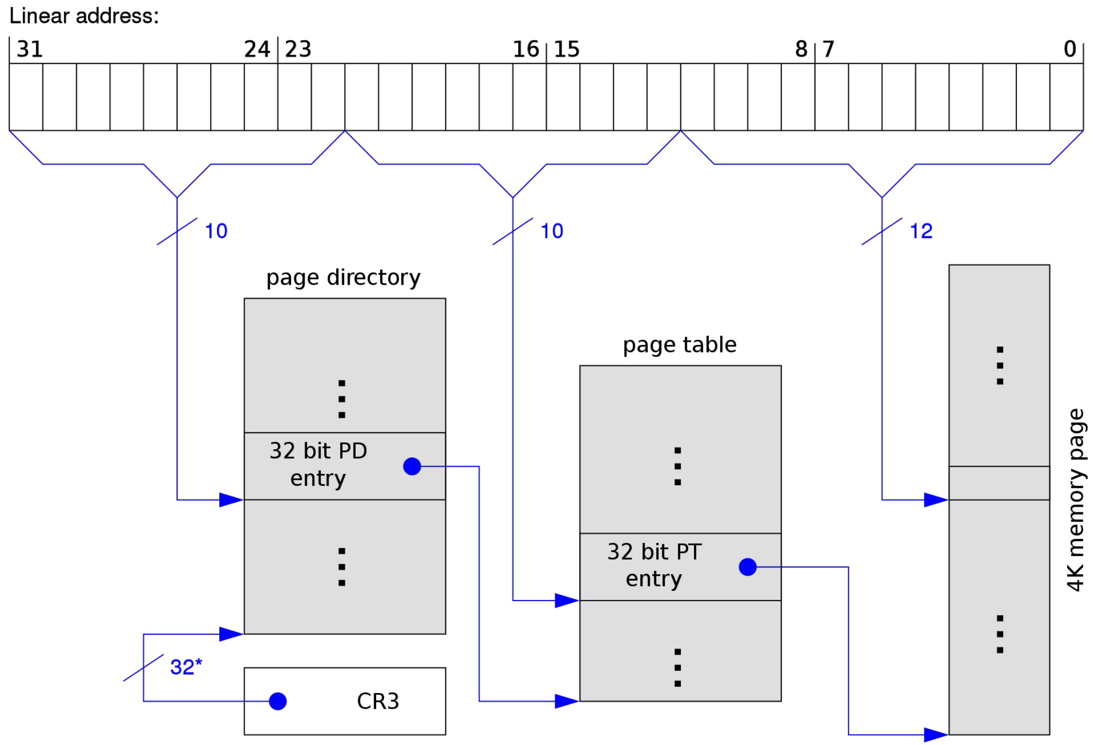
JOS memory management

ECE 469, Jan 27

Aravind Machiry

Recap: Two-level paging (32-bit)

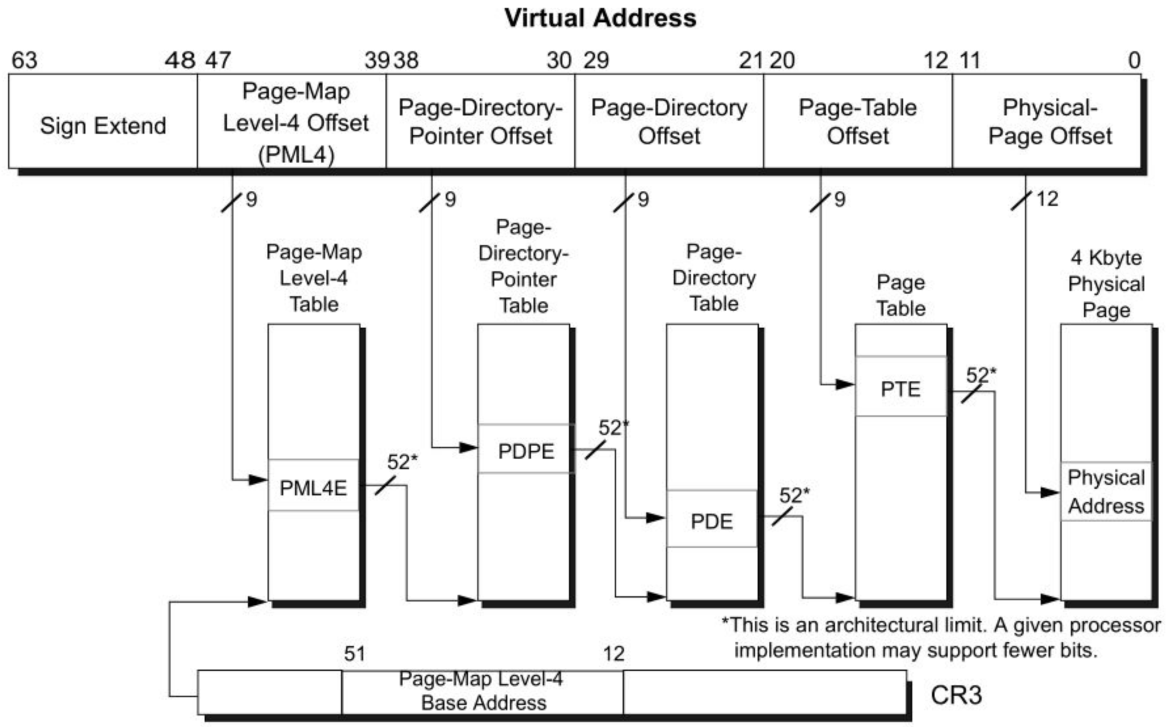


*) 32 bits aligned to a 4-KByte boundary

Recap: x86_64: 48-bit address space

- Initial amd64 processors use only 48-bit virtual address space
  - 2^48 = 256 TB
- Each level of table tree can process 9 bits (512 entries in table)
- We ignore lower 12 bits
  - 48 – 12 = 36 (total 2^36 pages)
  - 36 / 9 = 4   (each table can process 9 bits of address space)
- **We use 4-level page table**

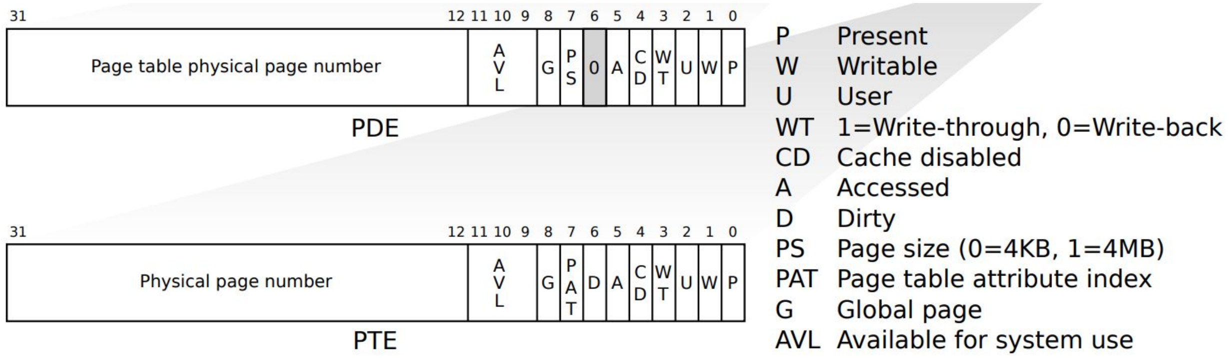
Recap: Four-level paging (64-bit)



Recap: Page Directory / Table Entry (PDE/PTE)

- Top 20 bits: physical page number
  - Physical page number of a page table (PDE)
  - Physical page number of the requested virtual address (PTE)

- Lower 12 bits: some flags
  - Permission
  - Etc.

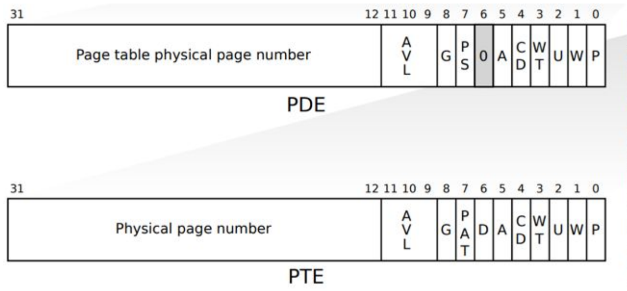


P	Present
W	Writable
U	User
WT	1=Write-through, 0=Write-back
CD	Cache disabled
A	Accessed
D	Dirty
PS	Page size (0=4KB, 1=4MB)
PAT	Page table attribute index
G	Global page
AVL	Available for system use

Recap: Permission Flags

- 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)

	Page Table Entry	
0	Addr PT	
0x48	0x10000 << 12 | PTE_U | PTE_W	Invalid
0x49	0x11000 << 12 | PTE_P | PTE_W	Kernel, writable
0x4a	0x50000 << 12 | PTE_P | PTE_U	User, read-only



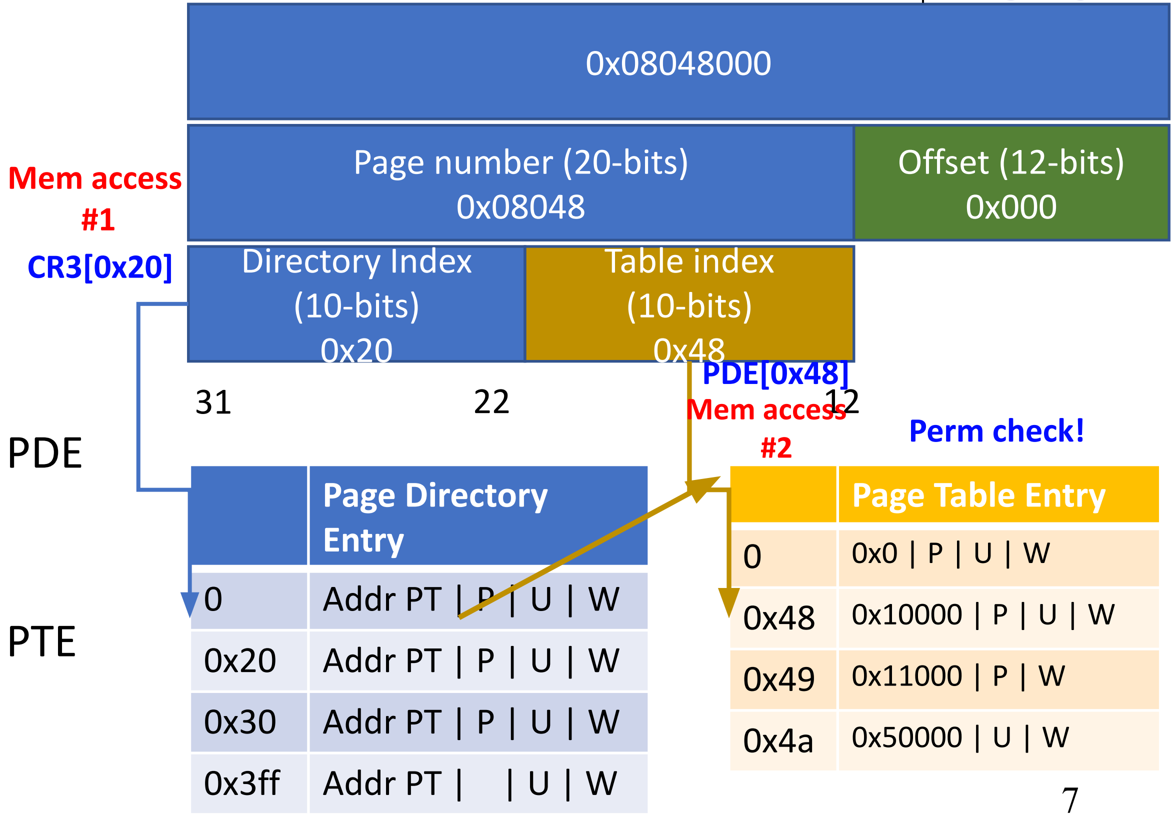
P	Present
W	Writable
U	User
WT	1=Write-through, 0=Write-back
CD	Cache disabled
A	Accessed
D	Dirty
PS	Page size (0=4KB, 1=4MB)
PAT	Page table attribute index
G	Global page
AVL	Available for system use

Recap: When CPU Checks Permission Bits?

- A virtual memory address is inaccessible if PDE disallows the access
- A virtual memory address is inaccessible if PTE disallows the access
- Both **PDE and PTE should allow the access**…

Recap: When CPU Checks Permission Bits?

