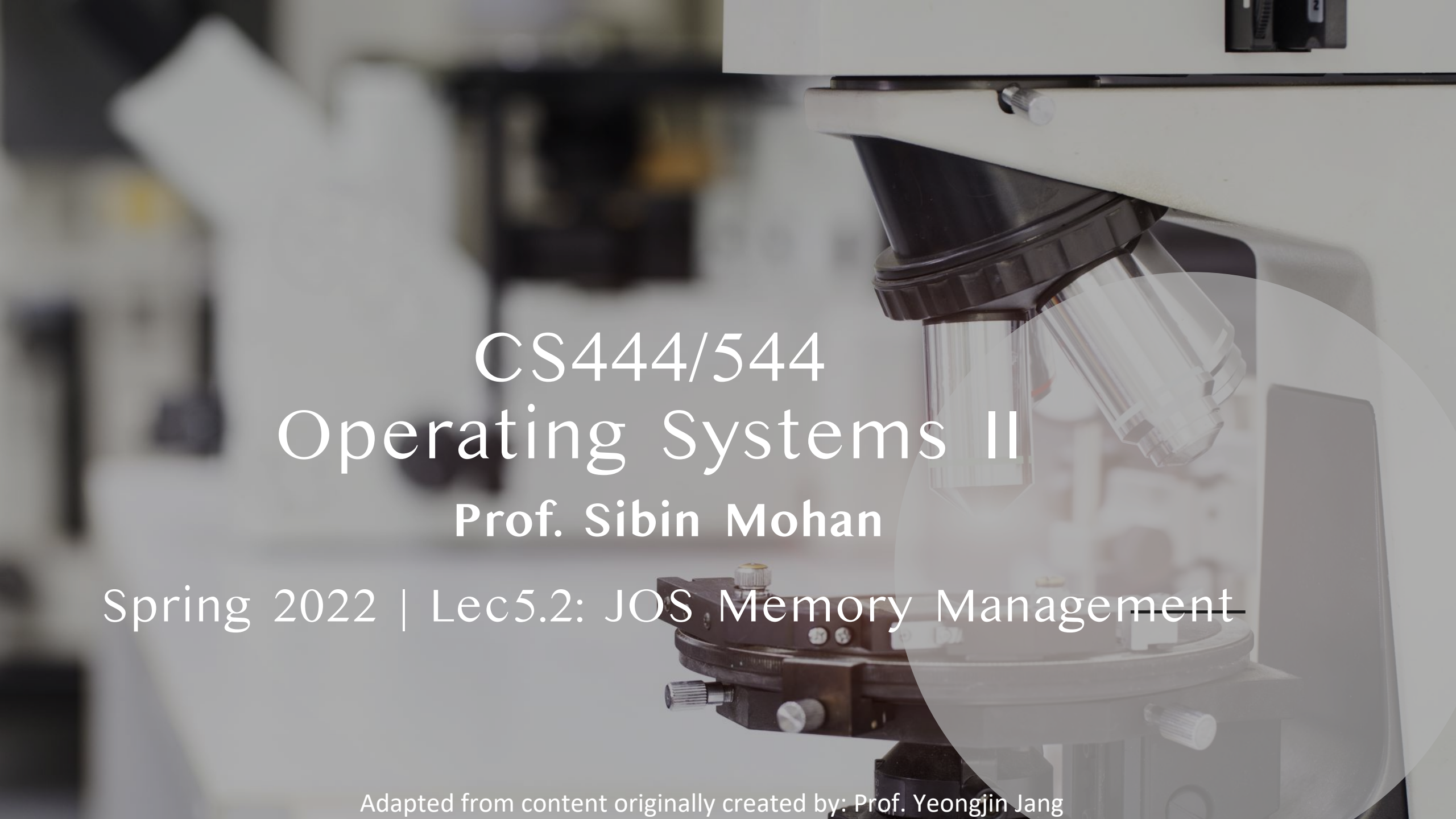


CS444/544
Operating Systems II

Prof. Sibin Mohan

Spring 2022 | Lec5.2: JOS Memory Management

Adapted from content originally created by: Prof. Yeongjin Jang

Administrivia

- Lab 1 due date: **April 15, 2022** [**Friday**] at **11:59 PM**!
  - Lots of office hours this week → USE THEM WELL!
- Quiz 1 on **April 14, 2022** [**Thursday**] at **8:30 AM**!
  - Available until **April 15, 2022** [**Friday**], **11:59 PM**
- Lab 2 announced [Memory Management]
  - Due **April 29, 2022** [**Friday**] at **11:59 PM**

- **Tutorial 3:** helpful for both, Quiz 1 and Lab 2.

Today's Topics

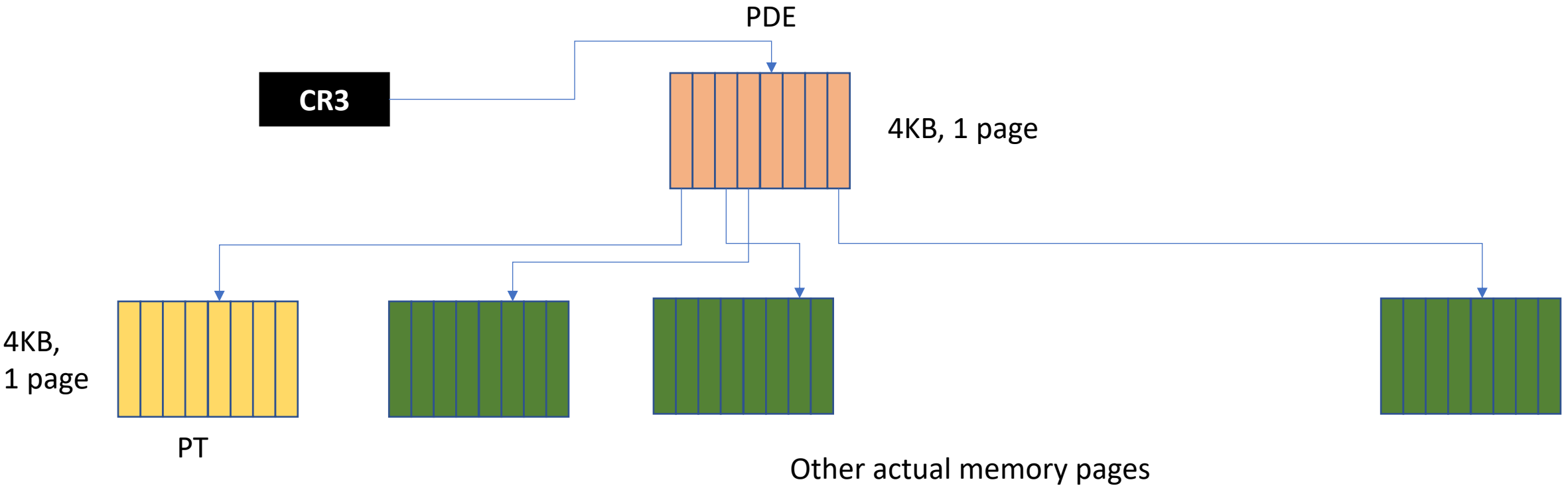
Managing Physical/Virtual Memory in JOS

Prep for Quiz 1

JOS Memory Management

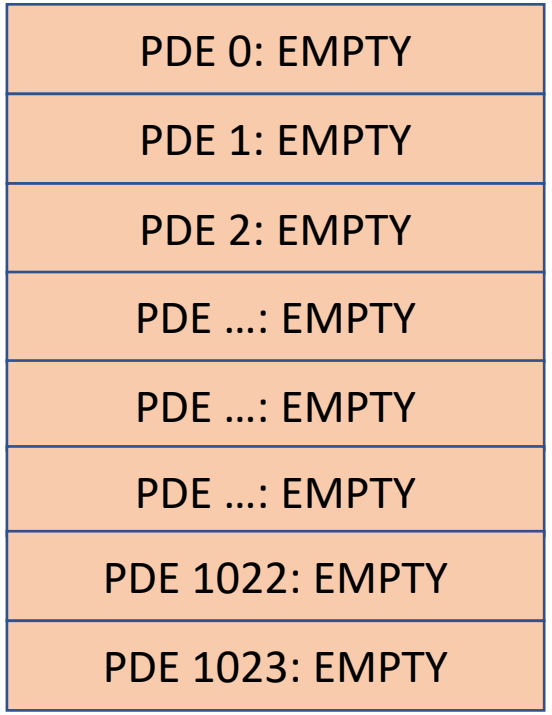
Creating a Virtual Memory Space

- **Page Directory** [PDE] manages entire virtual memory space for a process
- **CR3** points to the PDE
  - each PDE entry points to multiple PTs



