Virtual Memory: Multi-level Paging and the TLB

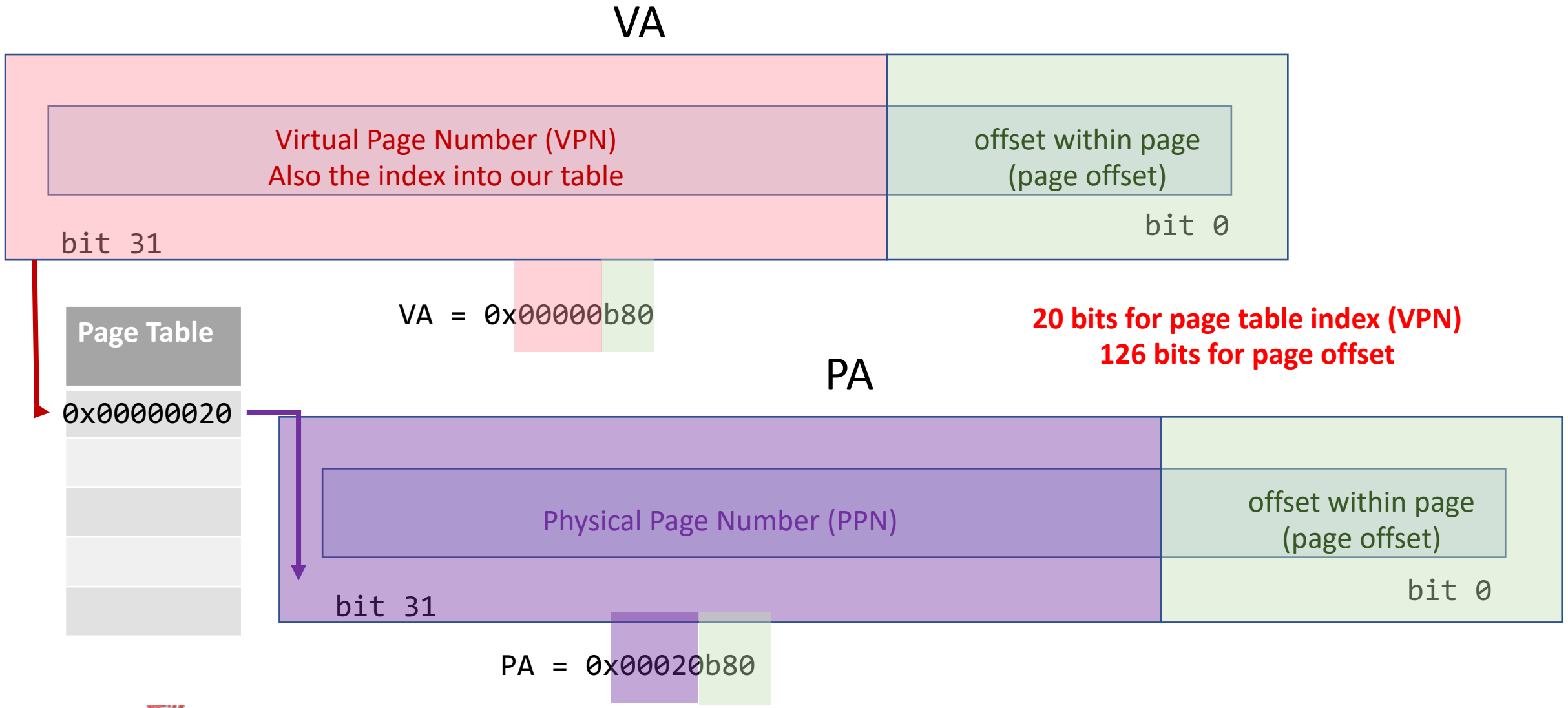
Questions Answered in this Lecture:

- How do we decrease page table overhead without tweaking the page size?
- How do we avoid unnecessary page tables?
- How do we avoid the extra memory references caused by page table lookups?

Announcements

- P2A is out! Due next Thursday. Plenty of background reading involved so get started early!
- This time you won't be able to hand-in without your info.txt file. If you manage to do it, you'll get a zero
- Hale will be back on Mon.
- Keep up with your reading!

Reminder: This was our lookup scheme



Check:

- How big is the page table?
- 2^20 PTEs * (2^2 bytes per PTE) = 2^22 B = 2MB
- PER PROCESS!
- I have 413 procs running on my laptop now = ***826MB of page table crap!***
- Unacceptable

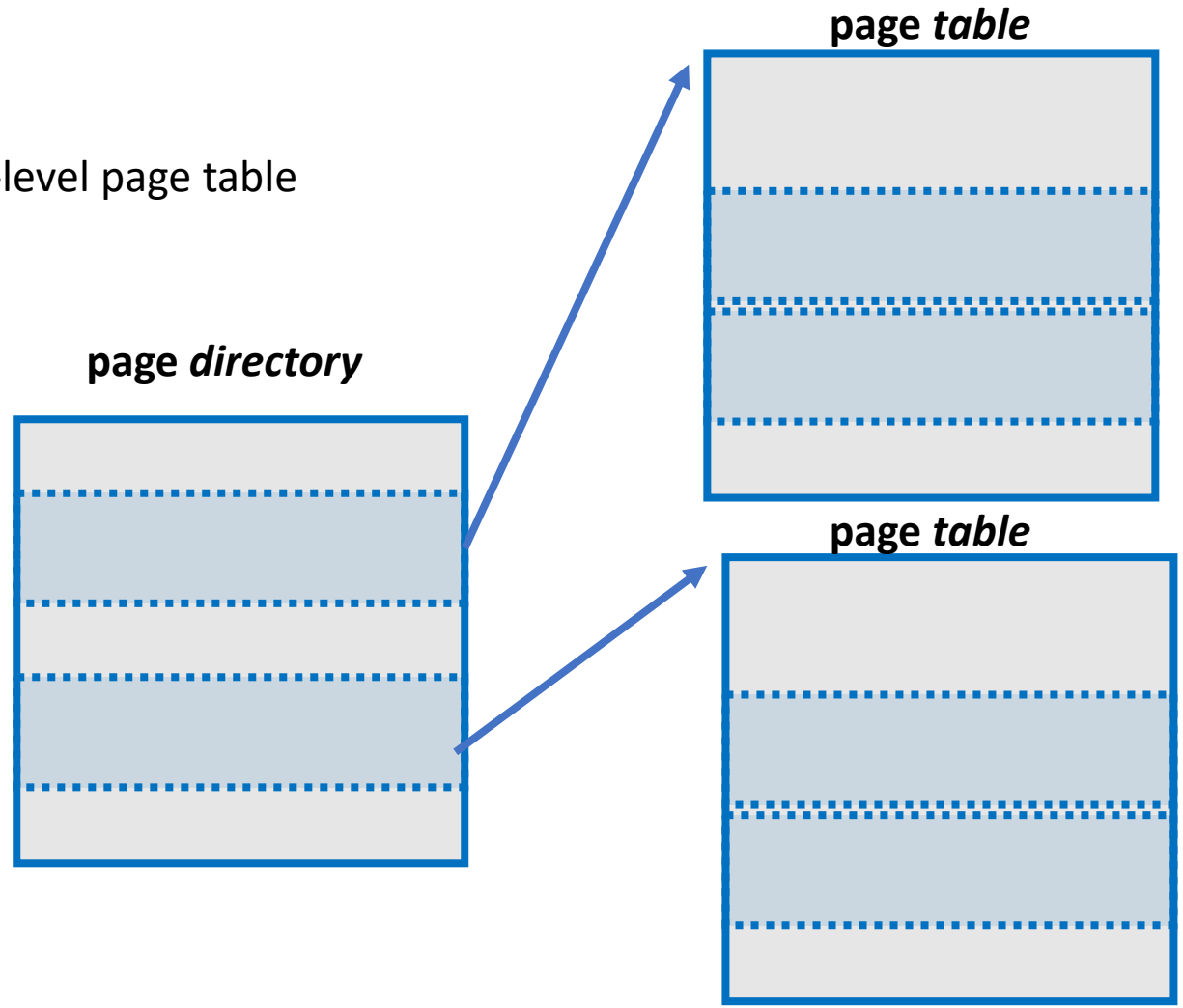
Observations

- We have one big page table. ***Will all the PTEs have valid mappings?***
  - **ALMOST NEVER**. Especially with a 64-bit address space, since you never (yet) have 2^64 bytes of RAM installed in your system
  - Wasted space!
- Typically **only *portions* of the address space map to anything** (heap, stack, code, data, mapped files and libraries, a few kernel regions here and there)
- We should **only pay the overhead for page tables when *valid mappings* are in place**!

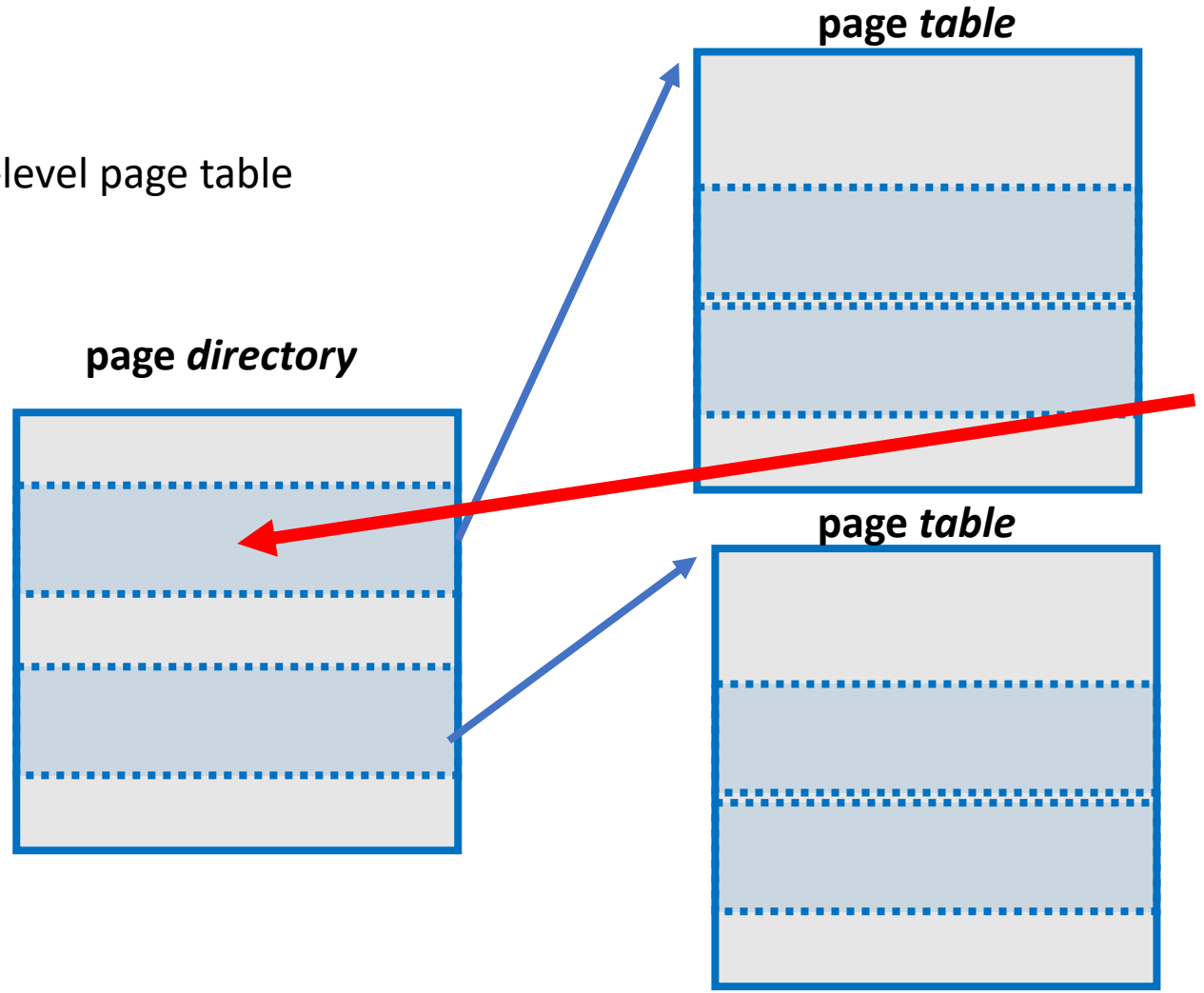
How? Guesses?

