Virtual Memory: Paging

Questions Answered in this Lecture:

- How do we do better than dynamic relocation?
- What is paging?
- Where are page tables stored, how are they created?
- How are page tables managed?

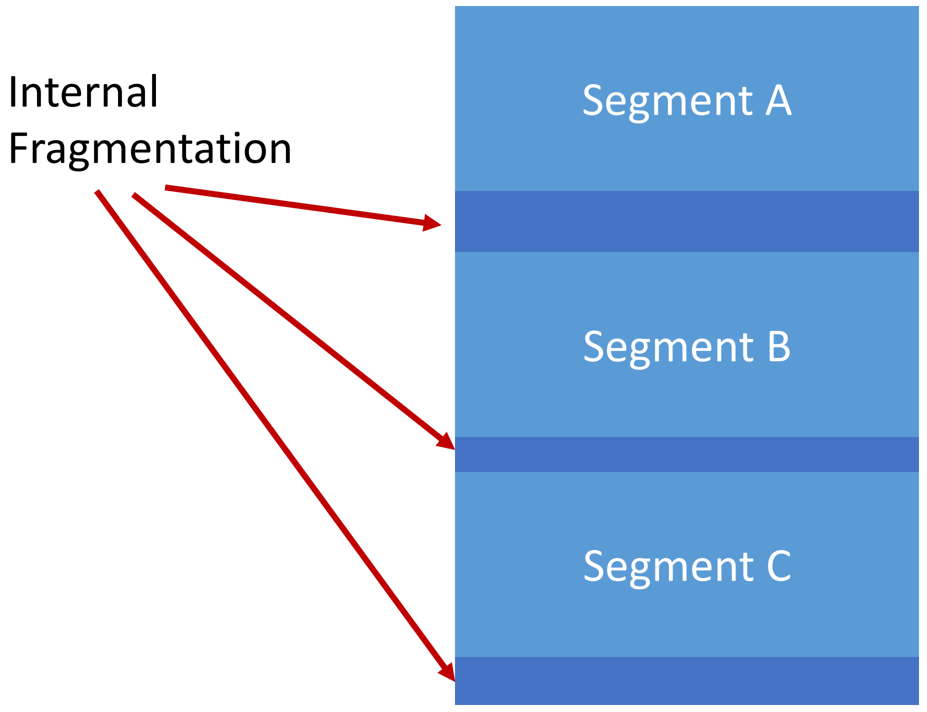
Announcements

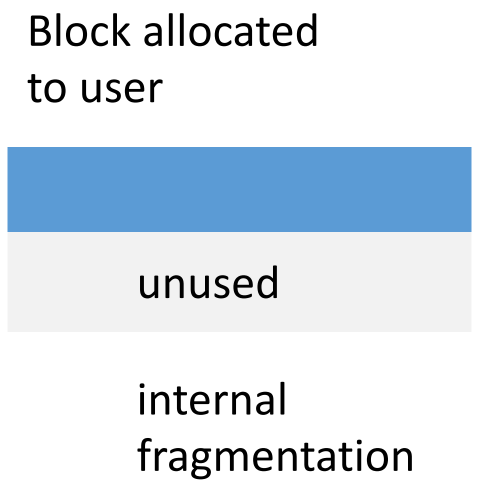
- P1a due tonight
- P1b out tonight
- Keep up with your reading!

Main Problem with Segmentation: Fragmentation

- Free memory which cannot be allocated for useful things
- Why does it happen?
  - Large allocations leave small pockets of free space
  - Allocator prohibits use of this space
- Types?
  - External: Visible to the allocator (i.e. the OS)
  - Internal: Visible to the requester (e.g. if allocations must be a power of 2 size)

Example


Internal Fragmentation
Segment A
Segment B
Segment C


Block allocated to user
unused
internal fragmentation

Aside on address translation

$$\underset{\substack{\uparrow\\ \text{physical address}}}{p} = \underset{\substack{\uparrow\\ \text{translation function}}}{f}(\underset{\substack{\uparrow\\ \text{virtual address}}}{v})$$

Aside on address translation

$$p = f(v)$$

identity mapping	$f(v) = v$
offset mapping	$f(v) = v + c$
arbitrary mapping	$f(v) = M[v]$

Address translation can be a function of *time*

$$p = f(v, t)$$

*this gives us **dynamic mappings**!*

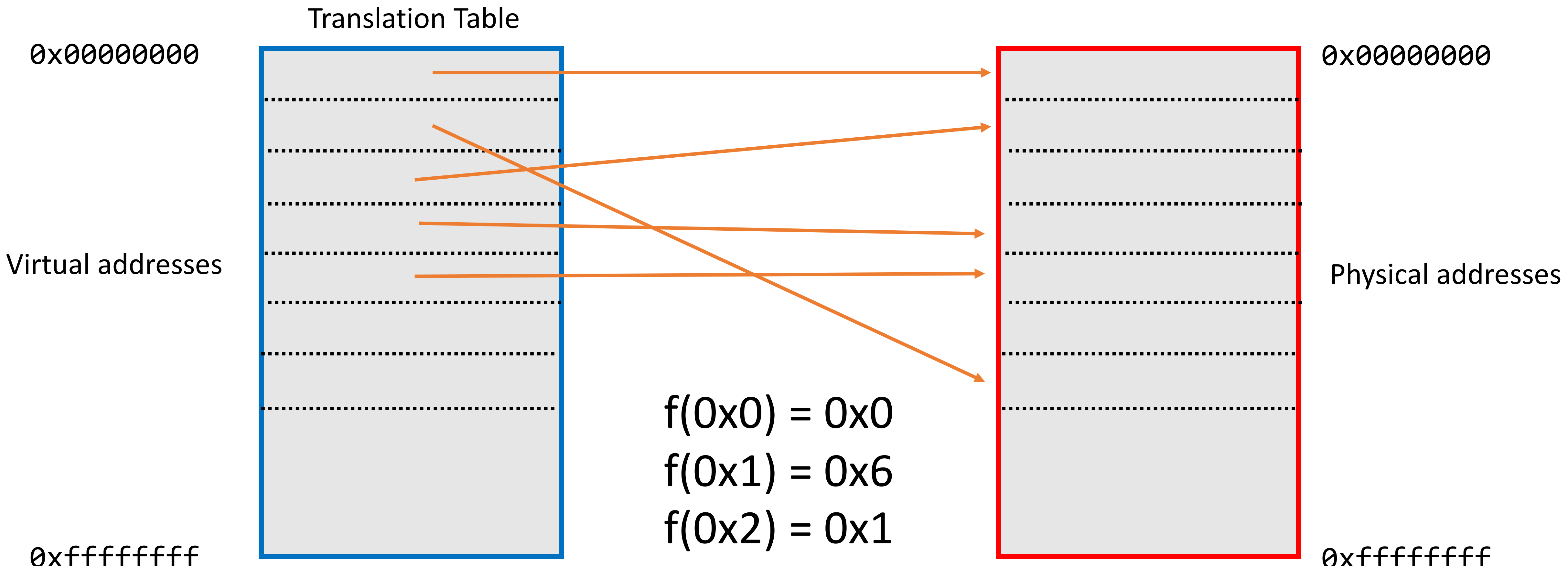
Where we're going

- We need a way to reduce fragmentation, and to allow *arbitrary* mappings from ***virtual addresses to physical addresses***
- We want to ***remove the contiguous address space restriction***
- What we'll end up with is more flexible than segmentation
- We'll use a *translation table,* with one entry per translation
- ***Each process has its own*** translation table

Translation (attempt 1)

- ***Every*** VA has a different translation
- Maintain a table *somewhere* to hold these translations

