

CS 61C: Great Ideas in Computer Architecture

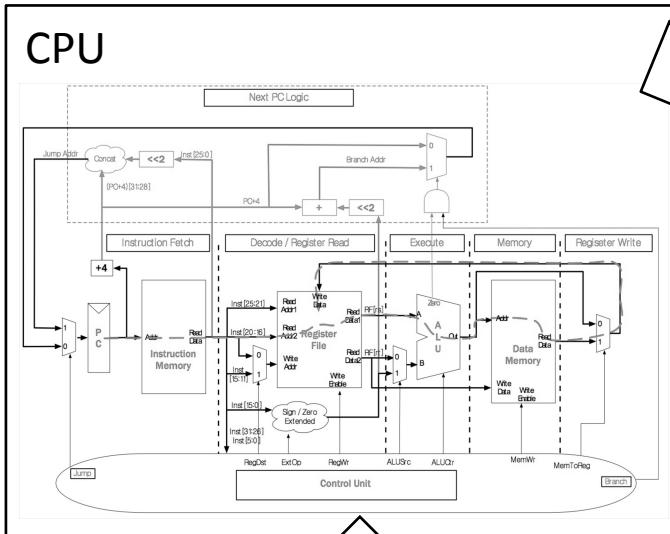
Lecture 22: *Operating Systems, Interrupts, and Virtual Memory Part 1*

Instructor: Sagar Karandikar
sagark@eecs.berkeley.edu

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

CS61C so far...

Project 2



MIPS Assembly

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

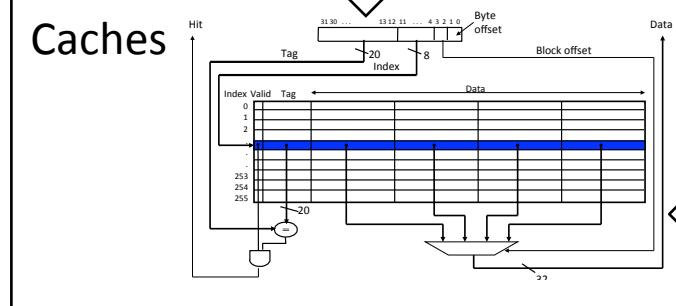
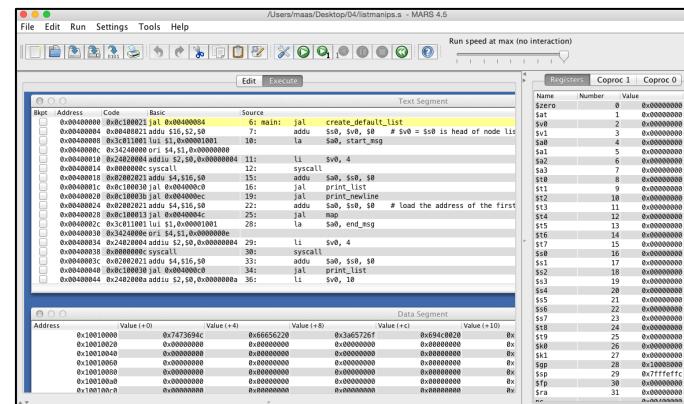
C Programs

```
#include <stdlib.h>

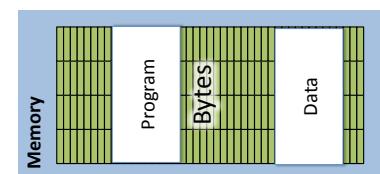
int fib(int n) {
    return
        fib(n-1) +
        fib(n-2);
}
```