We'll *deconstruct* our page tables!

top-level page table



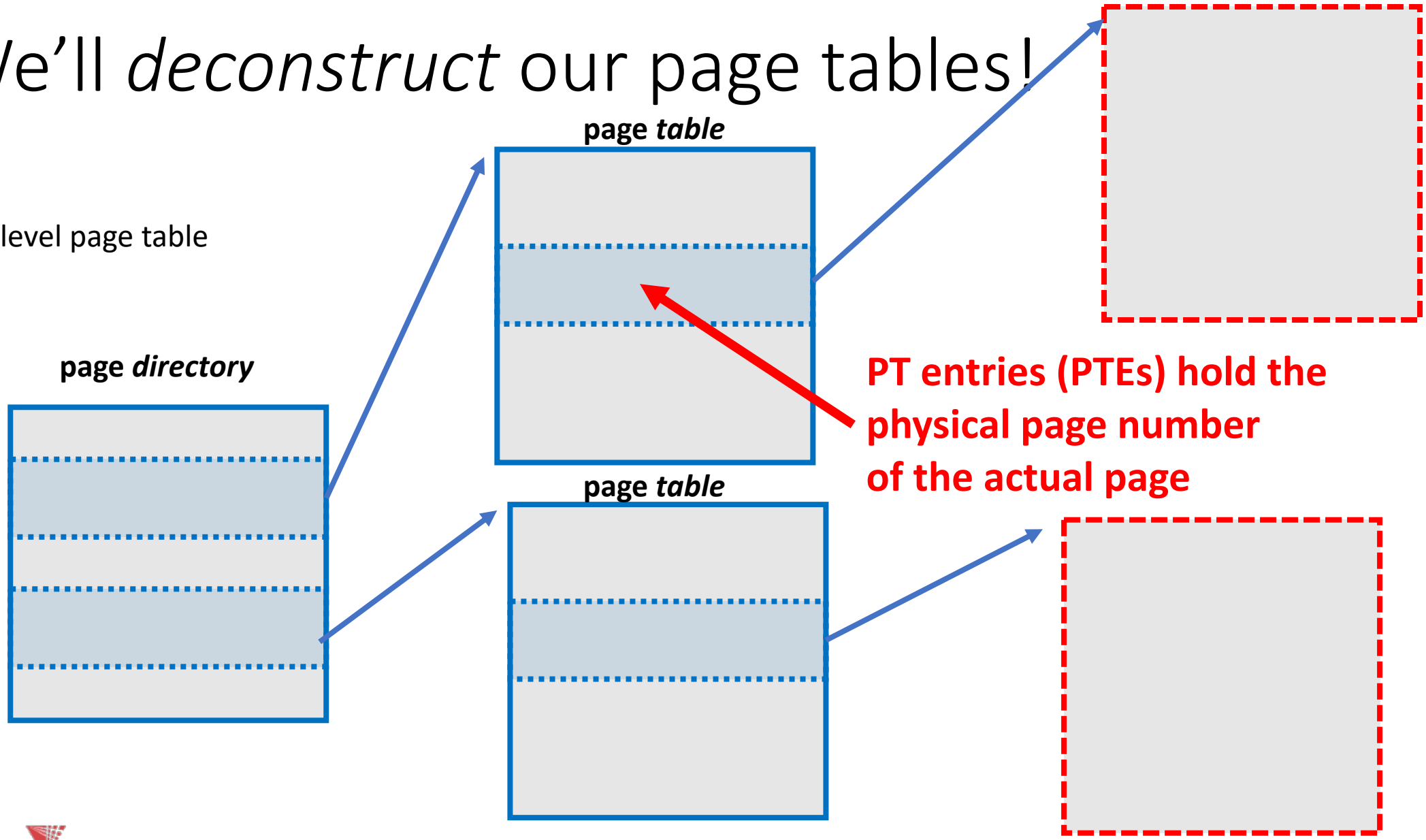
We'll *deconstruct* our page tables!



PD's entries (PDEs) hold *physical addre*
of the second-level table (PT)

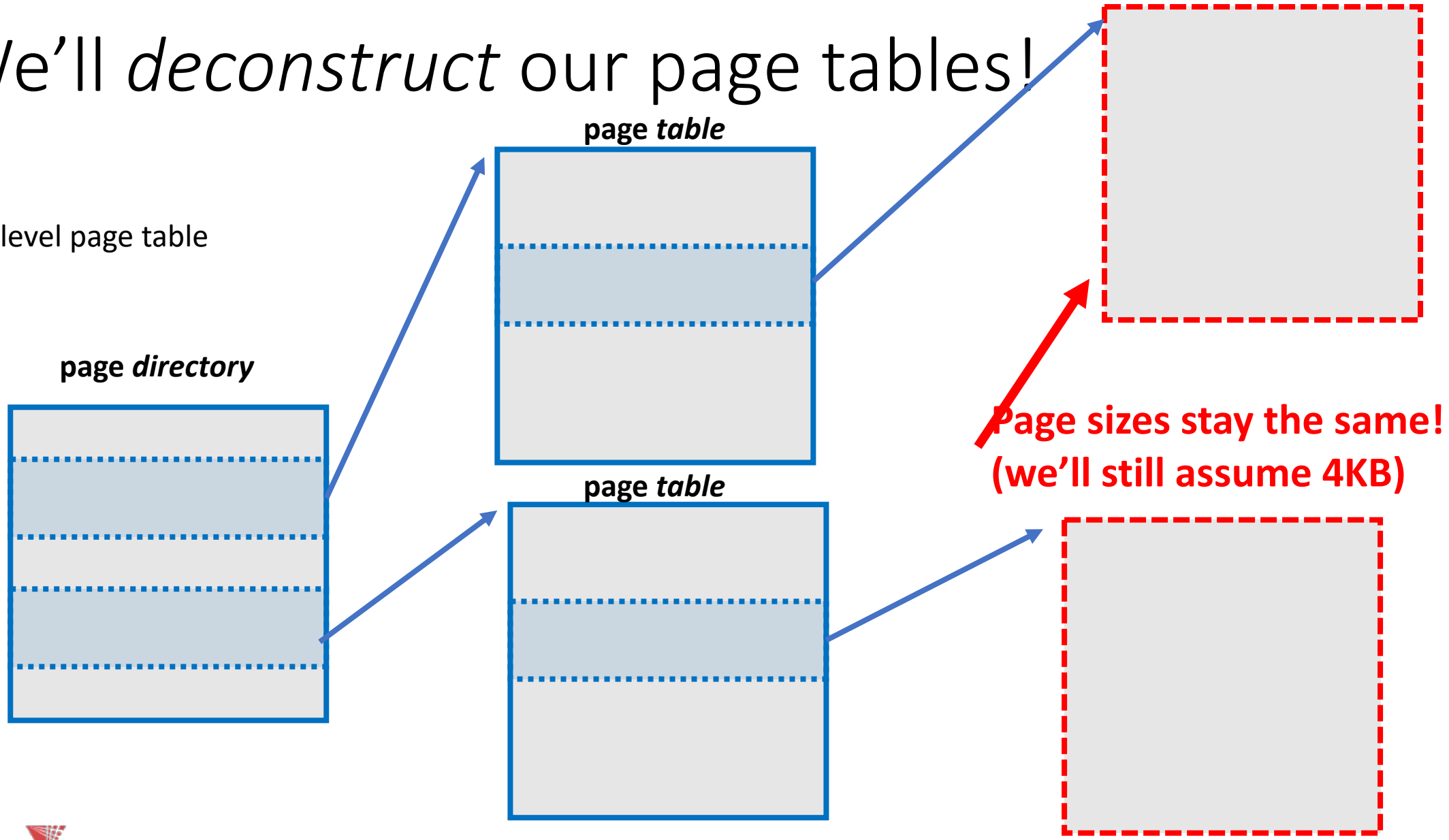
We'll *deconstruct* our page tables!

top-level page table



We'll *deconstruct* our page tables!

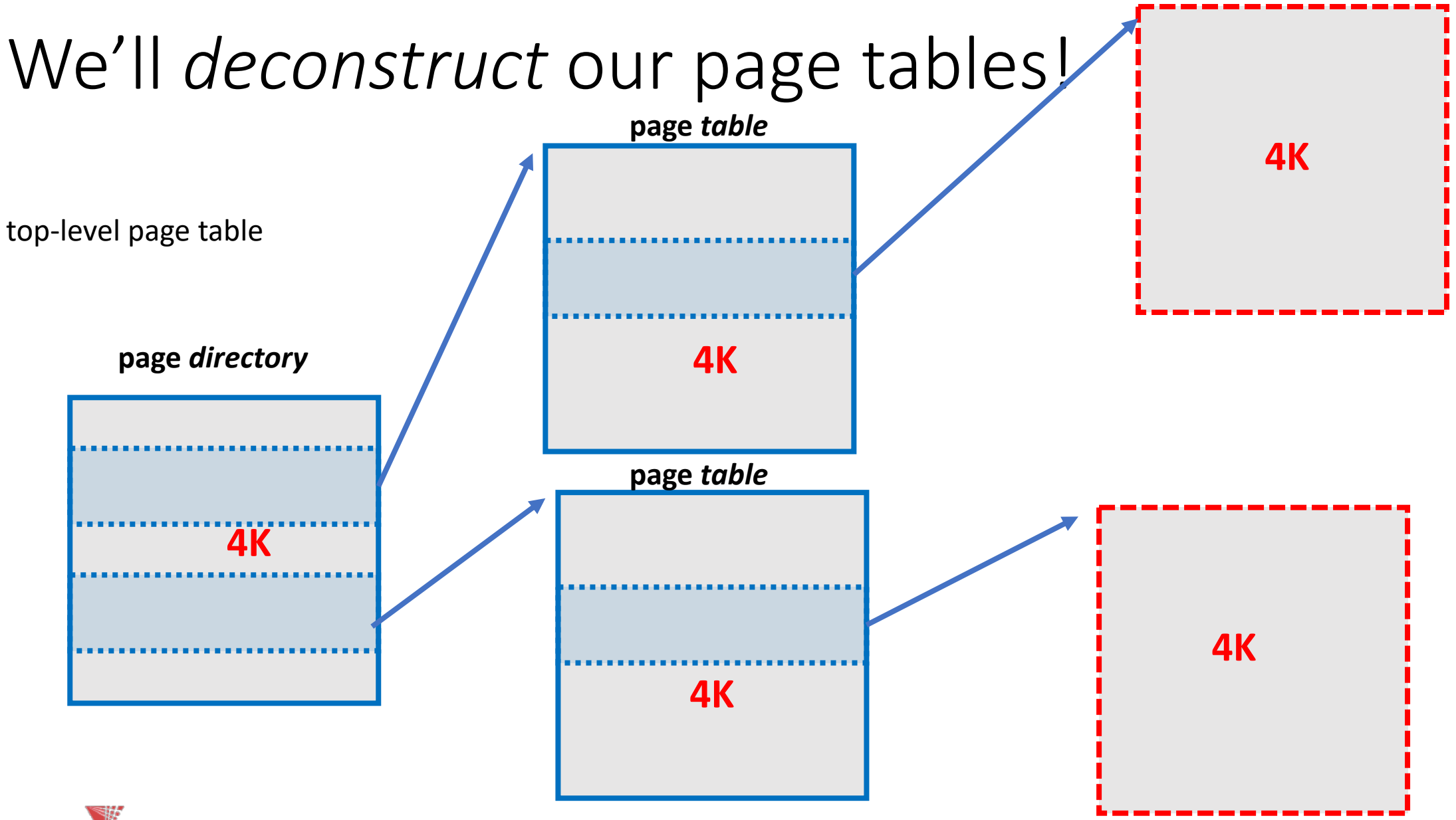
top-level page table



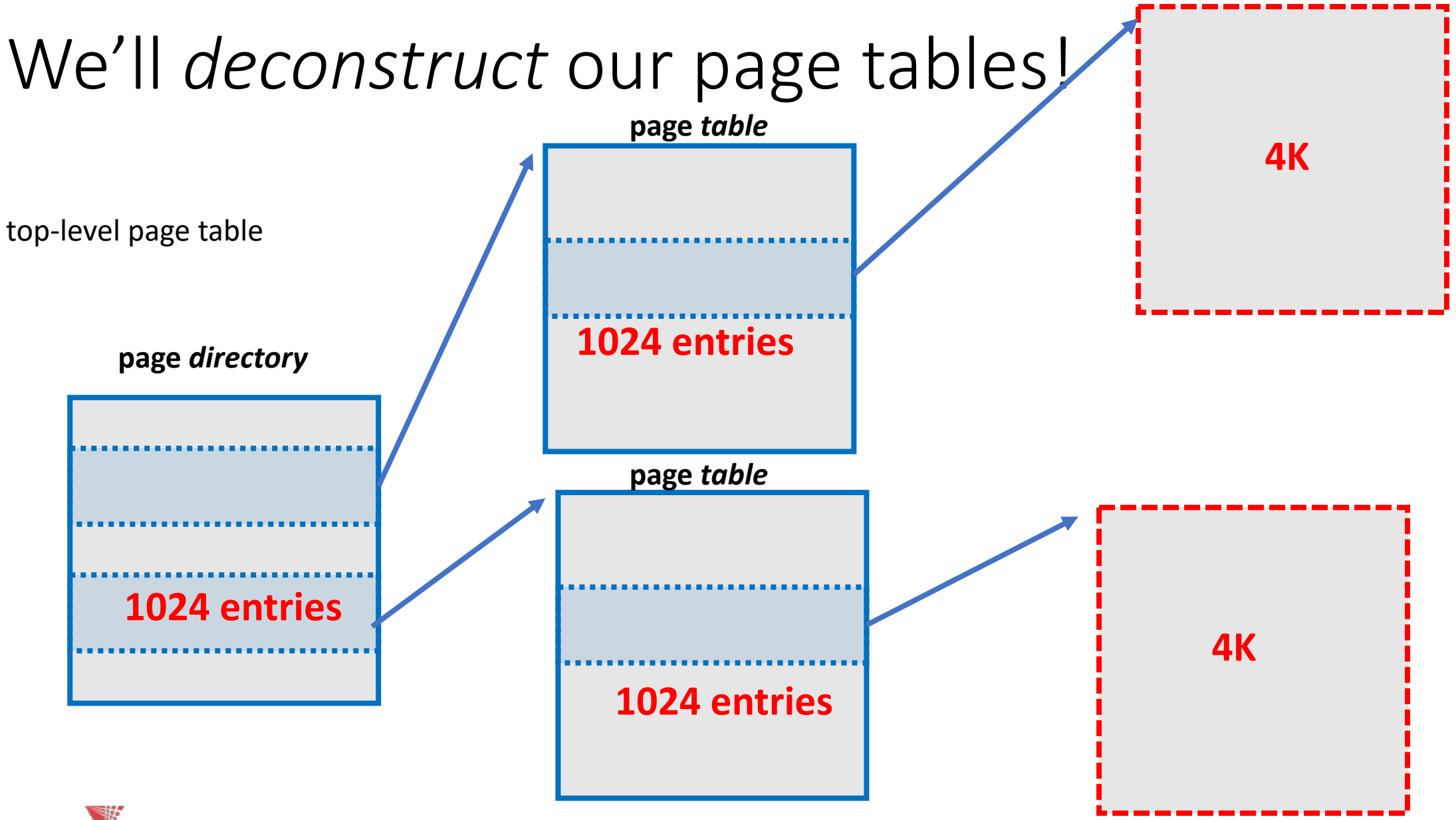
How big should the page *tables* be?

