6.033 ~~in the news~~ on Mars

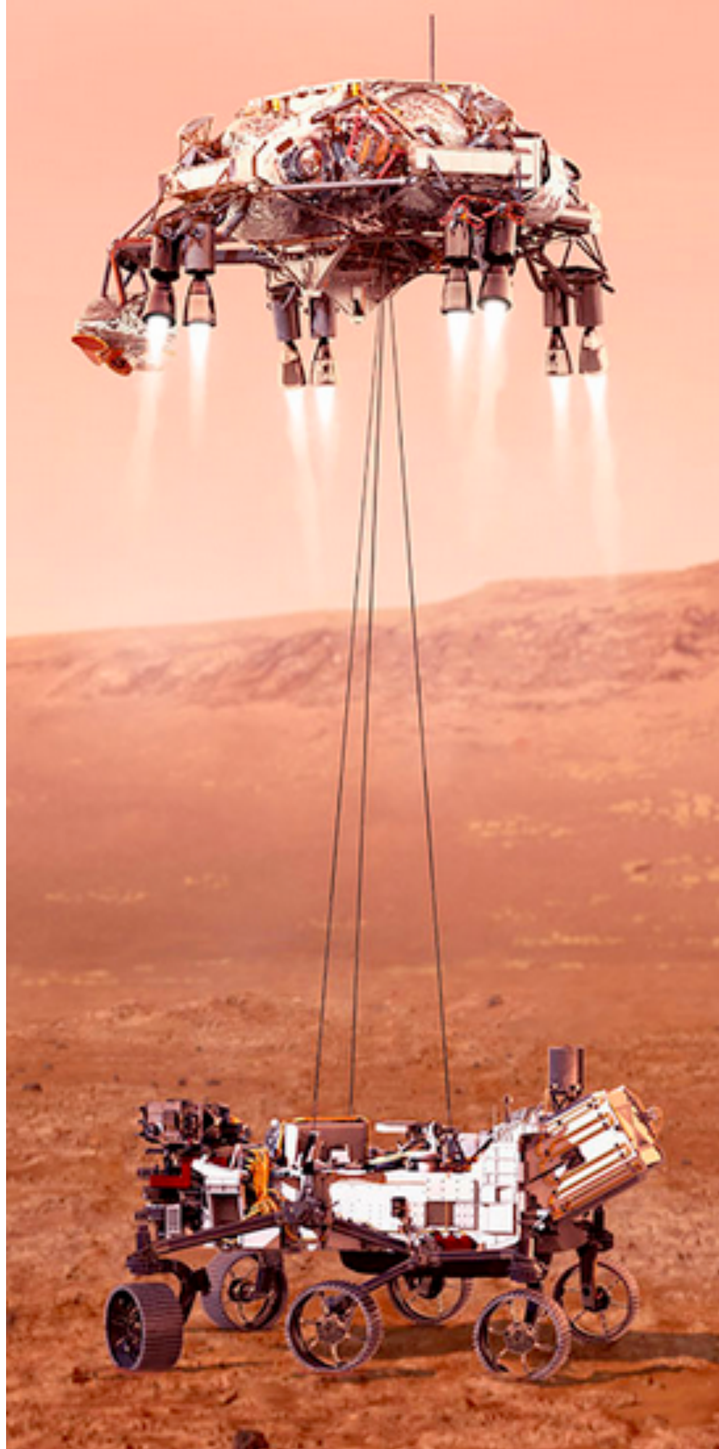
Tech Specs

Processor	• Radiation-hardened central processor with PowerPC 750 Architecture: a BAE RAD 750 • Operates at up to 200 megahertz speed, 10 times the speed in Mars rovers Spirit and Opportunity's computers
Memory	• 2 gigabytes of flash memory (~8 times as much as Spirit or Opportunity) • 256 megabytes of dynamic random access memory • 256 kilobytes of electrically erasable programmable read-only memory

https://mars.nasa.gov/mars2020/spacecraft/rover/brains/

It generally takes about 5 to 20 minutes for a radio signal to travel the distance between Mars and Earth, depending on planet positions. Using orbiters to relay messages is beneficial because they are much closer to Perseverance than the Deep Space Network (DSN) antennas on Earth. The mass- and power-constrained rover can achieve high data rates of up to 2 megabits per second on the relatively short-distance relay link to the orbiters overhead. The orbiters then use their much larger antennas and transmitters to relay that data on the long-distance link back to Earth.