- In address translation
- 1. Virtual address
- 2. PDE = CR3[PDX]
  - Checks permission bits in PDE
- 3. PTE = PDE[PTX]
  - Checks permission bits in PTE



	Page Directory Entry
0	Addr PT | P | U | W
0x20	Addr PT | P | U | W
0x30	Addr PT | P | U | W
0x3ff	Addr PT | | U | W

	Page Table Entry
0	0x0 | P | U | W
0x48	0x10000 | P | U | W
0x49	0x11000 | P | W
0x4a	0x50000 | U | W

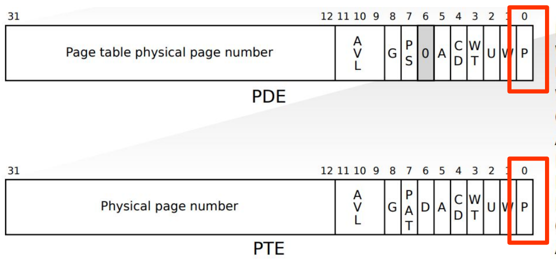
Recap: Allocating Virtual Memory

- Static allocation is inefficient:
  - Why don’t we just allocate entire virtual address space to a process?
  - Inefficient: The process may not access entire virtual address space

- Solution: Dynamic, Request based

What happens when we call malloc?

- Before malloc()?
  - No PTEs



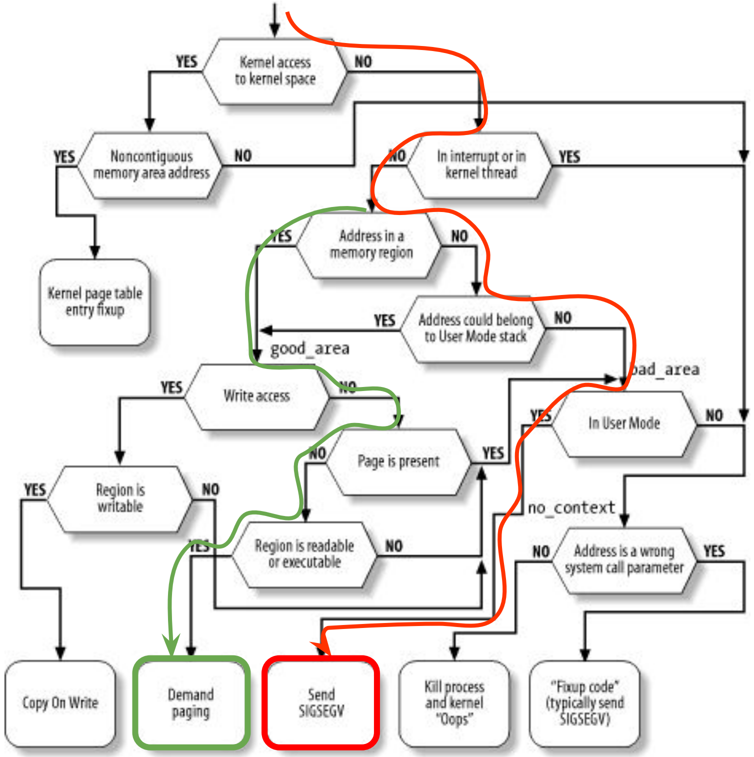
P	Present
W	Writable
U	User
WT	1=Write-through, 0=Write-back
CD	Cache disabled
A	Accessed
D	Dirty
PS	Page size (0=4KB, 1=4MB)
PAT	Page table attribute index
G	Global page
AVL	Available for system use

- After malloc()?
  - **PDE/PTE updated but present bit not-set**

- Upon first access?
  - Assign physical page (and page table) and set the valid bit.

Handling Page faults in Kernel

- Accessing unallocated memory: **Demand Paging**



Kernel access to kernel space
YES
NO
Noncontiguous memory area address
YES
NO
In interrupt or in kernel thread
NO
YES
Kernel page table entry fixup
Address in a memory region
YES
NO
Address could belong to User Mode stack
YES
NO
good_area
bad_area
Write access
YES
NO
In User Mode
YES
NO
Page is present
NO
YES
no_context
Region is writable
YES
NO
Region is readable or executable
YES
NO
Address is a wrong system call parameter
NO
YES
Copy On Write
Demand paging
Send SIGSEGV
Kill process and kernel "Oops"
"Fixup code" (typically send SIGSEGV)

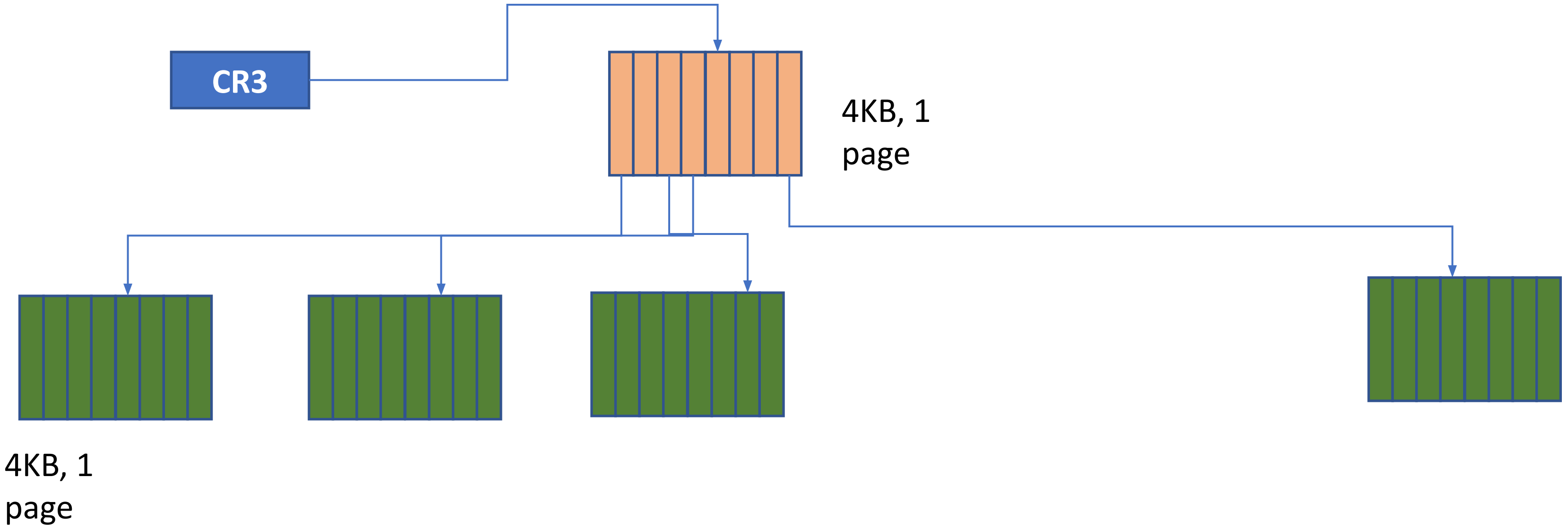
- Accessing Unassigned Virtual Memory: **Segmentation Fault**

JOS Memory Management

- Handling VA <-> PA mapping
- Managing Physical pages

Creating a Virtual Memory Space

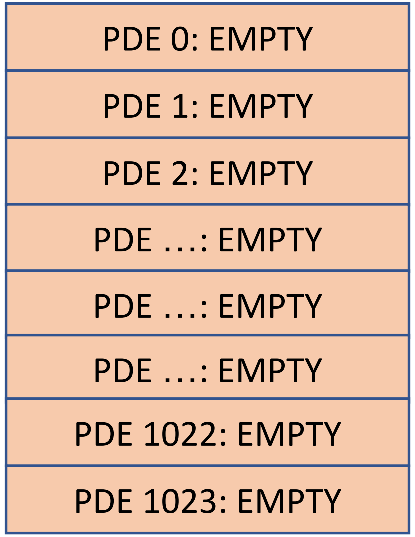
- A page directory manages the entire virtual memory space
  - Of a process
- CR3 points to the Page directory, and each PDE entry points to PTs..



Assigning VA -> PA mapping

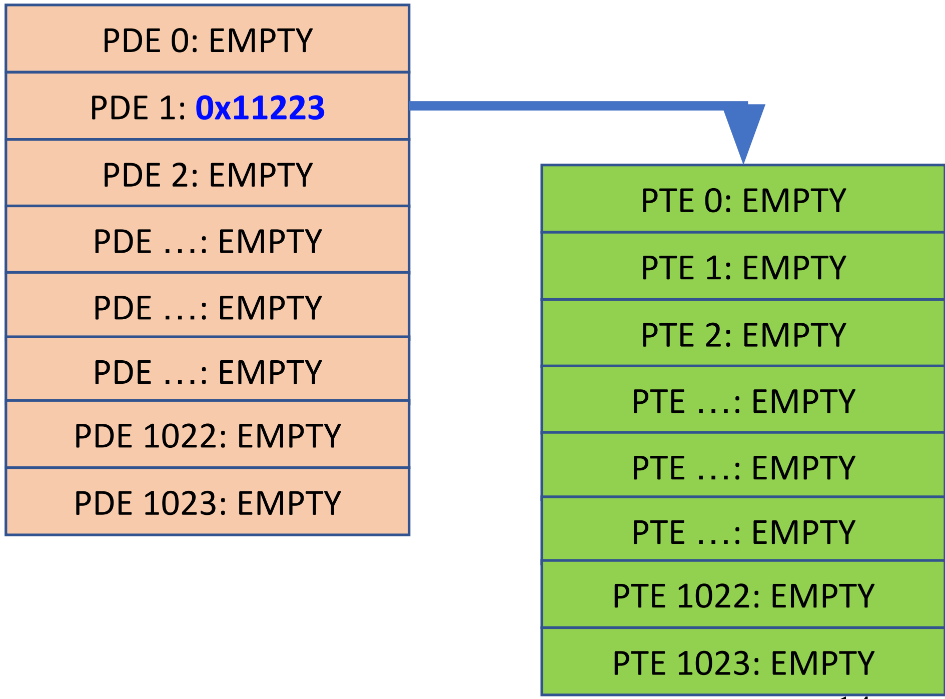
- Suppose a process would like to use a virtual address
  - 0x800000 (RW from user)

- Allocation procedure
  - Check page directory entry (PDE)
    - If not set with PTE_P, **allocate a physical page** for a new page table



Assigning VA -> PA mapping

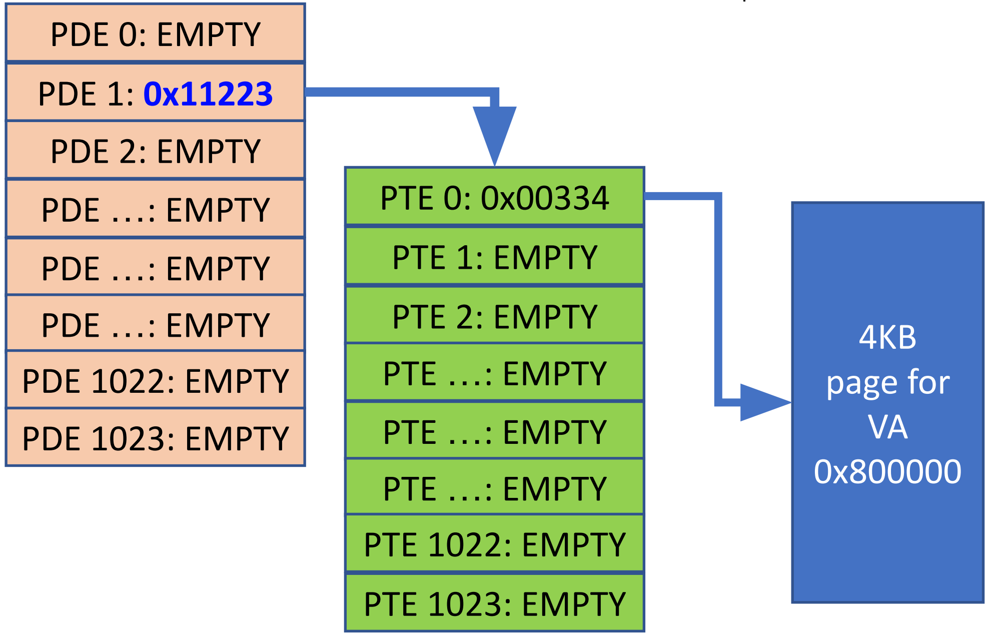
- Suppose a process would like to use a virtual address
  - 0x800000 (RW from user)
- Allocation procedure
  - Check page directory entry (PDE)
    - If not set with PTE_P, **allocate a physical page** for a new page table



Assigning VA -> PA mapping

- Suppose a process would like to use a virtual address
  - 0x800000 (RW from user)

- Allocation procedure
  - Check page directory entry (PDE)
    - If not set with PTE_P, **allocate a physical page** for a new page table
  - Check page table entry (PTE)
    - If not set with PTE_P, **allocate a physical page** to enable access



Assigning VA -> PA mapping

- Suppose a process would like to use a virtual address
  - 0x800000 (RW from user)

- Allocation procedure
  - Check page directory entry (PDE)
    - If not set with PTE_P, **allocate a physical page** for a new page table
  - Check page table entry (PTE)
    - If not set with PTE_P, **allocate a physical page** to enable access

