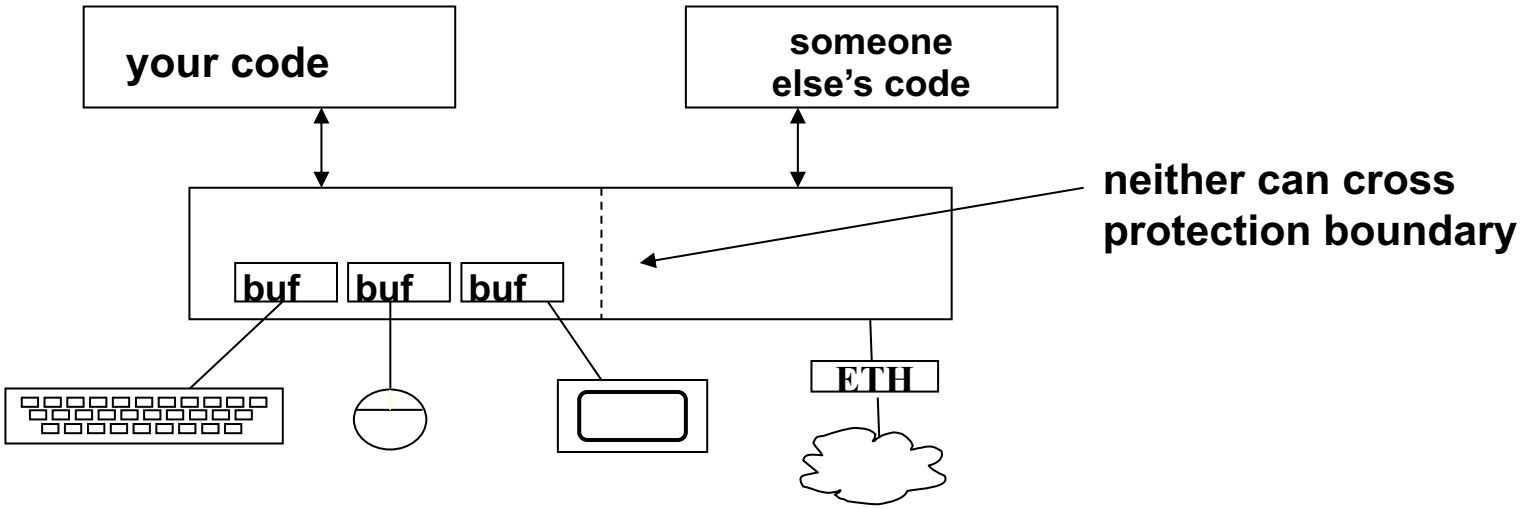
Role of System Software (1)

- 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)

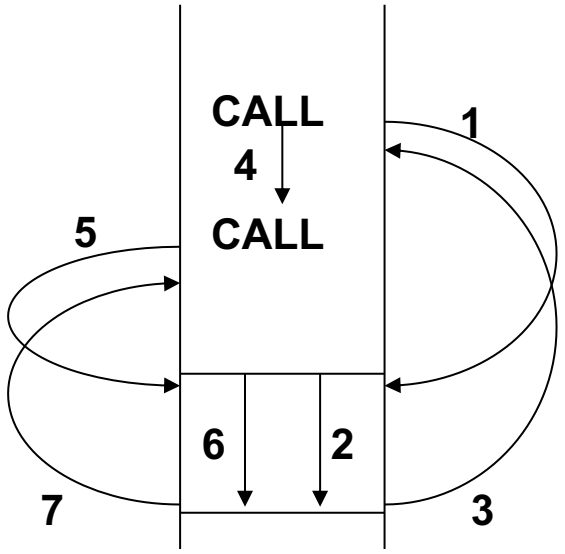
- abstraction:
  - hide fundamentally asynchronous nature of processor/device interaction
  - provide simpler and more powerful interfaces (integrated w/protection)



System Calls, Interrupts, and Exceptions (1)