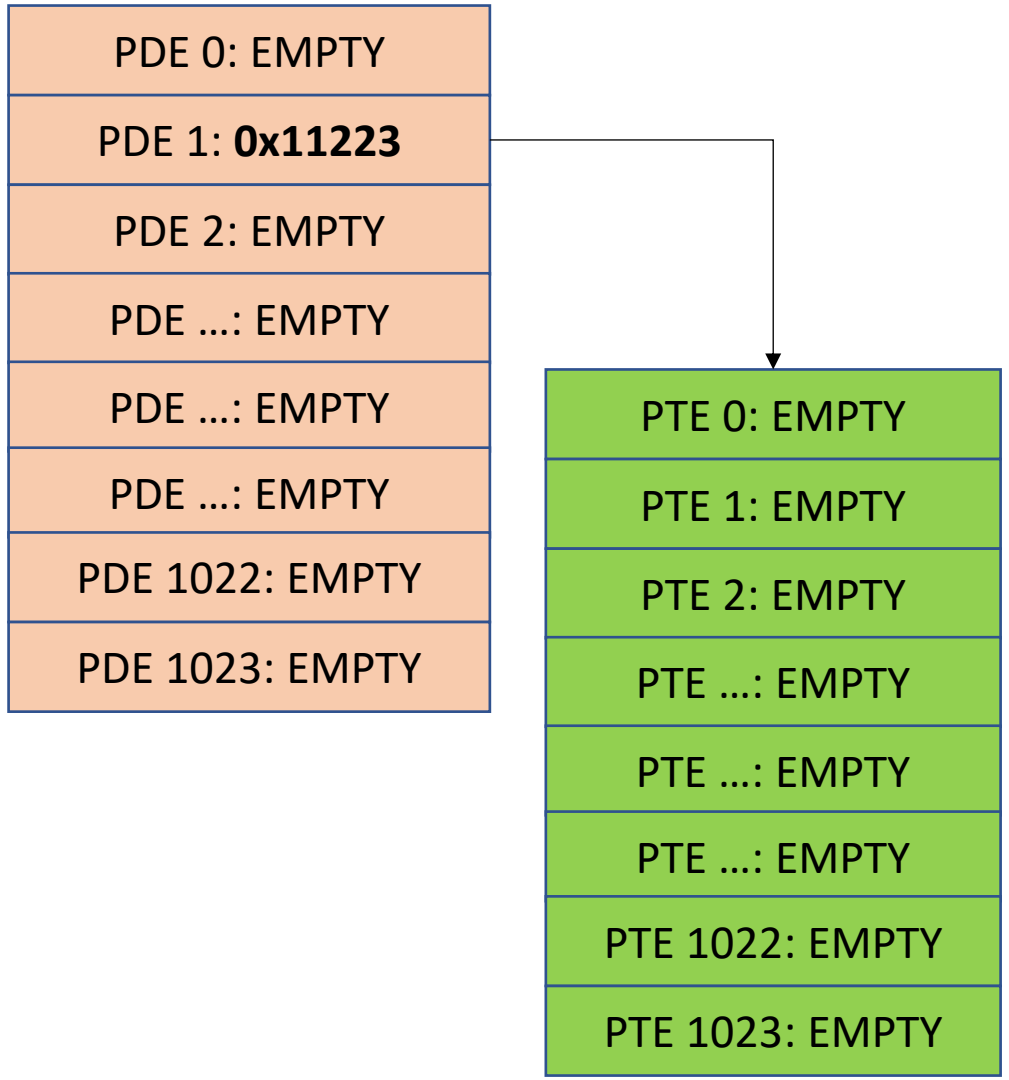
Assigning VA -> PA mapping

- Say, a process wants to access a virtual address
  - E.g., 0x800000
- Allocation procedure
  1. **Check page directory entry** (PDE)
  2. If not set with **PTE_P**
  3. **allocate a physical page** for a new page table



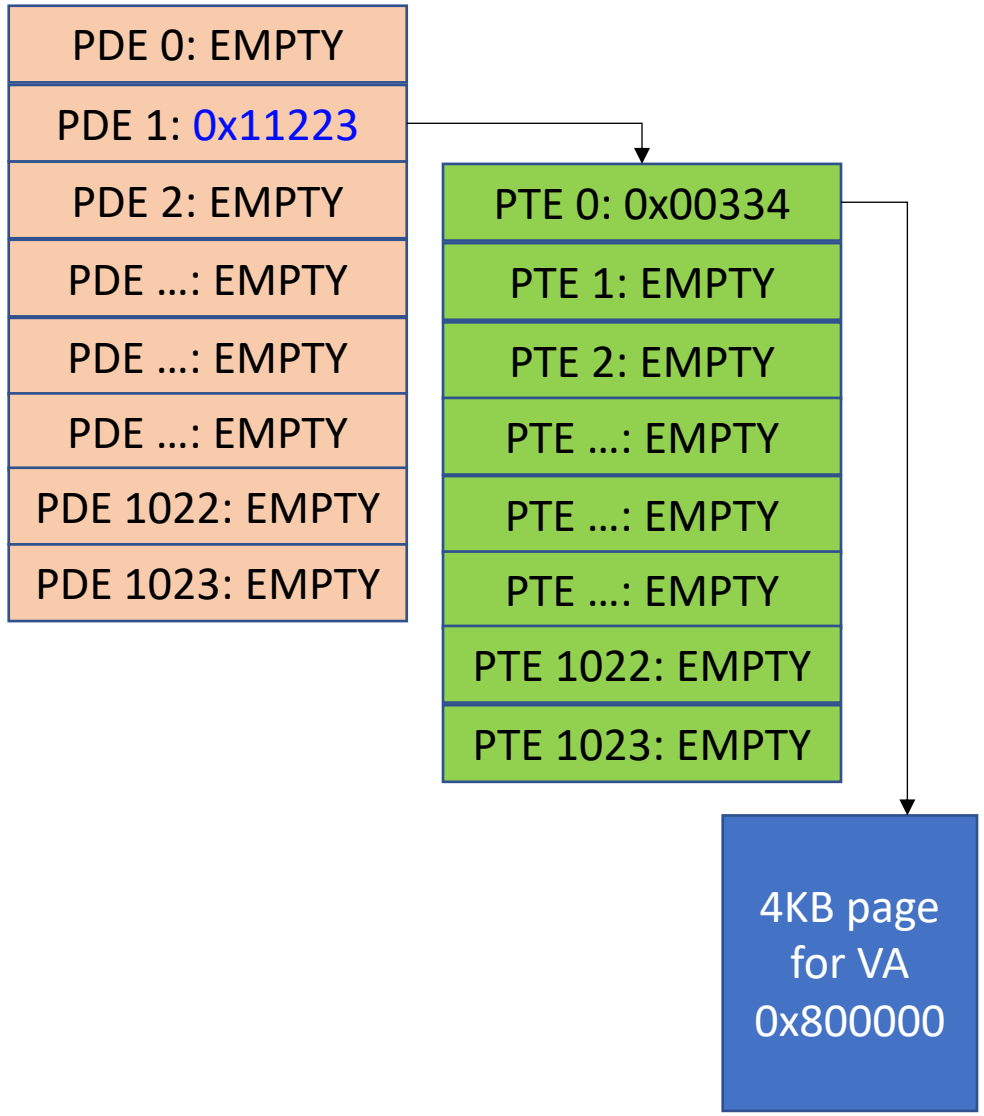
Assigning VA -> PA mapping

- Say, a process wants to access a virtual address
  - E.g., 0x800000
- Allocation procedure
  1. **Check page directory entry** (PDE)
  2. If not set with **PTE_P**
  3. **allocate a physical page** for a new page table



Assigning VA -> PA mapping

- Say, a process wants to access a virtual address
  - E.g., 0x800000
- Allocation procedure
  1. **Check page directory entry** (PDE)
  2. If not set with **PTE_P**
  3. **allocate a physical page** for a new page table
  4. **Check page table entry** (PTE)
  5. If not set with **PTE_P**
  6. **allocate a physical page** to enable access



Assigning VA -> PA mapping

- Say, a process wants to access a virtual address
  - E.g., 0x800000
- Allocation procedure
  1. **Check page directory entry** (PDE)
  2. If not set with **PTE_P**
  3. **allocate a physical page** for a new page table
  4. **Check page table entry** (PTE)
  5. If not set with **PTE_P**
  6. **allocate a physical page** to enable access

We need to
track the free pages!

How does JOS manage Physical Memory?

struct PageInfo

A metadata type that counts number of 'references' to a physical page

pp_ref

If page not in use, **pp_ref = 0**

struct PageInfo* page_free_list

A linked list that tracks **free** physical pages

Create **struct PageInfo** for each physical page

Create a **linked list of free pages**

Struct PageInfo *pages in JOS

A **one-to-one** mapping from a **struct PageInfo** to a physical page

```
struct PageInfo {
    // Next page on the free list.
    struct PageInfo *pp_link;

    // pp_ref is the count of pointers (usually in page table entries)
    // to this page, for pages allocated using page_alloc.
    // Pages allocated at boot time using pmap.c's
    // boot_alloc do not have valid reference count fields.

    uint16_t pp_ref;
};
```

Struct PageInfo *pages in JOS

A **one-to-one** mapping from a **struct PageInfo** to a physical page

```
struct PageInfo {
    // Next page on the free list.
    struct PageInfo *pp_link;

    // pp_ref is the count of pointers (usually in page table entries)
    // to this page, for pages allocated using page_alloc.
    // Pages allocated at boot time using pmap.c's
    // boot_alloc do not have valid reference count fields.

    uint16_t pp_ref;
};
```

8 byte struct per each physical memory page
For 128MB memory,
total physical pages: 128 * 1048576 / 4096 = **32768**
Total size of page structs = **32768 * 8** = 262,144 = **256 KB**

Struct PageInfo *pages in JOS

A **one-to-one** mapping from a **struct PageInfo** to a physical page

```
struct PageInfo {
    // Next page on the free list.
    struct PageInfo *pp_link;

    // pp_ref is the count of pointers (usually in page table entries)
    // to this page, for pages allocated using page_alloc.
    // Pages allocated at boot time using pmap.c's
    // boot_alloc do not have valid reference count fields.

    uint16_t pp_ref;
};
```

