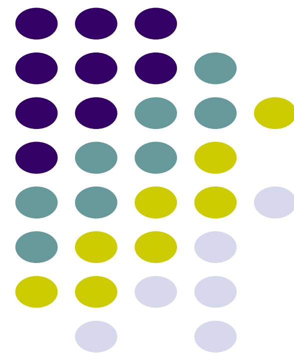
More Virtual Memory and JOS memory management

ECE 469, Jan 28

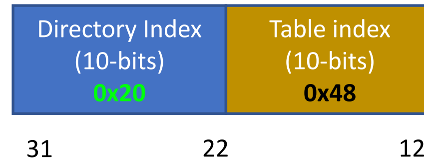
Aravind Machiry



Recap – Page Table & Addr Translation

Mem access #1

CR3[0x20]

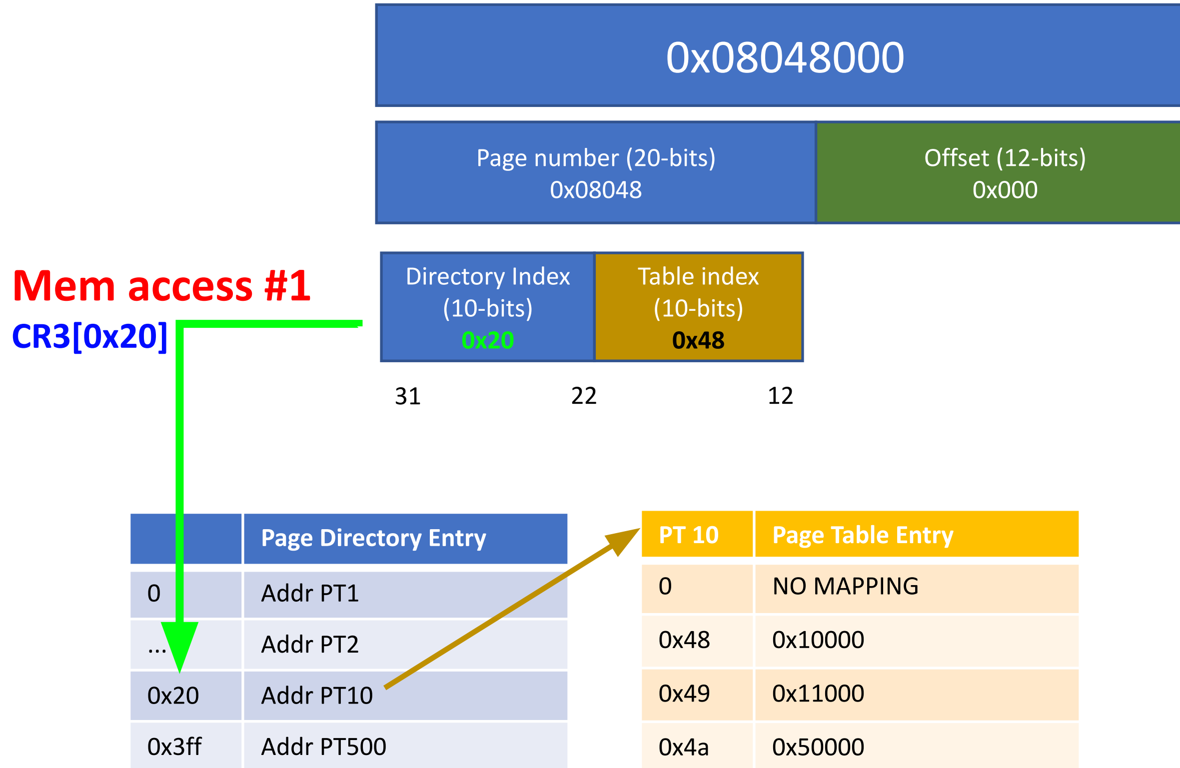


Virtual	Physical
0x8048000	0x10000
0x8049000	0x11000
0x804a000	0x50000

	Page Directory Entry
0	Addr PT1
...	Addr PT2
0x20	Addr PT10
0x3ff	Addr PT500

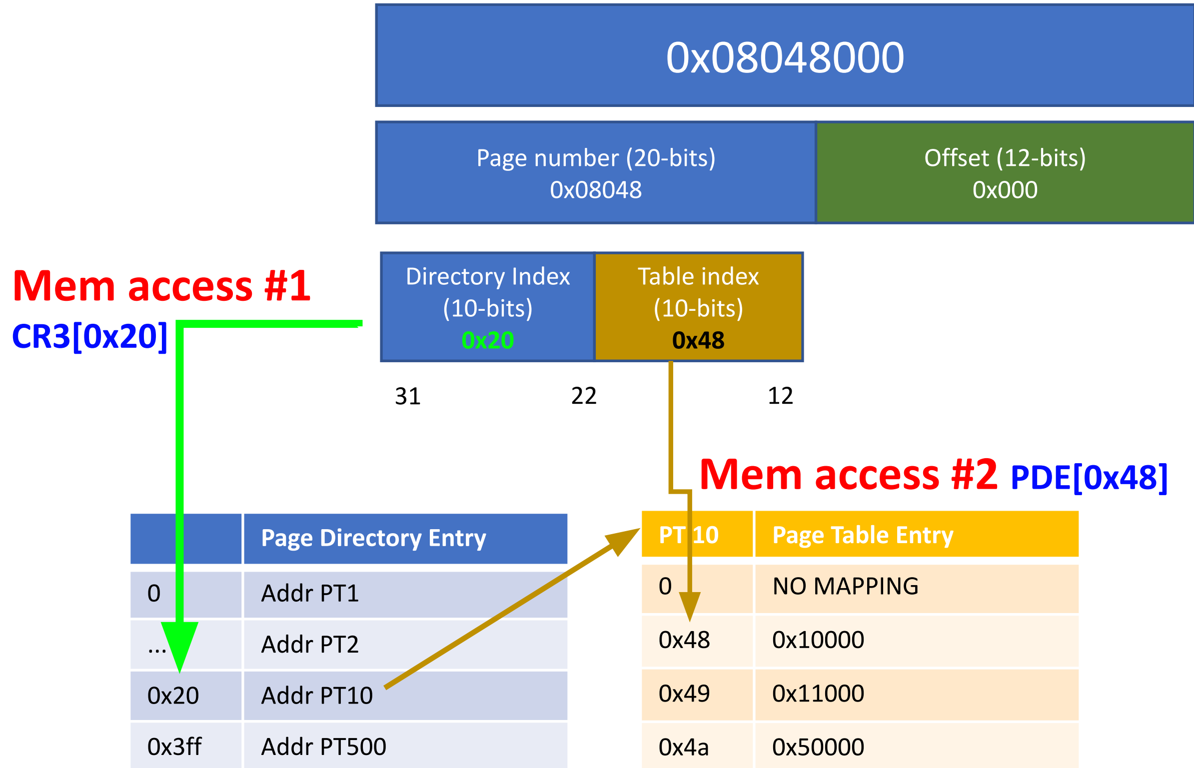
PT 10	Page Table Entry
0	NO MAPPING
0x48	0x10000
0x49	0x11000
0x4a	0x50000

