

**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 5](#_Toc1250404169)

[1 实验目的 5](#_Toc757159588)

[2 实验内容 5](#_Toc376929113)

[2.1 Part 1: PC Bootstrap 5](#_Toc2074243663)

[2.1.1 Getting Started with x86 assembly 5](#_Toc295919820)

[2.1.2 Simulating the x86 6](#_Toc660035081)

[2.1.3 The PC's Physical Address Space 6](#_Toc615193219)

[2.1.4 The ROM BIOS 8](#_Toc1109115755)

[2.2 Part 2: The Boot Loader 10](#_Toc437273114)

[2.2.1 Loading the Kernel 13](#_Toc1550792079)

[2.3 Part 3: The Kernel 18](#_Toc1500683989)

[Lab2 Memory Management 31](#_Toc1063311119)

[1 实验目的 31](#_Toc1333207272)

[2 实验内容 31](#_Toc167147757)

[2.1 Part 1: Physical Page Management 31](#_Toc1184939597)

[2.2 Part 2: Virtual Memory 34](#_Toc1424625108)

[2.2.1 Virtual, Linear, and Physical Addresses 36](#_Toc34301048)

[2.2.2 Page Table Management 37](#_Toc330220868)

[2.3 Part 3: Kernel Address Space 40](#_Toc1982787601)

[2.3.1 Permissions and Fault Isolation 40](#_Toc1092937015)

[2.3.2 Initializing the Kernel Address Space 40](#_Toc1439336623)

[3 实验问题 41](#_Toc272577067)

[Lab3 User Environments 44](#_Toc496245446)

[1 实验目的 44](#_Toc792536964)

[2 实验内容 44](#_Toc985544069)

[2.1.1 Environment State 44](#_Toc704740031)

[2.1.2 Allocating the Environments Array 45](#_Toc193833516)

[2.1.3 Creating and Running Environments 45](#_Toc243527936)

[2.1.4 Handling Interrupts and Exceptions 50](#_Toc799552707)

[2.2.1 Handling Page Faults & Breakpoint Exception 54](#_Toc1620096077)

[2.2.2 System calls 55](#_Toc25697653)

[2.2.3 User-mode startup 57](#_Toc1089008772)

[2.2.4 Page faults and memory protection 58](#_Toc233691985)

[Lab4 Preemptive Multitasking 65](#_Toc1761879192)

[1 实验目的 65](#_Toc616568312)

[2 实验内容 65](#_Toc1072288128)

[2.1.1 Multiprocessor Support 65](#_Toc165379325)

[2.1.2 Application Processor Bootstrap 67](#_Toc981090228)

[2.1.3 Per-CPU State and Initialization 68](#_Toc82860954)

[2.1.4 Locking 70](#_Toc1736060136)

[2.1.5 Round-Robin Scheduling 71](#_Toc272943203)

[2.1.6 System Calls for Environment Creation 73](#_Toc84821934)

[2.2.1 Setting the Page Fault Handler 75](#_Toc1065480167)

[2.2.2 Invoking the User Page Fault Handler 76](#_Toc1340982091)

[2.2.3 User-mode Page Fault Entrypoint 78](#_Toc1259313683)

[2.2.4 Implementing Copy-on-Write Fork 79](#_Toc572107469)

[2.3.1 Clock Interrupts and Preemption 83](#_Toc543414217)

[2.3.2 Inter-Process communication (IPC) 85](#_Toc1999372491)

[3 实验问题 89](#_Toc16026646)

