JOS 实习 Lab2 报告

1100012836 崔治丞

1. Part 1: Physical Page Management

1.1 Exercise1

2. Part 2: Virtual Memory

2.1 Exercise 2
2.2 Exercise 3
2.3 Question
2.4 Exercise 4

3. Part 3: Kernel Address Space

3.1 Exercise 5
3.2Question

4. Challenge

1 Part 1: Physical Page Management

1.1Exercise 1

Exercise 1. In the file `kern/pmap.c`, you must implement code for the following functions (probably in the order given).

`boot_alloc()`
`mem_init()` (only up to the call to `check_page_free_list(1)`)
`page_init()`
`page_alloc()`
`page_free()`

`check_page_free_list()` and `check_page_alloc()` test your physical page allocator. You should boot JOS and see whether `check_page_alloc()` reports success. Fix your code so that it passes. You may find it helpful to add your own `assert()`s to verify that your assumptions are correct.

boot_alloc:

```
        // LAB 2: Your code here.
//---------------------------------------My Code-----------------------------------------------------------
        result = nextfree;
        if (n > 0)
            nextfree = ROUNDUP((char*)(nextfree + n), PGSIZE);
        return result;
//---------------------------------------My Code-----------------------------------------------------------
        return NULL;
```

mem_init(only up to the call to check_page_free_list(1)):

```
//---------------------------------------My Code-----------------------------------------------------------
        pages = (struct PageInfo *)boot_alloc(sizeof(struct PageInfo) * npages);
        //Array of PageInfo
        //cprintf("npages: %d\n", npages);
        //cprintf("pages: %x\n", pages);
//---------------------------------------My Code-----------------------------------------------------------
```

page_init:

```
//---------------------------------------My Code-----------------------------------------------------------
        //extern char end[];
        size_t i;
        //cprintf("address and IOPHYSMEM: %x -- %x -- %x\n", npages_basemem * PGSIZE, IOPHYSMEM, EXTPHYSMEM);
        //It shows that npages_basemem * PGSIZE == IOPHYSMEM
        page_free_list = NULL;
        pages[0].pp_ref = 1;
        pages[0].pp_link = page_free_list;
        for (i = 1; i < npages_basemem; i++) {
                pages[i].pp_ref = 0;
                pages[i].pp_link = page_free_list;
                page_free_list = &pages[i];
        }
        //boot_alloc(0) returns the pointer nextfree points to the next free page. Already aligned.
        //So we should initialize pages after usedpages.
        size_t usedpages = PADDR(boot_alloc(0)) / PGSIZE;
        for (i = usedpages; i < npages ;i++) {
                pages[i].pp_ref = 0;
                pages[i].pp_link = page_free_list;
                page_free_list = &pages[i];
        }
//---------------------------------------My Code-----------------------------------------------------------
```

page_init 初始化空闲页的链表结构。PageInfo 结构体管理页。通过注释，我们知道第 0 页和 IO hole 是不可以分配的。而且[EXTPHYSMEM, ...)也有部分空间不是空闲的。通过输出查询 ，得知 EXTPHYSMEM 的值为 0x100000，而通过 lab1 中的信息我们知道内核就是加载到这个地址上的，内核的结束地址用 extern char exd[]保存（这里 end 是虚拟地址，为 0xf0118970，所以实际的物理地址就是 end - KERNBASE 为 0x118970）。再通过 boot_alloc 函数分配空闲页时，是从 end 位置开始的。而在 mem_init 函数中，先为页目录分配了一页，又为 PageInfo 结构体分配了若干页。这些页都是已分配的页，不能再做初始化，所以除了第 0 页，IO hole 和上述的已分配页，其余页都是需要初始化的。

page_alloc:

```
struct PageInfo *
page_alloc(int alloc_flags)
{
        // Fill this function in
//-------------------------------------My Code-------------------------------------------------------------

        if (page_free_list != NULL) {
                struct PageInfo *newpage = page_free_list;
                page_free_list = newpage->pp_link;
                if (alloc_flags & ALLOC_ZERO)
                        memset(page2kva(newpage), 0, PGSIZE);
                return newpage;
        }
        else return NULL;

//-------------------------------------My Code-------------------------------------------------------------
        return 0;
}
```

page_free:

```
void
page_free(struct PageInfo *pp)
{
        // Fill this function in
//-------------------------------------My Code-------------------------------------------------------------
        if (pp->pp_ref != 0)
                assert(pp->pp_ref != 0);
        pp->pp_link = page_free_list;
        page_free_list = pp;
//-------------------------------------My Code-------------------------------------------------------------
}
```

