

# 15-410

*“Computers make very fast, very accurate mistakes.”  
--Brandon Long*

## Hardware Overview Jan. 23, 2005

**Dave Eckhardt**

**Bruce Maggs**

# Synchronization

## Today's class

- Not exactly Chapter 2 or 13

## Project 0

- Due Wednesday at midnight
- Consider *not* using a late day
  - Could be a valuable commodity later
- Remember, this is a warm-up
  - Reliance on these skills will increase rapidly

## Upcoming

- Lecture on “The Process”
- Project 1

# Synchronization

## Personal Simics licenses

- Simics machine-simulator software is licensed
- We have enough “seats” for the class
  - Should work on most CMU-network machines
  - Will *not* work on most non-CMU-network machines
- Options
  - CMU “Address extension” service (non-encrypted VPN)
  - “Personal academic license” for a personal Linux box
    - » locked to your personal machine (MAC address)
    - » apply at [www.simics.net](http://www.simics.net) (top right of page)

# Synchronization

## Simics on Windows?

- Simics simulator itself is available for Windows
- 15-410 build/debug infrastructure is not

## Options

- Dual-boot, VMware
- Usability via X depends on network latency
  - Translation: performance ranges from “slow” to “unusably slow”
- Port to cygwin (may be non-trivial)
- There *are* those Andrew cluster machines...

