计算机组成与系统结构
Computer Organization & System Architecture

Huang Kejie (黄科杰) 百人计划研究员

Office: 玉泉校区老生仪楼 304

Email address: huangkejie@zju.edu.cn

HP: 17706443800

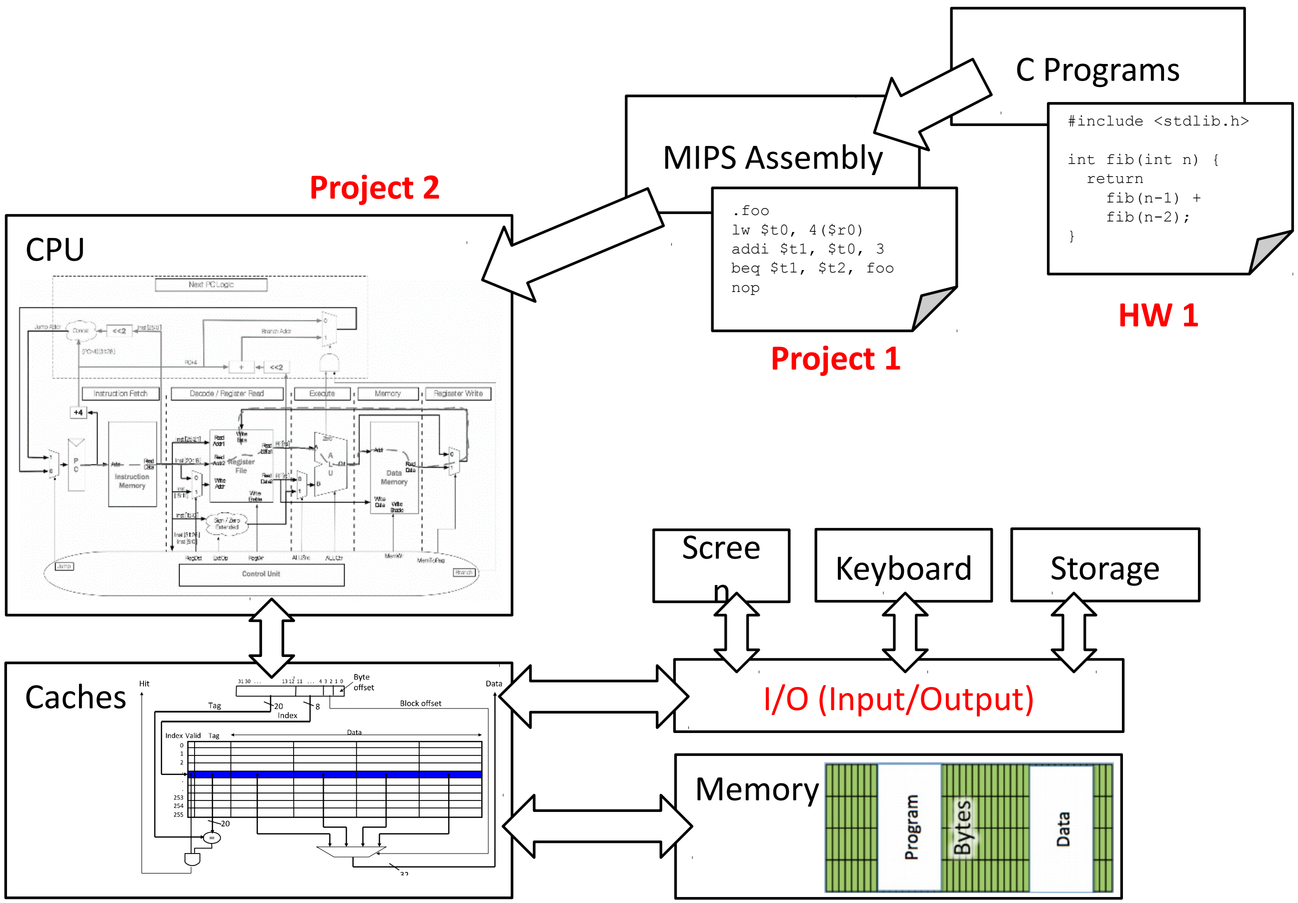
So How is a Laptop Any Different?

Screen

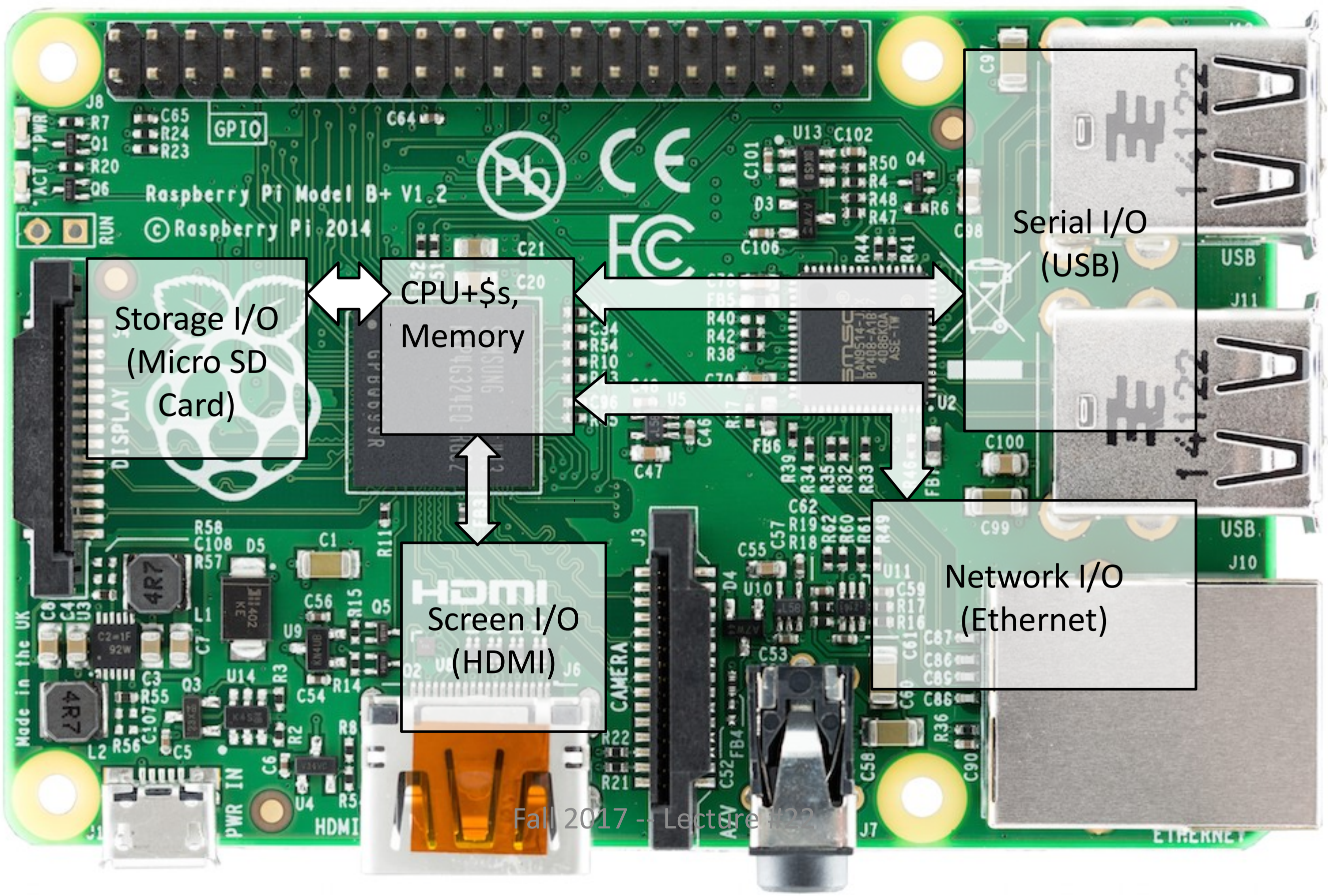
Keyboard

Storage

Adding I/O


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

Raspberry Pi (<$40 on Amazon in 2017)

