6.1800 Spring 2025

Lecture #3: Virtual Memory

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

Katrina LaCurts | lacurts@mit.edu | 6.1800 2025

6.1800 in the news

caveat: this does not appear to be a large-scale measurement study, we should not draw huge conclusions about the performance of Tubi vs. YouTube TV from these results alone

We measured latency against the over-the-air (OTA) broadcast the local San Francisco Fox affiliate KTVU delivered. Although our TV supports NextGen TV and KTVU broadcasts a NextGen signal we stuck with the regular ATSC 1.0 broadcast.[i] Note that the TV broadcast is delayed behind the actual live game.

The Tubi stream was consistently at or slightly ahead of the OTA. The Tubi browser was three or more seconds ahead, while the connected TV app was about a second behind. Even T-Mobile's network delivered slightly ahead of the OTA broadcast. These delays are very small, and stream viewers had no reason to worry about social media posts calling a play before they had seen it.

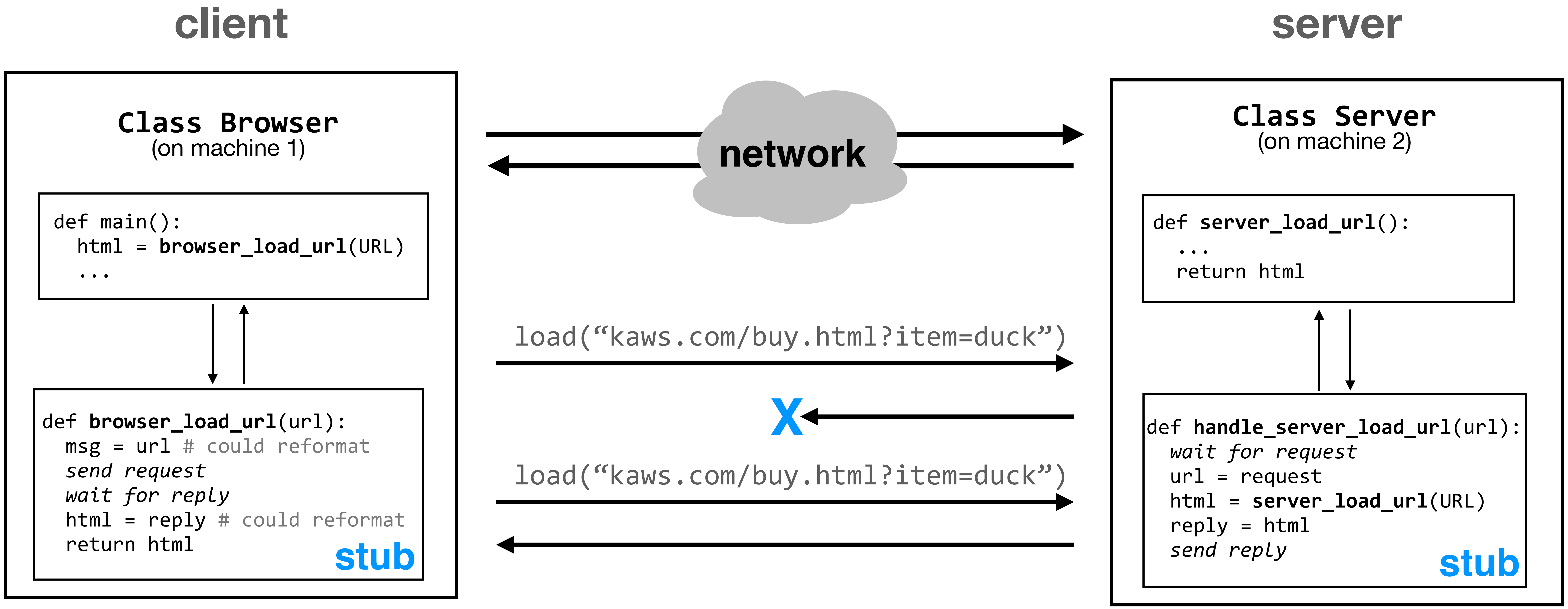
	Tubi	Tubi Mobile	YouTube TV	Tubi Browser	Tubi CTV
1st Quarter					
Delay	-3.2	4.4	35.3	-5.5	
Start Time	2	9.9	1.9	1.7	
2nd Quarter					
Delay	0	1	30.1	-7	
Start Time	7.3	27.9	2	1.5	
3rd Quarter					
Delay	-2.5	-4.5	29.9	-5.5	2
Start Time	7.4	25	2.2	1.7	5.5
4th Quarter					
Delay	-0.5	-2.3	31.4	5	0.5
Start Time	3	17.5	3	7	3

All times are seconds. Delay is relative to local TV over-the-air broadcast

Source: nScreenMedia

On the other hand, YouTube TV viewers needed to be very careful about monitoring their social accounts during the game. The vMVPD consistently delivered almost a full down behind the action on the OTA broadcast. When the Chiefs went into hurry-up offense in the fourth quarter, the YouTube TV stream was almost two downs behind at one point!