Example



f(0x0) = 0x0
f(0x1) = 0x6
f(0x2) = 0x1
f(0x3) = 0x3
f(0x4) = 0x4

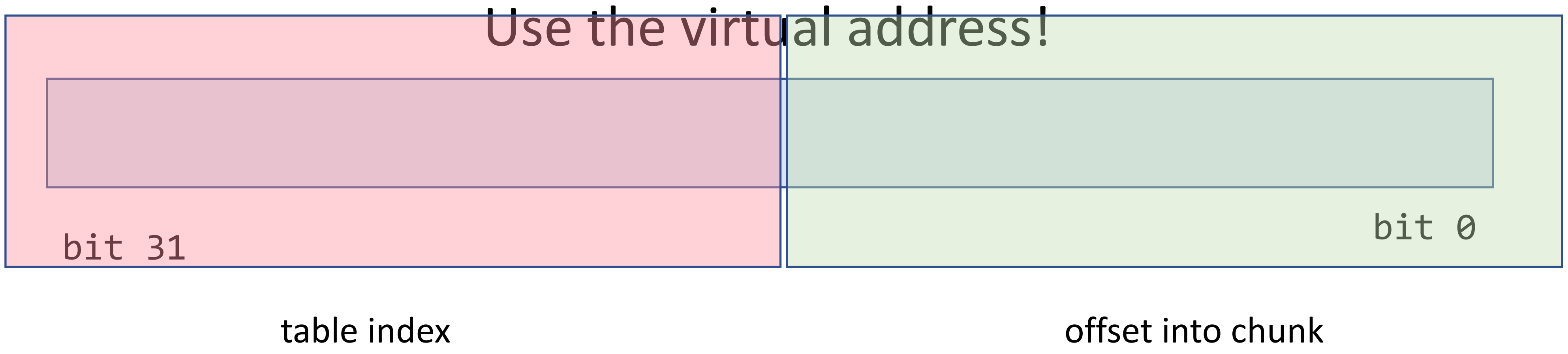
What’s wrong with this?

- Way too much overhead! (4 bytes for every address on a 32-bit machine)
- If we had a 4GB machine we’d need another 16GB (4bytes/address * 4G addresses) ***just for the translation table*** ***FOR EACH PROCESS!***

Translation (attempt 2)

- Let’s translate addresses in bigger chunks
- The bigger the chunk, the less space we need for our table (one entry for every chunk)
- But the bigger we make the chunk, the greater the chance of external fragmentation!

Indexing into the translation table



16 bits for table index
16 bits for offset

How big is a chunk?

How many table entries?

How much space used by table?

Indexing into the translation table

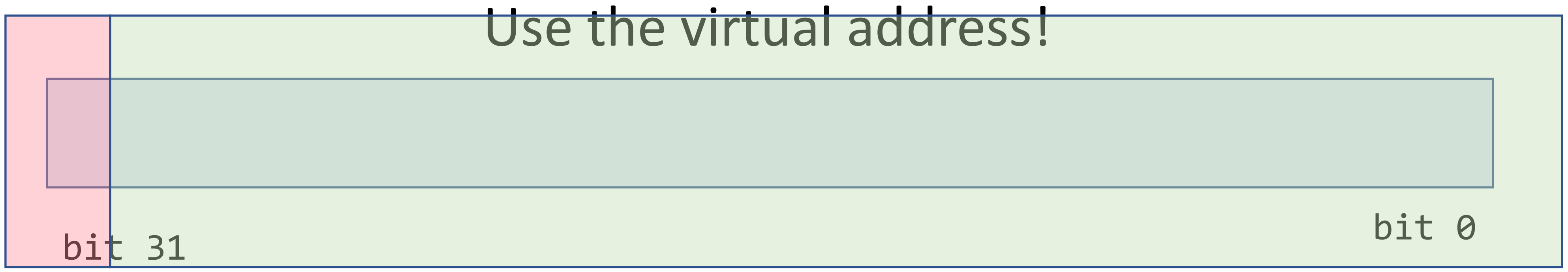


table index

offset into chunk

1 bit for table index
31 bits for offset

How big is a chunk?

How many table entries?

How much space used by table?

Indexing into the translation table

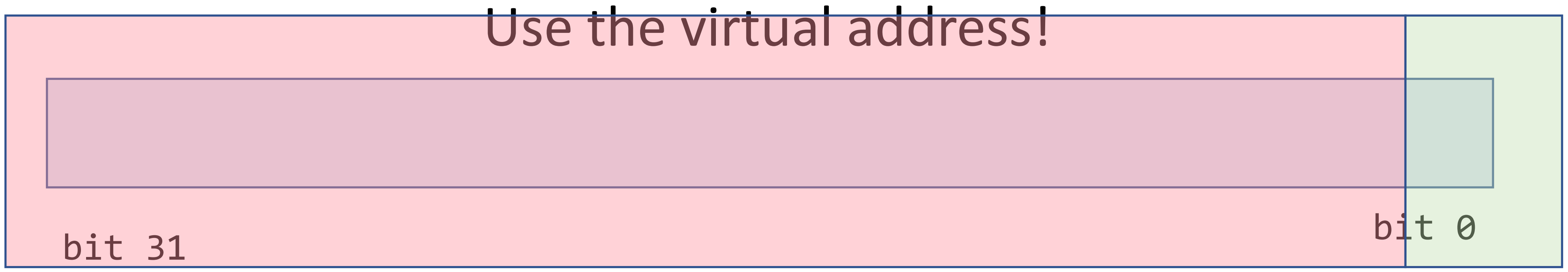


table index

offset into chunk

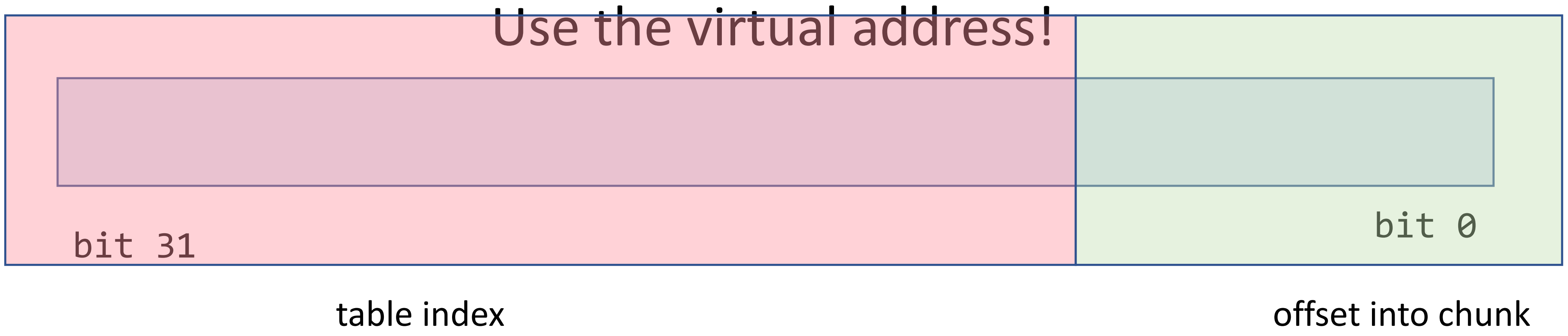
31 bit for table index
1 bits for offset

How big is a chunk?

How many table entries?

How much space used by table?

Indexing into the translation table



20 bits for table index
12 bits for offset

How big is a chunk?

How many table entries?

