

# CS 110

## Computer Architecture

### *Operating Systems, Interrupts, Virtual Memory*

Instructor:  
Sören Schwertfeger

<https://robotics.shanghaitech.edu.cn/courses/ca>

School of Information Science and Technology SIST

ShanghaiTech University

Slides based on UC Berkley's CS61C

# CA so far...

C Programs

```
#include <stdlib.h>

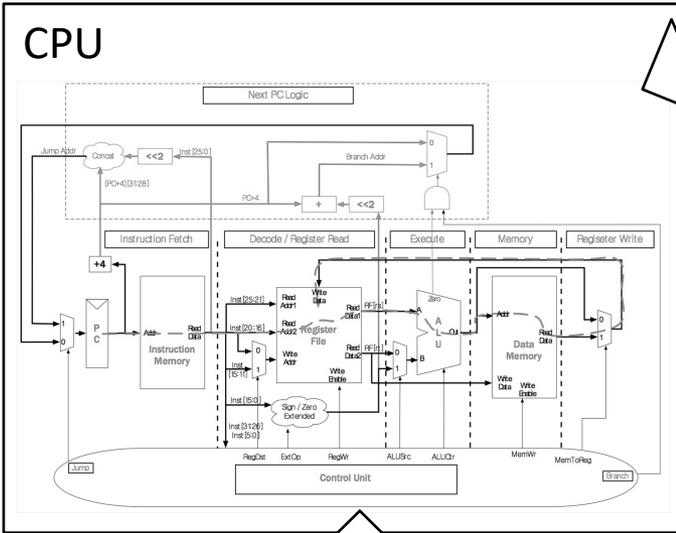
int fib(int n) {
    return
        fib(n-1) +
        fib(n-2);
}
```

Project 1

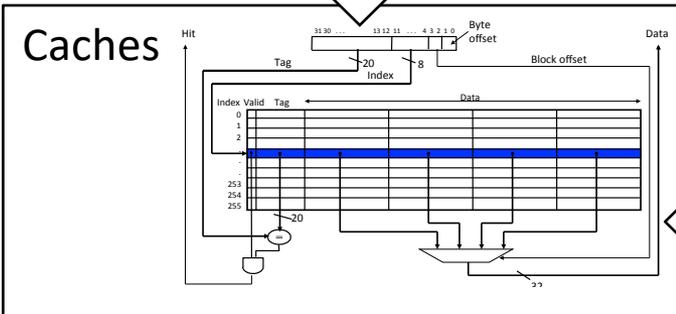
RISC-V Assembly

```
.foo
lw  t0, 4(s1)
addi t1, t0, 3
beq  t1, t2, foo
nop
```

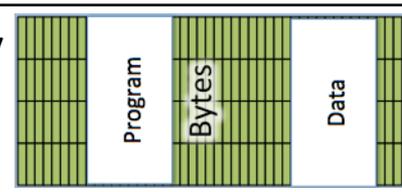
Project 2



PC	Machine Code	Basic Code	Original Code
0x0	0x00111001	lw x2, x1, 4	lw x2, 4(s1)
0x4	0x10000317	auipc x6, 65536	la x6, somedata # address of "somedata" in x6
0x8	0xPFC30313	addi x6, x6, -4	la x6, somedata # address of "somedata" in x6
0xc	0x00032893	lw x5, 0(x6)	lw x5, 0(x6) # (initial) value of "somedata" to x5
0x10	0x00128293	addi x5, x5, 1	addi, x5, x5, 1 # x5 += 1 (label "loop" points here)
0x14	0xPFDFF0EF	jal x1, -4	jal loop # jump to loop



Memory



# So how is this any different?



# Adding I/O

C Programs

```
#include <stdlib.h>

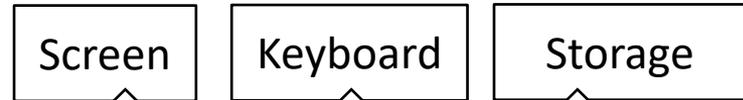
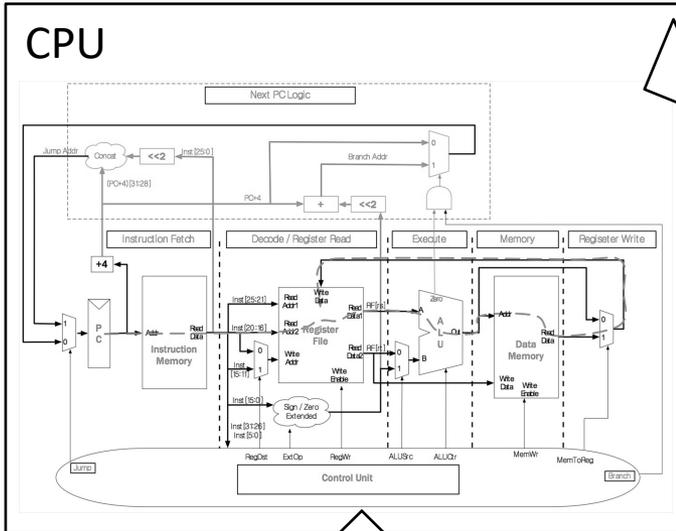
int fib(int n) {
    return
        fib(n-1) +
        fib(n-2);
}
```

Project 1

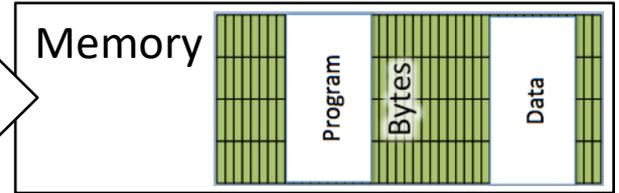
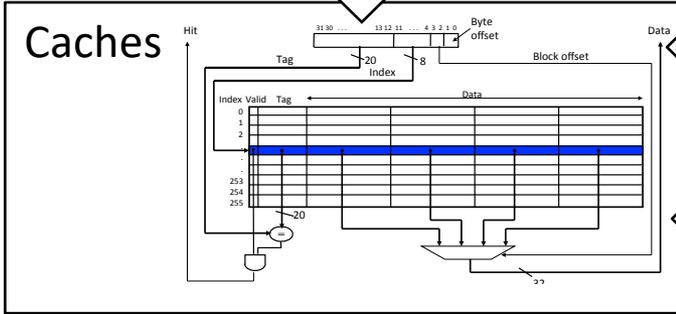
RISC-V Assembly

```
.foo
lw    t0, 4(s1)
addi  t1, t0, 3
beq   t1, t2, foo
nop
```

