15-410

"Computers make very fast, very accurate mistakes."
--Brandon Long

Hardware Overview
Sep. 4, 2015

Dave Eckhardt

Synchronization

Today's class

- Not exactly OSC Chapter 2 or 13
- Not exactly OS:P+P Chapter 2, Section 3.0/3.5

Upcoming

- Project 1
- Lecture on “The Process”

Outline

Computer hardware

CPU State

Fairy tales about system calls

CPU context switch (intro)

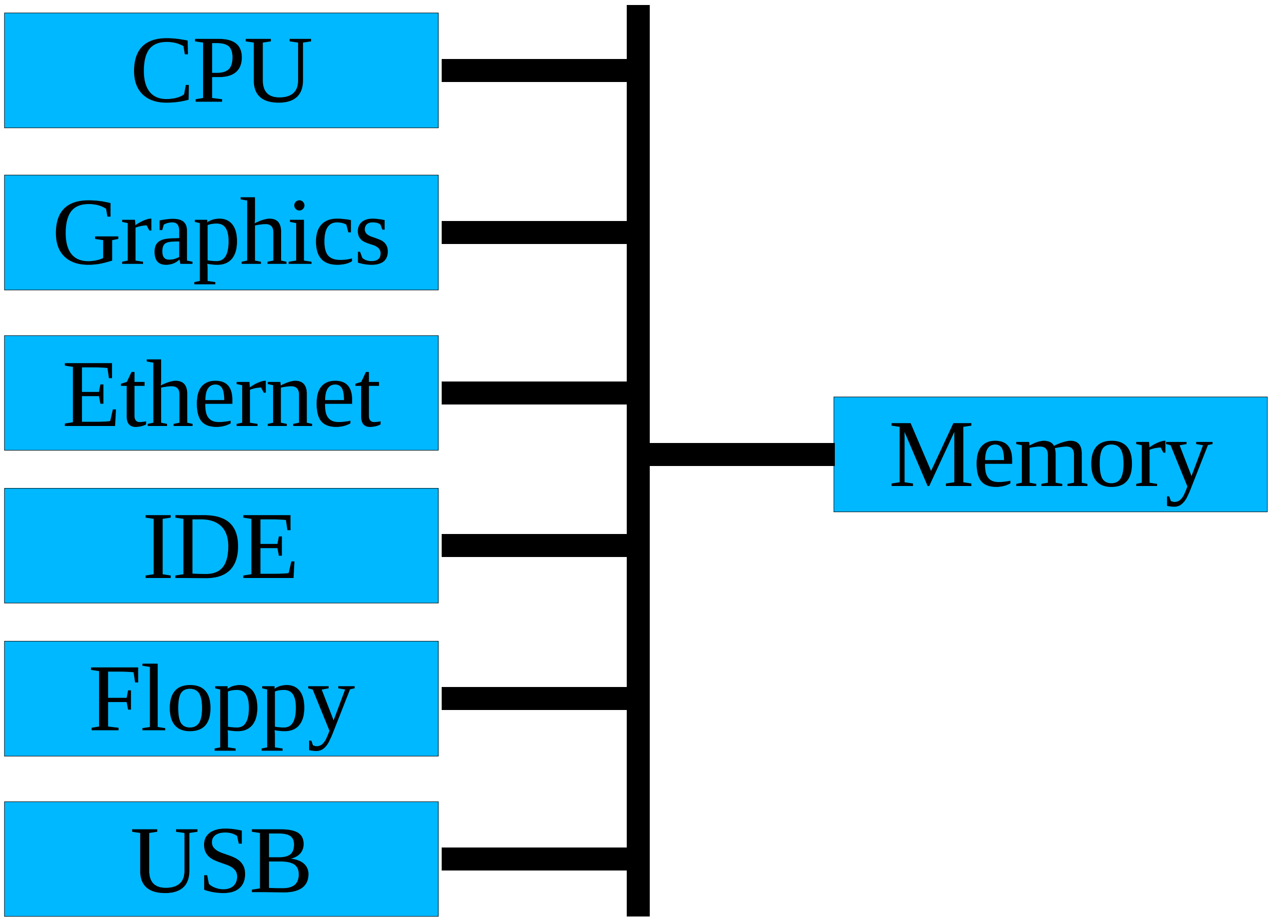
Interrupt handlers

Race conditions

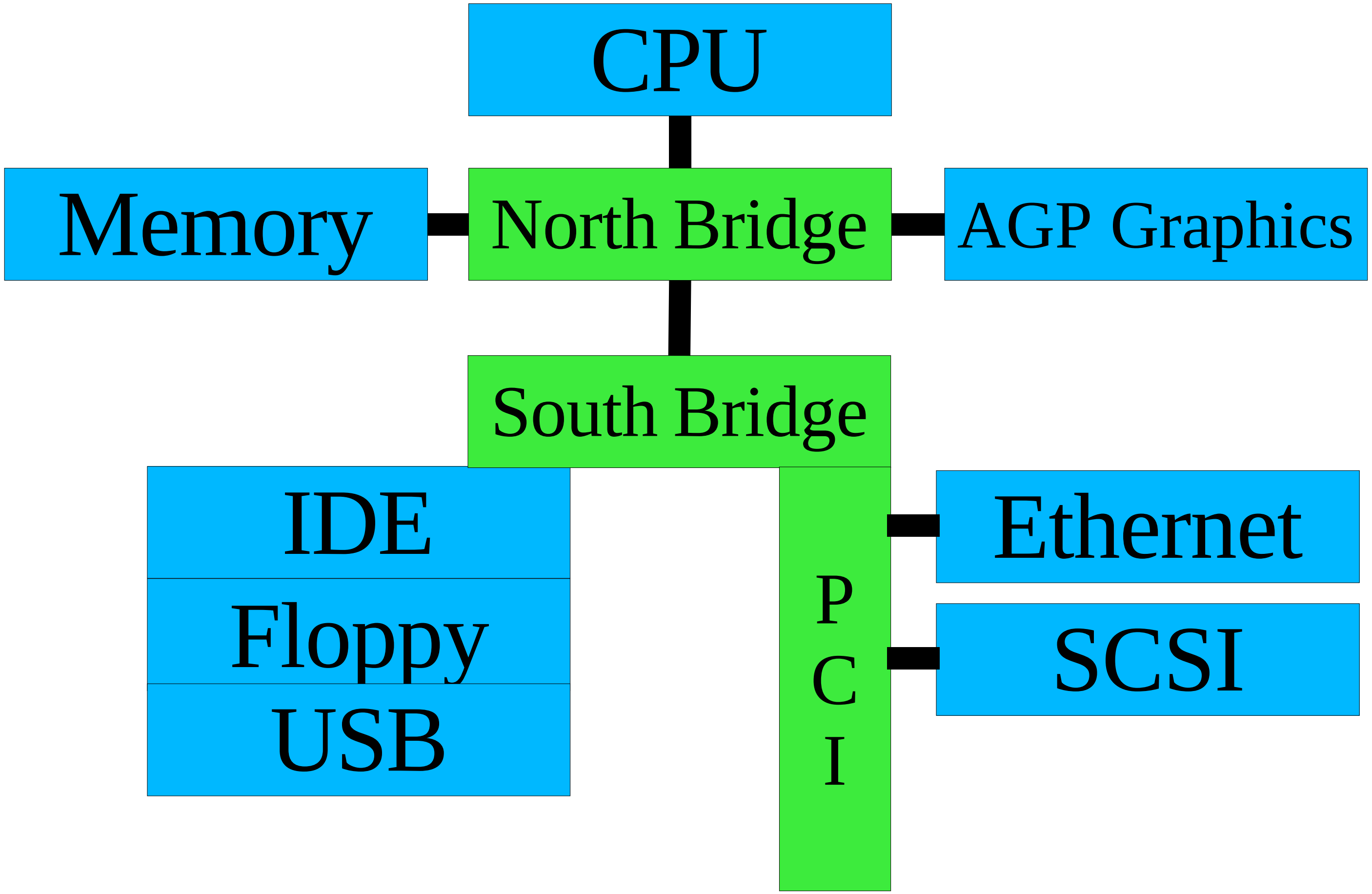
Interrupt masking

