Virtual Memory

Questions Answered in this Lecture:

- What is an address space? (quick review)
- How do we implement virtual memory? (relocation, base+bounds, segmentation)
- What hardware do we need?

Announcements

- Project 1a due Monday
- We go live Tues. Hermann Hall 100 (auditorium)
- Currently 33 people interested in in-person attendance. Max is 40.
  - You **must** wear a mask
  - I will be passing around an attendance sheet (for contact tracing)
- **Reading**: OSTEP 13, 15, 16 + optionals

A bit of history on timesharing systems

1955-56: MIT TX-0

- Early computer using transitors (which were still very new at the time)
- Developed at MIT's Lincoln Laboratory
- Used vacuum tube-based transistor packaging
- Heavily influences DEC's PDP-1

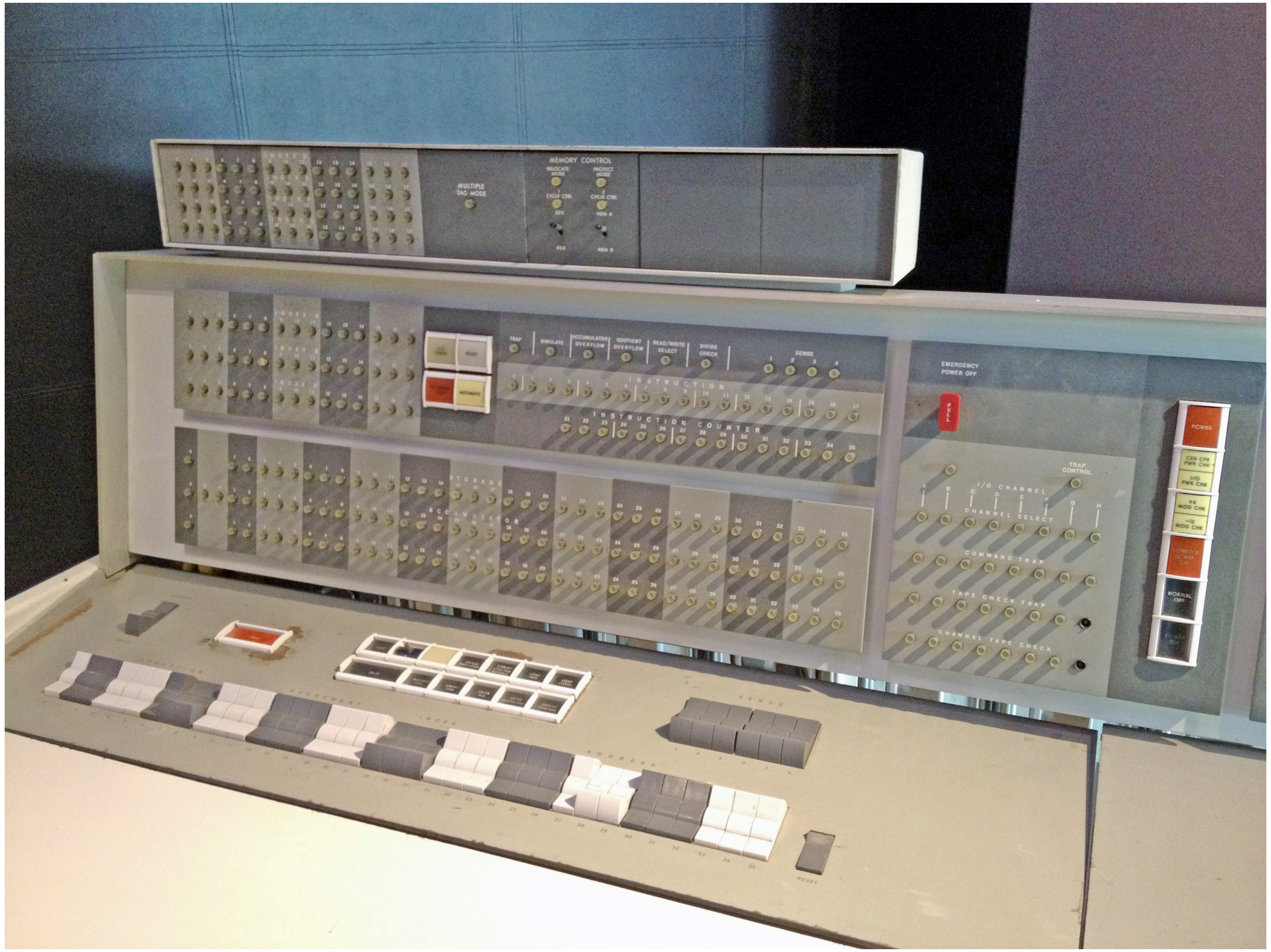

TX-0 COMPUTER

Jack Dennis

- Worked on TX line. Helped cultivate hacker culture at MIT, a precursor of what would come in the 80s
- One of the founders of Multics project, which would inspire Ken Thompson for UNIX, and later Linux.
- Designed a lot of cool architectures. His dataflow architectures are still inspiring new chips (including Google's TPU)

IBM 7094 (~1960)

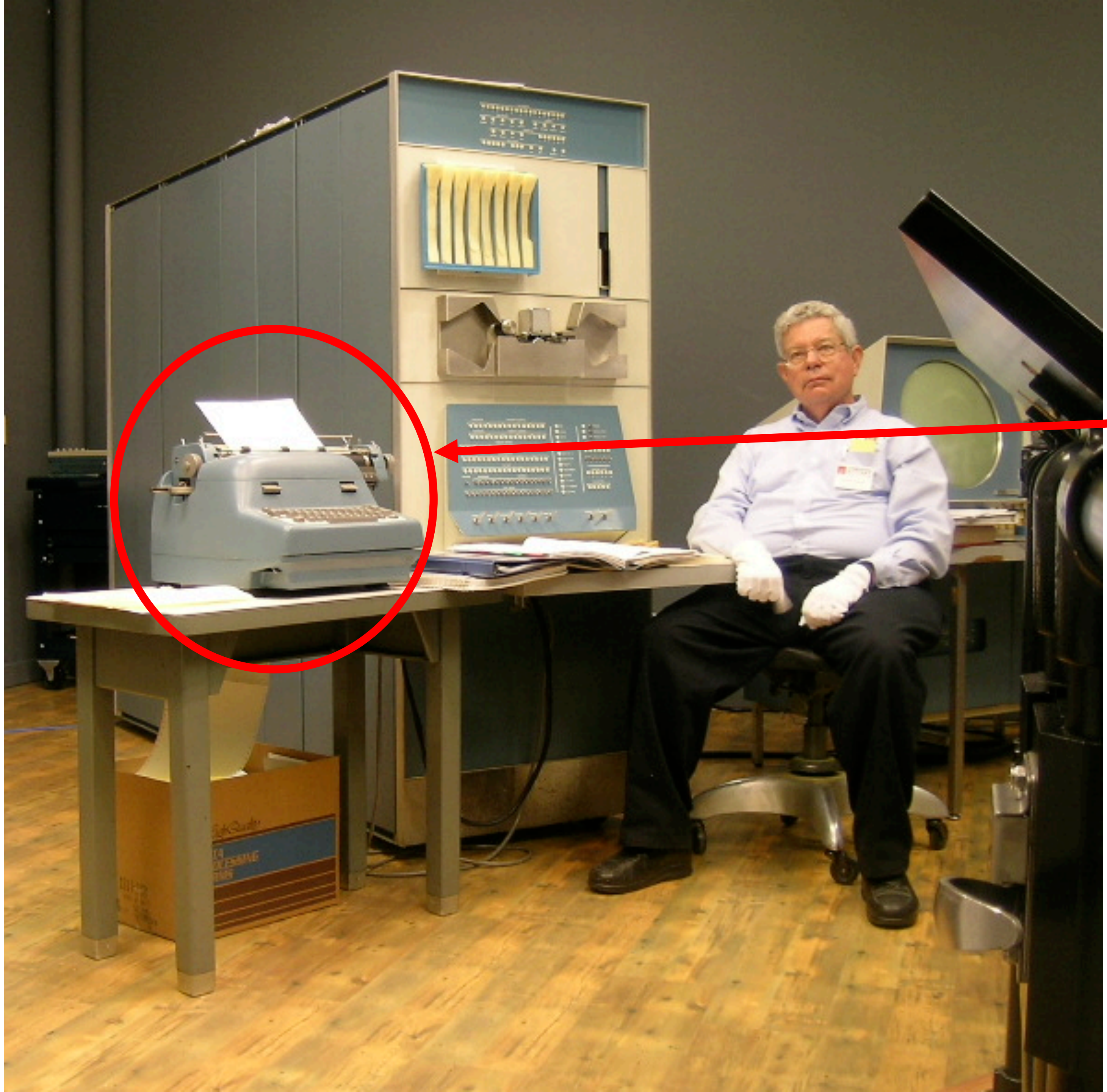
IBM 7094 (~1960)



1961: CTSS on the IBM 709

- MIT (again) develops the Compatible Time Sharing System, one of the first time sharers
- The goal was debugging: batch processing sucks!
- https://archive.org/details/large-fast-computers/page/n5/mode/2up
- Interestingly, LISP was written on the 709 series

First sold to Bolt, Beranek, and Newman (BBN) in 1960 ($1M in todays dollars)



1960

DEC PDP-1

Typewriter

DEC later eaten by Compaq in the late 90s

ILLINOIS INSTITUTE OF TECHNOLOGY

- 1962: BBN develops early prototype time-sharing system
- “The purpose of the BBN time-sharing system is to increase the effectiveness of the PDP-1 computer for those applications involving manmachine interaction by allowing each of the five users, each at his own typewriter to interact with the computer just as if he had a computer all to' himself”
- White paper in 1963: https://www.computer.org/csdl/pds/api/csdl/proceedings/download-article/12OmNvpew7R/pdf

Meanwhile in CT

- John Kemeny and Thomas Kurtz want a machine that can be used by all students, and not just math/science students
- 1964: They get funding from NSF to build a time-sharing system for the GE-225 (would become DTSS)
- Teletypes allow hundreds of undergrads to use the machine
- BASIC is born!