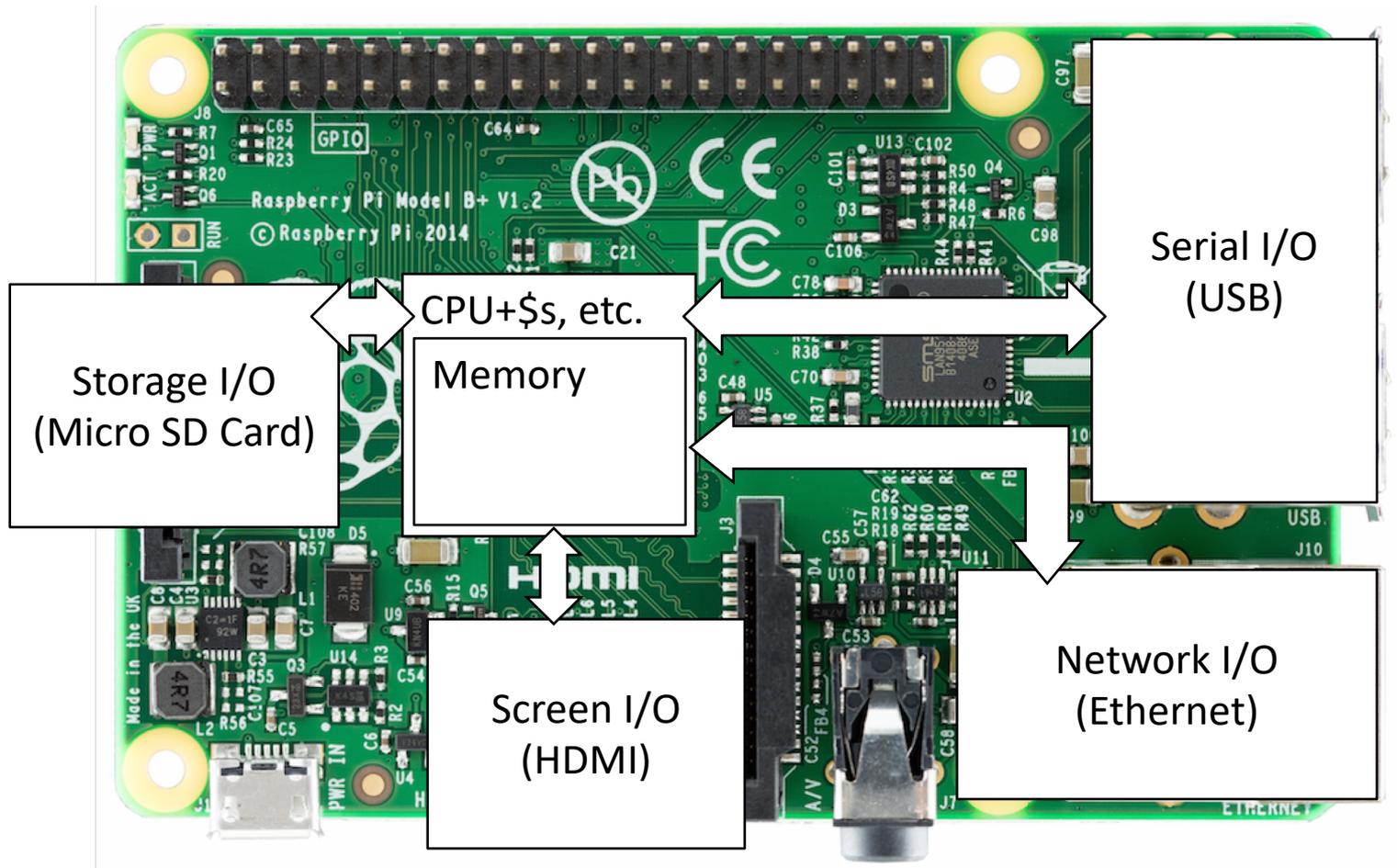
Project 2



I/O (Input/Output)



# Raspberry Pi (< 300RMB on jd.com)



It's a real computer!



# But wait...

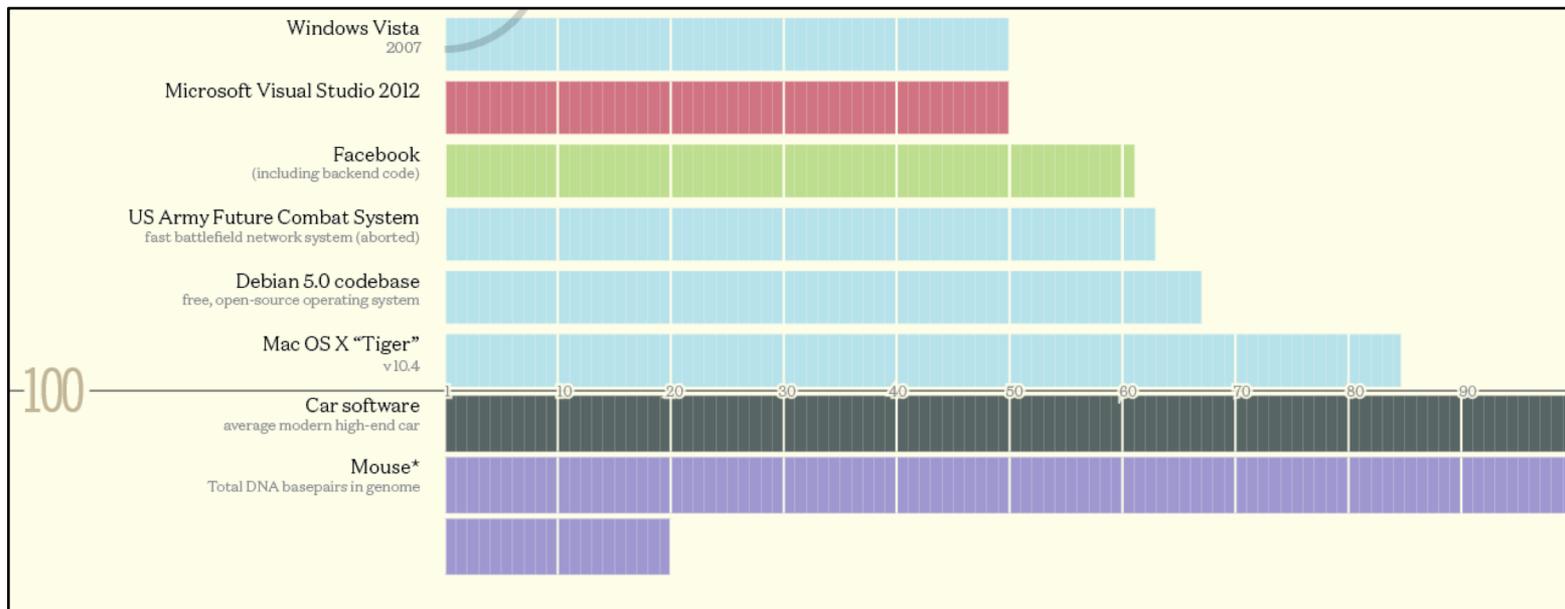
- That's not the same! When we run VENUS, it only executes one program and then stops.
- When I switch on my computer, I get this:



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

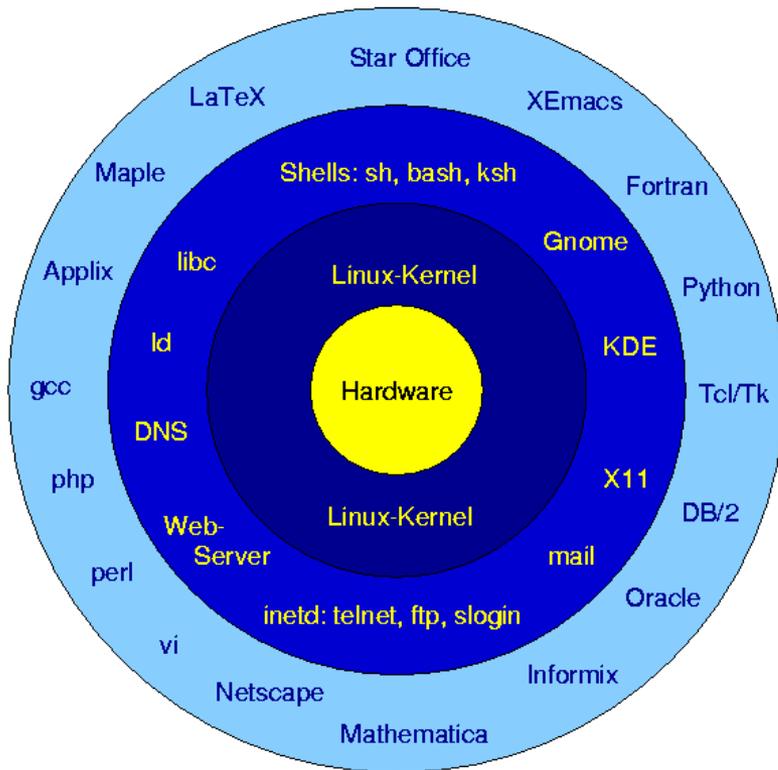
# Well, “just software”

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

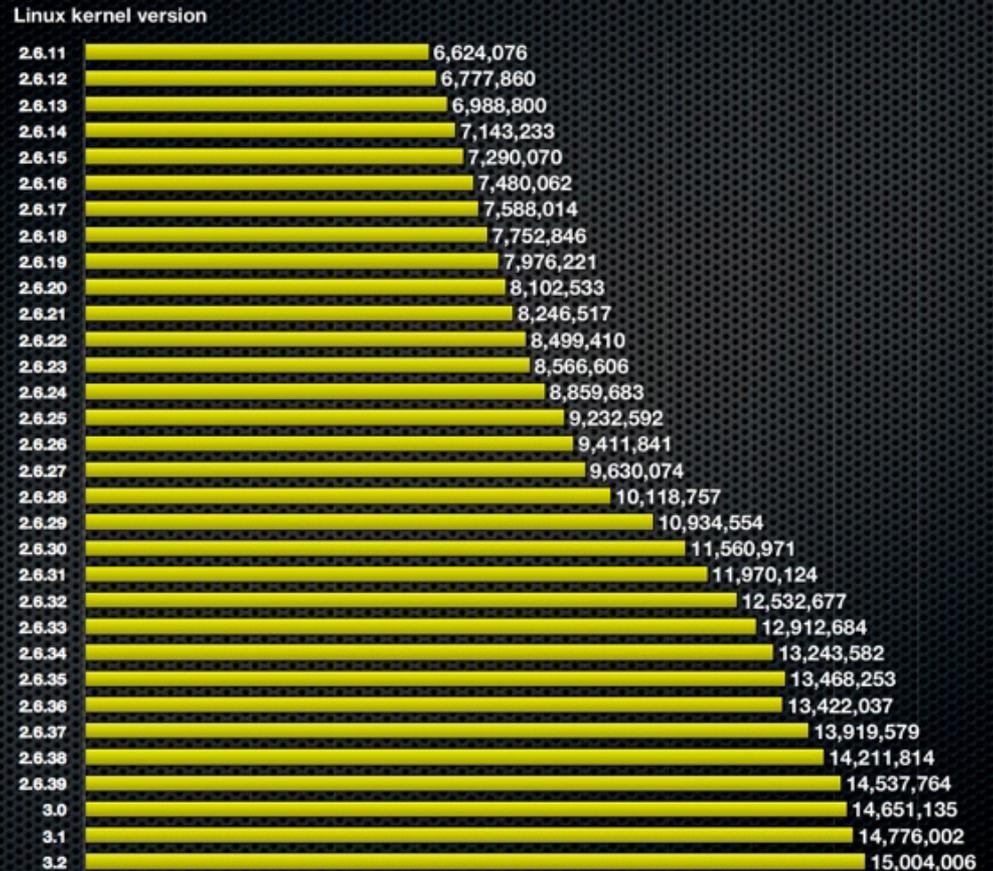


Codebases (in millions of lines of code). CC BY-NC 3.0 — David McCandless © 2013  
<http://www.informationisbeautiful.net/visualizations/million-lines-of-code/>