- Convenient decision: **Make them the size of the page!**
  - Otherwise it complicates our MMU hardware significantly
- It should become clearer as we go why this should be the case

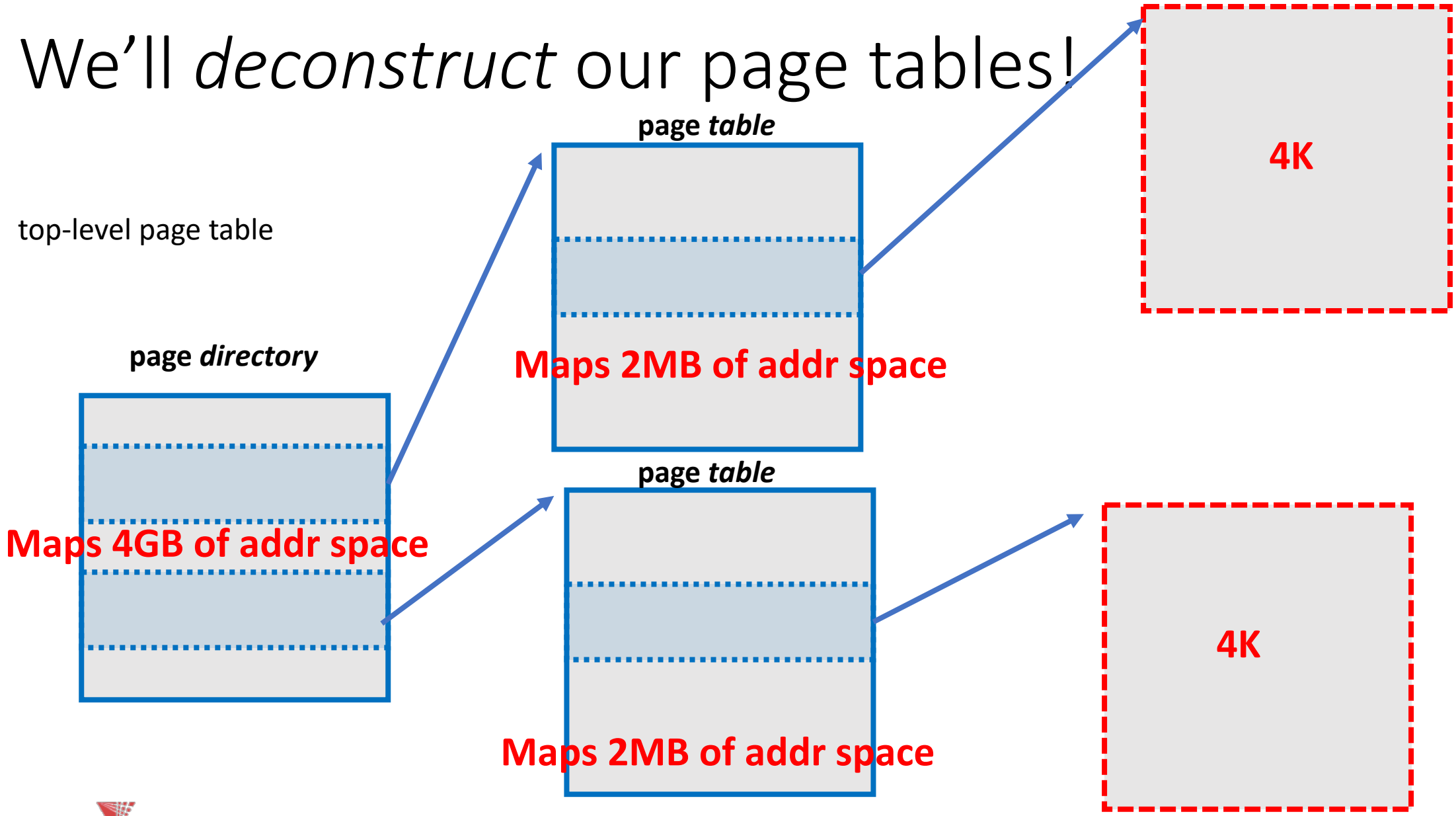
We'll *deconstruct* our page tables!



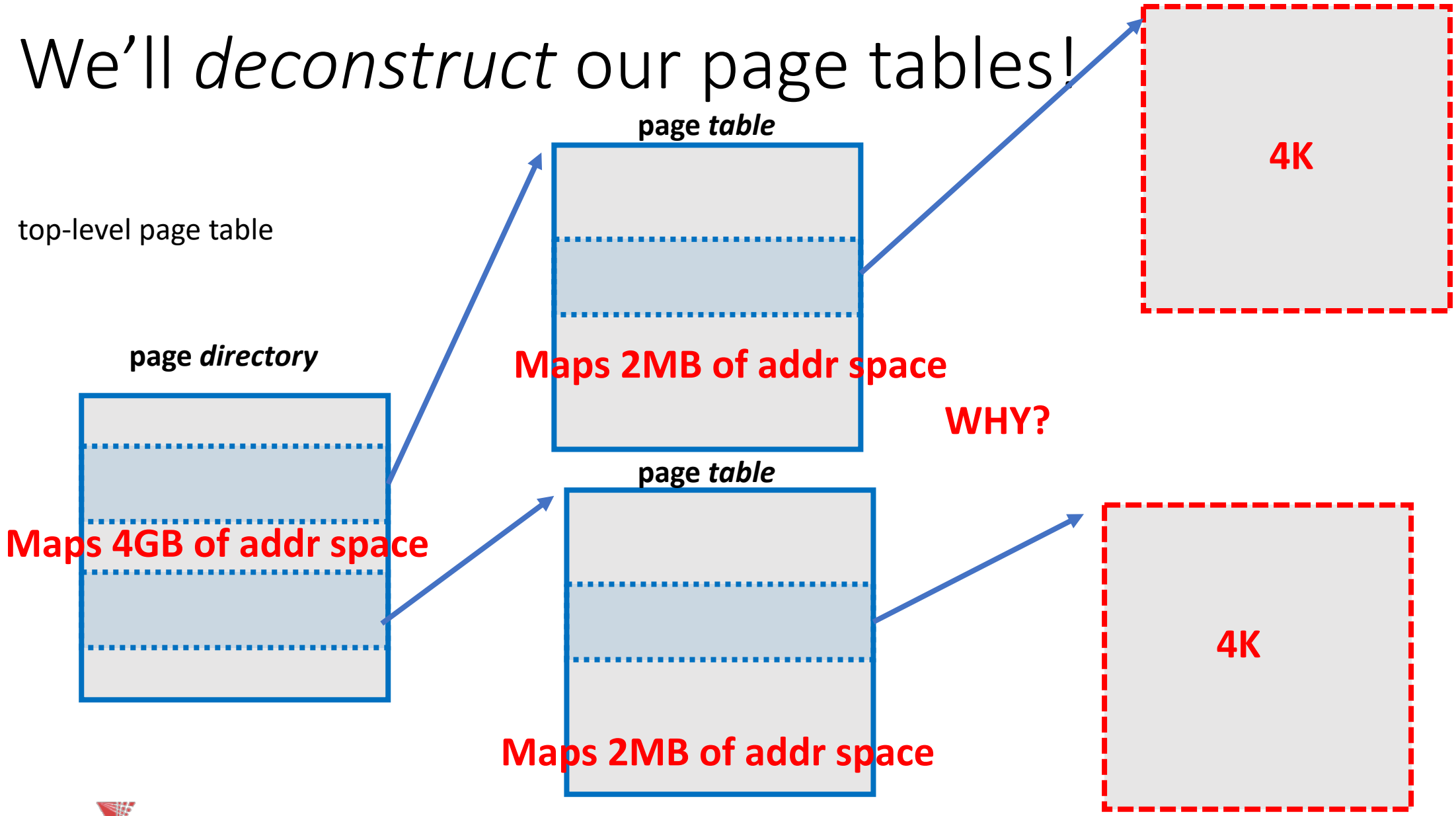
We'll *deconstruct* our page tables!



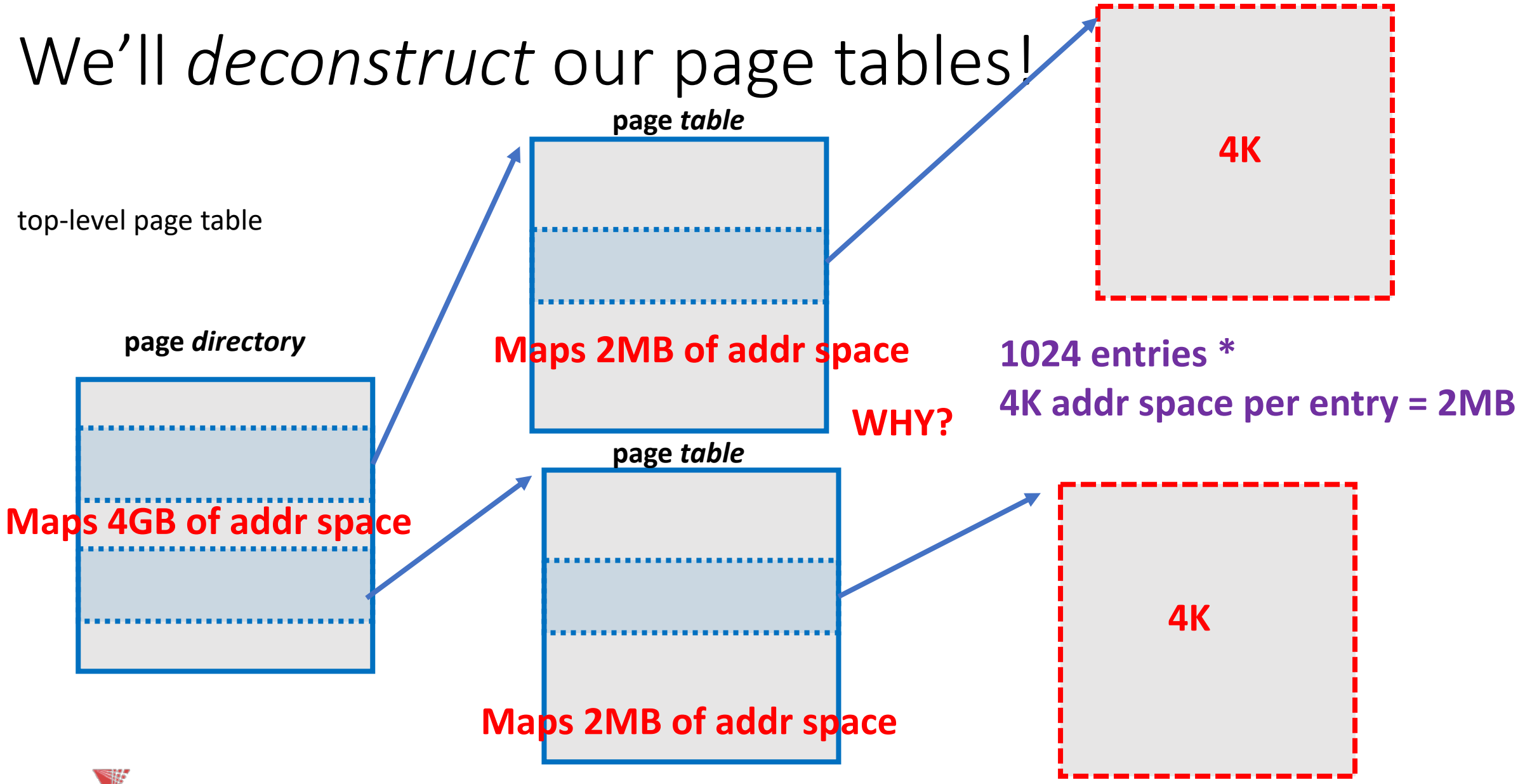
We'll *deconstruct* our page tables!



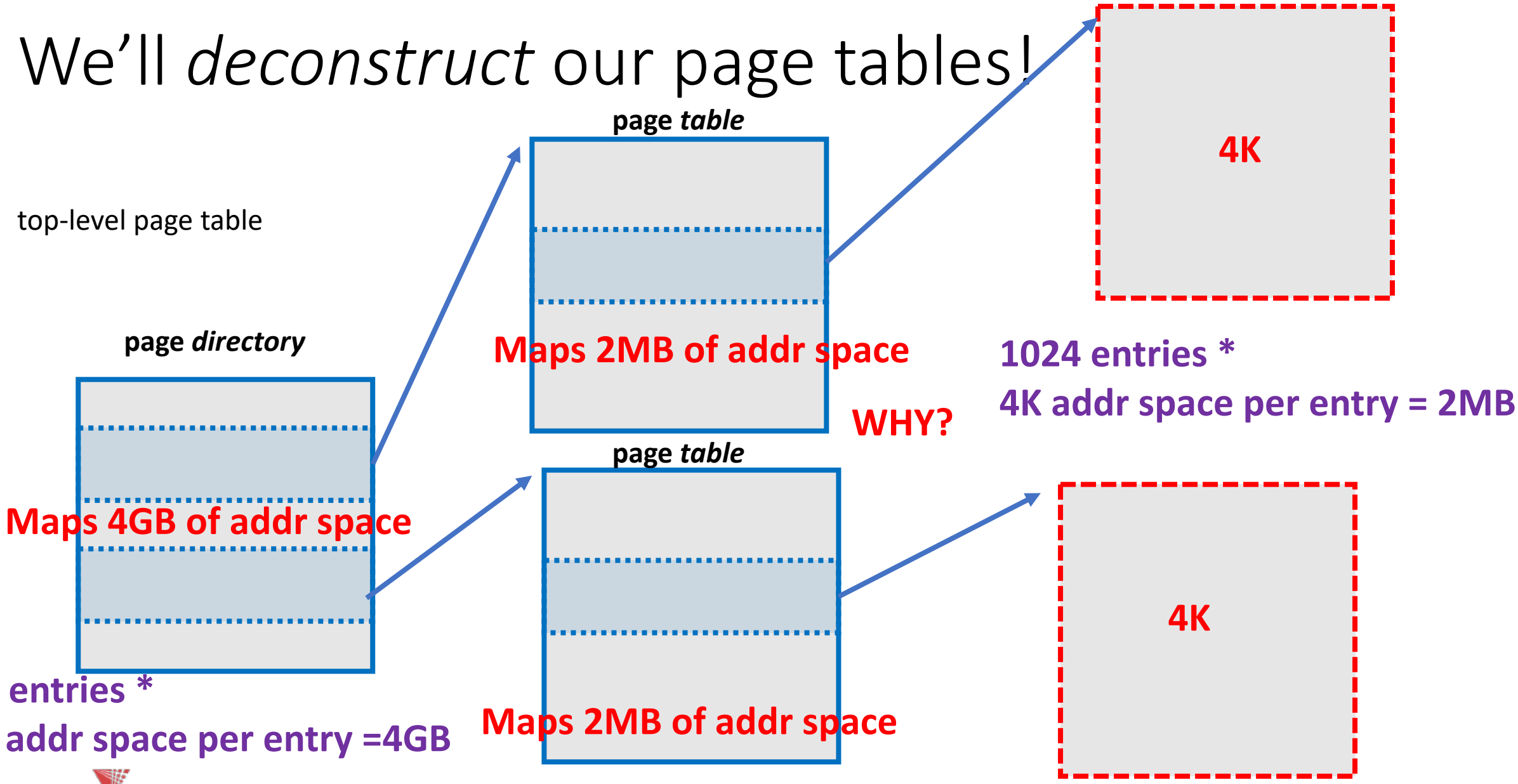
We'll *deconstruct* our page tables!