2 Part 2: Virtual Memory

2.1Exercise2

> **Exercise 2.** Look at chapters 5 and 6 of the Intel 80386 Reference Manual, if you haven't done so already. Read the sections about page translation and page-based protection closely (5.2 and 6.4). We recommend that you also skim the sections about segmentation; while JOS uses paging for virtual memory and protection, segment translation and segment-based protection cannot be disabled on the x86, so you will need a basic understanding of it.

操作系统及软件所操作的地址都是虚拟地址（逻辑地址），根据系统机制的不同而进行不同的地址转换。采用段页式管理，则先将虚拟地址（逻辑地址）通过段式转换为线性地址，然后再采用页式转换为物理地址。

2.2Exercise3

Exercise 3. While GDB can only access QEMU's memory by virtual address, it's often useful to be able to inspect physical memory while setting up virtual memory. Review the QEMU monitor commands from the lab tools guide, especially the `xp` command, which lets you inspect physical memory. To access the QEMU monitor, press `Ctrl-a c` in the terminal (the same binding returns to the serial console).

Use the `xp` command in the QEMU monitor and the `x` command in GDB to inspect memory at corresponding physical and virtual addresses and make sure you see the same data.

Our patched version of QEMU provides an `info pg` command that may also prove useful: it shows a compact but detailed representation of the current page tables, including all mapped memory ranges, permissions, and flags. Stock QEMU also provides an `info mem` command that shows an overview of which ranges of virtual memory are mapped and with what permissions.

Info pg

```
K> QEMU 0.15.0 monitor - type 'help' for more information
(qemu) info pg
VPN range     Entry         Flags        Physical page
[ef000-ef3ff]  PDE[3bc]     -------UWP
  [ef000-ef3ff]  PTE[000-3ff] -------U-P 0011a-00519
[ef400-ef7ff]  PDE[3bd]     -------U-P
  [ef7bc-ef7bc]  PTE[3bc]     -------UWP 003fd
  [ef7bd-ef7bd]  PTE[3bd]     -------U-P 00119
  [ef7bf-ef7bf]  PTE[3bf]     -------UWP 003fe
  [ef7c0-ef7d0]  PTE[3c0-3d0] ----A--UWP 003ff 003fc 003fb 003fa 003f9 003f8 ..
  [ef7d1-ef7ff]  PTE[3d1-3ff] -------UWP 003ec 003eb 003ea 003e9 003e8 003e7 ..
[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-f0117]  PTE[116-117] --------WP 00116-00117
  [f0118-f0119]  PTE[118-119] ---DA---WP 00118-00119
  [f011a-f011a]  PTE[11a]     ----A---WP 0011a
  [f011b-f011b]  PTE[11b]     ---DA---WP 0011b
  [f011c-f013a]  PTE[11c-13a] ----A---WP 0011c-0013a
  [f013b-f03bd]  PTE[13b-3bd] ---DA---WP 0013b-003bd
  [f03be-f03ff]  PTE[3be-3ff] --------WP 003be-003ff
[f0400-f3fff]  PDE[3c1-3cf] ----A--UWP
  [f0400-f3fff]  PTE[000-3ff] ---DA---WP 00400-03fff
[f4000-f43ff]  PDE[3d0]     ----A--UWP
  [f4000-f40fe]  PTE[000-0fe] ---DA---WP 04000-040fe
  [f40ff-f43ff]  PTE[0ff-3ff] --------WP 040ff-043ff
[f4400-fffff]  PDE[3d1-3ff] -------UWP
  [f4400-fffff]  PTE[000-3ff] --------WP 04400-0ffff
(qemu)
```

2.3Question1

Question

1. 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;
```

x 保存的是一个指针，所以应该是 uintptr_t 类型

2.4Exercise 4

Exercise 4. In the file kern/pmap.c, you must implement code for the following functions.

