

操作系统上机实验

Lab 2: 内存管理

实验报告

November 11, 2019

曹元议 1711425
黄艺璇 1711429
王雨奇 1711299
张瑞宁 1711302
阔坤柔 1611381
冯晓妍 1513408

Abstract

完成了 Lab 1 的练习之后，终于可以进入 Lab 2 的练习了。我们将在 Lab 1 环境的基础上进一步探究操作系统的内存管理。本次实验主要分为两个部分：一是通过探究 JOS 内核的物理内存分配器、维护一个记录空闲物理页的链表结构来了解物理页管理，二是通过初步建立内存管理单元（MMU）实现从虚拟地址到物理地址的映射来了解虚拟内存和页表管理。

Keywords: 操作系统；QEMU；JOS；内存管理；物理页；虚拟内存；页表

Contents

1	小组分工情况	1
2	物理页管理	1
2.1	作业 3	2
3	虚拟内存	14
3.1	练习 4	14
3.2	练习 5	19
3.3	问题 3	23
3.4	作业 4	24
4	内核地址空间	36
4.1	权限和故障隔离	36
4.2	作业 5	36
4.3	问题 4	40
5	总结	44

1 小组分工情况

	问题解决	报告整理	报告检查	小组讨论	Lab 2 任务
曹元议	√	√		√	练习 4、练习 5
黄艺璇	√		√	√	作业 4
王雨奇	√		√	√	作业 5
张瑞宁	√		√	√	作业 3、问题 3
阔坤柔			√	√	
冯晓妍	√		√	√	问题 4

Table 1: 小组分工

还是一样，由于分工的问题，上次做 Lab 1 的这次就不用做了。但是 Lab 3 一定人人有份的。

2 物理页管理

在本次实验中，我们仍使用 Lab 1 配置好的实验环境。根据实验指导取得 Lab 2 的代码，并把分支切换到 Lab 2 的分支。

我们取得了这些新的实验文件：

- `inc/memlayout.h`（描述虚拟地址空间的布局，原来 Lab 1 也有这个文件，但是这里的跟 Lab 1 的不一样）
- `kern/pmap.c`（读取物理内存大小，对虚拟地址空间进行布局）
- `kern/pmap.h`
- `kern/kclock.h`（操纵 PC 的时钟以及 CMOS RAM 等设备，这些设备记录了物理内存的大小）
- `kern/kclock.c`

其中，前三个是最主要的。

以下是一些常用的宏或函数定义，本报告可能会用到：

定义	所在文件	意义
PGSIZE	inc/mmu.h	页的大小
PTSIZE	inc/mmu.h	页表大小
PGNUM	inc/mmu.h	页号
PDX	inc/mmu.h	页所在页目录号
PTX	inc/mmu.h	页所在页表号
KADDR	kern/pmap.h	物理地址转换成虚拟地址
PADDR	kern/pmap.h	虚拟地址转换成物理地址
PTE_ADDR	inc/mmu.h	取得页所在页目录或页表的地址
KERNBASE	inc/memlayout.h	内核所在虚拟地址的起始地址
page2kva	kern/pmap.h	取得PageInfo 对应的虚拟地址
pa2page	kern/pmap.h	取得物理地址对应的PageInfo
page2pa	kern/pmap.h	取得PageInfo 对应的物理地址

Table 2: 常用的宏或函数定义

2.1 作业 3

2.1.1 题目原文

在文件 `kern/pmap.c` 中，你需要实现以下函数的代码（如下，按序给出）：

```
boot_alloc()
mem_init() (在调用 check_page_free_list(1) 之前的部分)
page_init()
page_alloc()
page_free()

check_page_free_list() 和 check_page_alloc() 将测试你的物理页分配器。你需要引导 JOS 然后查看 check_page_alloc() 的成功报告。加入你自己的 assert() 来验证你的假设是否正确将会有所帮助。
```

2.1.2 题目大意

上述需要实现的函数组成了 JOS 的物理页分配器。其中函数 `mem_init()` 只需要实现 `check_page_free_list(1)` 之前的部分。函数 `check_page_free_list()` 和 `check_page_alloc()` 会检查这些需要实现的函数的正确性。

2.1.3 回答

以上需要修改的函数都在kern/pmap.c中。

首先，先看boot_alloc()函数。该函数如下图：

```
// This simple physical memory allocator is used only while JOS is setting
// up its virtual memory system.  page_alloc() is the real allocator.
//
// If n>0, allocates enough pages of contiguous physical memory to hold 'n'
// bytes.  Doesn't initialize the memory.  Returns a kernel virtual address.
//
// If n==0, returns the address of the next free page without allocating
// anything.
//
// If we're out of memory, boot_alloc should panic.
// This function may ONLY be used during initialization,
// before the page_free_list list has been set up.
static void *
boot_alloc(uint32_t n)
{
    static char *nextfree; // virtual address of next byte of free memory
    char *result;

    // Initialize nextfree if this is the first time.
    // 'end' is a magic symbol automatically generated by the linker,
    // which points to the end of the kernel's bss segment:
    // the first virtual address that the linker did *not* assign
    // to any kernel code or global variables.
    if (!nextfree) {
        extern char end[];
        nextfree = ROUNDUP((char *) end, PGSIZE);
    }

    // Allocate a chunk large enough to hold 'n' bytes, then update
    // nextfree.  Make sure nextfree is kept aligned
    // to a multiple of PGSIZE.
    //
    // LAB 2: Your code here.

    return NULL;
}
```

Figure 1: 函数 boot_alloc()

注释反映了本函数的功能是一个简单的物理内存分配器，它仅用于 JOS 进入虚拟内存系统、加载空闲物理页链表前的初始化（也就是只用于函数mem_init()中，之后的内存分配使用page_alloc()）。在注释中也提示了这个函数如何实现的大致思路：如果传入的参数n>0，就需要分配足够的空间（n个字节）并返回这个空间的首地址，但不要初始化这些空间；如果n==0则不分配空间，返回指向的下一个可用内存空间的首地址。其次，还需要判断分配了以后内存是否够（不能超过页的总大小：npages*PGSIZE，npages在kern/pmap.h中），否则应提示内存不足（使用panic），分配失败，即这个函数的执行应该要满足内存充足的断言。

panic在inc/assert.h中，是为了实现一些断言所需要的函数，不满足断言会报错。

```

/* See COPYRIGHT for copyright information. */

#ifndef JOS_INC_ASSERT_H
#define JOS_INC_ASSERT_H

#include <inc/stdio.h>

void _warn(const char*, int, const char*, ...);
void _panic(const char*, int, const char*, ...) __attribute__((noreturn));

#define warn(...) _warn(__FILE__, __LINE__, __VA_ARGS__)
#define panic(...) _panic(__FILE__, __LINE__, __VA_ARGS__)

#define assert(x) \
    do { if (!(x)) panic("assertion failed: %s", #x); } while (0)

// static_assert(x) will generate a compile-time error if 'x' is false.
#define static_assert(x) switch (x) case 0: case (x):

#endif /* !JOS_INC_ASSERT_H */

```

Figure 2: inc/assert.h

其中，`nextfree`变量是一个静态变量，由于未显式初始化，因此初次进入该函数会被初始化成 0，并且只初始化一次，这个特性决定了它总指向下一个空闲内存空间的首地址。在初次调用本函数时会进入第一个 `if` 分支并执行其中的语句。需要注意字符数组 `end` 和宏 `ROUNDUP` 的含义。

由注释可知，`end` 由链接器生成，它指向内核的 `bss` 段。在终端中输入命令 `objdump -h kernel` 查看内核的段信息。

```

user@pc: ~/oslab/lab1/src/lab1_2
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
user@pc:~/oslab/lab1/src/lab1_2$ objdump -h obj/kern/kernel

obj/kern/kernel:      文件格式 elf32-i386

节:
Idx Name              Size      VMA      LMA      File off  Algn
 0 .text              00001b5f  f0100000 00100000 00001000 2**4
CONTENTS, ALLOC, LOAD, READONLY, CODE
 1 .rodata             00000810  f0101b60 00101b60 00002b60 2**5
CONTENTS, ALLOC, LOAD, READONLY, DATA
 2 .stab               00004315  f0102370 00102370 00003370 2**2
CONTENTS, ALLOC, LOAD, READONLY, DATA
 3 .stabstr            00001beb  f0106685 00106685 00007685 2**0
CONTENTS, ALLOC, LOAD, READONLY, DATA
 4 .data               0000a300  f0109000 00109000 0000a000 2**12
CONTENTS, ALLOC, LOAD, DATA
 5 .bss                00000650  f0113300 00113300 00014300 2**5
ALLOC
 6 .comment            0000002b  00000000 00000000 00014300 2**0
CONTENTS, READONLY
user@pc:~/oslab/lab1/src/lab1_2$

```

Figure 3: 利用 objdump 查看内核的段信息

由于 `.comment` 段是一些程序的注释信息，不存入内存。因此 `end` 指向的 `.bss` 段就是内核的最后一个段，因此在 `end` 之后的所有内存都是空闲的，因此将要从这个区域开

始分配n个字节。

ROUNDUP是一个宏，它的定义在inc/types.h中。

```
// Rounding operations (efficient when n is a power of 2)
// Round down to the nearest multiple of n
#define ROUNDDOWN(a, n) \
({ \
    uint32_t __a = (uint32_t) (a); \
    (typeof(a)) (__a - __a % (n)); \
})
// 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)); \
})
```

Figure 4: inc/types.h 中 ROUNDUP 宏的定义

由注释可知，这个宏的意义就是四舍五入到n的最近倍数（说向上取整到最近的n的倍数可能更准确一些），在if分支中对这个宏的调用，n的值为PGSIZE，为一个页的大小（4KB，在inc/mmu.h），故在这里调用的意义是使分配的空间 12 字节对齐。

由以上的分析，应补齐的代码为：

```
1 static void *
2 boot_alloc(uint32_t n)
3 {
4     static char *nextfree; // virtual address of next byte of free memory
5     char *result;
6
7     // Initialize nextfree if this is the first time.
8     // 'end' is a magic symbol automatically generated by the linker,
9     // which points to the end of the kernel's bss segment:
10    // the first virtual address that the linker did *not* assign
11    // to any kernel code or global variables.
12    if (!nextfree) {
13        extern char end[];
14        nextfree = ROUNDUP((char *) end, PGSIZE);
15    }
16
17    // Allocate a chunk large enough to hold 'n' bytes, then update
18    // nextfree. Make sure nextfree is kept aligned
19    // to a multiple of PGSIZE.
20    //
21    // LAB 2: Your code here.
22    if(n==0)//如果n为0，则直接返回指向下一个空闲内存区域的首地址
23        return nextfree;
24    char *result;
```



```

25 result = nextfree;
26 nextfree += n; // 否则分配n字节空间, 这个空间在result和nextfree之间, result
    是这个空间的首地址
27 nextfree = ROUNDUP((char*)nextfree, PGSIZE); // 使分配的空间12字节对齐
28
29 // 还需要判断一下分配了以后是否内存不足
30 // 由于nextfree是虚拟地址, 因此要先转换成相应的物理地址
31 if((uint32_t)PADDR(nextfree) > npages*(PGSIZE)) {
32     nextfree = result;
33     panic("Out of memory!\n");
34     return NULL;
35 }
36 return result;
37 }

```

成功执行了一次代码之后, `n`、`result`和`nextfree`的关系如下图(`nextfree`右边的是空闲但未分配的空间,`result`到`nextfree`是本次函数调用时分配的空间,`end`到`result`之间是已经分配了的空间):

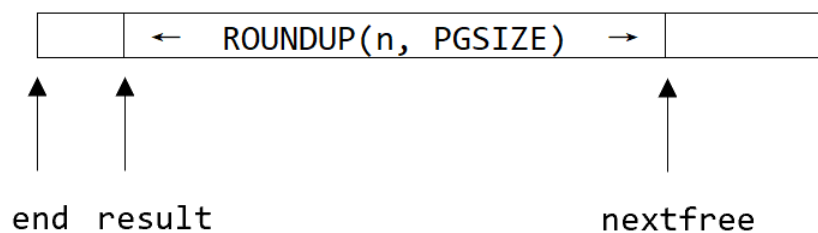


Figure 5: 示意图

下一个函数是`mem_init()`函数。需要用到一个`PageInfo`结构体, 它存储一个物理页的描述。它的定义在`inc/memlayout.h`中:

```

/*
 * Page descriptor structures, mapped at UPAGES.
 * Read/write to the kernel, read-only to user programs.
 *
 * Each struct PageInfo stores metadata for one physical page.
 * Is it NOT the physical page itself, but there is a one-to-one
 * correspondence between physical pages and struct PageInfo's.
 * You can map a struct PageInfo * to the corresponding physical address
 * with page2pa() in kern/pmap.h.
 */
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.
    // Pages allocated at boot time using pmap.c's
    // boot_alloc do not have valid reference count fields.

    uint16_t pp_ref;
};

```

Figure 6: PageInfo

这是一个链表结构。其中，`pp_ref` 表示有多少个指针指向该页（该页被引用了多少次，如果为 0 则为空闲页），`pp_link` 表示空闲内存列表中的下一页。非空闲页的 `pp_link` 总是为 `NULL`。值得注意的是 `pp_link` 是该结构体的第一个成员变量，是指针类型，因此占 4 Byte；第二个成员变量是 `pp_ref`，类型为 `uint16_t`，占 2 Byte；按照字节对齐的规则，两个成员变量共占 6 Byte（第二个成员变量相对于结构体的首地址的偏移已经是 2 的倍数）；由于该结构体占空间最大成员变量是指针 `pp_link`，因此结构体所占的空间应以指针的 4 Byte 对齐（最终占的空间是 4 的倍数），因此 `sizeof(struct PageInfo)=8 Byte`，这个值会在问题 4 中用到。