We'll *deconstruct* our page tables!



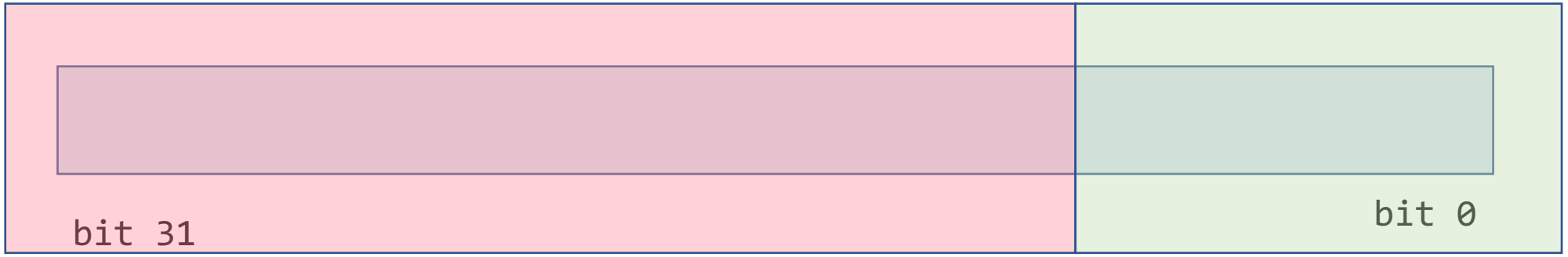
We'll *deconstruct* our page tables!



This changes our address breakdown. How?

Recall…

Use the virtual address!

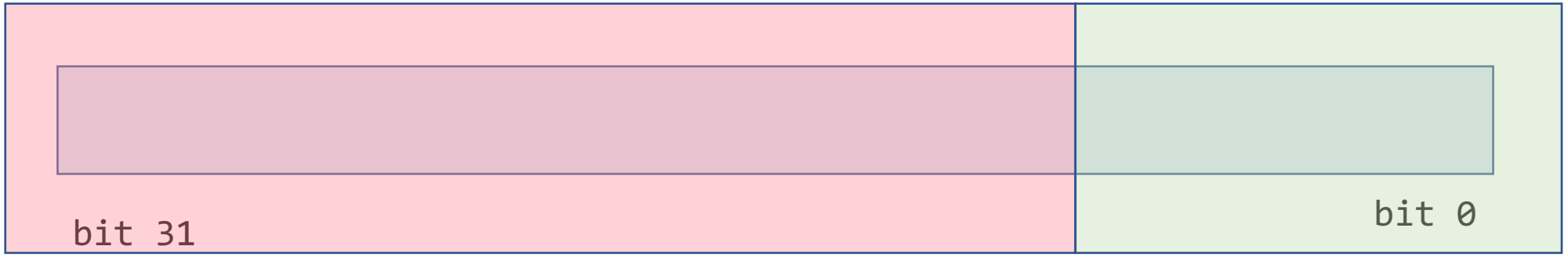


20 bits for table index
12 bits for offset

But one table index is now not enough. We need one more. What to do?

Recall…

Use the virtual address!



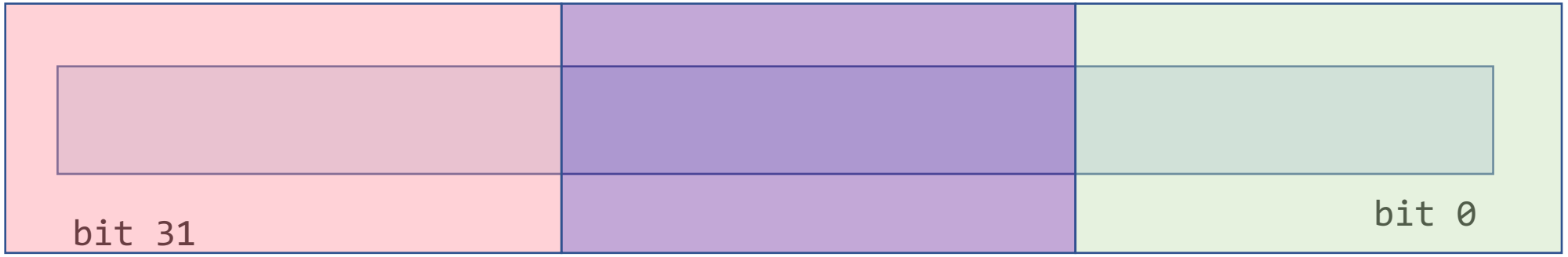
20 bits for table index
12 bits for offset

HINT: how many entries in the PD? How many entries in the PT?

How many bits do you need to index that many entries???

This changes our address breakdown. How?

Use the virtual address!



first-level table (PD) index
10 BITS

second-level table (PT) index
10 BITS

offset into page
(stays the same)

Who's in charge

- The OS still needs to create these page tables
- ***Only mapped regions in the address space will have entries*** which point to them
- When OS maps *new* regions, it must create new entries ***in both levels*** of page tables

The *Page Walk*

- Hardware (MMU) is in charge of *traversing* the page tables
- But it's more complicated now. Now longer just a table lookup
- ***Double* table lookups**
- Another way (more technical) to see it: our page table is no longer an associative array, but rather **a *radix tree***
- Hardware must do a ***radix tree* lookup**

Example

- OS is creating a new process
- Needs to create a stack for that process, and create valid mappings
- in the page tables for it
- First has to create the page tables!

Example

```
struct proc *  new_process() {
        struct proc * proc = kmalloc(sizeof(*proc));
        proc->pid          = next_pid++;
        proc->pd           = alloc_page();
        proc->stack_phys   = alloc_page();

        stack_va = create_mapping(proc->pd, proc->stack_phys)
        proc->regs->sp = stack_va + 0x1000 – 1;

}
```

Example

```
struct proc *  new_process() {
    struct proc * proc = kmalloc(sizeof(*proc));
    proc->pid          = next_pid++;
    proc->pd           = alloc_page();
    proc->stack_phys   = alloc_page();

    stack_va = create_mapping(proc->pd, proc->stack_phys)
    proc->regs->sp = stack_va + 0x1000 – 1;

}
```

allocate a new process struct

Example

```
struct proc *  new_process() {
        struct proc * proc = kmalloc(sizeof(*proc));
        proc->pid          = next_pid++;
        proc->pd           = alloc_page();
        proc->stack_phys   = alloc_page();

        stack_va = create_mapping(proc->pd, proc->stack_phys)
        proc->regs->sp = stack_va + 0x1000 – 1;

}
```

give it the next PID

Example

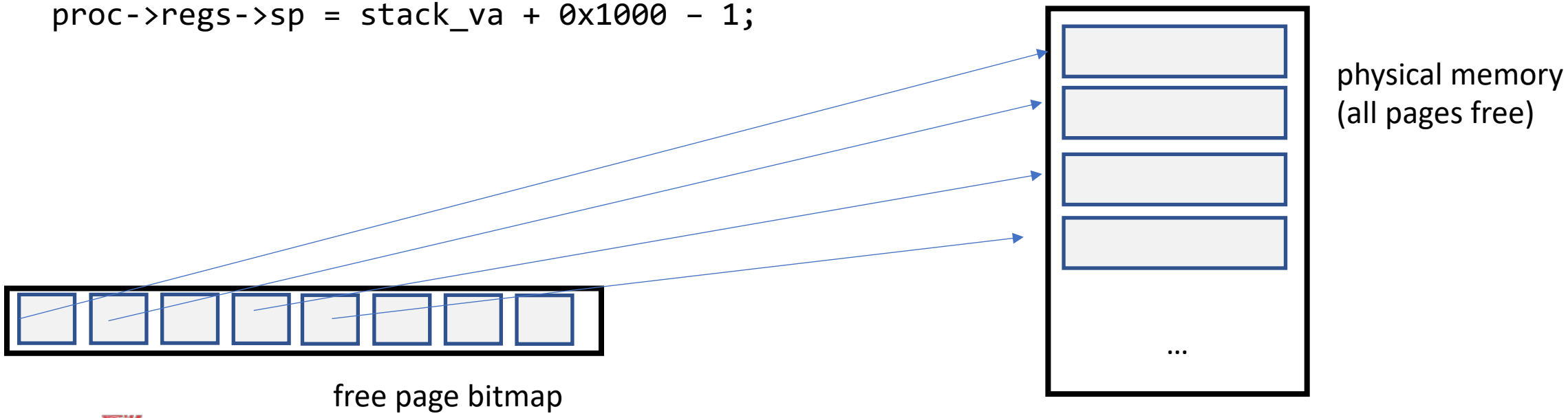
```
struct proc *  new_process() {
        struct proc * proc = kmalloc(sizeof(*proc));
        proc->pid          = next_pid++;
        proc->pd           = alloc_page();
        proc->stack_phys   = alloc_page();

        stack_va = create_mapping(proc->pd, proc->stack_phys)
        proc->regs->sp = stack_va + 0x1000 – 1;

}
```

allocate a physical page for this process's top-level page table (the page directory)

Example

```
struct proc *  new_process() {
        struct proc * proc = kmalloc(sizeof(*proc));
        proc->pid          = next_pid++;
        proc->pd           = alloc_page();
        proc->stack_phys   = alloc_page();

        stack_va = create_mapping(proc->pd, proc->stack_phys)
        proc->regs->sp = stack_va + 0x1000 – 1;

}
```

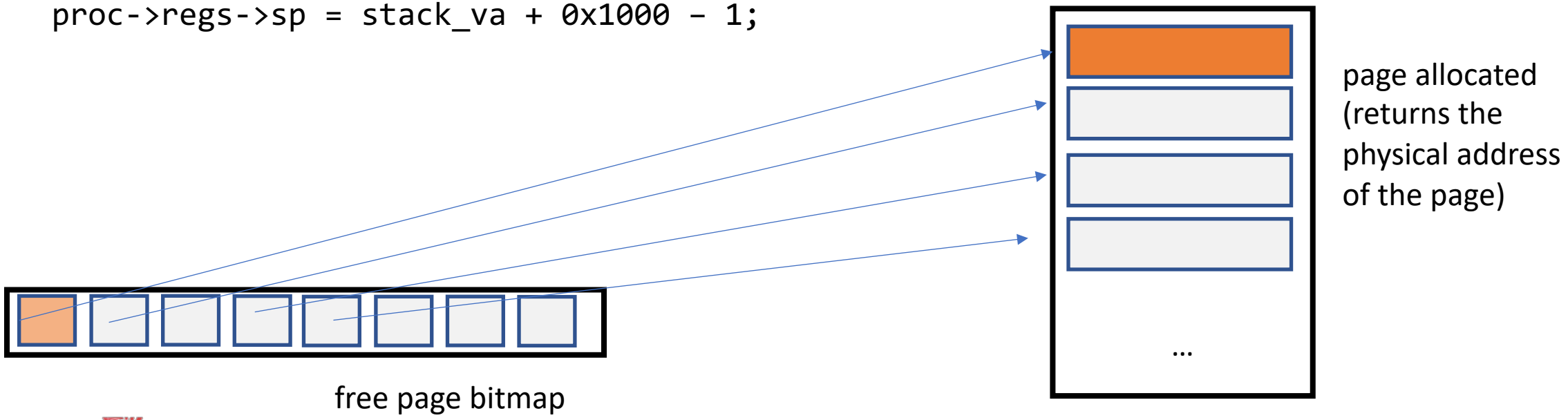
allocate a physical page for this process's top-level page table (the page directory)



Example

```
struct proc *  new_process() {
        struct proc * proc = kmalloc(sizeof(*proc));
        proc->pid          = next_pid++;
        proc->pd           = alloc_page();
        proc->stack_phys   = alloc_page();

        stack_va = create_mapping(proc->pd, proc->stack_phys)
        proc->regs->sp = stack_va + 0x1000 – 1;

}
```

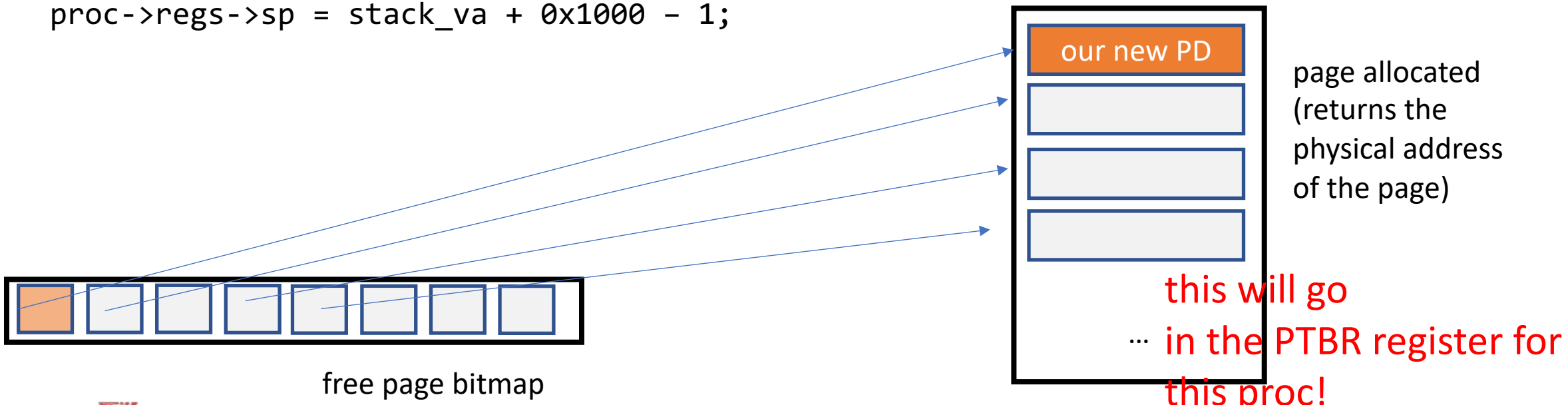
allocate a physical page for this process's top-level page table (the page directory)



