

**操作系统原理课程设计报告**

|  |  |
| --- | --- |
| 姓 名： | 陈学海 |
| 学 院： | 计算机科学与技术学院 |
| 专 业： | 计算机科学与技术 |
| 班 级： | ACM1601 |
| 学 号： | U201614789 |
| 指导教师： | 张杰 |

|  |  |
| --- | --- |
| 分数 |  |
| 教师签名 |  |

2019年 2 月 17 日

目录

[Lab1 Booting a PC 4](#_Toc59927398)

[1 实验目的 4](#_Toc234787900)

[2 实验内容 4](#_Toc1404417749)

[2.1 Part 1: PC Bootstrap 4](#_Toc505454250)

[2.1.1 Getting Started with x86 assembly 4](#_Toc1914550950)

[2.1.2 Simulating the x86 5](#_Toc156392663)

[2.1.3 The PC's Physical Address Space 5](#_Toc160504411)

[2.1.4 The ROM BIOS 7](#_Toc1313321860)

[2.2 Part 2: The Boot Loader 9](#_Toc910290533)

[2.2.1 Loading the Kernel 12](#_Toc768342642)

[2.3 Part 3: The Kernel 17](#_Toc1066751662)

[图1-9 分页前后0xf0100000处的内存值 18](#_Toc337801695)

[图1-10 函数帧栈布局图 21](#_Toc1377869855)

[图1-11 ELF段信息 23](#_Toc1342978124)

[图1-12 .stab段信息 23](#_Toc1545104331)

[图1-13 init.s中的调试信息 24](#_Toc152271285)

[图1-14 .stabstr段LVM处内存值 25](#_Toc439085555)

[图1-15 kernel的第一条指令 26](#_Toc124975568)

[1) int x = 1, y = 3, z = 4; 27](#_Toc1243388647)

[2) cprintf("x %d, y %x, z %d\n", x, y, z); 27](#_Toc604316254)

[1) unsigned int i = 0x00646c72; 28](#_Toc595519365)

[2) cprintf("H%x Wo%s", 57616, &i); 28](#_Toc1505700325)

[1) cprintf("x=%d y=%d", 3); 28](#_Toc1630244446)

[Lab2 Memory Management 30](#_Toc826337801)

[1 实验目的 30](#_Toc245366614)

[2 实验内容 30](#_Toc1455957876)

[2.1 Part 1: Physical Page Management 30](#_Toc1921269279)

[2.2 Part 2: Virtual Memory 30](#_Toc1630289217)

[2.3 Part 3: Kernel Address Space 30](#_Toc2115576871)

[3 实验问题 30](#_Toc886916743)

[Lab3 User Environments 31](#_Toc542410455)

[1 实验目的 31](#_Toc708442933)

[2 实验内容 31](#_Toc1121704643)

[3 实验问题 31](#_Toc1946828205)

[Lab4 Preemptive Multitasking 32](#_Toc1213897183)

[1 实验目的 32](#_Toc888771945)

[2 实验内容 32](#_Toc2103220868)

[3 实验问题 32](#_Toc1374401594)

### Lab1 Booting a PC

#### 1 实验目的

1. 熟悉x86汇编语言，计算机引导过程，并熟悉实验所需环境QEMU模拟器以及使用QEMU和GDB进行调试的过程；
2. 熟悉本实验环境中的引导程序，熟悉计算机操作系统引导过程；
3. 初步熟悉本实验环境所使用的操作系统内核JOS。

#### 2 实验内容

##### 2.1 Part 1: PC Bootstrap

###### 2.1.1 Getting Started with x86 assembly

|  |
| --- |
| Exercise 1. Familiarize yourself with the assembly language materials available on [the 6.828 reference page](https://pdos.csail.mit.edu/6.828/2018/reference.html). You don't have to read them now, but you'll almost certainly want to refer to some of this material when reading and writing x86 assembly.  We do recommend reading the section "The Syntax" in [Brennan's Guide to Inline Assembly](http://www.delorie.com/djgpp/doc/brennan/brennan_att_inline_djgpp.html). It gives a good (and quite brief) description of the AT&T assembly syntax we'll be using with the GNU assembler in JOS. |

Exercise 1的任务是要熟悉X86汇编，能看懂并使用AT&T汇编语法。AT&T语法与Intel语法大体相同，如下是两者的主要区别：

1. 源、目的操作数顺序相反。AT&T 先写源，再写目的操作数，Intel反之。
2. 指令命名不同。AT&T由指令后缀b,w,l来指示操作数大小，而Intel由操作数前缀XXX PTR来确定。
3. 取址方式不同。AT&T 语法为offset(base, index, width)，Intel则为[INDEX \* WIDTH + BASE + OFFSET]的格式。
4. 寄存器、立即数表示不同，AT&T在寄存器前加%，立即数前加$，Intel无这些前缀。

由于之前已经学过Intel语法，所以对于AT&T语法也能较快上手，注意好上述区别后可以看懂本实验中的汇编文件并书写简单的汇编指令了。

###### 2.1.2 Simulating the x86

本实验使用QEMU虚拟机来模拟X86环境，在实验文件的主目录下使用make命令来编译生成kernel镜像文件，之后键入make qemu-nox 即可运行JOS。退出时可使用组合键ctrl + A X。需额外提及的一点是如何在QEMU中查看各类寄存器。由于我们在GDB中只能查看一些通用寄存器，对于某些专用的寄存器如EIP,GDTR需用QEMU的info registers命令进行打印。具体而言：

1. 使用组合键ctrl + A C进入qemu模式。
2. 在qemu模式下使用inforegisters输出寄存器信息。

###### 2.1.3 The PC's Physical Address Space

PC的启动过程与其物理空间的布局密切相关，PC只有从磁盘加载kernel并执行后才能进入保护模式，而在这之前的BIOS和boot loader都是在实模式下和物理地址打交道。因而有必要详细了解PC的物理地址空间布局。此外kernel必须要设置好虚拟地址到物理地址的映射关系，了解物理地址空间布局也会对我们后续工作带来帮助。详细信息可参见下图：

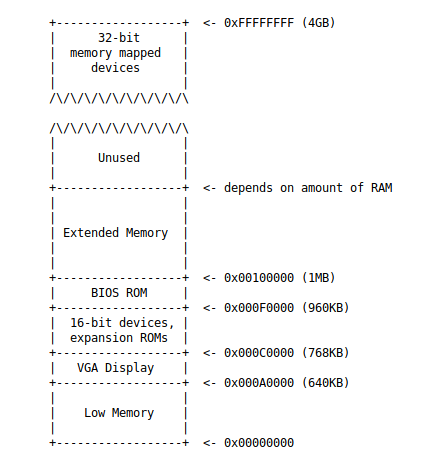


图1-1 PC物理地址空间布局

早期的8088处理器只有20根地址线，寻址空间为1MB，因此它们的物理地址始于0x00000并止于0xFFFFF。在这1MB物理地址空间中低640KB用作RAM,供各类程序随机读写。之后从0x000A0000到0x000FFFFF这384KB地址保留给各类硬件使用，如用作video display buffers、firmware等，因此这块区域又命名为I/O hole。在I/O hole中最重要的部分是基本输入/输出系统BIOS,BIOS首先执行系统初始化，如激活显卡、检测内存大小然后从启动设备读取boot loader 并将控制权转交给boot loader，之后boot loader加载kernel并执行从而完成PC的启动过程。

现在的X86处理器早已经支持4GB的寻址空间，为了实现向后兼容这类处理器依旧保留了低1M空间的物理布局。因此在现代计算机在0x000A0000到 0x00100000有一个hole将RAM分割成low memory和 extended memory.此外在32位地址的顶部空间也有一个hole保留给BIOS,供其为各类32位PCI设备使用。

如上即为PC物理地址空间布局的介绍。

###### 2.1.4 The ROM BIOS

本部分将探究BIOS在PC启动过程中所做的工作。PC在加电之后，CS和IP即被置为0xf000以及0xfff0。在实模式下物理地址是根据CS<<4+IP计算得出，也即PC执行的第一条指令位于物理地址0xFFFF0.该指令是一条长跳转指令，用于执行BIOS ROM较低地址处的其余指令。

|  |
| --- |
| Exercise 2. Use GDB's si (Step Instruction) command to trace into the ROM BIOS for a few more instructions, and try to guess what it might be doing. You might want to look at [Phil Storrs I/O Ports Description](http://web.archive.org/web/20040404164813/members.iweb.net.au/~pstorr/pcbook/book2/book2.htm), as well as other materials on the [6.828 reference materials page](https://pdos.csail.mit.edu/6.828/2018/reference.html). No need to figure out all the details - just the general idea of what the BIOS is doing first. |