- We need to keep track of '**free**' physical pages…

Managing Physical memory in JOS

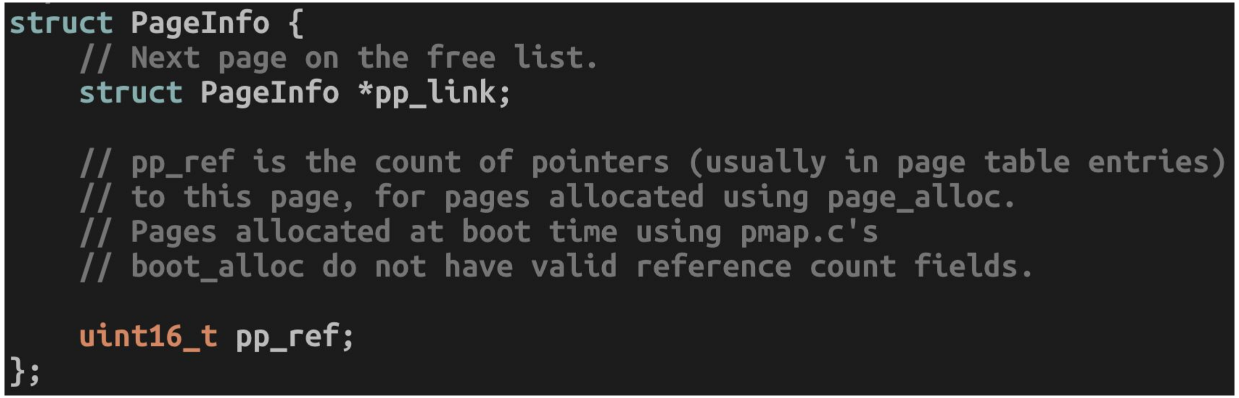
- Struct PageInfo
  - A metadata type that counts number of ‘references’ of the page
  - NOT IN USE : pp_ref == 0

- Struct PageInfo * page_free_list
  - A linked list that contains free physical pages

- We will create Struct PageInfo per each Physical Page and then
  - Create a linked list of free pages…

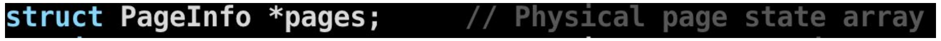
Struct PageInfo in JOS

- A **one-to-one** mapping from a struct **PageInfo** to a physical page
  - An 8 byte struct per each physical memory page
  - If we support 128MB memory, then we will create
    - Total number of physical pages: 128 * 1048576 / 4096 = 32768
  - Total size = **32768 * 8** = 262,144 = 256KB
- A linked-list for managing free physical pages
  - Starting from `page_free_list->pp_link`…
- `pp_ref`
  - Count references
  - Non-zero – in-use
  - Zero – free



```
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;
};
```

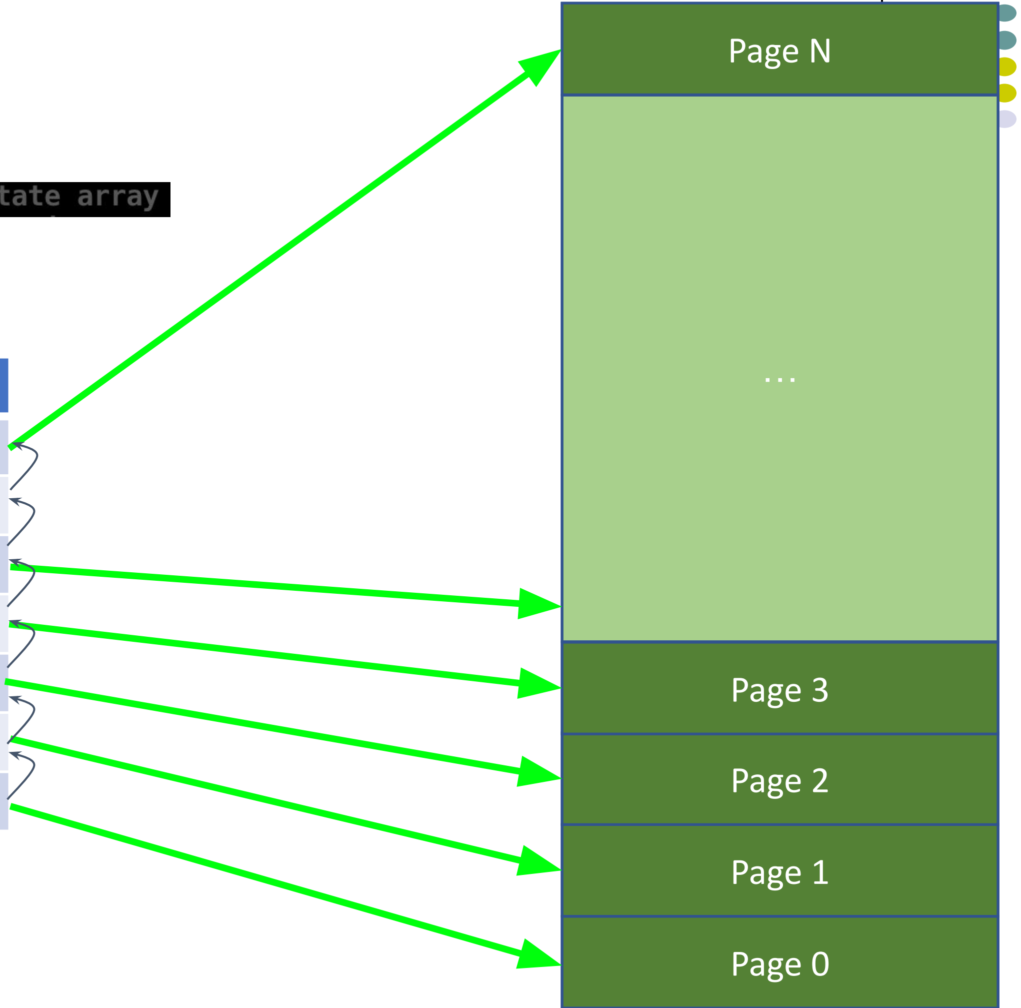
Struct PageInfo



```
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	



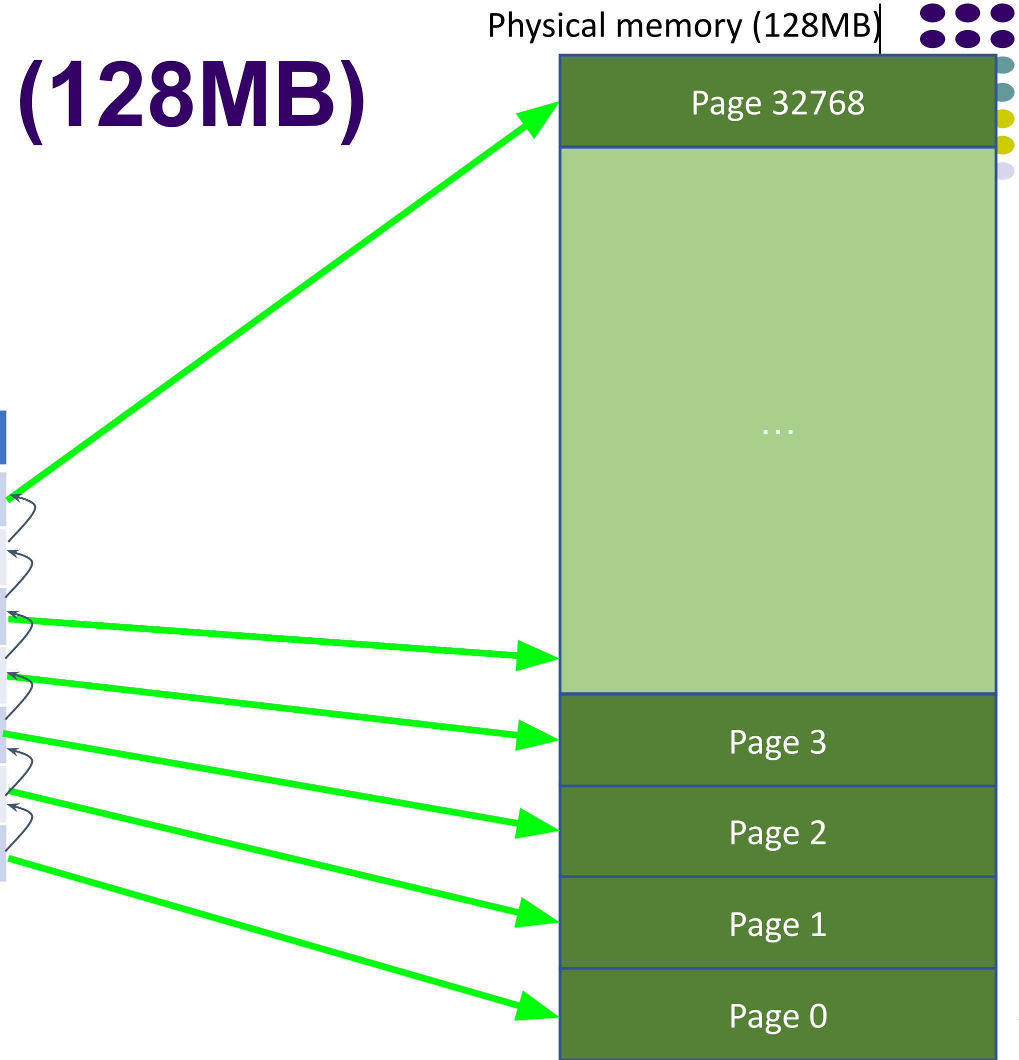
Struct PageInfo (128MB)

128 * 1048576 / 4096 = 32768 Pages

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

Struct PageInfo * pages (array)

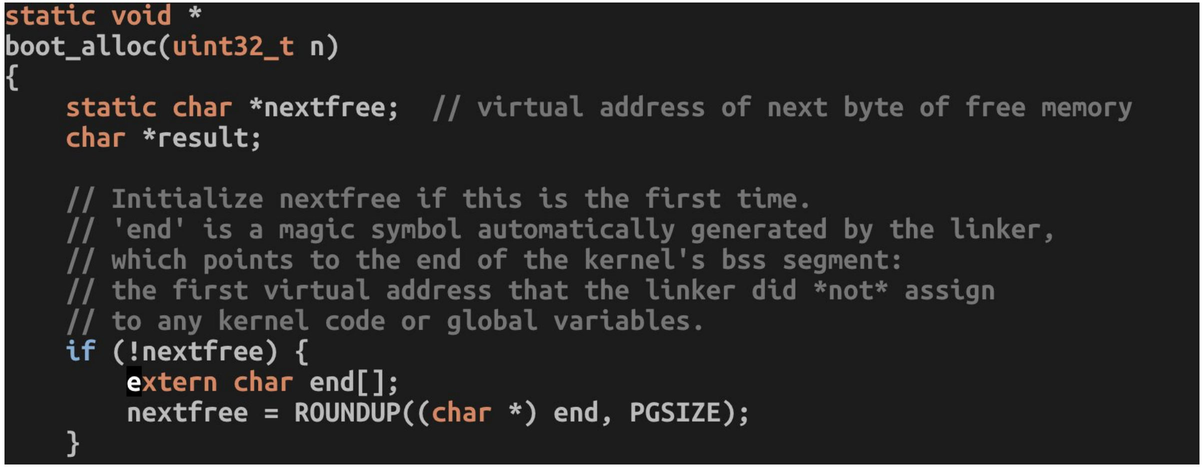
idx	pp_ref	pp_link
32K	0	
…	0	
…	0	
3	0	
2	0	
1	0	
0	0	



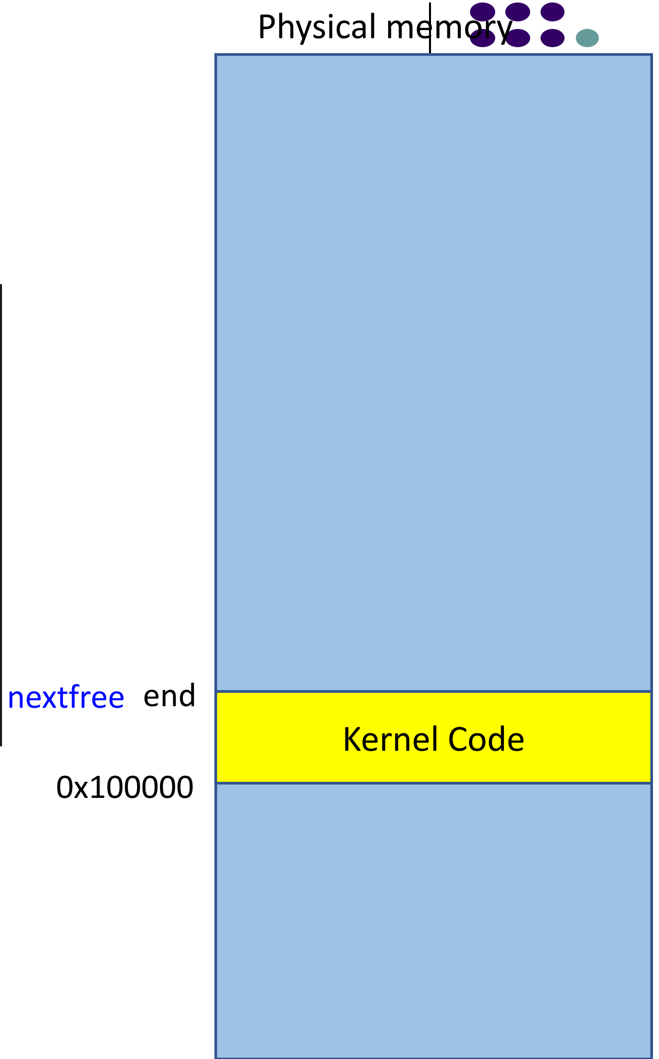
We can put this array into our physical memory

Free Physical Memory (init)

In kern/pmap.c, boot_alloc



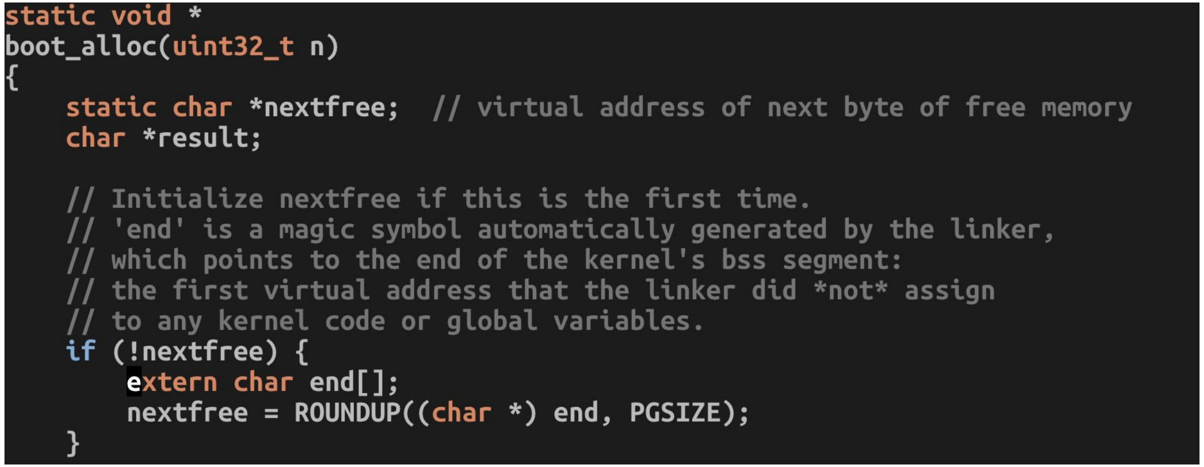
```
static void *
boot_alloc(uint32_t n)
{
    static char *nextfree;  // virtual address of next byte of free memory
    char *result;