Early terminals: The Teletype

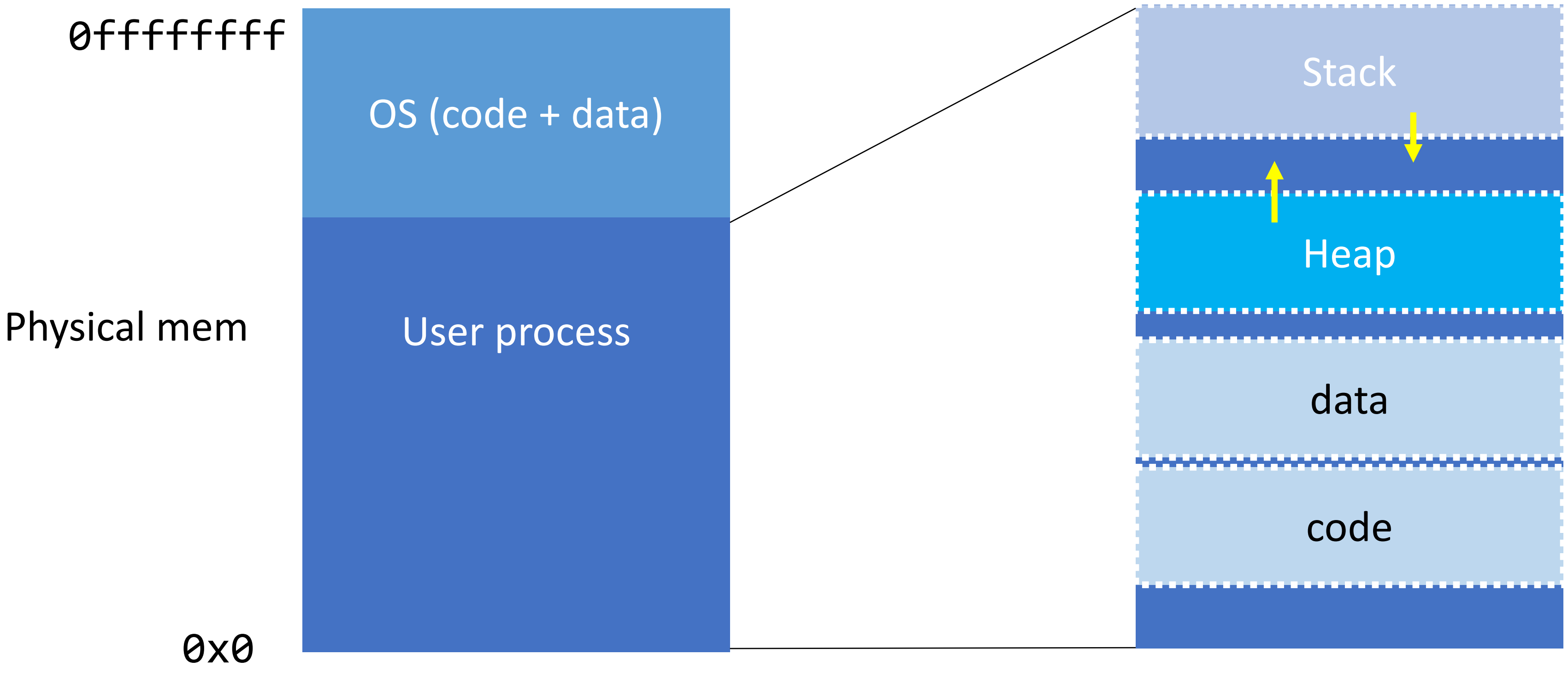
TeleTYpe

DEC VT100

We’re going for more virtualization

- Virtualizing the CPU: give the *illusion* of **private CPU (registers)**
- Virtualize memory: give the *illusion* of **private memory**

Anatomy of a uniprocess address space



The physical address space is more complicated than this!

What's wrong with this?

What we want from multiprogramming

- ***Protection***
  - Process can't corrupt OS and other processes
  - Can't *read* their data either (privacy/security)
- ***Efficiency***
  - Don't waste resources (primarily fragmentation)
- ***Sharing*** (of resources, of addr. space portions)
- ***Transparency***
  - Users not aware of sharing
  - Works regardless of proc count

Address Space Refresher

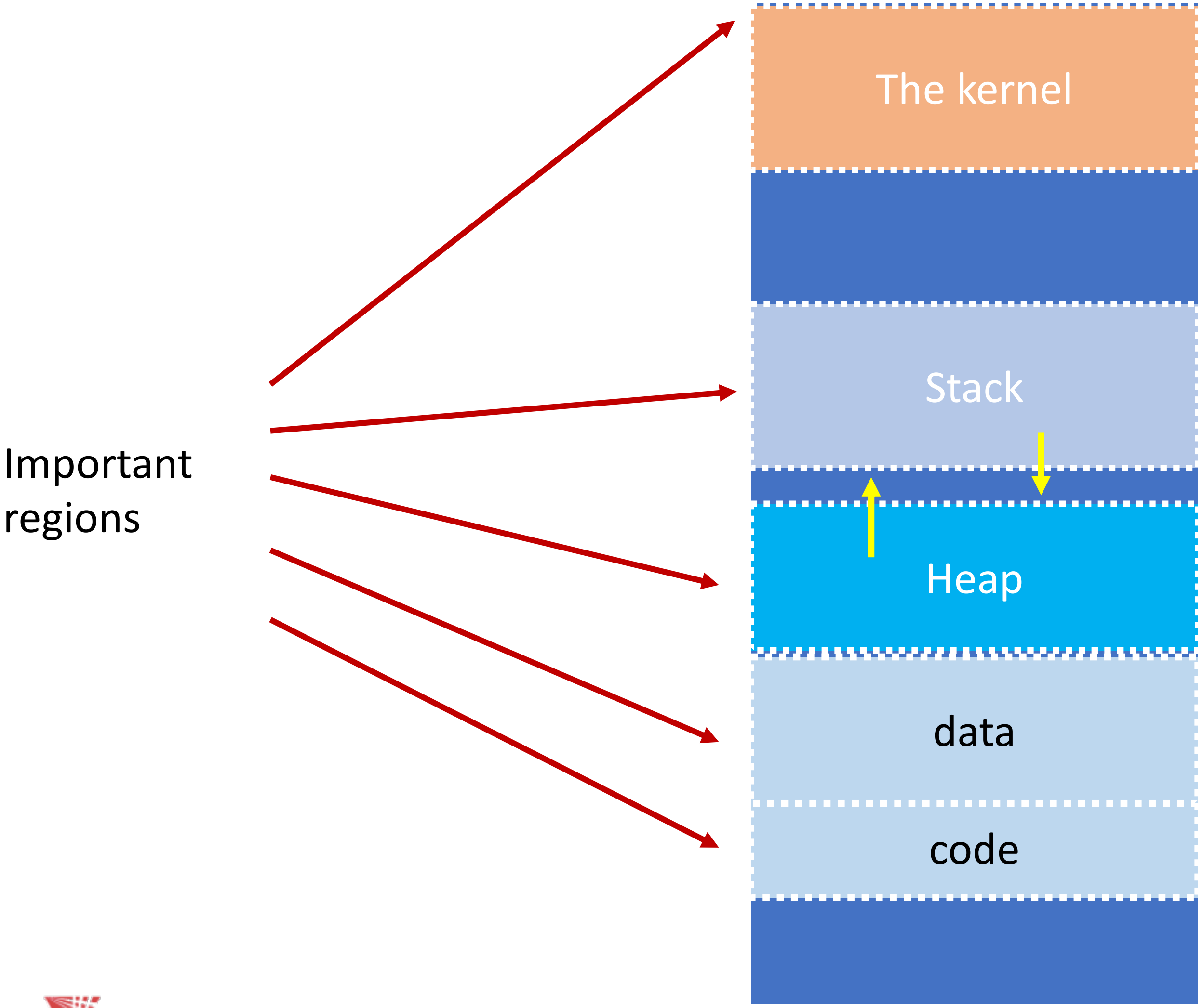
What is an address space?

- Most often just a finite set of numbers which we can map (uniquely) to objects in the real world
- Examples, memory address space (physical, virtual), IP address space, MAC address space, postal box address space, etc.
- For our purposes, a set of 2^n n-bit addresses, each of which maps to one memory location (a single byte on x86)

Address Space Regions

- The address space itself is pretty uninteresting
- Certain *regions* of the address space (subsets) usually have meaning attached to them by the OS