Recap – Page Table & Addr Translation



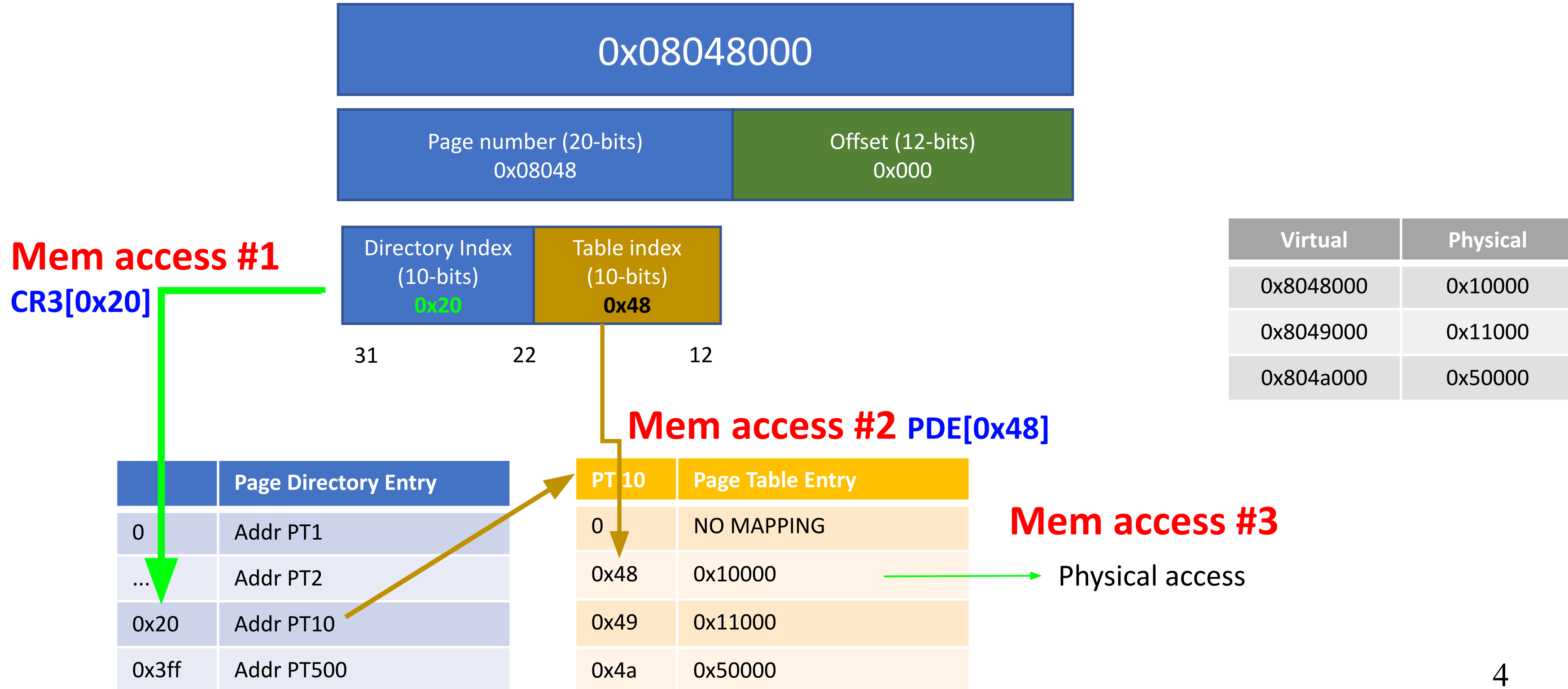
Virtual	Physical
0x8048000	0x10000
0x8049000	0x11000
0x804a000	0x50000

Recap – Page Table & Addr Translation

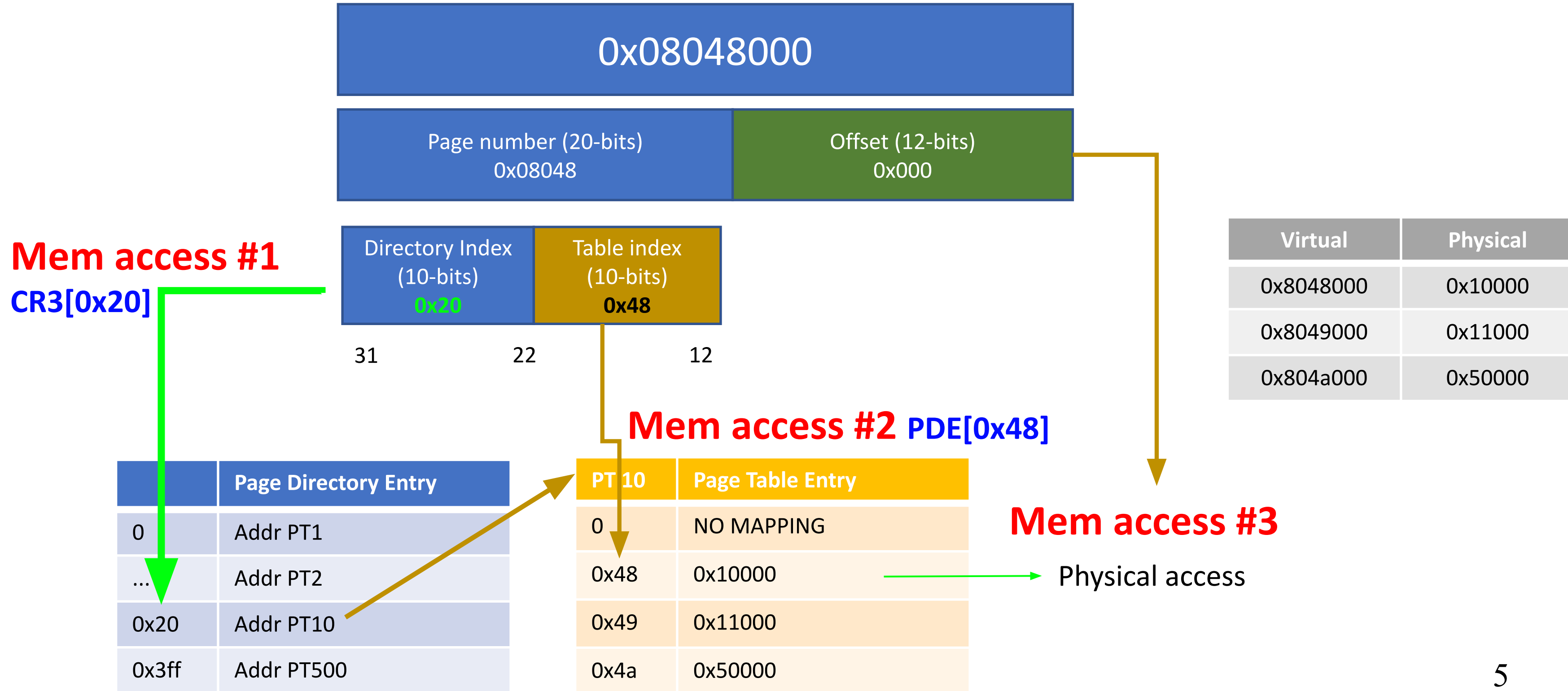


Virtual	Physical
0x8048000	0x10000
0x8049000	0x11000
0x804a000	0x50000

Recap – Page Table & Addr Translation



Recap – Page Table & Addr Translation

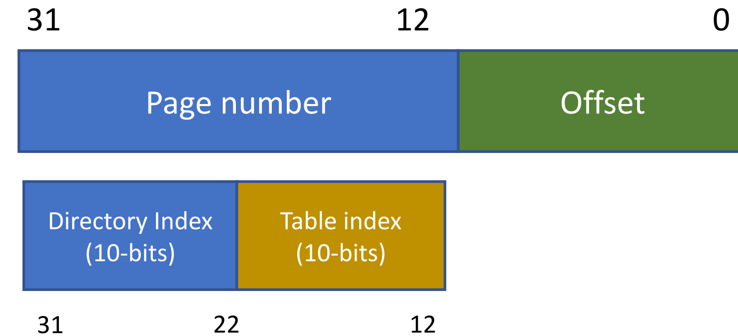


More Virtual Memory

- Multi-level page tables
- Page Table Permissions
- Physical Memory Management

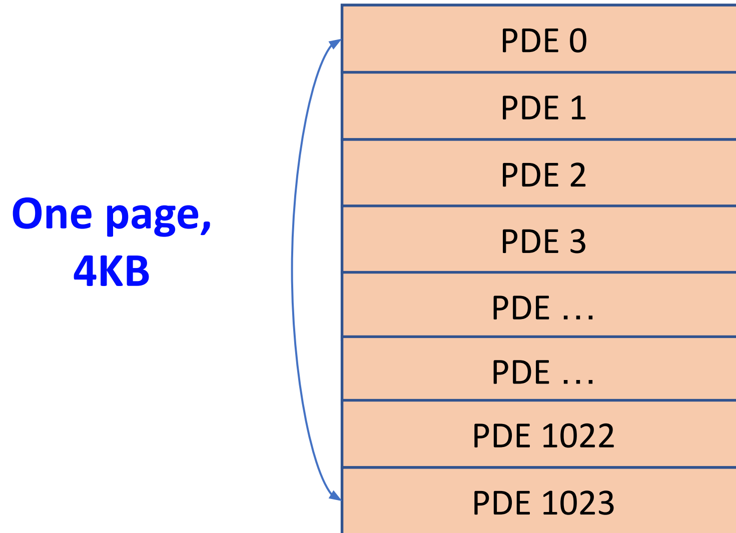
Page Directory / Table

- In x86 (32-bit), CPU uses 2-level page table
- 10-bit directory index
- 10-bit page table index
- 12-bit offset
- **2-level paging**



Size of Page Directory!

