CS 444/544 OS II
Lab Tutorial #3

Physical Memory Management for Lab 2

Today’s Topic

- Lab2, Exercise 1
  - `boot_alloc()`
  - `mem_init()`
  - `page_init()`
  - `page_alloc()`
  - `page_free()`

Before Start

- Please finish your lab1…

- After that, run the following commands

  - `git checkout lab2`

  - `git merge lab1`

Please Read inc/memlayout.h Thoroughly

```
* Virtual memory map:                                Permissions
*                                                    kernel/user
*
*    4 Gig -------->  +------------------------------+
*                     |                              | RW/--
*                     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
*                     :              .               :
*                     :              .               :
*                     :              .               :
*                     |~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~| RW/--
*                     |                              | RW/--
*                     |   Remapped Physical Memory   | RW/--
*                     |                              | RW/--
*    KERNBASE, ---->  +------------------------------+ 0xf0000000      --+
*    KSTACKTOP        |     CPU0's Kernel Stack      | RW/--  KSTKSIZE   |
*                     | - - - - - - - - - - - - - - -|                   |
*                     |      Invalid Memory (*)      | --/--  KSTKGAP    |
*                     +------------------------------+                   |
*                     |     CPU1's Kernel Stack      | RW/--  KSTKSIZE   |
*                     | - - - - - - - - - - - - - - -|                 PTSIZE
*                     |      Invalid Memory (*)      | --/--  KSTKGAP    |
*                     +------------------------------+                   |
*                     :              .               :                   |
*                     :              .               :                   |
*    MMIOLIM ------>  +------------------------------+ 0xefc00000      --+
*                     |       Memory-mapped I/O      | RW/--  PTSIZE
* ULIM, MMIOBASE -->  +------------------------------+ 0xef800000
*                     |  Cur. Page Table (User R-)   | R-/R-  PTSIZE
*    UVPT      ---->  +------------------------------+ 0xef400000
*                     |          RO PAGES            | R-/R-  PTSIZE
*    UPAGES    ---->  +------------------------------+ 0xef000000
*                     |           RO ENVS            | R-/R-  PTSIZE
* UTOP,UENVS ------>  +------------------------------+ 0xeec00000
* UXSTACKTOP -/       |     User Exception Stack     | RW/RW  PGSIZE
*                     +------------------------------+ 0xeebff000
*                     |       Empty Memory (*)       | --/--  PGSIZE
*    USTACKTOP  --->  +------------------------------+ 0xeebfe000
```

In lab2, we are setting up
the virtual memory space for JOS Kernel

This diagram will help you a lot…

Trust me, it’s your **must have item** in
one of your tmux pane..

Tool: Using `ctags`

- You can move around functions in vim by using ctags

- Initialize
  - At the top directory for JOS, run
  - `ctags -R .`

- Use
  - Open the file from the top directory, e.g., `vim kern/pmap.c`
  - Press `CTRL+]` to move
  - Press `CTRL+t` to get back

boot_alloc()

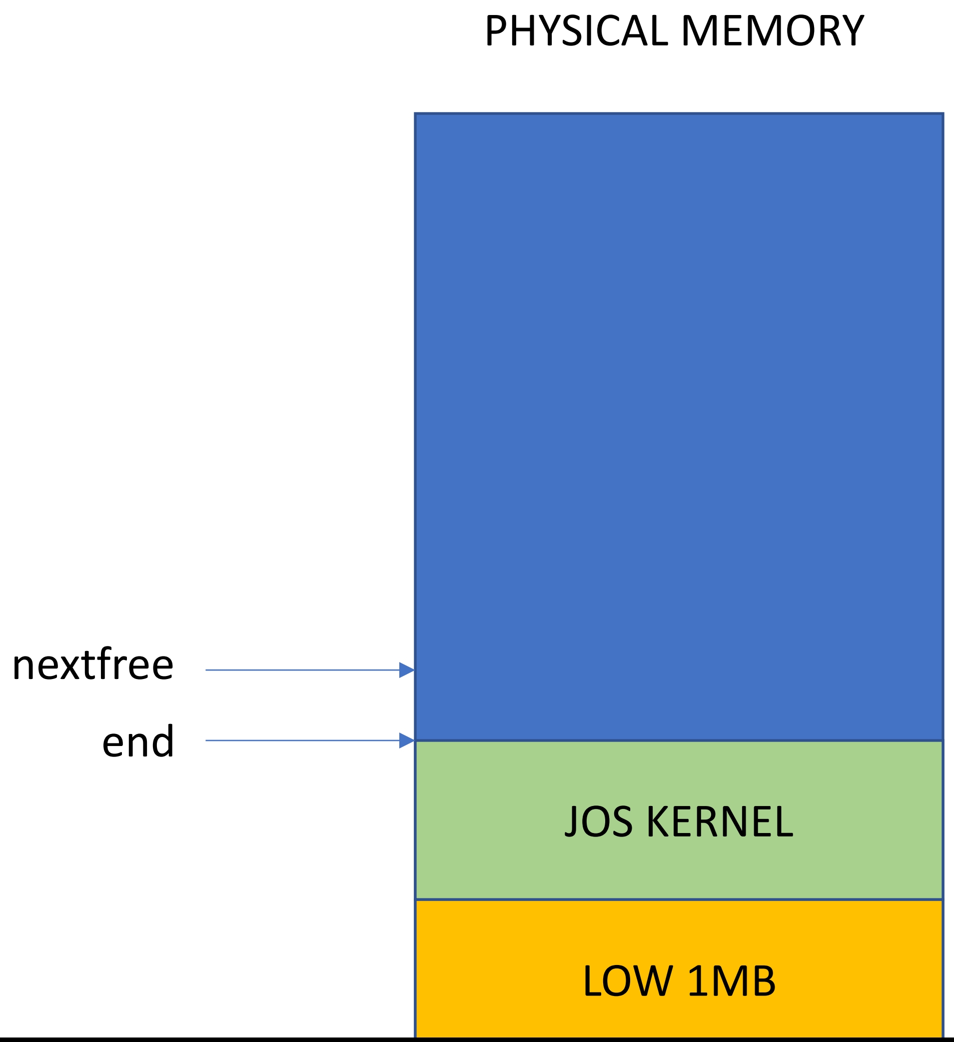
- An allocator for **physical memory**
- We will use this for bootstrapping virtual memory space in JOS
- The real allocator will be page_alloc(), which we will implement based on boot_alloc()

```
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);
	}

	// Allocate a chunk large enough to hold 'n' bytes, then update
	// nextfree.  Make sure nextfree is kept aligned
	// to a multiple of PGSIZE.
	//
	// LAB 2: Your code here.

	return NULL;
}
```

boot_alloc(); Read the Description!

```c
// This simple physical memory allocator is used only while JOS is setting
// up its virtual memory system.  page_alloc() is the real allocator.
//
// If n>0, allocates enough pages of contiguous physical memory to hold 'n'
// bytes.  Doesn't initialize the memory.  Returns a kernel virtual address.
//
// If n==0, returns the address of the next free page without allocating
// anything.
//
// If we're out of memory, boot_alloc should panic.
// This function may ONLY be used during initialization,
// before the page_free_list list has been set up.
static void *
boot_alloc(uint32_t n)
```

How boot_alloc works?

