Document of Lab2 in *Operating System*

Student ID: 7170302100006

March 28, 2019

Contents

0.1 Phisical Memory Allocator

In this section we implemented a set of functions that splits the entire physical memory into 4KB page frames and keeps these frames under management. Only after this can we set up a complete vitual memory mechanism.

We use a data structure **pages** to keep track of the position and using condition of every page frame, and a link list **page_free_list** to record the unused frames.

boot allocator

Before the entire physical page management mechanism is built, we need to implement a temporary physical page allocator so we can allocate the physical memory

required by the page management mechanism itself.

To achieve this we should firstly distinguish which physical memory is not occupied by the useful code and data. Since there is no data structure recording this information yet, we can only get this information from the previous process of booting the PC and loading the kernel.

In lab1 we know that when the kernel was just loaded there is only a small page table that maps merely 4MB virtual memory to physical memory, and the mapping relationship is linear. We also know the kernel's ELF is loaded at virtual address **0xf0100000**, and the kernel's stack is even below that address. There is not any other programs loaded yet, so we are confident that all virturl address after the kernel's ELF is free. And since the virtual address is mapped **linearly** to the physical address and there is no other process being executed currently, we can sure that using physical address mapped to the virtual address after the kernel's ELF is safe.

I disassembled the ELF of the kernel:

```
objdump -h obj/kern/kernel
```

And got this

```
Sections:
Idx Name          Size      VMA       LMA
  0 .text         00003fc9  f0100000  00100000

  1 .rodata       00001168  f0103fe0  00103fe0

  2 .stab         00006991  f0105148  00105148

  3 .stabstr      00001e2e  f010bad9  0010bad9

  4 .data         00009300  f010e000  0010e000

  5 .got          00000008  f0117300  00117300

  6 .got.plt      0000000c  f0117308  00117308

  7 .data.rel.local
                  00001000  f0118000  00118000
```

```
  8 .data.rel.ro.local
                  00000044   f0119000   00119000

  9 .bss          00000674   f0119060   00119060

 10 .comment      0000002a   00000000   00000000
```

The last section is .comment, which is useless after the kernel is loaded. So we can used the memory after the second last section .bss. And this address is provided by a magic number **end**.

```
extern char end[];
```

So we allocate the physical number from the physical address coressponding to the virtual address **end**. Use static variable **nextfree** to keep track of the used address. Note that in Lab1 we have turn on an incomplete page machenism, so every address we now use in C code should be virtual address except it's decleard as physaddr_t.

```
result = nextfree;
if(n > 0) {
    // update nextfree and keep it aligned
    nextfree += ((n - 1) / PGSIZE + 1) * PGSIZE;
    // or use:
    // nextfree += ROUNDUP(n, PGSIZE);
}
// If we're out of memory, panic.
if(PADDR(nextfree) >= npages * PGSIZE) {
    panic("Out of memory!\n");
}
return (void *)result;
```

allocate *pages*

Now we have boot_alloc() so we can allocate physical address for data structure **pages**.

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

initialize *pages*

We initialize the physical page management data structure by setting the already-used pages' **pp_ref** to 1 and insert the unused pages to **page_free_list**. We know which pages are taken now from the discription in the comments as well as from the function calling *boot_alloc(0)*, which returns the first unused page after physical address **0x100000** or virtual address **0xf0100000**.

```
physaddr_t firstfreepa = PADDR(boot_alloc(0));
for (i = 0; i < npages; i++) {
  // First physical address of current page frame
  physaddr_t pa = page2pa(pages + i);

  // If the page is free
  if((PGSIZE <= pa && pa < npages_basemem*PGSIZE) ||
      pa >= firstfreepa) {
      pages[i].pp_ref = 0;
      // Remember that page_free_list has
      // been automatically initlized to NULL,
      // so we needn't pay more attention
      // to deal with the tail of the link list.
      pages[i].pp_link = page_free_list;
      page_free_list = &pages[i];
  }

  // If the page is not free or can not be allocated
  else if(pa == 0 ||
            (IOPHYSMEM <= pa && pa < EXTPHYSMEM) ||
            pa < firstfreepa) {
      pages[i].pp_ref = 1;
      pages[i].pp_link = NULL;
  }
}
```

physical page alloc and free

Now we can write the allocator and freer with ease. To allocate a physical page, we first check the free list:

```
// If out of free memory
if(!page_free_list) {
    return NULL;
}
```

If there are free pages avaliable, we get it from the free list, and set the content to 0 if required.

