

操作系统实验报告



学 院：计算机学院 网络空间安全

专 业：计算机科学技术 信息安全

成 员：徐晖宇 付宇 顾知晨 姜奕兵 孙铭 肖阳

指导教师： 蒲凌君

目录

1	Lab1	2
1.1	Lab1_1 启动计算机部分	2
1.1.1	小组分工	2
1.1.2	问题一	2
1.1.3	问题 2	5
1.1.4	作业 1	12
1.1.5	作业 2	13
1.2	Lab1_2 内存管理部分	16
1.2.1	小组分工	16
1.2.2	物理页管理-作业 3	16
1.2.3	问题 3	25
1.2.4	作业 4	25
1.2.5	页表管理-作业 5	33
1.2.6	问题 4	34

1 Lab1

1.1 Lab1_1 启动计算机部分

1.1.1 小组分工

问题 2, 作业 1: 徐晖宇 (1713666)

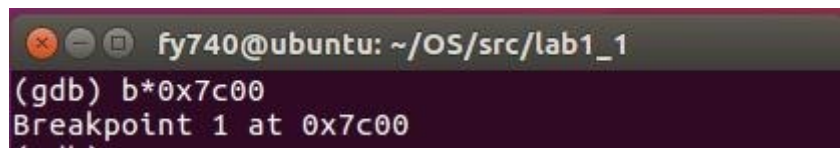
作业 2, 问题 1: 付宇 (1612120)

参与讨论与资料收集: 顾知晨(1711323) 姜奕兵(1710218) 孙铭(1711377) 肖阳(1610292)

1.1.2 问题一

1) 处理器从哪开始执行 32 位代码? 是什么导致了从 16 位到 32 位代码的切换?

利用 b*0x7c00 设置断点。如下图:



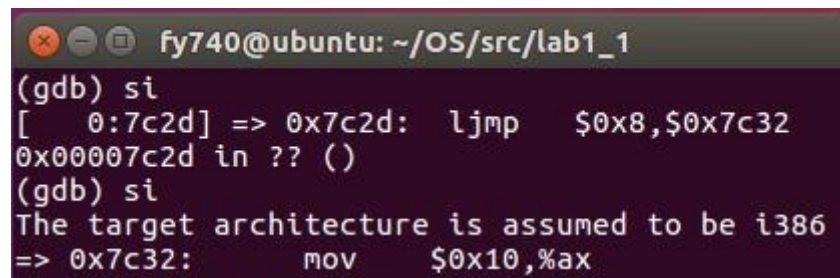
```
fy740@ubuntu: ~/OS/src/lab1_1
(gdb) b*0x7c00
Breakpoint 1 at 0x7c00
```

打开.../boot/boot.s 代码, 代码中第一个注释:

```
# Start the CPU: switch to 32-bit protected mode, jump into C.
```

从该段代码得知, boot.s 主要是将处理器从实模式转换到 32 位的保护模式。

接下来我们使用 si 命令单步执行, 至下图停止 si 命令:



```
fy740@ubuntu: ~/OS/src/lab1_1
(gdb) si
[ 0:7c2d] => 0x7c2d:  ljmp    $0x8,$0x7c32
0x00007c2d in ?? ()
(gdb) si
The target architecture is assumed to be i386
=> 0x7c32:      mov     $0x10,%ax
```

我们从上图可知, The target architecture is assumed to be i386 这段表明处理器在此开始执行 32 位代码 (注: i386 即 intel 80386, 其实 i386 通常被用来作为对 intel32 位微处理器的统称)。

找到 boot.s 中对应代码部分:

```

53      # Jump to next instruction, but in 32-bit code segment.
54      # Switches processor into 32-bit mode.
55      ljmp    $PROT_MODE_CSEG, $protcseg
56
57      .code32                                # Assemble for 32-bit mode
58  protcseg:
59      # Set up the protected-mode data segment registers
60      movw    $PROT_MODE_DSEG, %ax    # Our data segment selector

```

可见，处理器应该是从.code32 处开始执行 32 位代码。

为了探明 16 位转换至 32 位的原因，我们退回前面研究代码，在 boot.s 中得知如下信息：

```

44      # Switch from real to protected mode, using a bootstrap GDT
45      # and segment translation that makes virtual addresses
46      # identical to their physical addresses, so that the
47      # effective memory map does not change during the switch.
48      lgdt    gdt_desc
49      movl    %cr0, %eax
50      orl     $CR0_PE_ON, %eax
51      movl    %eax, %cr0

```

我们对上述内容非常陌生，查询资料后得知：GDT 全称 Global Descriptor Table，译为全局描述表，是在保护模式下一个重要的数据结构，它是在保护模式所必须的数据结构，也是唯一的。cr0 中包含 6 个预定义标志，第 0 位是保护允许位 PE (protected enable)，用于启动保护模式，若 PE 置为 1，则保护模式启动，0 则在实模式下运行。第 1 位是监控协处理器位 (monitor coprocessor)，它与第 3 位一起决定当 TS=1 时，操作码 WAIT 是否产生一个“协处理器不能使用”的出错信号。第 3 位是任务转换位 (task switch)，当一个任务转换完成之后，自动将它置为 1，随着 TS=1，就不能使用协处理器。第 2 位是模拟协处理器位 EM (emulate coprocessor)，若 EM=1，则不能使用协处理器。第 4 位是微处理器的扩展类型位 ET (processor extension type)，其内保存着处理器扩展类型的信息，若 ET=0，则表示系统使用的是 287 协处理器，若 ET=1，则表示系统使用的是 387 浮点协处理器。第 31 位是分页允许位 PG (paging enable)，它表示芯片上的分页部件是否允许工作。

在此我们只考虑 PE。如下，CR0_PE_ON 在 boot.s 开头处定义为值 1：

```

10      .set CR0_PE_ON, 0x1    # protected mode enable flag

```

至此我们得知，上述代码通过将 cr0 寄存器的 PE 位置为 1 来开启 32 位保护模式（注：如果 PE=0，PG=0，处理器处在实地址模式下；若 PE=1，PG=0，处理器工作在没有开启分页机制的保护模式下；若 PE=0，PG=1，此时由于不在保护模式下不能启用分页机制，因此处理器会产生保护异常；若 PE=1，PG=1，则处理器工作在开启了分页机制的保护模

式下)。

2) Boot loader 执行的最后一条指令是什么? Boot loader 加载内核后, 内核的第一条指令是什么?

从 boot.asm 中得到以下信息:

```
408      7d6b:  ff 15 18 00 01 00      call    *0x10018
```

从 main.c 中得到以下信息:

```
14      * * The kernel image must be in ELF format.
```

```
33      #define ELFHDR      ((struct Elf *) 0x10000) // scratch space
```

我们可以知道 boot loader 执行的最后一条指令就是:

7d6b: ff 15 18 00 01 00 call *0x10018

设置断点 b*7d6b, 依次执行 c、si 命令, 发现如下情况:

```
7=> 0x7d6b:      call    *0x10018
j
7Breakpoint 2, 0x00007d6b in ?? ()
j(gdb) si
7=> 0x10000c:     movw    $0x1234,0x472
0x0010000c in ?? ()
```

实际跳转地址至 0x10000c。执行 objdump -f kernel 验证, 结果同上。我们可以得出结论, 内核的加载在 bootmain 函数中完成, 加载完成后, 内核的第一条指令是 0x10000c: movw \$0x1234,0x472。

```
fy740@ubuntu:~/OS/src/lab1_1/obj/kern$ objdump -f kernel
kernel:      file format elf32-i386
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0010000c
```

3) 内核的第一条指令在哪?

同上题, 内核的第一条指令的地址为 0x10000c。

4) Bootloader 是如何知道为了从磁盘获取整个内核所必须读取的扇区数目? 它从哪找到这些信息的?

我们从 main.c 中, 发现下列加载扇区的代码:

```

50 // load each program segment (ignores ph flags)
51 ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);
52 eph = ph + ELFHDR->e_phnum;
53 for (; ph < eph; ph++)
54     // p_pa is the load address of this segment (as well
55     // as the physical address)
56     readseg(ph->p_pa, ph->p_memsz, ph->p_offset);