Project 1

Labs



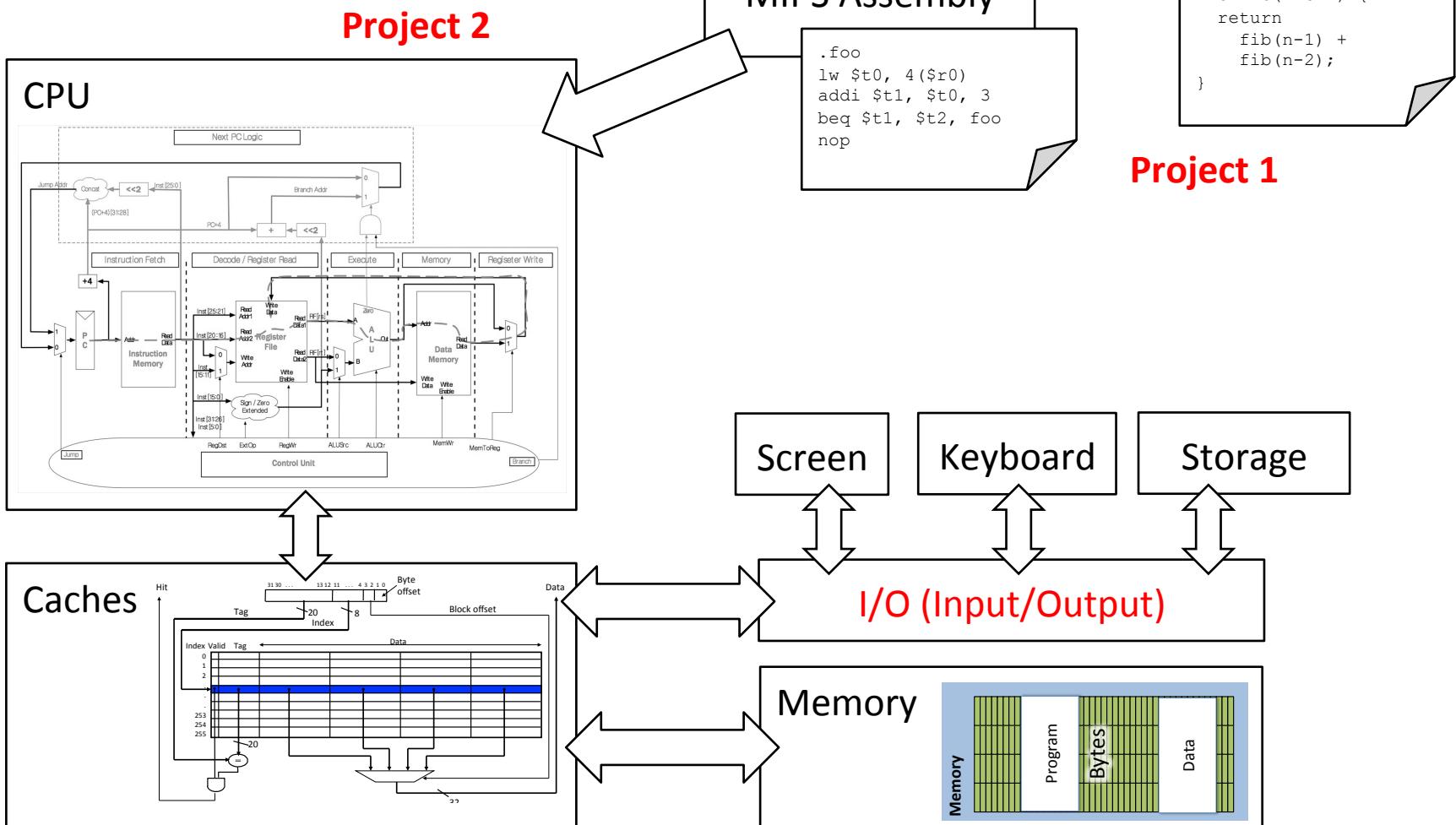
Memory



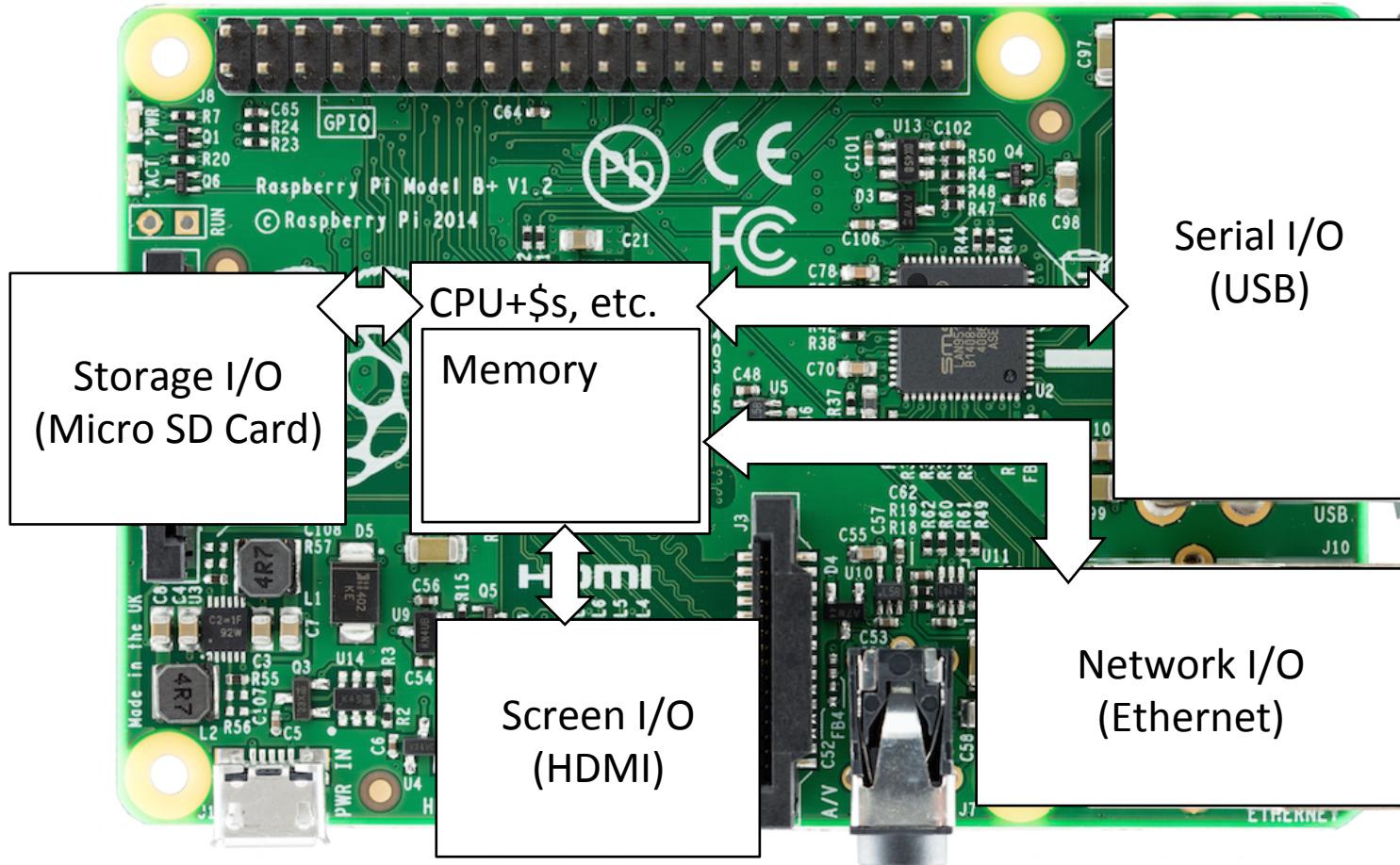
So how is this any different?



Adding I/O



Raspberry Pi (\$40 on Amazon)

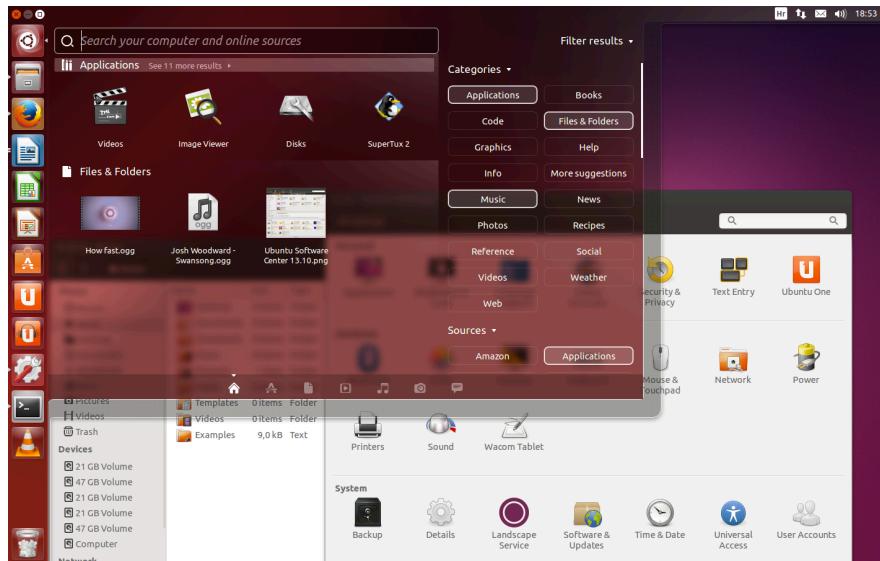


It's a real computer!



But wait...

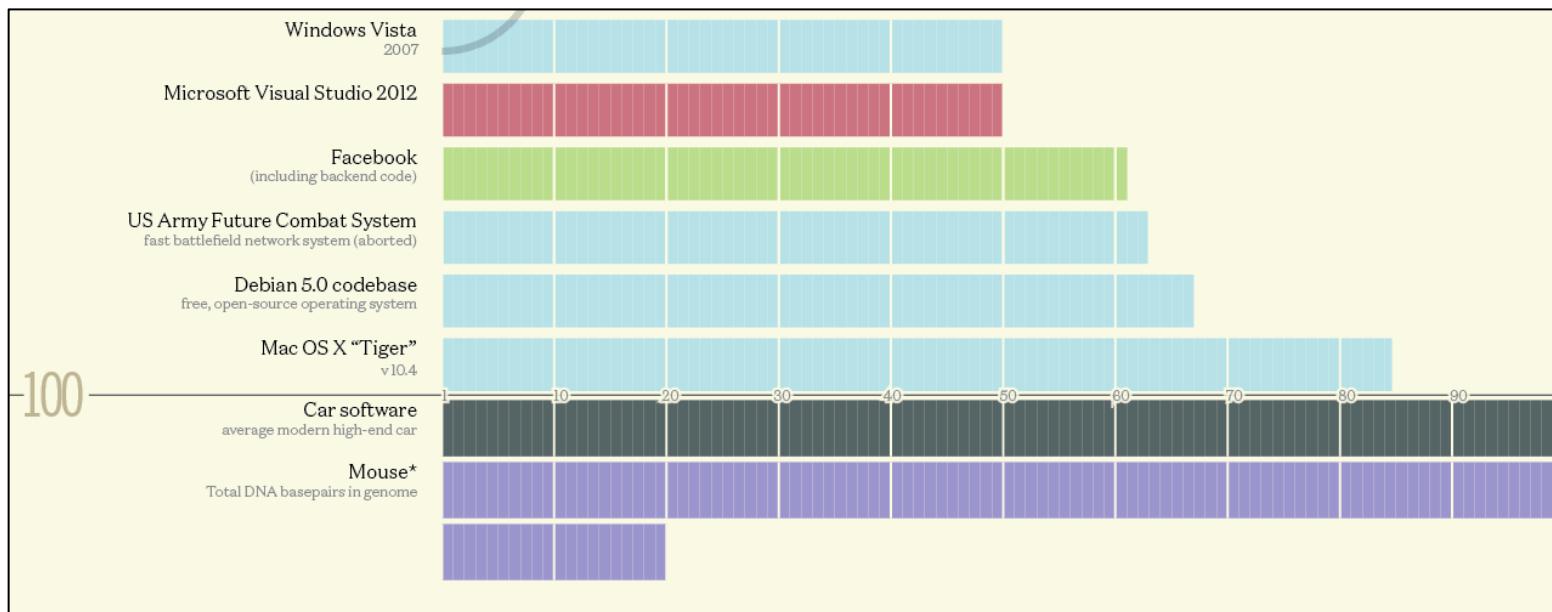
- That's not the same! When we run MARS, it only executes one program and then stops.
- When I switch on my computer, I get this:



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/>

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

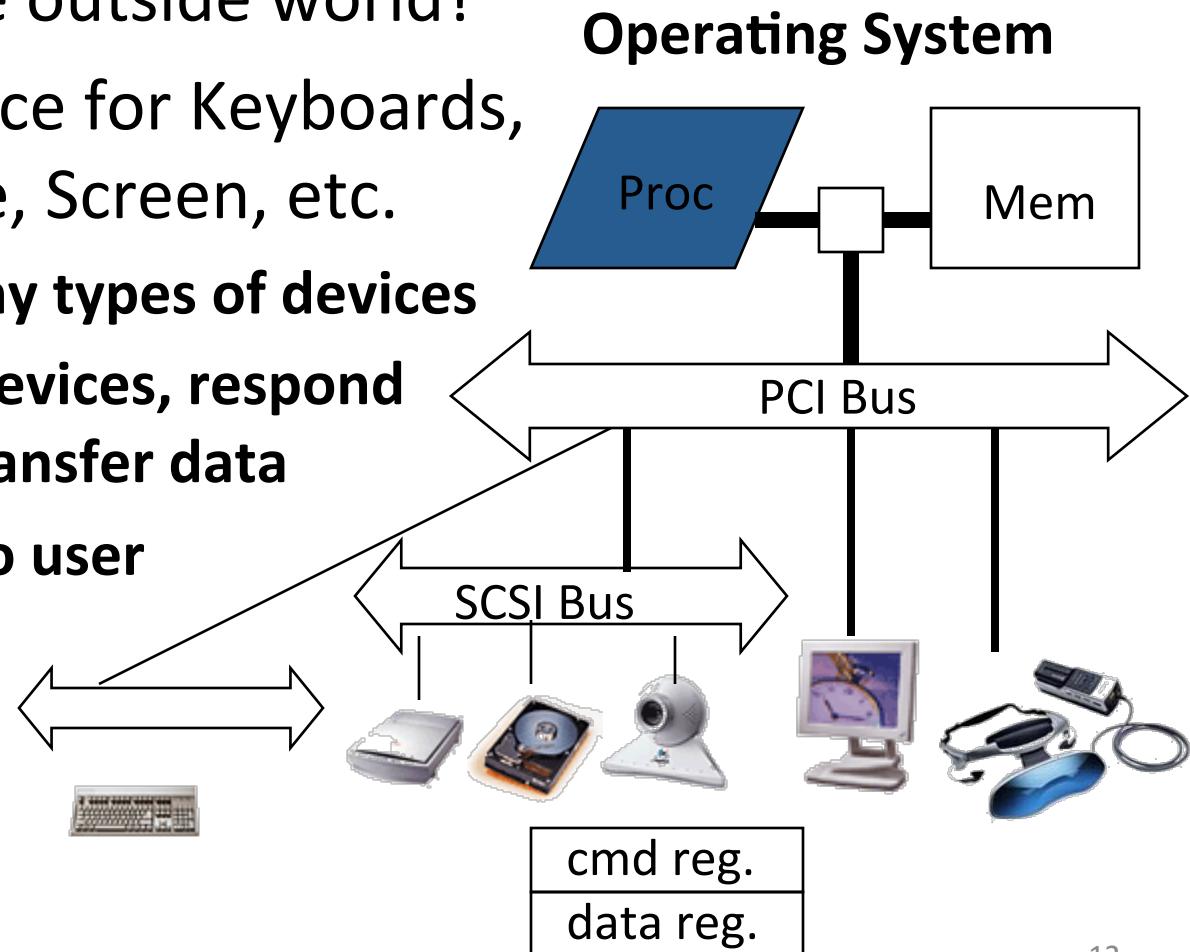
- Devices and I/O
- OS Boot Sequence and Operation
- Multiprogramming/time-sharing
- Introduction to Virtual Memory

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

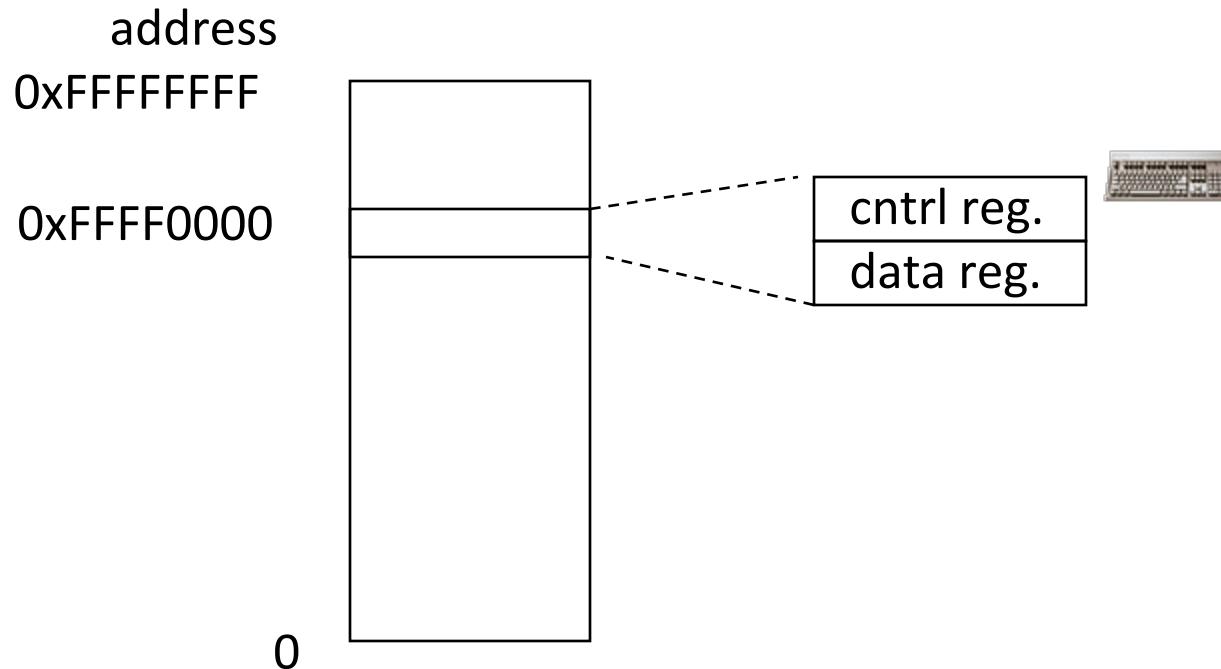


Instruction Set Architecture for I/O

- 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 ($0 \Rightarrow 1$) 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 ($1 \Rightarrow 0$) of Control Register
- This is called “**Polling**”

I/O Example (polling)

- Input: Read from keyboard into \$v0

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

- Output: Write to display from \$a0

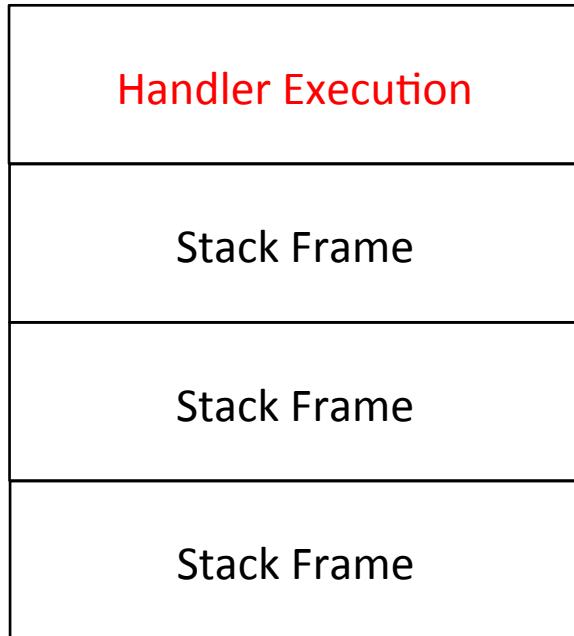
```
lui      $t0, 0xffff #ffff0000  
Waitloop:    lw       $t1, 8($t0) #control  
                  andi    $t1,$t1,0x1  
                  beq     $t1,$zero, Waitloop  
                  sw       $a0, 12($t0) #data
```

“Ready” bit is from processor’s point of view!

