CS1217 - Spring 2023 - Lab 3

Gautam Ahuja, Nistha Singh

Exercise 1:

We first go through the files `inc/pmap.h`, `inc/mmu.h` and `inc/memlayout.h` and analyse the memory layout and structure. We find that:

- All physical memory mapping starts from `KERNBASE` at `0xF0000000`.

```c
// All physical memory mapped at this address
#define KERNBASE    0xF0000000
```

- The information related to Page directory and page table is stored in `mmu.h`

```c
// Page directory and page table constants.
#define NPDENTRIES  1024        // page directory entries per page directory
#define NPTENTRIES  1024        // page table entries per page table

#define PGSIZE      4096        // bytes mapped by a page
```

- The information regarding the kernel stack is stored in `memorylayout.h`

```c
// Kernel stack.
#define KSTACKTOP   KERNBASE
#define KSTKSIZE    (8*PGSIZE)          // size of a kernel stack
#define KSTKGAP     (8*PGSIZE)          // size of a kernel stack guard
```

Editing `boot_alloc()` function:

Here addresses are of type `char` and are declared as follows:

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

In above code, `ROUNDUP` is a macro function defined in `types.h`. It is used to round up the number a to the nearest multiple of n.

```c
// Round up to the nearest multiple of n
#define ROUNDUP(a, n)                       \
({                                          \
    uint32_t __n = (uint32_t) (n);          \
    (typeof(a)) (ROUNDDOWN((uint32_t) (a) + __n - 1, __n)); \
})
```

Now we need to define conditions as mentioned in comments as follows:

```c
// kern > pmap.c > boot_alloc(uint32_t)
	// LAB 2: Your code here.
	// If n>0 allocate enough pages of contiguous physical memory to hold 'n' bytes.
	result = nextfree;
	if (n > 0) {
		nextfree = ROUNDUP(nextfree + n, PGSIZE);
		if (((uint32_t) nextfree - KERNBASE) > (npages * PGSIZE)) {
			panic("Panic! boot_alloc() failed! Out of memory!");
		}
	}
	// If n==0, return nextfree which is the address of the next free page
	return result;
}
```

Here, in case of `n==0`, we just return the `nextfree` address in type of `(void *)` since it is the function type.
Else, we allocate n free pages (rounded up to `PGSIZE`). If the allocation goes out of memory, i.e., allocated page is out of bounds, we do a `panic` and exit. Since function `boot_alloc()` is a `static void` type, we return a `NULL`.

Editing `mem_init()` function:

For this we only had to edit at one place before the call to `check_page_free_list(1)`. As per given comments we allocate an array of `npages` and store it in a variable `pages` of type `struct PageInfo *`. We use the function `boot_alloc()` to allocate the memory (it is a simple physical memory allocater used only while JOS is setting up its virtual memory system).
Then we use `memset()` to initialize all fields of each struct `PageInfo` to 0.

```c
// kern > pmap.c > mem_init(void)
	//////////////////////////////////////////////////////////////////////
	// 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 = (struct PageInfo *)boot_alloc(npages*sizeof(struct PageInfo));
	memset(pages, 0, npages*sizeof(struct PageInfo));
```

The `memset()` is a function defined in `string.c`. It makes some difference on real hardware. Its definition contains assembly instructions.

```c
// lib > string.c > memmove(void *, const void *, size_t)
void *
memset(void *v, int c, size_t n)
{
	char *p;

	if (n == 0)
		return v;
	if ((int)v%4 == 0 && n%4 == 0) {
		c &= 0xFF;
		c = (c<<24)|(c<<16)|(c<<8)|c;
		asm volatile("cld; rep stosl\n"
			:: "D" (v), "a" (c), "c" (n/4)
			: "cc", "memory");
	} else
		asm volatile("cld; rep stosb\n"
			:: "D" (v), "a" (c), "c" (n)
			: "cc", "memory");
	return v;
}
```

Editing `page_init()` function:

We can see that the `PageInfo` is just a linked-list of free pages. The already given code also shows a creation of a linked-list.

```
kern > pmap.c > page_init(void)
    size_t i;
    for (i = 0; i < npages; i++) {
        pages[i].pp_ref = 0;
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }    github-classroom[bot], 2 weeks ago • Initial commit
}
```

But this code makrs all pages as free which is not the case. As per the memory map given in `memlayout.h`, the page 0 is never to be accessed as BIOS Structure is located at the first page - index 0.
Next, the base memory starts at second (1) page and ends at `npages_basemem` and this memory canbe populated.
The next is IO hole. It starts from `IOPHYSMEM` and ends at `EXTPHYSMEM` and this should not be allocated.
The rest of the extended memory is free memory and can be used for allocation. However, we need to note that the kernel is also located in the extended memory. So we will use the `boot_alloc(0)` as we know it returns the address of next free page. So if the return value of `boot_alloc(0)` falls within kernel memory (which can be checked by mapping back to physical memory using `PADDR` defined in `pmap.h` which takes a kernel virtual address and returns the corresponding physical address. It panics if you pass it a non-kernel virtual address), we do not assign it to `page_free_list`.
The final code according to conditions is:

```
kern > pmap.c > page_init(void)
    size_t i;
    pages[0].pp_ref = 1;                          // Assign first page to nothing -- BIOS and IDT
    for(i = 1; i < npages; i++){
        if(i < npages_basemem){                   // Free pages in base memory
            pages[i].pp_ref = 0;
            pages[i].pp_link = page_free_list;
            page_free_list = &pages[i];
        }
        else if(i >= IOPHYSMEM/PGSIZE && i < EXTPHYSMEM/PGSIZE){    // IO Hole
            pages[i].pp_ref = 1;
        }
        else if(i >= EXTPHYSMEM/PGSIZE && i < PADDR(boot_alloc(0))/PGSIZE){     // Kernel Memory
            pages[i].pp_ref = 1;
        }
        else{                                     // Free pages in extended memory
            pages[i].pp_ref = 0;
            pages[i].pp_link = page_free_list;
            page_free_list = &pages[i];
        }
    }
}
```

Editing `page_alloc()` function:

As per the mentioned comments, we make a a new page structure `sendPage` to allocate a physical page. Here we first check if there are any free pages available in the linked list. We assign one page out of `page_free_list` to `sendPage` and assign its values to `0`, if it passes the `alloc_flags & ALLOC_ZERO` checks.
The `page2kva` check that the pages on the `page_free_list` are reasonable.

```c
struct PageInfo *
page_alloc(int alloc_flags)
{
	// Fill this function in
	struct PageInfo *sendPage;
	if(page_free_list == NULL){
		return NULL;
	}
	sendPage = page_free_list;
	if(alloc_flags & ALLOC_ZERO){
		memset(page2kva(sendPage), 0, PGSIZE);
	}
	page_free_list = page_free_list->pp_link;
	sendPage->pp_link = NULL;
	return sendPage;
}
```