```

其中 ELFHDR 内的某些信息影响着循环次数。接着我们利用 `objdump -p kernel` 查看内核程序的短信息，如下：

```

fy740@ubuntu:~/05/src/lab1_1/obj/kern$ objdump -p kernel

kernel:      file format elf32-i386

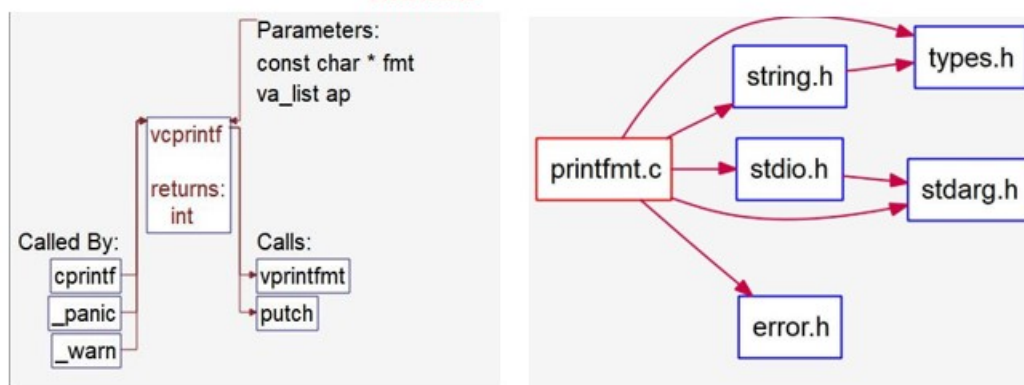
Program Header:
  LOAD off   0x00001000 vaddr 0xf0100000 paddr 0x00100000 align 2**12
    filesz 0x0000712f memsz 0x0000712f flags r-x
  LOAD off   0x00009000 vaddr 0xf0108000 paddr 0x00108000 align 2**12
    filesz 0x0000a300 memsz 0x0000a944 flags rw-
  STACK off  0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**4
    filesz 0x00000000 memsz 0x00000000 flags rwx

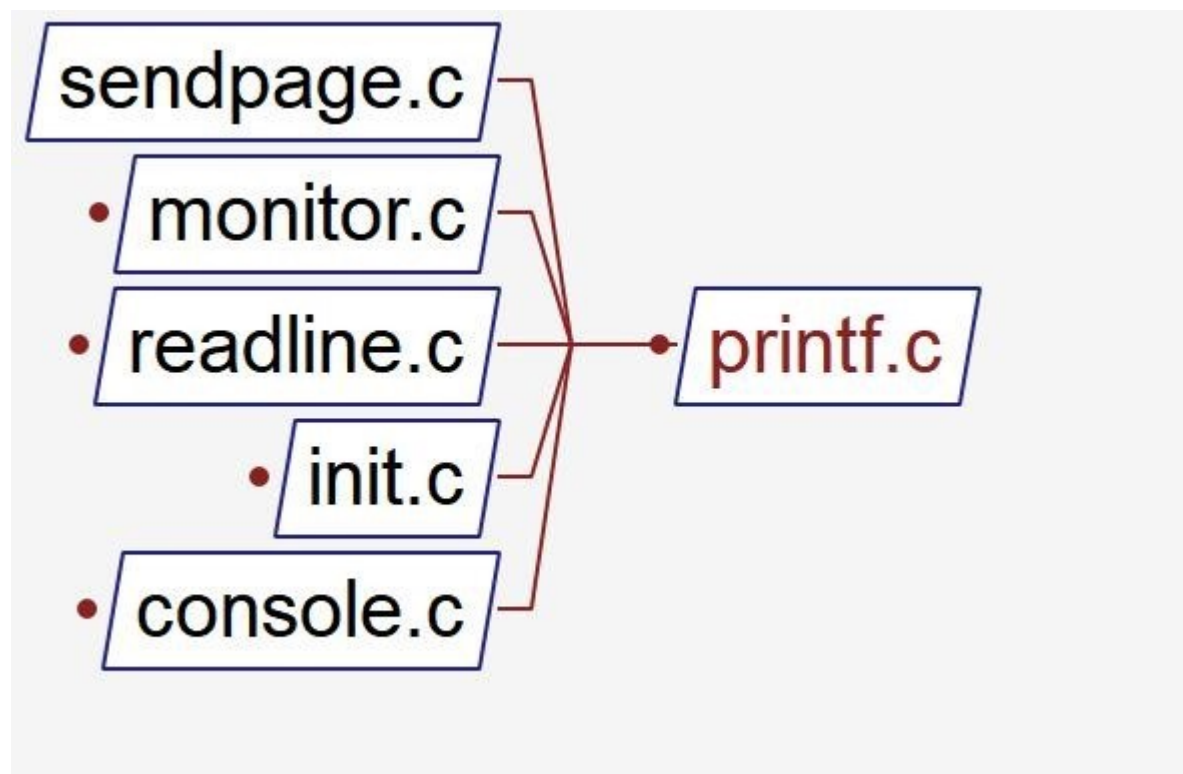
```

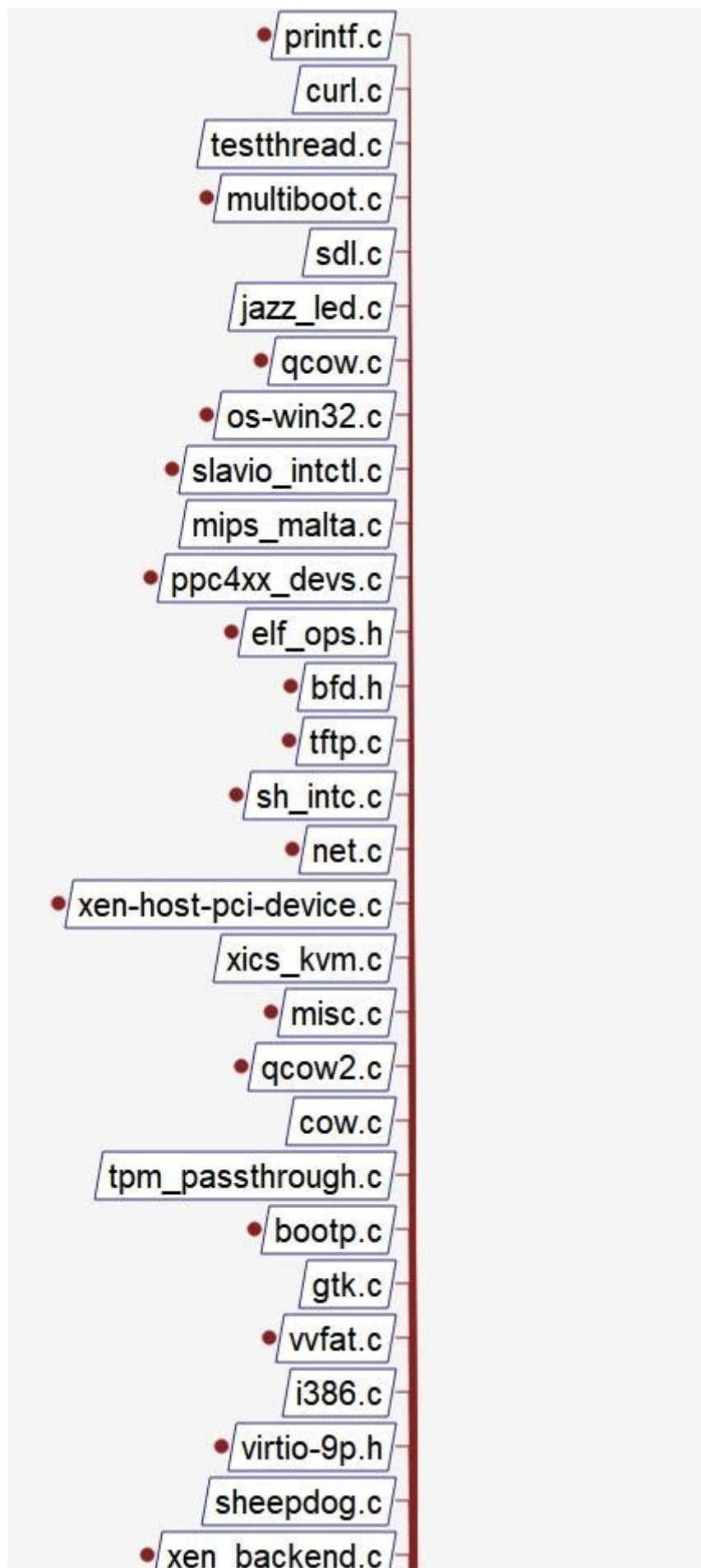
至此，我们得出结论，boot loader 通过 program header 中的段数目、每个段的偏移和字节数来知道需要加载的扇区数目。

1.1.3 问题 2

使用 `understand` 查看 `printf` 的父声明关系：







引用关系为 console.c 调用 printf.c, printf.c 调用 printfmt.c
具体进入 console.c

```
void
cputchar(int c)
{
    cons_putc(c);
}

int
getchar(void)
{
    int c;

    while ((c = cons_getc()) == 0)
        /* do nothing */;
    return c;
}

int
iscons(int fdnum)
{
    // used by readline
    return 1;
}
```

在上面的代码中我发现两点:

- 1.cputchar 代码的注释中说: 这个程序时最高层的 console 的 IO 控制程序。
- 2.cputchar 的实现其实是通过调用 cons_putc 完成的, cons_putc 程序的功能在它的备注中已经被叙述的很清楚了, 即输出一个字符到控制台 (计算机的屏幕)。所以我们就知道了 cputchar 的功能也是向屏幕上输出一个字符。

```

static void
cga_putc(int c)
{
    // if no attribute given, then use black on white
    if (!(c & ~0xFF))
        c |= 0x0700;

    switch (c & 0xff) {
    case '\b':
        if (crt_pos > 0) {
            crt_pos--;
            crt_buf[crt_pos] = (c & ~0xff) | ' ';
        }
        break;
    case '\n':
        crt_pos += CRT_COLS;
        /* fallthru */
    case '\r':
        crt_pos -= (crt_pos % CRT_COLS);
        break;
    case '\t':
        cons_putc(' ');
        cons_putc(' ');
        cons_putc(' ');
        cons_putc(' ');
        cons_putc(' ');
        break;
    default:
        crt_buf[crt_pos++] = c;      /* write the character */
        break;
    }

    // What is the purpose of this?
    if (crt_pos >= CRT_SIZE) {
        int i;

        memmove(crt_buf, crt_buf + CRT_COLS, (CRT_SIZE - CRT_COLS) * sizeof(uint16_t));
        for (i = CRT_SIZE - CRT_COLS; i < CRT_SIZE; i++)
            crt_buf[i] = 0x0700 | ' ';
        crt_pos -= CRT_COLS;
    }

    /* move that little blinky thing */
    outb(addr_6845, 14);
    outb(addr_6845 + 1, crt_pos >> 8);
    outb(addr_6845, 15);
    outb(addr_6845 + 1, crt_pos);
}