Sample hardware device – countdown timer

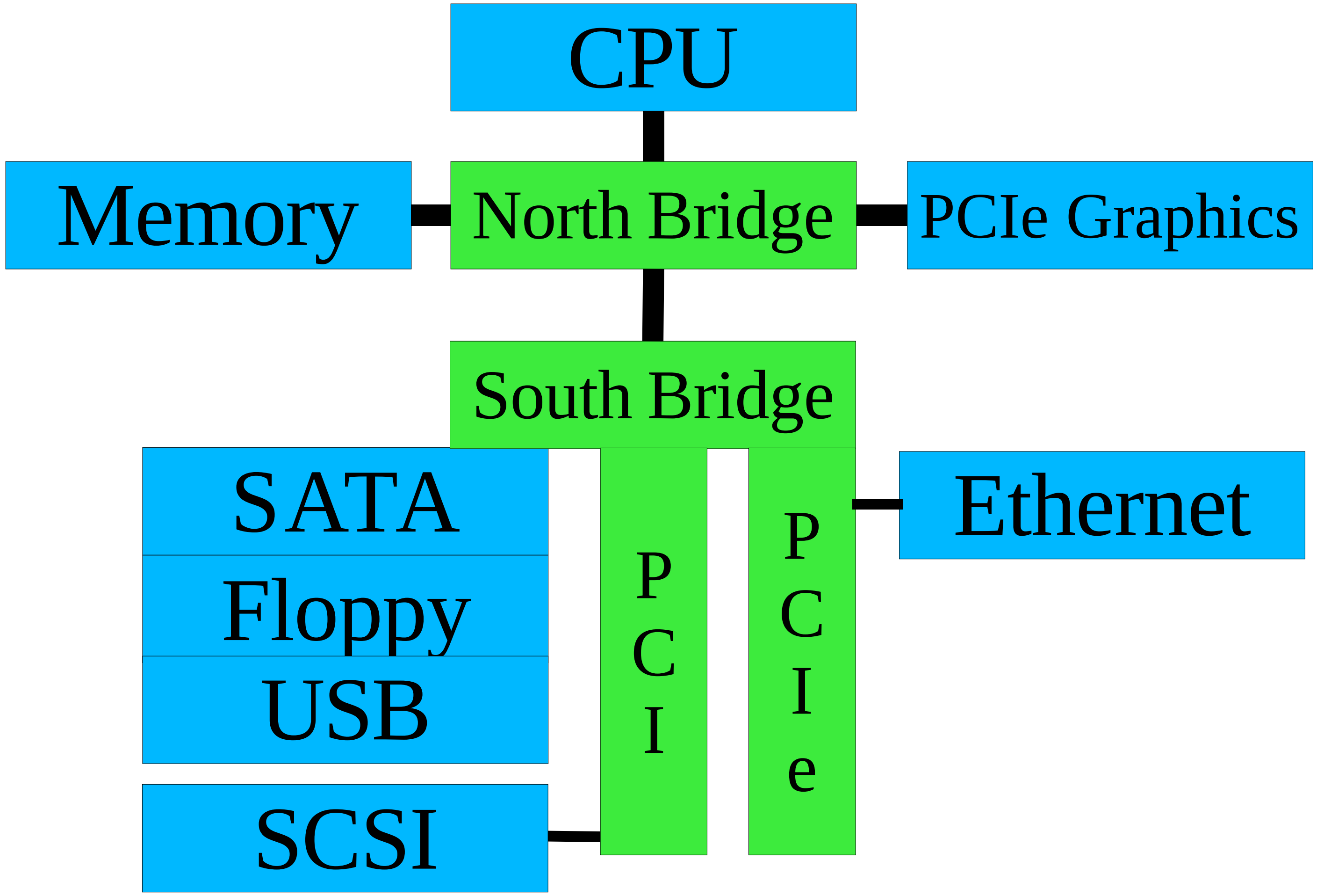
Inside The Box - Historical/Logical



Inside The Box - 1997-2004



Inside The Box - 2004-



CPU State

User registers (on Planet IA32)

- 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