https://mars.nasa.gov/mars2020/spacecraft/rover/communications/



https://mars.nasa.gov/mars2020/

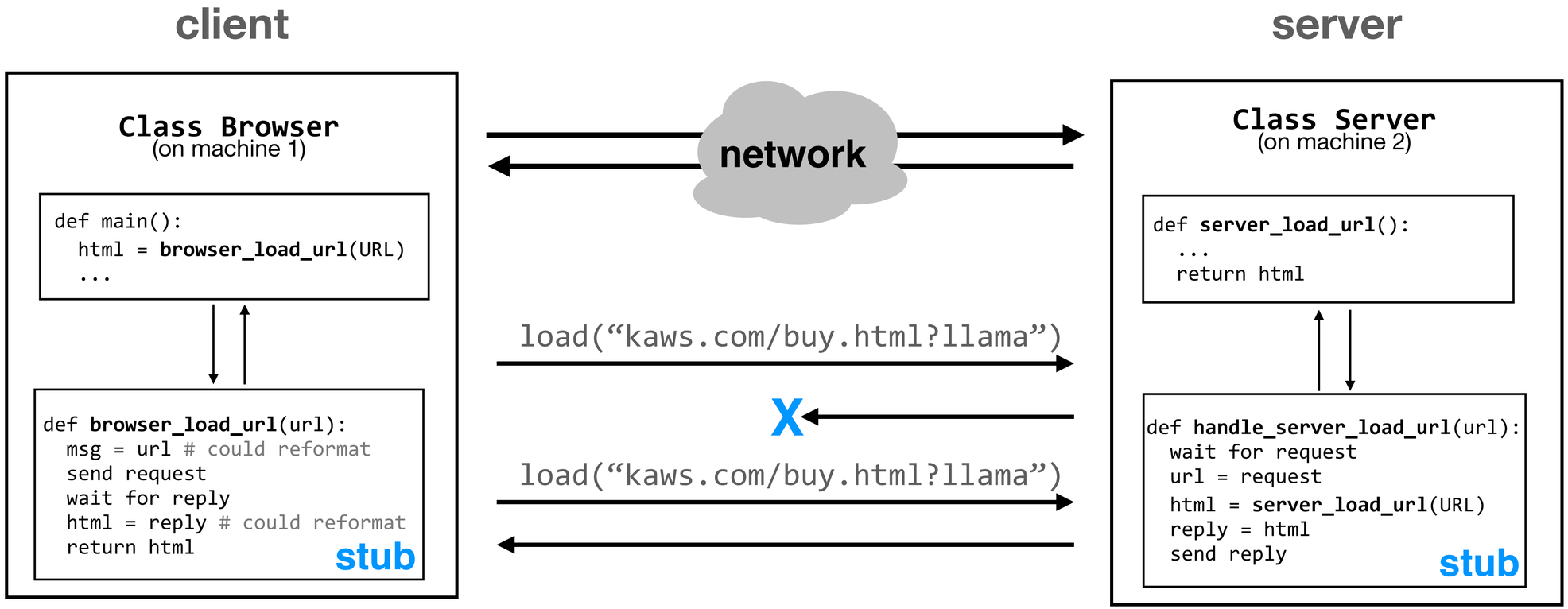
6.033 Spring 2021

Lecture #3: Virtual Memory

how does it work, but more importantly, why does an OS use it?

Katrina LaCurts | lacurts@mit.edu | 6.033 2021

last time: enforced modularity via client/server + naming



today: what if we *don't* want to put each module on a separate machine?

operating systems enforce modularity on a single machine

in order to enforce modularity + have an effective operating system, a few things need to happen

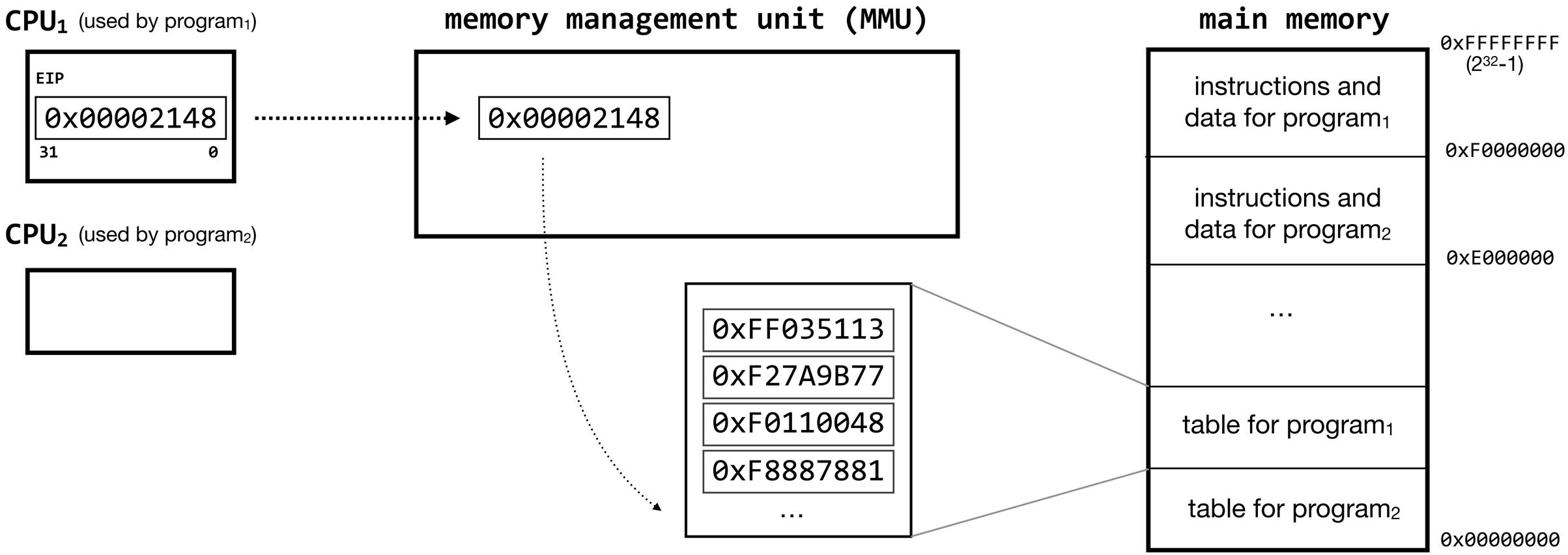
1. programs shouldn't be able to refer to (and corrupt) each others' **memory** → virtualize **memory**
2. programs should be able to **communicate** with each other → assume they don't need to (for today)
3. programs should be able to **share a CPU** without one program halting the progress of the others → assume one program per CPU (for today)

the primary technique that an operating system uses to enforce modularity is **virtualization**

in some sense, we want every program to *think* that it has access to the full physical hardware, when of course they don't; the OS *virtualizes* different components of hardware

what we want: every program to be able to access a full 32-bit address space

what we have: 2^{32} bytes of memory; every program can't *actually* have access to the full 32-bit space



