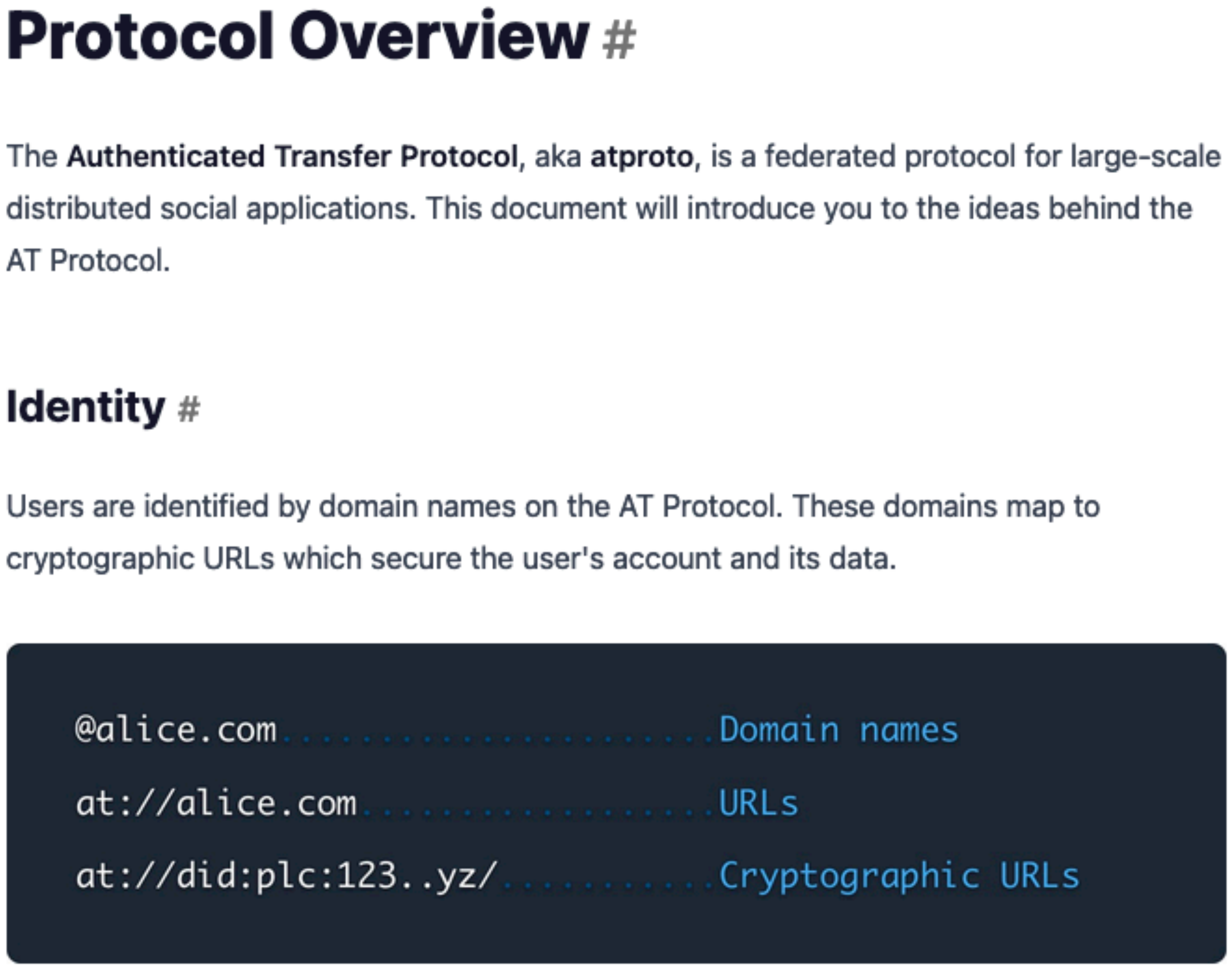
6.1800 Spring 2024

Lecture #3: Virtual Memory

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

6.1800 in the news



Protocol Overview

The **Authenticated Transfer Protocol**, aka **atproto**, is a federated protocol for large-scale distributed social applications. This document will introduce you to the ideas behind the AT Protocol.

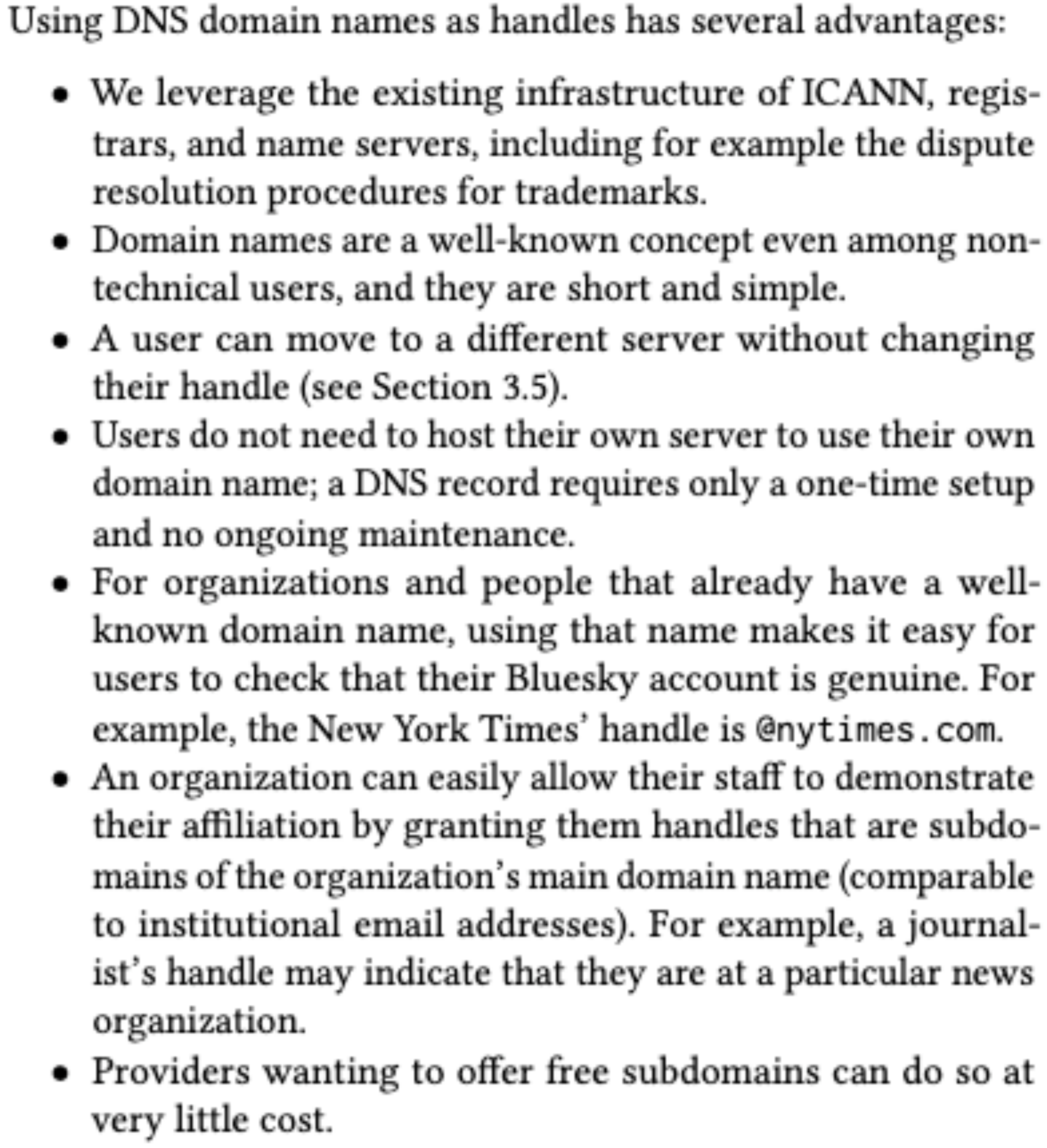
Identity

Users are identified by domain names on the AT Protocol. These domains map to cryptographic URLs which secure the user's account and its data.

```
@alice.com.....................Domain names
at://alice.com.................URLs
at://did:plc:123..yz/..........Cryptographic URLs
```

https://atproto.com/guides/overview

6.1800 in the news

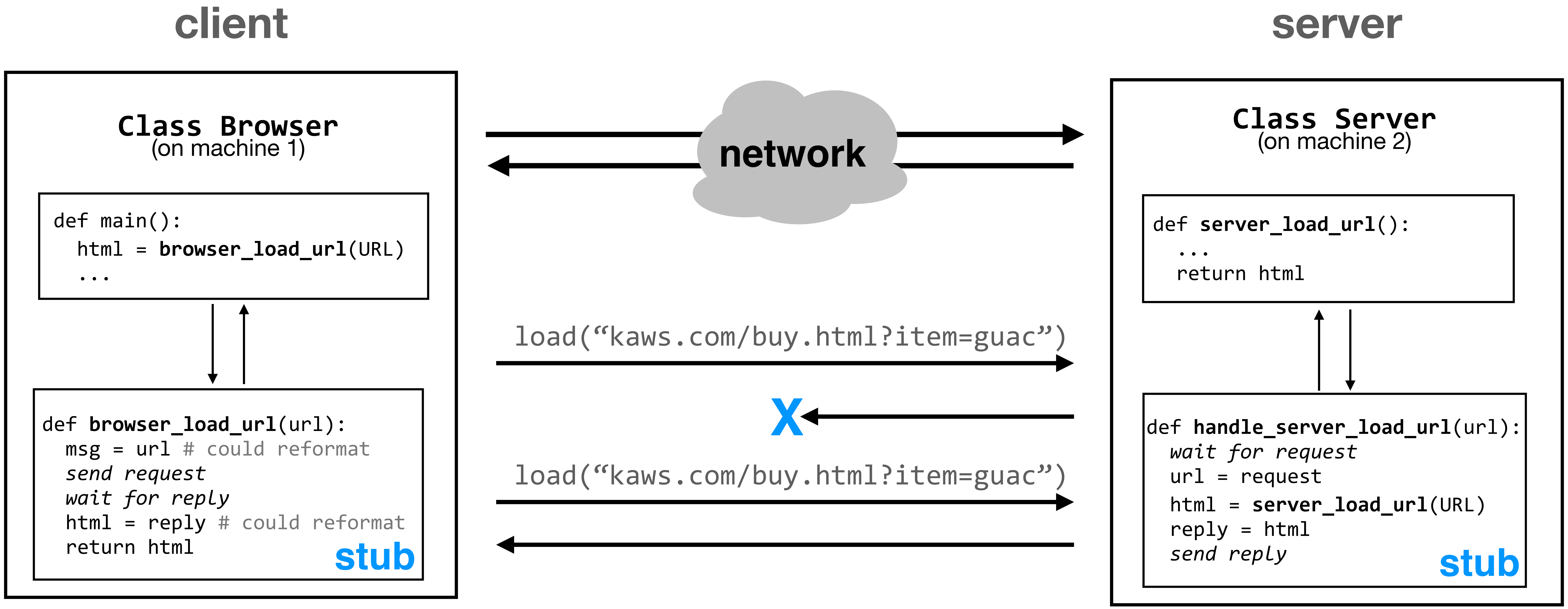


Using DNS domain names as handles has several advantages:

- We leverage the existing infrastructure of ICANN, registrars, and name servers, including for example the dispute resolution procedures for trademarks.
- Domain names are a well-known concept even among non-technical users, and they are short and simple.
- A user can move to a different server without changing their handle (see Section 3.5).
- Users do not need to host their own server to use their own domain name; a DNS record requires only a one-time setup and no ongoing maintenance.
- For organizations and people that already have a well-known domain name, using that name makes it easy for users to check that their Bluesky account is genuine. For example, the New York Times' handle is `@nytimes.com`.
- An organization can easily allow their staff to demonstrate their affiliation by granting them handles that are subdomains of the organization's main domain name (comparable to institutional email addresses). For example, a journalist's handle may indicate that they are at a particular news organization.
- Providers wanting to offer free subdomains can do so at very little cost.

https://arxiv.org/pdf/2402.03239.pdf

last time: enforced modularity via client/server + naming


client
server
network
load("kaws.com/buy.html?item=guac")
X
load("kaws.com/buy.html?item=guac")
stub
stub

Class Browser
(on machine 1)

```
def main():
  html = browser_load_url(URL)
  ...
```

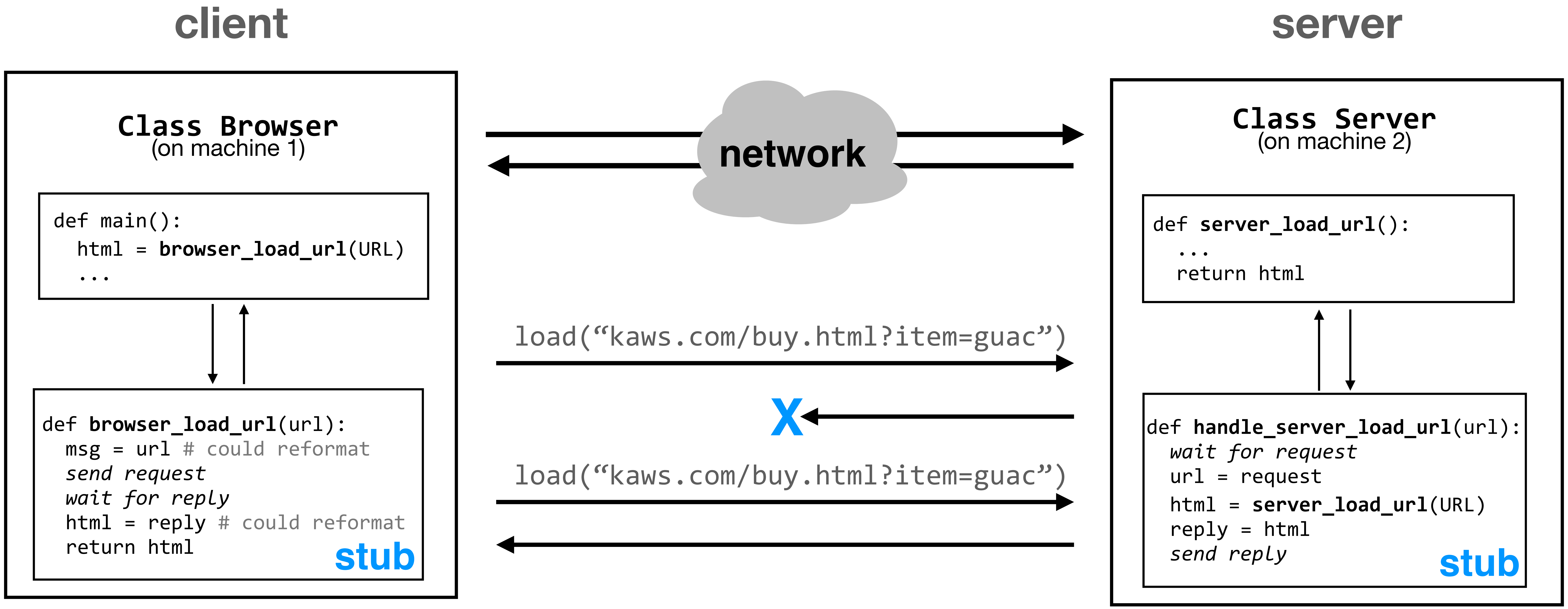
```
def browser_load_url(url):
  msg = url # could reformat
  send request
  wait for reply
  html = reply # could reformat
  return html
```

Class Server
(on machine 2)

```
def server_load_url():
  ...
  return html
```

```
def handle_server_load_url(url):
  wait for request
  url = request
  html = server_load_url(URL)
  reply = html
  send reply
```

last time: enforced modularity via client/server + naming



Class Browser
(on machine 1)

```
def main():
  html = browser_load_url(URL)
  ...
```

```
def browser_load_url(url):
  msg = url # could reformat
  send request
  wait for reply
  html = reply # could reformat
  return html
```

Class Server
(on machine 2)

```
def server_load_url():
  ...
  return html
```

```
def handle_server_load_url(url):
  wait for request
  url = request
  html = server_load_url(URL)
  reply = html
  send reply
```

load(“kaws.com/buy.html?item=guac”)

load(“kaws.com/buy.html?item=guac”)

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

operating systems enforce modularity on a single 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

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**

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**

2. programs should be able to **communicate** with each other

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**
2. programs should be able to **communicate** with each other
3. programs should be able to **share a CPU** without one program halting the progress of the others

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**

2. programs should be able to **communicate** with each other

3. programs should be able to **share a CPU** without one program halting the progress of the others

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

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**

2. programs should be able to **communicate** with each other

3. programs should be able to **share a CPU** without one program halting the progress of the others

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

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 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

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

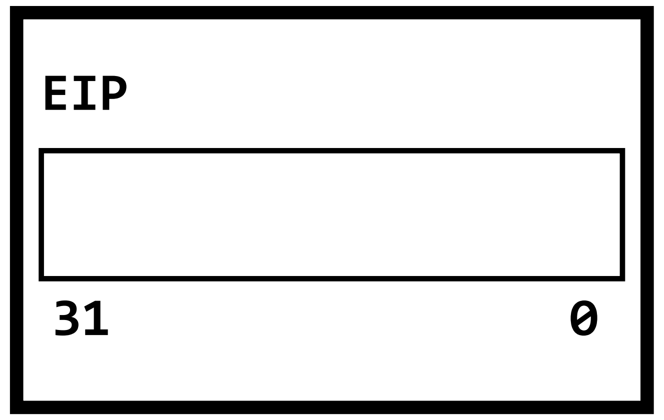
what we have: 2^{32} bytes of memory; every program can't *actually* have access to the full 32-bit space

CPU$_1$ (used by program$_1$)

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

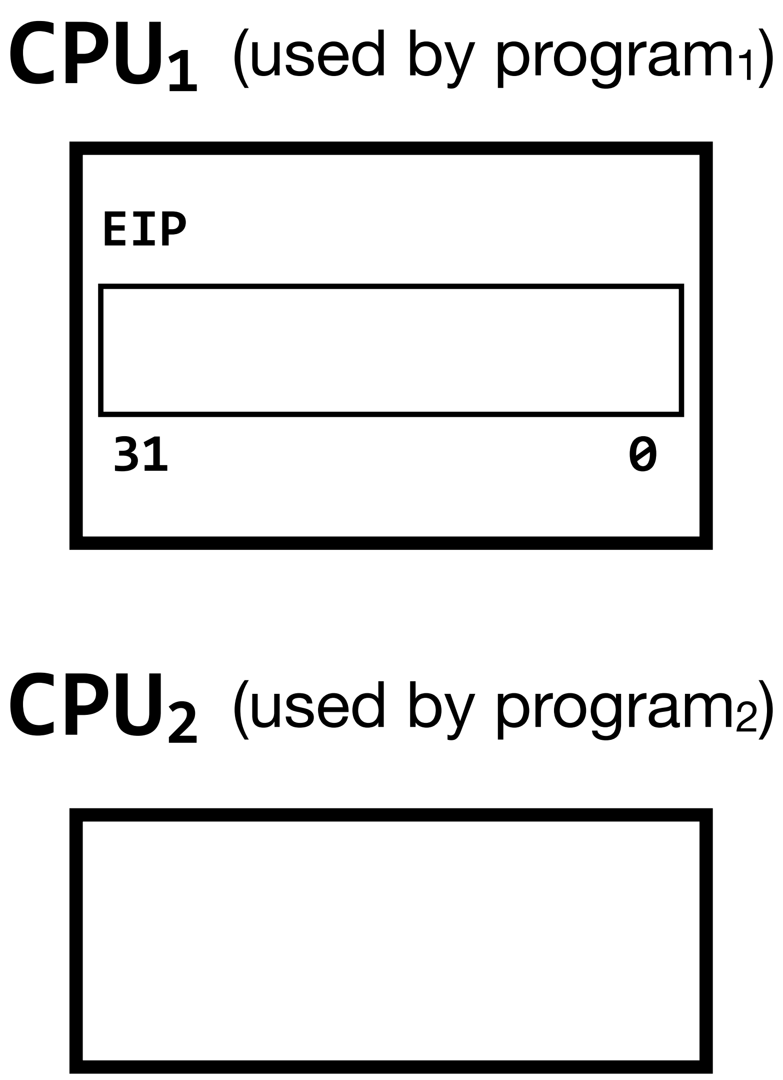
what we have: 2^{32} bytes of memory; every program can't *actually* have access to the full 32-bit space

CPU$_1$ (used by program$_1$)



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



CPU$_1$ (used by program$_1$)

CPU$_2$ (used by program$_2$)

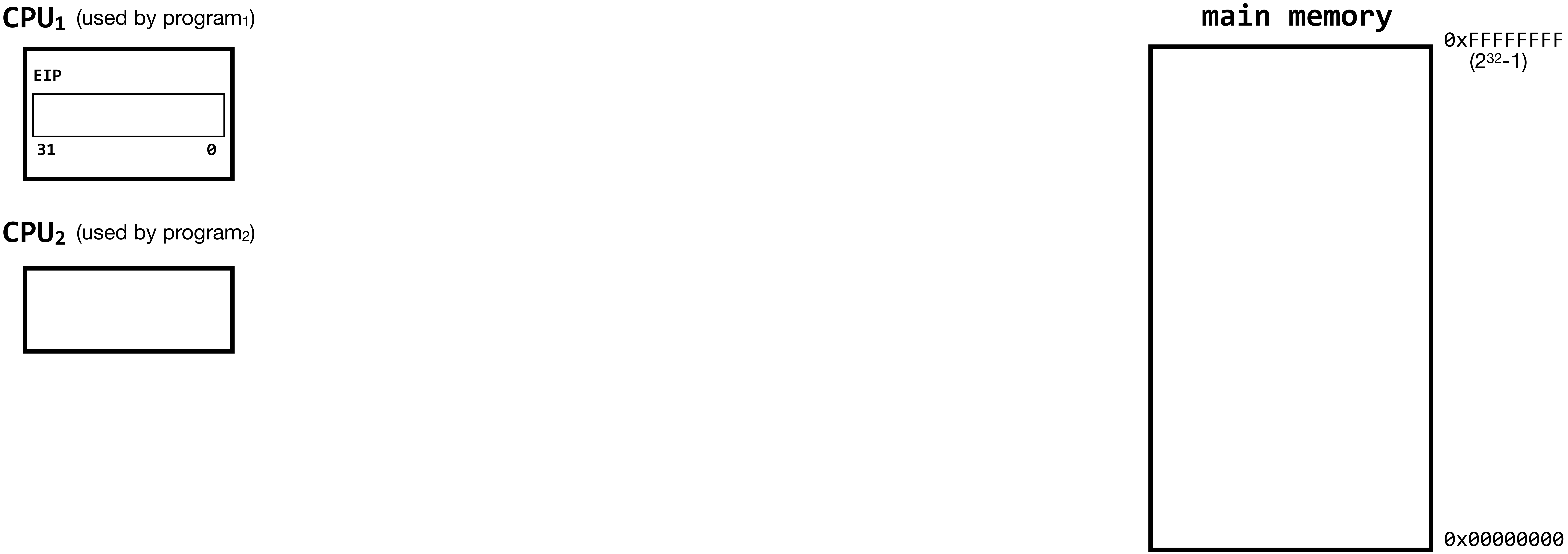
what we want: **virtualization.** every program should appear to have access to a full 32-bit address space

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

CPU₁ (used by program₁)
EIP
31
0
CPU₂ (used by program₂)
main memory

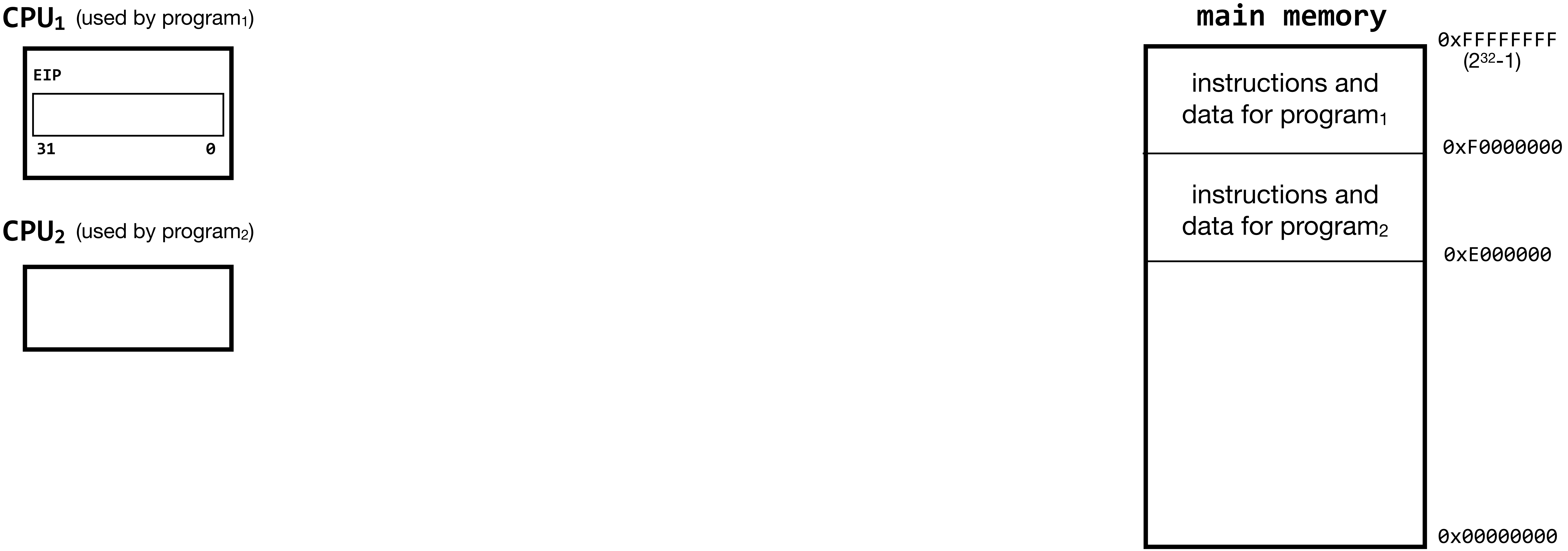
what we want: **virtualization.** every program should appear to have access to a full 32-bit address space

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



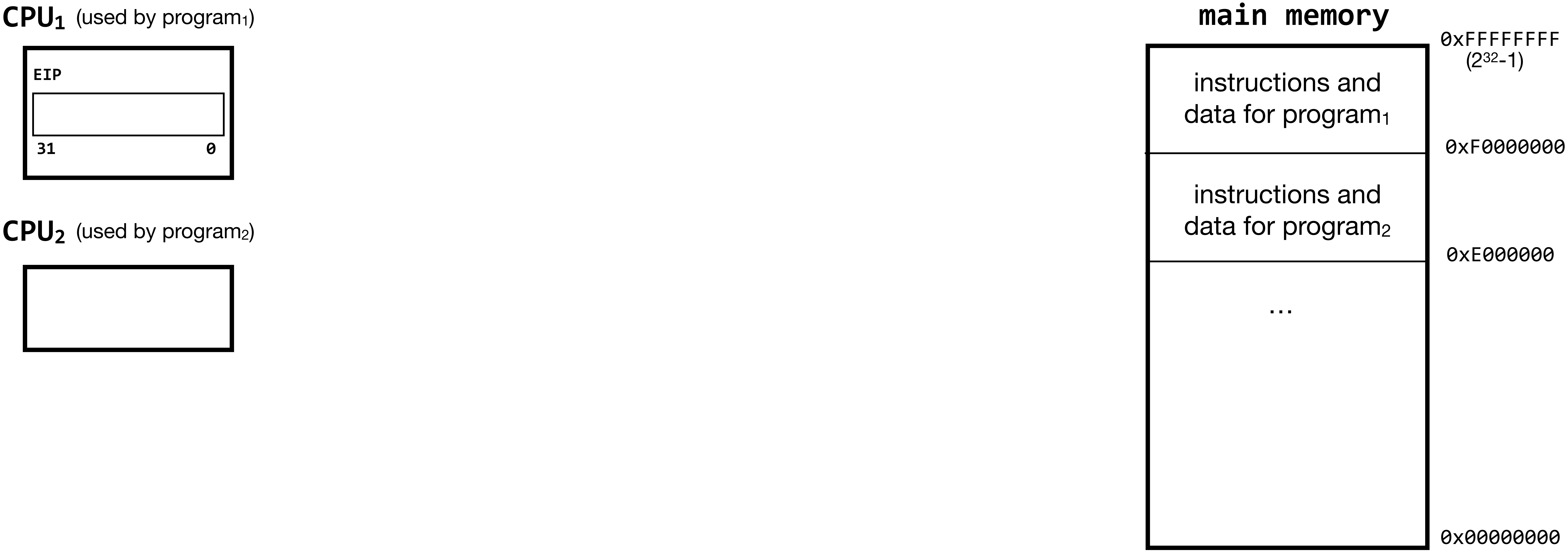
what we want: virtualization. every program should appear to have access to a full 32-bit address space