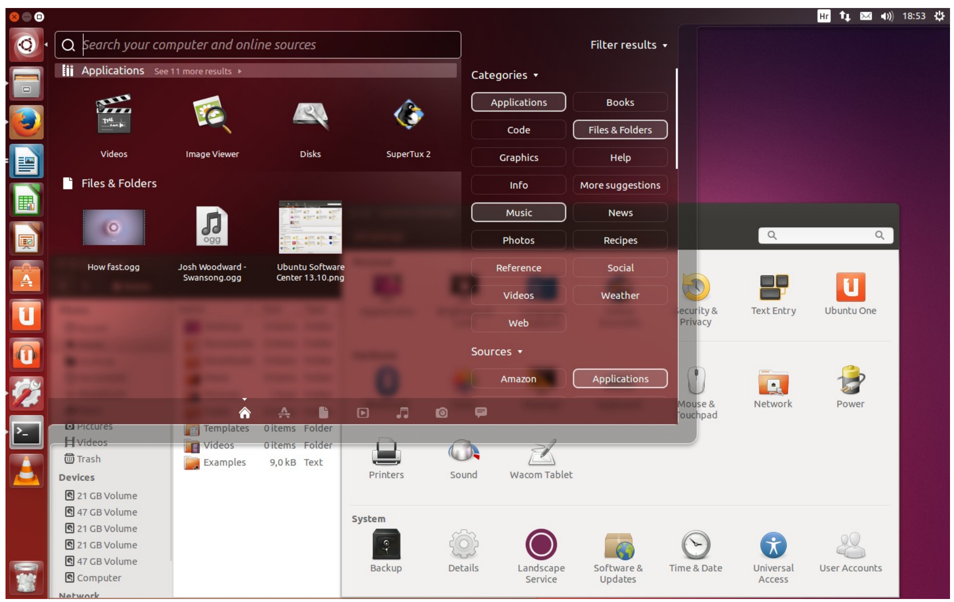


It’s a Real Computer!

But Wait…

- That’s not the same! When we run VENUS, it only executes one program and then stops.

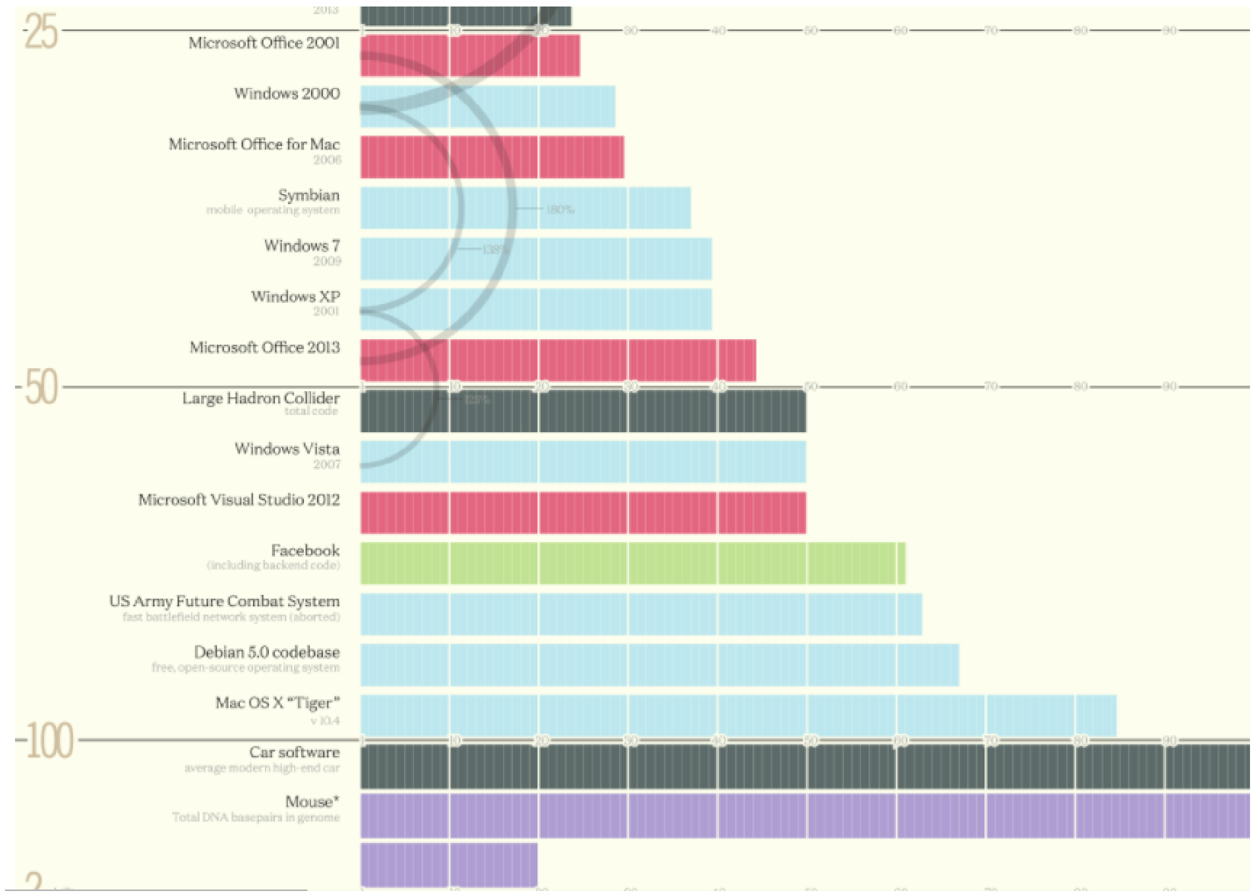
- When I switch on my computer, I get this:



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

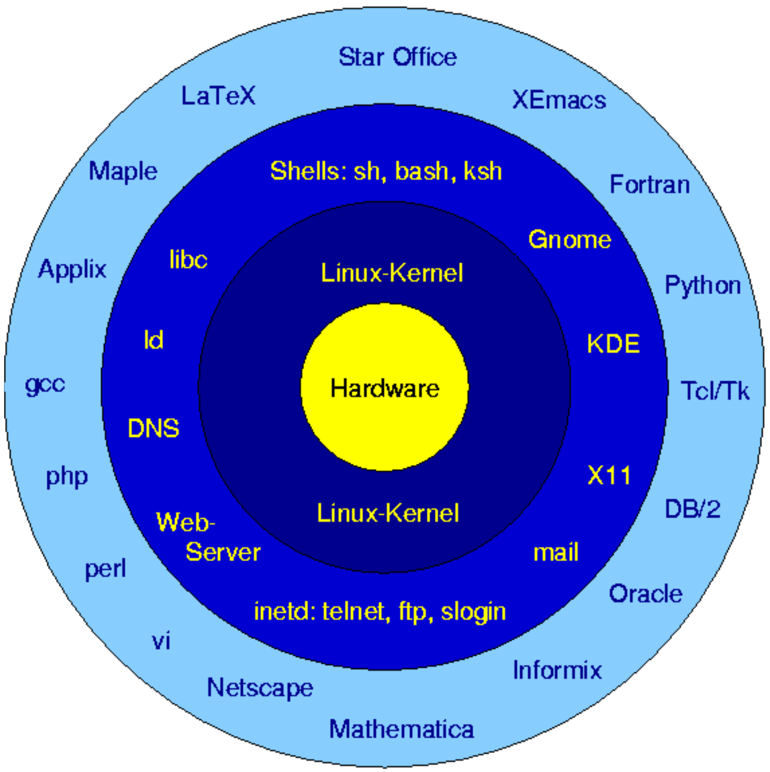
Well, "Just Software"

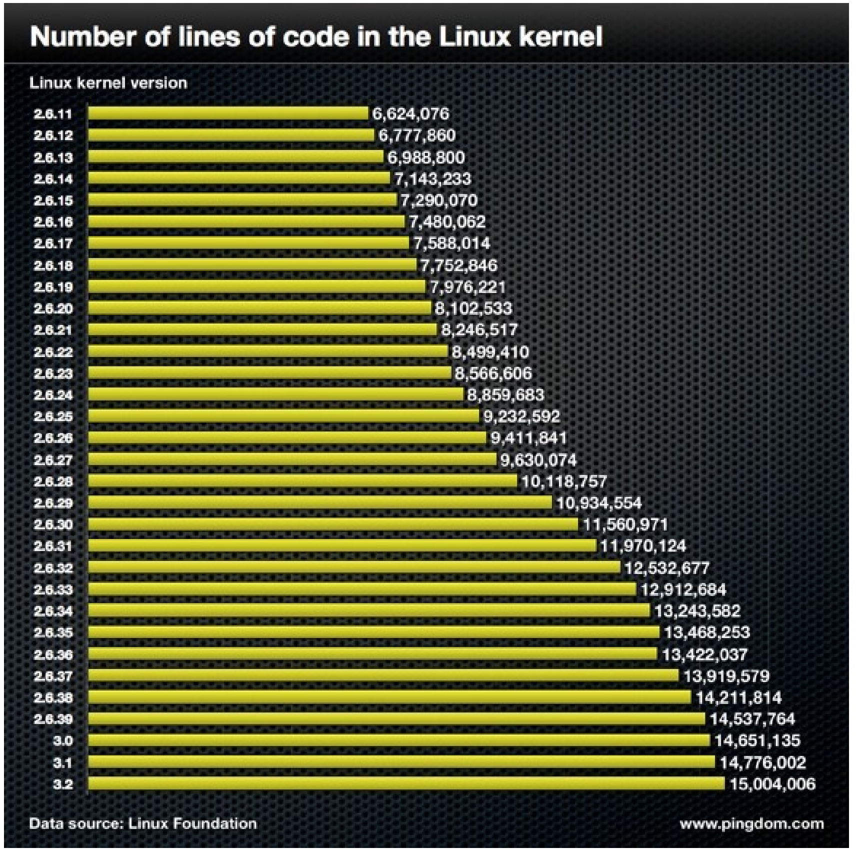
- The biggest piece of software on your machine?
- How many lines of code? These are guesstimates:



Codebases (in millions of lines of code). 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)

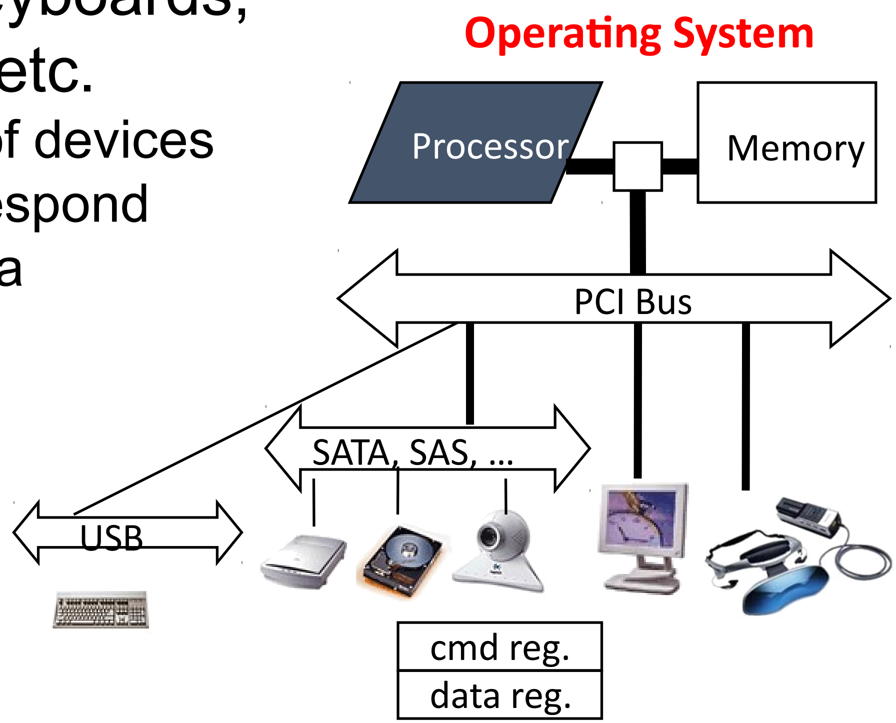
Agenda

- 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

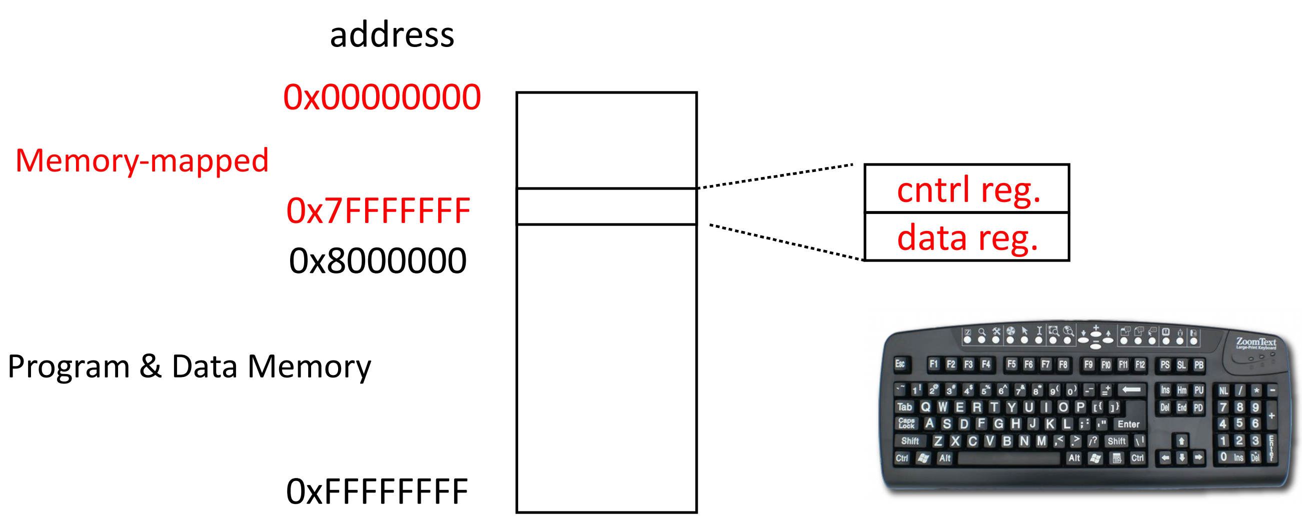


Instruction Set Architecture for I/O

- What must the processor do for I/O?
  - Input: read a sequence of bytes
  - Output: write a sequence of bytes
- Interface options
  - Special input/output instructions & hardware
  - 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



Processor-I/O Speed Mismatch

- 1 GHz 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

Agenda

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

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  #7ffff0000
Waitloop: lw   t1,8($t0)   #write control
          andi t1,t1,0x1   #ready bit
          beq  t1,zero,Waitloop
          sw   a1,12(t0)   #data
```

- "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)

Peer Instruction

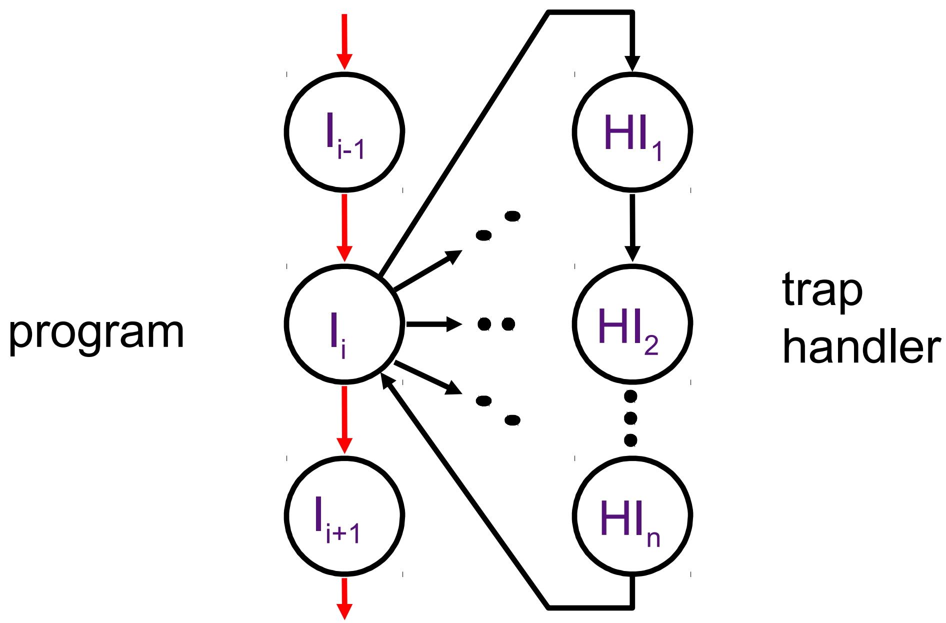
- Hard disk: transfers data in 16-Byte chunks and can transfer at 16 MB/second. No transfer can be missed. What percentage of processor time is spent in polling (assume 1 GHz clock)?

- 2%
- 4%
- 20%
- 40%

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”

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.

Interrupt-Driven I/O

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

Handler Execution
Stack Frame
Stack Frame
Stack Frame

```
Label: sll  t1,s3,2
       add  t1,t1,s5
       lw   t1,0(t1)

       or   s1,s1,t1
       add  s3,s3,s4
       bne  s3,s2,Label
```

Interrupt(SPI0)