# Outline

**Computer hardware**

**CPU State**

**Fairy tales about system calls**

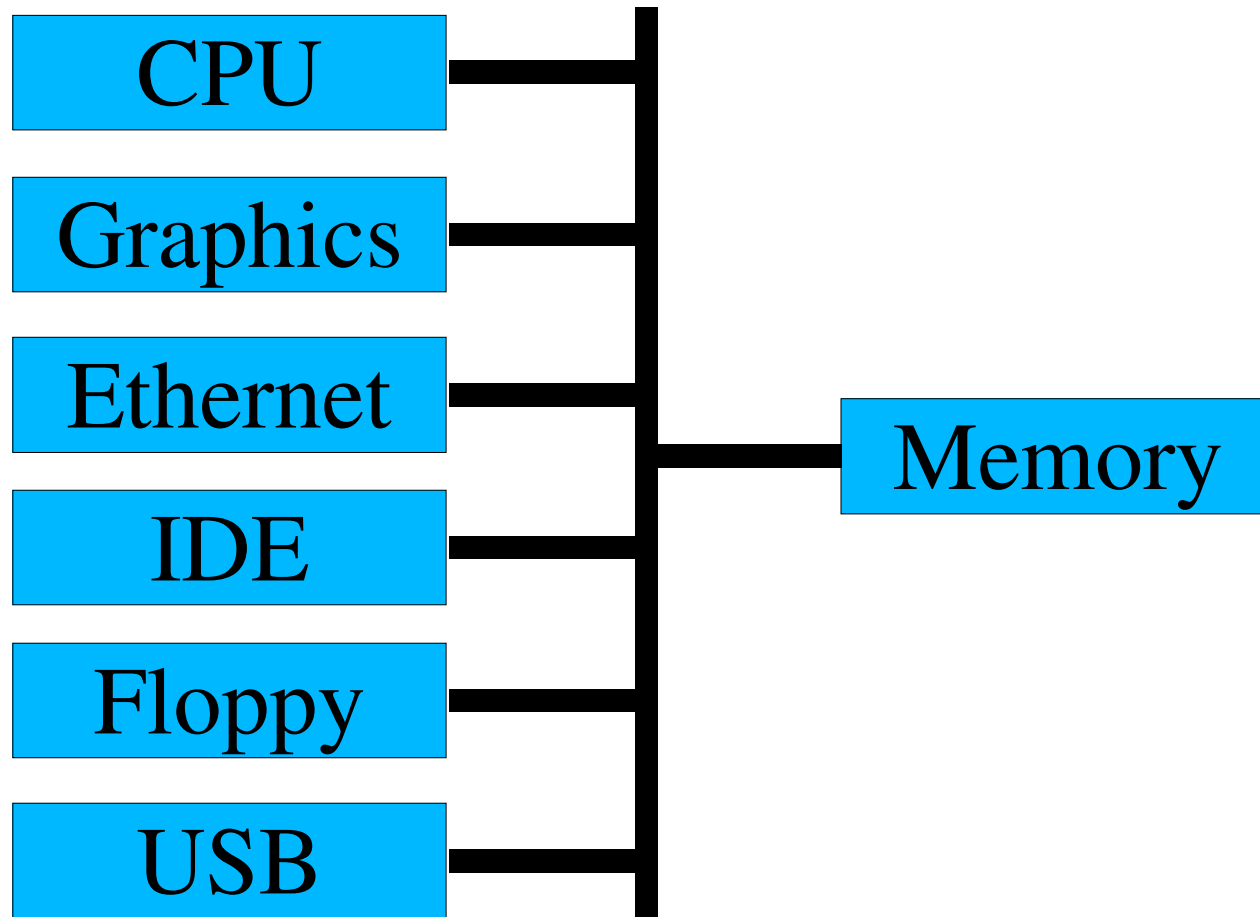
**CPU context switch (intro)**

**Interrupt handlers**

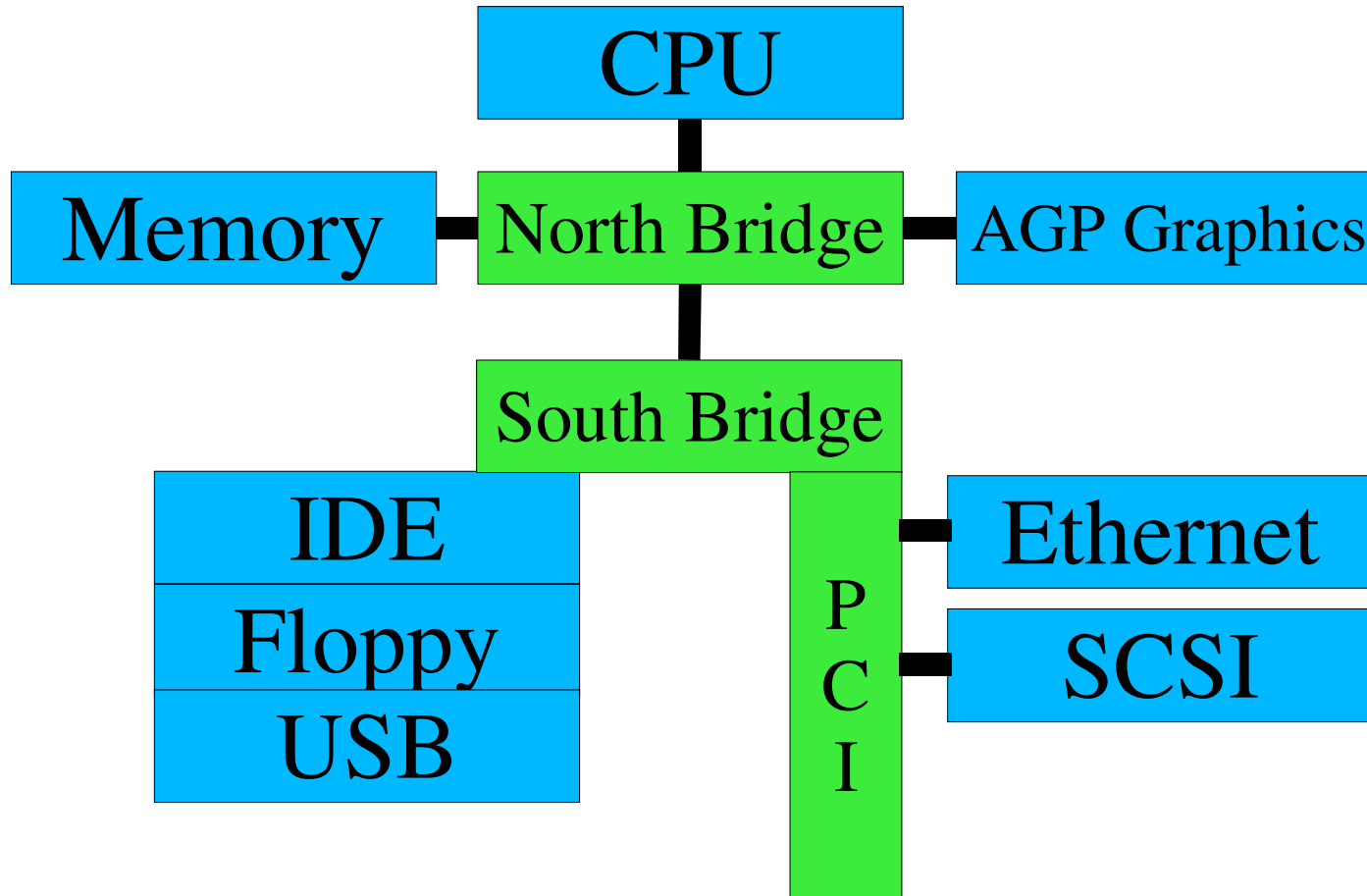
**Interrupt masking**

**Sample hardware device – countdown timer**

# Inside The Box - Historical/Logical



# Inside The Box - Really



# CPU State

## User registers (on Planet Intel)

- General purpose - %eax, %ebx, %ecx, %edx
- Stack Pointer - %esp
- Frame Pointer - %ebp
- Mysterious String Registers - %esi, %edi



# CPU State

*Non-user* registers, a.k.a....

## Processor status register(s)

- Currently running: user code / kernel code?
- Interrupts on / off
- Virtual memory on / off
- Memory model