为了确定BIOS的执行流程，在一个终端中执行make qemu-nox-gdb命令，在另一个终端中使用make gdb即可实现对BIOS的单步跟踪。GDB跟踪的前几条指令主要是堆栈设置相关，结果如下：

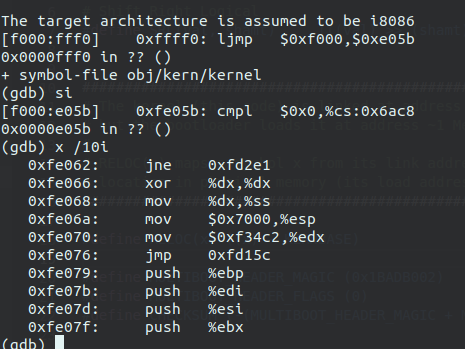


图1-2 BIOS的堆栈设置

BIOS首先跳转到[f000:e05b]处执行比较指令做分支检查。Xor %dx,%dx ; mov %dx，%ss将ss段寄存器置0。mov $0x7000,%esp使esp指向栈顶0x7000.设置好BIOS的堆栈后执行jmp，跳转到0xfd15c。

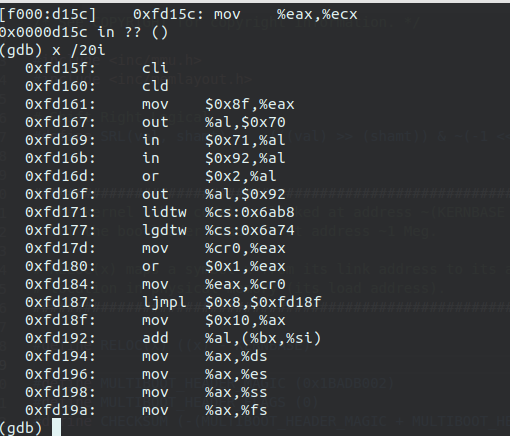


图1-3 BIOS进入保护模式，设置IDT,GDT

0xfd15c的前20条指令图1-3所示，cli关中断，cld设置字符串指令的方向。mov $0x8f,%eax out %al,$0x70 in 0x71,%al这几条指令用于读取CMOS的相关参数。在端口手册中可以查询到70端口和71端口参数的作用。

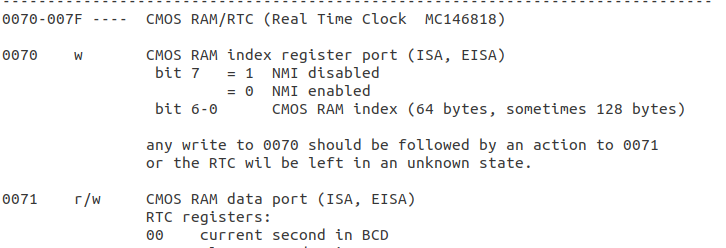


图1-4 70端口参数示意

往70端口写一个字节时，bit 7用于设置NMI, 0-6表示RTS寄存器的索引。指令中0x8f表示将NMI 关闭并索引0x0f号RTS寄存器。该寄存器存shutdown status byte，通过71号端口的in指令递交给BIOS。

In $0x92,%al or $0x2,%al out %al,$0x92这3条指令则是将0x92号端口寄存器的bit 1置为1，用来表明激活A20模式。

Lidtw lgdtw分别表示设置好IDTR、GDTR的值。

mov %cr0,%eax or $0x1,%eax mov %eax,%cr0打开cr0控制寄存器的PE位允许分段。ljmpl $0x8,$0xfd18f将CS设置为8,IP置为0xfd18f正式进入保护模式。

后续指令跟踪则比较困难，大体是设置好IDT并循环读取以及初始化各类设备。最后BIOS会从启动设备中读取boot loader并转交控制权。这部分细节此处不在详述。

##### **2.2 Part 2: The Boot Loader**

当BIOS找到一个启动磁盘时，会将该磁盘第一个扇区（启动扇区）加载到物理地址0x7c00处，该扇区存放boot loader程序。之后BIOS将控制权转交给boot loader并执行相应指令。 Boot loader主要实现两个主要功能：

1. 将处理器从实模式转变为32位保护模式，只有在保护模式下程序才能访问所有高于1MB的内存。
2. 通过特殊的I/O指令直接访问磁盘设备读取kernel。

Boot loader的上述两个功能分别在boot/boot.s以及boot/main.c文件中予以实现。在分析这两个文件之前我们先熟悉一下GDB的调试命令：

|  |
| --- |
| Exercise 3. Take a look at the [lab tools guide](https://pdos.csail.mit.edu/6.828/2018/labguide.html), especially the section on GDB commands. Even if you're familiar with GDB, this includes some esoteric GDB commands that are useful for OS work.  Set a breakpoint at address 0x7c00, which is where the boot sector will be loaded. Continue execution until that breakpoint. Trace through the code in boot/boot.S, using the source code and the disassembly file obj/boot/boot.asm to keep track of where you are. Also use the x/i command in GDB to disassemble sequences of instructions in the boot loader, and compare the original boot loader source code with both the disassembly in obj/boot/boot.asm and GDB.  Trace into bootmain() in boot/main.c, and then into readsect(). Identify the exact assembly instructions that correspond to each of the statements in readsect(). Trace through the rest of readsect() and back out into bootmain(), and identify the begin and end of the for loop that reads the remaining sectors of the kernel from the disk. Find out what code will run when the loop is finished, set a breakpoint there, and continue to that breakpoint. Then step through the remainder of the boot loader. |

GDB中常用的命令包括用b \*addr、b function\_name来设置断点、c命令执行到下一个断点、x /Nx addr打印addr处连续N个的内存值以及用x /Ni addr 来打印addr处的N条连续指令。

在exercise3中可以在0x7c00处设置一个断点，并用continue命令执行到0x7c00处,之后便可使用si来单步跟踪指令的执行，我们可以通过查看obj/boot/boot.asm文件来比较源代码和反汇编代码的区别。同样地通过分析该文件可以很方便地定位以及查找相应函数的汇编指令，这里不在赘述。

下面简要分析boot.s以及main.c的执行流程。

Boot.s首先开启A20模式，处理器在默认情况下第21根地址线是关闭的，这样做是为了兼容实模式1M的寻址空间，但是在进入保护模式后寻址达到4G,需要启用第21根地址线，因此我们在进入该模式之前需先开启A20。

开启A20之后可以设置相关寄存器来正式进入保护模式了，首先需要设置好GDT并加载好GDTR,然后将cr0控制寄存器的PE位置1表示允许分段,最后使用ljmp指令重置CS和EIP即完成分段功能。如下是与这部分相关的汇编代码：

|  |
| --- |
| lgdt gdtdesc //load GDT describer to GDTR  movl %cr0, %eax  orl $CR0\_PE\_ON, %eax //open PE in cr0  movl %eax, %cr0  ljmp $PROT\_MODE\_CSEG, $protcseg  gdt: //GDT  SEG\_NULL  SEG(STA\_X|STA\_R, 0x0, 0xffffffff)  SEG(STA\_W, 0x0, 0xffffffff)  gdtdesc: //gdt describer(48 bits)  .word 0x17  .long gdt |

最后boot.s通过call bootmain指令开始执行main.c文件中的代码。

Main.c主要完成kernel的加载并执行的功能。我们重点分析bootmain函数，其余函数可以参见实验文件的相关注释。

|  |
| --- |
| Void bootmain(void)  {  struct Proghdr \*ph, \*eph;  // read 1st page off disk  readseg((uint32\_t) ELFHDR, SECTSIZE\*8, 0);  // is this a valid ELF?  if (ELFHDR->e\_magic != ELF\_MAGIC)  goto bad;  // load each program segment (ignores ph flags)  ph = (struct Proghdr \*) ((uint8\_t \*) ELFHDR + ELFHDR->e\_phoff);//ph指向program header  eph = ph + ELFHDR->e\_phnum;//eph指向程序头表的尾部  for (; ph < eph; ph++)//循环读取程序段  // p\_pa is the load address of this segment (as well  // as the physical address)  readseg(ph->p\_pa, ph->p\_memsz, ph->p\_offset);  // call the entry point from the ELF header  // note: does not return!  ((void (\*)(void)) (ELFHDR->e\_entry))();//开始执行kernal的代码  bad:  outw(0x8A00, 0x8A00);  outw(0x8A00, 0x8E00);  while (1)  /\* do nothing \*/;  } |