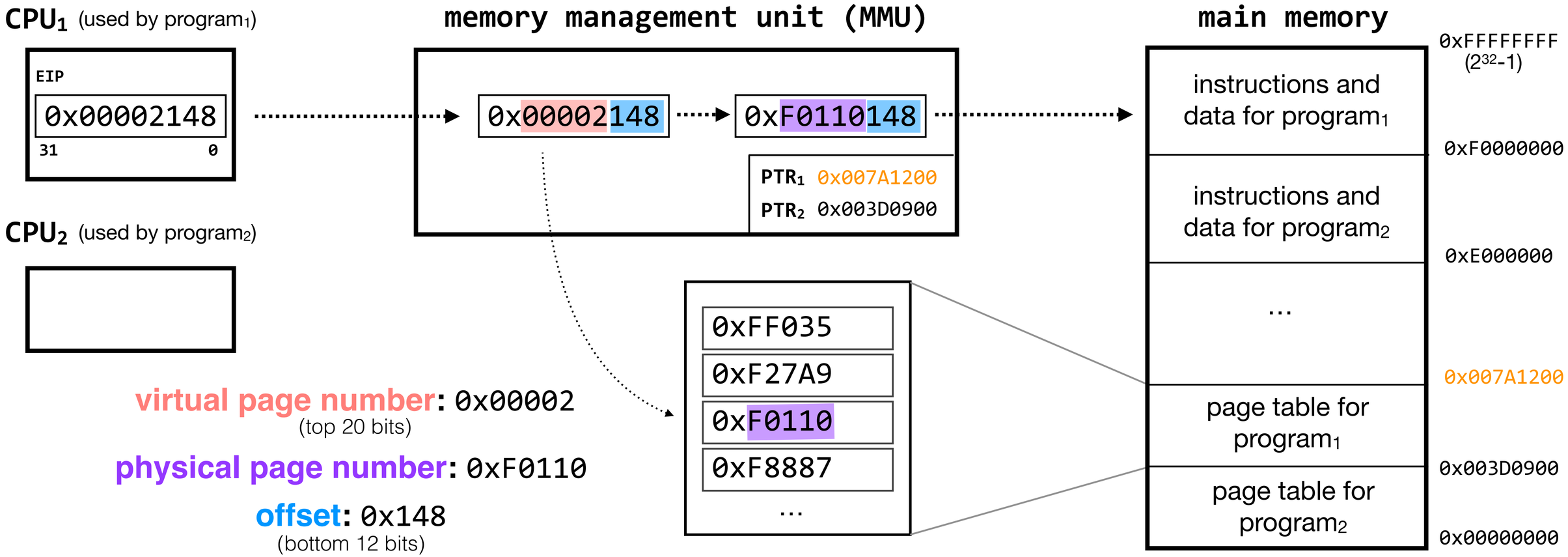
attempt 1: each virtual address acts as an index into this table; there is one entry for every virtual address

2^{32} virtual addresses each mapping to a 32-bit physical address → **16GB to store this table**

we don't even have 16GB of memory

what we want: every program to be able to access a full 32-bit address space

what we have: 2^{32} bytes of memory; every program can't *actually* have access to the full 32-bit space



virtual page number: 0x00002
(top 20 bits)

physical page number: 0xF0110

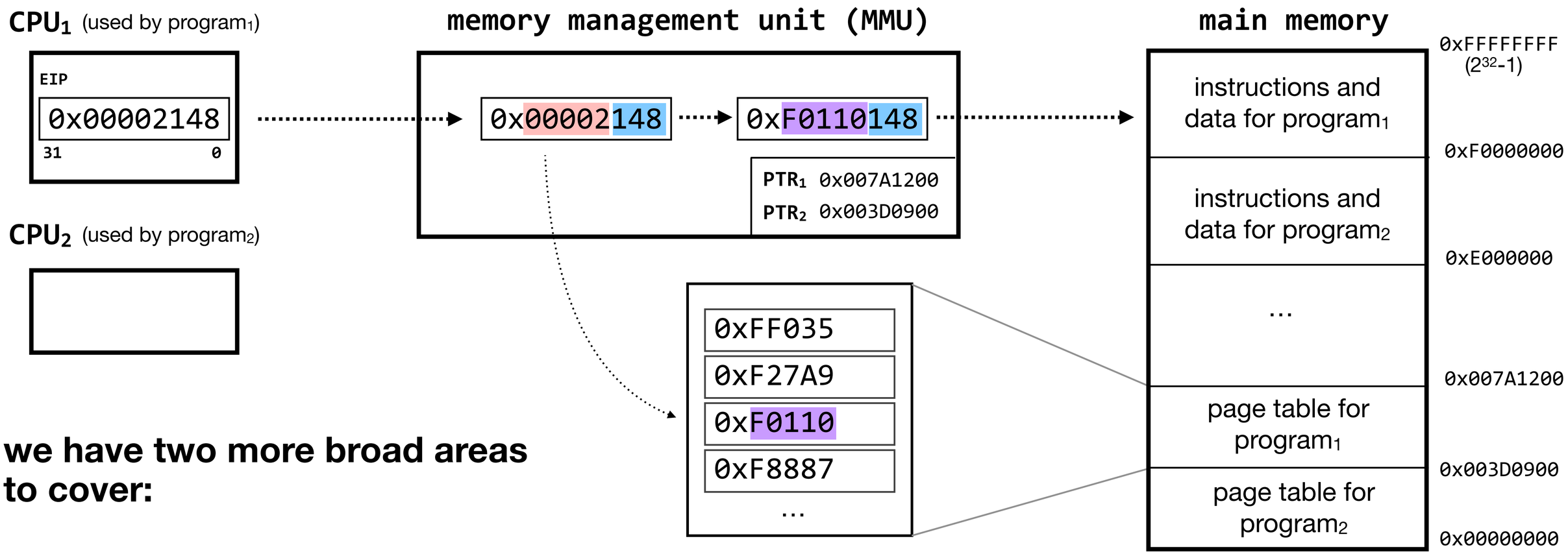
offset: 0x148
(bottom 12 bits)