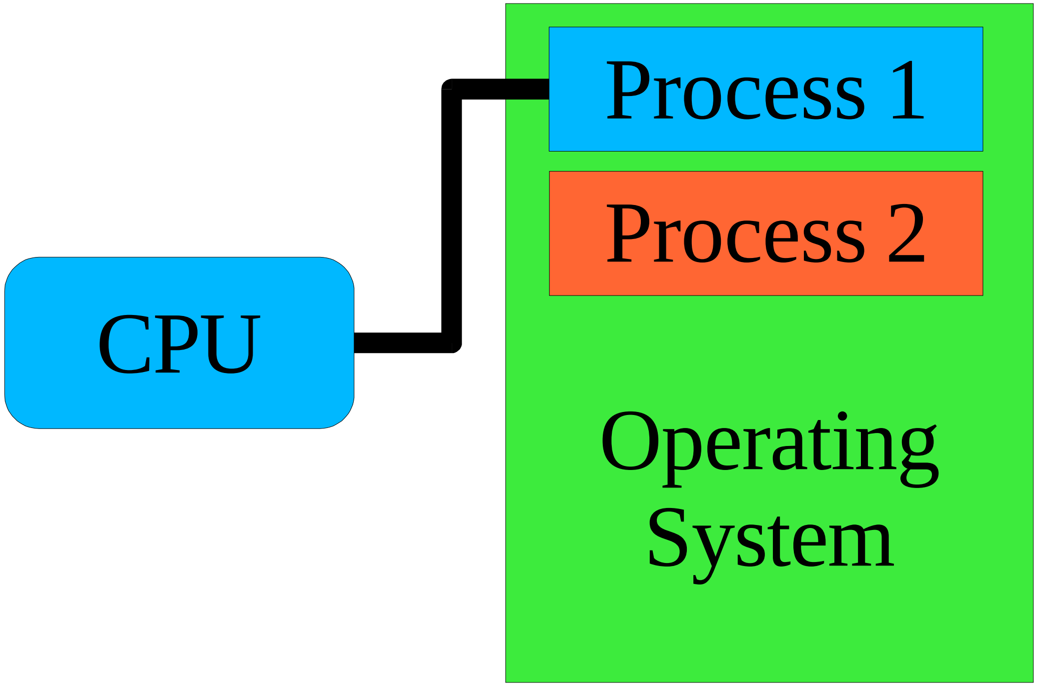
- User process calls getpid() library routine
- Library routine executes `TRAP $314159`
  - In Intel-land, `TRAP` is called "`INT`" (because it isn't one)
    - » REMEMBER: "`INT`" is *not an interrupt*