Editing `page_free()` function:

To free a page, we add it back to the `page_free_list`. We do this only if the value of `pp_ref` is non zero and value of `pp_link` is not `NULL`.

```c
void
page_free(struct PageInfo *pp)
{
	// Fill this function in
	// Hint: You may want to panic if pp->pp_ref is nonzero or
	// pp->pp_link is not NULL.
	if(pp->pp_ref != 0){
		panic("Page reference count is not zero");
	}
	if(pp->pp_link != NULL){
		panic("Page link is not NULL");
	}
	pp->pp_link = page_free_list;
	page_free_list = pp;
}
```

At last we remove the line: `panic("mem_init: This function is not finished \n");` from `mem_init()` and we get the following results:

```
qemu-system-i386 -nographic -drive file=obj/kern/kernel.img,index=0,media=disk,format=raw -serial mon:stdio -gdb tcp::26000 -D qemu.log
6828 decimal is XXX octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_free_list() succeeded!
check_page_alloc() succeeded!
kernel panic at kern/pmap.c:726: assertion failed: page_insert(kern_pgdir, pp1, 0x0, PTE_W) < 0
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

We see that `check_page_free_list()` and `check_page_alloc()` succeeded.

Exercise 2:

As referenced in Section 5.2 of the Intel Reference Manual.

Figure 5-9. Page Translation

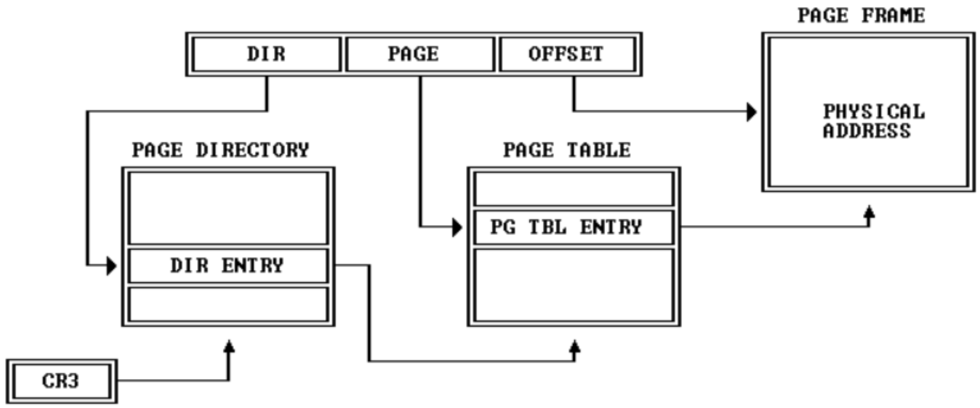
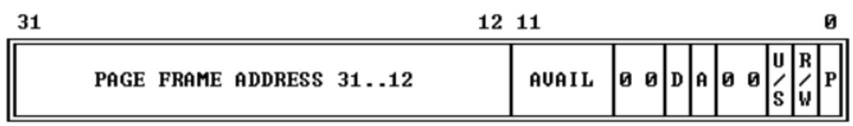


Figure 5-10. Format of a Page Table Entry



P - PRESENT
R/W - READ/WRITE
U/S - USER/SUPERVISOR
D - DIRTY
AVAIL - AVAILABLE FOR SYSTEMS PROGRAMMER USE

NOTE: 0 INDICATES INTEL RESERVED. DO NOT DEFINE.

Page translation is a technique used in x86 architecture to implement virtual memory. It involves dividing a process's virtual address space into fixed-size pages and assigning each page a physical address in memory. When a program accesses a virtual address, the CPU looks up the corresponding physical address in a page table maintained by the operating system.

The page table is organized as a hierarchical tree structure with multiple levels of page tables. To translate a virtual address to a physical address, the CPU starts by looking up the page directory entry corresponding to the most significant bits of the virtual address. It then looks up the entry in the second-level page table corresponding to the middle bits of the virtual address, which gives the physical page address. The lower bits of the virtual address are used as an offset within the physical page.

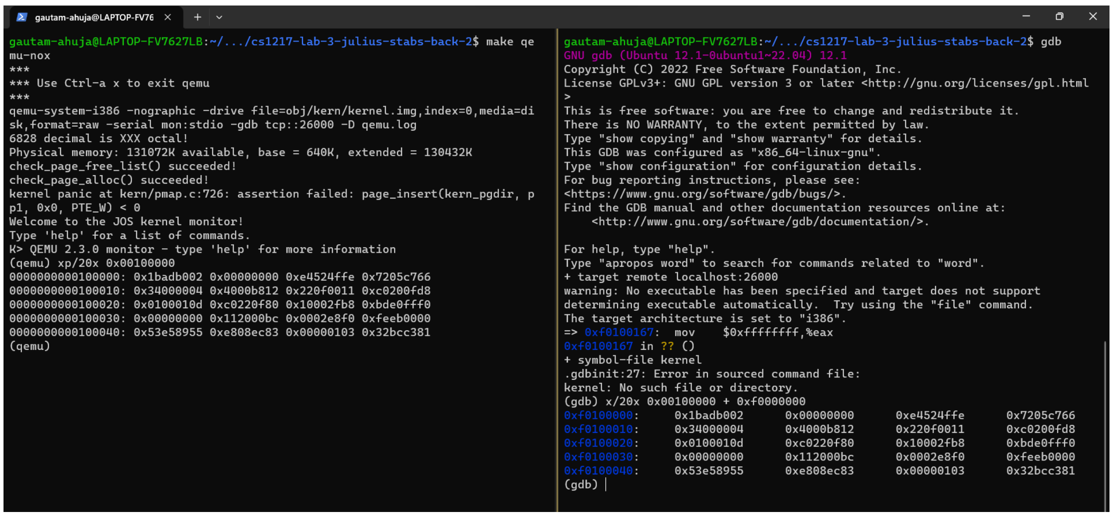
If the corresponding page table entry is not present in the page table, a page fault occurs, and the operating system loads the page from disk into physical memory. This process is called demand paging and allows the operating system to allocate memory to applications on demand.

Exercise 3:

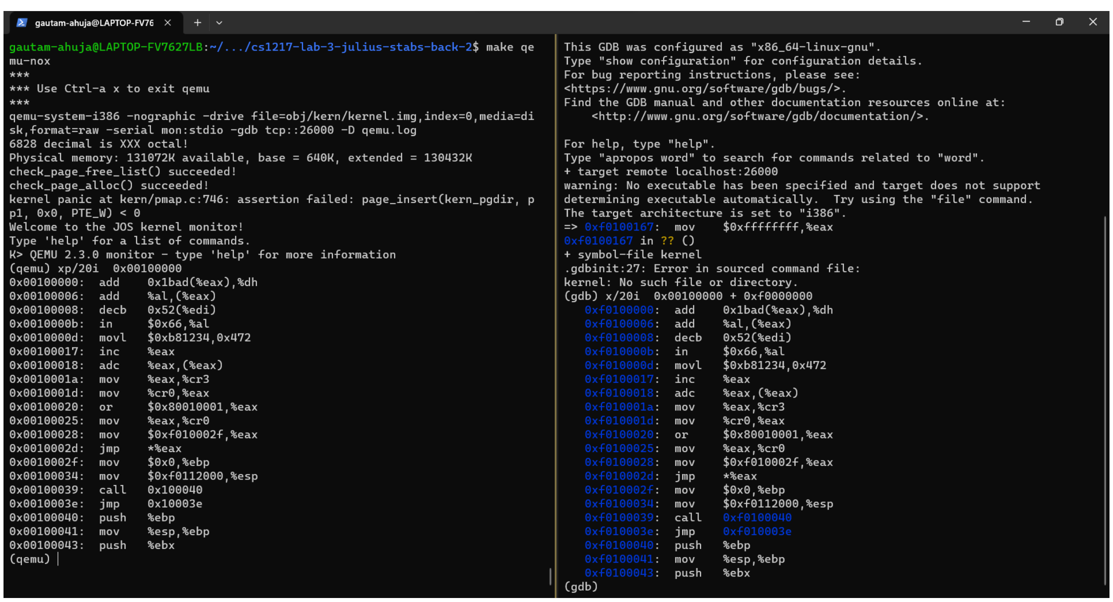
As did in LAB 1, we saw that the page table mapped a physical memory address of `0x00100000` to virtual address `0xf0100000`. Since the Page Table is only 4Mb, we could do (till now), we can actually figure out the mapping ourselves.

The virtual address is `0xf0000000` + physical address

We can use this fact to examine the memory. We can see the physical memory in `QEMU` and the virtual memory in `GDB`. We will check the first 20 values of the stack at physical address `0x00100000` and virtual address `0xf0100000`. We will use the provided `6.828 lab tools guide` to check stack values and instructions.



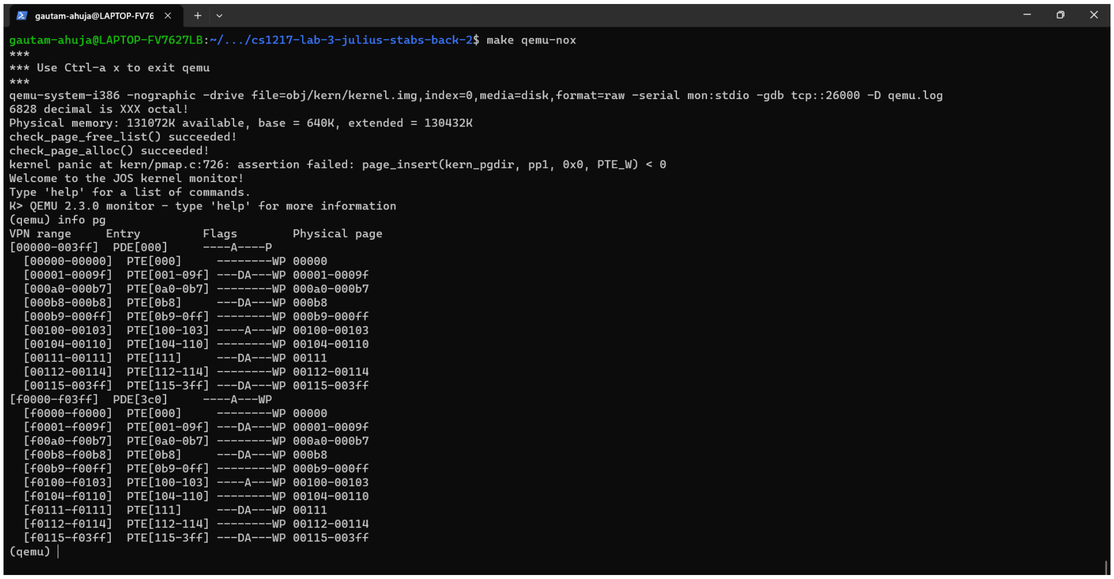
We can see that the values of stack are the same and the physical to virtual translation is just an addition of `0xf0000000`.
Checking the instructions at the address:



We can also see that the instructions are also the same for the addresses mentioed

Other `QEMU` Commands:

1. `info pg`: It shows a compact but detailed representation of the current page tables, including all mapped memory ranges, permissions, and flags.