```

crt_pos 是当前光标位置，CRT_SIZE 是屏幕上总共的可以输出的字符数（其值等于

行数乘以每行的列数), 这段代码的意思是当屏幕输出满了以后, 将屏幕上的内容都向上移一行, 即将第一行移出屏幕, 同时将最后一行用空格填充, 最后将光标移动到屏幕最后一行的开始处。

结论:

printf.c 调用 console.c 提供的接口 cputchar, 将这个函数封装在 putchar, 并将这个封装好的函数作为参数传给 vprintfmt 函数, 用于向屏幕上输出一个字符。

kern/printf.c 提供用户实际需要调用的接口 cprintf lib/printfmt.c 提供了供 cprintf 函数调用的接口 vprintf, vprintf 的作用是对输出进行格式化, 把不同类型的输出 (kern/console.c 提供了供 vprintf 调用的回调函数 cputchar

printfmt.c:

```
// Stripped-down primitive printf-style formatting routines,  
// used in common by printf, sprintf, fprintf, etc.  
// This code is also used by both the kernel and user programs.
```

这一段注释解释了 printfmt 的功能是为 printf 等常见输出定制格式化

```
* The special format %e takes an integer error code  
* and prints a string describing the error.  
* The integer may be positive or negative,  
* so that -E_NO_MEM and E_NO_MEM are equivalent.
```

该段表示 %e 为特殊的控制符, 表示输出错误, 每次输出一个错误声明 printnum 函数递归地打印一个数字串:

```

static void
printrnum(void (*putch)(int, void*), void *putdat,
          unsigned long long num, unsigned base, int width, int padc)
{
    // first recursively print all preceding (more significant) digits
    if (num >= base) {
        printrnum(putch, putdat, num / base, base, width - 1, padc);
    } else {
        // print any needed pad characters before first digit
        while (--width > 0)
            putch(padc, putdat);

        // then print this (the least significant) digit
        putch("0123456789abcdef"[num % base], putdat);
    }
}

```

其中各个参数的含义：

void(*putch)(int,void*) 这个参数是一个函数指针，传入的 int 表示要打印的单个字符的值，void* 表示该字符存在的地址单元的值，传入该函数指针的目的在于不同的打印方式对应的 putch 不一样，看到这里的程序段中调用的 putch 是参数指定的。

void *putdat: 表示输入的字符要存放在的地址的指针

unsigned long long num: 指需要打印的数字 unsigned base: 表示进制

int width: 表示输入字符的宽度 int padc: 表示填充字符

这里为什么要递归呢，因为多位数（正如程序段中的 num 和 base 的比较）需要一位一位的输出，每一次输出都要对该数字求 base 的商。

getint 函数：

```

static long long
getint(va_list *ap, int lflag)
{
    if (lflag >= 2)
        return va_arg(*ap, long long);
    else if (lflag)
        return va_arg(*ap, long);
    else
        return va_arg(*ap, int);
}

```

该函数指定需要返回的 int 类型，lflag 变量则是专门在输出数字的时候起作用，在我

们这个实验中为了简单起见实际上是不支持输出浮点数的，于是 `vprintfmt` 函数只能够支持输出整形数，当 `lflag=0` 时，表示将参数当做 `int` 型的来输出，当 `lflag=1` 时，表示当做 `long` 型的来输出，而当 `lflag=2` 时表示当做 `longlong` 型的来输出。最后 `altflag` 变量表示当 `altflag=1` 时函数若输出乱码则用 `'?'` 代替。其中 `va_arg` 宏返回可变的参数，第一个形参表示指向参数的指针，第二个参数表示输出的数据类型。

`vprintfmt` 函数是该文件对外的接口，完成格式化字符的打印第一个 `while` 循环：

剩余的代码都是在处理 `'%'` 符号后面的格式化输出，比如是 `%d`，则按照十进制输出对应参数。另外还有一些其他的特殊字符比如 `%5d` 代表显示 5 位，其中的 5 要特殊处理。

1.1.4 作业 1

过程参照对应十六进制数的输出的 `%u` 函数段改写，对于八进制数，输出前有一个特殊的 `/0` 需要通过 `putch` 打印出来，然后第二步需要获取该数的返回类型，然后设置 `base=8`，然后进入 `number` 处理，`number` 段处理的方式是调用之前的 `printnum` 完成格式化输出。

```
case 'o':
    // Replace this with your code.
    num = getuint(&ap, lflag);
    if ((long long) num < 0) {
        putch('-', putdat);
        num = -(long long) num;
    }
    base = 8;
    goto number;
```

运行结果 (make qemu 对应八进制输出):


```
(process:5283): GLib-WARNING **: 02:48:08.923: ../../../../
om memory allocation vtable not supported
6828 decimal is 15254 octal!
entering test_backtrace 5
entering test_backtrace 4
entering test_backtrace 3
entering test_backtrace 2
entering test_backtrace 1
entering test_backtrace 0
leaving test_backtrace 0
leaving test_backtrace 1
leaving test_backtrace 2
leaving test_backtrace 3
leaving test_backtrace 4
leaving test_backtrace 5
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
```

1.1.5 作业 2

作业 2

You can do `mon_backtrace()` entirely in C. You'll also have to hook this new function into the kernel monitor's command list so that it can be invoked interactively by the user. The backtrace function should display a listing of function call frames in the following format:

Warning:

`read_ebp()` (较为底层的函数，返回值为当前的 `ebp` 寄存器的值)

display format

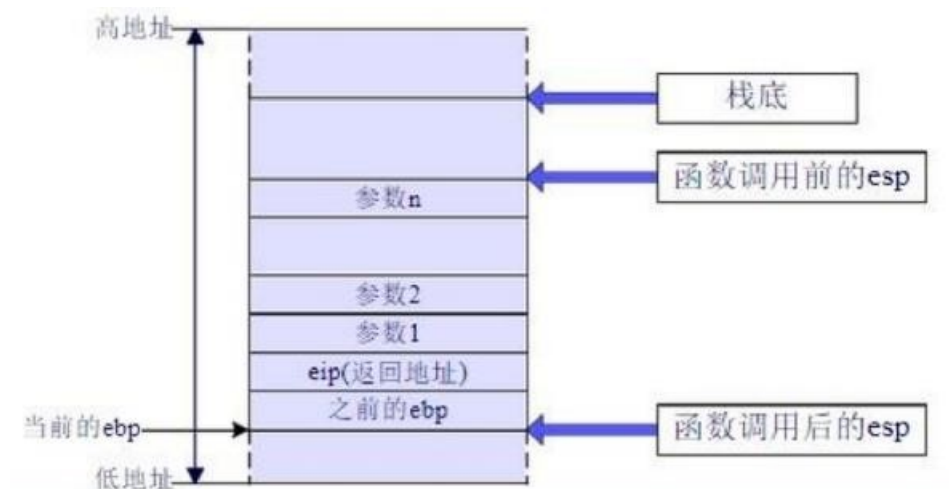
Stack backtrace:

```
ebp f0109e58 eip f0100a62 args 00000001 f0109e80 f0109e98 f0100ed2 00000031
ebp f0109ed8 eip f01000d6 args 00000000 00000000 f0100058 f0109f28 00000061
...
```

题目要求我们打印调用栈中 `ebp`、`eip` 以及参数的值。

从题中知道通过调用 `read_ebp()`，我们可以得到当前 `ebp` 寄存器的值。(注：`eip` 存储当前执行指令的下一条指令在内存中的偏移地址，`esp` 存储指向栈顶的指针，`ebp` 存储指向当前函数需要使用的参数的指针。)