    // 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 will point to the end of the kernel code/data

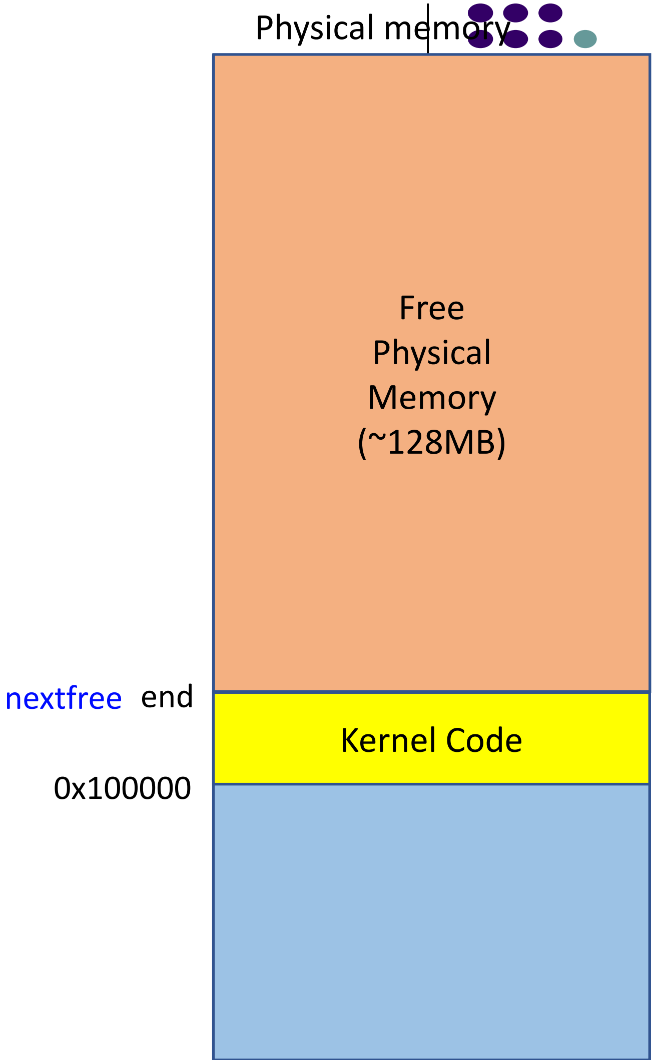
Free Physical Memory (init)

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 will point to the end of the kernel code/data



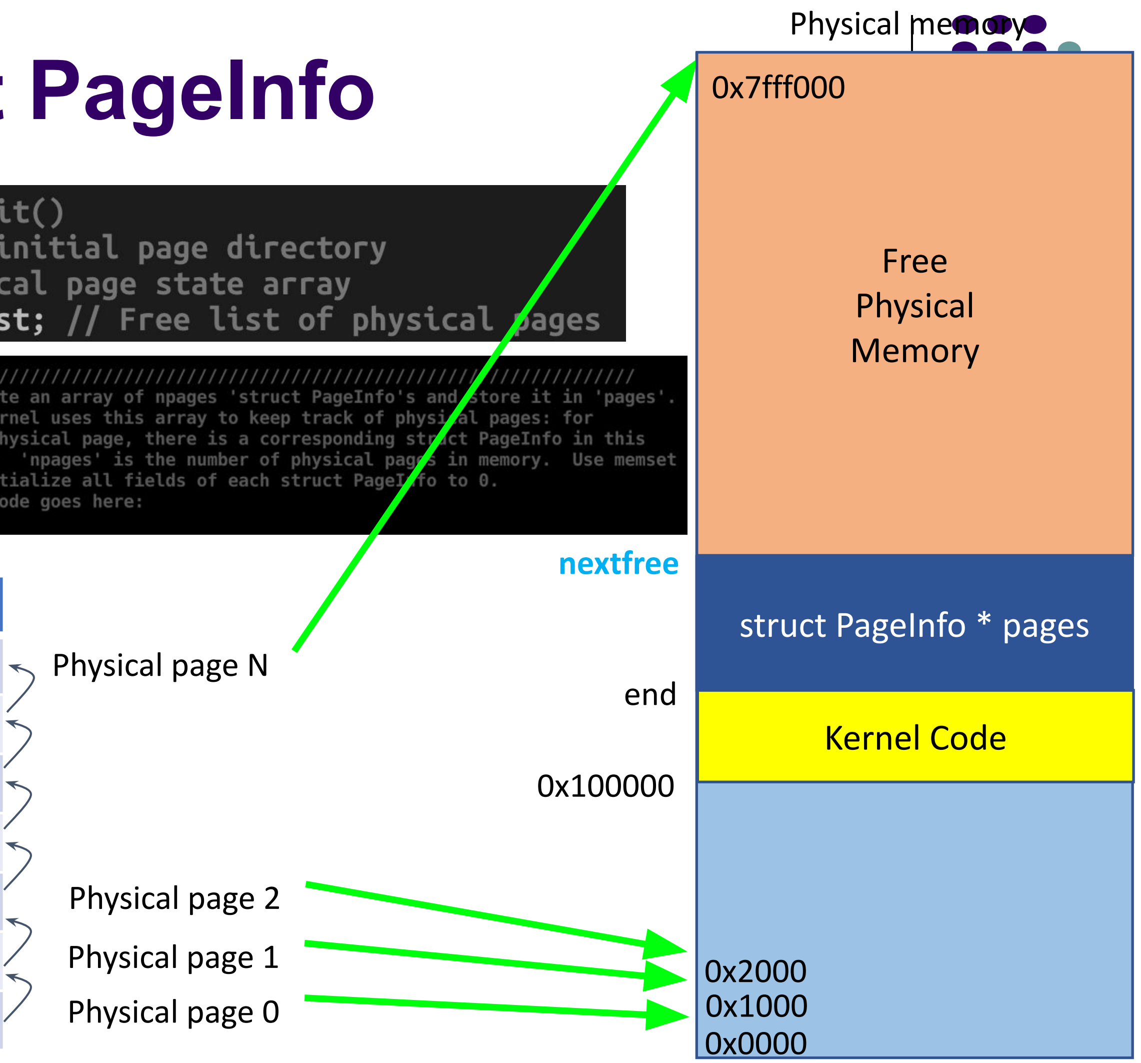
Allocating struct PageInfo

```c
// 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
```

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

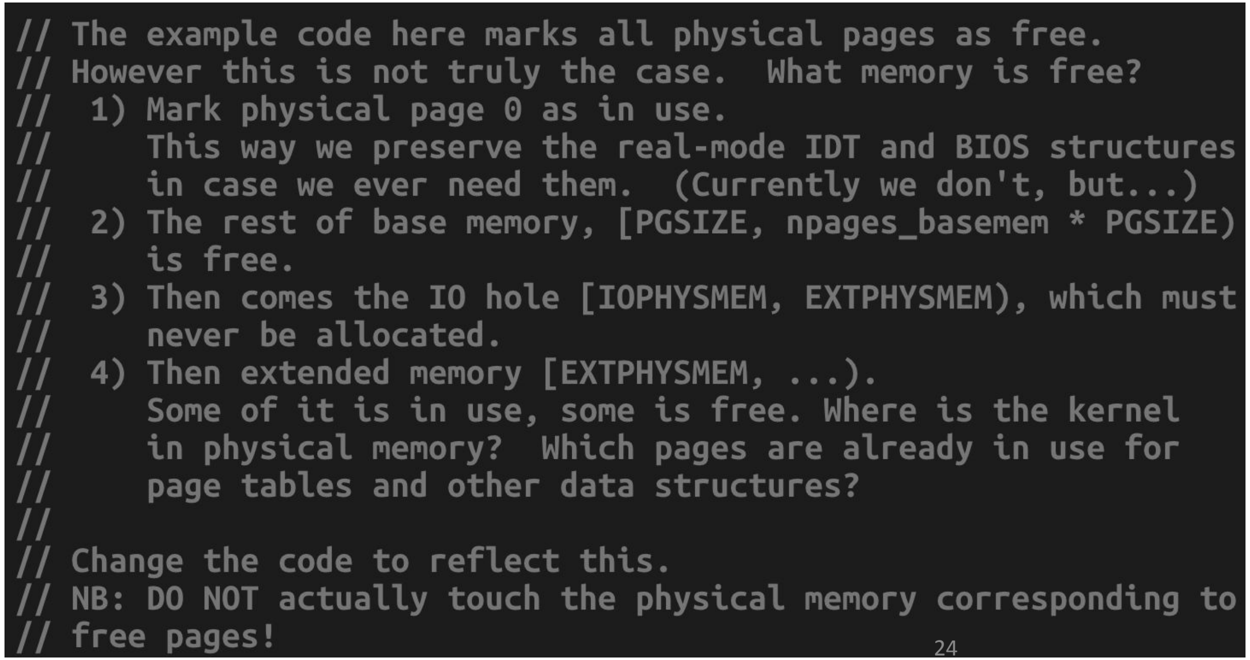
```c
//////////////////////////////////////////////////////////////////////
// Allocate an array of npages 'struct PageInfo's and store it in 'pages'.
// The kernel uses this array to keep track of physical pages: for
// each physical page, there is a corresponding struct PageInfo in this
// array.  '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:
```

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

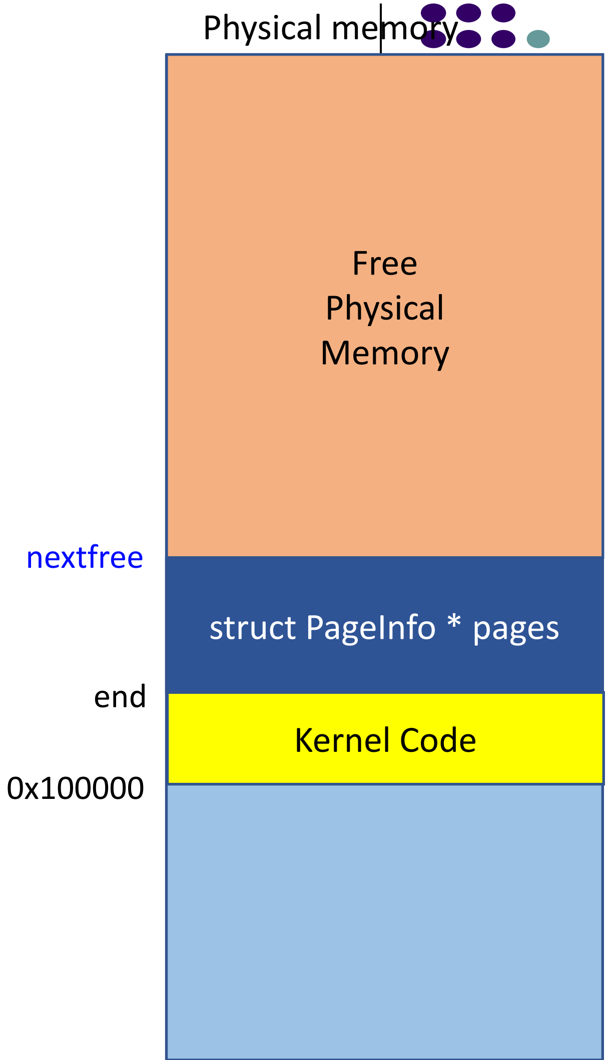


free pages!