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)

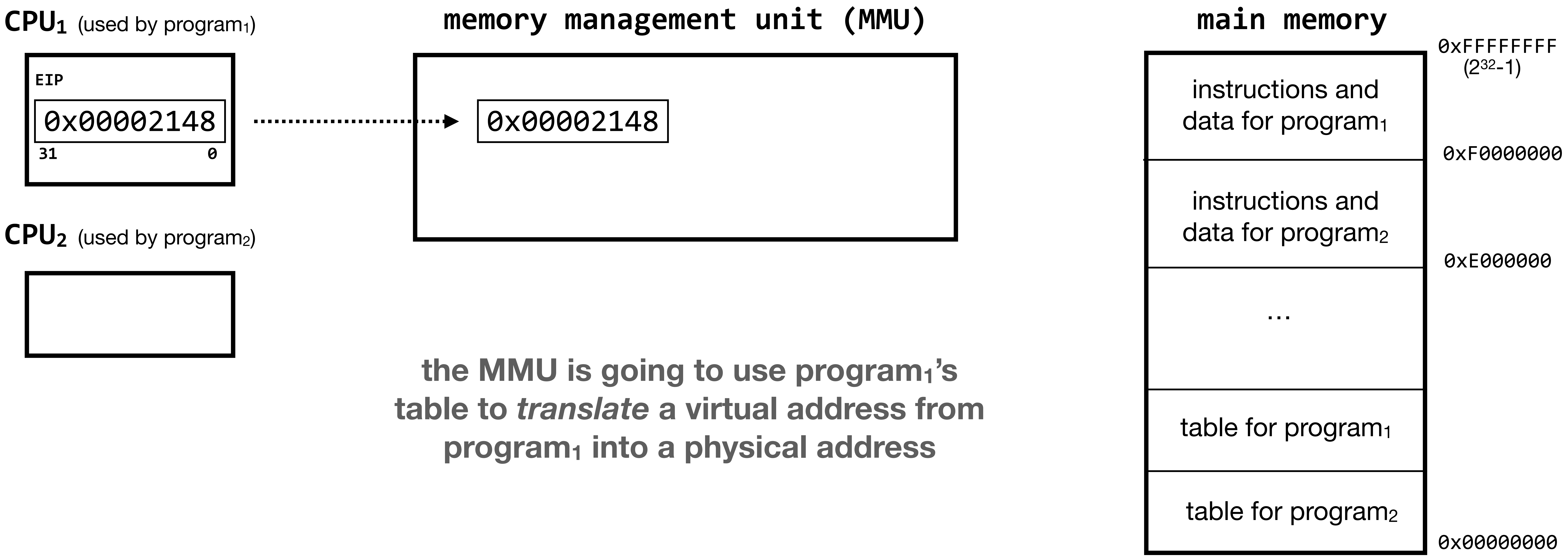
3. programs should be able to **share a CPU** without one program halting the progress of the others → assume one program per CPU (for today)

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: **virtualization.** every program should appear to have access to a full 32-bit address space

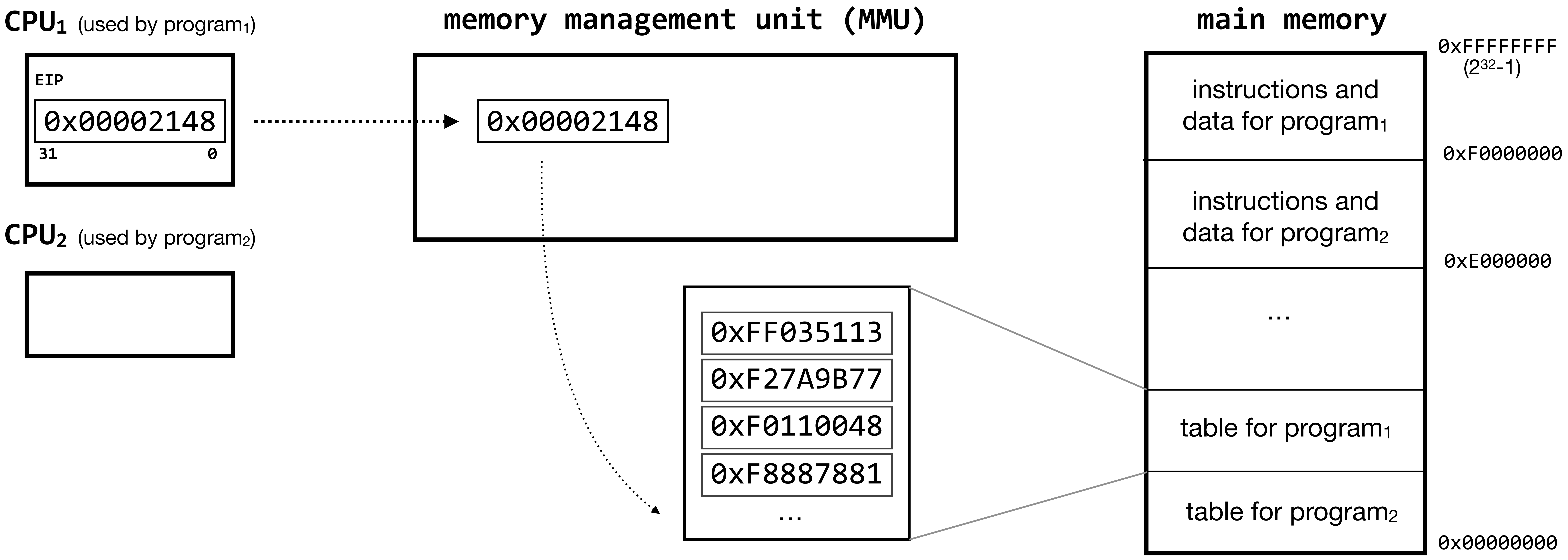
what we have: 2^{32} bytes of memory; every program can't *actually* have access to the full 32-bit space



the MMU is going to use program$_1$'s table to *translate* a virtual address from program$_1$ into a physical address

what we want: **virtualization.** every program should appear to have access to 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