Example

```
struct proc *  new_process() {
        struct proc * proc = kmalloc(sizeof(*proc));
        proc->pid          = next_pid++;
        proc->pd           = alloc_page();
        proc->stack_phys   = alloc_page();

        stack_va = create_mapping(proc->pd, proc->stack_phys)
        proc->regs->sp = stack_va + 0x1000 – 1;

}
```

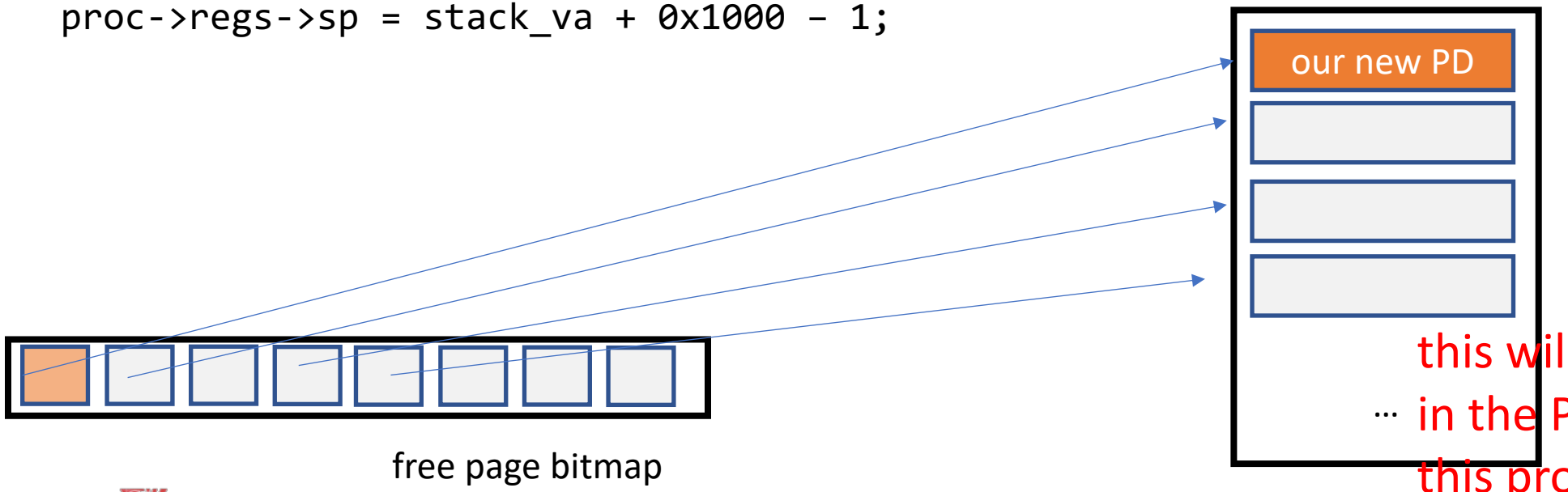
allocate a physical page for this process's top-level page table (the page directory)



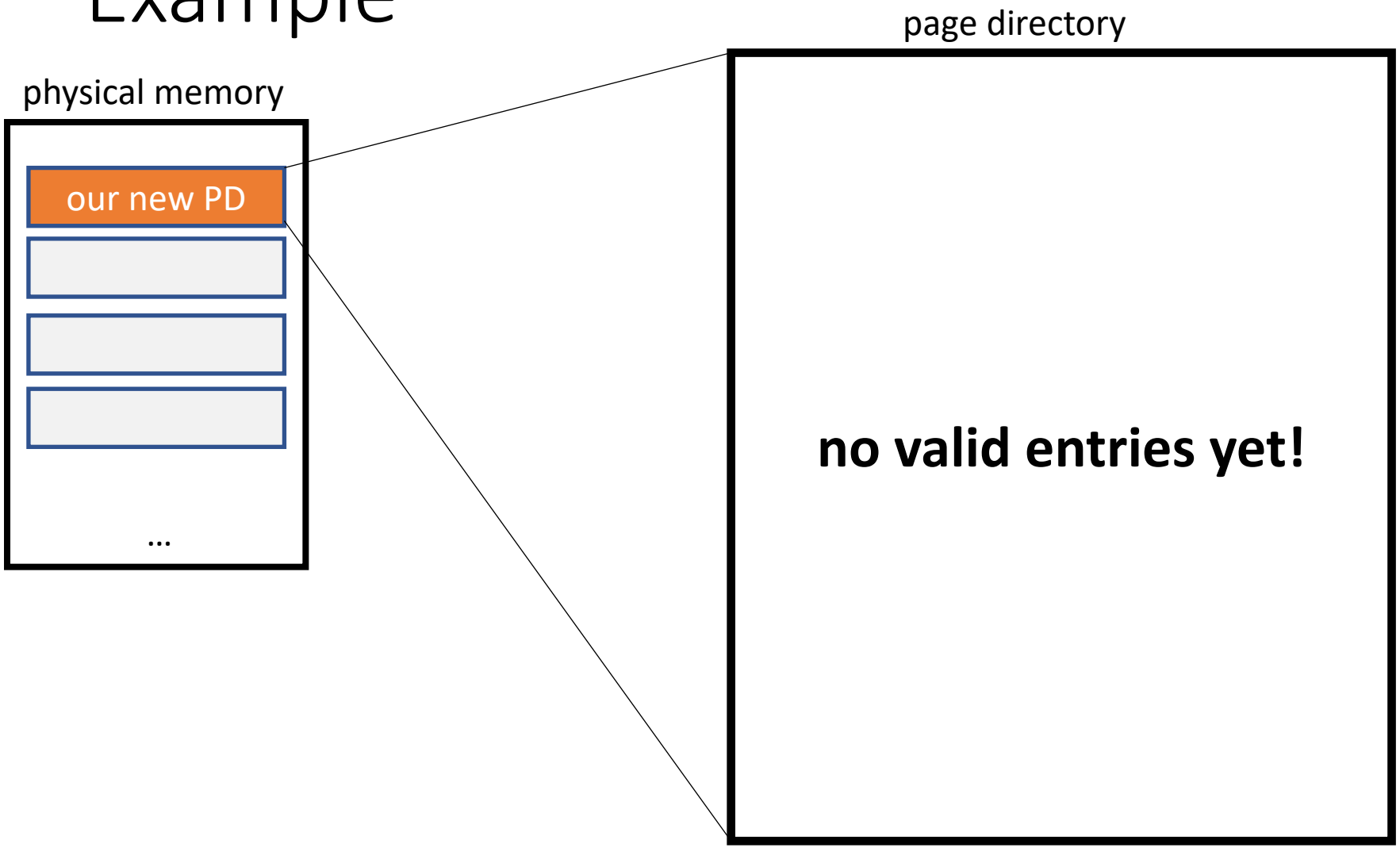
Example

```
struct proc *  new_process() {
        struct proc * proc = kmalloc(sizeof(*proc));
        proc->pid          = next_pid++;
        proc->pd           = alloc_page();
        proc->stack_phys   = alloc_page();

        stack_va = create_mapping(proc->pd, proc->stack_phys)
        proc->regs->sp = stack_va + 0x1000 – 1;

}
```

allocate a physical page for this process's top-level page table (the page directory)



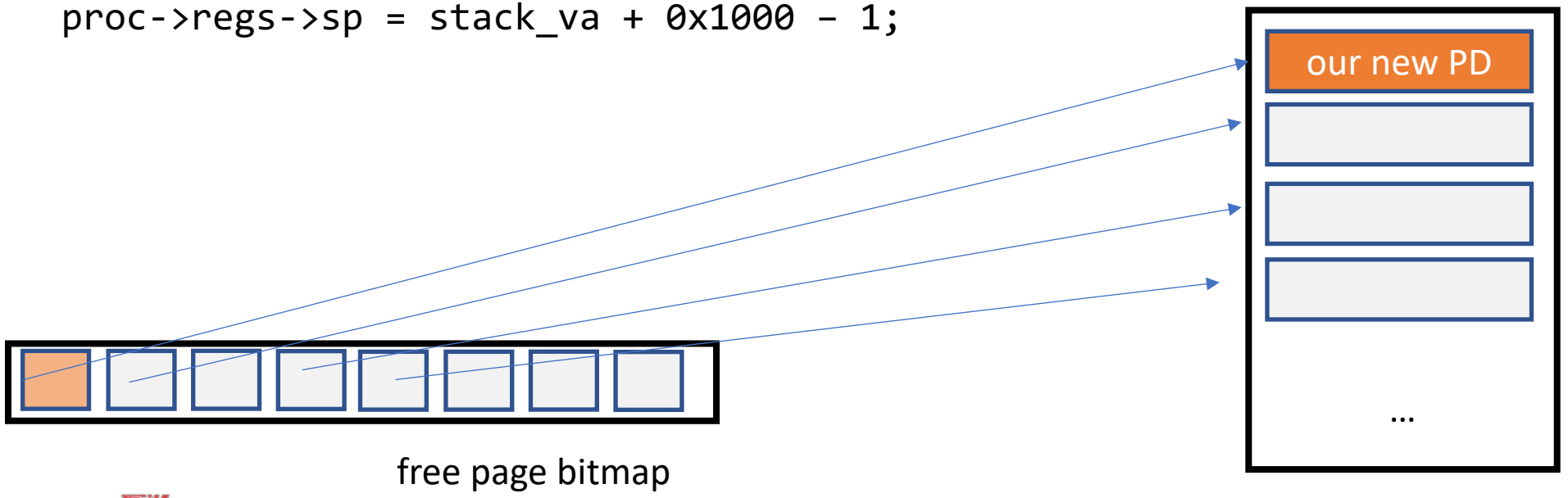
Example


page directory
physical memory
our new PD
...
no valid entries yet!

Example

```
struct proc *  new_process() {
        struct proc * proc = kmalloc(sizeof(*proc));
        proc->pid          = next_pid++;
        proc->pd           = alloc_page();
        proc->stack_phys   = alloc_page();

        stack_va = create_mapping(proc->pd, proc->stack_phys)
        proc->regs->sp = stack_va + 0x1000 – 1;

}
```

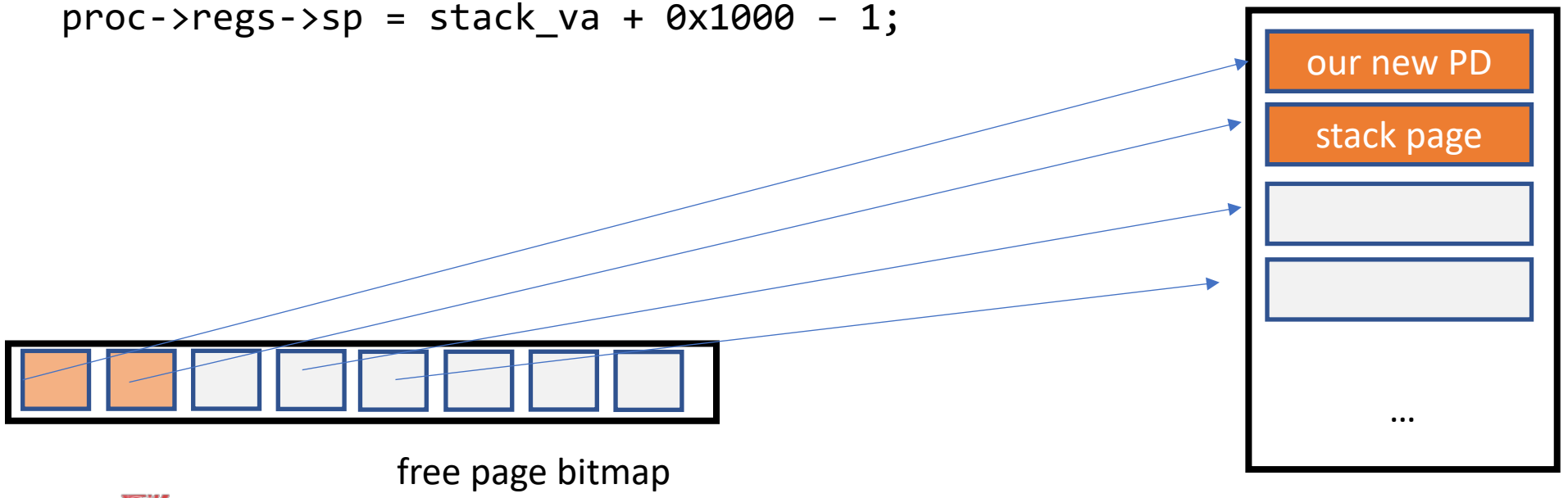
allocate a physical page for this process's stack



Example

```
struct proc *  new_process() {
        struct proc * proc = kmalloc(sizeof(*proc));
        proc->pid          = next_pid++;
        proc->pd           = alloc_page();
        proc->stack_phys   = alloc_page();

        stack_va = create_mapping(proc->pd, proc->stack_phys)
        proc->regs->sp = stack_va + 0x1000 – 1;

}
```