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 **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: lui $t0, 0xffff  
         lw  $t1, 0($t0)  
         andi $t1,$t1,0x1  
         lw  $v0, 4($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,Label
```

Interrupt(SPI0)

CPU Interrupt Table

SPI0	handler
...	...

Administrivia

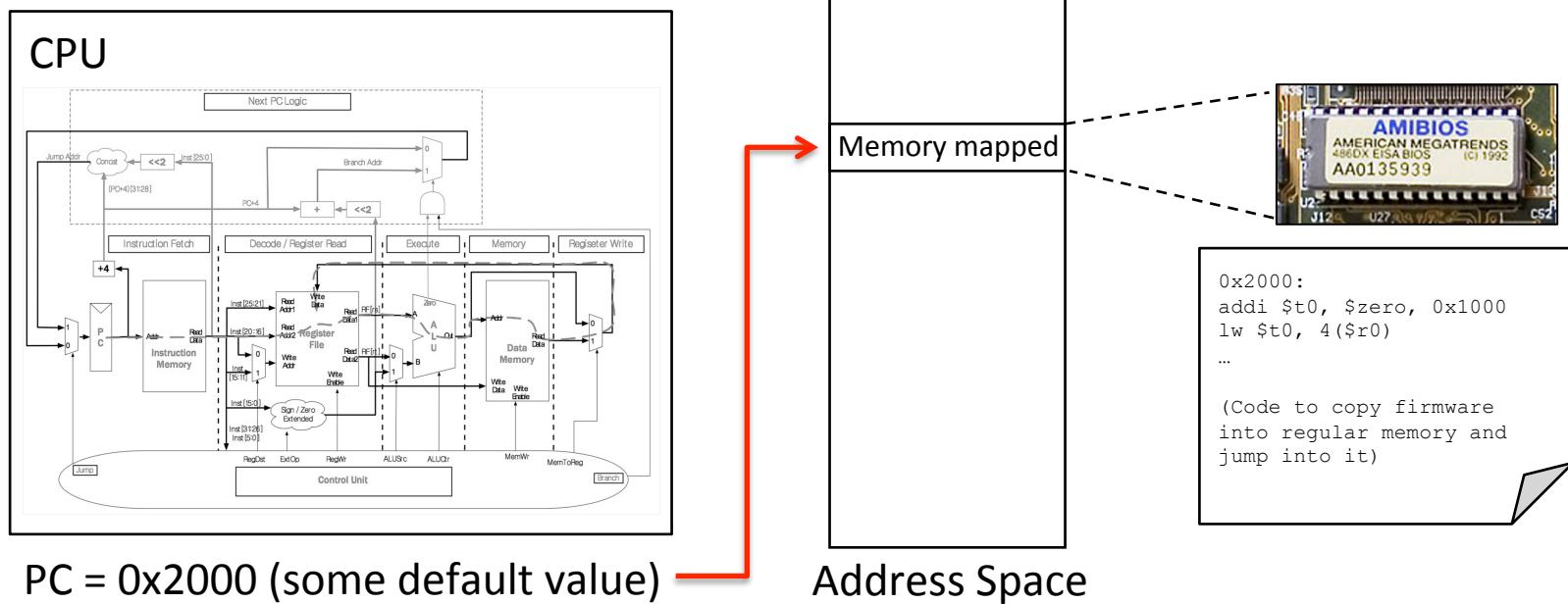
- Project 3-2 Out
- HW5 Out – Performance Programming
- Guerrilla Section on Performance Programming on Thursday, 5-7pm, Woz

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



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)

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

Diskette Drive B : None Serial Port(s) : 3F0 ZFO										
Pri. Master Disk : IDE ATA 100, 250GB Parallel Port(s) : 2F0										
Pri. Slave Disk : IDE ATA 100, 250GB DMA at Bank(s) : 0 1 2										
Sec. Master Disk : None										
Sec. Slave Disk : None										
PCI Dev Fun Vendor Device SUBS SSID Class Device Class IRQ										
0	27	0	8006	2666	1450	0005	0003	Multimedia Device		5
0	29	0	8006	2650	1450	2658	0003	USB 1.1 Host Contrlr		
0	29	1	8006	2659	1450	2659	0003	USB 1.1 Host Contrlr		
0	29	2	8006	2658	1450	2658	0003	USB 1.1 Host Contrlr		
0	29	3	8006	2659	1450	2659	0003	USB 1.1 Host Contrlr		
0	29	7	8006	2652	1450	2652	0003	USB 1.1 Host Contrlr		
0	31	2	8006	2651	1450	2651	1001	IDE Contrlr		
0	31	3	8006	2660	1450	2660	0005	SCSI Contrlr		11
1	0	0	10DE	0421	10DE	0479	0300	Display Contrlr		5
2	0	0	12B3	0212	0000	0000	0180	Mass Storage Contrlr		10
2	5	0	11AB	4320	1450	E000	0200	Network Contrlr		12
								ACPI Controller		9

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

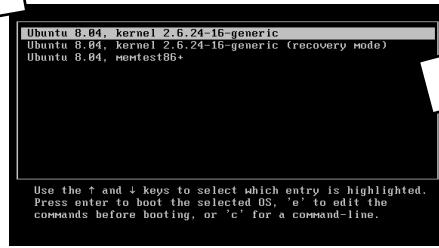
The diagram shows a circular flow of data. It starts with a red-bordered box containing a terminal window with BIOS output. An arrow points from this box to a white-bordered box containing a terminal window with bootloader (proj3_starter) output. Another arrow points from this box to a black-bordered box containing a terminal window with OS kernel (Ubuntu 8.04) output. Finally, an arrow points back from the black box to the red box, completing the cycle.

```
QUESTION 3:
copy: <speedup> x
relu: <speedup> x
pool: <speedup> x
fc: <speedup> x
softmax: <speedup> x

Which layer should we optimize?
which layers>
c-ti@hive22:~$ cd /src/proj3/proj3_starter
c-ti@hive22:~/src/proj3/proj3_starter$ ls
answers.txt cnn cnn.py data LICENSE Makefile test web
cnn.c main.c python.c util.c

c-ti@hive22:~/src/proj3/proj3_starter$ cd /src/proj3/proj3_starter
c-ti@hive22:~/src/proj3/proj3_starter$ make cnn
make: 'cnn' is up to date.

c-ti@hive22:~/src/proj3/proj3_starter$ cd /src/proj3/proj3_starter
c-ti@hive22:~/src/proj3/proj3_starter$
```



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

The diagram shows a white-bordered box containing a terminal window with a penguin icon and the text "Welcome to the KNOPPIX live GNU/Linux on DVD!". An arrow points from this box to a black-bordered box containing a terminal window with a long list of system boot logs. The logs include kernel initialization, module loading, and various system services starting up.

```
Welcome to the KNOPPIX live GNU/Linux on DVD!
Running Linux Kernel 2.6.24.4
Memory available: 124132KB, Memory free: 118180KB.
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=99394K) on shared memory...Done.
Mounting initial filesystem and splitfs on /randisk...
-> Read-only DVD system successfully merged with read-write /randisk.
Done.
Starting INIT (process 1).
INIT: version 2.66 booting
Configuring for Linux Kernel 2.6.24.4.
Processor 0 is Pentium II (Klamath) 1662MHz, 128 KB Cache
apmd16001.apmd12.1.1 Interfacing with apm driver 1.16ac and APM BIOS 1.2
Processor 0 has no power management functions enabled.
USB Friend, managed by udev
udev found, managed by udev
Starting udev hot-plug hardware detection... Started.
Toconfigure devices...
```

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

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` (see OpenMP threads) 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 can crash the entire machine. What about malware, etc.?
- The OS may need to enforce resource constraints to applications (e.g., access to devices).
- To protect the OS from the application, CPUs have a **supervisor mode** bit (also need isolation, more later).
 - You can only access a subset of instructions and (physical) memory when not in supervisor mode (user mode).
 - You can change out of supervisor mode using a special instruction, but not into it (unless there is an interrupt).

Syscalls

- How to switch back to OS? OS sets timer interrupt, when interrupts trigger, drop into supervisor mode.
- 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
- This way, the OS can mediate access to all resources, including devices, the CPU itself, etc.

Syscalls in MARS

- MARS provides many simple syscalls using the `syscall` MIPS instruction
- How to issue a syscall?
 - Place the syscall number in `$v0`
 - Place arguments to the syscall in the `$a*` registers
 - Issue the `syscall` instruction
- This is how your MIPS code has been able to produce output all along
 - MARS is providing a “shim” layer that acts like an OS for simple tasks

Table of Available Services

Service	Code in \$v0	Arguments	Result
print integer	1	\$a0 = integer to print	
print float	2	\$f12 = float to print	
print double	3	\$f12 = double to print	
print string	4	\$a0 = address of null-terminated string to print	
read integer	5		\$v0 contains integer read
read float	6		\$f0 contains float read
read double	7		\$f0 contains double read
read string	8	\$a0 = address of input buffer \$a1 = maximum number of characters to read	<i>See note below table</i>
sbrk (allocate heap memory)	9	\$a0 = number of bytes to allocate	\$v0 contains address of allocated memory
exit (terminate execution)	10		
print character	11	\$a0 = character to print	<i>See note below table</i>
read character	12		\$v0 contains character read

Example Syscall

- Let's say we want to print an integer stored in \$s3:

Print integer is syscall #1