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



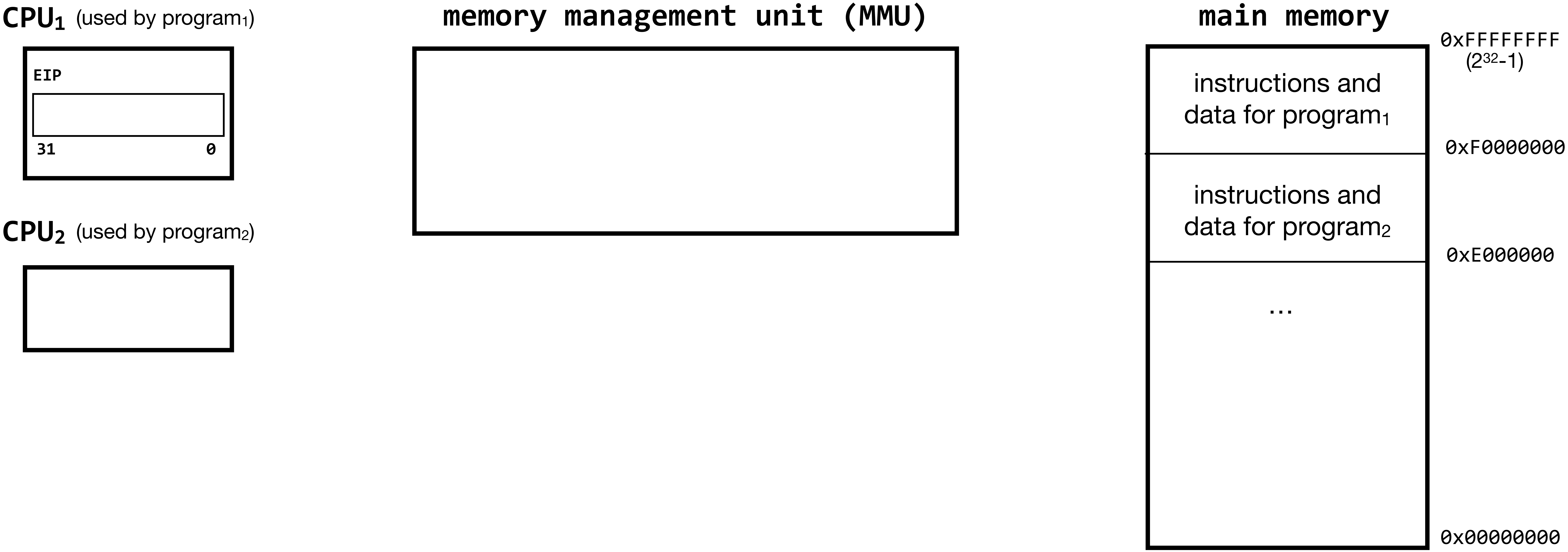
what we want: virtualization. every program should appear to have access to a full 32-bit address space

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



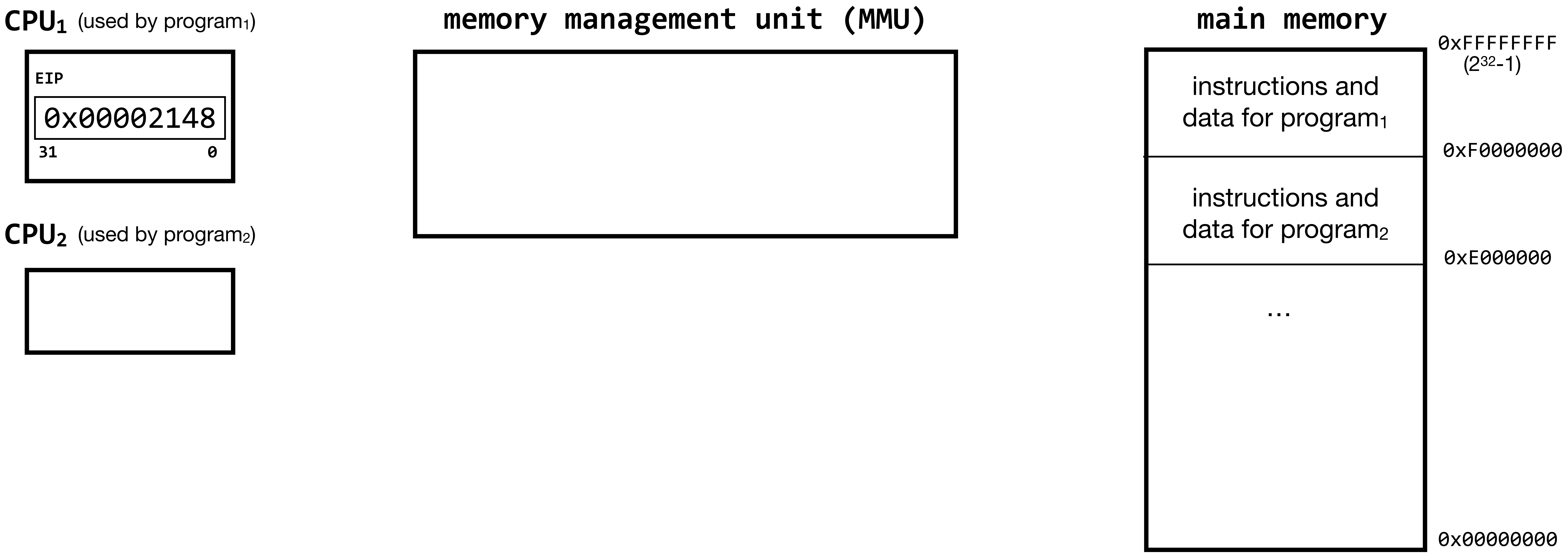
what we want: **virtualization.** every program should appear to have access to a full 32-bit address space

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



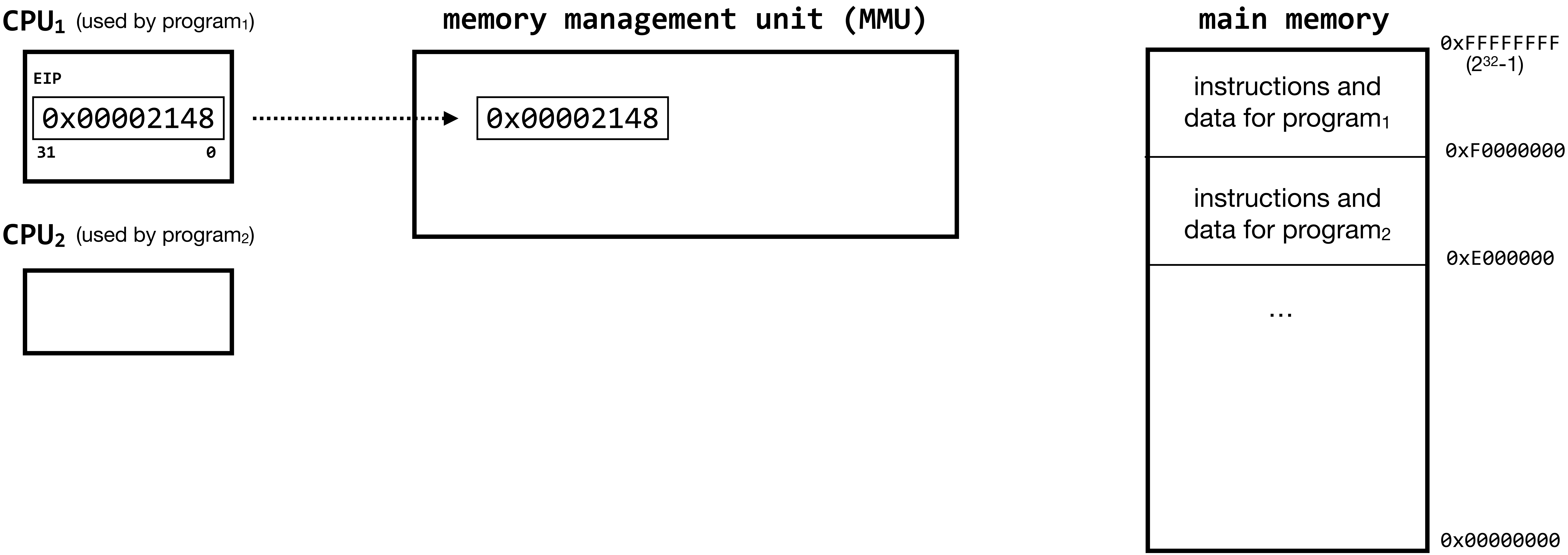
what we want: **virtualization.** every program should appear to have access to a full 32-bit address space

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



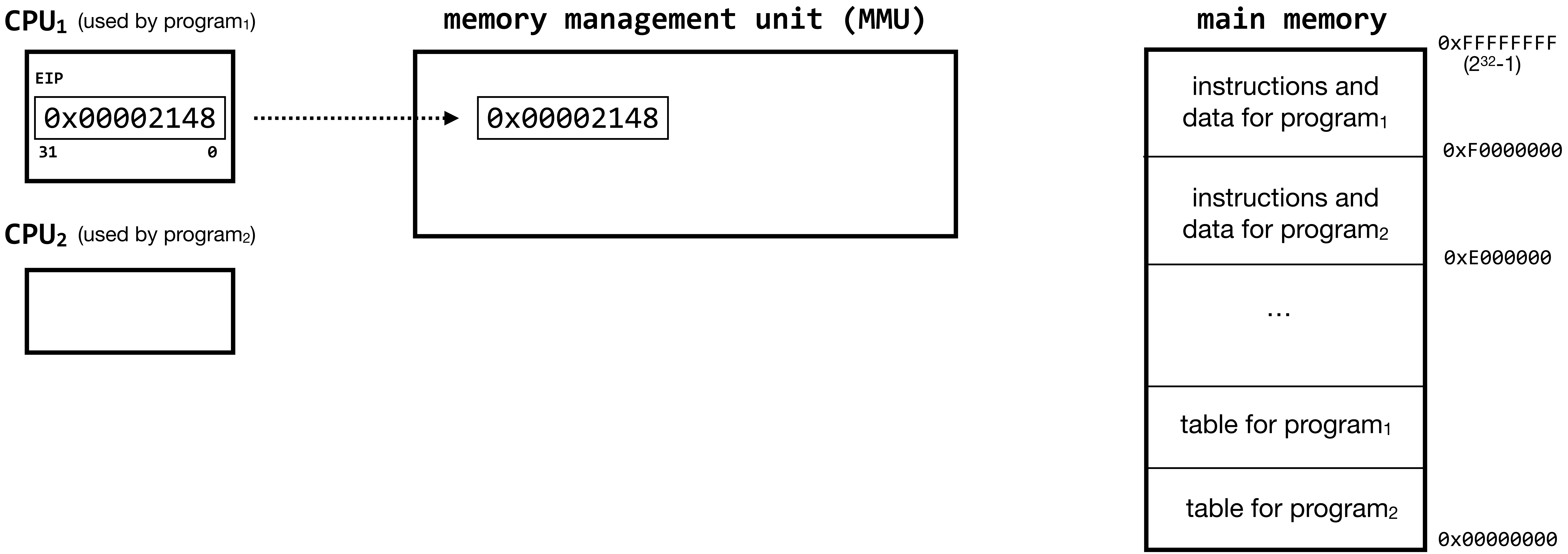
what we want: **virtualization.** every program should appear to have access to a full 32-bit address space

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



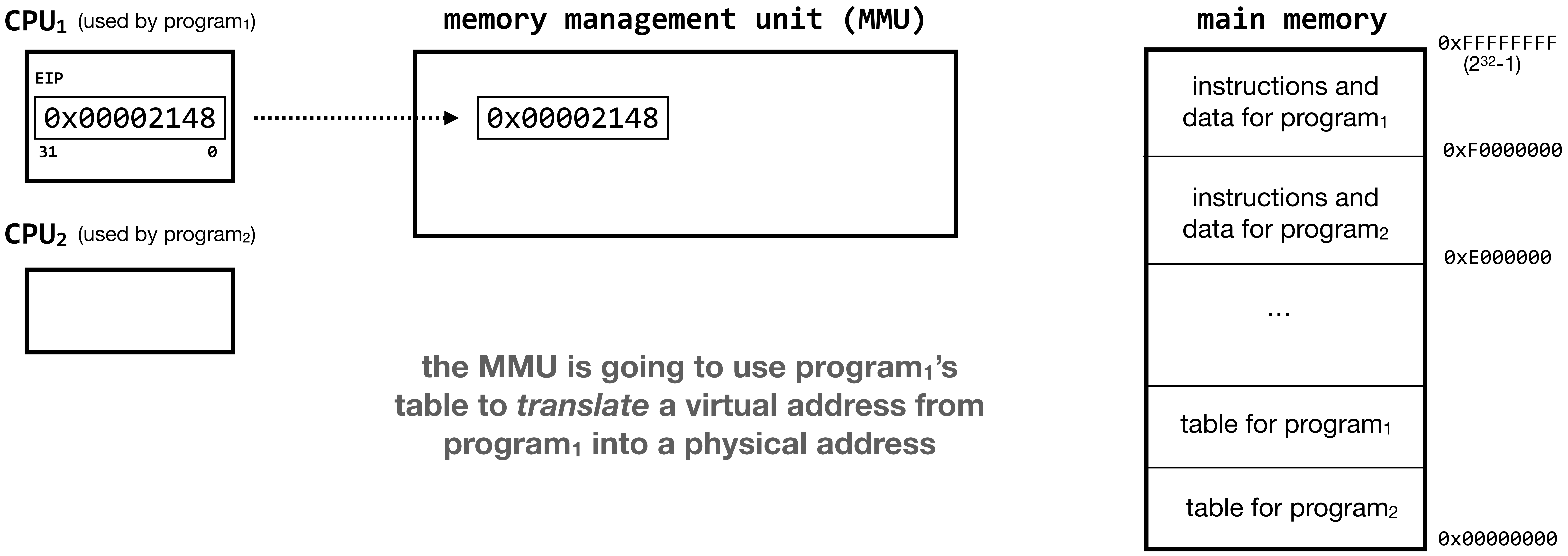
what we want: virtualization. every program should appear to have access to a full 32-bit address space

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



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

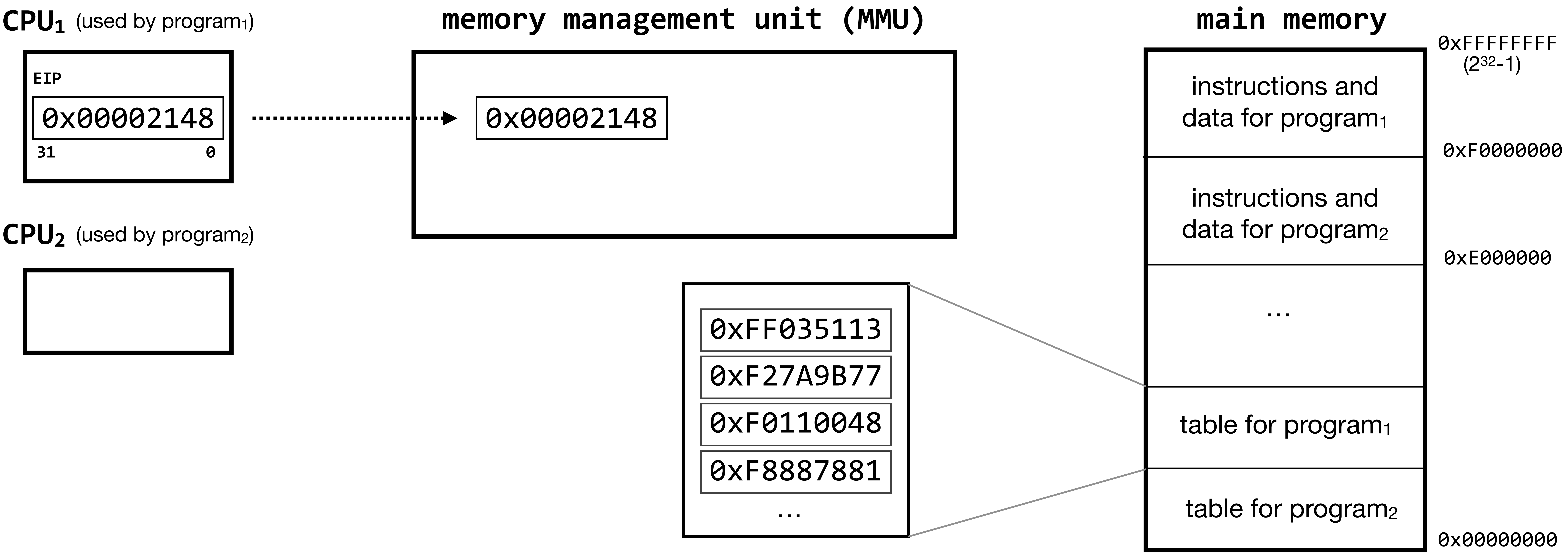
what we have: 2^{32} bytes of memory; every program can't *actually* have access to the full 32-bit space



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

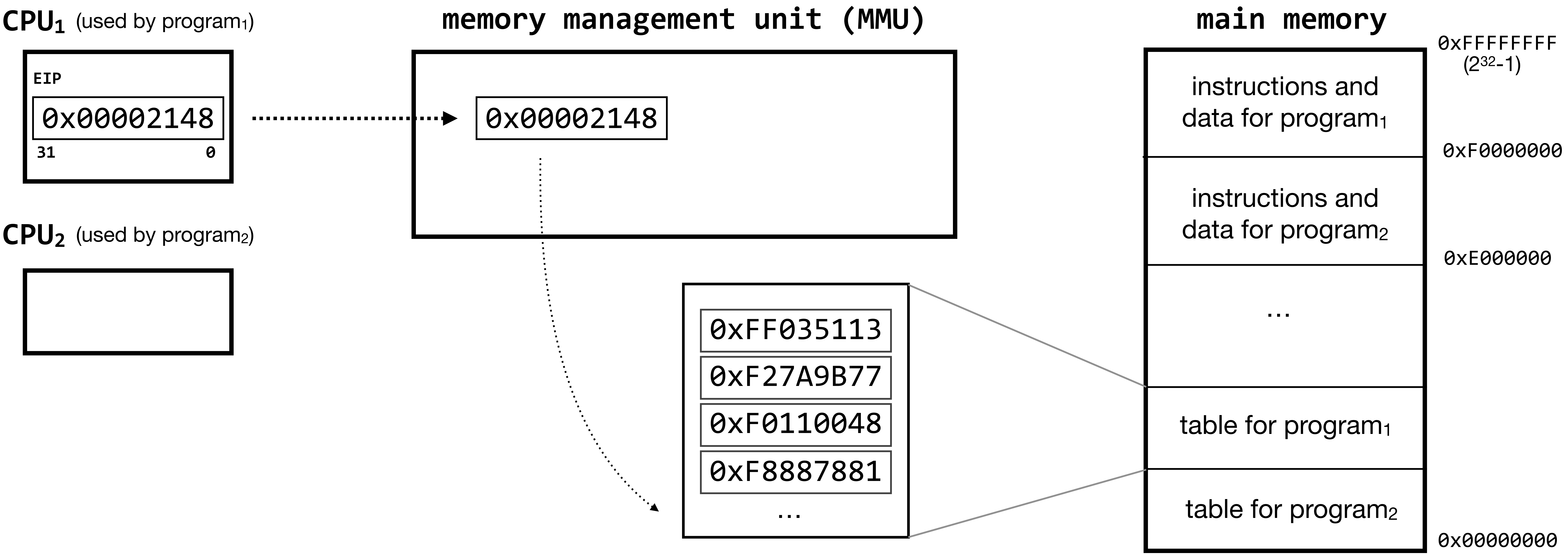
what we have: 2^{32} bytes of memory; every program can't *actually* have access to the full 32-bit space



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

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

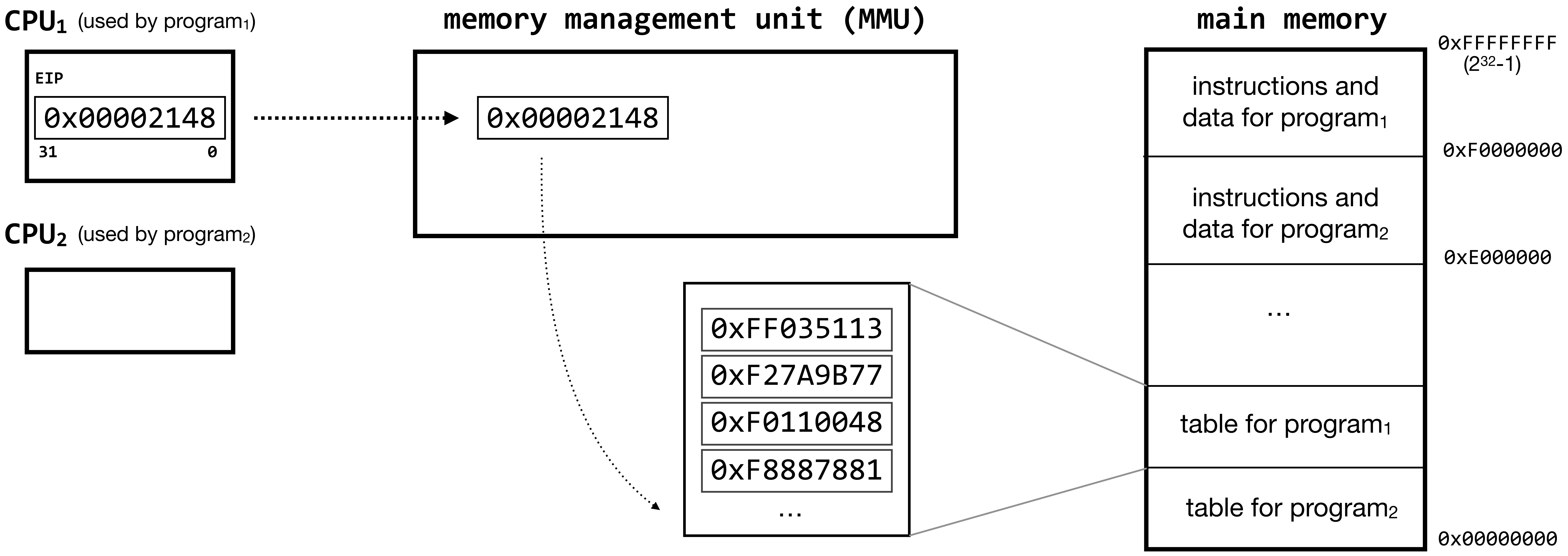
what we have: 2^{32} bytes of memory; every program can't *actually* have access to the full 32-bit space



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

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

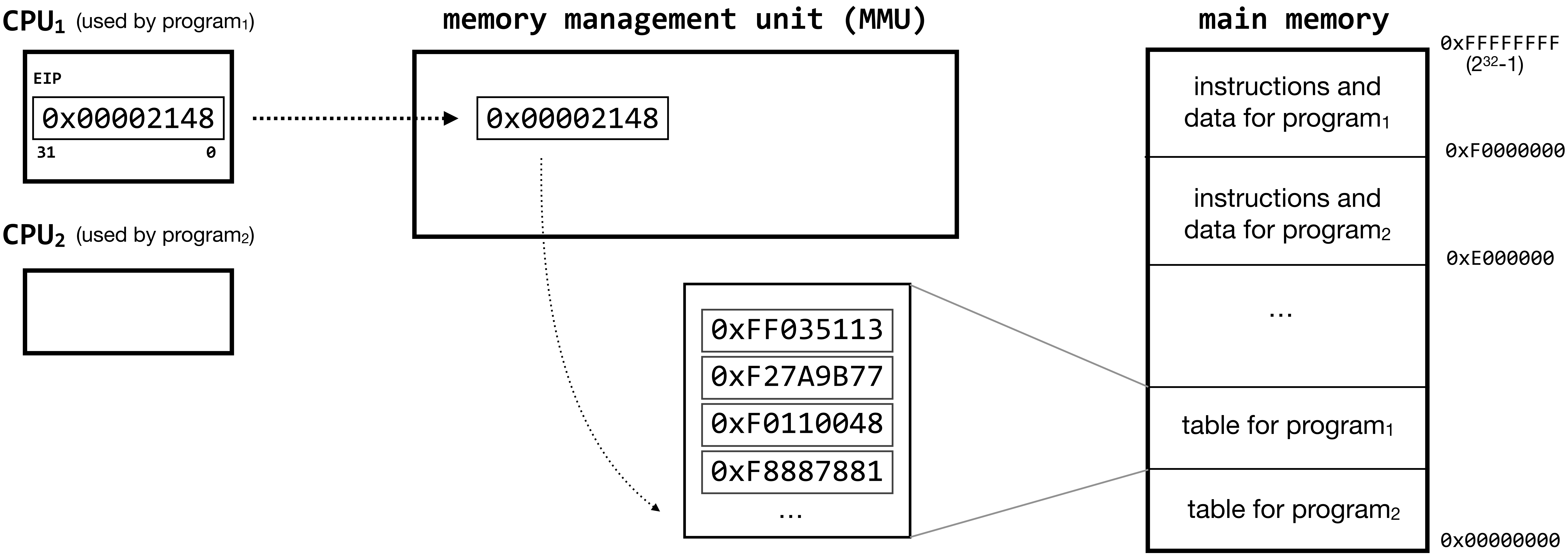


attempt 1: each virtual address acts as an index into this table; there is one entry for every virtual address

2^{32} virtual addresses each mapping to a 32-bit physical address →

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

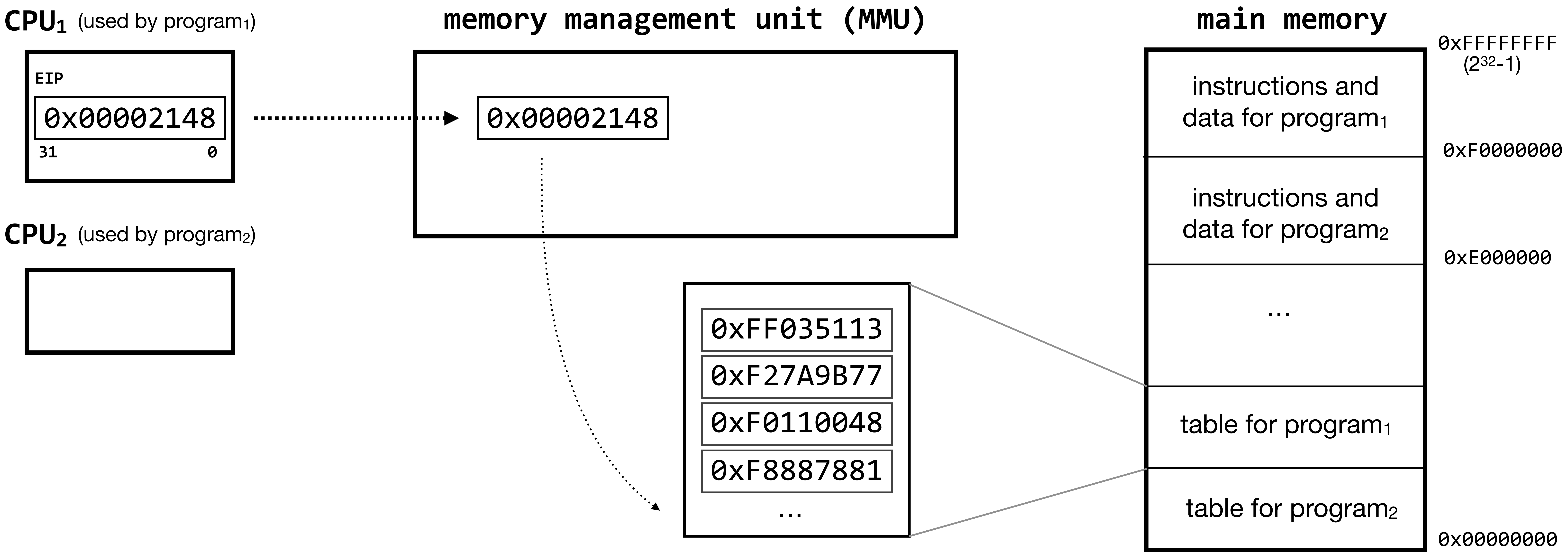


attempt 1: each virtual address acts as an index into this table; there is one entry for every virtual address

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

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



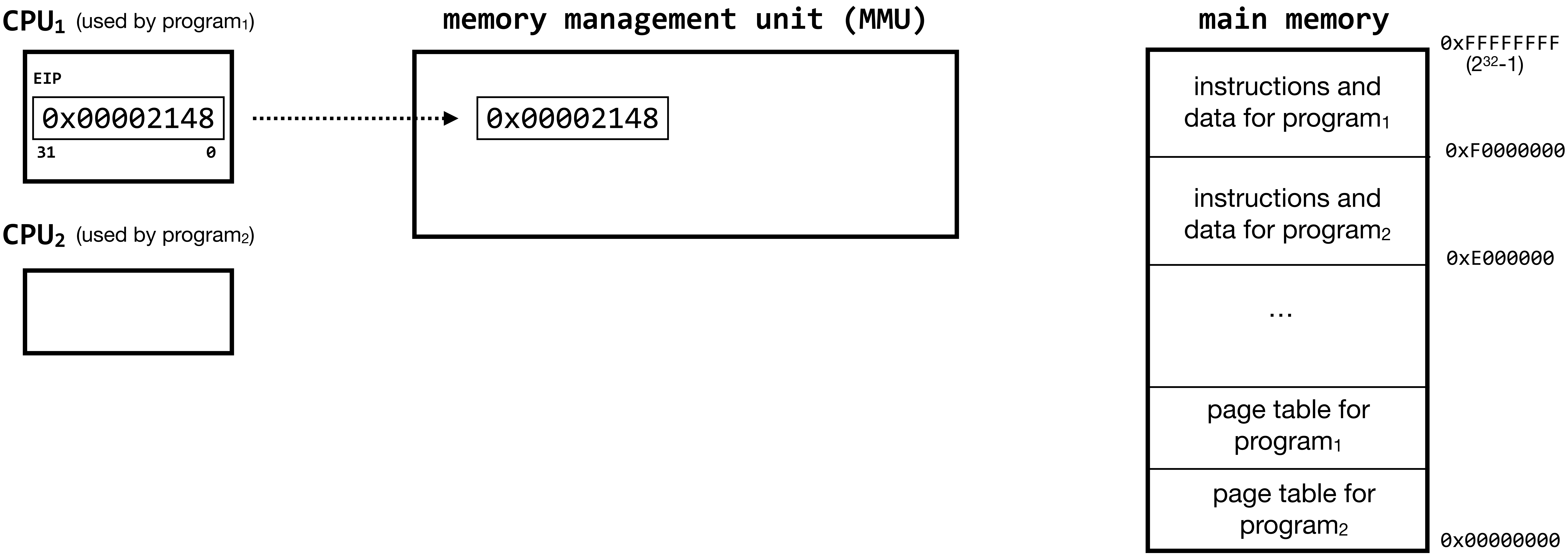
attempt 1: each virtual address acts as an index into this table; there is one entry for every virtual address

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

we don't even have 16GB 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



CPU_1 (used by program$_1$)

EIP

0x00002148

31 0

CPU_2 (used by program$_2$)

memory management unit (MMU)

0x00002148

main memory

0xFFFFFFFF (2^{32}-1)

instructions and data for program$_1$

0xF0000000

instructions and data for program$_2$

0xE000000

...

