CS 61C:
Great Ideas in Computer Architecture

Lecture 22:
Operating System

Bernhard Boser & Randy Katz

http://inst.eecs.berkeley.edu/~cs61c

CS61C so far…

C Programs

```
#include <stdlib.h>

int fib(int n) {
  return
    fib(n-1) +
    fib(n-2);
}
```

Project 1

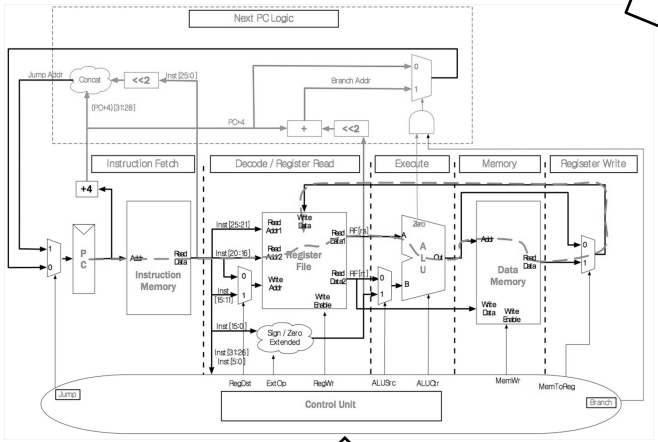
MIPS Assembly

```
.foo
lw $t0, 4($r0)
addi $t1, $t0, 3
beq $t1, $t2, foo
nop
```

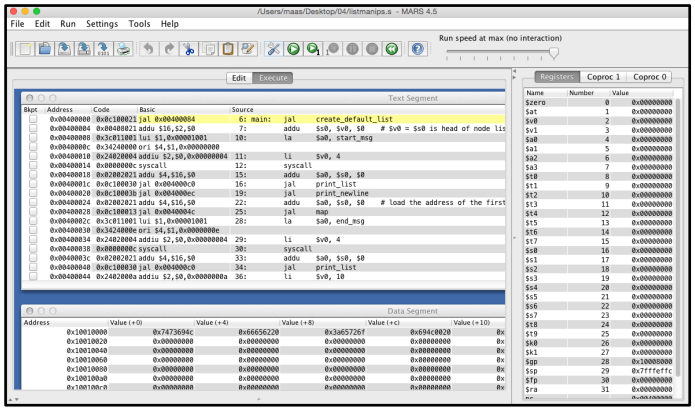
Project 2

Project 3

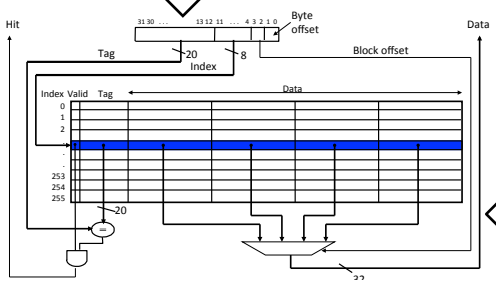
CPU



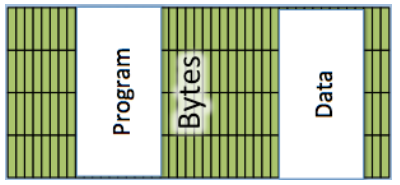
Labs



Caches



Memory



So how is this any different?

Adding I/O

C Programs

```
#include <stdlib.h>

int fib(int n) {
  return
    fib(n-1) +
    fib(n-2);
}
```

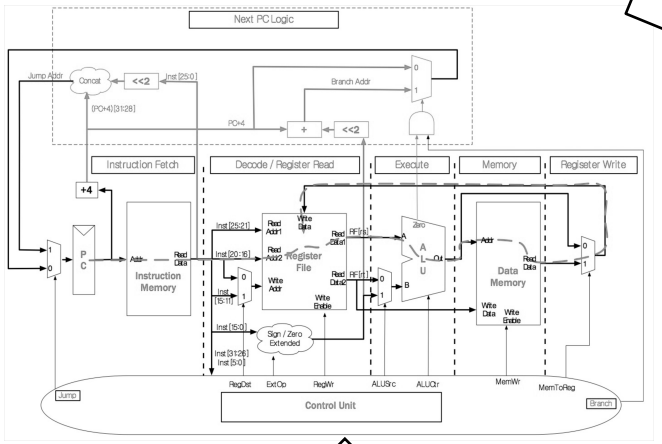
Project 1

MIPS Assembly

```
.foo
lw $t0, 4($r0)
addi $t1, $t0, 3
beq $t1, $t2, foo
nop
```

Project 3

CPU



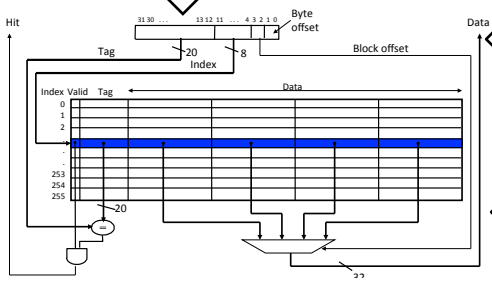
Screen

Keyboard

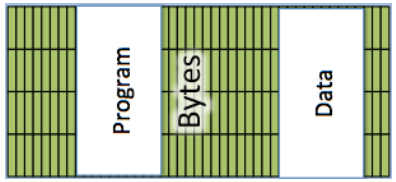
Storage

I/O (Input/Output)