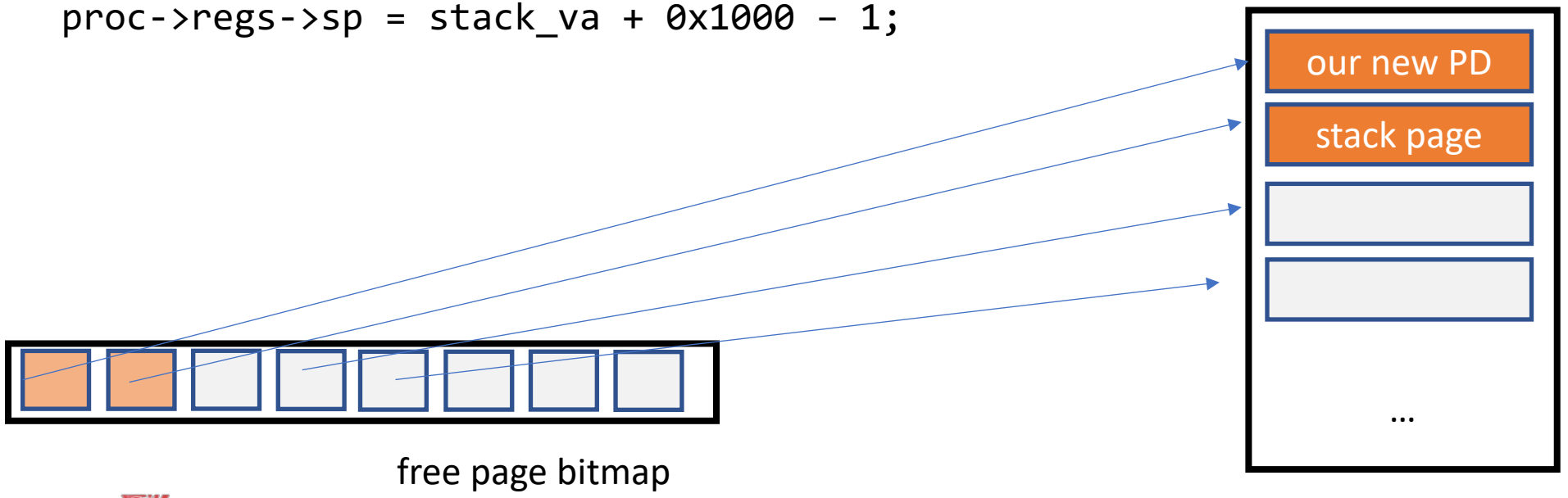
allocate a physical page for this process's stack



Example

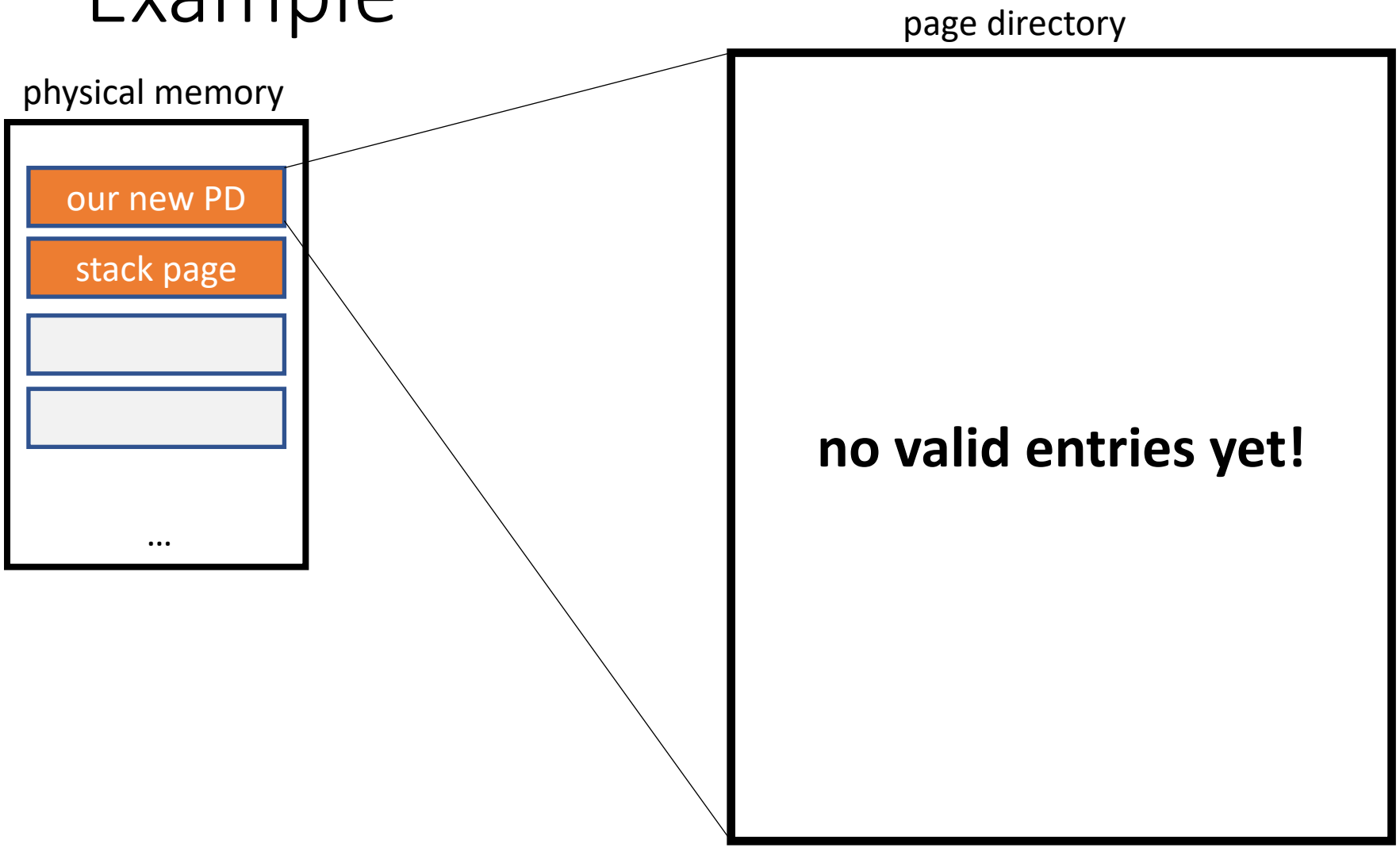
```
struct proc *  new_process() {
        struct proc * proc = kmalloc(sizeof(*proc));
        proc->pid          = next_pid++;
        proc->pd           = alloc_page();
        proc->stack_phys   = alloc_page();

        stack_va = create_mapping(proc->pd, proc->stack_phys)
        proc->regs->sp = stack_va + 0x1000 – 1;

}
```

tie everything together by mapping it



```
create_mapping(proc->pd, proc->stack_phys)
```

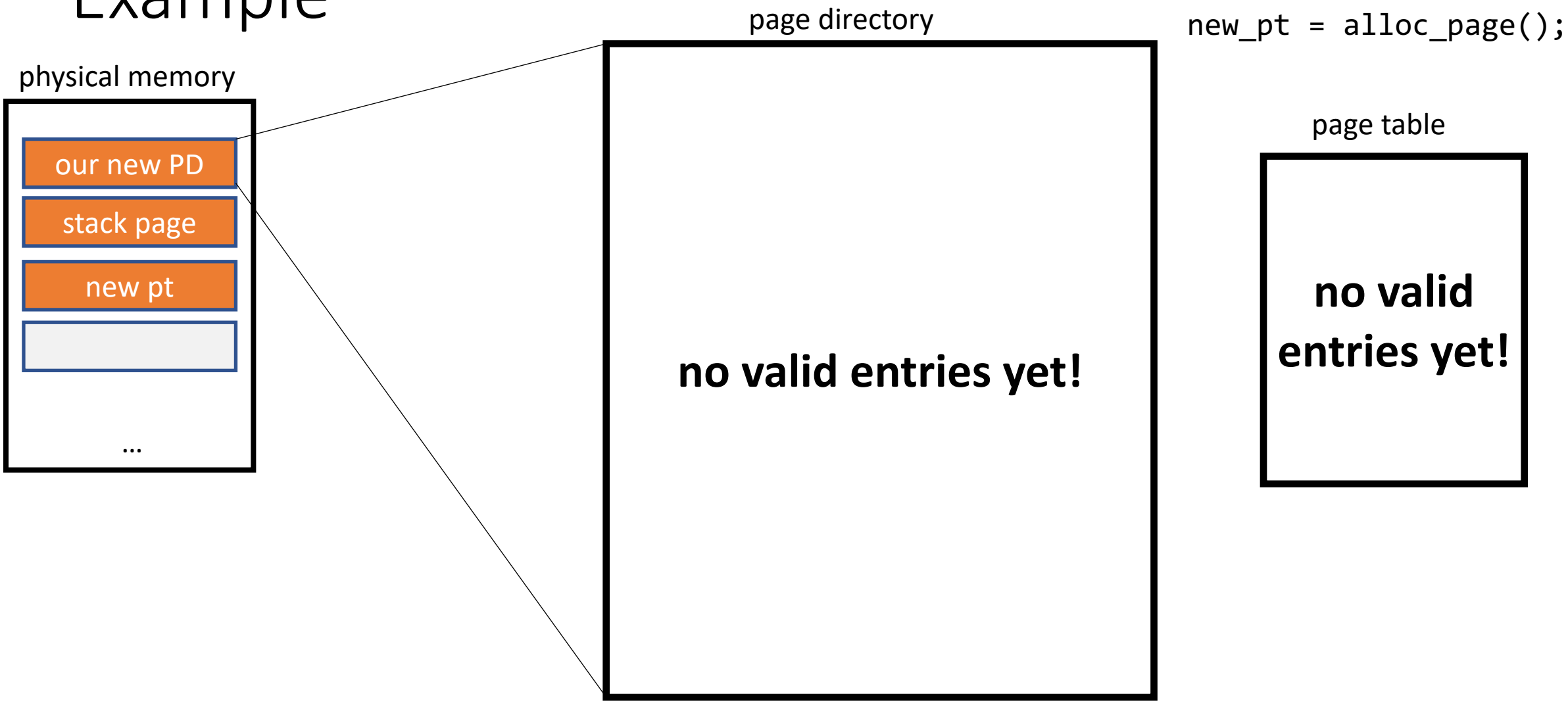
Example



Example

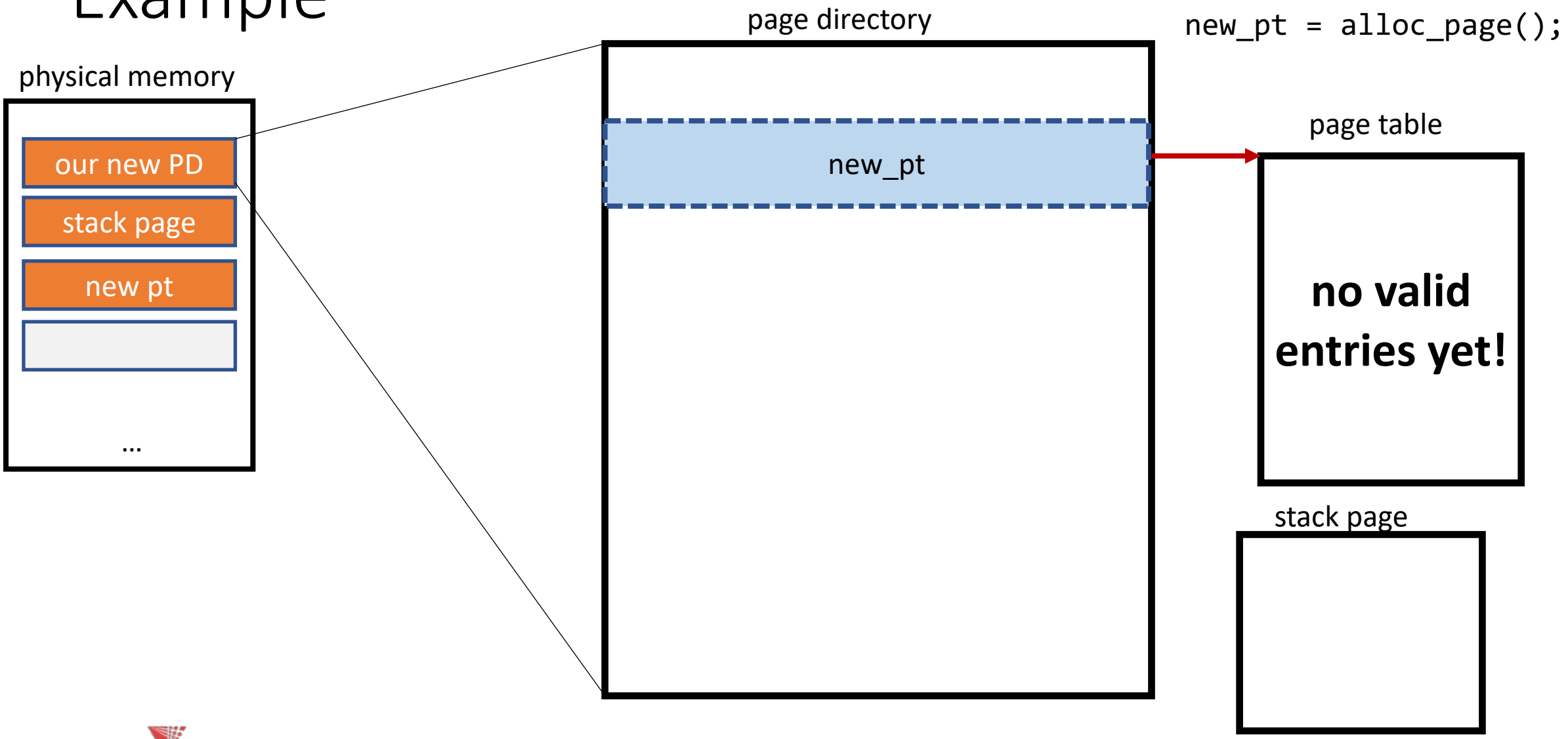
```
create_mapping(proc->pd, proc->stack_phys)
```

```
new_pt = alloc_page();
```