The world changes

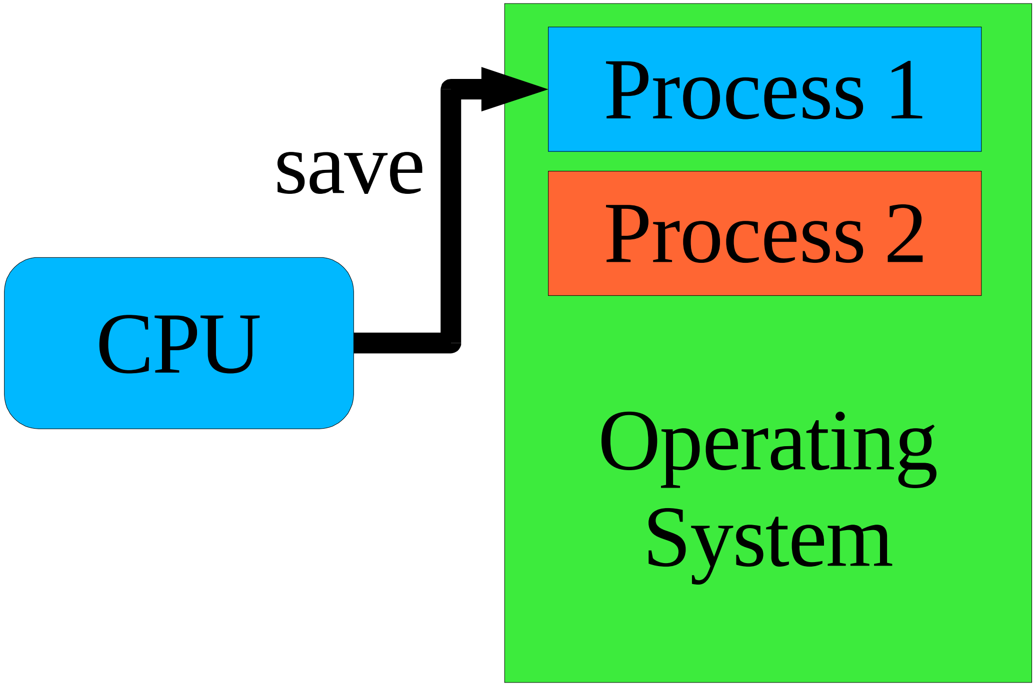
- Some registers dumped into memory somewhere
- Some registers loaded from memory somewhere

