Hardware Overview

David A. Eckhardt
School of Computer Science
Carnegie Mellon University

de0u@andrew.cmu.edu

Administrative Overhead

Communication

- Web site is stirring - http://www.cs.cmu.edu/~412
- Official announcements - academic.cs.15-412
  - having it beep or wiggle might be good

"How do I stop this thing?"

- Intermission?

Today's class

- Not *exactly* Chapter 2

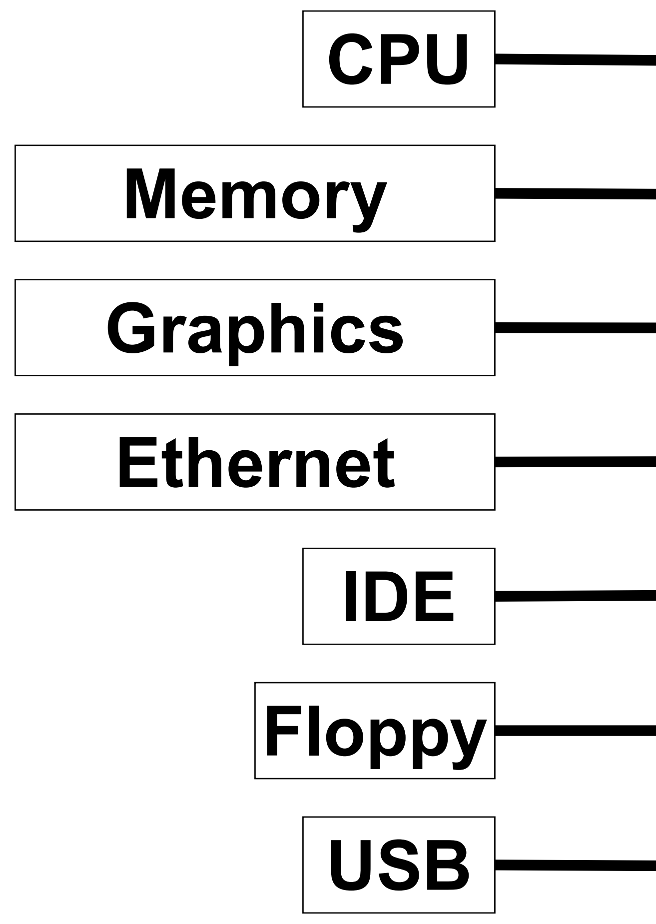
Monday's class

- Project 1 talk (good thing to attend)
- Class ends 12:00 (MLK Day activities)
- Being registered is good
  - disk space, access control lists, etc.