- in page_init()

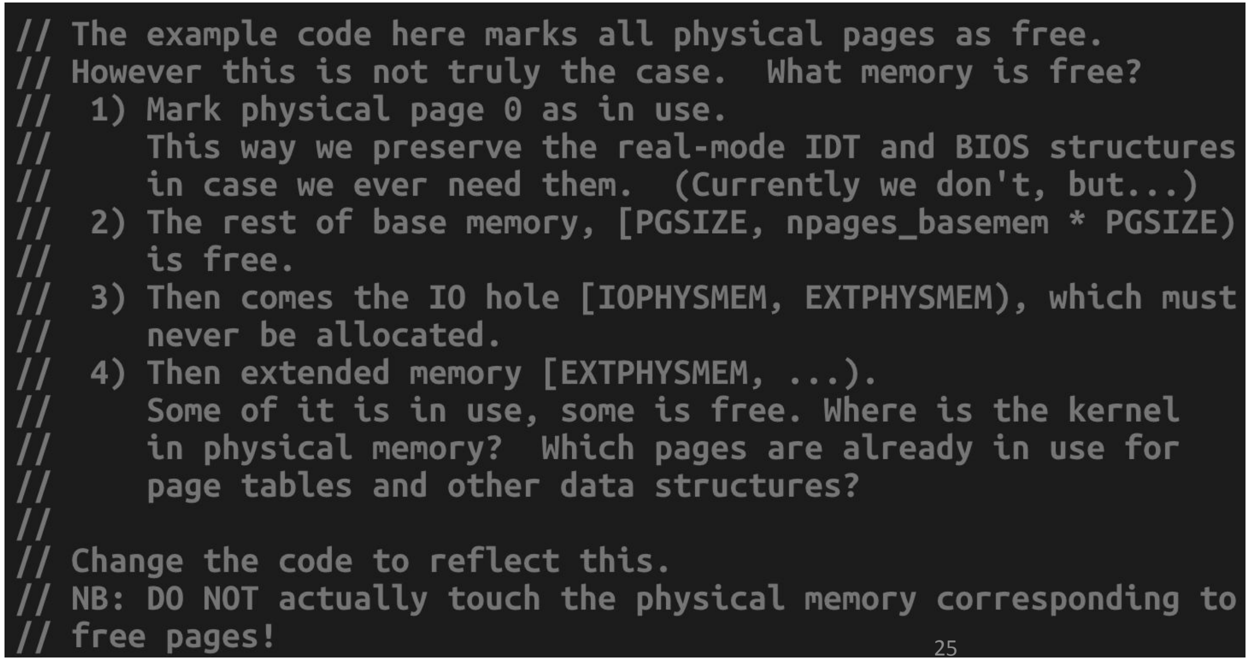


```
// The example code here marks all physical pages as free.
// However this is not truly the case.  What memory is free?
//  1) Mark physical page 0 as in use.
//     This way we preserve the real-mode IDT and BIOS structures
//     in case we ever need them.  (Currently we don't, but...)
//  2) The 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!
```

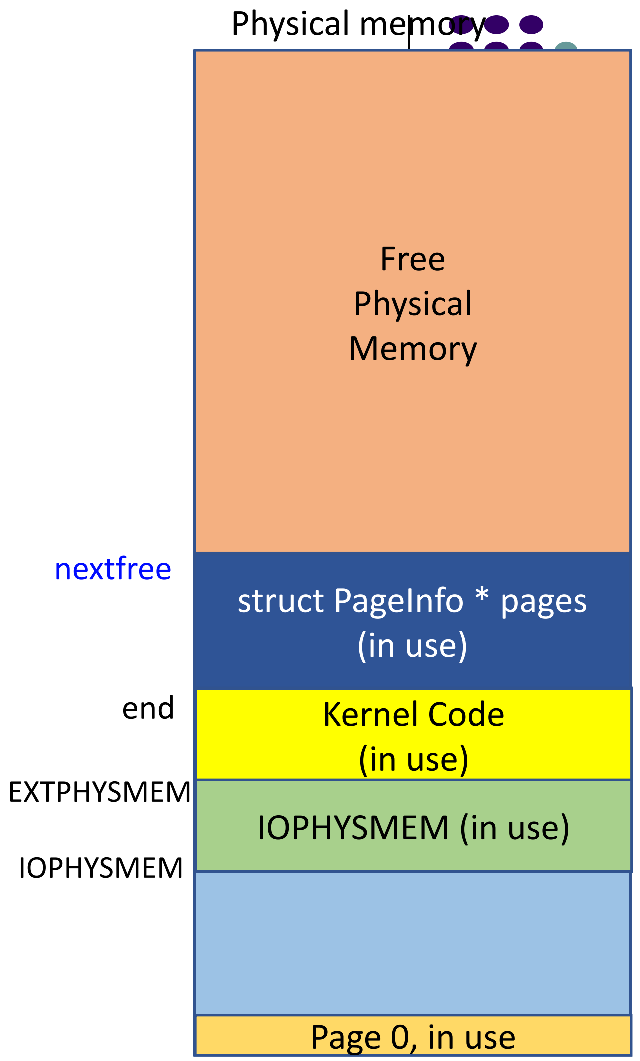


free pages!

- in page_init()

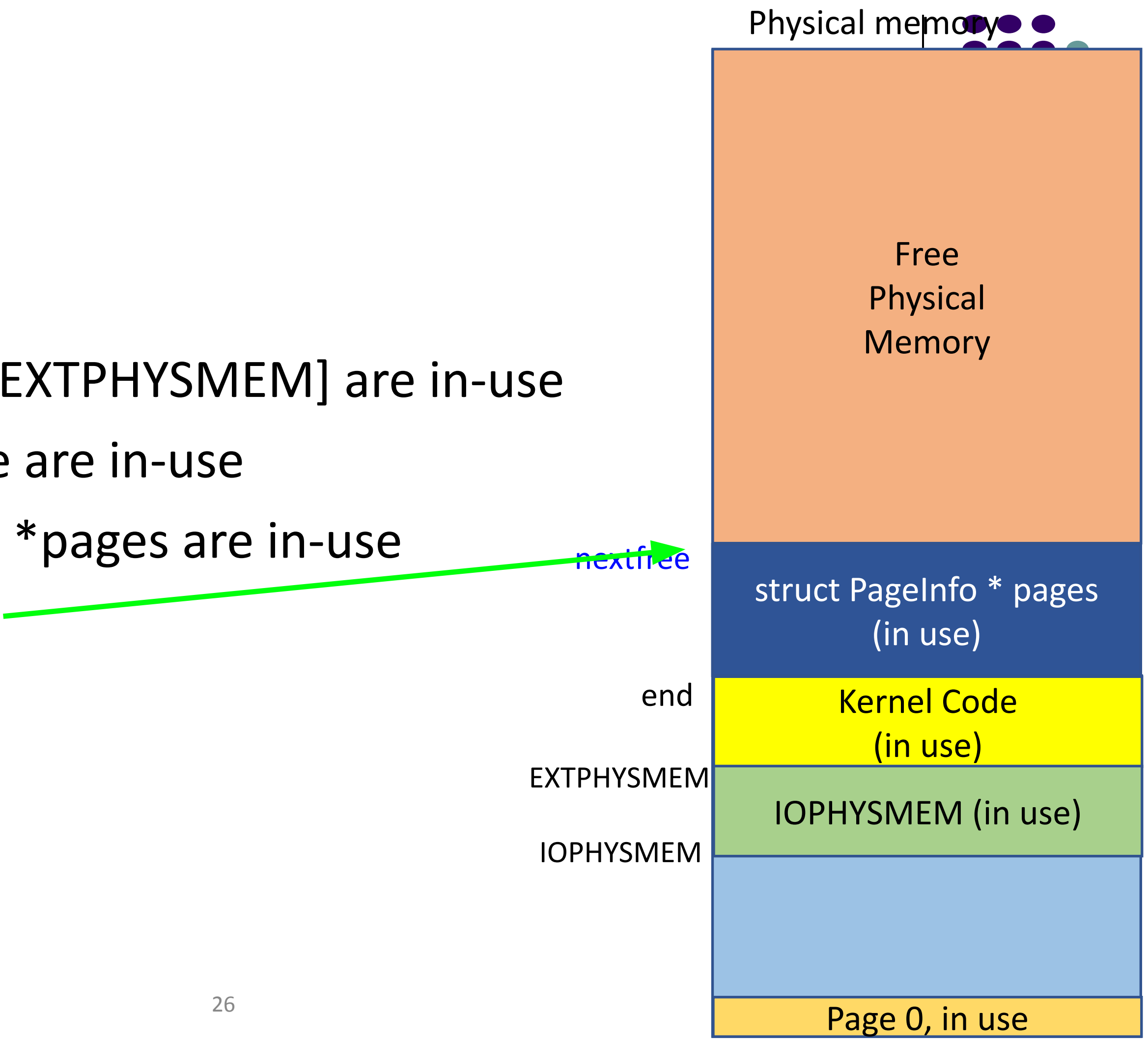


```
// The example code here marks all physical pages as free.
// However this is not truly the case.  What memory is free?
//  1) Mark physical page 0 as in use.
//     This way we preserve the real-mode IDT and BIOS structures
//     in case we ever need them.  (Currently we don't, but...)
//  2) The 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!
```



free pages!

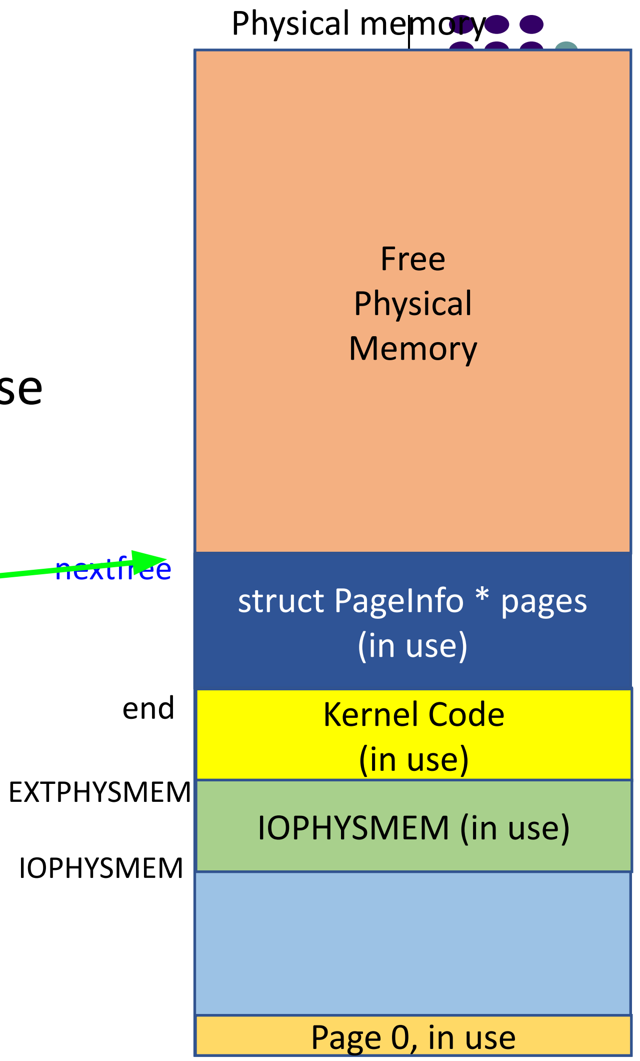
- Page 0 is in-use
- Pages in [IOPHYSMEM ~ EXTPHYSMEM] are in-use
- Pages for the kernel code are in-use
- Pages for struct PageInfo *pages are in-use
- How can you point this?
  - pages + npages ?
  - boot_alloc(0)?