A linked list to track all the free pages

Struct PageInfo *pages in JOS

A **one-to-one** mapping from a **struct PageInfo** to a physical page

```
struct PageInfo {
    // Next page on the free list.
    struct PageInfo *pp_link;

    // pp_ref is the count of pointers (usually in page table entries)
    // to this page, for pages allocated using page_alloc.
    // Pages allocated at boot time using pmap.c's
    // boot_alloc do not have valid reference count fields.

    uint16_t pp_ref;
};
```

pp_ref

Count references

Non-zero – in-use

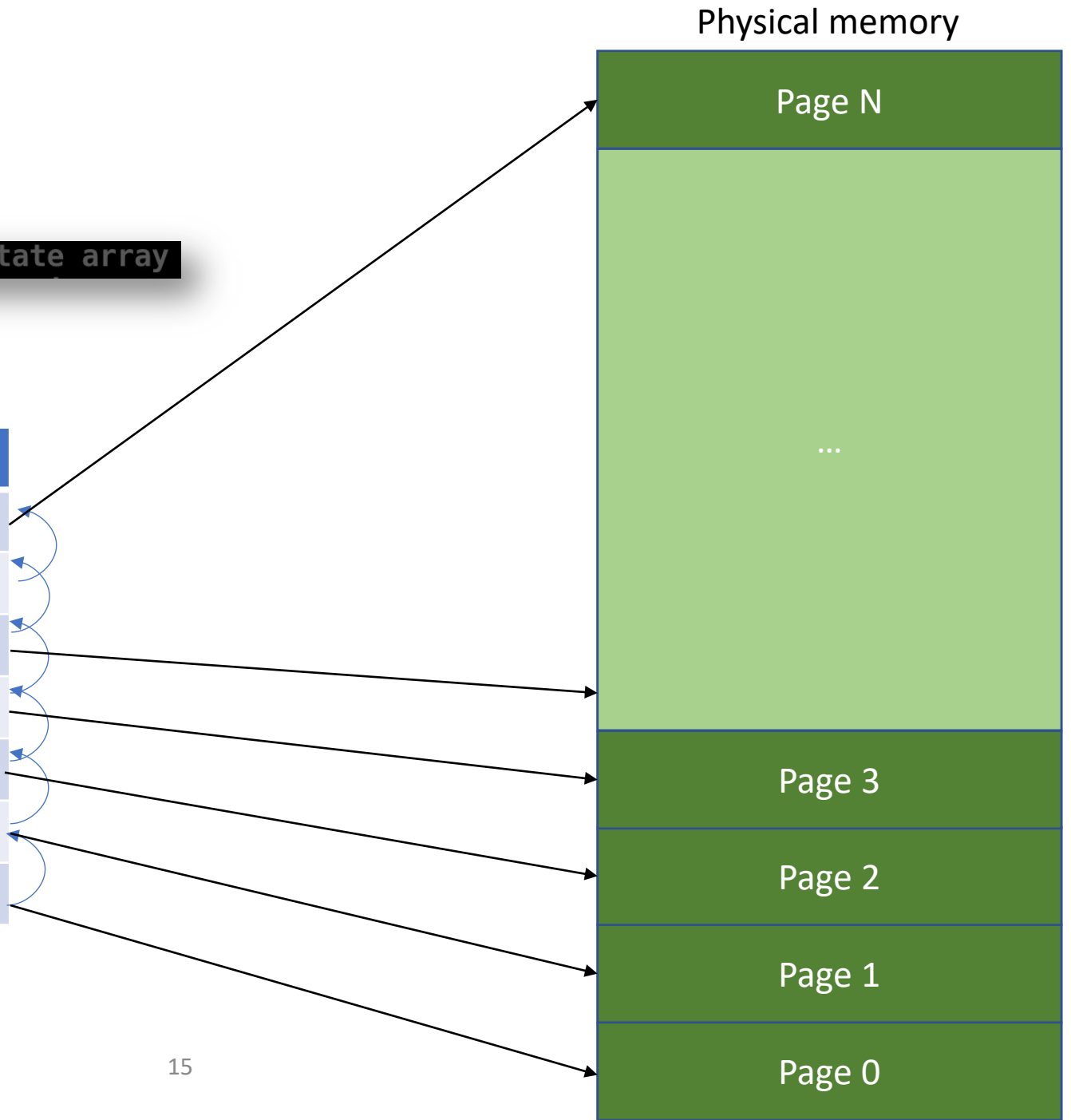
Zero – free

Example

```
struct PageInfo *pages;     // Physical page state array
```

Struct PageInfo * pages (array)

idx	pp_ref	pp_link
N	0	
…	0	
…	0	
3	0	
2	0	
1	0	
0	0	



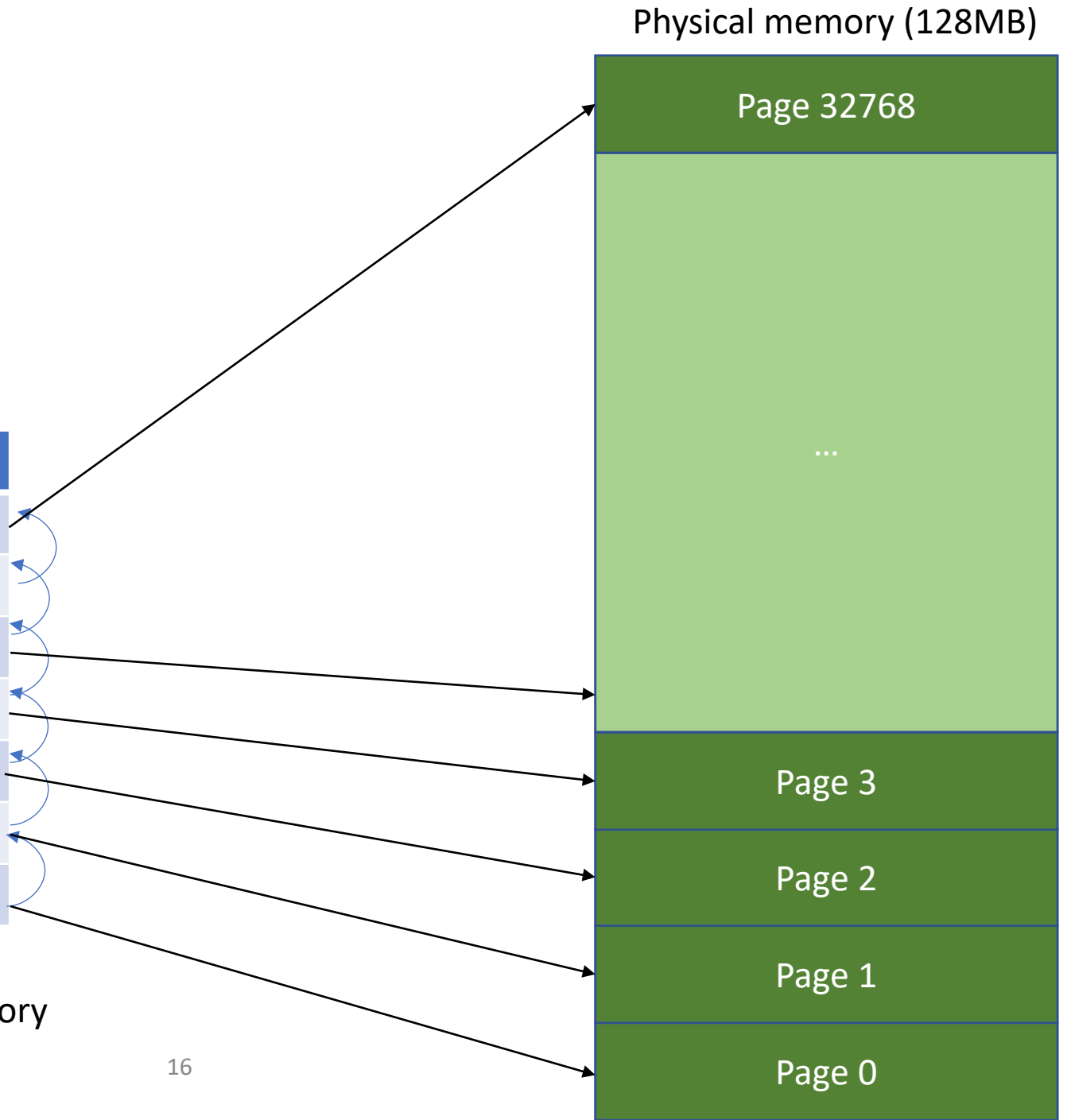
Example

128 * 1048576 / 4096 = 32768 Pages

8 byte per each entry = 32K * 8 = 256KB

Struct PageInfo * pages (array)