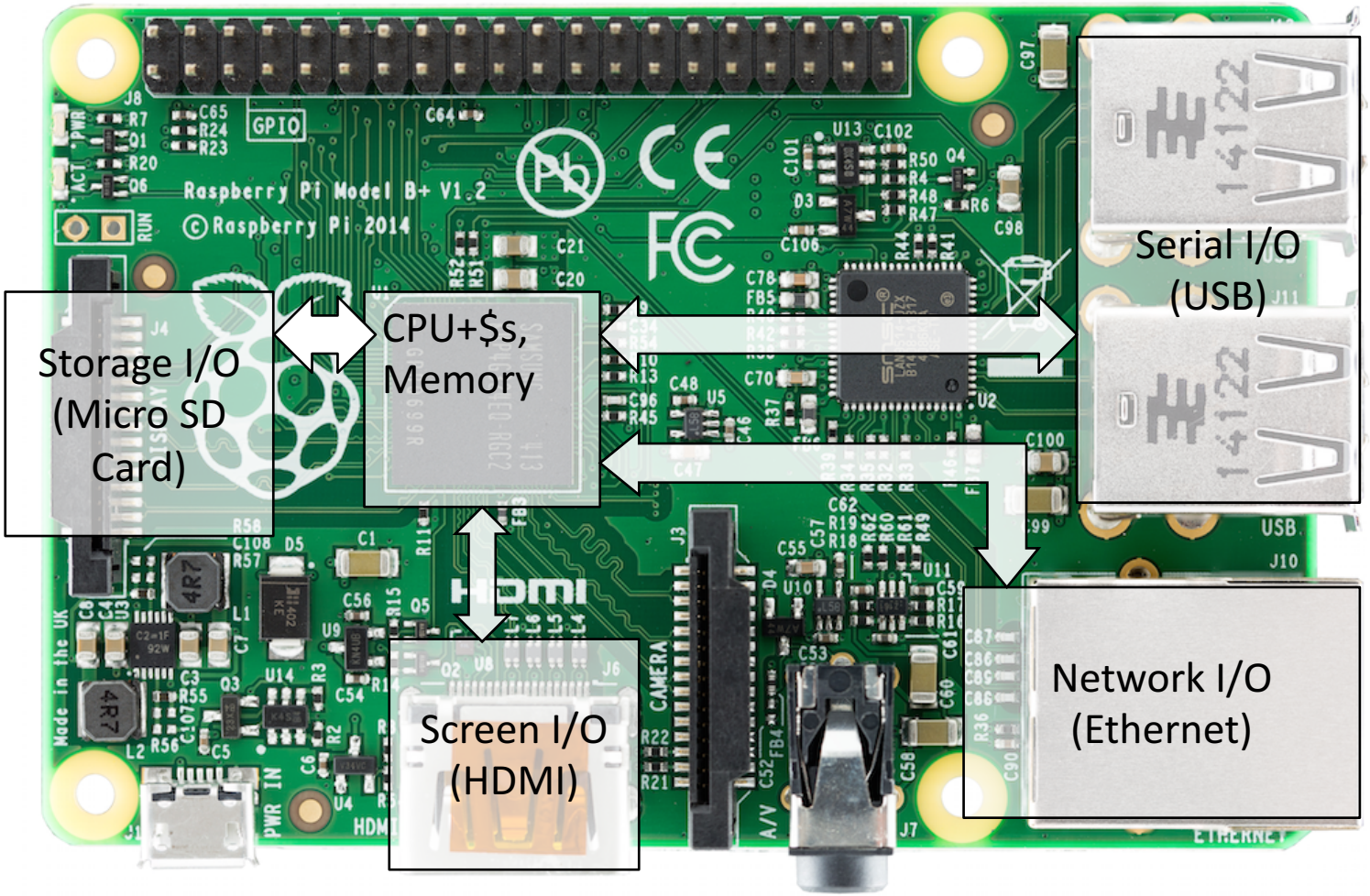
Caches



Memory



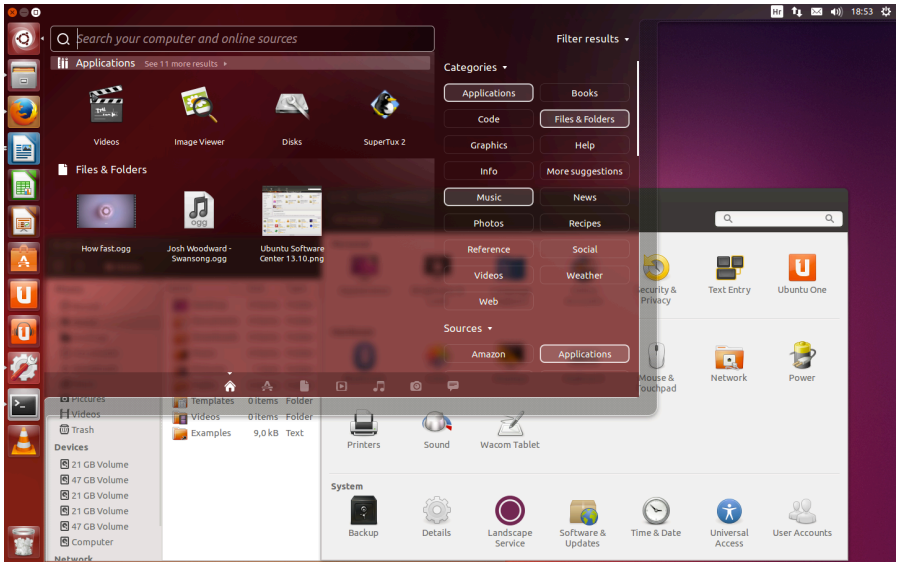
Raspberry Pi ($40 on Amazon)



It’s a real computer!

But wait…

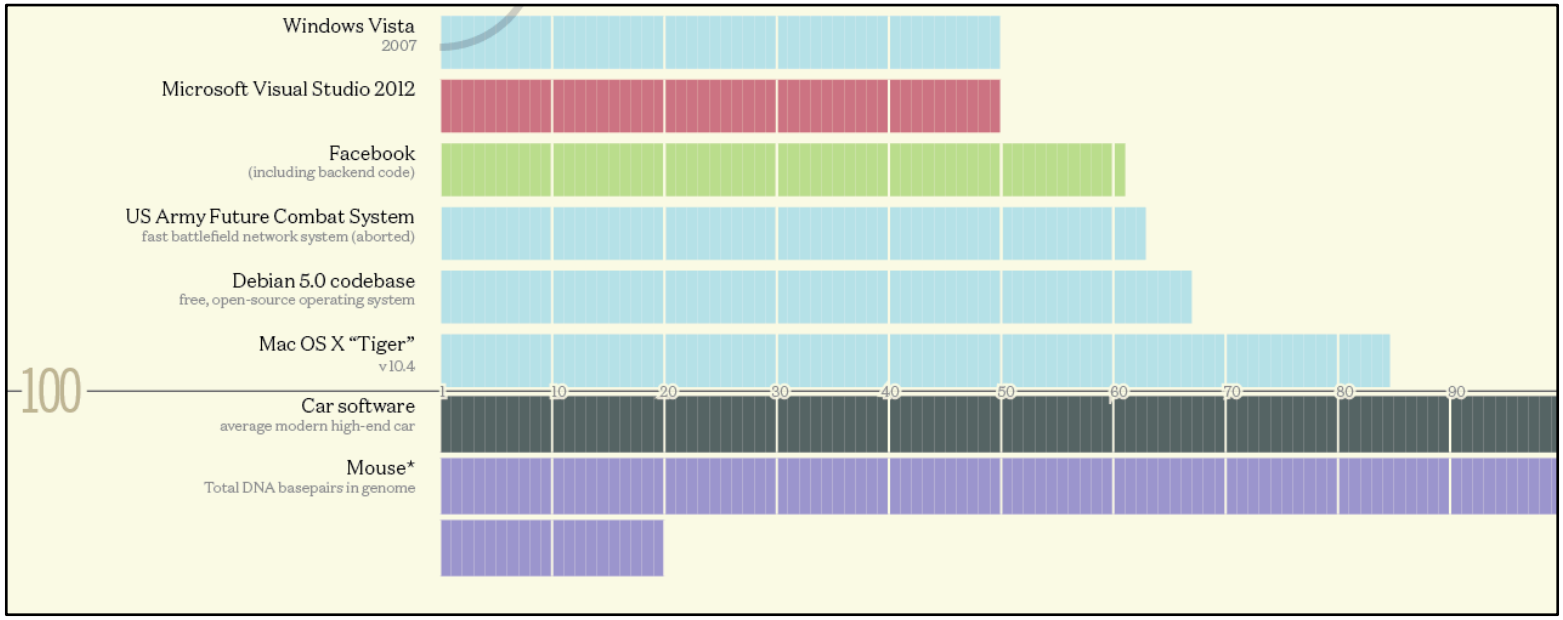
- That's not the same! When we run MARS, it only executes one program and then stops.
- When I switch on my computer, I get this:



Yes, but that's *just* software! The Operating System (OS)

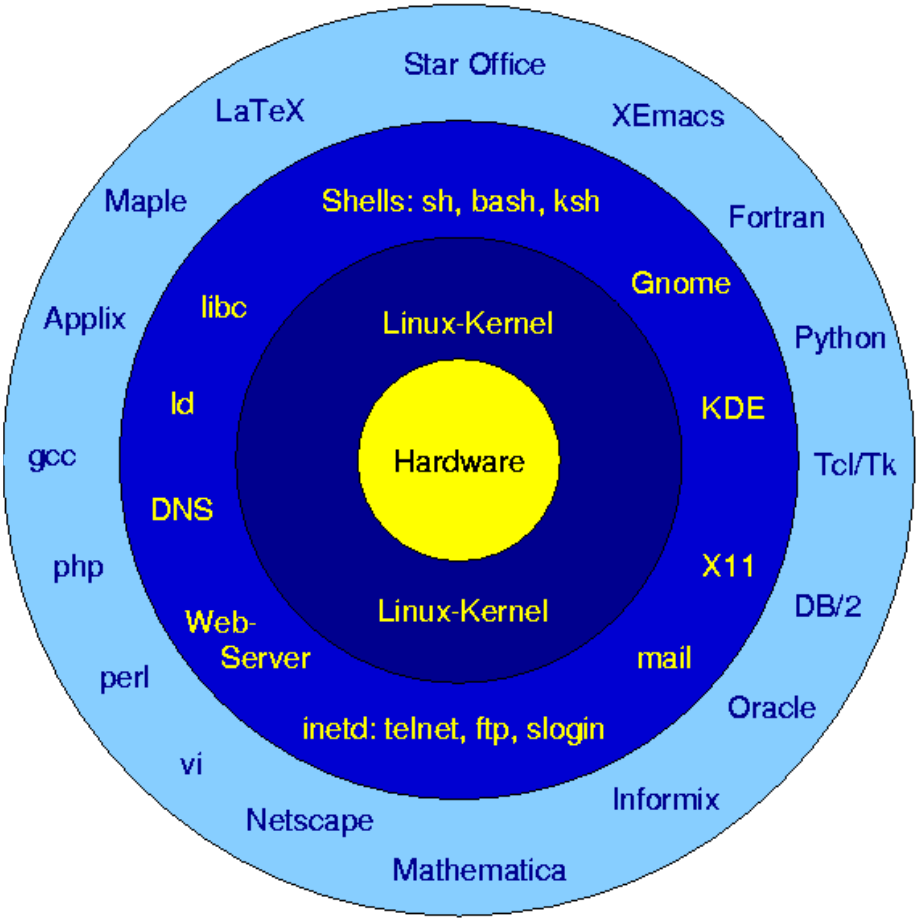
Well, "just software"

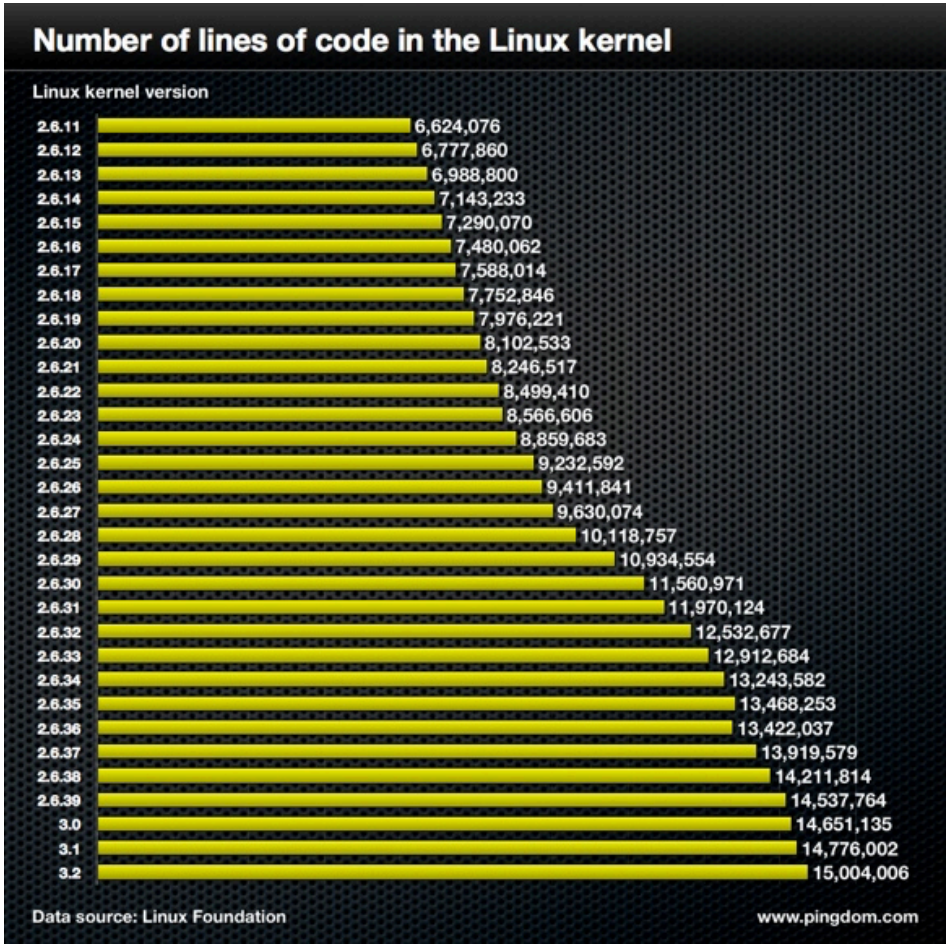
- The biggest piece of software on your machine?
- How many lines of code? These are guesstimates:



Codebases (in millions of lines of code). CC BY-NC 3.0 — David McCandless © 2013
http://www.informationisbeautiful.net/visualizations/million-lines-of-code/

Operating System





What does the OS do?

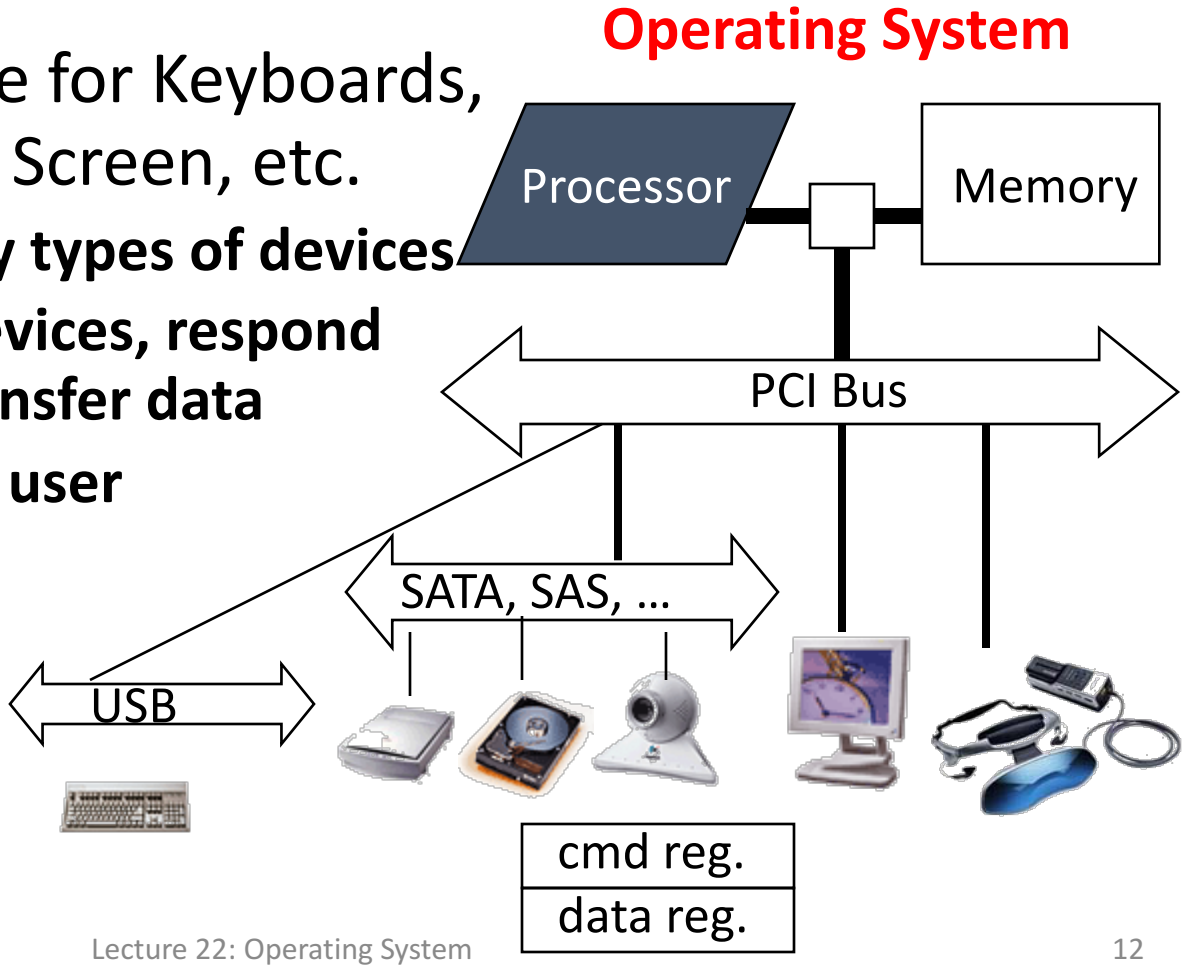
- OS is first thing that runs when computer starts
- Finds and controls all devices in the machine in a general way
  - Relying on hardware specific “device drivers”
- Starts services (100+)
  - File system,
  - Network stack (Ethernet, WiFi, Bluetooth, …),
  - TTY (keyboard),
  - …
- Loads, runs and manages programs:
  - Multiple programs at the same time (time-sharing)
  - Isolate programs from each other (isolation)
  - Multiplex resources between applications (e.g., devices)

Agenda

- **Devices and I/O**
- Polling
- Interrupts
- OS Boot Sequence
- Multiprogramming/time-sharing

How to interact with devices?

- Assume a program running on a CPU. How does it interact with the outside world?
- Need I/O interface for Keyboards, Network, Mouse, Screen, etc.
  - **Connect to many types of devices**
  - **Control these devices, respond to them, and transfer data**
  - **Present them to user programs so they are useful**