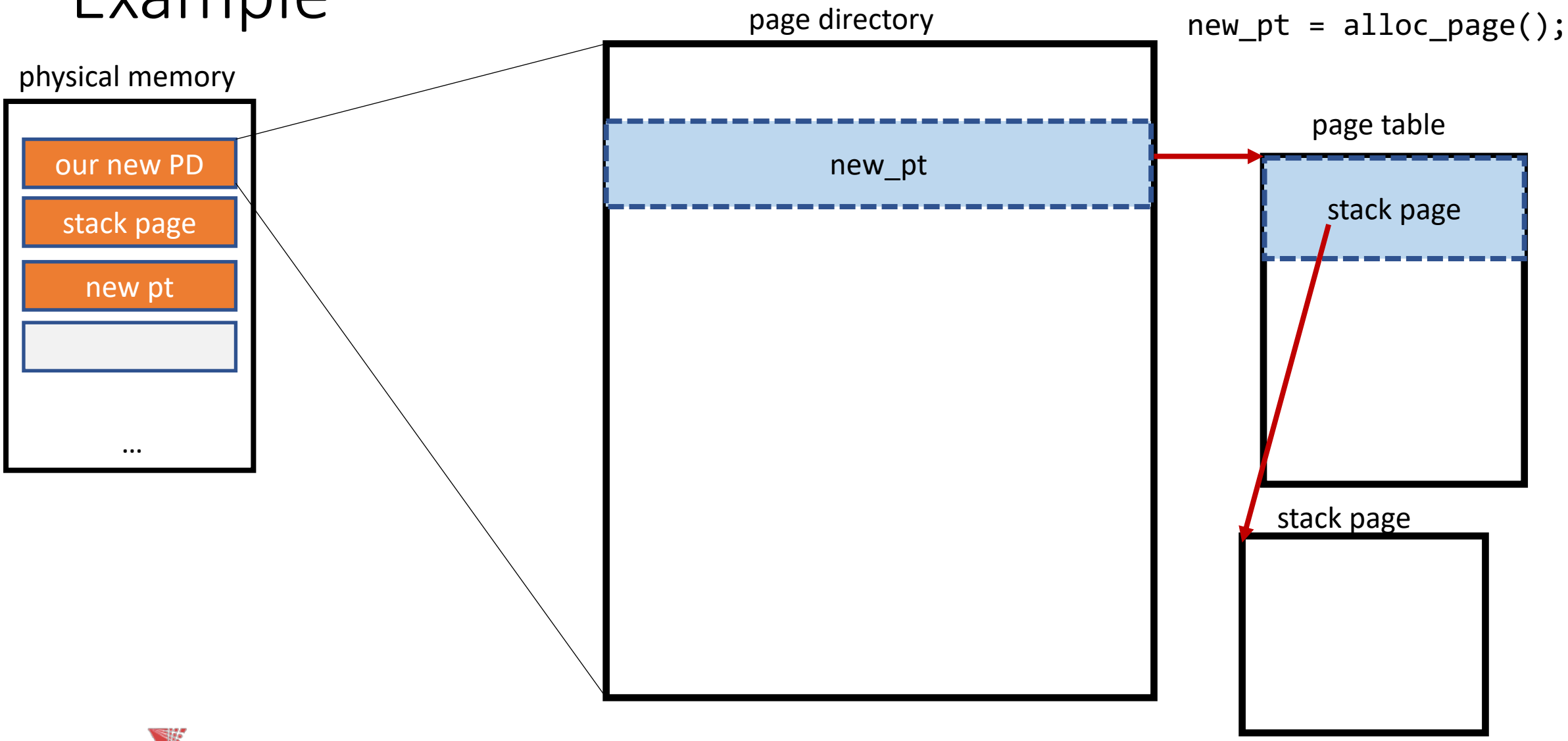
Example

```
create_mapping(proc->pd, proc->stack_phys)
```



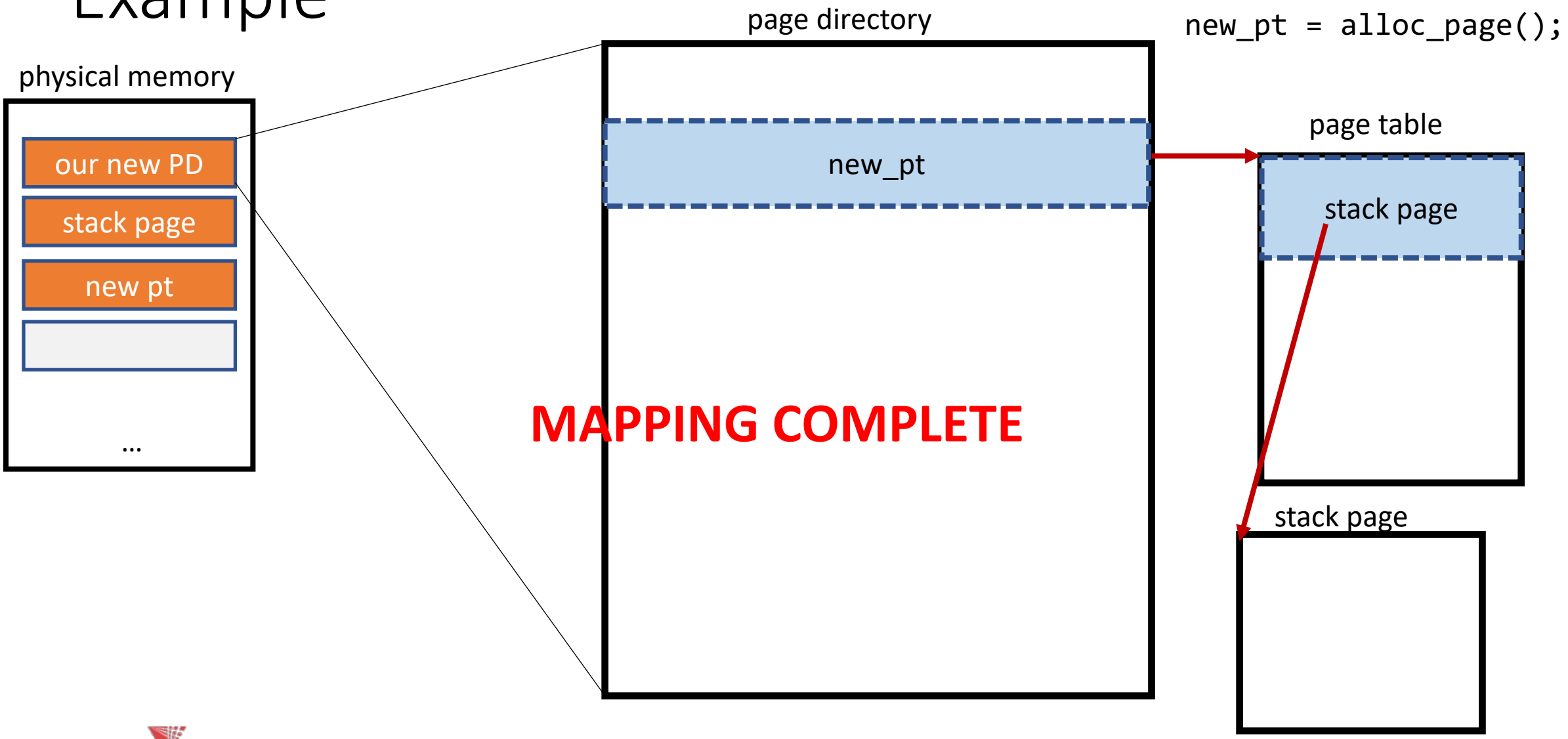
Example

```
create_mapping(proc->pd, proc->stack_phys)
```



Example

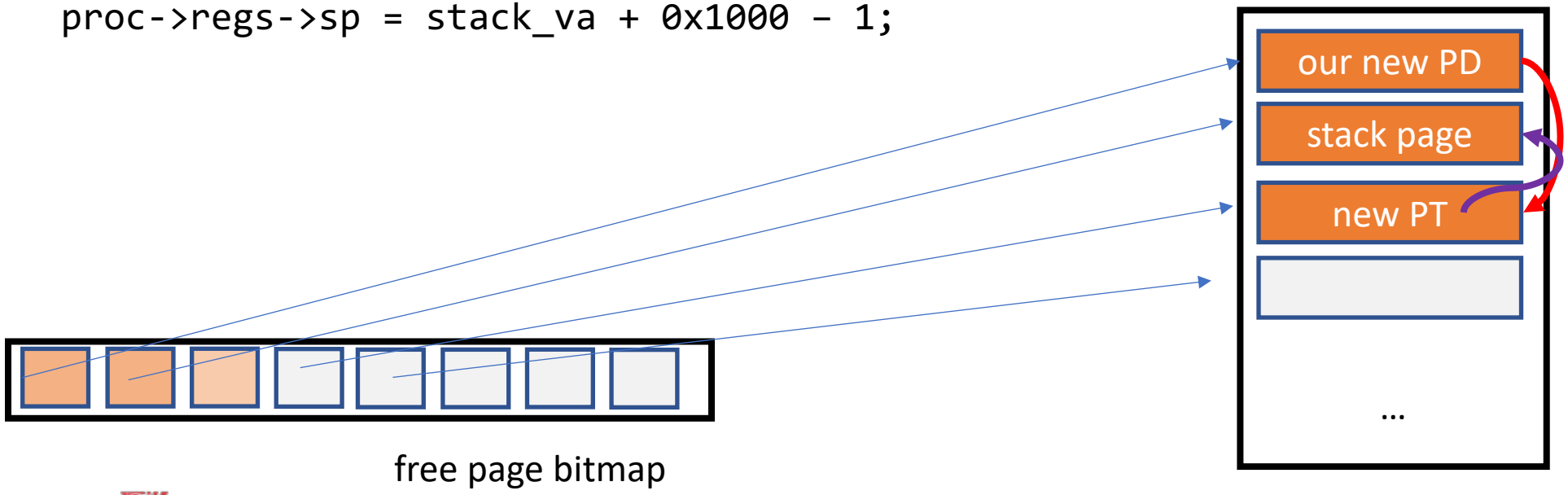
```
create_mapping(proc->pd, proc->stack_phys)
```



Example

mapped together

```
struct proc *  new_process() {
        struct proc * proc = kmalloc(sizeof(*proc));
        proc->pid          = next_pid++;
        proc->pd           = alloc_page();
        proc->stack_phys   = alloc_page();

        stack_va = create_mapping(proc->pd, proc->stack_phys)
        proc->regs->sp = stack_va + 0x1000 – 1;

}
```



Example: userspace code within our new process

```
void foo () {
    int some_var = 10;
    return bar(some_var);
}
```

Example: userspace code within our new process

```
void foo () {
    int some_var = 10;
    return bar(some_var);
}
```

```
push $0xa, 4(%ebp)
call bar
```

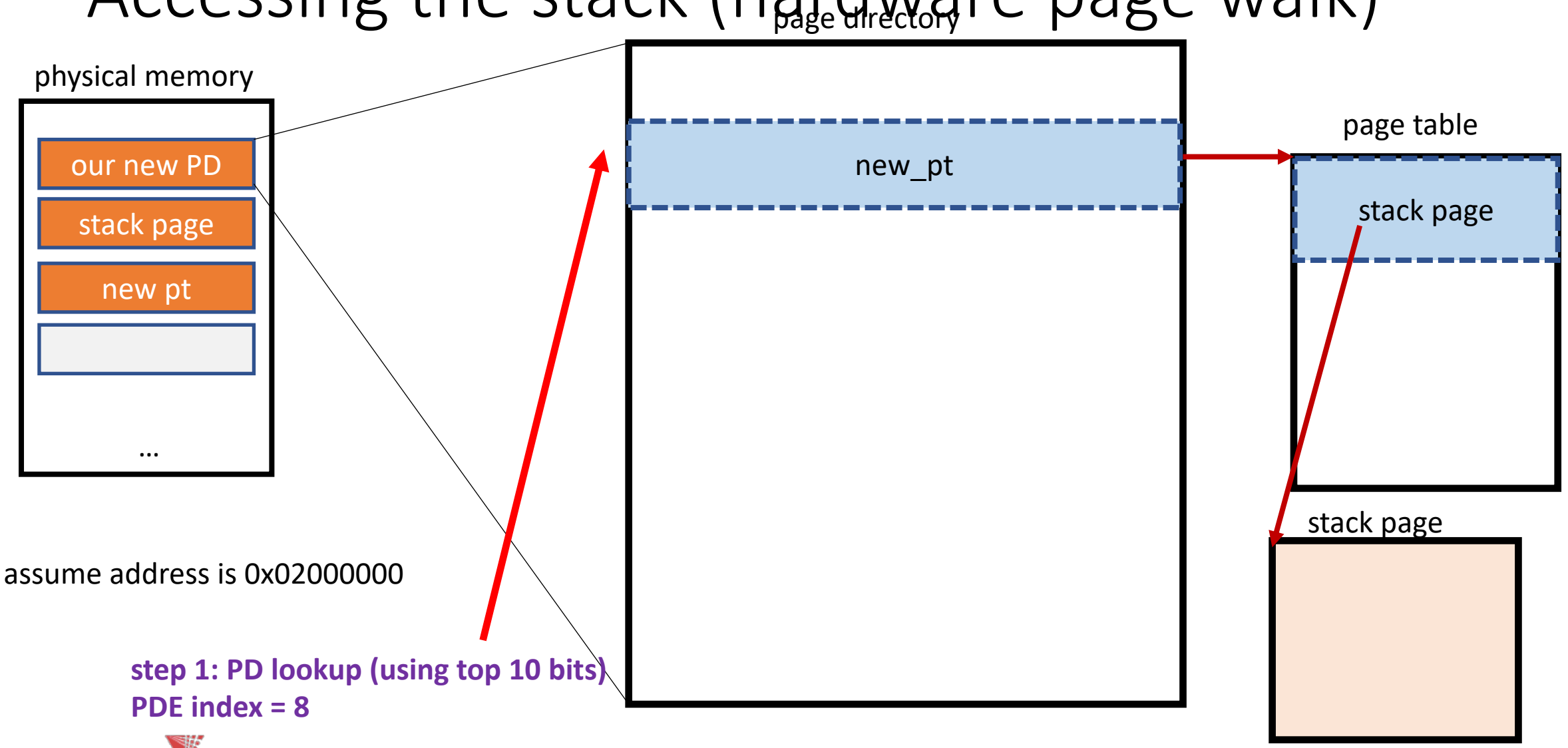
Example: userspace code within our new process

```
void foo () {
    int some_var = 10;
    return bar(some_var);
}
```