The processor has *entered kernel mode*

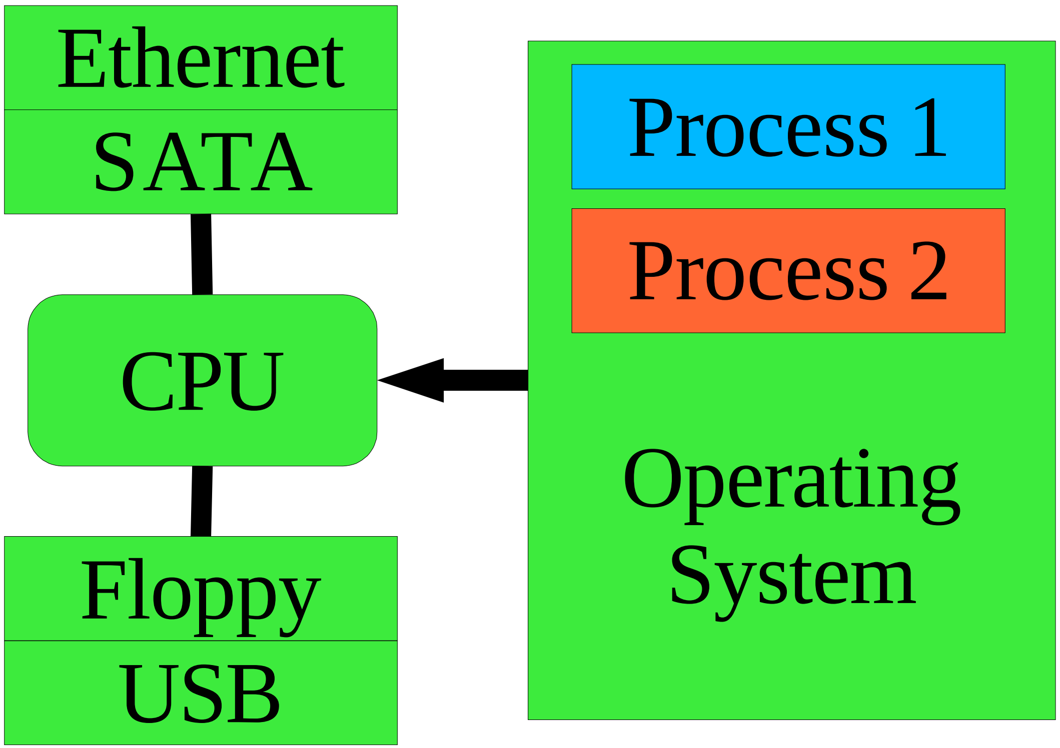
User Mode



Entering Kernel Mode



Entering Kernel Mode



The Kernel Runtime Environment

Language runtimes differ

- ML: may have 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 runs in kernel mode

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

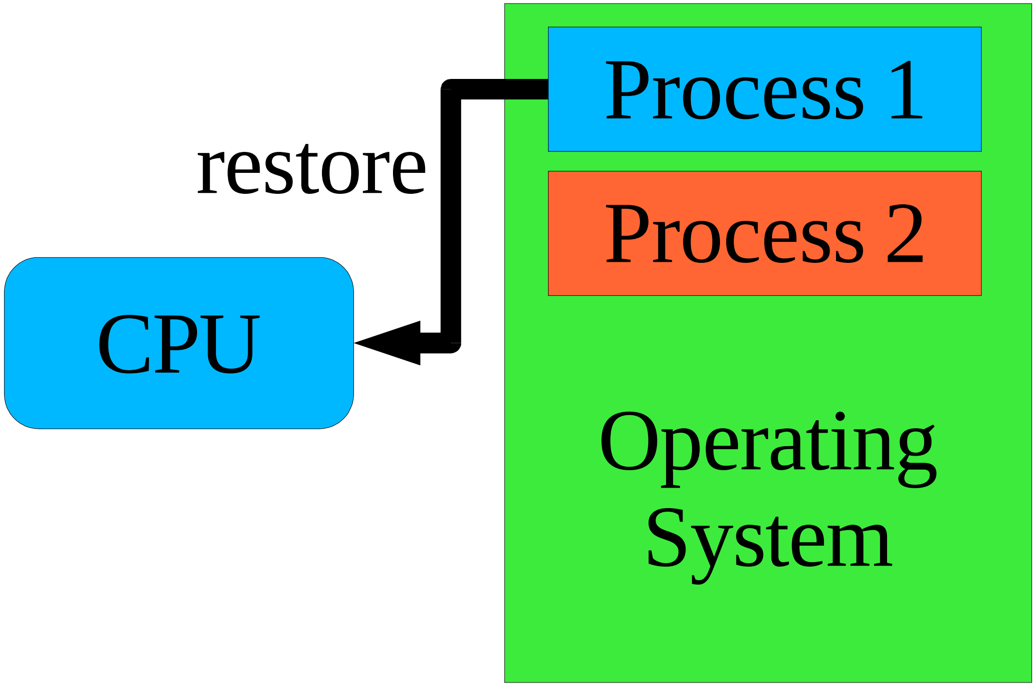
"Return from interrupt"

- 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" (`INT`) 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 "stops running"

Operating system issues disk read

Time passes

Operating system copies data to user buffer

User process "starts running again"

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 a system call"
  - P1's %eip is somewhere in the kernel
    - » (details later)
  - 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
- Interrupts to kernel mode
- Runs "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
  - P2 is not running
  - But it is not "blocked"
  - It is "runnable"

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

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

Interrupt Return

"Return from interrupt" operation

- **Load saved processor state back into registers**
- **Restoring program counter reactivates "old" code**
- **Hardware instruction typically restores some state**
- **Kernel code must restore the remainder**

Example: x86/IA32

CPU saves old processor state

- Stored on "kernel stack" (picture follows)