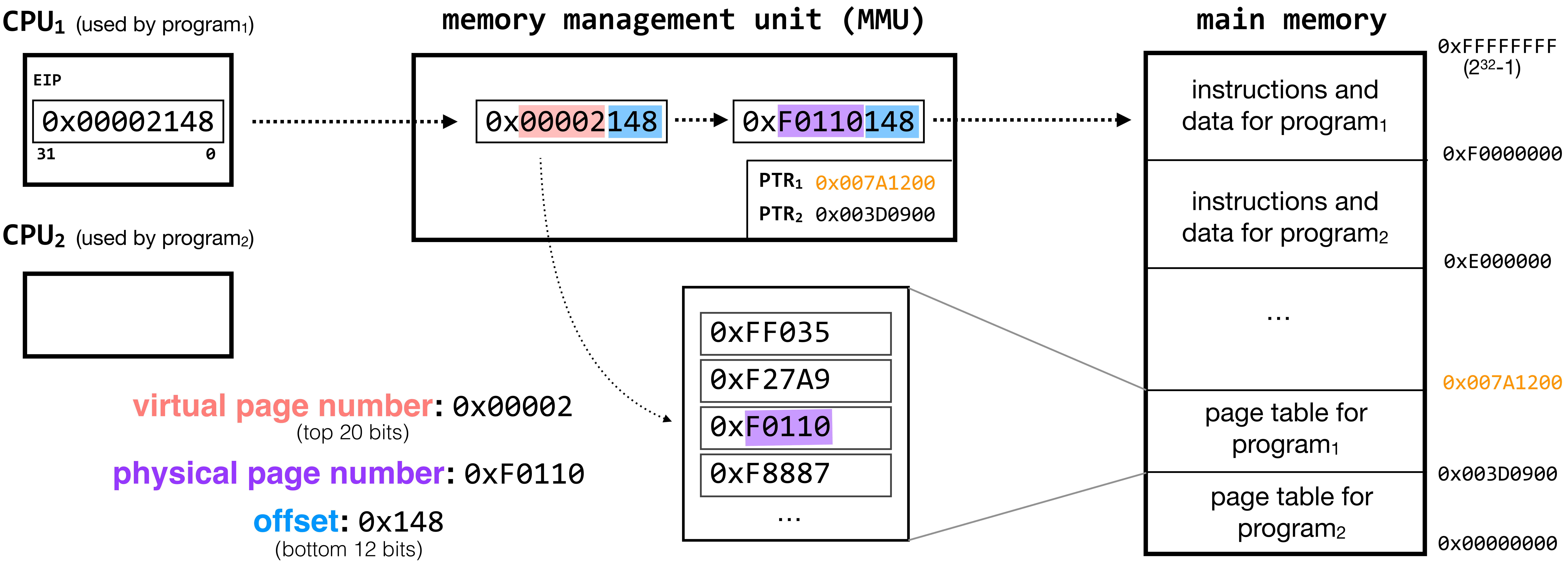
attempt 1: each virtual address acts as an index into this table; there is one entry for every virtual address

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

16GB is *quite a lot* of memory

what we want: **virtualization.** every program should appear to have access to 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



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

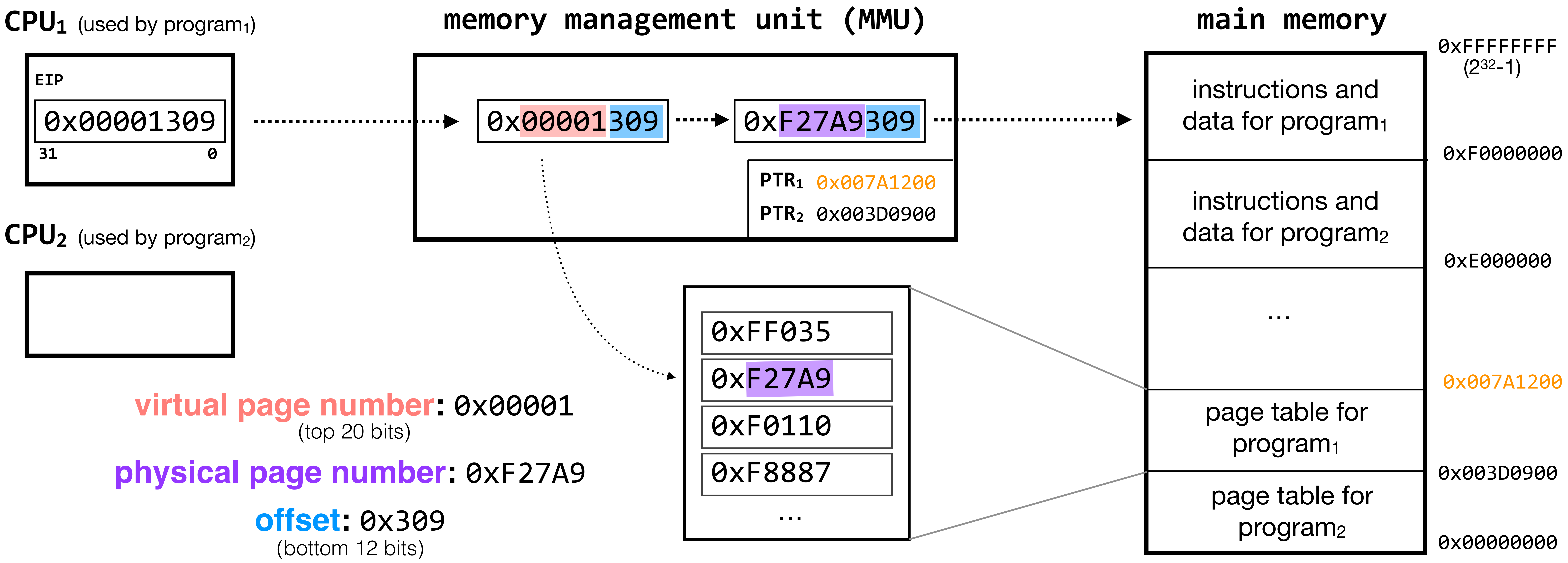
(a *page* of memory is $2^{32-20}=2^{12}$ bytes)