下图是 `mem_init()` 的注释：

```

// Set up a two-level page table:
//   kern_pgdir is its linear (virtual) address of the root
//
// This function only sets up the kernel part of the address space
// (ie. addresses >= UTOP). The user part of the address space
// will be setup later.
//
// From UTOP to ULIM, the user is allowed to read but not write.
// Above ULIM the user cannot read or write.
void
mem_init(void)
{
    uint32_t cr0;
    size_t n;

    // Find out how much memory the machine has (npages & npages_basemem).
    i386_detect_memory();

    // Remove this line when you're ready to test this function.
    panic("mem_init: This function is not finished\n");

    ////////////////////////////////////////////////////
    // create initial page directory.
    kern_pgdir = (pde_t *) boot_alloc(PGSIZE);
    memset(kern_pgdir, 0, PGSIZE);

    ////////////////////////////////////////////////////
    // Recursively insert PD in itself as a page table, to form
    // a virtual page table at virtual address UVPT.
    // (For now, you don't have understand the greater purpose of the
    // following line.)

    // Permissions: kernel R, user R
    kern_pgdir[PDX(UVPT)] = PADDR(kern_pgdir) | PTE_U | PTE_P;

    ////////////////////////////////////////////////////
    // 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:

```

Figure 7: 函数 mem_init() 的注释

由注释可知，该函数是对内存进行初始化的函数，它会建立起二级页表并建立起内核地址空间。在 `check_page_free_list(1)` 的调用之前应实现部分的注释的提示为：为 `PageInfo` 数组 `pages` 申请足够的空间（`npages` 个页面），并将这些空间使用 `memset()` 函数初始化为 0。内核将使用 `pages` 数组追踪物理页的使用情况。

因此，应补齐的代码为：

```

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

```

下一个是 `page_init()` 函数。下面的注释反映了这个函数的主要目的是遍历并初始化物理页的描述信息，如果该页是空闲页，需要将这个 `PageInfo` 链接到表示空闲

页的链表`page_free_list`。它同时还提醒如果这个函数执行成功了，之后不要再调用`boot_alloc()`。

```
// -----
// Tracking of physical pages.
// The 'pages' array has one 'struct PageInfo' entry per physical page.
// Pages are reference counted, and free pages are kept on a linked list.
// -----

//
// Initialize page structure and memory free list.
// After this is done, NEVER use boot_alloc again. ONLY use the page
// allocator functions below to allocate and deallocate physical
// memory via the page_free_list.
//
void
page_init(void)
{
    // The example code here marks all physical pages as free.
    // However this is not truly the case. What memory is free?
    // 1) Mark physical page 0 as in use.
    //    This way we preserve the real-mode IDT and BIOS structures
    //    in case we ever need them. (Currently we don't, but...)
    // 2) The rest of base memory, [PGSIZE, npages_basemem * PGSIZE)
    //    is free.
    // 3) Then comes the IO hole [IOPHYSMEM, EXTPHYSMEM), which must
    //    never be allocated.
    // 4) Then extended memory [EXTPHYSMEM, ...).
    //    Some of it is in use, some is free. Where is the kernel
    //    in physical memory? Which pages are already in use for
    //    page tables and other data structures?
    //
    // Change the code to reflect this.
    // NB: DO NOT actually touch the physical memory corresponding to
    // free pages!
    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];
    }*/
}
```

Figure 8: 函数 `page_init()` 的注释

函数体内提示了哪些范围的物理页是可分配的，哪些范围的不可分配的。如果是可分配的就把该`PageInfo`的`pp_ref`置为 0 表示没有被引用并将该`PageInfo`加入到链表`page_free_list`表明初始状态所有可分配的页都是空闲的。否则把该`PageInfo`的`pp_ref`置为 1，`pp_link`置`NULL`表明该物理页不空闲。

一共分为四种情况：

1. 第 0 号物理页不能被分配，它是保存实模式 IDT 和 BIOS 结构用的。
2. `[PGSIZE, npages_basemem*PGSIZE)` 可以被分配。
3. `[IOPHYSMEM, EXTPHYSMEM)` 是一个 I/O 空洞，留给 I/O 设备，详见 Lab 1 的物理地址空间布局。不能被分配。

4. [EXTPHYSMEM,...)为扩展内存空间,其中有一部分可以被分配,有一部分不能。需要找出第一个空闲页的首地址,因此在[EXTPHYSMEM,first_free_address)不能被分配,而[first_free_address,...)都可以被分配,但是页号不能超过npages。

因此,循环体的部分应改为:

```

1 void
2 page_init(void)
3 {
4     // Change the code to reflect this.
5     // NB: DO NOT actually touch the physical memory corresponding to
6     // free pages!
7     size_t i;
8
9     // 1) Mark physical page 0 as in use.
10    //     This way we preserve the real-mode IDT and BIOS structures
11    //     in case we ever need them. (Currently we don't, but...)
12    pages[0].pp_ref = 1;
13    // 2) The rest of base memory, [PGSIZE, npages_basemem * PGSIZE)
14    //     is free.
15    for (i = 1; i < npages_basemem; i++) {
16        pages[i].pp_ref = 0;
17        pages[i].pp_link = page_free_list;
18        page_free_list = &pages[i];
19    }
20    // 3) Then comes the IO hole [IOPHYSMEM, EXTPHYSMEM), which must
21    //     never be allocated.
22    for (i = IOPHYSMEM/PGSIZE; i < EXTPHYSMEM/PGSIZE; i++) {
23        pages[i].pp_ref = 1;
24    }
25    // 4) Then extended memory [EXTPHYSMEM, ...).
26    //     Some of it is in use, some is free. Where is the kernel
27    //     in physical memory? Which pages are already in use for
28    //     page tables and other data structures?
29    size_t first_free_address = PADDR(boot_alloc(0)); //利用boot_alloc()找到
30    //     第一个空闲页面的首地址
31    for (i = EXTPHYSMEM/PGSIZE; i < first_free_address/PGSIZE; i++) {
32        pages[i].pp_ref = 1;
33    }
34    for (i = first_free_address/PGSIZE; i < npages; i++) {
35        pages[i].pp_ref = 0;
36        pages[i].pp_link = page_free_list;
37        page_free_list = &pages[i];

```

```

37 }
38 }

```

在上述代码的循环体中，如果是一个空闲物理页，则按照原先给的循环体的操作插入进链表：

```

pages[i].pp_ref = 0;
pages[i].pp_link = page_free_list;
page_free_list = &pages[i];

```

这是链表的头插法，即每个新的结点都插到链表的开头，然后使头指针指向这个新结点。链表`page_free_list`结构如下图所示：

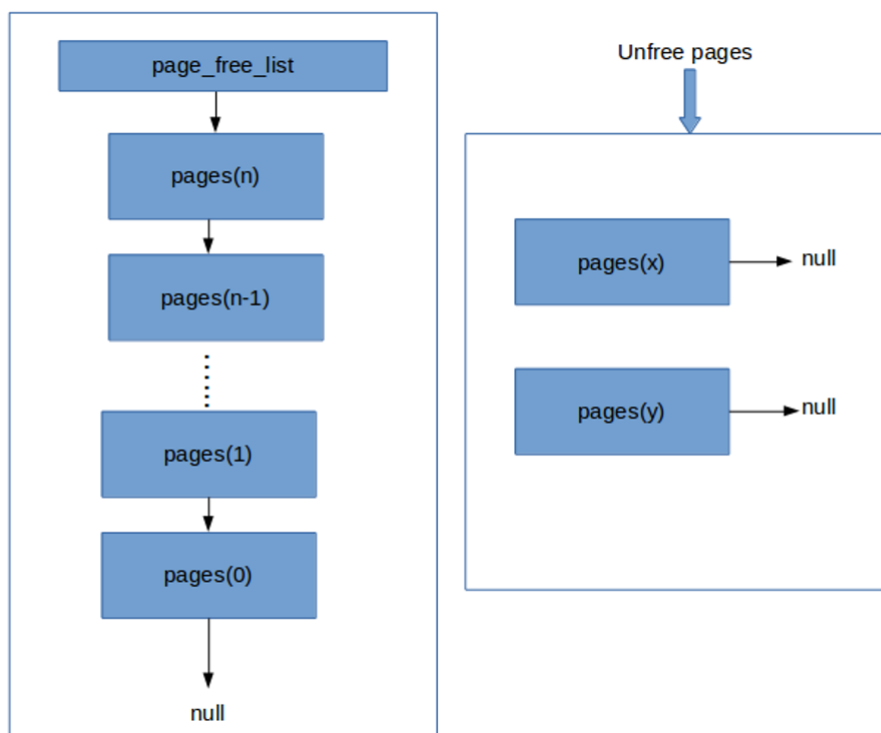


Figure 9: `page_free_list` 结构

`/PGSIZE`是为了得到相应的物理页号，用这个页号去索引相应的`PageInfo`，也可以使用宏`PGNUM`来代替来获取页号。

下面是函数`page_alloc()`。注释如下：

```

//
// Allocates a physical page. If (alloc_flags & ALLOC_ZERO), fills the entire
// returned physical page with '\0' bytes. Does NOT increment the reference
// count of the page - the caller must do these if necessary (either explicitly
// or via page_insert).
//
// Be sure to set the pp_link field of the allocated page to NULL so
// page_free can check for double-free bugs.
//
// Returns NULL if out of free memory.
//
// Hint: use page2kva and memset
struct PageInfo *
page_alloc(int alloc_flags)
{
    // Fill this function in
    return 0;
}

```

Figure 10: 函数 page_alloc() 的注释

由上图可知，该函数的作用是从page_free_list中取出第一个空闲的页的信息，对这个页进行分配，并从这个链表中删除，让下一个结点成为链表头。由于这个分配操作只是对PageInfo的操作，不是真正的分配页，因此可理解为将该页标记为已使用。上面的提示有，如果alloc_flags & ALLOC_ZERO为真，则将这个物理页填充为0（使用page2kva()和memset()）。需要将被分配的页的PageInfo分配的pp_link修改为NULL（表明此页不再空闲，将其信息从链表中拿出来），不用修改pp_ref，因为调用者会修改它。如果已经没有空闲页可分配（链表指向NULL），则函数返回NULL。否则返回被分配的物理页的PageInfo的地址。

由上述的分析，函数应修改为：

```

1 struct PageInfo *
2 page_alloc(int alloc_flags)
3 {
4     // Fill this function in
5     if(page_free_list == NULL)//如果已经没有空闲物理页可分配，则返回NULL
6         return NULL;
7
8     struct PageInfo* page = page_free_list;
9     page_free_list = page->pp_link;
10    page->pp_link = 0;
11    if(alloc_flags & ALLOC_ZERO)//如果满足这个条件就把这个物理页清0
12        memset(page2kva(page), 0, PGSIZE);
13    return page;
14 }

```

alloc_flags可理解为分配页的行为标志。如果其和ALLOC_ZERO相与的值的最低位不为0（为1）则要进行这个页的初始化的动作。

在kern/pmap.h找到了ALLOC_ZERO的定义:

```
enum {
    // For page_alloc, zero the returned physical page.
    ALLOC_ZERO = 1<<0,
};
```

Figure 11: ALLOC_ZERO

是一个枚举常量。不知道把1写成1<<0有什么特殊含义，推测是为了突出相与的是最低位，增加代码的可读性。

最后一个函数是page_free()，作用是释放一个物理页。由下面的注释，就是把一个变为空闲的物理页信息加入到链表中（插入原理同page_init()）。在此之前需要判断一下这个物理页是否还在被引用（pg_ref>0），或者是它已经在这个链表当中了（pp_link!=NULL），如果是就panic并返回：

```
//
// Return a page to the free list.
// (This function should only be called when pp->pp_ref reaches 0.)
//
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.
}
```

Figure 12: 函数 page_free() 的注释

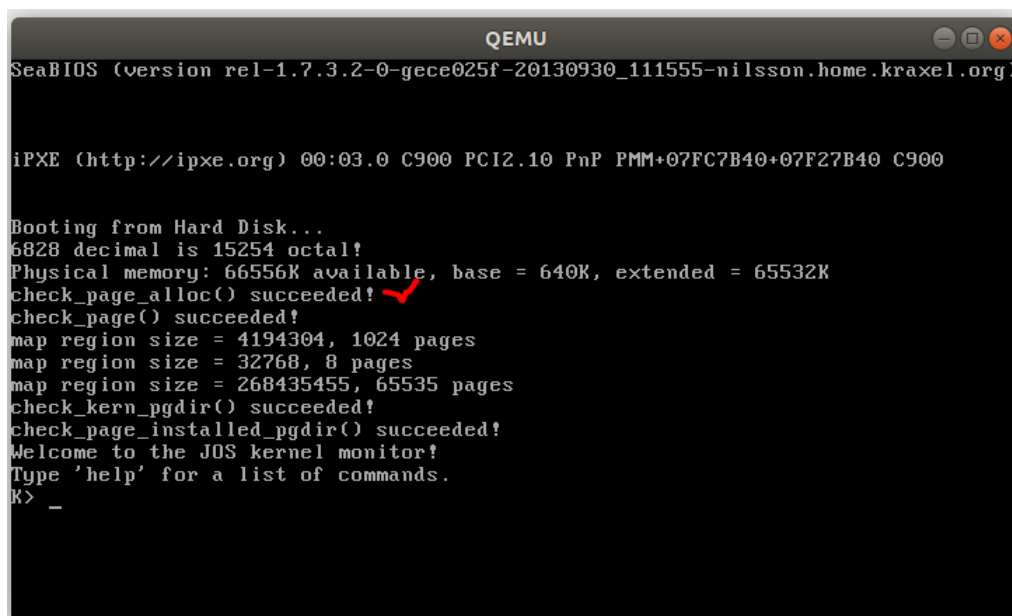
因此，函数应修改为：

```
1 //
2 // Return a page to the free list.
3 // (This function should only be called when pp->pp_ref reaches 0.)
4 //
5 void
6 page_free(struct PageInfo *pp)
7 {
8     // Fill this function in
9     // Hint: You may want to panic if pp->pp_ref is nonzero or
10    // pp->pp_link is not NULL.
11    if (pp->pp_ref > 0 || pp->pp_link != NULL) {
12        panic("Double check failed when dealloc page");
13        return;
14    }
15    pp->pp_link = page_free_list;
```



```
16 page_free_list = pp;  
17 }
```