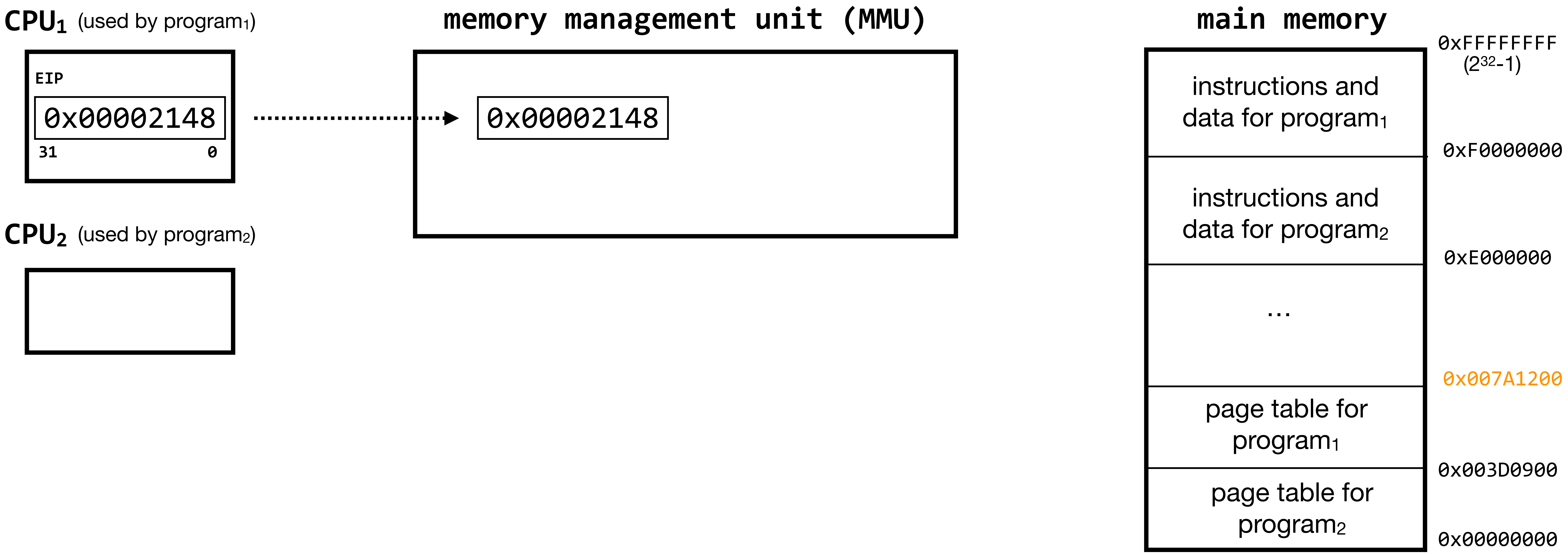
page table for program$_1$

page table for program$_2$

0x00000000

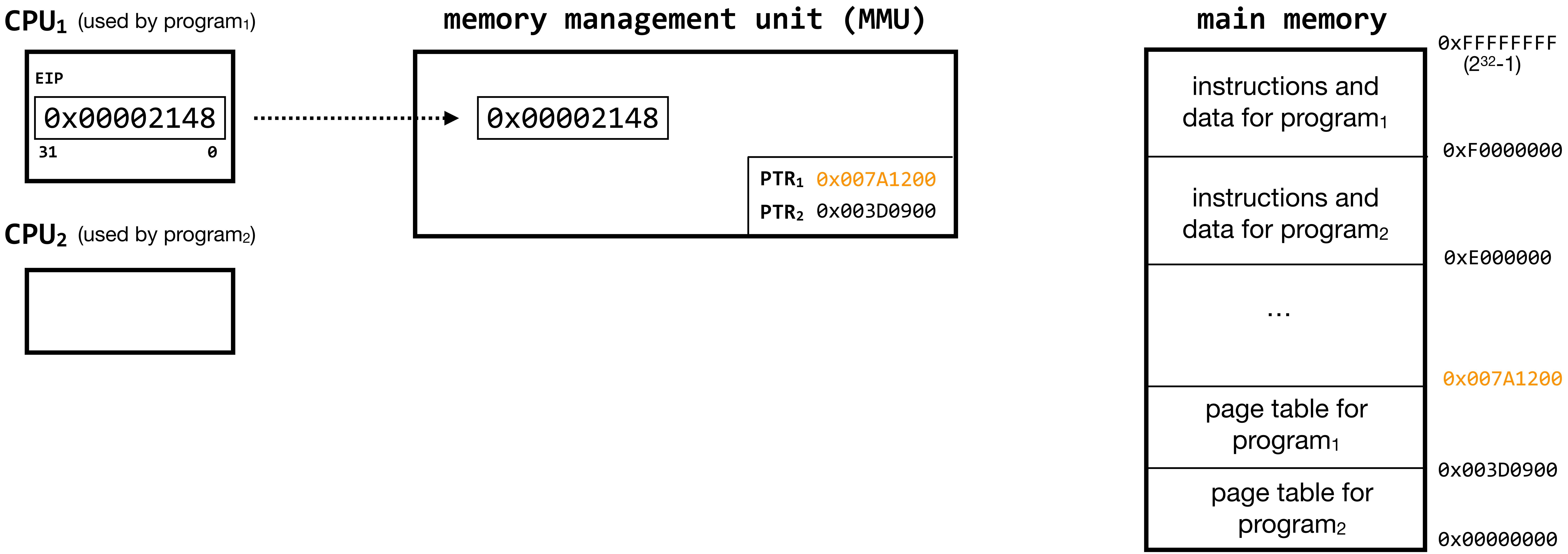
what we want: **virtualization.** every program should appear to have access to a full 32-bit address space

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



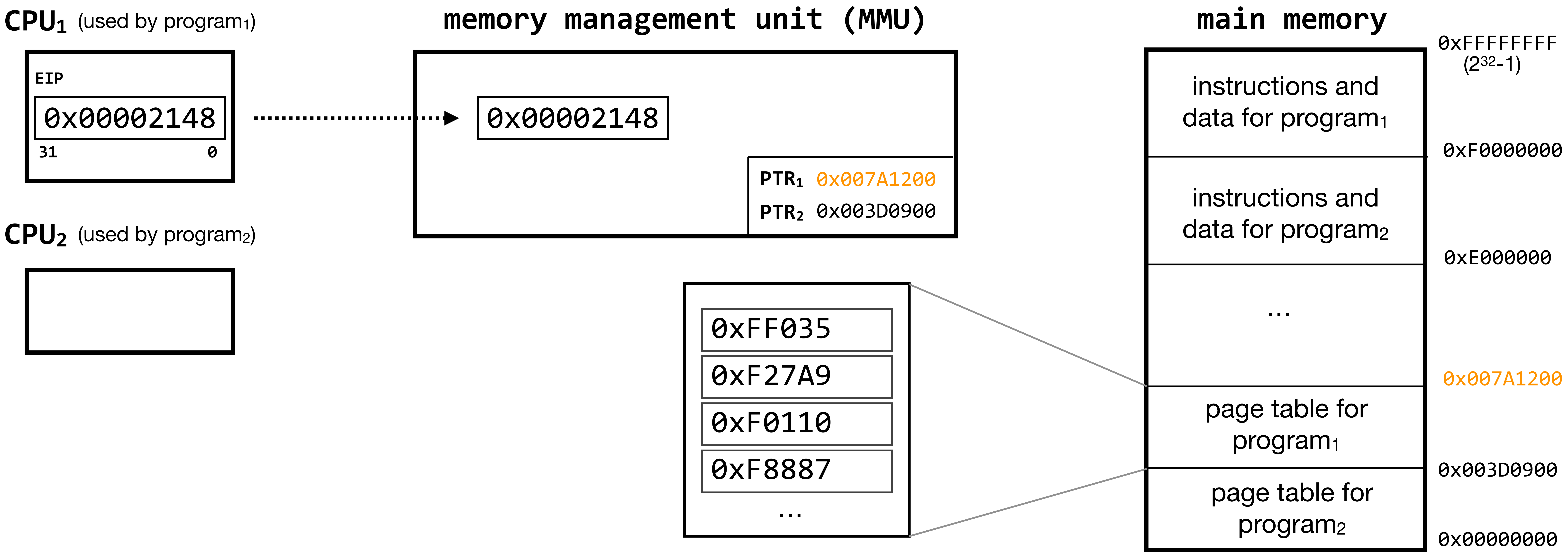
what we want: **virtualization.** every program should appear to have access to a full 32-bit address space

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



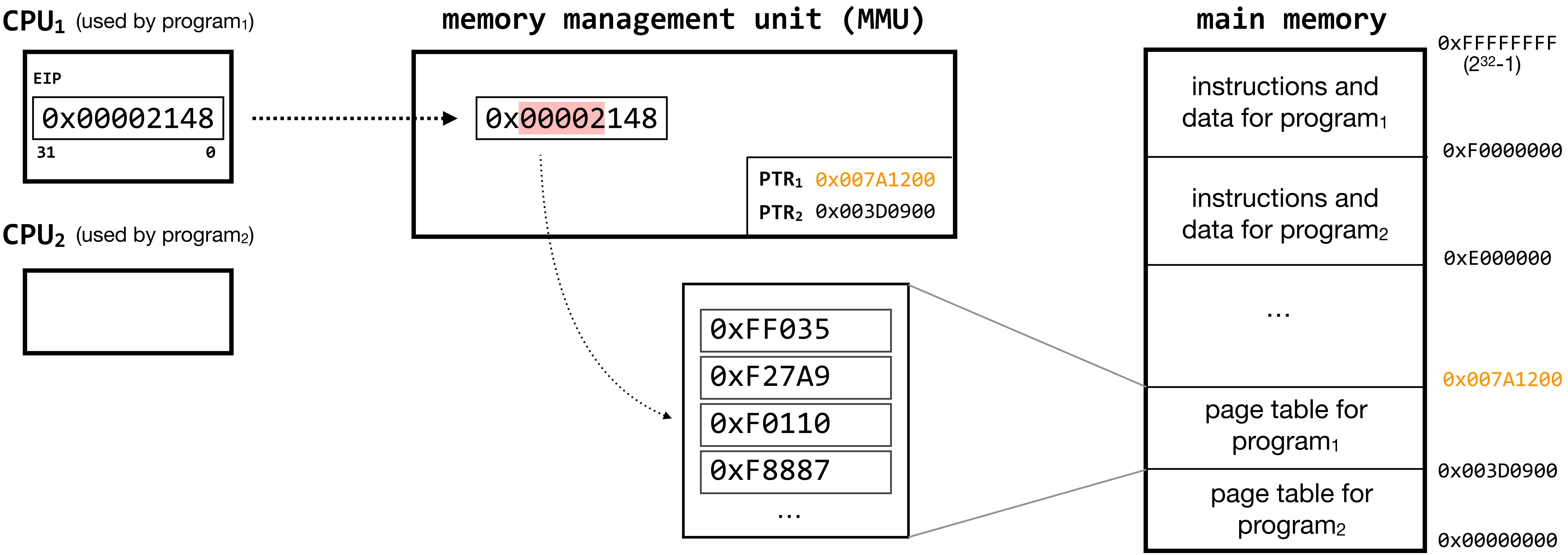
what we want: virtualization. every program should appear to have access to a full 32-bit address space

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



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

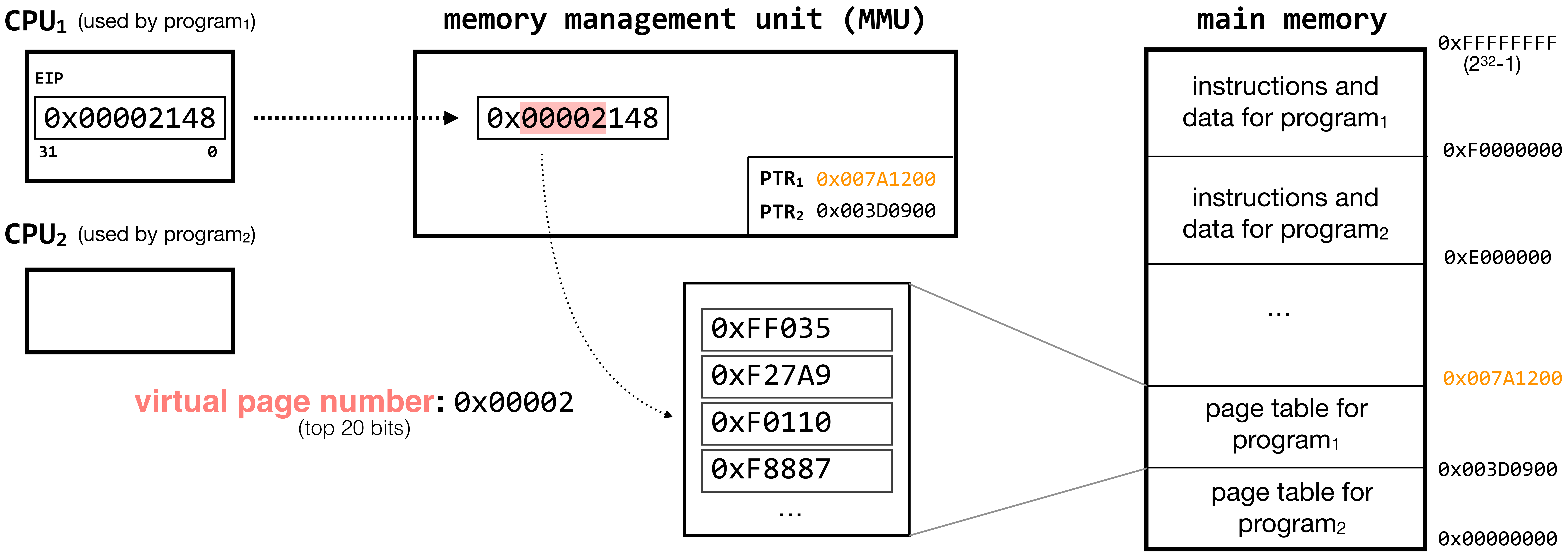


page tables: top 20 bits of the virtual address act as an index into this table

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

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



CPU$_1$ (used by program$_1$)

EIP

0x00002148

31 0

CPU$_2$ (used by program$_2$)

memory management unit (MMU)

0x00002148

PTR$_1$ 0x007A1200

PTR$_2$ 0x003D0900

virtual page number: 0x00002

(top 20 bits)

0xFF035

0xF27A9

0xF0110

0xF8887

…

main memory

0xFFFFFFFF (2^{32}-1)

instructions and data for program$_1$

0xF0000000

instructions and data for program$_2$

0xE000000

…

0x007A1200

page table for program$_1$

0x003D0900

page table for program$_2$

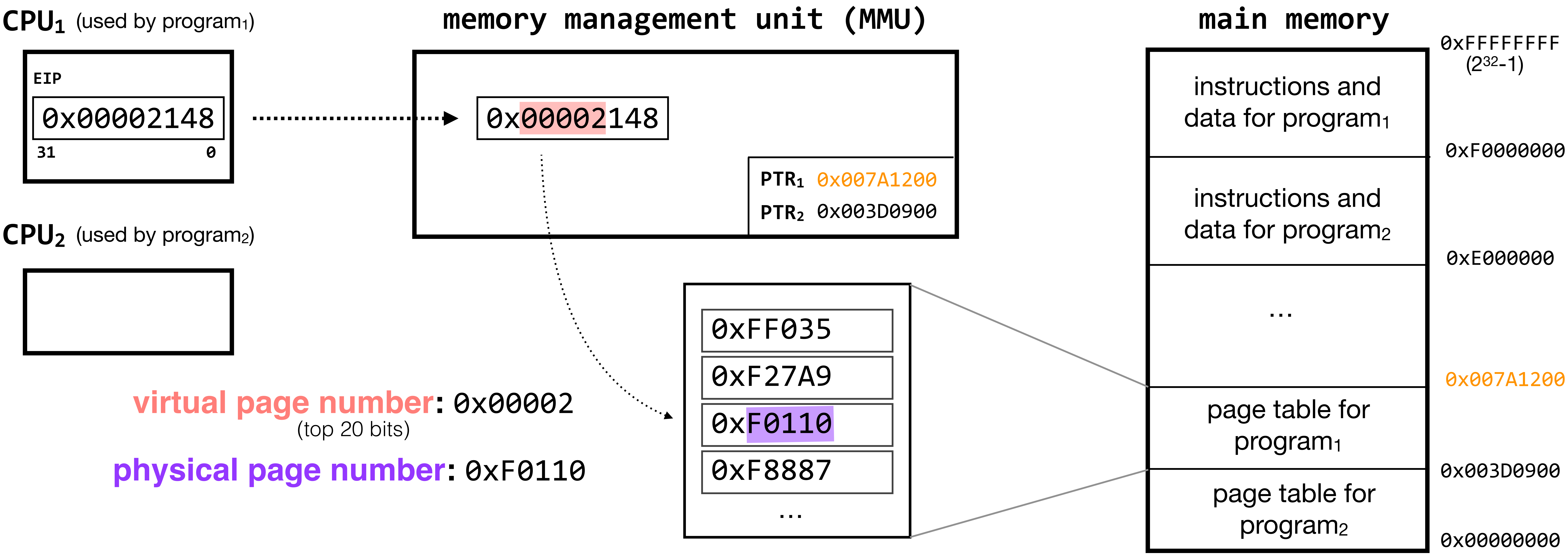
0x00000000

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)

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



virtual page number: 0x00002 (top 20 bits)

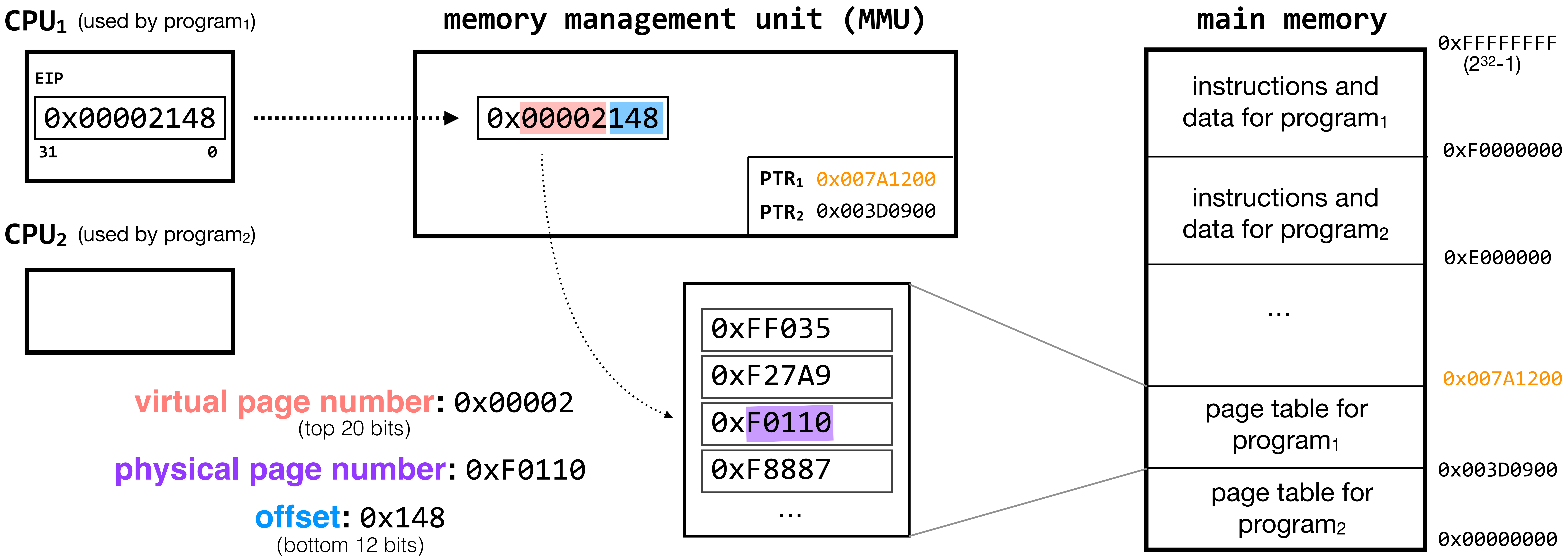
physical page number: 0xF0110

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)

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



virtual page number: 0x00002
(top 20 bits)

physical page number: 0xF0110

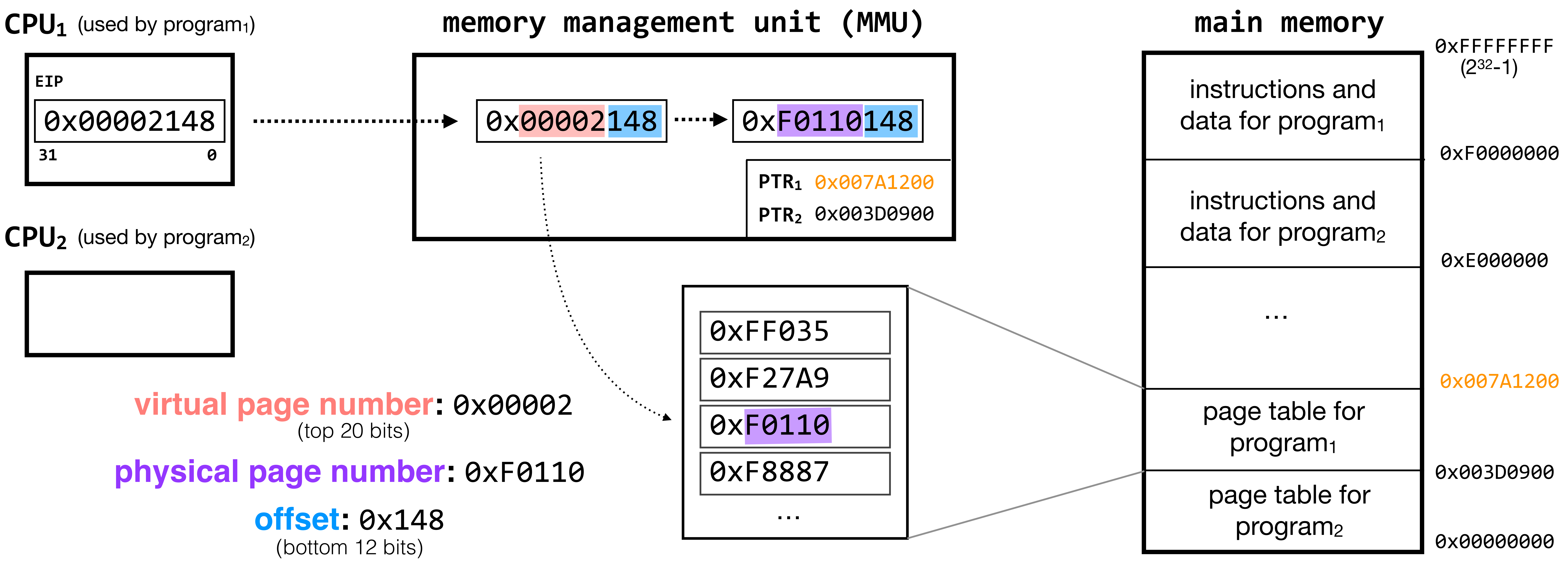
offset: 0x148
(bottom 12 bits)

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)

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



CPU₁ (used by program₁)

EIP

0x00002148

31 0

CPU₂ (used by program₂)

memory management unit (MMU)

0x00002148

0xF0110148

PTR₁ 0x007A1200

PTR₂ 0x003D0900

0xFF035

0xF27A9

0xF0110

0xF8887

…

main memory

0xFFFFFFFF (2^{32}-1)

instructions and data for program₁

0xF0000000

instructions and data for program₂

0xE0000000

…

0x007A1200

page table for program₁

0x003D0900

page table for program₂

0x00000000

virtual page number: 0x00002 (top 20 bits)

physical page number: 0xF0110

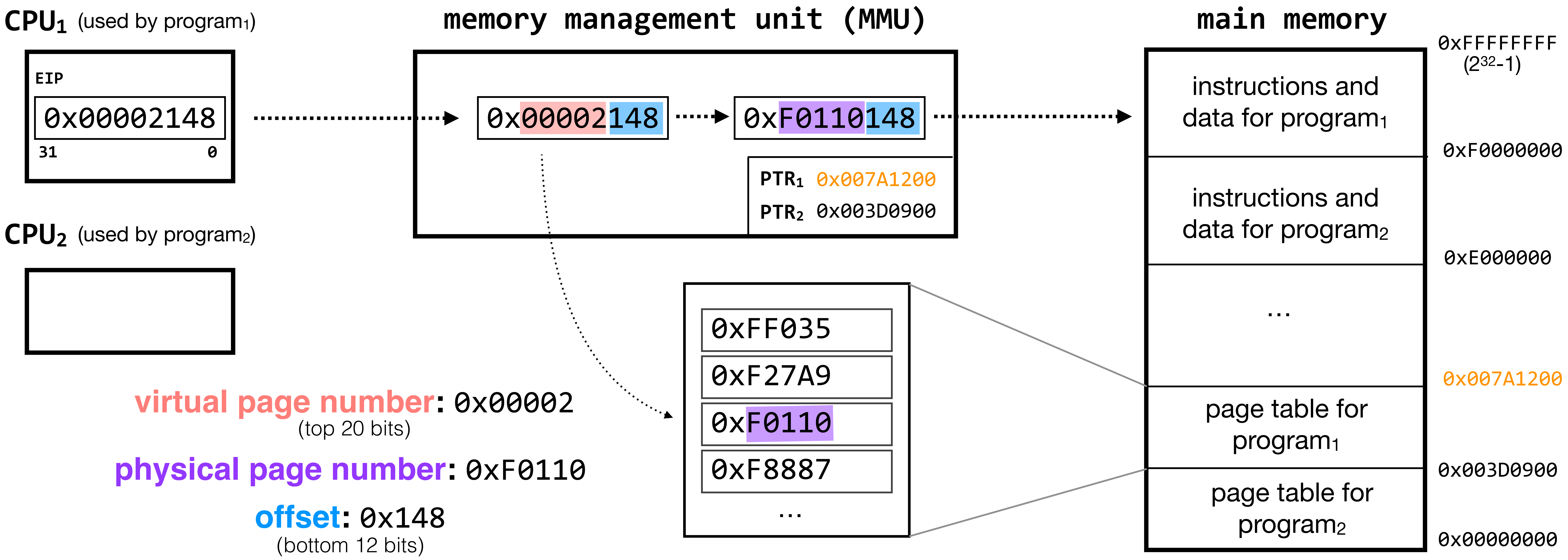
offset: 0x148 (bottom 12 bits)

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)

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



virtual page number: 0x00002
(top 20 bits)

physical page number: 0xF0110

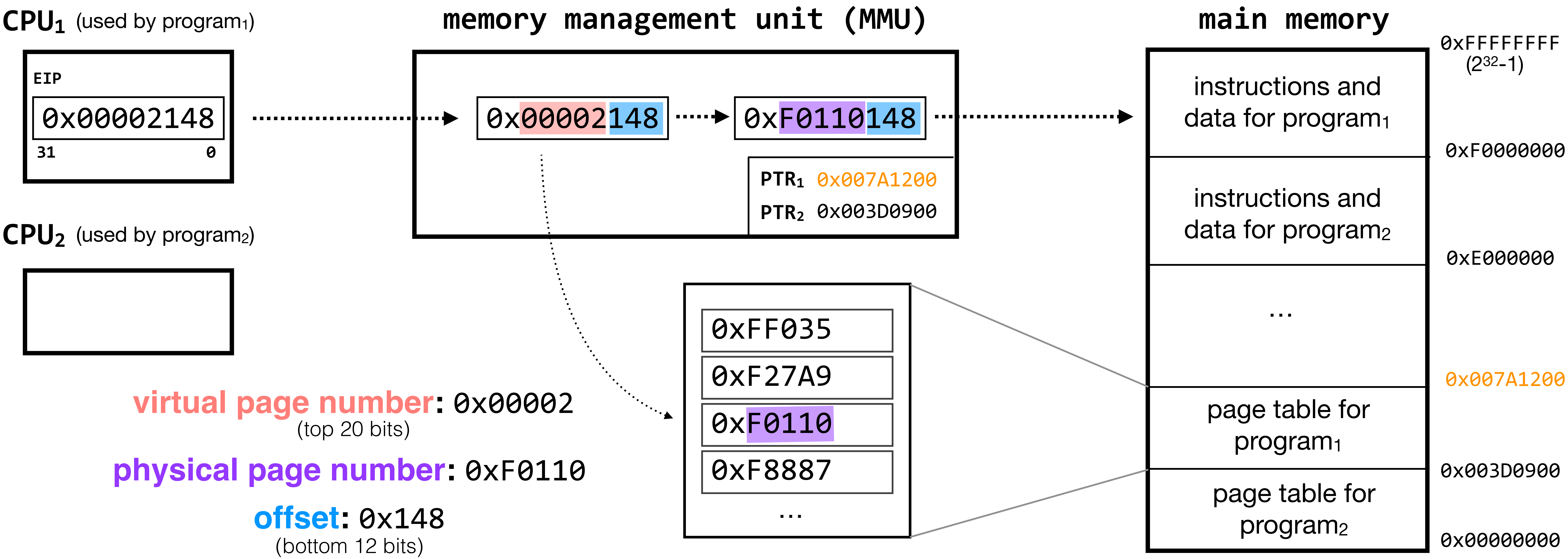
offset: 0x148
(bottom 12 bits)