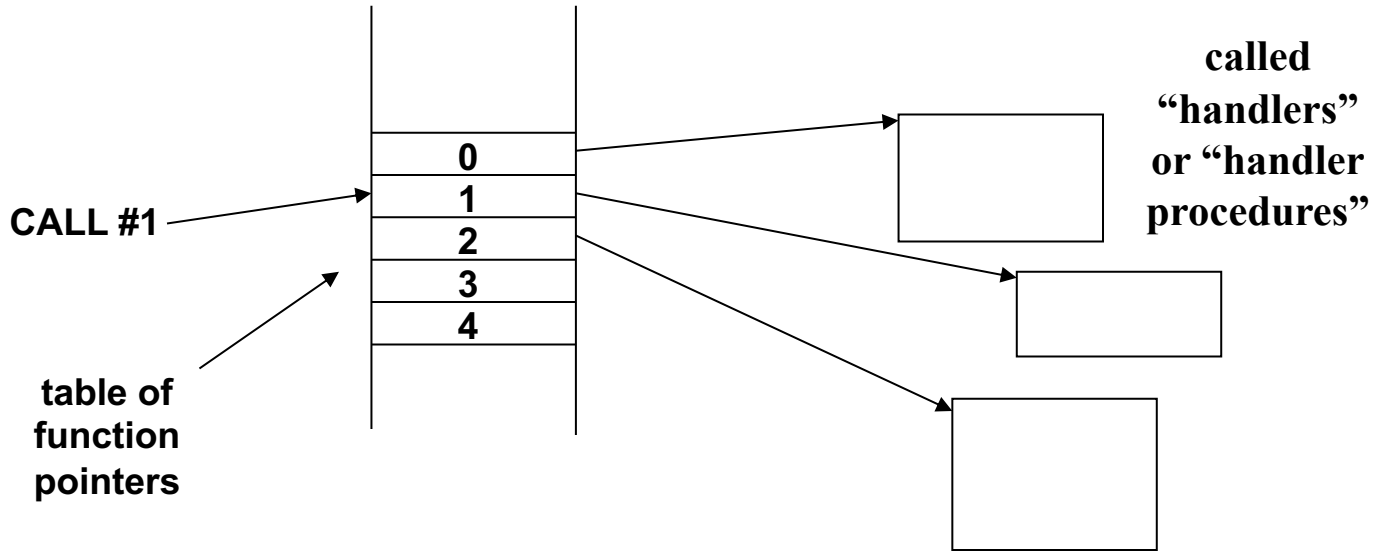
- recall that subroutines allow a programmer to encapsulate common operations

- 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?”

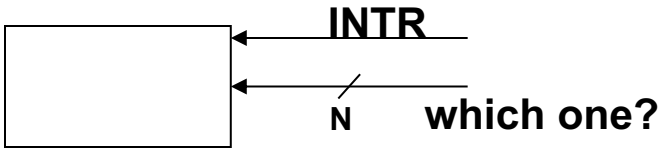
System Calls, Interrupts, and Exceptions (6)

- 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
- x86 ISA
  - uses one table for all three kinds
  - called the Interrupt Descriptor Table (IDT)

Processor Support for Interrupts (1)