以下是check_page_alloc()的成功报告:



```
QEMU  
SeaBIOS (version rel-1.7.3.2-0-gece025f-20130930_111555-nilsson.home.kraxel.org)  
  
iPXE (http://ipxe.org) 00:03.0 C900 PCI2.10 PnP PMM+07FC7B40+07F27B40 C900  
  
Booting from Hard Disk...  
6828 decimal is 15254 octal!  
Physical memory: 66556K available, base = 640K, extended = 65532K  
check_page_alloc() succeeded! ✓  
check_page() succeeded!  
map region size = 4194304, 1024 pages  
map region size = 32768, 8 pages  
map region size = 268435455, 65535 pages  
check_kern_pgdir() succeeded!  
check_page_installed_pgdir() succeeded!  
Welcome to the JOS kernel monitor!  
Type 'help' for a list of commands.  
K> _
```

Figure 13: 成功报告

3 虚拟内存

3.1 练习 4

3.1.1 题目原文

阅读 Intel 80386 Reference Manual 的第 5、6 章, 阅读关于页转换和基于页的保护 <http://video.mobisys.cc/pages/pdos.csail.mit.edu/6.828/2014/readings/i386/toc.htm> (5.2 和 6.4)。我们推荐你们简单了解一下关于分段模式的章节, 因为虽然JOS使用的基于页机制的虚拟内存和保护, 但是段转换和基于段的保护无法在 x86 上被禁用, 所以你需要对其有一个基础的了解。

3.1.2 题目大意

分段和分页是 x86 保护模式的两种内存管理架构。了解它们的机制。

3.1.3 回答

首先要了解逻辑地址、虚拟地址、线性地址和物理地址是什么以及它们之间的关系。

在 x86 体系结构中，保护模式的逻辑地址格式为 [段选择子 (给出段基址): 段内偏移] (段基址总是被省略)。格式与实模式的逻辑地址格式 [段寄存器 (给出段基址): 段内偏移] 相类似。段基址和段内偏移共同给出线性地址。在保护模式中，如未开启分页机制，则线性地址就是物理地址。开启了分页机制则是虚拟地址。

虚拟地址是在保护模式中才有的概念，顾名思义，它是在虚拟内存中的地址。引入虚拟内存是为了方便操作系统对内存的管理。虚拟内存是操作系统对主存和磁盘 I/O 设备的抽象表示，它为进程提供独占主存空间的假象。虚拟地址引用的数据可能在主存，也可能在磁盘。

物理地址就是实际物理主存的地址。它用于内存芯片级的单元寻址，与地址总线相对应。物理地址是经过段和页转换后我们最终得到的地址，也是最终通过硬件总线输出到 RAM 的地址。

内存中的地址转换由内存控制单元 (MMU) 完成，它通过一种称为分段单元 (segmentation unit) 的硬件电路把一个逻辑地址转换成线性地址；接着，第二个称为分页单元 (paging unit) 的硬件电路把线性地址转换成一个物理地址。对于 Intel CPU 来说，分页标志位是 CR0 寄存器的第 31 位，为 1 表示使用分页，为 0 表示不使用分页。CPU 在执行代码时，自动检测 CR0 寄存器中的分页标志位是否被设定，若被设定就自动完成虚拟地址到物理地址的转换。保护模式下逻辑地址、虚拟地址、线性地址和物理地址的关系可表示如下图：

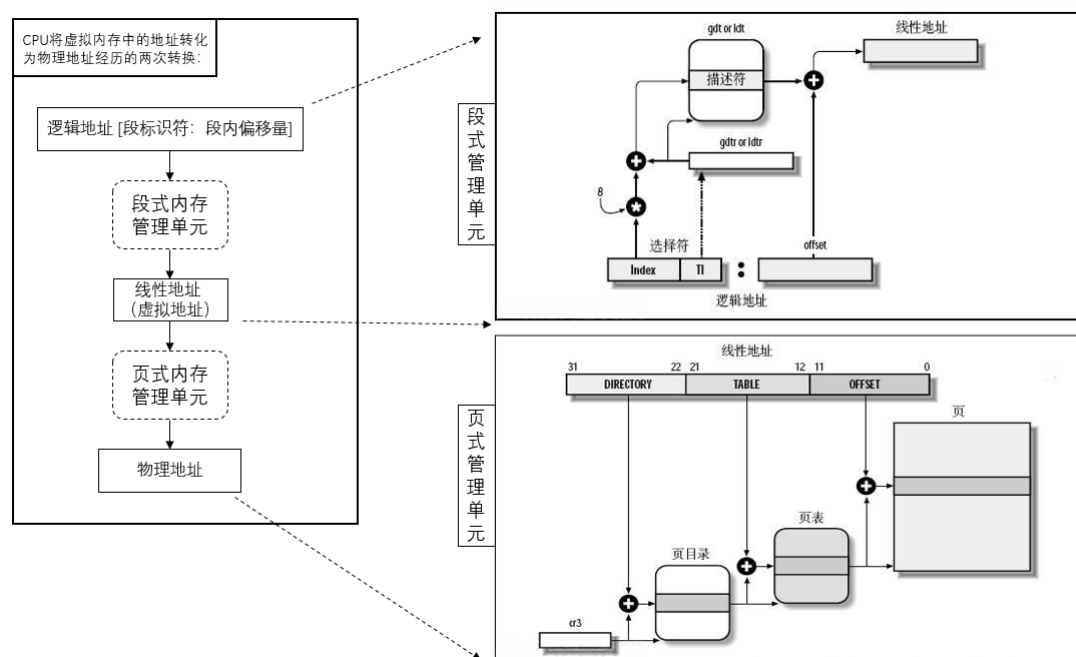


Figure 14: 保护模式下逻辑地址、虚拟地址、线性地址和物理地址的关系

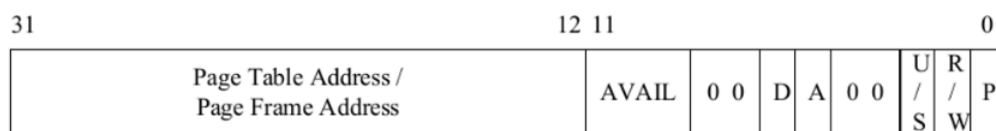
从上图中可以看出来，我们编程时使用的 C 语言指针用的是虚拟地址来描述的（更准确地说逻辑地址中的偏移，这是在编程时实际可以给出来的）。在 JOS 中的 `boot/boot.S` 中加载了一个全局描述符表（GDT），通过将所有段基址设置为 0 并将其有效地址上限设置为 `0xffffffff`，这覆盖了整个 32 位的地址空间。因此，在这里“段选择器”是不起作用的（段基址为 0），线性地址总是等于虚拟地址，因此在本实验中可以忽略分段机制，并且只关注页转换。JOS 会将所有物理内存从物理地址 `0x00000000` 映射到虚拟地址 `0xf0000000`。因为由 Lab 1 可以知道，JOS 内核只能使用 256MB 物理内存。整个底部的 256MB 物理地址空间即 `0x00000000~0xffffffff` 会被映射到 `0xf0000000~0xffffffff`。

上述的图已经大概描述了分段机制和分页机制的原理。

分段机制大概表述为：为了向前兼容，80386 的段寄存器和 8086 的段寄存器都是 16 位的。只是在 80386 保护模式中中段寄存器的含义发生了变化，线性地址的计算再也不是原来的段寄存器值 $\times 16 + \text{offset}$ 了。80386 的段选择子的高 13 位提供了一个段描述符（共 64 位）在其段描述符表中的索引（index），这个索引值 $\times 8$ （左移 3 位变为 16 位）加上段描述符表的基址（保存在一个寄存器中）即可得到该段描述符。其中，段描述表分为全局描述符表（GDT）和局部描述符表（LDT），其基址分别用 GDTR 和 LDTR 寄存器来存储。把这个段描述符的 base 字段（基址）加上 offset 即得线性地址。

分页机制是为了解决纯分段机制导致内存利用率不高的问题的。分页机制分为一级分页机制和二级分页机制。操作系统管理内存的粒度为页，页的大小一般为 4KB (2^{12})。

Byte)。因此在一个线性地址中，低 12 位为页内的偏移。在一级分页机制中，除了低 12 位为页内偏移以外，其余高位都是在页表中的索引。在二级分页机制中，低 12 位为页内偏移，中间 10 位为在页表内的索引，高 10 位是在页目录中的索引。页目录和页表都是把线性地址映射到物理地址的数据结构（一般是数组）。在一级分页机制中，由于每个进程都有高达 2^{20} 个页表项，占用大量内存空间。但是一个进程往往用不到所有的地址，因此这样的内存利用率会降低。使用了二级分页机制以后，把高 20 位继续拆分为高 10 位为页目录索引第 10 位为页表索引，一个页面对应页表中的一个页表项，一个页表对应页目录中的一个页目录项。一个页表和一个页目录都有 $2^{10}=1024$ 项，由于一个页表项和一个页目录项都为 4 Byte，原因是一个页表项或页目录项只需要存 20 位的页号或页表号，由于按字节寻址，因此一个页表项至少占 $\text{ceil}(20/8)=3\text{B}$ ，取 2 的整数次幂即为 4B，除了高 20 位外其他多余的 12 位都作为标志位。如下图所示：



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.

Figure 15: 页表项或页目录项的结构

上图的结构中，低 3 位的标志位 U、W、P 是最主要的：

- **P:** 代表页面是否有效，若为 1，表示页面有效。否则，表示页面无效，不能映射页面，否则会发生错误。
- **W:** 表示页面是否可写。若为 1，则页面可以进行写操作，否则，页面是只读页面，不能进行修改。
- **U:** 表示用户程序是否可以使用该页面。若为 1，表示此页面是用户页面，用户程序可以使用并且访问该页面。若为 0，则表示用户程序不能访问该页面，只有内核才能访问页面。

这些标志位都可以有效的保护系统的安全，实现操作系统和用户程序的隔离，加强了系统的稳定性。

由于一个页表项或页目录项为 4B，则一个页表和一个页目录的大小都和一个页一样大。页转换的过程大致如下：CR3 寄存器包含页目录的起始地址，用 32 位线性地址的最高 10 位作为页目录的页目录项的索引，将它乘以 4（左移 2 位），与 CR3 中的页目录的起始地址相加，形成相应页表的地址。然后从指定的地址中取出 32 位页目录项，它的低 12 位为 0，这 32 位是页表的起始地址。用 32 位线性地址中的中间 10 位作为页表中的页面的索引，将它乘以 4（左移 2 位），与页表的起始地址相加，形成 32 位页面地址。最后将低 12 位作为相对于页面地址的偏移量，与 32 位页面地址相加，形成 32 位物理地址。

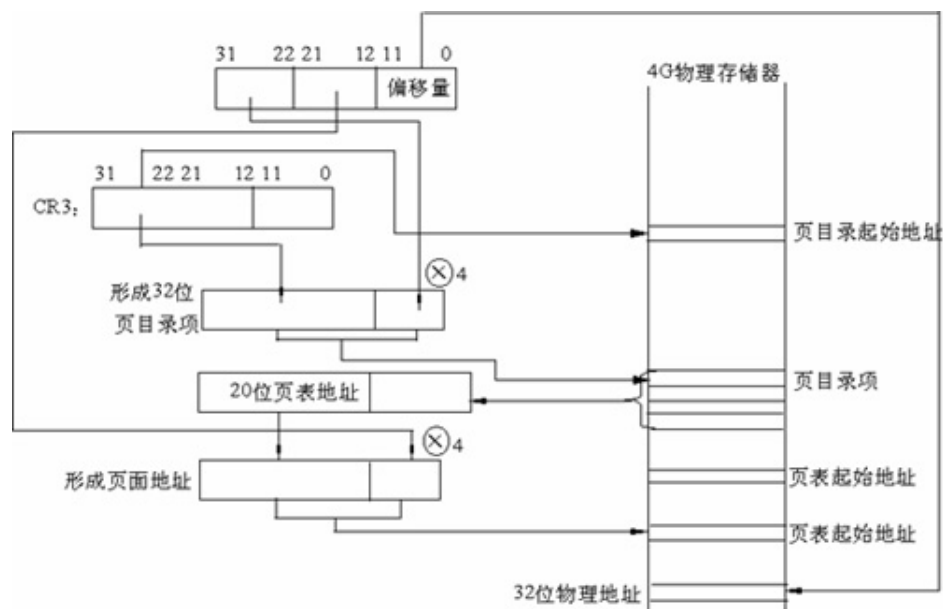


Figure 16: 线性地址和物理地址的映射

分页管理中，页目录以及页表都存放在内存中，而由于 CPU 和内存速度的不匹配，并且可能需要多次访存才能拿到所需的数据，这样地址转换时势必会降低系统的效率。为了提高地址转换的速度，x86 处理器引入了 TLB 来缓存最近转换过的地址。