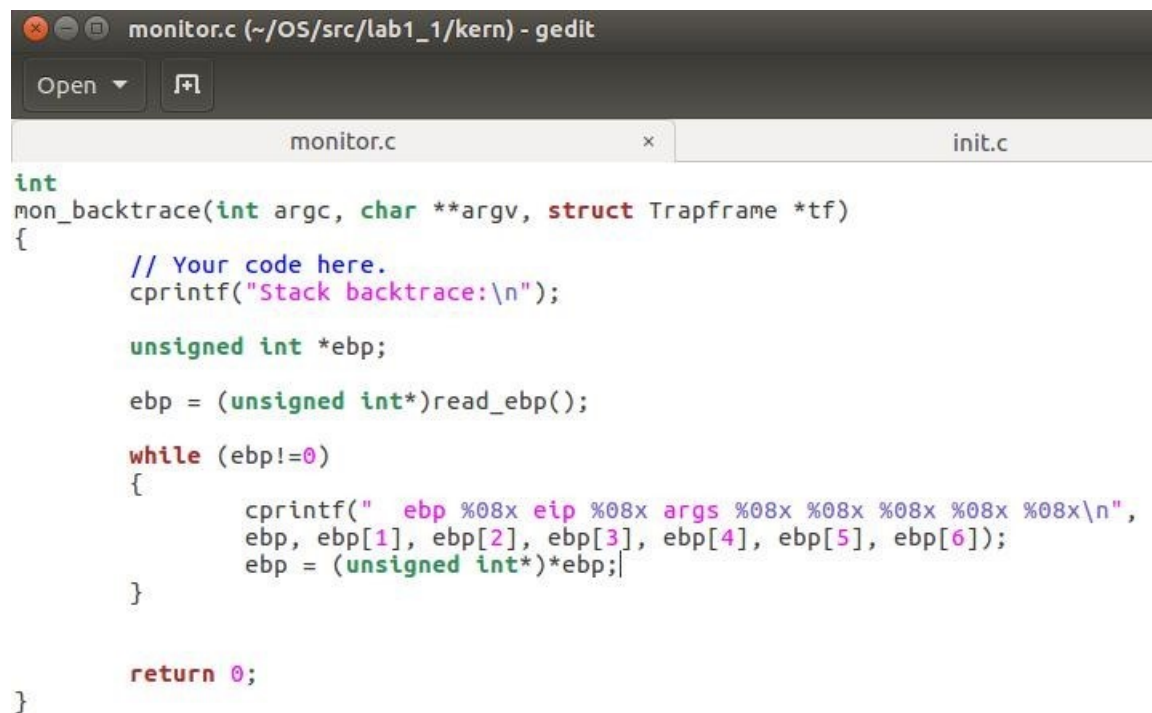
观察下图：



从文档描述中我们知道：一进入调用函数的时候，第一件事便是将 `ebp` 进栈，然后将当前的 `esp` 的值赋给 `ebp`，而此时 `ebp` 便指向了堆栈中存储 `ebp`、`eip` 和函数参数的地方，所以 `ebp` 通常都是指向当前函数所需要的参数，相当于每个函数都有一个自己的 `ebp`。根据前述信息，我们不难看出 `ebp` 的值实际上是指针值，亦可把它当作数组使用。结合上图，`ebp`，`ebp[1]`，`ebp[2]`，`ebp[3]`，`ebp[4]`，`ebp[5]`，`ebp[6]` 分别对应当前 `ebp`、`eip`、参数 1-5 的值。我们采用 `while` 循环来实现打印，但是存在一个问题，我们并不知道何时中止循环，为了找到跳出循环的临界值，我们查阅了 `kern/entry.S` 代码，有如下发现：

```
entry.S
# Turn on paging.
movl    %cr0, %eax
orl     $(CR0_PE|CR0_PG|CR0_WP), %e
movl    %eax, %cr0
```

我们发现，在内核初始化的时候，`ebp` 会被置为 0，也就是说在 `ebp=0` 的时候，循环就应该中止了。因此，我们的代码如下：



```

monitor.c (~OS/src/lab1_1/kern) - gedit
Open
monitor.c
init.c

int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
    // Your code here.
    cprintf("Stack backtrace:\n");

    unsigned int *ebp;

    ebp = (unsigned int*)read_ebp();

    while (ebp!=0)
    {
        cprintf("  ebp %08x eip %08x args %08x %08x %08x %08x %08x\n",
            ebp, ebp[1], ebp[2], ebp[3], ebp[4], ebp[5], ebp[6]);
        ebp = (unsigned int*)*ebp;
    }

    return 0;
}

```

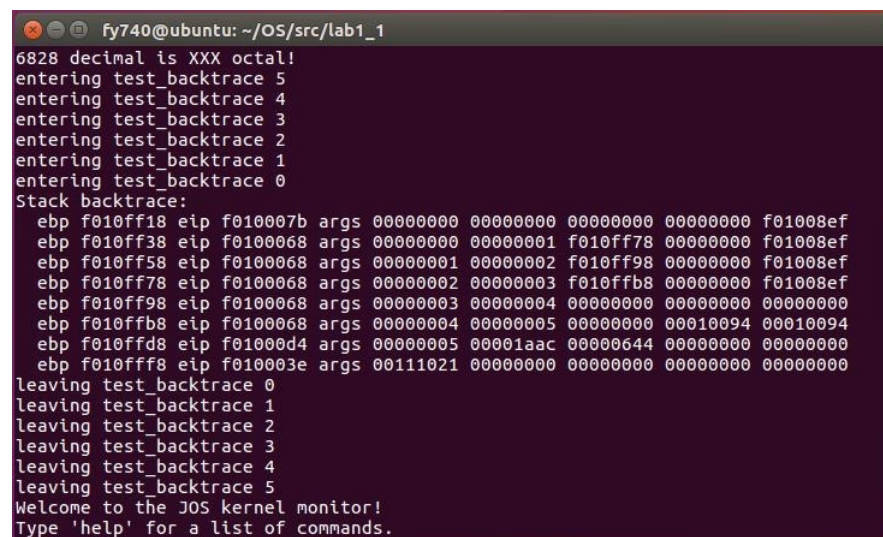
接着设置断点:

```

(gdb) b kern/monitor.c:4
Breakpoint 1 at 0xf0100678: file kern/monitor.c, line 4.

```

不断执行 c 命令直至结束, 得到以下结果:



```

fy740@ubuntu: ~/OS/src/lab1_1
6828 decimal is XXX octal!
entering test_backtrace 5
entering test_backtrace 4
entering test_backtrace 3
entering test_backtrace 2
entering test_backtrace 1
entering test_backtrace 0
Stack backtrace:
  ebp f010ff18 eip f010007b args 00000000 00000000 00000000 00000000 f01008ef
  ebp f010ff38 eip f0100068 args 00000000 00000001 f010ff78 00000000 f01008ef
  ebp f010ff58 eip f0100068 args 00000001 00000002 f010ff98 00000000 f01008ef
  ebp f010ff78 eip f0100068 args 00000002 00000003 f010ffb8 00000000 f01008ef
  ebp f010ff98 eip f0100068 args 00000003 00000004 00000000 00000000 00000000
  ebp f010ffb8 eip f0100068 args 00000004 00000005 00000000 00010094 00010094
  ebp f010ffd8 eip f01000d4 args 00000005 00001aac 00000644 00000000 00000000
  ebp f010fff8 eip f010003e args 00111021 00000000 00000000 00000000 00000000
leaving test_backtrace 0
leaving test_backtrace 1
leaving test_backtrace 2
leaving test_backtrace 3
leaving test_backtrace 4
leaving test_backtrace 5
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.

```

至此，作业二结束。

1.2 Lab1_2 内存管理部分

1.2.1 小组分工

作业 5，问题 4：孙铭（1711377）姜奕兵（1710218）

作业 3，问题 3：肖阳（1610292）

作业 4：顾知晨（1711323）

参与讨论与资料收集：徐晖宇（1713666）付宇（1612120）

1.2.2 物理页管理—作业 3

作业 3 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。

这几个函数都定义在 pmap.c 中，并且给出了函数的大体框架，包括函数的形式参数和返回值类型，我们需要做的是补充完整函数具体的内容。

打开源代码 pmap.c 文件：

首先观察到这个文件包含了许多别的头文件：

```
#include <inc/x86.h>
#include <inc/mmu.h>
#include <inc/error.h>
#include <inc/string.h>
#include <inc/assert.h>

#include <kern/pmap.h>
#include <kern/kclock.h>
```

图 1: 包含的头文件

对我们这次实验比较有帮助的是：

inc/mmu.h 包含 x86 内存管理单元 (MMU) 的一些定义好的常量以及页和段相关的数据结构。

kern/pmap.h 包含几个后续经常用到的函数，主要功能是实现页到物理地址的转换，物理地址到内核虚拟地址的转换，内核虚拟地址到物理地址的转换以及页到内核虚拟地址的转换，已经实现了我们需要用到的所有转换关系。

然后是一些定义的全局变量：

```
// These variables are set by i386_detect_memory()
size_t npages;           // Amount of physical memory (in pages)
static size_t npages_basemem; // Amount of base memory (in pages)

// These variables are set in mem_init()
pde_t *kern_pgdir;       // Kernel's initial page directory
struct PageInfo *pages;   // Physical page state array
static struct PageInfo *page_free_list; // Free list of physical pages
```

图 2: 全局变量

从注释可以知道，有两个 `size_t` 类型的全局变量 `npages`, `npages_basemem` 它们是由 `i386_detect_memory()` 这个函数来赋值的，其中 `npages` 是物理用来描述物理内存的大小的，单位是页，并且是一个非静态的全局变量，意味着在整个工程文件中都可以访问到它，我们可以在 `pmap.h` 中发现它被使用到了；而 `npages_basemem` 是用来描述 `basemem` 内存的大小的（即 `0xA0000` 之下的空间），单位是页，并且是一个静态的全局变量，只在本文件中可以访问。

接下来是一个 `pte_t*` 类型的指针，是指向内核初始化的页目录的，两个 `PageInfo*` 类型的指针，指向的是一个物理页的链表的首地址，`pages_free_list` 指向的是空闲物理页的链表的首地址。

然后我们可以进入 `mem_init()` 函数了，这个函数是用来初始化内核部分的地址空间，通过调用别的函数来进行初始化，并且建立起一个两级的页表结构，即页目录和页表的结构。

