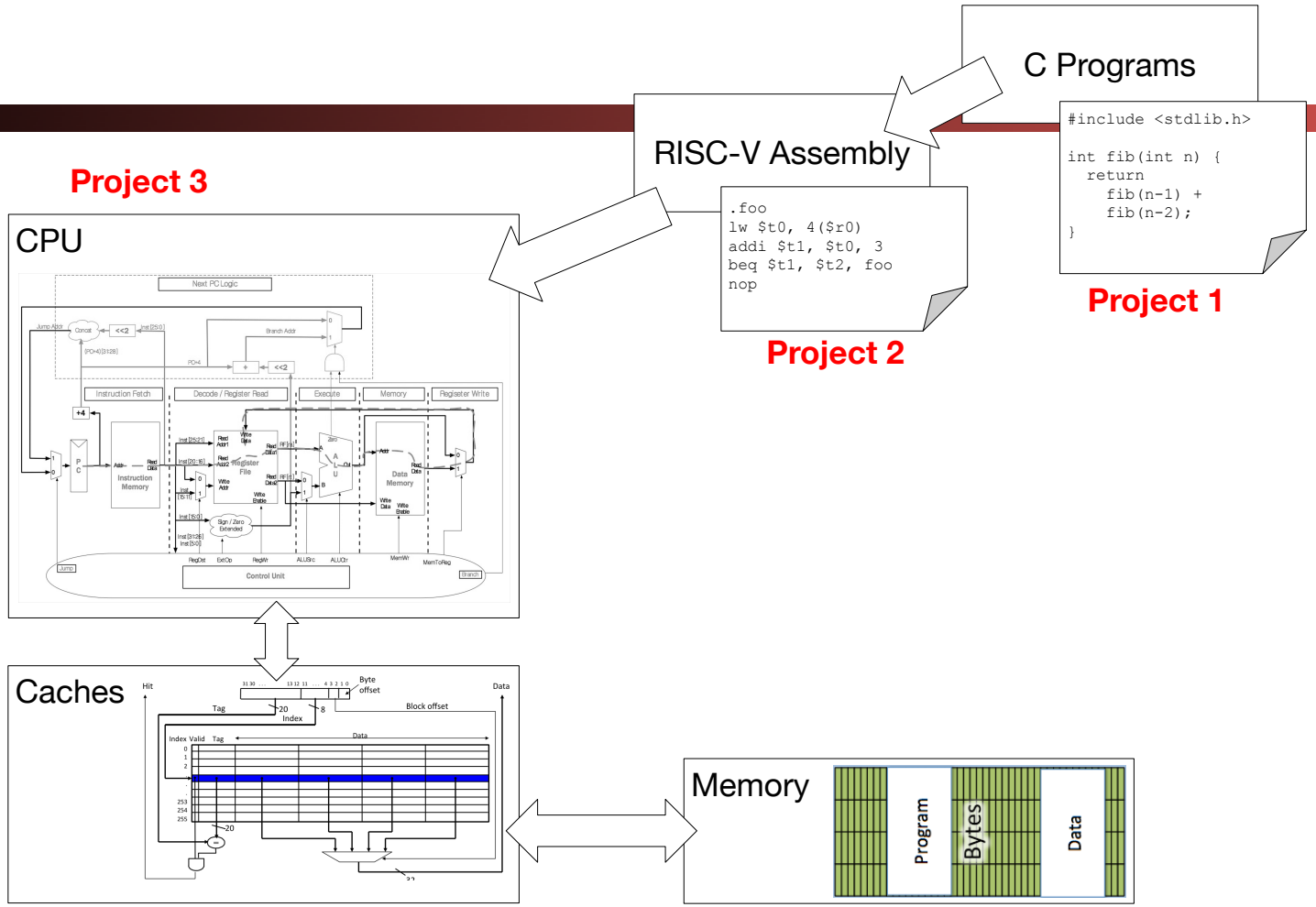
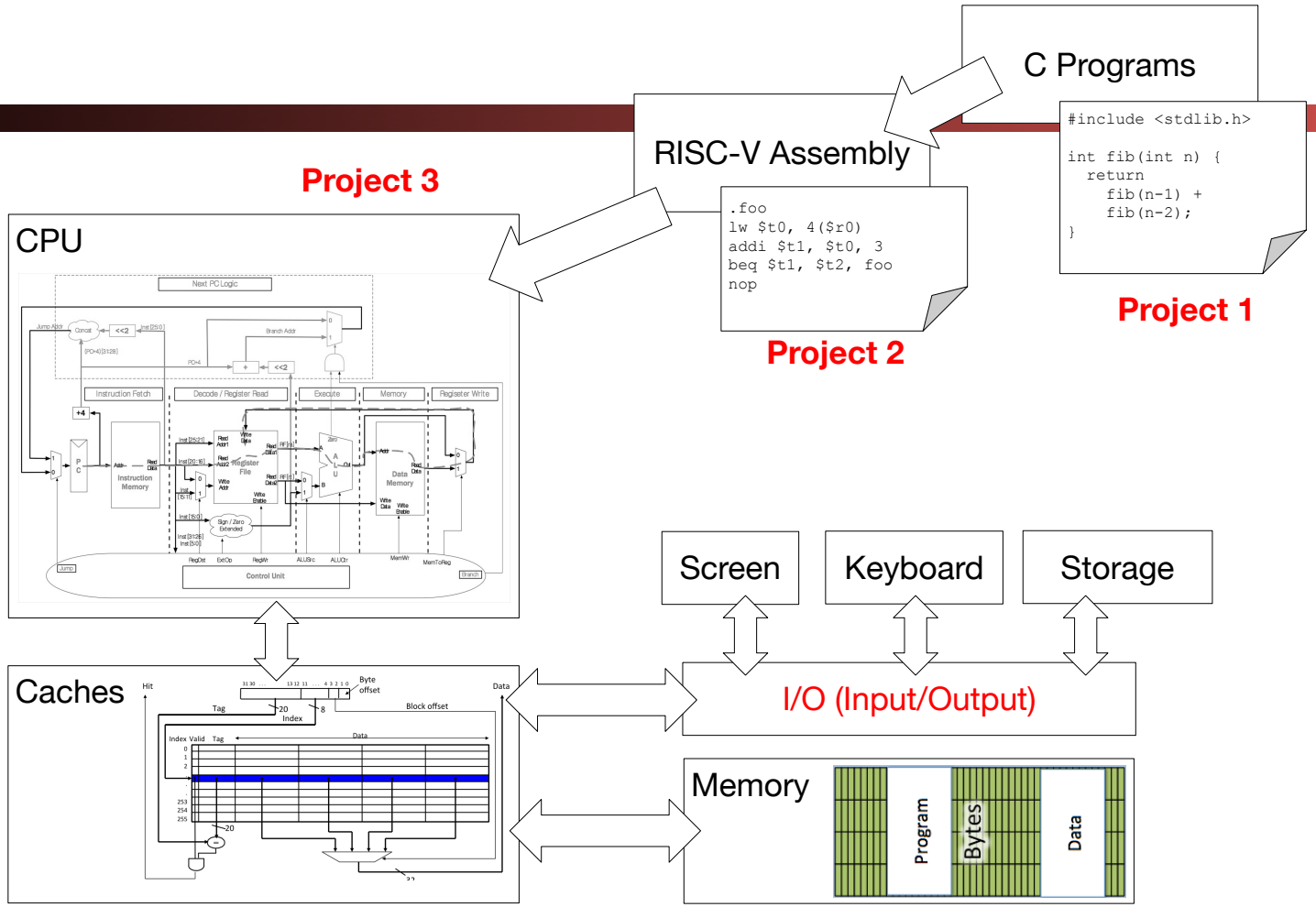
Operating Systems & Virtual Memory