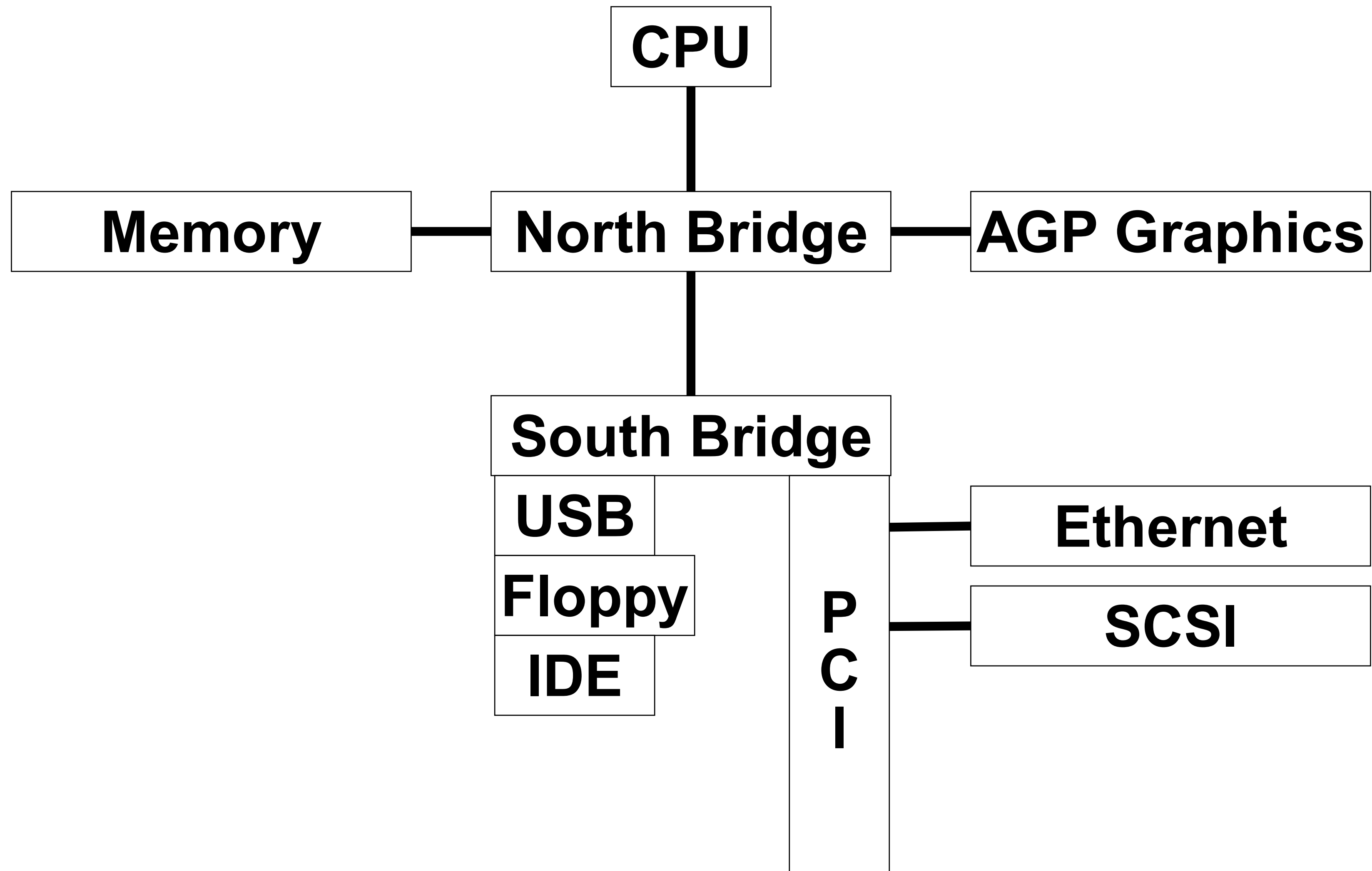
Outline

Computer parts
CPU State
Fairy tales about system calls
System memory layout
Device drivers
Interrupt Vector Table
Direct Memory Access (DMA)

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

Non-user registers

- Processor status register(s)
  - User process / Kernel process
  - Interrupts on / Interrupts off
  - Memory model (small, medium, large, purple, dinosaur)

Floating Point Number registers

- Logically part of “User registers”
- Sometimes “special” instead
  - Maybe this machine doesn’t *have* floating point
  - Maybe most 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 calls TRAP(314159)

The world changes

- Some registers dumped into memory somewhere
- Some registers loaded from memory (somewhere else)
- Trap handler builds kernel runtime environment

Process “in kernel mode”

- u.u_reg[R_EAX] = u.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()

A story about read()

User process is computing

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

User process “goes to sleep”