page tables: top 20 bits of the virtual address act as an index into this table

(a *page* of memory is $2^{32-20}=2^{12}$ bytes)

2^{20} virtual page numbers each mapping to a 32-bit page-table entry (PTE) → **4MB to store this table**

(why 32-bit PTEs, not 20-bit? hang on)



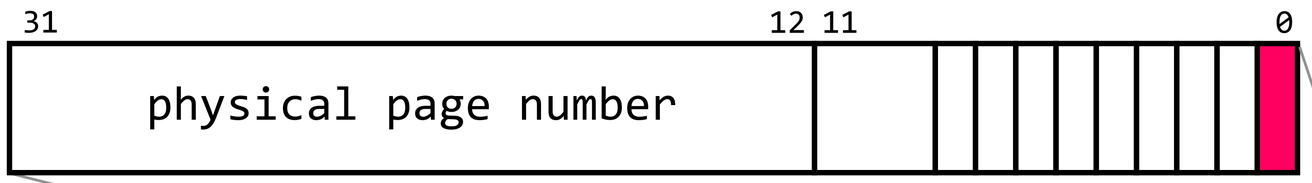
we have two more broad areas to cover:

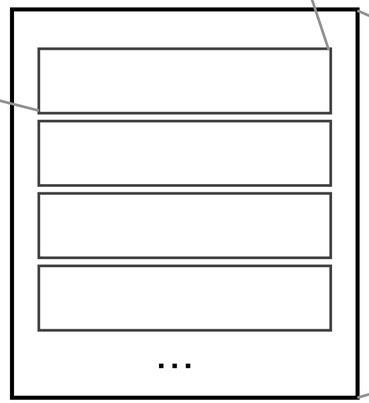
does virtual memory protect programs from accessing each other's memory?
(to answer this, we'll need to address some other issues first)

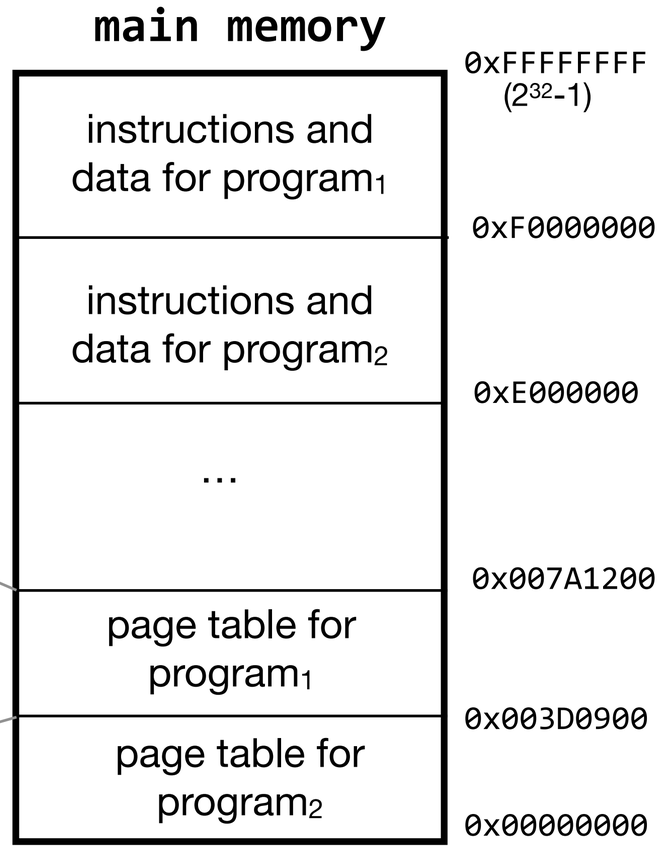
what performance issues matter here?

what happens if we don't have enough memory to store all of our programs' instructions and data?

page table entries contain additional bits that help us deal with this problem (and others)







present (P) bit: is the page currently in memory?

if the page is not in memory, the access triggers an **exception** (known a "page fault" in this case), which the OS handles.