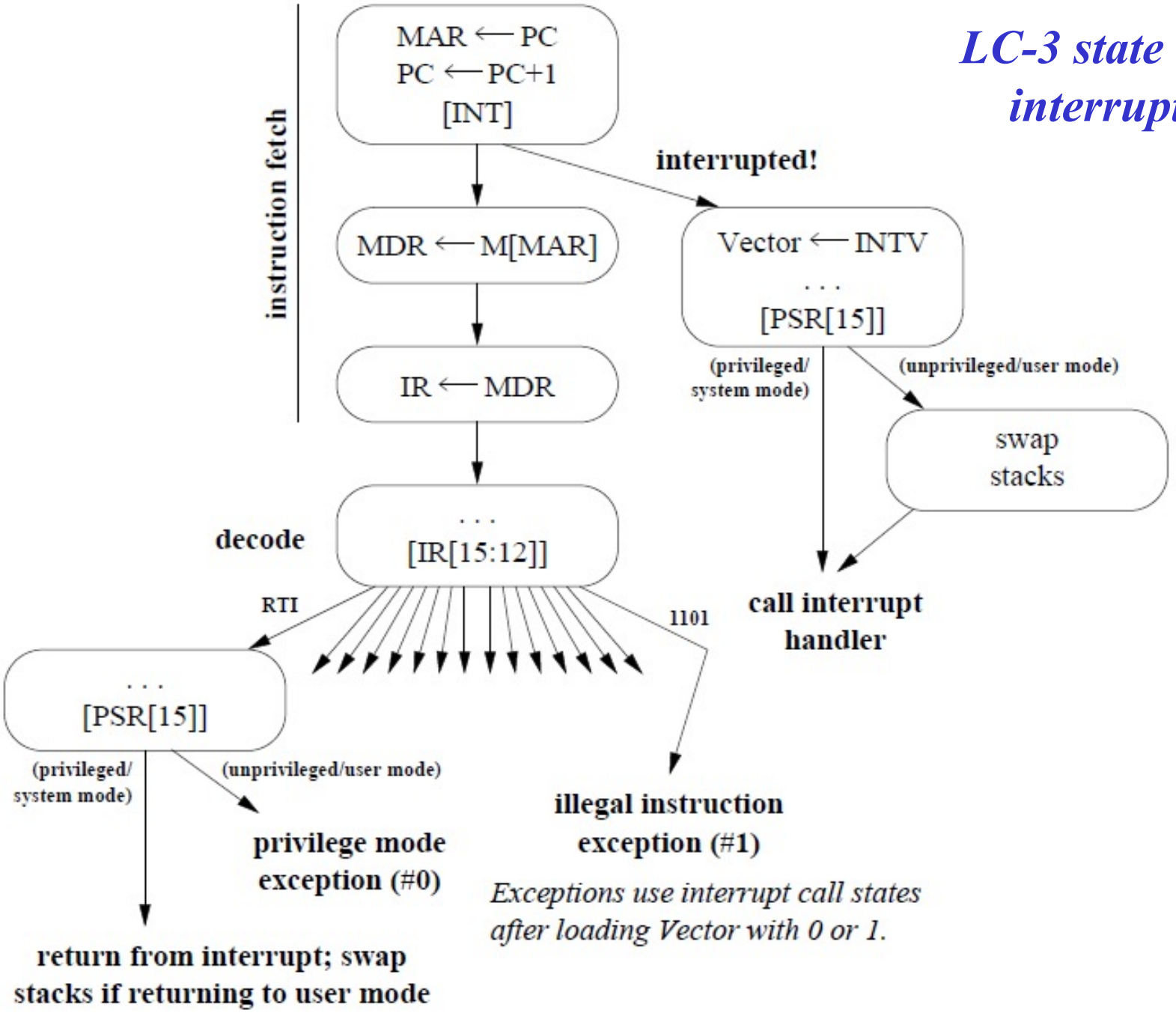
- How does a processor support interrupts?
- Logically…



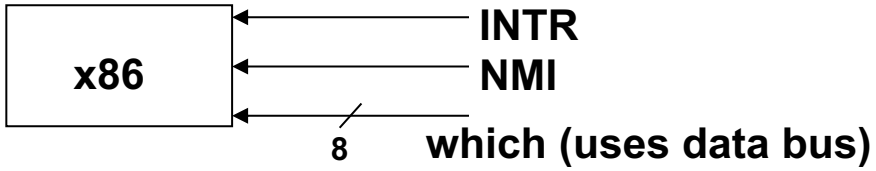
N=8 on x86 (and LC-3)

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

LC-3 state machine interrupt support



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, and is shown for Linux 2.4
  - note that a single entry is used for all system calls in Linux