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

To free a page, just insert the page to the free list. But it is important to check if the page can be freed legally.

```
if(pp->pp_ref) {
    panic("Error! Free a page with referrence!\n");
}
if(pp->pp_link) {
    panic("Error! Double free!\n");
}
```

0.2 Virtual Memory Management

Get Page Table Entry

It's time to set up page table, page directory and a complete virtual memory machenism. We have allocated a page directory in *mem_init()*.

```
kern_pgdir[PDX(UVPT)]=PADDR(kern_pgdir) | PTE_U | PTE_P;
```

Now we'll implement a function to get the address of a corresponding the page table entry given a virtual address. If the page table entry doesn't exist, create a new one.

```
pte_t *
pgdir_walk(pde_t *pgdir, const void *va, int create)
```

First calculate the address of the page directory entry by adding pgdir with the offset specified by the highest 10 bits of the virtual address.

```
pgdir += PDX(va);
```

Then get the page directory entry at that address, check the existence of the corresponding page table by testing if the PTE_P bit of the page directory entry is set to 1. If it exists, get the physical address of the page table entry by reading the highest 20 bits of the page directory entry and adding it with the offset specified by the middle 10 bits of the virtual address. Before returning cast physical address to virtual address because all addresses we'll **use** in C language code now should be virtual address.

```
pde_t pde = *pgdir;
// If the page table exists
if(pde & PTE_P) {
    // Before return, cast physical address to virtual
    return (pte_t *)(KADDR(PTE_ADDR(pde))) + PTX(va);
}
```

But if the page table doesn't exists, we should create one if the **create** tag is specified.

```
struct PageInfo *newpt_info = page_alloc(ALLOC_ZERO);
if(!newpt_info) {
    return NULL;
}
```

Also it is important to change **pp_ref** and set permission bits.

```
// increment the new page's referrence
newpt_info->pp_ref++;
// update the page directory entry
*pgdir = (uint32_t)page2pa(newpt_info);
// set the permissions a little more permissive
*pgdir|=(PTE_P|PTE_W|PTE_U|PTE_PWT|PTE_PCD|PTE_A);
// Before return, cast physical address to virtual
return (pte_t *)(page2kva(newpt_info)) + PTX(va);
```

Region Map

Now we'll implement a function to map a coutinuos virtual address sapce to a continious physical address space. It is an easy task. In every outer loop we find a new page table:

```
while(size > 0) {
```

```
        // Find a page table entry
        pt = pgdir_walk(pgdir, (void *)va, 1);
```

And in every inner loop we map a corresponding page.

```
*pt = (pte_t)pa;
*pt |= (perm|PTE_P);
// Update variables
va = ((uint32_t)va + PGSIZE);
pa += PGSIZE;
size -= PGSIZE;
pt++;
```

Large Region Map

It resembles *boot_map_region()*, but a page is 4MB this time instead of 4KB and the physical address of the page frame is stored in page directory instead of page table.

```
pgdir = &pgdir[PDX(va)];
while(size > 0) {
    *pgdir = pa|perm|PTE_P|PTE_PS;
    size -= PTSIZE;
    pa += PTSIZE;
    pgdir++;
}
```

Page Looking up

page_lookup() returns a page in which the given virtual address *va* exists. The essential point is to test the error cases, such as the page table doesn't exist or the virtual address is not mapped.

```
// If the page table dosen't exist and alloc failed
if(!pte_ptr) {
    return NULL;
}

// If no page mapped at va, reutrn NULL;
if(!(*pte_ptr & PTE_P)) {
    return NULL;
```

```
    }
    return pa2page((PTE_ADDR((*pte_ptr))));
```

Page Removing

Moving a page is easy but remember to check the case in which the given virtual address is not mapped.

```
if(!pg_info) {
    return;
}
page_decref(pg_info);
```

Page Inserting

To insert a page we just need to modify the corresponding page table entry.

```
*pte_ptr = (uint32_t)page2pa(pp);
*pte_ptr |= (perm|PTE_P);
```

But the most significant point is handle the corner case in which the same pp is re-inserted at the same virtual address in the same pgdir. I did not find that elegant method to handle everything in one way...So I solve this problem directly...and don't forget to reset the permission bits if **va** is already mapped to **pp**.

```
// If va is mapped
if(*pte_ptr & PTE_P) {
    // If va is already mapped to pp,
    // reset the permission and return
    if(pa2page(PTE_ADDR(*pte_ptr)) == pp) {
        // clear the old permission, important
        *pte_ptr &= (~0x00000FFF);
        *pte_ptr |= (perm|PTE_P);
        return 0;
    }
    page_remove(pgdir, va);
}
```

0.3 Kernel Address Space

We get the corresbonding areas mapped as the comments in *mem_init()* instructs.

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

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

In function *check_kern_pgdir()* I found that all physical address should be mapped to kernel base space by large page size:(*Not until T.A. updated the lab text had I known this is one of the* ***challenges***.)

```
boot_map_region_large(kern_pgdir,
KERNBASE, 0x100000000 - KERNBASE, 0, PTE_W|PTE_P);
.
```

And now, the most significant thing. Since we have used a 4MB page size, we should change regiser *%cr4* to invoke the page extension of the hardware.

```
cr4 = rcr4();
cr4 |= CR4_PSE;
lcr4(cr4);
```

This completes a basic memory management system for JOS.

0.4 Challenge

I choose the first one... As a matter of fact I didn't even realize that it's a challenge when I was trying to solve it because there was not any discription of any challenges in the lab text of the website. I was wondering why can't I get the last 20 scores and I looked into check_kern_pgdir() and find that it checks a large page machenism. So I used the page directory instead of page table to map 4MB pages, and reset register cr4 to invoke the mmu of thin function.

0.5 Conclusion

In this Lab I complete a memory management machenism under the guidance of the lab text and the comments in the souce code. Though having a hardship coding and debugging(and writing this document, from which I sufferred most...), I gain a relatively good command of how the memory management system works in an OS. It's worth sufferring.