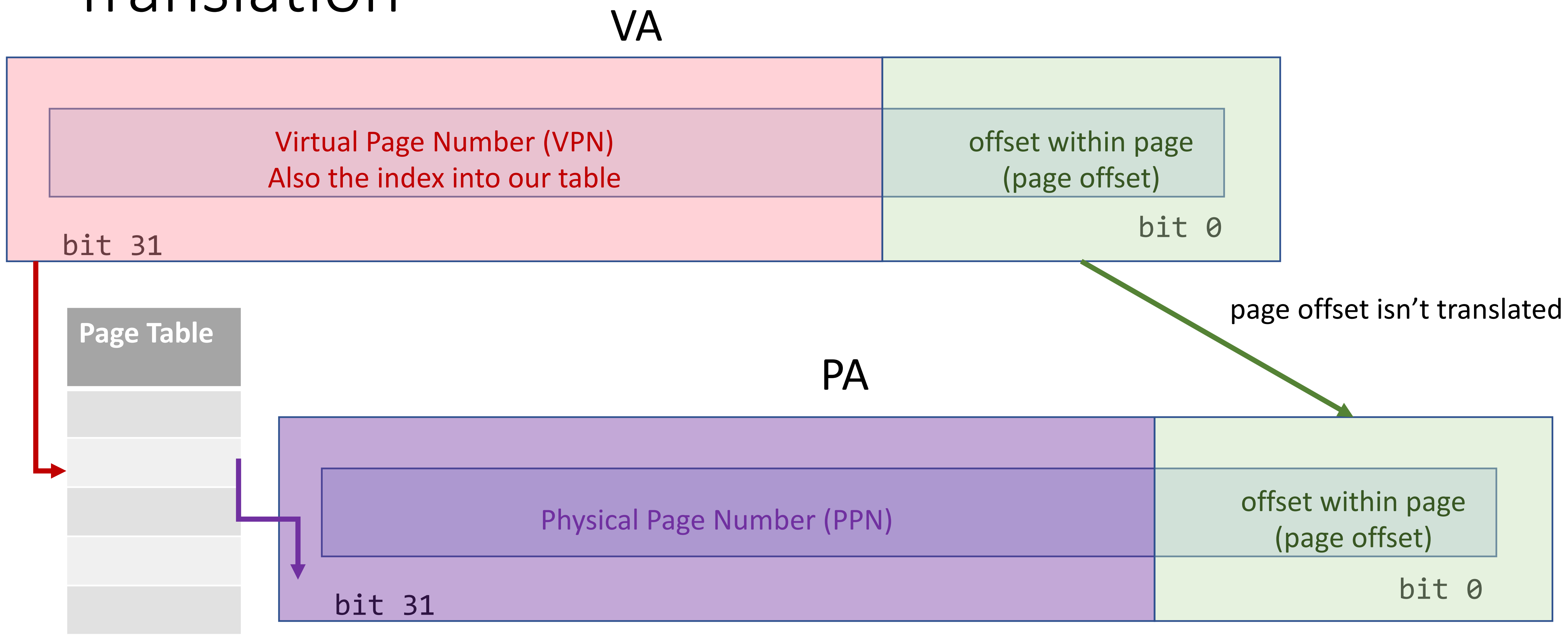
How much space used by table?

Translation using *paging*

- 4K is a standard chunk size
- We call each chunk a *page*
- Good tradeoff between table overhead and fragmentation
- How do we translate from VA to PA? (remember, ***virtual address*** is the analogy to our ***logical address*** from before)

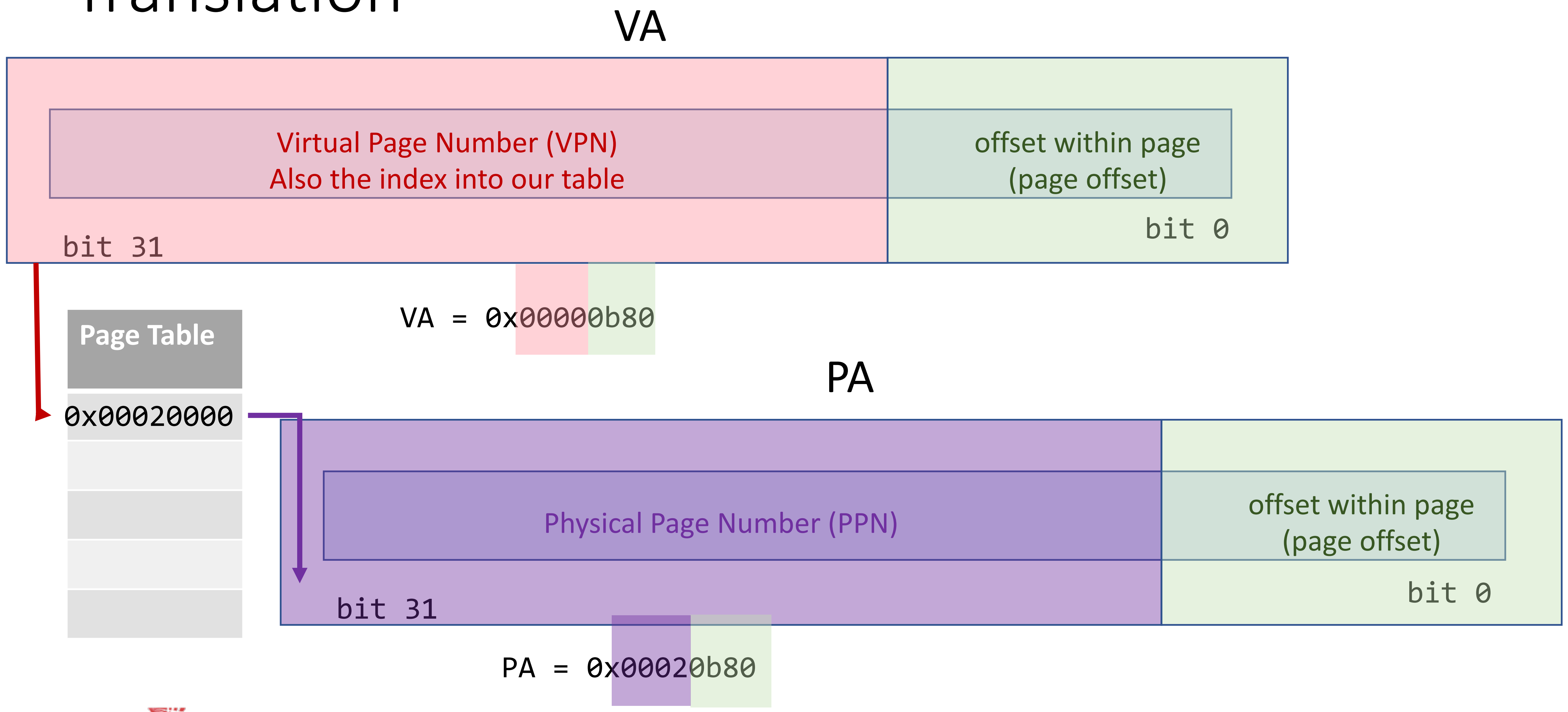
16 bits for table index
16 bits for offset

