15-410

"Dude, what were you thinking?"

Hardware Overview
Jan. 19, 2004

Dave Eckhardt

Bruce Maggs

Synchronization

Today's class

- Not exactly Chapter 2 or 13

Wednesday

- Project 0 is due
- Lecture on “The Process”

Outline

Computer hardware

CPU State

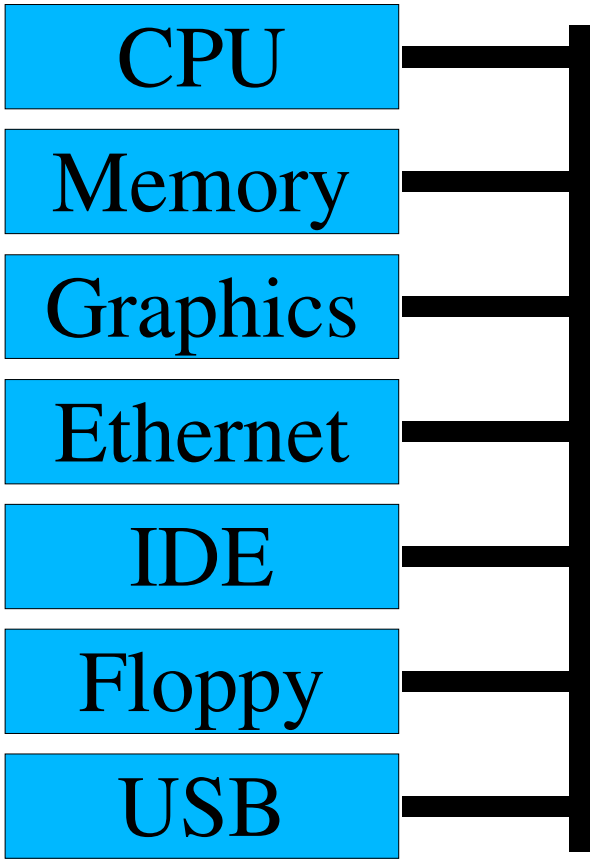
Fairy tales about system calls

CPU context switch (intro)