this also answers the question of why PTEs are 32 bits, not 20: they store information beyond the page number

interlude: handling exceptions

(such as page faults)

this idea will remain relevant, as we are going to find that there are quite a few exceptions for the OS to handle

the operating system's **kernel** manages page faults and other **exceptions**

```
// special instruction that calls the exception handler for exception x
exception(x):
  // switch from user mode to kernel mode
  // call the handler for this particular exception
  // switch from kernel mode to user mode
```

interlude: handling exceptions

(such as page faults)

this idea will remain relevant, as we are going to find that there are quite a few exceptions for the OS to handle

the operating system's **kernel** manages page faults and other **exceptions**

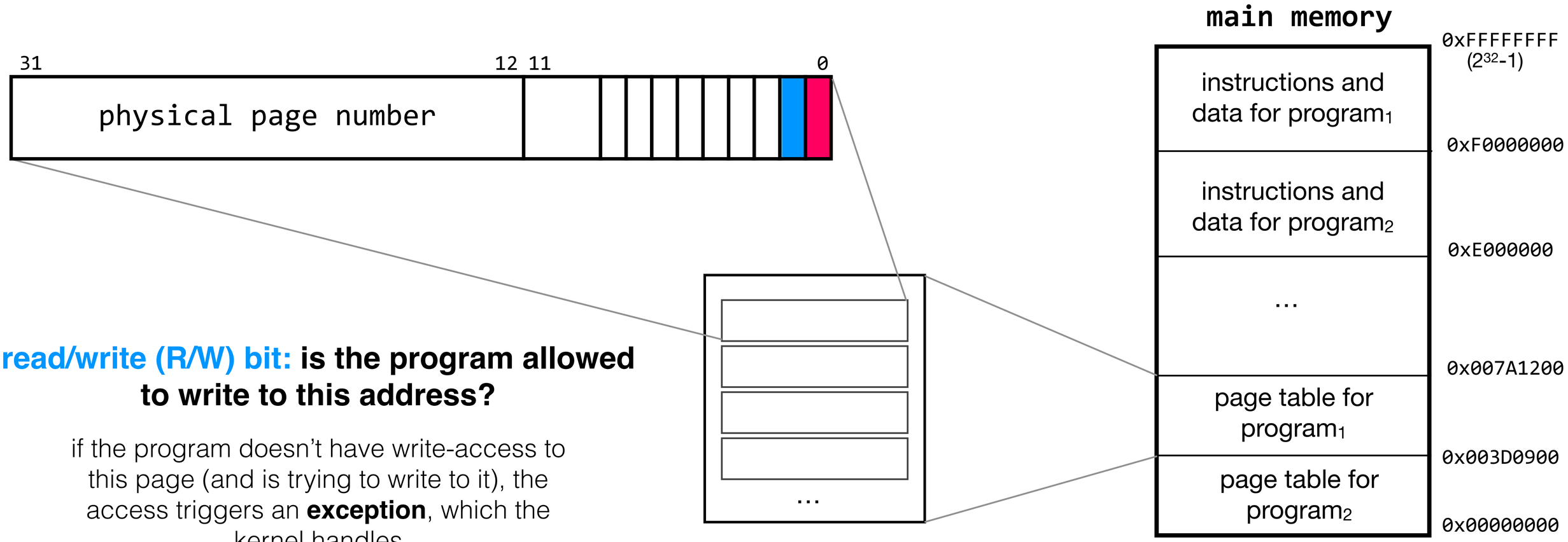
```
// special instruction that calls the exception handler for exception x
exception(x):
  U/K bit = K
  call handlers[x]
  U/K bit = U
```

the processor stores a **user/kernel (U/K) bit**, which indicates whether it's operating in user mode or kernel mode. this bit helps the processor control access to certain kernel-specific actions

each handler is different. as an example, the page-fault handler would take care of bringing the requested page into memory

what happens if a program tries to write to memory that it doesn't have write-access to?

after all, it's conceivable that we want program$_1$ to be able to read some data, but not to modify it