  - small, medium, large, purple, dinosaur

# CPU State

## Floating Point Number registers

- Logically part of “User registers”
- Sometimes another “special” set of registers
  - Some machines don't have floating point
  - Some processes don't use floating point

# Story time!

## Time for some fairy tales

- The getpid() story (shortest legal fairy tale)
- The read() story (toddler version)
- The read() story (grade-school version)

# The Story of getpid()

## User process is computing

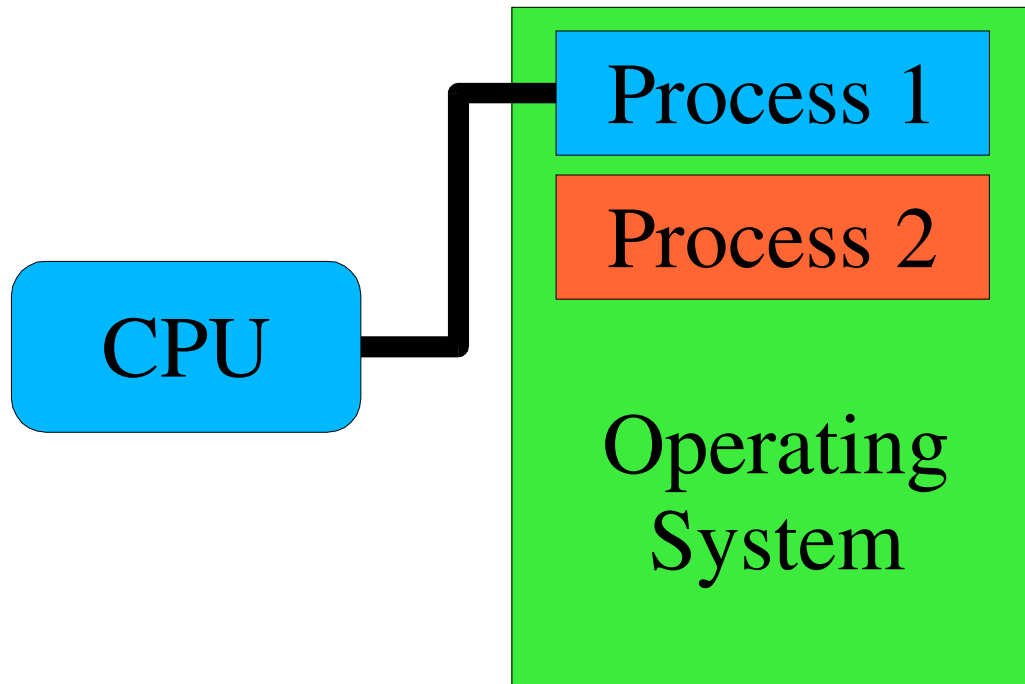
- User process calls getpid() library routine
- Library routine executes TRAP \$314159
  - In Intel-land, TRAP is called “INT” (because it isn't one)
    - » REMEMBER: “INT” is *not an interrupt*