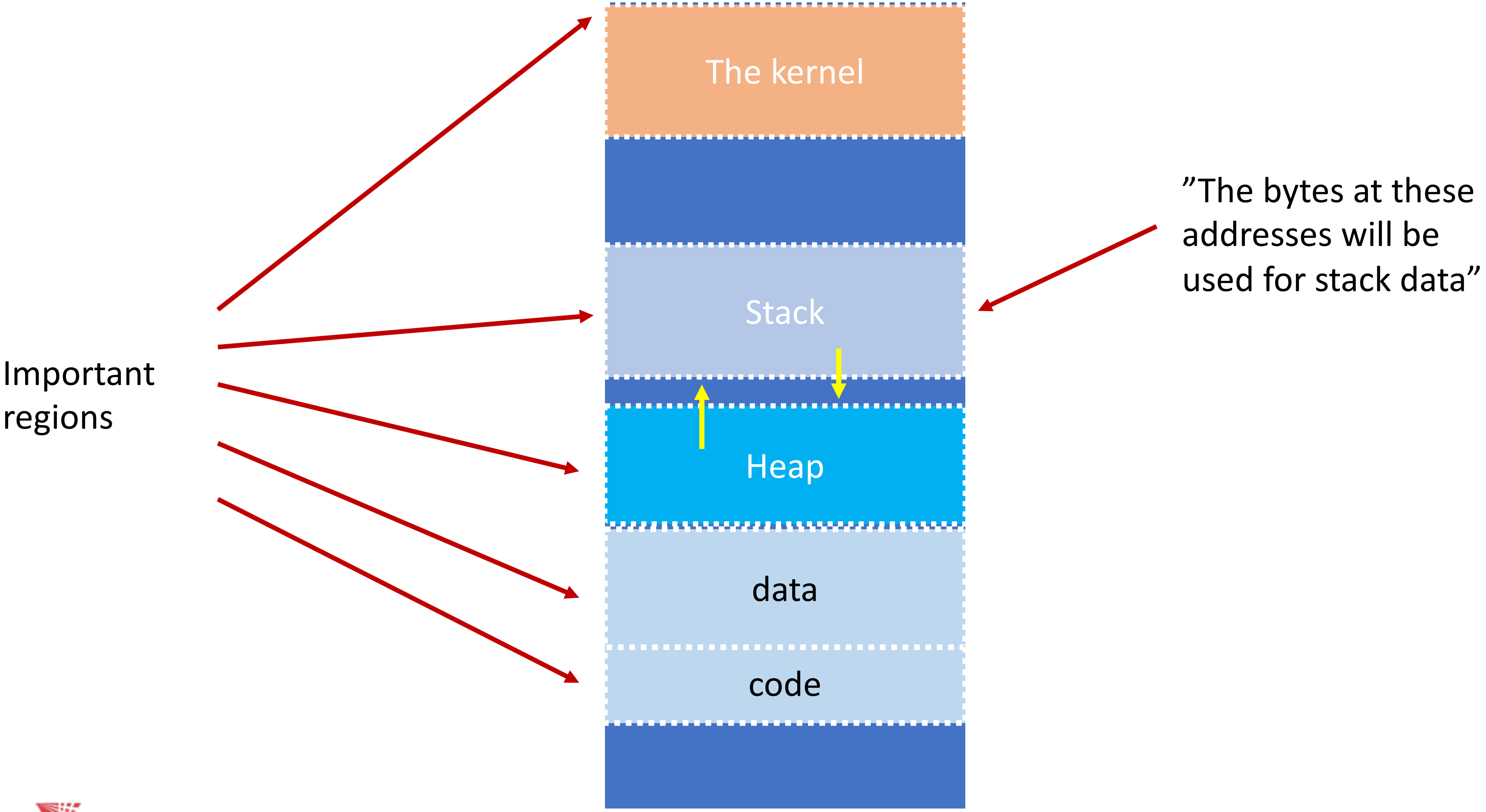
The usual process address space



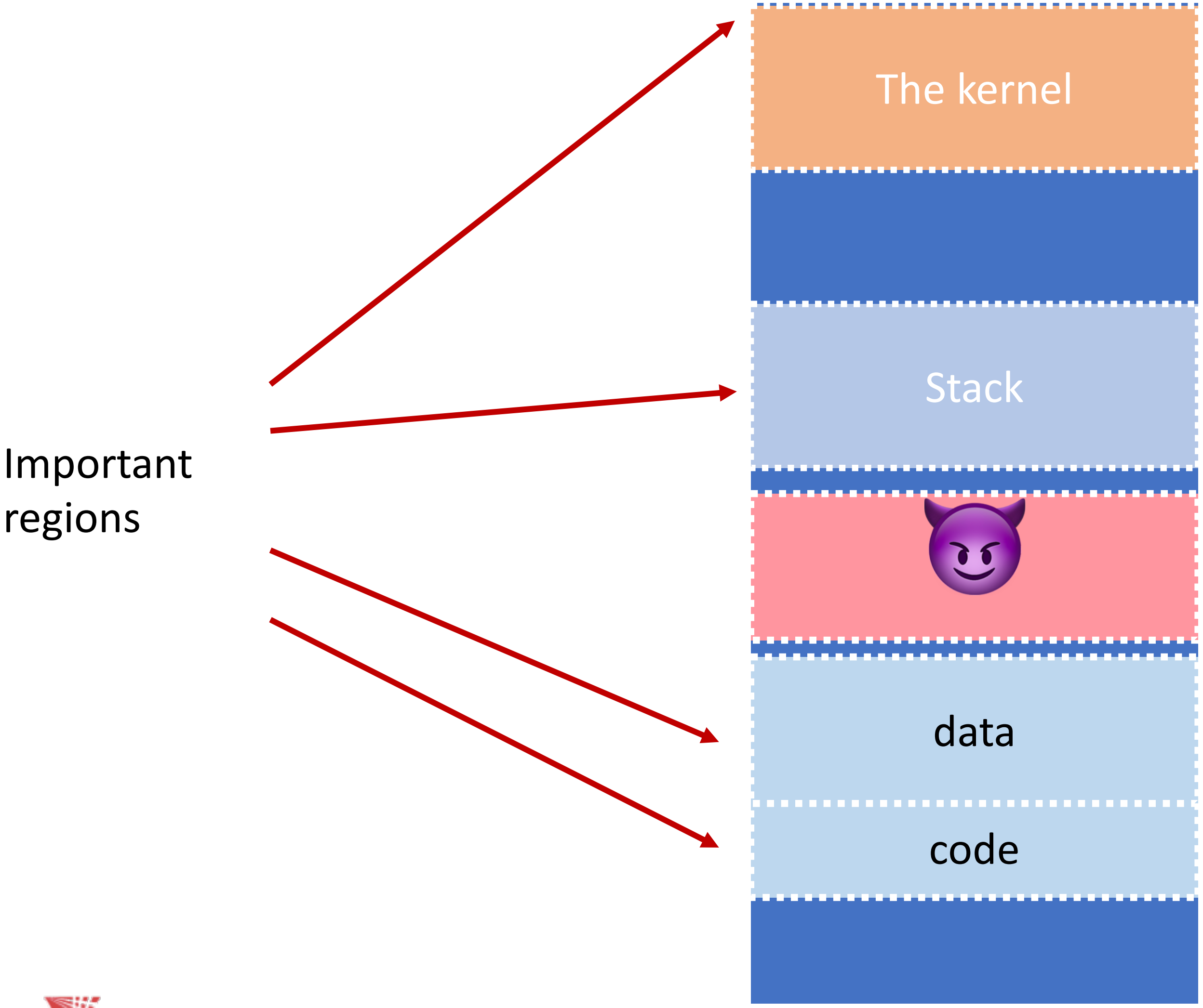
Address space regions have meaning

- What makes these regions within an address space interesting is *what meaning* is attached to the bytes they map to

The usual process address space



Address spaces are malleable…



`mmap(shellcode_fd, …);`

"The bytes at these addresses will hold my evil shellcode"

Getting the VA map on Linux

```
khale1@cs450-spring-2019:~$ cat /proc/self/maps
5563e53d5000-5563e53dd000 r-xp 00000000 fd:00 1831469                    /bin/cat
5563e55dc000-5563e55dd000 r--p 00007000 fd:00 1831469                    /bin/cat
5563e55dd000-5563e55de000 rw-p 00008000 fd:00 1831469                    /bin/cat
5563e5d98000-5563e5db9000 rw-p 00000000 00:00 0                          [heap]
7f889f5c0000-7f889f89e000 r--p 00000000 fd:00 263416                     /usr/lib/locale/locale-archive
7f889f89e000-7f889fa85000 r-xp 00000000 fd:00 1051912                    /lib/x86_64-linux-gnu/libc-2.27.so
7f889fa85000-7f889fc85000 ---p 001e7000 fd:00 1051912                    /lib/x86_64-linux-gnu/libc-2.27.so
7f889fc85000-7f889fc89000 r--p 001e7000 fd:00 1051912                    /lib/x86_64-linux-gnu/libc-2.27.so
7f889fc89000-7f889fc8b000 rw-p 001eb000 fd:00 1051912                    /lib/x86_64-linux-gnu/libc-2.27.so
7f889fc8b000-7f889fc8f000 rw-p 00000000 00:00 0
7f889fc8f000-7f889fcb6000 r-xp 00000000 fd:00 1051899                    /lib/x86_64-linux-gnu/ld-2.27.so
7f889fe76000-7f889fe9a000 rw-p 00000000 00:00 0
7f889feb6000-7f889feb7000 r--p 00027000 fd:00 1051899                    /lib/x86_64-linux-gnu/ld-2.27.so
7f889feb7000-7f889feb8000 rw-p 00028000 fd:00 1051899                    /lib/x86_64-linux-gnu/ld-2.27.so
7f889feb8000-7f889feb9000 rw-p 00000000 00:00 0
7ffd96b25000-7ffd96b46000 rw-p 00000000 00:00 0                          [stack]
7ffd96b85000-7ffd96b88000 r--p 00000000 00:00 0                          [vvar]
7ffd96b88000-7ffd96b8a000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
khale1@cs450-spring-2019:~$
```

code

heap

Dynamically Linked libraries

Dynamic linker's Code & data

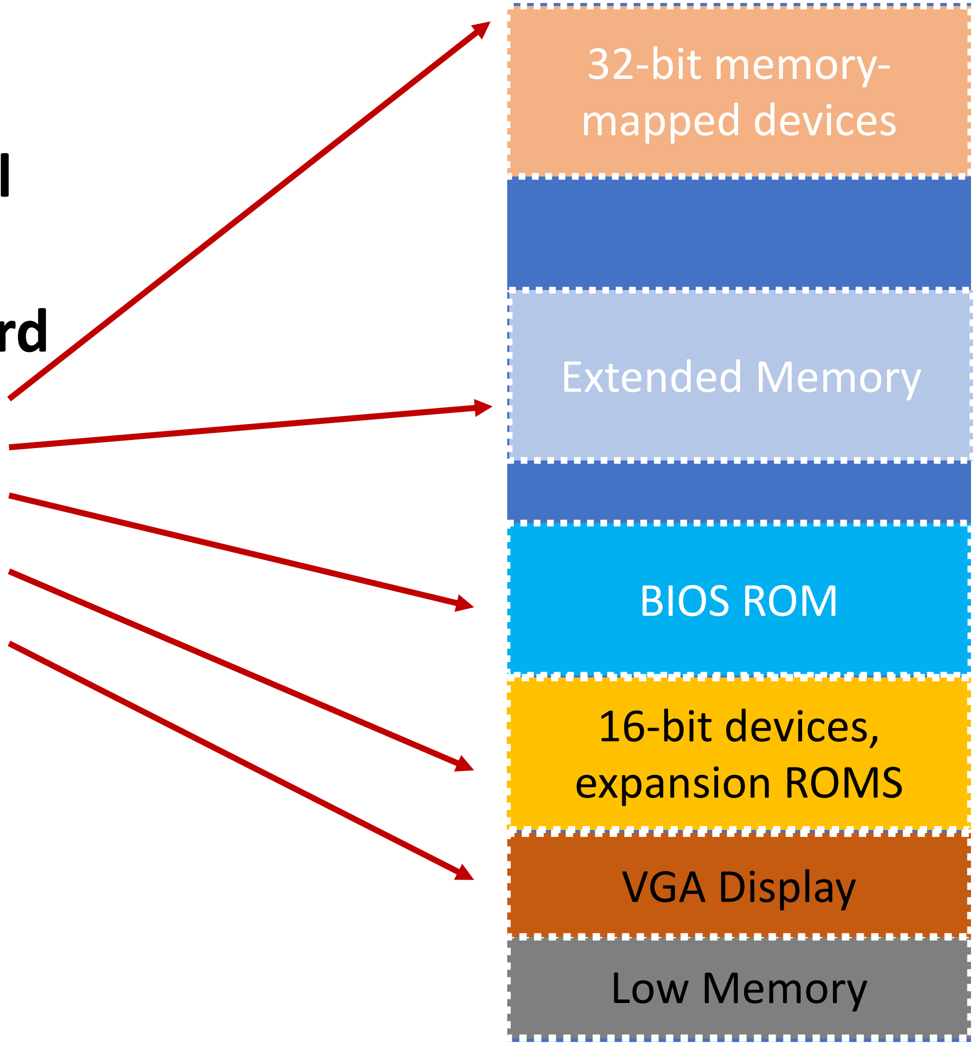
stack

Kernel being clev

Address spaces don't necessarily map bytes to RAM

PC 32-bit physical address space (differs from board to board)

Important regions



`0xffffffff (4GB)`

Only dark blue area gets routed to RAM chips by the memory controller!