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 this?
  - pages + npages ?
  - boot_alloc(0)?

boot_alloc(0) is better…



Reference counting: Knowing when a page is free

- A typical mechanism for tracking free memory blocks

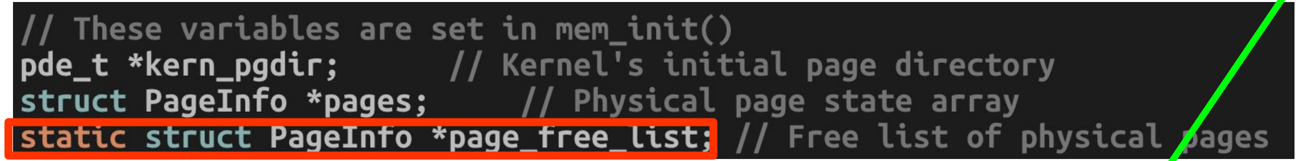
- Mechanism
  - Count up the value (pp_ref++) if the page is referenced by others (in use!)
  - Count down the value (pp_ref--) if not used for one of usages anymore
  - Free if pp_ref == 0

- 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

Reference counting with pp_ref

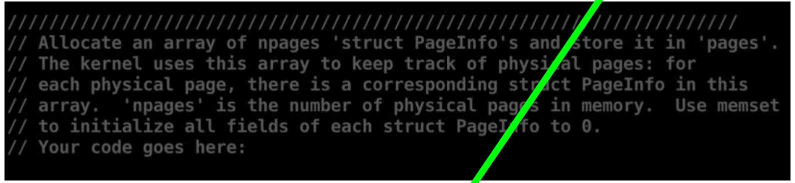
- For in-use memory
  - Set `pp_ref = 1`
- For not-in-use memory
  - Invariant: `pp_ref == 0`
  - **Must be linked with pages_free_list**
- When assigning the page to a virtual address
  - `pp_ref++`
- When releasing the page from a virtual address
  - `pp_ref--`

Allocating struct PageInfo



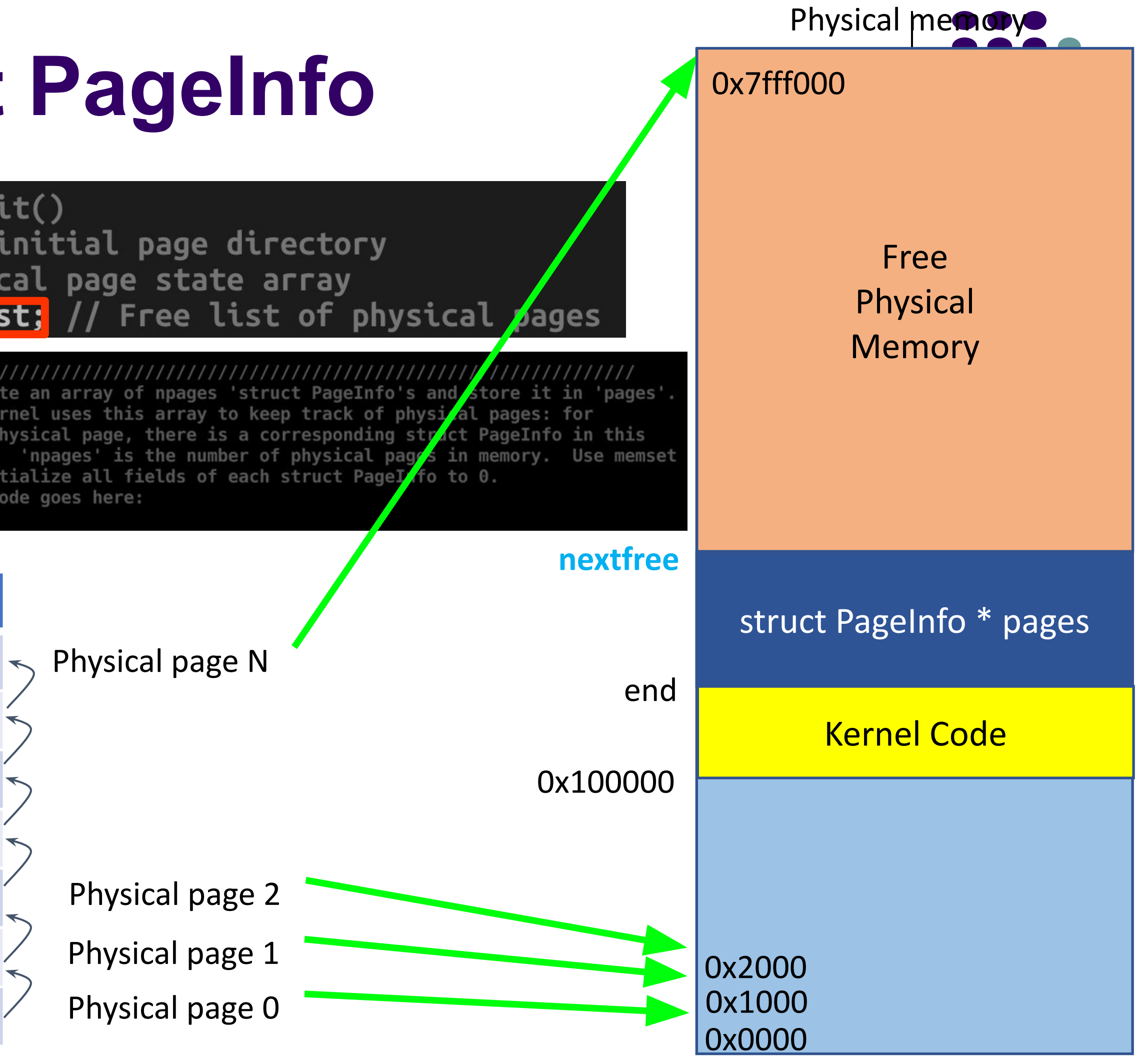
```
// These variables are set in mem_init()
pde_t *kern_pgdir;          // Kernel's initial page directory
struct PageInfo *pages;         // Physical page state array
static struct PageInfo *page_free_list; // Free list of physical pages
```

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



```
//////////////////////////////////////////////////////////////////////
// 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:
```

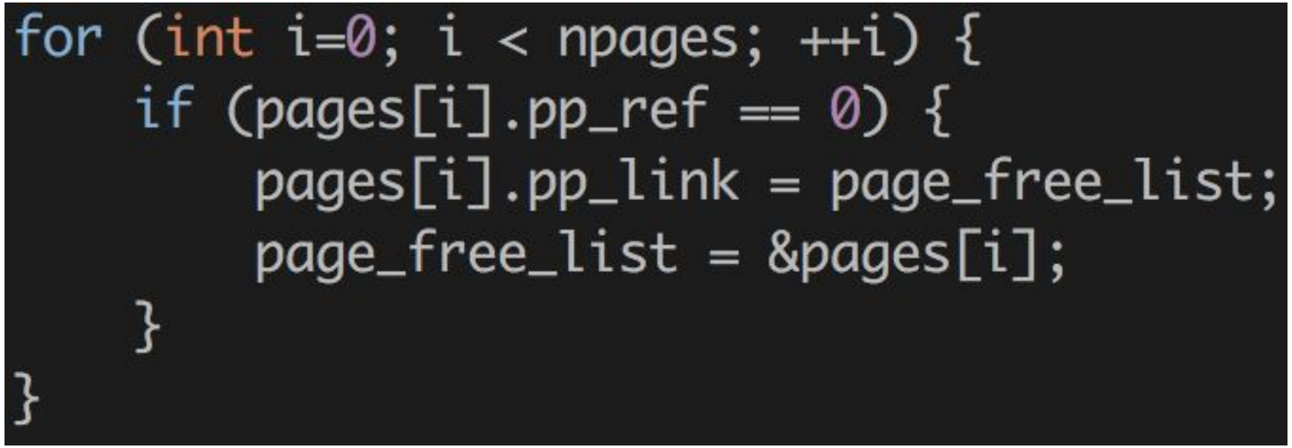
idx	pp_ref	pp_link
N	0	
…	0	
…	0	
3	0	
2	0	
1	0	
0	0	



Linked list of free pages

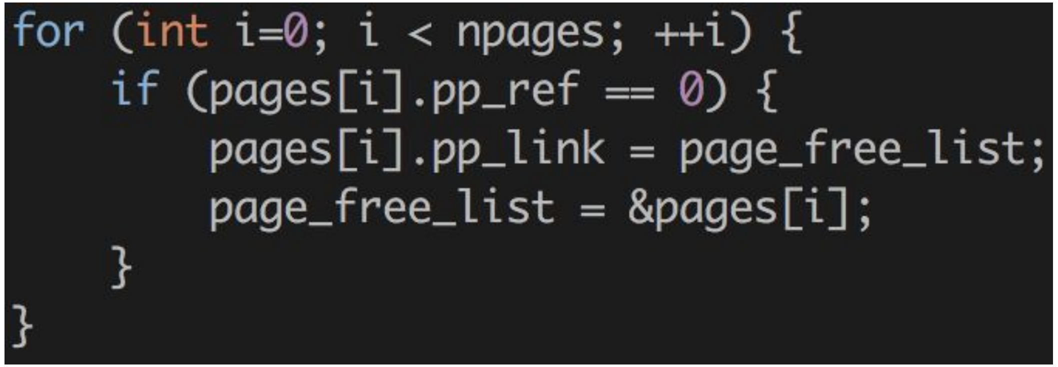
- Start with NULL at the head
  - `page_free_list = NULL;`
- After set `pp_ref` of all pages, do something like the following..

This will build a linked list…



```
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];
    }
}
```

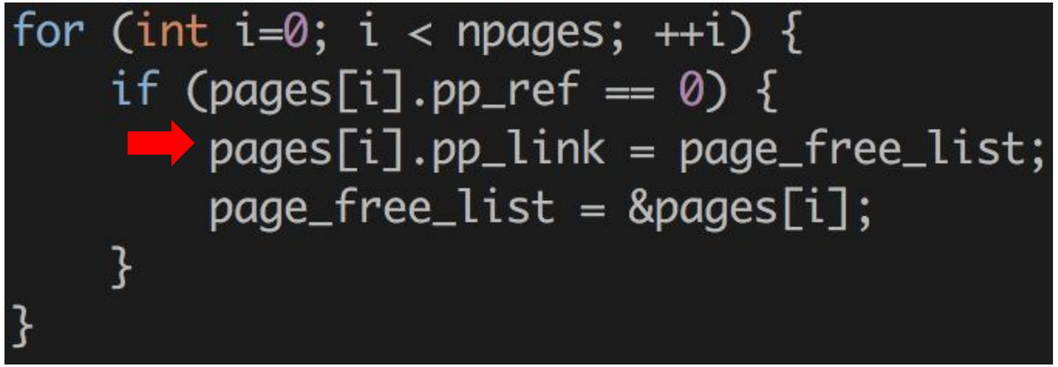
Building free list



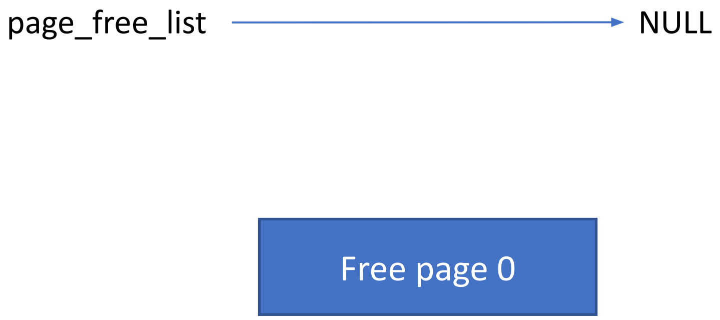
```
for (int i=0; i < npages; ++i) {
    if (pages[i].pp_ref == 0) {
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}
```

$$\begin{tikzcd}\text{page\_free\_list} \arrow[r] & \text{NULL}\end{tikzcd}$$

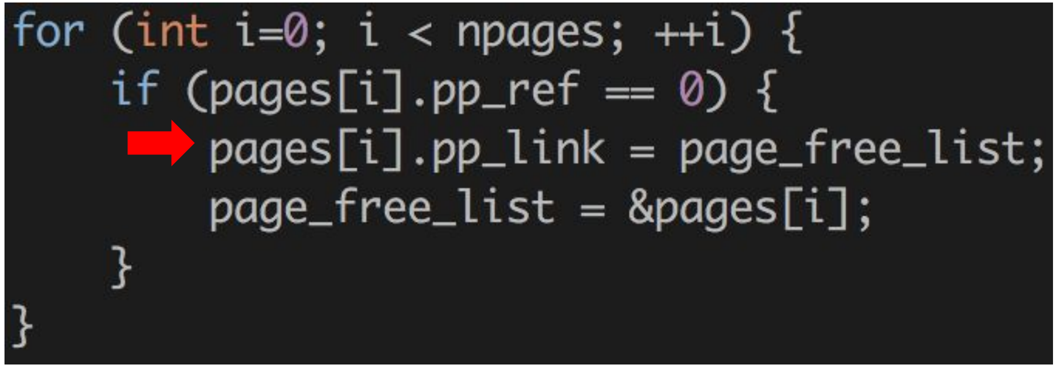
Building free list



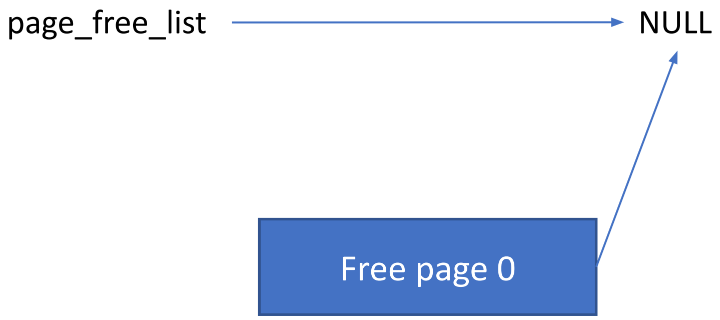
```
for (int i=0; i < npages; ++i) {
    if (pages[i].pp_ref == 0) {
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}
```