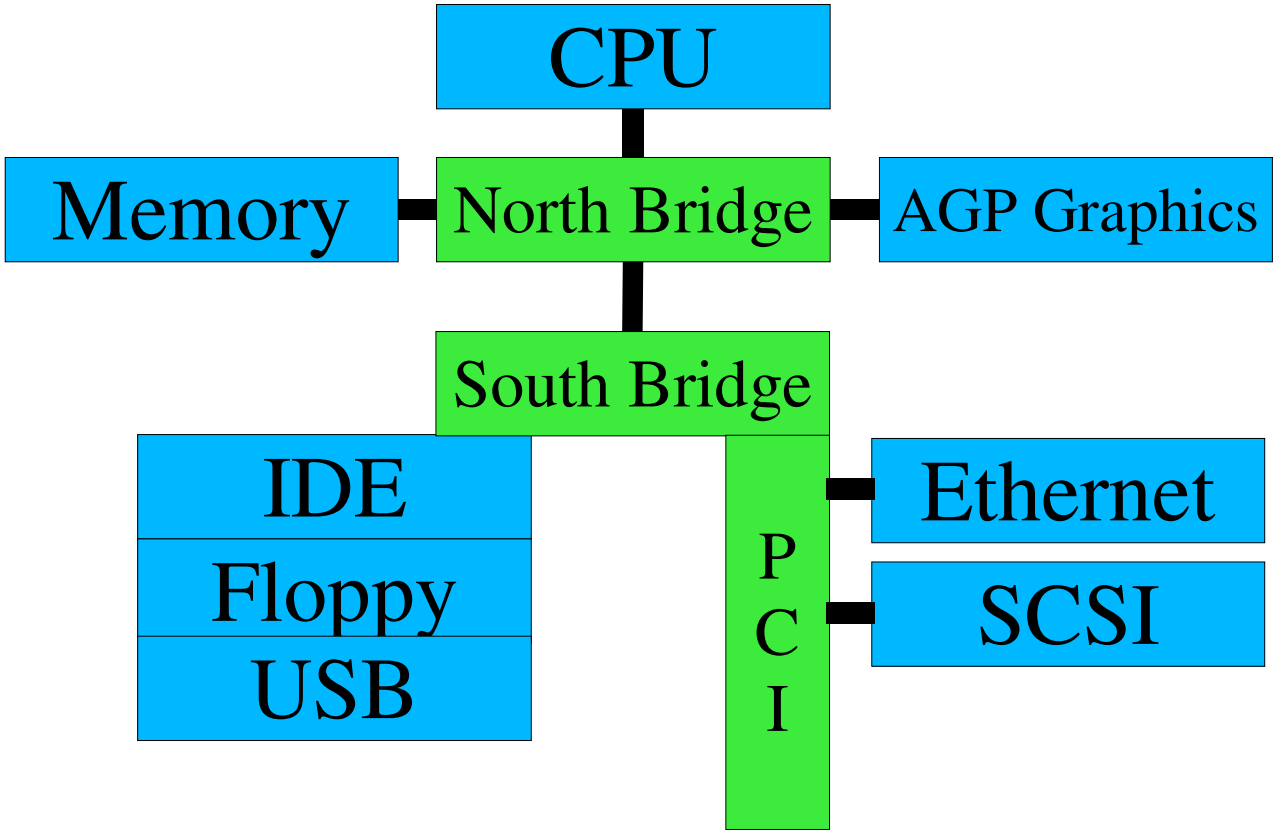
Interrupt handlers

Interrupt masking

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)

- **User process / Kernel process**
- **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 special instead
  - 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

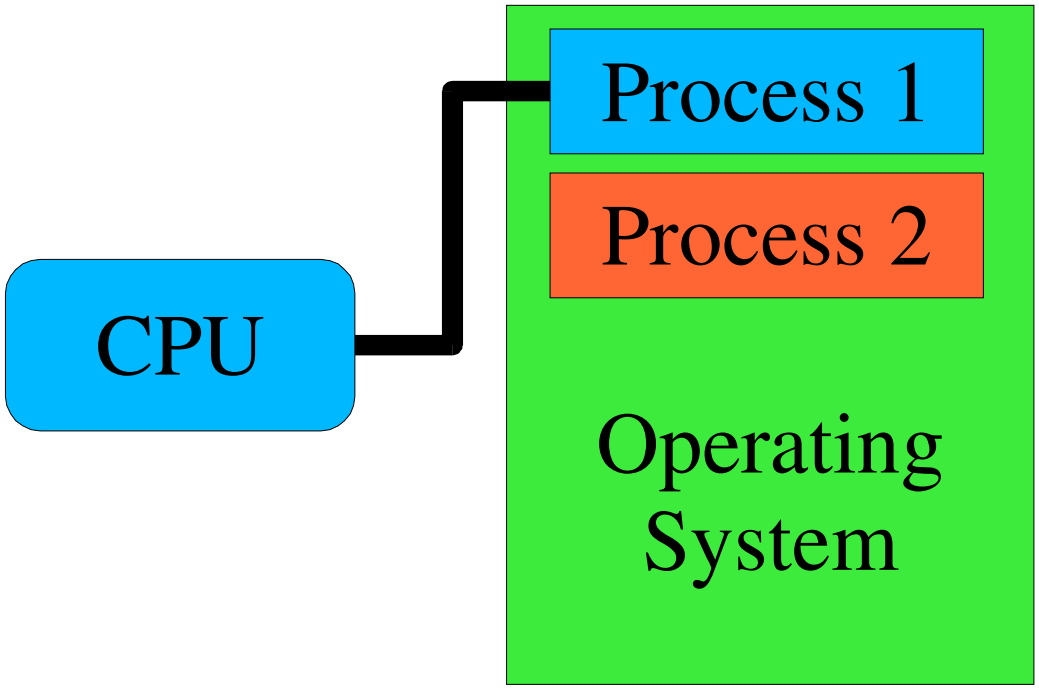
- User process calls getpid() library routine
- Library routine executes TRAP(314159)