## The world changes

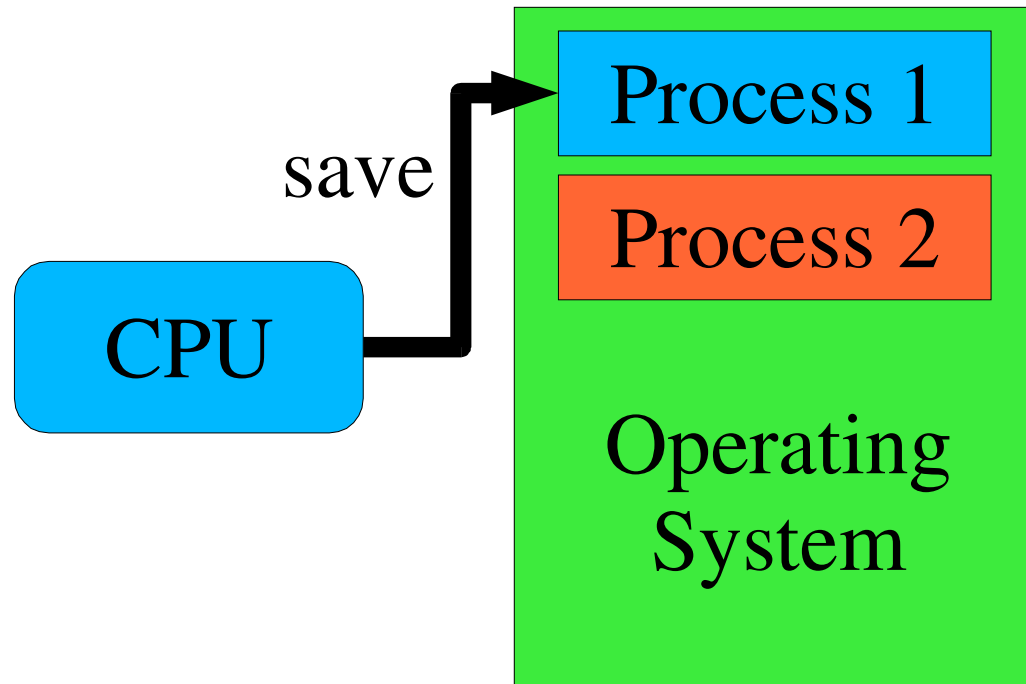
- Some registers dumped into memory somewhere
- Some registers loaded from memory somewhere