`Depends on System RAM`

`0x00100000 (1MB)`

`0x000f0000`

`0x000c0000`

`0x000a0000`

`0x00000000`

Getting the PA map on Linux

```
cs450_inst@cs450-spring-2019:~$ sudo cat /proc/iomem
00000000-00000fff : Reserved
00001000-0009fbff : System RAM
0009fc00-0009ffff : Reserved
000a0000-000bffff : PCI Bus 0000:00
000c0000-000c0dff : Video ROM
000f0000-000fffff : Reserved
  000f0000-000fffff : System ROM
00100000-bffdefff : System RAM
  87400000-880031d0 : Kernel code
  880031d1-88a6a1ff : Kernel data
  88ce2000-88f3dfff : Kernel bss
bffdf000-bfffffff : Reserved
c0000000-febfffff : PCI Bus 0000:00
  fe000000-fe1fffff : PCI Bus 0000:02
    fe000000-fe07ffff : 0000:02:02.0
    fe080000-fe0800ff : 0000:02:02.0
      fe080000-fe0800ff : 8139cp
  fe200000-fe3fffff : PCI Bus 0000:01
    fe200000-fe27ffff : 0000:01:02.0
    fe280000-fe2800ff : 0000:01:02.0
```

A memory-mapped PCI device

BIOS

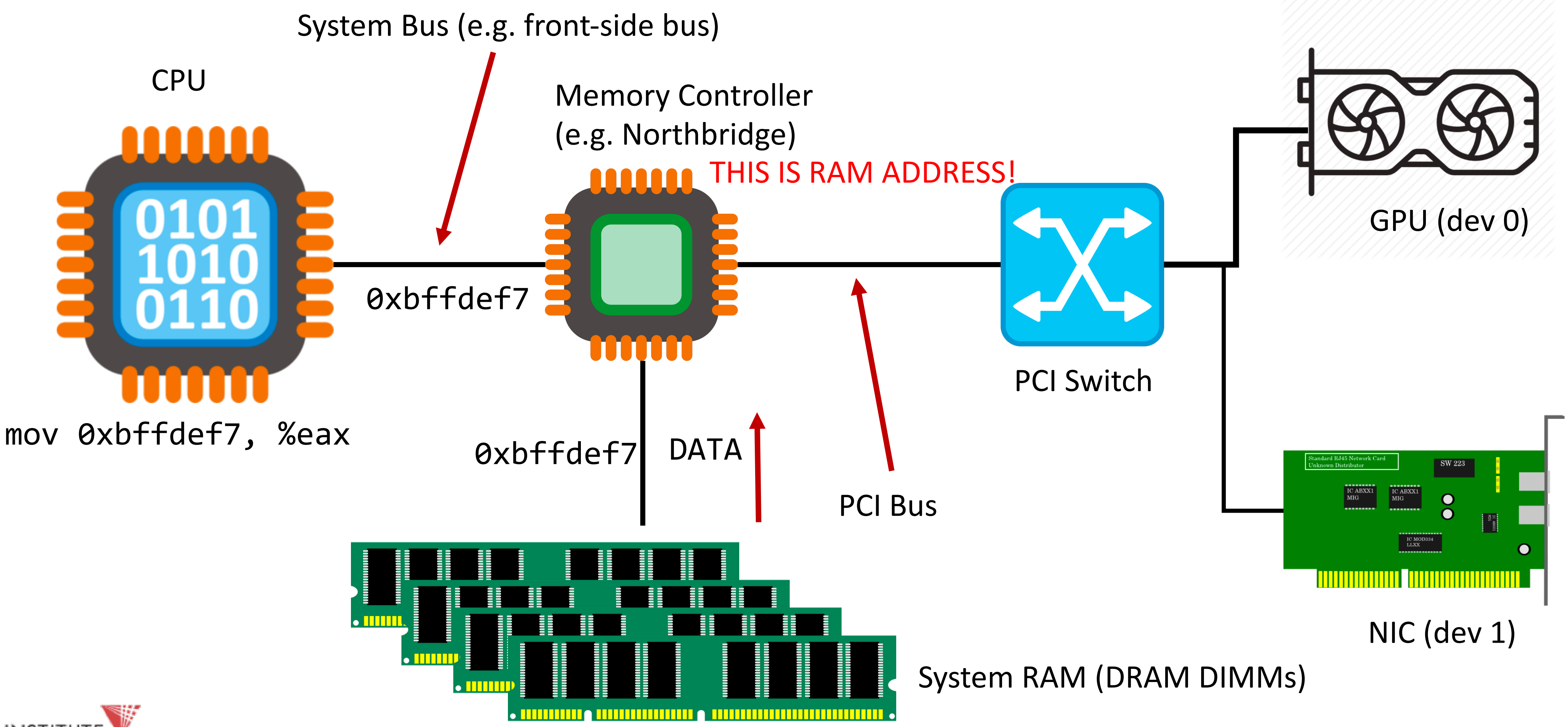
To the DRAM DIMMs

Where the bootloader put the kernel

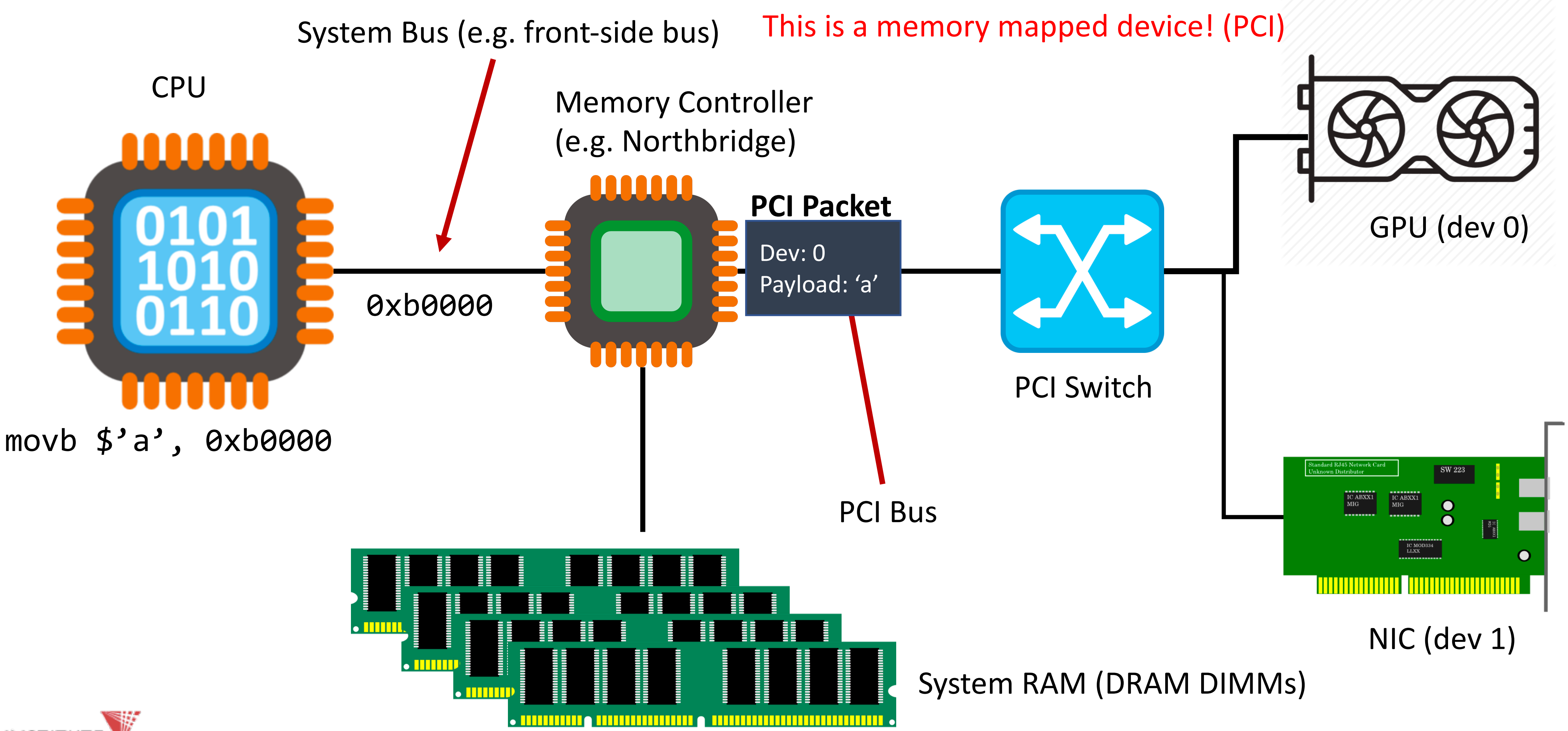
A PCI Bridge

A fancy PCI device

How the physical address space works



How the physical address space works



Our First Memory Virtualization Mechanisms

- Manual coordination
- Timesharing (mem dumping)
- Static relocation (compiler)
- Programmable Base
- Programmable Base + Bounds
- Segmentation

Coordination

- Have users coordinate so that memory addresses they use don’t collide
- If they *do* collide, it’s not the OS’s problem!
- We’ll need to know what processes will be running *beforehand*

Coordination Example

program 1

```
mov %eax, 0x1000
mov %ebx, 0x3000
```

program 2

```
mov %ecx, %edx
mov %edx, 0x2000
mov 0x3000, %ebx
```

there are collisions in the address space!

Coordination Example

program 1

```
mov %eax, 0x1000
mov %ebx, 0x3000
```

program 2

```
mov %ecx, %edx
mov %edx, 0x12000
mov 0x13000, %ebx
```

manual relocation

Problems with coordination

- A lot of effort! Not ***transparent***.
- Does not scale well when we add more and more users (programs) to the system. Not a good way to ***share resources***.
- Not portable
- Can’t add processes dynamically to the system (without rebooting)

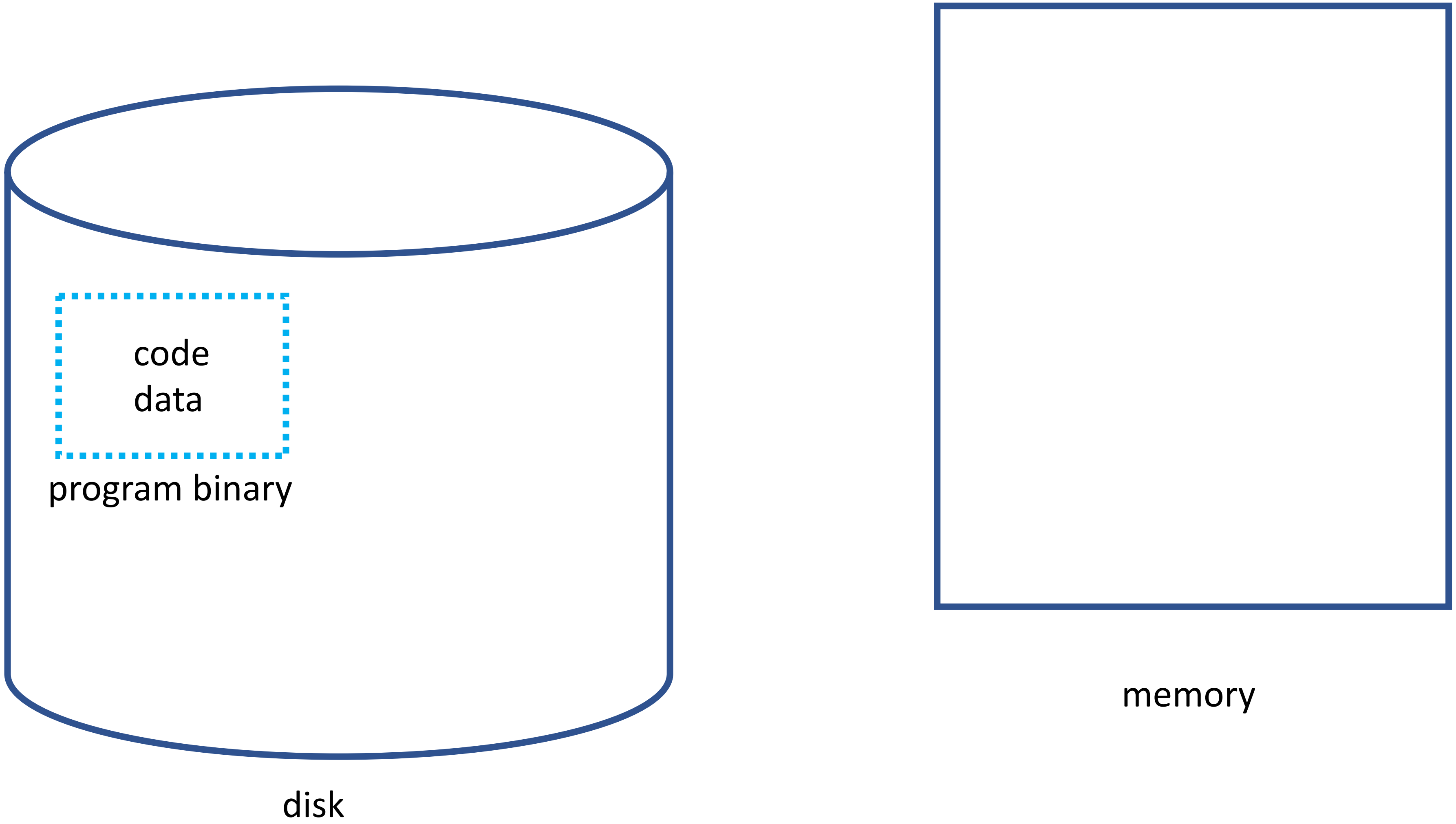
Our First Memory Virtualization Mechanisms