```
pgdir_walk()
boot_map_region()
page_lookup()
page_remove()
page_insert()
```

check_page(), called from mem_init(), tests your page table management routines. You should make sure it reports success before proceeding.

page_walk:

```
pte_t *
pgdir_walk(pde_t *pgdir, const void *va, int create)
{
        // Fill this function in
//----------------------------------------My Code-----------------------------------------------------------
        int dir_index, table_index;
        pte_t *result;
        struct PageInfo *newpage;
        dir_index = PDX(va);
        table_index = PTX(va);
        if (pgdir[dir_index] & PTE_P)   {
                result = KADDR(PTE_ADDR(pgdir[dir_index]));
                //PTE is a physical address, but we need a virtual address.
                result += table_index;
                return result;
        }
        else if (create != false) {
                newpage = page_alloc(ALLOC_ZERO);
                if (newpage == NULL) return NULL;       //allocation fail
                newpage->pp_ref++;
                pgdir[dir_index] = page2pa(newpage) | PTE_P | PTE_U | PTE_W;
                //make the new page PTE_U and PTE_W so we can pass the check.
                result = KADDR(PTE_ADDR(pgdir[dir_index]));
                result += table_index;
                return result;
        }
        return NULL;
//----------------------------------------My Code-----------------------------------------------------------
}
```

page_walk 的作用是返回给定虚拟地址所对应的 page table entry 指针。但是所对应的 page table 可能不在 page directory 中，此时就要根据 create 来决定是否新建一个 page table。若确实要新建一个 page table，则还要将其加入道 page directory 中，并设定其保护位，再返回所需要的指针。这里需要注意的是，page directory 和 page tabl 存储的都是实际的物理地址，而系统所能使用的都是虚拟地址，所以必须在返回前使用 KADDR 函数将物理地址转换为虚拟地址，

boot_map_region

```
static void
boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa, int perm)
{
        // Fill this function in
//--------------------------------------My Code-------------------------------------------------------------
        int i;
        pte_t *newpage;
        uintptr_t temp_va = va;
        physaddr_t temp_pa = pa;
        for (i = 0; i < size / PGSIZE;  ++i)    {
                newpage = pgdir_walk(pgdir, (void*)temp_va , true);             //map page by page
                if (!newpage)                                                   //pgdir_walk fail
                        panic("boot_map_region, pgdir_walk failed!");
                *newpage = temp_pa | perm | PTE_P;
                temp_va += PGSIZE; temp_pa += PGSIZE;
                //must increment temp_va and temp_pa after those instructions above.
        }
//--------------------------------------My Code-------------------------------------------------------------
}
```

boot_map_region 函数将虚拟地址[va, va + size)映射到物理地址[pa, pa + size)上，并加以给定的权限。我们知道页式管理是通过页目录和页表将虚拟地址转换道物理地址的，所以这里我们就需要修改页表了。通过 page_walk 函数找到给定虚拟地址所对应的 page table entry 的指针，然后修改其指向的值即可。

page_lookup

```
struct PageInfo *
page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
{
        // Fill this function in
//--------------------------------------My Code-------------------------------------------------------------
        pte_t *pte;
        pte = pgdir_walk(pgdir, va, false);
        if (!pte)                       //no page
                return NULL;
        if (!(*pte & PTE_P))            //no page
                return NULL;
        if (!pte_store)
                panic("pte_store is empty!");
        *pte_store = pte;
        return pa2page(PTE_ADDR((*pte)));

//--------------------------------------My Code-------------------------------------------------------------
}
```

给定虚拟地址，返回其对应的页的 PageInfo。而实际上一个页的物理地址就是对应虚拟地址通过页式转换查到的 PTE 的高 20 位。通过 page_walk 函数得到 pte 后，再通过 pa2page 函数将高 20 位转化为 PageInfo 结构返回即可。

page_remove

```
void
page_remove(pde_t *pgdir, void *va)
{
	// Fill this function in
//-------------------------------------My Code-----------------------------------------------------------
	pte_t *pte_store_n;
	struct PageInfo *oldpage;
	oldpage = page_lookup(pgdir, va, &pte_store_n);
	if (oldpage && ((*pte_store_n) & PTE_P)) {	//if the page exist
		page_decref(oldpage);			//In the function page_decref(), it first decrements the pp_ref
							//If pp_ref == 0 then free the page
		*pte_store_n = 0;			//set the p table entry to 0
		tlb_invalidate(pgdir, va);		//invalidate the TLB
	}