Translation

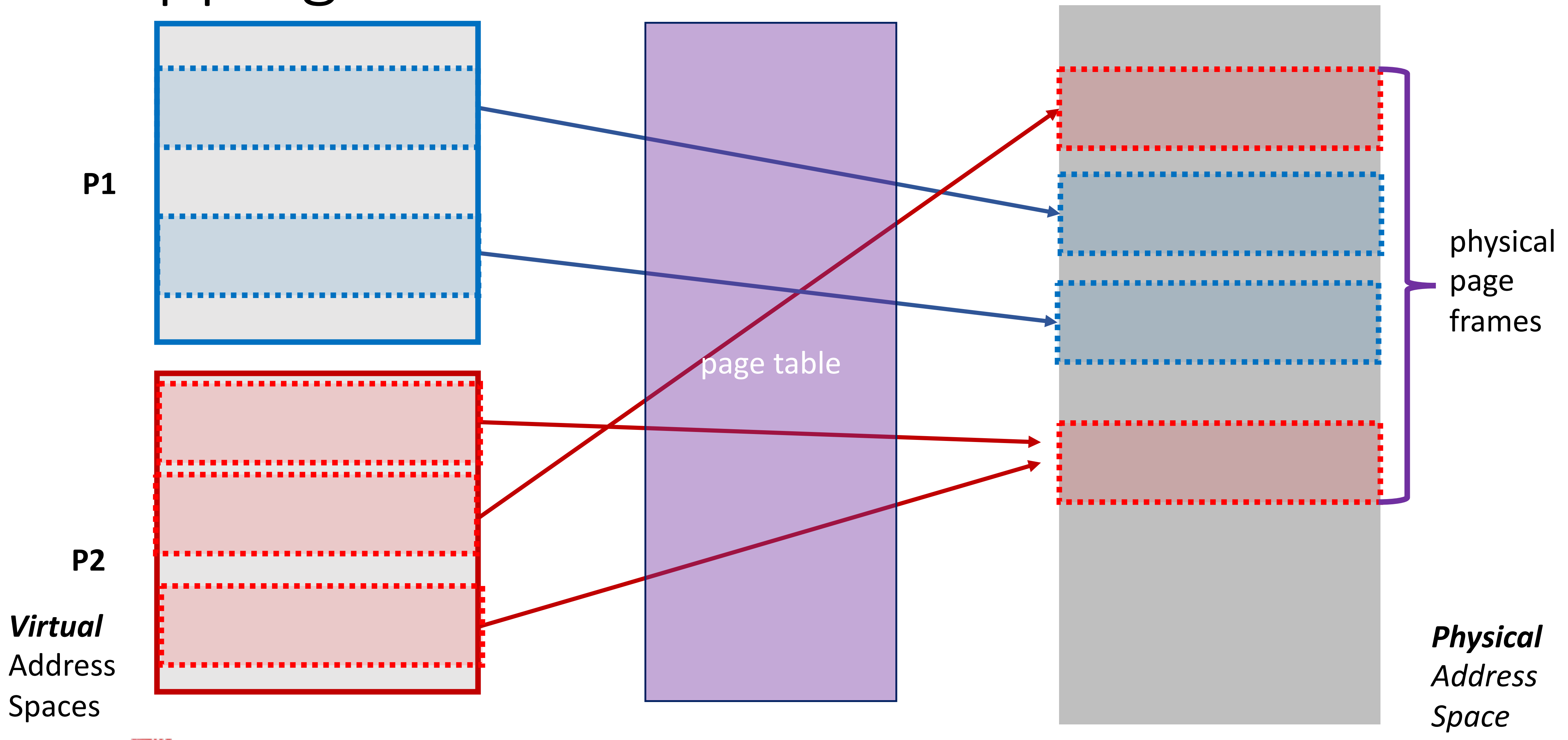

VA
Virtual Page Number (VPN)
Also the index into our table
offset within page
(page offset)
bit 31
bit 0
page offset isn't translated
Page Table
PA
Physical Page Number (PPN)
offset within page
(page offset)
bit 31
bit 0

Translation

16 bits for table index
16 bits for offset

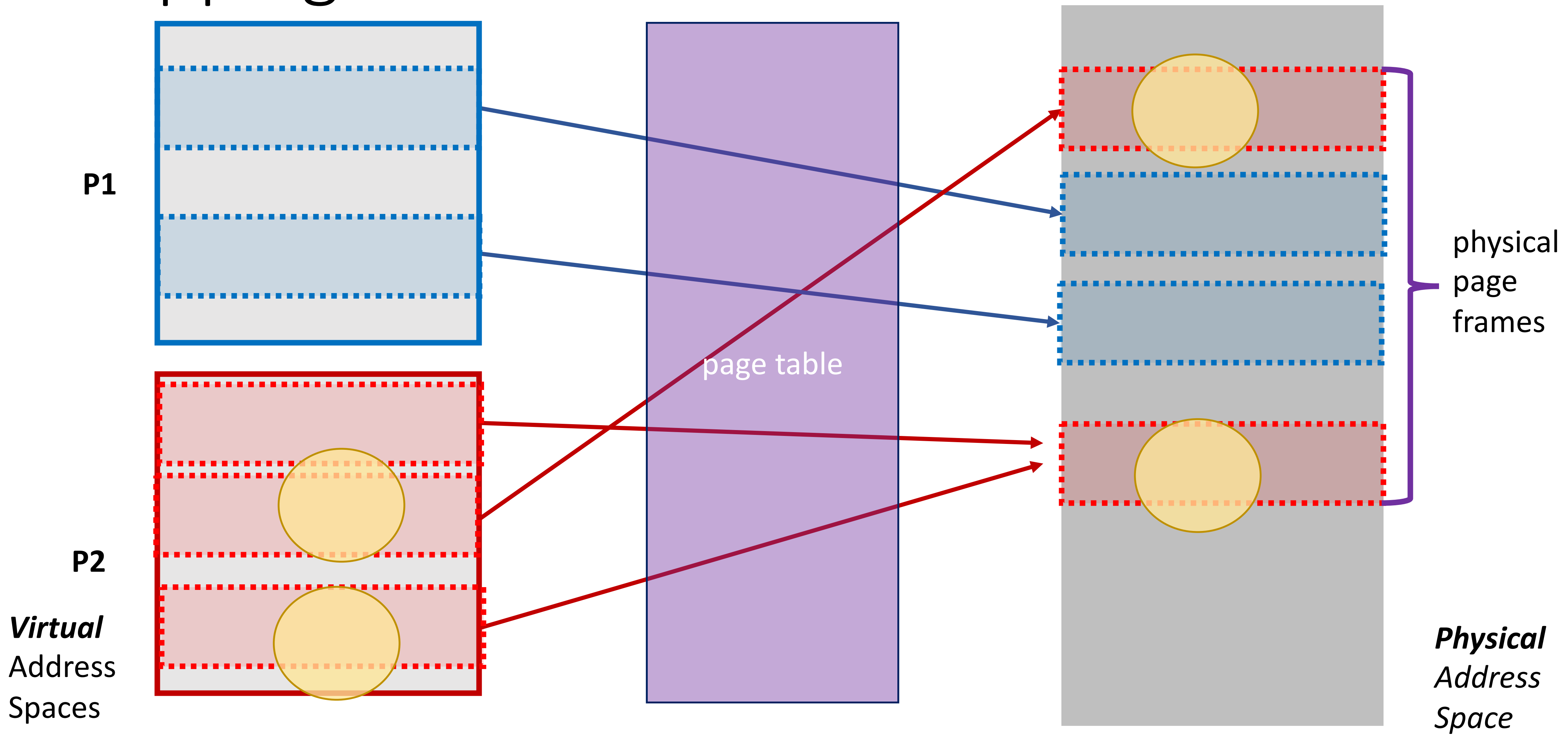


Mapping


P1
P2
Virtual
Address
Spaces
page table
physical
page
frames
Physical
Address
Space