- Manual coordination
- Timesharing (mem dumping)
- Static relocation (compiler)
- Programmable Base
- Programmable Base + Bounds
- Segmentation

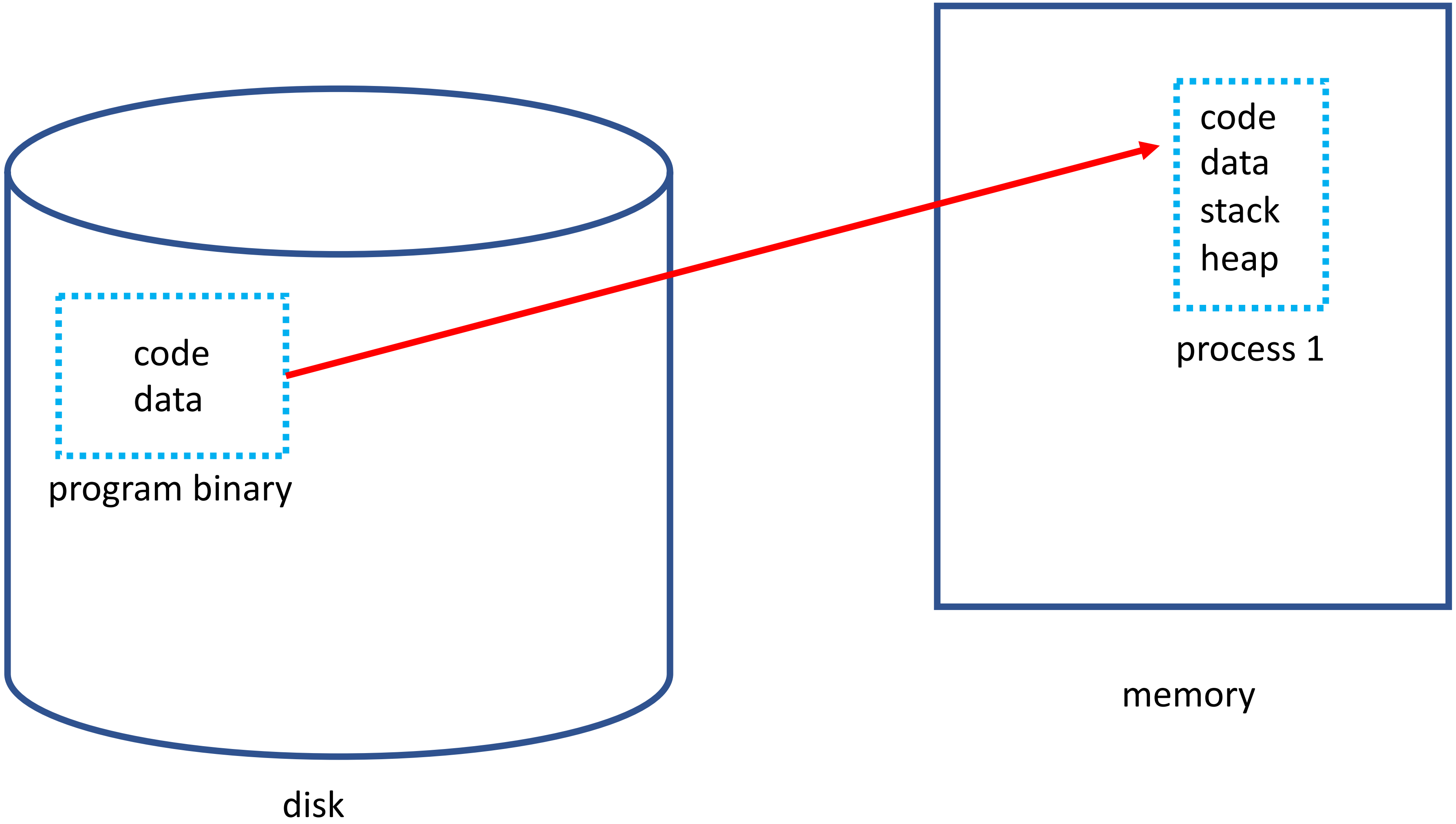
Timesharing

- Just like we virtualize the CPU, let's virtualize memory in *time*
- Give the illusion of many **virtual memories** by saving the memory of one process to disk when we context switch