//-------------------------------------My Code-----------------------------------------------------------
}
```

删除给定虚拟地址的映射。首先要删除对应的 pte 的信息，然后将对应页的引用次数减少，最后更新 tlb。

page_insert

```
int
page_insert(pde_t *pgdir, struct PageInfo *pp, void *va, int perm)
{
	// Fill this function in
//-------------------------------------My Code-----------------------------------------------------------
	pte_t *newpage;

	newpage = pgdir_walk(pgdir, va, true);
	if (!newpage)
		return -E_NO_MEM;
	pp->pp_ref++;			//To avoid the corner-case, we increment the pp_ref first.
	if (newpage && (*newpage & PTE_P)) {
		page_remove(pgdir, va);
	}
	*newpage = page2pa(pp) | perm | PTE_P;
	return 0;

//-------------------------------------My Code-----------------------------------------------------------
}
```

page_insert 增加给定虚拟地址到给定页的映射，同样需要修改对应页表的 pte，若该虚拟地址 va 已经有了映射，则需要先删除原有映射。提示中提到了一个情况，就是给定虚拟地址 va 到给定页的映射已经存在，若先进行删除映射操作，会把给定页删除，这是我们不希望看到的。通过查看 page_remove 和 page_decref 函数，可以知道，只有当页的引用数为 0 时，才会真正 free 掉。所以我们可以在 page_insert 中先将给定页的引用数增加，再判断是否有映射重复问题，这样即使上述映射存在，给定页的引用数也是先增加再减少，不会被 free 掉，也就不会产生问题。

3 Part 3: Kernel Address Space

按照提示中的划分，JOS 系统只有 ULIM 以下的地址空间是用户可以使用的。

其中：

1. [UPAGES， UPAGES + PTSIZE) 映射到[pages, pages + PTSIZE)上。

2. [KSTACKTOP-KSTKSIZE, KSTACKTOP) 映射到[bootstack, bootstacktop)上。另，[KSTACKTOP - PTSIZE, KSTACKTOP - KSTKSIZE）是作为 guard page 用来监测内核栈溢出。
3. [KERNBASE, 2^32)映射到[0, 2^32 - KERNBASE)

。

3.1Exercise5

Exercise 5. Fill in the missing code in `mem_init()` after the call to `check_page()`.

Your code should now pass the `check_kern_pgdir()` and `check_page_installed_pgdir()` checks.

代码如下：

```
//-------------------------------------My Code-------------------------------------------------------------
	boot_map_region(kern_pgdir, UPAGES, PTSIZE, PADDR(pages), PTE_U);
//-------------------------------------My Code-------------------------------------------------------------

//-------------------------------------My Code-------------------------------------------------------------
	boot_map_region(kern_pgdir, KSTACKTOP - KSTKSIZE, KSTKSIZE, PADDR(bootstack), PTE_W);
//-------------------------------------My Code-------------------------------------------------------------

//-------------------------------------My Code-------------------------------------------------------------
	boot_map_region(kern_pgdir, KERNBASE, 0 - KERNBASE, 0, PTE_W);
//-------------------------------------My Code-------------------------------------------------------------
```

3.2Question 2

Q2

2. 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:

Entry	Base Virtual Address	Points to (logically):
1023	?	Page table for top 4MB of phys memory
1022	?	?
.	?	?
.	?	?
.	?	?
2	0x00800000	?
1	0x00400000	?
0	0x00000000	[see next question]

进入 qemu 的控制台后，利用 info pg 命令查看映射：