```
gautam-ahuja@LAPTOP-FV76
gautam-ahuja@LAPTOP-FV7627LB:~/.../cs1217-lab-3-julius-stabs-back-2$ make qemu-nox
***
*** Use Ctrl-a x to exit qemu
***
qemu-system-i386 -nographic -drive file=obj/kern/kernel.img,index=0,media=disk,format=raw -serial mon:stdio -gdb tcp::26000 -D qemu.log
6828 decimal is XXX octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_free_list() succeeded!
check_page_alloc() succeeded!
kernel panic at kern/pmap.c:726: assertion failed: page_insert(kern_pgdir, pp1, 0x0, PTE_W) < 0
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> QEMU 2.3.0 monitor - type 'help' for more information
(qemu) info pg
VPN range     Entry         Flags        Physical page
[00000-003ff]  PDE[000]     ----A----P
  [00000-00000]  PTE[000]     --------WP 00000
  [00001-0009f]  PTE[001-09f] ---DA---WP 00001-0009f
  [000a0-000b7]  PTE[0a0-0b7] --------WP 000a0-000b7
  [000b8-000b8]  PTE[0b8]     ---DA---WP 000b8
  [000b9-000ff]  PTE[0b9-0ff] --------WP 000b9-000ff
  [00100-00103]  PTE[100-103] ----A---WP 00100-00103
  [00104-00110]  PTE[104-110] --------WP 00104-00110
  [00111-00111]  PTE[111]     ---DA---WP 00111
  [00112-00114]  PTE[112-114] --------WP 00112-00114
  [00115-003ff]  PTE[115-3ff] ---DA---WP 00115-003ff
[f0000-f03ff]  PDE[3c0]     ----A---WP
  [f0000-f0000]  PTE[000]     --------WP 00000
  [f0001-f009f]  PTE[001-09f] ---DA---WP 00001-0009f
  [f00a0-f00b7]  PTE[0a0-0b7] --------WP 000a0-000b7
  [f00b8-f00b8]  PTE[0b8]     ---DA---WP 000b8
  [f00b9-f00ff]  PTE[0b9-0ff] --------WP 000b9-000ff
  [f0100-f0103]  PTE[100-103] ----A---WP 00100-00103
  [f0104-f0110]  PTE[104-110] --------WP 00104-00110
  [f0111-f0111]  PTE[111]     ---DA---WP 00111
  [f0112-f0114]  PTE[112-114] --------WP 00112-00114
  [f0115-f03ff]  PTE[115-3ff] ---DA---WP 00115-003ff
(qemu)
```

2. `info mem`: It shows an overview of which ranges of virtual addresses are mapped and with what permissions

```
(qemu) info mem
0000000000000000-0000000000400000 0000000000400000 -r-
00000000f0000000-00000000f0400000 0000000000400000 -rw
(qemu)
```

In-text Questions:

Given Code:

```
mystery_t x;
char* value = return_a_pointer();
*value = 10;
x = (mystery_t) value;
```

Answer: `uintptr_t`
Here the `return_a_pointer()` return a virtual address as we know the kernel can't sensibly dereference a physical address. Hence this address is a virtual address.

Exercise 4:

Editing `page_free()` function:

We will be using these from the file `mmu.h`:

1. The `PDX()` and `PTX()` macros to calculate the of page directory and page table respectively.