Administrivia

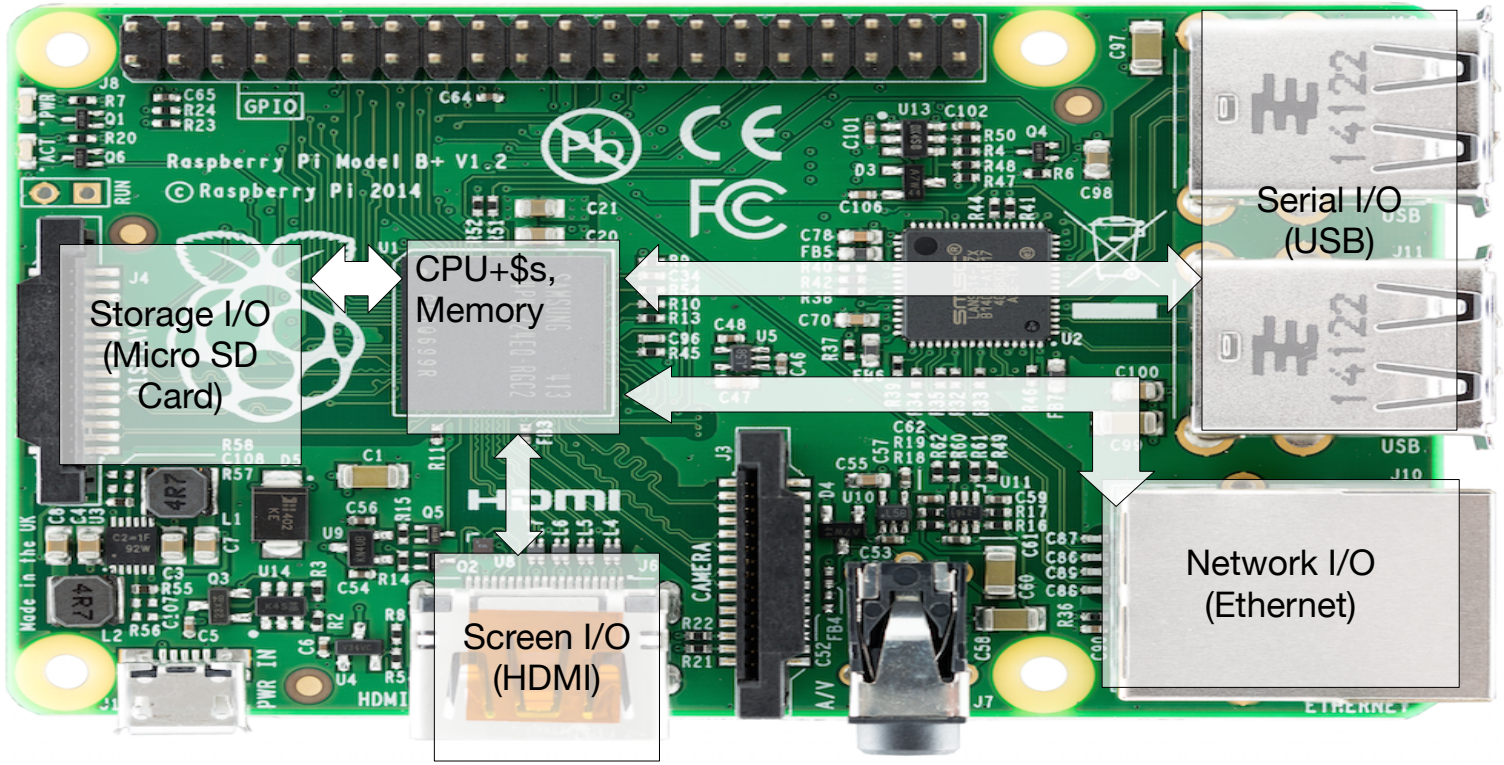
CS61C so far…


C Programs
#include <stdlib.h>
int fib(int n) {
return
fib(n-1) +
fib(n-2);
}
Project 1
RISC-V Assembly
.foo
lw $t0, 4($r0)
addi $t1, $t0, 3
beq $t1, $t2, foo
nop
Project 2
Project 3
CPU
Caches
Memory
Program
Bytes
Data

Adding I/O

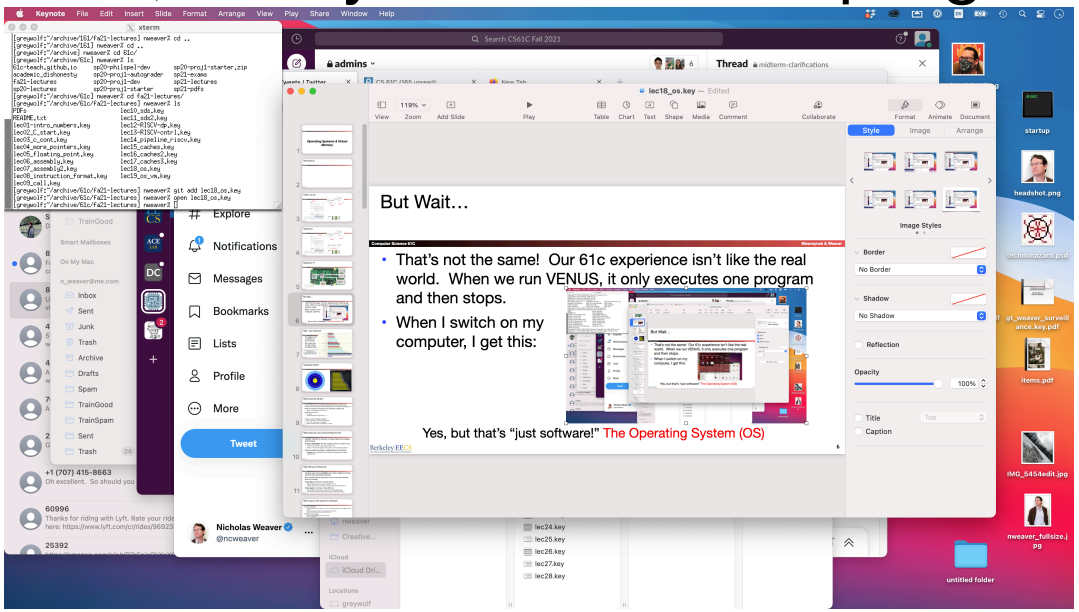

C Programs
#include <stdlib.h>
int fib(int n) {
return
fib(n-1) +
fib(n-2);
}
Project 1
RISC-V Assembly
.foo
lw $t0, 4($r0)
addi $t1, $t0, 3
beq $t1, $t2, foo
nop
Project 2
Project 3
CPU
Caches
Hit
Byte offset
Tag
Index
Block offset
Data
Screen
Keyboard
Storage
I/O (Input/Output)
Memory
Program
Bytes
Data