idx	pp_ref	pp_link
32K	0	
…	0	
…	0	
3	0	
2	0	
1	0	
0	0	



We can put this array into our physical memory

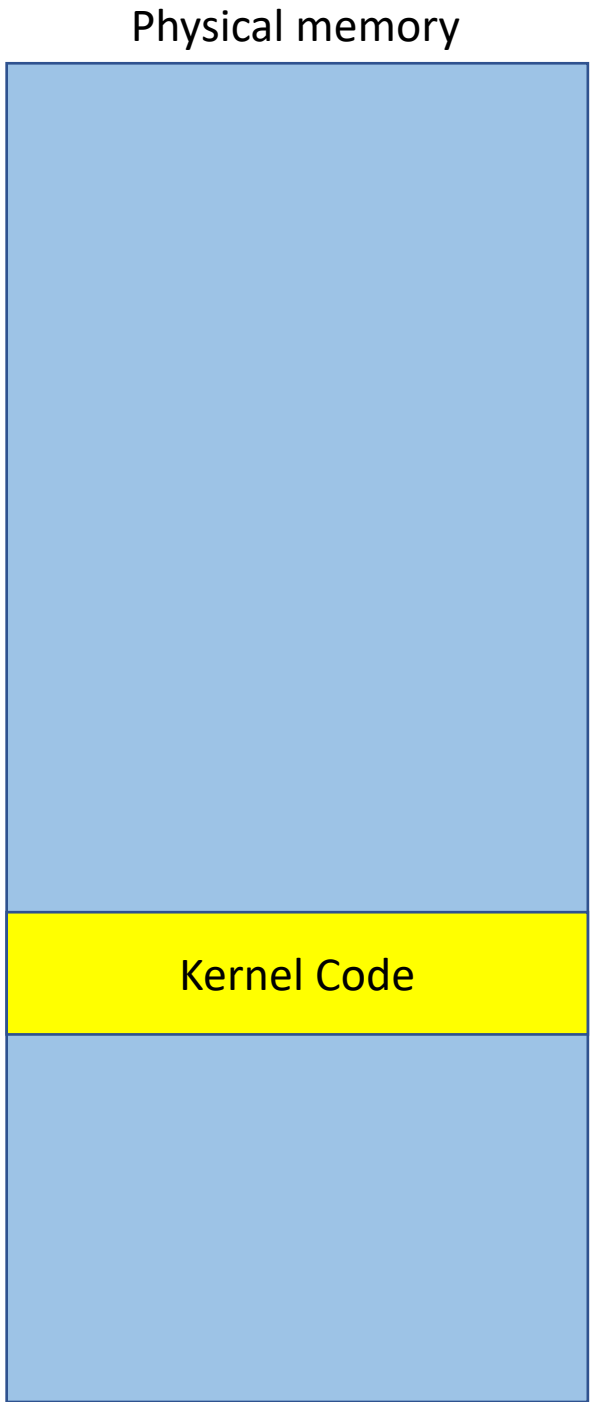
Free Physical Memory (init)

In kern/pmap.c, boot_alloc

```
static void *
boot_alloc(uint32_t n)
{
    static char *nextfree;  // virtual address of next byte of free memory
    char *result;

    // Initialize nextfree if this is the first time.
    // 'end' is a magic symbol automatically generated by the linker,
    // which points to the end of the kernel's bss segment:
    // the first virtual address that the linker did *not* assign
    // to any kernel code or global variables.
    if (!nextfree) {
        extern char end[];
        nextfree = ROUNDUP((char *) end, PGSIZE);
    }
```

nextfree → points to the end of the kernel code/data



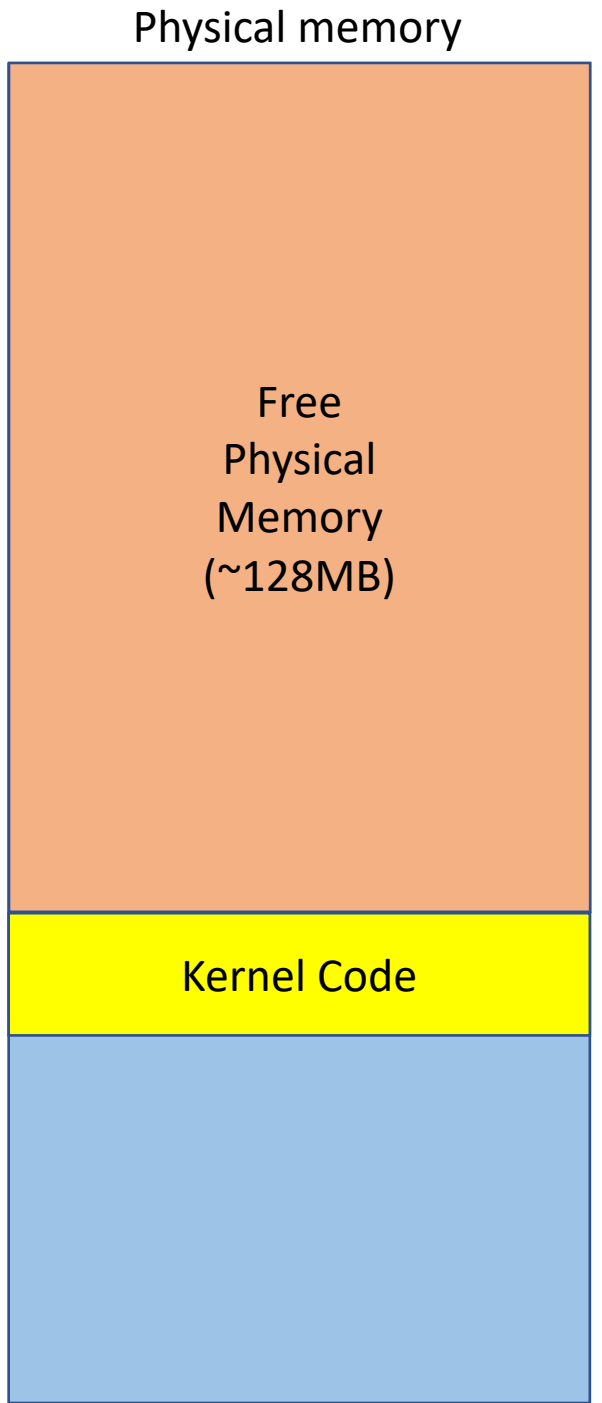
Free Physical Memory (init)

In kern/pmap.c, **`boot_alloc()`**

```
static void *
boot_alloc(uint32_t n)
{
    static char *nextfree;  // virtual address of next byte of free memory
    char *result;

    // Initialize nextfree if this is the first time.
    // 'end' is a magic symbol automatically generated by the linker,
    // which points to the end of the kernel's bss segment:
    // the first virtual address that the linker did *not* assign
    // to any kernel code or global variables.
    if (!nextfree) {
        extern char end[];
        nextfree = ROUNDUP((char *) end, PGSIZE);
    }
```

nextfree → points to the end of the kernel code/data



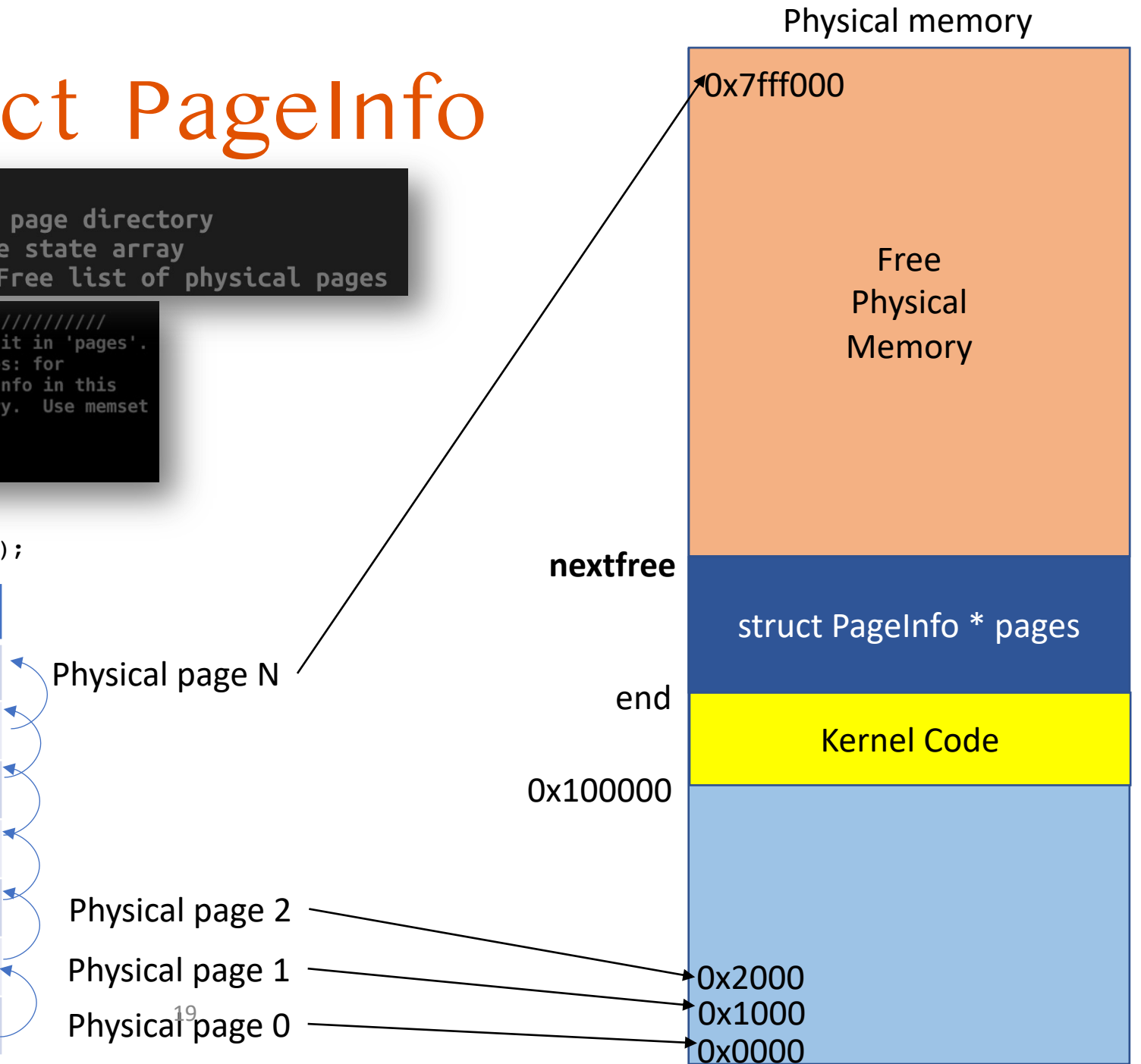
Allocating struct PageInfo

```
// These variables are set in mem_init()
pde_t *kern_pgdir;      // Kernel's initial page directory
struct PageInfo *pages;     // Physical page state array
static struct PageInfo *page_free_list; // Free list of physical pages
```

```
////////////////////////////////////////////////////////////////////
// Allocate an array of npages 'struct PageInfo's and store it in 'pages'.
// The kernel uses this array to keep track of physical pages: for
// each physical page, there is a corresponding struct PageInfo in this
// array.  'npages' is the number of physical pages in memory.  Use memset
// to initialize all fields of each struct PageInfo to 0.
// Your code goes here:
```

```
pages =
boot_alloc(npages * sizeof(struct PageInfo));
```