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)

what we want: virtualization. every program should appear to have access to 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

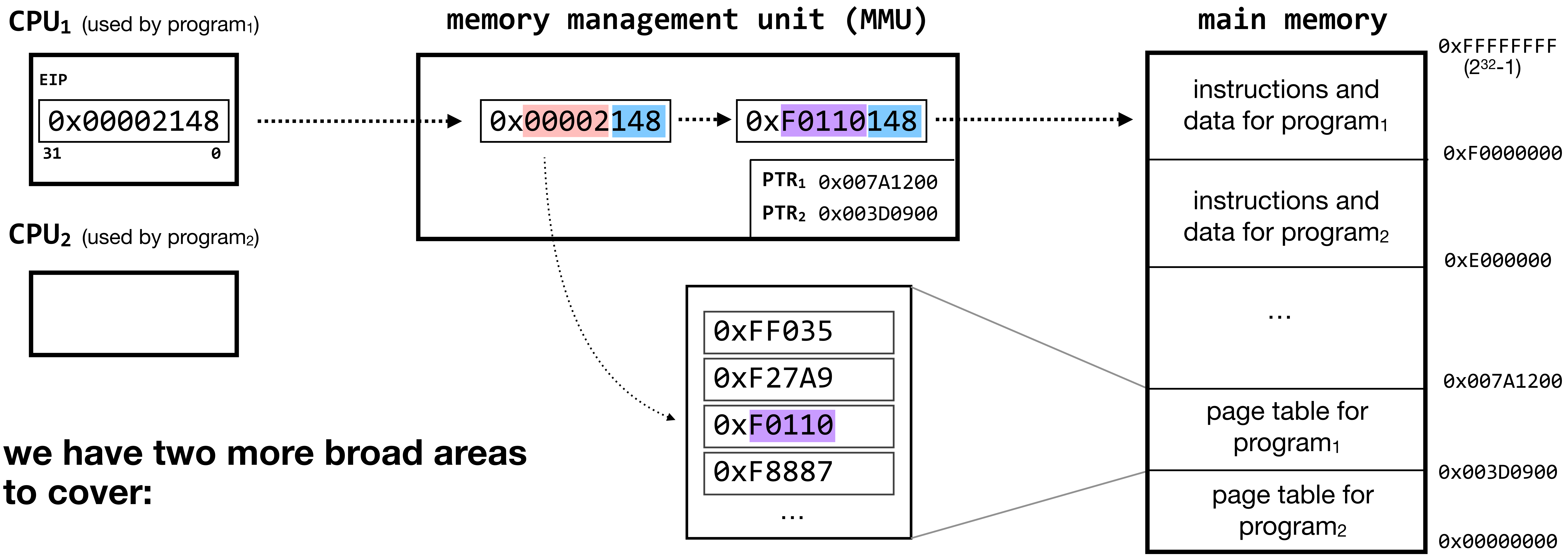


page tables: top 20 bits of the virtual address act as an index into this table

(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:

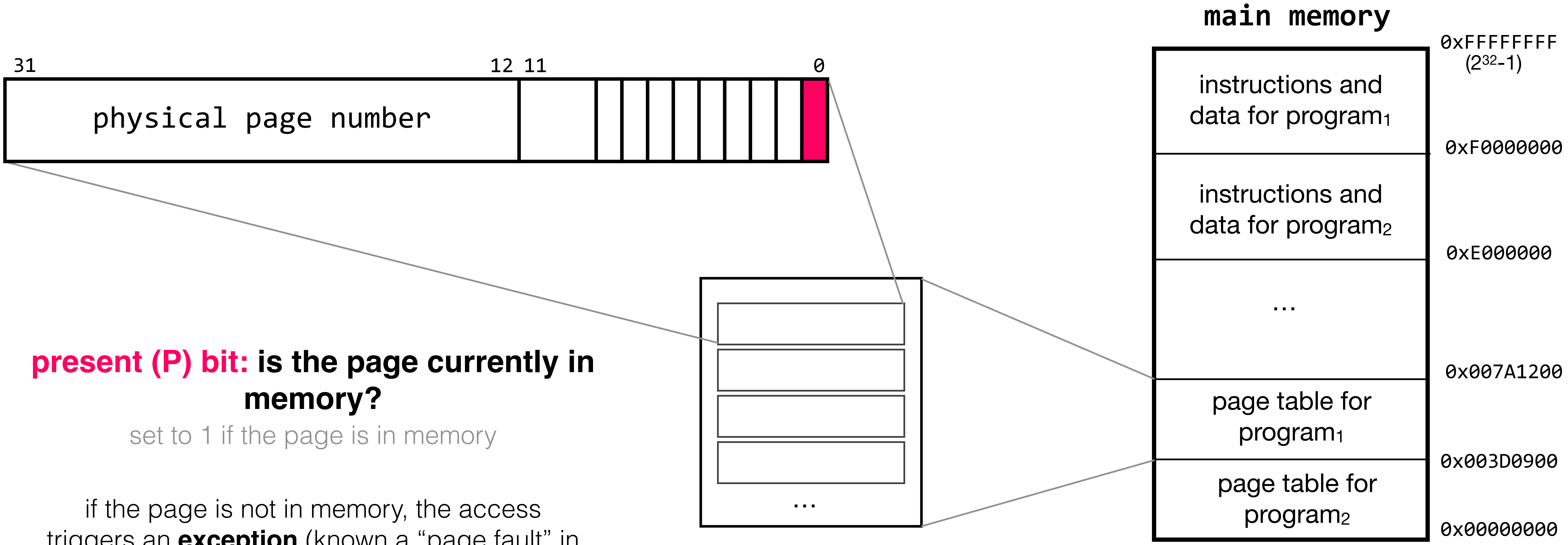
does virtual memory protect programs from accessing each other's memory?