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)

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



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

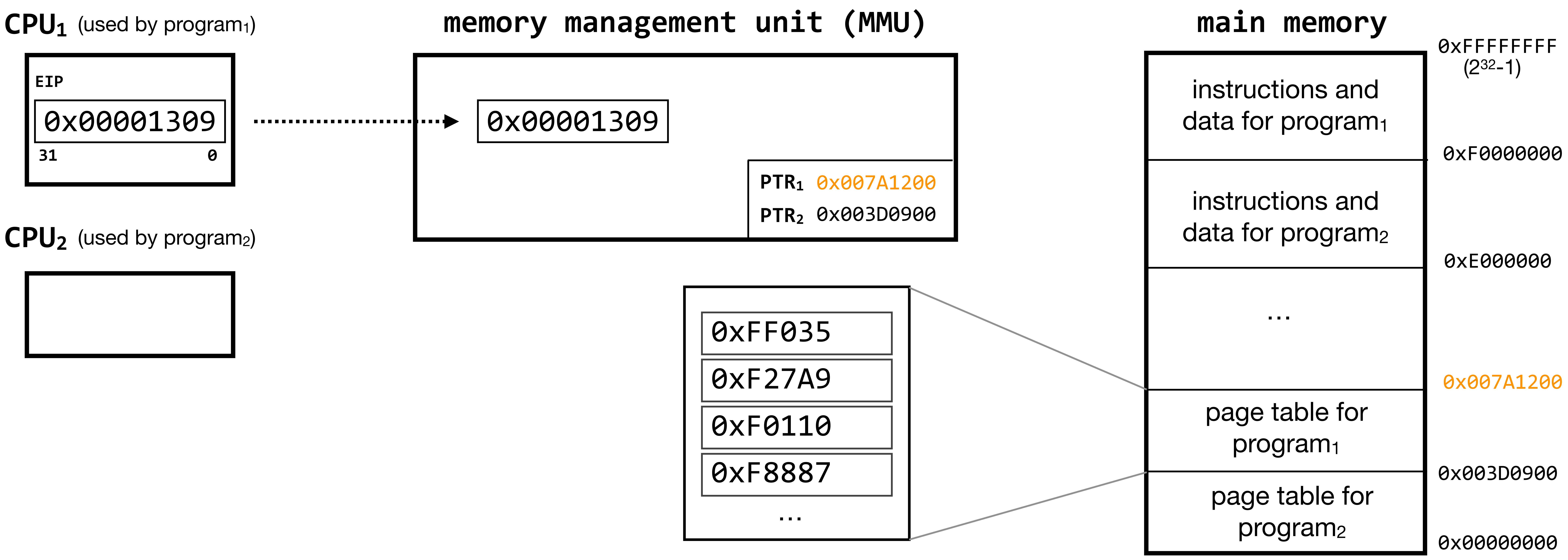
(a *page* of memory is $2^{32-20}=2^{12}$ bytes)

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)

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



CPU$_1$ (used by program$_1$)

EIP

0x00001309

31 0

CPU$_2$ (used by program$_2$)

memory management unit (MMU)

0x00001309

PTR$_1$ 0x007A1200

PTR$_2$ 0x003D0900

0xFF035

0xF27A9

0xF0110

0xF8887

...

main memory

0xFFFFFFFF (2^{32}-1)

instructions and data for program$_1$

0xF0000000

instructions and data for program$_2$

0xE000000

...

0x007A1200

page table for program$_1$

0x003D0900

page table for program$_2$

0x00000000

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

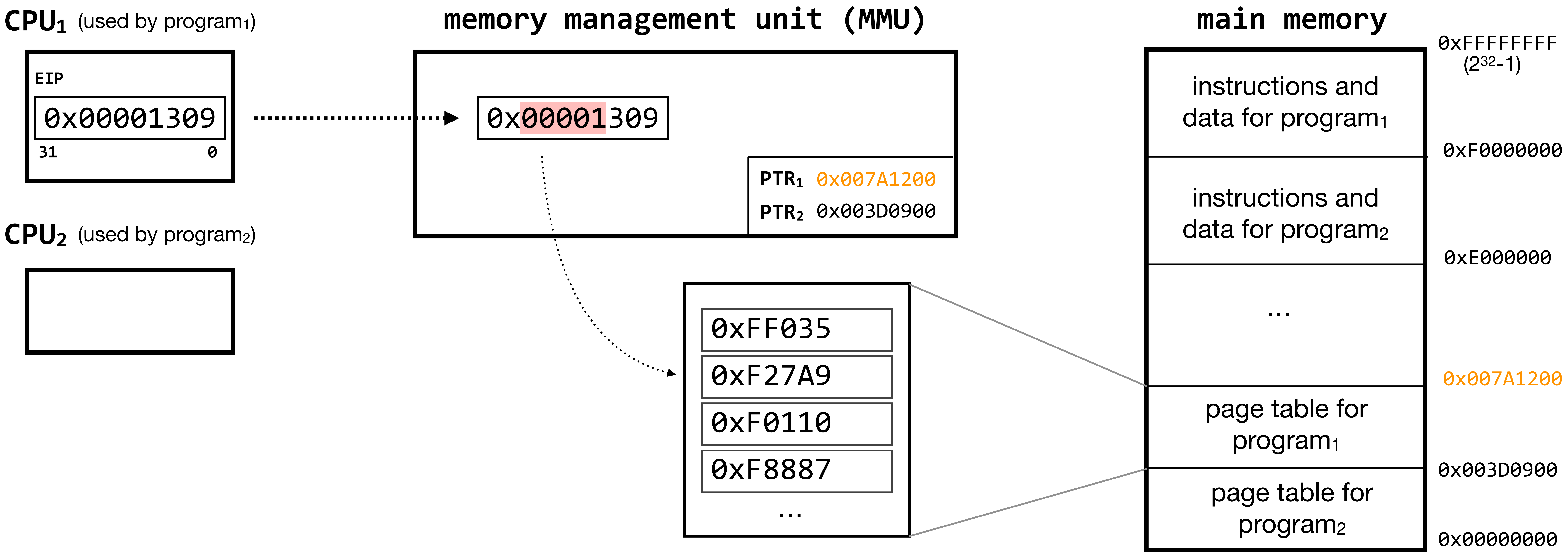
(a *page* of memory is $2^{32-20}=2^{12}$ bytes)

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

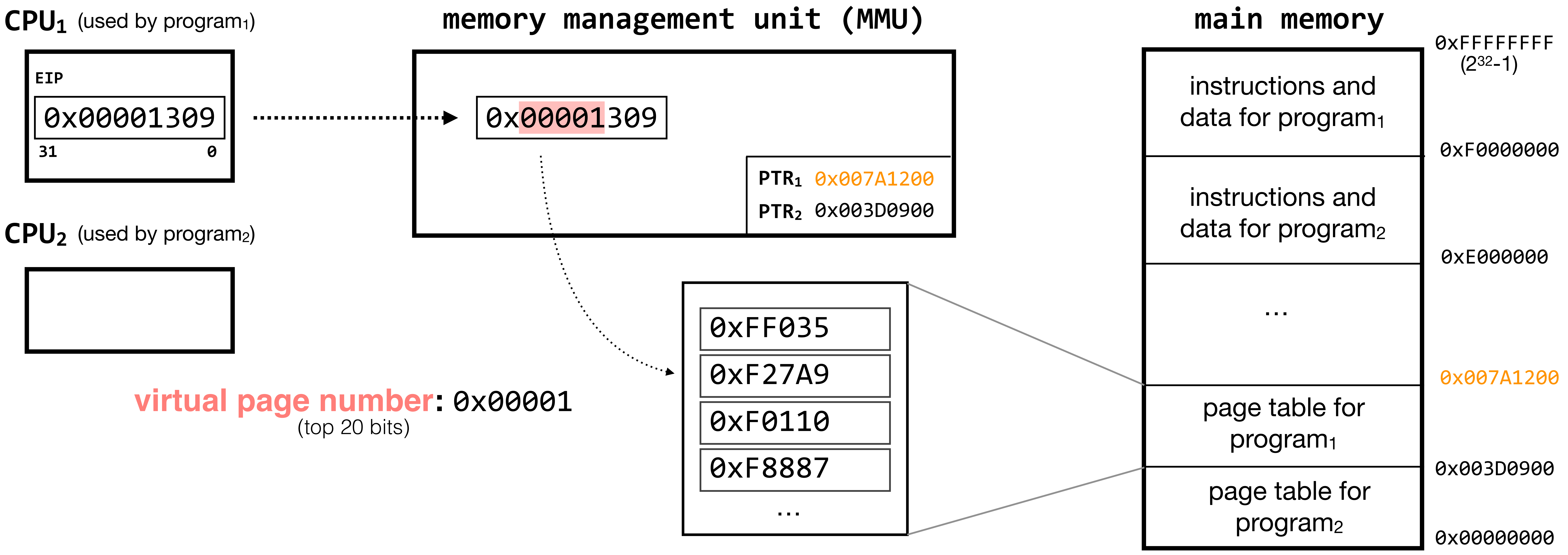
(a *page* of memory is $2^{32-20}=2^{12}$ bytes)

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)

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

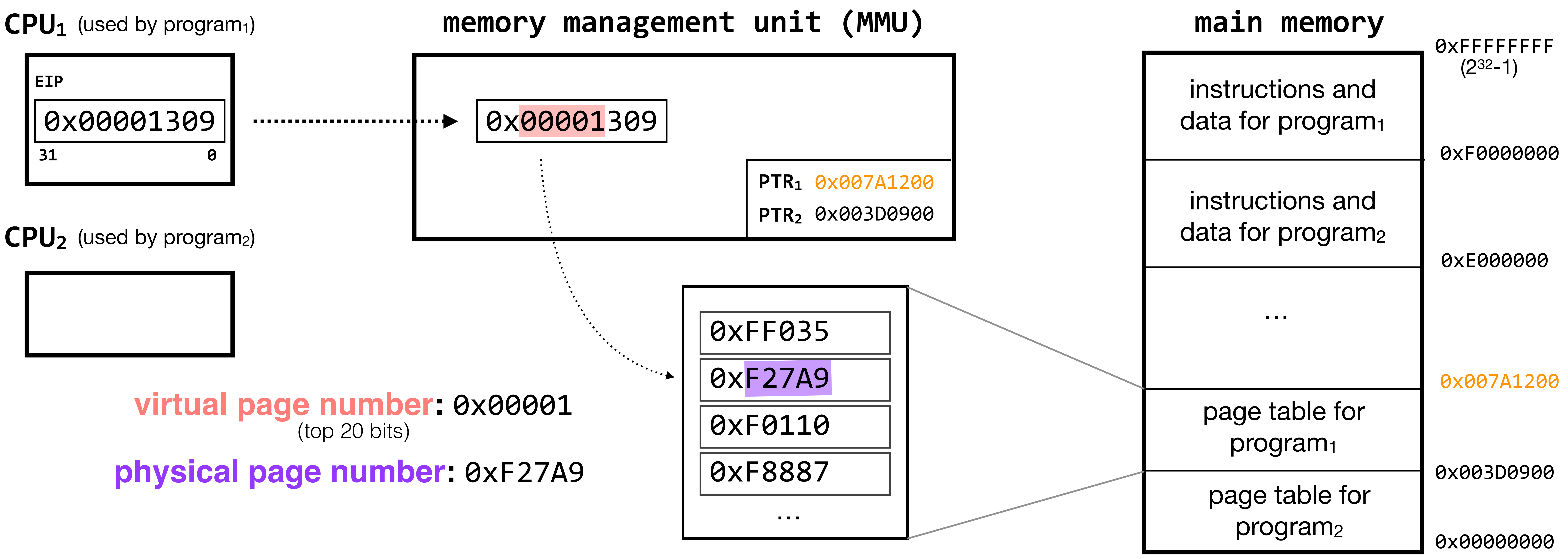
(a *page* of memory is $2^{32-20}=2^{12}$ bytes)

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


CPU$_1$ (used by program$_1$)
EIP
0x00001309
31
0
CPU$_2$ (used by program$_2$)
memory management unit (MMU)
0x00001309
PTR$_1$ 0x007A1200
PTR$_2$ 0x003D0900
0xFF035
0xF27A9
0xF0110
0xF8887
...
main memory
0xFFFFFFFF
(2^{32}-1)
instructions and data for program$_1$
0xF0000000
instructions and data for program$_2$
0xE000000
...
0x007A1200
page table for program$_1$
0x003D0900
page table for program$_2$
0x00000000

virtual page number: 0x00001
(top 20 bits)

physical page number: 0xF27A9

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

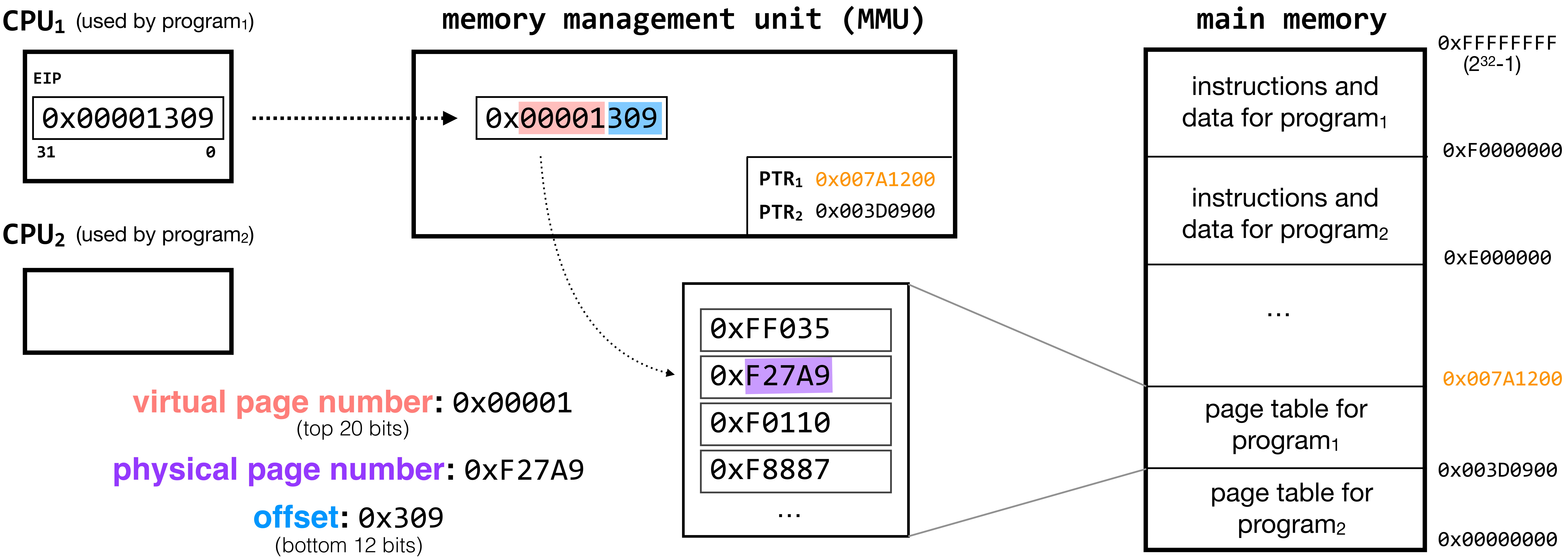
(a *page* of memory is $2^{32-20}=2^{12}$ bytes)

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



CPU_1 (used by program_1)
EIP
0x00001309
31
0
CPU_2 (used by program_2)
memory management unit (MMU)
0x00001309
PTR_1 0x007A1200
PTR_2 0x003D0900
0xFF035
0xF27A9
0xF0110
0xF8887
...
main memory
0xFFFFFFFF
(2^{32}-1)
instructions and data for program_1
0xF0000000
instructions and data for program_2
0xE000000
...
0x007A1200
page table for program_1
0x003D0900
page table for program_2
0x00000000

virtual page number: 0x00001
(top 20 bits)

physical page number: 0xF27A9

offset: 0x309
(bottom 12 bits)

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

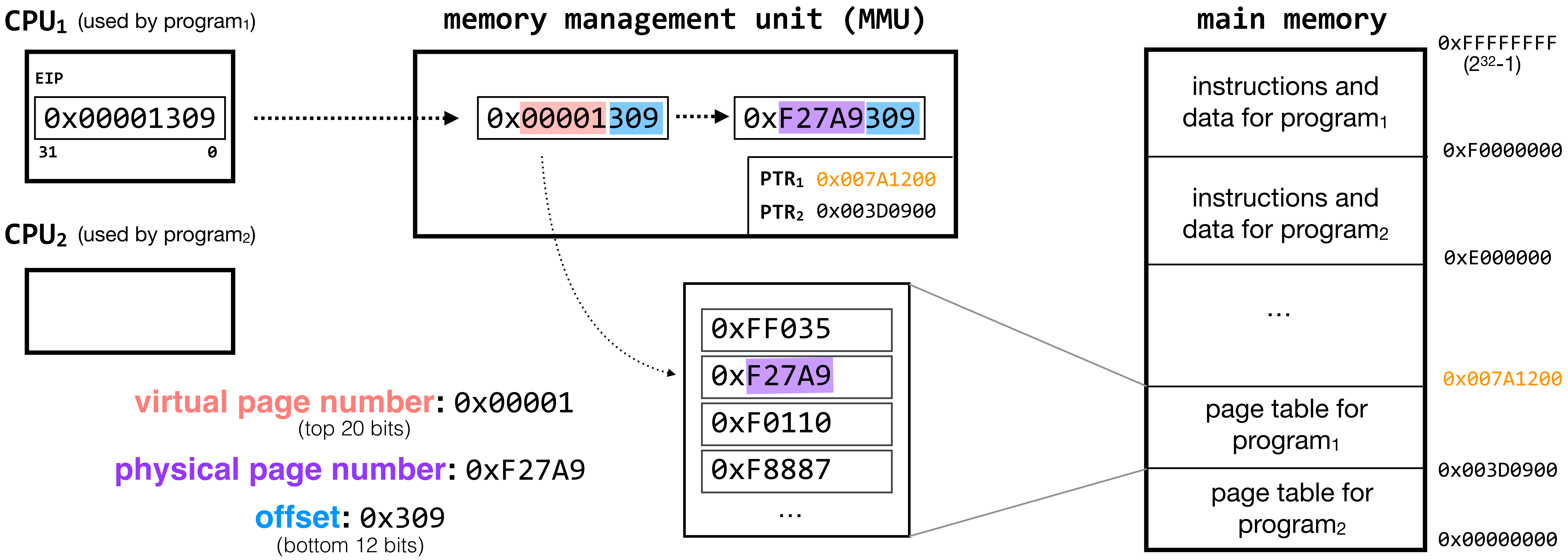
(a *page* of memory is 2^{32-20}=2^{12} bytes)

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)

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



CPU₁ (used by program₁)

EIP

0x00001309

31 0

CPU₂ (used by program₂)

memory management unit (MMU)

0x00001309 → 0xF27A9309

PTR₁ 0x007A1200

PTR₂ 0x003D0900

0xFF035

0xF27A9

0xF0110

0xF8887

...

main memory

instructions and data for program₁

instructions and data for program₂

...

page table for program₁

page table for program₂

0xFFFFFFFF (2^{32}-1)

0xF0000000

0xE0000000

0x007A1200

0x003D0900

0x00000000

virtual page number: 0x00001
(top 20 bits)

physical page number: 0xF27A9

offset: 0x309
(bottom 12 bits)

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

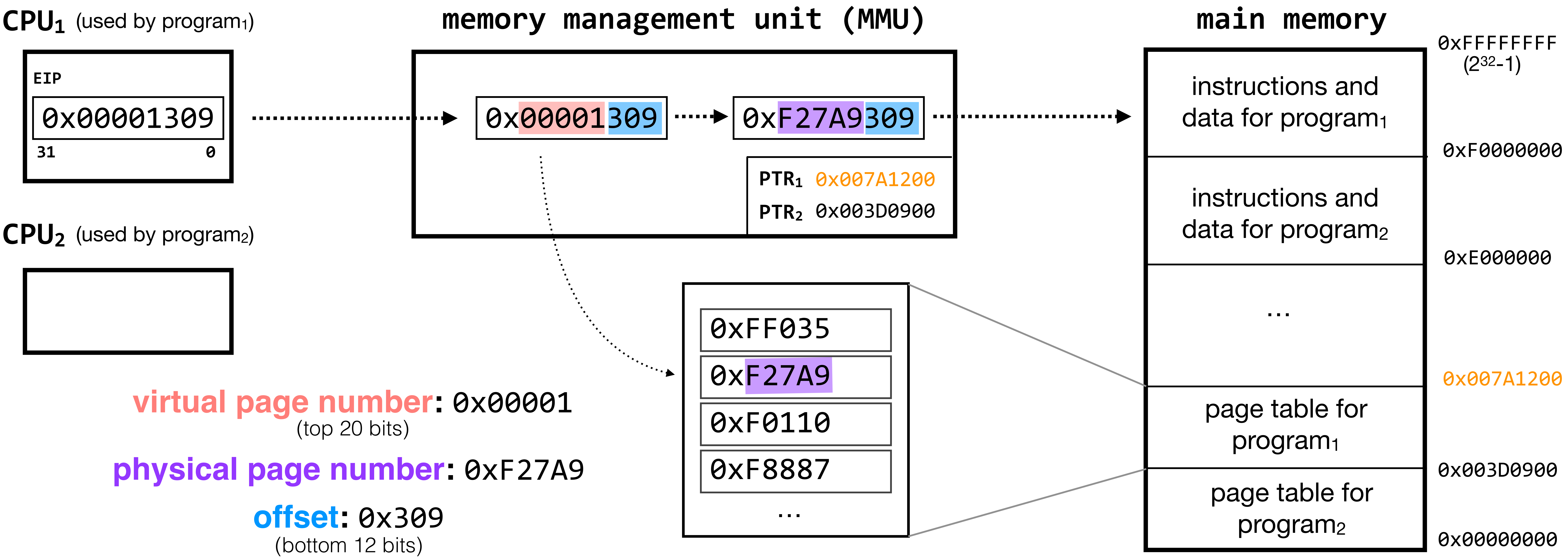
(a *page* of memory is $2^{32-20}=2^{12}$ bytes)

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)

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



virtual page number: 0x00001
(top 20 bits)

physical page number: 0xF27A9

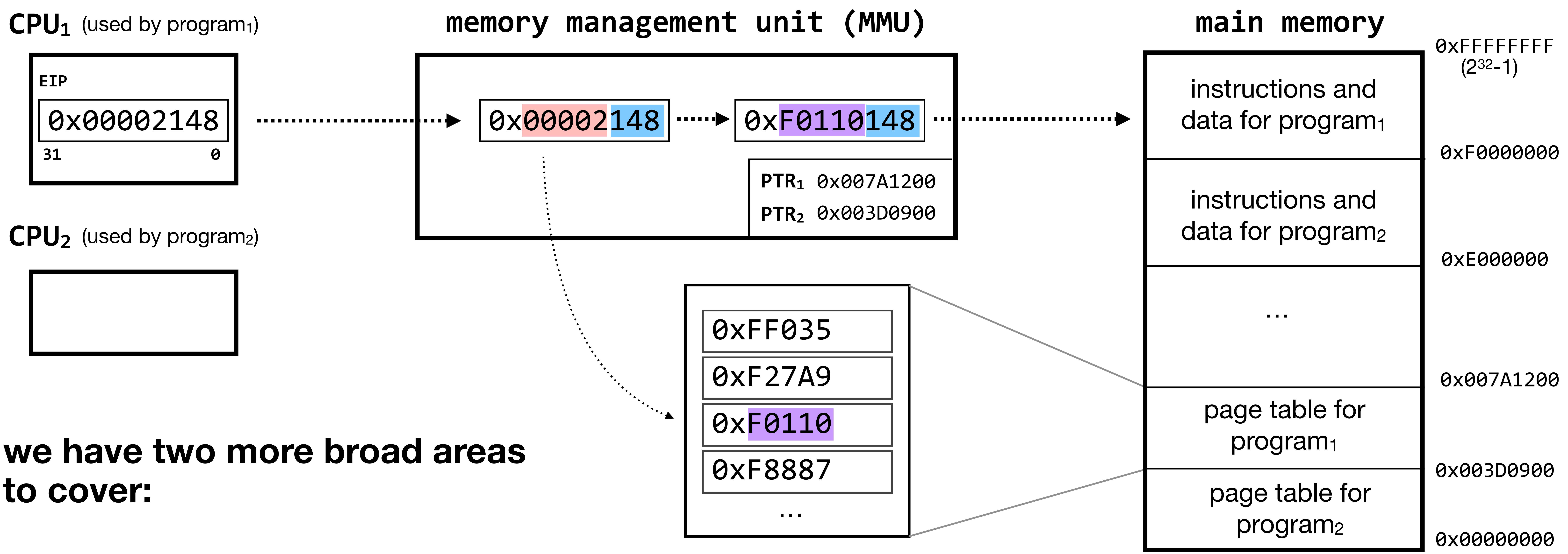
offset: 0x309
(bottom 12 bits)

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

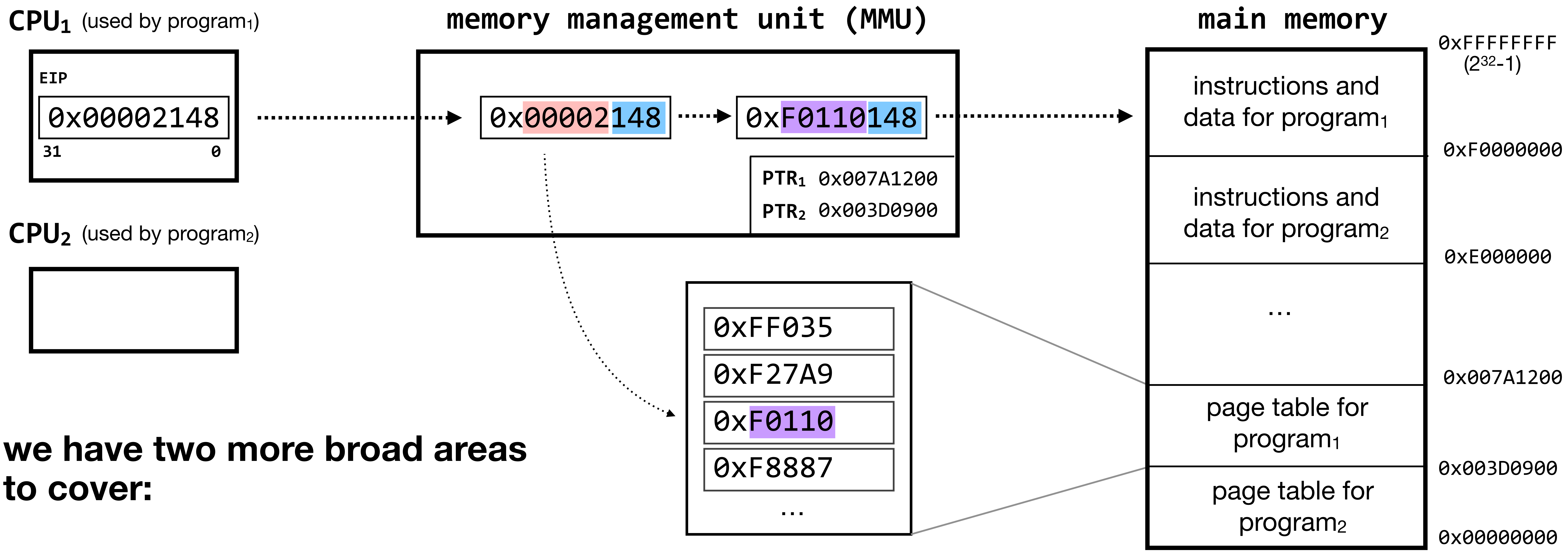
(a *page* of memory is $2^{32-20}=2^{12}$ bytes)

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)