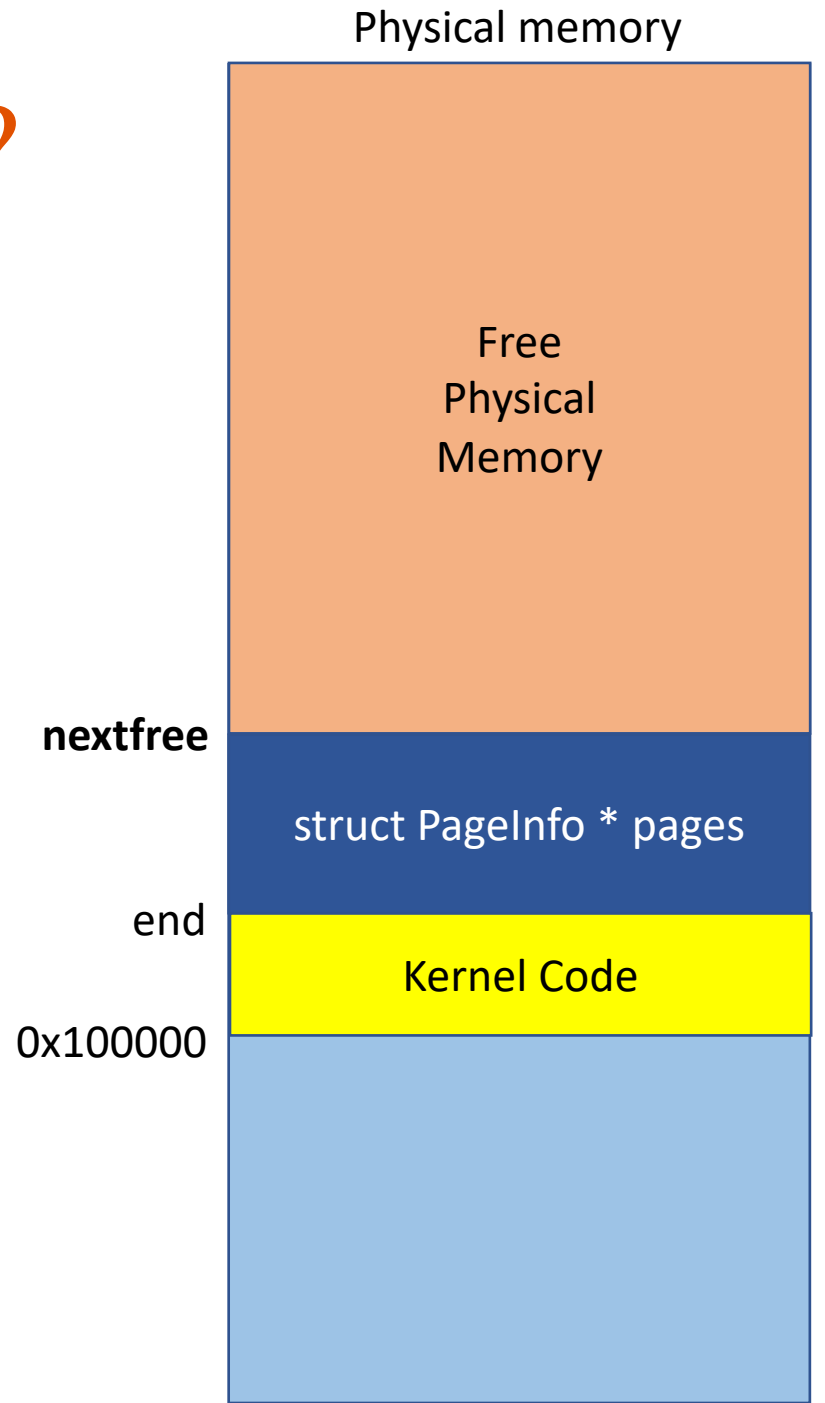
idx	pp_ref	pp_link
N	0	
…	0	
…	0	
3	0	
2	0	
1	0	
0	0	



Where are the free pages?

- in page_init()

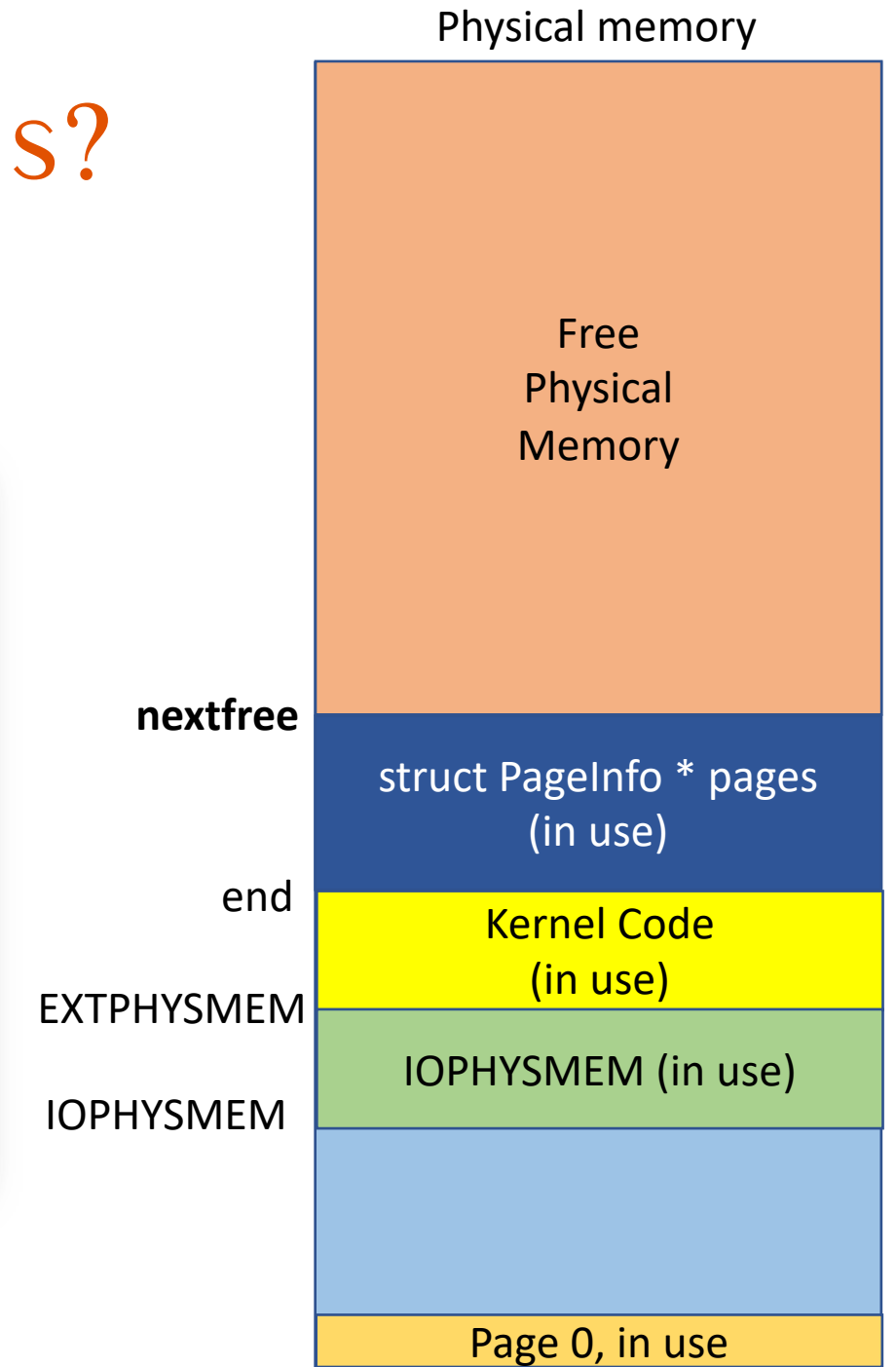
```
// The example code here marks all physical pages as free.
// However this is not truly the case.  What memory is free?
//  1) Mark physical page 0 as in use.
//     This way we preserve the real-mode IDT and BIOS structures
//     in case we ever need them.  (Currently we don't, but...)
//  2) The rest of base memory, [PGSIZE, npages_basemem * PGSIZE)
//     is free.
//  3) Then comes the IO hole [IOPHYSMEM, EXTPHYSMEM), which must
//     never be allocated.
//  4) Then extended memory [EXTPHYSMEM, ...).
//     Some of it is in use, some is free. Where is the kernel
//     in physical memory?  Which pages are already in use for
//     page tables and other data structures?
//
// Change the code to reflect this.
// NB: DO NOT actually touch the physical memory corresponding to
// free pages!
```



Where are the free pages?