```
K> QEMU 0.15.0 monitor - type 'help' for more information
(qemu) info pg
VPN range     Entry         Flags        Physical page
[ef000-ef3ff]  PDE[3bc]     -------UWP
  [ef000-ef3ff]  PTE[000-3ff] -------U-P 0011a-00519
[ef400-ef7ff]  PDE[3bd]     -------U-P
  [ef7bc-ef7bc]  PTE[3bc]     -------UWP 003fd
  [ef7bd-ef7bd]  PTE[3bd]     -------U-P 00119
  [ef7bf-ef7bf]  PTE[3bf]     -------UWP 003fe
  [ef7c0-ef7d0]  PTE[3c0-3d0] ----A--UWP 003ff 003fc 003fb 003fa 003f9 003f8 ..
  [ef7d1-ef7ff]  PTE[3d1-3ff] -------UWP 003ec 003eb 003ea 003e9 003e8 003e7 ..
[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-f0117]  PTE[116-117] --------WP 00116-00117
  [f0118-f0119]  PTE[118-119] ---DA---WP 00118-00119
  [f011a-f011a]  PTE[11a]     ----A---WP 0011a
  [f011b-f011b]  PTE[11b]     ---DA---WP 0011b
  [f011c-f013a]  PTE[11c-13a] ----A---WP 0011c-0013a
  [f013b-f03bd]  PTE[13b-3bd] ---DA---WP 0013b-003bd
  [f03be-f03ff]  PTE[3be-3ff] --------WP 003be-003ff
[f0400-f3fff]  PDE[3c1-3cf] ----A--UWP
  [f0400-f3fff]  PTE[000-3ff] ---DA---WP 00400-03fff
[f4000-f43ff]  PDE[3d0]     ----A--UWP
  [f4000-f40fe]  PTE[000-0fe] ---DA---WP 04000-040fe
  [f40ff-f43ff]  PTE[0ff-3ff] --------WP 040ff-043ff
[f4400-fffff]  PDE[3d1-3ff] -------UWP
  [f4400-fffff]  PTE[000-3ff] --------WP 04400-0ffff
(qemu)
```

通过查询 memlayout.h 文件以及 lab1 中关于内核加载的内容，可以分析出这些地址锁指向的内容。

VA	Point to
0xef000000 - 0xef020fff	PAGES
0xefbc0000 - 0xef7fffff	Cur. Page Table (User R-)
0xefff8000 - 0x0xffffffff	Kernel stack and guard pages
0xf0000000 - 0xffffffff	KERNEL

Q3

3. (From Lecture 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?

kernel memory 的页的 PTE_U 设为 0，用户程序不能访问。

Q4

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

通过 memlayout.h 中的图例可知，UPAGES 的大小为 PTSIZE = 4MB ，而一个 PageInfo 的大小为 8B，所以一共可以管理 4MB / 8B = 512K 个页，一个页为 4KB 大小，所以总共可管理的物理存储空间为 512K * 4KB = 2GB

Q5

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?

由 Q4 可知，物理内存一共 2GB 大小，即 512K 个页，需要 512K 个 PageInfo 结构体进行管理，则一共需要 4MB 的空间保存这些 PageInfo，同时页表中共有 512K 项，则页表大小为 512K * 4B = 2MB，页表一共 2MB / 4KB = 512 个页，对应页目录需要 1 页。

则总共需要 4MB + 2MB + 4KB = 6MB + 4KB 大小的空间。

Q6

6. 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?

在 entry.S 中有如下代码：

```
mov    $relocated, %eax
jmp    *%eax
```

mov 之后，eip 的值已经变为 KERNBASE 之上。

JOS 将虚拟地址[0, 4MB)和[KERNBASE, KERNBASE + 4MB)都映射到[0, 4MB)上，所以 eip 继续保持低值也是可行的。但将 eip 转化到高值是必须的，因为 kern_pgdir 被加载后，虚拟地址[0, 4MB)到物理地址[0, 4MB)的映射就被修改了，此后若 eip 仍为低值，程序就无法继续运行了。

至此，lab2 的所有 exercise 完成，得分如下：

```
czc@ubuntu:~/test/lab$ ./grade-lab2
+ cc kern/monitor.c
+ ld obj/kern/kernel
+ mk obj/kern/kernel.img
running JOS: (1.5s)
  Physical page allocator: OK
  Page management: OK
  Kernel page directory: OK
  Page management 2: OK
Score: 70/70
czc@ubuntu:~/test/lab$
```

4 Challenge

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.
- Explicitly set, clear, or change the permissions of any mapping in the current address space.
- 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!
- Do anything else that you think might be useful later for debugging the kernel. (There's a good chance it will be!)

为了处理命令行中的参数，我编写了下列函数：