(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?

set to 1 if the page is in memory

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

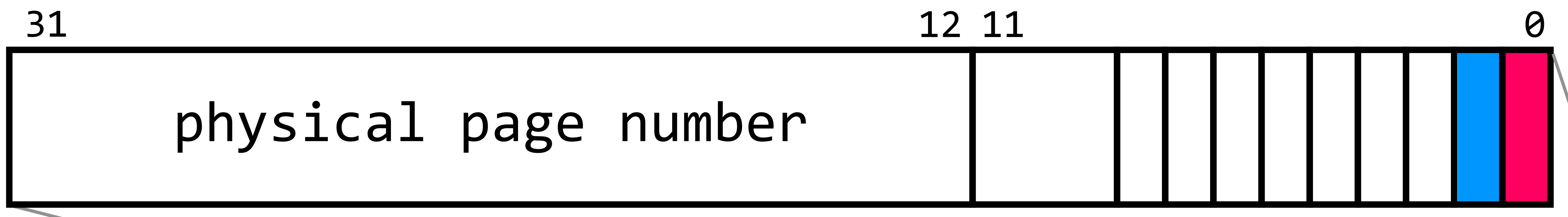
```
// special instruction that calls the exception handler for exception x
exception(x):
  U/K bit = K
  call handlers[x]
  U/K bit = U
```

the processor stores a **user/kernel (U/K) bit** that indicates whether its 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?

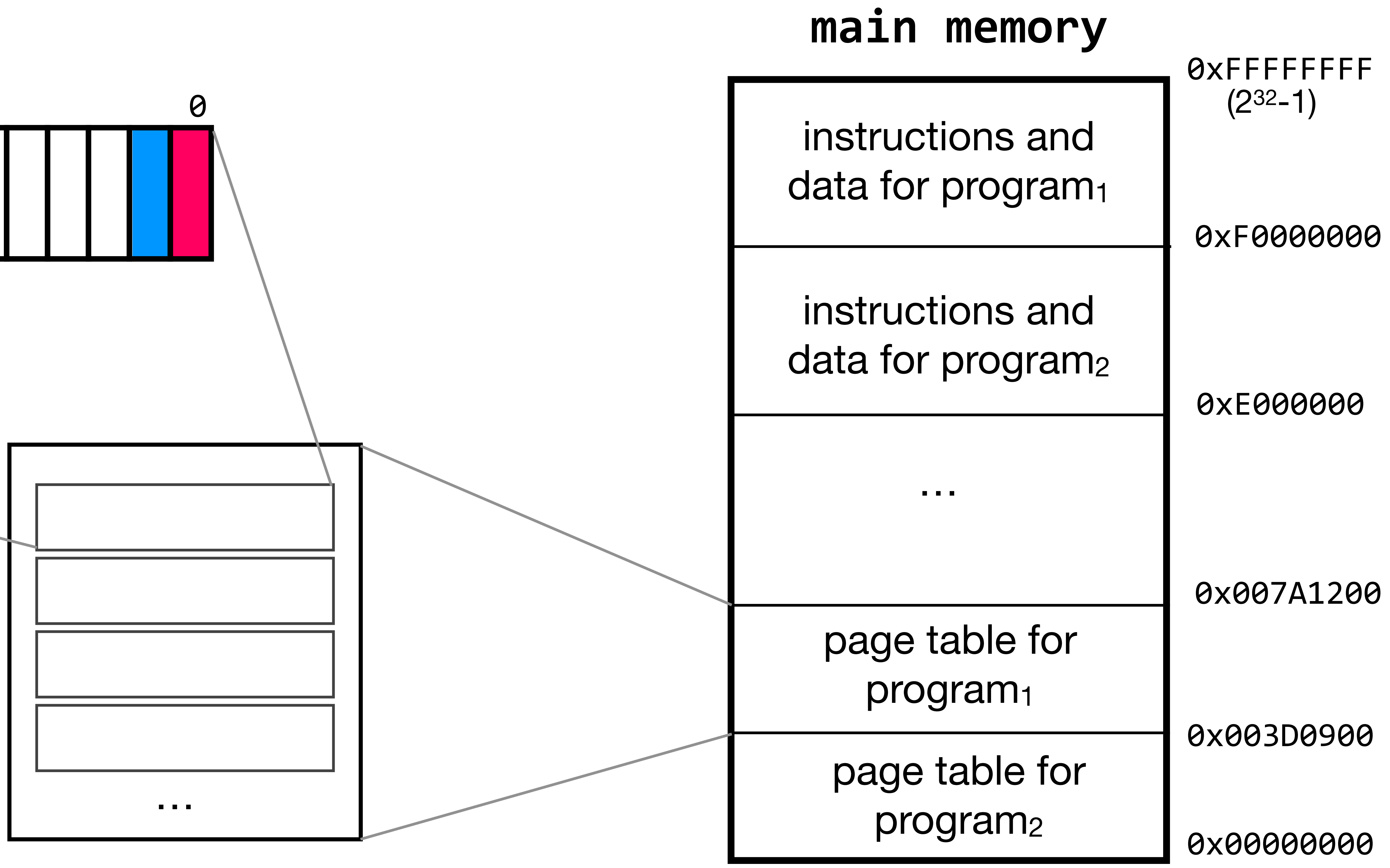
after all, it's conceivable that we want program$_1$ to be able to read some data, but not to modify it