- in page_init()

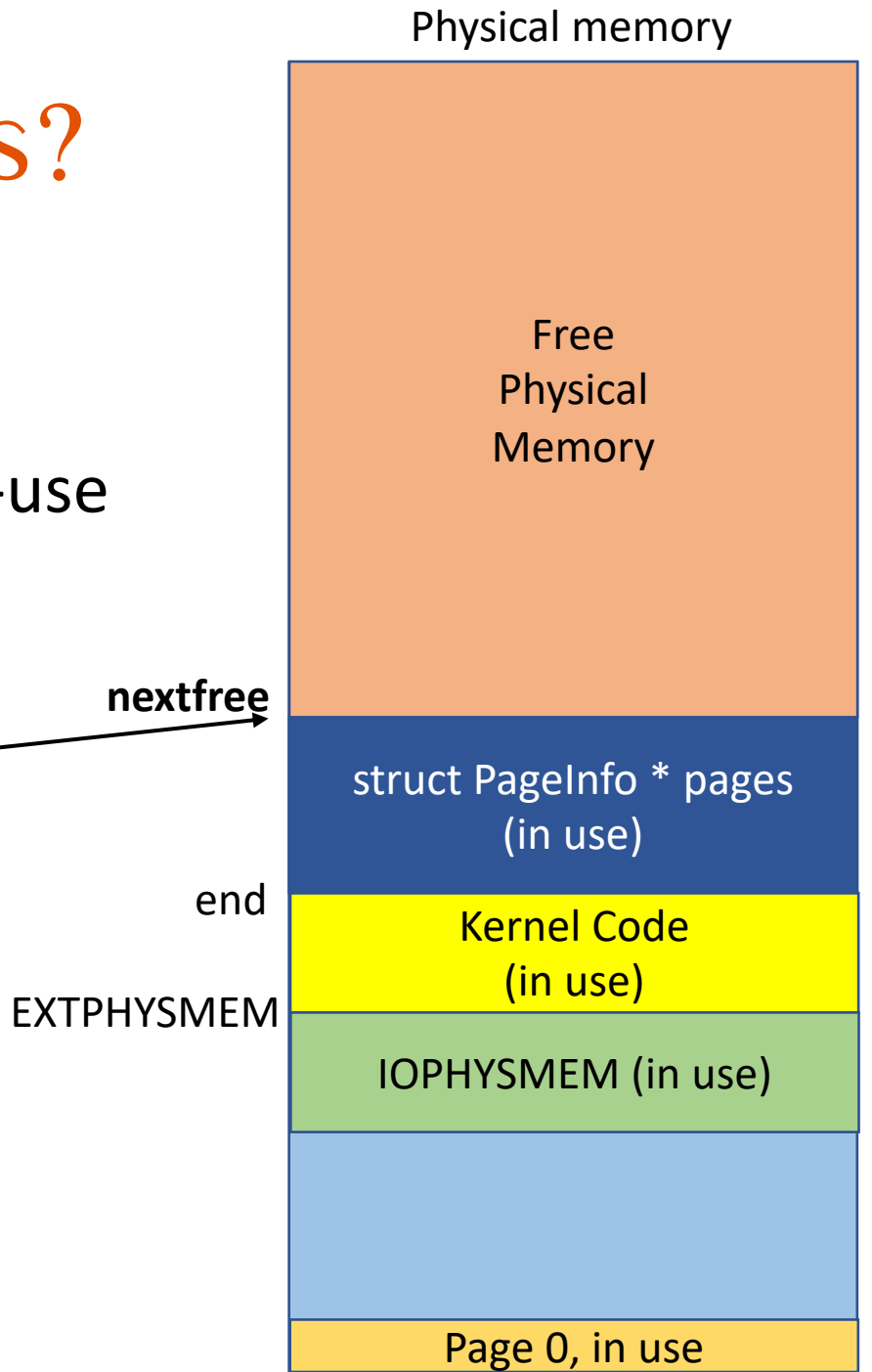
```
// The example code here marks all physical pages as free.
// However this is not truly the case.  What memory is free?
//  1) Mark physical page 0 as in use.
//     This way we preserve the real-mode IDT and BIOS structures
//     in case we ever need them.  (Currently we don't, but...)
//  2) The rest of base memory, [PGSIZE, npages_basemem * PGSIZE)
//     is free.
//  3) Then comes the IO hole [IOPHYSMEM, EXTPHYSMEM), which must
//     never be allocated.
//  4) Then extended memory [EXTPHYSMEM, ...).
//     Some of it is in use, some is free. Where is the kernel
//     in physical memory?  Which pages are already in use for
//     page tables and other data structures?
//
// Change the code to reflect this.
// NB: DO NOT actually touch the physical memory corresponding to
// free pages!
```



Where are the free pages?

- Page 0 is in-use
- Pages in [IOPHYSMEM ~ EXTPHYSMEM] are in-use
- Pages for the kernel code are in-use
- Pages for struct PageInfo *pages are in-use
- How can you point to this?
  - pages + npages ?
  - `boot_alloc(0)`?

`boot_alloc(0)` is better

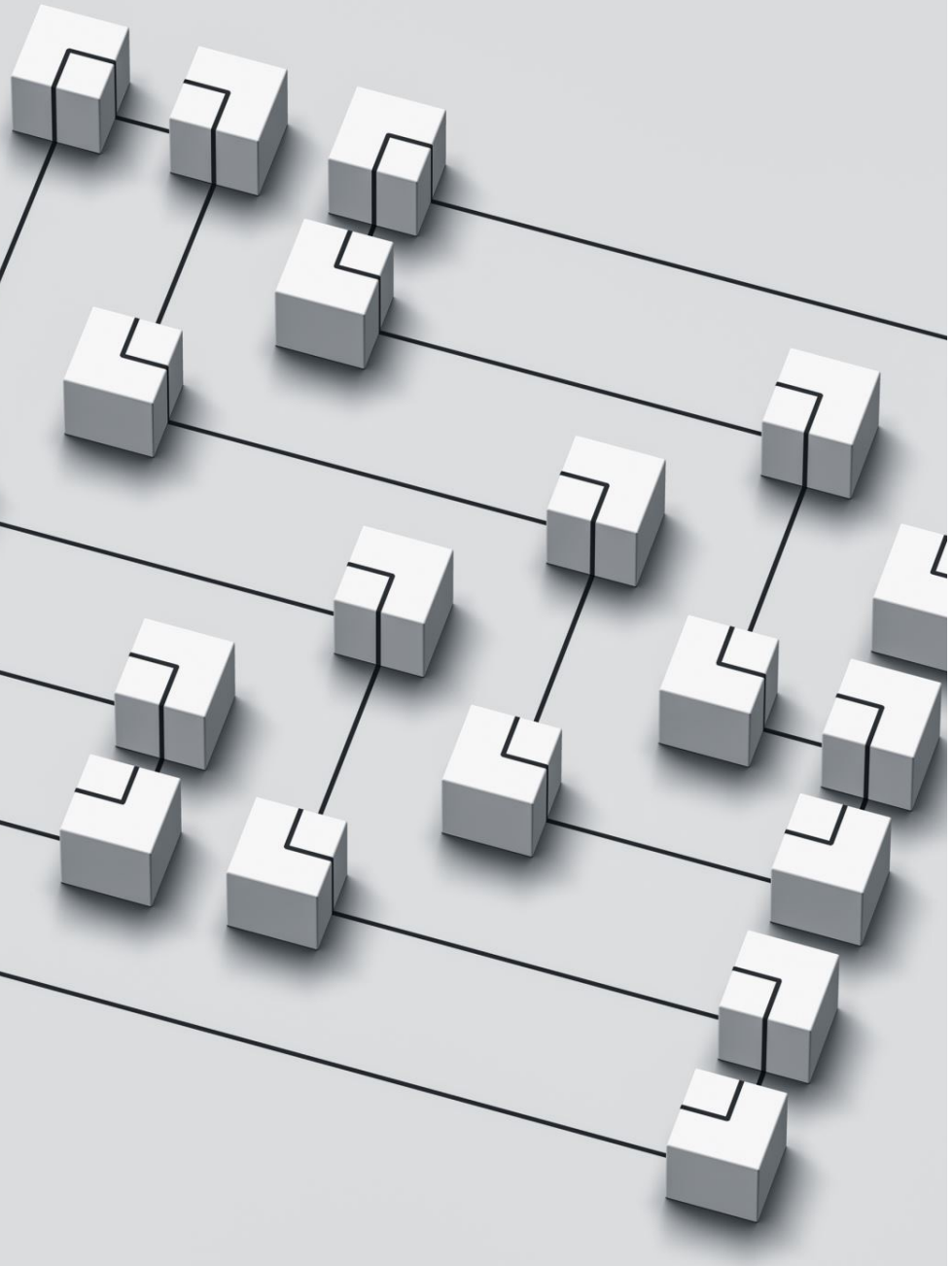


Caveat

- Some pages are mapped but does not have to be marked as in-use
- Make sure you do not count up pages for dirmap
  - **`0xf0000000 ~ 0xffffffff`**

- Read the comment at the top of **`boot_map_region`** thoroughly

```
// This function is only intended to set up the ``static'' mappings
// above UTOP. As such, it should *not* change the pp_ref field on the
// mapped pages.
//
// Hint: the TA solution uses pgdir_walk
static void
boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa, int perm)
```



Reference Counting

- A typical mechanism for tracking objects in programs
  - E.g., how many pointers are referencing the same object?
- Can use it for tracking free/used pages
- Mechanism
  - Count up the value (**pp_ref++**) if the page is referenced by others (in use!)
  - Count down the value (**pp_ref--**) if not used for one of usages anymore
  - A page is free if **pp_ref == 0**
- In C++, **`shared_ptr<T>`**
  - When a pointer is assigned to a variable, count up!
  - When the variable no longer uses the variable, count down!
  - Free the memory when the count become 0