- Page Size = 4 KB



4096 / **4** = 1024 entries

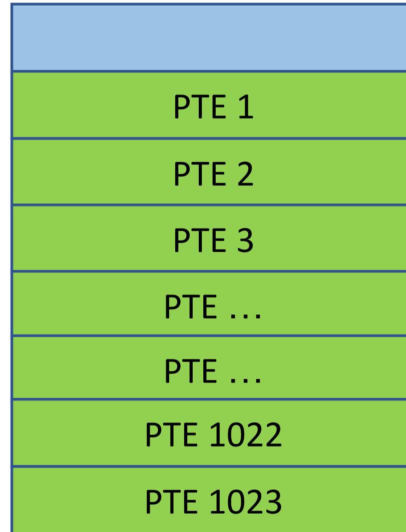
$1024 == 2^{10}$

10-bit index for PD

Size of Page Table!

- Page Size = 4 KB

One page, 4KB



Each entry is 4-byte (32 bits)

4096 / **4** = 1024 entries

1024 == 2^{10}

10-bit index for PT

Increasing Virtual Address Space

- 32-bit address:
  - 2^32 == total 4GB
  - We ignore lower 12 bits = 2^52 total pages
  - Page directory: 4KB.
  - Page table chunk or Second level page table: 4KB.
  - PTE/PDE entry size = 4 bytes

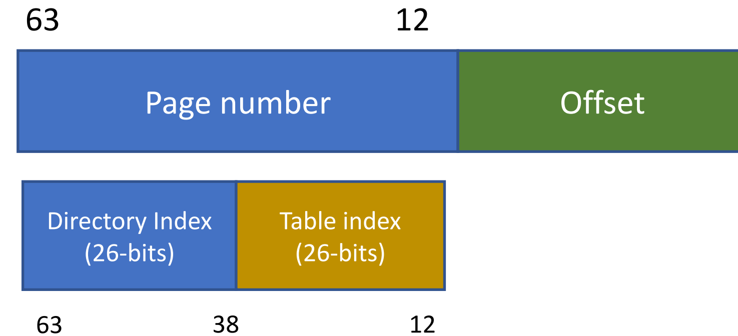
- **All metadata is of size 4KB**!!

Increasing Virtual Address Space

- 64-bit addresses:
  - 2^64 == 16 EB == 16,384 PB == 16,777,216 TB
  - We ignore lower 12 bits = 2^52 total pages
- **How should we handle paging?**

Choices of paging in x64

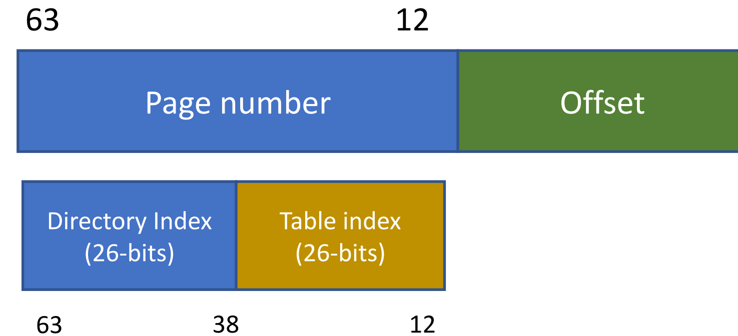
- 2-level paging
  - 26-bit directory index
  - 26-bit page table index
  - 12-bit offset
  - Each PTE/PDE size = 8 bytes



Choices of paging in x64

- 2-level paging
  - 26-bit directory index
  - 26-bit page table index
  - 12-bit offset
  - Each PTE/PDE size = 8 bytes

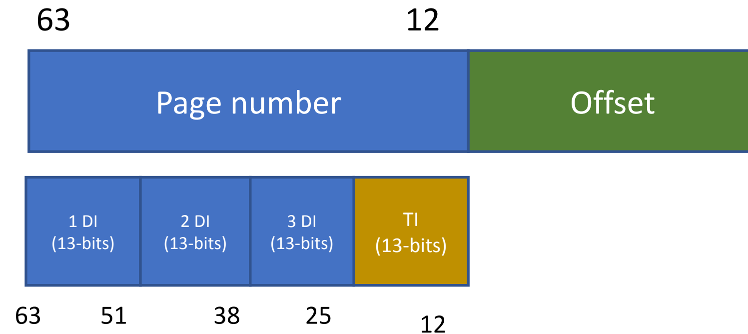
- **Page directory size or Page table size:**
  - **(2^26)*8 = 512 MB**



Choices of paging in x64

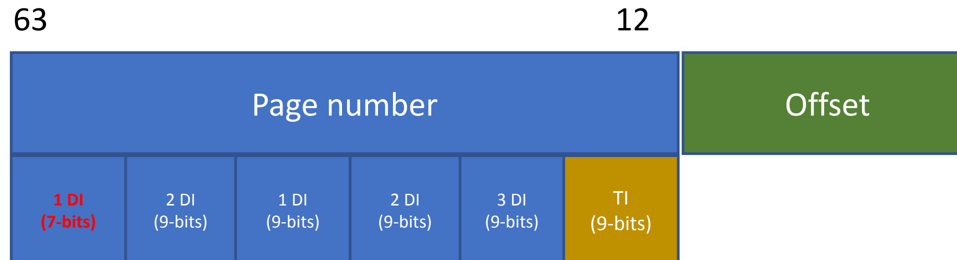
- 4-level paging
  - 13-bit directory indexes - 3 level
  - 13-bit page table index
  - 12-bit offset
  - Each PTE/PDE size = 8 bytes

- **Page directories size or Page table size:**
  - **(2^13)*8 = 64 KB**



Choices of paging in x64

- All metadata should be of 4KB (contiguous memory) chunks for efficient memory usage
- What should be the ideal level of paging for 64-bit address?
  - Each entry (PTE/PDE) size = 8 bytes
  - Number of entries in 4KB = (4KB/8) = 512 = (2^9) = 9-bit index
  - **64-bit address:**
    - **12-bit offset**
    - **52-bit page number:**
      - **52/9 = 5.777 = 6**
      - **6-level paging**



x86_64: 48-bit address space

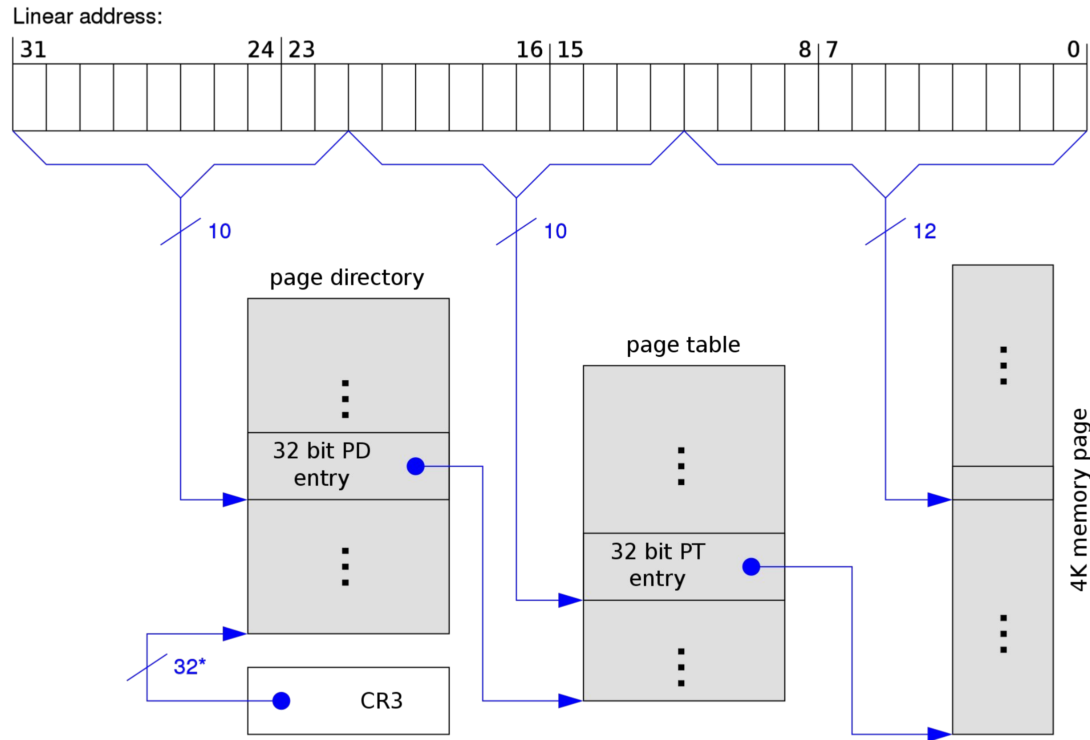
- Initial amd64 processors use only 48-bit virtual address space
  - 2^48 = 256 TB