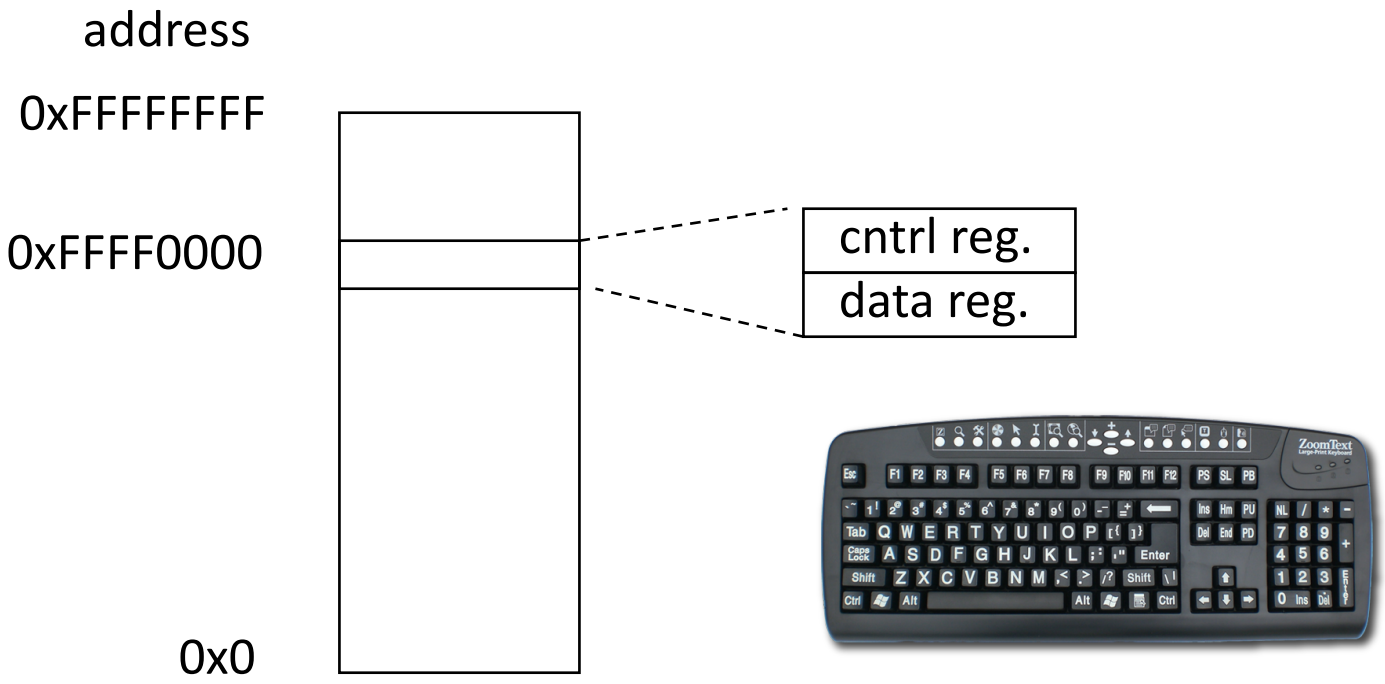


Instruction Set Architecture for I/O

- What must the processor do for I/O?
  - Input: read a sequence of bytes
  - Output: write a sequence of bytes
- Interface options
  a) Special input/output instructions & hardware
  b) Memory mapped I/O
     - Portion of address space dedicated to I/O
     - I/O device registers there (no memory)
     - Use normal load/store instructions, e.g. **`lw/sw`**
     - Very common, used by MIPS

Memory Mapped I/O

- Certain addresses are not regular memory
- Instead, they correspond to registers in I/O devices



Processor-I/O Speed Mismatch

- 1GHz microprocessor I/O throughput:
  - 4 Gi-B/s (**`lw/sw`**)
  - Typical I/O data rates:
    - 10 B/s (keyboard)
    - 100 Ki-B/s (Bluetooth)
    - 60 Mi-B/s (USB 2)
    - 100 Mi-B/s (Wifi, depends on standard)
    - 125 Mi-B/s (G-bit Ethernet)
    - 550 Mi-B/s (cutting edge SSD)
    - 1.25 Gi-B/s (USB 3.1 Gen 2)
    - 6.4 GiB/s (DDR3 DRAM)
  - These are peak rates – actual throughput is lower
- Common I/O devices neither deliver nor accept data matching processor speed

No Endorsement. Just a break from the lecture.

Agenda

- Devices and I/O
- **Polling**
- Interrupts
- OS Boot Sequence
- Multiprogramming/time-sharing

Processor Checks Status before Acting

- Device registers generally serve 2 functions:
  - Control Register, says it's OK to read/write (I/O ready) [think of a flagman on a road]
  - Data Register, contains data
- Processor reads from Control Register in loop
  - waiting for device to set Ready bit in Control reg (0 → 1)
  - Indicates "data available" or "ready to accept data"
- Processor then loads from (input) or writes to (output) data register
  - I/O device resets control register bit (1 → 0)
- Procedure called "Polling"

I/O Example (polling)

- Input: Read from keyboard into **$v0**

```
                lui   $t0, 0xffff #ffff0000 (io addr)
Waitloop:       lw    $t1, 0($t0) #read control
                andi  $t1,$t1,0x1 #ready bit
                beq   $t1,$zero, Waitloop
                lw    $v0, 4($t0) #data
```

- Output: Write to display from **$a0**

```
                lui   $t0, 0xffff #ffff0000
Waitloop:       lw    $t1, 8($t0) #write control
                andi  $t1,$t1,0x1 #ready bit
                beq   $t1,$zero, Waitloop
                sw    $a0,12($t0) #data
```

"Ready" bit is from processor's point of view!

Cost of Polling?

- Assume for a processor with
  - 1GHz clock rate
  - Taking 400 clock cycles for a polling operation
    - Call polling routine
    - Check device (e.g. keyboard or wifi input available)
    - Return
  - What's the percentage of processor time spent polling?
- Example:
  - Mouse
  - Poll 30 times per second
    - Set by requirement not to miss any mouse motion (which would lead to choppy motion of the cursor on the screen)

Percent Processor Time to Poll Mouse

- Mouse Polling [instructions/sec]
  = 30 [polls/s] * 400 [instructions/poll]
  = 12K [instructions/s]

- % Processor for polling:
  - Total processor throughput: T = $1*10^9$ [instructions/s]
  - Polling: P = $12*10^3$ [instructions/s]
  - Ratio: P/T = 0.0012%

  → Negligible:
    - Polling mouse has little impact on processor throughput

Clicker Time

Hard disk: transfers data in 16-Byte chunks and can transfer at 16 MB/second. No transfer can be missed. What percentage of processor time is spent in polling (assume 1GHz clock)?

- A: 2%
- B: 4%
- C: 20%
- D: 40%
- E: 80%

% Processor time needed poll disk

- Frequency of Polling Disk
  = 16 [MB/s] / 16 [B/poll] = 1M [polls/s]
- Disk Polling, instructions/sec
  = 1M [polls/s] * 400 [instructions/poll]
  = 400M [instructions/s]
- % Processor for polling:
  $400*10^6$ [instructions/s] / $1*10^9$ [instructions/s] = 40%
  → Unacceptable

What is the alternative to polling?