CPU$_1$ (used by program$_1$)
EIP
0x00002148
31
0
CPU$_2$ (used by program$_2$)
memory management unit (MMU)
0x00002148
0xF0110148
PTR$_1$ 0x007A1200
PTR$_2$ 0x003D0900
0xFF035
0xF27A9
0xF0110
0xF8887
...
main memory
0xFFFFFFFF
$(2^{32}-1)$
instructions and data for program$_1$
0xF0000000
instructions and data for program$_2$
0xE000000
...
0x007A1200
page table for program$_1$
0x003D0900
page table for program$_2$
0x00000000

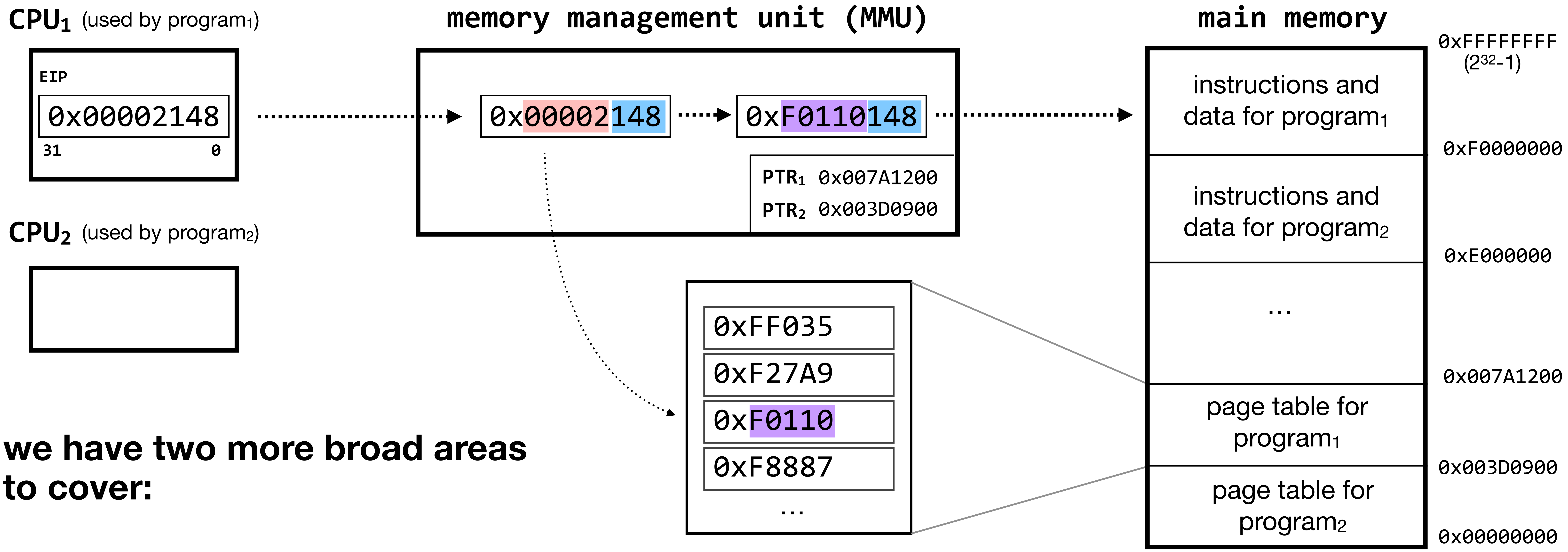
we have two more broad areas to cover:


CPU$_1$ (used by program$_1$)
EIP
0x00002148
31
0
CPU$_2$ (used by program$_2$)
memory management unit (MMU)
0x00002148
0xF0110148
PTR$_1$ 0x007A1200
PTR$_2$ 0x003D0900
0xFF035
0xF27A9
0xF0110
0xF8887
...
main memory
0xFFFFFFFF
(2^{32}-1)
instructions and data for program$_1$
0xF0000000
instructions and data for program$_2$
0xE000000
...
0x007A1200
page table for program$_1$
0x003D0900
page table for program$_2$
0x00000000

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)

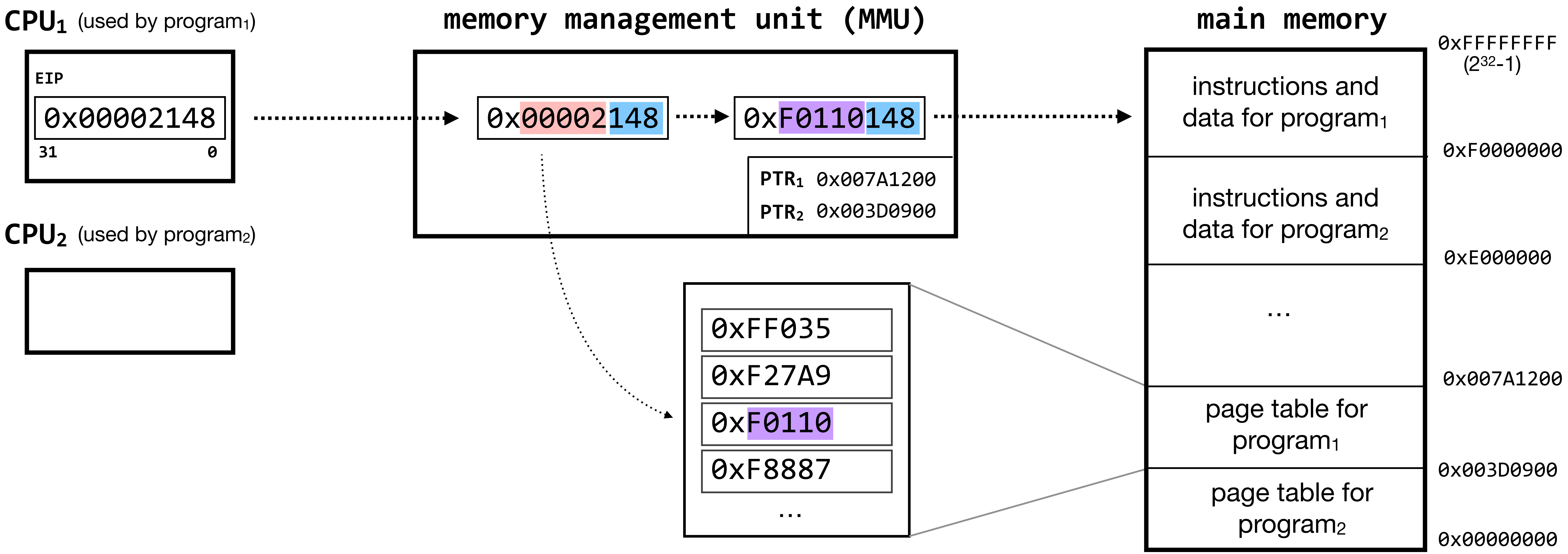


we have two more broad areas to cover:

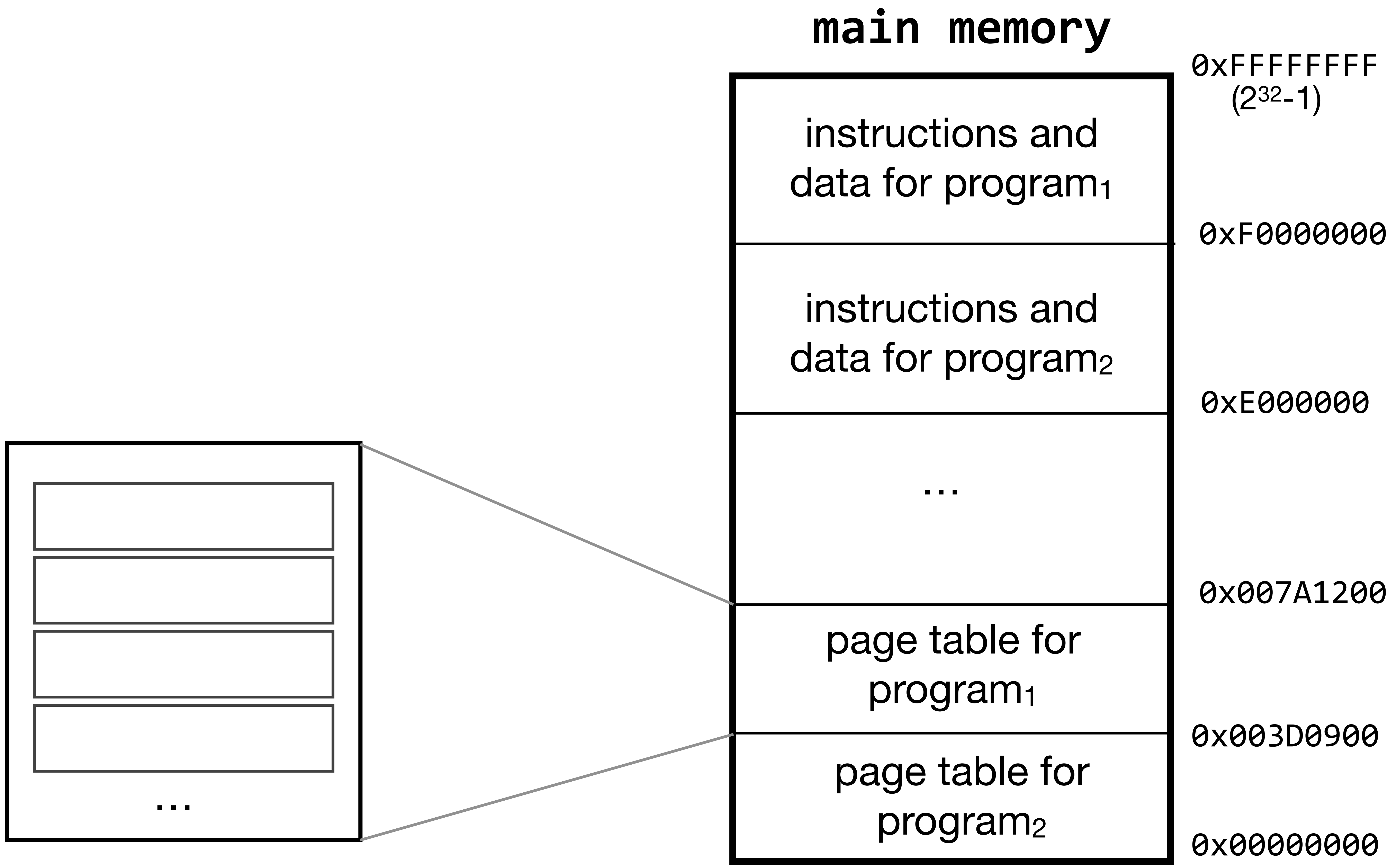
does virtual memory protect programs from accessing each other's memory?
(to answer this, we'll need to address some other issues first)

what performance issues matter here?

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

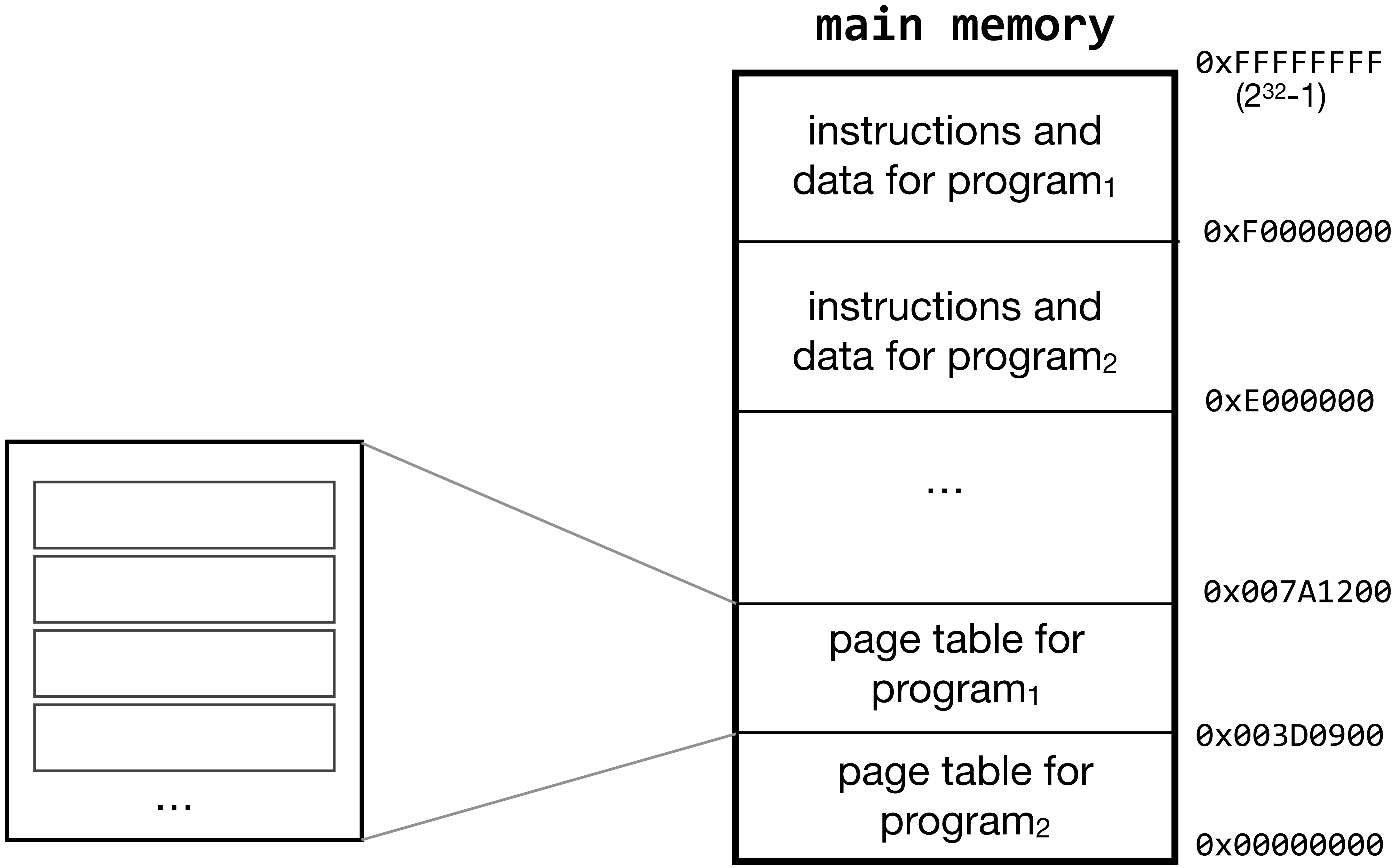


what happens if we don't have enough memory to store all of our programs' instructions and data?


main memory
0xFFFFFFFF
(2^{32}-1)
instructions and data for program$_1$
0xF0000000
instructions and data for program$_2$
0xE000000
…
0x007A1200
page table for program$_1$
0x003D0900
page table for program$_2$
0x00000000
…

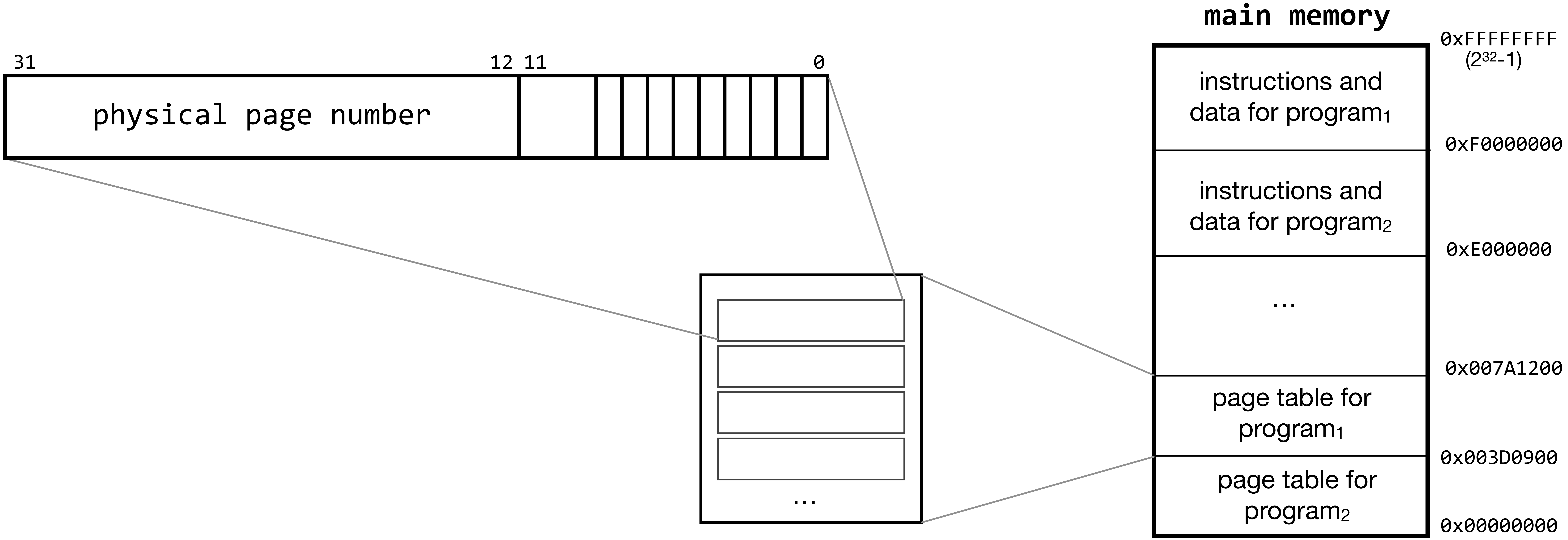
what happens if we don't have enough memory to store all of our programs' instructions and data?

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


main memory
0xFFFFFFFF
(2^{32}-1)
instructions and data for program$_1$
0xF0000000
instructions and data for program$_2$
0xE000000
…
0x007A1200
page table for program$_1$
0x003D0900
page table for program$_2$
0x00000000
…

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

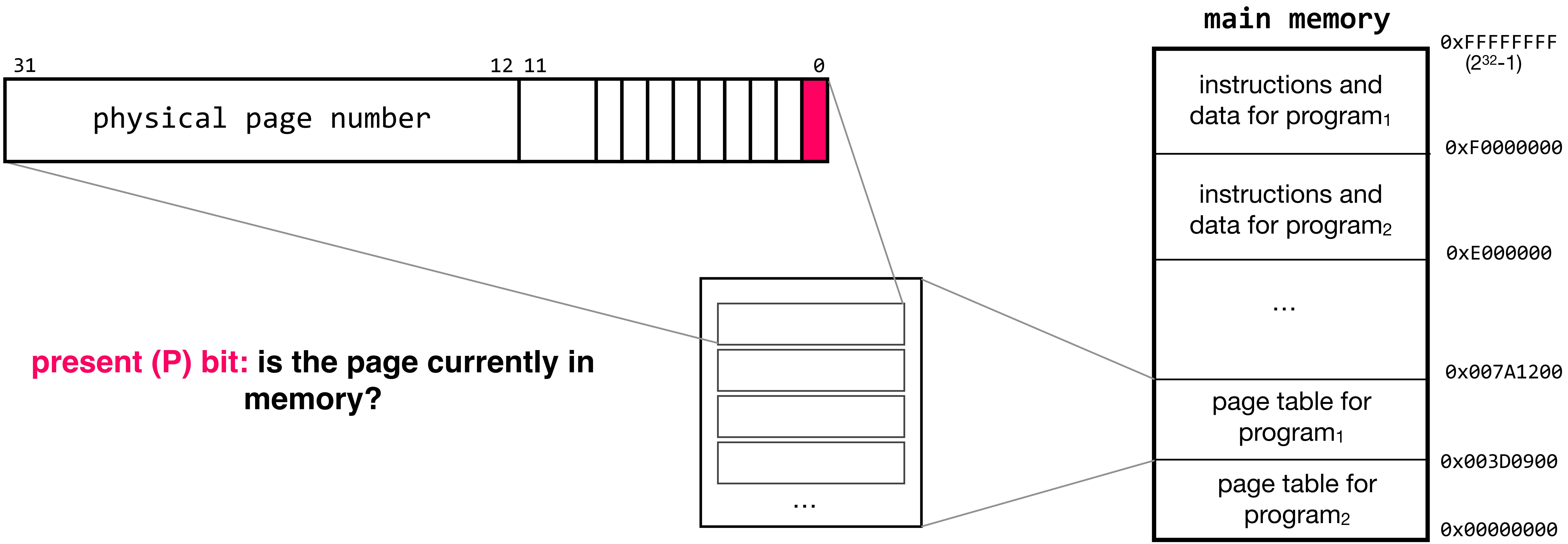
page table entries contain additional bits that help us deal with this problem (and others)



main memory
31
12 11
0
physical page number
...
0xFFFFFFFF
(2^{32}-1)
instructions and data for program$_1$
0xF0000000
instructions and data for program$_2$
0xE000000
...
0x007A1200
page table for program$_1$
0x003D0900
page table for program$_2$
0x00000000

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

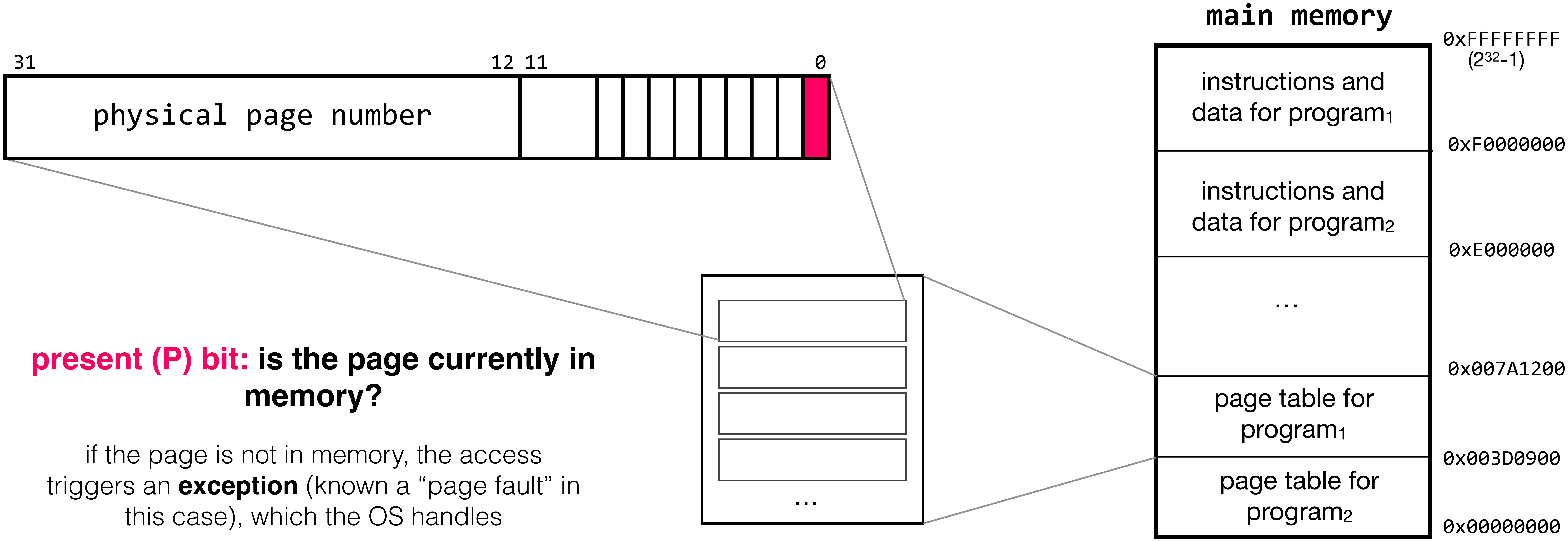
page table entries contain additional bits that help us deal with this problem (and others)



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

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)

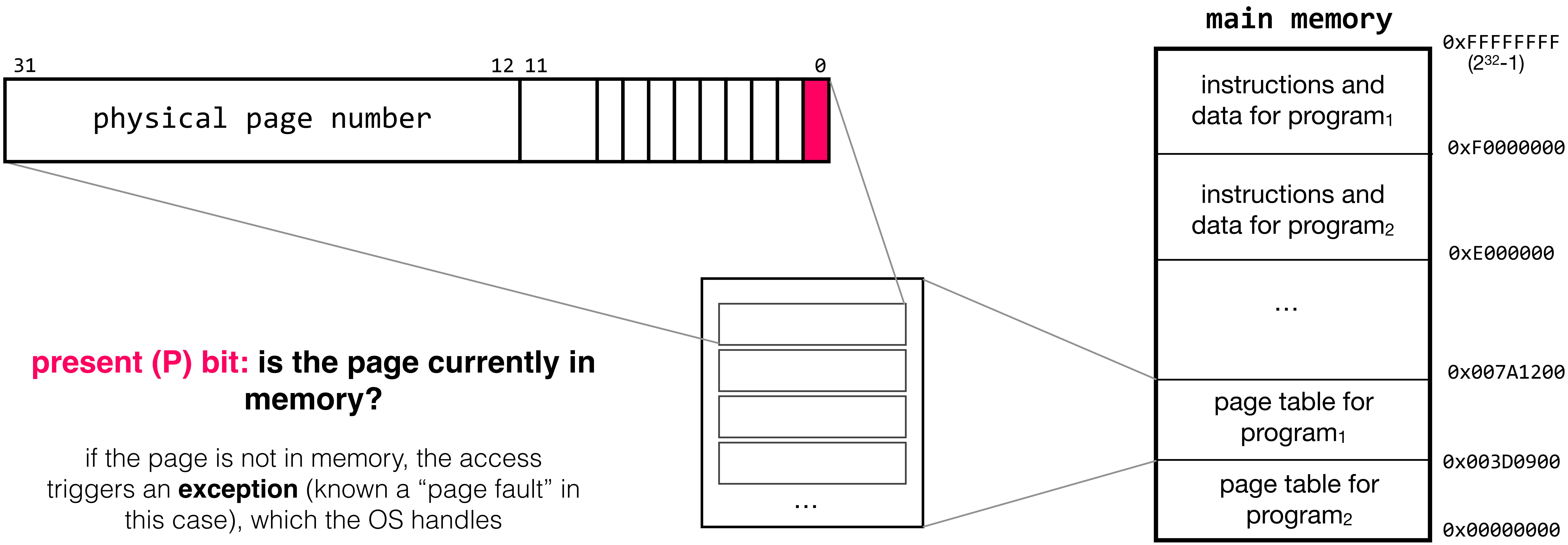


present (P) bit: is the page currently in memory?