```c
uint32_t myxtou_lab2(char *buf)         //transfer address to number
{
        uint32_t r = 0;
        buf += 2;               //buf = "0x....."
        while (*buf)    {
                r = r * 16;
                if (*buf >= 'a')
                        r = r + *buf - 'a' + 10;                        //a...f
                else
                        r += *buf - 48;                                 //0...9
                buf++;
        }
        return r;
}

void myprint_lab2(pte_t *pte)           //print the permission bits
{
        cprintf("PTE_P: %x, PTE_W: %x, PTE_U: %x\n", *pte & PTE_P, (*pte & PTE_W) >> 1, (*pte & PTE_U) >> 2);
}

bool check_pa_lab2(uint32_t pa, uint32_t *result)               //infomation from pmap.c/mem_int
{
        if (pa >= PADDR(pages) && pa < PADDR(pages) + PTSIZE)   {
                *result = *(uint32_t *)(pa - PADDR(pages) + UPAGES);
                return true;
        }
        if (pa >= PADDR(bootstack) && pa < PADDR(bootstack) + KSTKSIZE) {
                *result = *(uint32_t *)(pa - PADDR(bootstack) + KSTACKTOP - KSTKSIZE);
                return true;
        }
        if (pa >= 0 && pa < 0 - KERNBASE)       {
                *result = *(uint32_t *)(pa + KERNBASE);
                return true;
        }
        return false;
}
```

myxtou_lab2 函数用于将形如 0x...的 16 进制字符串转换为数字

myprint_lab2 函数用于输出 permission bit

check_pa_lab2 函数用于检查给定物理地址是否合法

showmappings 函数需要输出给定虚拟地址对应的物理地址，只需要利用 pgdir_walk 函数即可。这里为了调试后续功能的方便，也输出了对应页的 permission bit。

```c
int
mon_showmappings(int argc, char **argv, struct Trapframe *tf)
{
        pte_t *pte;
        if (argc != 3) {
                cprintf("Your input should be: showmappings begin_addr(0x...) end_addr(0x...)\n");
                return 0;
        }
        uint32_t left = myxtou_lab2(argv[1]);                           //begin address
        uint32_t right = myxtou_lab2(argv[2]);                          //end address
        cprintf("left: %x  right: %x\n", left, right);
        if (left > right)                                               //check
        {
                cprintf("Wrong Input! Your begin_addr should be less than end_addr!\n");
                return 0;
        }
        for (; left <= right; left += PGSIZE)                           //display the mapping page by page
        {
                pte = pgdir_walk(kern_pgdir, (void*)left, true);
                if (!pte || ((*pte & PTE_P) == 0))
                {
                        cprintf("page not exist: %x\n", left);
                }
                else
                {
                        cprintf("page %x mapped to 0x%08x with permission: ", left, PTE_ADDR(*pte));
                        myprint_lab2(pte);
                }
        }
        return 0;
}
```

结果如下：

```
K> showmappings 0xf0000000 0xf0004000
0xf0000000

0xf0004000

left: f0000000  right: f0004000
page f0000000 mapped to 0x00000000 with permission: PTE_P: 1, PTE_W: 1, PTE_U: 0
page f0001000 mapped to 0x00001000 with permission: PTE_P: 1, PTE_W: 1, PTE_U: 0
page f0002000 mapped to 0x00002000 with permission: PTE_P: 1, PTE_W: 1, PTE_U: 0
page f0003000 mapped to 0x00003000 with permission: PTE_P: 1, PTE_W: 1, PTE_U: 0
page f0004000 mapped to 0x00004000 with permission: PTE_P: 1, PTE_W: 1, PTE_U: 0
K> qemu: terminating on signal 2
```

setperm 函数修改给定虚拟地址对应的物理页的 permission bit，同样利用 pgdir_walk 函数找到对应页，再根据用户输入修改其 permission bit