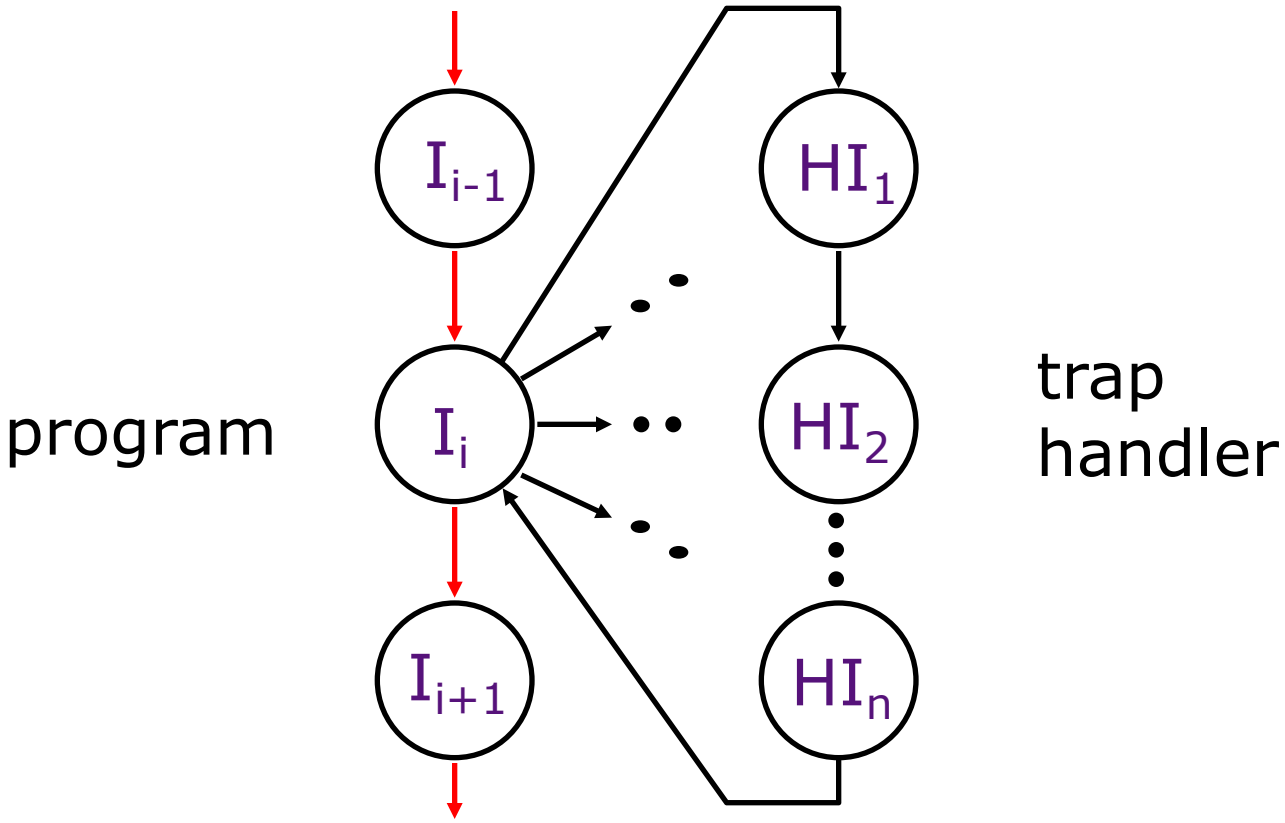
- Polling wastes processor resources
- Akin to waiting at the door for guests to show up
  - What about a bell?
- Computer lingo for bell:
  - Interrupt
  - Occurs when I/O is ready or needs attention
    - Interrupt current program
    - Transfer control to special code "interrupt handler"

Agenda

- Devices and I/O
- Polling
- **Interrupts**
- OS Boot Sequence
- Multiprogramming/time-sharing

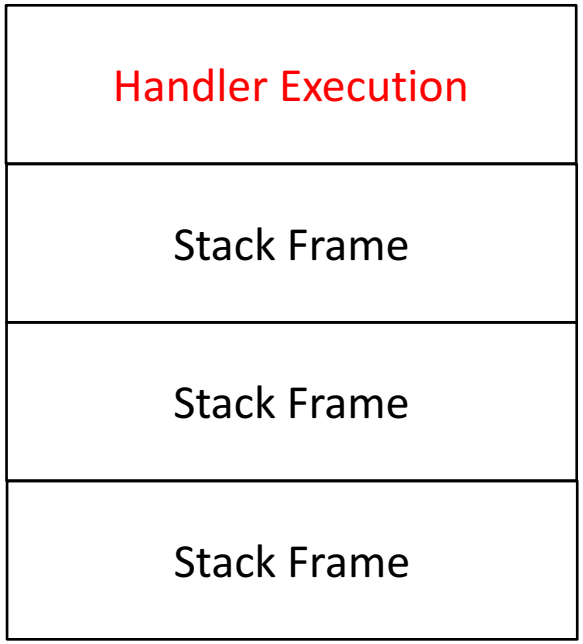
Traps/Interrupts/Exceptions:

altering the normal flow of control (P&H Section 4.9)



An *external or internal event* that needs to be processed - by another program – the OS. The event is often unexpected from original program's point of view.

Interrupt-driven I/O



1. Incoming interrupt suspends instruction stream
2. Looks up the vector (function address) of a handler in an interrupt vector table stored within the CPU
3. Perform a jal to the handler (save PC in EPC* register)
4. Handler run on current stack and returns on finish (thread doesn't notice that a handler was run)

```
handler:  lui  $t0, 0xffff
          lw   $t1, 0($t0)
          andi $t1,$t1,0x1
          lw   $v0, 4($t0)
          sw   $t1, 8($t0)
          ret
```

Code for saving/restoring $t0, $t1, $v0 not shown

```
Label: sll  $t1,$s3,2
       addu $t1,$t1,$s5
       lw   $t1,0($t1)
       or   $s1,$s1,$t1
       addu $s3,$s3,$s4
       bne  $s3,$s2,Label
```

Interrupt(SPI0)