```
handler: save registers
         decode interrupt cause
         handle interrupt
             clear interrupt
             restore registers
             mret
```

CPU Vector Interrupt Table

SPI0	handler
...	...

*MEPC: Machine Exception Program Counter

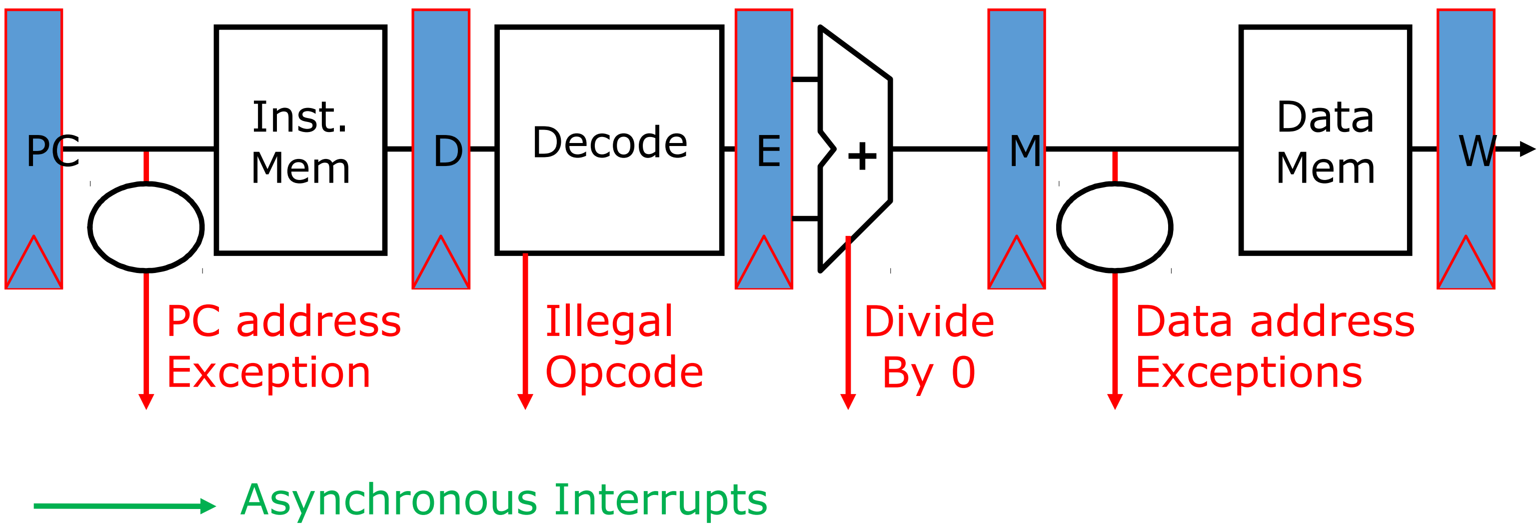
Terminology

- 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., divide by zero, 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

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

Trap Pipeline Diagram

		time												
	t0	t1	t2	t3	t4	t5	t6	t7	. . . .					
(I_1) 096: DIV		IF_1	ID_1	EX_1	MA_1		-		*divide by zero!*					
(I_2) 100: XOR			IF_2	ID_2	EX_2	-	-							
(I_3) 104: SUB				IF_3	ID_3	-	-	-						
(I_4) 108: ADD						IF_4	-	-	-	-				
(I_5) Trap Handler code*									IF_5	ID_5	EX_5	MA_5		WB_5

*MEPC = 100 (instruction following offending ADD)

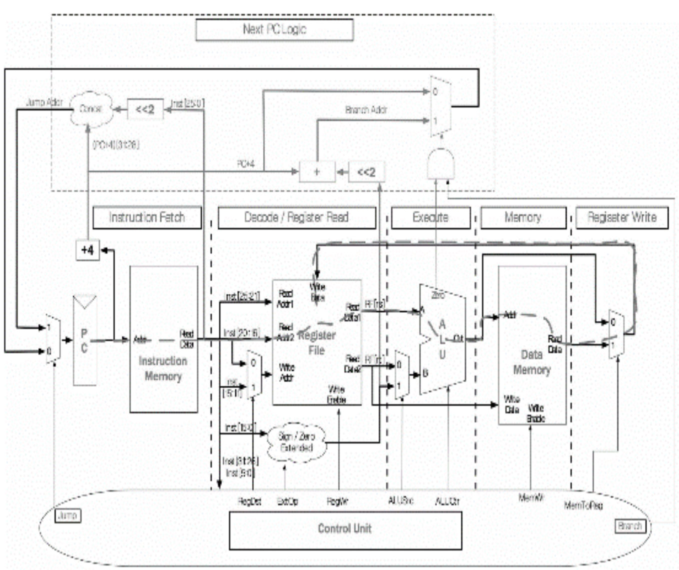
Agenda

- 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 VENUS: the CPU executes instructions from some start address (stored in Flash ROM)

CPU



PC = 0x2000 (some default value)

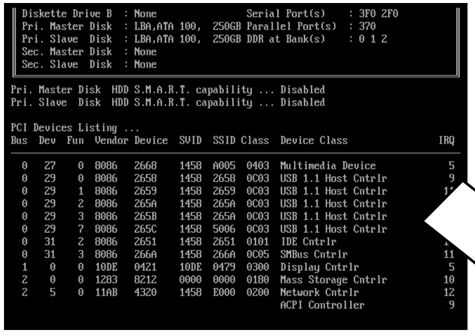
Memory mapped

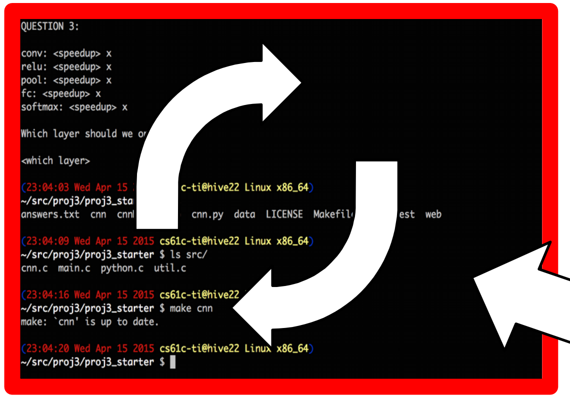
Address Space