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

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)


31
12 11
0
physical page number
main memory
0xFFFFFFFF
(2^{32}-1)
instructions and data for program$_1$
0xF0000000
instructions and data for program$_2$
0xE000000
…
0x007A1200
page table for program$_1$
0x003D0900
page table for program$_2$
0x00000000
…

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

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**

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 the handler for this particular exception
  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

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**

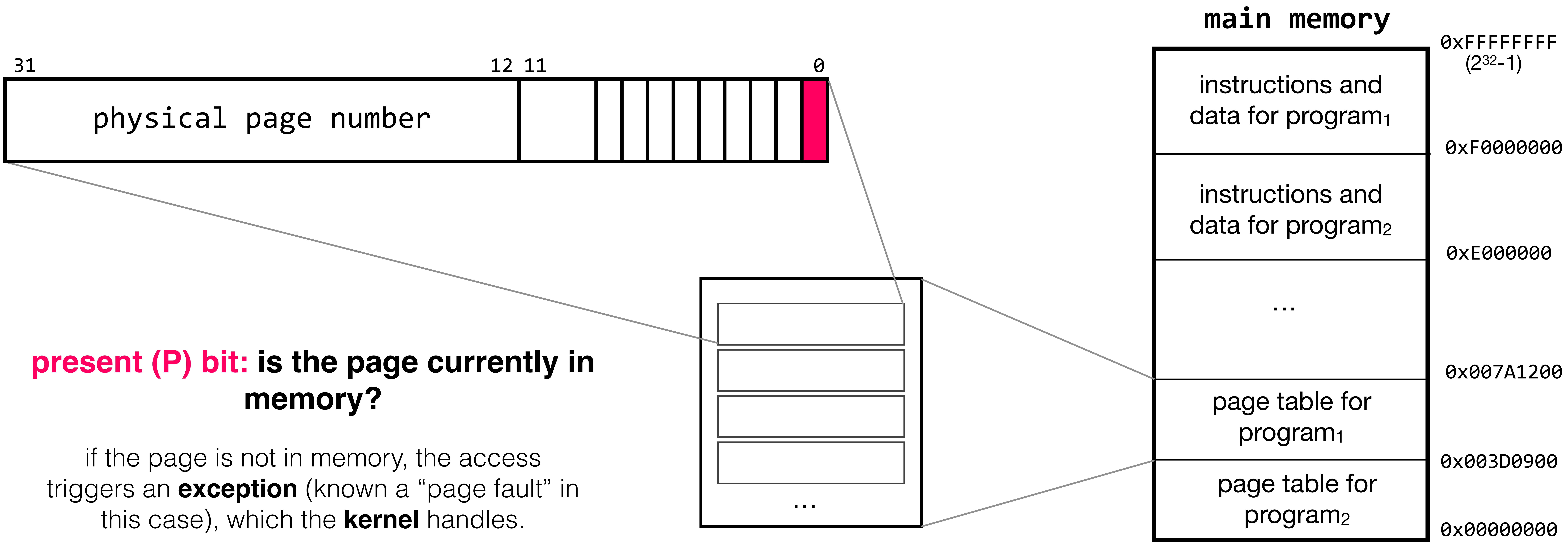
```
// special instruction that calls the exception handler for exception x
exception(x):
  U/K bit = K
  call handlers[x]
  U/K bit = U
```

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 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)

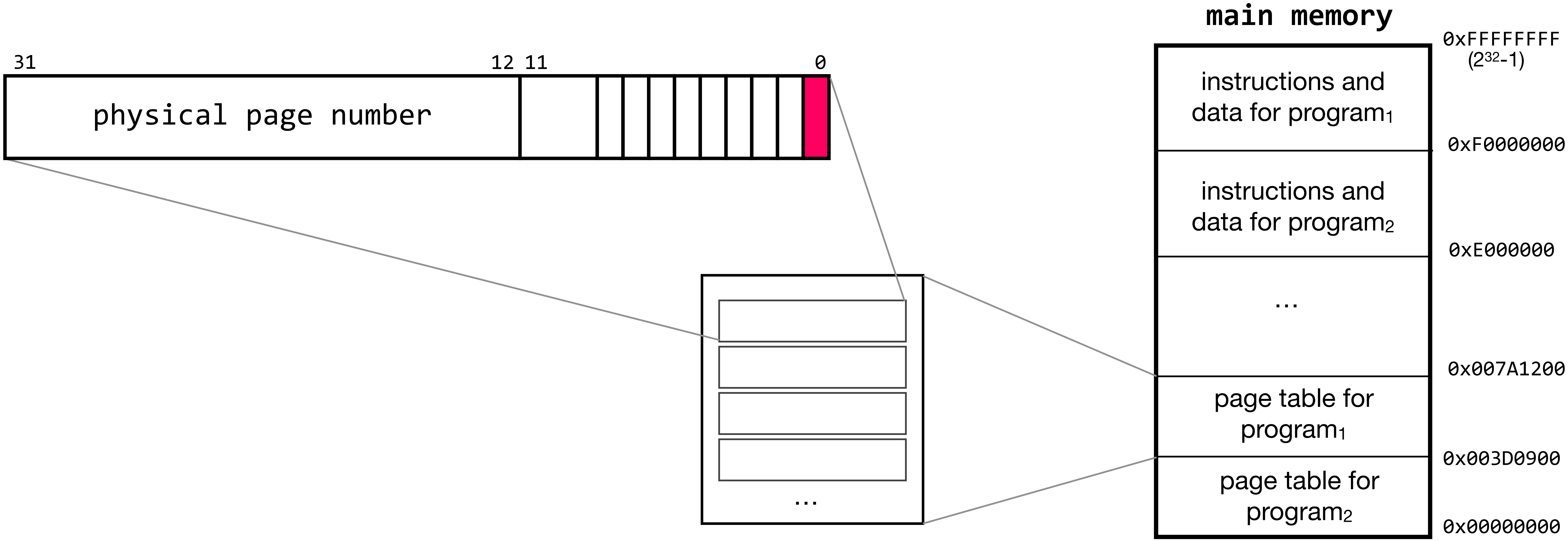


31
12 11
0
physical page number
...
main memory
0xFFFFFFFF
(2^{32}-1)
instructions and data for program$_1$
0xF0000000
instructions and data for program$_2$
0xE000000
...
0x007A1200
page table for program$_1$
0x003D0900
page table for program$_2$
0x00000000

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

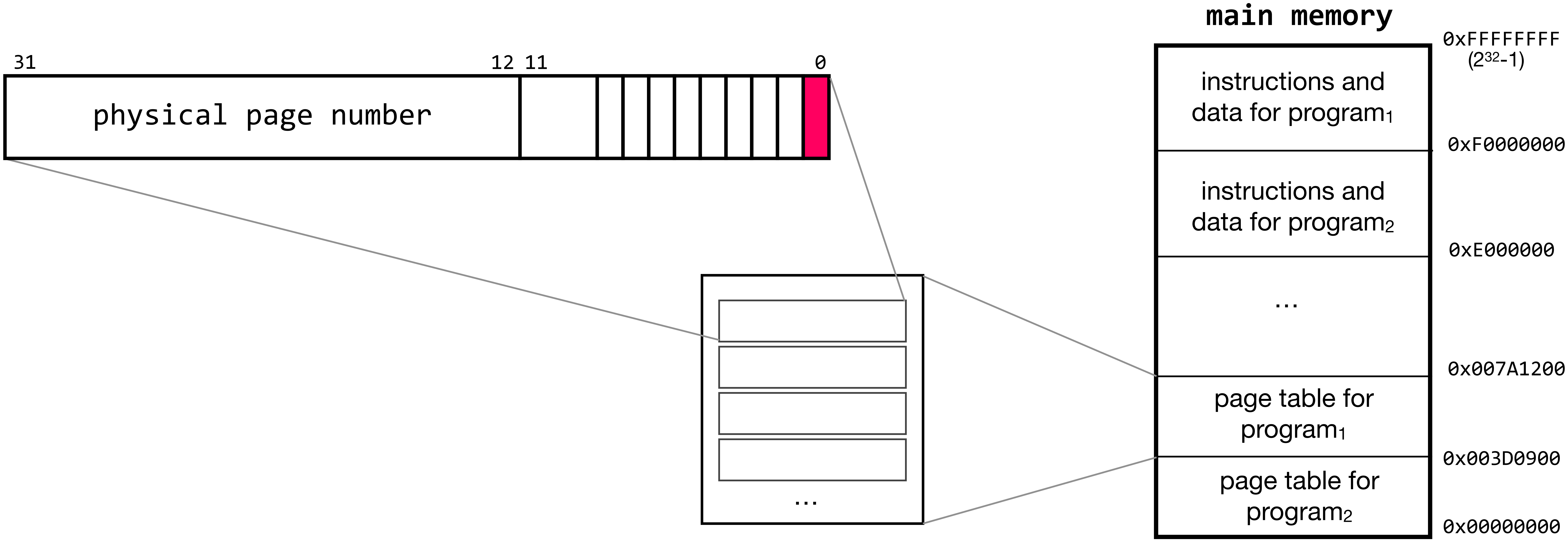
if the page is not in memory, the access triggers an **exception** (known a “page fault” in this case), which the **kernel** handles.

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



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

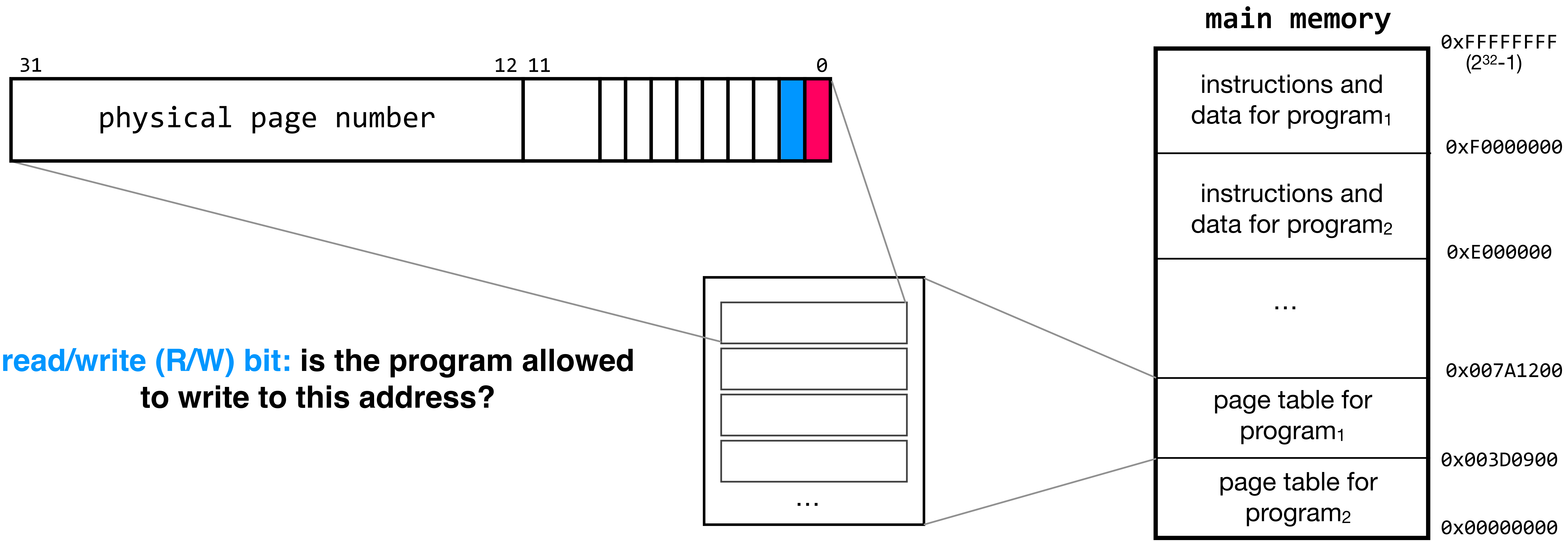
after all, it's conceivable that we want program$_1$ to be able to read some data, but not to modify it



main memory
31
12 11
0
physical page number
...
0xFFFFFFFF
(2^{32}-1)
instructions and data for program$_1$
0xF0000000
instructions and data for program$_2$
0xE000000
...
0x007A1200
page table for program$_1$
0x003D0900
page table for program$_2$
0x00000000

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

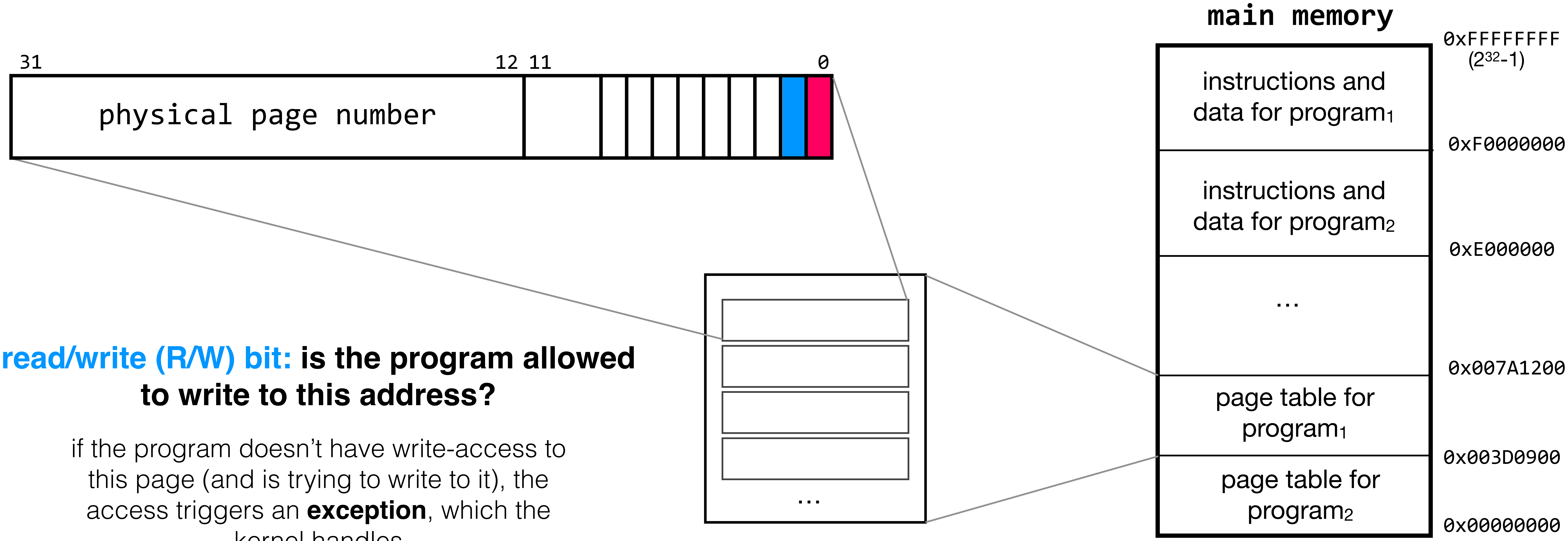
after all, it's conceivable that we want program$_1$ to be able to read some data, but not to modify it


31
12 11
0
physical page number
main memory
0xFFFFFFFF
(2^{32}-1)
instructions and data for program$_1$
0xF0000000
instructions and data for program$_2$
0xE000000
...
0x007A1200
page table for program$_1$
0x003D0900
page table for program$_2$
0x00000000
...

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

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

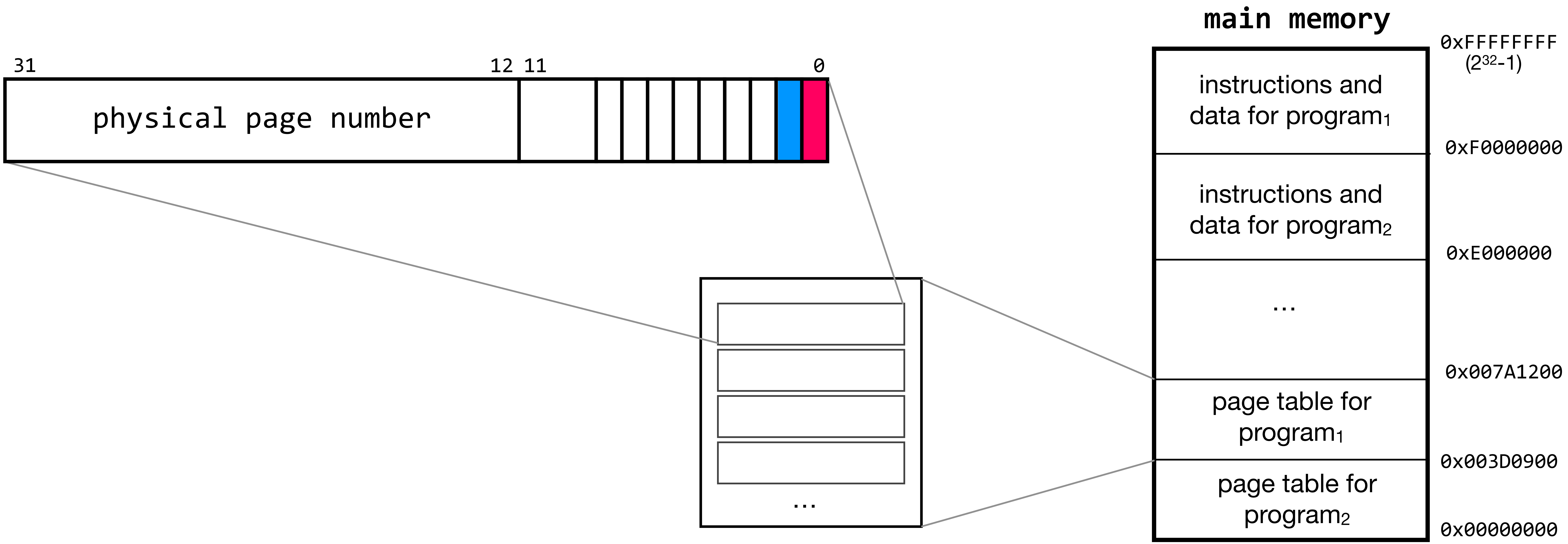


main memory
31
12 11
0
physical page number
0xFFFFFFFF (2^{32}-1)
instructions and data for program$_1$
0xF0000000
instructions and data for program$_2$
0xE000000
…
0x007A1200
page table for program$_1$
0x003D0900
page table for program$_2$
0x00000000
…

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

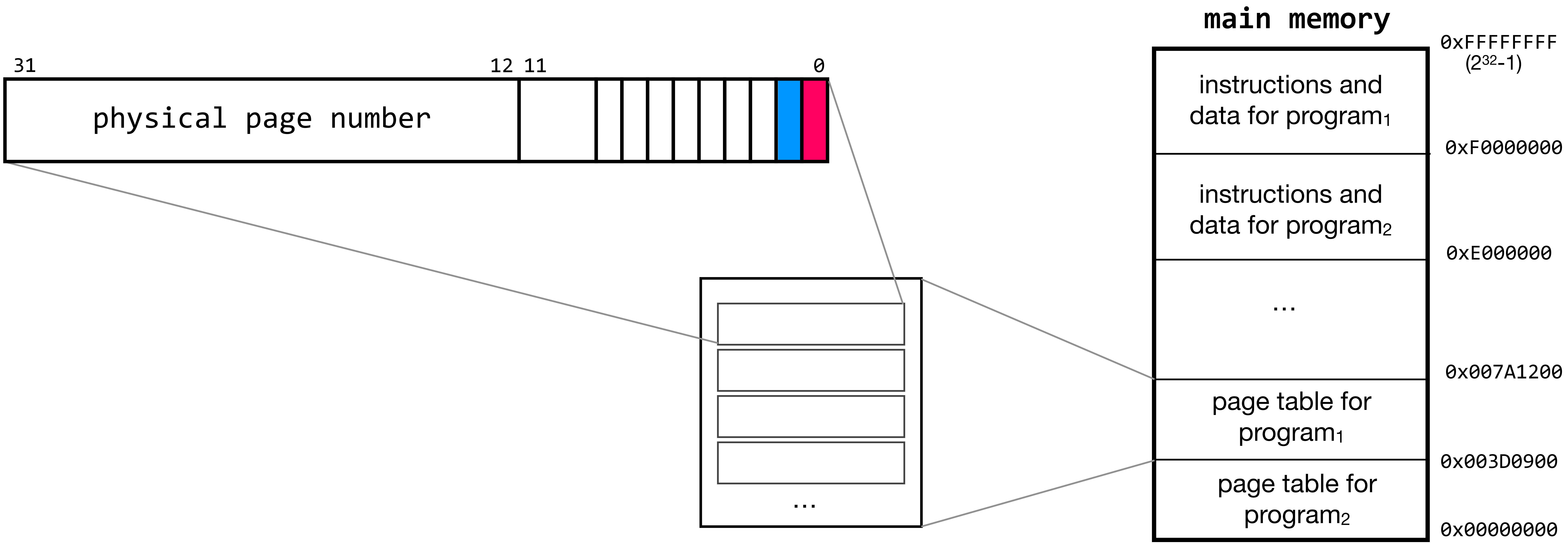
if the program doesn't have write-access to this page (and is trying to write to it), the access triggers an **exception**, which the kernel handles

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



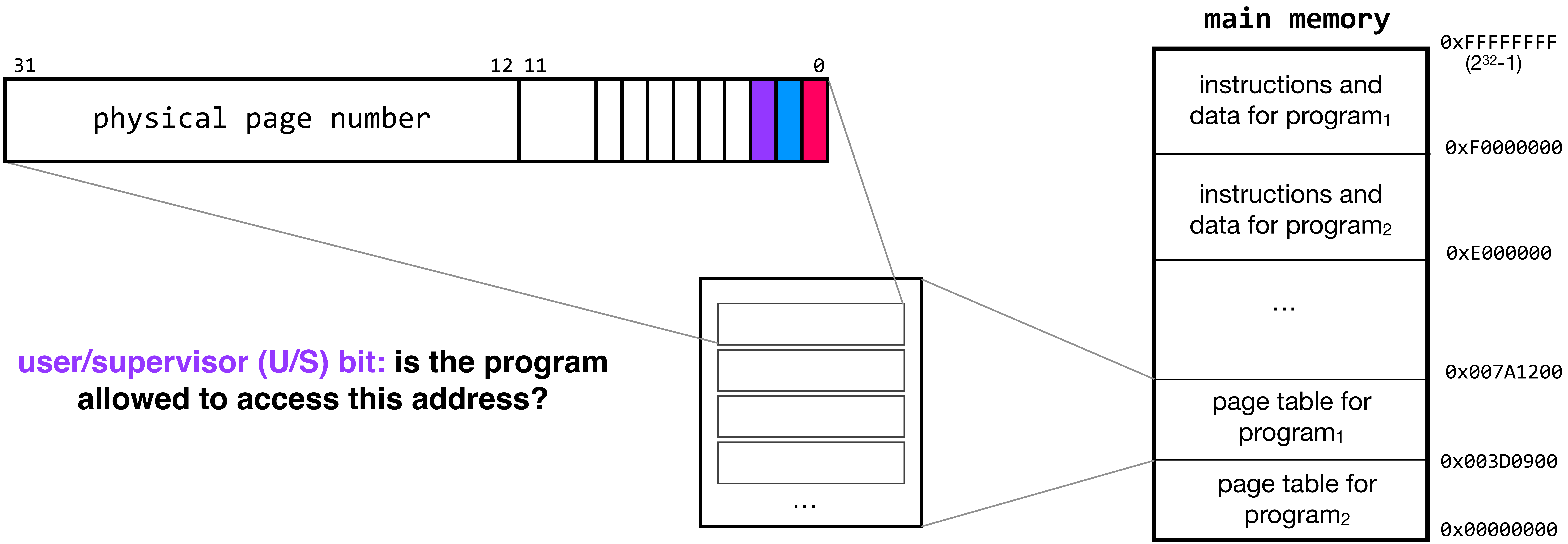
what happens if a program tries to access memory that only the kernel should have access to?

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


31
12 11
0
physical page number
...
main memory
0xFFFFFFFF
(2^{32}-1)
instructions and data for program$_1$
0xF0000000
instructions and data for program$_2$
0xE000000
...
0x007A1200
page table for program$_1$
0x003D0900
page table for program$_2$
0x00000000

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

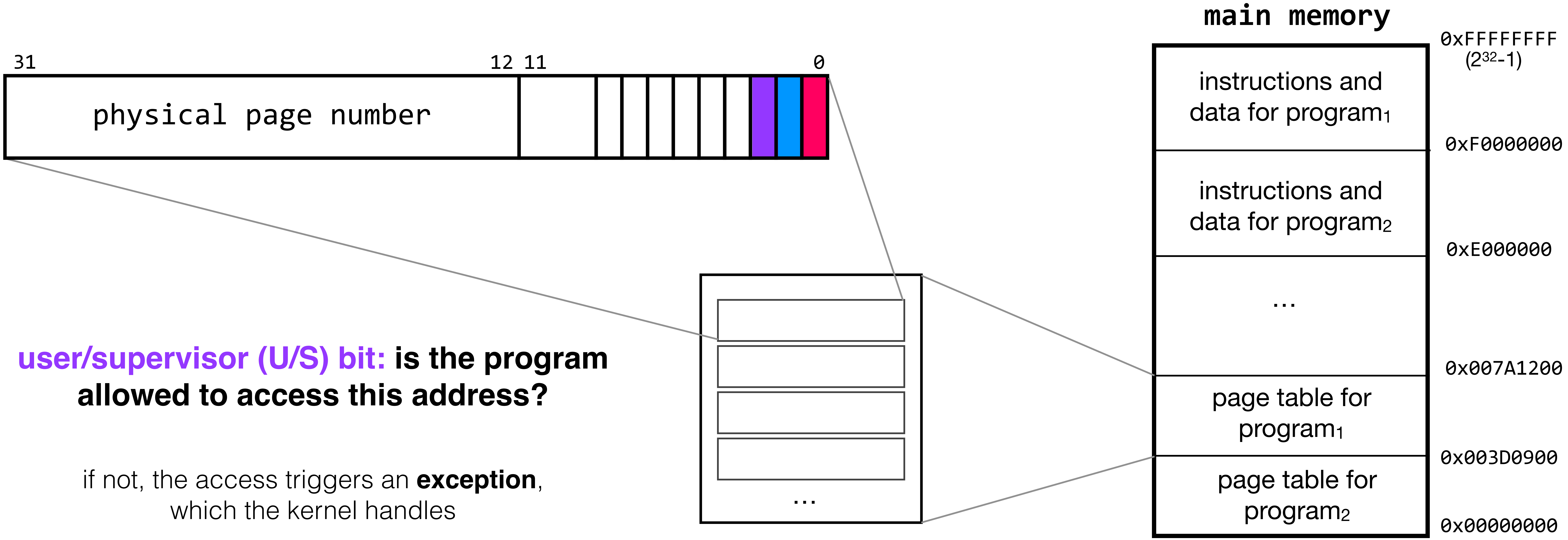
we need to enforce modularity between programs and the kernel, not just between programs


31
12
11
0
physical page number
main memory
0xFFFFFFFF
(2^{32}-1)
instructions and data for program$_1$
0xF0000000
instructions and data for program$_2$
0xE000000
...
0x007A1200
page table for program$_1$
0x003D0900
page table for program$_2$
0x00000000
...

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

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



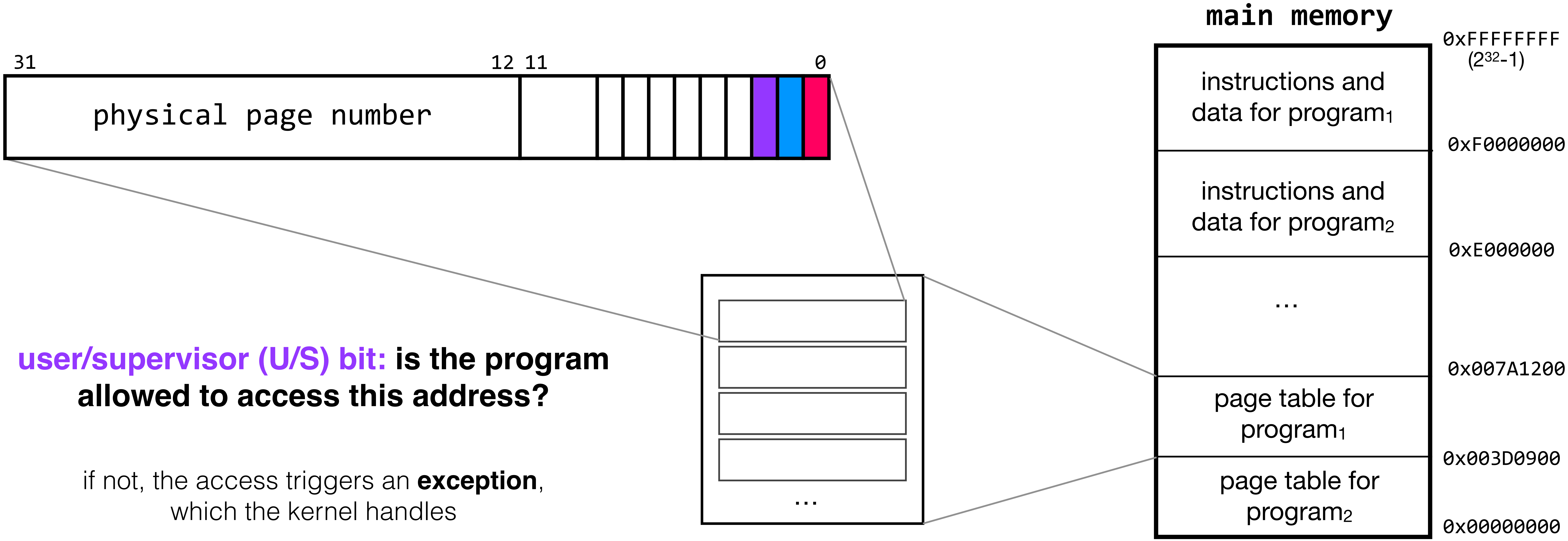
main memory
31
12 11
0
physical page number
0xFFFFFFFF
$(2^{32}-1)$
instructions and data for program$_1$
0xF0000000
instructions and data for program$_2$
0xE000000
…
0x007A1200
page table for program$_1$
0x003D0900
page table for program$_2$
0x00000000
…