## The processor has *entered kernel mode*

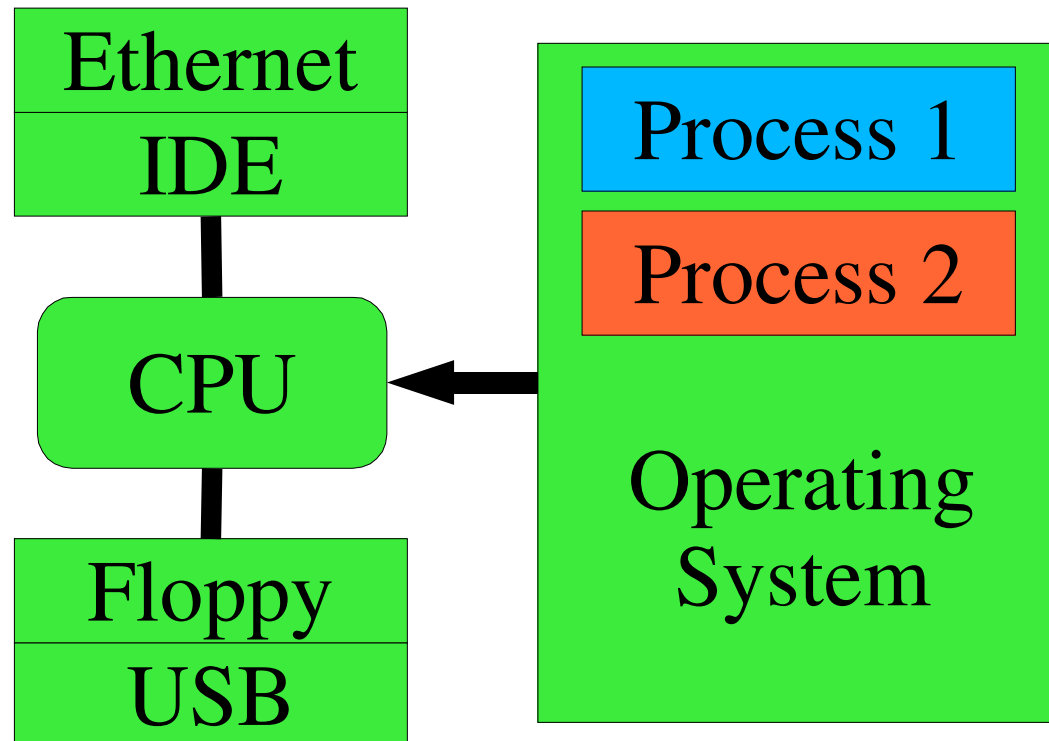
# User Mode



# Entering Kernel Mode



# Entering Kernel Mode



# The Kernel Runtime Environment

## Language runtimes differ

- ML: may be no stack (“nothing but heap”)
- C: stack-based

## Processor is more-or-less agnostic

- Some assume/mandate a stack

## Trap handler builds kernel runtime environment

### Depending on processor

- Switches to correct stack
- Saves registers
- Turns on virtual memory
- Flushes caches



# The Story of getpid()

## Process in kernel mode

- `running->u_reg[R_EAX] = running->u_pid;`

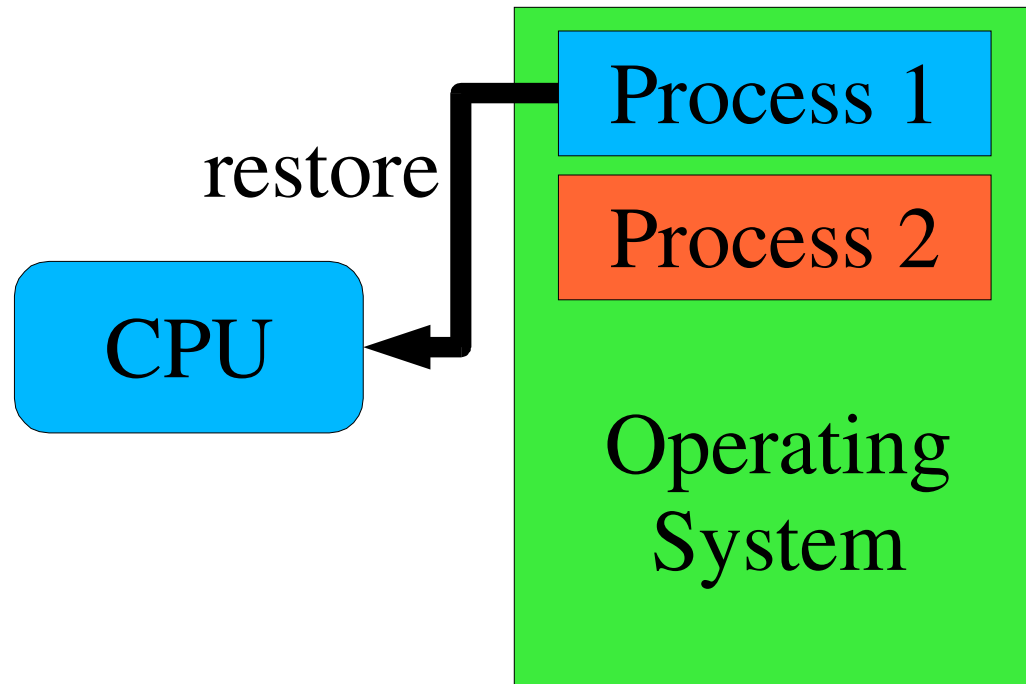
## Return from interrupt

- Processor state restored to user mode
  - (modulo `%eax`)

## User process returns to computing

- Library routine returns `%eax` as value of `getpid()`

# Returning to User Mode



# The Story of getpid()

## What's the getpid() system call?

- C function you call to get your process ID
- “Single instruction” which modifies %eax
- Privileged code which can access OS internal state