```
0x0002000:
Code to copy
firmware into
regular memory
and jump into
it)
```

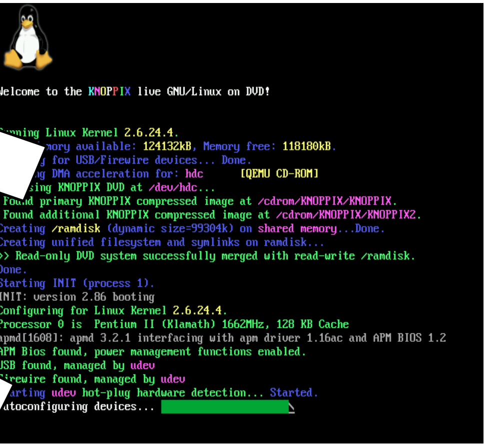
What Happens at Boot?

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

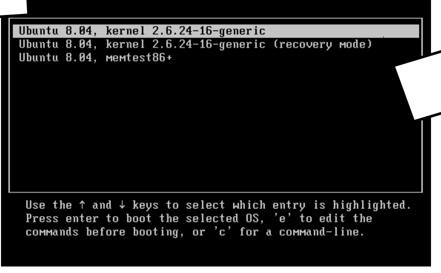




4. Init: Launch an application that waits for input in loop (e.g., Terminal/ Desktop/...



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.

*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 (execute file command) 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.?

Meltdown

Meltdown breaks the most fundamental isolation between user applications and the operating system. This attack allows a program to access the memory, and thus also the secrets, of other programs and the operating system.

Spectre

Spectre breaks the isolation between different applications. It allows an attacker to trick error-free programs, which follow best practices, into leaking their secrets. In fact, the safety checks of said best practices actually increase the attack surface and may make applications more susceptible to Spectre

Supervisor Mode

- 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 (e.g., set by a status bit in a special register)
  - 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. 0x8FFFFFFF
    - How can 100's of programs share 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

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. divide by 0, page fault (next lecture)
  - “Precise” handling: immediately at offending instruction
- Interrupts
  - Notification of external events, e.g., keyboard input, disk or Ethernet traffic
- Multiprogramming and supervisory mode
  - Enables and isolates multiple programs

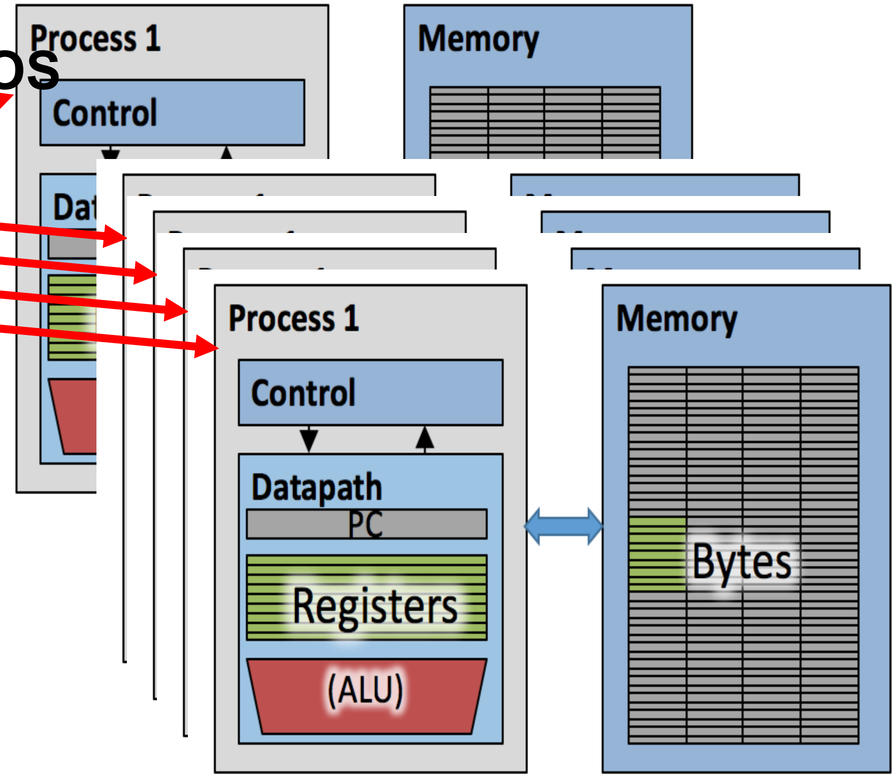
Agenda

- Virtual Memory
- Paged Physical Memory
- Swap Space
- Page Faults
- Hierarchical Page Tables
- Caching Page Table Entries (TLB)

Virtual Machine

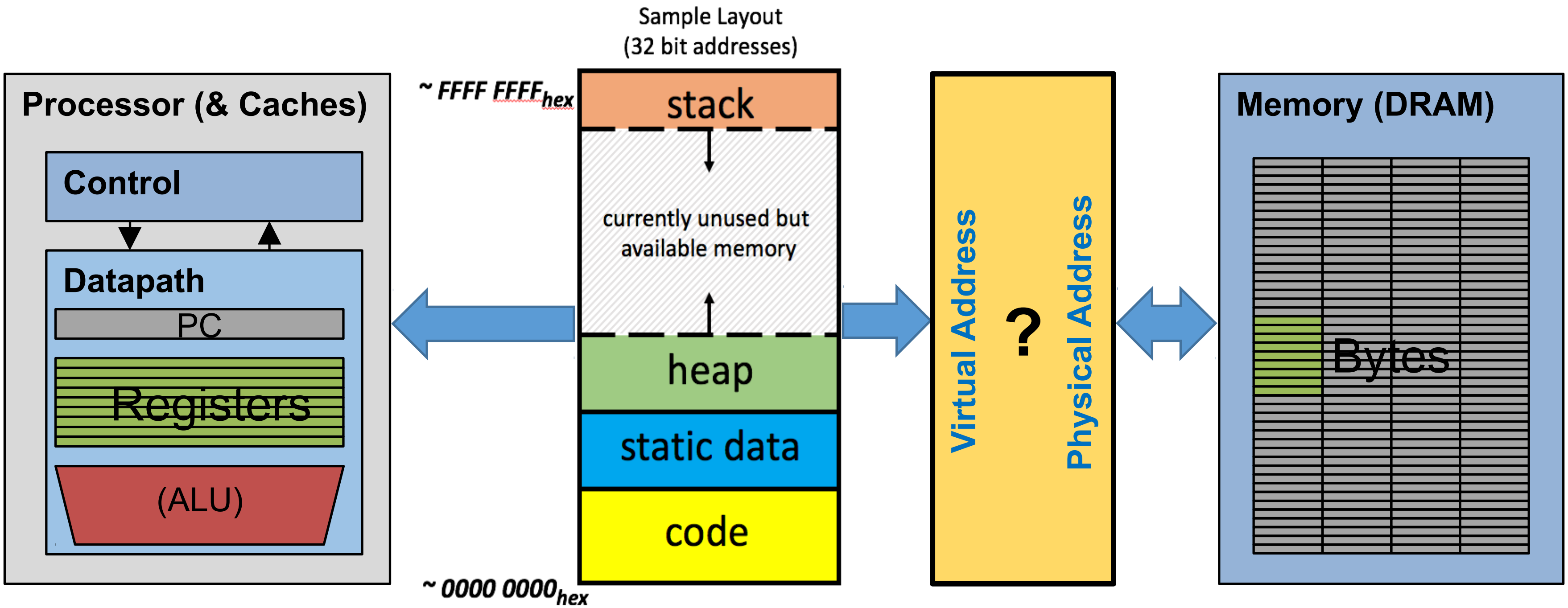
100+ Processes, managed by OS