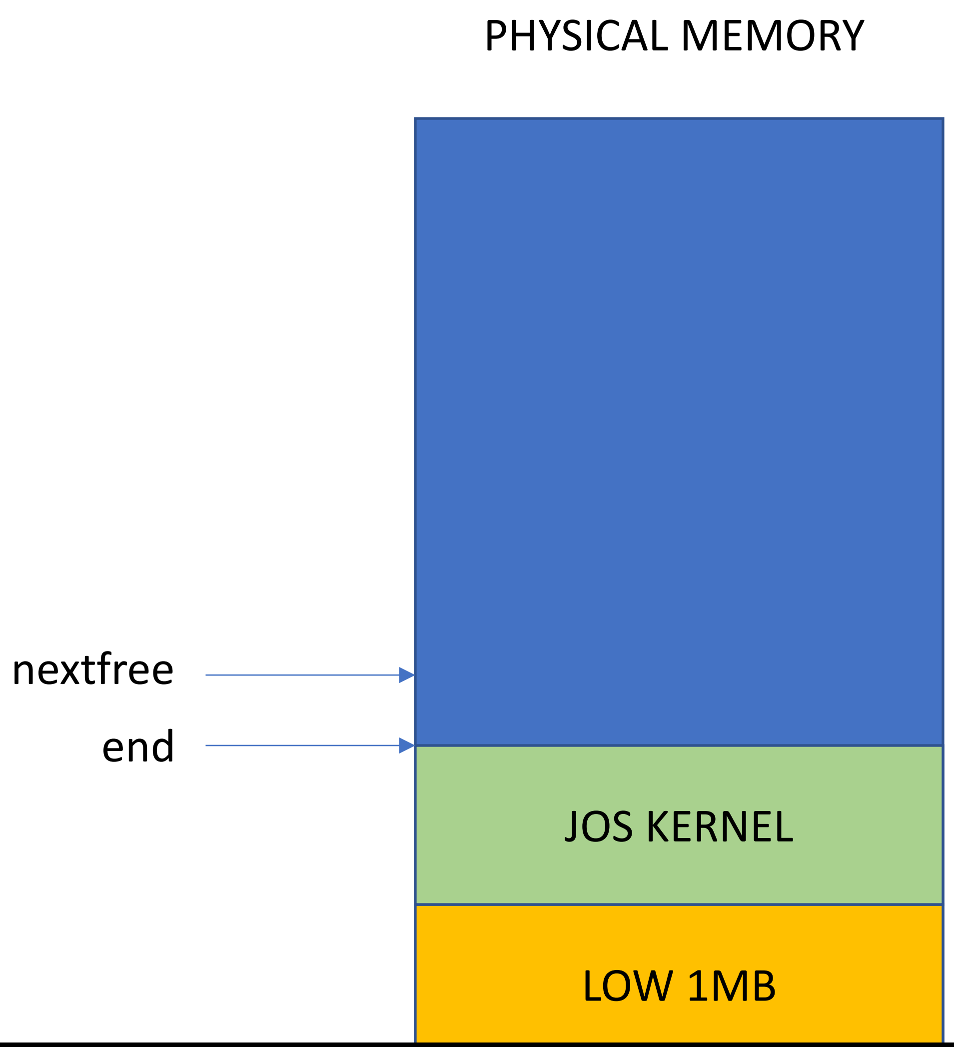


- Nextfree points to the addr
  - That is free (not used at all)
- How?
  - It first points to the ’end’ of the kernel (next page)
  - Whenever allocation request comes, move this to the next free address and return the previous value..

```
if (!nextfree) {
    extern char end[];
    nextfree = ROUNDUP((char *) end, PGSIZE);
}
```

How boot_alloc works?

- Boot_alloc INVARIANT
  - The physical memory region at above nextfree is free…

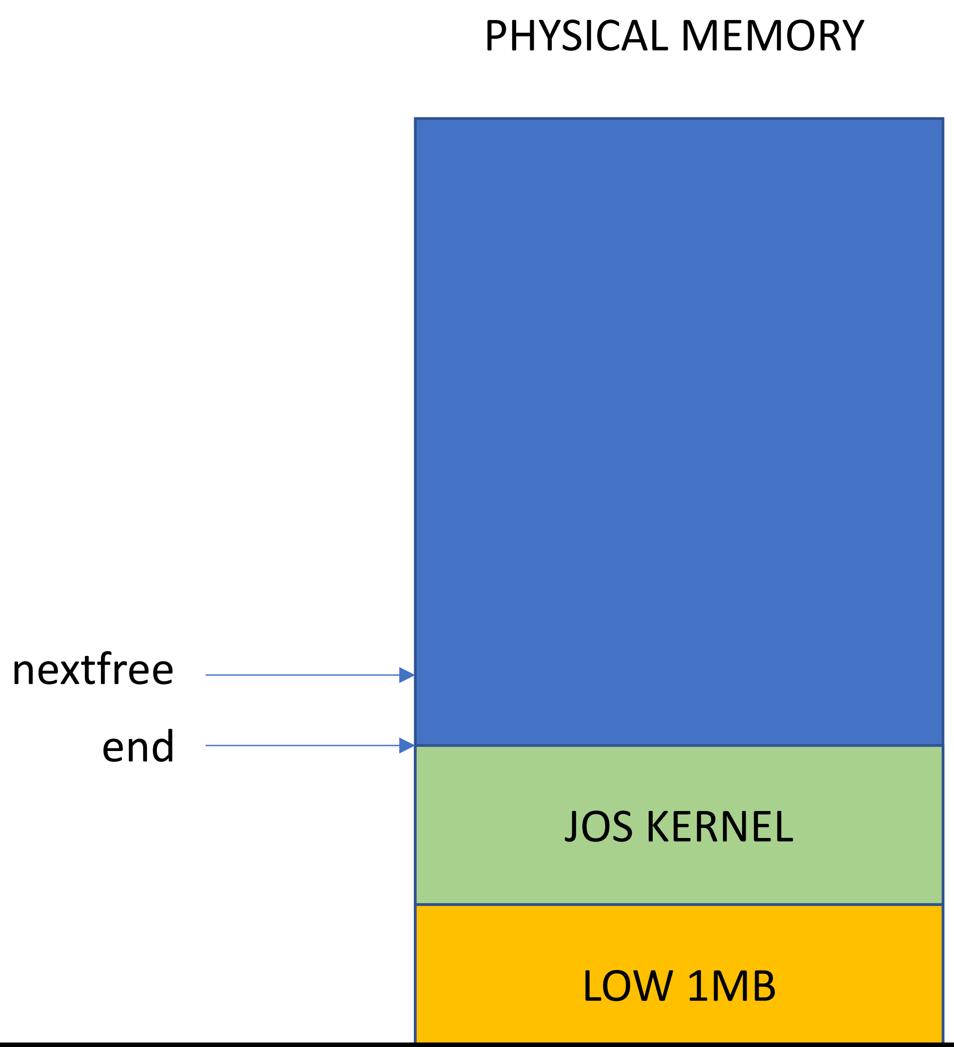



```
if (!nextfree) {
    extern char end[];
    nextfree = ROUNDUP((char *) end, PGSIZE);
}
```