# Operating System



## Number of lines of code in the Linux kernel



Data source: Linux Foundation

www.pingdom.com

# What does the OS do?

- One of the first things that runs when your computer starts (right after firmware/ bootloader)
- 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)
- Services: File System, Network stack, etc.
- Finds and controls all the devices in the machine in a general way (using “device drivers”)

# Agenda

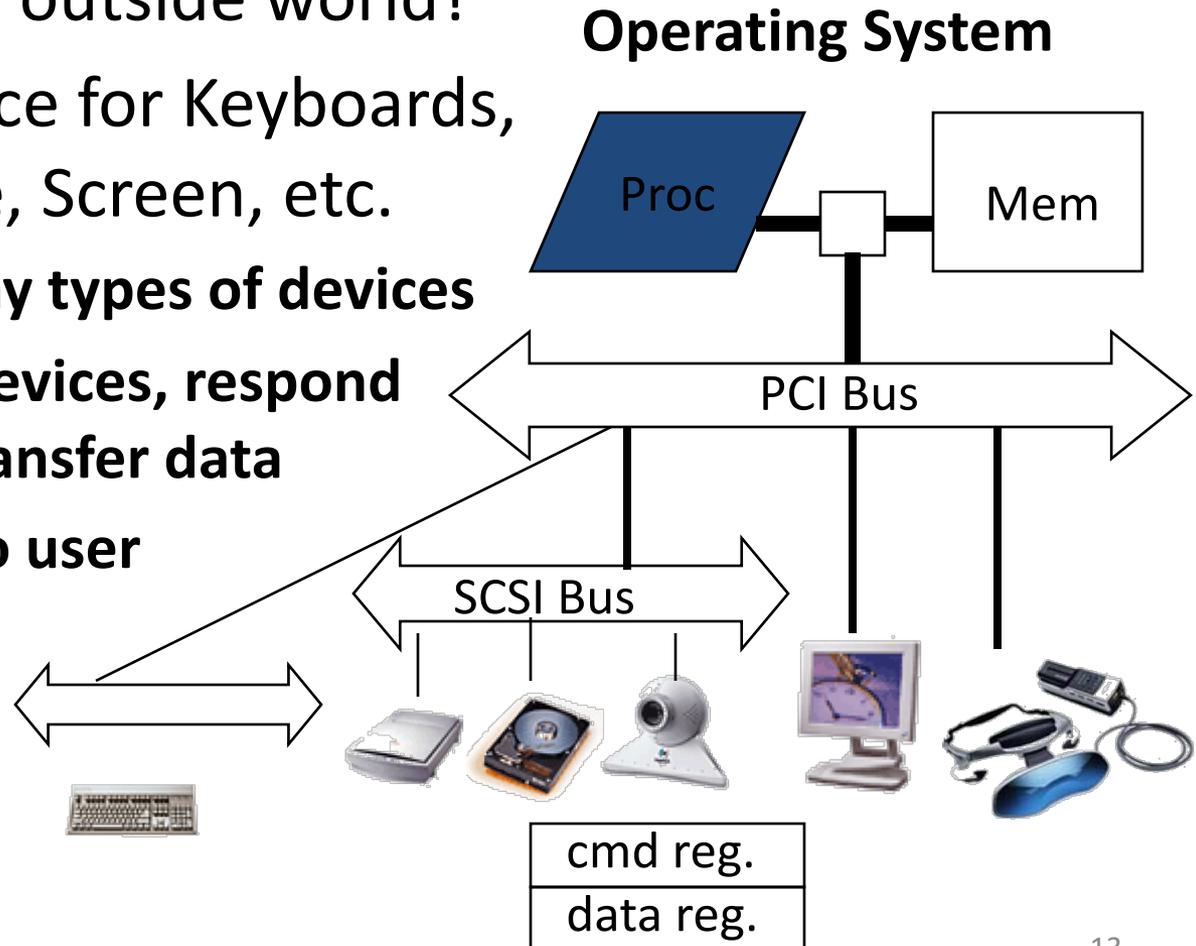
- Devices and I/O
- OS Boot Sequence and Operation
- Multiprogramming/time-sharing
- Introduction to Virtual Memory

# Agenda

- **Devices and I/O**
- OS Boot Sequence and Operation
- Multiprogramming/time-sharing
- Introduction to Virtual Memory

# 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**

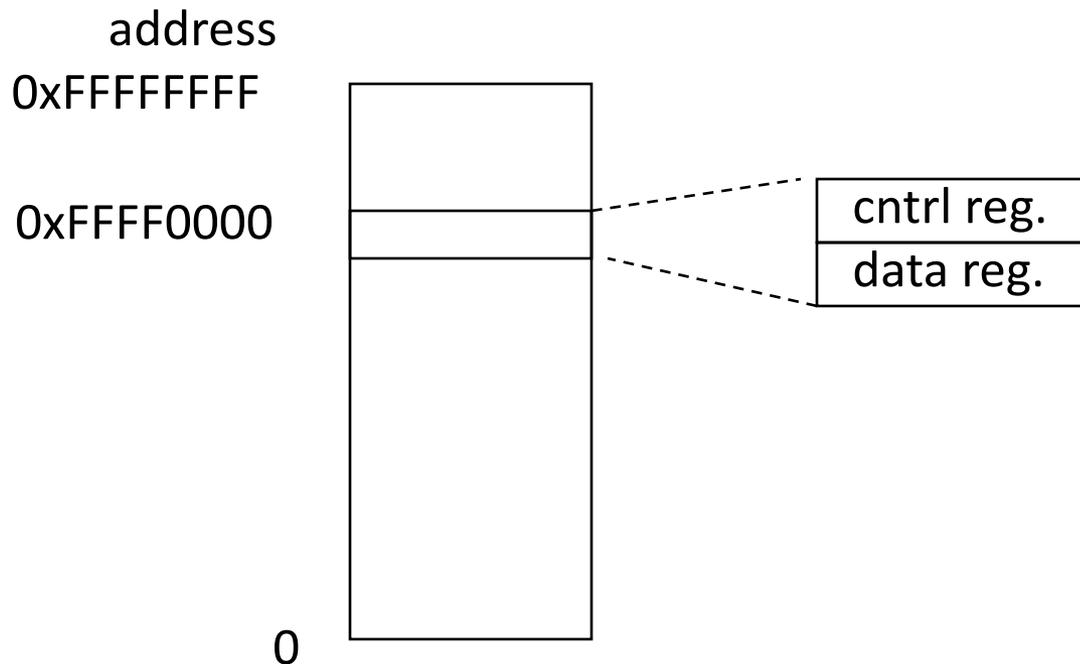


# Instruction Set Architecture for I/O

- What must the processor do for I/O?
  - Input: reads a sequence of bytes
  - Output: writes a sequence of bytes
- Some processors have special input and output instructions
- Alternative model (used by MIPS):
  - Use loads for input, stores for output (in small pieces)
  - Called **Memory Mapped Input/Output**
  - A portion of the address space dedicated to communication paths to Input or Output devices (no memory there)

# 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 can execute 1B load or store instructions per second, or 4,000,000 KB/s data rate
  - I/O data rates range from 0.01 KB/s to 1,250,000 KB/s
- Input: device may not be ready to send data as fast as the processor loads it
  - Also, might be waiting for human to act
- Output: device not be ready to accept data as fast as processor stores it
- **What to do?**