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

if not, 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

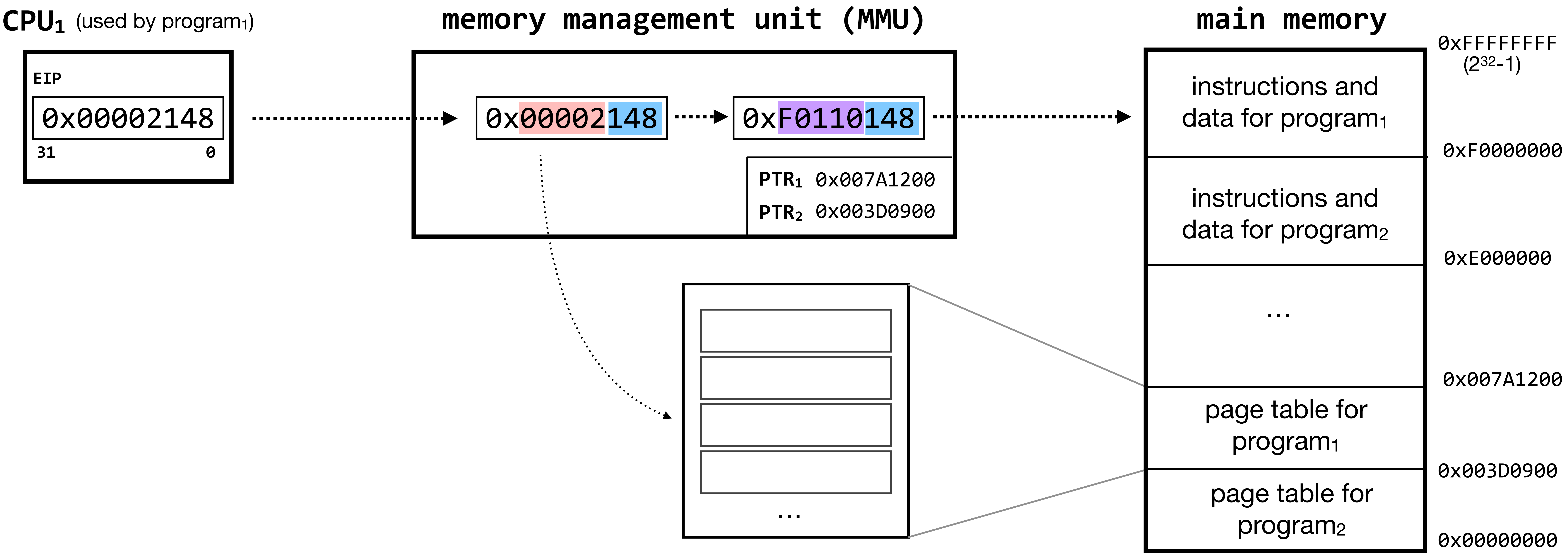


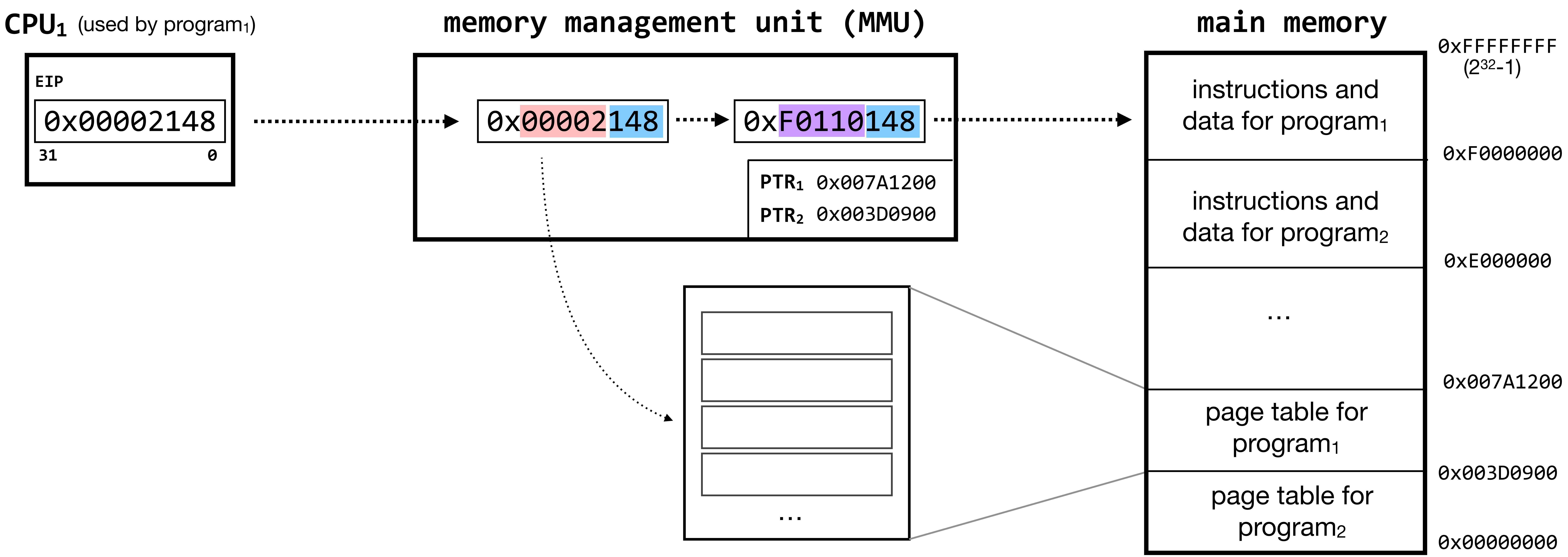
main memory
31
12 11
0
physical page number
0xFFFFFFFF
(2^{32}-1)
instructions and data for program$_1$
0xF0000000
instructions and data for program$_2$
0xE000000
…
0x007A1200
page table for program$_1$
0x003D0900
page table for program$_2$
0x00000000
…

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

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

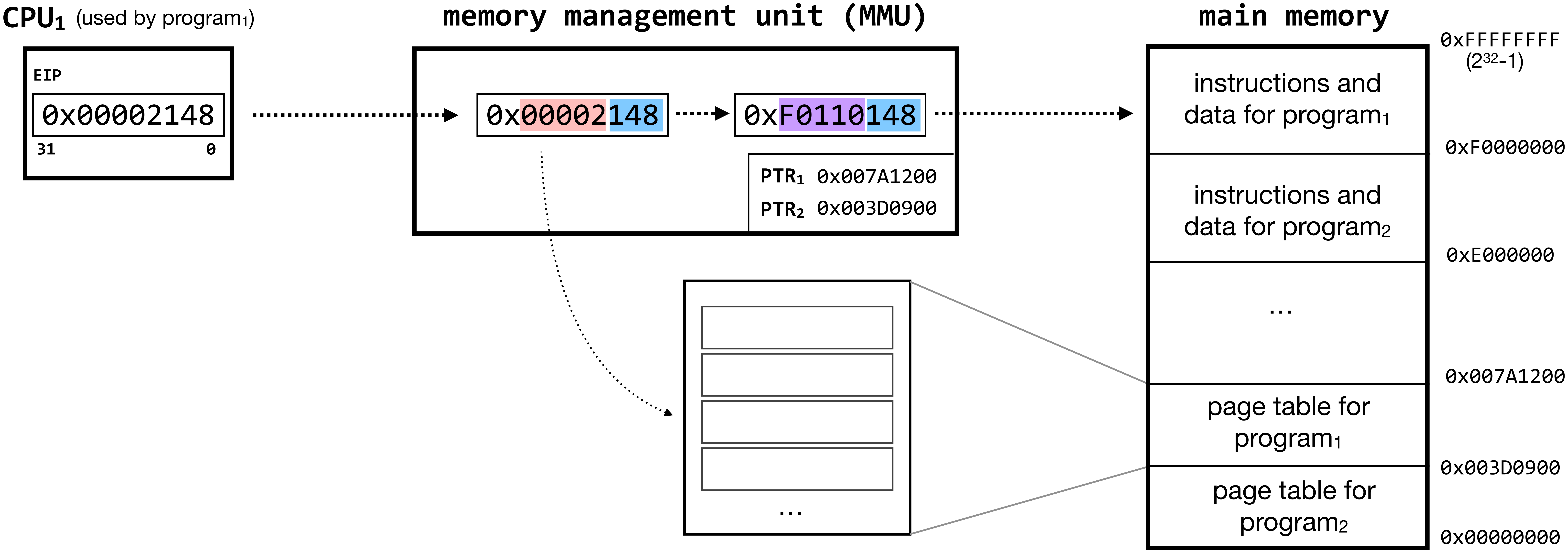
without this last piece, a determined program could still attempt to circumvent modularity by doing things such as modifying the page-table registers


CPU₁ (used by program₁)
EIP
0x00002148
31
0
memory management unit (MMU)
0x00002148
0xF0110148
PTR₁ 0x007A1200
PTR₂ 0x003D0900
...
main memory
0xFFFFFFFF
(2³²-1)
instructions and data for program₁
0xF0000000
instructions and data for program₂
0xE000000
...
0x007A1200
page table for program₁
0x003D0900
page table for program₂
0x00000000


CPU$_1$ (used by program$_1$)
EIP
0x00002148
31
0
memory management unit (MMU)
0x00002148
0xF0110148
PTR$_1$ 0x007A1200
PTR$_2$ 0x003D0900
...
main memory
0xFFFFFFFF
(2^{32}-1)
instructions and data for program$_1$
0xF0000000
instructions and data for program$_2$
0xE000000
...
0x007A1200
page table for program$_1$
0x003D0900
page table for program$_2$
0x00000000

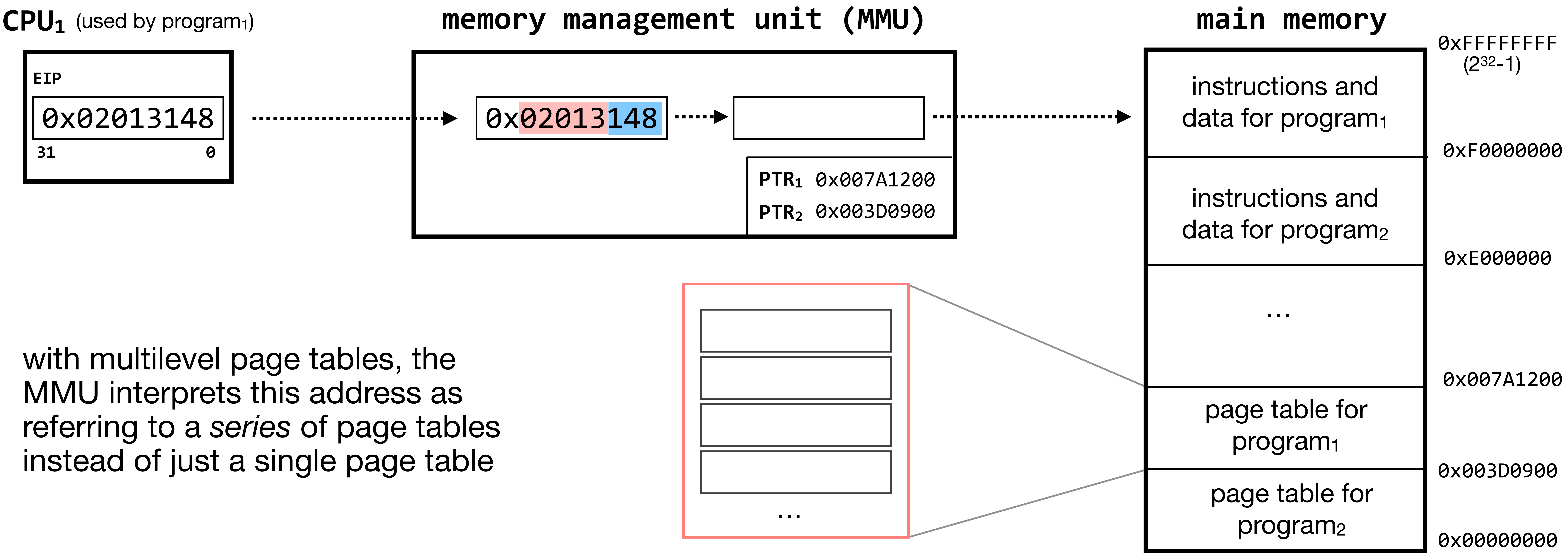
2^{20} virtual addresses each mapping to a 32-bit page-table entry (PTE)
→ **4MB to store this table**

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


CPU$_1$ (used by program$_1$)
EIP
0x00002148
31
0
memory management unit (MMU)
0x00002148
0xF0110148
PTR$_1$ 0x007A1200
PTR$_2$ 0x003D0900
…
main memory
0xFFFFFFFF
(2^{32}-1)
instructions and data for program$_1$
0xF0000000
instructions and data for program$_2$
0xE000000
…
0x007A1200
page table for program$_1$
0x003D0900
page table for program$_2$
0x00000000

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

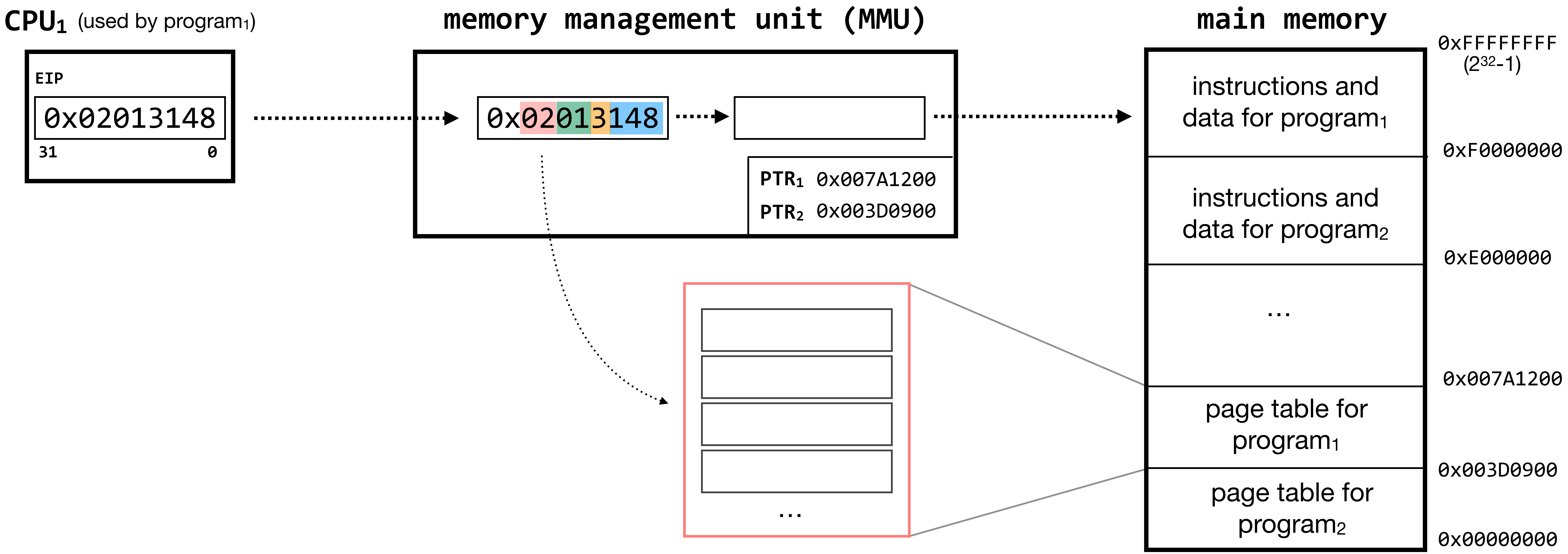
multilevel page tables often use less space



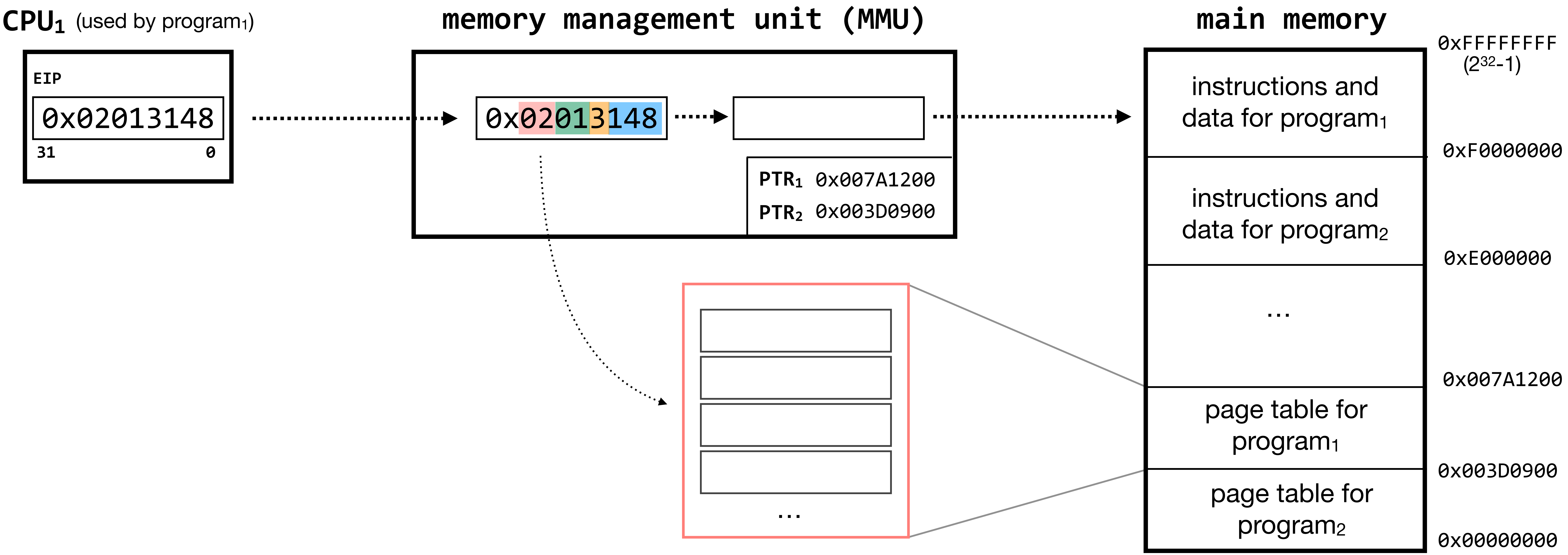
CPU$_1$ (used by program$_1$)
EIP
0x02013148
31
0
memory management unit (MMU)
0x02013148
PTR$_1$ 0x007A1200
PTR$_2$ 0x003D0900
...
main memory
0xFFFFFFFF
(2^{32}-1)
instructions and data for program$_1$
0xF0000000
instructions and data for program$_2$
0xE000000
...
0x007A1200
page table for program$_1$
0x003D0900
page table for program$_2$
0x00000000

with multilevel page tables, the MMU interprets this address as referring to a *series* of page tables instead of just a single page table

multilevel page tables often use less space

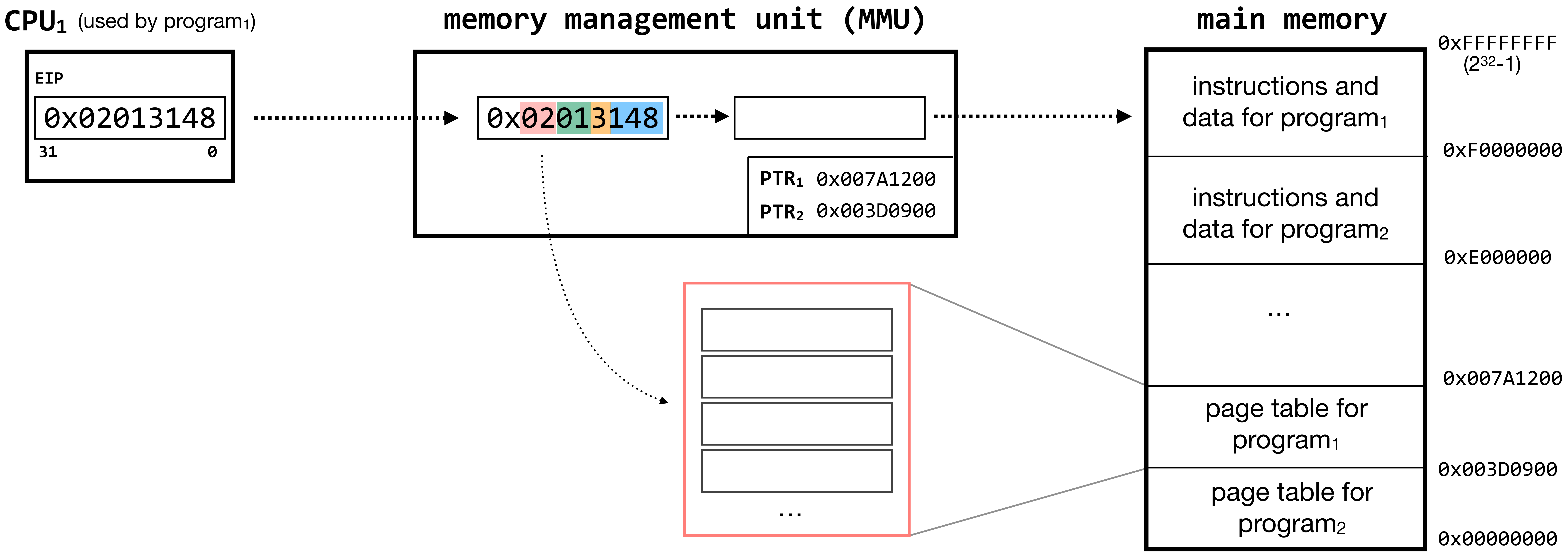


multilevel page tables often use less space



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}

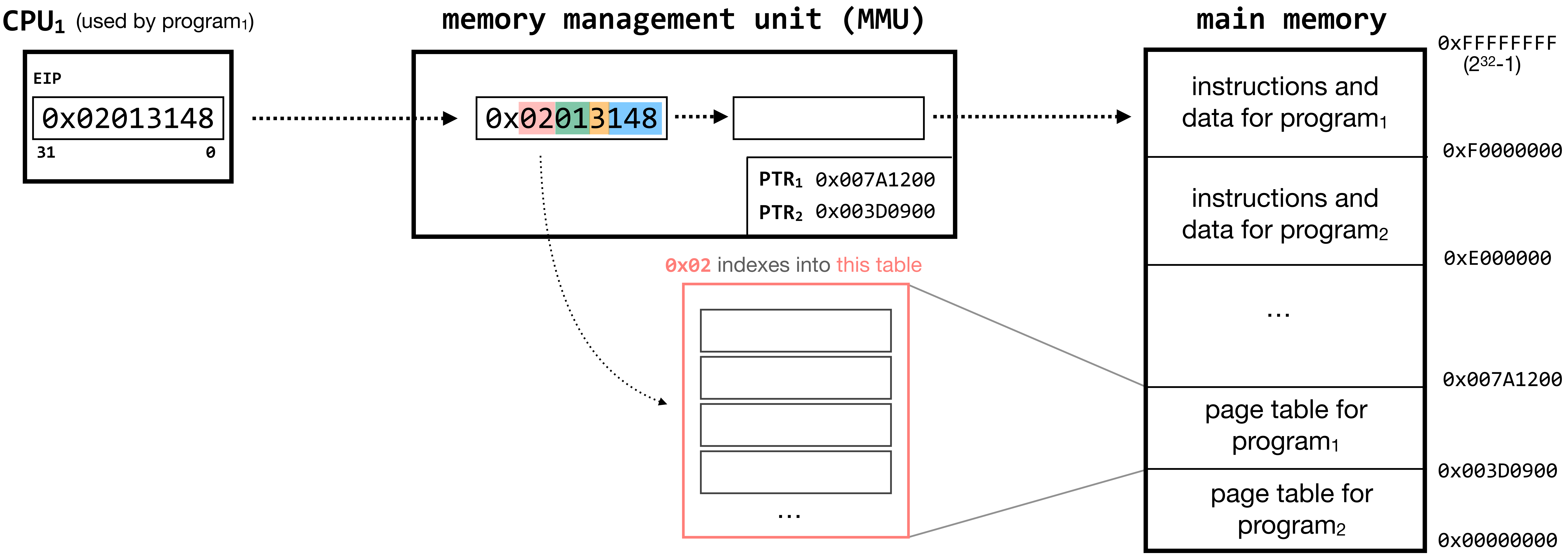
multilevel page tables often use less space



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)

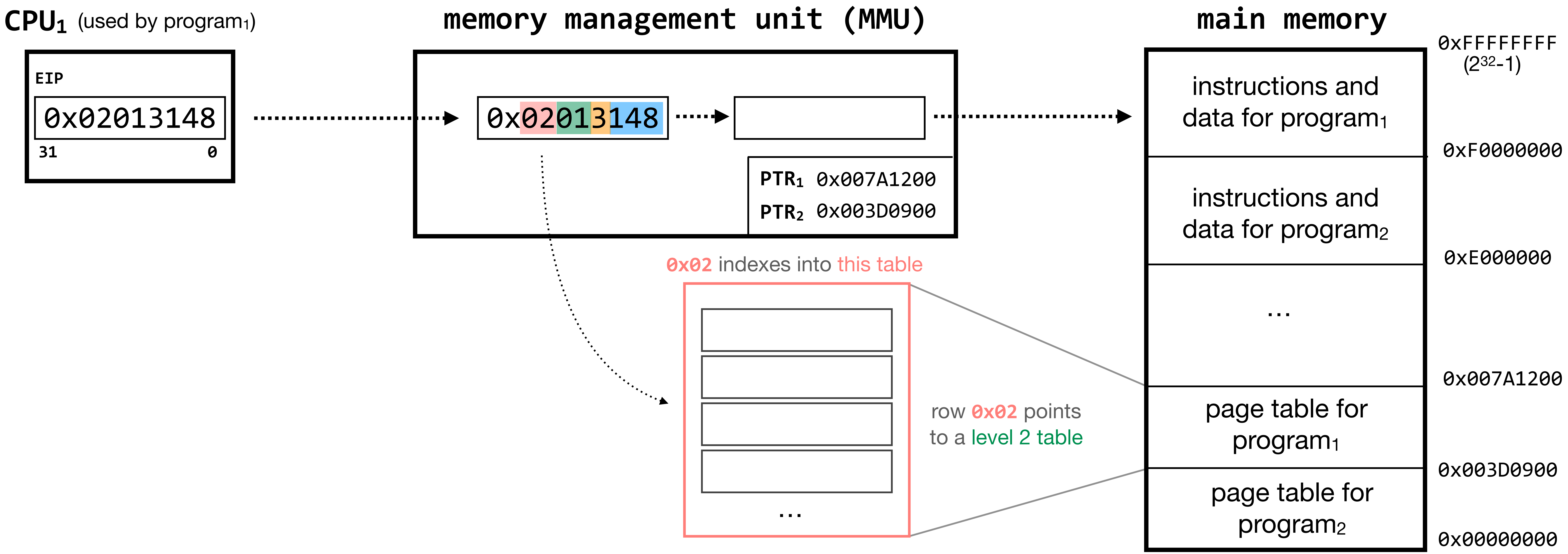
multilevel page tables often use less space



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)

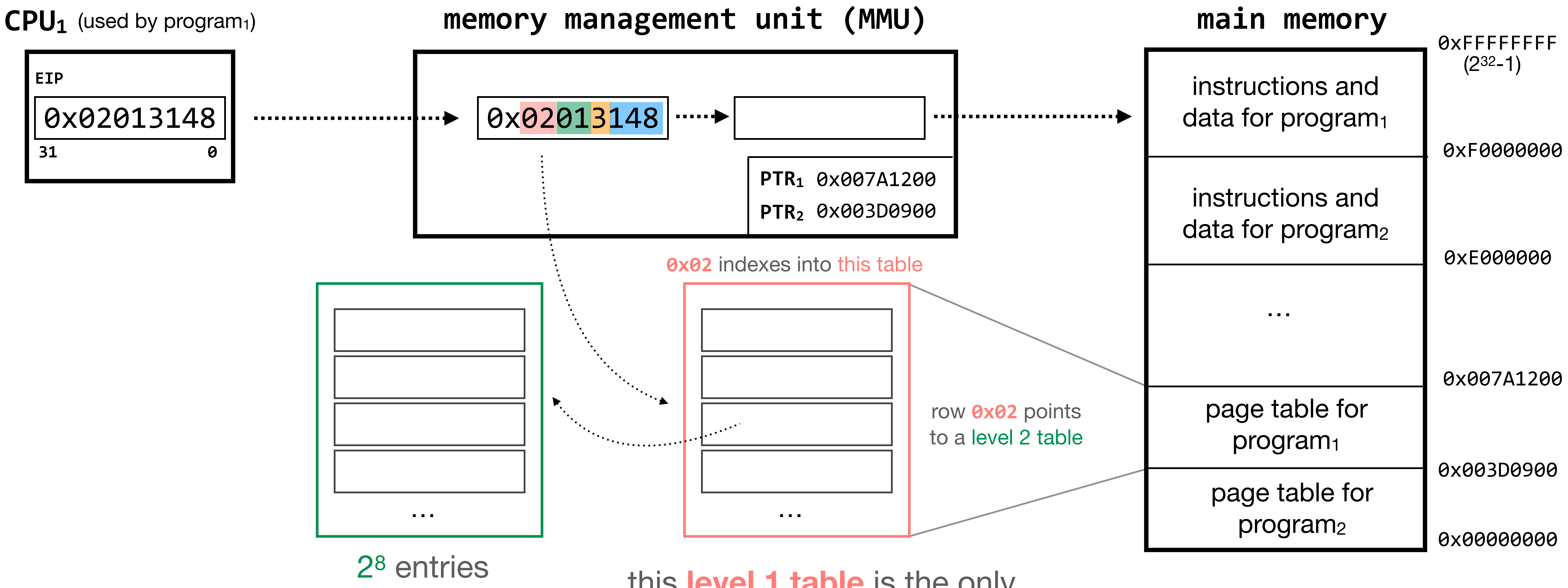
multilevel page tables often use less space



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)

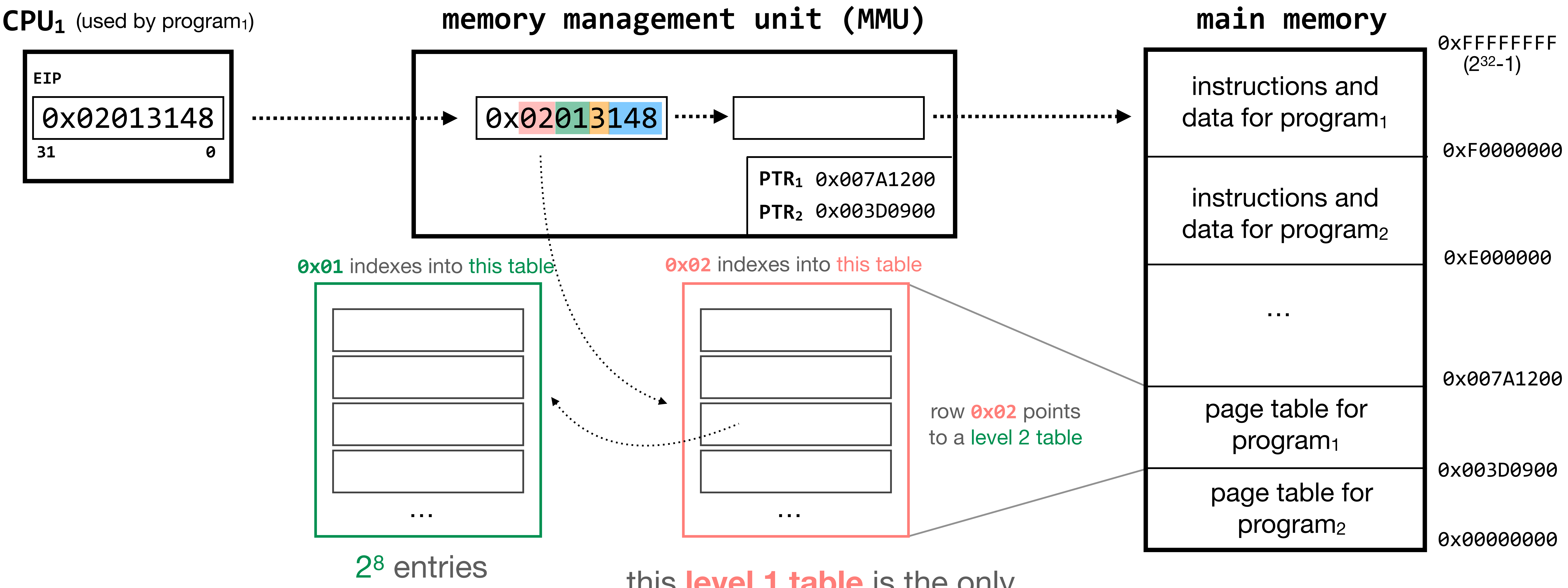
multilevel page tables often use less space