### 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：

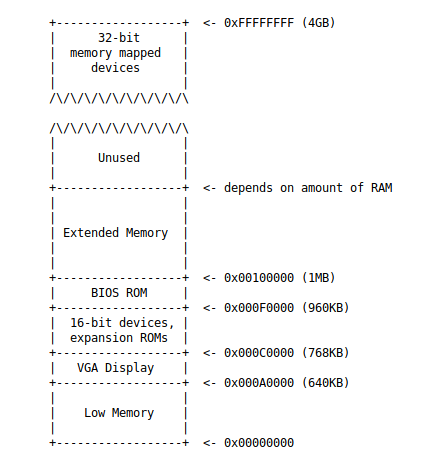


图1-1 PC物理地址空间布局

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：

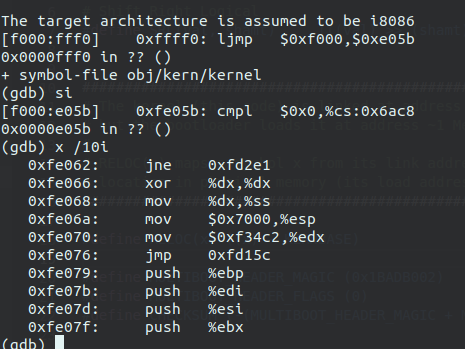


图1-2 BIOS的堆栈设置

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。

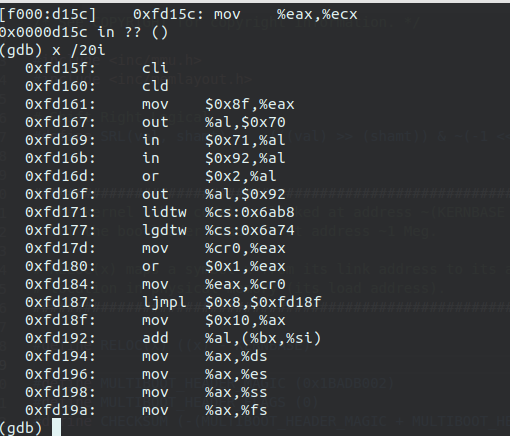


图1-3 BIOS进入保护模式，设置IDT,GDT

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。

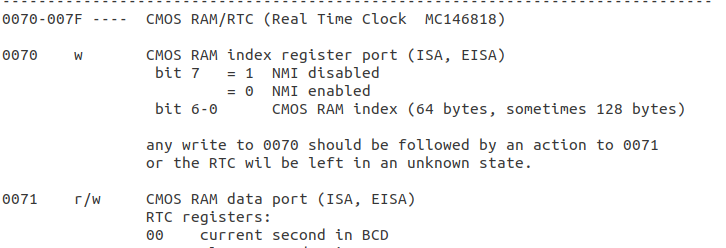


图1-4 70端口参数示意

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：

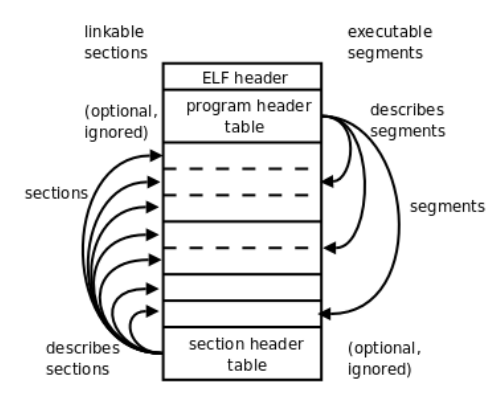


图1-4 ELF文件的两种视图

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。

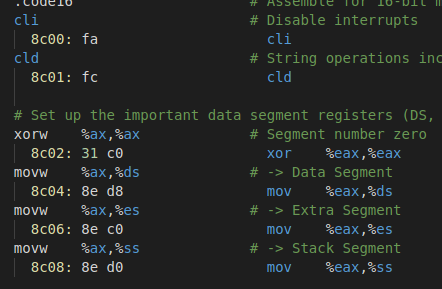


图1-5 修改后的VMA

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。

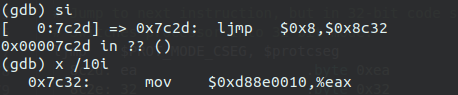


图1-6 第一条出错指令

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。

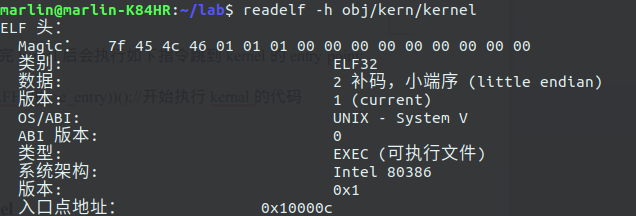


图1-7 kernel的entry point

|  |
| --- |
| 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：

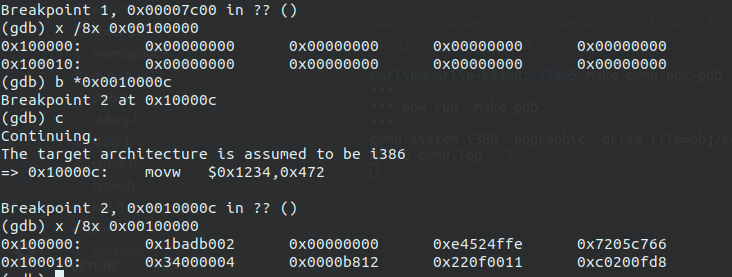


图1-8 两次输出结果

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：

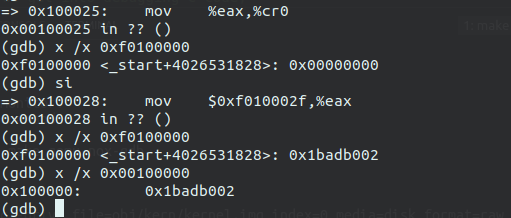


图1-9 分页前后0xf0100000处的内存值

###### 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。

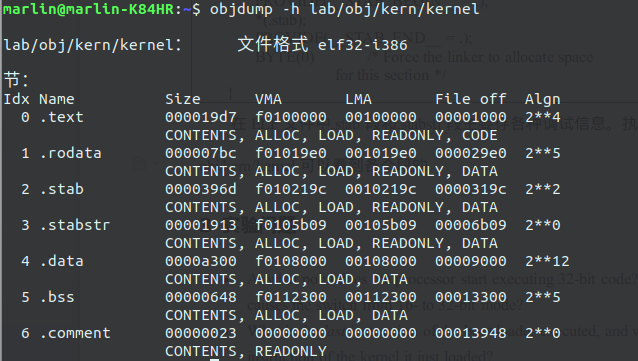


图1-11 ELF段信息

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。

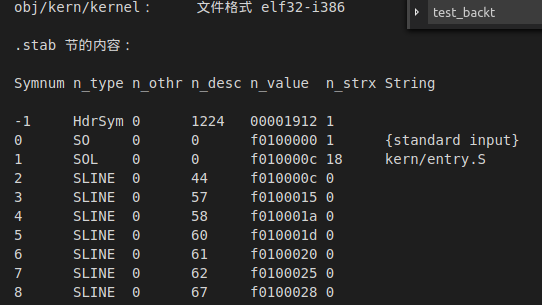


图1-12 .stab段信息

.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。

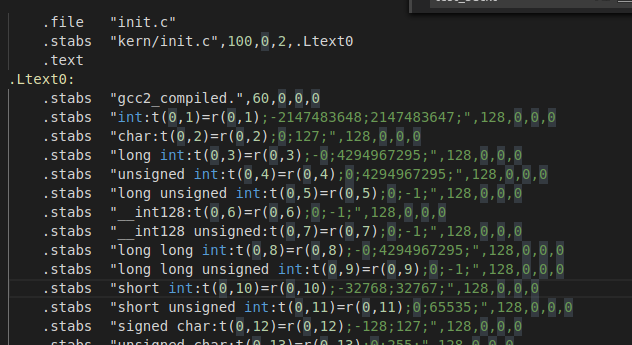


图1-13 init.s中的调试信息

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。

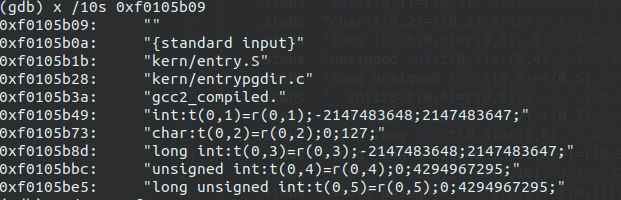


图1-14 .stabstr段LVM处内存值

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。

执行自动评分脚本：

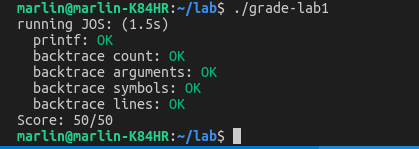


图1-15 lab1测试结果

#### 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。

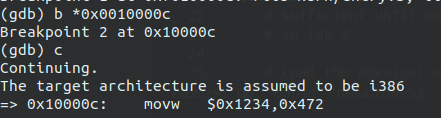


图1-15 kernel的第一条指令

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：

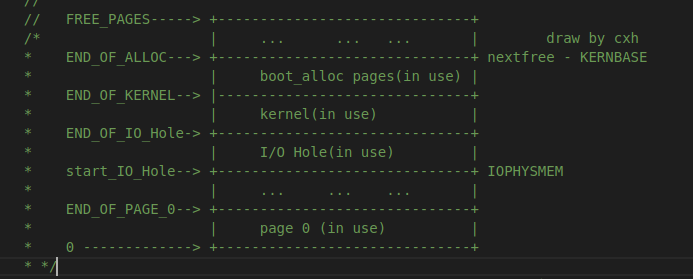


图2-1 物理页状态图

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：

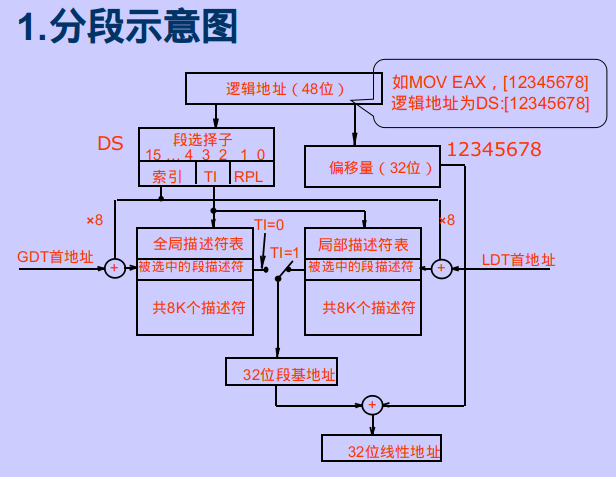


图2-2 分段示意图

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：

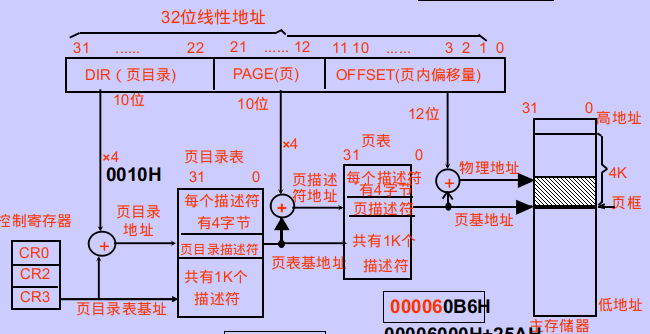


图2-3 分页示意图

###### 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：

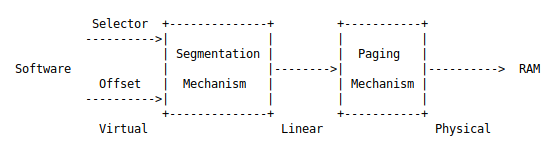


图2-4 虚拟、线性、物理地址的关系

|  |
| --- |
| 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。

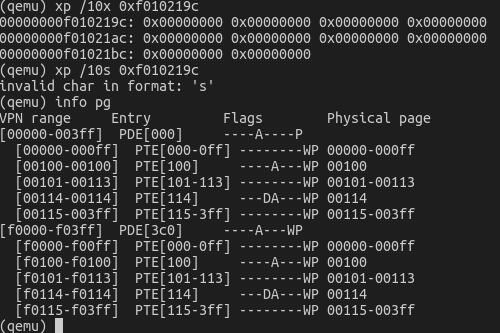


图2-5 qemu 调试命令

###### 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); |

最后执行评分脚本：

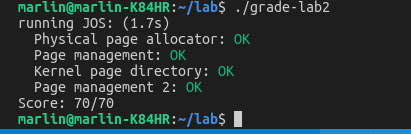


图2-6 lab2测试结果

#### **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**

本次实验需要实现保护模式下单进程的创建运行。主要包括一下操作要求：

1. 掌握如何设置数据结构跟踪用户进程，并加载程序镜像文件使其投入运行。
2. 理解操作系统创建、管理各个进程的方式和手段。
3. 实现系统调用。明白中断、异常所触发的中断处理流程。

#### **2 实验内容**

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

###### Environment State

在kern/env.c中包含了几个与进程相关的全局变量：

|  |
| --- |
| struct Env \*envs = NULL; // All environments  struct Env \*curenv = NULL; // The current env  static struct Env \*env\_free\_list; // Free environment list |

Envs是进程控制块数组，其大小由NENV决定，在pmap.c中使用boot\_alloc为其分配内存。Curenv记录了当前运行的进程控制块，而env\_free\_list则维护所有空闲的env链表。有关于进程控制块env的结构如下：

|  |
| --- |
| struct Env {  struct Trapframe env\_tf; // Saved registers  struct Env \*env\_link; // Next free Env  envid\_t env\_id; // Unique environment identifier  envid\_t env\_parent\_id; // env\_id of this env's parent  enum EnvType env\_type; // Indicates special system environments  unsigned env\_status; // Status of the environment  uint32\_t env\_runs; // Number of times environment has run  // Address space  pde\_t \*env\_pgdir; // Kernel virtual address of page dir  }; |

其主要成员的作用在注释中均给出了详细说明，无须过多解释。需要引起关注的是env\_tf,它记录了该进程上次退出时的现场（各寄存器值）或是第一次运行时的初始寄存器值。当此进程第一次或再次投入运行时会利用env\_tf恢复现场继续运行。

###### Allocating the Environments Array

本部分需要做的工作是为envs数组分配空间并完成虚拟地址映射。

|  |
| --- |
| Exercise 1. Modify mem\_init() in kern/pmap.c to allocate and map the envs array. This array consists of exactly NENV instances of the Env structure allocated much like how you allocated the pages array. Also like the pages array, the memory backing envs should also be mapped user read-only at UENVS (defined in inc/memlayout.h) so user processes can read from this array.  You should run your code and make sure check\_kern\_pgdir() succeeds. |

在lab2中已经完成过类似的操作，用boot\_alloc分配空间、boot\_map\_region建立映射关系即可。此处直接贴代码：

|  |
| --- |
| envs = (struct Env\*)boot\_alloc(NENV\*sizeof(struct Env)); //分配空间memset(envs,0,sizeof(struct Env)\*NENV); //清零  boot\_map\_region(kern\_pgdir,UENVS,PTSIZE,PADDR(envs),PTE\_U);//内存映射 |

###### Creating and Running Environments

到目前为止，JOS依然运行在内核态，为了使得JOS能运行用户进程。需要完成部分关于用户进程创建以及运行的代码。

由于尚未实现文件系统，无法从磁盘读取用户程序，因此这里采用将用户二进制文件内嵌kernel并由kernel加载出文件并执行。这一执行机制依赖于链接器ld的设置，当指定ld的参数--format=binary时，会将其后的文件原封不动地加载到.data段。并自动产生符号\_binary\_xxx\_start, \_binary\_xxx\_end用于指示该文件的起始和结束地址。在obj/kern/kern.sym中显示了所有符号的地址以及所在段。