使用虚拟内存的好处是显而易见的：

- a. 为每个程序都以为自己独占计算机内存空间的假象，概念清晰，方便程序的编译和装载。
- b. 有些程序可以忽略物理内存的限制，因为有些内存存储在磁盘上，这意味着程序可以使用比物理内存大的虚拟内存。
- c. 通过为不同的进程设置不同的页表，操作系统可以防止进程访问其他进程的地址。

d. 通过为不同的进程设置相同的物理页映射，操作系统可以允许进程之间共享部分资源（例如内核代码等）。

虽然分页由于是对内存等大的划分，提高了内存的利用率，但是它打散了程序的逻辑结构。而分段对内存的利用率虽然没有分页高，但是它能反映程序的逻辑结构，利于段的保护。因此，在很多操作系统中分段和分页往往是结合映射的：先分段，段内再分页。

3.2 练习 5

3.2.1 题目原文

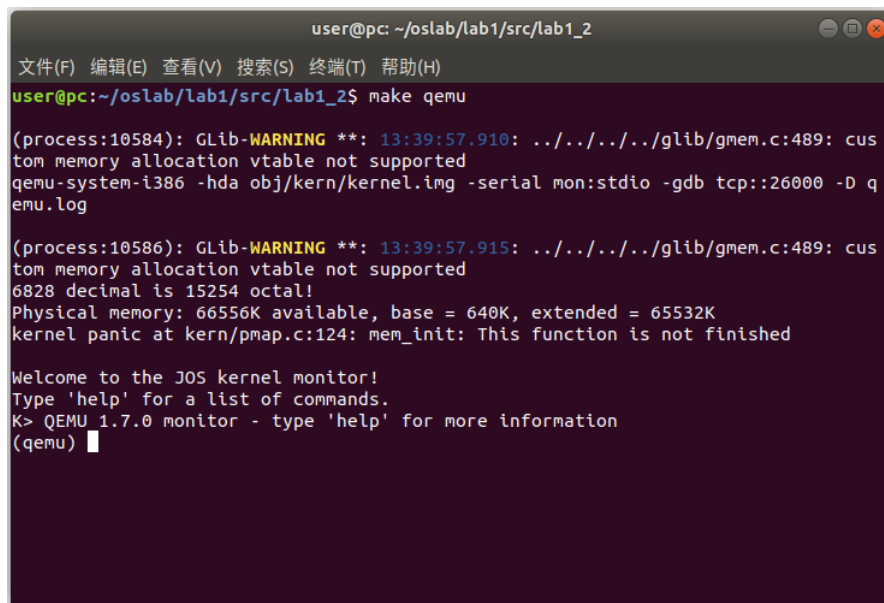
GDB 仅能通过虚拟地址访问 QEMU 的内存，但是在学习建立虚拟地址的时候，我们还需要同时检查物理地址。学习 QEMU 的 `monitor command`，特别是 `xp` 命令，它能够使你检查物理内存。在 `Terminal` 中按 `Ctrl-a c`（注：这个表示先同时按下 `ctrl` 和 `a`，然后再按下 `c`）打开 QEMU monitor。

3.2.2 题目大意

学习使用 QEMU 的 `monitor command`。

3.2.3 回答

先打开终端，输入 `make qemu` 来启动 QEMU，然后在 Ubuntu 的终端（不是 QEMU 的终端）中先同时按下 `ctrl` 和 `a`，然后按下 `c`，进入 QEMU monitor。



```

user@pc: ~/oslab/lab1/src/lab1_2
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
user@pc:~/oslab/lab1/src/lab1_2$ make qemu

(process:10584): Glib-WARNING **: 13:39:57.910: ../../../../glib/gmem.c:489: custom memory allocation vtable not supported
qemu-system-i386 -hda obj/kern/kernel.img -serial mon:stdio -gdb tcp::26000 -D qemu.log

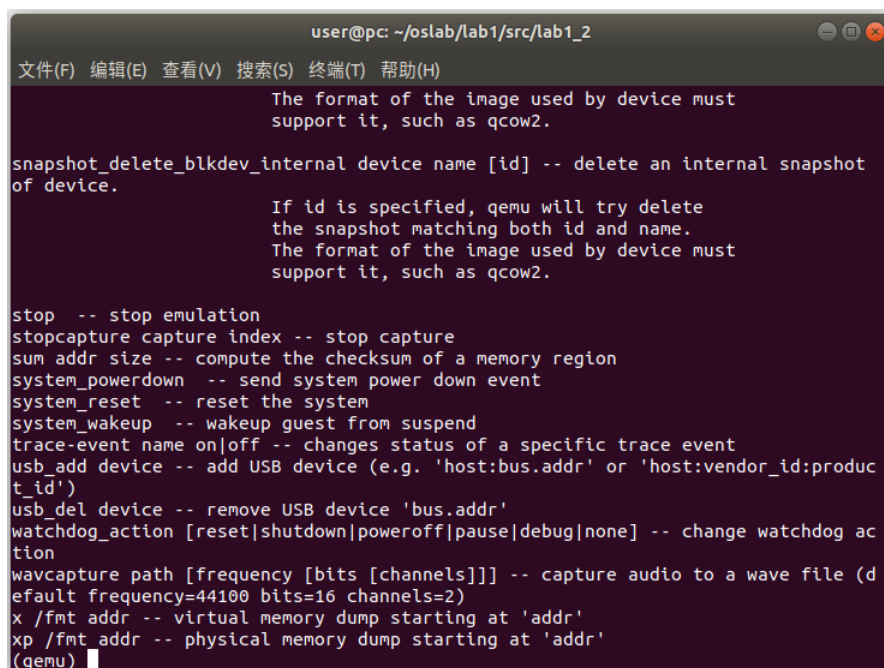
(process:10586): Glib-WARNING **: 13:39:57.915: ../../../../glib/gmem.c:489: custom memory allocation vtable not supported
6828 decimal is 15254 octal!
Physical memory: 66556K available, base = 640K, extended = 65532K
kernel panic at kern/pmap.c:124: mem_init: This function is not finished

Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> QEMU 1.7.0 monitor - type 'help' for more information
(qemu)

```

Figure 17: 进入 QEMU monitor

按照上图的提示，输入命令`help`可获取 QEMU monitor 支持的所有命令：



```

user@pc: ~/oslab/lab1/src/lab1_2
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
The format of the image used by device must support it, such as qcow2.

snapshot_delete_blkdev_internal device name [id] -- delete an internal snapshot of device.

If id is specified, qemu will try delete the snapshot matching both id and name.
The format of the image used by device must support it, such as qcow2.

stop -- stop emulation
stopcapture capture index -- stop capture
sum addr size -- compute the checksum of a memory region
system_powerdown -- send system power down event
system_reset -- reset the system
system_wakeup -- wakeup guest from suspend
trace-event name on/off -- changes status of a specific trace event
usb_add device -- add USB device (e.g. 'host:bus.addr' or 'host:vendor_id:product_id')
usb_del device -- remove USB device 'bus.addr'
watchdog_action [reset|shutdown|poweroff|pause|debug|none] -- change watchdog action
wavcapture path [frequency [bits [channels]]] -- capture audio to a wave file (default frequency=44100 bits=16 channels=2)
x /fmt addr -- virtual memory dump starting at 'addr'
xp /fmt addr -- physical memory dump starting at 'addr'
(qemu)

```

Figure 18: 输入命令 help

输入命令`info registers`可查看当前寄存器的信息：

```

user@pc: ~/oslab/lab1/src/lab1_2
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
x /fmt addr -- virtual memory dump starting at 'addr'
xp /fmt addr -- physical memory dump starting at 'addr'
(qemu) info registers
EAX=f0100383 EBX=f0100383 ECX=000003d4 EDX=ffffffff
ESI=00000000 EDI=00000001 EBP=f0110ec8 ESP=f0110ebc
EIP=f0100383 EFL=00000086 [--S--P-] CPL=0 II=0 A20=1 SMM=0 HLT=0
ES =0010 00000000 ffffffff 00cf9300 DPL=0 DS [-WA]
CS =0008 00000000 ffffffff 00cf9a00 DPL=0 CS32 [-R-]
SS =0010 00000000 ffffffff 00cf9300 DPL=0 DS [-WA]
DS =0010 00000000 ffffffff 00cf9300 DPL=0 DS [-WA]
FS =0010 00000000 ffffffff 00cf9300 DPL=0 DS [-WA]
GS =0010 00000000 ffffffff 00cf9300 DPL=0 DS [-WA]
LDT=0000 00000000 0000ffff 00008200 DPL=0 LDT
TR =0000 00000000 0000ffff 00008b00 DPL=0 TSS32-busy
GDT= 00007c4c 00000017
IDT= 00000000 000003ff
CR0=80010011 CR2=00000000 CR3=00111000 CR4=00000000
DR0=00000000 DR1=00000000 DR2=00000000 DR3=00000000
DR6=ffff0fff DR7=00000400
EFER=0000000000000000
FCW=037f FSW=0000 [ST=0] FTW=00 MXCSR=00001f80
FPR0=0000000000000000 0000 FPR1=0000000000000000 0000
FPR2=0000000000000000 0000 FPR3=0000000000000000 0000
FPR4=0000000000000000 0000 FPR5=0000000000000000 0000
FPR6=0000000000000000 0000 FPR7=0000000000000000 0000
XMM00=00000000000000000000000000000000 XMM01=00000000000000000000000000000000
XMM02=00000000000000000000000000000000 XMM03=00000000000000000000000000000000
XMM04=00000000000000000000000000000000 XMM05=00000000000000000000000000000000
XMM06=00000000000000000000000000000000 XMM07=00000000000000000000000000000000
(qemu)

```

Figure 19: 输入命令 info registers

输入命令 `info pg` 可查看当前页表的信息，可以看到虚拟页的地址、对应的页表项、标志位、对应的物理页地址：

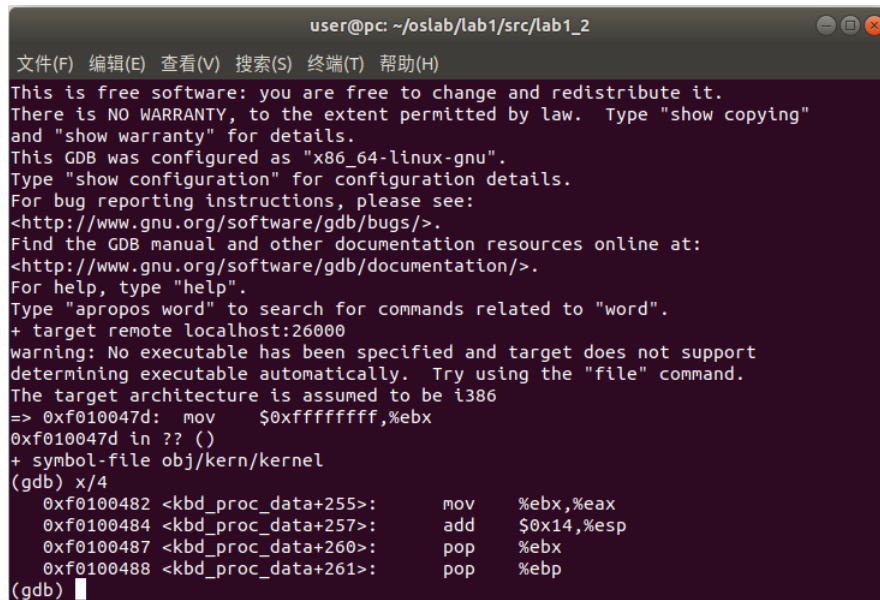
```

user@pc: ~/oslab/lab1/src/lab1_2
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
XMM06=00000000000000000000000000000000 XMM07=00000000000000000000000000000000
(qemu) info pg
VPN range      Entry      Flags      Physical page
[00000-003ff] PDE[000]   ----A----P
  [00000-000b7] PTE[000-0b7] -----WP 00000-000b7
  [000b8-000b8] PTE[0b8]   ---DA---WP 000b8
  [000b9-000ff] PTE[0b9-0ff] -----WP 000b9-000ff
  [00100-00102] PTE[100-102] ----A---WP 00100-00102
  [00103-0010f] PTE[103-10f] -----WP 00103-0010f
  [00110-00110] PTE[110]   ---DA---WP 00110
  [00111-00112] PTE[111-112] -----WP 00111-00112
  [00113-00113] PTE[113]   ---DA---WP 00113
  [00114-003ff] PTE[114-3ff] -----WP 00114-003ff
[f0000-f03ff] PDE[3c0]   ----A---WP
  [f0000-f00b7] PTE[000-0b7] -----WP 00000-000b7
  [f00b8-f00b8] PTE[0b8]   ---DA---WP 000b8
  [f00b9-f00ff] PTE[0b9-0ff] -----WP 000b9-000ff
  [f0100-f0102] PTE[100-102] ----A---WP 00100-00102
  [f0103-f010f] PTE[103-10f] -----WP 00103-0010f
  [f0110-f0110] PTE[110]   ---DA---WP 00110
  [f0111-f0112] PTE[111-112] -----WP 00111-00112
  [f0113-f0113] PTE[113]   ---DA---WP 00113
  [f0114-f03ff] PTE[114-3ff] -----WP 00114-003ff
(qemu)

```

Figure 20: 输入命令 info pg

打开另一个终端，进入 gdb，输入命令 `x/4` 反汇编后面 4 条指令，可以看到下一条将要执行的指令的地址是 `0xf0100482`，对应到物理地址就是 `0x00100482`。因此，在 QEMU monitor 终端输入命令 `xp /4iw 0x00100482` 反汇编后面 4 条指令（`i` 表示反汇编指令，`w` 指定输出的地址为 32 位）。