# Processor Checks Status before Acting

- Path to a device generally has 2 registers:
  - **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) to say it's OK
- Processor then loads from (input) or writes to (output) data register
  - Load from or Store into Data Register resets Ready bit (1 => 0) of Control Register
- This is called "**Polling**"

# I/O Example (polling)

- Input: Read from keyboard into a0

```
li    t0, 0xffff0000 #ffff0000
Waitloop: lw    t1, 0(t0)      #control
        andi  t1, t1, 0x1
        beq   t1, zero, Waitloop
        lw    a0, 4(t0)      #data
```

- Output: Write to display from a0

```
li    t0, 0xffff0000 #ffff0000
Waitloop: lw    t1, 8(t0)      #control
        andi  t1, t1, 0x1
        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 a 1GHz clock it takes 400 clock cycles for a polling operation (call polling routine, accessing the device, and returning).  
Determine % of processor time for polling
  - Mouse: polled 30 times/sec so as not to miss user movement
  - Hard disk: assume transfers data in 16-Byte chunks and can transfer at 16 MB/second. Again, no transfer can be missed. (we'll come up with a better way to do this)

## % 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

# Question

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; 400 cycles per poll)?

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

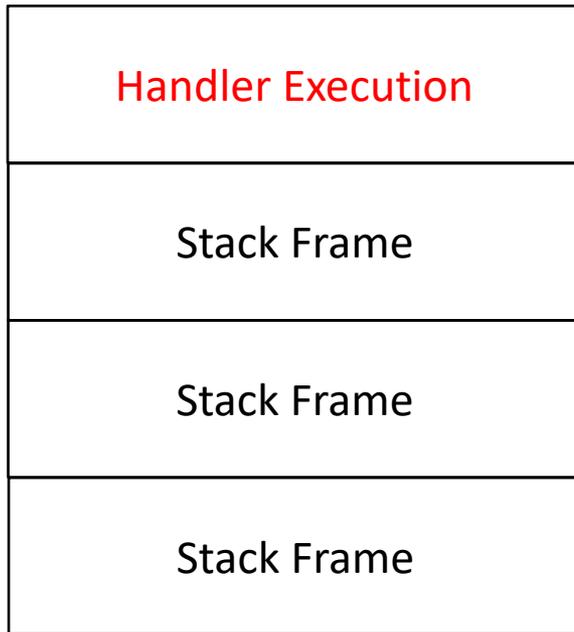
# % Processor time to poll hard disk

- Frequency of Polling Disk  
=  $16 \text{ [MB/s]} / 16 \text{ [B/poll]} = 1\text{M [polls/s]}$
- Disk Polling, Clocks/sec  
=  $1\text{M [polls/s]} * 400 \text{ [clocks/poll]}$   
=  $400\text{M [clocks/s]}$
- % Processor for polling:  
 $400 * 10^6 \text{ [clocks/s]} / 1 * 10^9 \text{ [clocks/s]} = 40\%$   
=> Unacceptable  
(Polling is only part of the problem – main problem is that accessing in small chunks is inefficient)

# What is the alternative to polling?

- Wasteful to have processor spend most of its time “spin-waiting” for I/O to be ready
- Would like an unplanned procedure call that would be invoked only when I/O device is ready
- Solution: use **exception mechanism** to help I/O. **Interrupt** program when I/O ready, return when done with data transfer
- Allow to register (post) **interrupt handlers**: functions that are called when an interrupt is triggered

# 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 (**needs to store any state**)
4. Handler run on current stack and returns on finish (thread doesn't notice that a handler was run)

```
handler: li t0, 0xffff0000  
lw t1, 0(t0)  
andi t1, t1, 0x1  
lw a0, 4(t0)  
sw t1, 8(t0)  
ret
```

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



Interrupt(SPI0)

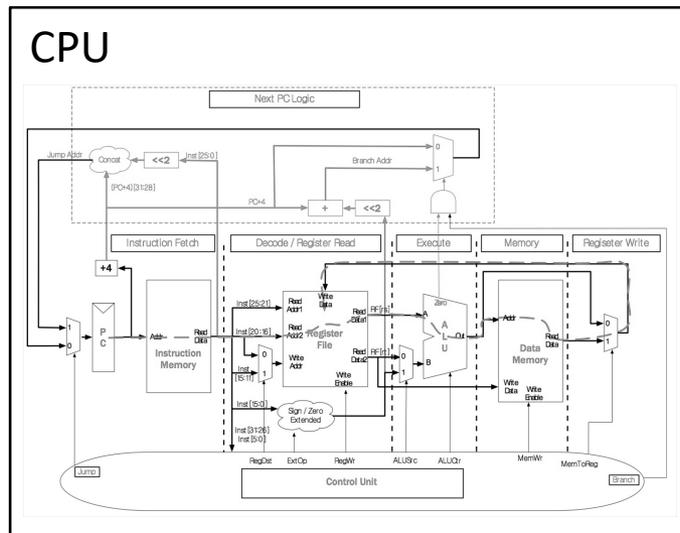
CPU Interrupt Table	
SPI0	handler
...	...

# Agenda

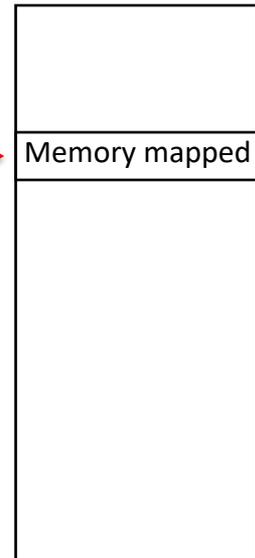
- Devices and I/O
- **OS Boot Sequence and Operation**
- Multiprogramming/time-sharing
- Introduction to Virtual Memory

# 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)



PC = 0x2000 (some default value)



Address Space



```
0x2000:  
addi t0, zero, 0x1000  
lw t0, 4(t0)  
...  
  
(Code to copy firmware into  
regular memory and jump  
into it)
```

- Bootstrapping:

<https://en.wikipedia.org/wiki/Bootstrapping>

# 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)

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

```
Diskette Drive B : None          Serial Port(s) : 3F0 2F0
Pri. Master Disk : LBA,ATA 100, 250GB Parallel Port(s) : 3F0
Pri. Slave Disk : LBA,ATA 100, 250GB DDR at Bank(s) : 0 1 2
Sec. Master Disk : None
Sec. Slave Disk : None

Pri. Master Disk HDD S.M.A.R.T. capability ... Disabled
Pri. Slave Disk HDD S.M.A.R.T. capability ... Disabled

PCI Devices Listing ...
Bus Dev Fun Vendor Device SUID SSID Class Device Class IRQ
-----
0 27 0 8086 2668 1458 A005 0403 Multimedia Device 5
0 29 0 8086 2658 1458 2658 0C03 USB 1.1 Host Contrlr 5
0 29 1 8086 2659 1458 2659 0C03 USB 1.1 Host Contrlr 5
0 29 2 8086 265A 1458 265A 0C03 USB 1.1 Host Contrlr 5
0 29 3 8086 265B 1458 265B 0C03 USB 1.1 Host Contrlr 5
0 29 7 8086 265C 1458 265C 0C03 USB 1.1 Host Contrlr 5
0 31 2 8086 2651 1458 2651 0101 IDE Contrlr 11
0 31 3 8086 266A 1458 266A 0C05 SMBus Contrlr 11
1 0 0 1002 0421 100E 0479 0300 Display Contrlr 5
2 0 0 1283 8212 0090 0000 0180 Mass Storage Contrlr 10
2 5 0 11AB 4320 1458 E000 0200 Network Contrlr 12
2 5 0 11AB 4320 1458 E000 0200 Network Contrlr 9
ACPI Controller
```

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

```
QUESTION 3:
conv: <speedup> x
relu: <speedup> x
pool: <speedup> x
fc: <speedup> x
softmax: <speedup> x
Which layer should we opti
<which layer>
c-ti@hive22 Linux x86_64
~/src/proj3/proj3_sta
answers.txt cnn cnn
cnn.py data LICENSE Makefile test web

c-ti@hive22 Linux x86_64
~/src/proj3/proj3_starte
cnn.c main.c python.c util.c

c-ti@hive22 Linux x86_64
~/src/proj3/proj3_starte
make: 'cnn' is up to date.
```

```
Ubuntu 8.04, kernel 2.6.24-16-generic
Ubuntu 8.04, kernel 2.6.24-16-generic (recovery mode)
Ubuntu 8.04, hometest8*
```

Use the ↑ and ↓ keys to select which entry is highlighted. Press enter to boot the selected OS, 'e' to edit the commands before booting, or 'c' for a command-line.

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

```
Welcome to the KNOPPIX live GNU/Linux on DVD!