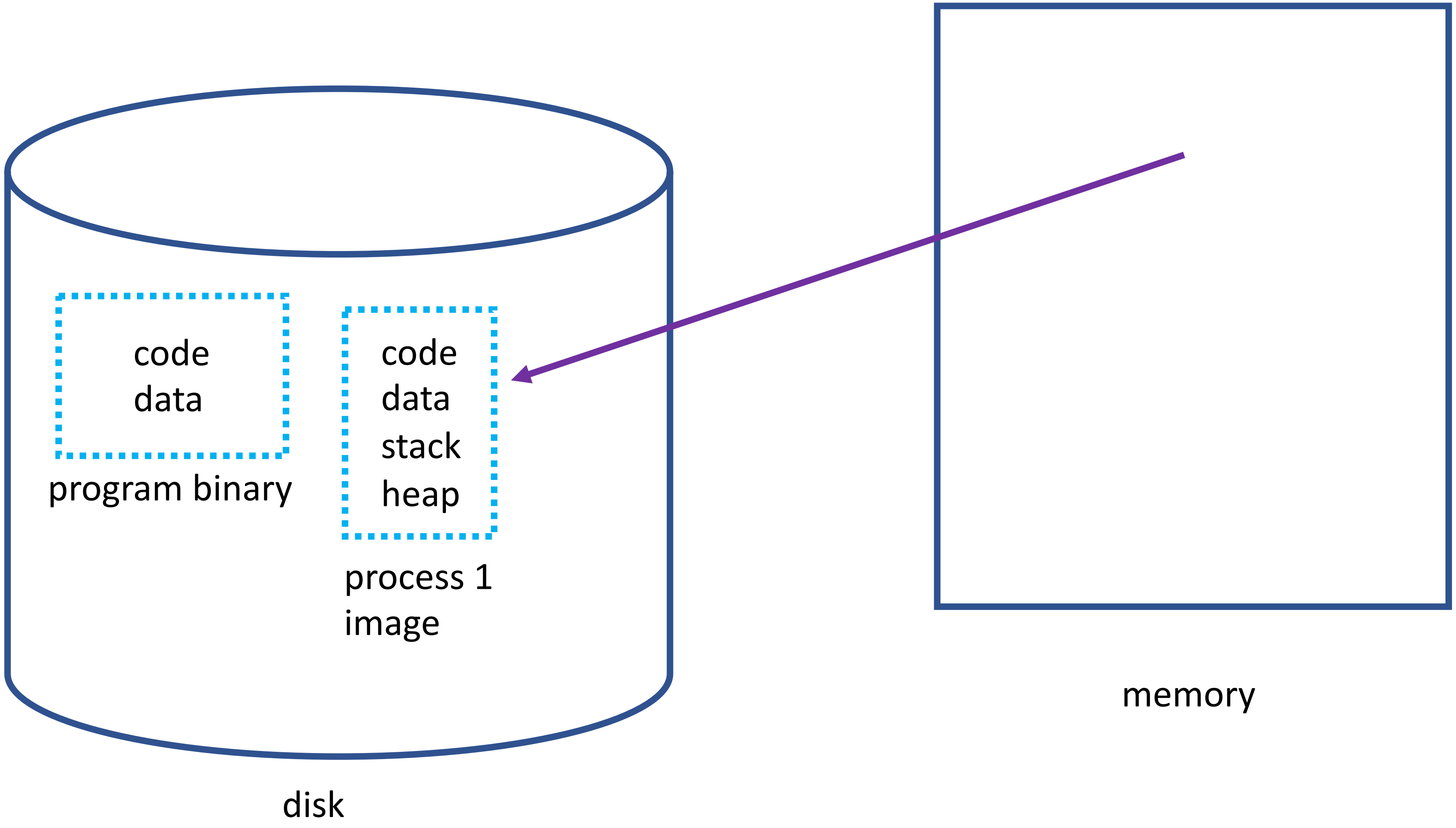
Timesharing Example



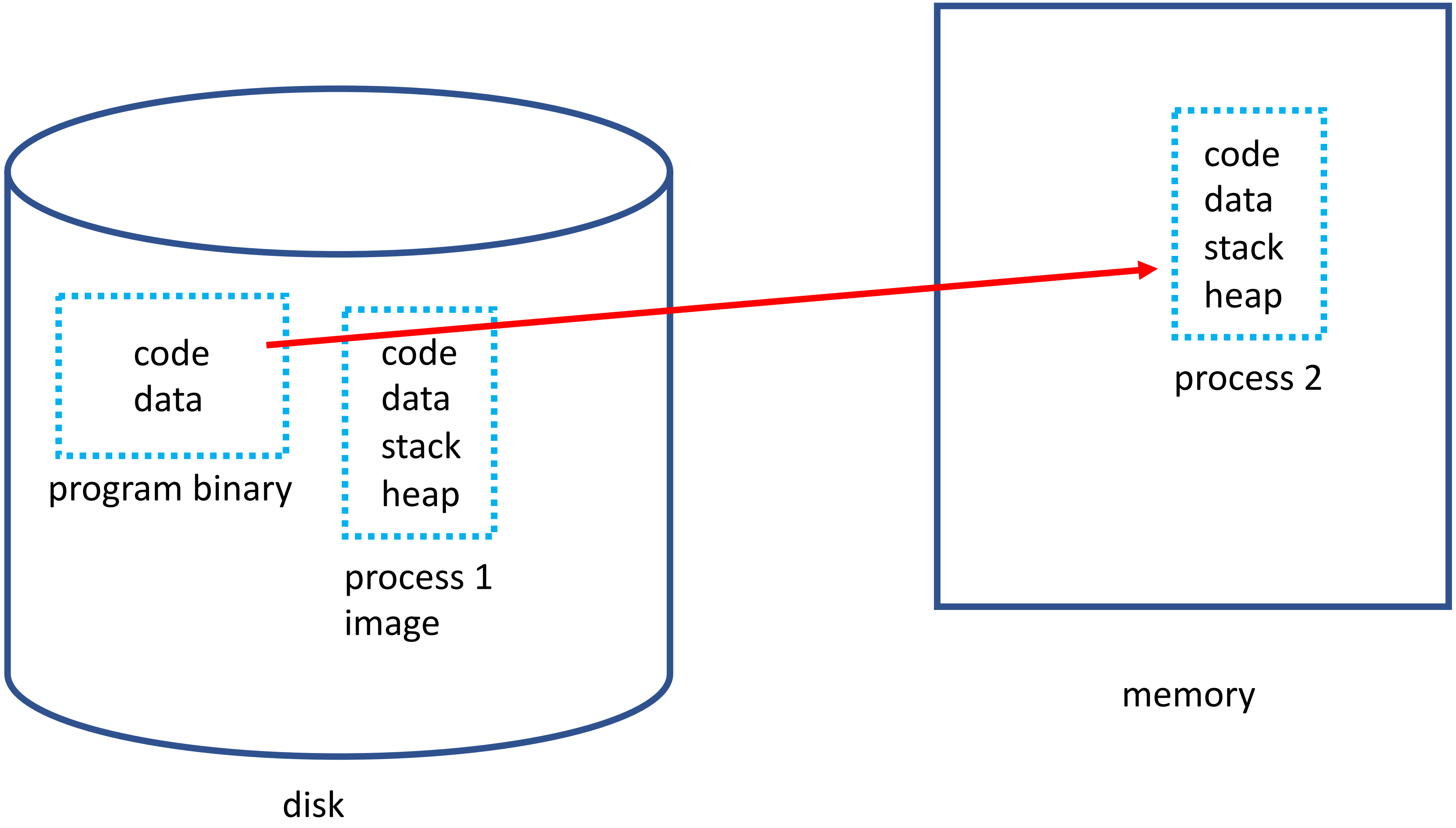
Timesharing Example



Timesharing Example



Timesharing Example



Problems with timesharing

- Very bad ***performance***
  - Disk is slow
  - We're saving the *entire* process image. That mean's *all* of the memory it uses

- We should go back to *spacesharing* (like in the coordination example), where we split memory up physically
- But we need to make things cleaner (and more ***transparent***)

Our First Memory Virtualization Mechanisms

- Manual coordination
- Timesharing (mem dumping)
- Static relocation (compiler)
- Programmable Base
- Programmable Base + Bounds
- Segmentation

Static relocation

- ***Spacesharing*** like coordination, but users aren't involved
- OS rewrites each program before loading it into memory as a process

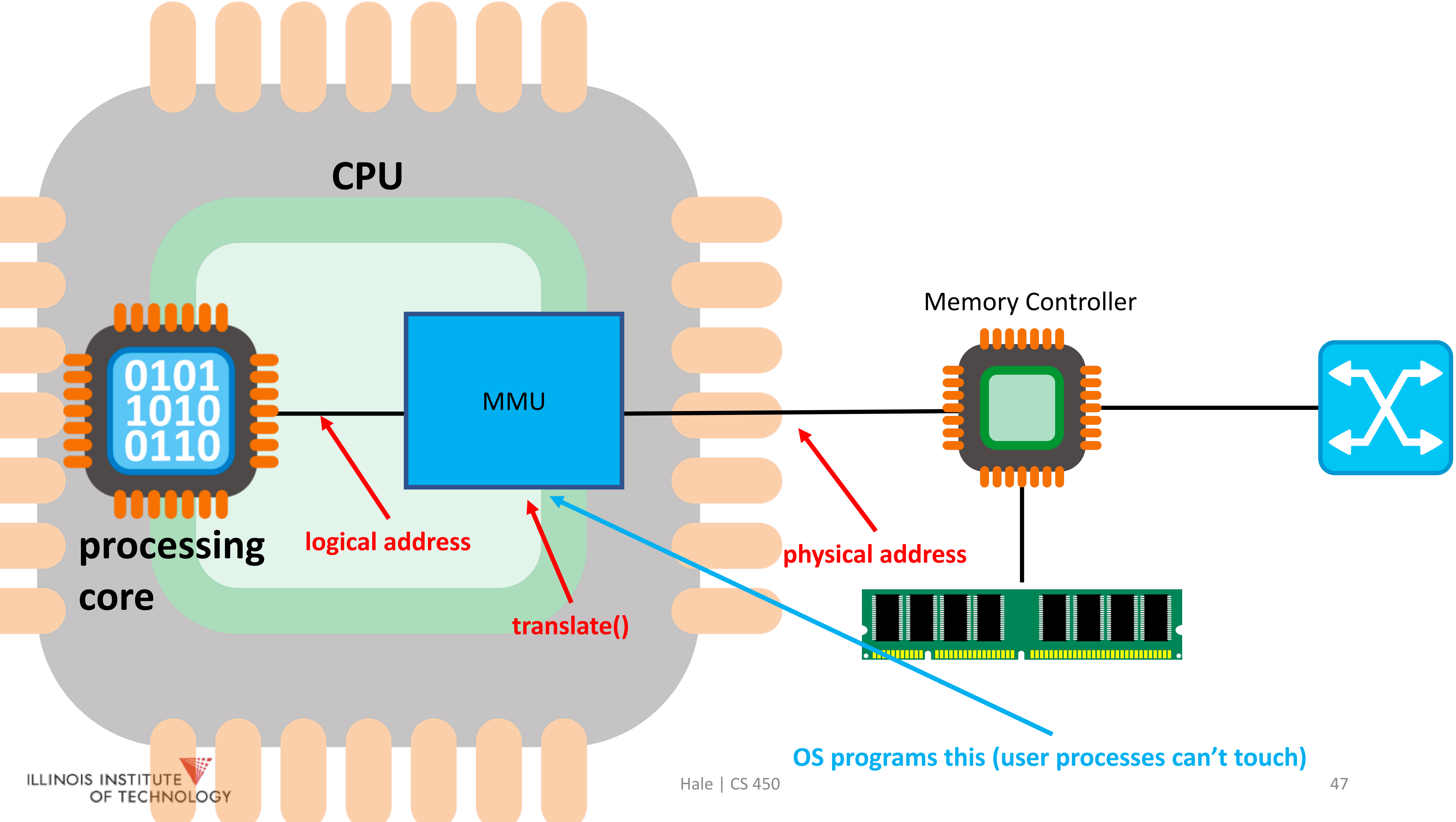
Problems with static relocation

- Better than manual coordination because ***we get some transparency***
- No ***protection/privacy***
  - Processes can overwrite each other's memory
  - And can read (no privacy)

- We can't move the address space after it's been created (unless we're willing to rewrite again)
- Scaling this is a pain when we add more procs

Dynamic Relocation

- We need to ***transparently protect*** processes from each other
- We stop ***trusting*** the user/programmer
- Hardware support!
  - Processor already has a memory management unit (MMU). We'll just add some more logic to it
- MMU changes addresses (behind process's back) on every memory reference
  - Process loads/stores/jmps/etc use ***logical addresses***.
  - The MMU translates these (automatically) into ***physical addresses***



CPU
Memory Controller
MMU
processing core
logical address
physical address
translate()
OS programs this (user processes can't touch)

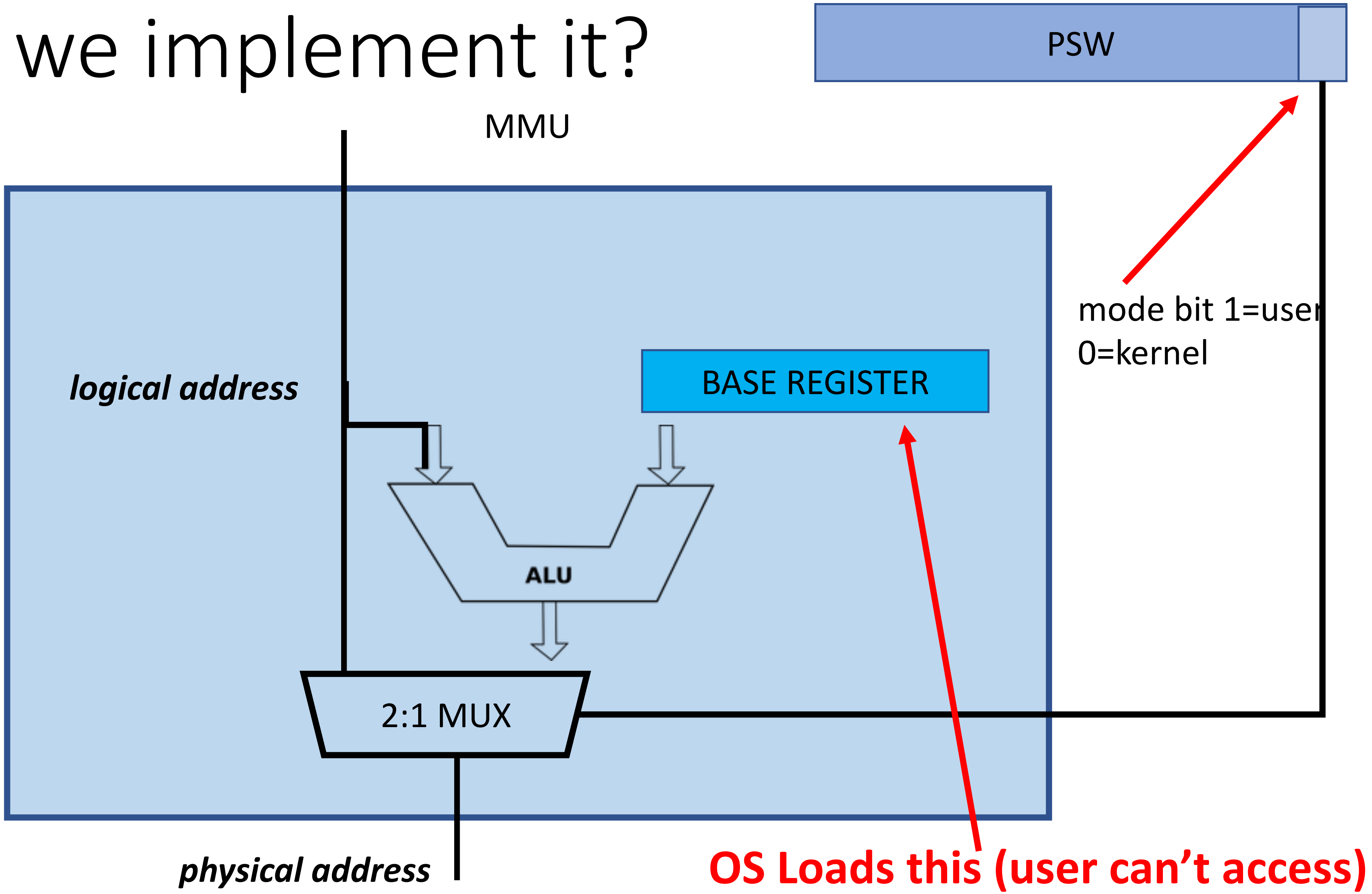
What hardware support do we need?