Ref. Counting with struct PageInfo

- For in-use memory page
  - Set **pp_ref = 1**
- For memory page that's not-in-use
  - Invariant: **pp_ref == 0**
  - **Must be linked with pages_free_list**
- When **assigning** the page to a virtual address
  - **pp_ref++**
- When **releasing** the page from a virtual address
  - **pp_ref--**

Linked-list for Free Pages

- Start with NULL at the head
  - **`page_free_list = NULL;`**
- After setting **`pp_ref`** of all pages

```
for (int i=0; i < npages; ++i) {
    if (pages[i].pp_ref == 0) {
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}
```

This will build a linked list of free pages

List Building

```
for (int i=0; i < npages; ++i) {
    if (pages[i].pp_ref == 0) {
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}
```

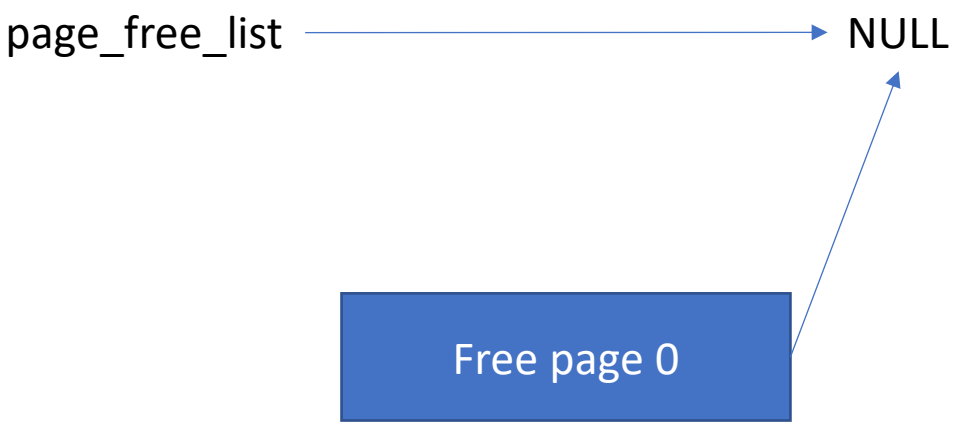
List Building

```
for (int i=0; i < npages; ++i) {
    if (pages[i].pp_ref == 0) {
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}
```

Free page 0

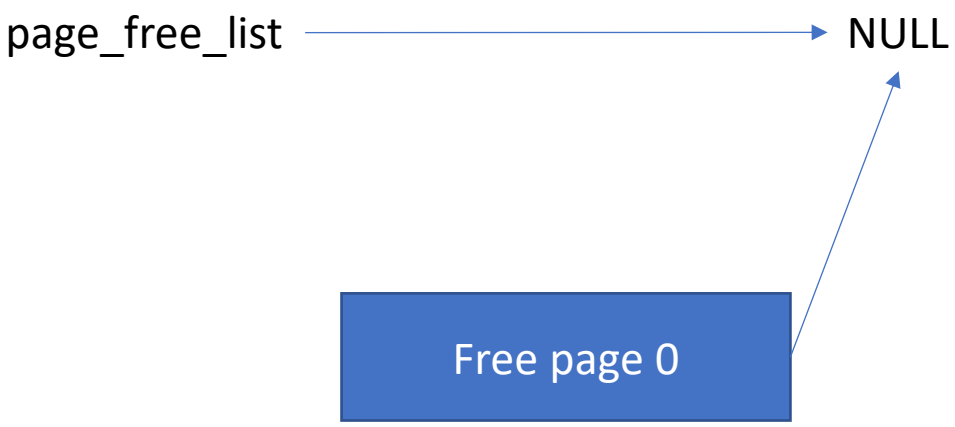
List Building

```
for (int i=0; i < npages; ++i) {
    if (pages[i].pp_ref == 0) {
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}
```



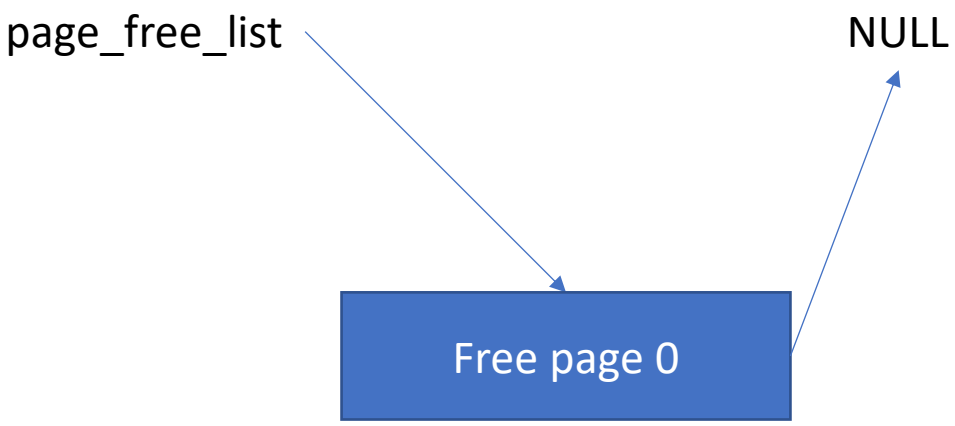
List Building

```
for (int i=0; i < npages; ++i) {
    if (pages[i].pp_ref == 0) {
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}
```



List Building

```
for (int i=0; i < npages; ++i) {
    if (pages[i].pp_ref == 0) {
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}
```



List Building

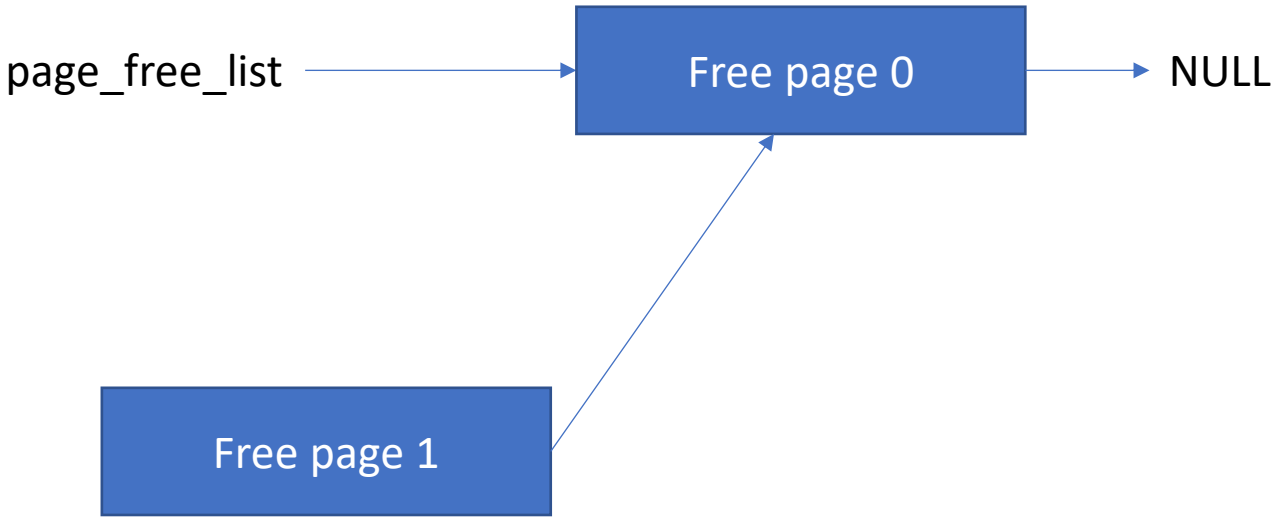
```
for (int i=0; i < npages; ++i) {
    if (pages[i].pp_ref == 0) {
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}
```

List Building

```
for (int i=0; i < npages; ++i) {
    if (pages[i].pp_ref == 0) {
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}
```

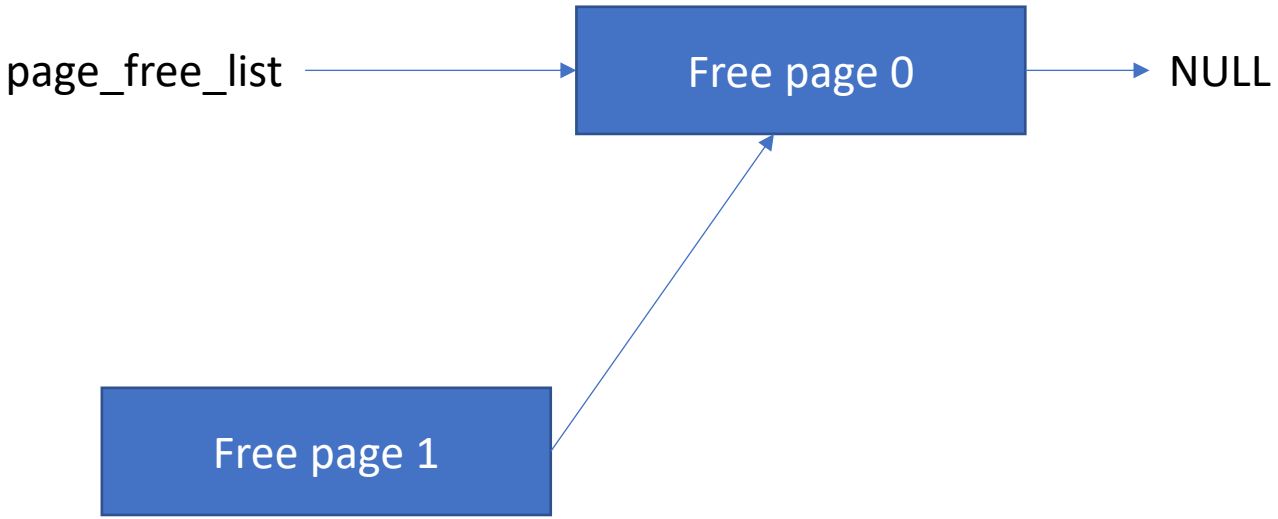
List Building

```
for (int i=0; i < npages; ++i) {
    if (pages[i].pp_ref == 0) {
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}
```