The world changes

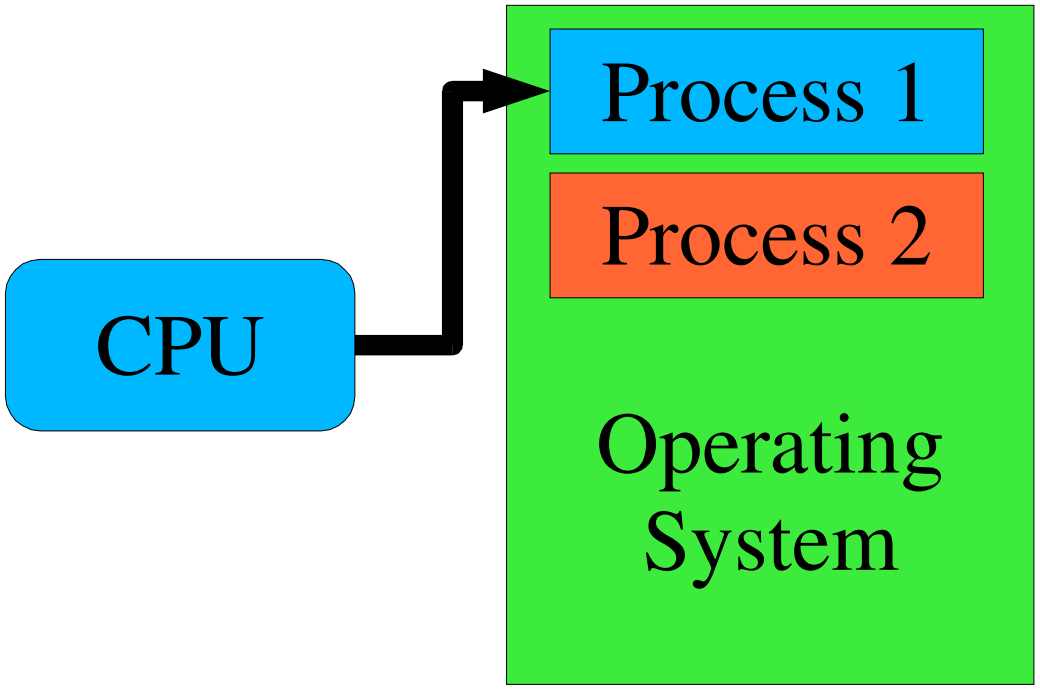
- Some registers dumped into memory somewhere
- Some registers loaded from memory somewhere