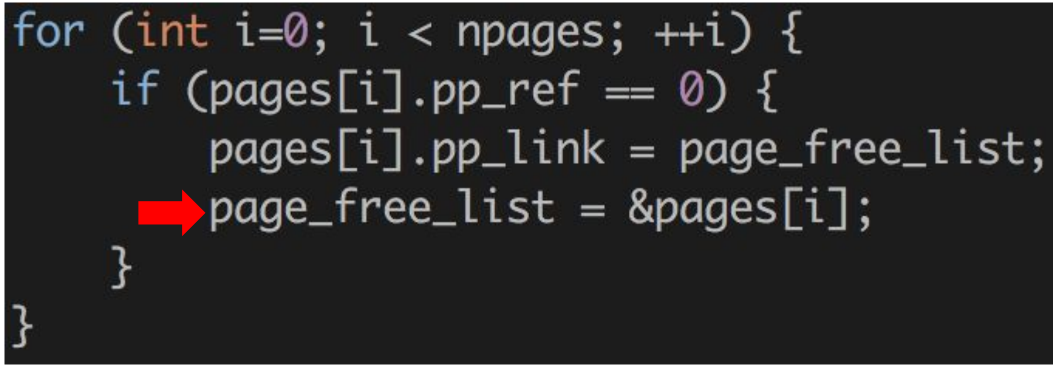
Building free list



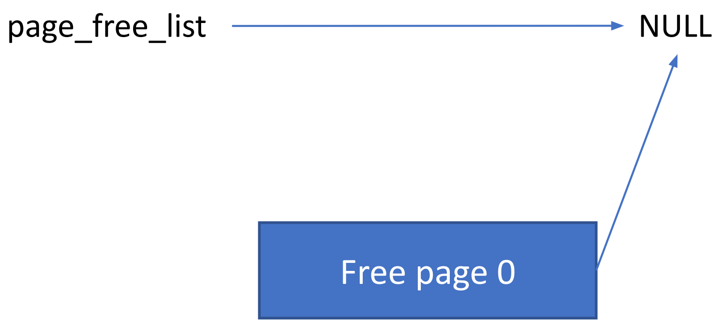
```
for (int i=0; i < npages; ++i) {
    if (pages[i].pp_ref == 0) {
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}
```



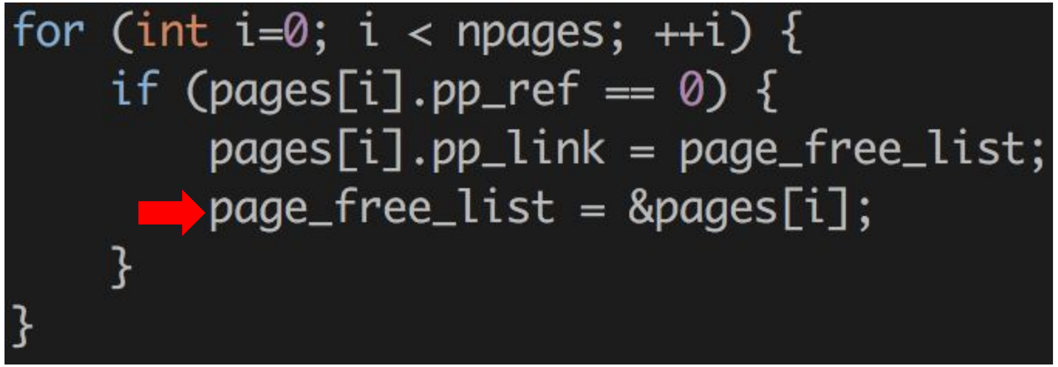
Building free list



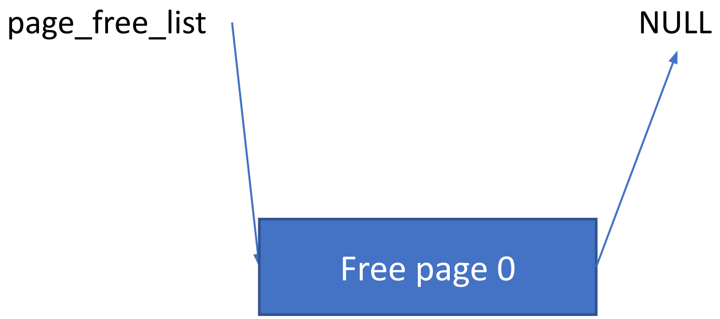
```
for (int i=0; i < npages; ++i) {
    if (pages[i].pp_ref == 0) {
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}
```



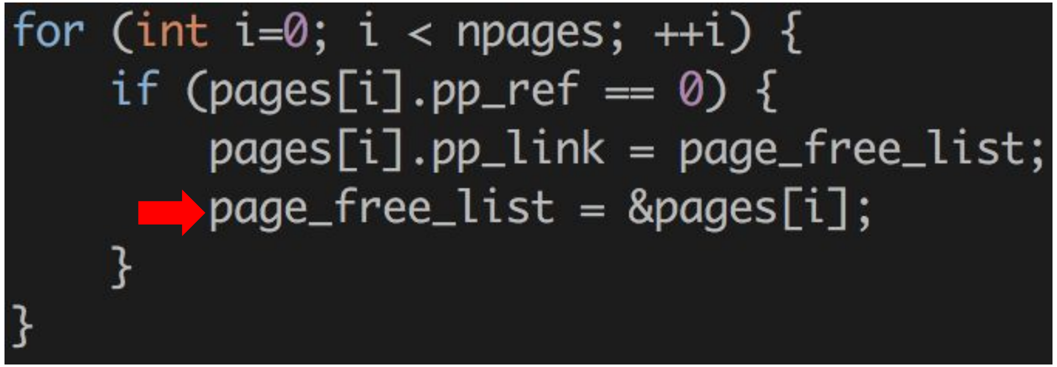
Building free list



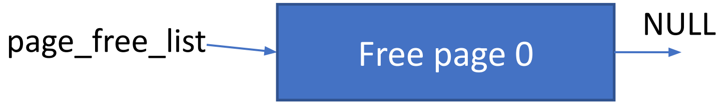
```
for (int i=0; i < npages; ++i) {
    if (pages[i].pp_ref == 0) {
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}
```



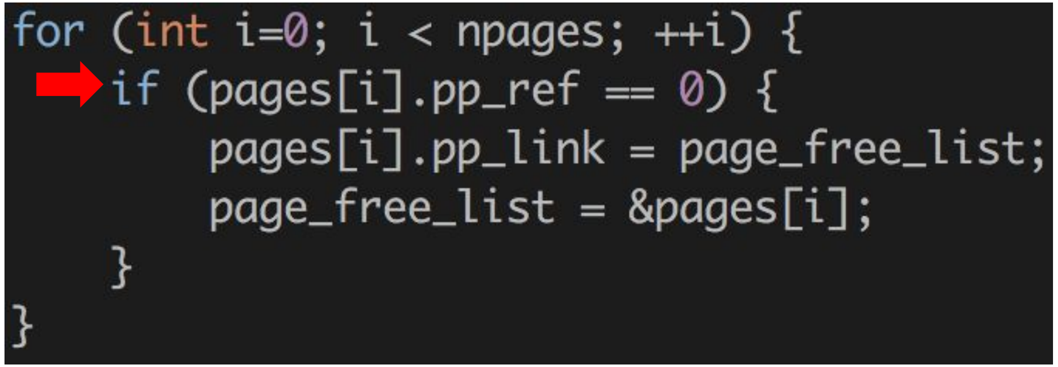
Building free list



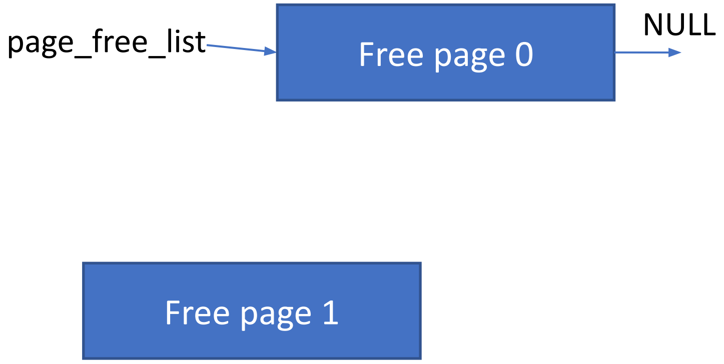
```
for (int i=0; i < npages; ++i) {
    if (pages[i].pp_ref == 0) {
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}
```



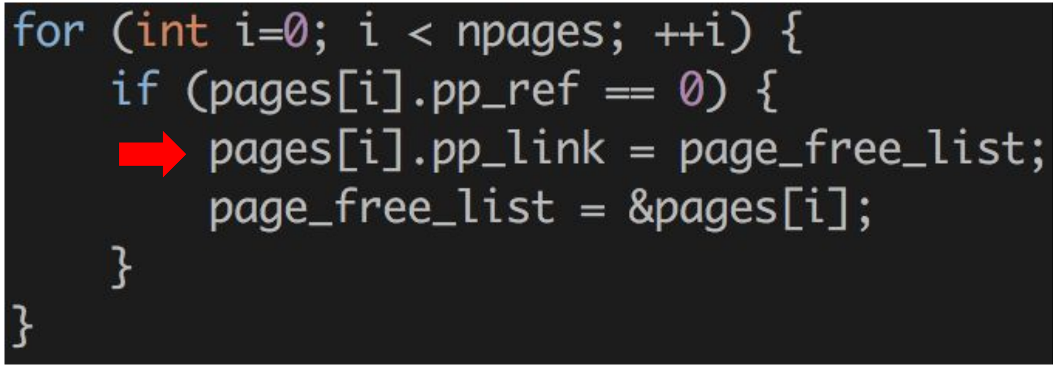
Building free list



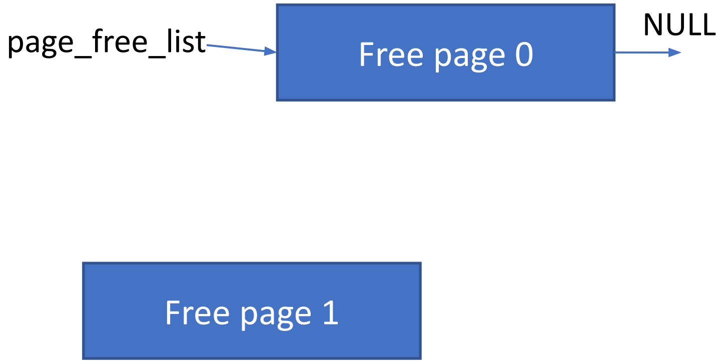
```
for (int i=0; i < npages; ++i) {
    if (pages[i].pp_ref == 0) {
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}
```



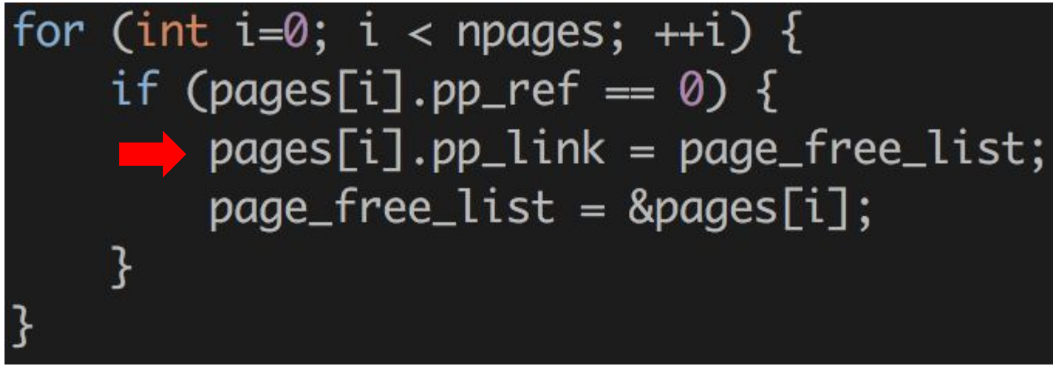
Building free list



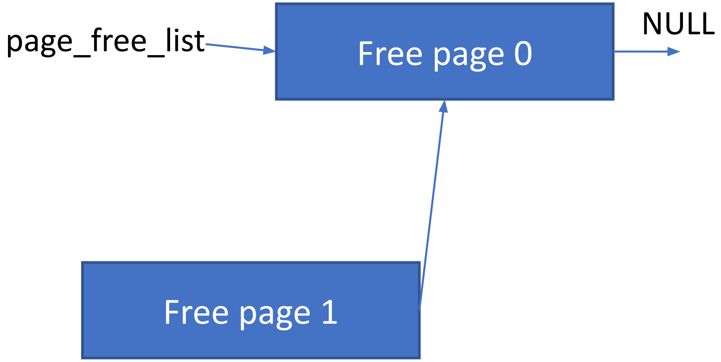
```
for (int i=0; i < npages; ++i) {
    if (pages[i].pp_ref == 0) {
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}
```