```
li $v0, 1  
add $a0, $s3, $zero  
syscall
```

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.
 - Remember your Project 1 linker: application assumes that code is in certain location. How to prevent overlaps?
 - 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**

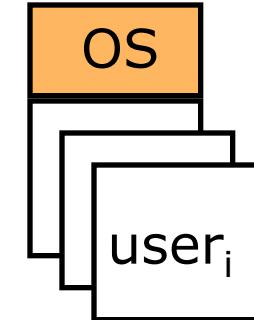
Modern Virtual Memory Systems

Illusion of a large, private, uniform store

Protection

several users, each with their private address space and one or more shared address spaces

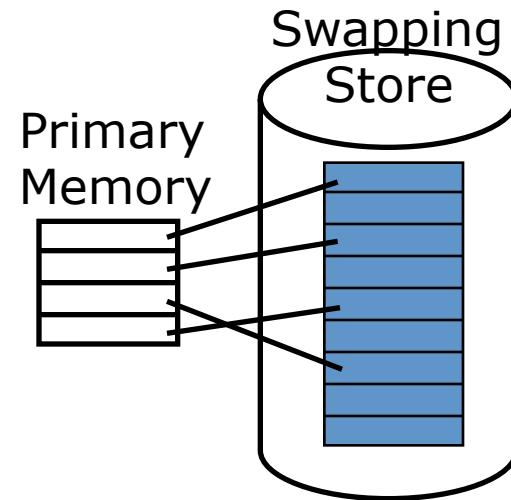
page table \equiv name space



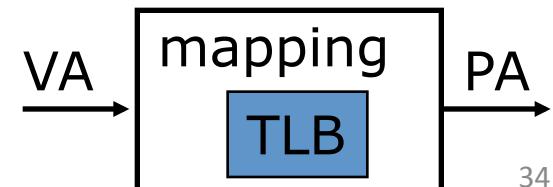
Demand Paging

Provides the ability to run programs larger than the primary memory

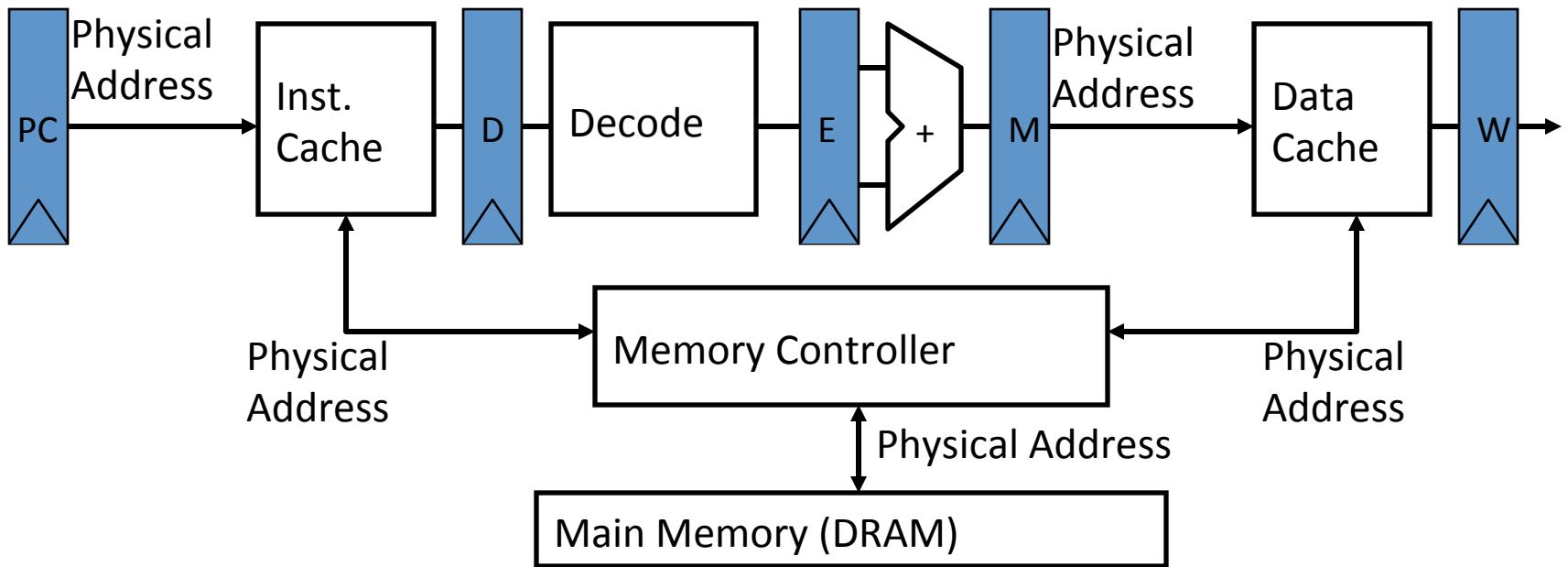