Main.c 首先从1号扇区开始读取一页文件到ELFHDR处，然后检查该ELF文件的magic number是否有效，无效表示磁盘数据被损坏执行bad处的指令，否则通过ELF文件头找到程序头表（program header table)的起始地址并赋值给ph指针，同时读取ELF header中program header table的表项数进而计算出程序头表的结束地址并将其赋值给eph指针,之后移动ph指针循环读取各个程序段到指定物理地址即完成了kernel的加载。有关ELF文件格式相关的内容可参见2.2.1节相关部分。

###### 2.2.1 Loading the Kernel

在深入探索boot/main.c文件之前先回顾一下C语言的相关基础知识。

|  |
| --- |
| Exercise 4. Read about programming with pointers in C. The best reference for the C language is *The C Programming Language* by Brian Kernighan and Dennis Ritchie (known as 'K&R'). We recommend that students purchase this book (here is an [Amazon Link](http://www.amazon.com/C-Programming-Language-2nd/dp/0131103628/sr=8-1/qid=1157812738/ref=pd_bbs_1/104-1502762-1803102?ie=UTF8&s=books)) or find one of [MIT's 7 copies](http://library.mit.edu/F/AI9Y4SJ2L5ELEE2TAQUAAR44XV5RTTQHE47P9MKP5GQDLR9A8X-10422?func=item-global&doc_library=MIT01&doc_number=000355242&year=&volume=&sub_library=).  Read 5.1 (Pointers and Addresses) through 5.5 (Character Pointers and Functions) in K&R. Then download the code for [pointers.c](https://pdos.csail.mit.edu/6.828/2018/labs/lab1/pointers.c), run it, and make sure you understand where all of the printed values come from. In particular, make sure you understand where the pointer addresses in printed lines 1 and 6 come from, how all the values in printed lines 2 through 4 get there, and why the values printed in line 5 are seemingly corrupted.  There are other references on pointers in C (e.g., [A tutorial by Ted Jensen](https://pdos.csail.mit.edu/6.828/2018/readings/pointers.pdf) that cites K&R heavily), though not as strongly recommended.  *Warning:* Unless you are already thoroughly versed in C, do not skip or even skim this reading exercise. If you do not really understand pointers in C, you will suffer untold pain and misery in subsequent labs, and then eventually come to understand them the hard way. Trust us; you don't want to find out what "the hard way" is. |

有关C语言部分的知识在大一就已经学过，对其还是比较熟悉，唯一需要提及的是有关指针的加减法。一般而言,对于类型为type \*的指针p,p+i所指向的地址是p所指向的地址加上sizeof(type)\*i字节的偏移。

需要完全弄懂boot/main.c还需要了解ELF文件的基本格式，ELF文件有两种类型，一种是可重定向文件，另外一种是可执行文件。这两种类型文件分别对应下图左边的链接视图以及右边的可执行视图：

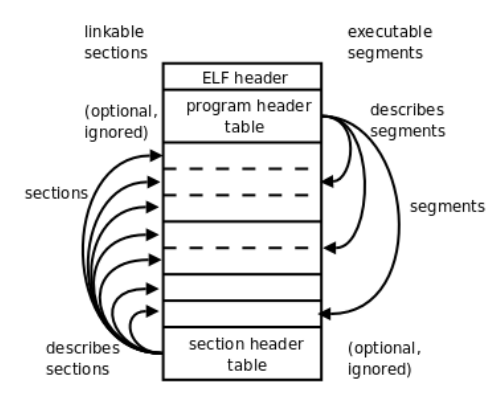


图1-4 ELF文件的两种视图

显然Kernel是一个可执行文件，因此我们着重分析ELF文件的可执行视图。从上图不难发现ELF文件由ELF header、program header table以及各个程序段组成。在inc/elf.h中找到ELF header、program header的结构声明：

|  |
| --- |
| struct Elf { //ELF header 的结构声明  uint32\_t e\_magic; // must equal ELF\_MAGIC  uint8\_t e\_elf[12];  uint16\_t e\_type;  uint16\_t e\_machine;  uint32\_t e\_version;  uint32\_t e\_entry; //.text段的入口地址  uint32\_t e\_phoff; //program header table起始地址的偏移，用于计算起始地址  uint32\_t e\_shoff;  uint32\_t e\_flags;  uint16\_t e\_ehsize;  uint16\_t e\_phentsize;  uint16\_t e\_phnum; //program header num,用于计算程序段头表的结束地址  uint16\_t e\_shentsize;  uint16\_t e\_shnum;  uint16\_t e\_shstrndx;  };  struct Proghdr { //program header table 的表项，描述每个program header的信息  uint32\_t p\_type;  uint32\_t p\_offset; //该程序段的偏移，用来计算程序段的起始地址  uint32\_t p\_va;  uint32\_t p\_pa; //该程序段被加载到的物理地址  uint32\_t p\_filesz; //文件大小  uint32\_t p\_memsz;//内存大小，p\_memsz>=p\_filesz,因为.bss段占用内存  uint32\_t p\_flags;  uint32\_t p\_align;  }; |

上述结构中比较重要的成员均给出了注释说明，结合boot/main.c中的代码我们可以清楚地知道kernel的加载过程了。Boot loader首先从1号扇区读取一页的文件到物理地址ELFHDR(0x10000)处，在这一页的文件中包含了kernel的ELF header以及program header table。然后通过分析ELF header的成员信息e\_phoff以及e\_phnum可以计算出program header table的起始和终止地址，确定program header的个数。接着依次遍历每个program header将相应程序段从磁盘特定位置（由p\_offset计算得出）读取到指定的物理地址（p\_pa)。这样便完成了kernel的加载了。

ELF文件的加载需要提及的一点是装载地址LMA和链接地址VMA的区别。LMA是程序装载到内存的物理地址。VMA则是程序执行时所处的地址空间，由链接器ld决定。一般而言，LMA与VMA是相同的，如果不同，那么在执行与绝对地址相关的指令时就会产生错误。

|  |
| --- |
| Exercise 5. Trace through the first few instructions of the boot loader again and identify the first instruction that would "break" or otherwise do the wrong thing if you were to get the boot loader's link address wrong. Then change the link address in boot/Makefrag to something wrong, run make clean, recompile the lab with make, and trace into the boot loader again to see what happens. Don't forget to change the link address back and make clean again afterward |

在boot/Makefrag中将-Ttext后的VMA修改为0x8c00,使用make命令重新编译后观察obj/boot/boot.asm。可以发现所有指令的运行地址是以0x8c00开始。

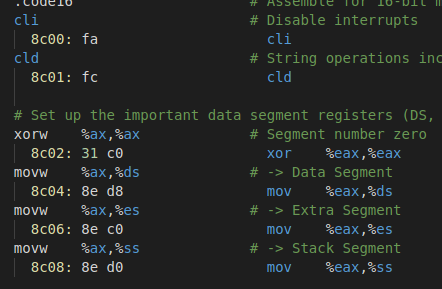


图1-5 修改后的VMA

在boot.s中与地址相关的指令有jnb,jnz,ljmp。由于条件跳转指令计算的是相对地址，属于地址无关指令，而ljmp是绝对地址跳转，因此修改VMA之后第一条出错的指令是ljmp.在GDB中可以看到这条指令将跳到0x8c32处而不是下一条指令0x7c32。执行该指令，qemu 将发生triple fault。

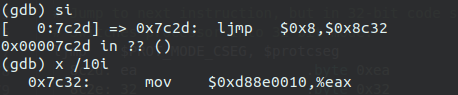


图1-6 第一条出错指令

Boot/main.c在加载完kernel后会执行如下指令跳到kernel的entry point:

((void (\*)(void)) (ELFHDR->e\_entry))();//开始执行kernal的代码

使用readelf -h obj/kern/kernel可以查看到kernel的entry point是0x10000c。

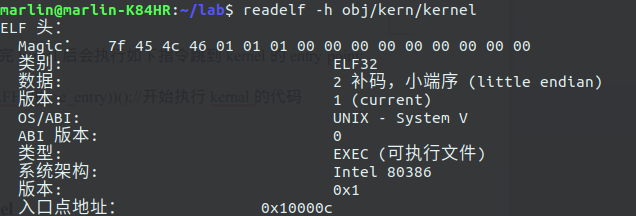


图1-7 kernel的entry point

|  |
| --- |
| Exercise 6. We can examine memory using GDB's x command. The [GDB manual](https://sourceware.org/gdb/current/onlinedocs/gdb/Memory.html) has full details, but for now, it is enough to know that the command x/*N*x *ADDR* prints *N* words of memory at *ADDR*. (Note that both 'x's in the command are lowercase.) *Warning*: The size of a word is not a universal standard. In GNU assembly, a word is two bytes (the 'w' in xorw, which stands for word, means 2 bytes).  Reset the machine (exit QEMU/GDB and start them again). Examine the 8 words of memory at 0x00100000 at the point the BIOS enters the boot loader, and then again at the point the boot loader enters the kernel. Why are they different? What is there at the second breakpoint? (You do not really need to use QEMU to answer this question. Just think.) |