- Two operating modes
  - Privileged operation (protected mode, kernel mode): OS runs
    - **Only in this mode after: trap, system call, interrupt, exception**
    - Only some instructions in the ISA can be executed in this mode (e.g. **those that deal with the MMU**)
    - OS has access to ***all of physical memory***
  - User mode: user processes run in this mode. Can't touch privileged instructions!
    - Addresses are translated from logical addresses to physical addresses
  - We can implement these with a single bit in a control register

Our First Memory Virtualization Mechanisms

- Manual coordination
- Timesharing (mem dumping)
- Static relocation (compiler)
- Programmable Base
- Programmable Base + Bounds
- Segmentation

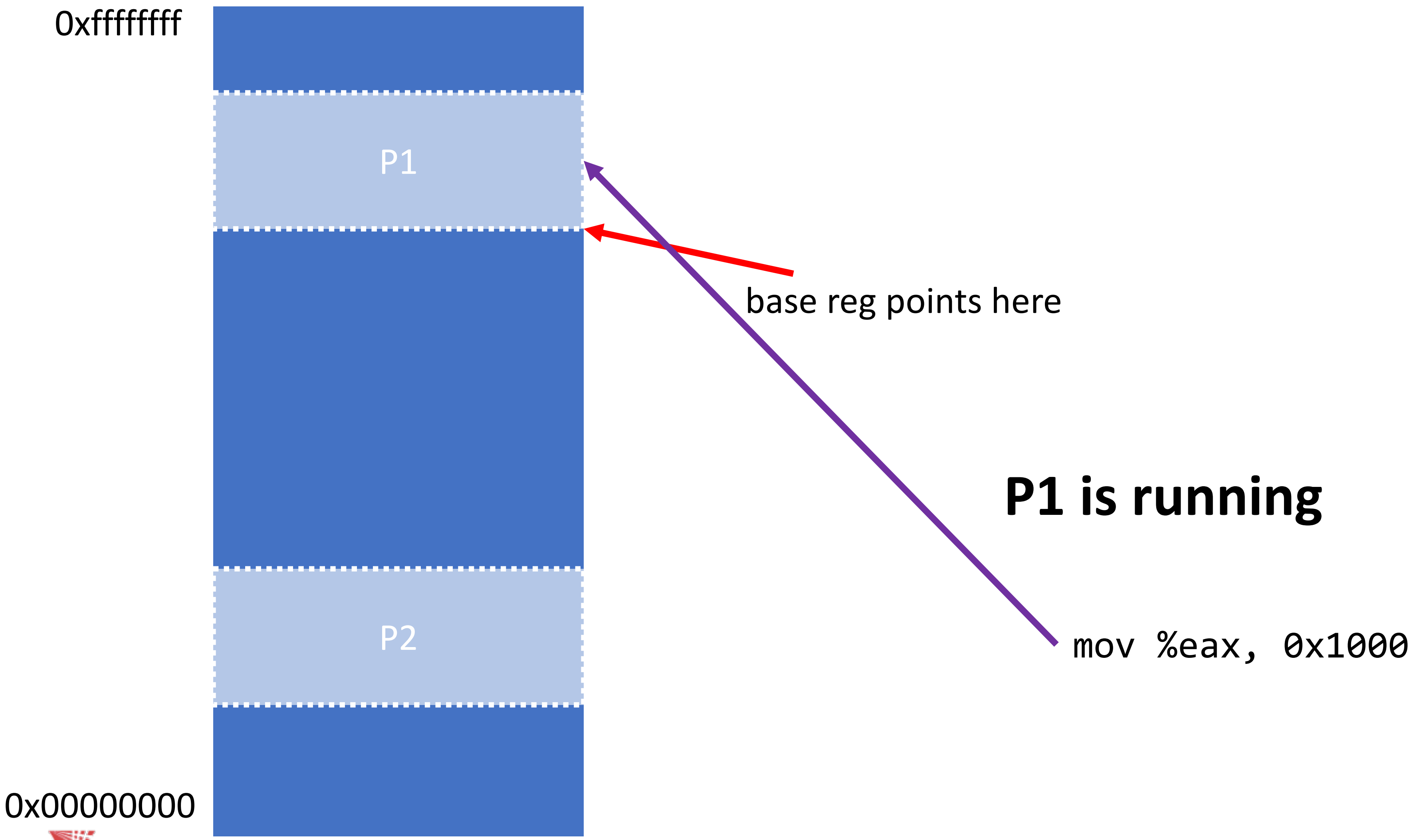
How do we implement it?



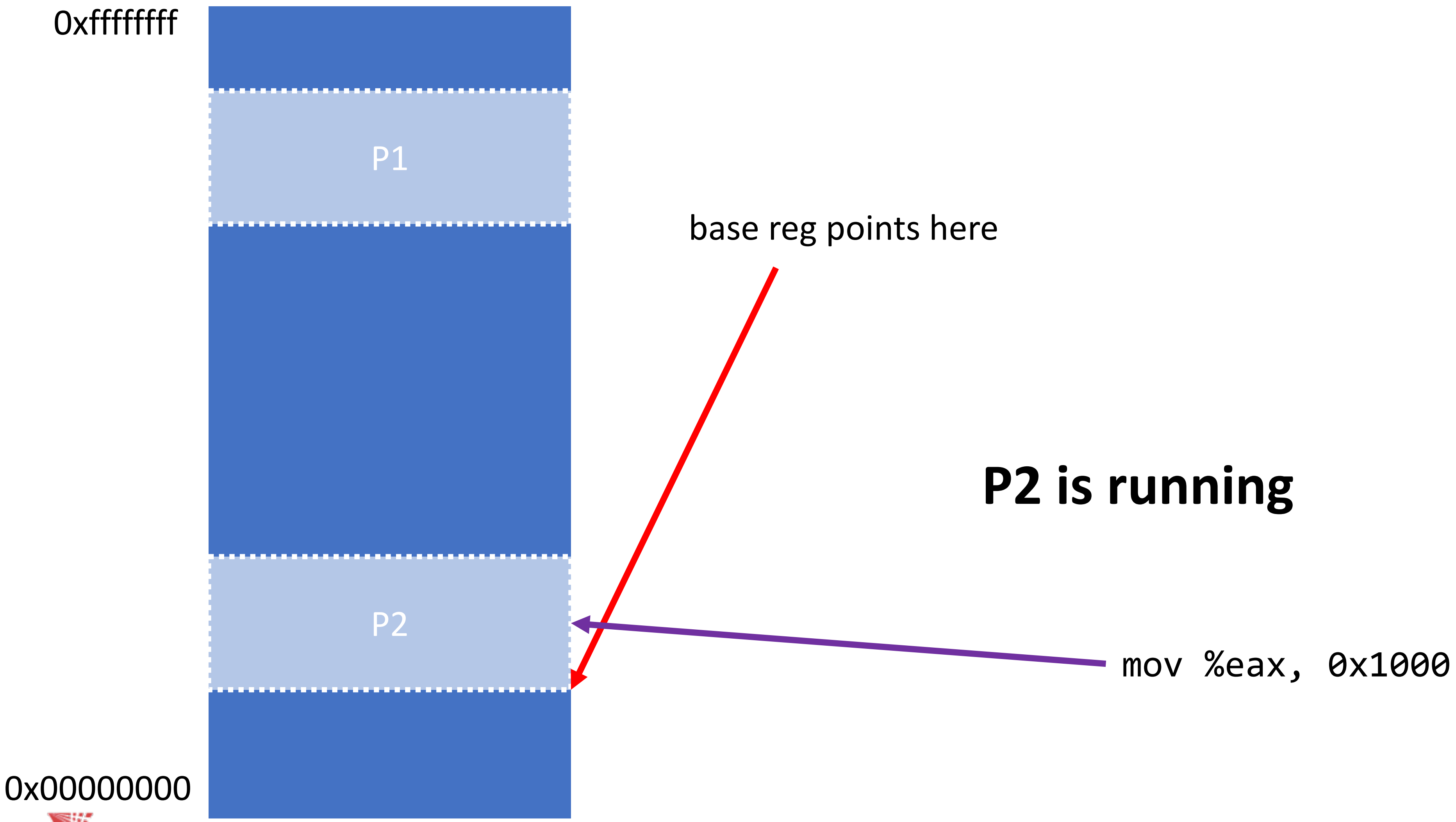
Programmable base register

- OS can write a new base register value (an offset) each time it loads a new process
- Translate a logical address into a physical address by adding the offset to the logical addr
- **Each process has its own base register value (determined by the OS)**