How boot_alloc works?

- Nextfree points to the addr
  - That is free (not used at all)

- Requesting 16KB..

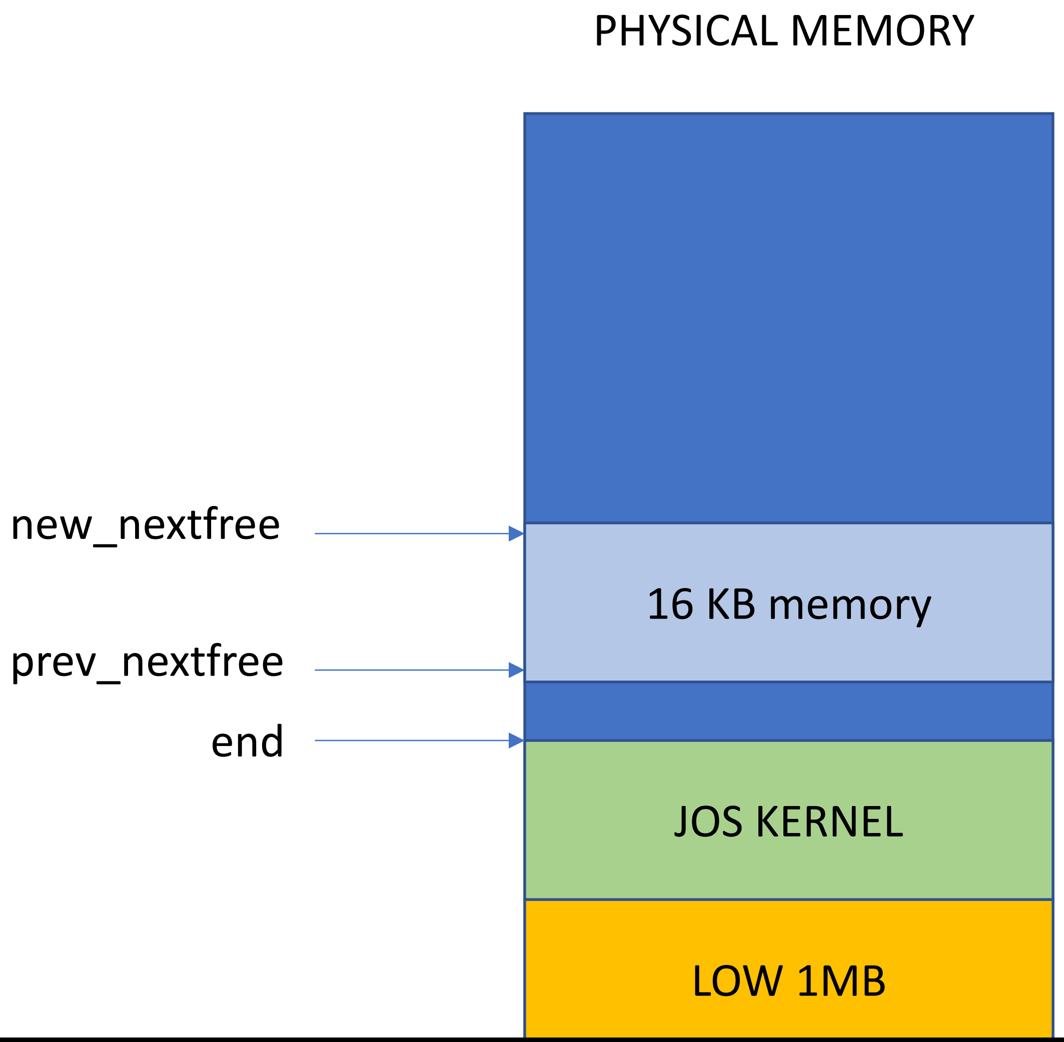



```
if (!nextfree) {
    extern char end[];
    nextfree = ROUNDUP((char *) end, PGSIZE);
}
```

How boot_alloc works?

- Nextfree points to the addr
  - That is free (not used at all)