函数首先调用了 `i386_detect_memory()` 这个函数，用来确定机器还有多少内存可以使用，分别保存在 `npages` 和 `npages_basemem` 这两个全局变量中。

```

static void
i386_detect_memory(void)
{
    size_t npages_extmem;

    // Use CMOS calls to measure available base & extended memory.
    // (CMOS calls return results in kilobytes.)
    npages_basemem = (nvram_read(NVRAM_BASELO) * 1024) / PGSIZE;
    npages_extmem = (nvram_read(NVRAM_EXTLO) * 1024) / PGSIZE;

    // Calculate the number of physical pages available in both base
    // and extended memory.
    if (npages_extmem)
        npages = (EXTPHYSMEM / PGSIZE) + npages_extmem;
    else
        npages = npages_basemem;

    cprintf("Physical memory: %uK available, base = %uK, extended = %uK\n",
        npages * PGSIZE / 1024,
        npages_basemem * PGSIZE / 1024,
        npages_extmem * PGSIZE / 1024);
}

```

图 3: i386_detect_mem() 函数用来确定可分配内存大小

我们可以知道整个内存的大小 `npages` 就是在这里确定下来的。

接下来是一个提示信息，告诉我们测试这个函数还没有完成，当我们需要测试的时候记得注释掉这一行。

```
// Remove this line when you're ready to test this function.
```

```
//panic("mem_init: This function is not finished");
```

然后是创建初始的页目录。

```
kern_pgdir = (pde_t *) boot_alloc(PGSIZE);
```

```
memset(kern_pgdir, 0, PGSIZE);
```

这里调用了—个需要我们完成的函数 `boot_alloc()`，先看源代码中的注释信息，里面对 `boot_alloc()` 这个函数做了具体的说明和解释：

```


// 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.
```

从注释中告诉我们，这个函数只是 JOS 用来建立虚拟内存系统时才调用的，而另外一个函数 `page_alloc()` 才是真正的分配器。如果要分配的字节数大于 0，那么就需要分配足够多的连续物理页来容纳这个 `n` 个字节，这里暗示我们需要内存对齐，即以页的大小 4KB 来分配空间，不足向上取整，但是不用初始化内存，返回一个虚拟内存的地址即可。若 `n` 为 0，则返回下一个空闲的页的地址。

这个函数的代码如下：



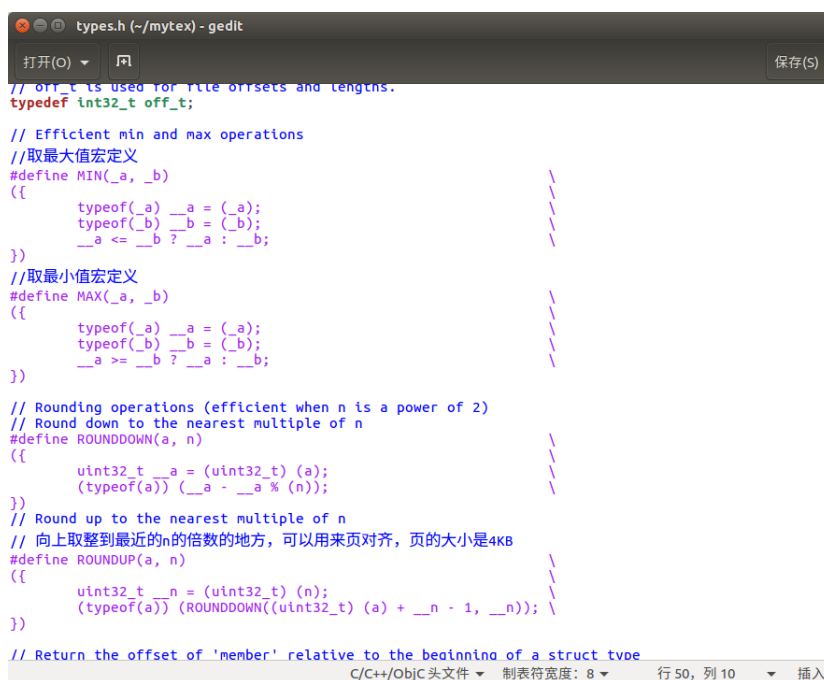
```
static void *
boot_alloc(uint32_t n)
{
    static char *nextfree; //局部静态变量默认初始化为0，每一次调用函数都会保留上次的值，指向的就是当前空闲页的首地址
    char *res;             //函数返回的指针，用来保存当前空闲的页的首地址，nextfree会更新

    // 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) {        //若nextfree没有显式初始化，即第一次调用boot_alloc函数时给nextfree赋值
        extern char end[]; //end是链接器自动生成的魔术符号，它指向内核的bss段的末尾：链接器未分配给任何内核代码或全局变量的第一个虚拟地址
        nextfree = ROUNDUP((char *) end, PGSIZE); //更新nextfree，使它指向对齐一页后的end，这里用到了ROUNDUP宏定义，它定义在types.h中，主要
        //用来4KB对齐
    }

    // 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.
    result = nextfree; //保存nextfree的值，用来返回
    nextfree = ROUNDUP((char *)result + n, PGSIZE); //更新nextfree，使其指向分配了足够多的内存页之后的首地址，注意4KB对齐
    return result; //函数返回值
}
```

图 4: `boot_alloc()` 函数及其实现

需要注意的是在这个函数中是用了 `ROUNDUP` 宏定义，它定义在 `types.h` 中：



```

types.h (~/mytex) - gedit
打开(O) 保存(S)

// off_t is used for file offsets and lengths.
typedef int32_t off_t;

// Efficient min and max operations
//取最大值宏定义
#define MIN(_a, _b) \
({ \
    typeof(_a) __a = (_a); \
    typeof(_b) __b = (_b); \
    __a <= __b ? __a : __b; \
})

//取最小值宏定义
#define MAX(_a, _b) \
({ \
    typeof(_a) __a = (_a); \
    typeof(_b) __b = (_b); \
    __a >= __b ? __a : __b; \
})

// 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
// 向上取整到最近的n的倍数的地方, 可以用来页对齐, 页的大小是4KB
#define ROUNDUP(a, n) \
({ \
    uint32_t __n = (uint32_t) (n); \
    (typeof(a)) (ROUNDDOWN((uint32_t) (a) + __n - 1, __n)); \
})

// Return the offset of 'member' relative to the beginning of a struct type

```

图 5: types.h 头文件

types.h 头文件中还定义了一些需要用到的宏定义, 如 `max()`, `min()` 等等。

至此, `boot__alloc()` 函数就大致分析完成了, 这个函数主要作用是分配一定大小的空间, 返回结果就是所分配空间的首地址, 这样 `kern_pgdir` 这个变量就指向了一个页的首地址, 那么我们的页目录就是一个页的大小, 首地址就是 `kern_pgdir` 所指。

然后使用了 `memset()` 函数来初始化这个页的内容为 0。

之后的一行代码, 是把刚分配的这个页插入到自身, 作为页目录的一个表项。

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

这行代码用到了 `PADDR` 这个宏定义, 实现的是把虚拟内核地址转换到对应的物理地址, 这个宏在 `pmap.h` 头文件中。

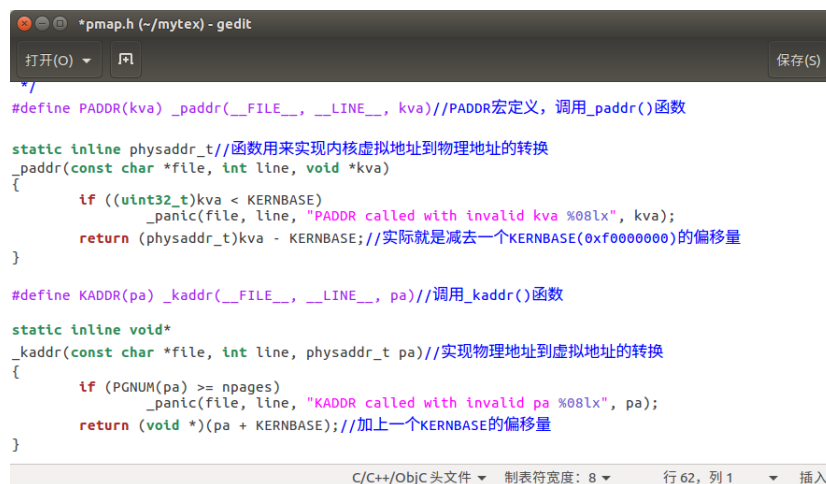


图 6: pmap.h 头文件中的宏定义, 实现虚拟地址和物理地址的各种转换

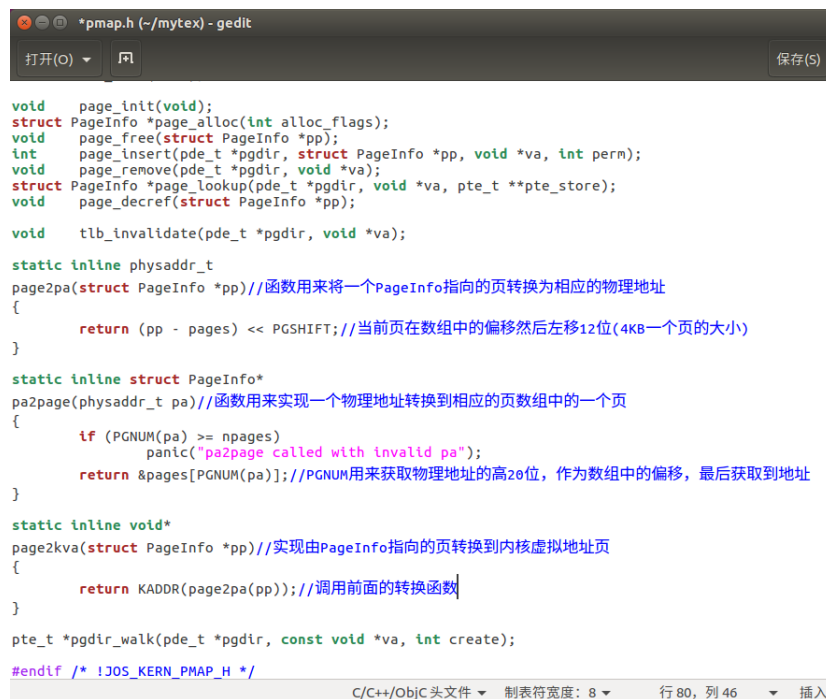


图 7: pmap.h 头文件

PTE_U|PTE_P 是页表项的标识符宏定义, 用来记录相关页的信息, 定义在 MMU.h 头文件中。

PTE_P 是判断对应物理页面是否存在, 若存在为 1, 不存在为 0。

PTE_U 是用来规定访问权限的，若为 1 表示 USER 即可，为 0 表示需要更高的权限。这样等号左边就形成了一个页目录的项，对应的是自身页的物理地址。把这个地址加入到页目录中即可。