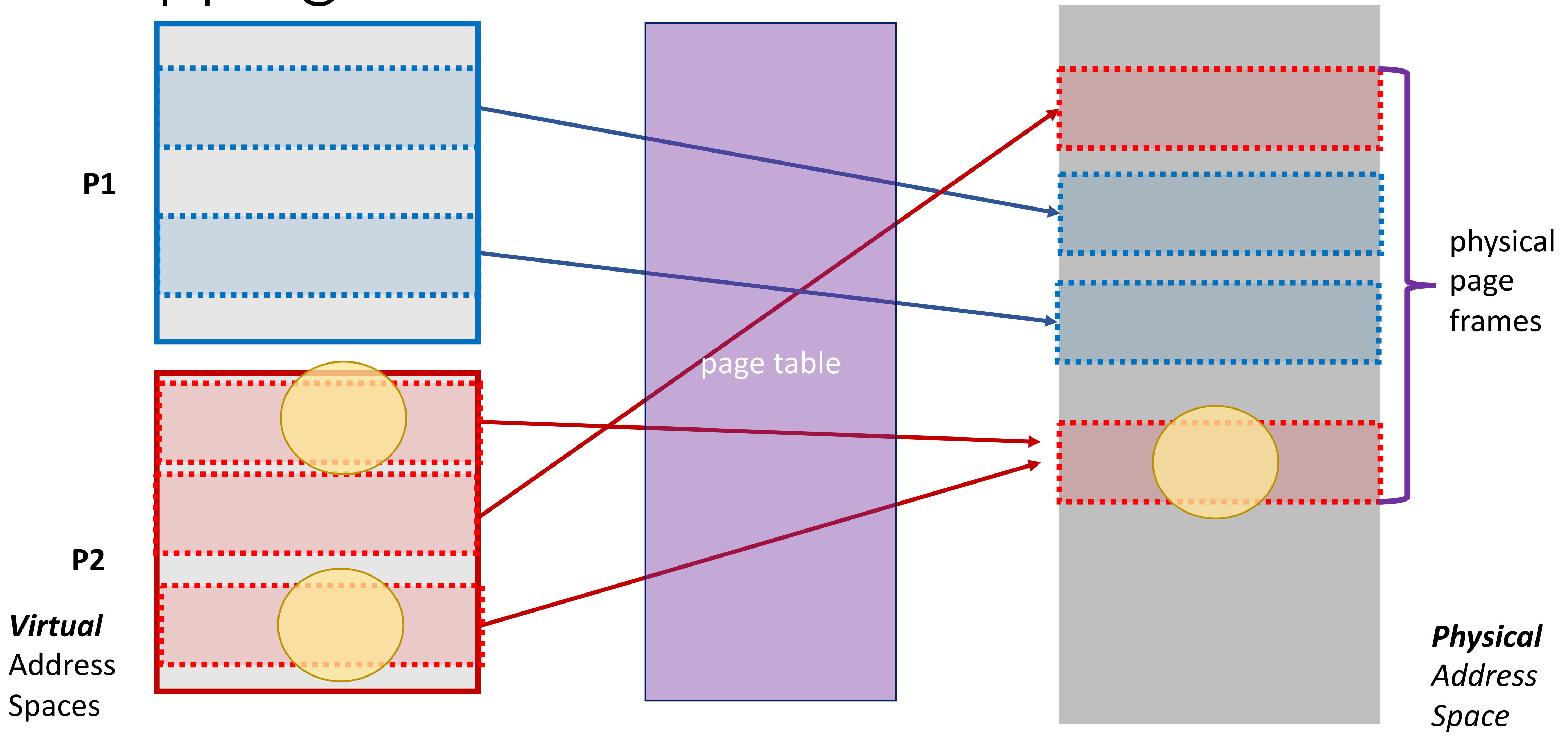
Address spaces are *not* physically contiguous!

Mapping



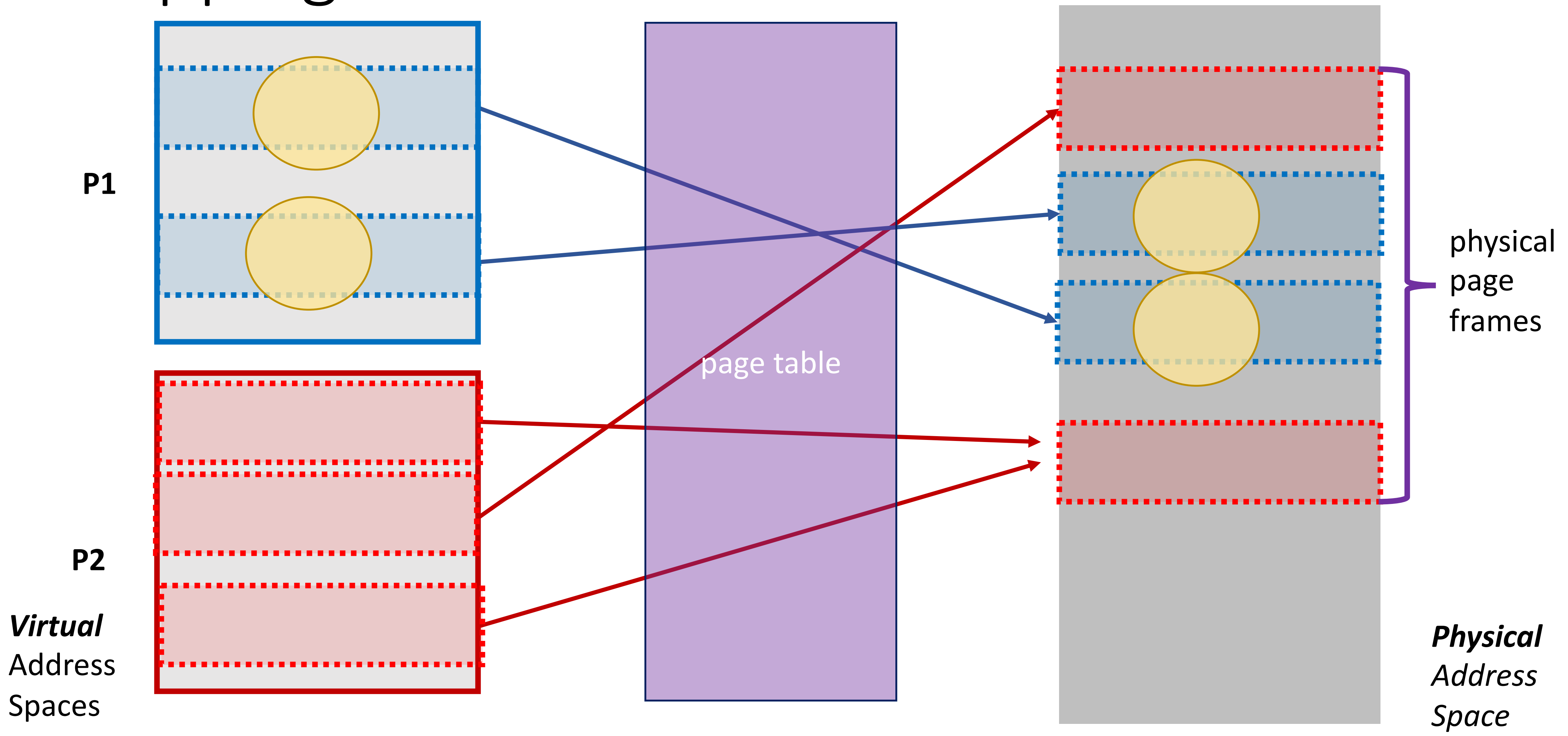
Mapping

Translations can alias!
(mapping function is not always one-to-one)



Mapping

Virtual Contiguity does not imply Physical Contiguity!



Where do we put the pagetables?