Raspberry Pi



But Wait…

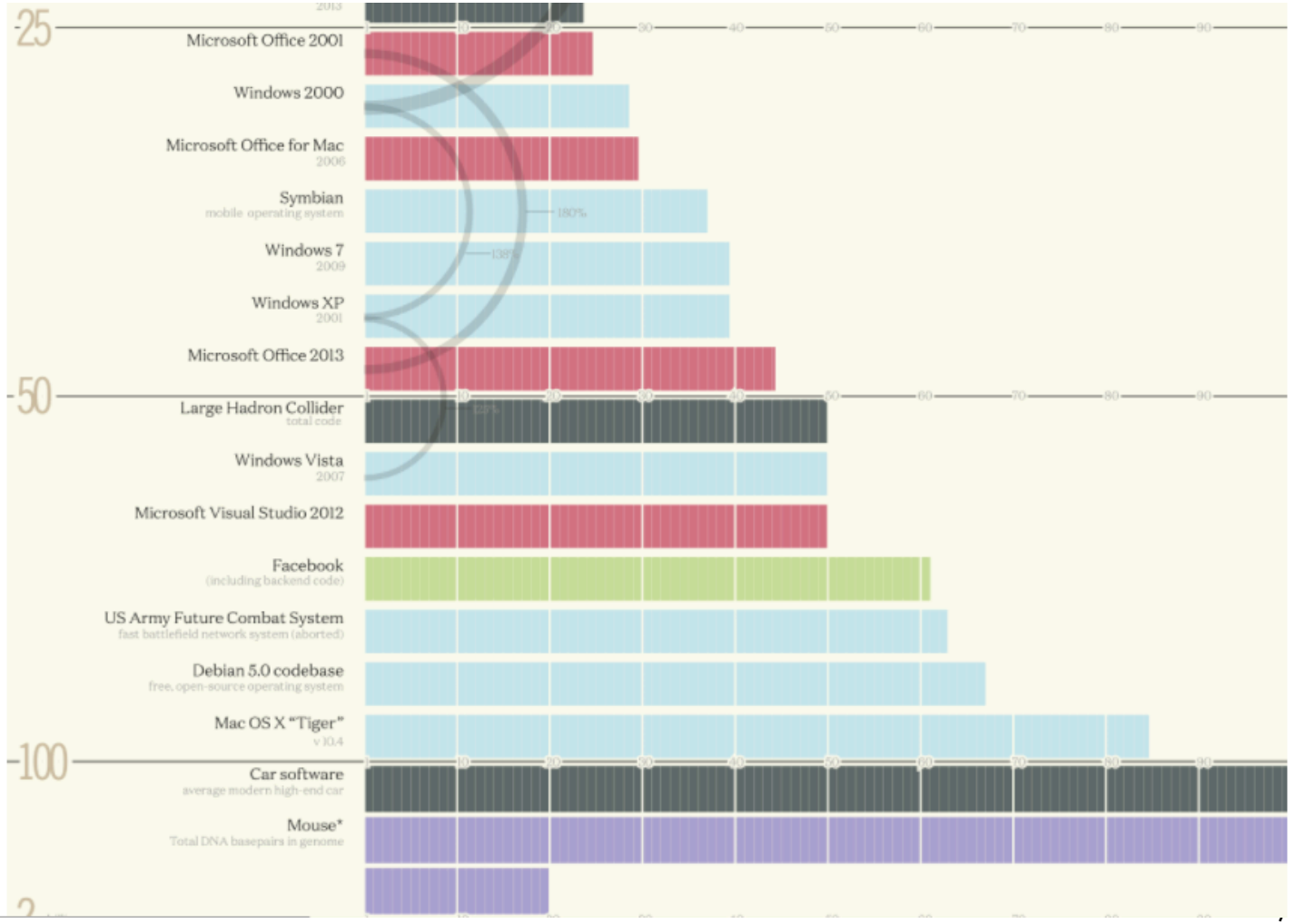
- That's not the same! Our 61c experience isn't like the real world. When we run VENUS, it only executes one program and then stops.
- When I switch on my computer, I get this:



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

Well, “Just Software”

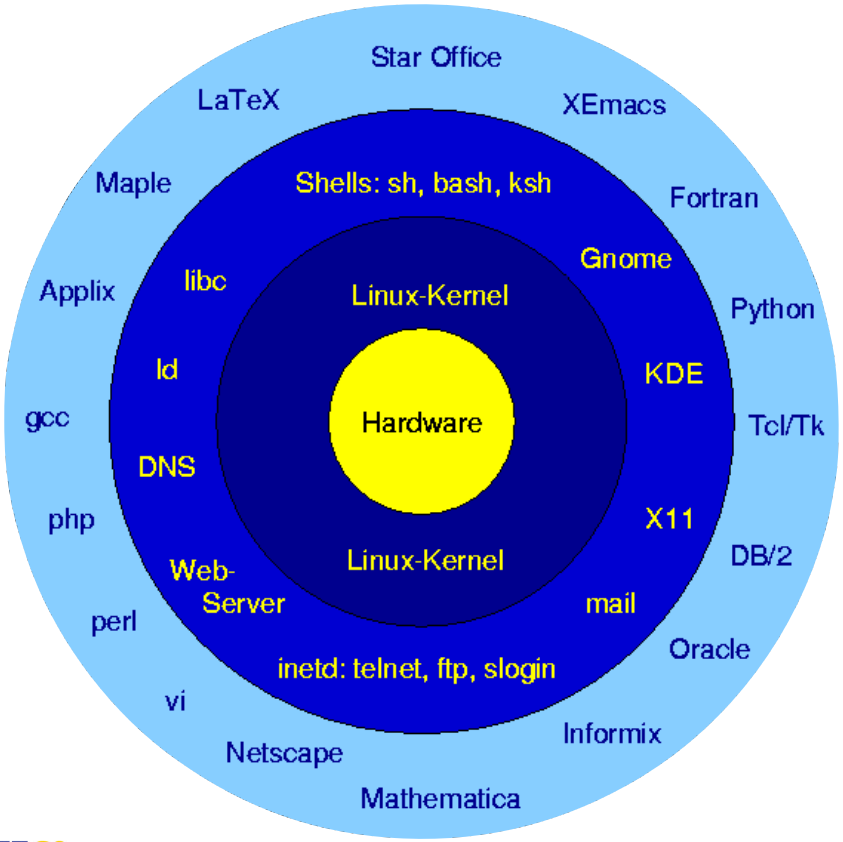
- The biggest piece of software on your machine?
- How many lines of code? These are guesstimates:

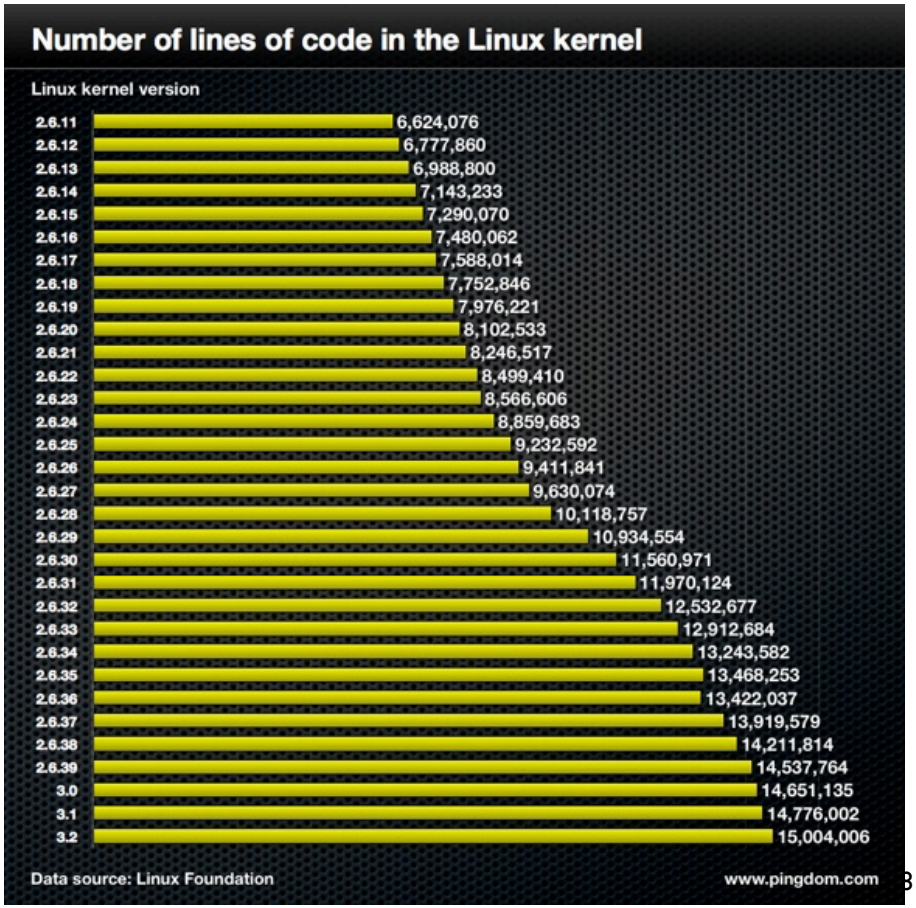


Codebases (in millions of lines of code).

http://www.informationisbeautiful.net/visualizations/million-lines-of-code/

Operating System





What Does the OS do?