首先在0x7c00处设置断点并执行到此处，使用x /8x 0x00100000打印0x00100000处的8个words。然后在0x0010000c处设置一个断点并用相同的方式执行到该处打印8个words。如下是两次输出的结果：

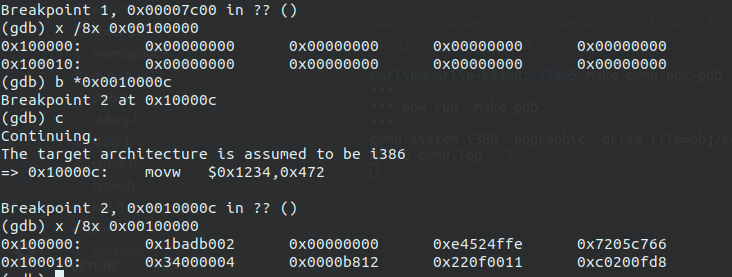


图1-8 两次输出结果

第一次输出时的8个words全0，而第二次输出的并不是全0。这是因为在进入kernel的entry point的时候boot loader已经将kernel程序段加载到了0x00100000处。

##### **2.3 Part 3: The Kernel**

###### 2.3.1 Using virtual memory to work around position dependen ce

通过上述分析我们知道kernel被装载到了物理地址0x00100000处，也即其LVM是0x00100000,但是为了实现虚拟内存工作的位置无关性，往往会将其链接地址（或运行地址）设置为较高的虚拟地址。因此我们必须设置一种从高虚拟地址到低物理地址之间的映射关系。在本次实验中我们只需要完成虚拟地址[0x0000000 0,0x0ffffffff]以及[0xf0000000,0xffffffff]到物理地址[0x00000000,0x0fffffff]的映射。这可以通过静态建立页目录和页表并开启分页功能来实现。

具体而言，在entrypgdir.s中设置了一个页目录和一个页表：

|  |
| --- |
| pde\_t entry\_pgdir[NPDENTRIES] = {  // Map VA's [0, 4MB) to PA's [0, 4MB)  [0]  = ((uintptr\_t)entry\_pgtable - KERNBASE) + PTE\_P,  // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)  [KERNBASE>>PDXSHIFT]  = ((uintptr\_t)entry\_pgtable - KERNBASE) + PTE\_P + PTE\_W  };  // Entry 0 of the page table maps to physical page 0, entry 1 to  // physical page 1, etc.  pte\_t entry\_pgtable[NPTENTRIES] = {  0x000000 | PTE\_P | PTE\_W,  0x001000 | PTE\_P | PTE\_W, |

在页目录中设置了两个页目录项，分别对应低4MB和高4MB虚拟地址，两者均映射到相同的entry\_patable了。而entry\_pgtable中第i项直接映射第i页。这样就可以实现虚拟地址空间中的高低4MB均映射到物理地址的低4MB。

|  |
| --- |
| Exercise 7. Use QEMU and GDB to trace into the JOS kernel and stop at the movl %eax, %cr0. Examine memory at 0x00100000 and at 0xf0100000. Now, single step over that instruction using the stepi GDB command. Again, examine memory at 0x00100000 and at 0xf0100000. Make sure you understand what just happened.  What is the first instruction *after* the new mapping is established that would fail to work properly if the mapping weren't in place? Comment out the movl %eax, %cr0 in kern/entry.S, trace into it, and see if you were right. |

在开启分页之前，虚拟地址0xf0100000直接映射的是物理地址0xf0100000.因此此时内存值为0，而开启分页之后，0xf0100000映射到了物理地址0x00100 000处，所以内存值与虚拟地址0xf0100000是相同的了。如下是分页前后的结果：

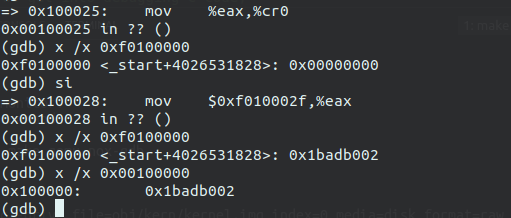


图1-9 分页前后0xf0100000处的内存值

###### 2.3.2 Formatted Printing to the Console

|  |
| --- |
| Exercise 8. We have omitted a small fragment of code - the code necessary to print octal numbers using patterns of the form "%o". Find and fill in this code fragment. |

追踪cprintf函数的调用关系可以知道其先后调用顺序为cprintf->vcprintf-> vprintfmt。因此最终是由vprintfmt完成格式化字符串的分析工作。稍加分析不难发现该函数首先执行普通字符的输出直至遇到‘%’。

|  |
| --- |
| while ((ch = \*(unsigned char \*) fmt++) != '%') {  if (ch == '\0')  return;  putch(ch, putdat);  } |

之后对‘%’后面的特殊字符进行判断分别处理，整个判断是由一个大的switch完成，找到有关于‘%u’和‘%o’两种格式的处理代码，可以发现8进制输出与10进制输出除了进制不同外其余均相同。因此我们只需复制‘%u’的处理代码修改基数即可：

|  |
| --- |
| // unsigned decimal  case 'u':  num = getuint(&ap, lflag);  base = 10;  goto number;  // (unsigned) octal  case 'o': //仿照上面的将base改成8即可  // Replace this with your code.  num = getuint(&ap,lflag);  base = 8;  goto number; |

###### 2.3.3 The Stack

本部分将探究C语言是如何在X86上使用栈并完成kernel monitor函数回溯打印该栈。

|  |
| --- |
| Exercise 9. Determine where the kernel initializes its stack, and exactly where in memory its stack is located. How does the kernel reserve space for its stack? And at which "end" of this reserved area is the stack pointer initialized to point to? |

在kern/entry.s中可以发现与kernel stack初始化相关的代码：

|  |
| --- |
| movl $0x0,%ebp # nuke frame pointer  # Set the stack pointer  movl $(bootstacktop),%esp  .data  # boot stack  .p2align PGSHIFT # force page alignment  .globl bootstack  bootstack:  .space KSTKSIZE  .globl bootstacktop  bootstacktop: |

易知kernel分配了一个KSTKSIZE大小的栈位于[bootstack,bootstacktop]。并将栈顶地址$(bootstacktop)赋给esp。其中KSTKSIZE的值为4KB,即该栈大小为一页。Bootstacktop的偏移在obj/kern/kernel.asm中显示为0xf0110000。即esp指向0xf0110000.

|  |
| --- |
| Exercise 10. To become familiar with the C calling conventions on the x86, find the address of the test\_backtrace function in obj/kern/kernel.asm, set a breakpoint there, and examine what happens each time it gets called after the kernel starts. How many 32-bit words does each recursive nesting level of test\_backtrace push on the stack, and what are those words?  Note that, for this exercise to work properly, you should be using the patched version of QEMU available on the [tools](https://pdos.csail.mit.edu/6.828/2018/tools.html) page or on Athena. Otherwise, you'll have to manually translate all breakpoint and memory addresses to linear addresses. |

在test\_backtrace函数中会先后递归调用本身6次打印entering xxx，然后再回溯输出leving xxx。先enter的后leave。关于每层函数占有的stack words可参见下表的分析：

|  |
| --- |
| push %ebp  mov %esp,%ebp //将ebp压栈，ebp记录帧的起始地址  push %ebx //保存ebx寄存器，该寄存器是由被调用者保护的寄存器  sub $0x14,%esp //为该函数预留空间用作临时变量或子函数实参  call f0100040 <test\_backtrace> //call会将eip压栈 |

由上表易知共保留了4+4+20+4 = 32字节（4words)为函数使用。

|  |
| --- |
| Exercise 11. Implement the backtrace function as specified above. Use the same format as in the example, since otherwise the grading script will be confused. When you think you have it working right, run make grade to see if its output conforms to what our grading script expects, and fix it if it doesn't. *After* you have handed in your Lab 1 code, you are welcome to change the output format of the backtrace function any way you like. |