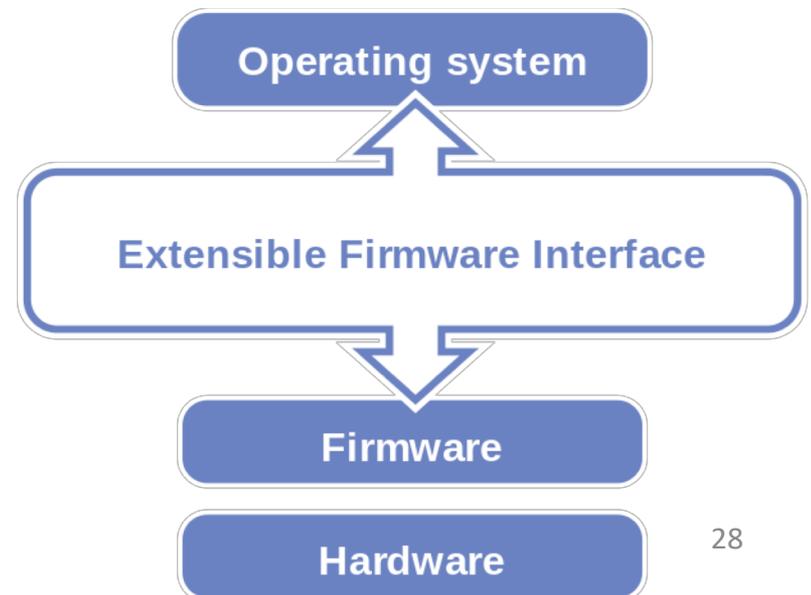
Loading Linux Kernel 2.6.24.4.
Memory available: 124132KB. Memory free: 110100KB.
for USB/Firewire devices... Done.
DMA acceleration for: hdc [QEMU CD-ROM]
Loading KNOPPIX DVD at /dev/hdc...
Found primary KNOPPIX compressed image at /cdrom/KNOPPIX/KNOPPIX.
Found additional KNOPPIX compressed image at /cdrom/KNOPPIX/KNOPPIX2.
Creating /randisk (dynamic size=39304k) on shared memory...Done.
Creating unified filesystem and symlinks on /randisk...Done.
-> Read-only DVD system successfully merged with read-write /randisk.
Done.
Starting INIT (process 1).
INIT: version 2.85 booting
Configuring for Linux Kernel 2.6.24.4.
Processor 0 is Pentium III (Klamath) 1662MHz, 128 KB Cache
apmd(1680): apmd 2.2.1 interfacing with apm driver 1.16ac and APM BIOS 1.2
APM Bios found, power management functions enabled.
USB found, managed by udev
firewire found, managed by udev
Using ide0 hot-plug hardware detection... Started.
Detecting and configuring devices...
```

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

# UEFI

## Unified Extensible Firmware Interface

- Successor of BIOS
- Much more powerful and complex
- E.g. graphics menu; networking; browsers
- All modern Intel & AMD based computer use UEFI



# Intel: ModernFW, Open Source UEFI

- <https://github.com/intel/ModernFW>
- Announced a few days ago at the Intel Open Source Technology Summit (OSTS)
- First hardware systems expected to boot in one year
- Benefits:
  - Vendors & users could verify ALL code of the system -> security
  - In the end, more high quality UEFI implementations

# Launching Applications

- Applications are called “processes” in most OSs.