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

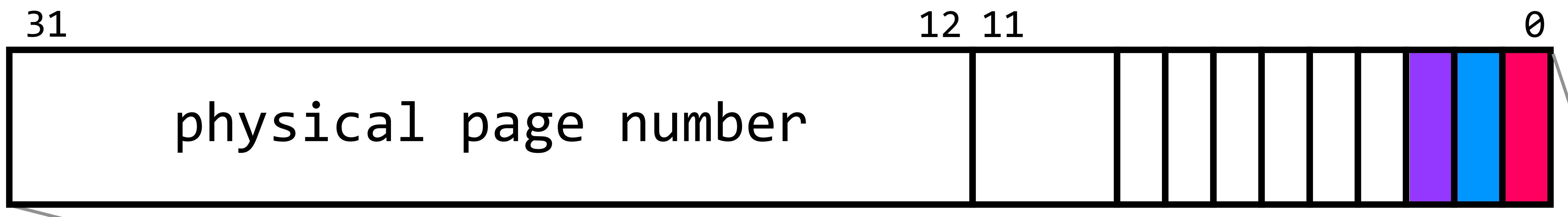
set to 1 if the program can write to this address

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?

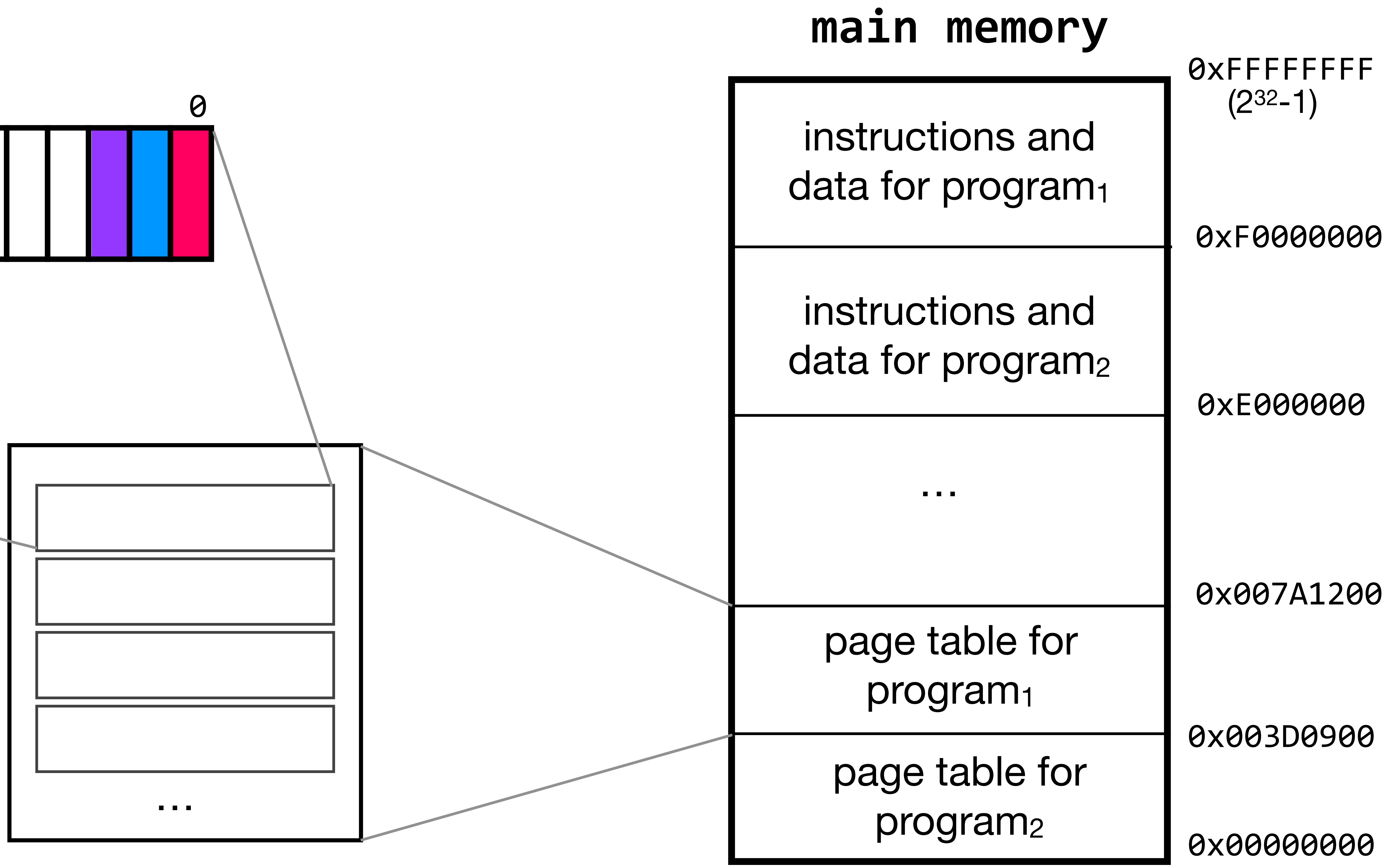
we need to enforce modularity between programs and the kernel, not just between programs