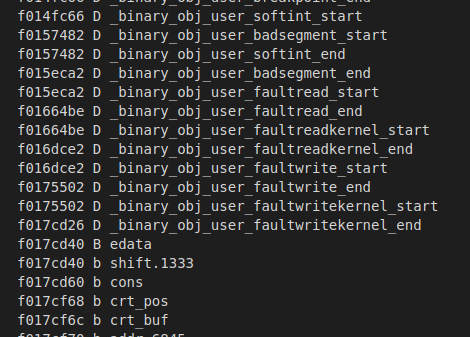


图3-1 内嵌kernel的各个用户程序

第一列是符号所代表的偏移地址，第二列表示所在段如D表示.data,B表示.bss。第三列是符号名称，上图显示所有用户程序被加载到.data段末尾并紧邻.bss段。至此，准备工作基本完成。

|  |
| --- |
| Exercise 2. In the file env.c, finish coding the following functions:  env\_init()  Initialize all of the Env structures in the envs array and add them to the env\_free\_list. Also calls env\_init\_percpu, which configures the segmentation hardware with separate segments for privilege level 0 (kernel) and privilege level 3 (user).  env\_setup\_vm()  Allocate a page directory for a new environment and initialize the kernel portion of the new environment's address space.  region\_alloc()  Allocates and maps physical memory for an environment  load\_icode()  You will need to parse an ELF binary image, much like the boot loader already does, and load its contents into the user address space of a new environment.  env\_create()  Allocate an environment with env\_alloc and call load\_icode to load an ELF binary into it.  env\_run()  Start a given environment running in user mode.  As you write these functions, you might find the new cprintf verb %e useful -- it prints a description corresponding to an error code. For example,  r = -E\_NO\_MEM;  panic("env\_alloc: %e", r);  will panic with the message "env\_alloc: out of memory". |

为完成上述几个函数，我们对其稍加分析：

1. env\_init用来初始化envs数组并将所有进程控制块加入空闲链表。只需遍历一遍envs数组即可。
2. env\_setup\_vm为用户进程进行必要的地址映射，主要是为其分配页目录并设置有关页目录项，如内核部分的映射。可以复制kern\_pgdir的部分表项完成。
3. region\_alloc分配指定大小的物理内存给进程e,并将这部分映射到va地址起始处。直接调用page\_alloc,page\_insert即可。
4. load\_icode是需要特别注意的，该函数将内嵌kernel的用户ELF文件加载到用户进程的指定虚拟地址空间。类似lab1中bootmain()函数，我们先对ELF header进行分析找到program header table的首尾地址，之后循环读取并加载每个program header即可。由于.bss段需要置0，为了达到这一目的，可以先用lcr3切换到用户页目录然后使用memset将该段置0。细节参见源代码。
5. env\_create创建一个用户进程并设置好用户程序。需调用env\_alloc以及load\_icode。
6. env\_run对待运行的进程控制块成员env\_status以及env\_run进行更新，设置好eip使其指向用户程序的entry\_point。最后切换页目录将其投入运行。

如下是这部分练习的源代码：

|  |
| --- |
| void  env\_init(void)  {  // Set up envs array  // LAB 3: Your code here.  env\_free\_list=NULL;  for(int i=NENV-1;i>=0;i--){//fix a bug  envs[i].env\_id = 0;  envs[i].env\_link = env\_free\_list;  env\_free\_list = &envs[i];  }  // Per-CPU part of the initialization  env\_init\_percpu();  }  static int  env\_setup\_vm(struct Env \*e)  {  int i;  struct PageInfo \*p = NULL;  // Allocate a page for the page directory  if (!(p = page\_alloc(ALLOC\_ZERO)))  return -E\_NO\_MEM;  // LAB 3: Your code here.  p->pp\_ref++;  e->env\_pgdir =(pde\_t\*)page2kva(p);  uint32\_t idx = PDX(UTOP);  memcpy(&e->env\_pgdir[idx],&kern\_pgdir[idx],PGSIZE - idx\*sizeof(pde\_t));  e->env\_pgdir[PDX(UVPT)] = PADDR(e->env\_pgdir) | PTE\_P | PTE\_U;  return 0;  }  static void  region\_alloc(struct Env \*e, void \*va, size\_t len)  {  // LAB 3: Your code here.  va = ROUNDDOWN(va,PGSIZE);  for(size\_t i=0;i<len;i+=PGSIZE){  struct PageInfo\* newpg = page\_alloc(0);  if(newpg == NULL) panic("region\_alloc: page\_alloc failed!\n");  if(page\_insert(e->env\_pgdir,newpg,va,PTE\_U|PTE\_W) < 0)  panic("region\_alloc: page\_insert failed!\n");  va+=PGSIZE;  }  }  static void  load\_icode(struct Env \*e, uint8\_t \*binary)  {  // LAB 3: Your code here.  struct Elf\* pelf = (struct Elf\*)binary;  if(pelf->e\_magic != ELF\_MAGIC)  panic("load\_icode: ELF file broken!\n");  struct Proghdr\* ph = (struct Proghdr\*)(binary + pelf->e\_phoff);  struct Proghdr\* eph = ph + pelf->e\_phnum;  lcr3(PADDR(e->env\_pgdir));//切换页目录，没有想到ORZ  for(;ph<eph;ph++){  if(ph->p\_type == ELF\_PROG\_LOAD){  region\_alloc(e,(void\*)ph->p\_va,ph->p\_memsz);//分配用户程序空间  memset((void\*)ph->p\_va,0,ph->p\_memsz); //全部初始化为0，这样.bss段就自动初始化了  memcpy((void\*)ph->p\_va,(void\*)(binary+ph->p\_offset),ph->p\_filesz);//填充其他段  }  }  e->env\_tf.tf\_eip = pelf->e\_entry; //设置好该程序的入口地址  lcr3(PADDR(kern\_pgdir));//切换回kernel的页目录  // Now map one page for the program's initial stack  // at virtual address USTACKTOP - PGSIZE.  // LAB 3: Your code here.  region\_alloc(e,(void\*)(USTACKTOP - PGSIZE),PGSIZE);  }  void  env\_create(uint8\_t \*binary, enum EnvType type)  {  // LAB 3: Your code here.  struct Env\* newenv;  int stat;  if((stat=env\_alloc(&newenv,0)))  panic("env\_create: %e!\n",stat);  load\_icode(newenv,binary);  newenv->env\_type = type;  }  void  env\_run(struct Env \*e)  {  // LAB 3: Your code here.  if(curenv!=NULL && curenv->env\_status == ENV\_RUNNING)  curenv->env\_status = ENV\_RUNNABLE;  curenv = e;  e->env\_status = ENV\_RUNNING;  e->env\_runs++;  lcr3(PADDR(e->env\_pgdir));  env\_pop\_tf(&e->env\_tf);//恢复各个寄存器的值  // panic("env\_run not yet implemented");  } |

###### Handling Interrupts and Exceptions

|  |
| --- |
| Exercise 3. Read [Chapter 9, Exceptions and Interrupts](https://pdos.csail.mit.edu/6.828/2018/readings/i386/c09.htm) in the [80386 Programmer's Manual](https://pdos.csail.mit.edu/6.828/2018/readings/i386/toc.htm) (or Chapter 5 of the [IA-32 Developer's Manual](https://pdos.csail.mit.edu/6.828/2018/readings/ia32/IA32-3A.pdf)), if you haven't already. |

中断分为内部中断以及外部中断，一般称外部中断为中断，内部中断为异常。两者都是指处理器在执行程序时接收到中断信号，停止当前任务的执行，转去处理中断服务程序的过程。中断一般由外部设备触发，如I/O操作。而异常则通常由运行时的代码引起。可以进一步将中断细分为可屏蔽中断以及不可屏蔽中断，异常分为处理器保护异常以及可编程异常（中断软指令）。

中断的实现需要两种机制的支持：中断向量表和任务状态段（TSS段）

IDT表有256个中断描述符，通过中断号进行索引。每个中断描述符由8个字节组成，主要包括中断服务程序的段选择子（CS）以及偏移。之后就可以利用CS：EIP的逻辑地址找到服务程序的入口地址。如下是中断程序的索引流程：

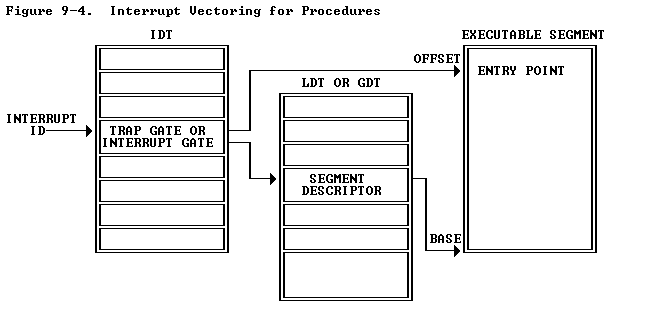


图3-2 中断服务程序索引流程

在中断处理过程中，处理器需要一定空间存放中断源的状态，例如原始程序的EIP,CS寄存器信息以便中断返回后能继续执行后续指令。同时又要将这些信息保护起来以防遭到用户态其他程序的破坏。

鉴于这些原因，中断发生时首先会将当前的特权级从用户态变成内核态，然后将现场数据压入一个特定的内核栈保护。用于指示该内核栈地址的数据结构就是TSS。其成员SS0是该内核栈的段选择子，ESP0是其偏移。