- Each level of table tree can process 9 bits (512 entries in table)

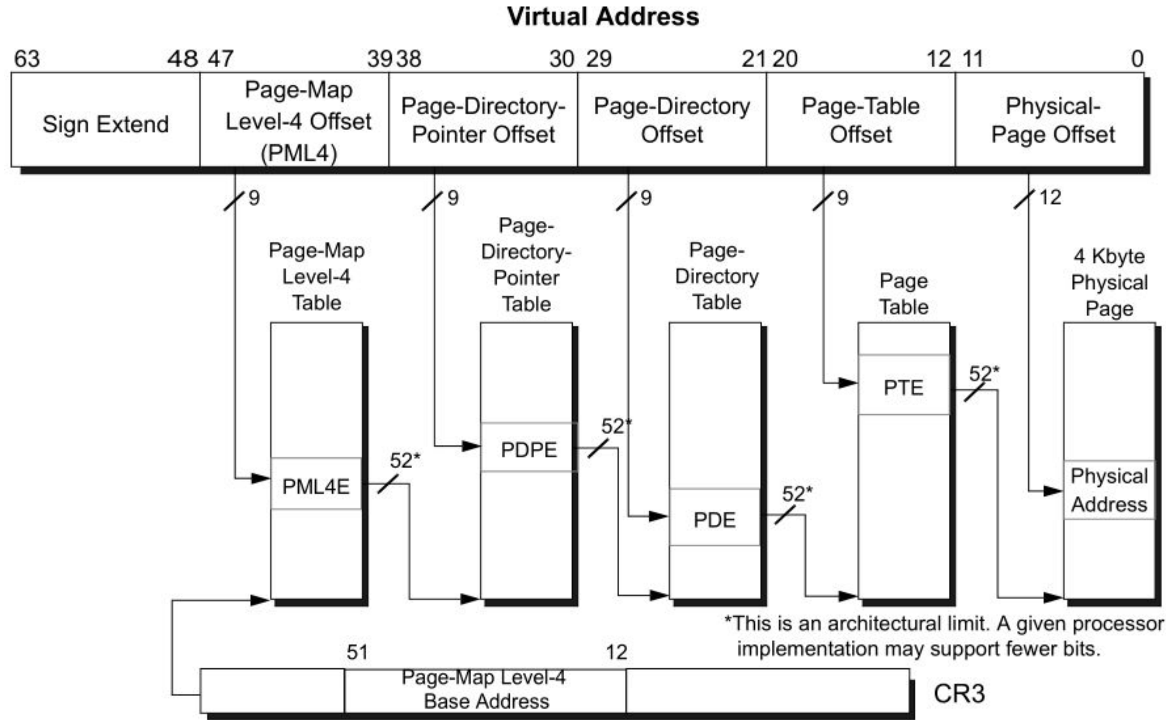
- We ignore lower 12 bits
  - 48 – 12 = 36 (total 2^36 pages)
  - 36 / 9 = 4   (each table can process 9 bits of address space)

- **We use 4-level page table**

Two-level paging (32-bit)



Four-level paging (64-bit)



More memory with increasing address space…

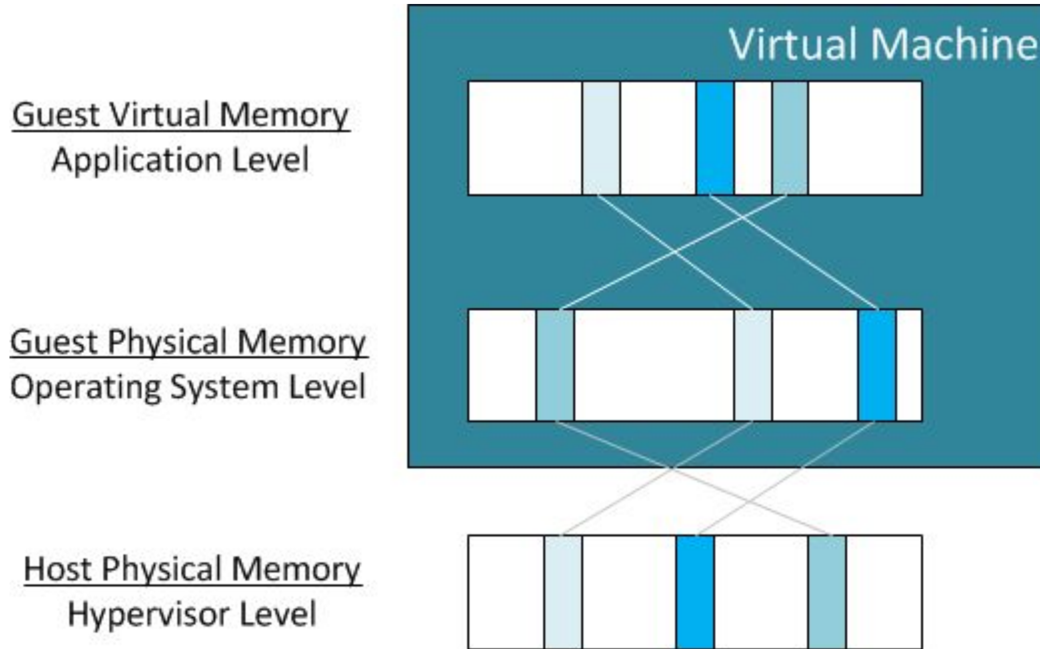
- 256 TB might not enough for big data processing
    - E.g., analyzing online social network of users in Facebook
        - More than 1 billion users, more than 1 trillion edges among users
        - 1 byte per edge = 1TB

- 4 levels, 48 bit

- 5 levels? Yes we can, 48 + 9 = 57 bits = 128PB

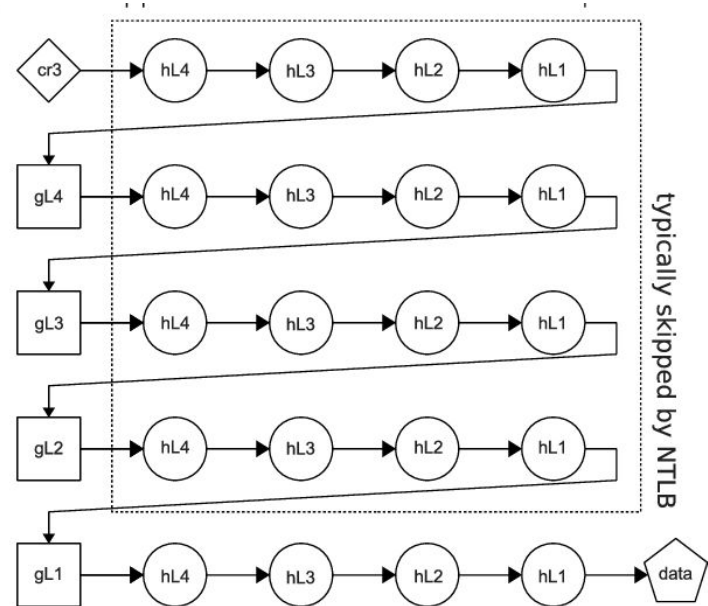
- 6 levels? Maybe, but 57 + 9 = 68 bits, out of 64 bits…

