姜海天 19307110022

Part 1 Physical Page Management

Exercise 1

In this part, we ought to write a physical page allocator. We need a separate page allocator because we need to use page table to record the mapping between physical address and virtual address, but the page table is in virtual memory. So we use a page allocator to allocate memory in a fixed position.

1.1 `boot_alloc()`

This function will only be called when JOS is initializing the virtual memory. By looking at `mem_init()`, we can know that it is used for initialing page directory.

The following given code in `boot_alloc()` shows that if `nextfree` is not initialized, we set if to the first unused virtual address after the `end` by the times of page size. The comment says `end` points to the end of the kernel's bss segment. So this part is used for initialize `nextfree` pointer.

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

If `n` is 0, which means we don't want to allocate a size, we can just return the `nextfree` pointer.

Otherwise, we return `ROUNDUP((char *)(nextfree+n),PGSIZE)`. If `KERNBASE+npages*PGSIZE` is less than the address we get, it means we are running out of memory, so there should be a panic. So the code is:

```
if (n == 0) {
    return nextfree;
}
result = nextfree;
if ((uint32_t)result > (KERNBASE+npages*PGSIZE)){
    panic("boot_alloc: there is no enough space!\n");
}
nextfree += ROUNDUP(n, PGSIZE);
return result;
```

1.2 `mem_init()`

Here we have to use the struct `PageInfo`. This is apparently a linked list. What the variables mean are clearly put in the comments. Notice that the `pp_link` of non-free pages are always `NULL`.

```c
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.
    uint16_t pp_ref;
};
```

As we haven't implement `page_alloc()`, we still have to use `boot_alloc()` to allocate memory.

The code is quite simple.

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

1.3 `page_init()`

1.3.1 Mark physical page 0 as in use.

By mocking the given code, we can just set `pages[0]`.

```c
pages[0].pp_ref = 1;
pages[0].pp_link = page_free_list; // null
```

1.3.2 The rest of base memory, [PGSIZE, npages_basemem * PGSIZE) is free.

Other than `pages[0]`, all memory pages below `npages_basemem` are free, so just modify the given code.

```c
size_t i;
for (i = 1; i < npages_basemem; i++) {
    pages[i].pp_ref = 0;
    pages[i].pp_link = page_free_list;
    page_free_list = &pages[i];
}
```

1.3.3 Then comes the IO hole [IOPHYSMEM, EXTPHYSMEM), which must never be allocated.

Since the IO hole won't be allocated, we have to set it not free. Given by the comment, it starts from `IOPHYSMEM` and ends at `EXTPHYSMEM`. The code is still similar.

```c
for (i = IOPHYSMEM/PGSIZE; i < EXTPHYSMEM/PGSIZE; i++) {
    pages[i].pp_ref = 1;
    pages[0].pp_link = page_free_list; // null
}
```

1.3.4 Then extended memory [EXTPHYSMEM, ...)

We use `boot_alloc()` to find the first page that we can allocate. And we shall use `PADDR` use convert the virtual address from `boot_alloc()` to physical address. Then we set the `pages[]` accordingly.

```
size_t first_free_physical_address = PADDR(boot_alloc(0));
for (i = EXTPHYSMEM/PGSIZE; i < first_free_physical_address/PGSIZE; i++) {
    pages[i].pp_ref = 1;
    pages[0].pp_link = page_free_list; // null
}
for (i = first_free_physical_address/PGSIZE; i < npages; i++) {
    pages[i].pp_ref = 0;
    pages[i].pp_link = page_free_list;
    page_free_list = &pages[i];
}
```

1.4 `page_alloc()`

This function is for allocating pages by labeling the used pages. It is based on `PageInfo`, so it is just reading the head of the linked list, without really allocating.

```
if (page_free_list == NULL) {
    return NULL;
}
struct PageInfo* ret_page = page_free_list;
page_free_list = page_free_list->pp_link;
ret_page -> pp_link = NULL;
memset(page2kva(ret_page),0,PGSIZE);
if (alloc_flags & ALLOC_ZERO)
{
    memset(page2kva(ret_page),'\0',PGSIZE);
}
return ret_page;
```

1.5 `page_free()`

According to the hint, we can only free the page if its `pp_ref` is 0 and `pp_link` is `NULL`.

```
if (pp->pp_ref > 0 || pp->pp_link != NULL) {
    panic("Deallocating page failed");
    return;
}
pp->pp_link = page_free_list;
page_free_list = pp;
```

Part 2 virtual memory

Question 1: Assuming that the following JOS kernel code is correct, what type should variable `x` have, `uintptr_t` or `physaddr_t`?

Answer: Apparently it is a virtual address, so it should be `uintptr_t`.

