JOS Lab2

Jen-Tse Huang, 1500017836

Exercise 1

- `boot_alloc()`

This function serves to allocate enough pages of contiguous physical memory.

When n > 0, we need to check if we are out of memory firstly. Function `i386_detect_memory()` set variable `npages` which tells us the number of pages. And we can find `PGSIZE` in `mmu.h` and `KERNBASE` in `memlayout.h` and calculate the maximum size of address by $KERNBASE + npages * PGSIZE$.

Finally we need to modify the pointer to next free address. Remember to use macro `ROUNDUP()`.

Here is the code:

```
if (n == 0) {
    return nextfree;
} else {
    result = nextfree;
    nextfree = ROUNDUP(nextfree + n, PGSIZE);
    if ((uint32_t)nextfree > KERNBASE + npages * PGSIZE) {
        panic("Out of memory.\n");
    } else {
        return result;
    }
}
```

- `mem_init()`

First, allocate a array of `struct PageInfo` to keep track of all pages, set the array to `0`.

```
pages = (struct PageInfo *)boot_alloc(npages * sizeof(struct PageInfo));
memset(pages, 0, npages * sizeof(struct PageInfo));
```

- `page_init()`

This function is to initiate all `struct PageInfo` in `pages`. `pp_ref` is the number of pointers to this page. And `pp_link` points to the next free page. `pp_link` is not NULL i.i.f. `pp_ref` equals zero.

`IOPHYSMEM` and `EXTPHYSMEM` are defined in `memlayout.h`.

Data structure that kernel uses can be found by `boot_alloc(0)`, which returns the next free page.

```
size_t i;

for (i = 0; i < npages; i++) {
    if (i == 0) {
```

```
            // Page 0
            pages[i].pp_ref = 1;
            pages[i].pp_link = NULL;
        } else if (i >= IOPHYSMEM / PGSIZE && i < EXTPHYSMEM / PGSIZE) {
            // I/O hole
            pages[i].pp_ref = 1;
            pages[i].pp_link = NULL;
        } else if (i >= EXTPHYSMEM / PGSIZE && i < ((uint32_t)boot_alloc(0) - KERNBASE) / PGSIZE) {
            // Some data structure that kernel has used in boot_alloc()
            pages[i].pp_ref = 1;
            pages[i].pp_link = NULL;
        } else {
            // Rest parts are free
            pages[i].pp_ref = 0;
            pages[i].pp_link = page_free_list;
            page_free_list = &pages[i];
        }
    }
```

- `page_alloc()`

This function allocate a new page and maintain `page_free_list`. Also set memory with zero if needed.

```
    if (page_free_list == NULL) {
        // Returns NULL if out of free memory
        return NULL;
    }
    struct PageInfo *ret = page_free_list;
    page_free_list = page_free_list->pp_link;
    ret->pp_link = NULL;
    if (alloc_flags & ALLOC_ZERO) {
        // After mem_init(), we need to use virtual address
        memset(page2kva(ret), 0, PGSIZE);
    }
    return ret;
```

- `page_free`

This function free a page. It will panic when `pp_ref` does not equal to zero (there are still some processes using this page), or `pp_link` is not NULL (it has been freed).

```
    if (pp->pp_ref != 0) {
        panic("There are still some processes using this page\n");
    }
    if (pp->pp_link != NULL) {
        panic("This page has been freed\n");
    }
    pp->pp_link = page_free_list;
    page_free_list = pp;
```

Finally, type `make qemu`, and we will get:

```
check_page_free_list() succeeded!
check_page_alloc() succeeded!
```

Exercise 2

I have studied these in courses like Introduction to Computer System or Operation System.