- OS runs before any user programs (after BIOS and boot loader) when computer is first turned on, and intermittently runs as long as computer is on
- Finds and controls all I/O devices in the machine in a general way
  - Relying on hardware specific “device drivers”
- Starts services (100+)
  - File system,
  - Network stack (Ethernet, WiFi, Bluetooth, …),
  - …
- 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)

What Does the core of the OS Need To Do?

- Mediate interactions between running programs (processes) and hardware
- Enables ***interaction*** with the outside world in a uniform way
  - Interact with “devices": Disk, display, network, etc.
  - Project 1: We can use `fopen` to read a file on any kind of storage device
- Allows multiple processes to safely share one computer
  - ***Protection*** for the hardware, the OS, and other processes from a faulty/ malicious proc.

Safe Sharing of Resources

- OS gives each process **isolation** even when multiple processes share the same hardware resources
- Each process has the view that it "owns" the whole machine when it is running
- Share **time** on the CPU: *Context Switch*
  - Need to change from one process to another on CPU
  - Need to save and restore state so a process can pick up where it left off
- Share **space** in memory: *Virtual Memory*
  - Each process has the "illusion" of access to the full address space
  - One process cannot see what another process has stored in memory

What does an OS need from hardware?

- Memory translation
  - Each running process has a mapping from "virtual" to "physical" addresses that are different for each process
  - When you do a load or a store, the program issues a virtual address...
    But the actual memory accessed is a physical address
- Protection and privilege
  - Split the processor into at least two running modes:
    "User" and "Supervisor"
    - RISC-V also has "Machine" below "Supervisor"
  - Lesser privilege ***can not*** change its memory mapping
    - But "Supervisor" ***can*** change the mapping for any given program
    - And supervisor has its own set of mapping of virtual->physical
- Traps & Interrupts:
  - A way of going into Supervisor mode on demand

Hardware "Control and Status Registers"

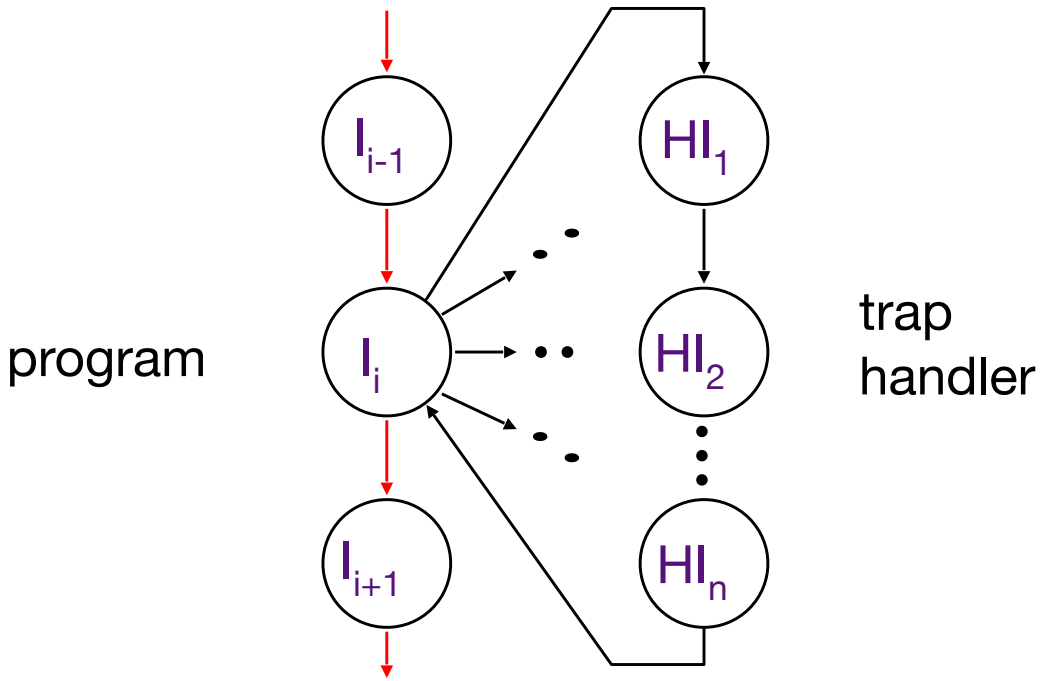
- A set of separate registers that can be read/written atomically
  - **`CSRRW rd rs csr`**
    - Read the old value of the specific control and status register and put it into **`rd`**
    - If **`rs`** != **`x0`**, place the new value in the CSR
  - Similar versions to just set or clear some bits in the CSR, and versions with a 5 bit immediate instead of **`rs`** Csrrwi
- These aren't normal registers...
  - Instead they are used to communicate requests with the hardware
- The hardware enforces privileges
  - So a program running at User level can't change Supervisor-level CSRs

Interrupts and Exceptions...

- We need to transition into Supervisor mode when "something" happens
- Interrupt: Something external to the running program
  - Something happens from the outside world
- Exception: Something done by the running program
  - Accessing memory it isn't "supposed" to, executing an illegal instruction, reading a `csr` not supposed at that privilege
  - **`ECALL`**: Trigger an exception to the higher privilege
    - How you communicate with the operating system:
      Used to implement "syscalls"
  - **`EBREAK`**: Trigger an exception within the current privilege

Traps/Interrupts/Exceptions:

altering the normal flow of control



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.

Terminology

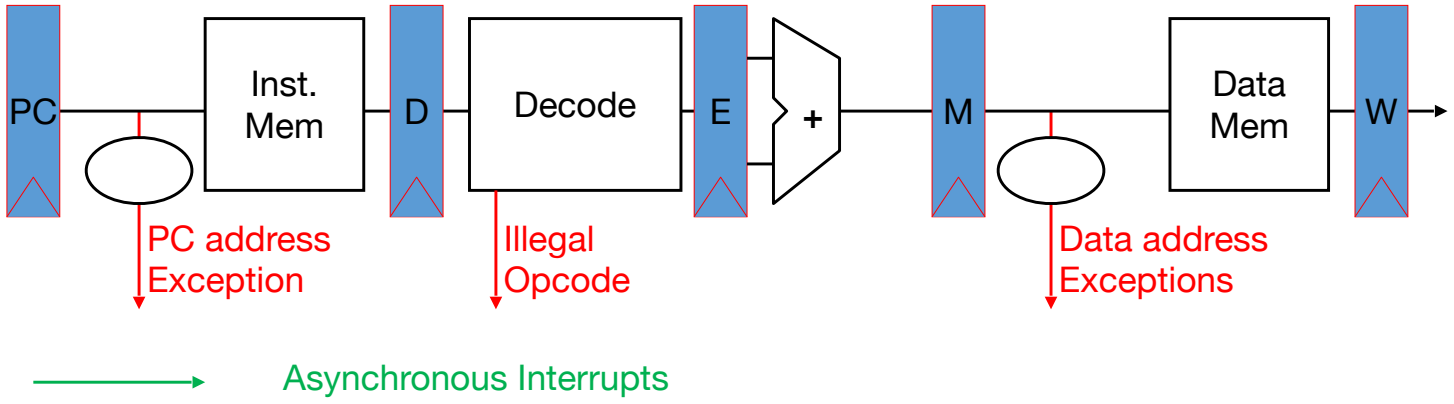
In CS61C (other definitions in use elsewhere):

- Interrupt – caused by an event *external* to current running program
  - E.g., key press, disk I/O
  - 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., memory error, bus error, illegal instruction, raised exception
  - 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., memory error) 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 for things to actually work right