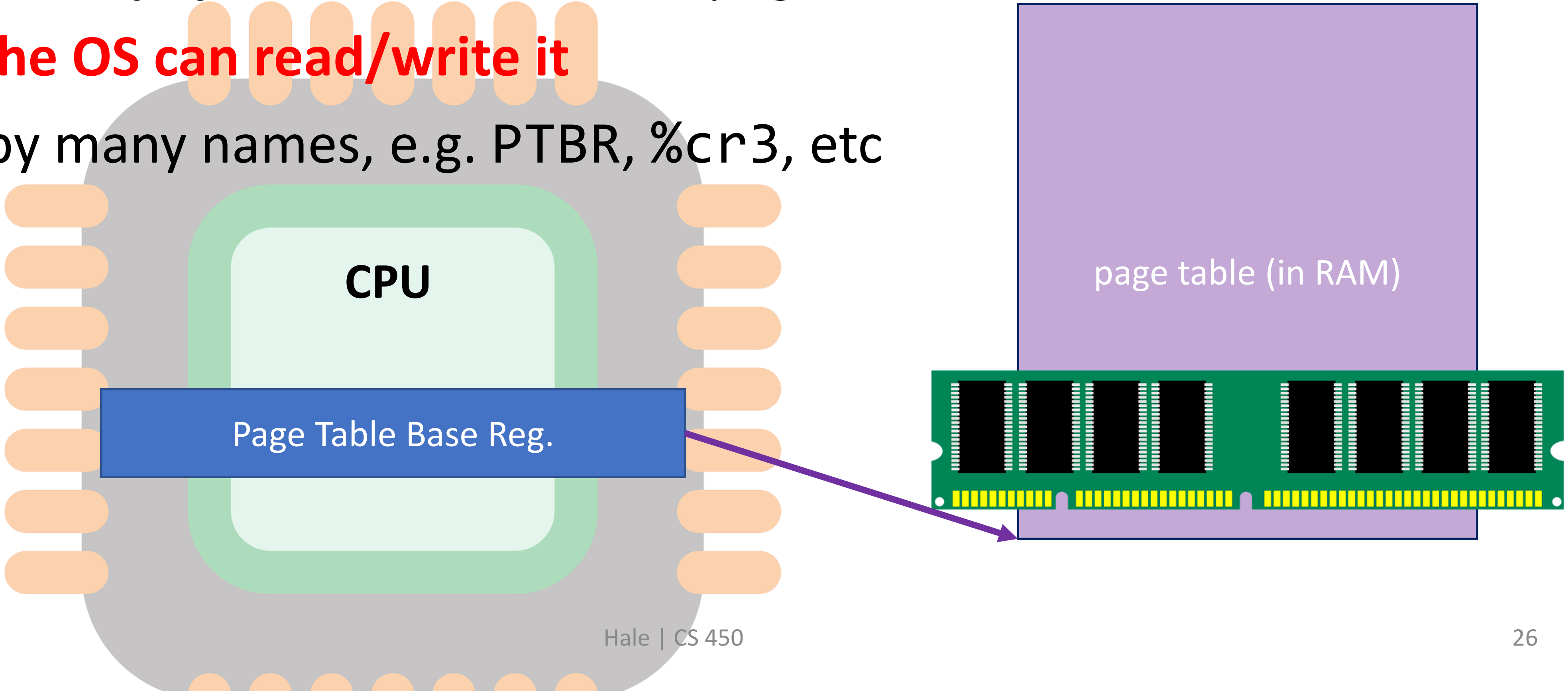
- We could create special logic on our chip…
- But too expensive!
- **Put them in memory!** (headscratcher right?)
- OS ***installs*** (and ***manages***) page tables
  - Install: create the mappings by writing the table somewhere in memory
  - Manage: update, delete, handle errors (more on this later)
- Hardware does the translation (**by doing lookups in the page tables**)
  - This lookup is called a ***page walk*** (we'll see why in a later lecture)
  - Raises errors (for OS to handle) when it can't grok the page tables

How does the hardware perform a page walk?

- The page tables are in memory, but *where*?
- OS needs a way to tell the hardware where the PT is

Page Table Base Register

- A register which points to the current page table
- It holds the ***physical address*** of the page table
- **Only the OS can read/write it**
- Goes by many names, e.g. PTBR, `%cr3`, etc



What does this mean for memory access?

- ***Every memory reference*** must be translated
- Therefore ***every memory reference*** goes through the PT

```
mov 0x80000, 8(%ebx)
```

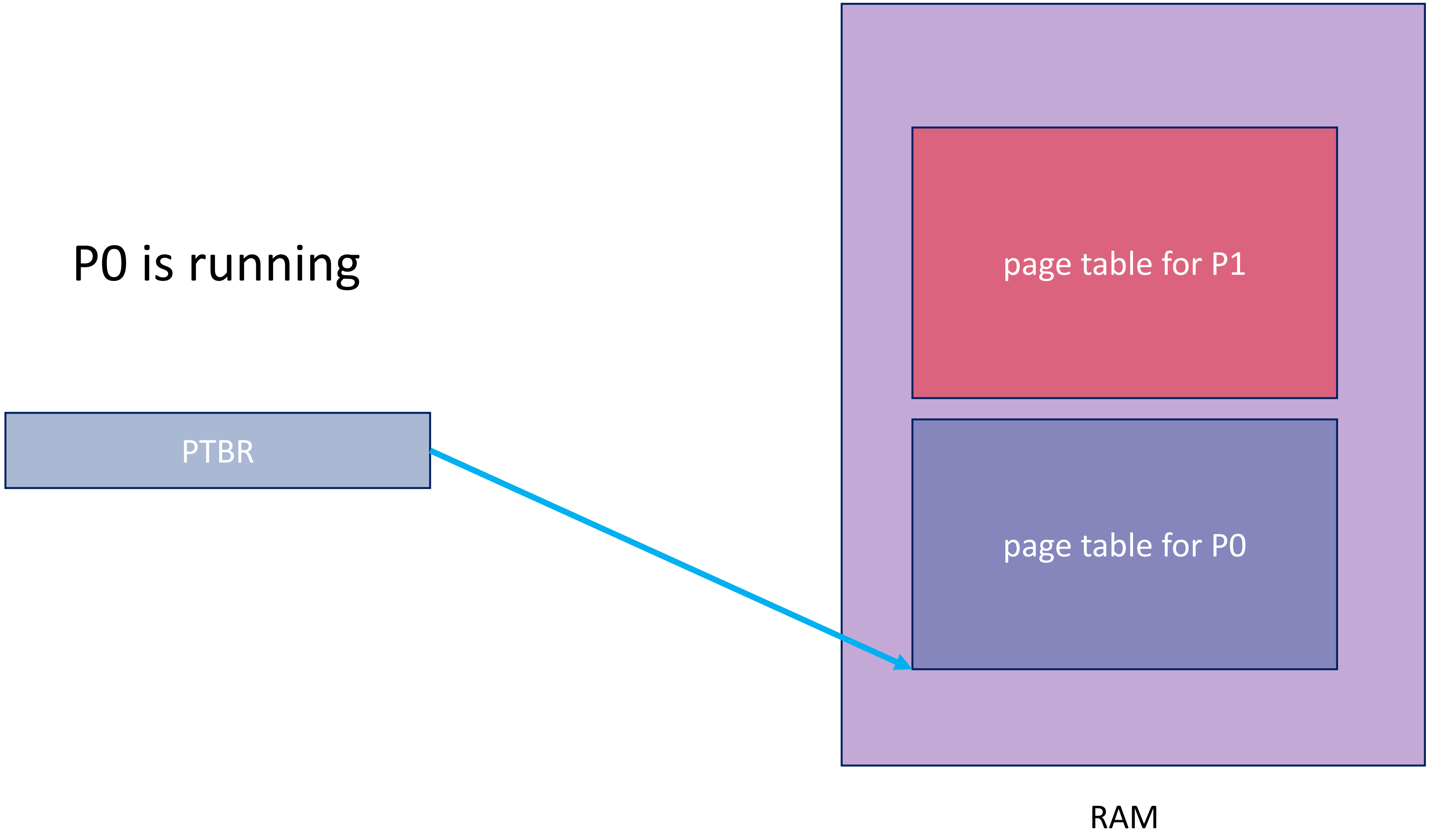
How many memory references?