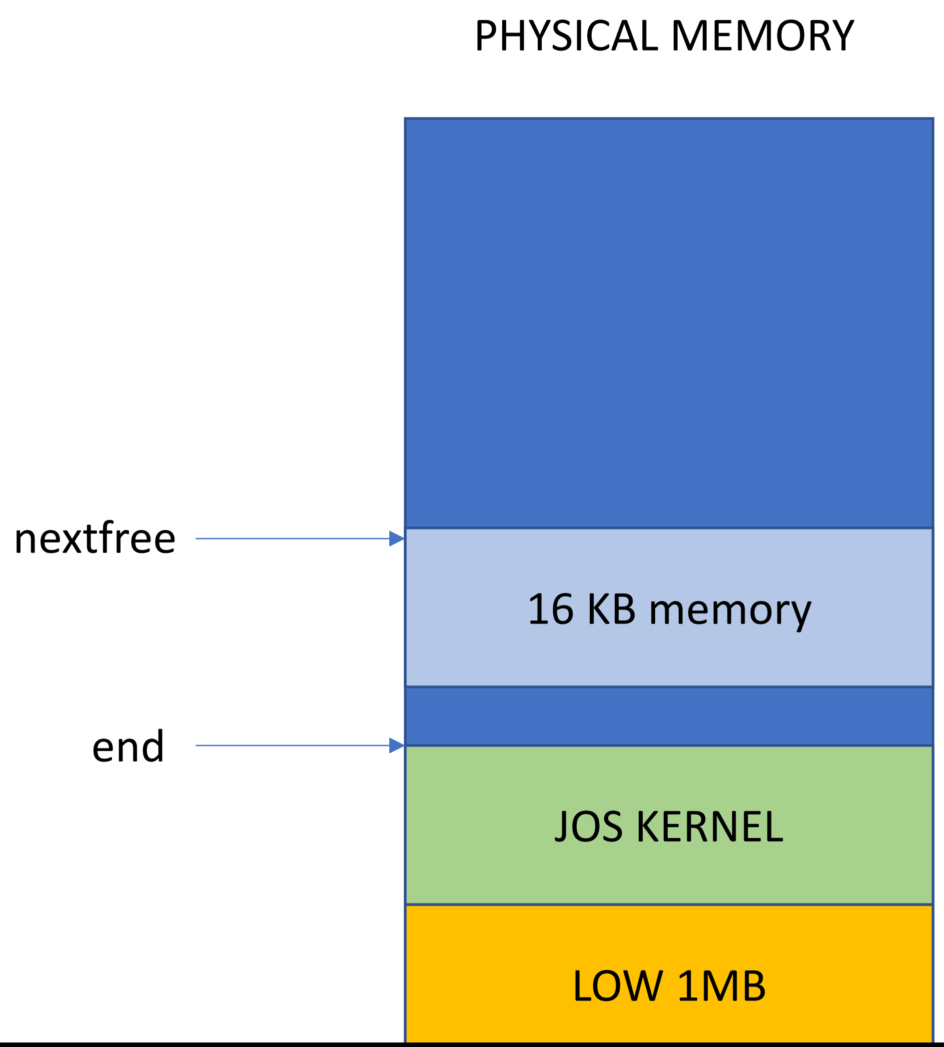
- Allocating 16KB..
  - new_nextfree =
    - ROUNDUP(nextfree + 16KB, PGSIZE)…
  - return prev_nextfree
  - update nextfree…




```
if (!nextfree) {
    extern char end[];
    nextfree = ROUNDUP((char *) end, PGSIZE);
}
```

How boot_alloc works?

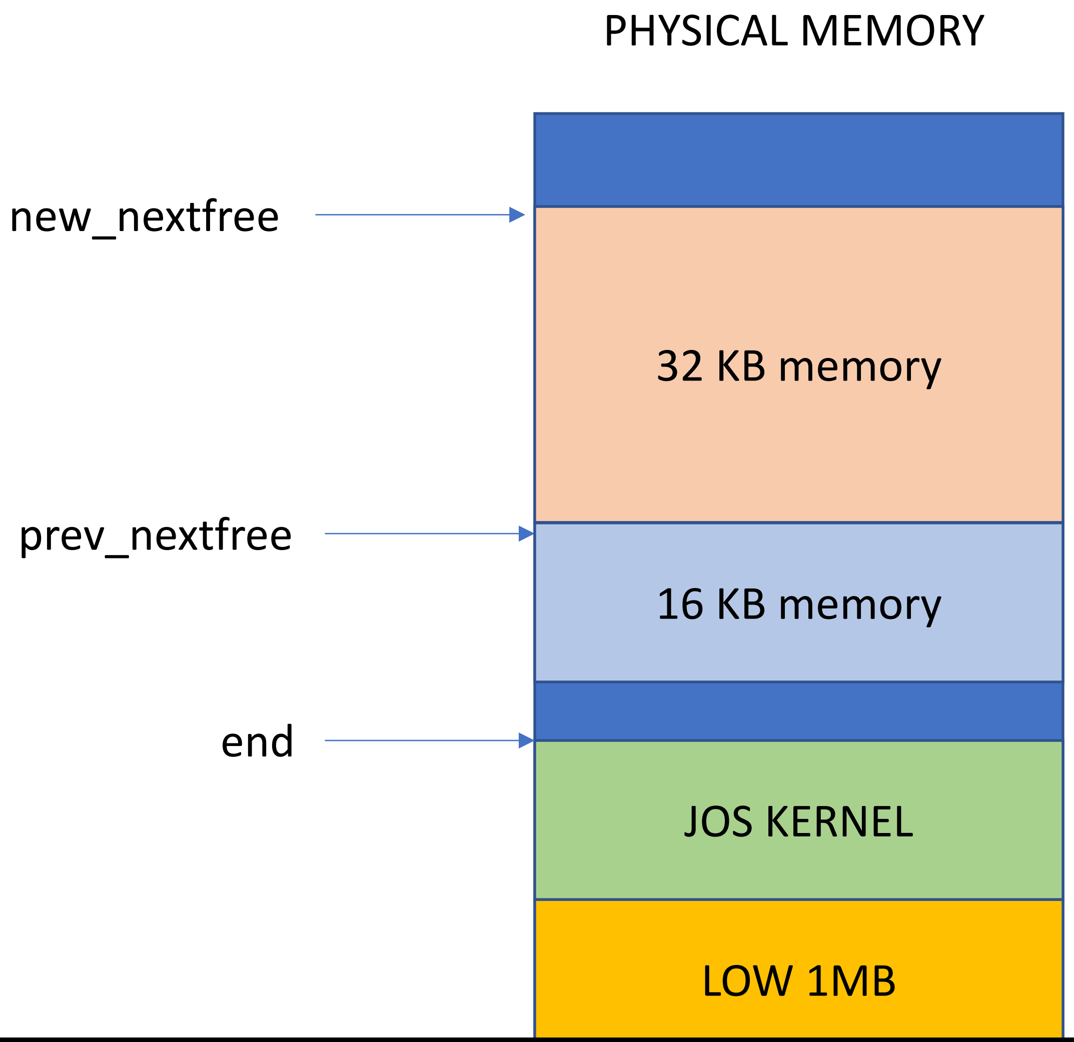
- Requesting 32KB..
  - new_nextfree =
    - ROUNDUP(nextfree + 32KB, PGSIZE)…
  - return prev_nextfree
  - update nextfree




```
if (!nextfree) {
    extern char end[];
    nextfree = ROUNDUP((char *) end, PGSIZE);
}
```

How boot_alloc works?