中断产生后,处理器自动将寄存器SS,ESP,FLAGS,CS,EIP以及一个可选的error code放到TSS描述的内核栈中。然后利用中断号找到相应的描述符，从中取出服务程序的段选择子和偏移来设置CS,EIP并设置ESP,SS指向内核栈。这样就使得处理器进入中断服务程序了。

|  |
| --- |
| Exercise 4. Edit trapentry.S and trap.c and implement the features described above. The macros TRAPHANDLER and TRAPHANDLER\_NOEC in trapentry.S should help you, as well as the T\_\* defines in inc/trap.h. You will need to add an entry point in trapentry.S (using those macros) for each trap defined in inc/trap.h, and you'll have to provide \_alltraps which the TRAPHANDLER macros refer to. You will also need to modify trap\_init() to initialize the idt to point to each of these entry points defined in trapentry.S; the SETGATE macro will be helpful here.  Your \_alltraps should:   1. push values to make the stack look like a struct Trapframe 2. load GD\_KD into %ds and %es 3. pushl %esp to pass a pointer to the Trapframe as an argument to trap() 4. call trap (can trap ever return?)   Consider using the pushal instruction; it fits nicely with the layout of the struct Trapframe.  Test your trap handling code using some of the test programs in the user directory that cause exceptions before making any system calls, such as user/divzero. You should be able to get make grade to succeed on the divzero, softint, and badsegment tests at this point. |

结合实验材料的说明，我们需要在trapentry.s中完成一些中断服务程序的汇编入口，关于需要实现哪些中断入口可参见inc/trap.h中定义的中断号。

在查找并执行断处理函数之前，处理器已经自动将部分寄存器压栈。

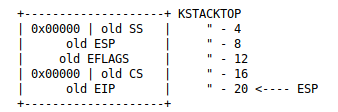


图3-3 处理器自动压栈

需要我们完成的工作是继续压栈使得KSTACKTOP构成一个trapframe。trapframe结构记录了一个进程的现场信息，由诸多寄存器值构成。由于有的中断会产生error code,需要根据情况调用trapentry.s中的两个宏TRAPHANDER\_ NOEC以及TRAPHANDLER。各中断是否会产生error code在PPT中均有标识。

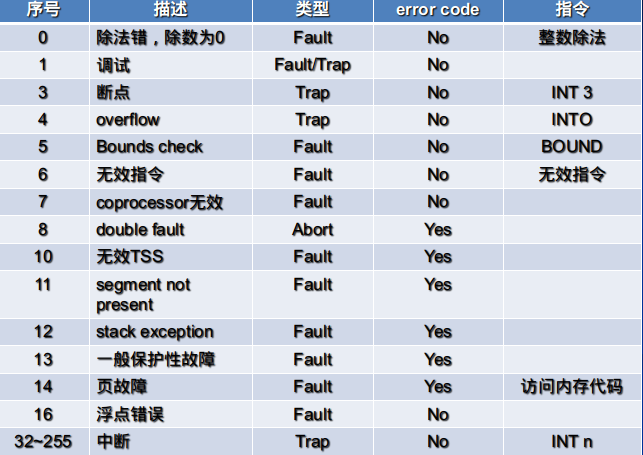


图3-4各中断信息图

根据是否产生error code我们可以调用不同的宏将中断号以及error code压栈。实现如图所示：

|  |
| --- |
| TRAPHANDLER\_NOEC(trap0, 0);  TRAPHANDLER\_NOEC(trap1, 1);  TRAPHANDLER\_NOEC(trap3, 3);  TRAPHANDLER\_NOEC(trap4, 4);  TRAPHANDLER\_NOEC(trap5, 5);  TRAPHANDLER\_NOEC(trap6, 6);  TRAPHANDLER\_NOEC(trap7, 7);  TRAPHANDLER(trap8, 8);  TRAPHANDLER(trap10, 10)  TRAPHANDLER(trap11, 11)  TRAPHANDLER(trap12, 12)  TRAPHANDLER(trap13, 13)  TRAPHANDLER(trap14, 14)  TRAPHANDLER\_NOEC(trap16, 16)  TRAPHANDLER\_NOEC(t\_syscall, T\_SYSCALL) |

完成此步骤后这两个宏均会跳到\_alltraps标号处将所有中断共同的寄存器参数入栈。只需根据提示编写对应的汇编指令即可。直接给出代码：

|  |
| --- |
| \_alltraps:  pushl %ds;  pushl %es;  pushal  movl $GD\_KD,%eax;  movw %ax,%ds;  movw %ax,%es;  pushl %esp;//作为trap()的参数  call trap; |

现在栈中恰好构成了一个struct trapframe，并且转去执行trap.c中的trap函数，该函数会利用该trapframe的信息调用合适的处理函数处理此中断，最后使用env\_run 完成中断返回。

在trap.c中我们需要完成trap\_init函数。该函数将相关中断服务程序安装到IDT中，实验为我们提供了一个有用的宏SETGATE(gate, istrap, sel, off, dpl)。直接调用即可完成安装。

|  |
| --- |
| extern void trap0();  extern void trap1();  extern void trap3();  extern void trap4();  extern void trap5();  extern void trap6();  extern void trap7();  extern void trap8();  extern void trap10();  extern void trap11();  extern void trap12();  extern void trap13();  extern void trap14();  extern void trap16();  extern void t\_syscall();  SETGATE(idt[0],0,GD\_KT,trap0,0);  SETGATE(idt[1],0,GD\_KT,trap1,0);  SETGATE(idt[3],0,GD\_KT,trap3,3);  SETGATE(idt[4],0,GD\_KT,trap4,0);  SETGATE(idt[5],0,GD\_KT,trap5,0);  SETGATE(idt[6],0,GD\_KT,trap6,0);  SETGATE(idt[7],0,GD\_KT,trap7,0);  SETGATE(idt[8],0,GD\_KT,trap8,0);  SETGATE(idt[10],0,GD\_KT,trap10,0);  SETGATE(idt[11],0,GD\_KT,trap11,0);  SETGATE(idt[12],0,GD\_KT,trap12,0);  SETGATE(idt[13],0,GD\_KT,trap13,0);  SETGATE(idt[14],0,GD\_KT,trap14,0);  SETGATE(idt[16],0,GD\_KT,trap16,0);  SETGATE(idt[T\_SYSCALL],0,GD\_KT,t\_syscall,3); |

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

###### Handling Page Faults & Breakpoint Exception

|  |
| --- |
| Exercise 5. Modify trap\_dispatch() to dispatch page fault exceptions to page\_fault\_handler(). You should now be able to get make grade to succeed on the faultread, faultreadkernel, faultwrite, and faultwritekernel tests. If any of them don't work, figure out why and fix them. Remember that you can boot JOS into a particular user program using make run-*x* or make run-*x*-nox. For instance, make run-hello-nox runs the *hello* user program. |
| Exercise 6. Modify trap\_dispatch() to make breakpoint exceptions invoke the kernel monitor. You should now be able to get make grade to succeed on the breakpoint test. |

Trap\_dispatch接收一个struct trapframe指针，可以根据trapframe的tf\_trapno判断中断号，进而执行特定的中断处理函数。具体来说，如果中断号是T\_PGFLT,执行page\_fault\_handler函数，若是T\_BRKPT，则调用monitor()即可。

|  |
| --- |
| switch (tf->tf\_trapno)  {  case T\_PGFLT:  page\_fault\_handler(tf);  return;  case T\_BRKPT:  monitor(tf);  return;  default:  break;  } |

###### System calls

用户进程通过系统功能调用进入内核态完成指定的操作，通常这些操作只有内核才能完成。系统功能调用产生时，处理器从用户态进入内核态并保存用户进程的相关状态信息。待内核处理完之后再返回用户进程。

进行系统调用时，应用会将功能号以及调用参数保存到相关寄存器中，其中功能号被放到eax寄存器，参数（最多5个）依次放入edx,ecx,ebx,edi和esi。Kernel处理完之后会将返回结果放到eax中。

|  |
| --- |
| Exercise 7. Add a handler in the kernel for interrupt vector T\_SYSCALL. You will have to edit kern/trapentry.S and kern/trap.c's trap\_init(). You also need to change trap\_dispatch() to handle the system call interrupt by calling syscall() (defined in kern/syscall.c) with the appropriate arguments, and then arranging for the return value to be passed back to the user process in %eax. Finally, you need to implement syscall() in kern/syscall.c. Make sure syscall() returns -E\_INVAL if the system call number is invalid. You should read and understand lib/syscall.c (especially the inline assembly routine) in order to confirm your understanding of the system call interface. Handle all the system calls listed in inc/syscall.h by invoking the corresponding kernel function for each call.  Run the user/hello program under your kernel (make run-hello). It should print "hello, world" on the console and then cause a page fault in user mode. If this does not happen, it probably means your system call handler isn't quite right. You should also now be able to get make grade to succeed on the testbss test. |

练习的第一步是安装处理系统功能调用的入口函数以及设置IDT表项，这在exercise 4中实现过，此处不再赘述。接下来是在trap\_dispatch中添加处理system call的分支调用。结合练习中的说明可写出如下代码：

|  |
| --- |
| case T\_SYSCALL:  tf->tf\_regs.reg\_eax = syscall(tf->tf\_regs.reg\_eax,  tf->tf\_regs.reg\_edx,  tf->tf\_regs.reg\_ecx,  tf->tf\_regs.reg\_ebx,  tf->tf\_regs.reg\_edi,  tf->tf\_regs.reg\_esi);  return ; |

代码中的syscall是内核的系统功能调用处理函数，区别于用户态的syscall函数。

用户态的syscall使用内联汇编int 0x30指令显示进行系统调用。当处理器执行该指令时会触发中断并调用相应的中断处理程序，该过程包括IDT的检索以及内核栈的切换。通过中断描述符找到中断处理程序的段选择子和偏移、并跳到TSS中指定的内核栈后开始执行中断程序的入口函数。当处理程序执行到trap\_ dispatch时会进入上面的分支代码进行相应的处理，处理完后通过env\_run返回用户进程。如下是用户系统功能调用的处理流程：