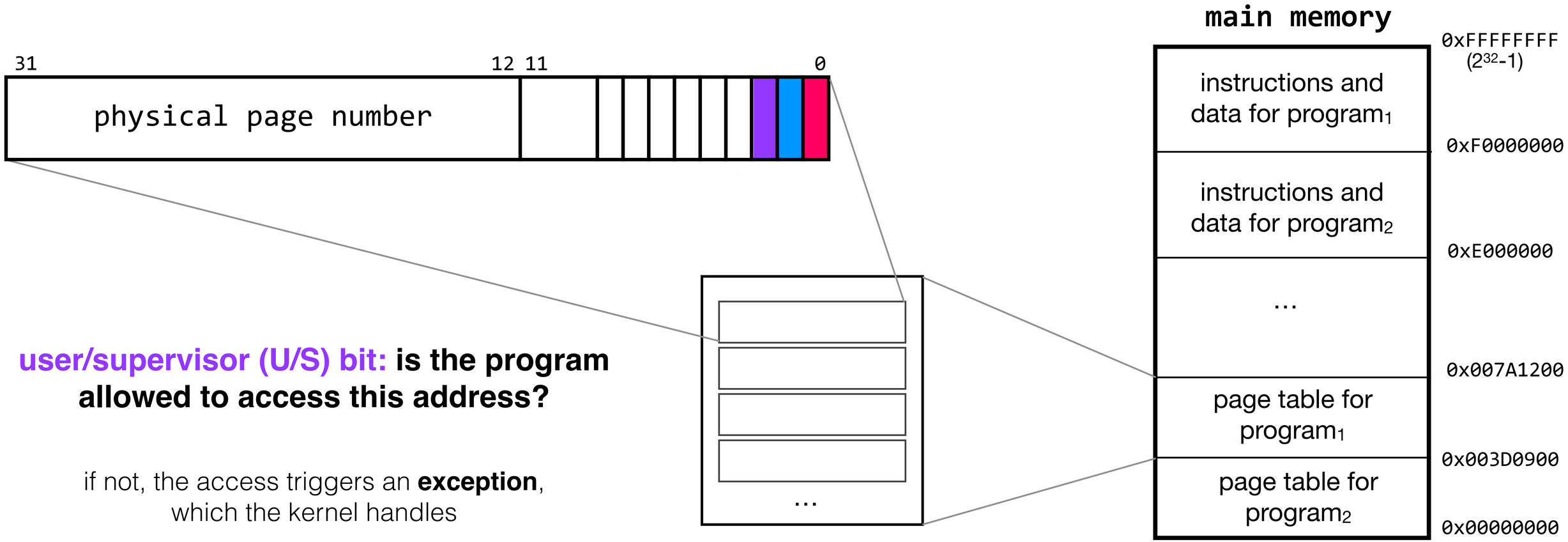


read/write (R/W) bit: is the program allowed to write to this address?

if the program doesn't have write-access to this page (and is trying to write to it), the access triggers an **exception**, which the kernel handles

what happens if a program tries to access memory that only the kernel should have access to?

we need to enforce modularity between programs and the kernel, not just between programs

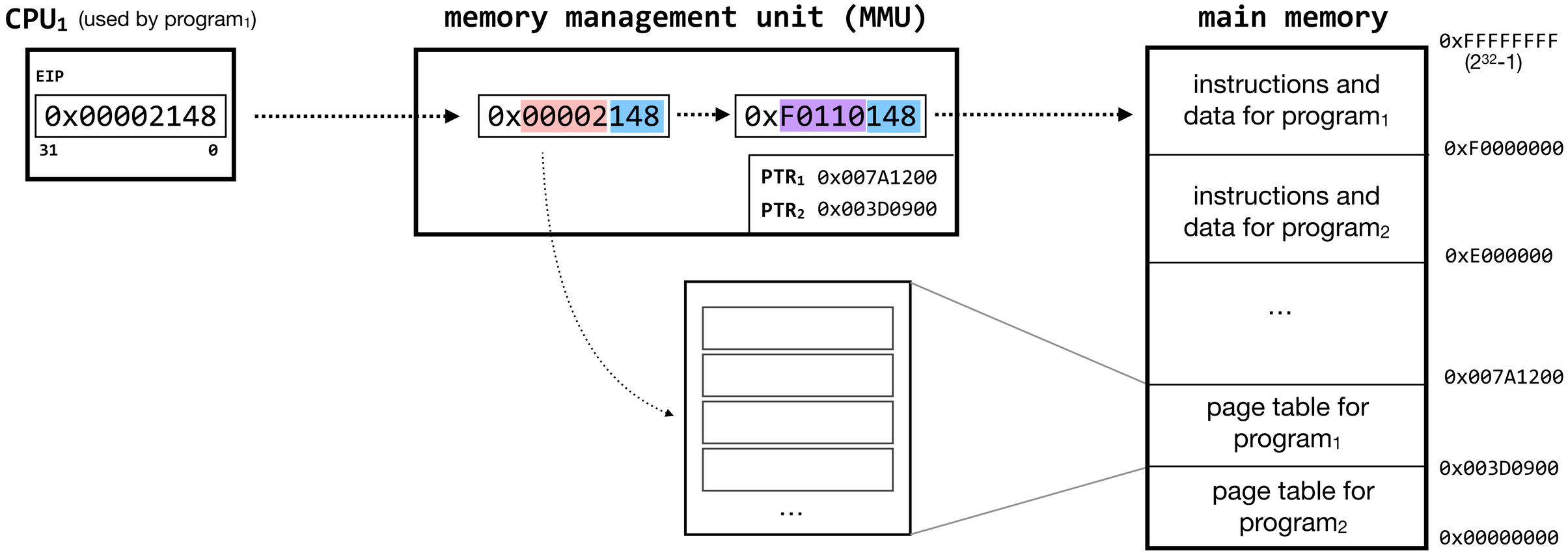


user/supervisor (U/S) bit: is the program allowed to access this address?