```c
inc > C mmu.h > ...
// page directory index
#define PDX(la)     ((((uintptr_t) (la)) >> PDXSHIFT) & 0x3FF)

// page table index
#define PTX(la)     ((((uintptr_t) (la)) >> PTXSHIFT) & 0x3FF)
```

2. The `PTE_ADDR()` macros that is used to find the address in page table or page directory entry

```c
inc > C mmu.h > CR0_WP
// Address in page table or page directory entry
#define PTE_ADDR(pte)   ((physaddr_t) (pte) & ~0xFFF)
```

3. And finally the permissions and flags associated with the page table/directory entry.

```c
inc > C mmu.h > ...
// Page table/directory entry flags.
#define PTE_P       0x001   // Present
#define PTE_W       0x002   // Writeable
#define PTE_U       0x004   // User
#define PTE_PWT     0x008   // Write-Through
#define PTE_PCD     0x010   // Cache-Disable
#define PTE_A       0x020   // Accessed
#define PTE_D       0x040   // Dirty
#define PTE_PS      0x080   // Page Size
#define PTE_G       0x100   // Global
```

Another macros function as mentioned in question paper is `KADDR()` loacted in `pmap.h`. Since the addresses in page directory and page table are all physical address. We will use this to return the kernel virtual address of a physical address.

We are referenceing a two level page table. As discussed in question two, The page directory entry points to an address of a page table. Linear address always equals the offset of the virtual address. To walk down page directory, we work as follows:

1. Get the page directory index of given `va`
2. Check if a page table exists for given page directory index.
   - If it exists, get the address stored at that index (which is the address of page table index), convert it virtual address, as `PTE_ADDR` return a physical address.
   - Return the address to the index of Page Table Entry (which we get through `PTX()` function).
3. If the page table does not exist. We check the `create` argument. If true:
   - Allocate new page using `page_alloc()` and check if valid.

- Set the new page's physical address in page directory along with permissions (bitwise `OR`) using the `page2pa()` function.
- Return the address to page table entry of this new page.

4. If `create` argument is false, return `NULL`.

kern > C pmap.c > pgdir_walk(pde_t *, const void *, int)

```c
pte_t *
pgdir_walk(pde_t *pgdir, const void *va, int create)
{
    // Fill this function in
    pde_t *pde;                 // Page Directory Entry
    pte_t *pte;                 // Page Table Entry
    struct PageInfo *newPage;   // New Page
    pde = &pgdir[PDX(va)];      // Get Page Directory Entry
    if(*pde & PTE_P){           // Check if page table exists
        pte = (pte_t*)KADDR(PTE_ADDR(*pde));            // Get Page Table Entry -- Virtual Address
        return &pte[PTX(va)];                           // Return Page Table Entry -- Virtual Address
    }
    else if(create){                            // Create new page table
        newPage = page_alloc(ALLOC_ZERO);       // Allocate new page with zeroed
        if(newPage == NULL){                    // Failed to allocate new page
            return NULL;
        }
        newPage->pp_ref++;                      // Increment reference count
        *pde = page2pa(newPage) | PTE_P | PTE_W | PTE_U;        // Set page directory entry and permissions
        pte = (pte_t*)KADDR(PTE_ADDR(*pde));    // Get Page Table Entry -- Virtual Address
        return &pte[PTX(va)];                   // Return Page Table Entry -- Virtual Address
    }
    else{
        return NULL;
    }
}       github-classroom[bot], 2 weeks ago • Initial commit
```

Editing `boot_map_region()` function:

We follow the instructions mentioned in comments. We need to map a contiguous memory. Since `size` is of the multiple `PGSIZE` and `va` and `pa` are both page-aligned, we can use the `pgdir_walk` which maps the memory of size `PGSIZE`.
We first allocate a new new page table page for a given virtual address between [va, va+size). If it fails, panic and exit. Else we map this address space to physical [pa, pa+size) while setting the permissions bits perm—PTE_P for the entries.

kern > C pmap.c > boot_map_region(pde_t *, uintptr_t, size_t, physaddr_t, int)

```c
static void
boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa, int perm)
{
    // Fill this function in
    size_t i;
    for(i=0; i < size/PGSIZE; i++){
        pte_t *pte = pgdir_walk(pgdir, (void*)(va + i*PGSIZE), 1);
        if(pte == NULL){
            panic("boot_map_region: Failed to map region");
        }
        else{
            *pte = (pa + i*PGSIZE) | perm | PTE_P;
        }
    }
}       github-classroom[bot], 2 weeks ago • Initial commit
```

Editing `page_lookup()` function:

Since our page table is a two level, we need to check if a page table entry exists at the given page table directory. For this we will use the `page_pgdir_walk()` function.
If the address of entry exists and is entry itself is present (checking the `PTE_P` permission) then we move ahead to check if the provided `pte_store` address is valid.
At last we store `pte` as entry to `pte_store`, map and return the new page's physical address in page directory along with permissions (bitwise `OR`) using the `page2pa()` function.

kern > C pmap.c > ...

```c
struct PageInfo *
page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
{
    // Fill this function in
    pte_t *pte = pgdir_walk(pgdir, va, 0);
    if(pte == NULL || (*pte & PTE_P) == 0){
        return NULL;
    }
    if(pte_store){
        *pte_store = pte;
    }
    return pa2page(PTE_ADDR(*pte));
}
```

Editing `page_remove()` function:

As per the comments, we simply first check of a given page exists at given virtual address `va` using `page_lookup()` and then do the following:

- If the page does not exists, do nothing.
- If the page exists:
  - Decrement ref count on the physical page using `page_decref()`. This function also frees the page of the count reaches zero.
  - At last we invalidate the tlb entry using `tlb_invalidate()` function.

kern > C pmap.c > ...