The processor has *entered kernel mode*

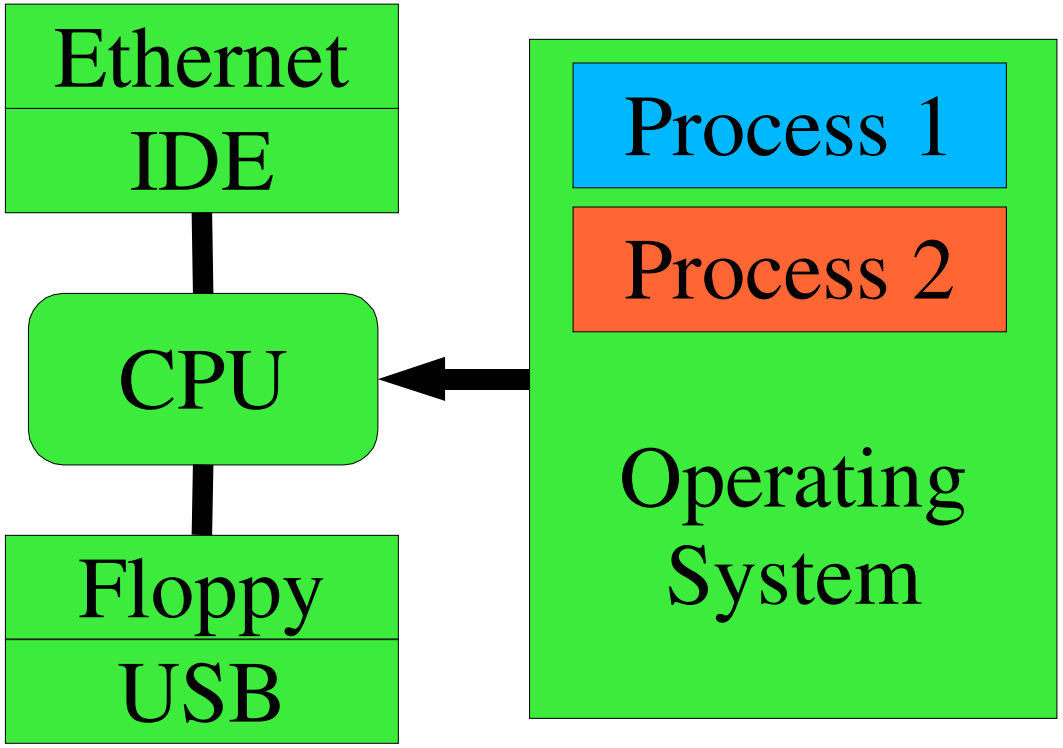
User Mode



Entering Kernel Mode



Entering Kernel Mode



The Kernel Runtime Environment

Language runtimes differ

- ML: no stack, “nothing but heap”
- C: stack-based

Processor is mostly agnostic

Trap handler builds kernel runtime environment

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

The Story of getpid()

Process in kernel mode

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

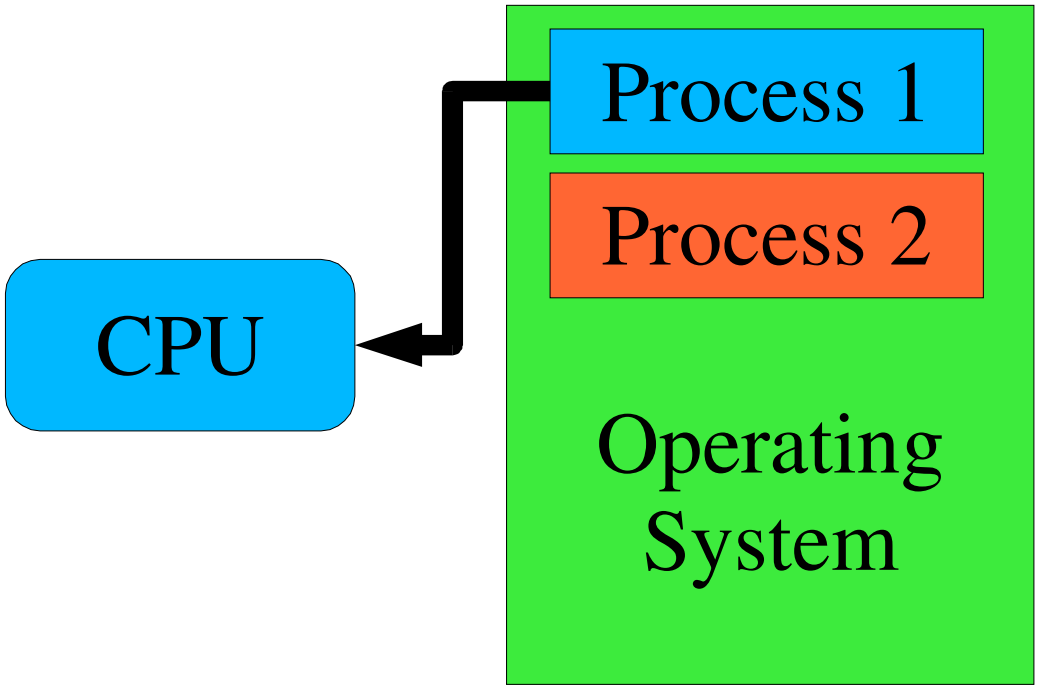
Return from interrupt

- 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



A Story About read()

User process is computing

- `count = read(0, buf, sizeof (buf));`

User process "goes to sleep"

Operating system issues disk read

Time passes

Operating system copies data

User process wakes up

Another Story About read()

P1: read()

- Trap to kernel mode

Kernel: issue disk read

Kernel: switch to P2

- Return from interrupt - but to P2, not P1!

P2: compute 1/3 of Mandelbrot set

Another Story About read()

Disk: done!