Interrupt Descriptor Table

0x00–0x1F defined by Intel	0x00	division error	
	⋮		
	0x02	NMI (non-maskable interrupt)	
	0x03	breakpoint (used by KGDB)	
	0x04	overflow	
	⋮		
	0x0B	segment not present	
	0x0C	stack segment fault	
	0x0D	general protection fault	
	0x0E	page fault	
	⋮		
0x20–0x27 primary 8259 PIC	0x20	IRQ0 — timer chip	example of possible settings
	0x21	IRQ1 — keyboard	
	0x22	IRQ2 — (cascade to secondary)	
	0x23	IRQ3	
	0x24	IRQ4 — serial port (KGDB)	
	0x25	IRQ5	
	0x26	IRQ6	
	0x27	IRQ7	
0x28–0x2F secondary 8259 PIC	0x28	IRQ8 — real time clock	
	0x29	IRQ9	
	0x2A	IRQ10	
	0x2B	IRQ11 — eth0 (network)	
	0x2C	IRQ12 — PS/2 mouse	
	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	

Shared Data and Resources (1)

- The question
  - interrupt handlers and programs share resources
    - What resources are shared between them?
    - How might interactions cause problems?
    - What can we do to fix those problems?

Thought Problem on Shared Resources (2)

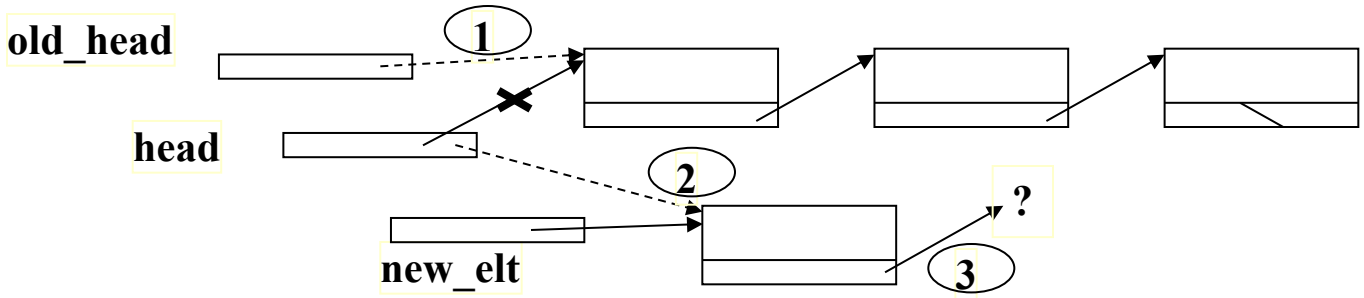
- Obvious things
  - registers
    - solution? save them to the stack
  - memory
    - solution? privatize
    - will still need to share some things; discussed later

Thought Problem on Shared Resources (3)

- Less obvious
  - condition codes
    - solution? again, save them to the stack
  - shared data
- More subtle
  - external state (e.g., on devices)
  - compiler optimization (e.g., volatility)
  - security leaks
    - e.g., application waits for interrupt, then observes values written by OS to stack
    - solution? use separate stack for kernel

Examples of Shared Resources:

Example #1: a shared linked list



step 1: `old_head = head;`

step 2: `head = new_elt;`

Oops! an interrupt!

step 3: `new_elt->next = old_head;`

Examples of Shared Resources:
Example #1: a shared linked list

- The problem?
  - linked list structure has invariant
  - head points to first, chained through last via next field, ends with NULL
  - complete operation maintains invariant
  - partial operation does not – need atomic update

Examples of Shared Resources:
Example #2: external state

- The core problem
  - devices have state
  - processors interact with devices using specific protocols
  - protocol often requires several steps (e.g., I/O instructions)
  - device cannot differentiate which piece of code performed an operation
- Example:
  - VGA controller operations for scrolling window with color modulation
  - interrupt handler drives color manipulations
  - program handles scrolling using pixel shift
  - both use VGA attribute register (port 0x3C0)

Examples of Shared Resources:
Example #2: external state

- Protocol for attribute control register
  - 22 different attributes accessed via this register
  - first send index
  - then send data
  - VGA tracks whether next byte sent is index or data

- Problem: processor can't know which one is expected
- Solution: reading from port 0x3DA forces VGA to expect index next