```c
void
page_remove(pde_t *pgdir, void *va)
{
    // Fill this function in
    pte_t *pte;
    struct PageInfo *checkPage = page_lookup(pgdir, va, &pte);      // Get Page Table Entry
    if(checkPage != NULL){              // Page exists
        page_decref(checkPage);         // Decrement reference count
        *pte = 0;                       // Set page table entry to 0
        tlb_invalidate(pgdir, va);      // Invalidate TLB
    }
    else{
        return;
    }
}
```

Editing `page_insert()` function:

This function maps a physical page `pp` to a virtual address `va`. Following comments, we use `page_walk()` with `create` flag as true. If the allocation fail, return `-E_NO_MEM` as given.

As given in comments, we figure out to increase the `pp_ref` first because using `page_remove()` itself decrement the count. We also consider the corner case, where the same `pp` is re-inserted at the same virtual address in the same `pgdir`. Not increasing the count will free the page.
At last, we set the permission of the page table entry and page directory.

```
kern > C pmap.c > ...
int
page_insert(pde_t *pgdir, struct PageInfo *pp, void *va, int perm)
{
    // Fill this function in
    pte_t *pte;
    pte = pgdir_walk(pgdir, va, 1);          // Get Page Table Entry
    if(pte == NULL){                         // Failed to get Page Table Entry
        return -E_NO_MEM;
    }
    pp->pp_ref++;                            // Increment reference count
    if(*pte & PTE_P){                        // Check if page is already mapped
        page_remove(pgdir, va);
    }
    *pte = page2pa(pp) | perm | PTE_P;       // Set page table entry and permissions
    pgdir[PDX(va)] |= perm;                  // Allocate and inserte page table into page directory
    return 0;
}
```

Results:

```
gautam-ahuja@LAPTOP-FV7627LB:~/.../cs1217-lab-3-julius-stabs-back-2$ make qemu-nox
***
*** Use Ctrl-a x to exit qemu
***
qemu-system-i386 -nographic -drive file=obj/kern/kernel.img,index=0,media=disk,format=raw -serial mon:stdio -gdb tcp::26
000 -D qemu.log
6828 decimal is XXX octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_free_list() succeeded!
check_page_alloc() succeeded!
check_page() succeeded!
```

The `check_page()` succeeded and the code is working properly.

Exercise 5:

Question 1:

We refer to the comments mentioned and add the mappings for specific address spaces using the `boot_map_region()` at three places.
First we map `pages` read-only by the user at linear address `UPAGES` with permission for both kernel and user set to read.

```
kern > C pmap.c > mem_init(void)
    // Your code goes here:
    // boot_map_region(kern_pgdir, UPAGES, PTSIZE, PADDR(pages), PTE_U | PTE_P);
    boot_map_region(kern_pgdir, UPAGES, npages * sizeof(struct PageInfo), PADDR(pages), PTE_U | PTE_P);
```

Then we map the physical memory that `bootstack` refers to in range `[KSTACKTOP-PTSIZE, KSTACKTOP)`. Since we need to physical memory and `bootstack` is virtual memory, we convert it to physical using `PADDR()` function.

```
kern > C pmap.c > mem_init(void)
    //      Permissions: kernel RW, user NONE
    // Your code goes here:
    boot_map_region(kern_pgdir, KSTACKTOP - KSTKSIZE, KSTKSIZE, PADDR(bootstack), PTE_W | PTE_P);
```

Lastly, we map the rest of the physical space starting from `KERNBASE` to 2^32 and set the permissions.

```
kern > C pmap.c > mem_init(void)
    // Your code goes here:
    boot_map_region(kern_pgdir, KERNBASE, 0xffffffff-KERNBASE, 0, PTE_W | PTE_P);
```

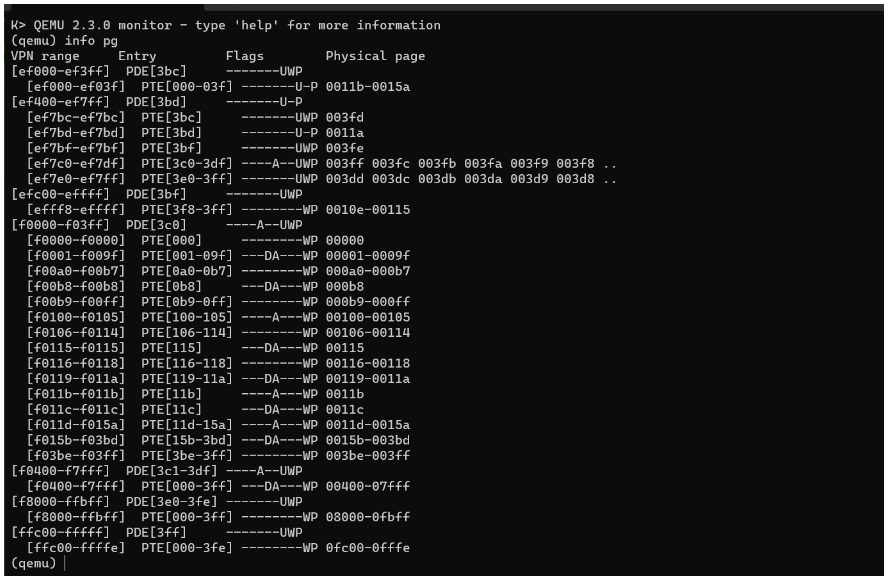
Results:

```
gautam-ahuja@LAPTOP-FV7627LB:~/.../cs1217-lab-3-julius-stabs-back-2$ make qemu-nox
***
*** Use Ctrl-a x to exit qemu
***
qemu-system-i386 -nographic -drive file=obj/kern/kernel.img,index=0,media=disk,format=raw -serial mon:stdio -gdb tcp::26
000 -D qemu.log
6828 decimal is XXX octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_free_list() succeeded!
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_free_list() succeeded!
check_page_installed_pgdir() succeeded!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> |
```

The `check_kern_pgdir()` and `check_page_installed_pgdir()` are successful.

Question 2:

To fill in the table, we refer to the memory map mentioned in the comments of `memlayout.h`. We also see the output of the `info pg` command in the `QEMU` monitor. We know the kernel reserves approximately 256MB of virtual address space.