Exercise 4

4.1 `pgdir_walk()`

The hint says, *the relevant page table page might not exist yet*. We should use the present bit to judge whether the page exists or not. The macro `PTE_P` defined in `mmu.h` is a bit mask for the present bit. So the present bit is got by `*pgdir_entry & PTE_P`. If it is true, then the page exists.

Also we need to add the permission bits for the entry.

Notice that we have to compulsory transfer the type to `pte_t`.

```
    pde_t * pgdir_entry = pgdir + PDX(va);
    bool page_exist = (*pgdir_entry & PTE_P);
    struct PageInfo * pginfo;
    if (!page_exist){
        if (!create) {
            return NULL;
        }
        pginfo = page_alloc(1);
        if (pginfo == NULL) {
            return NULL;
        }
        pginfo->pp_ref++;
        *pgdir_entry = page2pa(pginfo) | PTE_U | PTE_W | PTE_P;
    }
    return (pte_t *)KADDR(PTE_ADDR(*pgdir_entry)) + PTX(va);
```

4.2 `boot_map_region()`

The comment says that all the parameters have already been aligned, so it is greatly simplified.

We will map [va, va+size) of virtual address space to physical [pa, pa+size) by modifying the tree pointed by `pgdir`, so we just go through it once.

```
for (uintptr_t end = va+size; va != end; pa += PGSIZE,va += PGSIZE){
    pte_t *entry = pgdir_walk(pgdir,(const void*)va,1);
    if (entry == NULL){
        panic("pgdir_walk return NULL!");
    }
    *entry = pa | perm | PTE_P;
}
```

4.3 `boot_map_region_large()`

Similar to `boot_map_region()`

```
for (uintptr_t end = va+size; va != end; pa += PTSIZE, va += PTSIZE){
    pde_t * target_pde = pgdir + PDX(va);
    if ((*target_pde & (PTE_P | PTE_PS)) != (PTE_P | PTE_PS)){
        if(*target_pde & PTE_P){
            cprintf("DANGEROUS!COVER OLD PT,UNTRACK PT\n");
        }
        *target_pde = pa | perm | PTE_P | PTE_PS;
    }
}
```

But we need to use `cr4` to enable large page size. Add the following code in `entry.S`.

```
movl    %cr4,       %eax
orl     $(CR4_PSE), %eax
movl    %eax,       %cr4
```

4.4 `page_lookup()`

This function will get the page info mapped at the virtual address. Notice that we have to check the returned `*entry`, otherwise there will be an assertion error.

```
    pte_t * entry = pgdir_walk( pgdir, va, 0);
    if (!entry) {
        return NULL;
    }
    if (pte_store){
        *pte_store =  entry;
    }
    bool exist = *entry & PTE_P;
    if (!exist) {
        return NULL;
    }
    physaddr_t pa = PTE_ADDR(*entry);
    struct PageInfo* ret = pa2page(pa);
    return ret;
```

4.5 `page_remove()`

This function will remove the mapping between virtual address and the physical page. TLB is a buffer for speeding up looking-up procedure. We have to notice TLB that the mapping between the virtual address and the physical page is invalid. We use `tlb_invalidate()` for this purpose.

```
    pte_t *entry;
    struct PageInfo * pginfo = page_lookup(pgdir, va, &entry);
    if (!pginfo) {
        return;
    }
    page_decref(pginfo);
    *entry = 0;
    tlb_invalidate(pgdir,va);
```

4.6 `page_insert()`

If there is already a page mapped at `va`, it should be removed by `page_remove()`.
If necessary, on demand, a page table should be allocated and inserted into `pgdir`.
If the insertion succeeds, we increase `pp->pp_ref`.
If a page was formerly present at `va`, we invalidate its TLB.

Notice that the error code can be negative.

```
    pte_t *pgtab = pgdir_walk(pgdir, va, 1);
    if (!pgtab) {
        return -E_NO_MEM;
    }
    pp->pp_ref++;  // adding reference here is crucial
    if (*pgtab & PTE_P) {
        page_remove(pgdir, va);
    }
    *pgtab = page2pa(pp) | perm | PTE_P;
    return 0;
```

Part 3 Kernel Address Space

Exercise 5

1. Map 'pages' read-only by the user at linear address UPAGES

   Notice that there is only one page directory for now, which is `kern_pgdir`, so clearly the first parameter is it. The second parameter is the virtual address. The third parameter is the size of the mapped memory block. The fourth parameter is the physical address. The last parameter shows user has read access.

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

2. Kernel stack.

   `bootstack` is the lowest address of the stack, namely the top of stack. We need to map [KSTACKTOP-KSTKSIZE, KSTACKTOP) to virtual address.

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

3. Kernel base

   Map 0xf0000000 to 0xffffffff, a total of 256MB of memory.

```
boot_map_region_large(kern_pgdir, KERNBASE,
                ROUNDUP(0xffffffff-KERNBASE,PGSIZE),0,PTE_W|PTE_P);
```

Question 2: What entries (rows) in the page directory have been filled in at this point?