要实现回溯打印栈信息的功能，可以利用ebp指针依次跳到上一帧的起始处，然后输出相关信息。在此之前我们已经知道了函数调用的帧栈布局：

|  |
| --- |
| 实参5 |
| 实参... |
| 实参1 |
| EIP |
| EBP |
| ... |

图1-10 函数帧栈布局图

那么回溯什么时候结束呢？在kern/entry.s中ebp被赋初值0，执行第一个函数时压人的ebp显然就是0。我们只要发现某个帧中存的ebp值等于0就可以停止。如下是该函数的代码：

|  |
| --- |
| int  mon\_backtrace(int argc, char \*\*argv, struct Trapframe \*tf)  {  // Your code here.  uint32\_t\* ebp=(uint32\_t\*) read\_ebp();//读取ebp中的地址  cprintf("Stack backtrace:\n");  for(;ebp!=0;ebp=(uint32\_t\*)\*ebp){  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]);  }  return 0;  } |

|  |
| --- |
| Exercise 12. Modify your stack backtrace function to display, for each eip, the function name, source file name, and line number corresponding to that eip.  In debuginfo\_eip, where do \_\_STAB\_\* come from? This question has a long answer; to help you to discover the answer, here are some things you might want to do:   * look in the file kern/kernel.ld for \_\_STAB\_\* * run objdump -h obj/kern/kernel * run objdump -G obj/kern/kernel * run gcc -pipe -nostdinc -O2 -fno-builtin -I. -MD -Wall -Wno-format -DJOS\_KERNEL -gstabs -c -S kern/init.c, and look at init.s. * see if the bootloader loads the symbol table in memory as part of loading the kernel binary   Complete the implementation of debuginfo\_eip by inserting the call to stab\_binsearch to find the line number for an address.  Add a backtrace command to the kernel monitor, and extend your implementation of mon\_backtrace to call debuginfo\_eip and print a line for each stack frame of the form:  K> backtrace  Stack backtrace:  ebp f010ff78 eip f01008ae args 00000001 f010ff8c 00000000 f0110580 00000000  kern/monitor.c:143: monitor+106  ebp f010ffd8 eip f0100193 args 00000000 00001aac 00000660 00000000 00000000  kern/init.c:49: i386\_init+59  ebp f010fff8 eip f010003d args 00000000 00000000 0000ffff 10cf9a00 0000ffff  kern/entry.S:70: <unknown>+0  K> |

现分别完成上述任务。

\_STAB\_BEGIN和\_STAB\_END\_在kern/kern.ld中均有定义，分别表示了.stab段的开始和结束地址。

|  |
| --- |
| /\* Include debugging information in kernel memory \*/  .stab : {  PROVIDE(\_\_STAB\_BEGIN\_\_ = .);  \*(.stab);  PROVIDE(\_\_STAB\_END\_\_ = .);  BYTE(0) /\* Force the linker to allocate space  for this section \*/  } |

在ELF文件中.stab段和.stabstr段主要存各种调试信息。执行objdump -h obj/kern/kernel可以看到各段大小、装载地址等信息。

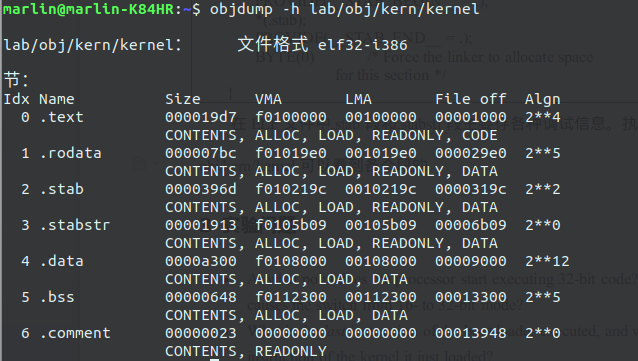


图1-11 ELF段信息

执行objdump -G obj/kern/kernel得到.stab段的具体内容，为了节省空间,此处仅截取部分内容。

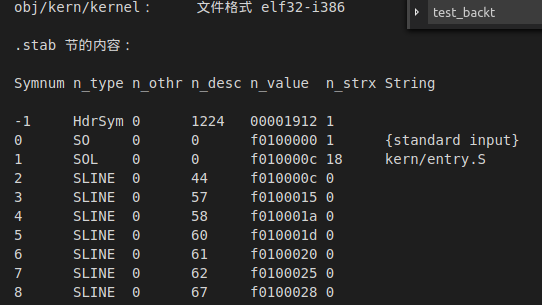


图1-12 .stab段信息

.stab段是一个结构数组。该结构的定义可在kern/stab.h中找到。

|  |
| --- |
| // Entries in the STABS table are formatted as follows.  struct Stab {  uint32\_t n\_strx; // index into string table of name  uint8\_t n\_type; // type of symbol  uint8\_t n\_other; // misc info (usually empty)  uint16\_t n\_desc; // description field  uintptr\_t n\_value; // value of symbol  }; |

这5个成员分别对应图1-12的5列信息。Symnum是数组索引，n\_type是调试信息的类型，如SO表示源文件，FUN表示函数，SLINE表示代码行。需要注意的是这些信息的存放是有规律的，该数组从低到高会依次遍历每个文件、函数、行号等，例如SO FUN SLINE SLINE FUN SLINE SO ...。两个SO之间是一个源文件的调试信息，同理可知两个FUN之间是一个函数的信息...，该特征可被用来二分查找调试信息。

执行命令gcc -pipe -nostdinc -O2 -fno-builtin -I. -MD -Wall -Wno-format -DJOS\_KERNEL -gstabs -c -S kern/init.c 在init.s中可以看到汇编文件中的各种调试信息被显示出来。

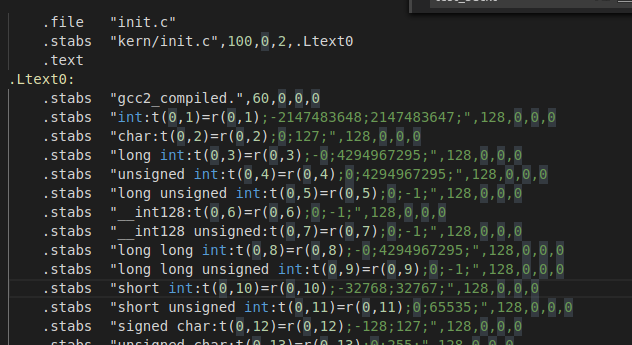


图1-13 init.s中的调试信息

为验证boot loader在加载kernel的时候会加载与调试信息有关的.stab段和.stabstr段。在gdb中打印.stabstr段LVM处的内存值，可以看到字符信息。

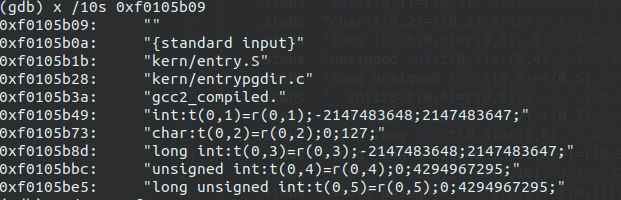


图1-14 .stabstr段LVM处内存值

理清思路后可以在debuginfo\_eip中添加查询行号的代码。实验文件已给出查询源文件、函数的代码。只需继续调用stab\_binsearch即可找到行号。如下是与之相关的实现：

|  |
| --- |
| stab\_binsearch(stabs, &lline, &rline, N\_SLINE, addr);  if(lline<=rline)  info->eip\_line=stabs[lline].n\_desc;  else return -1; |

本实验最后一步是安装backtrace命令。直接在monitor.c的commond数组中添加一行{ "backtrace", "show debug info of functions", mon\_backtrace}即可。此处不再赘述。

#### 3 实验问题

1. At what point does the processor start executing 32-bit code? What exactly causes the switch from 16- to 32-bit mode?

当处理器设置好GDTR并将CR0控制寄存器的PE位置1后即开始执行32位代码。当执行完ljmp $PROT\_MODE\_CSEG, $protcseg长跳转指令后正式进入32位模式

1. What is the *last* instruction of the boot loader executed, and what is the *first* instruction of the kernel it just loaded?

Boot loader最后执行的指令是跳转到kernel的entry point：

|  |
| --- |
| ((void (\*)(void)) (ELFHDR->e\_entry))();//开始执行kernal的代码  7d61: ff 15 18 00 01 00 call \*0x10018 |

Kernel执行的第一条指令是热启动指令：

|  |
| --- |
| entry:  movw $0x1234,0x472 # warm boot |

1. *Where* is the first instruction of the kernel?