HINT: this is a trick question…

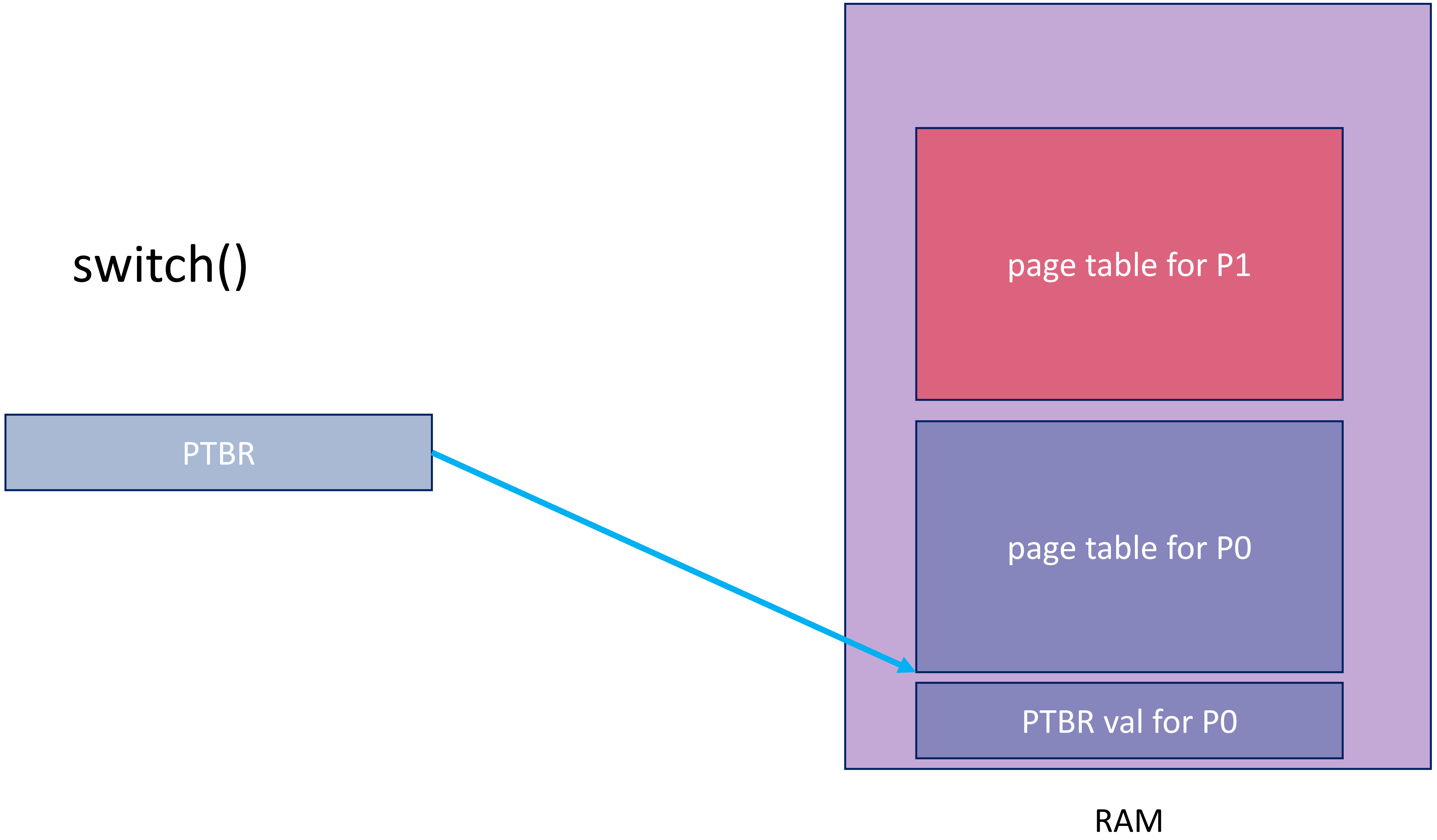
What happens on a context switch?