Answer:

Entry	Base Virtual Address	Points to (logically)
1023	0xffc00000	Page table for [252,256) MB of phys memory
...	...	...
961	0xf0400000	Page table for [4,8) MB of phys memory
960	0xf0000000	Page table for [0,4) MB of phys memory
959	0xefc00000	cpu0's kernel stack(0xefff8000)
958	0xef800000	ULIM
...	...	...
956	0xef000000	npages of PageInfo(0xef000000)
...	...	...
952	0xee000000	bootstack
...	...	...
2	0x00800000	Program Data & Heap
1	0x00400000	Empty Memory
0	0x00000000	Empty Memory

Question 3: 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?

Answer: There is permission bits in the page table entry. If we set `PTE_U` to 0, then user have no access to the kernel's memory.

Question 4: What is the maximum amount of physical memory that this operating system can support? Why?

Answer: `pages` is a array of type `PageInfo`. `pages` can take at most 4MB, while one `PageInfo` takes 8B, so there can be at most 512k pages. Since each page is 4K, the page table can hold at most 2GB of memory.

Question 5: 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?

Answer: 512k(pages) * 4B(PTE) + 4MB(array) + 4KB(PDE) $\approx$ 6MB

Question 6: 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?

Answer: After `jmp *%eax`, `EIP` is above `KERNBASE`. We can run when EIP is at a low address after enabling paging because VA[0,4MB) is also mapped to PA[0,4MB). This is necessary because operating system kernels often like to be linked and run at very high virtual address, such as 0xf0100000, in order to leave the lower part of the processor's virtual address space for user programs to use.

Challenge

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, add `int mon_showmappings(int argc, char **argv, struct Trapframe *tf);` to `monitor.h`.

Second, add to the array `commands`.

Third, implement the function `mon_showmappings`.

```
int
mon_showmappings(int argc, char **argv, struct Trapframe *tf)
{
    if (argc != 3) {
        cprintf("Requir 2 virtual address as arguments.\n");
        return -1;
    }
    char *errChar;
    uintptr_t start_addr = strtol(argv[1], &errChar, 16);
    if (*errChar) {
        cprintf("Invalid virtual address: %s.\n", argv[1]);
        return -1;
    }
    uintptr_t end_addr = strtol(argv[2], &errChar, 16);
    if (*errChar) {
        cprintf("Invalid virtual address: %s.\n", argv[2]);
        return -1;
    }
    if (start_addr > end_addr) {
        cprintf("Address 1 must be lower than address 2\n");
        return -1;
    }

    start_addr = ROUNDDOWN(start_addr, PGSIZE);
    end_addr = ROUNDUP(end_addr, PGSIZE);

    uintptr_t cur_addr = start_addr;
    while (cur_addr <= end_addr) {
        pte_t *cur_pte = pgdir_walk(kern_pgdir, (void *) cur_addr, 0);
        if ( !cur_pte || !(*cur_pte & PTE_P)) {
```

```
            cprintf( "Virtual address [%08x] - not mapped\n", cur_addr);
        } else {
            cprintf( "Virtual address [%08x] - physical address [%08x], permission: ", cur_addr, PTE_ADDR(*cur_pte));
            char perm_PS = (*cur_pte & PTE_PS) ? 'S':'-';
            char perm_W = (*cur_pte & PTE_W) ? 'W':'-';
            char perm_U = (*cur_pte & PTE_U) ? 'U':'-';
            cprintf( "-%c----%c%cP\n", perm_PS, perm_U, perm_W);
        }
        cur_addr += PGSIZE;
    }
    return 0;
}
```

Crucial point: use `strtol` to convert string to int.

running example:

```
K> showmappings 0xefff0000 0xf0000000
Virtual address [efff0000] - not mapped
Virtual address [efff1000] - not mapped
Virtual address [efff2000] - not mapped
Virtual address [efff3000] - not mapped
Virtual address [efff4000] - not mapped
Virtual address [efff5000] - not mapped
Virtual address [efff6000] - not mapped
Virtual address [efff7000] - not mapped
Virtual address [efff8000] - physical address [0010d000], permission: -------WP
Virtual address [efff9000] - physical address [0010e000], permission: -------WP
Virtual address [efffa000] - physical address [0010f000], permission: -------WP
Virtual address [efffb000] - physical address [00110000], permission: -------WP
Virtual address [efffc000] - physical address [00111000], permission: -------WP
Virtual address [efffd000] - physical address [00112000], permission: -------WP
Virtual address [efffe000] - physical address [00113000], permission: -------WP
Virtual address [effff000] - physical address [00114000], permission: -------WP
Virtual address [f0000000] - physical address [f000f000], permission: -------WP
```