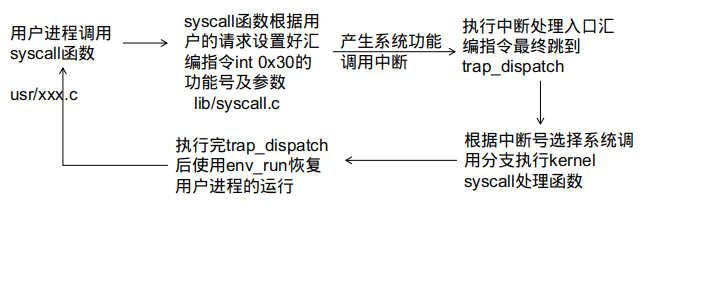


图3-5 系统调用处理流程

Kernel的syscall函数根据系统功能调用号（位于eax寄存器）调用相应的系统功能函数即可，可以用一个switch进行分支。代码如下：

|  |
| --- |
| int32\_t  syscall(uint32\_t syscallno, uint32\_t a1, uint32\_t a2, uint32\_t a3, uint32\_t a4, uint32\_t a5)  {  // Call the function corresponding to the 'syscallno' parameter.  // Return any appropriate return value.  // LAB 3: Your code here.  switch (syscallno) {  case SYS\_cgetc:  return sys\_cgetc();  case SYS\_cputs:  sys\_cputs((char\*)a1,a2);  return 0;  case SYS\_env\_destroy:  return sys\_env\_destroy(a1);  case SYS\_getenvid:  return sys\_getenvid();  default:  return -E\_INVAL;  }  // panic("syscall not implemented");  } |

现在usr/hello.c可以正常地通过系统功能调用打印字符了。

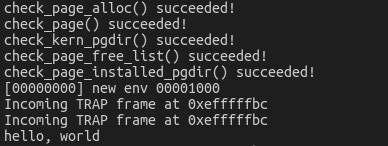


图3-6 make run-hello 结果

###### User-mode startup

Kernel 一旦启动env\_run就开始执行用户程序。在load\_icode中我们将ELF文件的entry\_point赋给了进程控制块的eip寄存器。所以env\_run可以顺利进入用户进程执行。

用户程序由lib/entry.s,lib/libmain.c以及usr/xxx.c三个文件编译链接生成，lib/entry.s以及lib/main.c是通用接口。Entry.s会调用libmain函数，后者接着调用用户主程序umain。因为用户程序无法直接使用指针curenv，因此需要我们通过设置thisenv来代替其功能。

|  |
| --- |
| Exercise 8. Add the required code to the user library, then boot your kernel. You should see user/hello print "hello, world" and then print "i am environment 00001000". user/hello then attempts to "exit" by calling sys\_env\_destroy() (see lib/libmain.c and lib/exit.c). Since the kernel currently only supports one user environment, it should report that it has destroyed the only environment and then drop into the kernel monitor. You should be able to get make grade to succeed on the hello test. |

在libmain中添加如下两行代码就可以设置thisenv了，先通过sys\_getenvid系统调用获得当前进程号，然后用宏ENVX算出其在envs数组中的下标。

|  |
| --- |
| envid\_t envid = sys\_getenvid();  thisenv = envs + ENVX(envid); |

###### Page faults and memory protection

内存保护是操作系统一个很重要的特征，它保证了一个程序的bug不会使其他程序崩溃。

操作系统通常依靠硬件支持来实现内存保护。当程序试图访问无效或者无权限的地址时，处理器停止当前指令的执行并陷入内核，如果该错误可以修复，内核将对其修复并继续运行否则停机或重启。常见的可修复错有自动扩展栈、缺页中断等。

系统功能调用呈现了一个有趣的内存保护问题，大多数系统调用接口让用户程序传递指针可内核，这些指针指向要读写的用户缓冲。当执行系统调用时，内核对这些指针解引用。这里需要关注两个问题：

1. 内核的页错误往往比用户页错误要严重地多，如果用户要访问地地址非法而在系统调用时内核不做检查的话就可能引发内核页错误！！！。从而导致停机或重启。
2. 内核比用户具有更多的权限，若用户请求访问内核权限的数据，而内核又不做检查直接在内核态对这些数据读写并返回用户很可能会使数据泄露，危害数据安全。

针对这两个问题，我们需要实现JOS的内存保护机制。

|  |
| --- |
| Exercise 9. Change kern/trap.c to panic if a page fault happens in kernel mode.  Hint: to determine whether a fault happened in user mode or in kernel mode, check the low bits of the tf\_cs.  Read user\_mem\_assert in kern/pmap.c and implement user\_mem\_check in that same file.  Change kern/syscall.c to sanity check arguments to system calls.  Boot your kernel, running user/buggyhello. The environment should be destroyed, and the kernel should *not* panic. You should see:  [00001000] user\_mem\_check assertion failure for va 00000001  [00001000] free env 00001000  Destroyed the only environment - nothing more to do!  Finally, change debuginfo\_eip in kern/kdebug.c to call user\_mem\_check on usd, stabs, and stabstr. If you now run user/breakpoint, you should be able to run backtrace from the kernel monitor and see the backtrace traverse into lib/libmain.c before the kernel panics with a page fault. What causes this page fault? You don't need to fix it, but you should understand why it happens. |

分步完成该练习要求：

首先在page\_fault\_handler中完成内核页错误的相关处理，判读该错误是由内核还是用户引起的，可以通过trapframe中的CS寄存器低两位CPL来确定。如果是0则说明是内核。直接给出要填充的代码：

|  |
| --- |
| if((tf->tf\_cs & 3) == 0){  cprintf("[%08x] user fault va %08x ip %08x\n",  curenv->env\_id, fault\_va, tf->tf\_eip);  panic("page\_fault\_handler: kernel page fault at %x!\n",fault\_va);  } |

阅读user\_mem\_assert函数，该函数调用user\_mem\_check来判断用户访问是否合法，如果访问非法则打印该地址并销毁此用户进程。因此重点就是实现user\_mem\_check函数。根据要求，user\_mem\_check检查进程env是否具有perm权限访问虚拟地址空间[va,va+len)。如果权限不符，则将引起错误的地址赋给静态变量user\_mem\_check\_addr并返回-E\_FAULT。完成这一功能需要使用pgdir\_ walk找到相应va的页表项地址，并判断页描述符中是否具有指定的权限。

|  |
| --- |
| int  user\_mem\_check(struct Env \*env, const void \*va, size\_t len, int perm)  {  // LAB 3: Your code here.  void \*va\_end = (void\*)va + len;  void \*va\_start = ROUNDDOWN((void\*)va,PGSIZE);  for(;va\_start<va\_end;va\_start+=PGSIZE){  pde\_t \*pde\_ptr = pgdir\_walk(env->env\_pgdir,va\_start,0);  int r\_perm = perm | PTE\_P;  if((uint32\_t)va\_start>=ULIM||!pde\_ptr||((\*pde\_ptr&r\_perm)!=r\_perm)){  user\_mem\_check\_addr = va\_start < va?(uintptr\_t)va:(uintptr\_t)va\_start;  return -E\_FAULT;  }  }  return 0;  } |

下一步是补全kern/syscall.c中的sys\_cputs函数，该系统功能函数打印指定长度的字符串，因此需要检查该字符串是否具有用户读权限，直接调用user\_mem\_ assert函数：

|  |
| --- |
| user\_mem\_assert(curenv,s,len,0); |

最后，需要在kdebug.c中修改debuginfo\_eip,增加对usd、stab、stabstr的权限检查。在user/user.ld中，链接器利用标号\_STAB\_BEGIN\_,\_STAB\_END\_, STABSTR\_BEGIN\_,STABSTR\_END分别表示.stab段和.stabstr段的起始、结束地址，并将这四个地址信息组成一个结构放到链接地址0x200000处，该位置对应宏USTABDATA。

|  |
| --- |
| .stab\_info 0x200000 : {  LONG(\_\_STAB\_BEGIN\_\_);  LONG(\_\_STAB\_END\_\_);  LONG(\_\_STABSTR\_BEGIN\_\_);  LONG(\_\_STABSTR\_END\_\_);  }  .stab : {  \_\_STAB\_BEGIN\_\_ = DEFINED(\_\_STAB\_BEGIN\_\_) ? \_\_STAB\_BEGIN\_\_ : .;  \*(.stab);  \_\_STAB\_END\_\_ = DEFINED(\_\_STAB\_END\_\_) ? \_\_STAB\_END\_\_ : .;  BYTE(0) /\* Force the linker to allocate space  for this section \*/  }  .stabstr : {  \_\_STABSTR\_BEGIN\_\_ = DEFINED(\_\_STABSTR\_BEGIN\_\_) ? \_\_STABSTR\_BEGIN\_\_ : .;  \*(.stabstr);  \_\_STABSTR\_END\_\_ = DEFINED(\_\_STABSTR\_END\_\_) ? \_\_STABSTR\_END\_\_ : .;  BYTE(0) /\* Force the linker to allocate space  for this section \*/  } |

在debuginfo\_eip中会用struct UserStabData usd来表示.stab\_info段，从中读取.stab和.stabstr的首尾位置信息，因此需要检查usd是否具有用户读权限。