Exercise 3

The output of `x/10i 0xf00100000` in `gdb` and `xp/10i 0x000100000` in `qemu`. We can see the same codes.

```
(qemu) xp/10i 0x000100000
0x00100000:  add    0x1bad(%eax),%dh
0x00100006:  add    %al,(%eax)
0x00100008:  decb   0x52(%edi)
0x0010000b:  in     $0x66,%al
0x0010000d:  movl   $0xb81234,0x472
0x00100017:  and    %dl,(%ecx)
0x00100019:  add    %cl,(%edi)
0x0010001b:  and    %al,%bl
0x0010001d:  mov    %cr0,%eax
0x00100020:  or     $0x80010001,%eax

(gdb) x/10i 0xf00100000
   0x100000:   add    0x1bad(%eax),%dh
   0x100006:   add    %al,(%eax)
   0x100008:   decb   0x52(%edi)
   0x10000b:   in     $0x66,%al
   0x10000d:   movl   $0xb81234,0x472
   0x100017:   and    %dl,(%ecx)
   0x100019:   add    %cl,(%edi)
   0x10001b:   and    %al,%bl
   0x10001d:   mov    %cr0,%eax
   0x100020:   or     $0x80010001,%eax
```

`info pg` and `info mem` show the mapping in memory.

- Q: Assuming that the following JOS kernel code is correct, what type should variable `x` have, `uintptr_t` or `physaddr_t`?

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

  `uintptr_t`, because `*value` is a virtual address.

Exercise 4

- `pgdir_walk()`

Given the base address of page directory and a virtual address, we need to return the page table entry. If the page table page does not exist, then allocate one.

To get the directory and table index of a virtual address, use `PDX()` and `PTX()` in `mmu.h`.

To find out if the page exists, use the flag provided in `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
```

The addresses in page directory and page table are all physical address. But we need to return virtual address. Use `KADDR()` in `pamp.h` for the transformation.

`PTE_ADDR()` in `mmu.h` serves to cover the flags in the lower bit.

```
// This pointer points to the page tale address in page directory
uintptr_t *pt_addr = pgdir + PDX(va);

// Check if the page table page exists
if (*pt_addr & PTE_P) {
  // Exists
  // Returns virtual address
  // Use PTE_ADDR() to cover some flags
  return (pte_t *)KADDR(PTE_ADDR(*pt_addr)) + PTX(va);
} else {
  // Does not exist
  if (create == false) {
    // Does not create new page
    return NULL;
  } else {
    // Create a new page
    struct PageInfo *new_pg = page_alloc(ALLOC_ZERO);
    if (new_pg == NULL) {
      // Allocation failed
      return NULL;
    } else {
      // Maintain the pp_ref
      new_pg->pp_ref ++;

      // Write the physical address in page directiry
      // And we need to add permission flag
      *pt_addr = page2pa(new_pg) | PTE_U | PTE_W | PTE_P;


      // Returns virtual address
```

```
            // Use PTE_ADDR() to cover some flags
            return (pte_t *)KADDR(PTE_ADDR(*pt_addr)) + PTX(va);
        }
    }
}
```

- `boot_map_region`

Use the function above to map a contiguous memory. Remember once `pgdir_walk` is called, memory of size `PGSIZE` will be mapped.

We do not need to modify the `pp_ref` of physical page of address `*p`.

```
uintptr_t *p;
for (size_t i = 0; i < size; i += PGSIZE) {
    p = pgdir_walk(pgdir, (void *)(va + i), 1);
    if (p == NULL) {
        panic("Mapping failed\n");
    } else {
        *p = (pa + i) | perm | PTE_P;
    }
}
```

- `page_lookup()`

We need to check both the page table and the page exist.

```
uintptr_t *p = pgdir_walk(pgdir, va, 0);

// Check both the page table and the page exist
if (p == NULL || (*p & PTE_P) == 0) {
    return NULL;
} else {
    if (pte_store != 0) {
        *pte_store = p;
    }
    // Use PTE_ADDR() to cover some flags
    return pa2page(PTE_ADDR(*p));
}
```

- `page_remove()`

We can use the function `page_lookup()` to check if the page exists and, by the way, the page table entry.

Follow the hint and use `page_decref()` and `tlb_invalidate()` to complete this function.