Hides differences in machine configurations



The price is address translation on each memory reference



“Bare” 5-Stage Pipeline



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

Dynamic Address Translation

Motivation

In early machines, I/O operations were slow and each word transferred involved the CPU

Higher throughput if CPU and I/O of 2 or more programs were overlapped.

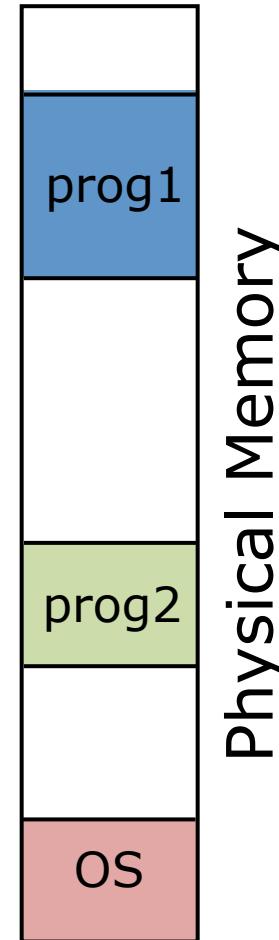
Location-independent programs

Programming and storage management ease
⇒ need for a *base register*

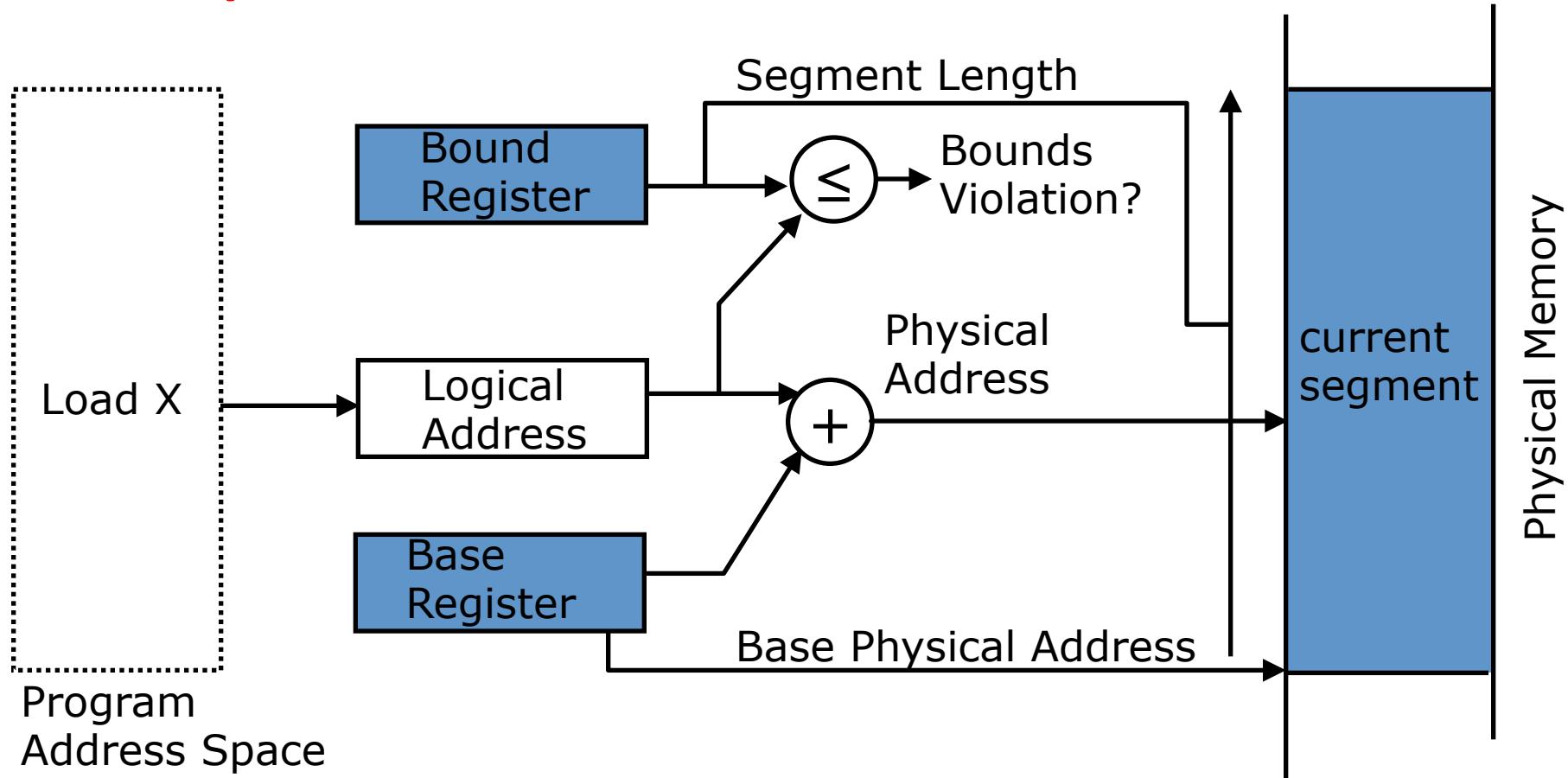
Protection

Independent programs should not affect each other inadvertently
⇒ need for a *bound register*

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