使用readelf -h obj/kern/kernel可以看到kernel的入口地址是0x0010000c。用GDB在此处设置断点运行后可以看到正是kernel的第一条指令。

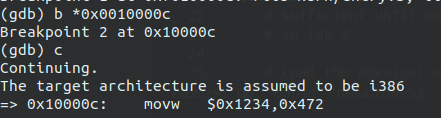


图1-15 kernel的第一条指令

1. How does the boot loader decide how many sectors it must read in order to fetch the entire kernel from disk? Where does it find this information?

Boot loader首先会加载kernel的前一页到物理地址0x10000(ELFHDR).读取ELF header中e\_phoff和e\_phnum可以确定program header table的开始和结束地址。然后boot loader会遍历program header table读取每个段所在的磁盘地址以及该段大小并放到指定的物理地址中。所以boot loader是从ELF header和program header中读取需要信息的。

1. Explain the interface between printf.c and console.c. Specifically, what function does console.c export? How is this function used by printf.c?

printf.c会需要调用putch函数完成字符的输出，追踪putch可以得到如下的调用链：putch->cputchar->cons\_putc->serial\_putc(c)/lpt\_putc(c)/cga\_putc(c)。其中cputchar以及之后的函数位于console.c文件中。因此console.c export的函数是cput\_char。而printf.c使用该接口封装自己的putc供其余函数使用。

1. Explain the following from console.c:

* 1 if (crt\_pos >= CRT\_SIZE) {
* 2 int i;
* 3 memmove(crt\_buf, crt\_buf + CRT\_COLS, (CRT\_SIZE - CRT\_COLS) \* sizeof(uint16\_t));
* 4 for (i = CRT\_SIZE - CRT\_COLS; i < CRT\_SIZE; i++)
* 5 crt\_buf[i] = 0x0700 | ' ';
* 6 crt\_pos -= CRT\_COLS;
* 7 }

上述代码实现的是翻屏功能。首先判断crt\_pos是否超过CRT\_SIZE，超过说明当前屏幕已被写满。这时需要调用memmove将第二行开始全部往上移动一行，将最后一行用空格填充并将光标移动到最后一行的第一列。

1. For the following questions you might wish to consult the notes for Lecture 2. These notes cover GCC's calling convention on the x86.

* Trace the execution of the following code step-by-step:

1. int x = 1, y = 3, z = 4;
2. cprintf("x %d, y %x, z %d\n", x, y, z);

In the call to cprintf(), to what does fmt point? To what does ap point?

fmt指向"x %d, y %x, z %d\n"。ap指向可变参数列表va\_list，此时指向第一个参数x.

* List (in order of execution) each call to cons\_putc, va\_arg, and vcprintf. For cons\_putc, list its argument as well. For va\_arg, list what ap points to before and after the call. For vcprintf list the values of its two arguments.

可以在这些函数处设置断点，然后依次运行便可以跟踪先后调用的函数以及它们的参数，由于信息过多，此处省略。

1. Run the following code.
2. unsigned int i = 0x00646c72;
3. cprintf("H%x Wo%s", 57616, &i);

* What is the output? Explain how this output is arrived at in the step-by-step manner of the previous exercise.
* The output depends on that fact that the x86 is little-endian. If the x86 were instead big-endian what would you set i to in order to yield the same output? Would you need to change 57616 to a different value?

上面两行代码的输出为“He110 World”.因为57616的16进制是e110,并且0x72,0x6c,0x64,0x00分别表示字符rld\0。如果机器是大端模式，需要将i改为0x726c6400，而57616不需要改变。

1. In the following code, what is going to be printed after 'y='? (note: the answer is not a specific value.) Why does this happen?
2. cprintf("x=%d y=%d", 3);

打印的将是栈中不确定的4字节内容。因为ap指针首先指向3所在地址，之后会移动到下一个参数，而栈中有效的实参只有一个，故会打印一个不确定值。

1. Let's say that GCC changed its calling convention so that it pushed arguments on the stack in declaration order, so that the last argument is pushed last. How would you have to change cprintf or its interface so that it would still be possible to pass it a variable number of arguments?

改变压栈顺序后各参数从左至右依次对应高到低的地址。因此va指针指向下一个参数时要减去一个参数所占字节。可以将va\_start、va\_list内建宏中的加号全改为减号。

### Lab2 Memory Management

#### **1 实验目的**

本次实验将完成操作系统内存管理代码。内存管理主要有两部分组成。

1. 实现kernel的内核分配器，理解kernel是如何建立数据结构记录各物理页的分配状态，并熟悉按页分配、回收的各个函数例程。
2. 建立内核的虚拟内存，初步熟悉JOS中二级页表在分页过程中的转换流程，能画出虚拟地址空间的内存布局。

#### **2 实验内容**

##### **Part 1: Physical Page Management**

操作系统必须能跟踪物理RAM的使用状态，将其记录在特定数据结构中以方便后续的使用和释放。本实验中使用struct PageInfo来实现该功能，PageInfo中包含有两个数据成员：struct PageInfo \*pp\_link和uint16\_t pp\_ref。这两个成员分别用来链接到下一个page以及记录本page的引用数。了解了基本数据后可以完成第一个任务：

|  |
| --- |
| Exercise 1. In the file kern/pmap.c, you must implement code for the following functions (probably in the order given).  boot\_alloc() mem\_init() (only up to the call to check\_page\_free\_list(1)) page\_init() page\_alloc() page\_free()  check\_page\_free\_list() and check\_page\_alloc() test your physical page allocator. You should boot JOS and see whether check\_page\_alloc() reports success. Fix your code so that it passes. You may find it helpful to add your own assert()s to verify that your assumptions are correct. |

boot\_alloc函数是kernel最先使用的内存分配函数，由于此时尚未完成记录各个物理页的数据结构，因而会利用该函数来建立页目录kern\_pgdir以及物理页数组pages。在kernel.ld中使用end作为.bss段末尾的标号，即end是内存中紧邻kernel的地址。

|  |
| --- |
| .bss : { //kernel最后一个段  PROVIDE(edata = .);  \*(.bss)  PROVIDE(end = .); //end flag  BYTE(0)  } |

boot\_alloc从end位置开始分配若干页来存放n字节内容，通过使用一个静态变量nextfree来记录下一次分配时的起始地址，利用ROUNDUP宏可以很方便地按页对齐。如下是boot\_alloc的实现：

|  |
| --- |
| 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.  result = nextfree ;  nextfree = ROUNDUP(result+n,PGSIZE);  if(nextfree < result) //内存越界,nextfree溢出会比result小  panic("boot alloc: OUT OF MEMORY!!!\n");  return result;  } |

在mem\_init中我们需要补齐check\_page\_free\_list(1)之前的代码。缺失的部分主要是分配pages数组以记录物理页状态，在i386\_detect\_momory中已经通过读取特定的端口信息提取出了物理内存大小以及计算出物理页个数npages。因此只需调用boot\_alloc函数。

|  |
| --- |
| pages = (struct PageInfo\*)boot\_alloc(npages\*sizeof(struct PageInfo));  memset(pages,0,sizeof(struct PageInfo)\*npages); |

Page\_init按照注释中的提示设置好各PageInfo即可。在写代码之前先画出物理页的状态图：

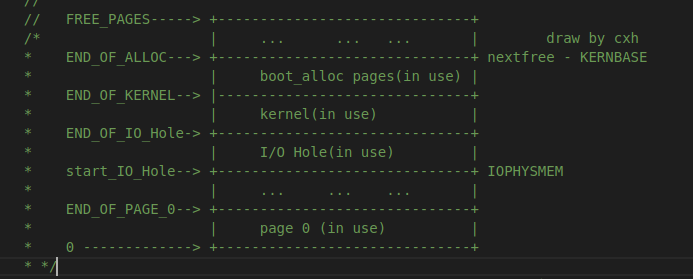


图2-1 物理页状态图

对照上图可以写出如下的代码：

|  |
| --- |
| void  page\_init(void)  {  size\_t start\_IO\_hole = IOPHYSMEM / PGSIZE;  size\_t end\_boot\_alloc = PADDR(boot\_alloc(0)) / PGSIZE;  size\_t i;  for (i = 0; i < npages; i++) {  if(i==0||(i>=start\_IO\_hole && i<=end\_boot\_alloc)){  pages[i].pp\_ref = 1;  pages[i].pp\_link = NULL;  }  else{  pages[i].pp\_ref = 0;  pages[i].pp\_link = page\_free\_list;  page\_free\_list = &pages[i];  }  }  } |