```
int
mon_setperm(int argc, char **argv, struct Trapframe *tf)
{
        if (argc != 6) {
                cprintf("Your input should be: setperm addr(0x...) [0|1](clear or set) [0|1](P) [0|1](W) [0|1]
(U)\n");
                return 0;
        }
        uint32_t addr = myxtou_lab2(argv[1]);                   //address
        pte_t *pte;
        pte = pgdir_walk(kern_pgdir, (void*)addr, true);        //find pte
        cprintf("Address: 0x%08x\n", addr);
        cprintf("Old Permission: ");
        myprint_lab2(pte);                                      //print permission
        uint32_t perm = 0;
        if (argv[3][0] == '1') perm |= PTE_P;
        if (argv[4][0] == '1') perm |= PTE_W;
        if (argv[5][0] == '1') perm |= PTE_U;
        if (argv[2][0] == '0') *pte = *pte & ~perm;             //change permission
        else *pte = PTE_ADDR(*pte) | perm;
        cprintf("New Permission: ");
        myprint_lab2(pte);                                      //print new permission
        return 0;
}
```

结果如下：

```
K> setperm 0xf0004000 1 1 1 1
0xf0004000

Address: 0xf0004000
Old Permission: PTE_P: 1, PTE_W: 1, PTE_U: 0
New Permission: PTE_P: 1, PTE_W: 1, PTE_U: 1
K> setperm 0xf0004000 0 1 1 0
0xf0004000

Address: 0xf0004000
Old Permission: PTE_P: 1, PTE_W: 1, PTE_U: 1
New Permission: PTE_P: 0, PTE_W: 0, PTE_U: 1
K> qemu: terminating on signal 2
```

dump 函数用于输出给定地址的内容，要区分虚拟地址和物理地址。这里为了方便处理，要求用户自己输入是虚拟地址还是物理地址。若是虚拟地址则直接输出其值即可。物理地址，则需要首先检查其合法性，再通过地址转换输出其内容。

```
int
mon_dump(int argc, char **argv, struct Trapframe *tf) {
        if (argc != 4) {
                cprintf("Yout input should be: dump [v|p](virtual or physical) begin_addr(0x...) length(0x...)
\n");
                return 0;
        }
        uint32_t addr = myxtou_lab2(argv[2]);
        uint32_t len = myxtou_lab2(argv[3]);
        pte_t *pte;
        addr = ROUNDDOWN(addr, 4);
        if (argv[1][0] == 'v') {                        //virtual memory just show the content
                int i;
                for (i = 0; i < len; i++){
                        if (i % 4 == 0) cprintf("virtual memory %08x: ", addr + 4 * i);
                        pte = pgdir_walk(kern_pgdir, (void*)ROUNDDOWN(addr + i * 4, PGSIZE), 0);
                        if (pte && (*pte & PTE_P))
                                cprintf("0x%08x ", *(uint32_t *)(addr + 4 * i));
                        else cprintf("-------- ");
                        if (i % 4 == 3) cprintf("\n");
                }
        }
        if (argv[1][0] == 'p') {                        //physical memory, first check the address
                int i;                                  //then show the content
                uint32_t result;
                for (i = 0; i < len; i++){
                        if (i % 4 == 0) cprintf("physical memory %08x: ", addr + 4 * i);
                        if (check_pa_lab2(addr + i * 4, &result))       //my function to check the address
                                cprintf("0x%08x ", result);
                        else
                                cprintf("-------- ");
                        if (i % 4 == 3) cprintf("\n");
                }
        }
        return 0;
}
```

结果如下：

```
K> dump v 0xefff8000 0xc
virtual memory efff8000: 0x00000000 0x00000000 0x00000000 0x00000000
virtual memory efff8010: 0x00000000 0x00000000 0x00000000 0x00000000
virtual memory efff8020: 0x00000000 0x00000000 0x00000000 0x00000000
K> dump v 0xf0000000 0x4
virtual memory f0000000: 0xf000ff53 0xf000ff53 0xf000e2c3 0xf000ff53
K> dump p 0x0 0x4
physical memory 00000000: 0xf000ff53 0xf000ff53 0xf000e2c3 0xf000ff53
K>
```

这里在处理物理地址的时候遇到了困难，因为没有注意到不同段的物理地址的处理不同，所以得到的结果一直都不对。感谢陈灏大神的帮助，提示我要按照 pmap.c 中的 mem_init 中的信息对物理地址进行处理，我才正确完成了这个函数。