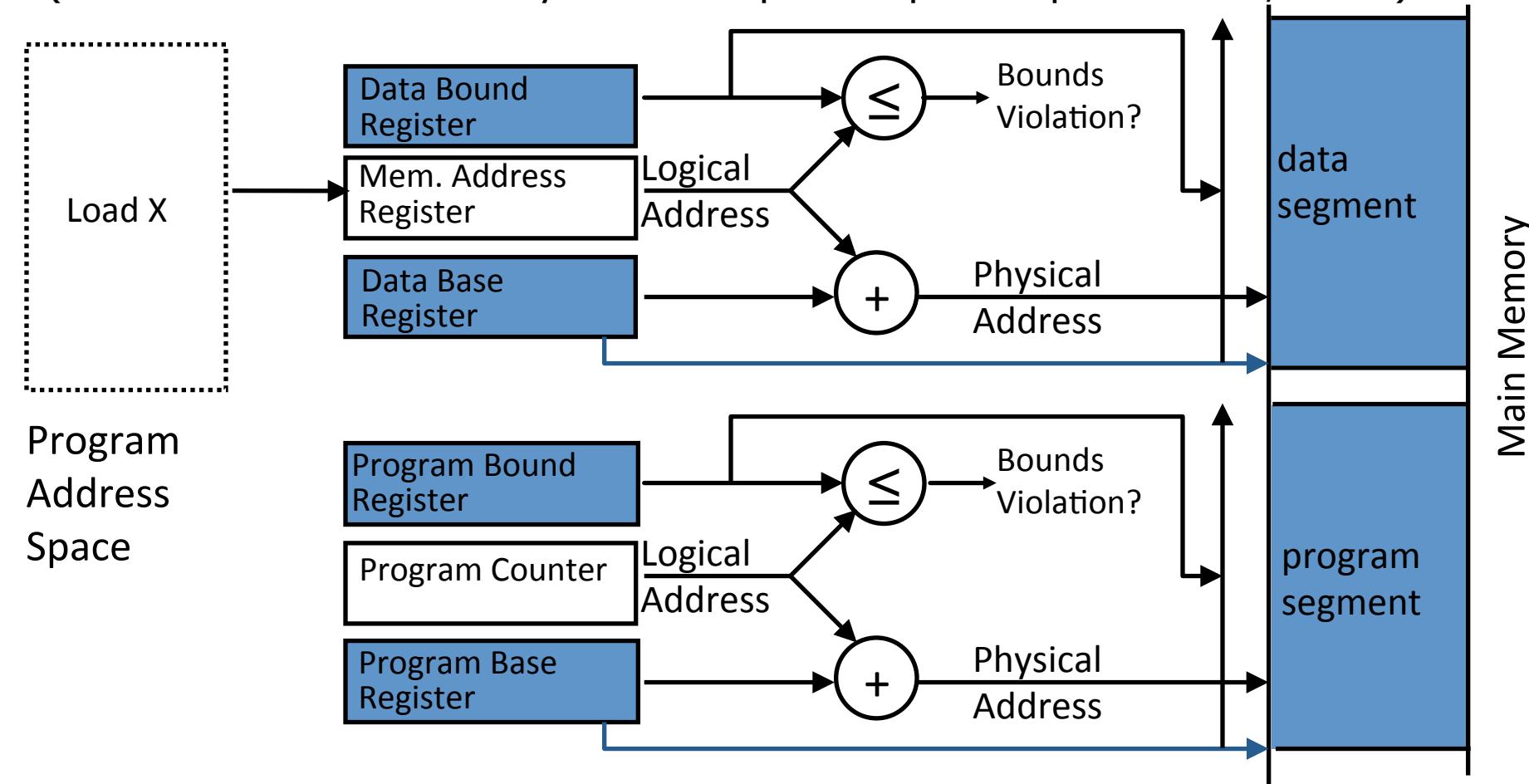
Simple Base and Bound Translation



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

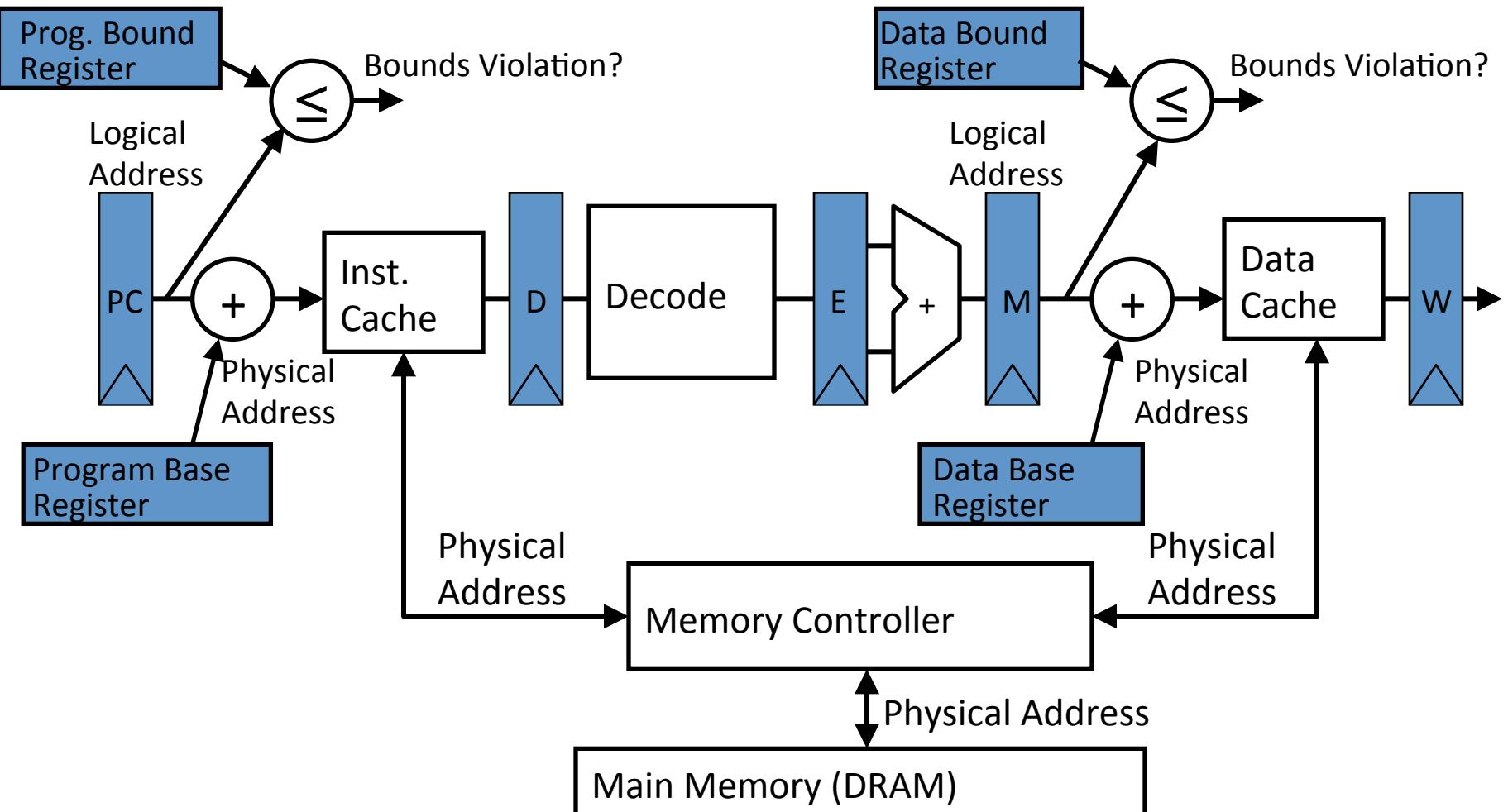
Separate Areas for Program and Data

(Scheme used on all Cray vector supercomputers prior to X1, 2002)



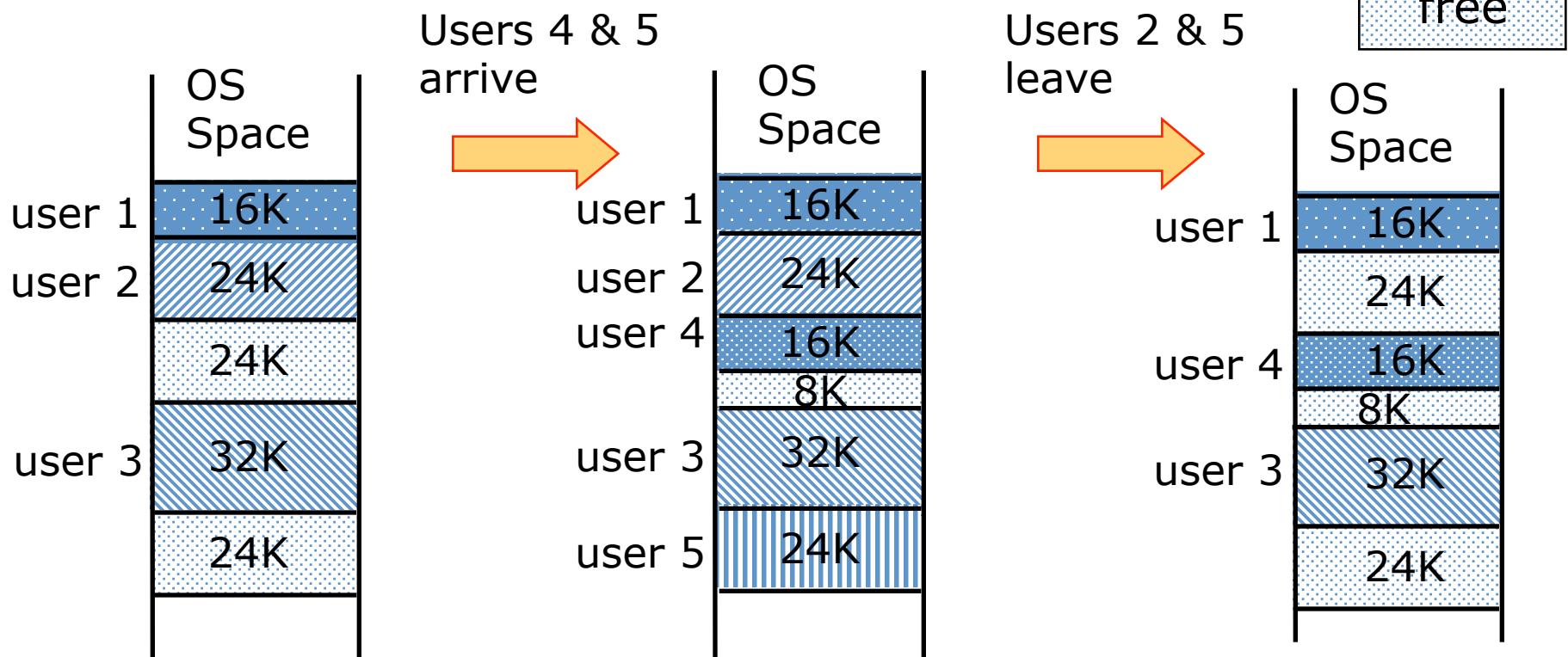
What is an advantage of this separation?

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



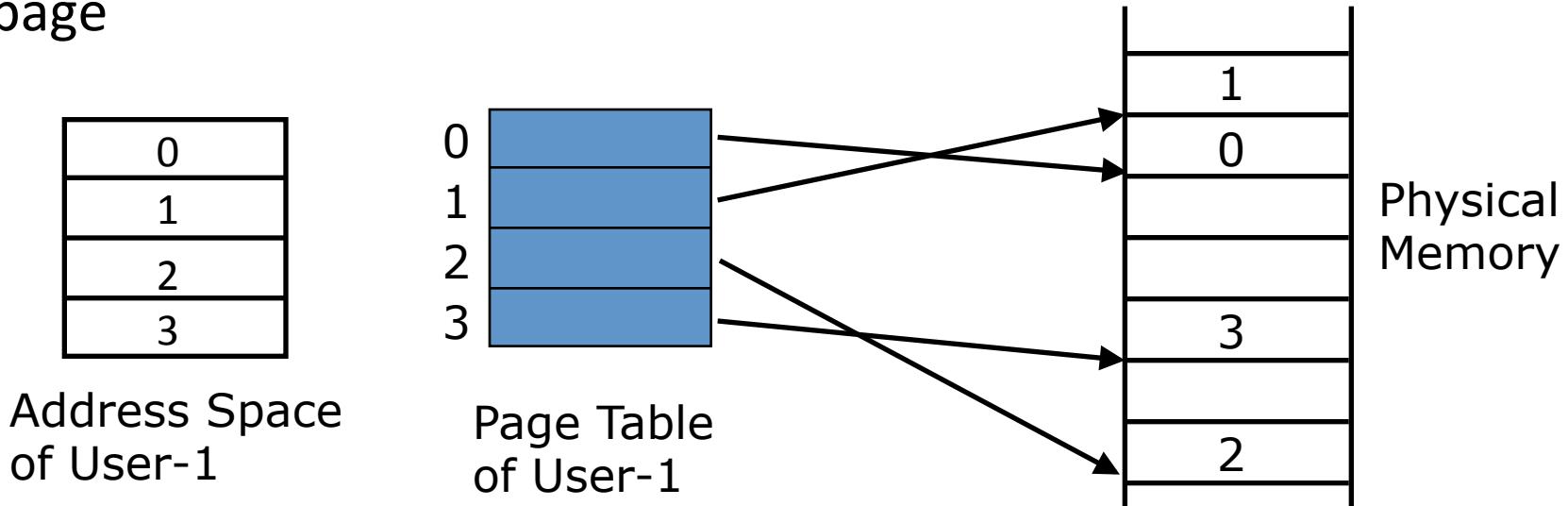
As users come and go, the storage is “fragmented”. Therefore, at some stage programs have to be moved around to compact the storage.

Paged Memory Systems

- Processor-generated address can be split into:

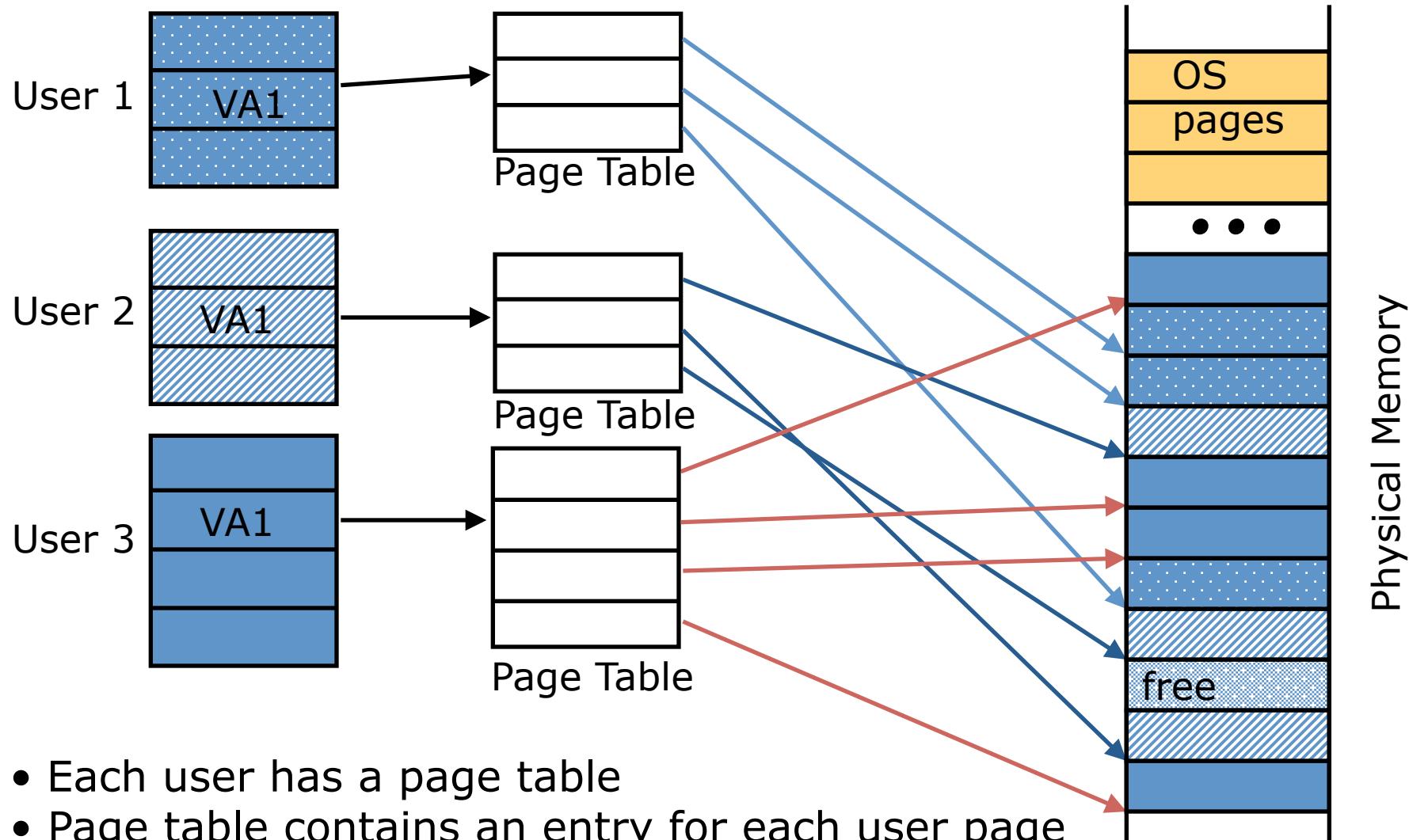


- A page table contains the physical address of the base of each page