Virtual Machines - Detour



Virtual Machines - Nested Page Tables (NPE/EPE)

- A case of 48-bit 4-level page table
- Suppose you run Windows
  - Install VMWARE
  - Install a Linux VM
- Host: Windows
- Guest: Linux
- Physical address of Linux
  - This is just a virtual address of Windows

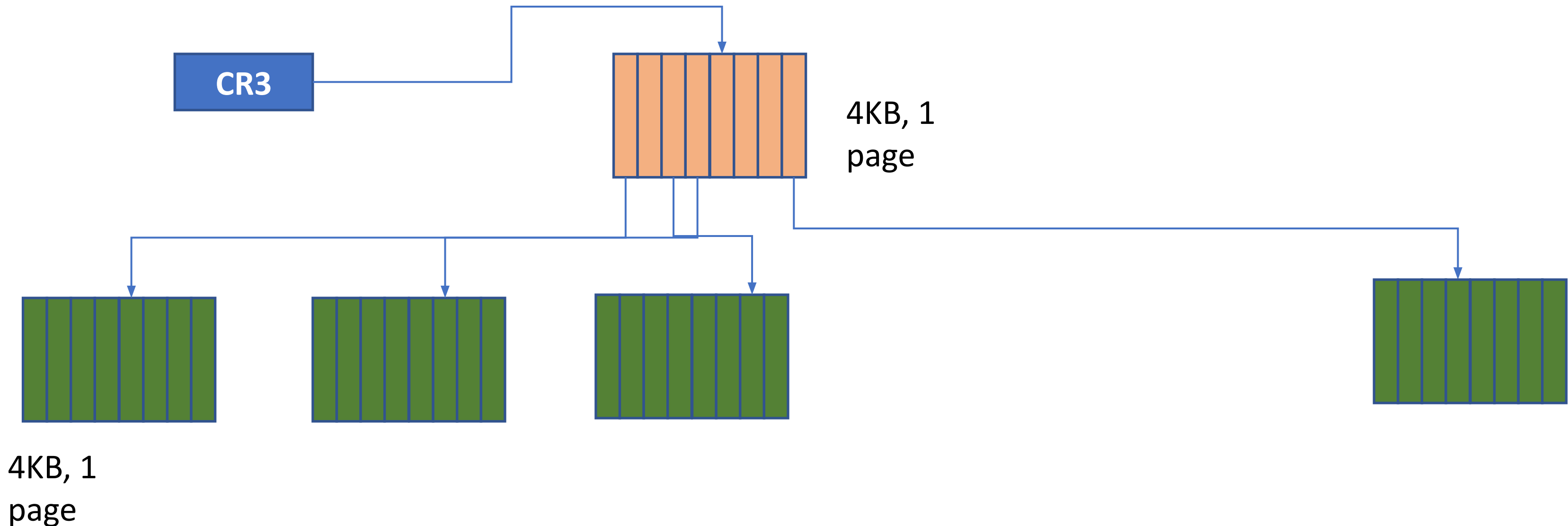


JOS Memory Management

- 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

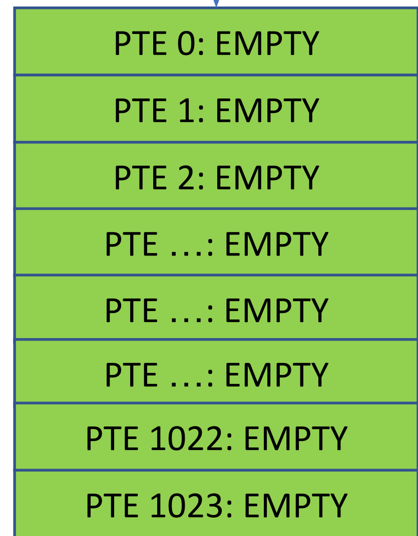
PDE 0: EMPTY
PDE 1: EMPTY
PDE 2: EMPTY
PDE …: EMPTY
PDE …: EMPTY
PDE …: EMPTY
PDE 1022: EMPTY
PDE 1023: EMPTY

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

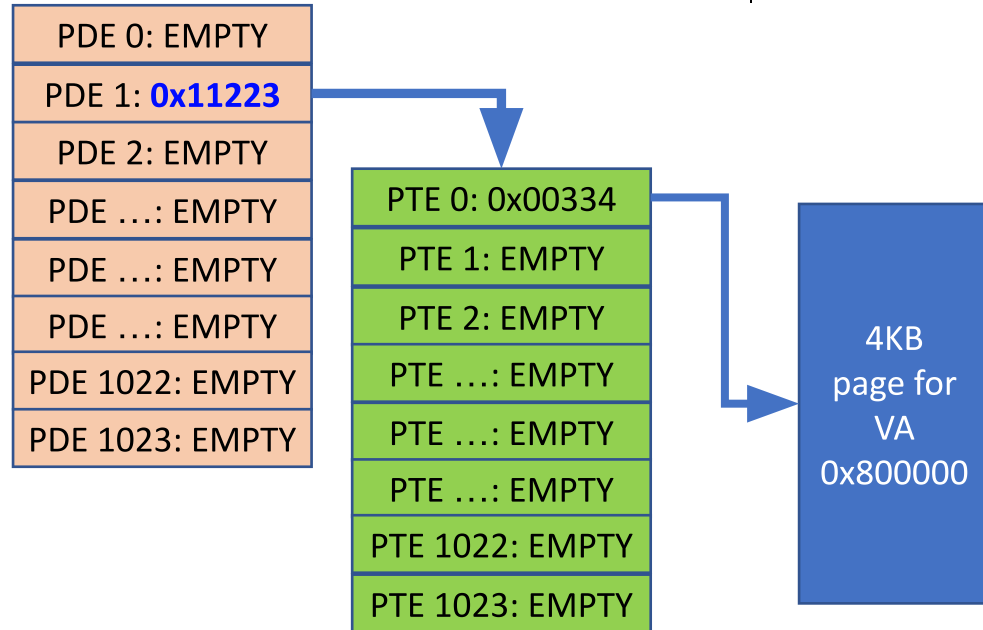
PDE 0: EMPTY
PDE 1: **0x11223**
PDE 2: EMPTY
PDE …: EMPTY
PDE …: EMPTY
PDE …: EMPTY
PDE 1022: EMPTY
PDE 1023: EMPTY