if not, the access triggers an **exception**, which the kernel handles

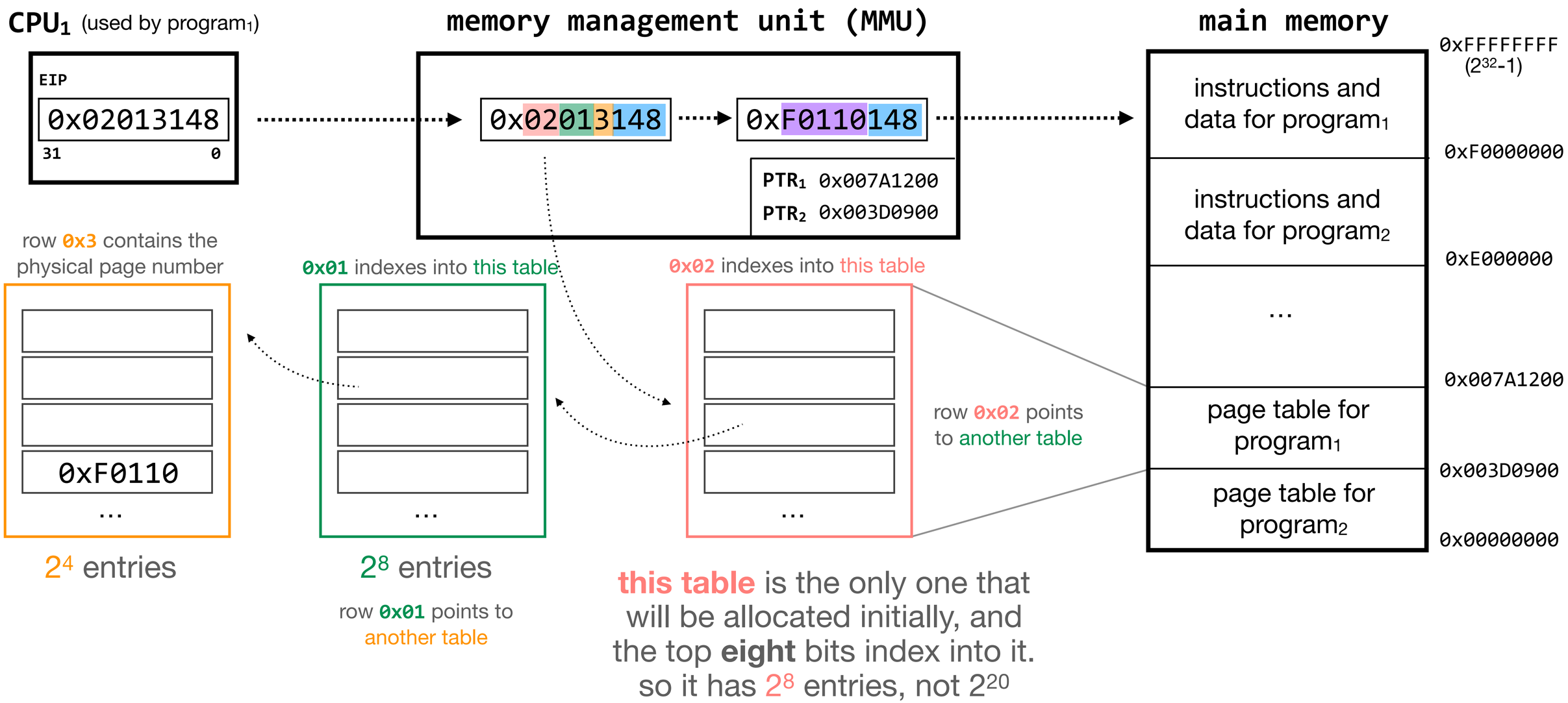
without this last piece, a determined program could still attempt to circumvent modularity by doing things such as modifying the page-table registers

performance issue #1: page tables are allocated contiguously in memory so that access into them is extremely fast; this means that *every* page table is 4MB, even if the program only need to make a few memory accesses



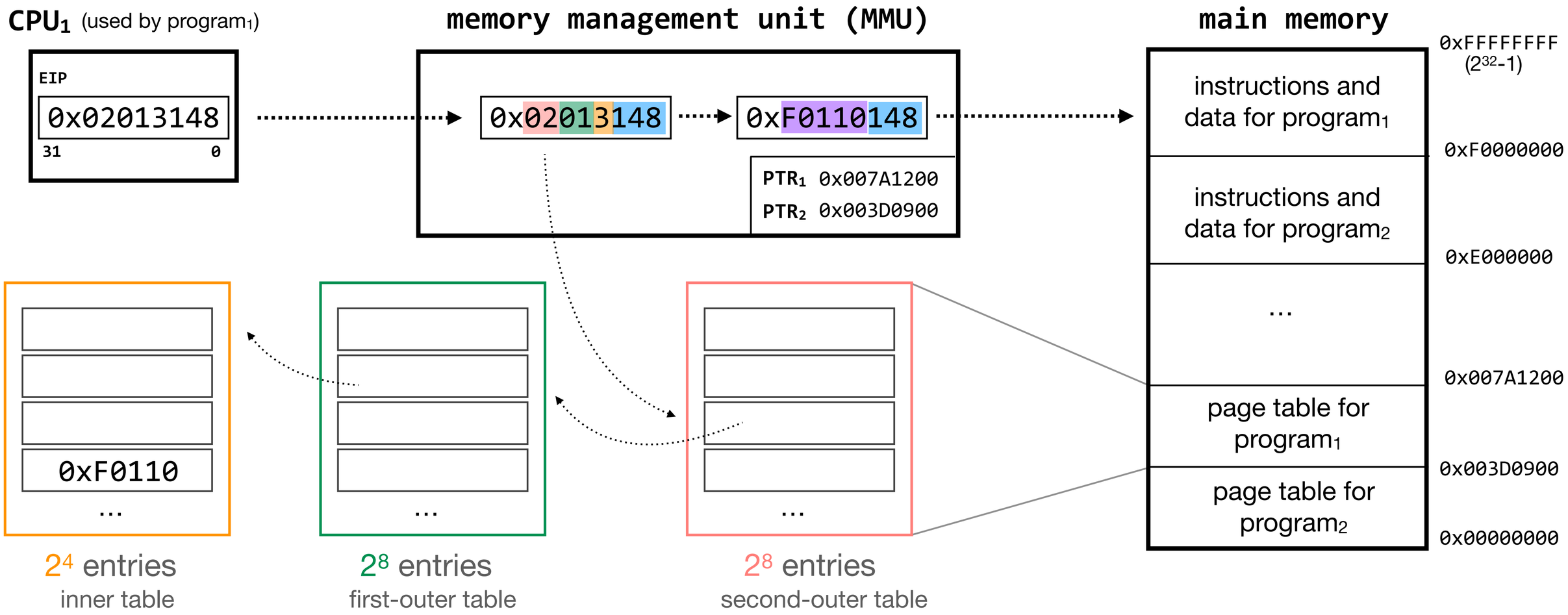
2^{20} virtual addresses each mapping to a 32-bit page-table entry (PTE)
→ **4MB to store this table**

hierarchical (or “multilevel”) page tables potentially use less space



(I used 8/8/4 in this example, but you can generalize to M/N/P)