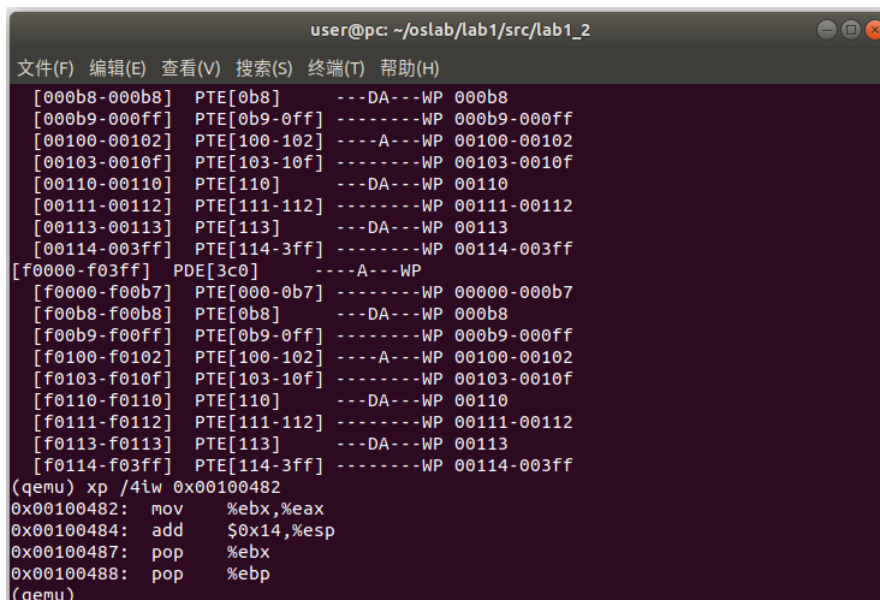


```

user@pc: ~/oslab/lab1/src/lab1_2
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
+ target remote localhost:26000
warning: No executable has been specified and target does not support
determining executable automatically. Try using the "file" command.
The target architecture is assumed to be i386
=> 0xf010047d: mov    $0xffffffff,%ebx
0xf010047d in ?? ()
+ symbol-file obj/kern/kernel
(gdb) x/4
0xf0100482 <kbd_proc_data+255>:      mov    %ebx,%eax
0xf0100484 <kbd_proc_data+257>:      add    $0x14,%esp
0xf0100487 <kbd_proc_data+260>:      pop    %ebx
0xf0100488 <kbd_proc_data+261>:      pop    %ebp
(gdb)

```

Figure 21: 在 gdb 中输入命令 `x/4`



```

user@pc: ~/oslab/lab1/src/lab1_2
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
[000b8-000b8] PTE[0b8] ---DA---WP 000b8
[000b9-000ff] PTE[0b9-0ff] -----WP 000b9-000ff
[00100-00102] PTE[100-102] ----A---WP 00100-00102
[00103-0010f] PTE[103-10f] -----WP 00103-0010f
[00110-00110] PTE[110] ---DA---WP 00110
[00111-00112] PTE[111-112] -----WP 00111-00112
[00113-00113] PTE[113] ---DA---WP 00113
[00114-003ff] PTE[114-3ff] -----WP 00114-003ff
[f0000-f03ff] PDE[3c0] ----A---WP
[f0000-f00b7] PTE[000-0b7] -----WP 00000-000b7
[f00b8-f00b8] PTE[0b8] ---DA---WP 000b8
[f00b9-f00ff] PTE[0b9-0ff] -----WP 000b9-000ff
[f0100-f0102] PTE[100-102] ----A---WP 00100-00102
[f0103-f010f] PTE[103-10f] -----WP 00103-0010f
[f0110-f0110] PTE[110] ---DA---WP 00110
[f0111-f0112] PTE[111-112] -----WP 00111-00112
[f0113-f0113] PTE[113] ---DA---WP 00113
[f0114-f03ff] PTE[114-3ff] -----WP 00114-003ff
(qemu) xp /4iw 0x00100482
0x00100482: mov    %ebx,%eax
0x00100484: add    $0x14,%esp
0x00100487: pop    %ebx
0x00100488: pop    %ebp
(qemu)

```

Figure 22: 在 QEMU monitor 中输入命令 `xp /4iw 0x00100482`

可以看到，这两张图中反汇编的 4 条指令是一样的，因此这两张图中虚拟地址和物

理地址是相互对应的。

3.3 问题 3

3.3.1 题目原文

```
假设以下内核代码是正确的，那么变量 x 将会是什么类型，uintptr_t 或者  
physaddr_t?  
mystery_t x;  
char* value = return_a_pointer();  
*value = 10;  
x = (mystery_t) value;
```

3.3.2 题目大意

变量x是表示物理地址还是虚拟地址？

3.3.3 回答

uintptr_t（虚拟地址）。

由实验文档可知，JOS 的两个 32 位地址值和 C 语言指针的关系如下：

C type	Address type	实际数据类型
T*	Virtual	指针类型
uintptr_t	Virtual	uint32_t
physaddr_t	Physical	uint32_t

Table 3: 3 种值的关系

从 CPU 执行的代码来看，一旦处在保护模式下，没有办法直接使用一个线性的或者物理地址，因为这两者是对程序员透明的。所有的内存引用被解释为虚拟地址，并且被 MMU 转换，它意味着所有在 C 程序中出现的指针都是虚拟地址。

uintptr_t和physaddr_t是 JOS 的两种地址值，分别表示虚拟地址和物理地址，都是uint32_t类型的。如果要取这些地址值对应的内容，必须先将它们强制转换为某种指针类型。如果将physaddr_t强制转换为指针类型，则 MMU 会将其作为虚拟地址处理，不会得到相应物理地址的值。

3.4 作业 4

3.4.1 题目原文

```
在文件 kern/pmap.c 文件中，你必须实现以下函数的代码。  
pgdir_walk()  
boot_map_region()  
page_lookup()  
page_remove()  
page_insert()  
  
mem_init()调用的 check_page()，用于测试你的页表管理方法。
```

3.4.2 题目大意

以上需要实现的函数组成了 JOS 的页表管理方法：实现虚拟地址到物理地址的映射，需要时还要创建页表页。mem_init()调用的函数check_page()会检查页表管理方法实现的正确性。

3.4.3 回答

关于虚拟内存，虚拟地址、线性地址和物理地址，分段和分页机制的背景知识已在练习 3 中阐述，故不再赘述。

参考计数就是每一个物理的页被映射的次数（一个物理页可能同时被多个虚拟地址映射），当参考计数的值为 0 时，这个物理页就需要被重新加入到free_page_list中。

下图是本次函数编写所用到的页表项或页目录项的标志位，在inc/mmu.h中定义：

```
// Page table/directory entry flags.  
#define PTE_P           0x001    // Present  
#define PTE_W           0x002    // Writeable  
#define PTE_U           0x004    // User
```

Figure 23: 标志位

其中PTE_P、PTE_W、PTE_U分别对应标志位 P、W、U。

现在要写的代码在 kern/pmap.c 中，都是关于虚拟地址和物理地址转换的代码。每一个物理页面对应一个PageInfo的结构体和一个物理页号 PPN（Physical Page Number）和物理首地址。

在 kern/pmap.h 中有这样几个函数定义：

```

static inline physaddr_t
page2pa(struct PageInfo *pp)
{
    return (pp - pages) << PGSHIFT;
}

static inline struct PageInfo*
pa2page(physaddr_t pa)
{
    if (PGNUM(pa) >= npages)
        panic("pa2page called with invalid pa");
    return &pages[PGNUM(pa)];
}

static inline void*
page2kva(struct PageInfo *pp)
{
    return KADDR(page2pa(pp));
}

```

Figure 24: 函数定义

其中,

- `page2pa()`实现从一个`PageInfo`到其对应的物理地址的转换。
- `pa2page()`实现从一个物理地址到相应的`PageInfo`的转换。
- `page2kva()`实现从一个`PageInfo`到其对应的虚拟内核地址的转换。

通过在`kern/pmap.c`文件中阅读这些函数注释, 可以知道它们对应的基本功能:

1. `pgdir_walk()`: 作用是查找一个虚拟地址对应的二级页表项地址 (PTE)
2. `boot_map_region()`: 映射一片指定虚拟页到指定物理页, 即`[va, va+size)`映射到`[pa, pa+size)`
3. `page_lookup()`: 返回虚拟地址对应的物理地址的页面描述
4. `page_remove()`: 对虚拟地址和其对应的物理页取消映射
5. `page_insert()`: 把虚拟地址映射到指定的物理页表

对函数`pgdir_walk()`的实现的关键是得到下图中红圈部分的二级页表项的虚拟地址:

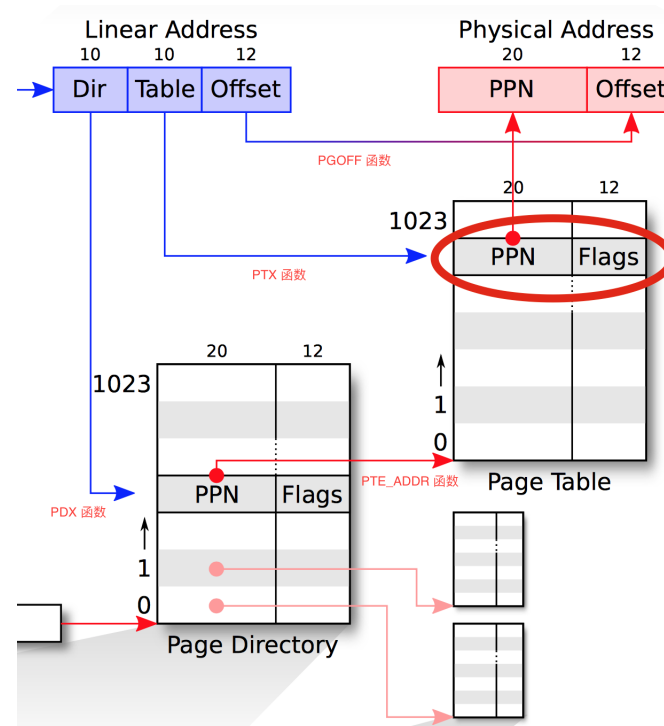


Figure 25: 转换流程及对应宏定义

```
// Given 'pgdir', a pointer to a page directory, pgdir_walk returns
// a pointer to the page table entry (PTE) for linear address 'va'.
// This requires walking the two-level page table structure.
//
// The relevant page table page might not exist yet.
// If this is true, and create == false, then pgdir_walk returns NULL.
// Otherwise, pgdir_walk allocates a new page table page with page_alloc.
//   - If the allocation fails, pgdir_walk returns NULL.
//   - Otherwise, the new page's reference count is incremented,
//     the page is cleared,
//     and pgdir_walk returns a pointer into the new page table page.
//
// Hint 1: you can turn a Page * into the physical address of the
// page it refers to with page2pa() from kern/pmap.h.
//
// Hint 2: the x86 MMU checks permission bits in both the page directory
// and the page table, so it's safe to leave permissions in the page
// directory more permissive than strictly necessary.
//
// Hint 3: look at inc/mmu.h for useful macros that manipulate page
// table and page directory entries.
//
```

Figure 26: 函数 pgdir_walk() 的注释

在上面的注释中，首先解释了pgdir_walk()函数中的参数的含义，pgdir是一个指向一级页表（页目录，page directory）的指针，然后va是虚拟地址。这个函数要返回这

个虚拟地址指向的页表项 (PTE)。如果这个 PTE 存在，那么返回这个 PTE 即可，如果不存在，参数 `create` 是指这个页表项是否需要被创建。若需要就创建一个页表项，不需要就返回 `NULL`。创建失败也返回 `NULL`，成功就为这个页表项的引用计数 +1。这里有一点需要注意的是，page directory 和 page table 中的 `pdt_t` 和 `pet_t` 都是物理地址，而返回值应该是一个虚拟地址，所以返回时需要使用 `KADDR` 宏实现从物理地址到虚拟内核地址的转换，或者手动加上 `KERNBASE`。其中，宏 `KADDR` 的定义在 `kern/pmap.h` 中，`KERNBASE` 的定义在 `inc/memlayout.h` 中。

```
/* This macro takes a physical address and returns the corresponding kernel
 * virtual address. It panics if you pass an invalid physical address. */
#define KADDR(pa) _kaddr(__FILE__, __LINE__, pa)

static inline void*
_kaddr(const char *file, int line, physaddr_t pa)
{
    if (PGNUM(pa) >= npages)
        _panic(file, line, "KADDR called with invalid pa %08lx", pa);
    return (void *) (pa + KERNBASE);
}
```