此外该函数还会读取整个.stab段和.stabstr段，同样需要对这两个段进行权限检查，代码如下：

|  |
| --- |
| const struct UserStabData \*usd = (const struct UserStabData \*) USTABDATA;  // Make sure this memory is valid.  // Return -1 if it is not. Hint: Call user\_mem\_check.  // LAB 3: Your code here.  if(user\_mem\_check(curenv,usd,sizeof(const struct UserStabData),PTE\_U)<0)  return -1;  stabs = usd->stabs;  stab\_end = usd->stab\_end;  stabstr = usd->stabstr;  stabstr\_end = usd->stabstr\_end;  // Make sure the STABS and string table memory is valid.  // LAB 3: Your code here. if(user\_mem\_check(curenv,stabs,(uint32\_t)stab\_end-(uint32\_t)stabs,PTE\_U)<0  ||user\_mem\_check(curenv,stabstr,(uint32\_t)stabstr\_end-(uint32\_t)stabstr,PTE\_U)<0)  return -1; |

至此，任务9顺利完成，最后就是测试user\_mem\_check的正确性了。

|  |
| --- |
| Exercise 10. Boot your kernel, running user/evilhello. The environment should be destroyed, and the kernel should not panic. You should see:  [00000000] new env 00001000  ...  [00001000] user\_mem\_check assertion failure for va f010000c  [00001000] free env 00001000 |

按照上面的要求，使用命令make run-evilhello得到预定的输出，测试正确。

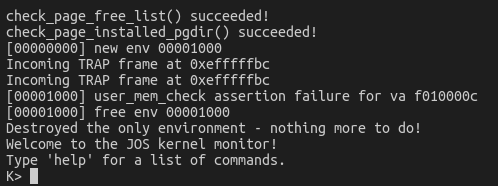


图3-7 user\_mem\_check 测试

执行评分脚本，各个测试全部顺利通过：

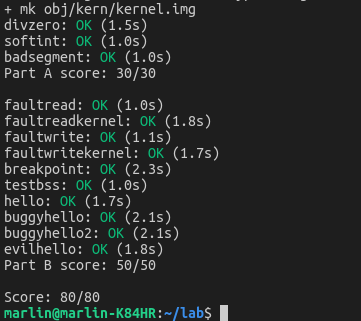


图3-8 lab3 测试结果

#### **3 实验问题**

1. What is the purpose of having an individual handler function for each exception/interrupt? (i.e., if all exceptions/interrupts were delivered to the same handler, what feature that exists in the current implementation could not be provided?)

如果所有中断共享一个中断处理函数那么该函数需要处理来自用户态和内核态的中断请求，特权位必须为3才能都响应，但是这样就无法做权限检查了。

1. Did you have to do anything to make the user/softint program behave correctly? The grade script expects it to produce a general protection fault (trap 13), but softint's code says int $14. *Why* should this produce interrupt vector 13? What happens if the kernel actually allows softint's int $14 instruction to invoke the kernel's page fault handler (which is interrupt vector 14)?

不行，user/softint程序用内联汇编调用14号中断，该中断的DPL位是0，用户程序没有该权限会使处理器产生通用保护错进而触发13号中断。

1. The break point test case will either generate a break point exception or a general protection fault depending on how you initialized the break point entry in the IDT (i.e., your call to SETGATE from trap\_init). Why? How do you need to set it up in order to get the breakpoint exception to work as specified above and what incorrect setup would cause it to trigger a general protection fault?

这是由断点中断的中断描述符决定的。在中断描述符中如果将DPL设置为0，那么用户程序将会产生通用保护错，只有设置为3才能实现预定的功能。

1. What do you think is the point of these mechanisms, particularly in light of what the user/softint test program does?

这些机制的重点是限制用户的权限，防止其通过中断破坏内核数据引发不可恢复的错误。用户只能使用不影响内核的系统调用如设置断点，读取进程id等。

### Lab4 **Preemptive** Multitasking

#### **1 实验目的**

本实验将完成多个运行进程的抢占式多任务。主要包括如下几个方面：

1. 了解多处理器的启动流程，在JOS中添加多处理器支持。
2. 理解、实现论询调度，并添加基本的进程管理系统调用函数。
3. 实现类UNIX的fork函数，用该函数在用户态创建本进程的完整拷贝。考虑写时拷贝机制的优点。
4. 添加进程间通信机制，掌握不同用户进程之间交流、同步的方法。

#### **2 实验内容**

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

###### Multiprocessor Support

本部分将使JOS支持SMP，在该模式下，所有CPU拥有相同的访问系统资源权限，如总线、内存资源。尽管该模式下的CPU功能一致，但是可以根据启动过程将其分为两类：启动处理器BSP和应用处理器AP。系统加载时会先启动BSP,即此时kernel运行在BSP中，当BSP初始化完一些参数后，会逐个激活其他AP。最后所有AP共同运行调度各个进程运行。

在SMP系统中，每个CPU有一个LAPIC单元，这些单元负责在系统间传递中断信息，LAPIC为其连接的CPU提供了一个唯一的编号，本实验会使用LAPIC单元的一些基本功能：

1. 读取LAIPC ID，确定正在运行当前代码的CPU。
2. 从BSP中发送STARUP IPI给APs,用来唤醒其他CPUs.
3. 编辑LAPIC的内建时钟触发时钟中断来支持多任务抢占。

处理器是通过memory-mapped I/O来访问它的LAPIC。在MMIO中，一个物理内存的端口被硬布线到一些I/O设备寄存器，这样就可以通过内存读写的方式来访问设备寄存器了。

|  |
| --- |
| Exercise 1. Implement mmio\_map\_region in kern/pmap.c. To see how this is used, look at the beginning of lapic\_init in kern/lapic.c. You'll have to do the next exercise, too, before the tests for mmio\_map\_region will run. |

在memlayout.h中，供MMIO使用的地址空间位于[MMIOBASE,MMIOLIM]。需要将各个处理器的lapic物理地址映射到此块区域以便直接使用虚拟地址访问。为了进一步理解MMIO的使用方法，在kern/lapic.c中观察lapic\_int函数：

|  |
| --- |
| static void  lapicw(int index, int value)  {  lapic[index] = value; //直接使用虚拟地址读写寄存器端口  lapic[ID]; // wait for write to finish, by reading  }  void  lapic\_init(void)  {  if (!lapicaddr)  return;  lapic = mmio\_map\_region(lapicaddr, 4096);//将lapicaddr处4字节映射  lapicw(SVR, ENABLE | (IRQ\_OFFSET + IRQ\_SPURIOUS));  lapicw(TDCR, X1);  .........  } |

在lapic\_init完成MMIO后，可以很方便地使用虚拟地址lapic[index]对特定的设备寄存器进行读写而不必使用内联汇编的in、out指令。现在来实现mmio\_ map\_region。该函数把从物理地址pa开始的size字节映射到指定区域。由于每个CPU初始化时（mp\_main）都会调用lapic\_init。因此会进行多次MMIO映射。在mmio\_region中使用静态指针base记录下一次分配的起始虚拟地址。初始时其值为MMIOBASE,每次分配完空间就会移动到下一个空闲地址。在此函数中可以使用boot\_map\_region对指定区域进行静态映射，代码如下：

|  |
| --- |
| void \*  mmio\_map\_region(physaddr\_t pa, size\_t size)  {  static uintptr\_t base = MMIOBASE;  size = ROUNDUP(pa+size,PGSIZE); //指向映射的上界  pa = ROUNDDOWN(pa,PGSIZE); //指向下界  size -= pa; //实际按页对齐的大小  if(base+size >= MMIOLIM)  panic("mmio\_map\_region: the map would overflow MMIOLIM\n");  boot\_map\_region(kern\_pgdir,base,size,pa,PTE\_PCD|PTE\_PWT|PTE\_W);  base += size; //base 移动  return (void\*)(base-size);  } |

###### Application Processor Bootstrap

在启动APs之前，BSP会先收集多处理机系统的信息。诸如CPU个数、APIC ID以及LAPIC单元的MMIO地址。这部分工作由kern/mpconfig.c中mp\_init函数完成。该函数通过读取位于BIOS的MP configuration table确定上述参数，其实现细节接近底层，此处不再深究。

位于kern/init.c中的boot\_aps函数被用来驱动AP引导程序。由于AP起始于实模式，因此boot\_aps会先复制其入口代码到物理地址0x7000 (MPENTRY\_ PADDR)处，然后逐一发送STARTUP IPI给各个AP并使其CS:IP寄存器指向汇编代码的入口点。接着便等待AP的CPU\_STARTED信号。

|  |
| --- |
| Exercise 2. Read boot\_aps() and mp\_main() in kern/init.c, and the assembly code in kern/mpentry.S. Make sure you understand the control flow transfer during the bootstrap of APs. Then modify your implementation of page\_init() in kern/pmap.c to avoid adding the page at MPENTRY\_PADDR to the free list, so that we can safely copy and run AP bootstrap code at that physical address. Your code should pass the updated check\_page\_free\_list() test (but might fail the updated check\_kern\_pgdir() test, which we will fix soon). |

Boot\_aps的主要流程上面已经介绍过了，即先加载AP的汇编入口代码到指定物理地址MPENTRY\_PADDR,然后逐一给其他AP发送STARTUP IPI使它们执行汇编引导代码。当boot\_aps收到一个AP的CPU\_STARTED信号后就可以继续启动其它AP了。在AP执行的汇编入口mpentry.s中，主要完成处理器的模式切换以及堆栈的设置，最后跳转到mp\_main函数执行C代码，该部分初始化了该CPU的cr3寄存器和LAPIC并对进程控制块及中断进行了设置。

由于将汇编代码载入到了物理地址MPENTRY\_PADDR处，因此需要修改pmap.c以防止内存分配器（page\_alloc等函数）占用并修改此块内存。如下是修改后的代码：

|  |
| --- |
| 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==MPENTRY\_PADDR/PGSIZE)|| //标记MPENTRY\_PADDR 处的一页为in use  (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];  }  }  } |