CPU Vector Interrupt Table	
SPI0	handler
…	…

*EPC: Exception program counter

Terminology

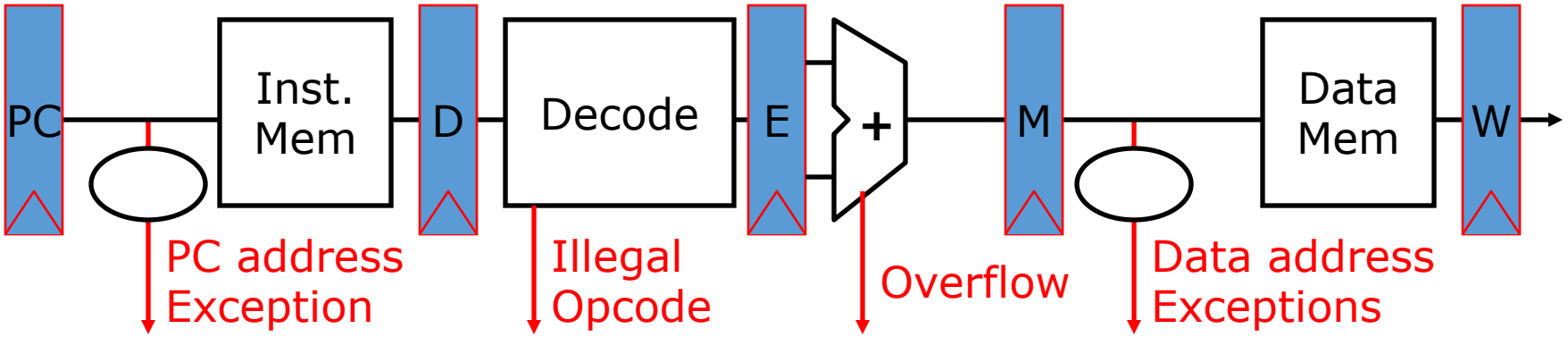
In CS61C (other definitions in use elsewhere):

- Interrupt – caused by an event *external* to current running program
  - E.g. key press, disk
  - Asynchronous to current program
    - Can handle interrupt on any convenient instruction
    - "Whenever it's convenient, just don't wait too long"
- Exception – caused by some event *during* execution of one instruction of current running program
  - E.g., overflow, bus error, illegal instruction
  - Synchronous
    - Must handle exception *precisely* on instruction that causes exception
    - "Drop whatever you are doing and act now"
- Trap – action of servicing interrupt or exception by hardware jump to "interrupt or trap handler" code

Precise Traps

- *Trap handler's view of machine state is that every instruction prior to the trapped one (e.g. overflow) has completed, and no instruction after the trap has executed.*
- Implies that handler can return from an interrupt by restoring user registers and jumping back to interrupted instruction
  - Interrupt handler software doesn't need to understand the pipeline of the machine, or what program was doing!
  - More complex to handle trap caused by an exception than interrupt
- Providing precise traps is tricky in a pipelined superscalar out-of-order processor!
  - But a requirement e.g. for
    - Virtual memory to function properly (see next lecture)

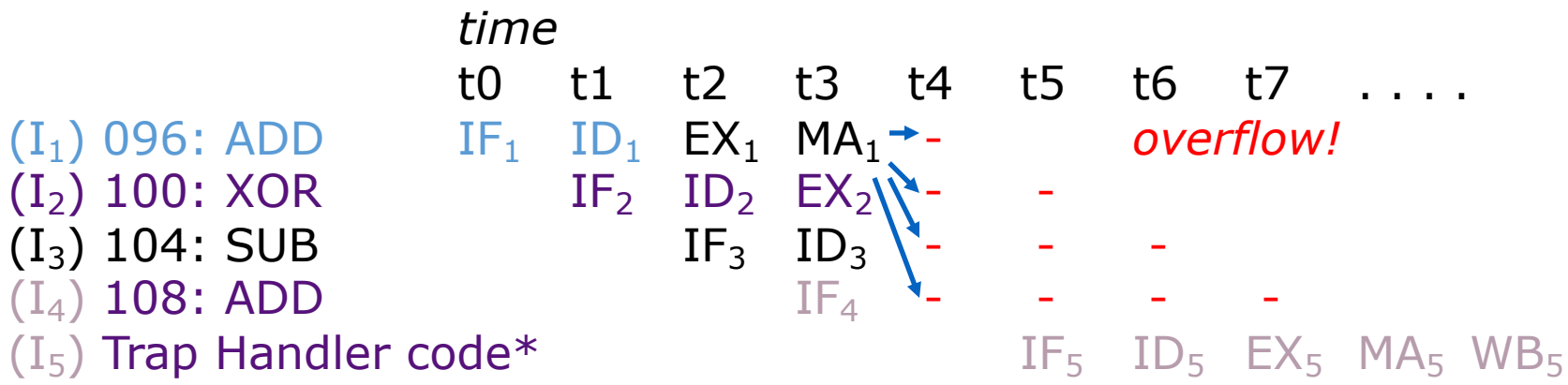
Trap Handling in 5-Stage Pipeline



Exceptions are handled *like pipeline hazards*

1) Complete execution of instructions before exception occurred
2) Flush instructions currently in pipeline (convert to **nop**s or "bubbles")
3) Optionally store exception cause in status register (e.g. MIPS)
   - Indicate type of exception
   - Note: several exceptions can occur in a single clock cycle!
4) Transfer execution to trap handler

Trap Pipeline Diagram



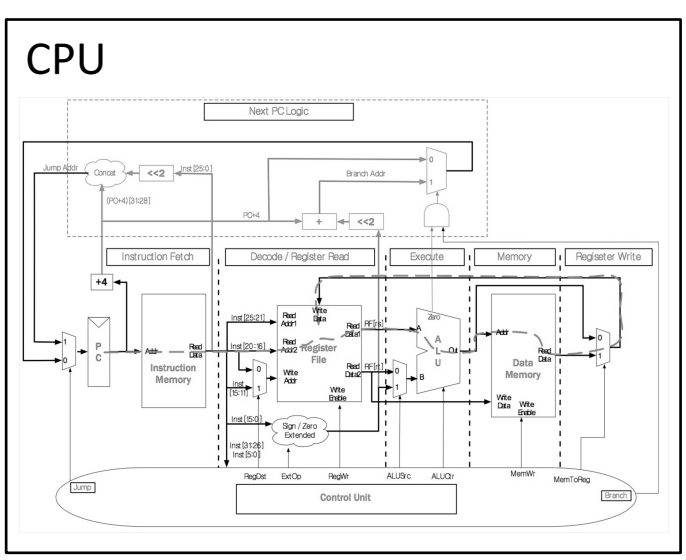
*EPC = 100 (instruction following offending ADD)

Agenda

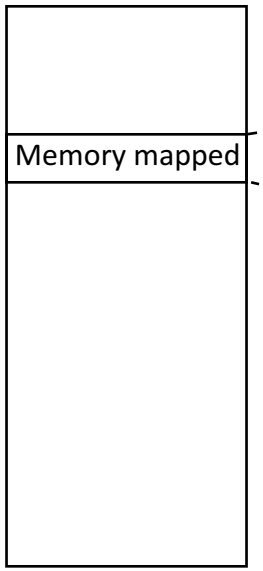
- Devices and I/O
- Polling
- Interrupts
- **OS Boot Sequence**
- Multiprogramming/time-sharing

What happens at boot?