- Interrupt to kernel mode

Kernel: switch to P1

- Return from interrupt - but to P1, not P2!

Interrupt Vector Table

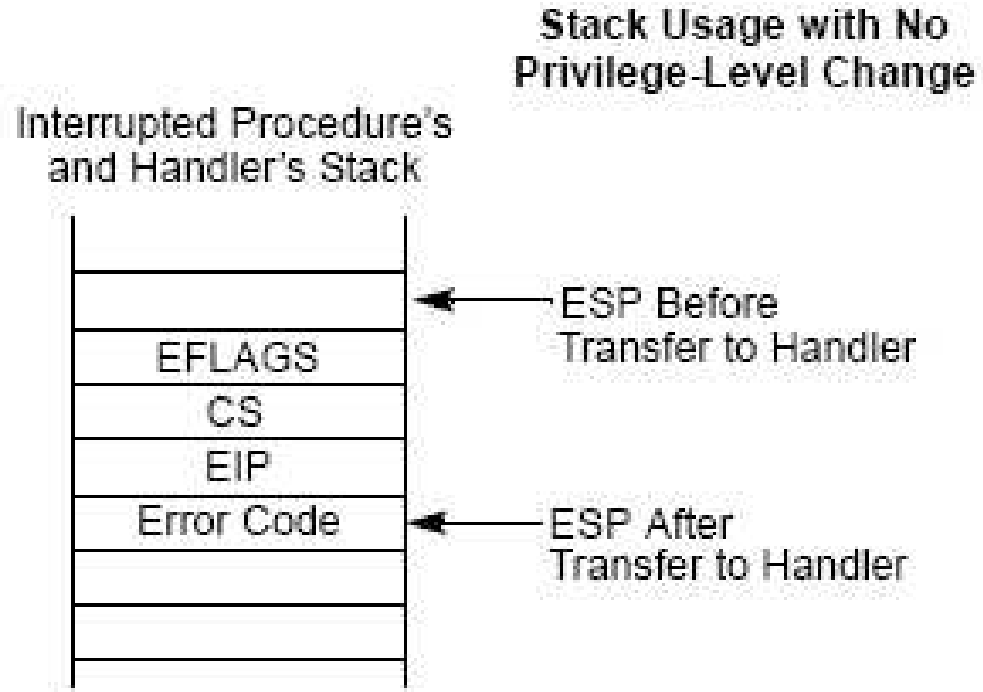
How does CPU handle *this* interrupt?

- Disk interrupt -> disk driver
- Mouse interrupt -> 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

IA32 Single-Task Mode Example



From intel-sys.pdf

- **Interrupt/Exception while in kernel mode (Project 1)**
- **Hardware pushes registers on current stack, NO STACK CHANGE**
  1. **EFLAGS (processor state)**
  2. **CS/EIP (return address)**
  3. **Error code (if interrupted exception procedure wants to)**

Interrupt Vector Table

Table lookup

- **Interrupt controller says: this is interrupt source #3**

CPU knows table base pointer, table entry size

Spew, slurp, off we go

Race Conditions

```
if (device_idle)
  start_device(request);
else
  enqueue(request);
```

Race Conditions

aaaaaa *User process*	*Interrupt handler*
`if (device_idle)`	
	`INTERRUPT`
	`...`
	`device_idle = 1;`
	`RETURN FROM INTERRUPT`
`enqueue(request)`	

Interrupt masking

Atomic actions

- Block device interrupt while checking and enqueueing
- Or use a lock-free data structure
  - [left as an exercise for the reader]

Avoid blocking *all* interrupts

- [not a big issue for 15-410]

Avoid blocking too long

- Part of Project 3 grading criteria

Timer – Behavior

Count something

- **CPU cycles, bus cycles, microseconds**

When you hit a limit, generate an interrupt

Reload counter (don't wait for software to do it)

Timer – Why?

Why interrupt a perfectly good execution?

Avoid CPU hogs

```
for (;;) ;
```

Maintain accurate time of day

- Battery-backed calendar counts only seconds (poorly)

Dual-purpose interrupt

- ++ticks_since_boot;
- force process switch (probably)