```
0:04.34 /usr/libexec/UserEventAgent (Aqua)
0:10.60 /usr/sbin/distnoted agent
0:09.11 /usr/sbin/cfprefsd agent
0:04.71 /usr/sbin/usernoted
0:02.35 /usr/libexec/nsurlsessiond
0:28.68 /System/Library/PrivateFrameworks/Calend
0:04.36 /System/Library/PrivateFrameworks/GameCe
0:01.90 /System/Library/CoreServices/cloudphotos
0:49.72 /usr/libexec/secinitd
0:01.66 /System/Library/PrivateFrameworks/TCC.fr
0:12.68 /System/Library/Frameworks/Accounts.fram
0:09.56 /usr/libexec/SafariCloudHistoryPushAgent
0:00.27 /System/Library/PrivateFrameworks/CallHi
0:00.74 /System/Library/CoreServices/mapspushd
0:00.79 /usr/libexec/fmfd
```



- 100’s of processes
  - OS multiplexes these over available cores
- But what about memory?
  - There is only one!
  - We cannot just ”save” its contents in a context switch …

Virtual vs. Physical Addresses



Many of these (software & hardware cores)

One main memory

- Processes use virtual addresses, e.g., 0 … 0xffff,ffff
  - Many processes, all using same (conflicting) addresses
- Memory uses physical addresses (also, e.g., 0 ... 0xffff,ffff)
- *Memory manager maps virtual to physical addresses*

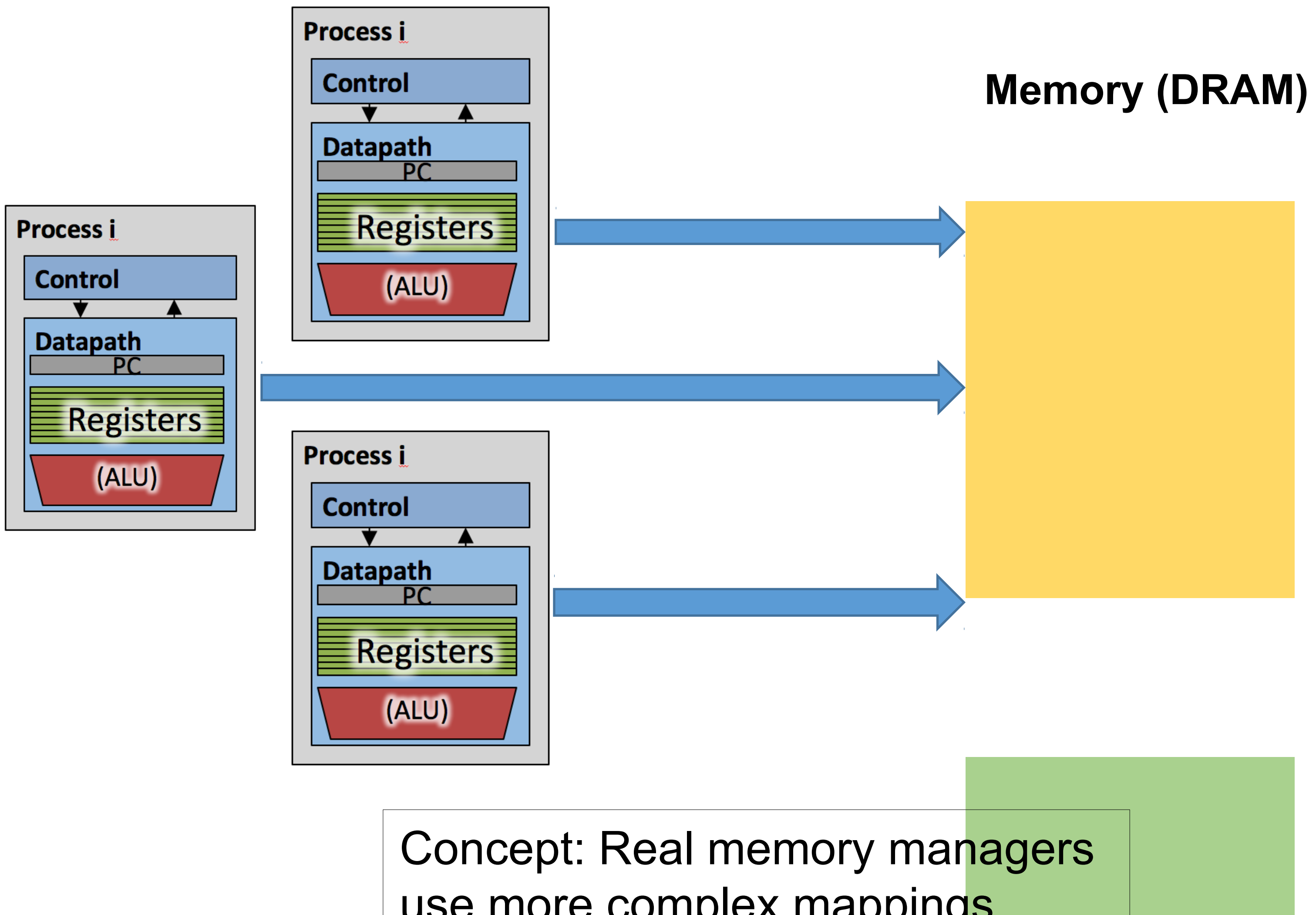
Address Spaces

- Address space = set of addresses for all available memory locations

- Now, two kinds of memory addresses:
  - Virtual Address Space
    - Set of addresses that the user program knows about
  - Physical Address Space
    - Set of addresses that map to actual physical locations in memory
    - Hidden from user applications

- Memory manager maps between these two address spaces

Conceptual Memory Manager



Concept: Real memory managers use more complex mappings.

Responsibilities of Memory Manager

- Map virtual to physical addresses

- Protection:
  - Isolate memory between processes