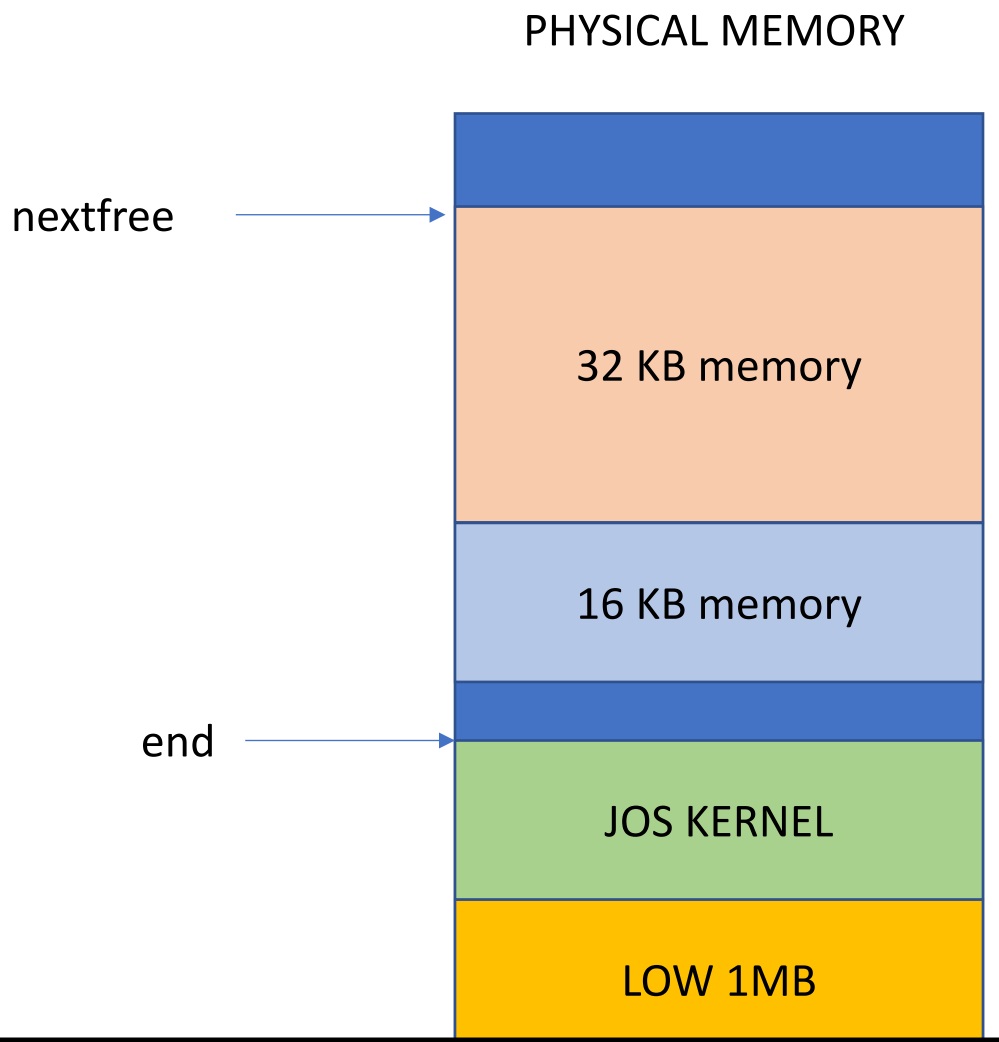
- Requesting 32KB..
  - new_nextfree =
    - ROUNDUP(nextfree + 32KB, PGSIZE)…
  - return prev_nextfree
  - update nextfree




```
if (!nextfree) {
    extern char end[];
    nextfree = ROUNDUP((char *) end, PGSIZE);
}
```

How boot_alloc works?

- Requesting 32KB..
  - new_nextfree =
    - ROUNDUP(nextfree + 32KB, PGSIZE)…
  - return prev_nextfree
  - update nextfree




```
if (!nextfree) {
    extern char end[];
    nextfree = ROUNDUP((char *) end, PGSIZE);
}
```

How can you maintain nextfree?

- **Static** in C…
  - Regard nextfree as a global variable in the function…

- 3 different meaning of static
  - 1. **global variable in a function**
  - 2. a function/variable accessible within a file
  - 3. class static function (C++)

```
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);
    }

    // Allocate a chunk large enough to hold 'n' bytes, then update
    // nextfree.  Make sure nextfree is kept aligned
    // to a multiple of PGSIZE.
    //
    // LAB 2: Your code here.

    return NULL;
}
```

mem_init()

- TODO
  - Allocate ‘pages’, an array of struct PageInfo
  - Remove the panic line below…

```
    // Remove this line when you're ready to test this function.
    panic("mem_init: This function is not finished\n");
```

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

mem_init()

- TODO
  - Allocate ‘`pages`’, an array of `struct PageInfo`
  - We need to have a corresponding `struct PageInfo` per each physical page
- Hint
  - Use boot_alloc to allocate ‘`pages`’
  - Total number of physical pages: `npages`
  - Size of memory to allocate: `npages * sizeof(struct PageInfo)`

  - How to initialize that with 0?
    - `memset(pages, 0, size…)`

How JOS manages free Phys mem?

- `struct PageInfo`
  - JOS will have a struct PageInfo entry per each physical pages..
  - 128MB (134,217,728 bytes) of available physical memory for now
  - 32768 entries (134217728 / 4096 = 32768)

```
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;
};
```

How JOS manages free Phys mem?

- page_free_list: Linked list of free physical pages

```
static struct PageInfo *page_free_list; // Free list of physical pages
```

```
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;
};
```

How JOS manages free Phys mem?

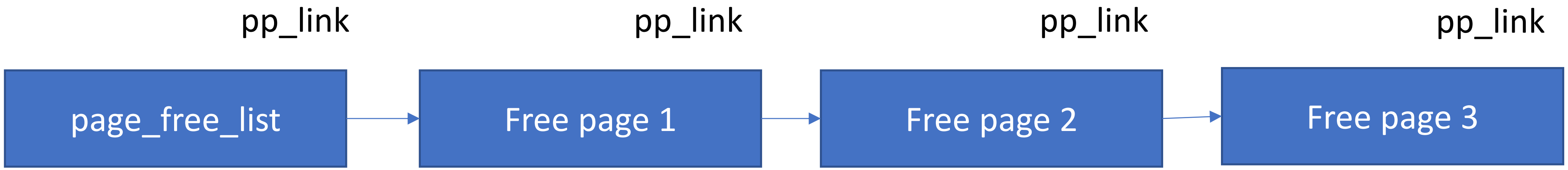
- pp_link: indicates a pointer to the PageInfo of the next free page

```
static struct PageInfo *page_free_list; // Free list of physical pages
```

```
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;
};
```

How JOS manages free Phys mem?