接着完成page\_alloc和page\_free。Page\_alloc从page\_free\_list中取出一页，page\_free将一页重新放到page\_free\_list中。直接给出这两个函数的代码实现：

|  |
| --- |
| struct PageInfo \*  page\_alloc(int alloc\_flags)  {  // Fill this function in  if(page\_free\_list == NULL) return NULL;  struct PageInfo \*pg = page\_free\_list;  page\_free\_list = page\_free\_list->pp\_link;  pg->pp\_link = NULL;  if(alloc\_flags & ALLOC\_ZERO)  memset(page2kva(pg),0,PGSIZE);  return pg;  }  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\_ref!=0||pp->pp\_link!=NULL)  panic("page\_free:pp->pp\_ref is nonzero or pp->pp\_link is not NULL\n");  else{  pp->pp\_link = page\_free\_list;  page\_free\_list = pp;  }  } |

##### **Part 2: Virtual Memory**

|  |
| --- |
| Exercise 2. Look at chapters 5 and 6 of the [Intel 80386 Reference Manual](https://pdos.csail.mit.edu/6.828/2018/readings/i386/toc.htm), if you haven't done so already. Read the sections about page translation and page-based protection closely (5.2 and 6.4). We recommend that you also skim the sections about segmentation; while JOS uses the paging hardware for virtual memory and protection, segment translation and segment-based protection cannot be disabled on the x86, so you will need a basic understanding of it. |

现代处理器一般都支持保护模式。在保护模式下虚拟地址通过段页式转换得到物理地址。分段时将段寄存器CS的高13位用作GDT表的索引，低2位表示CPL,还有一位表示索引的是GDT还是LDT.在GDT中找到段首址后加上32位的偏移就得到最终的线性地址了。如下是实验所给的PPT中的分段介绍：

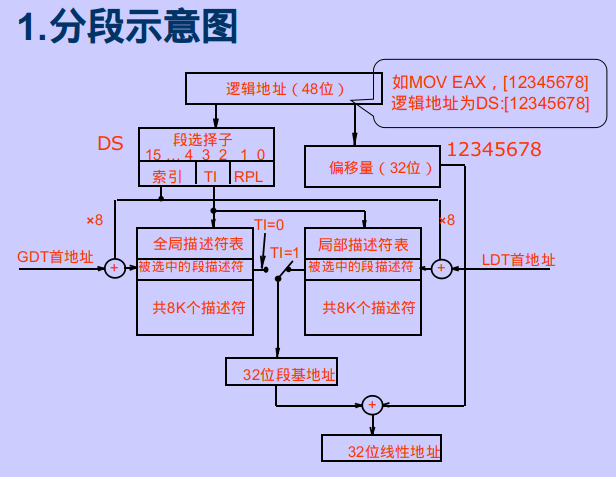


图2-2 分段示意图

分页较分段更复杂些，需要两级索引。首先用CR3寄存器做页目录基址，把线性地址分成10+10+12，高10位索引页目录得到页表基址，中间10位索引页表得到页基址。最后12位用作页内偏移得到物理地址。如下是分页示意图：

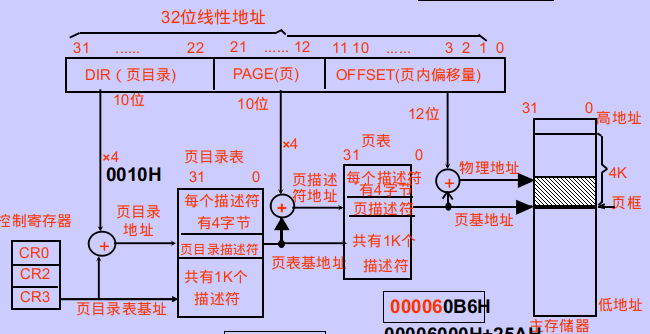


图2-3 分页示意图

###### Virtual, Linear, and Physical Addresses

在X86专业术语中，虚拟地址由段选择符和段内偏移组成，分段后得到的地址是线性地址。线性地址通过分页机制最终将得到RAM中的物理地址。关于分段和分页的具体流程可参见上面的示意图。如下是这3个术语的关系：

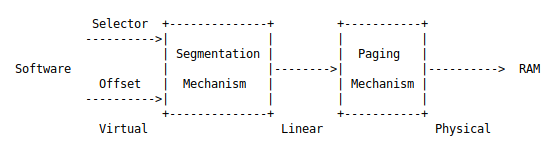


图2-4 虚拟、线性、物理地址的关系

|  |
| --- |
| Exercise 3. While GDB can only access QEMU's memory by virtual address, it's often useful to be able to inspect physical memory while setting up virtual memory. Review the QEMU [monitor commands](https://pdos.csail.mit.edu/6.828/2018/labguide.html" \l "qemu) from the lab tools guide, especially the xp command, which lets you inspect physical memory. To access the QEMU monitor, press Ctrl-a c in the terminal (the same binding returns to the serial console).  Use the xp command in the QEMU monitor and the x command in GDB to inspect memory at corresponding physical and virtual addresses and make sure you see the same data.  Our patched version of QEMU provides an info pg command that may also prove useful: it shows a compact but detailed representation of the current page tables, including all mapped memory ranges, permissions, and flags. Stock QEMU also provides an info mem command that shows an overview of which ranges of virtual addresses are mapped and with what permissions. |

此处是熟悉qemu的调试命令。常用的命令有xp,info mem,info pg。前两个是查看内存，后一个是查看当前page table的信息。

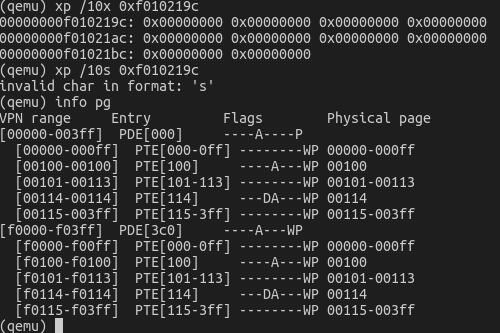


图2-5 qemu 调试命令

###### Page Table Management

本部分将书写一系列例程管理页表：插入、删除从线性地址到物理地址的映射关系以及在必要的时候创建页表页。

|  |
| --- |
| Exercise 4. In the file kern/pmap.c, you must implement code for the following functions.  pgdir\_walk()  boot\_map\_region()  page\_lookup()  page\_remove()  page\_insert()  check\_page(), called from mem\_init(), tests your page table management routines. You should make sure it reports success before proceeding. |

pgdir\_walk接受一个指定的线性地址va，需要我们返回该va对应的page table entry地址以方便后续读写此页表项。实现时可以根据va的高10位查找页目录，如果页目录中不存在相应的页表（PTE\_P为0）且create是1的话就调用page\_alloc分配一页用作页表，否则返回NULL。如果找到或成功创建页表后直接返回对应的页表项地址即可。如下是实现代码：

|  |
| --- |
| pte\_t \*  pgdir\_walk(pde\_t \*pgdir, const void \*va, int create)  {  // Fill this function in  pde\_t\* pgt = pgdir + PDX(va);  if(!(\*pgt & PTE\_P)){ //需要创建一个页表  if(create){  struct PageInfo\* newpg = page\_alloc(ALLOC\_ZERO);//分配一个页表页  if(newpg == NULL) return NULL;  newpg->pp\_ref++;  physaddr\_t pa = page2pa(newpg); //得到该页表所在页的物理地址  pgdir[PDX(va)] = pa | PTE\_U | PTE\_P | PTE\_W;  }  else return NULL;  }  physaddr\_t pte = PTE\_ADDR(\*pgt)+PTX(va)\*sizeof(pte\_t);  return (pte\_t \*)KADDR(pte);  } |

Boot\_map\_region用于将给定va处的size字节映射到物理地址pa处。此映射是静态映射。所谓的静态映射是指不需要分配具体的物理页，只需登记好页目录及页表即可。因此只需调用pgdir\_walk得到页表项地址然后写入相应的物理地址即可。

|  |
| --- |
| static void  boot\_map\_region(pde\_t \*pgdir, uintptr\_t va, size\_t size, physaddr\_t pa, int perm)  {  // Fill this function in  for(size\_t i=0;i<size;i+=PGSIZE){  pte\_t\* pte\_ptr = pgdir\_walk(pgdir,(void\*)(va+i),true);  \*pte\_ptr = (pa+i)|perm|PTE\_P;  }  } |