###### Per-CPU State and Initialization

当编写多处理机OS时，需要区分各CPU的私有状态以及共享状态。私有状态包括以下信息：

1. 各CPU内核栈。由于多个CPU可以同时陷入内核，需要为每个CPU分配各自独有的内核栈以防互相干扰，在数组percpu\_kstacks[NCPU][KSTKSIZE]中预留了NCPU个内核栈的空间。
2. 各CPU任务状态段及任务状态段描述符。需要为每个CPU分配TSS指明内核栈的位置。
3. 各CPU当前进程指针。由于现在可以有多个进程同时运行，不能再依靠原来的全局变量curenv来指示当前运行进程。取而代之的是对curenv重定义为thiscpu ->cpu\_env。
4. 各CPU系统状态寄存器。所有寄存器（包括系统状态寄存器）都由各CPU私有，因此所有CPU都需要调用env\_init\_percpu和trap\_init\_percpu函数对寄存器进行初始化。

|  |
| --- |
| Exercise 3. Modify mem\_init\_mp() (in kern/pmap.c) to map per-CPU stacks starting at KSTACKTOP, as shown in inc/memlayout.h. The size of each stack is KSTKSIZE bytes plus KSTKGAP bytes of unmapped guard pages. Your code should pass the new check in check\_kern\_pgdir(). |

完成各CPU内核栈的设置，已经做过多次相关映射了，直接给代码：

|  |
| --- |
| static void  mem\_init\_mp(void)  {      for(int i=0;i<NCPU;i++){          boot\_map\_region(kern\_pgdir,KSTACKTOP-KSTKSIZE-i\*(KSTKSIZE+KSTKGAP),          KSTKSIZE,PADDR(percpu\_kstacks[i]),PTE\_W);      }  } |

|  |
| --- |
| Exercise 4. The code in trap\_init\_percpu() (kern/trap.c) initializes the TSS and TSS descriptor for the BSP. It worked in Lab 3, but is incorrect when running on other CPUs. Change the code so that it can work on all CPUs. (Note: your new code should not use the global ts variable any more.) |

为了适应多CPU的运行环境，需要修改trap\_init\_percpu设置特定cpu的信息。根据函数的提示将相关变量或下标进行调整即可。

|  |
| --- |
| void  trap\_init\_percpu(void)  {  int i=thiscpu->cpu\_id; //得到cpu id，作为索引  thiscpu->cpu\_ts.ts\_esp0 = KSTACKTOP - i\*(KSTKSIZE+KSTKGAP);  thiscpu->cpu\_ts.ts\_ss0 = GD\_KD;  gdt[(GD\_TSS0 >> 3)+ i] = SEG16(STS\_T32A, (uint32\_t) (&(thiscpu->cpu\_ts)),  sizeof(struct Taskstate) - 1, 0);  gdt[(GD\_TSS0 >> 3) + i].sd\_s = 0;  ltr(GD\_TSS0 + (i<<3));  lidt(&idt\_pd);  } |

###### Locking

为了防止多个CPU同时运行内核带来竞争和险象，需要在适当的地方加上锁，保证同一时刻只有一个CPU运行在内核态。Kern/spinlock.h中定义了加锁函数lock\_kernel以及解锁函数unlock\_kernel。其实现原理是利用了原子操作xchg指令，此处不在细述。我们需要在4个地方使用该锁。分别如下：

1. 在i386\_init（）里需要在BSP唤醒其他CPU之前加锁。
2. 在mp\_main（）里，需要在轮询调度用户进程之前加锁。（防止多个CPU同时调度相同的用户进程）
3. 在trap（）里，当从用户态陷入内核时需要加锁。（防止同时多个进程由于中断陷入内核）
4. 在env\_run中，释放内核锁。

|  |
| --- |
| Exercise 5. Apply the big kernel lock as described above, by calling lock\_kernel() and unlock\_kernel() at the proper locations. |

直接按照上面的描述，在适当的函数内加锁即可。这里唯一需要注意的地方是env\_run中unlock\_kernel的位置。解锁必须在env\_pop\_tf之前，因为env\_pop\_tf执行后，eip,esp寄存器都被恢复为用户进程的现场，不会在执行env\_pop\_tf后的指令了。

###### Round-Robin Scheduling

本部分主要是实现用户进程的循环轮转调度，由函数sched\_yield实现。其主要功能是从进程控制块数组envs中依次挑选出所有可运行的用户进程投入运行。选择依据是从当前运行进程的后一个开始遍历envs数组找到处于RUNNABLE状态的进程。如果没有找到且当前运行的进程正处于RUNNING,则继续运行当前程序，否则调用sched\_halt停止运行。

|  |
| --- |
| Exercise 6. Implement round-robin scheduling in sched\_yield() as described above. Don't forget to modify syscall() to dispatch sys\_yield().  Make sure to invoke sched\_yield() in mp\_main.  Modify kern/init.c to create three (or more!) environments that all run the program user/yield.c.  Run make qemu. You should see the environments switch back and forth between each other five times before terminating, like below.  Test also with several CPUS: make qemu CPUS=2.  ...  Hello, I am environment 00001000.  Hello, I am environment 00001001.  Hello, I am environment 00001002.  Back in environment 00001000, iteration 0.  Back in environment 00001001, iteration 0.  Back in environment 00001002, iteration 0.  Back in environment 00001000, iteration 1.  Back in environment 00001001, iteration 1.  Back in environment 00001002, iteration 1.  ...  After the yield programs exit, there will be no runnable environment in the system, the scheduler should invoke the JOS kernel monitor. If any of this does not happen, then fix your code before proceeding. |

根据上面的要求可以写出sched\_yield的代码，详情见注释：

|  |
| --- |
| void  sched\_yield(void)  {  struct Env \*idle;  int head=0; //当无进程运行，head=0  if(curenv) head=(ENVX(curenv->env\_id)+1)%NENV; //有进程运行，head指向下一个进程  int i;  for(i=0;i<NENV;i++){//全部调度完RUNNABLE的进程  if(envs[head].env\_status == ENV\_RUNNABLE)  env\_run(&envs[head]);  head = (head+1)%NENV;  }  if(curenv && curenv->env\_status == ENV\_RUNNING) //本进程没运行完继续投入运行。  env\_run(curenv);  sched\_halt();//全部运行完毕  } |

接着在syscall中增加功能号为SYS\_yield的系统调用并在mp\_main中加上sched\_yield。实现较简单，故省略。

最后在init.c中创建3个进程进行测试，需要使用宏ENV\_CREATE。如下是测试结果：

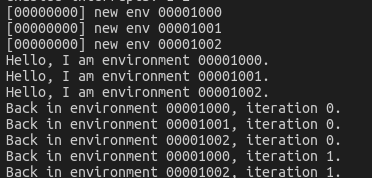


图4-1 循环调度测试结果

###### System Calls for Environment Creation

在用户态实现进程创建需要完成几个相关的系统调用函数:

1. sys\_exofork:该系统调用创建一个几乎空白的新进程，无地址映射且不可运行。该函数向父进程返回子进程的ID,向子进程返回0.
2. sys\_env\_set\_status:将进程的状态设置为ENV\_RUNNABLE或者ENV\_NOT\_RUNNABLE.
3. sys\_page\_alloc:向特定进程分配一个物理页并映射到给定虚拟地址。
4. sys\_page\_map:从一个进程地址空间中复制一页的映射关系到另一个进程，注意是只复制映射关系而非页内容。
5. sys\_page\_unmap:解除指定进程某个虚拟地址对应的页。

|  |
| --- |
| Exercise 7. Implement the system calls described above in kern/syscall.c and make sure syscall() calls them. You will need to use various functions in kern/pmap.c and kern/env.c, particularly envid2env(). For now, whenever you call envid2env(), pass 1 in the checkperm parameter. Be sure you check for any invalid system call arguments, returning -E\_INVAL in that case. Test your JOS kernel with user/dumbfork and make sure it works before proceeding. |

对于sys\_exofork,可通过env\_alloc分配一个新的进程，将进程状态设置为ENV\_NOT\_RUNNABLE并复制父进程的寄存器值（curenv->env\_tf)。最后返回即可。

|  |
| --- |
| static envid\_t  sys\_exofork(void)  {  struct Env \*pe;  int status = env\_alloc(&pe,curenv->env\_id);  if(status< 0 ) return status;  pe->env\_status = ENV\_NOT\_RUNNABLE;  pe->env\_tf = curenv->env\_tf;  pe->env\_tf.tf\_regs.reg\_eax = 0;  return pe->env\_id;  } |

sys\_env\_set\_status只需设置一下env\_status，直接给出代码：

|  |
| --- |
| static int  sys\_env\_set\_status(envid\_t envid, int status)  {  if(status !=ENV\_RUNNABLE && status!=ENV\_NOT\_RUNNABLE)  return -E\_INVAL;  struct Env\* pe;  int r = envid2env(envid,&pe,true);  if(r == 0)  pe->env\_status = status;  return r;  } |

sys\_page\_alloc先利用page\_alloc从空闲页链表中申请一个空闲页，然后使用page\_insert将其插入到指定的虚拟地址空间。如果插入失败则释放物理页。

|  |
| --- |
| static int  sys\_page\_alloc(envid\_t envid, void \*va, int perm)  {  if((uint32\_t)va >= UTOP || (uint32\_t)va%PGSIZE || (perm&PTE\_U)==0 || (perm&PTE\_P)==0  || (perm&(~(PTE\_AVAIL|PTE\_W|PTE\_U|PTE\_P))))  return -E\_INVAL;  struct Env\* pe;  int r1 = envid2env(envid,&pe,true);  if(r1 < 0) return r1;  struct PageInfo\* pg = page\_alloc(ALLOC\_ZERO); //分配物理页  if(pg == NULL) return -E\_NO\_MEM;  int r2 = page\_insert(pe->env\_pgdir,pg,va,perm); //插入虚拟地址空间  if(r2 < 0) {  page\_free(pg); //插入失败释放物理页  return r2;  }  return 0;  } |