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

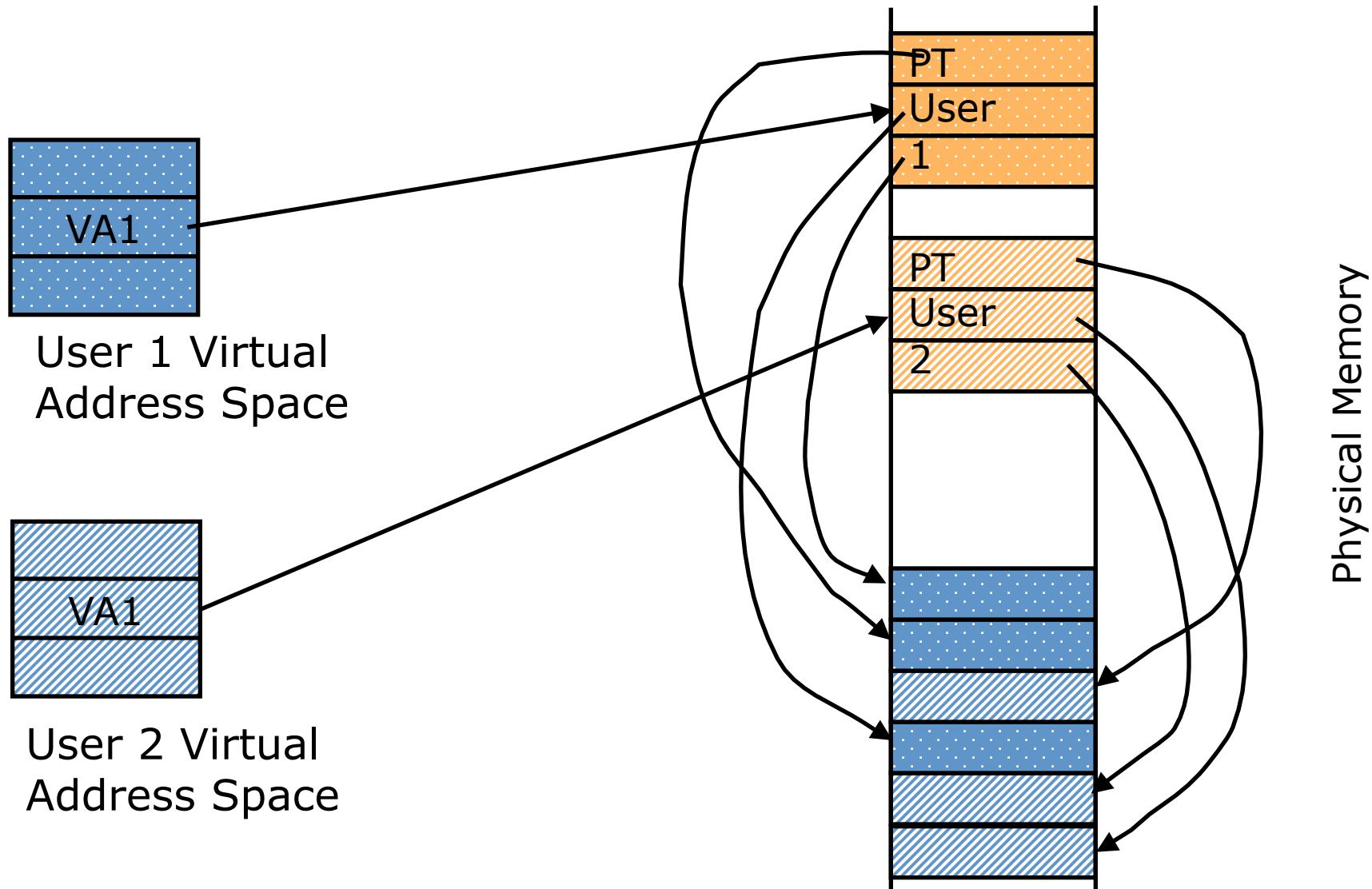
Private Address Space per User



Where Should Page Tables Reside?

- Space required by the page tables (PT) is proportional to the address space, number of users, ...
 \Rightarrow *Too large to keep in registers*
- Idea: Keep PTs in the main memory
 - Needs one reference to retrieve the page base address and another to access the data word
 \Rightarrow *doubles the number of memory references!*

Page Tables in 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).
- If you want to learn more about operating systems, you should take CS162!
- What's next in CS61C?
 - More details on I/O
 - More about Virtual Memory