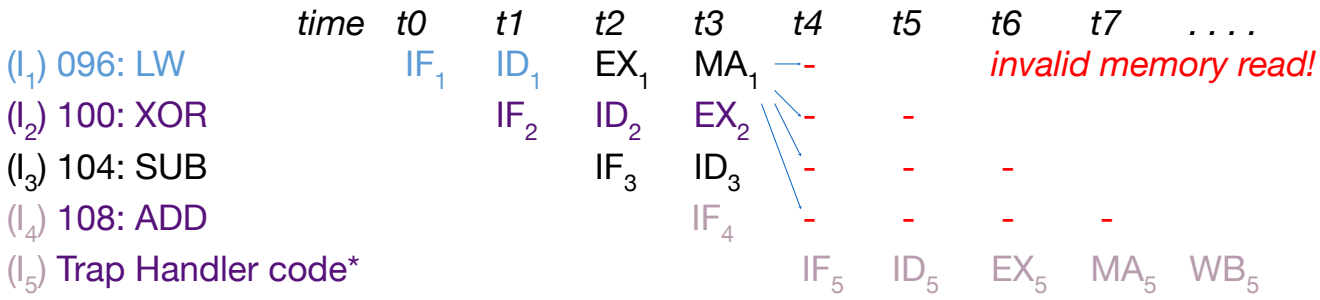
Trap Handling in 5-Stage Pipeline



Exceptions are handled *like pipeline hazards*

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

Trap Pipeline Diagram



*MEPC = 100 (instruction following offending LW)

So an Exception or Interrupt happens...
What does the hardware do?

- Note, slightly simplified and RISC-V specific
  - But most processors have a similar flow...
- All this happens as a single action ("Atomically")
- The hardware adjust the privilege level (if appropriate, some exceptions don't change privilege level)
  - So the processor is now in ***supervisor*** mode in most cases
- Disable interrupts!
  - Don't want to get interrupted when handling an interrupt
- Write the old program counter into the **`sepc`** CSR
  - Note: It is the PC that triggered the exception or the first instruction that hasn't yet executed if an interrupt
- Write the reason into the **`scause`** CSR
- Set the PC to the value in the **`stvec`** CSR
  - This is the address of the "trap handler":
    The single function that handles ***ALL*** exceptions and interrupts

And Now It Is In Software's Hands...
The Trap Handler's Job...

- Save ***all*** the registers
  - Intent is to make the previous program think that ***nothing whatsoever actually happened!***
- Figure out what the exception or interrupt is...
  - Read the appropriate CSRs and other pieces to do what is necessary
- Restore all the registers
- Return to the right point in execution

How To Save The Registers?

- Supervisor mode has a **`sscratch`** CSR
  - Use it to point to a piece of memory to store things for the trap handler
- Swap x1 for sscratch
  - **`csrrw x1 x1 sscratch`** # x1 = ra
- Now save all the other registers into that location
  - **`sw x2 4(x1)`**
    `sw x3 8(x1)`
    `...`
- And store the pc from the **`sepc`** CSR
  - **`csrrw x2 x0 sepc`** # x2 = sp
    `sw x2 124(x1)`
- And finally save x1 and restore **`sscratch`**
  - **`csrrw x2 x1 sscratch`**
    `sw x2 0(x1)`

original ra => sscratch
x1 => mem addr.

rd = sp, rs = x0 (zero)

original ra => x2, mem addr => sscratch
save original x1 to 0(mem addr)

Figure Out What To Do...

- We will cover that later
  - It really depends on what caused the exception or interrupt
- If it is an ECALL (so we're doing a "sys call")
  - What is user process asking OS to do? Do that and return to the ***next*** instruction...
- If it is an illegal instruction or similar error
  - Just kill the running program and do something else
- If it is some memory-related exception
  - Can we clean things up? If so, do that and return to the ***current*** instruction...
- If it is the "timer interrupt" - this process has used up its time slice
  - OK, we will execute a "context switch"
- If it is external I/O...

And Now To Return...

- Need to restore all the registers
- Restore the value for **`sepc`**
  - So we know where to return to...
    If ECALL, increment it by 4 first to make it seem like a function call
  - Otherwise, don't...
    - We will just redo the instruction that triggered the exception
- Restore all the other registers
  - May need to swap one temporarily using **`sscratch`**...
    But make sure **`sscratch`** is back to where it should be
  - And if an **`ecall`**, set **`a0`** to the return value...
- Now execute the **`SRET`** instruction

So with `sret` it is back to the hardware...

- Again, this is atomic
- Reenable interrupts
  - After all, now we are done with the trap handler we can get interrupted again
- Reset back down to user level
  - No longer in the privileged mode of operation
- Restore the pc to the value in **`sepc`**
- And now the program continues on like ***nothing ever happened***

So What Does This Mean?

- If it was an **`ecall`**, it worked just like a function call
  - We returned to the next instruction and just carried on:
    The return value is in **`a0`** and, we have what we wanted from the OS
- Otherwise, it is like nothing ever happened...
  - Except for a gap in time:
    Between one instruction and the next significant time may have passed
- And the caches probably got trashed
  - Because something else was using the memory
  - In fact, for security reasons the OS often needs to ***flush all caches*** before returning flow to the program

This Is A Very Powerful Primitive...

- In User mode the program is constrained
  - It can only access the memory it is allowed to:
    Controlled by the virtual memory system
  - Can only update the virtual memory->physical memory mapping by changing the **`satp`** (page table pointer) CSR or the data pointed to by the **`satp`** CSR
    - But can't update any of the supervisor CSRs in user mode
    - And the **`satp`** pointer points to a piece of physical memory that the user mode program *should not have access to*
- So this provides strong constraints
  - Have to stay in user mode except for invoking the trap handler
  - Can only affect the data it is supposed to
  - Can't affect the data that controls what accesses the program has

But It Isn't Cheap...

- Taking an interrupt or trap is ***expensive***...
- We have to flush the pipeline
  - So a big penalty on a modern processor with ~20 or more stages
- We have to save and restore all the registers
- The caches get ***trashed***
  - Since the trap handler is completely unrelated code
  - And isolation requires that ***no cache state is shared across processes***:
    Have to invalidate all caches to prevent information leakages when returning to a different process...
    So if we are returning to the same process, no problem. But if we return to something else...

So The First Thing We Build... Context Switching...

- The hardware provides the OS an interrupt: the "Timer Interrupt"
  - Triggers at a regular interval, e.g. 10 ms
  - So every process gets a 10ms “time slice” where it can use the CPU
- When triggered, trap handler can execute a context switch
  - Take those saved register that were stored in the area pointed to by **`sscratch`**...
    - And copy them to a bookkeeping data structure for the current process (Process Control Block)
    - Also copy the **`satp`** value to that data structure so we know its memory mapping
- Now pick some other process's data structure
  - Determined by the "scheduler": Lots of details in CS162
  - Load that process's registers, **`satp`**, **`sepc`**, etc...
  - Tell the caches to flush themselves
    - Needed for proper isolation: many security attacks involve one process using the cache state of another process to infer information
    - But we'd be taking a ton of misses anyway since the new process has no temporal locality with the old process
- And then return with **`sret`**.

So What Just Happened?