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

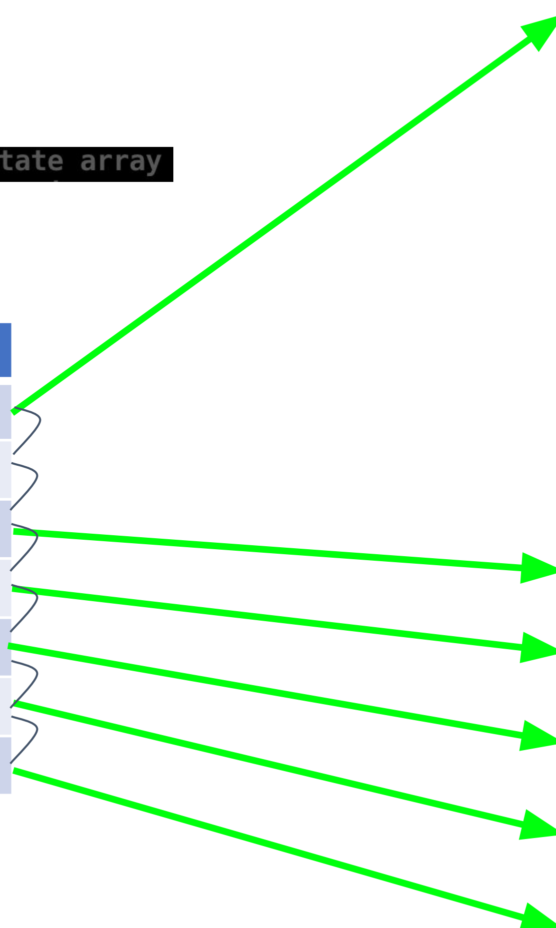
```
struct PageInfo {
    // Next page on the free list.
    struct PageInfo *pp_link;

    // 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	



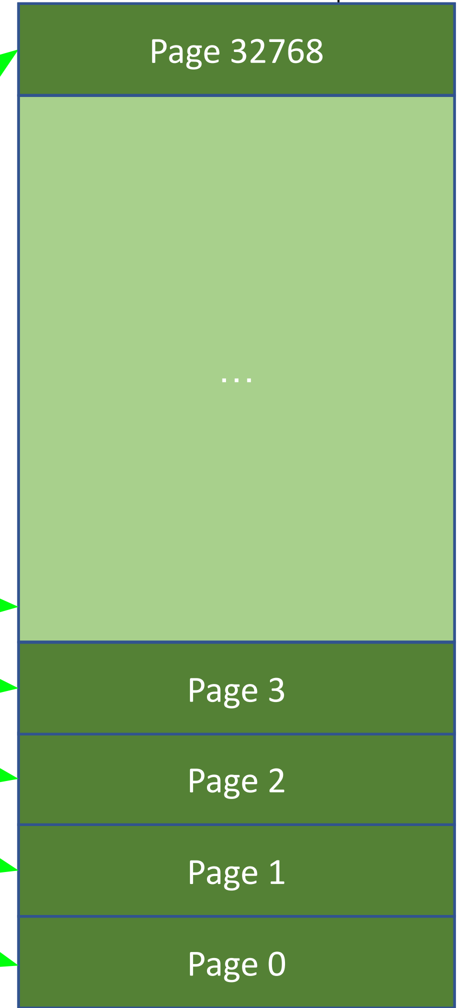
Struct PageInfo (128MB)

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	



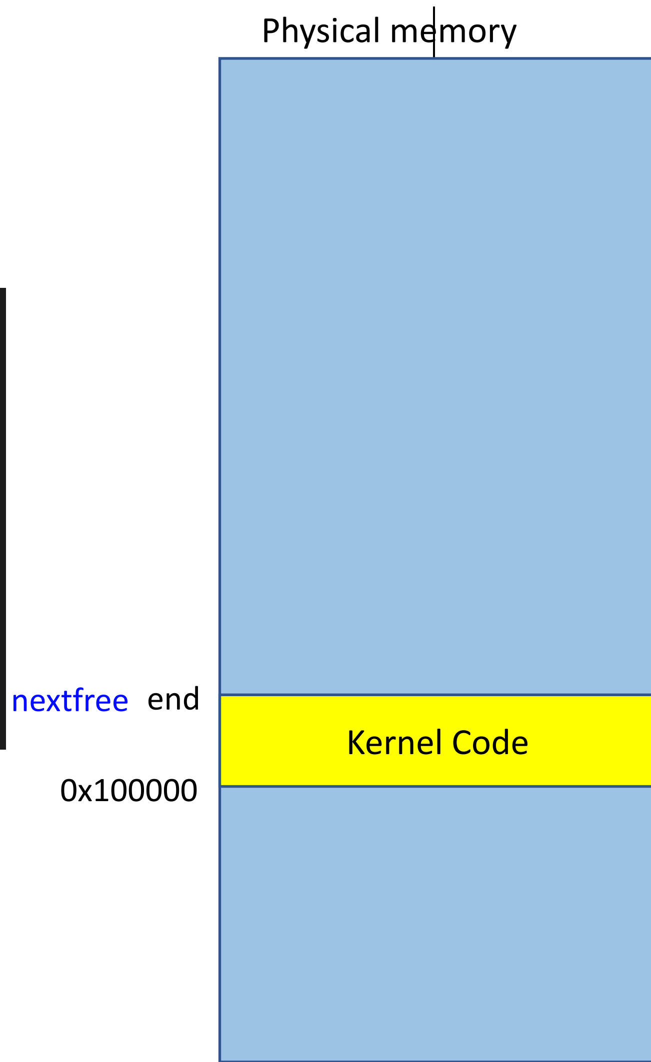
We can put this array into our physical memory

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

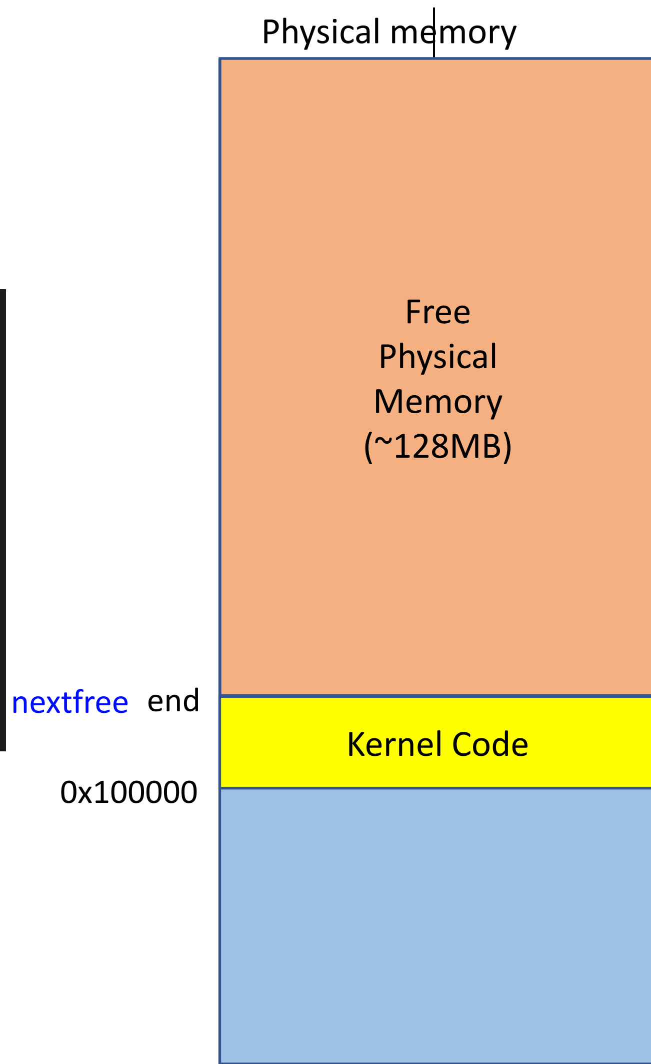


Free Physical Memory (init)

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



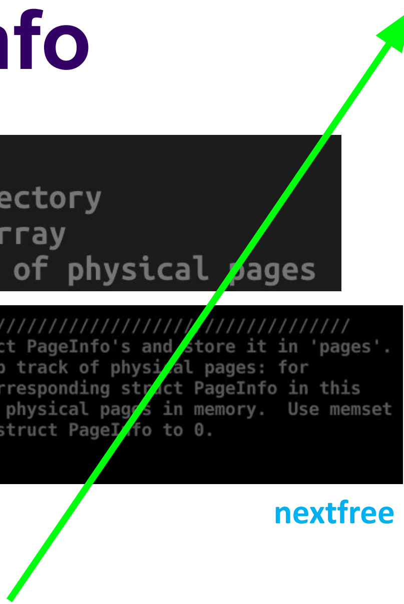
Allocating struct PageInfo

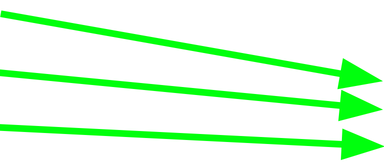
```
// These variables are set in mem_init()
pde_t *kern_pgdir;        // Kernel's initial page directory
struct PageInfo *pages;       // Physical page state array
static struct PageInfo *page_free_list; // Free list of physical pages
```