Example


0xffffffff
P1
base reg points here
P1 is running
P2
mov %eax, 0x1000
0x00000000

Example

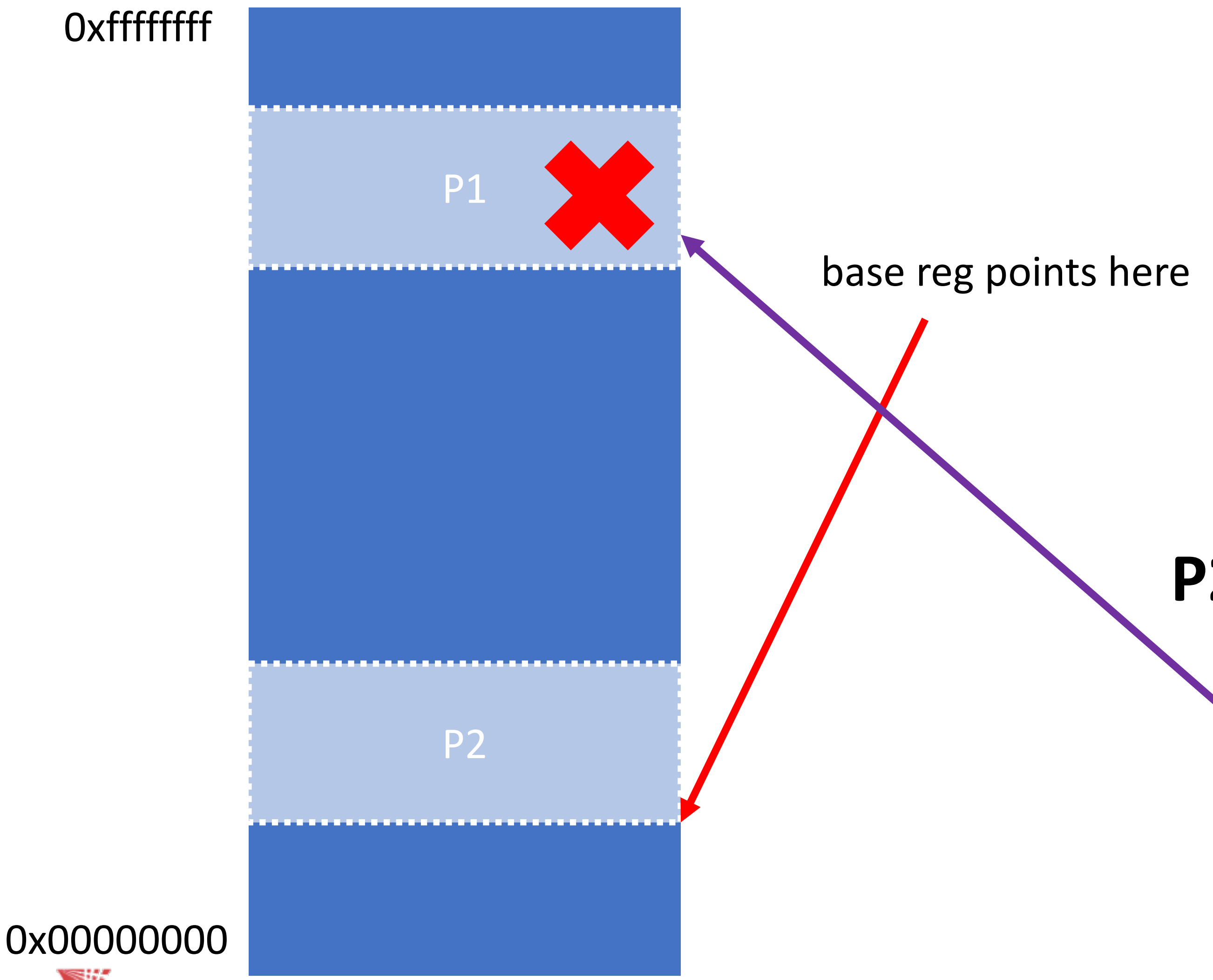


P2 is running

Problems with programmable base?

- We can run off the end!

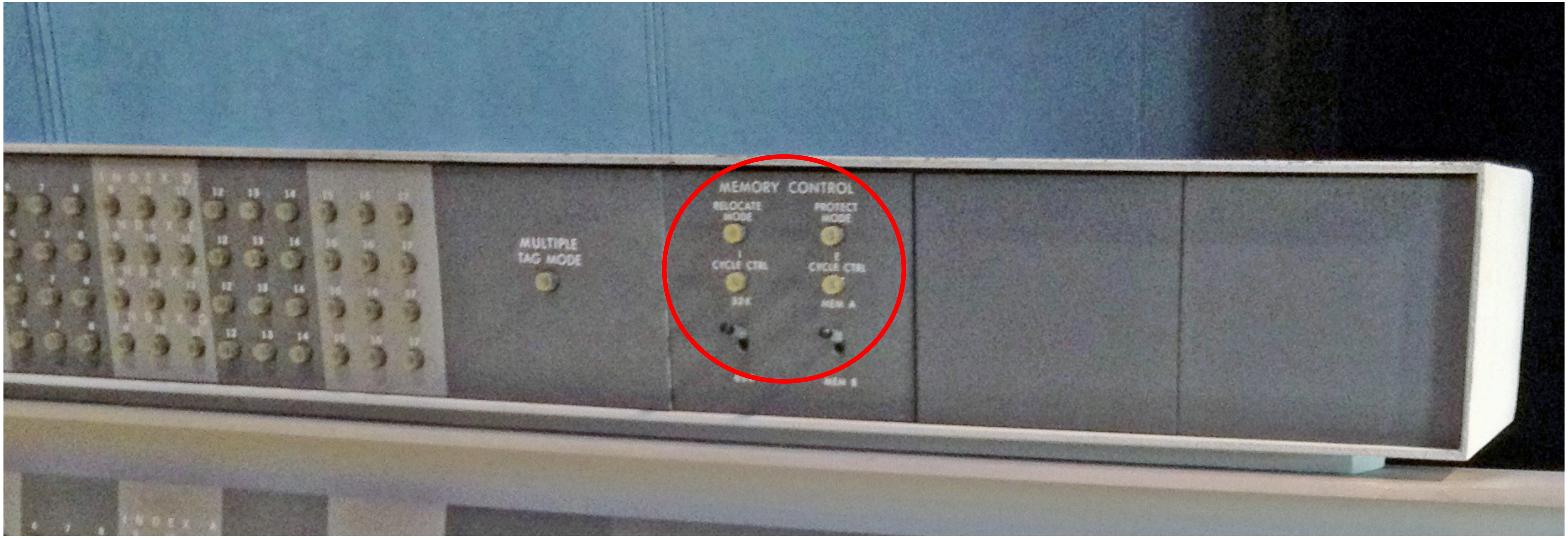
Example



P2 is running

`mov %eax, 0xf00000`

Dynamic Relocation on the IBM machine



Our First Memory Virtualization Mechanisms

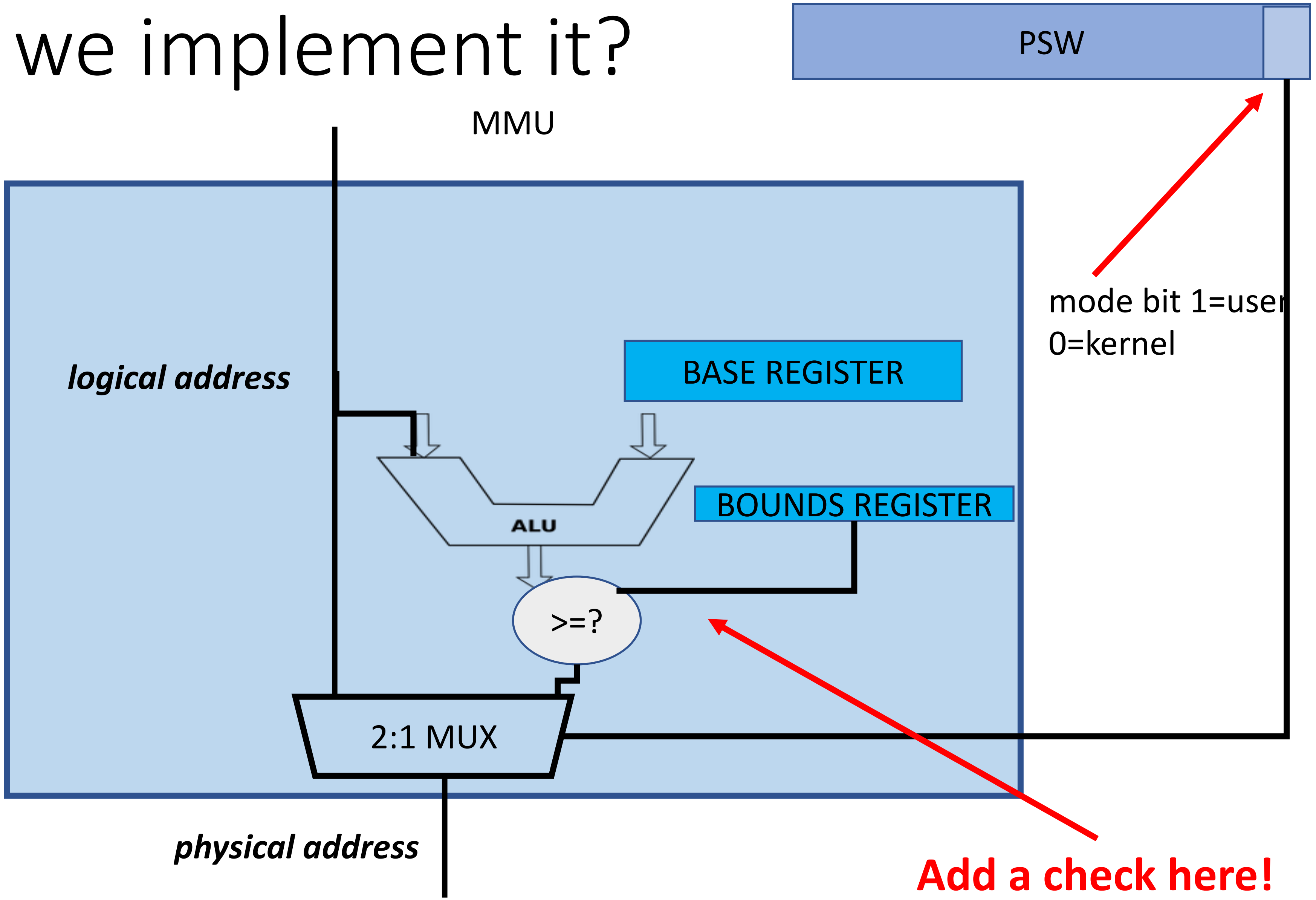
- Manual coordination
- Timesharing (mem dumping)
- Static relocation (compiler)
- Programmable Base
- Programmable Base + Bounds
- Segmentation

Base + Bounds

- Don’t allow memory references outside of the ***valid range***
- ***Constrain*** the address space
- Add one more piece of hardware: the bounds register
- Holds the highest valid address of an address space (can also hold the size)

```
if (base_reg + logical_addr < bounds) {
    phys_addr = base_reg + logical_addr;
} else {
    raise_exception();
}
```

How do we implement it?



Base + Bounds Advantages

- Provides ***protection*** (read *and* write) across address spaces
- Supports dynamic (***transparent***) relocation
- Simple and inexpensive to implement in hardware
- Fast (gives us good ***performance***)

Problems with Base + Bounds?

- Each process must be ***contiguously*** allocated in memory
- No ***sharing***: Can't share limited parts of the address space

Our First Memory Virtualization Mechanisms

- Manual coordination
- Timesharing (mem dumping)
- Static relocation (compiler)
- Programmable Base
- Programmable Base + Bounds
- Segmentation

Segmentation

- Divide the address space into logical segments
- Each segment corresponds to a logical region of the addr space (e.g. code, data, stack, heap, etc.)
- Each segment can independently:
  - Be placed separately in phys memory
  - grow and shrink
  - be protected (separate bits for read/write/execute permission)

Segmented Addressing

- Process now specifies segment and offset within segment
- How?
  - Use part of the logical address
    - Top bits of the address select the segment (segment selector)
    - Low bits specify offset *within* the segment
- What if our address space is too small?
  - Don't use special bits, instead use special registers (x86)

Address Translation with Segmentation

MMU

Segment Table

Segment	Base	Bounds	R W
0	0x2000	0x6ff	1 0
1	0x0000	0x4ff	1 1
2	0x3000	0xfff	1 1
3	0x0000	0x0000	0 0

Problems with Segmentation

- Each segment must be allocated contiguously
- May not have sufficient physical memory for large segments

Summary

- Next time we'll look at a more elegant approach to virtual memory (with HW support)
- Reminder: reading
- Reminder: Project 1a due Monday night!