```
K> QEMU 2.3.0 monitor - type 'help' for more information
(qemu) info pg
VPN range     Entry         Flags        Physical page
[ef000-ef3ff]  PDE[3bc]     -------UWP
  [ef000-ef03f]  PTE[000-03f] -------U-P 0011b-0015a
[ef400-ef7ff]  PDE[3bd]     -------U-P
  [ef7bc-ef7bc]  PTE[3bc]     -------UWP 003fd
  [ef7bd-ef7bd]  PTE[3bd]     -------U-P 0011a
  [ef7bf-ef7bf]  PTE[3bf]     -------UWP 003fe
  [ef7c0-ef7df]  PTE[3c0-3df] ----A--UWP 003ff 003fc 003fb 003fa 003f9 003f8 ..
  [ef7e0-ef7ff]  PTE[3e0-3ff] -------UWP 003dd 003dc 003db 003da 003d9 003d8 ..
[efc00-effff]  PDE[3bf]     -------UWP
  [efff8-effff]  PTE[3f8-3ff] --------WP 0010e-00115
[f0000-f03ff]  PDE[3c0]     ----A--UWP
  [f0000-f0000]  PTE[000]     --------WP 00000
  [f0001-f009f]  PTE[001-09f] ---DA---WP 00001-0009f
  [f00a0-f00b7]  PTE[0a0-0b7] --------WP 000a0-000b7
  [f00b8-f00b8]  PTE[0b8]     ---DA---WP 000b8
  [f00b9-f00ff]  PTE[0b9-0ff] --------WP 000b9-000ff
  [f0100-f0105]  PTE[100-105] ----A---WP 00100-00105
  [f0106-f0114]  PTE[106-114] --------WP 00106-00114
  [f0115-f0115]  PTE[115]     ---DA---WP 00115
  [f0116-f0118]  PTE[116-118] --------WP 00116-00118
  [f0119-f011a]  PTE[119-11a] ---DA---WP 00119-0011a
  [f011b-f011b]  PTE[11b]     ----A---WP 0011b
  [f011c-f011c]  PTE[11c]     ---DA---WP 0011c
  [f011d-f015a]  PTE[11d-15a] ----A---WP 0011d-0015a
  [f015b-f03bd]  PTE[15b-3bd] ---DA---WP 0015b-003bd
  [f03be-f03ff]  PTE[3be-3ff] --------WP 003be-003ff
[f0400-f7fff]  PDE[3c1-3df] ----A--UWP
  [f0400-f7fff]  PTE[000-3ff] ---DA---WP 00400-07fff
[f8000-ffbff]  PDE[3e0-3fe] -------UWP
  [f8000-ffbff]  PTE[000-3ff] --------WP 08000-0fbff
[ffc00-fffff]  PDE[3ff]     -------UWP
  [ffc00-ffffe]  PTE[000-3fe] --------WP 0fc00-0fffe
(qemu) |
```

The final table is:

Entry	**Base Virtual Address**	**Points to (logically)**
1023	`0xffc00000`	Page table for top 4MB of Physical memory
1022	`0xff800000`	Page table for second top 4MB of Physical memory
...	...	...
959	`0xf0000000`	`KERNBASE`, `KSTACKTOP` - Phys. Mem. Mapping Starts Here.
958	`0xefc00000`	`MMIOLIM` and Start of Kernel Stack
957	`0xef400000`	`UVPT` and Start of Cur. Page Table (Kernel page directory)
956	`0xef000000`	`UPAGES` and Start of Read Only Pages (`PageInfo` structure)
955	`0xeec00000`	UNMAPPED
...	...	...
0	`0x00000000`	UNMAPPED

Question 3:
The user programs not be able to read or write the kernel's memory because they lack permissions to do so. We did not update the user permission to read or write and this is done so only kernel has the access to all pages and mapping not the individual programs. This is done for security.

Question 4:
This operating system can only address up to 4 GB of memory.
Since we set the architecture to `i386` and according to Intel 80386 Reference (80386 is older name of i386), the page directory addresses up to 1K page tables of the second level. And each second level addresses 1K Pages. Because each page contains 4K bytes (2^{12}) bytes, the amount of physical space addressed is $1024 * 1024 * 2^{12} = 2^{32} = 4$GB.

Question 5:
To manage memory we require `PageInfo`, Page Table and 1 Page Directory.
Page directory is just one Page = 4KB

Each entry in Page Directory points to a Page Table. There are 1024 Page Tables and each Page Table is a page itself = $1024 * 4KB = 4$ MB. This is also defined similary in `mmu.h`
Now each of the entry (1024 entries in each table as defined by `NPTENTRIES` in `mmu.h`) of the 1024 page tables can hold a `struct PageInfo` which is of size 8 bytes (a pointer and an integer). Therefore $8 * 1024 * 1024 = 8$MB
Therefore, total overhead = 8MB + 4MB + 4KB = 12MB + 4KB

Question 6:
As we saw in lab 1, the machine first boots up at low address because the BIOS needs to set up everything meanwhile the kernel is loaded (not executed) at low addresses. At this point the value of `%eax` is changed to a high value. The exact instruction at which this happens is `jmp *%eax` in `entry.S`.
We see in `entrypgdir.c` that both both virtual address [0, 4MB) and [KERNBASE, KERNBASE+4MB) are mapped to the same physical address [0, 4MB). This lets us to continue executing at a low `EIP` between when we enable paging and when we begin running at an `EIP` above `KERNBASE`. After the paging is turned on, the kernel is then loaded at high addresses of `0xf0000000` which is why the transition is necessary.

Exercise 6:

Scrolling through the Volume 3 of the current Intel manuals we find the section where it explain the Paging Mechanism. Since it is mentioned in the lab, we will only be using Linear Address Translation and no segmentation.
Under section 3.7.1 of manual we find translation for 4-KByte Pages and in next section 3.7.2 we see the 4-MByte Pages translation. The figure[1] below shows how a page directory can be used to map linear addresses to 4-MByte pages.

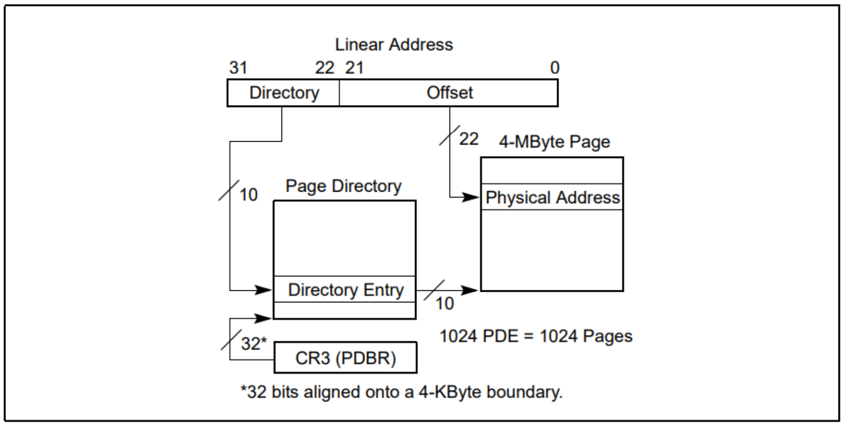


Figure 3-13. Linear Address Translation (4-MByte Pages)

From above we know the offset of page is 22 bit shift which can be obtained by doing a bit-wise `AND` on address with (00000000001111111111111111111111) (`0x3FFFFF` in hex), as per following the code for 4KB pages.
Similarly, the directory address can be obtained by doing a bit-wise `AND` on address with (11111111110000000000000000000000) (`0xFFC00000` in hex).

The later section 3.7.6 explains the Page-Directory and Page-Table Entries and the associated permissions as follows:

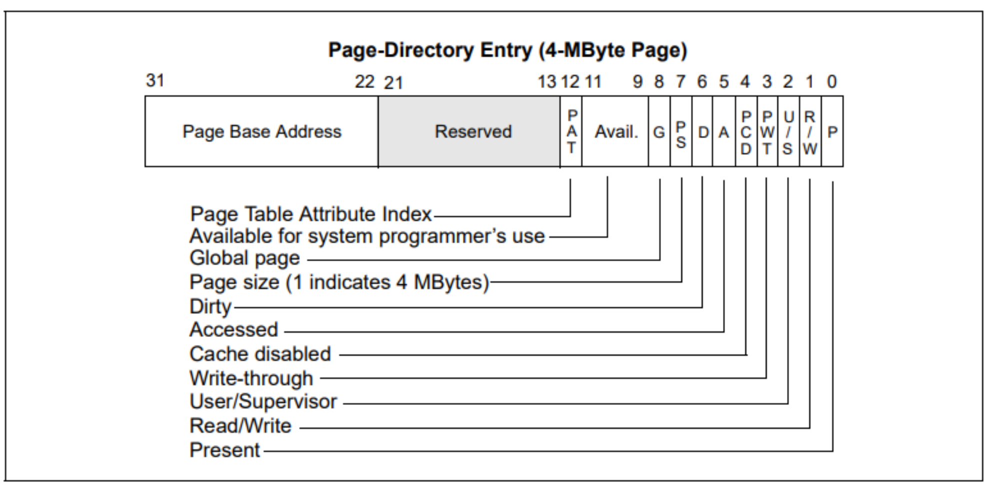


Figure 3-15. Format of Page-Directory Entries for 4-MByte Pages and 32-Bit Addresses

[1]Figure 3-13. Linear Address Translation (4-MByte Pages), Intel 64 and IA-32 Architectures Software Developer's Manual, Page 109

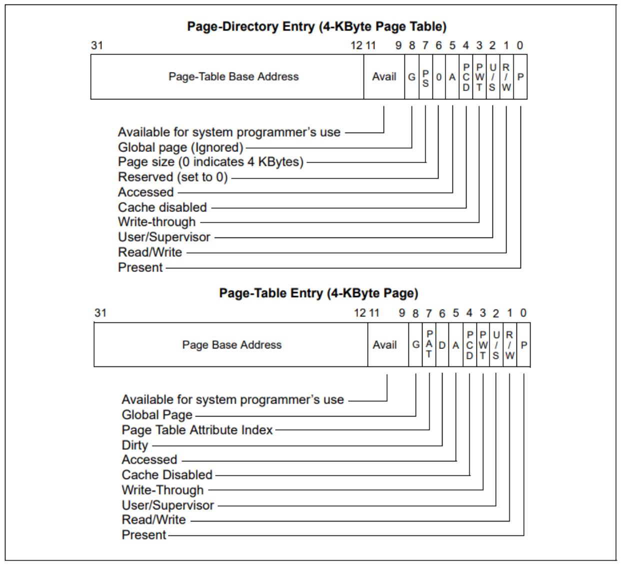


Figure 3-14. Format of Page-Directory and Page-Table Entries for 4-KByte Pages and 32-Bit Physical Addresses

Since there are no page tables for 4MB Pages, we only work with Directories. The `PS` bit represents whether the page size is 4 MB (1) or not (0). Using this we only need to modify `boot_map_region()` since only this function is used to map kernel region which needs to be remapped for 4 MB pages. We may modify our code as follows:

- Define new parameters in `mmu.h` — New Page Size (N_PGSIZE), New Page Offset (N_PGOFF), and New Address in page directory entry (PDE_ADDR).