- When the computer switches on, it does the same as MARS: the CPU executes instructions from some start address (stored in Flash ROM)



PC = 0x2000 (some default value)

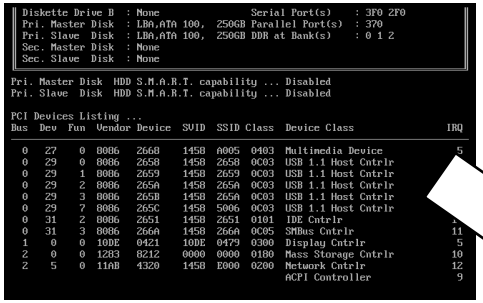


Address Space

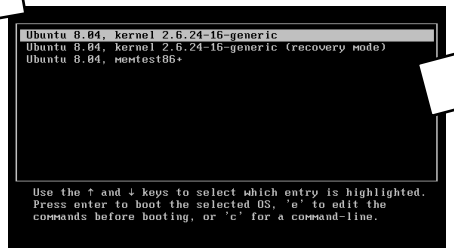

```
0x2000:
addi $t0, $zero, 0x1000
lw $t0, 4($r0)
…

(Code to copy firmware into
regular memory and jump
into it)
```

What happens at boot?

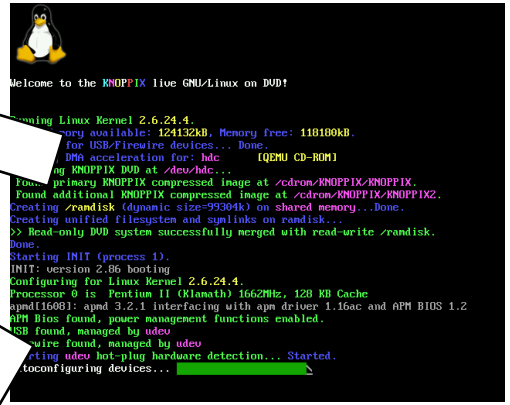
1. BIOS*: Find a storage device and load first sector (block of data)



2. Bootloader (stored on, e.g., disk): Load the OS *kernel* from disk into a location in memory and jump into it.



3. OS Boot: Initialize services, drivers, etc.



4. Init: Launch an application that waits for input in loop (e.g., Terminal/Desktop/…

*BIOS: Basic Input Output System

Launching Applications

- Applications are called “processes” in most OSs.
  - Thread: shared memory
  - Process: separate memory
  - Both threads and processes run (pseudo) simultaneously
- Apps are started by another process (e.g. shell) calling an OS routine (using a “syscall”)
  - Depends on OS, but Linux uses fork to create a new process, and execve to load application.
- Loads executable file from disk (using the file system service) and puts instructions & data into memory (.text, .data sections), prepares stack and heap.
- Set argc and argv, jump to start of main.
- Shell waits for main to return (join)

Supervisor Mode

- If something goes wrong in an application, it could crash the entire machine. And what about malware, etc.?
- The OS enforces resource constraints to applications (e.g., access to memory, devices).
- To help protect the OS from the application, CPUs have a supervisor mode (set e.g. by a status bit).
  - A process can only access a subset of instructions and (physical) memory when not in supervisor mode (user mode).
  - Process can change out of supervisor mode using a special instruction, but not into it directly – only using an interrupt.
  - Supervisory mode is a bit like "superuser"
    - But used much more sparingly
      (most of OS code does *not* run in supervisory mode)
    - Errors in supervisory mode often catastrophic
      (blue "screen of death", or "I just corrupted your disk")

Syscalls

- What if we want to call an OS routine? E.g.,
  - to read a file,
  - launch a new process,
  - ask for more memory (malloc),
  - send data, etc.
- Need to perform a syscall:
  - set up function arguments in registers,
  - raise software interrupt (with special assembly instruction)
- OS will perform the operation and return to user mode
- This way, the OS can mediate access to all resources, and devices.

Agenda

- Devices and I/O
- Polling
- Interrupts
- OS Boot Sequence
- **Multiprogramming/time-sharing**

Multiprogramming

- The OS runs multiple applications at the same time.
- But not really (unless you have a core per process)
- Switches between processes very quickly (on human time scale). This is called a “context switch”.
- When jumping into process, set timer interrupt.
  - When it expires, store PC, registers, etc. (process state).
  - Pick a different process to run and load its state.
  - Set timer, change to user mode, jump to the new PC.
- Deciding what process to run is called scheduling.

Protection, Translation, Paging

- Supervisor mode alone is not sufficient to fully isolate applications from each other or from the OS.
  - Application could overwrite another application's memory.
  - Typically programs start at some fixed address, e.g. 0x1000
    - How can 100's of programs share memory at location 0x1000?
  - Also, may want to address more memory than we actually have (e.g., for sparse data structures).
- Solution: Virtual Memory.
  - Gives each process the *illusion* of a full memory address space that it has completely for itself.

And, in Conclusion, …

- Basic machine (datapath, memory, IO devices) are application agnostic
- Same concepts / processor architecture apply to large variety of applications. E.g.
  - OS with command line and graphical interface (Linux, …)
  - Embedded processor in network switch, car engine control, …
- Input / output (I/O)
  - Memory mapped: appears like "special kind of memory"
  - Access with usual load/store instructions (e.g. **`lw, sw`**)
- Exceptions
  - Notify processor of special events, e.g. overflow, page fault (next lecture)
  - "precise" handling: immediately at offending instruction
- Interrupts
  - Notification of external events, e.g. keyboard input or Ethernet traffic
- Multiprogramming and supervisory mode
  - Enables and isolates multiple programs
- Take CS162 to learn more about operating systems