# A Story About read()

**User process is computing**

```
count = read(7, buf, sizeof (buf));
```

**User process “goes to sleep”**

**Operating system issues disk read**

**Time passes**

**Operating system copies data to user buffer**

**User process “wakes up”**

# Another Story About read()

**P1: read()**

- Trap to kernel mode

**Kernel: tell disk: “read sector 2781828”**

**Kernel: switch to running P2**

- Return to user mode - but to P2, not P1!
- P1 is “blocked in system call”
  - Part-way through driver code
  - Marked “unable to execute more instructions”

**P2: compute 1/3 of Mandelbrot set**

# Another Story About read()

## Disk: done!

- Asserts “interrupt request” signal
- CPU stops running P2's instructions
- Interrupt to kernel mode
- Run “disk interrupt handler” code

## Kernel: switch to P1

- Return from interrupt - but to P1, not P2!
- P2 is able to execute instructions, but not doing so

# Interrupt Vector Table

**How should CPU handle *this particular* interrupt?**

- Disk interrupt ⇒ invoke disk driver
- Mouse interrupt ⇒ invoke mouse driver

## **Need to know**

- Where to dump registers
  - Often: property of current process, not of interrupt
- New register values to load into CPU
  - Key: new program counter, new status register
    - » These define the new execution environment

# Interrupt Dispatch/Return

## Table lookup

- Interrupt controller says: this is interrupt source #3
- **CPU fetches table entry #3**
  - Table base-pointer programmed in OS startup
  - Table-entry size defined by hardware

## Save old processor state

## Modify CPU state according to table entry

## Start running interrupt handler

## “Return from interrupt” process

- Load saved processor state back into registers
- Restoring program counter reactivates “old” code



# Example: x86/IA32

## CPU saves old processor state

- Stored on “kernel stack”

## CPU modifies state according to table entry

- Loads new privilege information, program counter

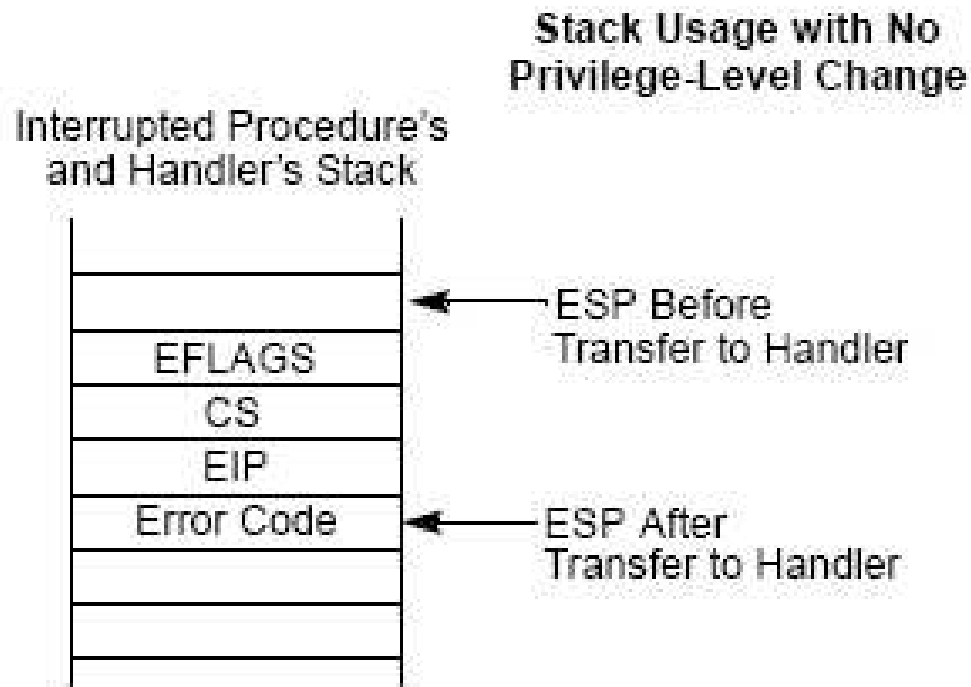
## Interrupt handler begins

- Uses kernel stack for its own purposes

## Interrupt handler completes

- Empties stack back to original state
- Invokes “interrupt return” (IRET) instruction
  - Registers loaded from kernel stack
  - Mode switched from “kernel” to “user”

