card on the bus, claim memory addresses!
 - now uses both, although ports are somewhat deprecated

Device I/O

- I/O instructions have not evolved since 8086
 - 16-bit port space
 - byte addressable
 - little-endian (looks like memory)
 - instructions
 - IN port, dest.reg
 - OUT src.reg, port
 - the register operands are NOT general-purpose registers
 - all data to/from AL, AX, or EAX
 - port is either an 8-bit immediate (not 16!) or DX