  - Each process gets dedicate ”private” memory
  - Errors in one program won’t corrupt memory of other program
  - Prevent user programs from messing with OS’ memory

- Swap memory to disk
  - Give illusion of larger memory by storing some content on disk
  - Disk is usually much larger and slower than DRAM
    - Use “clever” caching strategies

Agenda

- Virtual Memory
- Paged Physical Memory
- Swap Space
- Page Faults
- Hierarchical Page Tables
- Caching Page Table Entries (TLB)

Memory Manager

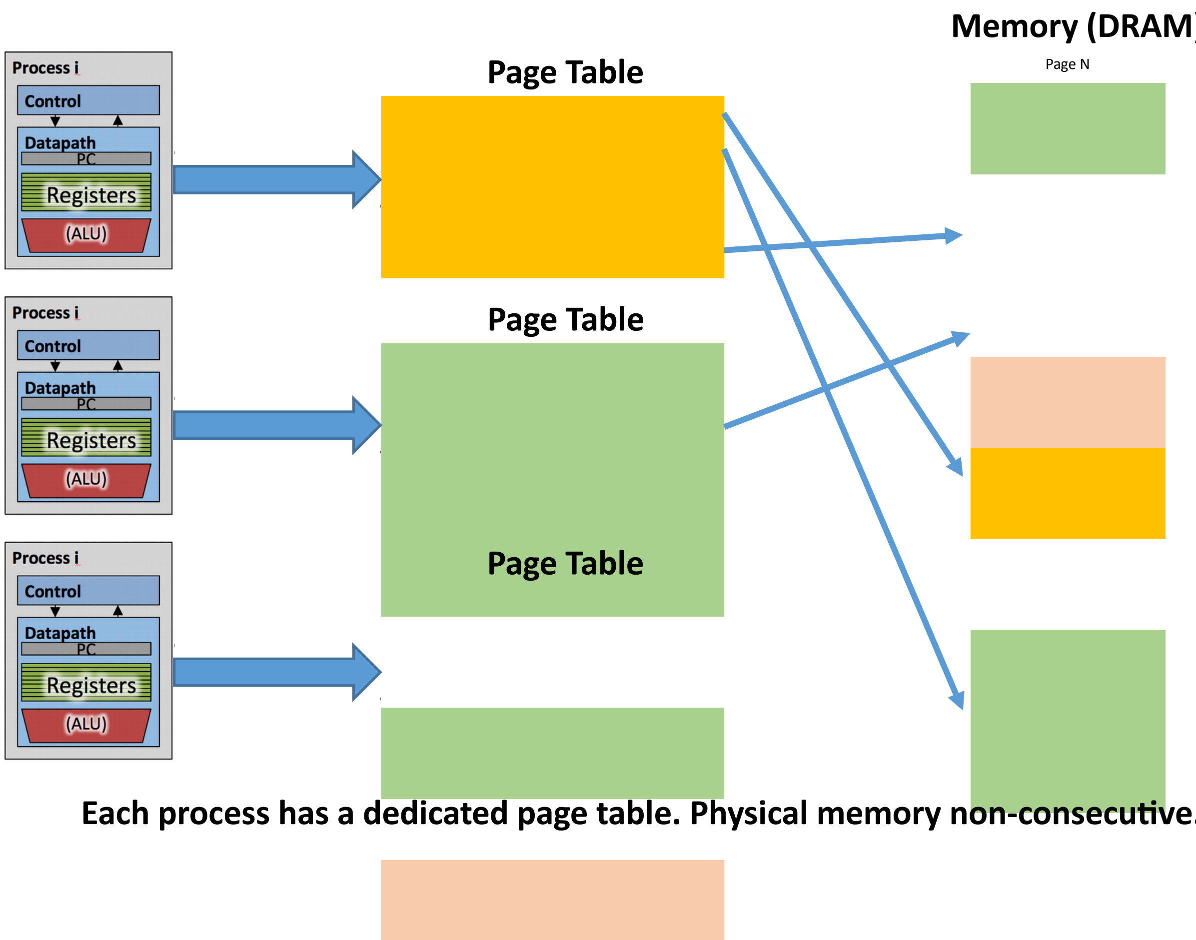
- Several options

- Today “paged memory” dominates
  - Physical memory (DRAM) is broken into pages
  - Typical page size: 4 KiB+

Virtual address (e.g., 32 Bits)

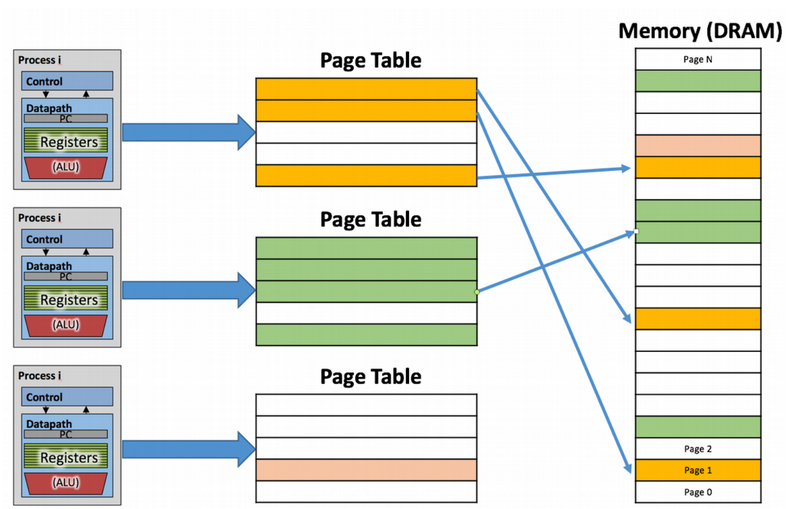
page number (e.g., 20 Bits)	offset (e.g., 12 Bits)

Paged Memory

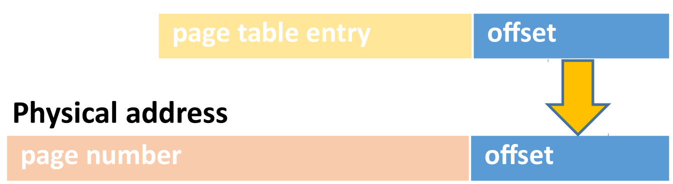


Each process has a dedicated page table. Physical memory non-consecutive.

Paged Memory Address Translation



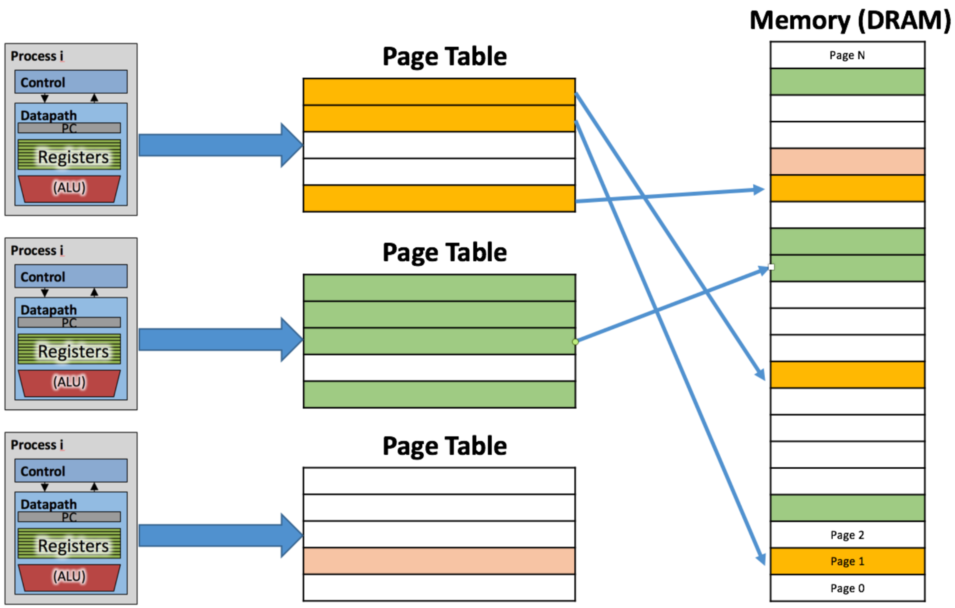
Virtual address (e.g. 32 Bits)



Physical addresses may (but do not have to) have more or fewer bits than virtual addresses

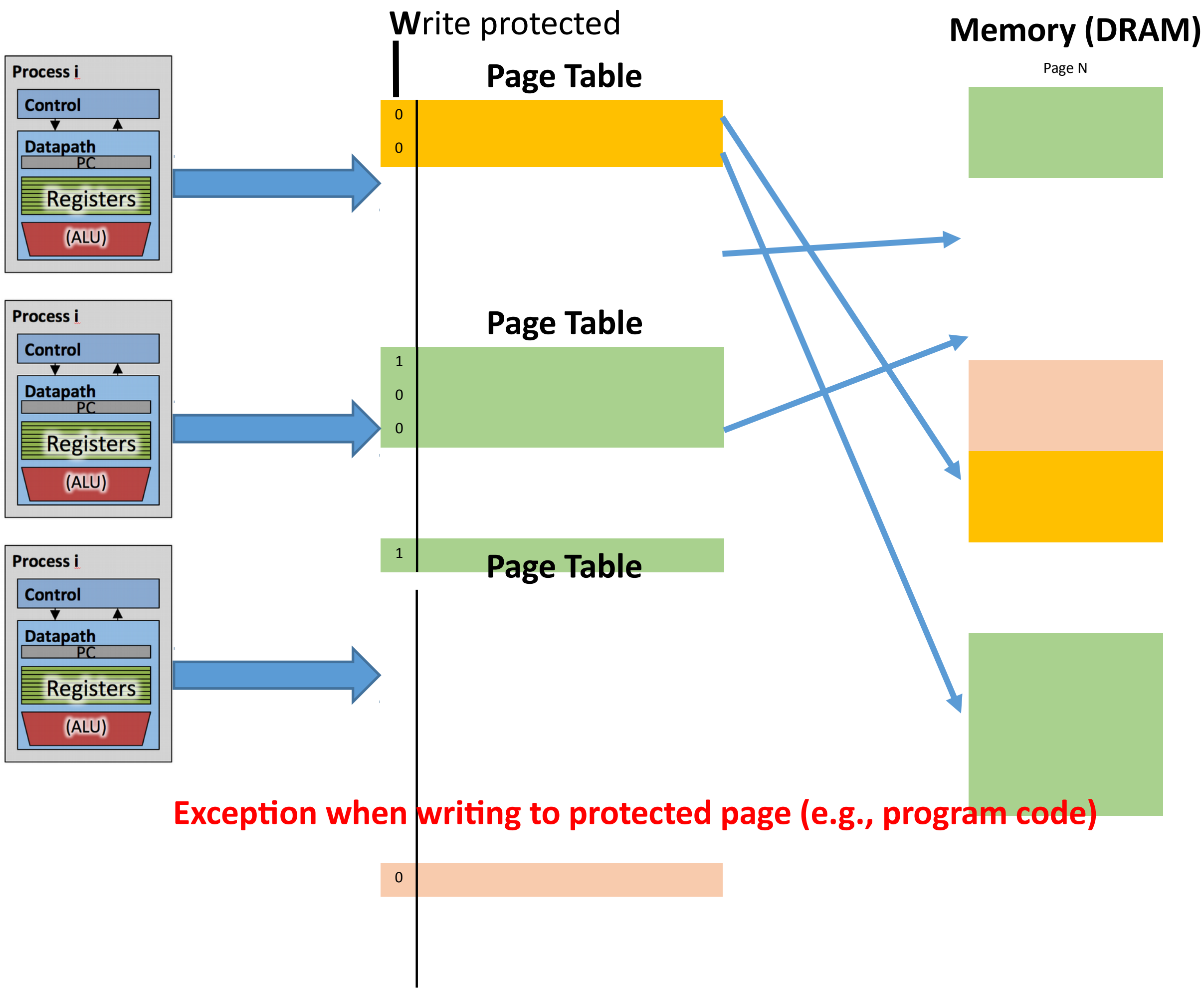
- OS keeps track of which process is active
  - Chooses correct page table
- Memory manager extracts page number from virtual address
- Looks up page address in page table
- Computes physical memory address from sum of
  - Page address and
  - Offset (from virtual address)

Protection



- Assigning different pages in DRAM to processes also keeps them from accessing each others memory
  - Isolation
  - Page tables handled by OS (in supervisory mode)
- Sharing is also possible
  - OS may assign same physical page to several processes

Write Protection

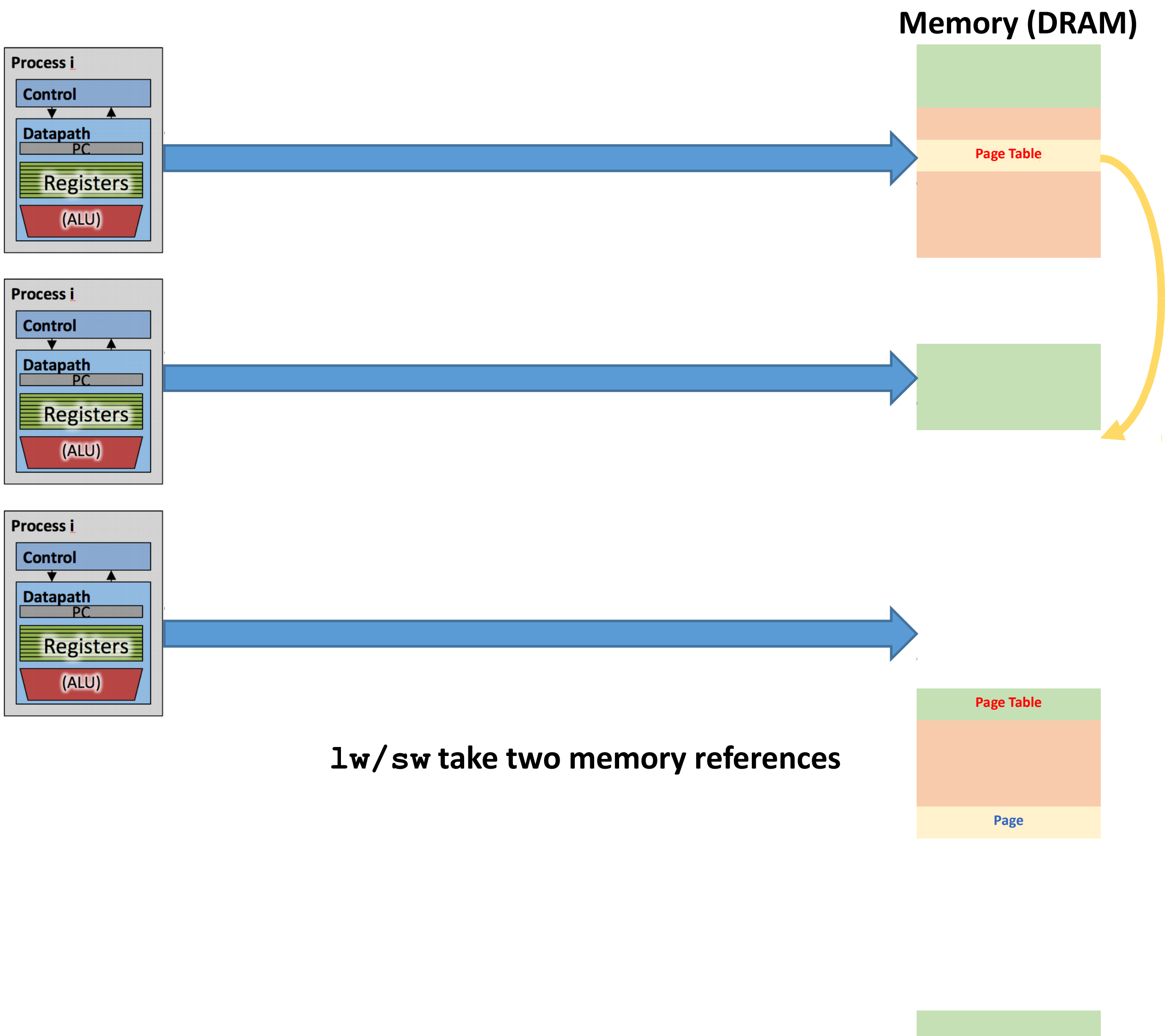


Exception when writing to protected page (e.g., program code)