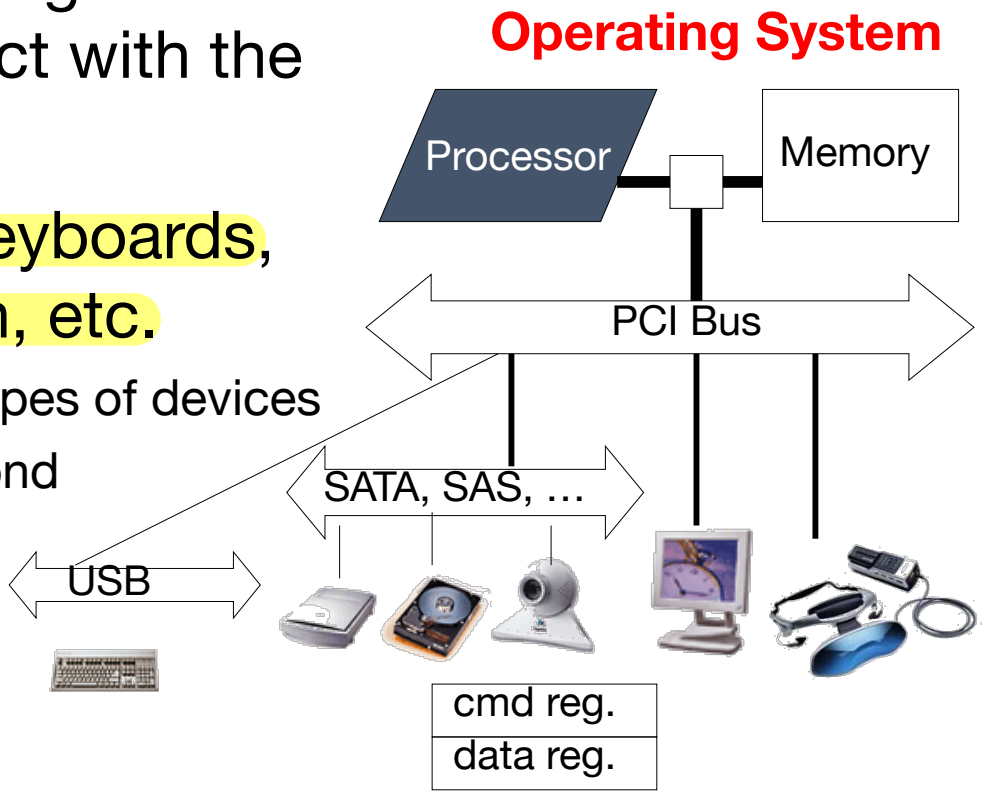
- One program was happily running along...
  - Then all of a sudden the timer triggered
- All its data got stored away...
- And now some other program got restored...
  - From where ***it*** had previously gotten hit by the timer interrupt!
- Congratulations! Now we have multiple programs running!
  - And simply switch between them all on a regular schedule
- Eventually we will get back to the first program...
  - And it is like nothing ever happened, except that between one instruction and the next a lot of time passed...

Let's use the same mechanism to talk to the "outside world"

- How do we talk to devices?
- How do we see if there is data ready to use?
  - E.g, data read in from a file on an SSD
- How do we respond?

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 different types of devices
  - Control these devices, respond to them, and transfer data
  - Present them to user programs so they are useful

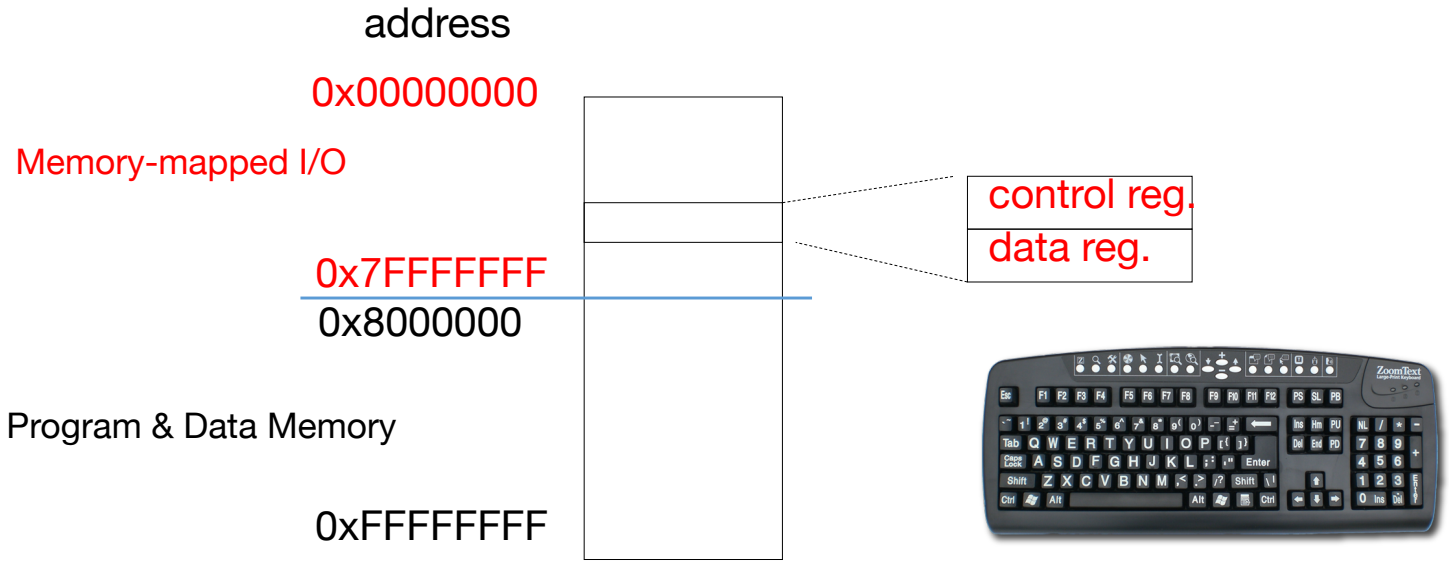


Instruction Set Architecture for I/O

- 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 RISC-V

Memory Mapped I/O

- Certain addresses are not regular memory
- Instead, they correspond to registers in I/O devices. We will see how in an upcoming lecture



Processor-I/O Speed Mismatch

- 3 GHz microprocessor I/O throughput:
  - 12 Gi-B/s (**`lw/sw`**)
    - And then add in all the superscalar-ness and additional cores on die...
  - 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)
    - 985 Mi-B/s (PCIe 3.0 x 1)
    - 1.25 Gi-B/s (USB 3.1 Gen 2 or 10 GigE networking)
    - 15 Gi-B/s (PCIe 3.0 x 16)
    - 40 Gi-B/s (DDR5 DRAM DIMM)
  - These are peak rates – actual throughput is often much lower
- Common I/O devices neither deliver nor accept data matching processor speed

Polling:
Processor Checks Status before Acting

- Device registers generally serve two 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)

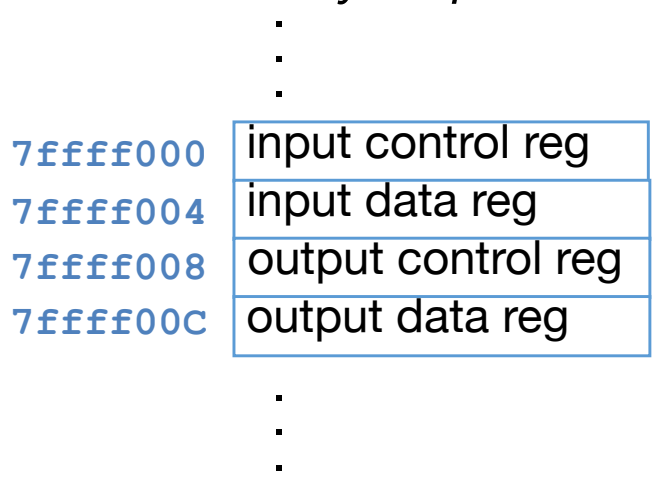
- Input: Read from keyboard into **a0**

```
                lui   t0 0x7ffff #7ffff000 (io addr)
Waitloop:       lw    t1 0(t0)   #read control
                andi  t1 t1 0x1  #ready bit
                beq   t1 zero Waitloop
                lw    a0 4(t0)   #data
```

- Output: Write to display from **a1**

```
                lui   t0 0x7ffff #7ffff000
Waitloop:       lw    t1 8($t0)  #write control
                andi  t1 t1 0x1  #ready bit
                beq   t1 zero Waitloop
                sw    a1 12(t0)  #data
```

Memory Map



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

Cost of Polling?