- Created by another process calling into an OS routine (using a “syscall”, more details later).
  - 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), prepare stack and heap.
- Set `argc` and `argv`, jump into the main function.

# Supervisor Mode

- If something goes wrong in an application, it could crash the entire machine. And what about malware, etc.?
- The OS may need to enforce resource constraints to applications (e.g., access to devices).
- To help protect the OS from the application, CPUs have a **supervisor mode** 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.

# Syscalls

- What if we want to call into an OS routine? (e.g., to read a file, launch a new process, send data, etc.)
  - Need to perform a **syscall**: set up function arguments in registers, and then raise **software interrupt**
  - OS will perform the operation and return to user mode
- Also, OS uses interrupts for scheduling process execution:
  - OS sets scheduler timer interrupt then drops to user mode and start executing a user task, when interrupts triggers, switch into supervisor mode, select next task to execute (& set timer) and drop back to user mode.
- This way, the OS can mediate access to all resources, including devices and the CPU itself.

# Agenda

- Devices and I/O
- OS Boot Sequence and Operation
- **Multiprogramming/time-sharing**
- Introduction to Virtual Memory

# 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. 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**.

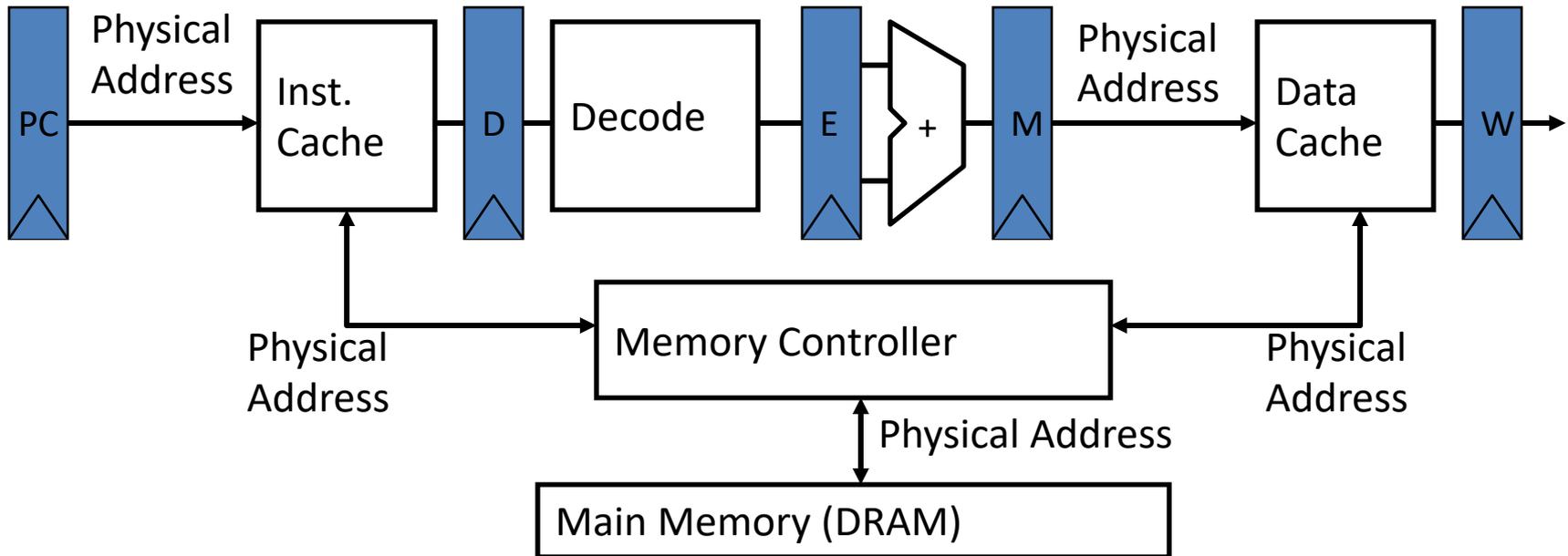