Figure 27: `kern/pmap.h` 中宏 `KADDR` 的定义

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

Figure 28: `inc/memlayout.h` 中 `KERNBASE` 的定义

因此代码实现如下：

```
1 // Given 'pgdir', a pointer to a page directory, pgdir_walk returns
2 // a pointer to the page table entry (PTE) for linear address 'va'.
3 // This requires walking the two-level page table structure.
4 // The relevant page table page might not exist yet.
5 // If this is true, and create == false, then pgdir_walk returns NULL.
6 // Otherwise, pgdir_walk allocates a new page table page with page_alloc.
7 //   - If the allocation fails, pgdir_walk returns NULL.
8 //   - Otherwise, the new page's reference count is incremented,
9 //   the page is cleared,
10 // and pgdir_walk returns a pointer into the new page table page.
11 // Hint 1: you can turn a Page * into the physical address of the
12 // page it refers to with page2pa() from kern/pmap.h.
13 //
14 // Hint 2: the x86 MMU checks permission bits in both the page directory
15 // and the page table, so it's safe to leave permissions in the page
16 // directory more permissive than strictly necessary.
```

```

17 //
18 // Hint 3: look at inc/mmu.h for useful macros that manipulate page
19 // table and page directory entries.
20 //
21 pte_t *
22 pgdir_walk(pde_t *pgdir, const void *va, int create)
23 {
24     // Fill this function in
25     // 参数*pgdir: 页目录项指针
26     // 参数*va: 线性地址, JOS 中等于虚拟地址
27     // 参数create: 若页表项不存在是否创建
28     // 返回: 页表项指针
29     uint32_t page_dir_idx = PDX(va); // 得到这个虚拟地址所在的页目录偏移
30     uint32_t page_tab_idx = PTX(va); // 计算该虚拟地址对应的页表偏移量
31     pte_t *pgtab; // 对应页表的指针
32     if (pgdir[page_dir_idx] & PTE_P) { // 如果存在这页, 即标志位P为1
33         pgtab = KADDR(PTE_ADDR(pgdir[page_dir_idx])); // 通过页目录表pgdir和页目
            录偏移求得这页在page directory的地址。
34     }
35     else { // 如果不存在这页, 就要根据create的值决定是否创建
36         if (create) {
37             struct PageInfo *new_pageInfo = page_alloc(ALLOC_ZERO);
38             if (new_pageInfo) {
39                 new_pageInfo->pp_ref += 1;
40                 pgtab = (pte_t *)page2kva(new_pageInfo);
41                 // 修改页目录的flag, 根据 check_page 函数中用到的属性。
42                 // 因为分配以页为单位对齐, 必然最后12bit为0
43                 pgdir[page_dir_idx] = PADDR(pgtab) | PTE_P | PTE_W | PTE_U; // 创建一
                    个新的页目录项
44             }
45             else {
46                 return NULL; // 没空间创建了返回NULL
47             }
48         }
49         else {
50             return NULL; // create为0直接返回NULL
51         }
52     }
53     return &pgtab[page_tab_idx]; // 返回对应的二级页表项指针
54 }

```

boot_map_region()函数的思路就是利用pgdir_walk()。此时的va类型是uintptr_t,

是无符号 32 位整数 `uint32_t`, 调用 `pgdir_walk()` 时需要先强制转换为指针类型 `void *`。

```
//
// Map [va, va+size) of virtual address space to physical [pa, pa+size)
// in the page table rooted at pgdir. Size is a multiple of PGSIZE, and
// va and pa are both page-aligned.
// Use permission bits perm|PTE_P for the entries.
//
// This function is only intended to set up the ``static'' mappings
// above UTOP. As such, it should *not* change the pp_ref field on the
// mapped pages.
//
// Hint: the TA solution uses pgdir_walk
```

Figure 29: 函数 `boot_map_region()` 的注释

这段注释说明了 `boot_map_region()` 函数的功能, 将虚拟内存空间 `[va, va+size)` 映射到物理空间 `[pa, pa+size)` 这个映射关系加入到页目录 `pgdir` 中。这个函数主要的目的是为了设置虚拟地址 `UTOP` 之上的地址范围, 这一部分的地址映射是静态的, 在操作系统的运行过程中不会改变, 所以这个页的 `PageInfo` 结构体中的 `pp_ref` 域的值不会发生改变。因此需要完成一个循环, 在每一轮中, 把一个虚拟页和物理页的映射关系放到响应的页表项中。直到把 `size` 个字节的内存都分配完。

因此代码实现如下:

```
1 // Map [va, va+size) of virtual address space to physical [pa, pa+size)
2 // in the page table rooted at pgdir. Size is a multiple of PGSIZE, and
3 // va and pa are both page-aligned.
4 // Use permission bits perm|PTE_P for the entries.
5 //
6 // This function is only intended to set up the ``static'' mappings
7 // above UTOP. As such, it should *not* change the pp_ref field on the
8 // mapped pages.
9 //
10 // Hint: the TA solution uses pgdir_walk
11 static void
12 boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa, int
    perm)
13 {
14     // Fill this function in
15     pte_t *pgtab;
16     size_t pg_num = PGNUM(size);
17     size_t i;
18     cprintf("map region size = %d, %d pages\n", size, pg_num);
```



```

19  for (i = 0; i < pg_num; i++)
20  {
21      //在每次循环中，把一个虚拟页和一个物理页的映射关系放到对应的页表项中
22      pgtab = pgdir_walk(pgdir, (void *)va, 1); //从页目录中得到va这块地址的的
        表项
23      if (!pgtab) { //如果并未获得相应表项，就直接返回
24          return;
25      }
26      *pgtab = pa | perm | PTE_P;
27      va += PGSIZE;
28      pa += PGSIZE;
29  }
30 }

```

page_lookup()函数的注释如下:

```

//
// Return the page mapped at virtual address 'va'.
// If pte_store is not zero, then we store in it the address
// of the pte for this page. This is used by page_remove and
// can be used to verify page permissions for syscall arguments,
// but should not be used by most callers.
//
// Return NULL if there is no page mapped at va.
//
// Hint: the TA solution uses pgdir_walk and pa2page.
//

```

Figure 30: 函数 page_lookup() 的注释

这一段注释说page_lookup()函数功能是：返回虚拟地址va映射的页PageInfo结构体的指针。然后如果参数pte_store!=0,那么我们将页表项pte的地址存储到pte_store中。如果va处没有物理页，那么返回NULL。因此只需要调用pgdir_walk()函数获取这个va对应的页表项，然后判断这个页是否已经在内存中，如果在则返回这个页的PageInfo结构体指针。并且把这个页表项的内容存放到pte_store中。

因此代码实现如下:

```

1 //
2 // Return the page mapped at virtual address 'va'.
3 // If pte_store is not zero, then we store in it the address
4 // of the pte for this page. This is used by page_remove and
5 // can be used to verify page permissions for syscall arguments,
6 // but should not be used by most callers.
7 //

```

```

8 // Return NULL if there is no page mapped at va.
9 //
10 // Hint: the TA solution uses pgdir_walk and pa2page.
11 struct PageInfo *
12 page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
13 {
14     // Fill this function in
15     // 参数pgdir: 页目录指针
16     // 参数va: 线性地址
17     // 参数3: 指向页表指针的指针
18     // 返回: 页描述结构体指针
19     pte_t *pgtab = pgdir_walk(pgdir, va, 0); // 不创建, 只查找
20     if (!pgtab) {
21         return NULL; // 未找到则返回 NULL
22     }
23     if (pte_store) {
24         *pte_store = pgtab; // 将页表项pte的地址存储到pte_store中
25     }
26     return pa2page(PTE_ADDR(*pgtab)); // 返回页面描述
27 }

```

page_remove() 函数的注释如下:

```

//
// Unmaps the physical page at virtual address 'va'.
// If there is no physical page at that address, silently does nothing.
//
// Details:
// - The ref count on the physical page should decrement.
// - The physical page should be freed if the refcount reaches 0.
// - The pg table entry corresponding to 'va' should be set to 0.
//   (if such a PTE exists)
// - The TLB must be invalidated if you remove an entry from
//   the page table.
//
// Hint: The TA solution is implemented using page_lookup,
//       tlb_invalidate, and page_decref.
//

```

Figure 31: 函数 page_remove() 的注释

这段注释说page_remove()函数功能是: 取消虚拟地址 va 处的物理页映射。如果这个地址上本来就没有物理页, 那么就不用取消。要注意以下细节: 此处物理页的引用计数要减 1, 然后要将它释放掉, 如果这个地址的页表项存在, 那么页表项要置 0, 从

页表中移除一个表项时要将 TLB 置为无效（否则 CPU 仍会根据该 TLB 表项认为该页表项有效）。

因此代码实现如下：

```

1 //
2 // Unmaps the physical page at virtual address 'va'.
3 // If there is no physical page at that address, silently does nothing.
4 //
5 // Details:
6 //   - The ref count on the physical page should decrement.
7 //   - The physical page should be freed if the refcount reaches 0.
8 //   - The pg table entry corresponding to 'va' should be set to 0.
9 //     (if such a PTE exists)
10 //   - The TLB must be invalidated if you remove an entry from
11 //     the page table.
12 //
13 // Hint: The TA solution is implemented using page_lookup,
14 //       tlb_invalidate, and page_deceref.
15 void
16 page_remove(pde_t *pgdir, void *va)
17 {
18     // Fill this function in
19     pte_t *pgtab;
20     pte_t **pte_store = &pgtab;
21     struct PageInfo *pInfo = page_lookup(pgdir, va, pte_store); // 找到va对应的
        物理页
22     if (!pInfo) { // 如果不存在，就直接返回
23         return;
24     }
25     page_deceref(pInfo); // 将pInfo页的引用计数-1
26     *pgtab = 0; // 将内容清0，即无法再根据页表内容得到物理地址。
27     // 取消va对应物理页之间的关联，相当于刷新TLB
28     // 每次我们调整虚拟页和物理页之间的映射关系的时候，我们都要刷新TLB
        ，调用这个函数或invlpg汇编指令
29     tlb_invalidate(pgdir, va); // 通知tlb失效，tlb是个高速缓存，用来缓存查找
        记录增加查找速度。
30 }

```

需要注意的是，这里我们调用了 `tlb_invalidate(pgdir, va)` 函数，在 `kern/pmap.c` 的后面部分，可以看到此函数体如下：

```
//
// Invalidate a TLB entry, but only if the page tables being
// edited are the ones currently in use by the processor.
//
void
tlb_invalidate(pde_t *pgdir, void *va)
{
    // Flush the entry only if we're modifying the current address space.
    // For now, there is only one address space, so always invalidate.
    invlpg(va);
}
```

Figure 32: 函数 tlb_invalidate(pgdir,va)

其中tlb_invalidate()函数中调用了inc/x86.h中的invlpg()函数, 在这里需要了解一下, 这些函数都使用了__asm __volatile()内嵌了汇编指令, 以下是invlpg()函数:

```
static __inline void
invlpg(void *addr)
{
    __asm __volatile("invlpg (%0)" : : "r" (addr) : "memory");
}
```

Figure 33: 函数 invlpg()

这段嵌入式汇编是没有输出的。作用是把addr读入到寄存器中去（这里r指定任意常用寄存器），%0则指向addr所在的寄存器。

处理器使用 TLB（Translation Lookaside Buffer）来缓存线性地址到物理地址的映射关系。实际的地址转换过程中，处理器首先根据线性地址查找 TLB，如果未发现该线性地址到物理地址的映射关系（TLB miss），将根据页表中的映射关系填充 TLB（TLB fill），然后再进行地址转换。

这里tlb_invalidate(pgdir,va)函数的目的就是取消 va 对应物理页之间的关联，相当于刷新 TLB，每次我们调整虚拟页和物理页之间的映射关系的时候，我们都要刷新 TLB，调用invlpg()函数或直接嵌入invlpg汇编指令。

最后是page_insert() 函数，注释如下图：

```
//
// Map the physical page 'pp' at virtual address 'va'.
// The permissions (the low 12 bits) of the page table entry
// should be set to 'perm|PTE_P'.
//
// Requirements
//   - If there is already a page mapped at 'va', it should be page_remove().
//   - If necessary, on demand, a page table should be allocated and inserted
//     into 'pgdir'.
//   - pp->pp_ref should be incremented if the insertion succeeds.
//   - The TLB must be invalidated if a page was formerly present at 'va'.
//
// Corner-case hint: Make sure to consider what happens when the same
// pp is re-inserted at the same virtual address in the same pgdir.
// However, try not to distinguish this case in your code, as this
// frequently leads to subtle bugs; there's an elegant way to handle
// everything in one code path.
//
// RETURNS:
//   0 on success
//   -E_NO_MEM, if page table couldn't be allocated
//
// Hint: The TA solution is implemented using pgdir_walk, page_remove,
// and page2pa.
//
```

Figure 34: 函数 page_insert() 的注释

这段注释说page_insert()函数功能是：把pp描写叙述的物理页与虚拟地址va建立映射。假设va所在的虚拟内存页不存在，那么应指定pgdir_walk()的create参数为1，创建这个虚拟地址页。假设va所在的虚拟内存页存在，那么取消当前va的虚拟内存页和之前物理页的关联，即调用page_remove()，而且为va创建新的物理页联系——pp所描述的物理页。