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)

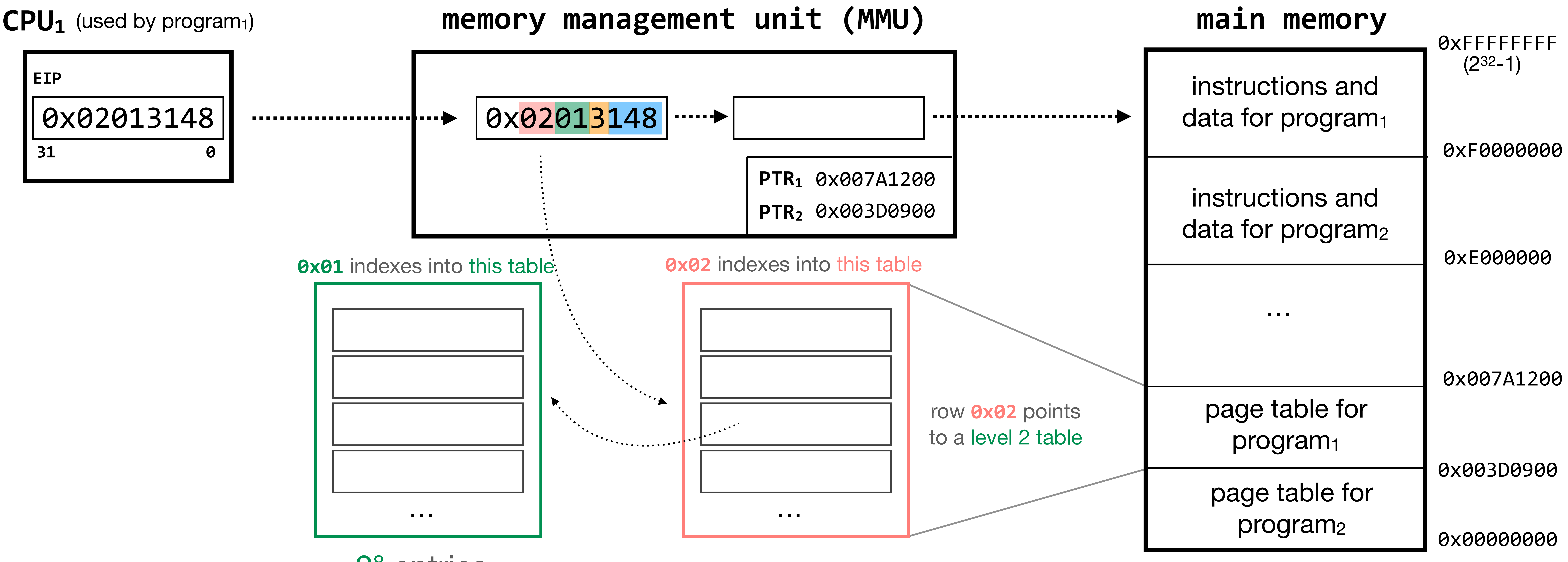
multilevel page tables often use less space



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



CPU₁ (used by program₁)

EIP

0x02013148

31 0

memory management unit (MMU)

0x02013148

PTR₁ 0x007A1200

PTR₂ 0x003D0900

main memory

instructions and data for program₁

instructions and data for program₂

...

page table for program₁

page table for program₂

0xFFFFFFFF (2^{32}-1)

0xF0000000

0xE000000

0x007A1200

0x003D0900

0x00000000

0x01 indexes into this table

0x02 indexes into this table

row 0x02 points to a level 2 table

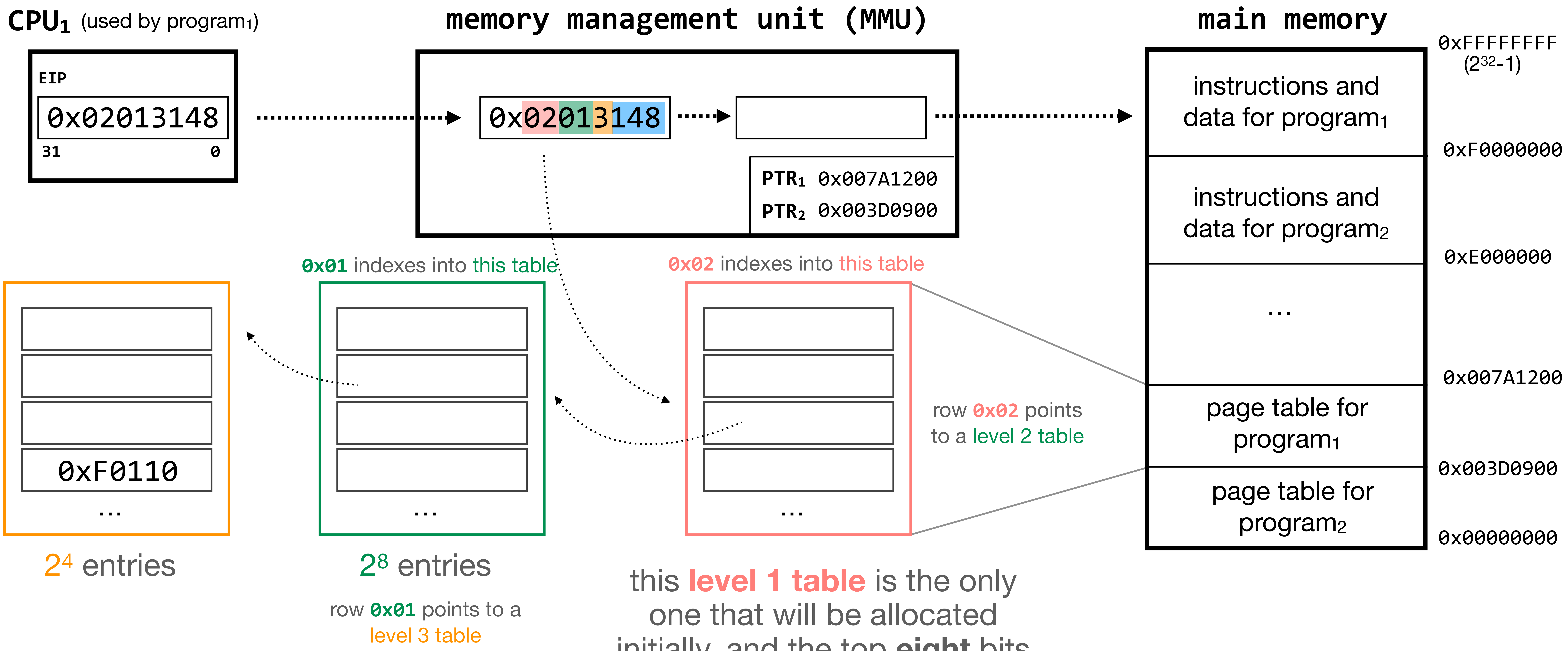
2^8 entries

row 0x01 points to a level 3 table

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



CPU$_1$ (used by program$_1$)

EIP

0x02013148

31 0

memory management unit (MMU)

0x02013148

PTR$_1$ 0x007A1200

PTR$_2$ 0x003D0900

main memory

instructions and data for program$_1$	0xFFFFFFFF (2^{32}-1)
instructions and data for program$_2$	0xF0000000
…	0xE000000
page table for program$_1$	0x007A1200
page table for program$_2$	0x003D0900
	0x00000000

0x01 indexes into this table

0x02 indexes into this table

0xF0110

…

2^4 entries

2^8 entries

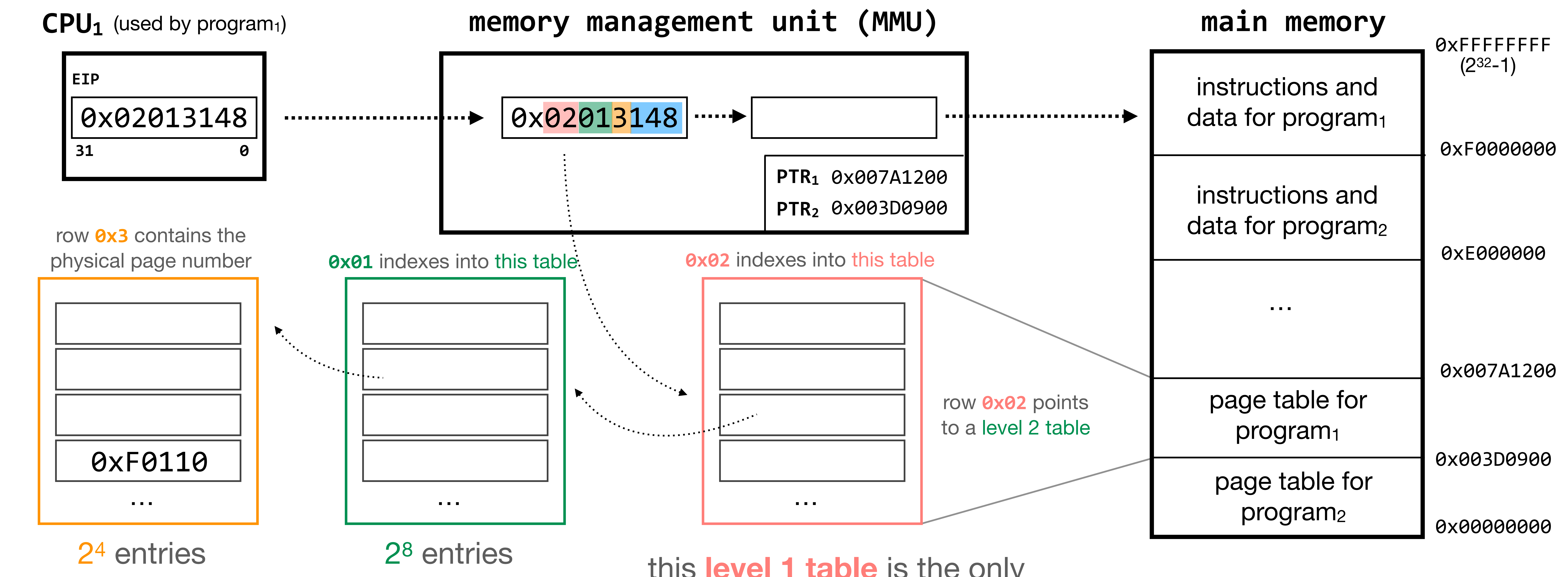
row **0x01** points to a level 3 table

row **0x02** points to a level 2 table

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



$\mathtt{CPU_1}$ (used by program_1)

EIP

0x02013148

31 0

memory management unit (MMU)

0x02013148

$\mathtt{PTR_1}$ 0x007A1200

$\mathtt{PTR_2}$ 0x003D0900

main memory

0xFFFFFFFF (2^{32}-1)

instructions and data for program_1

0xF0000000

instructions and data for program_2

0xE000000

...

0x007A1200

page table for program_1

0x003D0900

page table for program_2

0x00000000

row **0x3** contains the physical page number

0xF0110

...

2^4 entries

0x01 indexes into this table

...

2^8 entries

0x02 indexes into this table

...

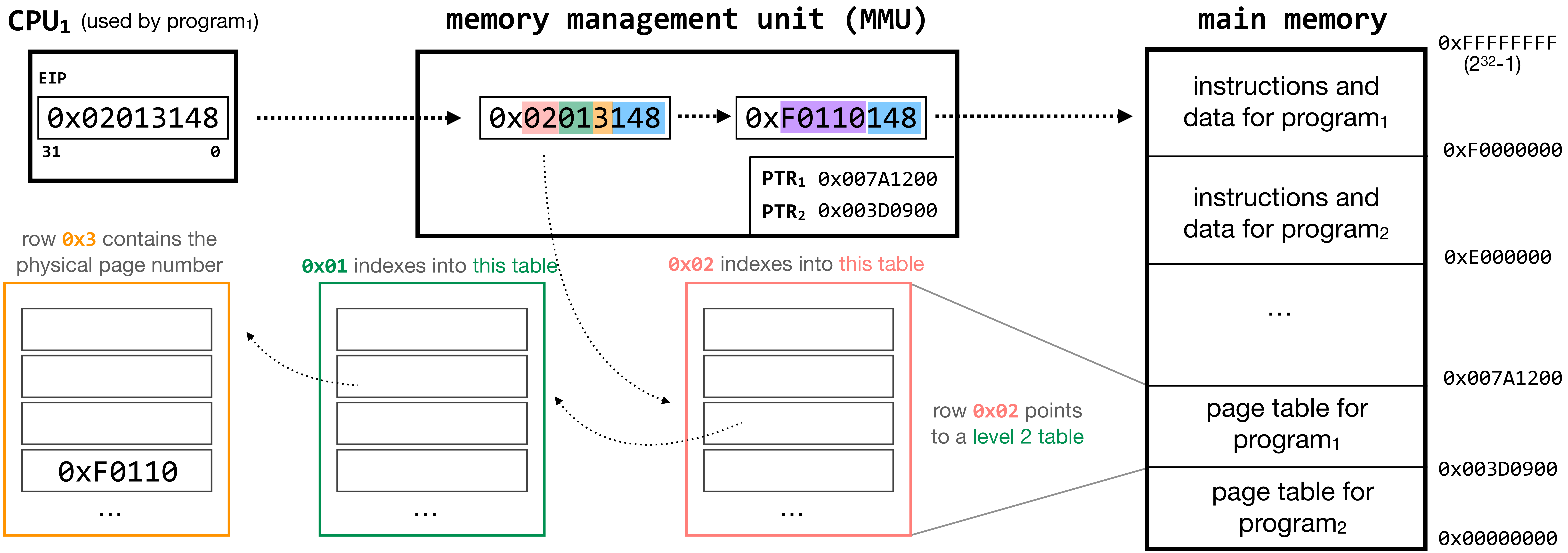
row **0x02** points to a level 2 table

row **0x01** points to a level 3 table

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


CPU$_1$ (used by program$_1$)
EIP
0x02013148
31
0
memory management unit (MMU)
0x02013148
0xF0110148
PTR$_1$ 0x007A1200
PTR$_2$ 0x003D0900
main memory
0xFFFFFFFF
(2^{32}-1)
instructions and data for program$_1$
0xF0000000
instructions and data for program$_2$
0xE000000
...
0x007A1200
page table for program$_1$
0x003D0900
page table for program$_2$
0x00000000
row 0x3 contains the physical page number
0x01 indexes into this table
0x02 indexes into this table
0xF0110
...
...
...
row 0x02 points to a level 2 table

2^4 entries

2^8 entries

row **0x01** points to a level 3 table

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)

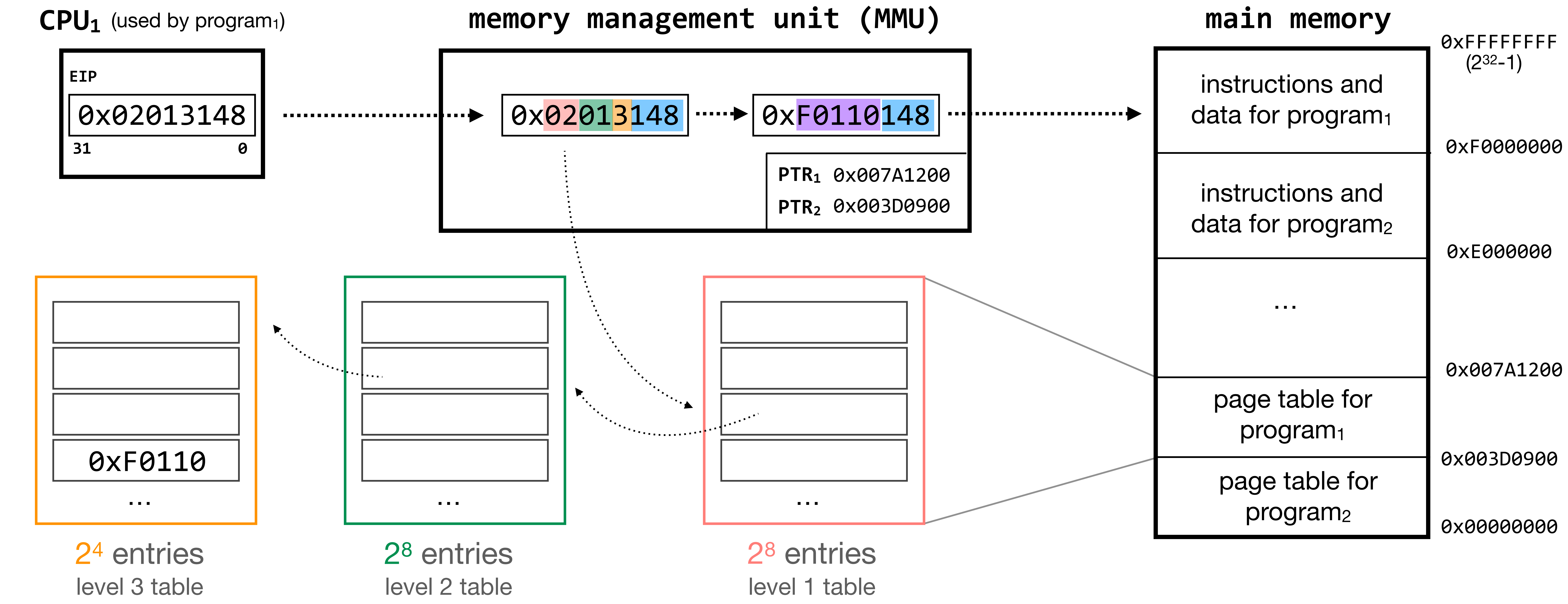
multilevel page tables often use less space

CPU₁ (used by program₁)
EIP
0x02013148
31
0
memory management unit (MMU)
0x02013148
0xF0110148
PTR₁ 0x007A1200
PTR₂ 0x003D0900
main memory
0xFFFFFFFF
(2³²-1)
instructions and data for program₁
0xF0000000
instructions and data for program₂
0xE000000
...
0x007A1200
page table for program₁
0x003D0900
page table for program₂
0x00000000
0xF0110
...
2⁴ entries
level 3 table
...
2⁸ entries
level 2 table
...
2⁸ entries
level 1 table

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)

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



CPU$_1$ (used by program$_1$)

EIP

0x02013148

31 0

memory management unit (MMU)

0x02013148 → 0xF0110148

PTR$_1$ 0x007A1200

PTR$_2$ 0x003D0900

main memory

0xFFFFFFFF (2^{32}-1)

instructions and data for program$_1$

0xF0000000

instructions and data for program$_2$

0xE000000

…

0x007A1200

page table for program$_1$

0x003D0900

page table for program$_2$

0x00000000

0xF0110

…

2^4 entries

level 3 table

…

2^8 entries

level 2 table

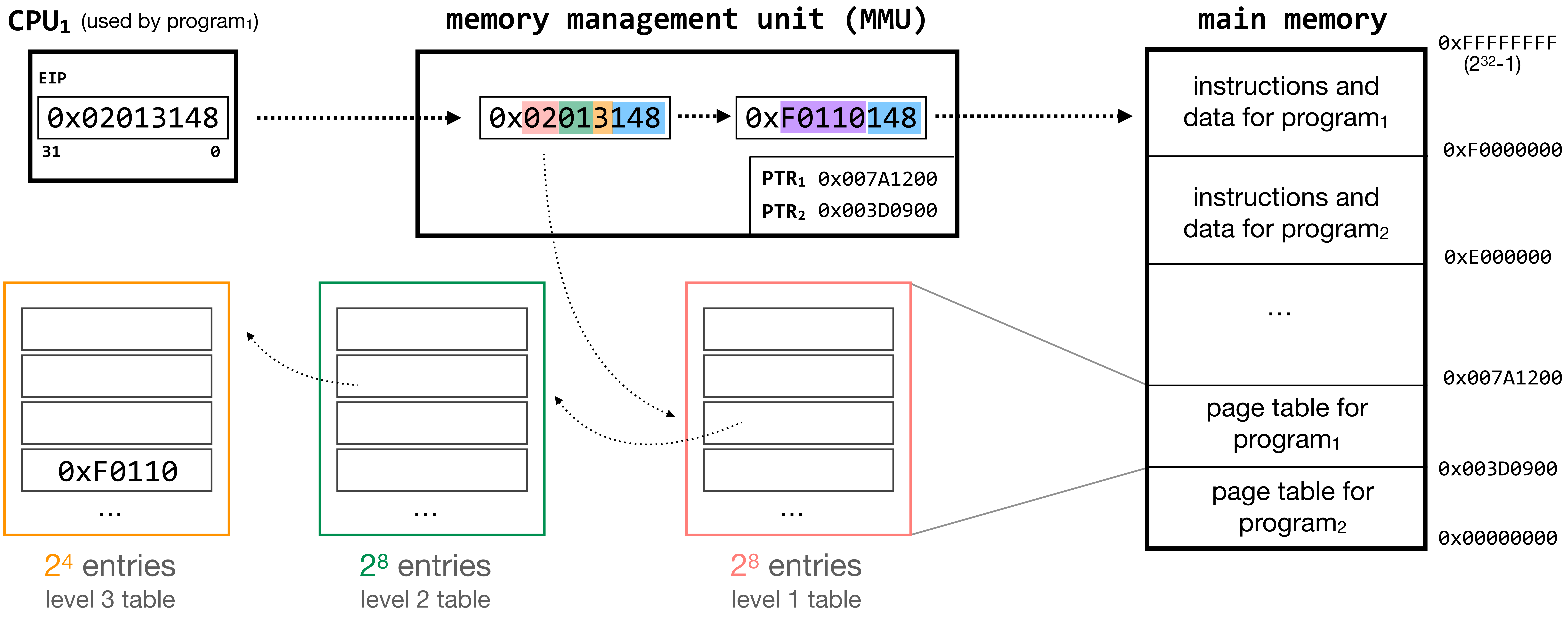
…

2^8 entries

level 1 table

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)



CPU$_1$ (used by program$_1$)

EIP

0x02013148

31 0

memory management unit (MMU)

0x02013148

0xF0110148

PTR$_1$ 0x007A1200

PTR$_2$ 0x003D0900

main memory

0xFFFFFFFF (2^{32}-1)

instructions and data for program$_1$

0xF0000000

instructions and data for program$_2$

0xE000000

…

0x007A1200

page table for program$_1$

0x003D0900

page table for program$_2$

0x00000000

0xF0110

…

2^4 entries

level 3 table

…

2^8 entries

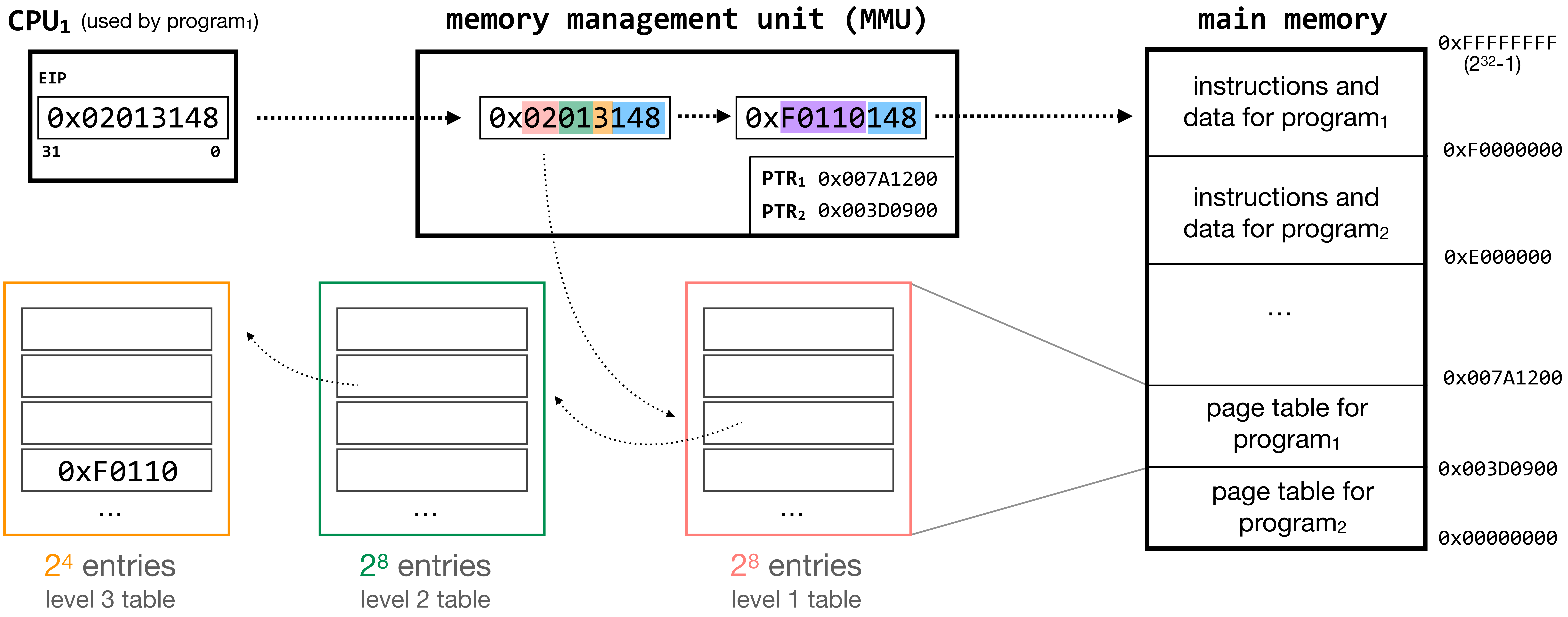
level 2 table

…

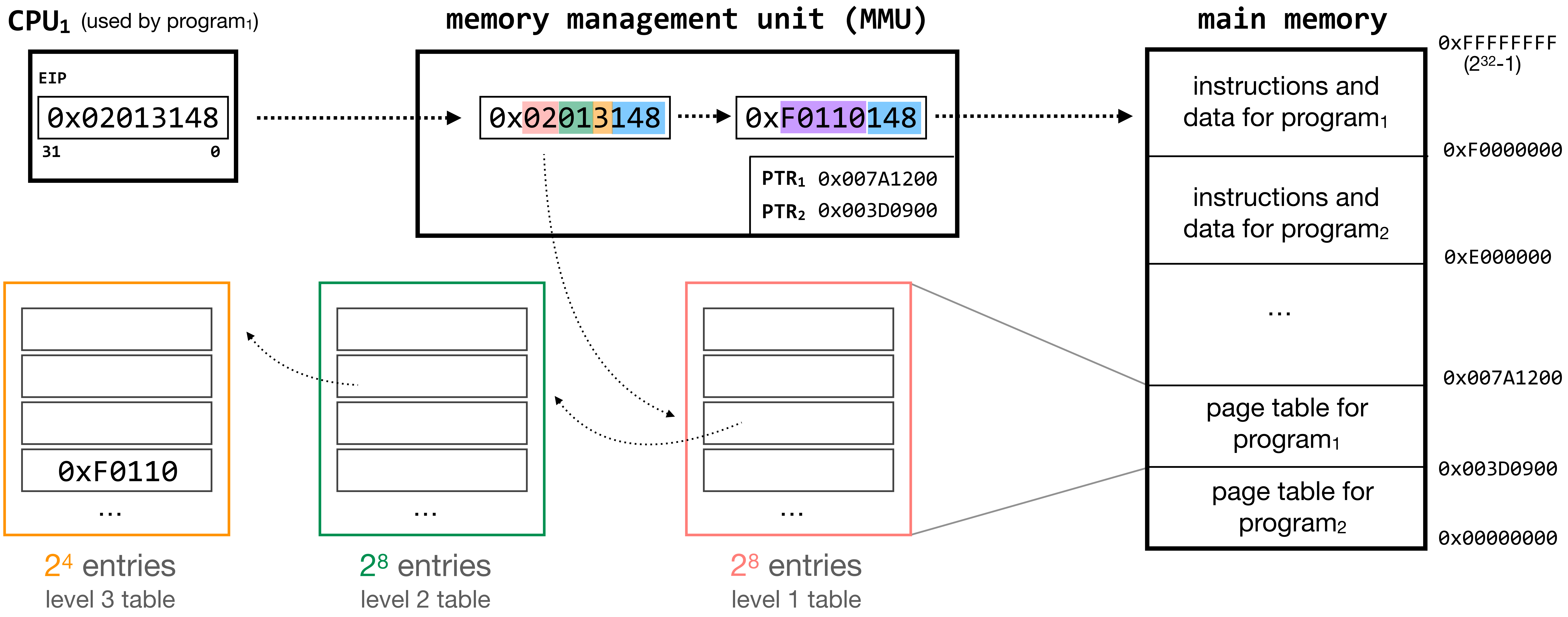
2^8 entries

level 1 table

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



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)

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**

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

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

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

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