CPU modifies state according to table entry

- Loads new privilege information, program counter

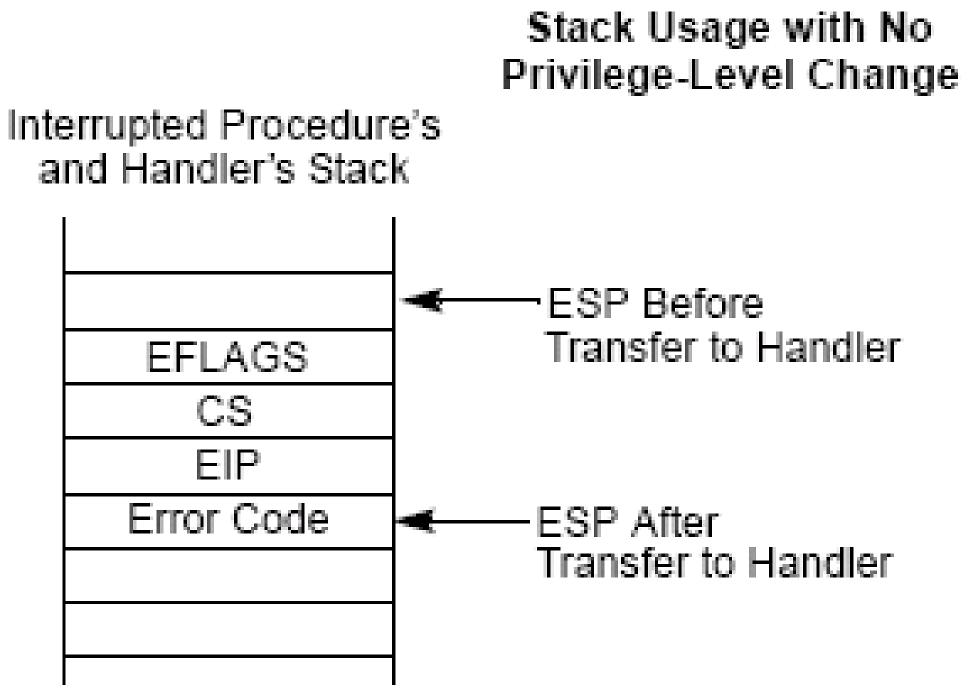
Interrupt handler begins

- 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 may switch from "kernel" to "user"
    - » Also possible to not-switch from kernel to kernel

IA32 Single-Task Mode Example



From intel-sys.pdf (please consult!)

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

1. **Two concurrent activities**
   - Computer program, disk drive
2. **Various execution sequences produce various "answers"**
   - Disk interrupt *before* or *after* function call?
3. **Execution sequence is 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 ⇒ transmit next request to disk
- Queue empty ⇒ let disk go idle

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 ⇒ transmit next request to disk
- Queue empty ⇒ let disk go idle

Various execution orders possible

- Disk interrupt *before* or *after* "disk is idle" test?

System produces random "answers"

- "Work in queue ⇒ transmit next request" (good)
- "Work in queue ⇒ let disk go idle" (what??)

Race Conditions – Driver Skeleton

```
dev_start(request) {
  if (device_idle) {
    device_idle = 0;
    send_device(request);
  } else {
    enqueue(request);
  }
}
dev_intr() {
  ...finish up previous request...
  if (new_request = head()) {
    send_device(new_request);
  } else
    device_idle = 1;
}
```

Race Conditions – Good Case

User process	*Interrupt handler*
`if (device_idle)`	
`/* no, so... */`	
`enqueue(request)`	
	`INTERRUPT`
	`...finish up...`
	`new = 0x80102044;`
	`send_device(new);`
	`RETURN FROM INTERRUPT`

Race Conditions – Bad Case

User process	*Interrupt handler*
`if (device_idle)`	
`/* no, so... */`	
	`INTERRUPT`
	`..finish up...`
	`new = 0;`
	`device_idle = 1;`
	`RETURN FROM INTERRUPT`
`enqueue(request)`	

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!

What To Do?

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, Project 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

Race conditions

Interrupt masking

Sample hardware device – countdown timer