分配页面失败时返回-E_NO_MEM，插入时分三种情况：

- 插入位置不存在页面，直接插入即可
- 插入位置存在不相同的页面，先移除该页面再插入
- 插入位置存在相同页面，设置权限即可

因此代码实现如下：

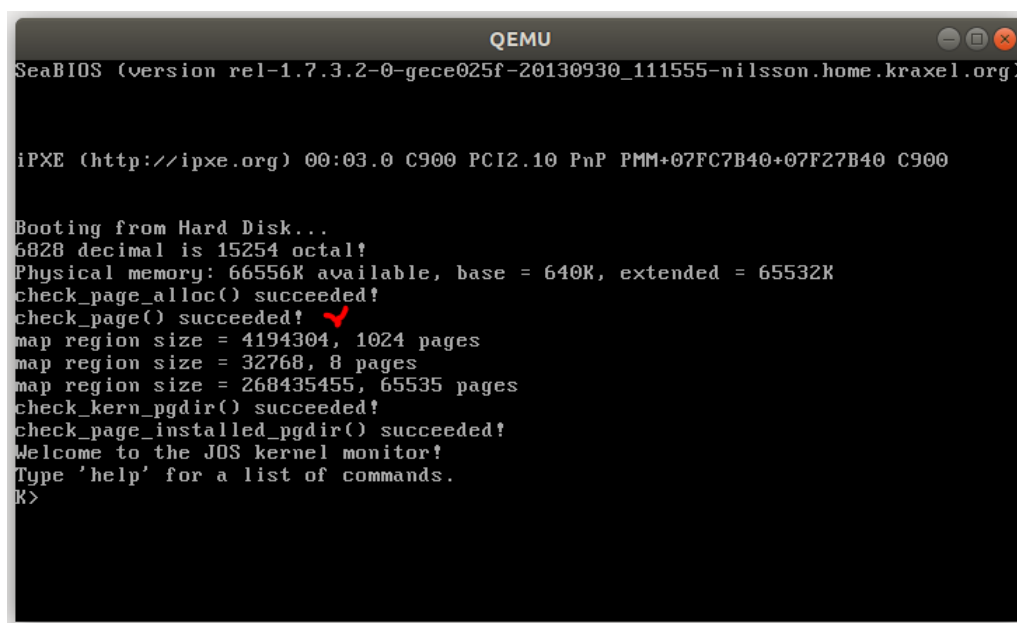
```
1 //
2 // Map the physical page 'pp' at virtual address 'va'.
3 // The permissions (the low 12 bits) of the page table entry
4 // should be set to 'perm|PTE_P'.
```

```

5 //
6 // Requirements
7 //   - If there is already a page mapped at 'va', it should be page_remove
    ()d.
8 //   - If necessary, on demand, a page table should be allocated and
    inserted
9 //     into 'pgdir'.
10 //   - pp->pp_ref should be incremented if the insertion succeeds.
11 //   - The TLB must be invalidated if a page was formerly present at 'va'.
12 //
13 // Corner-case hint: Make sure to consider what happens when the same
14 // pp is re-inserted at the same virtual address in the same pgdir.
15 // However, try not to distinguish this case in your code, as this
16 // frequently leads to subtle bugs; there's an elegant way to handle
17 // everything in one code path.
18 //
19 // RETURNS:
20 //   0 on success
21 //   -E_NO_MEM, if page table couldn't be allocated
22 //
23 // Hint: The TA solution is implemented using pgdir_walk, page_remove,
24 // and page2pa.
25 //
26 int
27 page_insert(pde_t *pgdir, struct PageInfo *pp, void *va, int perm)
28 {
29     // Fill this function in
30     pte_t *pgtab = pgdir_walk(pgdir, va, 1); //通过pgdir_walk函数求出va对应的
        页目录表项
31     if (!pgtab) { //表示va并不在页目录表中
32         return -E_NO_MEM;
33     }
34     // 这里一定要提前增加引用
35     pp->pp_ref++; //修改引用计数值
36     if (*pgtab & PTE_P) { //如果这个虚拟地址已有物理页与之映射
37         tlb_invalidate(pgdir, va); //TLB无效
38         page_remove(pgdir, va); //删除这个映射
39     }
40     *pgtab = page2pa(pp) | perm | PTE_P;
41     pgdir[PDX(va)] |= perm; //把va和pp的映射关系插到页目录中
42     return 0;
43 }

```

以下是check_page()的成功报告:



```
QEMU
SeaBIOS (version rel-1.7.3.2-0-gece025f-20130930_111555-nilsson.home.kraxel.org)

iPXE (http://ipxe.org) 00:03.0 C900 PCI2.10 PnP PMM+07FC7B40+07F27B40 C900

Booting from Hard Disk...
6828 decimal is 15254 octal!
Physical memory: 66556K available, base = 640K, extended = 65532K
check_page_alloc() succeeded!
check_page() succeeded! ✓
map region size = 4194304, 1024 pages
map region size = 32768, 8 pages
map region size = 268435455, 65535 pages
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

Figure 35: 成功报告

4 内核地址空间

4.1 权限和故障隔离

JOS 将处理器的 32 位线性地址空间分成了两部分。用户环境（进程）将会管理低位内存的内容,内核维护高位内存的内容。低位内存和高位内存的分隔线由 `inc/memlayout.h` 中的 ULIM 符号定义,为内核保留大约 256MB 的虚拟地址空间。用户对于 ULIM 的上面都没有访问权限,而 UTOP 到 ULIM 之间用户和内核都有访问权限（但是这部分内存是只读的,不可以修改,在这部分内存中,主要存储内核的一些只读数据）。 $[0, \text{ULIM})$ 的空间是留给用户的（除了空页以及符号表所在的页以外）。由 Lab 1 可知,把操作系统内核映射到高的虚拟地址空间中是为了给用户程序留出足够的虚拟地址空间。在页表中使用权限位的原因是使用户和内核隔离开来,避免用户出现的故障蔓延到内核。

4.2 作业 5

4.2.1 题目原文

在调用 `check_page()` 之后，填写 `mem_init()` 丢失的代码。你的代码需要通过 `check_kern_pgdir()` 和 `check_installed_pgdir()` 的检验。

4.2.2 题目大意

补充 `mem_init()` 在函数调用 `check_page()` 之后缺失的代码。`check_kern_pgdir()` 和 `check_installed_pgdir()` 将会检验补充代码的正确性。

4.2.3 回答

由 `mem_init()` 所在的文件 `kern/pmap.c` 可知，在 `check_page()` 之后一共需要补充 3 行代码。

首先在 `kern/pmap.c` 中定位到相应位置，要补充的第一段代码的注释如图所示。

```

////////////////////////////////////
// Now we set up virtual memory

////////////////////////////////////
// Map 'pages' read-only by the user at linear address UPAGES
// Permissions:
//   - the new image at UPAGES -- kernel R, user R
//   (ie. perm = PTE_U | PTE_P)
//   - pages itself -- kernel RW, user NONE
// Your code goes here:

```

Figure 36: 需要补充的第一段代码的注释

根据注释信息了解到，此处补充的代码需要实现将线性地址 `UPAGES` 以用户和内核均只读的方式映射到 `pages` 上的功能。由以上的作业 4 可知，`boot_map_region` 函数将用户空间的一块虚拟内存映射到存储该数据结构的物理内存上，它完成了这项功能，此函数的声明如下所示，这些参数的作用分别为页目录指针，虚拟地址，需要映射的地址长度，物理内存地址以及权限。

```
static void boot_map_region(pde_t *pgdir, uintptr_t va, size_t size,
physaddr_t pa, int perm);
```

因此根据要求，函数的这五个参数分别填写为：`kern_pgdir`，指向页表目录的指针；`UPAGE`，需要映射的虚拟地址；`PTSIZE`，需要映射的地址长度，由 `inc/memlayout.h` 中的注释代码，即第二张图可知 `UPAGES` 的大小为 `PTSIZE`；`PADDR((uintptr_t *) pages)`，映射的目标物理地址；`PTE_U`，表示用户可以访问的只读权限。因此需要填写的完整代码如下所示。

```
boot_map_region(kern_pgdir, UPAGES, PTSIZE, PADDR((uintptr_t *) pages), PTE_U);
```



```

+-----+
*      :          .          :
*      :          .          :
* MMIO LIM -----> +-----+ 0xefc00000 -
* | Memory-mapped I/O | RW/-- PTSIZE
* ULIM, MMIOBASE --> +-----+ 0xef800000
* | Cur. Page Table (User R-) | R-/R- PTSIZE
* UVPT -----> +-----+ 0xef400000
* | RO PAGES | R-/R- PTSIZE
* UPAGES -----> +-----+ 0xef000000
* | RO ENV S | R-/R- PTSIZE
* UTOP,UENV S -----> +-----+ 0xeec00000
* UXSTACKTOP -/ | User Exception Stack | RW/RW PGSIZE
* | Empty Memory (*) | --/-- PGSIZE
* USTACKTOP ----> +-----+ 0xebbf0000
* | Normal User Stack | RW/RW PGSIZE
* +-----+ 0xebfd0000
* |
* |
* |

```

Figure 37: UPAGE 的地址、权限、大小信息

下图显示了需要补充的第二段代码。

```

////////////////////////////////////
// Use the physical memory that 'bootstack' refers to as the kernel
// stack. The kernel stack grows down from virtual address KSTACKTOP.
// We consider the entire range from [KSTACKTOP-PTSIZE, KSTACKTOP)
// to be the kernel stack, but break this into two pieces:
//     * [KSTACKTOP-KSTKSIZE, KSTACKTOP) -- backed by physical memory
//     * [KSTACKTOP-PTSIZE, KSTACKTOP-KSTKSIZE) -- not backed; so if
//       the kernel overflows its stack, it will fault rather than
//       overwrite memory. Known as a "guard page".
// Permissions: kernel RW, user NONE
// Your code goes here:

```

Figure 38: 要补充的第二段代码要求

根据注释可知，代码需要将内核栈的地址映射到**bootstack**中，映射时，内核栈区域的地址**[KSTACKTOP-PTSIZE, KSTACKTOP)**需要分成两部分，一部分为**[KSTACKTOP-KSTKSIZE, KSTACKTOP)**，这部分被物理内存所支持，即操作系统会把这部分地址与物理内存之间做映射，并加入页表中，而另一部分**[KSTACKTOP-PTSIZE, KSTACKTOP-KSTKSIZE)**，则不被映射，其目的是设置一个守护页，来避免内核栈发生溢出时将低地址位的数据覆盖。同样使用**boot_map_region**函数，其参数分别为**kern_pgdir**，指向页表目录的指针；**KSTACKTOP-KSTKSIZE**，需要映射区域的起始地址；**KSTKSIZE**，需要映射的地址长度（为内核栈的长度）；**PADDR((uintptr_t*) bootstack)**，映射到的物理内存区域；以及**PTE_W**，因为根据注释内容或图四均可看到，此部分是内核具有读写权限而用户进程没有任何权限的。

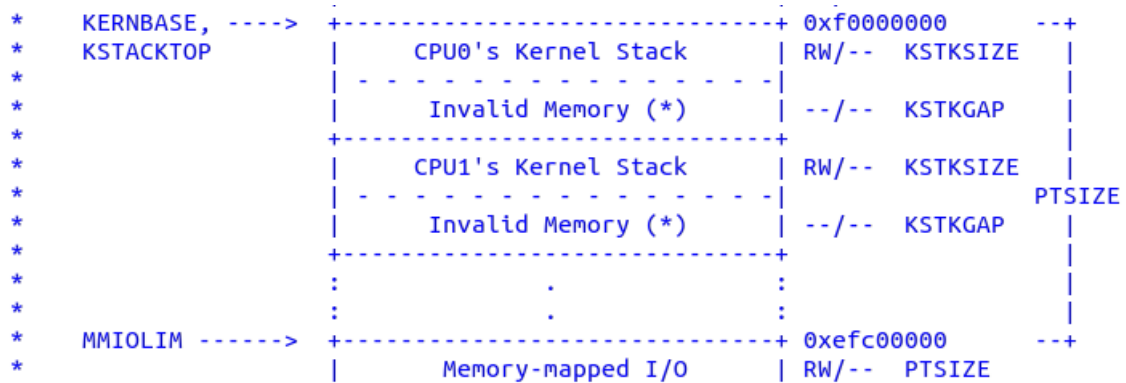


Figure 39: [KSTACKTOP-PTSIZE, KSTACKTOP) 区域的地址、权限、大小信息

所以，要补充的代码为：

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

需要补充的第三段代码的要求如下图所示。

```
////////////////////////////////////
// Map all of physical memory at KERNBASE.
// Ie. the VA range [KERNBASE, 2^32) should map to
// the PA range [0, 2^32 - KERNBASE)
// We might not have 2^32 - KERNBASE bytes of physical memory, but
// we just set up the mapping anyway.
// Permissions: kernel RW, user NONE
// Your code goes here:
```

Figure 40: 要补充的第三段代码要求

可知，此处的代码需要将内核虚拟地址区域映射到整个的物理内存地址上。内核地址区域为[KERNBASE, 2³²)，物理内存地址范围为[0, 2³² - KERNBASE)。同时，要求中也提到，尽管物理内存的大小并没有2³² - KERNBASE那么大，但我们还是将它做映射。这里依然使用boot_map_region函数做映射，其参数分别为kern_pgdir，指向页目录的指针；KERNBASE，内核虚拟地址的起始地址；0xffffffff - KERNBASE，内核区域的长度；(physaddr_t)0，映射到的物理内存区域的起始地址；以及PTE_W，代表内核具有读写权限，而用户不具有任何权限。因此，所需要输入的代码如下所示。

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

须注意，这里3个函数调用的权限位均不需要写PTE_P是因为函数内部已经自己默认赋予此权限了。详见作业4中boot_map_region()函数的具体实现。

最后，在终端输入make qemu，可以看到check_kern_pgdir()和check_installed_pgdir()都已经验证通过。

```

QEMU
SeaBIOS (version rel-1.7.3.2-0-gece025f-20130930_111555-nilsson.home.kraxel.org)

iPXE (http://ipxe.org) 00:03.0 C900 PCI2.10 PnP PMM+07FC7B40+07F27B40 C900

Booting from Hard Disk...
6828 decimal is 15254 octal!
Physical memory: 66556K available, base = 640K, extended = 65532K
check_page_alloc() succeeded!
check_page() succeeded!
map region size = 4194304, 1024 pages
map region size = 32768, 8 pages
map region size = 268435455, 65535 pages
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>

```

Figure 41: 验证通过作业 5

4.3 问题 4

4.3.1 题目原文

1) 在这一点上页目录的哪些行已经被填写了？他们映射了什么地址，指向了哪？换言之，尽量填写以下表：

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]

2) 我们已经将内核和用户环境放在了相同的地址空间，为什么用户程序不能读或者写内核内存？什么样的具体机制保护内核内存？

3) 这个操作系统最大能支持多大的物理内存？为什么？

4) 管理内存有多大的空间开销，如果我们拥有最大的物理内存？这个空间开销如何减

小？

4.3.2 回答

1) 根据 `inc/memlayout.h` 的 Virtual memory map, 可以得到下列虚拟内存布局:

```

/* Virtual memory map:                                     Permissions
                                                            kernel/user
*/
*      4 Gig -----> +-----+ | RW/--
*                      |         |
*                      |         |
*                      |         |
*                      |         |
*                      |         |
*                      |         |
* KERNBASE, ----> +-----+ | 0xf0000000 --+
KSTACKTOP          | CPU0's Kernel Stack   | RW/-- KSTKSIZ
                  | Invalid Memory (*)     | --/-- KSTKGAP
                  +-----+
                  | CPU1's Kernel Stack    | RW/-- KSTKSIZ
                  | Invalid Memory (*)     | --/-- KSTKGAP PTSIZE
                  +-----+
                  | .                       | :
                  |                         | :
MMIOLIM -----> +-----+ | 0xefc00000 ---+
                  | Memory-mapped I/O      | RW/-- PTSIZE
ULIM, MMIOBASE -> +-----+ | 0xef800000
                  | Cur. Page Table (User R-) | R-/R- PTSIZE
UVPT -----> +-----+ | 0xef400000
                  | RO PAGES                 | R-/R- PTSIZE
UPAGES -----> +-----+ | 0xef000000
                  | RO ENV                   | R-/R- PTSIZE
UTOP,UENVS ----> +-----+ | 0xeec00000
UXSTACKTOP -/    | User Exception Stack     | RW/RW PGSIZE
                  | Empty Memory (*)        | --/-- PGSIZE USTACKTOP ->| 0xeebf000
                  | Normal User Stack       | RW/RW PGSIZE
                  |                         | 0xeebfd000
                  +-----+
                  |                         |
                  |                         |
                  |                         |
                  |                         |
                  | Program Data & Heap     |
UTEVT -----> +-----+ | 0x00800000
PFTEMP -----> | Empty Memory (*)           | PTSIZE
UTEMP -----> +-----+ | 0x00400000 --+
                  | Empty Memory (*)        |
                  | User STAB Data (optional) | PTSIZE
USTABDATA ----> +-----+ | 0x00200000
                  | Empty Memory (*)        |
0 -----> +-----+
/*
(*) Note: The kernel ensures that "Invalid Memory" is *never* mapped.
      "Empty Memory" is normally unmapped, but user programs may map pages
      there if desired. JOS user programs map pages temporarily at UTEMP.
*/

```

Figure 42: 虚拟内存布局

以及物理内存布局和虚拟内存布局的映射关系如下图:

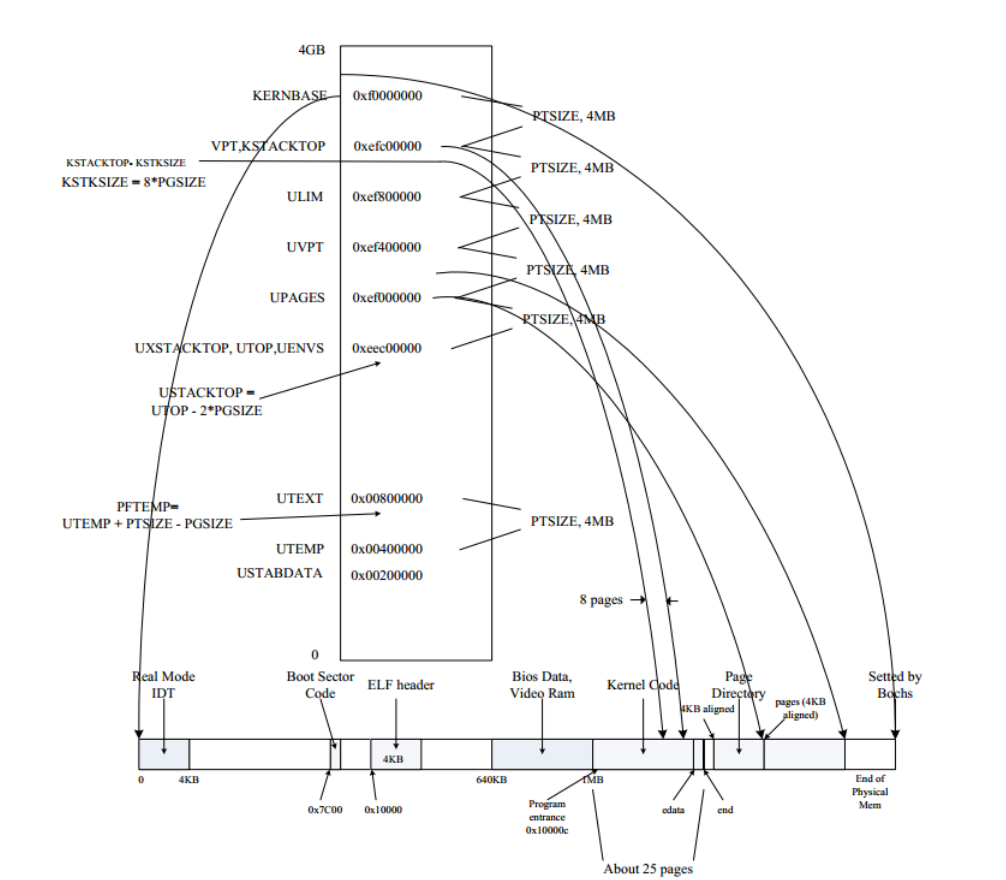


Figure 43: 物理内存布局和虚拟内存布局的映射关系

其中,

- **[USTABDATA, UTEXT]、[UTEXT, USTACKTOP]**: 用户程序的 `.text` 段以及用户堆栈公用区域, 两个区域从两头向中间生长;
- **[USTACKTOP, UPAGES]、[UPAGES, UVPT]**: 在虚拟地址中这段内存就是对应的在实际物理地址里 `pages` 数组对应存储位置;
- **[UVPT, ULIM) 和 [VPT, KERNBASE)**: 这两个地址映射了同一个系统页目录, 即 `pgdir`, 对用户开放只读权限, 使用户可以得到当前内存中某个虚拟地址对应的物理页面地址是多少。
- **[KERNBASE, 4GB)**: 这部分映射实际物理内存中从 0 开始的中断向量表 IDT、BIOS 程序、IO 端口以及操作系统内核等, 对应的物理地址范围是 **[0, 4GB-KERNBASE]**。

因此上表可补充如下:

Entry	Base Virtual Address	Points to (logically)
1023	0xffc00000	Page table for [252,256)MB (top 4MB) of phys memory, the last address finding page table that the kernel can use.
1022	0xff800000	Page table for [248,252)MB of phys memory
...
961	0xf0400000	Page table for [4,8)MB of phys memory
960	0xf0000000 (KERNBASE)	Page table for [0,4)MB of phys memory
959	0xefc00000 (KSTACK-PTSIZE)	Kernel stack
958	0xef800000	ULIM
957	0xef400000 (UVPT)	kern_pgdir
956	0xef000000 (UPAGES)	Array of PageInfo
...
2	0x00800000	NULL
1	0x00400000	NULL
0	0x00000000	The start of the virtual memory, same as 960

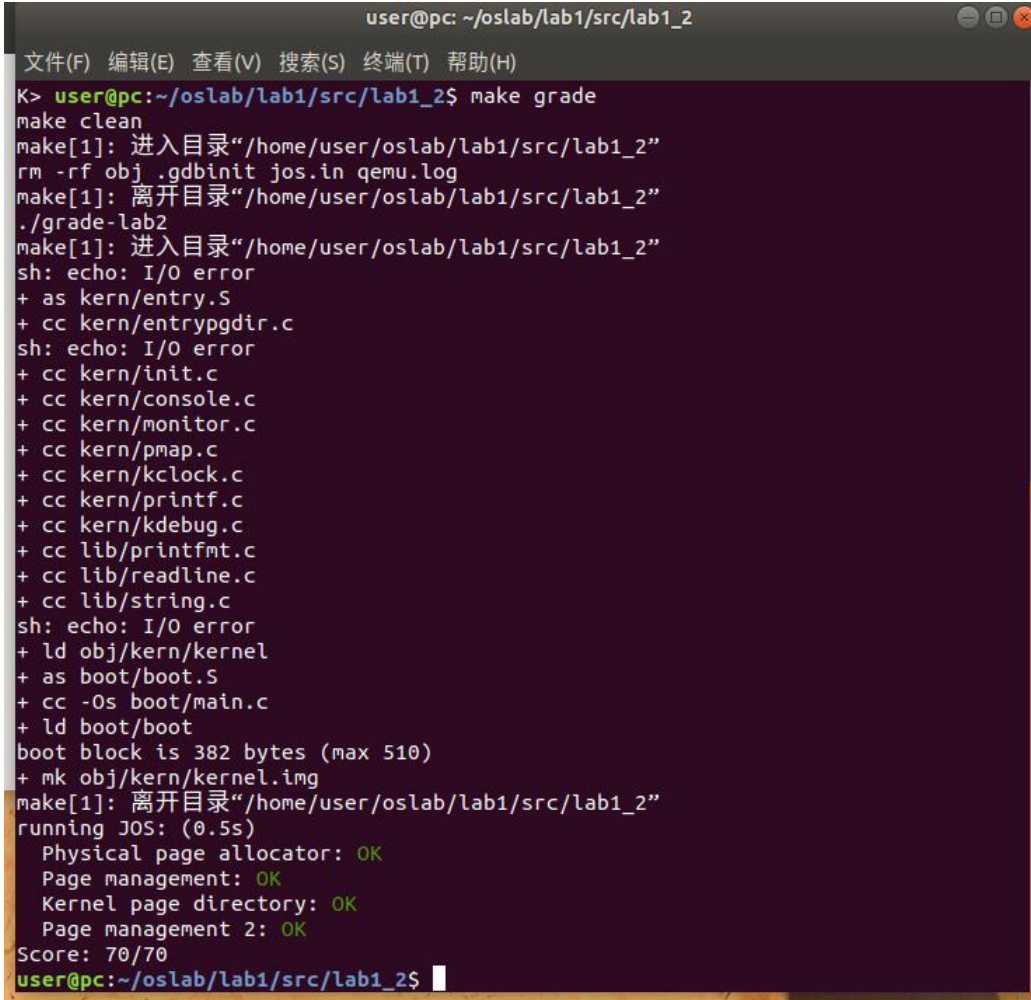
Table 4: 第 1 问的表

2) 在inc/mmu.h中可以看到页表有权限位 U (对应PTE_U), 如果将该权限位置 0 则用户无权限读写。

3) 所有的空闲的物理页面一开始都储存在了pages这样一个数组中, 这个数组存储的基本单元是PageInfo, 由PageInfo结构体的定义可知一个PageInfo的大小是sizeof(struct PageInfo)=8 Byte。由上述的虚拟内存布局可知, UPAGE是用来存放pages的数组的, 它的起始地址是0xef000000, 与之相邻的下一个符号UVPT的起始地址为0xef400000, 这两个的起始地址之间相隔了一个PTSIZE(一个页表的物理内存大小)。由inc/mmu.h可知, PTSIZE的值为 $1024 \times 4096B = 4MB$ 。即因此pages数组最多只能占4MB的空间, 也就是说最多 $4MB/8B = 512K$ 个页面, 每个页面容量 $2^{12} = 4K$ 。系统支持最大 $512K \times 4K = 2GB$ 的物理内存。

4) 对于 32 位操作系统而言, 有 32 根地址线, 虚拟地址空间为4GB。页表实际的空间开销只与虚拟地址空间有关而不是实际的物理地址空间, 尽管 JOS 只支持最大2GB物理内存。如果进行二级分页。高 10 位是在页目录中的索引, 中间 10 位是在页表中的索引, 最后 12 位是页内偏移。因此有一个页目录和 1024 个页表, 每个页表都有 1024 个页表项。一个页表项占4B。因此页表的空间开销为 $(1024+1) \times 4B \times 1024 = 4MB + 4KB$ 。由于pages数组占4MB, 因此总的空间开销为 $4MB + 4MB + 4KB = 8196KB$ 。消减管理内存开支的一个办法是在地址总线位数不变的情况下是增大页面容量, 比如把页面大小改成8KB, 这样可以使得相同的存储开销存储的页能够映射更多的物理地址空间来减少内存开销。

至此，Lab 2 的内容已完成。以下是在终端输入命令 `make grade` 之后得到的打分情况：



```
user@pc: ~/oslab/lab1/src/lab1_2
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
K> user@pc:~/oslab/lab1/src/lab1_2$ make grade
make clean
make[1]: 进入目录"/home/user/oslab/lab1/src/lab1_2"
rm -rf obj .gdbinit jos.in qemu.log
make[1]: 离开目录"/home/user/oslab/lab1/src/lab1_2"
./grade-lab2
make[1]: 进入目录"/home/user/oslab/lab1/src/lab1_2"
sh: echo: I/O error
+ as kern/entry.S
+ cc kern/entrypgdir.c
sh: echo: I/O error
+ cc kern/init.c
+ cc kern/console.c
+ cc kern/monitor.c
+ cc kern/pmap.c
+ cc kern/kclock.c
+ cc kern/printf.c
+ cc kern/kdebug.c
+ cc lib/printfmt.c
+ cc lib/readline.c
+ cc lib/string.c
sh: echo: I/O error
+ ld obj/kern/kernel
+ as boot/boot.S
+ cc -Os boot/main.c
+ ld boot/boot
boot block is 382 bytes (max 510)
+ mk obj/kern/kernel.img
make[1]: 离开目录"/home/user/oslab/lab1/src/lab1_2"
running JOS: (0.5s)
  Physical page allocator: OK
  Page management: OK
  Kernel page directory: OK
  Page management 2: OK
Score: 70/70
user@pc:~/oslab/lab1/src/lab1_2$
```

Figure 44: 打分情况

5 总结

做了 Lab 2 以后，感觉还是学到了不少关于内存管理的知识。有了 Lab 1 的经验之后，果然这次稍微轻松了一点，也知道该怎么下手了。通过尝试对代码的阅读和修改，上课听不懂的二级页表机制似乎已经明白了一些，也对页表项的结构和权限位有了更深入的认识。也认识到在页表管理机制并忽略分段机制的情况下权限位也可以保证安全。希望 Lab 3 能有更多的收获。