# IA32 Single-Task Mode Example



From intel-sys.pdf

**Picture: Interrupt/Exception while in kernel mode (Project 1)**

**Hardware pushes registers on current stack, NO STACK CHANGE**

- EFLAGS (processor state)
- CS/EIP (return address)
- Error code (certain interrupts/faults, not others: see intel-sys.pdf)
- IRET restores state from EIP, CS, EFLAGS

# Race Conditions

## Two concurrent activities

- Computer program, disk drive

## Various execution sequences produce various “answers”

- Disk interrupt *before* or *after* function call?

## Execution orders are not controlled

- So either outcome is possible “randomly”

## System produces random “answers”

- One answer or another “wins the race”

# Race Conditions – Disk Device Driver

## “Top half” wants to launch disk-I/O requests

- If disk is idle, send it the request
- If disk is busy, queue request for later

## Interrupt handler action depends on queue status

- Work in queue  $\Rightarrow$  feed disk queued request
- Queue empty  $\Rightarrow$  let disk go idle

## Various outcomes possible

- Disk interrupt *before* or *after* “queue empty” test?

## System produces random “answers”

- Work in queue  $\Rightarrow$  transmit next request
- Work in queue  $\Rightarrow$  let disk go idle

# Race Conditions – Driver Skeleton

```
dev_start(request) {
    if (device_idle)
        start_device(request);
    else
        enqueue(request);
}
dev_intr() {
    ...finish up previous request...
    if (new_request = head()) {
        start_device(new_request);
    } else
        device_idle = 1;
}
```

# Race Conditions – Good Case

<i>User process</i>	<i>Interrupt handler</i>
<small>aaaaa</small> <code>if (device_idle)</code>	
<code>/* no, so... */</code>	
<code>enqueue(request)</code>	
	<b>INTERRUPT</b>
	<b>...clean up...</b>
	<b>start_device</b>
	<b>(new_request);</b>
	<b>RETURN FROM</b>
	<b>INTERRUPT</b>

# Race Conditions – Bad Case

<small>aaaaa</small> <i>User process</i>	<i>Interrupt handler</i>
<code>if (device_idle)</code>	
<code>/* no, so... */</code>	
	<b>INTERRUPT</b>
	<b>...</b>
	<code>device_idle = 1;</code>
	<b>RETURN FROM INTERRUPT</b>
<code>enqueue (request)</code>	

# What Went Wrong?

## “Top half” ran its algorithm

- Examine state
- Commit to action

## Interrupt handler ran *its* algorithm

- Examine state
- Commit to action

## Various outcomes possible

- Depends on exactly when interrupt handler runs

## System produces random “answers”

- Study, avoid this in your P1!



# Interrupt Masking

## Two approaches

- Temporarily suspend/mask/defer device interrupt while checking and enqueueing
  - Will cover further before Project 1
- Or use a lock-free data structure
  - [left as an exercise for the reader]

## Considerations

- **Avoid blocking *all* interrupts**
  - [not a big issue for 15-410]
- **Avoid blocking too long**
  - Part of Project 1, 3 grading criteria

# Timer – Behavior

## Simple behavior

- Count something
  - CPU cycles, bus cycles, microseconds
- When you hit a limit, signal an interrupt
- Reload counter to initial value
  - Done “in background” / “in hardware”
  - (Doesn't wait for software to do reload)

## Summary

- No “requests”, no “results”
- Steady stream of evenly-distributed interrupts

# Timer – Why?

## Why interrupt a perfectly good execution?

### Avoid CPU hogs

```
while (1)
    continue;
```

### Maintain accurate time of day

- Battery-backed calendar counts only seconds (poorly)

### Dual-purpose interrupt

- Timekeeping  

```
++ticks_since_boot;
```
- Avoid CPU hogs: force process switch

# Summary

**Computer hardware**

**CPU State**

**Fairy tales about system calls**

**CPU context switch (intro)**

**Interrupt handlers**

**Interrupt masking**

**Sample hardware device – countdown timer**