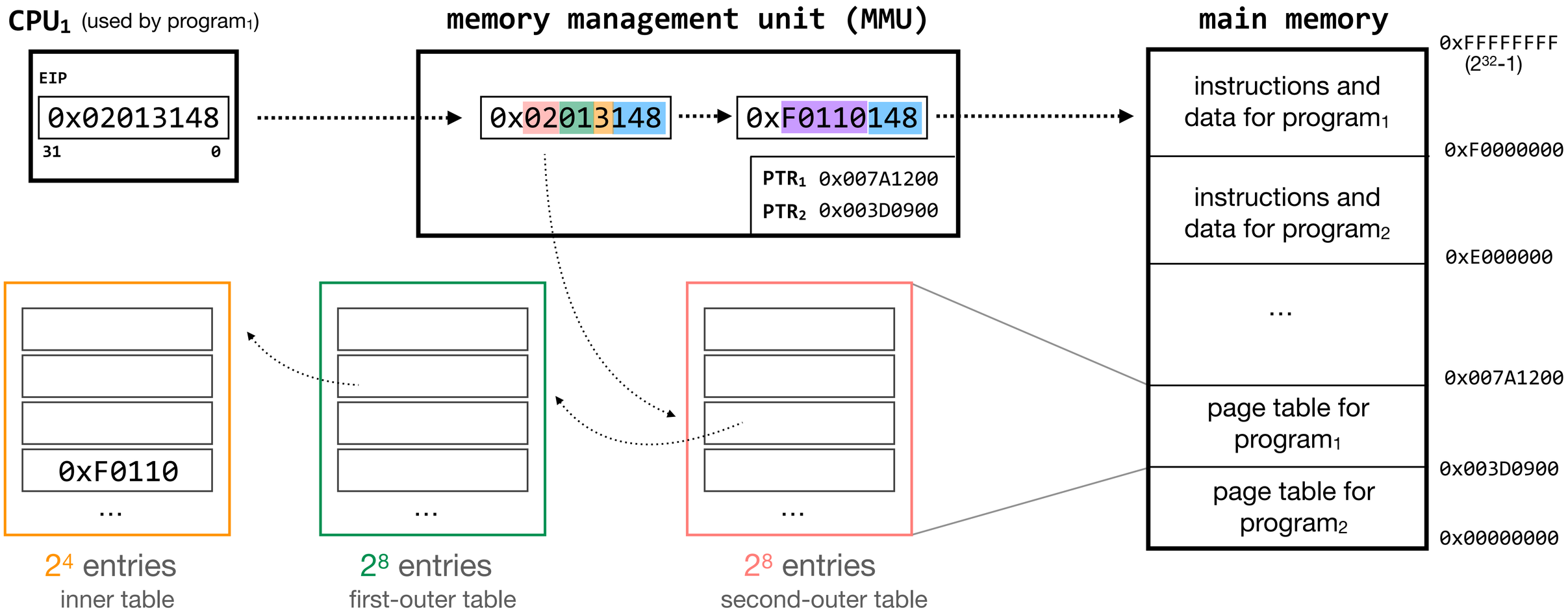
hierarchical (or “multilevel”) page tables potentially use less space, at the expense of more table look-ups and more exceptions (to allocate additional tables)



if the program never accesses a virtual memory address starting with 0x03 (say), no **first-outer table** will be allocated corresponding to row 0x03 in the **second-outer table**

(I used 8/8/4 in this example, but you can generalize to M/N/P)

performance issue #2: looking up the same piece of data over and over again takes time; can we make it faster?



yes. caches are involved in a variety of places here, to (in theory) make common look-ups faster. you've also seen caching in the context of DNS, now.

operating systems enforce modularity on a single machine

in order to enforce modularity + have an effective operating system, a few things need to happen

1. programs shouldn't be able to refer to (and corrupt) each others' **memory** → virtualize **memory**
2. programs should be able to **communicate** with each other → assume they don't need to (for today)
3. programs should be able to **share a CPU** without one program halting the progress of the others → assume one program per CPU (for today)

the primary technique that an operating system uses to enforce modularity is **virtualization**. some components are difficult to virtualize (e.g., the disk); for those, the operating system presents **abstractions**

operating systems enforce modularity on a single machine via **virtualization** and **abstraction**

you'll talk much more about abstractions during the recitations on UNIX; designing good abstractions is part of designing a good operating system

virtualizing memory prevents programs from referring to (and corrupting) each other's memory. the **MMU** translates virtual addresses to physical addresses using **page tables**, and there are a number of **performance issues** to take into account

amount of storage used, speed of access

the **kernel** handles any exceptions triggered in this process; protecting the kernel from user programs is just as important as protecting user programs from each other