- Assume for a processor with
  - 1 GHz 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)

% Processor time to poll

- Mouse Polling [clocks/sec]

  = 30 [polls/s] * 400 [clocks/poll] = 12K [clocks/s]

- % Processor for polling:

  $12*10^3$ [clocks/s] / $1*10^9$ [clocks/s] = 0.0012%

  => Polling mouse little impact on processor...

  (Except that you need to know you should be polling...)

% Processor time to poll hard disk

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)?

- Frequency of Polling Disk (rate at which chunks come could off disk)
  = 16 [MB/s] / 16 [B/poll] = 1M [polls/s]
- Disk Polling, Clocks/sec
  = 1M [polls/s] * 400 [clocks/poll]
  = 400M [clocks/s]
- % Processor for polling:

  $400*10^6$ [clocks/s] / $1*10^9$ [clocks/s] = 40%

  => Unacceptable

  (Polling is only part of the problem – main problem is that accessing in small chunks is inefficient)

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 the trap handler in the operating system

Have I/O trigger interrupts too!

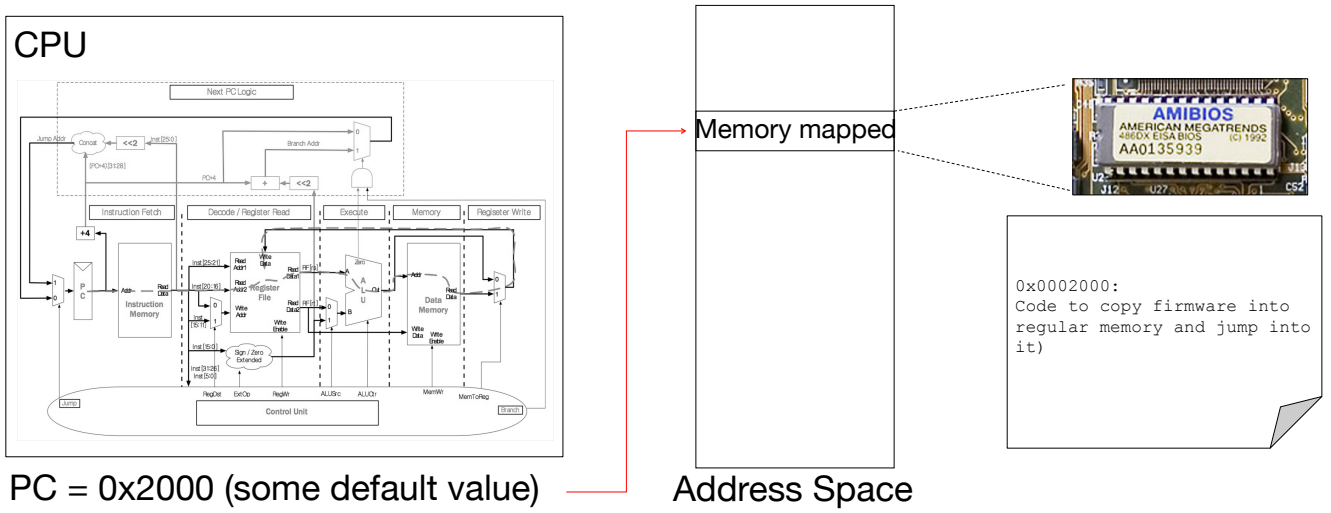
- It just jumps to the trap handler
  - The trap handler than figures out what to do...
    Does it...
  - And returns
- If there is no I/O...
  - This is wonderfully efficient because nothing needs to be done
- If there is a lot of I/O...
  - This gets costly:
    Taking an interrupt is effectively hundreds or thousands of instructions of effort...
    Between the pipeline flush, the cost of the trap handler itself, and the thrashing of caches

Real World:
Polling and Interrupts...

- Low data rate (e.g. mouse, keyboard)
  - In theory: Use interrupts. We ***could*** poll with the timer interrupt, but why?
  - Overhead of interrupts ends up being low
  - But in practice, the USB hardware only supports polling!
- High data rate (e.g. network, disk)
  - Start with interrupts...
    - After all, if there is no data, you don't do anything!
  - Once you start getting data...
    - Then you switch to polling... Keep grabbing the data until it stops...
    - Or, use a trick we will talk about later: Direct Memory Access... The device just writes the data into memory directly
    - And then after the data stops, shift back to interrupts

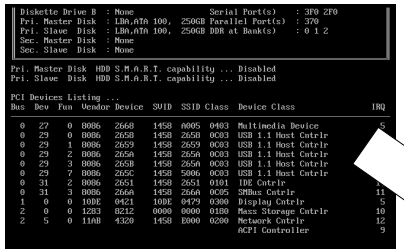
What Happens at Boot?

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

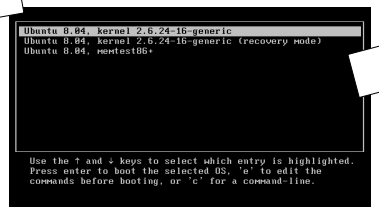


What Happens at Boot?

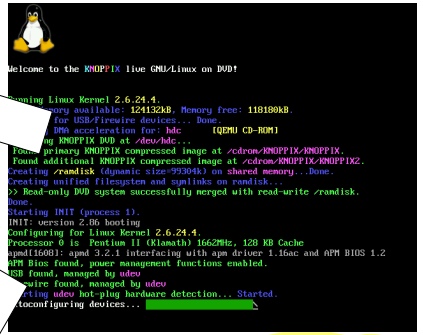
1. BIOS*: Find a storage device and load first sector (block of data)



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



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



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

*BIOS: Basic Input Output System
also "EFI Firmware"

Launching Applications

- Applications are called “processes” in most OSs
  - Each process has its own address space (so each process is **isolated**)
  - A process has one or more **threads** of execution
  - All threads in the system run (pseudo) simultaneously
  - Many user applications actually comprise multiple threads and/or processes (e.g., Chrome)
- Apps are started by another process (e.g., shell) calling an OS routine (This is a system call [“syscall”])
  - Depends on OS, but Linux uses fork to create a new process, and execve (execute file command) to load application
  - Under the hood these call something like **`ecall`**.
- 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 (wait)

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. 0x8FFFFFFF
    - How can hundreds of processes all use memory at location 0x8FFFFFFF?
  - 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

Modern *Virtual Memory* Systems

Illusion of a large, private, uniform store

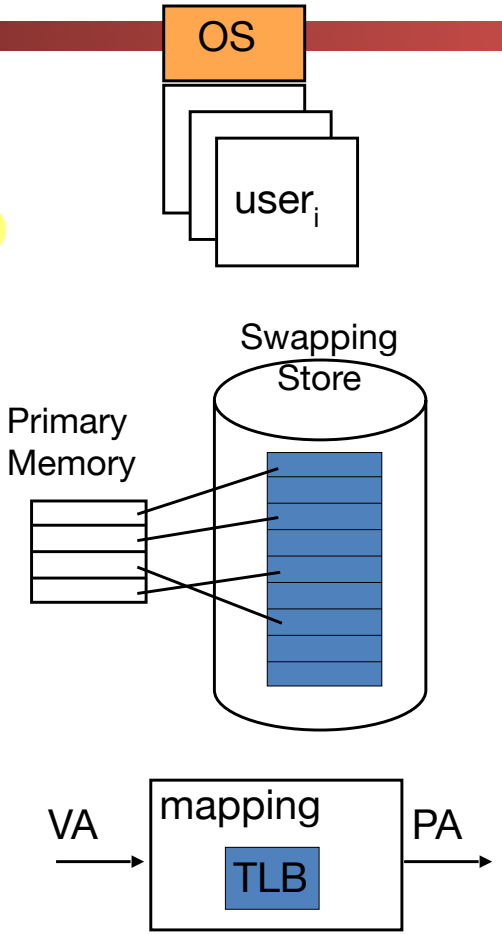
Protection

- several users (processes), each with their private address space and one or more shared address spaces

Demand Paging

- Provides the ability to run programs larger than the primary memory
- Hides differences in machine configurations

The price is address translation on each memory reference



Dynamic Address Translation

Motivation

Multiprogramming, multitasking: Desire to execute more than one process at a time (more than one process can reside in main memory at the same time).