Where Do Page Tables Reside?

- E.g., 32-Bit virtual address, 4-KiB pages
  - Single page table size:
  - 4×2^{20} Bytes = 4-MiB
  - 0.1% of 4-GiB memory
  - But much too large for a cache!
- Store page tables in memory (DRAM)
  - Two (slow) memory accesses per **`lw/sw`** on cache miss
  - How could we minimize the performance penalty?
    - Transfer blocks (not words) between DRAM and processor cache
      - Exploit spatial locality
  - Use a cache for frequently used page table entries …

Paged Table Storage in DRAM



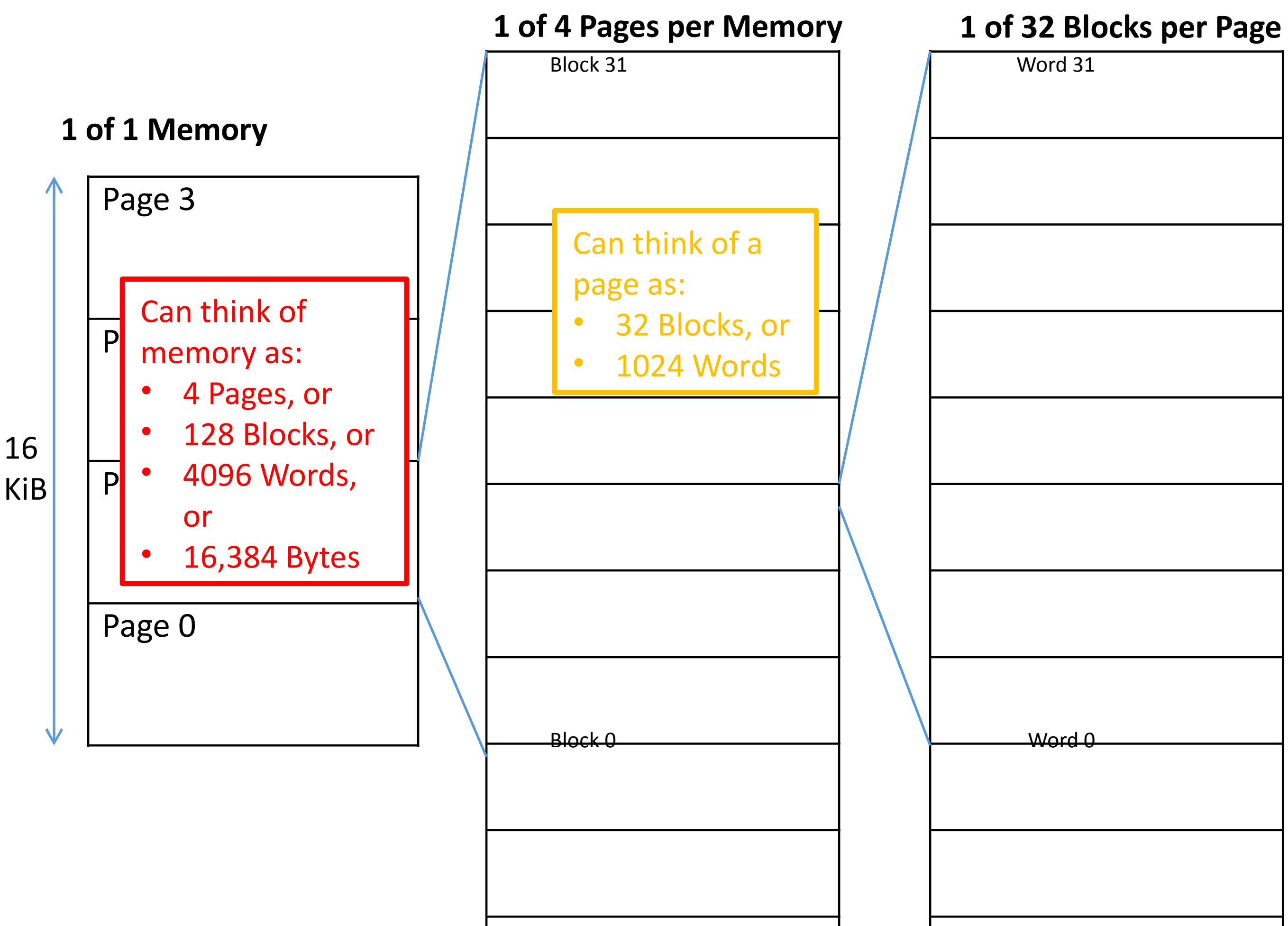
`lw`/`sw` take two memory references

Blocks vs. Pages

- In caches, we dealt with individual *blocks*
  - Usually ~64B on modern systems
- In VM, we deal with individual *pages*
  - Usually ~4 KB on modern systems
- Common point of confusion:
  - Bytes,
  - Words,
  - Blocks,
  - Pages
    - Are all just different ways of looking at memory!

Bytes, Words, Blocks, Pages

- E.g.: 16 KiB DRAM, 4 KiB Pages (for VM), 128 B blocks (for caches), 4 B words (for lw/sw)



Size of Page Tables

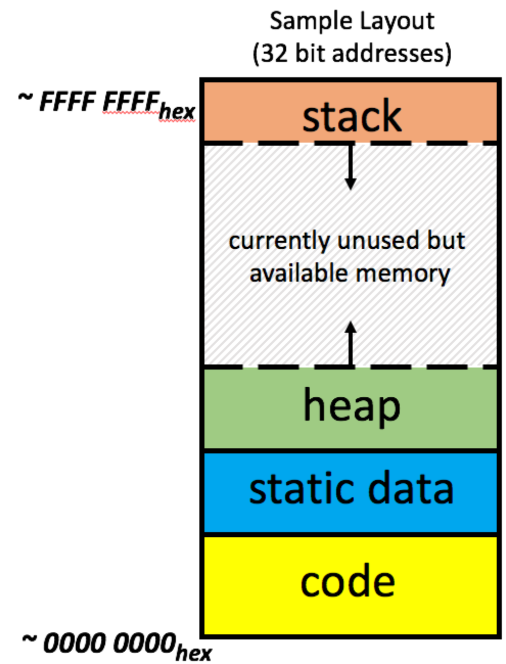
- E.g., 32-Bit virtual address, 4-KiB pages
  - Single page table size:
    - 4 x 2^{20} Bytes = 4-MiB
    - 0.1% of 4-GiB memory
  - Total size for 256 processes (each needs a page table)
    - 256 x 4 x 2^{20} Bytes = 256 x 4-MiB = 1-GiB
    - 25% of 4-GiB memory!

- What about 64-bit addresses?

How can we keep the size of page tables “reasonable”?

Options

- Increase page size
  - E.g., doubling page size cuts PT size in half
  - At the expense of potentially wasted memory
- Hierarchical page tables
  - With decreasing page size
- Most programs use only fraction of memory
  - Split PT in two (or more) parts



Hierarchical Page Table – Exploits Sparsity of Virtual Address Space Use