```
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	

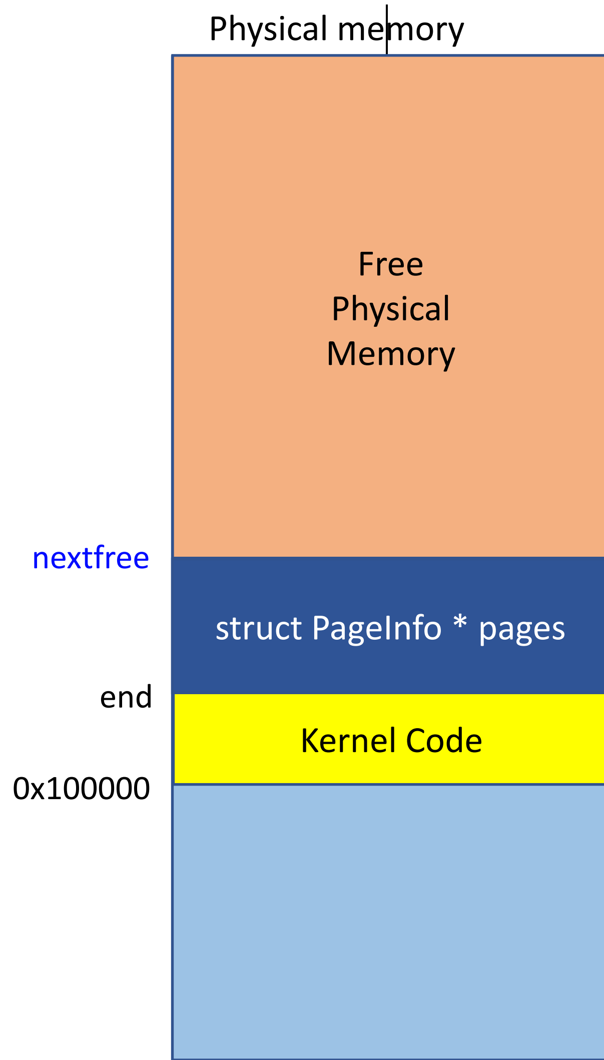




free pages!

- in page_init()

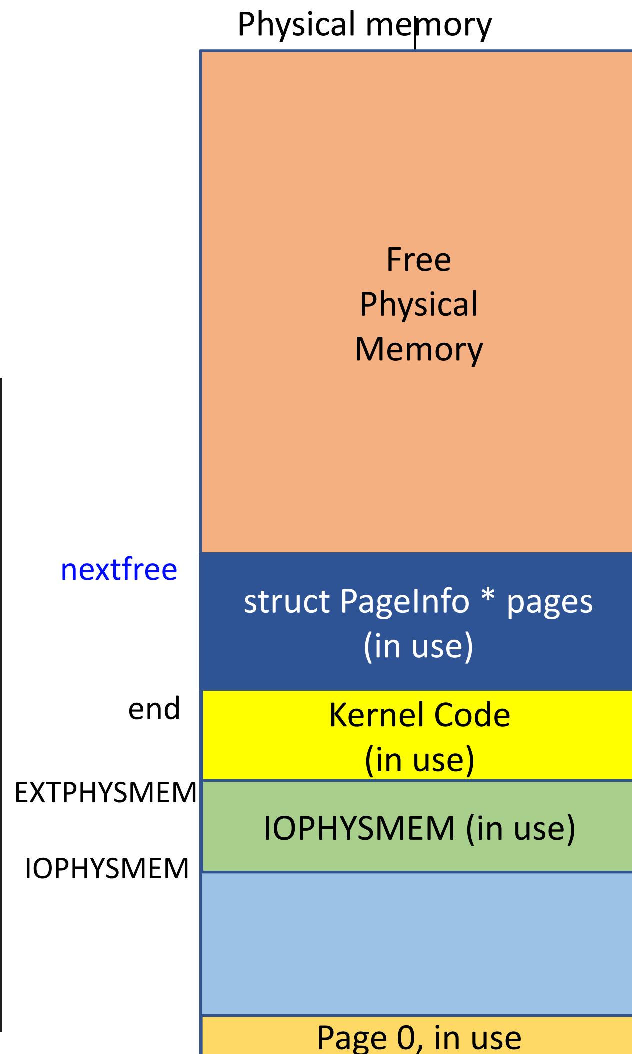
```
// The example code here marks all physical pages as free.
// However this is not truly the case.  What memory is free?
//  1) Mark physical page 0 as in use.
//     This way we preserve the real-mode IDT and BIOS structures
//     in case we ever need them.  (Currently we don't, but...)
//  2) The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

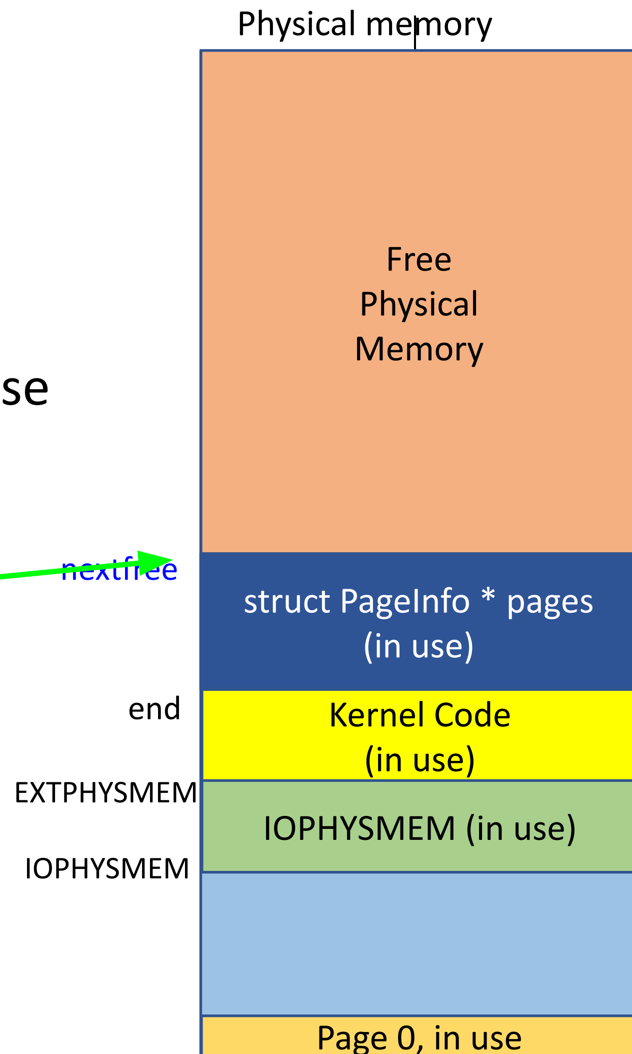
- in page_init()

```
// 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!

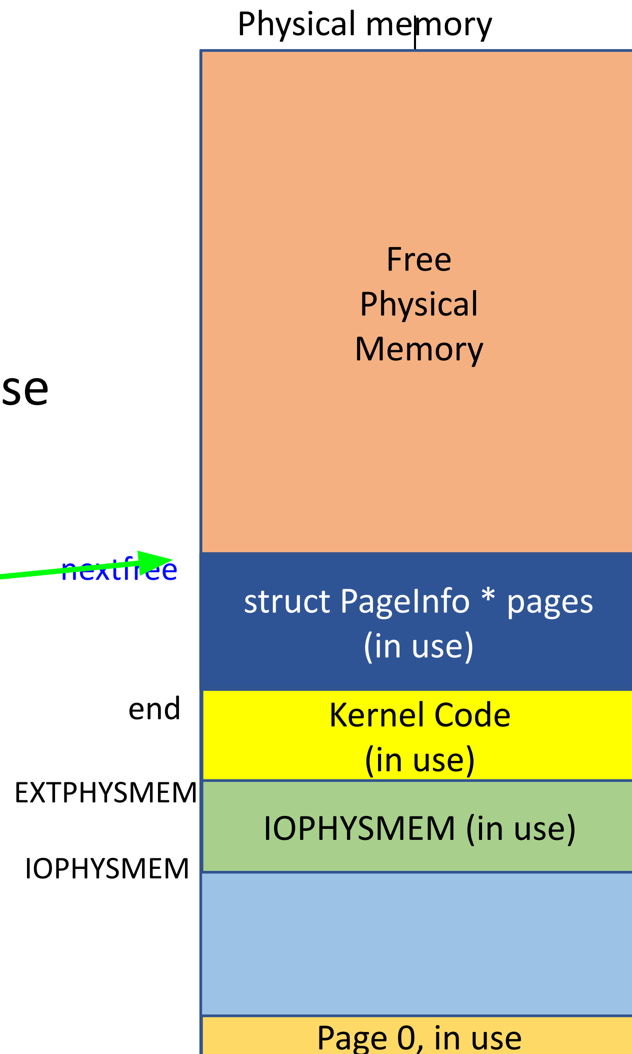
- Page 0 is in-use
- Pages in [IOPHYSMEM ~ EXTPHYSMEM] are in-use
- Pages for the kernel code are in-use
- Pages for struct PageInfo *pages are in-use
- How can you point this?
  - pages + npages ?
  - boot_alloc(0)?