剩下的三个函数与物理页的插入、查找、移除相关。Page\_insert完成对指定物理页的插入，首先使用pgdir\_walk查看该位置是否已经挂载了其他页，如果有则将其卸载，然后把给定的页安装即可。Page\_lookup用于查找va地址的物理页信息,调用pagedir\_walk即可。Page\_remove则是卸除相关物理页。这三个函数实现代码如下：

|  |
| --- |
| int  page\_insert(pde\_t \*pgdir, struct PageInfo \*pp, void \*va, int perm)  {  // Fill this function in  pte\_t \*pte\_ptr = pgdir\_walk(pgdir,va,true);  if(pte\_ptr == NULL)  return -E\_NO\_MEM;  ++pp->pp\_ref; //fixed a bug,this statement can't be placed at the end.  if((\*pte\_ptr)&PTE\_P)  page\_remove(pgdir,va);  physaddr\_t pa = page2pa(pp);  \*pte\_ptr = pa | perm | PTE\_P;  return 0;  }  struct PageInfo \*  page\_lookup(pde\_t \*pgdir, void \*va, pte\_t \*\*pte\_store)  {  // Fill this function in  \*pte\_store = pgdir\_walk(pgdir,va,true);  if(\*pte\_store==NULL||(\*\*pte\_store&PTE\_P)==0)  return NULL;  return pa2page(\*\*pte\_store);  }  void  page\_remove(pde\_t \*pgdir, void \*va)  {  // Fill this function in  pte\_t \* pte\_ptr;  struct PageInfo \* pginfo\_ptr = page\_lookup(pgdir,va,&pte\_ptr);  if(pginfo\_ptr!=NULL){  page\_decref(pginfo\_ptr);  \*pte\_ptr = 0;  tlb\_invalidate(pgdir,va);  }  } |

##### **Part 3: Kernel Address Space**

JOS将32位线性地址空间分割成用户环境和内核环境两部分。这两部分的分界线由宏ULIM表示，内核保留了高256MB的虚拟地址空间,这解释了为什么将kernel的VMA设置为0xf0100000。关于JOS的内存布局可以详细参考inc/ memlayout.h文件中的布局图。

###### Permissions and Fault Isolation

因为内核以及用户的内存存在于每个进程的地址空间中，必须利用权限位使用户程序运行在用户地址空间。否则可能会存在用户程序修改内核数据或者一个进程窃取另外一个进程私有信息等bug。需要牢记写权限位既可影响用户程序也能影响内核程序。

一般而言，用户进程没有权限访问ULIM之上的内存，而内核进程却可对该区域进行读写。对于地址空间[UTOP,ULIM)，内核和用户进程拥有相同的只读权限。该部分存放页表、pages数组和进程控制块env数组。在UTOP之下的空间则全是用户空间了。用户对此具有读写权限。

###### Initializing the Kernel Address Space

现在我们需要完成UTOP之上地址空间的设置，即内核空间的设置。

|  |
| --- |
| Exercise 5. Fill in the missing code in mem\_init() after the call to check\_page().  Your code should now pass the check\_kern\_pgdir() and check\_page\_installed\_pgdir() checks. |

在pmap.c文件中，根据注释的提示可知需要完成3块地址的映射。分别是将pages数组映射到虚拟地址UPAGES,将bootstack所在的内核栈映射到虚拟地址KSTACKTOP-KSTSIZE以及将整个内核(从kernbase到0xffffffff）的256MB映射到物理地址低256MB。通过调用boot\_alloc\_region可以很方便地实现上述功能。需要提及的是此处不能使用page\_insert。由于page\_insert是动态映射，即从page\_free\_list中取出一页映射到某个虚拟地址，而我们需要映射到的物理地址都是处于in use状态，不在page\_free\_list中，因此只能使用静态映射登记号页目录、页表即可。

|  |
| --- |
| boot\_map\_region(kern\_pgdir,UPAGES,PTSIZE,PADDR(pages),PTE\_U);  boot\_map\_region(kern\_pgdir,KSTACKTOP-KSTKSIZE,KSTKSIZE,PADDR(bootstack),PTE\_W);  boot\_map\_region(kern\_pgdir,KERNBASE,0xffffffff-KERNBASE,0,PTE\_W); |

#### **3 实验问题**

1. Assuming that the following JOS kernel code is correct, what type should variable x have, uintptr\_t or physaddr\_t?

|  |
| --- |
| mystery\_t x;  char\* value = return\_a\_pointer();  \*value = 10;  x = (mystery\_t) value; |

X是uintptr\_t类型。因为return\_a\_pointer()所返回的值是其运行地址空间中的地址，即虚拟地址。因此x也必定是uintptr\_t类型，否则和代码正确的假设矛盾。

1. What entries (rows) in the page directory have been filled in at this point? What addresses do they map and where do they point? In other words, fill out this table as much as possible:

|  |  |  |  |
| --- | --- | --- | --- |
| Entry | Base VA | Points to (logically): | location |
| 1023 | 0xffc00000 | Page table for top 4MB of phys memory | kernel |
| 1022 | 0xff800000 | Page table for 4MB of physmemory |
| ... | ... | Page table for 4MB of physmemory |
| 960 | 0xf0000000 | Page table for bottom 4MB of phys memory |
| 959 | 0xefc00000 | KSTSIZE(8\*PGSIZE) of kernel stack | KSTK |
| ? | ? | ? | ? |
| 957 | 0xef400000 | PGSIZE of Kern\_pgdir | UVPT |
| 956 | 0xef000000 | PTSIZE(4MB) of pages array | UPAGES |
| ? | ? | ? | ? |

在pmap.c中找到如下映射关系即可填写出上表：

|  |
| --- |
| kern\_pgdir[PDX(UVPT)] = PADDR(kern\_pgdir) | PTE\_U | PTE\_P;  boot\_map\_region(kern\_pgdir,UPAGES,PTSIZE,PADDR(pages),PTE\_U);  boot\_map\_region(kern\_pgdir,KSTACKTOP-KSTKSIZE,KSTKSIZE,PADDR(bootstack),PTE\_W);  boot\_map\_region(kern\_pgdir,KERNBASE,0xffffffff-KERNBASE,0,PTE\_W); |

1. 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?

这是通过检查CPL来实现的。CS寄存器由段选择子和特权级位组成。0表示内核态，3表示用户态。当用户程序访问内核空间地址时，首先会进行虚拟地址到线性地址再到物理地址的转换。因此需要查询页目录和页表，而页目录和页表描述符的低12位中有权限标记，当处理器发现权限不符时会引发缺页异常。

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

4GB,因为32位地址的寻址空间为4GB.

1. 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?

如果拥有最大的4GB内存，那么用来管理内存的空间开销包括：

1. 内核页目录（4KB)。
2. 1K个页表（1K\*4KB=4M)
3. Pages数组（npages==4G/4K==1M,需要1M\*sizeof(struct PageInfo).

因此总的空间开销至少（没包括用户页目录页表）为4K+4M+1M\*sizeof(struct PageInfo)。

1. Revisit the page table setup in kern/entry.S and kern/entrypgdir.c. Immediately after we turn on paging, EIP is still a low number (a little over 1MB). At what point do we transition to running at an EIP above KERNBASE? What makes it possible for us to continue executing at a low EIP between when we enable paging and when we begin running at an EIP above KERNBASE? Why is this transition necessary?

在执行完如下jmp \*%eax后，kernel开始运行在KERNBASE之上的虚拟地址。

|  |
| --- |
| mov $relocated, %eax  jmp \*%eax |

虽然将CR0的PE位打开后处理器仍运行在低地址，但是却能正确执行指令的原因是我们在entrypgdir.s中将低4M虚拟地址空间也映射到了低4M的物理地址空间。这是有必要的，因为如果不这么映射。那么在将分页enable之后，eip就会被映到其他地址，也就不会正确执行后面的指令了。

### Lab3 User **Environments**

#### **1 实验目的**

#### **2 实验内容**

##### **2.1 Part 1: User Environment and Exception Handling**

##### **2.2 Part 2: Page Fault, Breakpoints Exception, and System Call**

#### **3 实验问题**

### Lab4 **Preemptive** Multitasking

#### **1 实验目的**

#### **2 实验内容**

##### **2.1 Part 1: Multiprocessor Support and Cooperative Multitasking**

##### **2.2 Part 2: Copy-on-Write Fork**

##### **2.3 Part 3: Preemptive Multitasking and Inter-Process communication(IPC)**

#### **3 实验问题**