```
uintptr_t *p;
struct PageInfo *pg = page_lookup(pgdir, va, &p);
if (pg == NULL) {
    return ;
} else {
    page_decref(pg);
    *p = 0;
    tlb_invalidate(pgdir, va);
}
```

- `page_insert()`

This function maps a physical address to a virtual address. We need to remember that increase the `pp_ref` first because if we are handling the situation that the same `pp` is re-inserted at the same virtual address in the same `pgdir`, removing it first will lead to freeing this page.

```
uintptr_t *p = pgdir_walk(pgdir, va, 1);
if (p == NULL) {
    // Cannot allocate
    return -E_NO_MEM;
}
// If allocation is successful, increase the pp_ref first.
// By doing this before page_remove, we can handle the situation that
// the same pp is re-inserted at the same virtual address in the same pgdir.
pp->pp_ref ++;
if ((*p & PTE_P) == 1) {
    // If there is a page
    page_remove(pgdir, va);
}

// Modify permission flags, page directory is also needed
*p = page2pa(pp) | perm | PTE_P;
*(pgdir + PDX(va)) |= perm;
return 0;
```

After `make qemu`, we will get

```
check_page() succeeded!
```

Exercise 5

- `mem_init()`

Remember that pointers `pages, bootstack` is virtual address, and we need to `PADDR()` it first.

```
boot_map_region(kern_pgdir, UPAGES, npages * sizeof(struct PageInfo), PADDR(pages), PTE_U | PTE_P);
```

```
boot_map_region(kern_pgdir, KSTACKTOP - KSTKSIZE, KSTKSIZE, PADDR(bootstack), PTE_W);
```

```
boot_map_region(kern_pgdir, KERNBASE, 0xffffffff - KERNBASE, 0, PTE_W);
```

- Q: What entries (rows) in the page directory have been filled in at this point? What addresses do they map and where do they point? In other words, fill out this table as much as possible:

  A: Follow the `memlayout.h`:

Entry	Base Virtual Address	Points to (logically):
1023	0xffc00000	Page table for top 4MB of phys memory
1022	0xff800000	?
.	?	?
960	0xf0000000	KERNBASE
.	?	?
2	0x00800000	Program Data & Heap
1	0x00400000	?
0	0x00000000	[see next question]

- Q: We have placed the kernel and user environment in the same address space. Why will user programs not be able to read or write the kernel's memory? What specific mechanisms protect the kernel memory?

  A: Because of the permission bits.

- Q: What is the maximum amount of physical memory that this operating system can support? Why?

  A: 4GiB. Page directory contains 1024 pointers, each pointer points to a page table containing 1024 pointers, and these pointers in page table point to physical page. The size of a page is 4096B, so the maximum size is $1024 * 1024 * 4096B = 4GiB$.

- Q: How much space overhead is there for managing memory, if we actually had the maximum amount of physical memory? How is this overhead broken down?

  A: For physical pages, 8MiB will be used for `struct PageInfo`, and 4MiB for page tables, finally 4KB for one page directory.

- Q: Revisit the page table setup in `kern/entry.S` and `kern/entrypgdir.c`. Immediately after we turn on paging, EIP is still a low number (a little over 1MB). At what point do we transition to running at an EIP above KERNBASE? What makes it possible for us to continue executing at a low EIP between when we enable paging and when we begin running at an EIP above KERNBASE? Why is this transition necessary?

  A: After instruction `jmp *%eax`. It is possible because in `entrypgdir.c`, it maps virtual address [0, 4M) to physical address [0, 4M). It is necessary because later a `kern_pgdir` will be loaded and va [0, 4M) will be abandoned.

Challenge 2

Challenge! Extend the JOS kernel monitor with commands to:

- Display in a useful and easy-to-read format all of the physical page mappings (or lack thereof) that apply to a particular range of virtual/linear addresses in the currently active address space. For example, you might enter `'showmappings 0x3000 0x5000'` to display the physical page mappings and corresponding permission bits that apply to the pages at virtual addresses 0x3000, 0x4000, and 0x5000.

First, modify the `monitor.h`:

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

Then modify the `monitor.c`:

Add this code in `commands[]`:

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

And its implementation:

```
int
mon_showmappings(int argc, char **argv, struct Trapframe *tf)
{
    extern pte_t *pgdir_walk(pde_t *pgdir, const void *va, int create);
    extern pde_t *kern_pgdir;

    if (argc != 2 && argc != 3) {
        cprintf("Usage: showmappings ADDR1 ADDR2\n       showmappings ADDR\n");
        return 0;
    }

    // Convert string to long and satisfy some assertion
    long begin = strtol(argv[1], NULL, 0);
    long end = argc == 3 ? strtol(argv[2], NULL, 0) : begin;
    if (end < begin) {
        long tmp = end;
        end = begin;
        begin = tmp;
    }
    if (end > 0xffffffff) {
        end = 0xffffffff;
    }
    begin = ROUNDUP(begin, PGSIZE);
    end = ROUNDUP(end, PGSIZE);

    for (; begin <= end; begin += PGSIZE) {
        cprintf("%08x---%08x: ", begin, begin + PGSIZE);
        pte_t *p = pgdir_walk(kern_pgdir, (void *)begin, 0);
        if (p == NULL) {
            cprintf("No mapping\n");

            continue;
```

```
        }
        cprintf("page %08x ", PTE_ADDR(*p));
        cprintf("PTE_P: %x, PTE_W: %x, PTE_U: %x\n", (bool)(*p & PTE_P), (bool)(*p & PTE_W), (bool)(*p & PTE_U));
    }

    return 0;
}
```

Test it:

```
K> showmappings 0xf0100000 0xf0110000
f0100000---f0101000: page 00100000 PTE_P: 1, PTE_W: 2, PTE_U: 0
f0101000---f0102000: page 00101000 PTE_P: 1, PTE_W: 2, PTE_U: 0
f0102000---f0103000: page 00102000 PTE_P: 1, PTE_W: 2, PTE_U: 0
f0103000---f0104000: page 00103000 PTE_P: 1, PTE_W: 2, PTE_U: 0
f0104000---f0105000: page 00104000 PTE_P: 1, PTE_W: 2, PTE_U: 0
f0105000---f0106000: page 00105000 PTE_P: 1, PTE_W: 2, PTE_U: 0
f0106000---f0107000: page 00106000 PTE_P: 1, PTE_W: 2, PTE_U: 0
f0107000---f0108000: page 00107000 PTE_P: 1, PTE_W: 2, PTE_U: 0
f0108000---f0109000: page 00108000 PTE_P: 1, PTE_W: 2, PTE_U: 0
f0109000---f010a000: page 00109000 PTE_P: 1, PTE_W: 2, PTE_U: 0
f010a000---f010b000: page 0010a000 PTE_P: 1, PTE_W: 2, PTE_U: 0
f010b000---f010c000: page 0010b000 PTE_P: 1, PTE_W: 2, PTE_U: 0
f010c000---f010d000: page 0010c000 PTE_P: 1, PTE_W: 2, PTE_U: 0
f010d000---f010e000: page 0010d000 PTE_P: 1, PTE_W: 2, PTE_U: 0
f010e000---f010f000: page 0010e000 PTE_P: 1, PTE_W: 2, PTE_U: 0
f010f000---f0110000: page 0010f000 PTE_P: 1, PTE_W: 2, PTE_U: 0
f0110000---f0111000: page 00110000 PTE_P: 1, PTE_W: 2, PTE_U: 0
```

- Explicitly set, clear, or change the permissions of any mapping in the current address space.

First, modify the `monitor.h`:

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

Then modify the `monitor.c`:

Add this code in `commands[]`:

```
{"setperm", "Set the permission bits of an addresses", mon_setperm },
```

And its implementation:

```
int
mon_setperm(int argc, char **argv, struct Trapframe *tf)
{
    extern pte_t *pgdir_walk(pde_t *pgdir, const void *va, int create);
    extern pde_t *kern_pgdir;


    if (argc != 4) {
```

```
        cprintf("Usage: setperm ADDR [clear|set] [P|W|U]\n      setperm ADDR [change] perm\n");
        return 0;
    }

    long addr = strtol(argv[1], NULL, 0);
    pte_t *p = pgdir_walk(kern_pgdir, (void *)addr, 0);
    cprintf("Before: ");
    cprintf("PTE_P: %x, PTE_W: %x, PTE_U: %x\n", (bool)(*p & PTE_P), (bool)(*p & PTE_W), (bool)(*p & PTE_U));

    int perm;
    if (strcmp(argv[2], "change") == 0) {
        cprintf("...Change permission bits...\n");
        perm = (int)strtol(argv[3], NULL, 0);
        *p = *p | perm;
    } else {
        if (argv[3][0] == 'P') perm = PTE_P;
        if (argv[3][0] == 'W') perm = PTE_W;
        if (argv[3][0] == 'U') perm = PTE_U;
        if (strcmp(argv[2], "clear") == 0){
            cprintf("...Clear permission bits...\n");
            *p = *p & (~perm);
        }
        if (strcmp(argv[2], "set") == 0) {
            cprintf("...Set permission bits...\n");
            *p = *p | perm;
        }
    }
    cprintf("After: ");
    cprintf("PTE_P: %x, PTE_W: %x, PTE_U: %x\n", (bool)(*p & PTE_P), (bool)(*p & PTE_W), (bool)(*p & PTE_U));
    return 0;
}
```

Test it:

```
K> setperm 0xf0100000 clear W
Before: PTE_P: 1, PTE_W: 1, PTE_U: 0
...Clear permission bits...
After: PTE_P: 1, PTE_W: 0, PTE_U: 0
K> setperm 0xf0100000 set U
Before: PTE_P: 1, PTE_W: 0, PTE_U: 0
...Set permission bits...
After: PTE_P: 1, PTE_W: 0, PTE_U: 1
```

- Dump the contents of a range of memory given either a virtual or physical address range. Be sure the dump code behaves correctly when the range extends across page boundaries!

First, modify the `monitor.h`:

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

Then modify the `monitor.c`:

Add this code in `commands[]` :

```
{"showmem", "Show the contents of a range of given memory", mon_showmem },
```

And its implementation:

```
int
mon_showmem(int argc, char **argv, struct Trapframe *tf)
{
    if (argc != 4) {
        cprintf("Usage: showmem [Virtual|Physical] ADDR num\n");
        return 0;
    }
    long addr = strtol(argv[2], NULL, 0);
    long vaddr = argv[1][0] == 'V' ? addr : (long)KADDR(PTE_ADDR((void *)addr));
    int n = (int)strtol(argv[3], NULL, 0);
    for (int i = 0; i < n; i +=4) {
        cprintf("%s Memory at %08x is %08x\n", argv[1], addr + i, *((int *)(vaddr + i)));
    }
    return 0;
}
```

Test it:

```
K> showmem V 0xf0100000 32
V Memory at f0100000 is 1badb002
V Memory at f0100004 is 00000000
V Memory at f0100008 is e4524ffe
V Memory at f010000c is 7205c766
V Memory at f0100010 is 34000004
V Memory at f0100014 is 5000b812
V Memory at f0100018 is 220f0011
V Memory at f010001c is c0200fd8
K> showmem P 0x00100000 32
P Memory at 00100000 is 1badb002
P Memory at 00100004 is 00000000
P Memory at 00100008 is e4524ffe
P Memory at 0010000c is 7205c766
P Memory at 00100010 is 34000004
P Memory at 00100014 is 5000b812
P Memory at 00100018 is 220f0011
P Memory at 0010001c is c0200fd8
```

- Do anything else that you think might be useful later for debugging the kernel. (There's a good chance it will be!)

I haven't come up with something useful yet :D. Maybe I will do it in later labs.

Grade

Use `./grade-lab2`

```
+ cc kern/pmap.c
+ ld obj/kern/kernel
ld: warning: section `.bss' type changed to PROGBITS
+ mk obj/kern/kernel.img
running JOS: (0.8s)
  Physical page allocator: OK
  Page management: OK
  Kernel page directory: OK
  Page management 2: OK
Score: 70/70
```