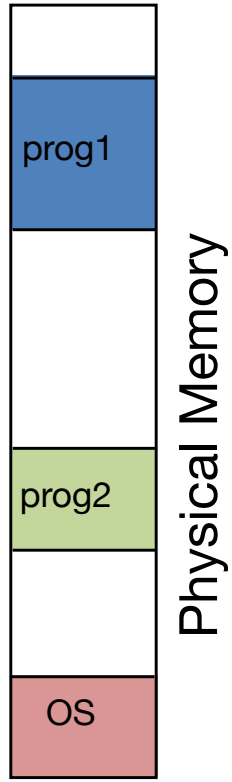
Location-independent programs

Programming and storage management ease

Protection

Independent programs should not affect each other inadvertently

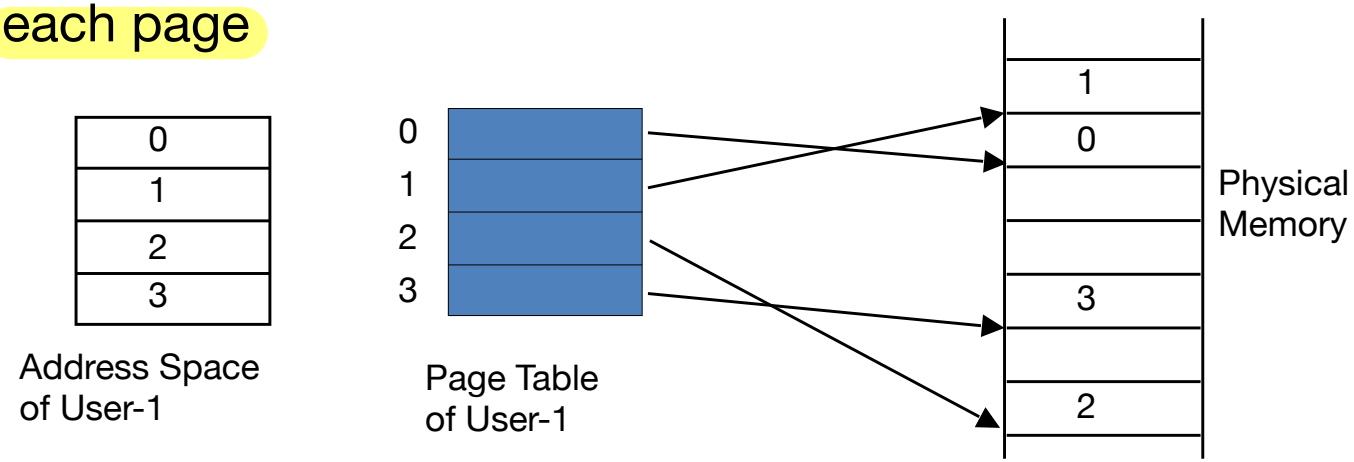
(Note: Multiprogramming drives requirement for resident *supervisor (OS)* software to manage context switches between multiple programs)



Paged Memory Systems

- Processor-generated address can be split into:

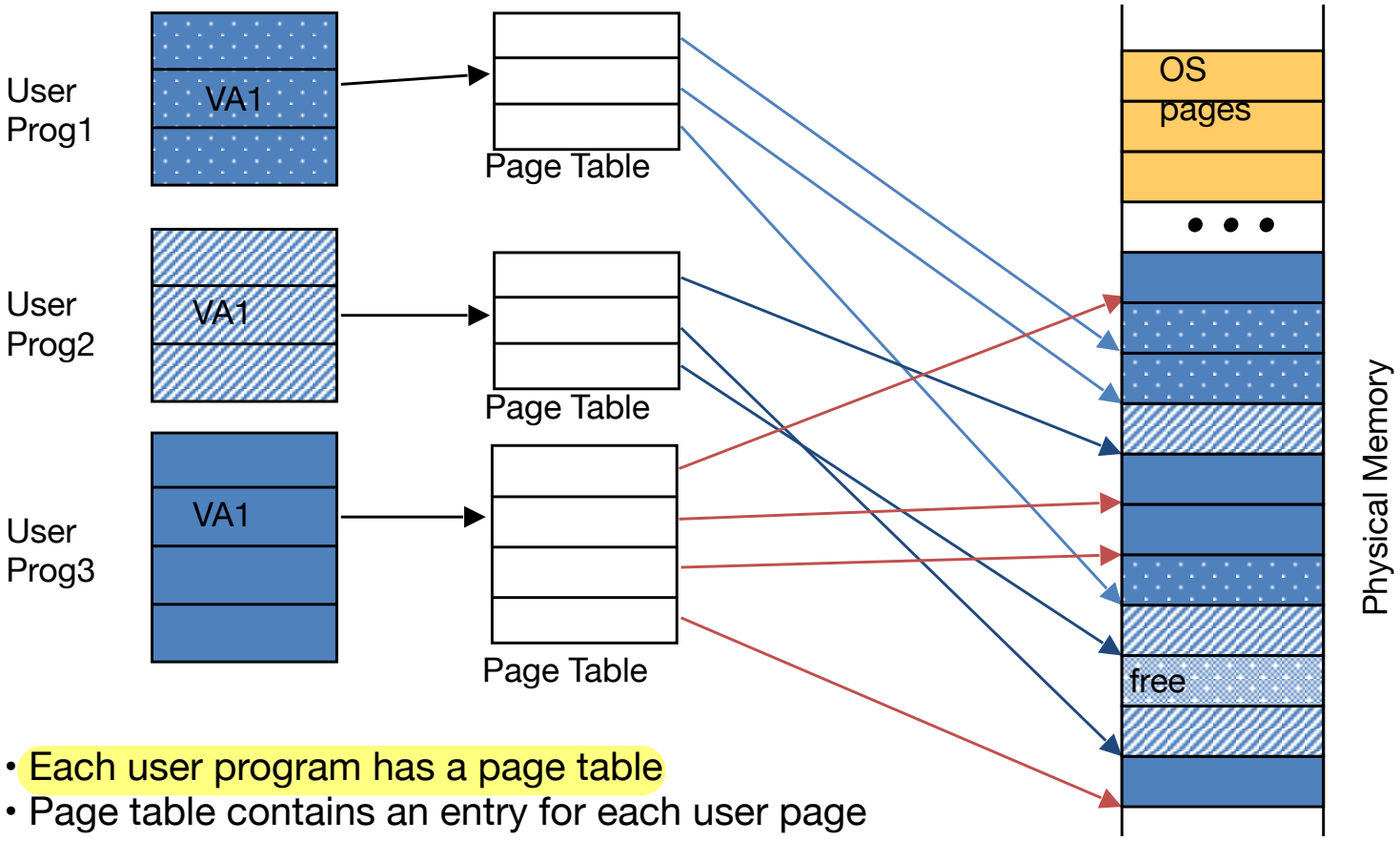
  - A page table contains the physical address of the base of each page



Page tables make it possible to store the pages of a program non-contiguously.

不连续

Private Address Space per User Program



- Each user program has a page table
- Page table contains an entry for each user page

Where Should Page Tables Reside?

- Space required by the page tables (PT) is proportional to the address space, number of users, ...

  ⇒ *Too large to keep in cpu registers*

- Idea: Keep PTs in the main memory
  - Needs one reference to retrieve the page base address and another to access the data word

    ⇒ *doubles the number of memory references!*

    well, until we add caching next time...

Page Tables in Physical Memory