然后是对全局变量 `pages` 进行初始化：

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

同样利用 `boot_alloc()` 函数来分配 `npages` 个页大小的空间，并把首地址返回到 `pages` 这个 `PageInfo*` 类型的指针，还有把这些空间清零，利用 `memset()` 函数。

现在我们已经分配来初始的内核的数据结构了，接下来是建立起一个空闲页的链表。调用 `page_init()` 函数。

先看注释的提示内容：

```
// 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.
```

告诉我们跟踪物理页，需要一个 `pages` 数组，每一个物理页都对应一个 `pages` 数组指针，这个指针指向一个 `PageInfo` 结构，结构里面保存了当前页被引用的次数，并且空闲页是由一个链表来维护的。

```
// 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.
```

初始化页结构和空闲内存的链表，在完成这些工作后，就不能在使用 `boot_alloc()` 函数了。

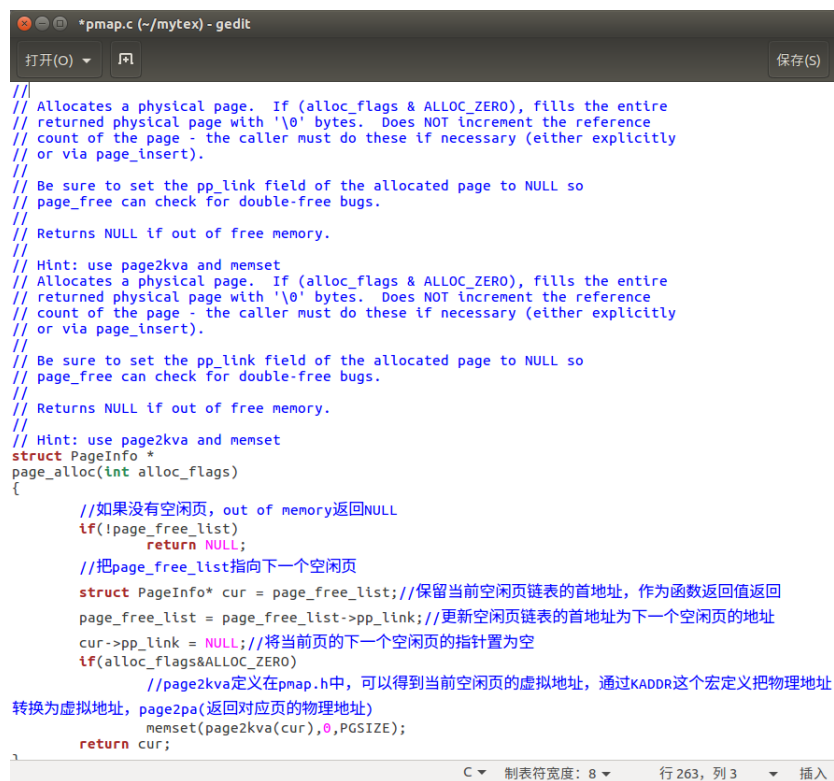
```

//
void
page_init(void)
{
    page_free_list = NULL;
    //第0页是被使用的
    pages[0].pp_ref = 1; //引用数非0,表示在使用中
    pages[0].pp_link = NULL; //下一个空闲页为空
    //余下的npages_basemem个页是空闲的
    size_t i;
    for(i=1; i<npages_basemem; i++){
        pages[i].pp_ref = 0; //引用为0表示空闲
        pages[i].pp_link = page_free_list; //将当前页的下一页链接到空闲页的首地址
        page_free_list = &pages[i]; //更新空闲的首地址为当前页
    }
    //IO hole是无法被分配的, EXTPHYSMEM宏定义了IO hole结束的位置
    for(i; i<(EXTPHYSMEM)/PGSIZE; i++){
        pages[0].pp_ref = 1;
        pages[0].pp_link = NULL;
    }
    //boot_alloc()是在内核之后(bss段)开始分配空间的, 所以boot_alloc(0)可以返回空闲页的首地址(虚拟地址), 即被占用的页的尾地址, 通过PADDR(pmap.h中)宏定义实现了虚拟地址到物理地址的转换
    for(i; i<PADDR(boot_alloc(0))/PGSIZE; i++){
        pages[0].pp_ref = 1;
        pages[0].pp_link = NULL;
    }
    //余下的所有页都是空闲的
    for(i; i<npages; i++){
        pages[i].pp_ref = 0;
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}
//

```

图 8: page_init() 函数的实现

然后是 page_alloc() 函数的实现如图所示。



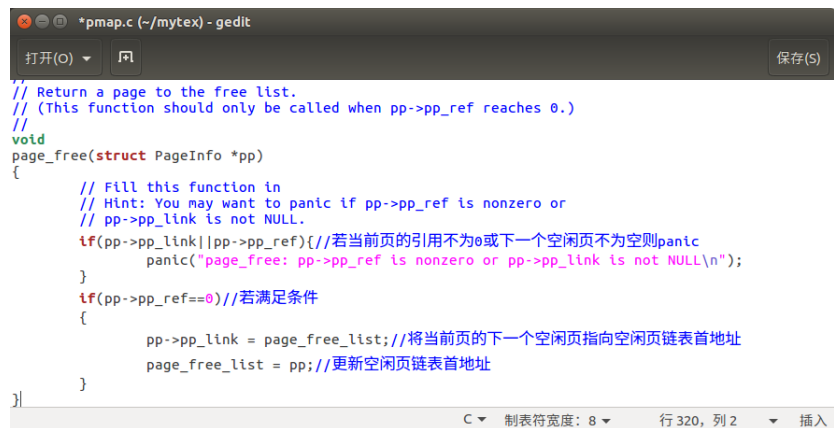
```

// 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
// 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)
{
    //如果没有空闲页, out of memory返回NULL
    if(!page_free_list)
        return NULL;
    //把page_free_list指向下一个空闲页
    struct PageInfo* cur = page_free_list; //保留当前空闲页链表的首地址, 作为函数返回值返回
    page_free_list = page_free_list->pp_link; //更新空闲页链表的首地址为下一个空闲页的地址
    cur->pp_link = NULL; //将当前页的下一个空闲页的指针置为空
    if(alloc_flags & ALLOC_ZERO)
        //page2kva定义在pmap.h中, 可以得到当前空闲页的虚拟地址, 通过KADDR这个宏定义把物理地址
        //转换为虚拟地址, page2pa(返回对应页的物理地址)
        memset(page2kva(cur), 0, PGSIZE);
    return cur;
}

```

图 9: page_alloc() 函数实现

最后是实现 page_free() 函数, 如图所示。



```

// 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.
    if(pp->pp_link || pp->pp_ref){ //若当前页的引用不为0或下一个空闲页不为空则panic
        panic("page_free: pp->pp_ref is nonzero or pp->pp_link is not NULL\n");
    }
    if(pp->pp_ref==0) //若满足条件
    {
        pp->pp_link = page_free_list; //将当前页的下一个空闲页指向空闲页链表首地址
        page_free_list = pp; //更新空闲页链表首地址
    }
}

```

图 10: page_free() 函数实现

1.2.3 问题 3

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;
```

从 `types.h` 头文件中可以知道, `uintptr_t` 和 `physaddr_t` 其实都是 `uint32_t` 的别名。

那么从最后一行的代码, 将 `value` 这个 `char*` 类型的指针强制转换为 `mystery_t` 类型, 可以知道 `mystery_t` 其实是一个指针类型, 在内核中所有的指针都是虚拟地址, 所以 `x` 是 `uintptr_t` 类型。

1.2.4 作业 4

作业 4 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。
```

题目要求我们实现以下的代码, 并用 `check_page()` 测试我的页表管理方法。

函数一:

`pgdir_walk()`, 返回指向为线性地址 `va` 分配的页表入口地址。如果相关页表没有在页目录中, 那么做如下工作:

如果 `create` 为 0, 那么返回 `NULL`。否则, `pgdir_walk()` 用 `page_alloc()` 分配一个新页, 将 `pp_ref` 设置为 1 实现代码:


```

pgdir_walk(pde_t *pgdir, const void *va, int create){
    // Fill this function in
    pde_t* pgd = pgdir+PDX(va);
    pte_t* pgt = NULL;
    if(!(*pgd&PTE_P))
    {
        if(!create)
            //若 create 为 0 则返回 null
            return NULL;
        else
        {
            struct PageInfo* newPg = page_alloc(1);
            if(!newPg)
                //p points to an entry in the pages table and set pp_ref to
                1
                return NULL;
            else
            {
                newPg->pp_ref += 1;
                //被访问次数+1
                *pgd = (page2pa(newPg) | PTE_P | PTE_W | PTE_U);
                //让此页目录指向那一页
            }
        }
    }
    //va 偏移量
    pgt = (pte_t*)KADDR(PTE_ADDR(*pgd)+PTX(va));
    //KADDR 宏定义，可以将物理地址转换成虚拟地址
    // Page table exists, return the VA of the page table entry
    return pgt;
}

```

函数二：

将虚拟地址对应的物理地址找到，找到其对应物理页和对于该页的偏移量。

```

boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa,
int perm)
{
    //将虚拟地址映射成物理地址
    // Fill this function in
    int pgnum = size/PGSIZE;
    //size 需要 PGsize 的整数倍 (若一倍的话页表失去了实际意义), va 与 pa 同步移动
    for(int i=0; i<pgnum; i++)
    {
        pte_t* pgt = pgdir_walk(pgdir, (void*)va, 1);
        //正确地寻找虚拟地址对应的第几页偏移量多少
        if(pgt)
        {
            *pgt = pa|PTE_P|perm;
            //不影响 pp—ref
            //根据关键位看权限 perm|PTE_P
            pa += PGSIZE;
            va += PGSIZE;
            //同时偏移对齐
        }
    }
}

```

函数三：

查找页表中虚拟地址有无被存储，若该地址已经被使用则现移除该页表。

```

int
page_insert(pde_t *pgdir, struct PageInfo *pp, void *va, int perm)
{
    // Fill this function in
    physaddr_t phyadd = page2pa(pp);
    //页表地址提取出来
    pte_t* pgt = pgdir_walk(pgdir, va, 1);
    if(!pgt)
        //未找到目录中的虚拟地址
        return -E_NO_MEM;
    else{
        pp->pp_ref += 1;
        if((*pgt)&PTE_P)
        {
            //如果原来就有，那么将其删除
            page_remove(pgdir, va);
        }
        *pgt = phyadd | perm | PTE_P;
        pgdir[PDX(va)] |= perm;
        //给予目录访问虚拟地址的权限
    }
    return 0;
}

```

函数四：

查找该页的信息，若其有效位有效（为 1）且目录偏移量不超过页的范围。则返回一个指向该页表的结构体指针。

```

struct PageInfo *
page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
{
    // Fill this function in
    pte_t* pgt = pgdir_walk(pgdir, va, 0);
    if(pgt==NULL)
        return NULL;
    //返回且有效且在目录内不超出页的范围
    if(!((*pgt)&PTE_P))
        return NULL;
    struct PageInfo* res = pa2page(PTE_ADDR(*pgt));
    //转换成结构体指针（取出页的物理地址）
    //创建一个指向结构体的指针
    if(pte_store)
        //标记位
        *pte_store = pgt;
    return res;
}

```

函数五：
移除页表。

```

void
page_remove(pde_t *pgdir, void *va)
{
    // Fill this function in
    pte_t* pte = NULL;
    struct PageInfo* pgin = page_lookup(pgdir, va, &pte);
    if(pgin)
    {
        page_decref(pgin);
        *pte = 0;
        tlb_invalidate(pgdir,va);
    }
}

```

函数六：check——page
static void

```

check_page(void)
{
    struct PageInfo *pp, *pp0, *pp1, *pp2;
    struct PageInfo *fl;
    pte_t *ptep, *ptep1;
    void *va;
    int i;
    extern pde_t entry_pgdir[];
    // should be able to allocate three pages
    pp0 = pp1 = pp2 = 0;
    assert((pp0 = page_alloc(0)));
    assert((pp1 = page_alloc(0)));
    assert((pp2 = page_alloc(0)));
    assert(pp0);
    assert(pp1 && pp1 != pp0);
    assert(pp2 && pp2 != pp1 && pp2 != pp0);
    // temporarily steal the rest of the free pages
    fl = page_free_list;
    page_free_list = 0;
    // should be no free memory
    assert(!page_alloc(0));
    // there is no page allocated at address 0
    assert(page_lookup(kern_pgdir, (void *) 0x0, &ptep) == NULL);
    // there is no free memory, so we can't allocate a page table
    assert(page_insert(kern_pgdir, pp1, 0x0, PTE_W) < 0);
    // free pp0 and try again: pp0 should be used for page table
    page_free(pp0);
    assert(page_insert(kern_pgdir, pp1, 0x0, PTE_W) == 0);
    assert(PTE_ADDR(kern_pgdir[0]) == page2pa(pp0));
    assert(check_va2pa(kern_pgdir, 0x0) == page2pa(pp1));
    assert(pp1->pp_ref == 1);
    assert(pp0->pp_ref == 1);
    // should be able to map pp2 at PGSIZE because pp0 is already allocated for page
table
    assert(page_insert(kern_pgdir, pp2, (void*) PGSIZE, PTE_W) == 0);
    assert(check_va2pa(kern_pgdir, PGSIZE) == page2pa(pp2));

```

```

assert(pp2->pp_ref == 1);
// should be no free memory
assert(!page_alloc(0));
// should be able to map pp2 at PGSIZE because it's already there
assert(page_insert(kern_pgdir, pp2, (void*) PGSIZE, PTE_W) == 0);
assert(check_va2pa(kern_pgdir, PGSIZE) == page2pa(pp2));
assert(pp2->pp_ref == 1);
// pp2 should NOT be on the free list
// could happen in ref counts are handled sloppily in page_insert
assert(!page_alloc(0));
// check that pgdir_walk returns a pointer to the pte
ptep = (pte_t *) KADDR(PTE_ADDR(kern_pgdir[PDX(PGSIZE)]));
assert(pgdir_walk(kern_pgdir, (void*)PGSIZE, 0) == ptep+PTX(PGSIZE));
// should be able to change permissions too.
assert(page_insert(kern_pgdir, pp2, (void*) PGSIZE, PTE_W|PTE_U) == 0);
assert(check_va2pa(kern_pgdir, PGSIZE) == page2pa(pp2));
assert(pp2->pp_ref == 1);
assert(*pgdir_walk(kern_pgdir, (void*) PGSIZE, 0) & PTE_U);
assert(kern_pgdir[0] & PTE_U);
// should be able to remap with fewer permissions
assert(page_insert(kern_pgdir, pp2, (void*) PGSIZE, PTE_W) == 0);
assert(*pgdir_walk(kern_pgdir, (void*) PGSIZE, 0) & PTE_W);
assert(!(*pgdir_walk(kern_pgdir, (void*) PGSIZE, 0) & PTE_U));
// should not be able to map at PTSIZE because need free page for page table
assert(page_insert(kern_pgdir, pp0, (void*) PTSIZE, PTE_W) < 0);
// insert pp1 at PGSIZE (replacing pp2)
assert(page_insert(kern_pgdir, pp1, (void*) PGSIZE, PTE_W) == 0);
assert(!(*pgdir_walk(kern_pgdir, (void*) PGSIZE, 0) & PTE_U));
// should have pp1 at both 0 and PGSIZE, pp2 nowhere, ...
assert(check_va2pa(kern_pgdir, 0) == page2pa(pp1));
assert(check_va2pa(kern_pgdir, PGSIZE) == page2pa(pp1));
// ... and ref counts should reflect this
assert(pp1->pp_ref == 2);
assert(pp2->pp_ref == 0);
// pp2 should be returned by page_alloc
assert((pp = page_alloc(0)) && pp == pp2);

```



```

// unmapping pp1 at 0 should keep pp1 at PGSIZE
page_remove(kern_pgdir, 0x0);
assert(check_va2pa(kern_pgdir, 0x0) == 0);
assert(check_va2pa(kern_pgdir, PGSIZE) == page2pa(pp1));
assert(pp1->pp_ref == 1);
assert(pp2->pp_ref == 0);
// test re-inserting pp1 at PGSIZE
assert(page_insert(kern_pgdir, pp1, (void*) PGSIZE, 0) == 0);
assert(pp1->pp_ref);
assert(pp1->pp_link == NULL);
// unmapping pp1 at PGSIZE should free it
page_remove(kern_pgdir, (void*) PGSIZE);
assert(check_va2pa(kern_pgdir, 0x0) == 0);
assert(check_va2pa(kern_pgdir, PGSIZE) == 0);
assert(pp1->pp_ref == 0);
assert(pp2->pp_ref == 0);
// so it should be returned by page_alloc
assert((pp = page_alloc(0)) && pp == pp1);
// should be no free memory
assert(!page_alloc(0));
// forcibly take pp0 back
assert(PTE_ADDR(kern_pgdir[0]) == page2pa(pp0));
kern_pgdir[0] = 0;
assert(pp0->pp_ref == 1);
pp0->pp_ref = 0;
// check pointer arithmetic in pgdir_walk
page_free(pp0);
va = (void*)(PGSIZE * NPDETRIES + PGSIZE);
ptep = pgdir_walk(kern_pgdir, va, 1);
ptep1 = (pte_t *) KADDR(PTE_ADDR(kern_pgdir[PDX(va)]));
assert(ptep == ptep1 + PTX(va));
kern_pgdir[PDX(va)] = 0;
pp0->pp_ref = 0;
// check that new page tables get cleared
memset(page2kva(pp0), 0xFF, PGSIZE);
page_free(pp0);