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

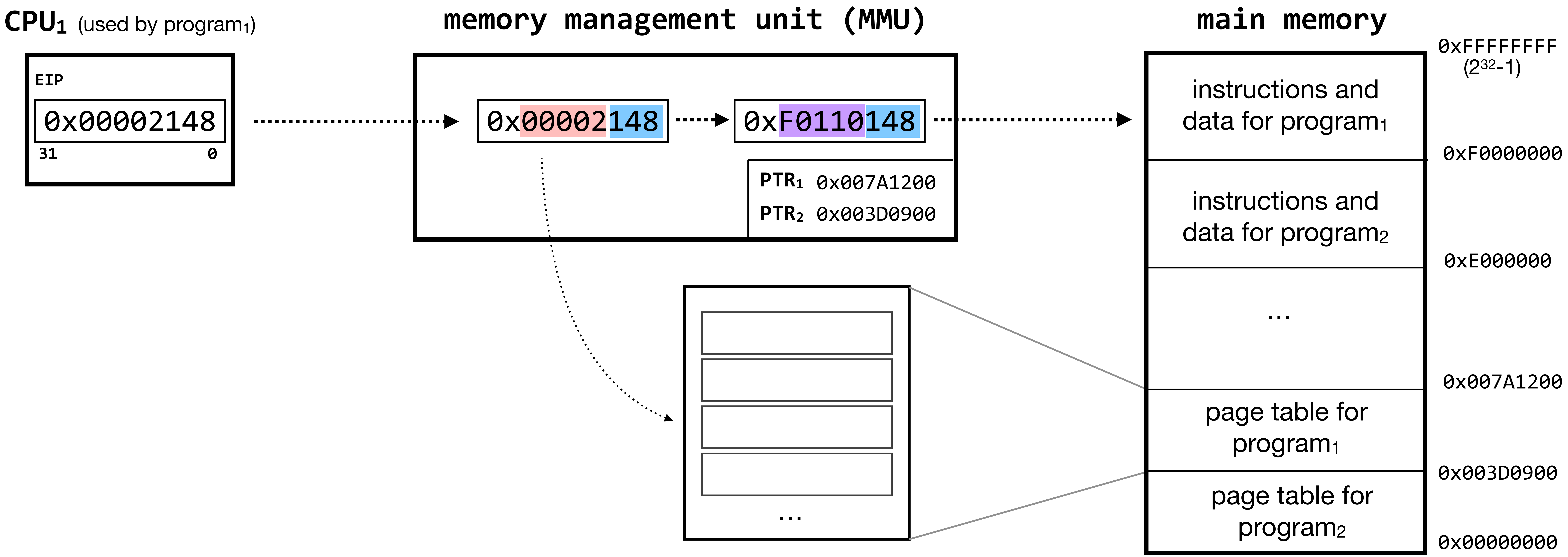
set to 1 if the program can access this address

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 needs to make a few memory accesses