```c
#define N_PGSIZE     4194304     // bytes mapped by new 4MB page
// offset in new 4MB page
#define N_PGOFF(la) (((uintptr_t) (la)) & 0x3FFFFF)
// Address in page directory entry for 4MB page
#define PDE_ADDR(pde)    ((physaddr_t) (pde) & ~0xFFC00000)
```

- Next we edit our `boot_map_region()`. First check if 4MB Support is available and `PTE_PS` bit is present in permissions and if true, map a every page in page directory for range `[va, va+size)`.

```c
boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa, int perm)
{
    // Fill this function in
    size_t i;
    // 4 MB Paging
    if(fourMB_page_support){
        if(perm & PTE_PS){
            for(i=0; i < size/N_PGSIZE; i++){
                pte_t *pte = pgdir + PDX(va + i*N_PGSIZE);
                if(pte == NULL){
                    panic("boot_map_region: Failed to map region");
                }
                else{
                    *pte = (pa + i*N_PGSIZE) | perm | PTE_P;
                }
            }
        }
    }
    // 4 KB Paging
    for(i=0; i < size/PGSIZE; i++){
        pte_t *pte = pgdir_walk(pgdir, (void*)(va + i*PGSIZE), 1);
        if(pte == NULL){
            panic("boot_map_region: Failed to map region");
        }
        else{
            *pte = (pa + i*PGSIZE) | perm | PTE_P;
        }
    }
}
```

- Now, to pass the `check_kern_pgdir()`; we see that the function is basically checking the assertions for conditions on `check_va2pa()` function. After seeing its definition, we add another condition for `PTE_PS`.

```c
// 4 MB paging
if (*pgdir & PTE_PS)
    return PDE_ADDR(*pgdir) | N_PGOFF(va);
```

- Noe we edit our `i386_detect_memory()` function to check for the `CR4_PSE` which checks if page size extensions are available by reading control register 4 through `rcr4()` function. If available, we set a global check to true.

```c
// If CR4_PSE is set, we can use 4MB pages to map Kernel memory.
if (rcr4() & CR4_PSE) {
    fourMB_page_support = true;
}
```

- At last edit `mem_init()` to check for the flag. If 4MB is not available, print a message.

```c
// Check for 4 MB pages Support
if(!fourMB_page_support){
    cprintf("4MB pages not supported!\n");
}       You, 5 minutes ago • Uncommitted changes
```

This roughly completes our allocation of 4 MB Pages. This may still give errors.
On running the `make` all checks are successful and the 4 MB page support is not available.

```
qemu-system-i386 -nographic -drive file=obj/kern/kernel.img,index=0,media=disk,format=raw -serial mon:stdio -gdb tcp::26
000 -D qemu.log
6828 decimal is XXX octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_free_list() succeeded!
check_page_alloc() succeeded!
check_page() succeeded!
4MB pages not supported!
check_kern_pgdir() succeeded!
check_page_free_list() succeeded!
check_page_installed_pgdir() succeeded!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

Exercise 7:

We follow the similar path we did in Lab 1 to make a custom command. To add any custom command we edit the `monitor.h` & `monitor.c` files.

1. `showmappings`
   First we edit `monitor.h` to add the declaration of the function.