Examples of Shared Resources:
Example #2: external state

- Consider the program code
  - the horizontal pixel panning register is register 0x13
  - assume that the code should write the value 0x03 to it

(discard) ← P[0x3DA]	MOVW $0x3DA, %DX
	INB (%DX),%AL
0x13 → P[0x3C0]	MOVW $0x3C0, %DX
	MOVB $0x13, %AL
	OUTB %AL, (%DX)
0x03 → P[0x3C0]	MOVB $0x03, %AL
	OUTB %AL, (%DX)

Examples of Shared Resources:
Example #2: external state

- What happens if the interrupt occurs after the first write to 0x3C0?
  - the interrupt handler is executing basically the same code
  - leaves the VGA expecting an index

- What is the solution?

Examples of Shared Resources:
Example #3: handshake synchronization

- A device generates an interrupt after it finishes executing a command
- Consider the following attempt to synchronize

the shared variable…

```
int device_is_busy = 0;
```

the interrupt handler…

```
device_is_busy = 0;
```

Examples of Shared Resources:
Example #3: handshake synchronization

The program function used to send a command to the device…

```
while (device_is_busy);/* wait until device is free */

device_is_busy = 1;

/* send new command to device */
```

- Q: Does the loop work?
- No.
  - Compiler assumes sequential program.
  - Variables can't change without code that changes them.

Examples of Shared Resources:
Example #3: handshake synchronization

```
LOOP: MOVL device_is_busy, %EAX
      CMPL $0, %EAX
      JNE  LOOP
```

- Nothing can change variable, so no need to reload (move LOOP down a line).
- Now nothing can change EAX, so move it down another line (to branch!).
- Will interrupt handler break you out of the resulting infinite loop?

Examples of Shared Resources:
Example #3: handshake synchronization

- Solution
  - mark variable as volatile
  - tells compiler to never assume that it hasn't changed between uses

- Why not mark everything volatile?
  - forces compiler to always re-load variables
  - more memory operations = slower program

Examples of Shared Resources:
Example #3: handshake synchronization

- Is it ok to swap setting the variable and sending the command?
- No.
  - introduces a race condition:

```
/* send new command to device */
---- INTERRUPT OCCURS HERE ----
device_is_busy = 1;
```

    - Next command call blocks (forever) for device to be free.

Examples of Shared Resources:
Example #3: handshake synchronization

- Unfortunately, writing your code correctly is not enough.
  - compiler optimization allowed to reorder
    - so long as code is equivalent
    - assuming sequential program
  - also (and much more subtly!)
    - ISA implementation is allowed to reorder

- Message
  - important to think about reordering possibilities by compiler and ISA and to prevent bad reorderings

Critical Sections

- Some parts of program need to appear to execute atomically, i.e., without interruption
- Full version: atomic with respect to code in interrupt handler
  - for now, the clause is implied i.e., only interrupt handlers can operate during our programs
  - however, multiprocessors may have >1 program executing at same time

Critical Sections

- Solution?
  - IF (the interrupt enable flag)

    critical section start (CLI)

    (the code to be executed atomically)

    critical section end (STI)
- What else must be prevented?
  - no moving memory ops into or out of critical section!

Critical Sections in Examples

- Example #2: external state

```
MOVW  $0x3DA, %DX
INB   (%DX),%AL

MOVW  $0x3C0, %DX
MOVB  $0x13, %AL

OUTB  %AL, (%DX)
MOVB  $0x03, %AL
OUTB  %AL, (%DX)
```

CLI ←

STI ←

the critical section should be as short as possible

Critical Sections in Examples

- Why should critical sections be short?
  - avoid delaying device service by interrupt handler
  - long delays can even crash system (e.g., swap disk driver timeout)
- Example #1: a shared linked list

```
old_head = head;
head = new_elt;
new_elt->next = old_head;
```

CLI

could skip first statement, but including is safer

STI

- If interrupt handler can change list, too, leaving out first inst. creates race
- Example #3: handshake synchronization—volatile suffices for this example