- pp_link: indicates a pointer to the PageInfo of the next free page
- Implement this for page_init



How to build a linked list?

- 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];
    }
}
```

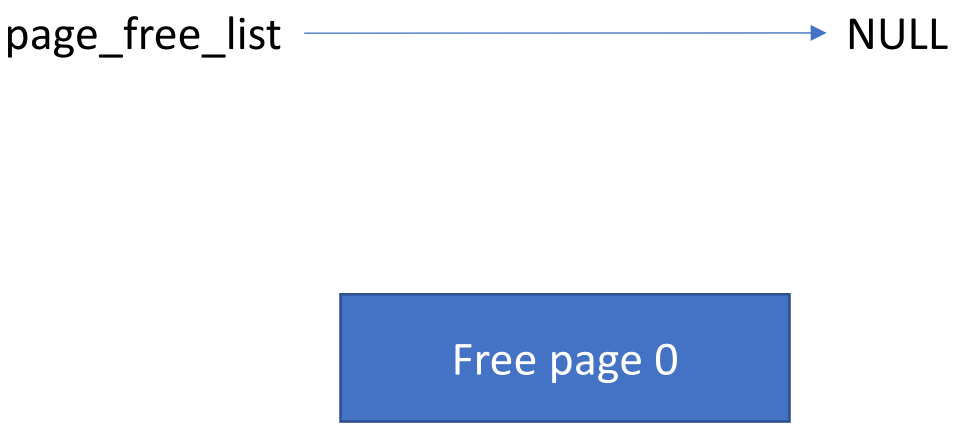
List Building

```
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];
    }
}
```

page_free_list → NULL

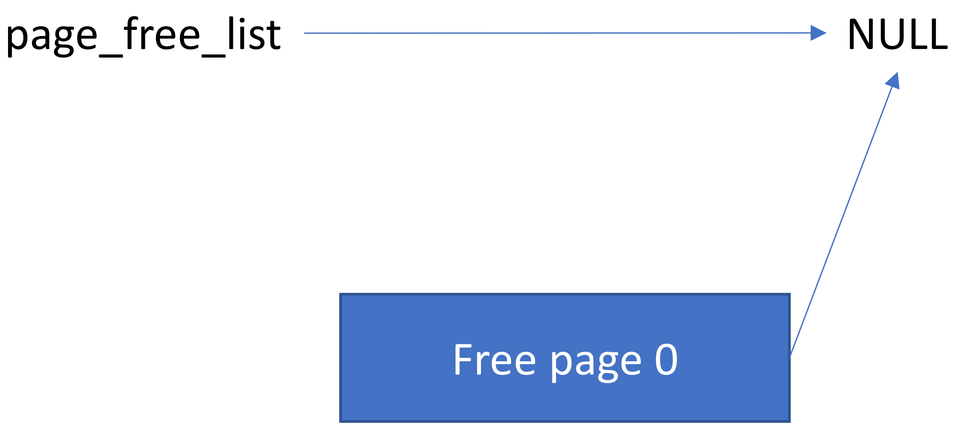
List Building

```c
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];
    }
}
```



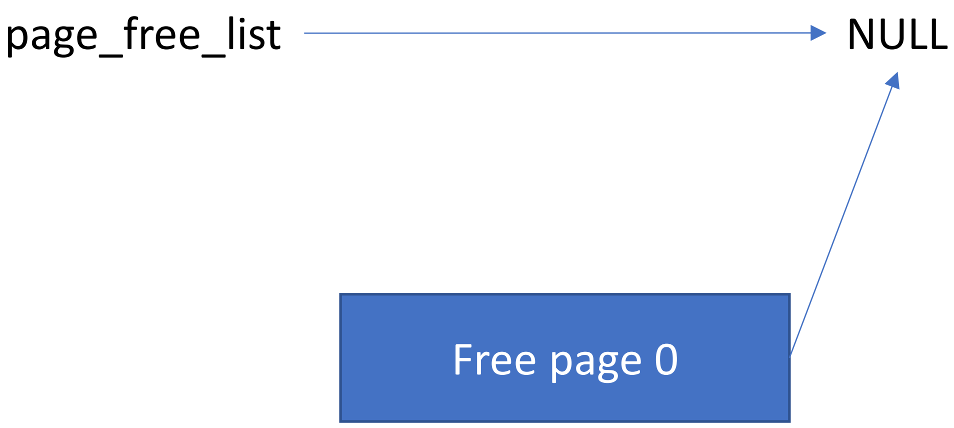
List Building

```
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];
    }
}
```



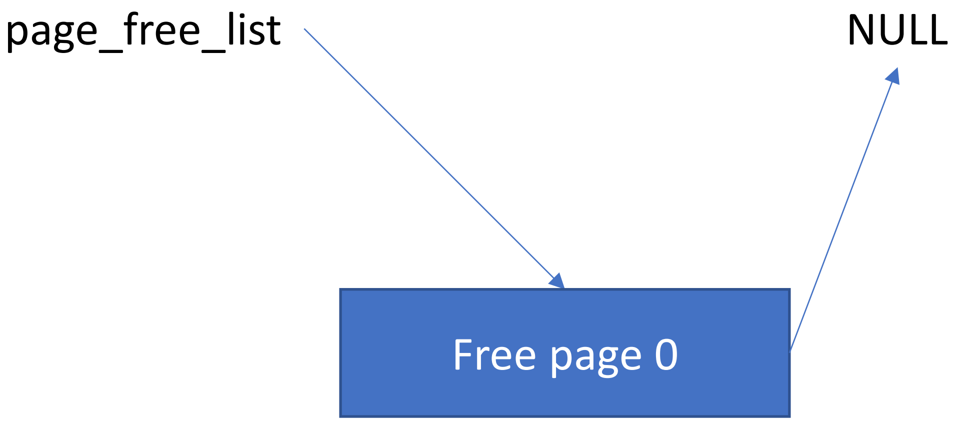
List Building