Operating system issues the disk read

Time passes

Disk operation complete

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

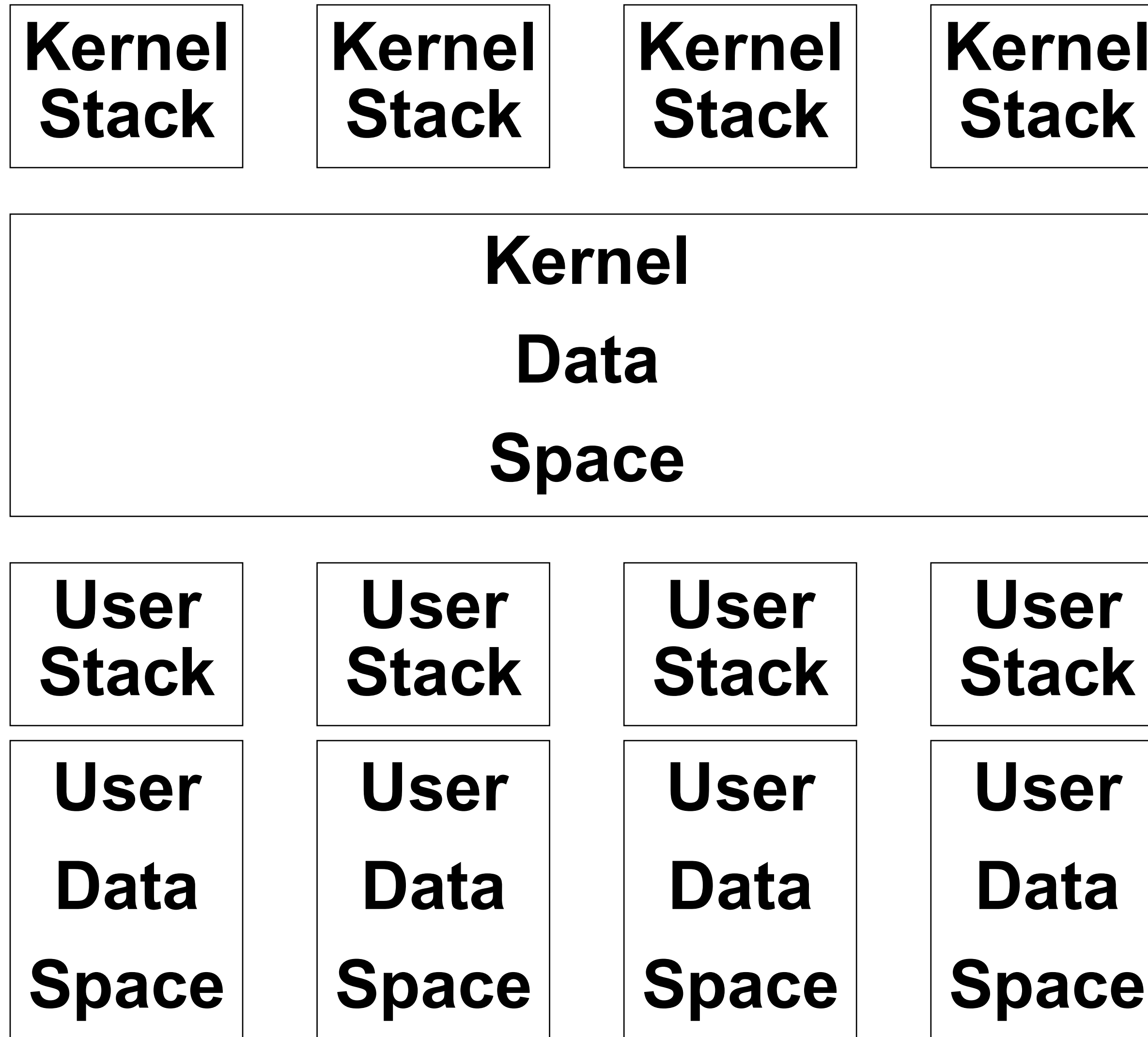
Disk: done!

- Interrupt to "kernel mode"

Kernel: switch to P1

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

The Big Picture



Thought experiments

What is the data flow for getpid()?

How does a message get between processes?

- Start: P1's stack
- ...whoosh, whoosh...
- Finish: P2's user data space

Why does *every* process have a kernel stack?

- On a uniprocessor, isn't one k-stack enough?
    - Only one process is "in kernel mode" at once, right?

What's a "device driver"?

Opposite of a user process

- Runs "forever"
- Three "threads" (but two *different* execution environments)

Initiation

- if (device_idle) start_device(request)
- else enqueue(request);
- condition_wait(request); /* switch to another process */

Interrupt handler

- condition_signal(cur_request);
- if (cur_request = queue_next()) start_device(cur_request);

Cleanup

- Transfer results from request buffer to user memory
- Return from trap

Interrupt Vector Table

How do I 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

Table lookup

- Interrupt controller says: this is interrupt source #3
- CPU knows table base pointer, table entry size
  - spew, slurp, off we go

Interrupt masking

"Race condition"

- First attempt
  - if (device_idle) start_device(request)
  - else enqueue(request);
- What about:
  - if (device_idle)
  - *INTERRUPT*...device_idle = 1;...*RETURN*
  - enqueue(request)
- Result: no initiation, so no completion

Atomic actions

- Block device interrupt while checking and enqueueing
- Avoid blocking *all* interrupts
- Avoid blocking "too long"

Direct Memory Access (DMA)

Moving the bits manually

- while (cnt--) *p++ = in_word(drive->data_port);
- Disk sector: 512 bytes = 128 32-bit words
- Disks *like* multi-sector I/O operations
- I/O bus is slower than memory bus
- So: sipping kilobytes over the bus will occupy the CPU

Real DMA

- Tell disk controller where your buffer is
- Disk controller stores words into memory
  - one-by-one: “cycle stealing”
- Legacy IBM PC DMA
  - A few “DMA channels”
  - CPU sets: pointer, length
  - Device says “here’s a word for channel 3”
  - Devices are cheap, but concurrency limited

The 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)

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;
- force process switch (probably)