   ```c
   int mon_showmappings(int argc, char **argv, struct Trapframe *tf);
   ```

   Next, we add the new command to the `commands[]` list in `monitor.c`

   ```c
   {"showmappings", "Show mappings between two addresses", mon_showmappings },
   ```

   Next, we need to add the function `mon_showmappings()` to `monitor.c`. We includes a check to make sure the user input three arguments (showmappings ¡start_address¿ ¡end_address¿). Here the start and end addresses will be present at `argv[1]` and `argv[2]` respectively
   Since the input will be in string, we will need a type conversion on addresses. We will use `strtol()` defined in `string.c` which converts a string to `long` datatype.
   Then to get the physical address mapping, we first get the page table entry for that virtual address using `pgdir_walk()` and then convert it to physical address using `PTE_ADDR`.
   We also print the permissions (which are stored in the 12 bit offset) doing a bitwise-`AND` with `PTE_U, PTE_W, PTE_P`. We will store the permissions in a string and print it as output. Repeat this process till we reach `end`.

   ```c
   kern > monitor.c > mon_showmappings(int, char **, Trapframe *)
   mon_showmappings(int argc, char **argv, struct Trapframe *tf)
   {
       // Your code here.
       if(argc != 3){
           cprintf("Usage: showmappings <start_address> <end_address>\n");
           return 0;
       }
       long start = strtol(argv[1], NULL, 16);
       long end = strtol(argv[2], NULL, 16);
       // Check if end address is greater than start address
       if(end < start){
           cprintf("Warning: End Address Should be Greater Than Start Address");
           // swap the values
           long temp = start;
           start = end;
           end = temp;
       }
       cprintf("Virtual Address\t\tPhysical Address\t\tPermissions\n");
       // Iterate over the virtual addresses
       for(long i = start; i <= end; i += PGSIZE){
           pte_t *pte = pgdir_walk(kern_pgdir, (void*)i, 0);
           if(pte == NULL){
               cprintf("%08x\t\tUnmapped\t\t-\n", i);
               continue;
           }
           // Get the physical address
           long physical_address = PTE_ADDR(*pte);
           // Get the permissions
           char permissions[4];
           permissions[0] = (*pte & PTE_U) ? 'U' : '-';
           permissions[1] = (*pte & PTE_W) ? 'W' : '-';
           permissions[2] = (*pte & PTE_P) ? 'P' : '-';
           permissions[3] = '\0';
           cprintf("%08x\t\t%08x\t\t%s\n", i, physical_address, permissions);
       }
       cprintf("\n");
       return 0;
   }       You, 36 seconds ago • Uncommitted changes
   ```

```
K> showmappings 0x3000 0x5000
Virtual Address                    Physical Address                Permissions
00003000                           Unmapped                -
00004000                           Unmapped                -
00005000                           Unmapped                -
```

```
K> showmappings 0xf0000000 0xf0010000
Virtual Address                    Physical Address                Permissions
f0000000                           00000000                - W P --- - - -- -
f0001000                           00001000                - W P --- A D -- -
f0002000                           00002000                - W P --- A D -- -
f0003000                           00003000                - W P --- A D -- -
f0004000                           00004000                - W P --- A D -- -
f0005000                           00005000                - W P --- A D -- -
f0006000                           00006000                - W P --- A D -- -
f0007000                           00007000                - W P --- A D -- -
f0008000                           00008000                - W P --- A D -- -
f0009000                           00009000                - W P --- A D -- -
f000a000                           0000a000                - W P --- A D -- -
f000b000                           0000b000                - W P --- A D -- -
f000c000                           0000c000                - W P --- A D -- -
f000d000                           0000d000                - W P --- A D -- -
f000e000                           0000e000                - W P --- A D -- -
f000f000                           0000f000                - W P --- A D -- -
f0010000                           00010000                - W P --- A D -- -
```

2. `Set Permissions`
   To set, clear, or change the permissions of any mapping in the current address we follow a similar trajectory.
   Take address and permissions bit and convert to `long`. Then find the PTE and change the permissions using `*pte = *pte | new_prem;`.
   We also create an array to store all permissions and print them before and after change.

```
K> setperm 0xf0000000 0x004
Virtual Address: f0000000
Old Permissions:
- W P --- - - -- -
Permissions Set Successful
New Permissions:
U W P --- - - -- -
K> |
```

3. `Dump`:
   To make this, we follow the same routine. We check for the input arguments format `dump <virtual(V)/physical(P)> <start_address> <end_address>`
   If the address is virtual, we find its PTE, and then map it to physical address (using `PTE_ADDR` and `KADDR`) and print its memory contents. At any other point we throw an error.

```
K> dump v 0xf0000000 0xf0000020
Address:                      Memory Content:
f0000000                      f000ff53
f0000004                      f000ff53
f0000008                      f000e2c3
f000000c                      f000ff53
f0000010                      f000ff53
f0000014                      f000ff53
f0000018                      f000ff53
f000001c                      f000ff53
f0000020                      f000fea5
K> |
```

4. `Do Something Extra` For this part we just copy paste the `backtrace` function we made in LAB 1.

Exercise 8:

Idea:
This is exercise is an example of implementation of buddy allocator. The idea behind it is to have pages of multiple sizes in powers of 2 such that $2^i \leq \texttt{size}$ for some i. Then have pages of size of powers of two such that their sum equals 2^i. While allocating a page, we choose the a page such that required page size $\leq 2^j$ for some j. If that size is unavailable, split the $(j+t)$th page into two j pages.

Implementation
To implements the idea, we will need a structure such (as a double linked list). Where the first linked list contains the head of linked lists. Where each index has the head of linked list containing pages of size 2^i where i is the index.
The second linked list is the chain of free pages of that size.

When a process needs a page to be allocated, the nearest size is calculated and one free page from the chain is allocated to the process. If no free page is available then the next bigger size page is split in two (or two smaller pages are joined, depending on the policy) and allocated to program.
When a Page is freed it is allocated back to the to end of chain of linked lists.

Advantages:
There are multiple advantages of using a buddy allocator. Now, we can allocate and deallocated a page faster as we know the exact amount it needs. We also get advantage in addressing. There will be less misses because the entire page is loaded when accessed.

Disadvantages:
One issue that might arise in this system is external fragmentation, where free blocks become scattered throughout memory, making it difficult to find contiguous blocks of memory for requests larger than largest page size.
Another issue is internal fragmentation, where small allocations leave unused space in larger blocks. This can be reduced by choosing appropriate block sizes and by splitting blocks only when necessary.

Implementation:
We were not able to make a buddy allocator for this LAB. Lack of expertise and time were the primary reason. However, we believe roughly we have to do the following:

1. Edit the `struct PagInfo` to store the size of the page.
2. Create a new structure `struct buddyPage` to store the heads of every first page of size equals 2^{index}.
3. Find an effecient enough size for `buddyPage` array.
4. Create a function to create atleast one page of each size at start.
5. When mapping pages in `page_alloc()`, `boot_alloc()` or `boot_map_region()`, we take the size and map them to nearest possible page of that size (split if no page present).