- Remember, ***each process*** has its own page table

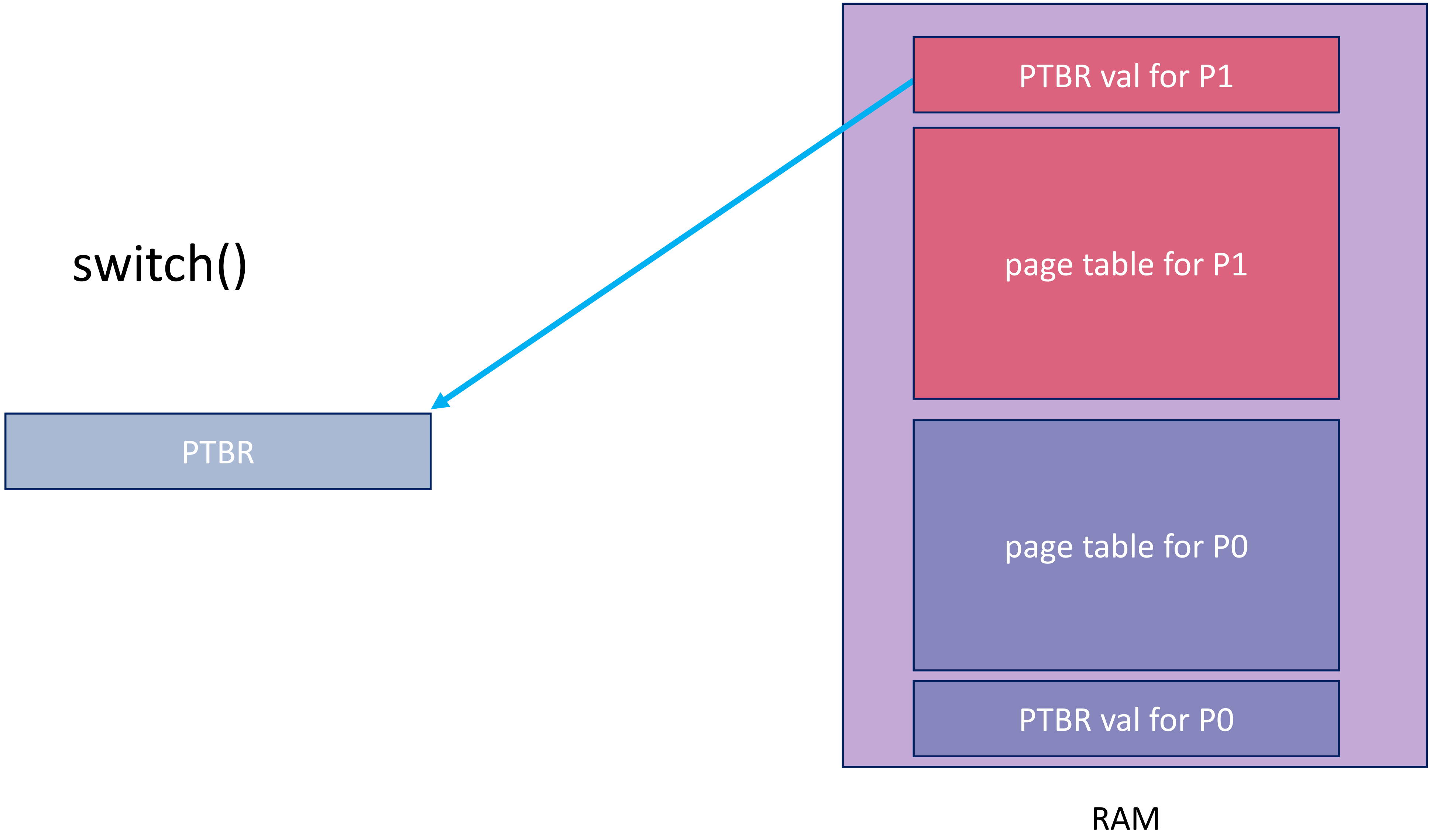
P0 is running



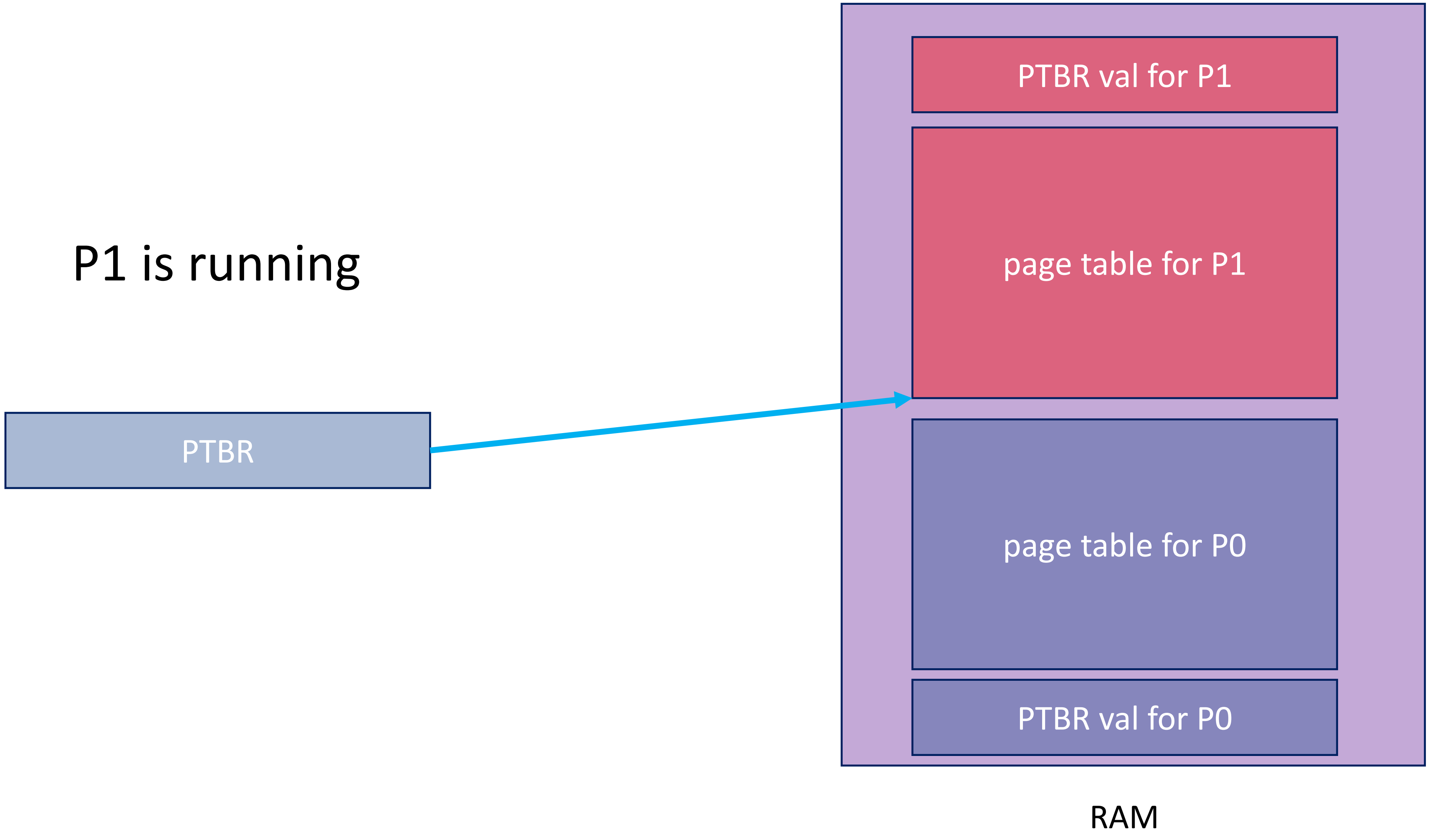
switch()



switch()

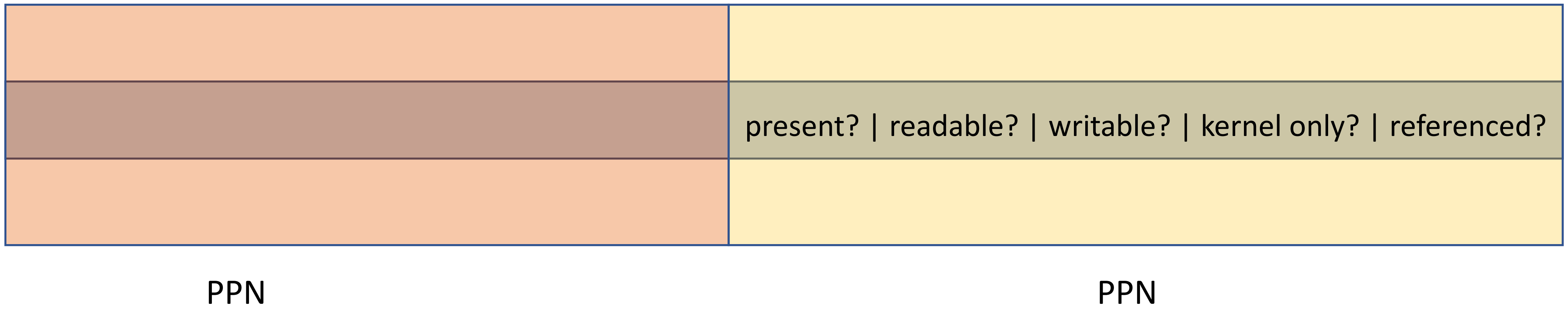

PTBR
PTBR val for P1
page table for P1
page table for P0
PTBR val for P0
RAM

P1 is running



Reusing some waste

- The page table entries only need to store the physical page number (PPN, sometimes called physical frame number, PFN)
- The PTE does not have to store the page offset (it can come directly from the VA)
- We can ***reuse the page offset bits*** for interesting stuff…


present? | readable? | writable? | kernel only? | referenced?
PPN
PPN

Paging: Advantages

- We **got rid of external fragmentation**
  - Any page can be placed in any frame in physical memory
- Fast to allocate and free
  - ***Allocating a fixed-size page is very fast*** (e.g. bitmap-based allocator, plenty of nice hardware instructions for this)
  - Freeing a page is simple (no need to merge blocks)
- **Simple to swap out** portions of memory to the disk (more later)

Paging: Disadvantages

- ***Internal fragmentation***: page size might be too big for process's needs
  - If we try to reduce page table overhead with large pages, this gets worse

- Additional memory references ***for every load and store***
  - Because page tables are in memory!
  - Solution: caching (next time)
- ***Storage overhead*** for page tables is still pretty high
- We're allocating a PTE for every page (even if it isn't used)
- Solutions next time