# Protection, Translation, Paging

- Supervisor mode does not fully isolate applications from each other or from the OS.
  - Application could overwrite another application's memory.
  - 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.

# Agenda

- Devices and I/O
- OS Boot Sequence and Operation
- Multiprogramming/time-sharing
- **Introduction to Virtual Memory**

# “Bare” 5-Stage Pipeline



- In a bare machine, the only kind of address is a physical address

# 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).

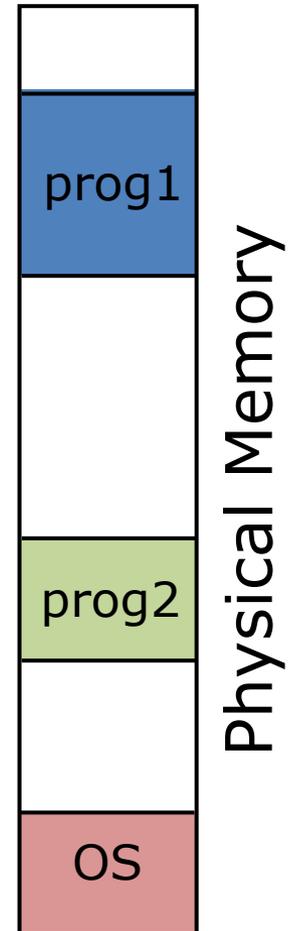
## Location-independent programs

Programming and storage management ease  
=> *base register* – *add offset to each address*

## Protection

Independent programs should not affect each other inadvertently  
=> *bound register* – *check range of access*

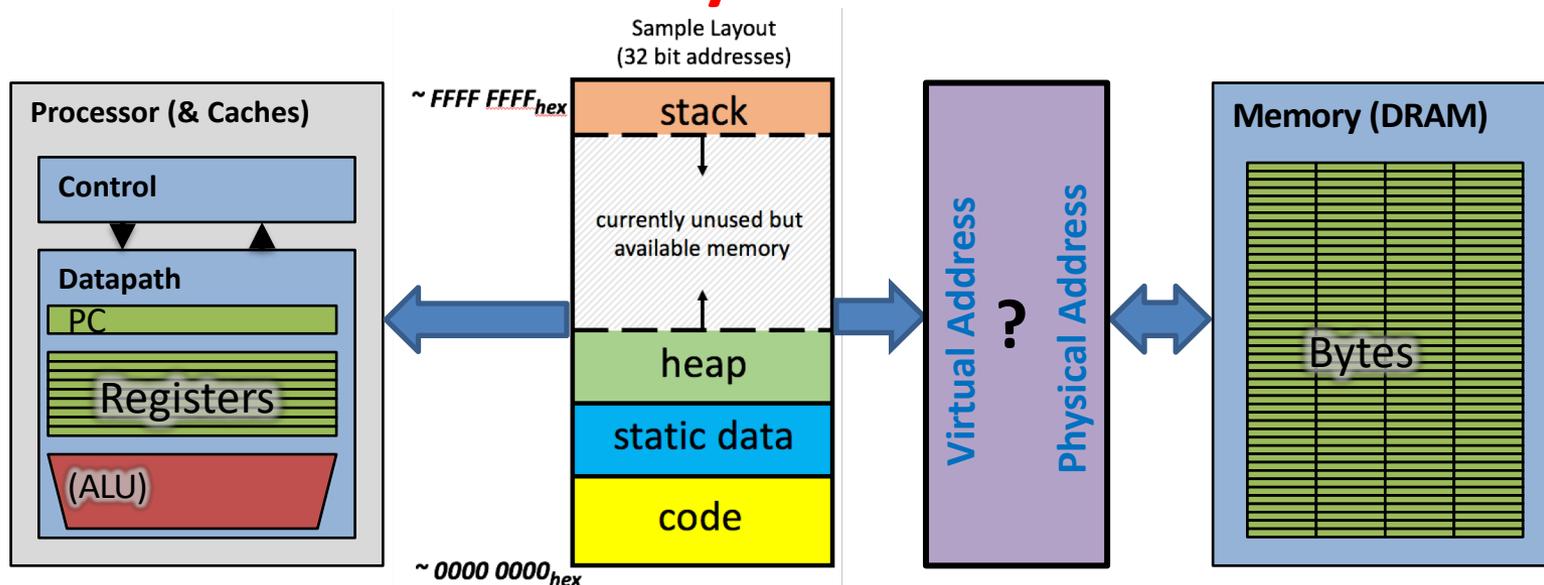
(Note: Multiprogramming drives requirement for resident *supervisor (OS)* software to manage context switches between multiple programs)



# Virtual Memory

- Next level in the memory hierarchy:
  - Provides program with illusion of a very large main memory:
  - Working set of “pages” reside in main memory - others reside on disk.
- Also allows OS to share memory, protect programs from each other
- Today, more important for **protection** than just another level of memory hierarchy
- Each process thinks it has all the memory to itself
- (Historically, it predates caches)

# Virtual vs. Physical Addresses

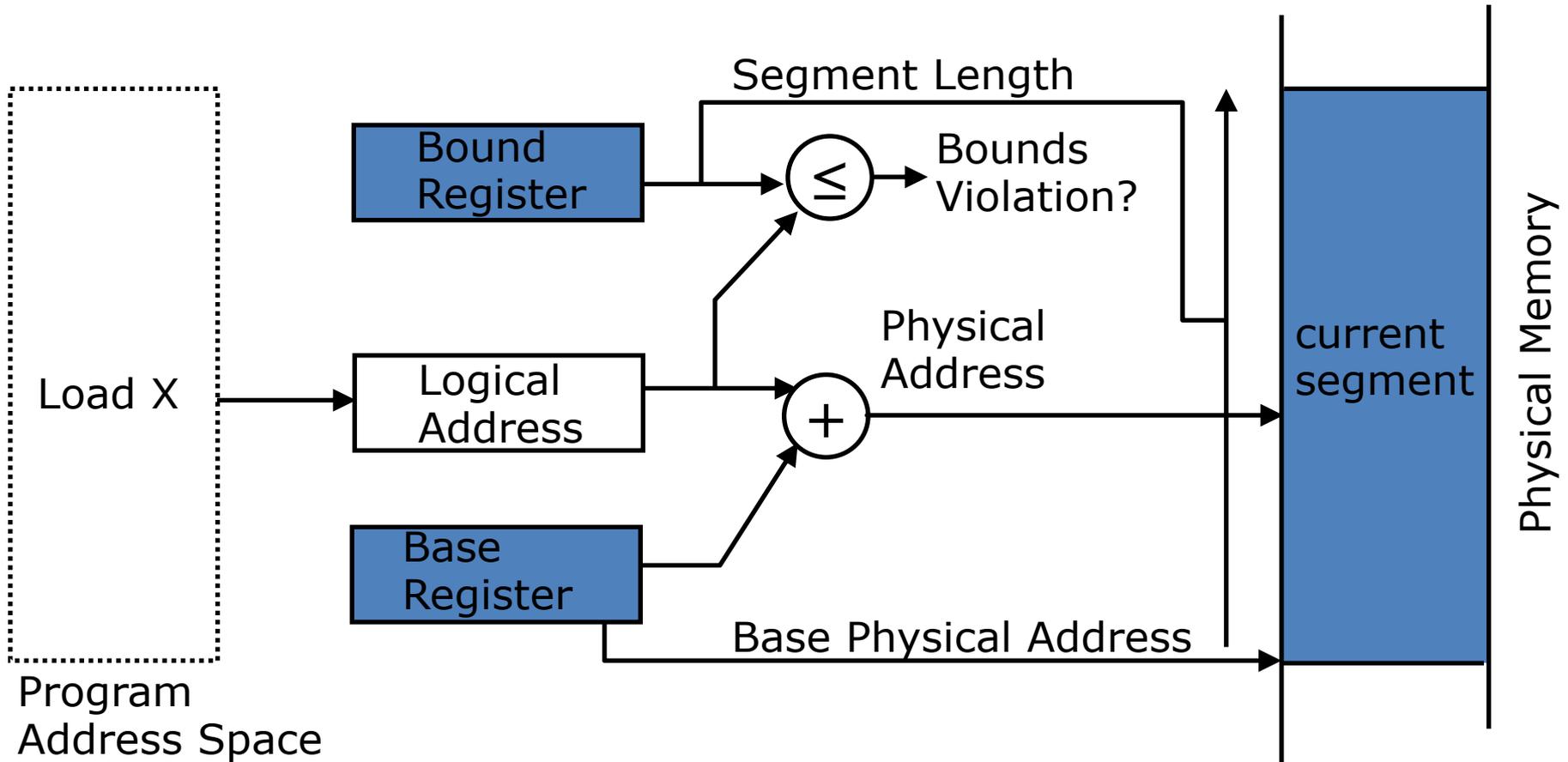


Many of these (software & hardware cores)

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***

# Simple Base and Bound Translation

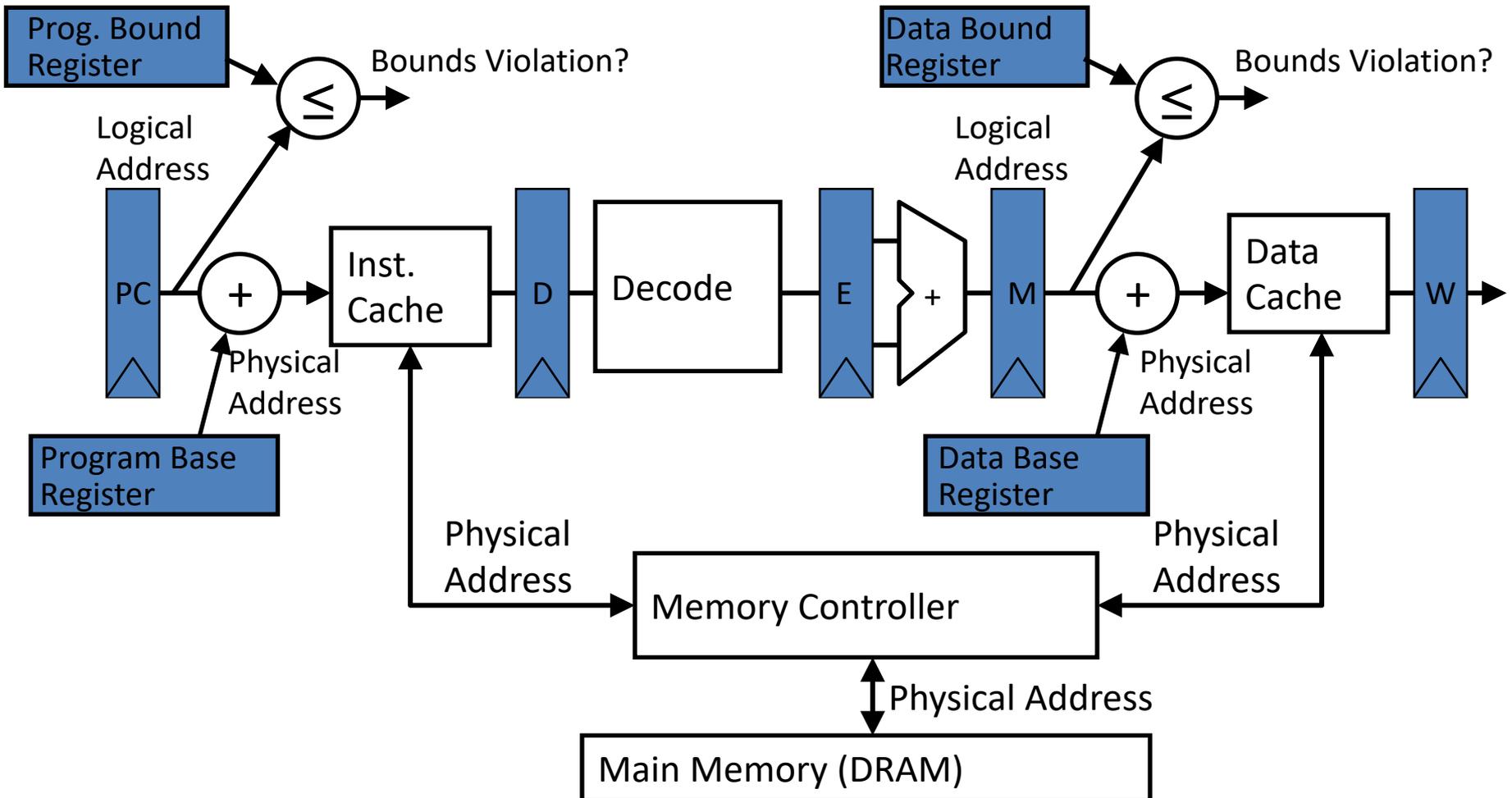