```
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];
    }
}
```

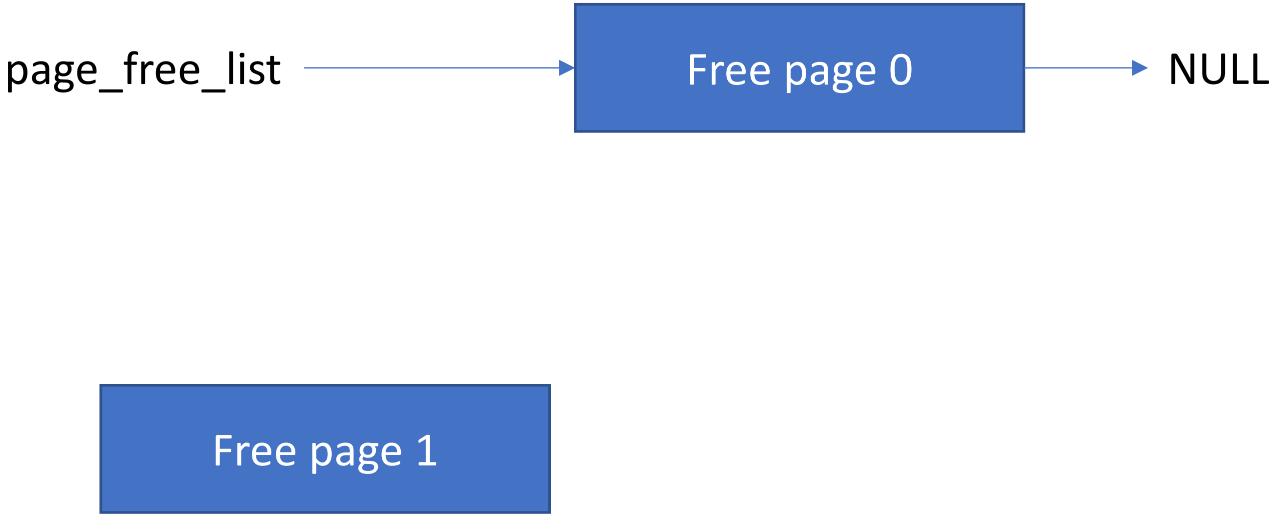


List Building

```
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];
    }
}
```

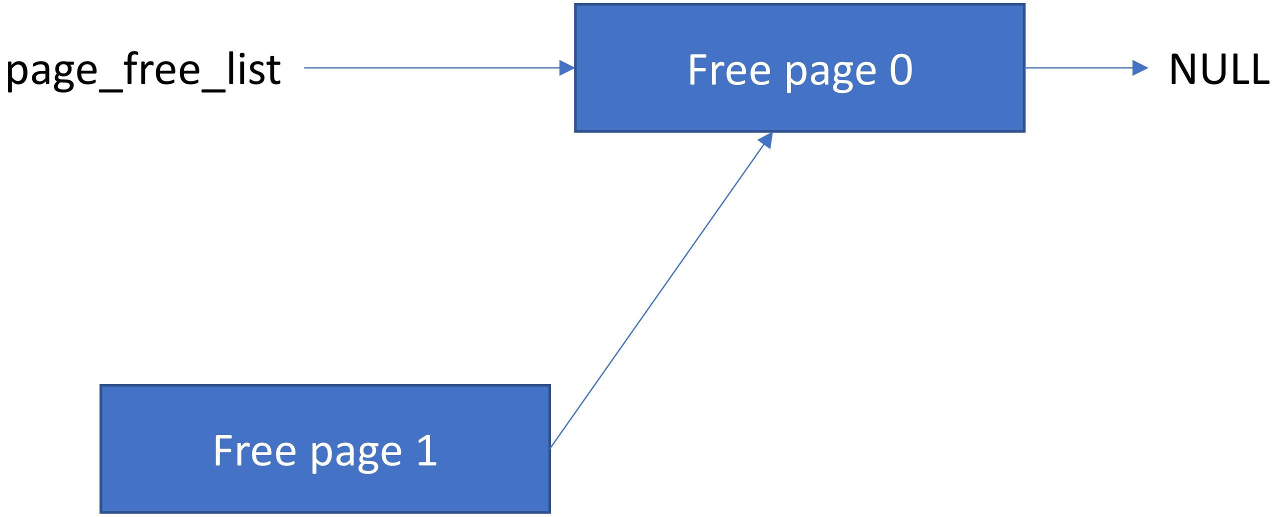
List Building

```
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];
    }
}
```



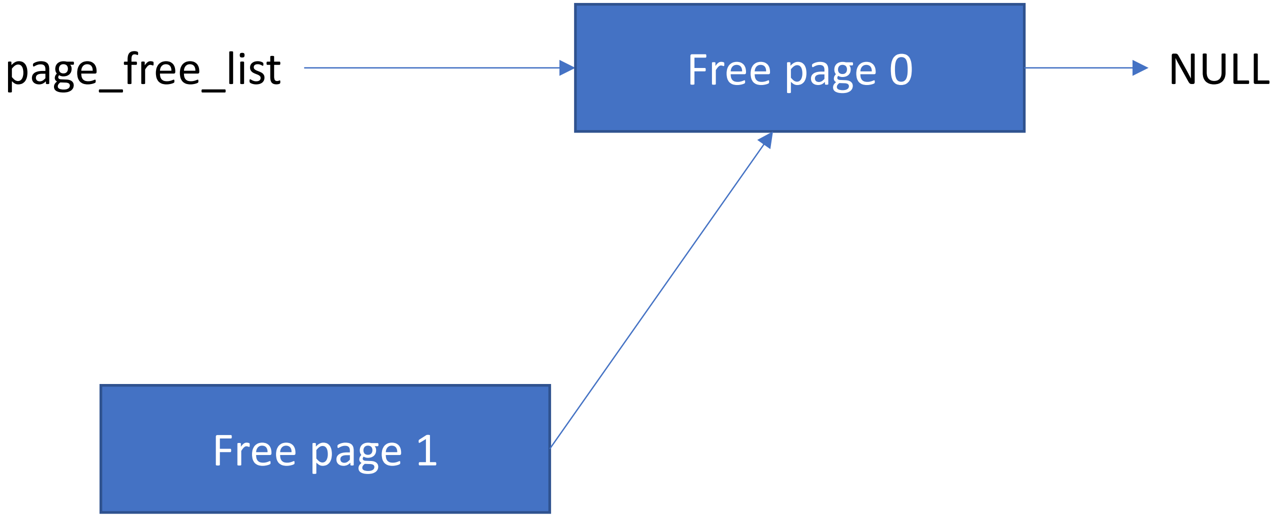
List Building