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	

Physical page N

Physical page 2

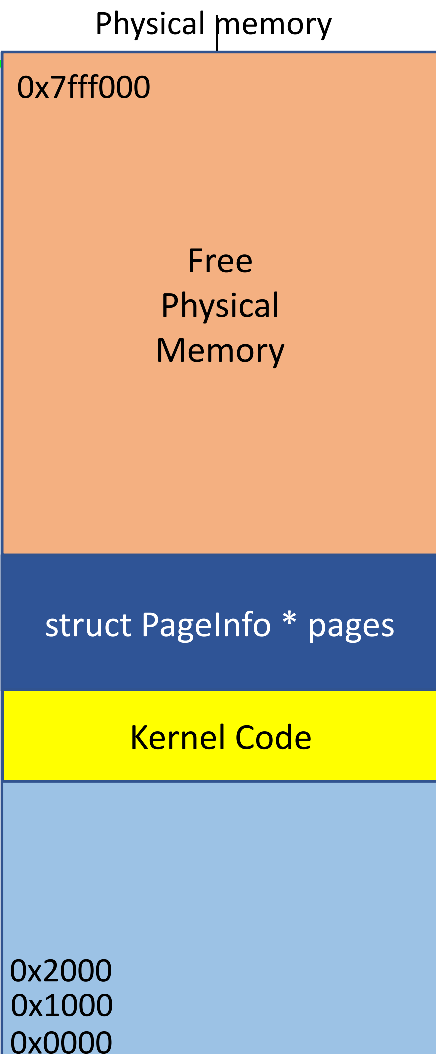
Physical page 1

Physical page 0

nextfree

end

0x100000



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];
    }
}
```

Building free list

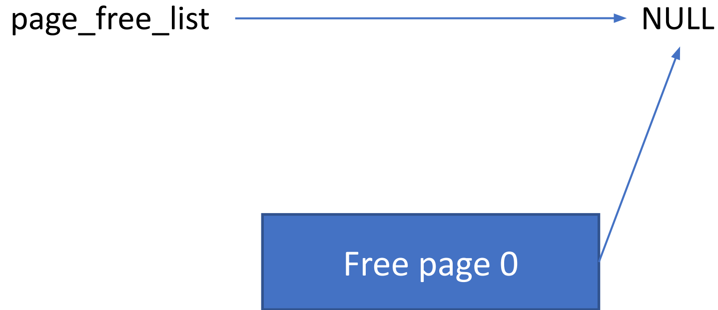
```
for (int i=0; i < npages; ++i) {
    if (pages[i].pp_ref == 0) {
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}
```

page_free_list → NULL

Free page 0

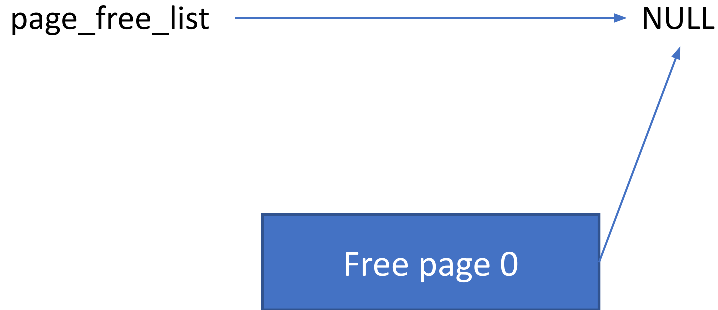
Building free list

```
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];
    }
}
```



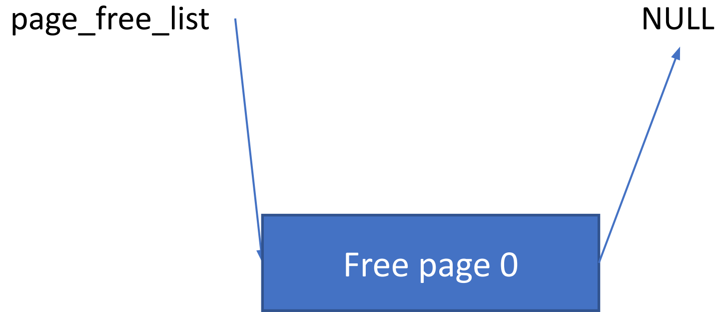
Building free list

```
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];
    }
}
```

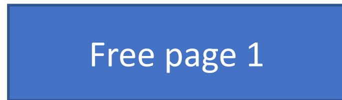


Building free list

```
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];
    }
}
```

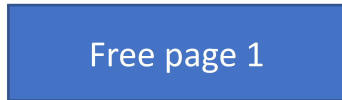
Building free list

```
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];
    }
}
```



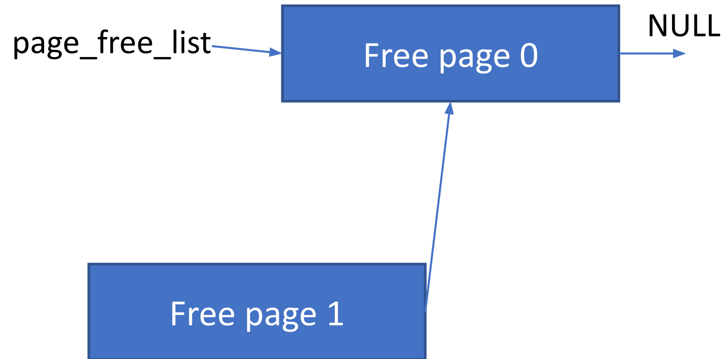
Building free list

```
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];
    }
}
```



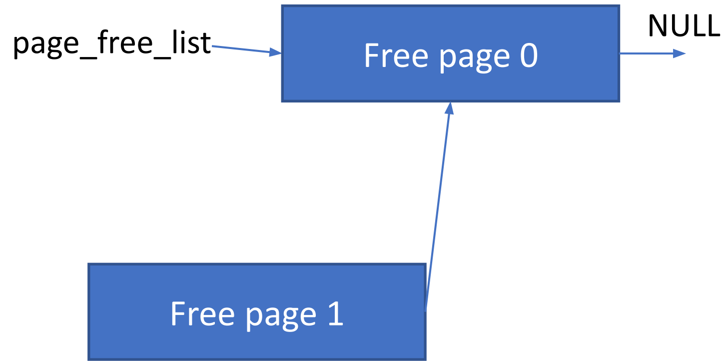
Building free list

```
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];
    }
}
```



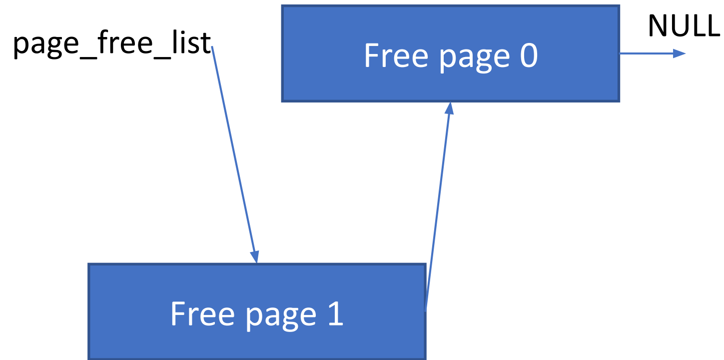
Building free list

```
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];
    }
}
```



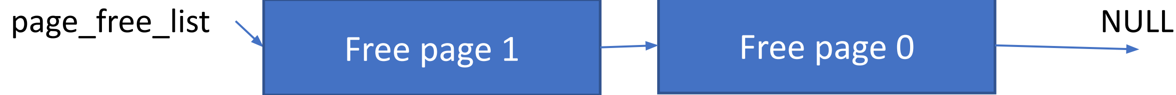
Building free list

```
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;
}
```

pp → (idx 4)

pages → (idx 0)

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)

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

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

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 kernel, in kern/entry.S)
  - C. Load kernel image from disk

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