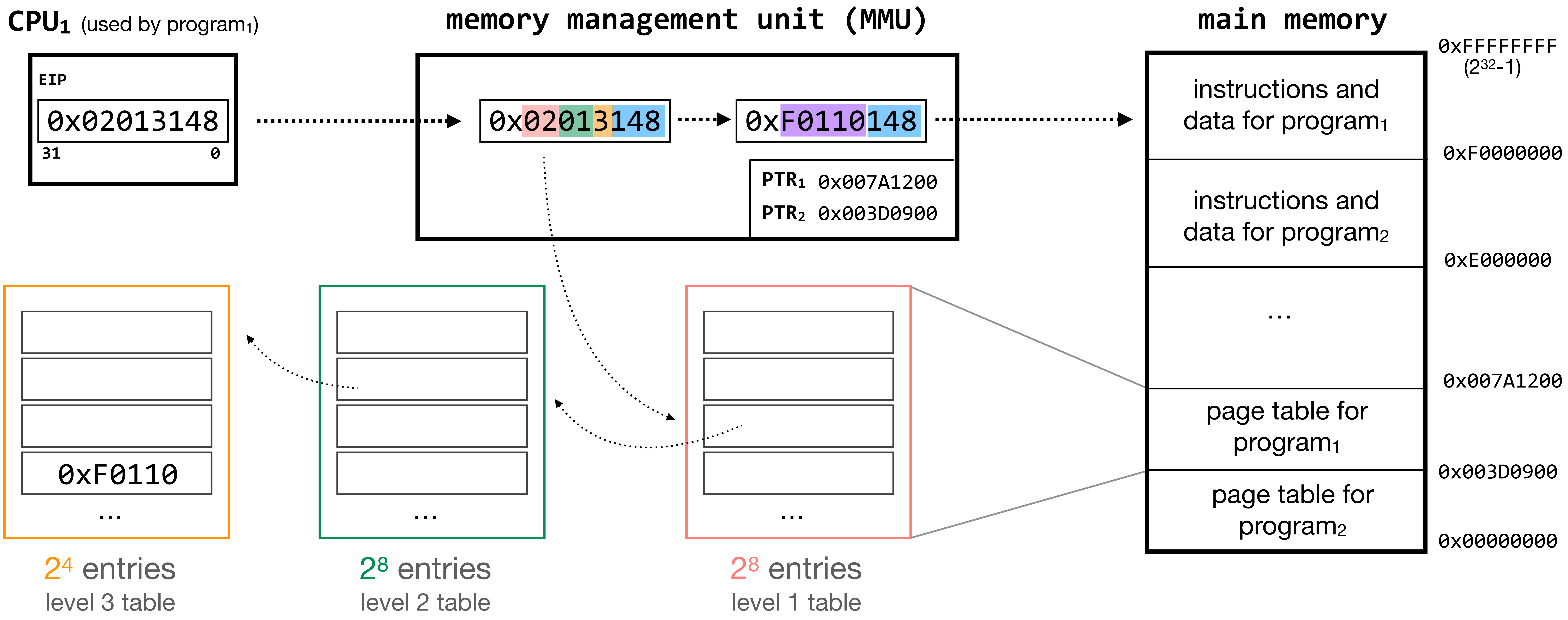
2^{20} virtual addresses each mapping to a 32-bit page-table entry (PTE)
→ **4MB to store this table**

multilevel page tables often use less space

this **level 1 table** is the only one that will be allocated initially, and the top **eight** bits index into it. so it has 2^8 entries, not 2^{20}

(we're using 8/8/4 in this example, but you can generalize to M/N/P)

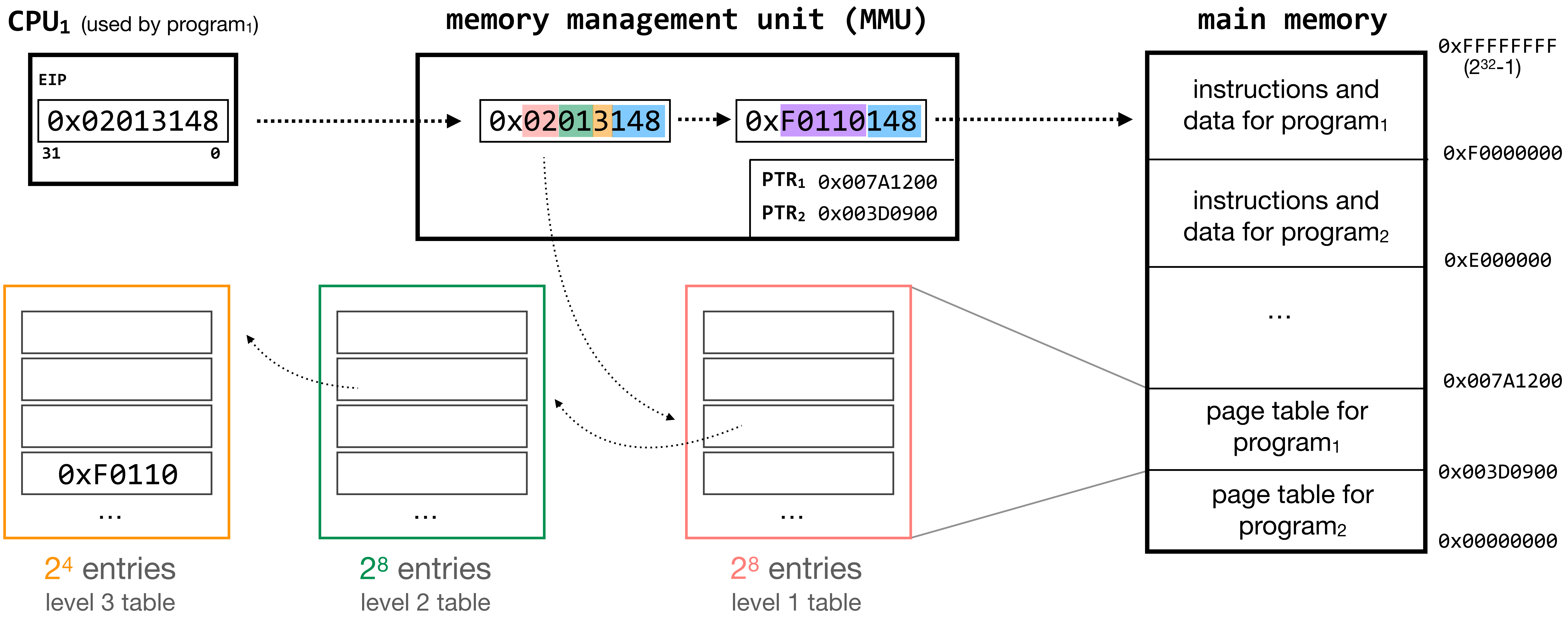
multilevel page tables often use less space, at the expense of more table look-ups and more exceptions (to allocate additional tables)



each row in the **level 1 table** (typically) corresponds to a different **level 2 table**, but each **level 2 table** (and **level 3 table**) is allocated as needed

(we're using 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.

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 memory 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