```

```

pgdir_walk(kern_pgdir, 0x0, 1);
ptep = (pte_t *) page2kva(pp0);
for(i=0; i<NPENTRIES; i++)
assert((ptep[i] & PTE_P) == 0);
kern_pgdir[0] = 0;
pp0->pp_ref = 0;
// give free list back
page_free_list = fl;
// free the pages we took
page_free(pp0);
page_free(pp1);
page_free(pp2);
cprintf("check_page() succeeded!");
}

```

1.2.5 页表管理-作业 5

其他的函数均已实现，现在需要调用 `boot_map_region()` 函数实现从虚拟地址到物理地址的映射关系。

1. // Map 'pages' read-only by the user at linear address UPAGES
2. // Permissions:
3. // - the new image at UPAGES - kernel R, user R
4. // (ie. perm = PTE_U | PTE_P)

根据注释，要将以 'UPAGES' (0xef000000) 开始的虚拟地址映射到 'pages' 位置的物理地址上。而要映射的内存大小是在调用 `i386_detect_memory()` 函数时已经计算过的 `npages` 变量，也就是可用物理内存的总数。所以调用函数的参数清空如下：

1. `boot_map_region(`
2. `kern_pgdir`, //页目录
3. `UPAGES`, // 虚拟地址
4. `ROUNDUP((sizeof(struct PageInfo)*npages), PGSIZE)`, //映射地址长度
5. `PADDR(pages)`, // 物理地址
6. `PTE_U`); //权限

对于每一块映射过的地址权限为应该标记当前页面有效，即应是 `PTE_U` 和 `PTE_P` 按位或处理，但在函数 `boot_map_region()` 内部已经将最后一个参数进行该操作，所以传入的参数是 `PTE_U` 用户程序可访问的权限，不再与 `PTE_P` 按位或。

1. // Use the physical memory that 'bootstack' refers to as the kernel
2. // stack. The kernel stack grows down from virtual address `KSTACKTOP`.

3. // We consider the entire range from [KSTACKTOP-PTSIZE, KSTACKTOP)
4. // to be the kernel stack, but break this into two pieces:
5. // * [KSTACKTOP-KSTKSIZE, KSTACKTOP) – backed by physical memory
6. // * [KSTACKTOP-PTSIZE, KSTACKTOP-KSTKSIZE) – not backed; so if
7. // the kernel overflows its stack, it will fault rather than
8. // overwrite memory. Known as a "guard page".
9. // Permissions: kernel RW, user NONE

同样根据注释提示，我们要将 [KSTACKTOP-KSTKSIZE, KSTACKTOP) 这一块虚拟地址映射到物理地址 bootstack 处，而 [KSTACKTOP-PTSIZE, KSTACKTOP-KSTKSIZE) 这一块作为保护页不做映射。

1. boot_map_region(kern_pgdir, // 页目录
2. (KSTACKTOP-KSTKSIZE), // 映射的虚拟地址位置起始 = 终止-长度
3. KSTKSIZE, // 长度
4. PADDR(bootstack), // bootstack 对应物理地址
5. PTE_W); // 内核可写

由于这一块权限是内核可写而用户没有权限访问，所以权限位是 PTE_W。

1. // Map all of physical memory at KERNBASE.
2. // Ie. the VA range [KERNBASE, 232) should map to
3. // the PA range [0, 232 - KERNBASE)
4. // We might not have 232 - KERNBASE bytes of physical memory, but
5. // we just set up the mapping anyway.
6. // Permissions: kernel RW, user NONE

根据提示，需要将 KERNBASE(0xf0000000) 以上的虚拟地址全部映射到物理地址 0 的位置。

1. boot_map_region(kern_pgdir, // 页目录
 2. KERNBASE, // 0xf0000000
 3. ROUNDUP((0xFFFFFFFF-KERNBASE), PGSIZE), // 长度
 4. 0, // 物理地址
 5. PTE_W); // 内核可写
- 这里用 16 进制 0xFFFFFFFF 表示 232。

1.2.6 问题 4

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.

题目要求表示出该实验中所有映射到的地址，根据代码文件 memlayout.h 中的结构显示：

Entry	Base Virtual Address	Points to (logically)
1024	0xffc00000	Page table for top 4MB of phys memory
.....		Kern space
.....		Kern space
960	0xf0000000	All physical memory mapped at this address
.....
959	0xefc00000	Kernel stack
958	0xef800000	The limit of user programing, Memory-mapped I/O
957	0xef400000	Read-only virtual page table for user(UVPT)
956	0xef000000	Struct page in memory(struct PageInfo)
955	0xeec00000	Top of one-page user exception stack, top of user-accessible. Read-only copies of the global env structures.
.....
2	0x00800000	The beginning of program space
1	0x00400000	Phys memory for user temporary data
0	0x00000000	Contains user-level STABS data structures

2) 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?

在做虚拟内存到物理内存的映射时，会有一个按位操作的权限值，其中各个变量对应权限如下：

权限变量	值	含义
PTE_P	0x001	标记了页面是否有效。只有对应位为 1 才可做映射访问
PTE_W	0x002	标记了页面是否可写。只有对应位为 1 才可对页面进行写操作
PTE_U	0x004	标记用户程序可使用访问该页面。若为 0 则只有内核态可访问

在用户程序要访问或写入内存时，需要对这个权限值进行判断，只有满足相应的权限位为 1 才可进行对应的权限操作。这种机制保护了系统的安全，避免了用户程序对内核地址中的内容进行修改，有效的将操作系统和用户程序隔离开来。

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

页结构体变量处于 (UPAGES, UVPT) 之间, 共占据内存:

$$page_info_all_size = 4MB \quad (1)$$

每一个结构体大小:

$$page_info_each_size = sizeof(structPageInfo) = 8Byte \quad (2)$$

所以页的结构体个数:

$$page_info_count = \frac{page_info_all_size}{page_info_each_size} = \frac{4MB}{8Byte} = 512K \quad (3)$$

而每个页的结构体变量对应一个物理内存块 (每块 4KB), 若都映射不同的物理内存, 则对应物理内存大小为:

$$max_mem_size = 4KB * page_info_count = 2GB \quad (4)$$

即最大物理内存为 2GB。

4) 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?

最大可能映射的物理内存为:

$$max_mem_size = 2GB \quad (5)$$

为了管理内存, 耗费了页目录, 页表以及页的结构体三部分的内存。这三部分分别记为:

$$page_dir_size, page_table_size, page_info_size \quad (6)$$

最大物理内存每一块均对应一个 page 结构体, 所以结构体数量等于物理内存块数, 即:

$$mem_block_count = \frac{max_mem_size}{mem_each_size} = \frac{2GB}{4KB} = 512K \quad (7)$$

每个页结构体占 8Byte, 所以页结构体共占用内存:

$$\begin{aligned} page_info_size &= mem_block_count * page_info_each_size \\ &= 512K * 8Byte \\ &= 4MB \end{aligned} \quad (8)$$

同样, 页表每一行要对应一个页结构体, 且页表每一行占 4Byte, 所以页表占用内存:

$$\begin{aligned} page_table_size &= mem_block_count * page_table_each_size \\ &= 512K * 4Byte \\ &= 2MB \end{aligned} \quad (9)$$

页表需要放在内存块中，每个内存块 4KB，所以页表要占用内存块个数：

$$page_table_count = \frac{page_table_size}{4KB} = \frac{2MB}{4KB} = 2MB/4KB = 512 \quad (10)$$

由于一个内存块 (4KB) 的页目录可以映射的页表数量：

$$\frac{4KB}{4Byte} = 1024 > 512 = page_table_count \quad (11)$$

所以页目录需要占用 4KB 内存。

综上，为管理内存空间共需要的内存为：

$$\begin{aligned} total_size &= page_dir_size + page_table_size + page_info_size \\ &= 4KB + 2MB + 4MB \\ &= 6148KB \end{aligned} \quad (12)$$

为了减少这个内存消耗，可以将每个内存块的大小由 4KB 适当增大，这样便减少了需要的页目录、页表、页结构的存储。

假设内存块大小由 4KB 增大到 8MB，则需要管理内存空间的消耗是：

$$\begin{aligned} new_size &= new_pdSize + new_ptSize + new_piSize \\ &= 8KB + 1MB + 2MB \\ &= 3080KB \end{aligned} \quad (13)$$

显然，耗费的内存减少了大约一半。

但是并不是内存块设置的越大越好。如果每个内存块大小设置的太大，反而会使这个耗费变得更大。例如将内存块大小由 4KB 增大至 4MB，此时耗费大小是：

$$\begin{aligned} new_size &= new_pdSize + new_ptSize + new_piSize \\ &= 4MB + 4MB + 4MB \\ &= 12MB \end{aligned} \quad (14)$$

所以要做的是将每个内存块的大小由 4KB 适当增大，而不是随意的增大。