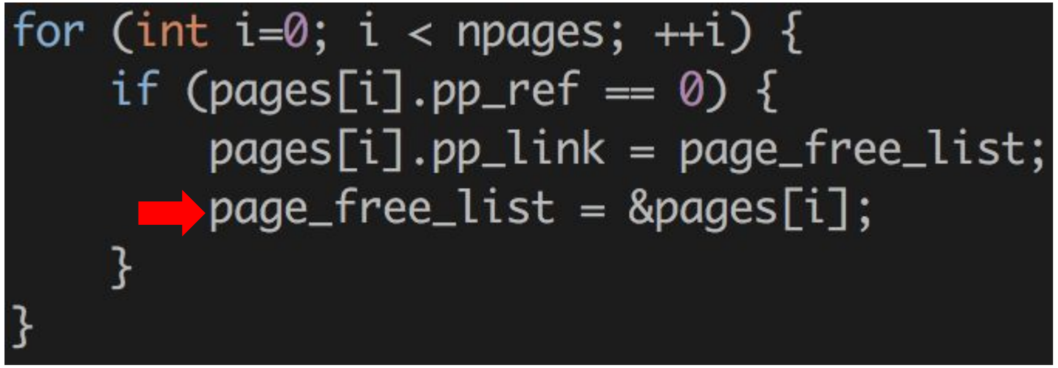
Building free list



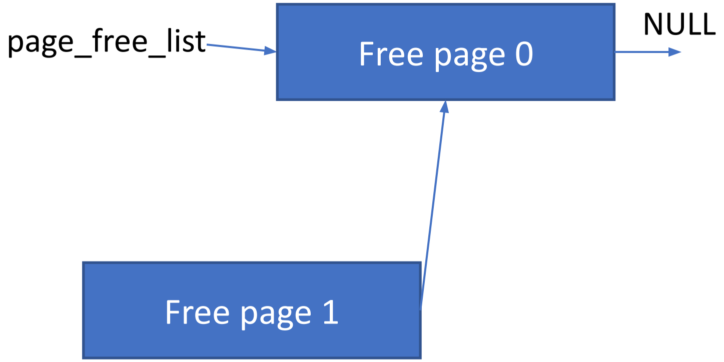
```
for (int i=0; i < npages; ++i) {
    if (pages[i].pp_ref == 0) {
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}
```



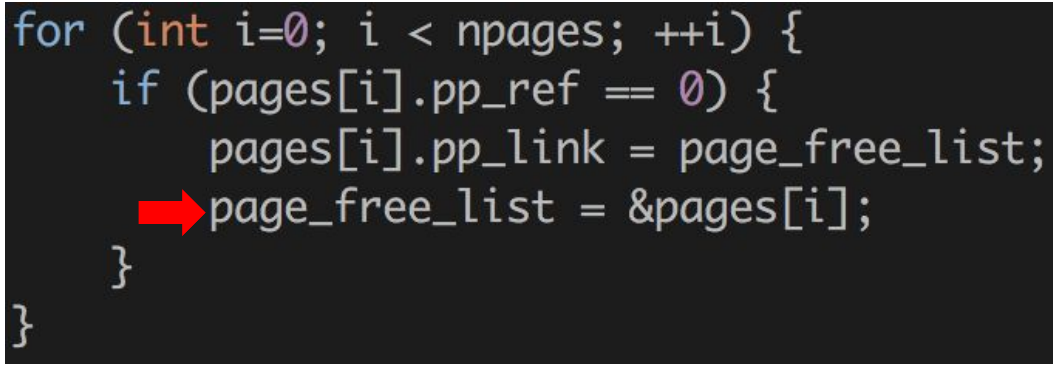
Building free list



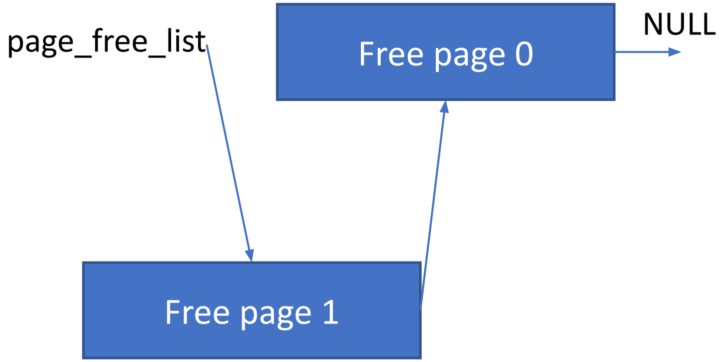
```
for (int i=0; i < npages; ++i) {
    if (pages[i].pp_ref == 0) {
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}
```



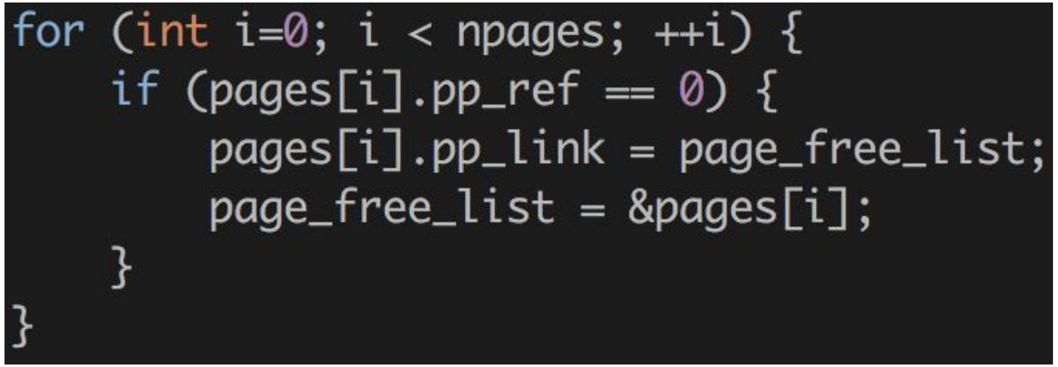
Building free list



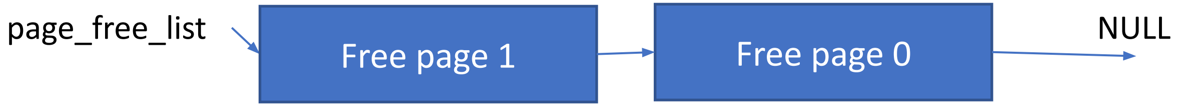
```
for (int i=0; i < npages; ++i) {
    if (pages[i].pp_ref == 0) {
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}
```



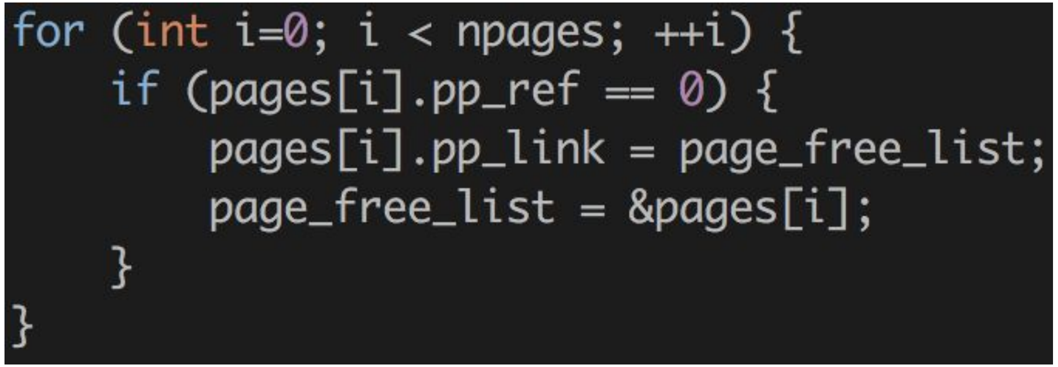
Building free list



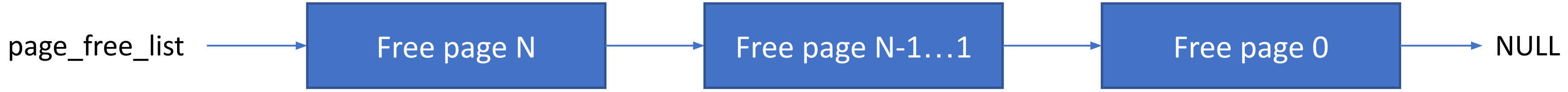
```
for (int i=0; i < npages; ++i) {
    if (pages[i].pp_ref == 0) {
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}
```



Building free list

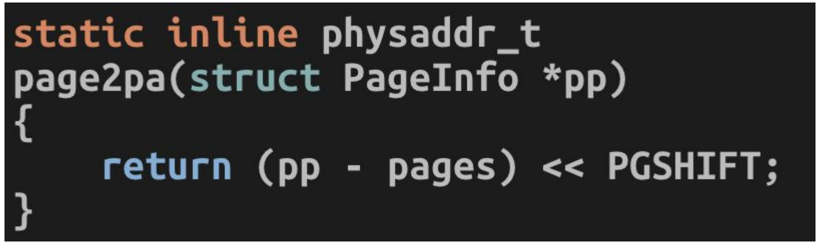


```
for (int i=0; i < npages; ++i) {
    if (pages[i].pp_ref == 0) {
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}
```

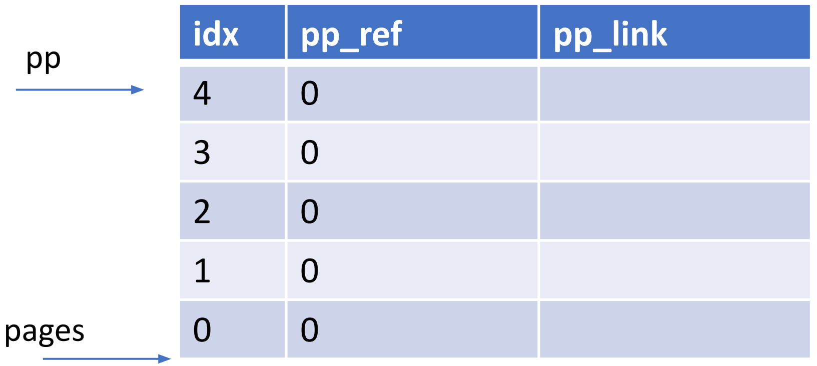


page2pa(struct PageInfo *pp)

- 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;
}
```

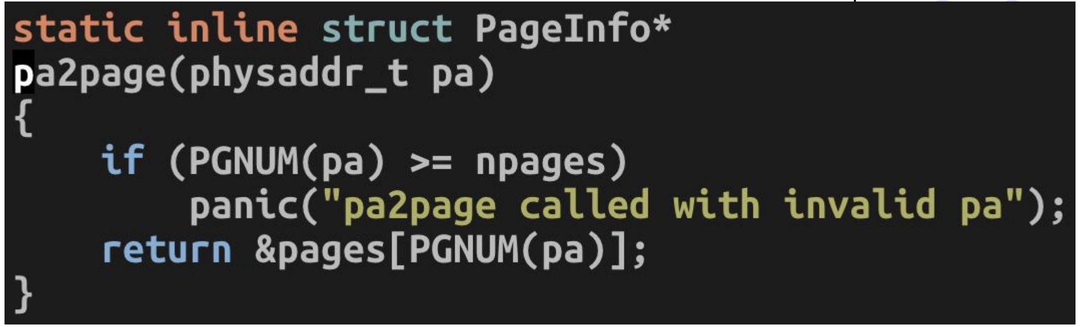


idx	pp_ref	pp_link
4	0	
3	0	
2	0	
1	0	
0	0	

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

pa2page(physaddr_t pa)

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



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

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

Sample Qs

- 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

Sample Qs

- 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

Sample Qs

- In the x86 real mode, which address the following segment:offset pair points to?
- 0x8000:0x3131
  - A. 0xb131
  - B. 0x3131
  - C. 0x83131
  - D. 0x103131
  - E. 0x11131

Sample Qs

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

Sample Qs

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

Sample Qs

- 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

Sample Qs

- 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

Sample Qs

- What kind of benefit can we enjoy by 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