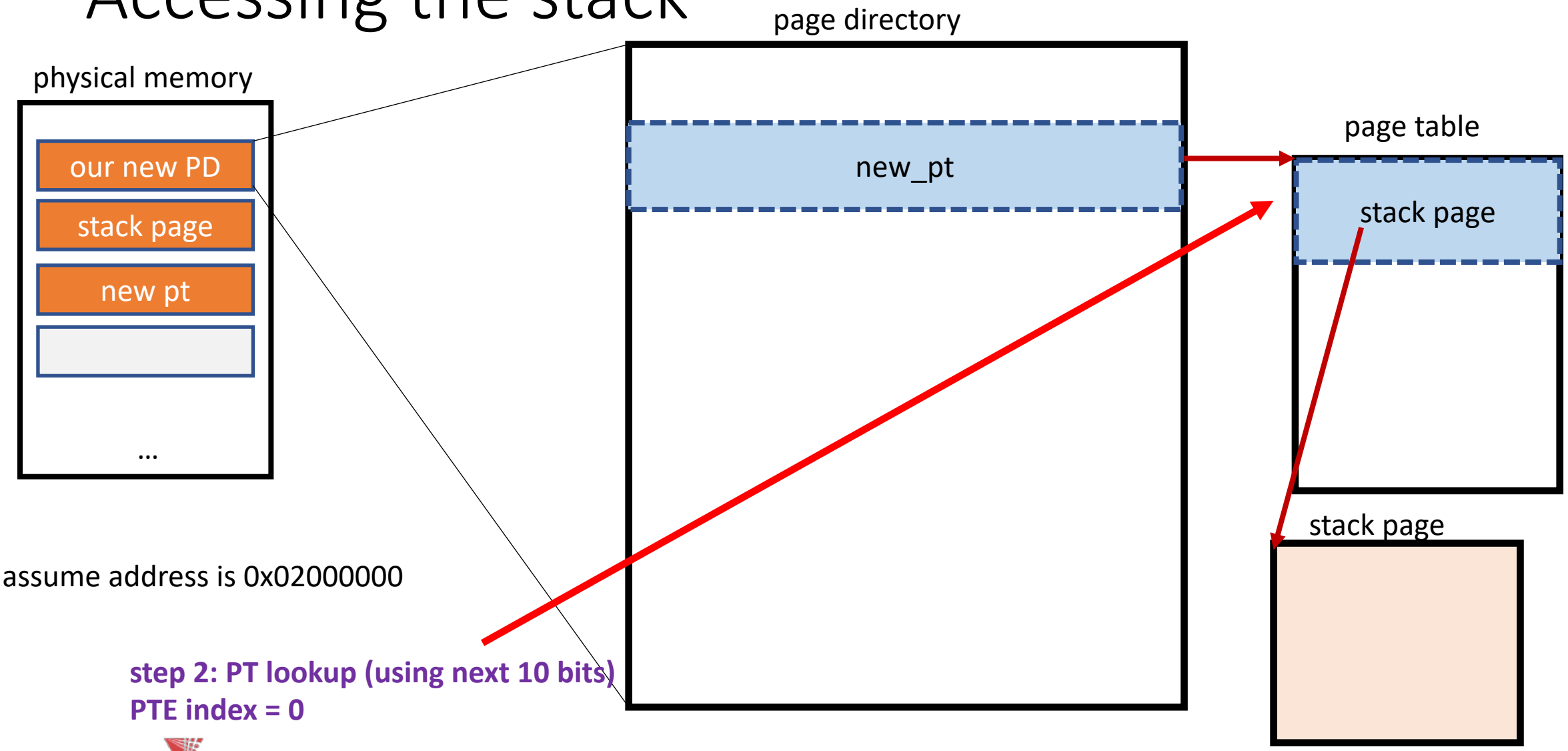
```
push $0xa, 4(%ebp)
call bar
```

frame pointer (address somewhere in stack page)

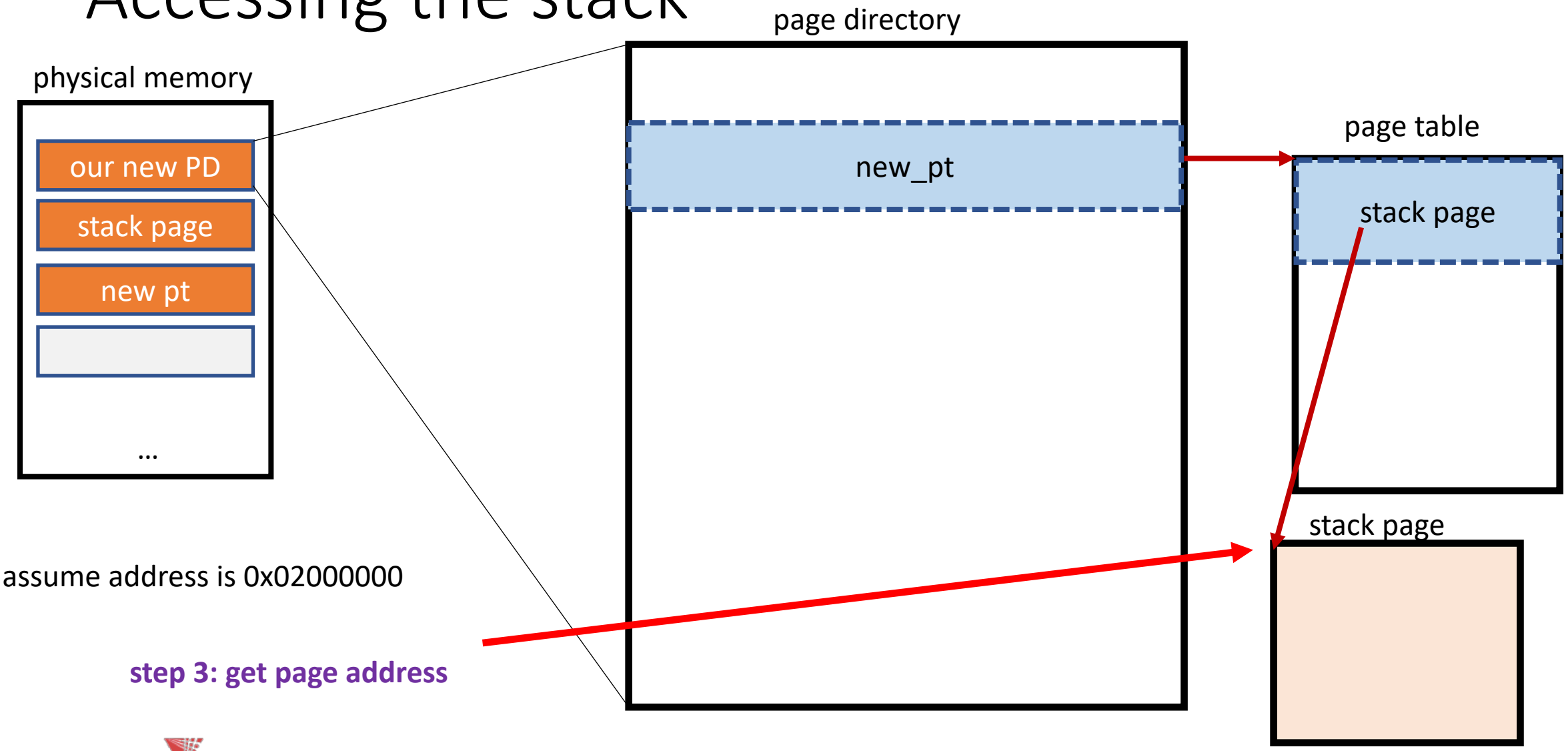
Accessing the stack (hardware page walk)



Accessing the stack



Accessing the stack



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: it depends now! On what?

How can we possibly recover from this?

$$ Cache $$

Enter the *Translation Lookaside Buffer (TLB)*

- Simple idea, fancy name: ***cache translations*** in very fast (on-chip) memory
  - Usually SRAM (just like L1$). Single-cycle (<1ns) access
- This is basically the single-level table we started with, just ***much smaller*** and ***much faster***

The TLB

- The TLB organization is almost *exactly* the same as the L1 cache
- SRAM, index bits, tag bits
- Entry # is computed by VA % entry_count
- Hit check is done comparing tag (lower order) bits of address
- Tag comparison can be done in parallel
- What is cached is the ***physical page number*** (not arbitrary data)
- TLB hit means **no page walk**!

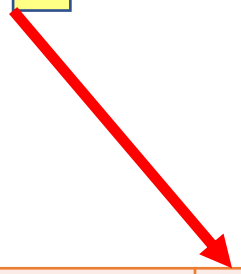
```
mov 0x00080000, 8(%ebx)
```

4 entries means 2 index bits

0	**0x70000000**
1	--
2	--
3	--

CPU

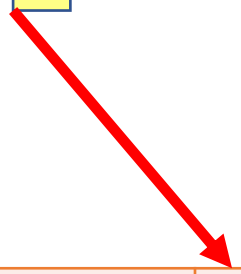
```
mov 0x00080000, 8(%ebx)
```



TLB Hit!

0	**0x70000000**
1	--
2	--
3	--

```
mov 0x00080000, 8(%ebx)
```

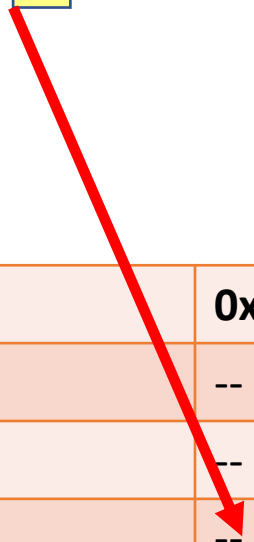


TLB Hit!

0	**0x70000000**
1	--
2	--
3	--

Single cycle! We only pay for one memory access (to physical memory at 0x70000000)

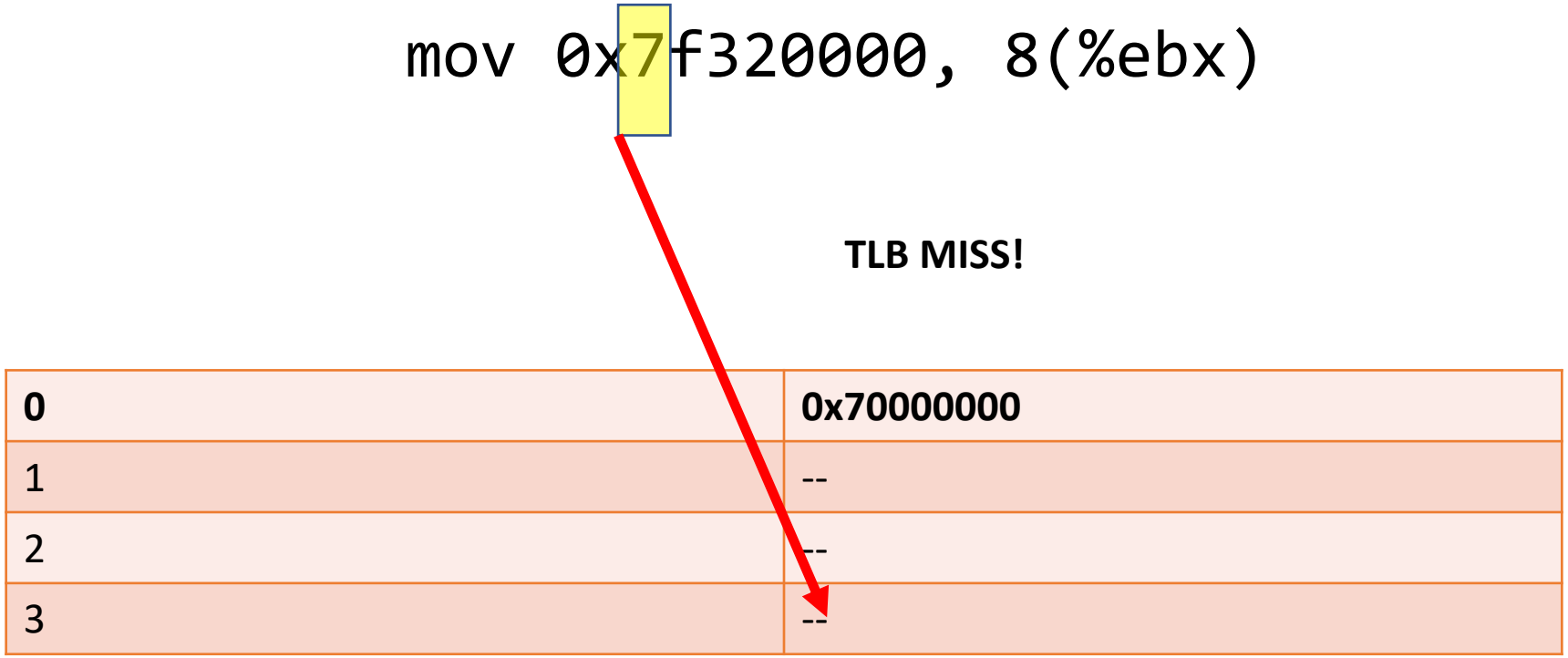
```
mov 0x7f320000, 8(%ebx)
```



TLB MISS!

0	**0x70000000**
1	--
2	--
3	--

```
mov 0x7f320000, 8(%ebx)
```



Page walk! = several more memory references

(The hardware will automatically start a page walk on a TLB miss)

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: it depends now! On what?

How big is my TLB?

on a mac

```
$ sysctl machdep.cpu | grep -i tlb
machdep.cpu.tlb.inst.large: 8
machdep.cpu.tlb.data.small: 64
machdep.cpu.tlb.data.small_level1: 64
# kyle
```

64-entry TLB for data (specifically for small pages). Do your reading to get more info on this

on Linux*

```
$ cpuid | grep -i tlb
   cache and TLB information (2):
      0x63: data TLB: 2M/4M pages, 4-way, 32 entries
            data TLB: 1G pages, 4-way, 4 entries
      0x03: data TLB: 4K pages, 4-way, 64 entries
      0x76: instruction TLB: 2M/4M pages, fully, 8 ent
      0xb5: instruction TLB: 4K, 8-way, 64 entries
```

Note the variable page sizes

***sorry, I don't have a Windows box**

Trends

- 64-bit machines use 4-level page tables (Intel is suggesting 5-level soon)
- How many memory references does that mean for a TLB miss?
- ***Cost of a TLB miss is going up*** (especially with a virtual machines). Take CS 562 with me to find out why
- Datasets are getting bigger (***more pressure*** on the TLB)

Summary

- Multi-level page tables are here to stay (for now)
- They ***decrease page table overhead, wasted space***
- But ***they increase page walk overhead***
- To mitigate this overhead, we introduced a ***translation cache (TLB)*** on-chip