Base and bounds registers are visible/accessible only when processor is running in *supervisor mode*

# Address Spaces

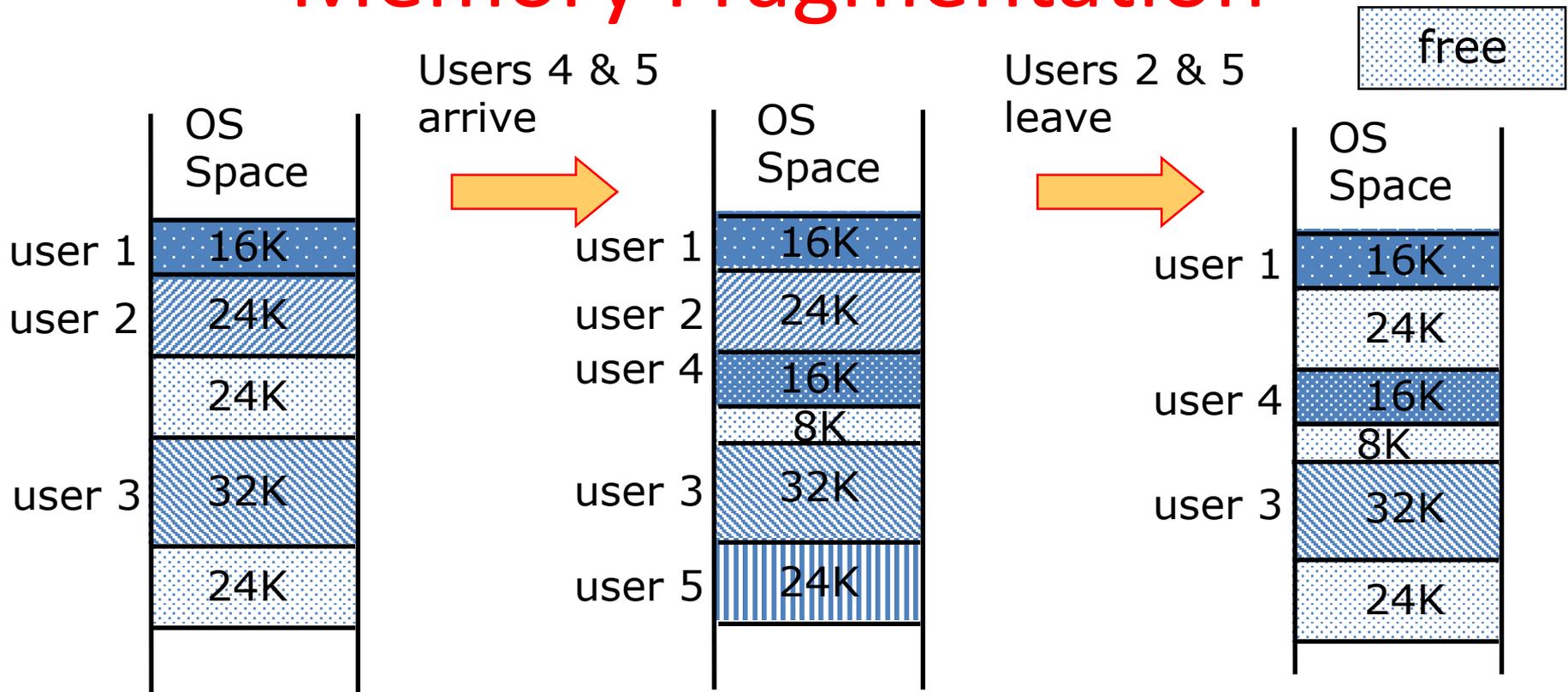
- Address space = set of addresses for all available memory locations
- Now, two kinds of memory addresses:
  - **Virtual Address Space**
    - Set of addresses that the user program knows about
  - **Physical Address Space**
    - Set of addresses that map to actual physical locations in memory
    - Hidden from user applications
- Memory manager maps between these two address spaces

# Base and Bound Machine



[ Can fold addition of base register into (register+immediate) address calculation using a carry-save adder (sums three numbers with only a few gate delays more than adding two numbers) ]

# Memory Fragmentation



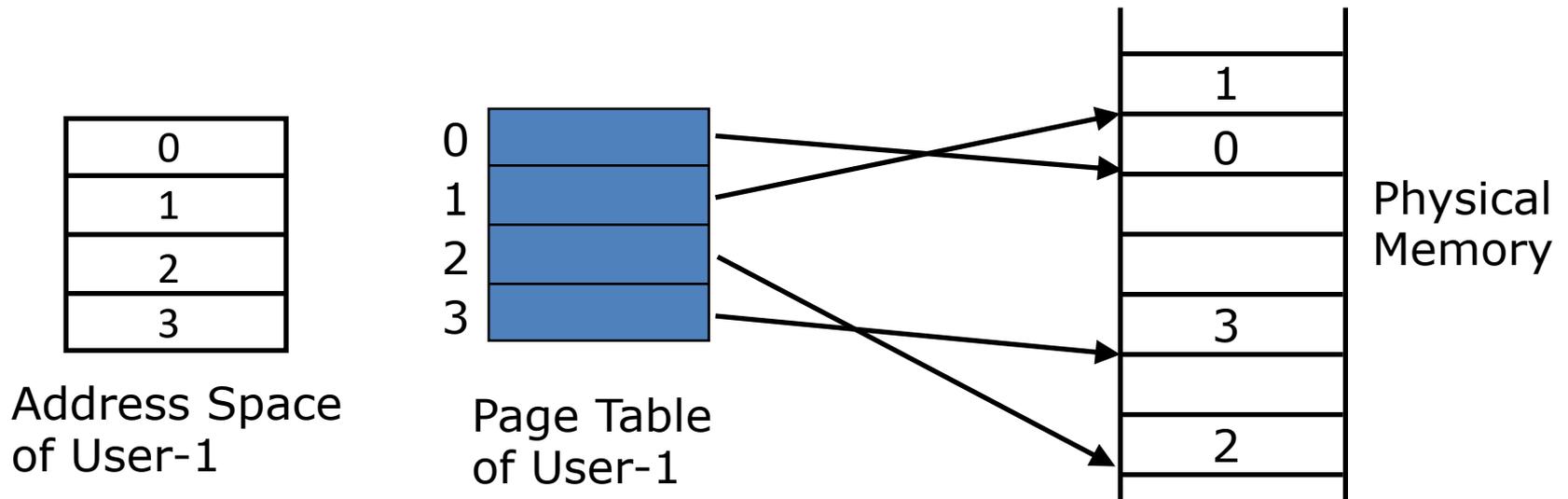
As users come and go, the storage is “fragmented”. Therefore, at some stage programs have to be moved around to compact the storage.

# Paged Memory Systems

- Processor-generated address can be split into:

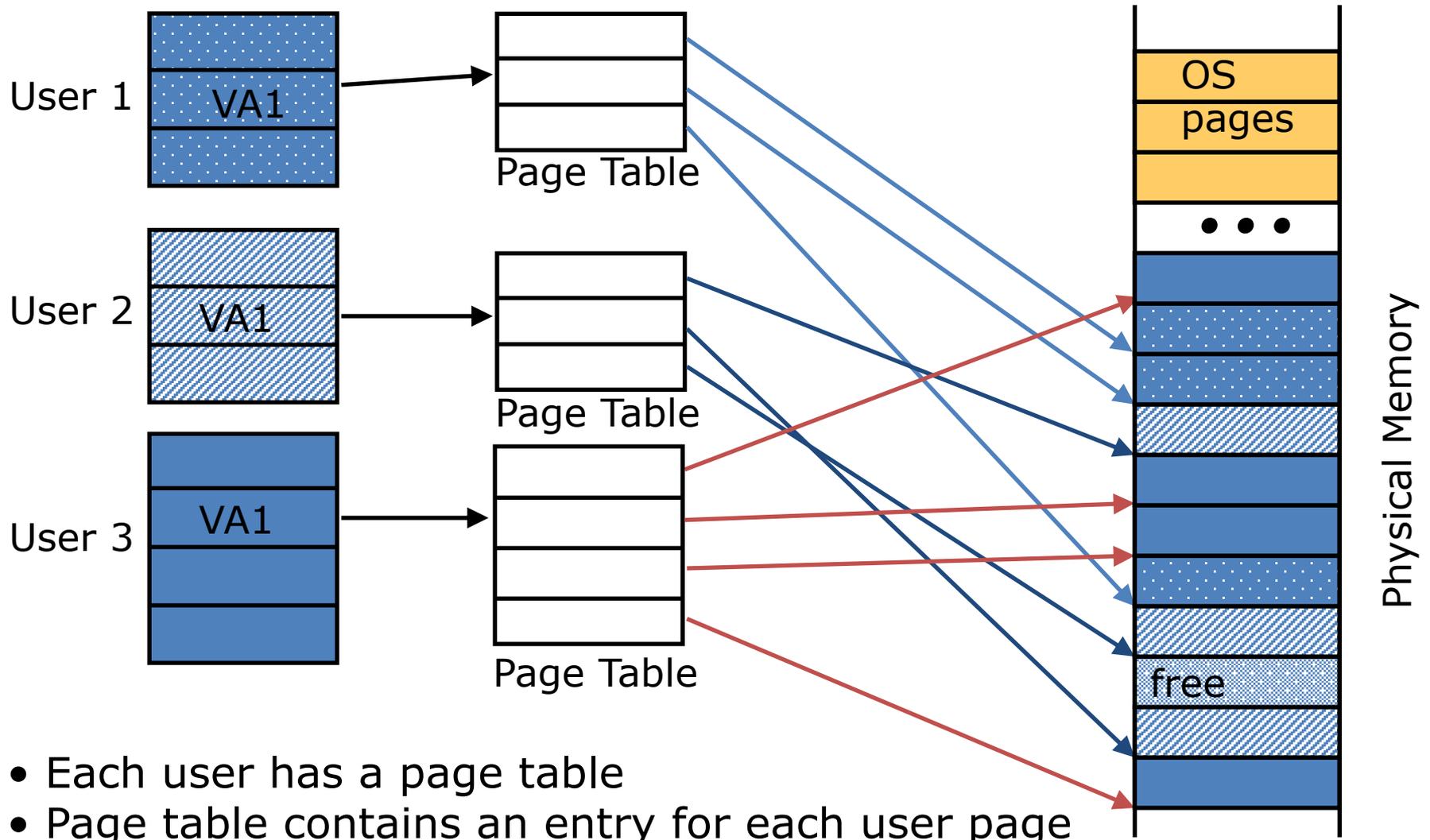


- A page table contains the physical address of the base of each page



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

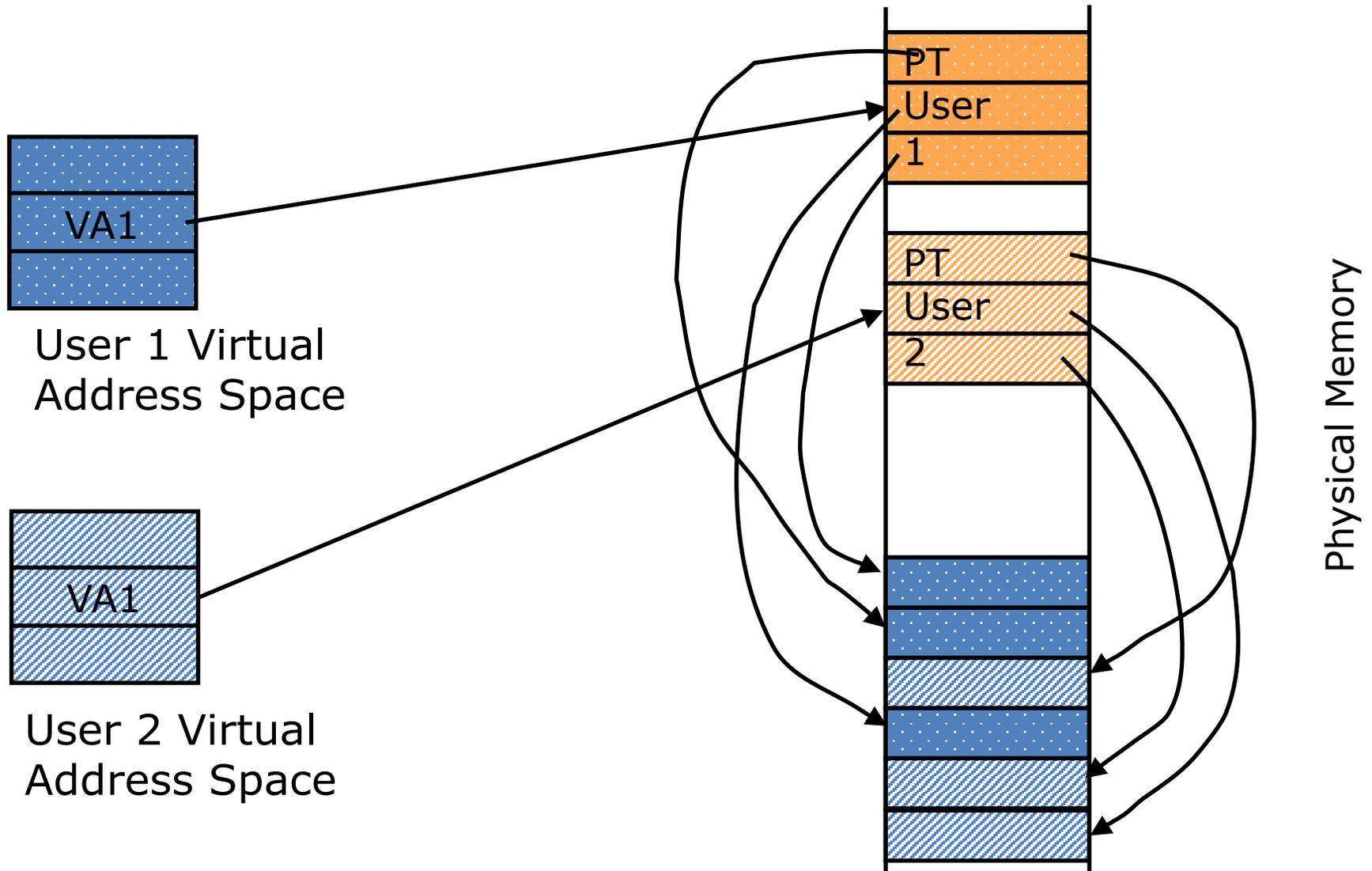
# Private Address Space per User



# 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!*

# Page Tables in Physical Memory



Physical Memory

# In Conclusion

- Once we have a basic machine, it's mostly up to the OS to use it and define application interfaces.
- Hardware helps by providing the right abstractions and features (e.g., Virtual Memory, I/O).