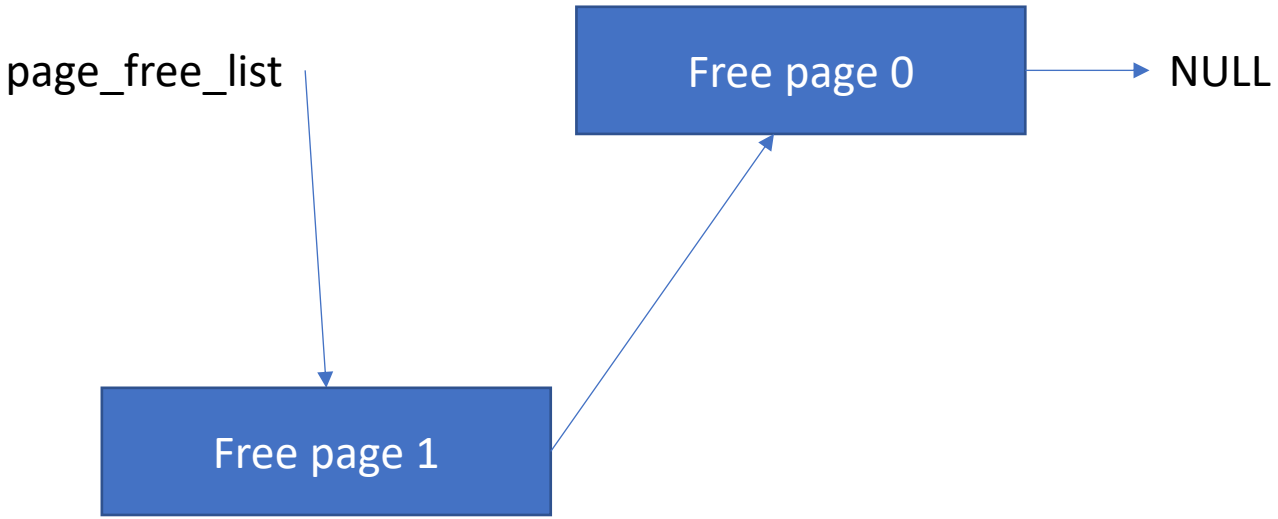
List Building

```
for (int i=0; i < npages; ++i) {
    if (pages[i].pp_ref == 0) {
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}
```



List Building

```
for (int i=0; i < npages; ++i) {
    if (pages[i].pp_ref == 0) {
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}
```



List Building

```
for (int i=0; i < npages; ++i) {
    if (pages[i].pp_ref == 0) {
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}
```

List Building

```
for (int i=0; i < npages; ++i) {
    if (pages[i].pp_ref == 0) {
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}
```

page2pa(struct PageInfo *pp)

- Changes a pointer to **struct PageInfo** to a physical address
- **`idx = (pp – pages)`**
  - Gets the index of **`pp`** in pages
  - E.g., **`&pages[idx] == pp`**
- idx here is a physical page number

```
static inline physaddr_t
page2pa(struct PageInfo *pp)
{
    return (pp - pages) << PGSHIFT;
}
```

pp →

idx	pp_ref	pp_link
4	0	
3	0	
2	0	
1	0	
0	0	

pages →

pp – pages = 4
0x4000 ← physical page address!

pa2page(physaddr_t pa)

- `PGNUM(pa)`
  - Returns page number

```
static inline struct PageInfo*
pa2page(physaddr_t pa)
{
    if (PGNUM(pa) >= npages)
        panic("pa2page called with invalid pa");
    return &pages[PGNUM(pa)];
}
```

- `&pages[PGNUM(pa)]`
  - Returns `struct PageInfo *` of that pa..

idx	pp_ref	pp_link
4	0	
3	0	
2	0	
1	0	
0	0	

Quiz 1 Details

Quiz 1 (04/14)

- Released via CANVAS
  - You can see Quiz 1 at **8:30 am** on **04/14** [**Thursday**]
  - Deadline: **04/15** [**Friday**] **11:59pm**
  - Duration: **unlimited**, but you can finish it around **30 min**
- You will be given up to **2 attempts** to take quiz
- **Open material** → you may refer to
  - Contents at our course website
  - Slides
  - Lab Tutorials
  - Your code for Lab 1 / Lab 2
  - Textbook (not required)

DO NOT DISCUSS WITH ANYONE!

- Not other students
- Not Tas
- Not instructor
- Not your friends online
- Your friendly neighborhood Spider-man
- **NO ONE!**

Quiz 1 (10/14)

- Question type: **multiple choice**, less than 15 questions
  - 1 point per each question
- All content taught so far will be covered in the Quiz 1, viz.,
  - BIOS/Booting/CPU, Real mode segmentation
  - Protected mode segmentation and Paging
  - Virtual address translation
  - Virtual memory layout
  - JOS Memory management
  - JOS Lab Assignment 1 [Lab Tutorials 1 & 2]
  - First part of JOS Lab Assignment 2 [Lab Tutorial 3]

Prep for Quiz 1

- Which one of the following is not a job that JOS Bootloader does?
  - A. Enable protected mode
  - B. Enable paging
  - C. Load kernel image from disk
  - D. Enable A20

Prep for Quiz 1

- Which one of the following is not a job that JOS Bootloader does?
  - A. Enable protected mode
  - **B. Enable paging (is done in kenrel, in kern/entry.S)**
  - C. Load kernel image from disk
  - D. Enable A20

Prep for Quiz 1

- In the x86 **real** mode, what address is the following segment:offset pair?
- 0x8000:0x3131
  - A. 0xb131
  - B. 0x3131
  - C. 0x83131 (0x8000 * 16 + 0x3131 = 0x80000 + 0x3131 = 0x83131)
  - D. 0x103131
  - E. 0x11131

Prep for Quiz 1

- In the x86 **real** mode, what address is the following segment:offset pair?
- 0x8000:0x3131
  - A. 0xb131
  - B. 0x3131
  - **C. 0x83131 (0x8000 * 16 + 0x3131 = 0x80000 + 0x3131 = 0x83131)**
  - D. 0x103131
  - E. 0x11131

Prep for Quiz 1

- Which of the following x86 registers stores the current privilege level?
  - A. ds
  - B. eip
  - C. ebp
  - D. esp
  - E. cs

Prep for Quiz 1

- Which of the following x86 register stores the end of the current stack frame (and moves if the CPU runs push/pop) ?
  - A. ds
  - B. eip
  - C. ebp
  - D. esp
  - E. cs

Prep for Quiz 1

- Which of the following x86 register stores the start of the current stack frame (also points to the address that stores previous frame's stack base pointer) ?
  - A. ds
  - B. eip
  - C. ebp
  - D. esp
  - E. cs

Prep for Quiz 1

- What are the benefits of enabling virtual memory?
- Choose all (no partial credits)
  - A. Performs faster execution than when using physical memory
  - B. Suffers less memory fragmentation than when using physical memory
  - C. Provides a better isolation / protection than when using physical memory
  - D. Provides memory transparency
  - E. Enables virtual reality

Quiz 1 (04/14)

- Released via CANVAS
  - You can see Quiz 1 at **8:30 am** on **04/14** [**Thursday**]
  - Deadline: **04/15** [**Friday**] **11:59pm**
  - Duration: **unlimited**, but you can finish it around **30 min**
- You will be given up to **2 attempts** to take quiz
- **Open material** → you may refer to
  - Contents at our course website
  - Slides
  - Lab Tutorials
  - Your code for Lab 1 / Lab 2
  - Textbook (not required)

DO NOT DISCUSS WITH ANYONE!

- Not other students
- Not Tas
- Not instructor
- Not your friends online
- Your friendly neighborhood Spider-man
- **NO ONE!**

Additional Slides

Recap: PDE/PTE Permission Examples

- Virtual address 0x01020304
- PDE: PTE_P | PTE_W
- PTE: PTE_P | PTE_U
- valid, inaccessible by ring3, not writable

- PTE_P (PRESENT)
  - 0: invalid entry
  - 1: valid entry
- PTE_W (WRITABLE)
  - 0: read only
  - 1: writable
- PTE_U (USER)
  - 0: kernel (only ring 0 can access)
  - 1: user (accessible by ring 3)

Recap: PDE/PTE Permissions CAVEAT

- A virtual address access is allowed if both **PDE** and **PTE** entries allows the access…
- General practice: put **a more permissive** permission bits in **PDE**, and **be strict** on setting permission bits in **PTE**
- For a conflicting permission setup for Kernel/User, add **an additional virtual address mapping** can enable such a setup

Recap: You can setup the following page permissions...

- Kernel: RW, User: R
  - VA 0x00001000 -> PA 0x50000, PTE_P | PTE_U (User R)
  - VA 0xf0050000 -> PA 0x50000, PTE_P | PTE_W (Kernel RW)
- Kernel: R, User: RW
  - VA 0x00002000 -> PA 0x60000, PTE_P | PTE_U | PTE_W (User RW)
  - VA 0xf0060000 -> PA 0x60000, PTE_P (Kernel R)

- Kernel: --, User: RW
  - VA 0x00003000 -> PA 0x70000, PTE_P | PTE_U | PTE_ W
  - VA 0xf0070000 -> PA 0x70000, 0 for flag…