```
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];
    }
}
```



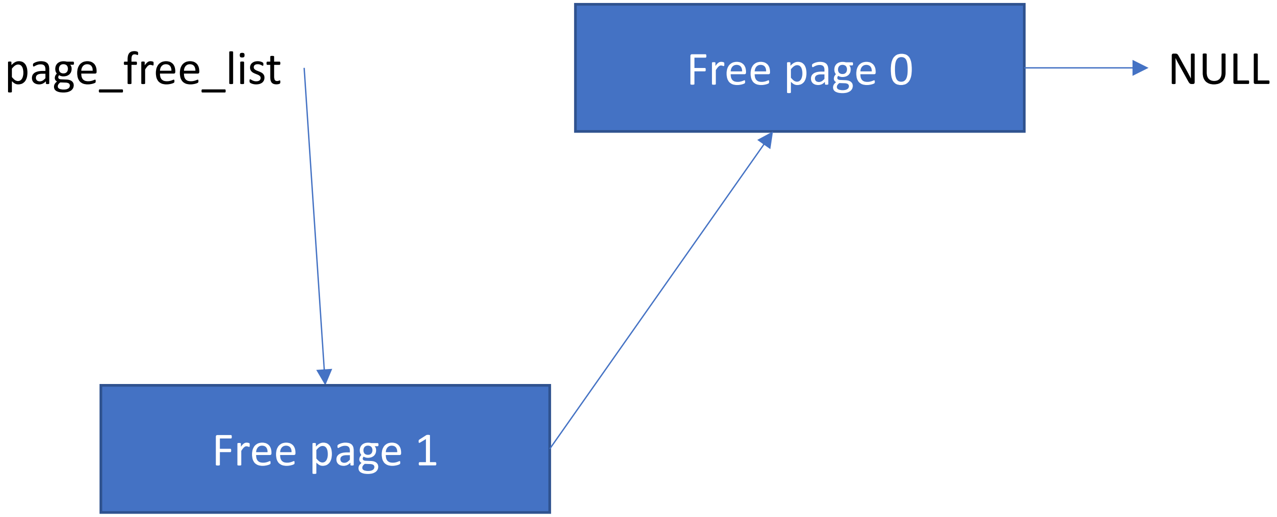
List Building

```
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];
    }
}
```



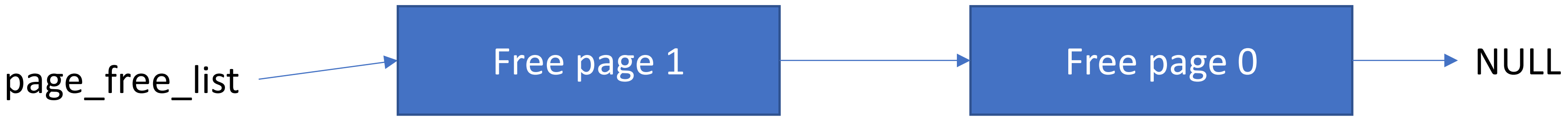
List Building

```
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];
    }
}
```



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];
    }
}
```

How JOS manages free Phys mem?

- pp_ref: indicates how many virtual allocation was made
  - pp_ref != 0 // page is being used
  - pp_ref == 0 // page is not being used at all – 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;
};
```

page_init()

- Mark some pages IN_USE
  - By setting pages[i].pp_ref = 1

- Build a linked list for
  - free physical pages

```
// 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!
```

Allocating a page (va->pa)

- Get a page from page_free_list (let this be new_page)
  - Must disconnect this from the list if allocated…
- Set page_free_list = new_page->pp_link
  - Maintain page_free_list

- page_alloc() does this

- Assigning PTE (done by other functions that uses page_alloc)
  - new_page->pp_ref += 1 (DO NOT DO THIS within PAGE_ALLOC)
  - Set page table entry (va -> pa)
    - Store it to page directory
    - invalidate TLB

Releasing a page

- page->pp_ref -= 1;
- if(page->pp_ref == 0) {
  - Add it to the page_free_list
    - page->pp_link = page_free_list;
    - page_free_list = page;
- }

- page_free() does this

Useful MACROs

- PGNUM(x)
  - Get the page number (x >> 12) of the address x
- PDX(x)
  - Get the page directory index (top 10 bits) of the address x
- PTX(x)
  - Get the page table index (mid 10 bits) of the address x
- PGOFF(x)
  - Get the page offset (lower 12 bits) of the address x
- PTE_ADDR(x)
  - Get the physical address pointed by an entry x
  - i.e., erasing all the flags (lower 12 bits), `x&0xfffff000`

Useful MACROs

- KADDR(pa)
  - Convert a physical address pa to a kernel virtual address
  - i.e., returns pa + KERNBASE
- PADDR(va)
  - Convert a kernel virtual address va to a physical address
  - i.e., returns va - KERNBASE
- Only works for kernel virtual memory
  - i.e., memory address range from 0xf0000000 to 0xffffffff

Useful MACROs

- page2kva(page)
  - Get the kernel virtual address of the page (struct PageInfo *)
  - E.g., physical_address_of_the_page + KERNBASE, 0xf???????
- Page2pa(page)
  - Get the physical address of the page (struct PageInfo *)
  - page2kva(page) == KADDR(page2pa(page))
- pa2page(pa)
  - Get the struct PageInfo *page that stores information about the pa
- To get the pa for kva?
  - pa2page(PADDR(kva))

Type and Casting

- uintptr_t
  - Used for indicating a virtual address (uint32_t)
  - Can be accessed in your C code
- physaddr_t
  - Used for indicating a physical address (uint32_t)
  - Cannot be accessed in your C code (must be converted by KADDR)
- (void *), (char *)
  - Pointers, virtual address

Type and Casting

- How to use uintptr_t va?
  - (void *)va
  - char * c = (char *)va; c[0];
  - int *i = (int *)va; i[0];
  - struct PageInfo *pp = (struct PageInfo *) va;
- How to use physaddr_t pa?
  - We can't access physical address directly; use KADDR(x)
    - KADDR(x) adds KERNBASE to x
  - char *c = (char *c) KADDR(pa); c[0];
  - struct PageInfo *pp = (struct PageInfo *) KADDR(pa);