sys\_page\_map先使用page\_lookup找到源进程虚拟地址srcva对应的物理页，然后用page\_insert插入目的进程的虚拟地址dstva。

sys\_page\_unmap则通过调用page\_remove卸除特定地址va的页。代码如下:

|  |
| --- |
| static int  sys\_page\_map(envid\_t srcenvid, void \*srcva,envid\_t dstenvid, void \*dstva, int perm)  {  struct Env\* pesrc,\*pedst;  int r;  if((r=envid2env(srcenvid,&pesrc,true))||(r=envid2env(dstenvid,&pedst,true)))  return r;  if((uint32\_t)srcva>= UTOP || (uint32\_t)srcva%PGSIZE ||(uint32\_t)dstva>=UTOP ||(uint32\_t)dstva%PGSIZE)  return -E\_INVAL;  pte\_t \* pte\_ptr;  struct PageInfo \*pg;  if((pg=page\_lookup(pesrc->env\_pgdir,srcva,&pte\_ptr))==NULL)  return -E\_INVAL;  if((perm&PTE\_U)==0||(perm&PTE\_P)==0|| (perm&(~(PTE\_P|PTE\_U|PTE\_AVA IL|PTE\_W))))  return -E\_INVAL;  if( (perm&PTE\_W) && (((\*pte\_ptr)&PTE\_W)==0) )  return -E\_INVAL;  r = page\_insert(pedst->env\_pgdir,pg,dstva,perm);  return r;  }  static int  sys\_page\_unmap(envid\_t envid, void \*va)  {  if((uint32\_t)va >= UTOP || (uint32\_t)va%PGSIZE)  return -E\_INVAL;  int r;  struct Env \*pe;  r=envid2env(envid,&pe,true);  if(r) return r;  page\_remove(pe->env\_pgdir,va);  return 0;  } |

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

所谓的写时拷贝是在进程创建时不实际复制父进程的所有也给子进程而是仅仅拷贝映射关系，并将这些页标记为只读。当父、子进程想要往其中写入时才会将该页进行完整的拷贝。这样做的原因是当我们使用fork时，往往会接着调用exec,原来完整复制的页会被重新写入，造成极大的浪费。而使用写时拷贝只拷贝了映射关系因此代价很低。

###### Setting the Page Fault Handler

为了处理用户态写时拷贝触发的页错误，需要登记页错误处理函数的入口点。

通过执行sys\_env\_set\_pgfault\_upcall系统调用可以在用户进程控制块中设置env\_pgfault\_upcall。

|  |
| --- |
| Exercise 8. Implement the sys\_env\_set\_pgfault\_upcall system call. Be sure to enable permission checking when looking up the environment ID of the target environment, since this is a "dangerous" system call |

代码如下：

|  |
| --- |
| static int  sys\_env\_set\_pgfault\_upcall(envid\_t envid, void \*func)  {  struct Env\* pe;  int r = envid2env(envid,&pe,true);  if(r<0) return r;  pe->env\_pgfault\_upcall = func;  return 0;  } |

###### Invoking the User Page Fault Handler

在正常执行过程中，用户进程会在普通用户栈上运行。即位于地址空间[USTACKTOP-PGSIZE,USTACKTOP-1],然而当触发用户页错误时，kernel会将用户进程转到用户异常栈上执行，具体如下。

如果进程没有登记异常处理函数，kernel会销毁该进程，否则在用户异常栈上建立如下的trap frame：

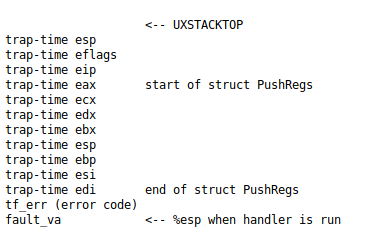


图4-2 trap frame

接着，kernel会在此异常栈上继续执行后续的异常处理函数。需要提及的是如果当异常发生时用户程序已经在内核栈上运行，那么要在当前栈下面压入栈帧而不是从UXSTACKTOP开始压帧。此外还需记得在两个UTrapframe中间预留一个字供后面的任务使用。

|  |
| --- |
| Exercise 9. Implement the code in page\_fault\_handler in kern/trap.c required to dispatch page faults to the user-mode handler. Be sure to take appropriate precautions when writing into the exception stack. (What happens if the user environment runs out of space on the exception stack?) |

在写代码之前先梳理一下用户页错误的处理流程，首先用户程序往写时拷贝的地址空间中写入数据，由于这些页被标记为PTE\_COW(copy on write),不具写权限从而引发页错误陷入内核。内核在内核栈上执行缺页中断处理函数page\_fault\_handle，该函数先检查该错误是由谁引发，如果是内核页错误，直接panic。否则在用户异常栈上建立struct UTrapframe并使用户进程运行其pg\_fault\_upcall重新进入用户态。

根据上面的描述在page\_fault\_hanler中补充如下代码：

|  |
| --- |
| if(curenv->env\_pgfault\_upcall){  char \*top = (char\*)(UXSTACKTOP-1+4);  if(tf->tf\_esp>=UXSTACKTOP-PGSIZE&& tf->tf\_esp<= UXSTACKTOP-1)  top=(char\*)tf->tf\_esp;  size\_t size = sizeof(struct UTrapframe) + 4;  if(top == (char\*)(UXSTACKTOP+3))//内存保护检查  user\_mem\_assert(curenv,top-size,sizeof(struct UTrapframe),PTE\_U|PTE\_W);  Else //内存保护检查  user\_mem\_assert(curenv,top-size,size,PTE\_U|PTE\_W);  struct UTrapframe\* utf\_ptr = (struct UTrapframe\*)(top-size); //设置UTrapf rame起始位置  utf\_ptr->utf\_eflags = tf->tf\_eflags; //设置UTrapframe结构  utf\_ptr->utf\_eip = tf->tf\_eip;  utf\_ptr->utf\_err = tf->tf\_err;  utf\_ptr->utf\_esp = tf->tf\_esp;  utf\_ptr->utf\_fault\_va = fault\_va;  utf\_ptr->utf\_regs = tf->tf\_regs;  tf->tf\_esp = (uintptr\_t)utf\_ptr; //使用户进程进入env\_pgfault\_upcall  tf->tf\_eip = (uintptr\_t)curenv->env\_pgfault\_upcall;  env\_run(curenv);//运行用户进程  return;  } |

###### User-mode Page Fault Entrypoint

|  |
| --- |
| Exercise 10. Implement the \_pgfault\_upcall routine in lib/pfentry.S. The interesting part is returning to the original point in the user code that caused the page fault. You'll return directly there, without going back through the kernel. The hard part is simultaneously switching stacks and re-loading the EIP. |

Kernel通过env\_pgfault\_upcall使用户进程进入pfentry.s。利用该文件完成页错误处理以及返回原始用户进程。

处理页错误是通过调用函数指针\_pgfault\_handler所指代的函数来完成的。可以通过设置该函数指针执行不同的处理函数。其指令如下：

|  |
| --- |
| .text  .globl \_pgfault\_upcall  \_pgfault\_upcall:  pushl %esp // function argument: pointer to UTF  movl \_pgfault\_handler, %eax  call \*%eax //执行异常处理函数  addl $4, %esp // pop function argument |

完成异常处理后需要返回用户原始进程，也就是我们要完成的工作。回忆一下UTrapframe的布局图：

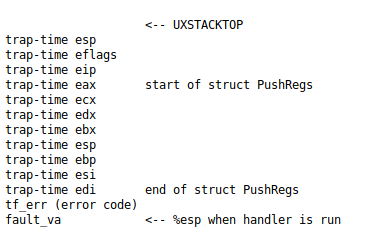


图4-3 UTrapframe布局图

可以利用该布局图还原原始进程的trapframe。对照上图，可以写出如下代码。各指令的作用可参见注释：

|  |
| --- |
| addl $8,%esp //忽略error code和fault\_va  movl 32(%esp),%eax //把原始eip暂存eax  movl 40(%esp),%ebx //把原始esp暂存ebx  movl %eax,-4(%ebx) //把原始eip放到我们在栈中预留的一个字  popal //恢复通用寄存器组  addl $4,%esp //跳过eip  popfl //还原eflags  popl %esp //回到原来的esp  lea -4(%esp),%esp//把esp指向它下一个字（就是我们预留放eip的）  ret //还原eip. |

通过上述这段指令，我们可以顺利回到出错时的环境。

最后我们要完成设置\_pgfault\_handler指针的函数set\_pgfault\_handler.

|  |
| --- |
| Exercise 11. Finish set\_pgfault\_handler() in lib/pgfault.c. |

直接给出代码：

|  |
| --- |
| void  set\_pgfault\_handler(void (\*handler)(struct UTrapframe \*utf))  {  int r;  if (\_pgfault\_handler == 0) { //第一次设置  r = //设置用户异常栈 = sys\_page\_alloc(0,(void\*)(UXSTACKTOP-PGSIZE),PTE\_U|PTE\_W|PTE\_P);  if(r<0)  panic("set\_pgfault\_handler %e\n:",r);  sys\_env\_set\_pgfault\_upcall(0,\_pgfault\_upcall); //设置page\_fault\_upcall  }  \_pgfault\_handler = handler;//设置异常处理函数指针  } |

###### Implementing Copy-on-Write Fork

完成写时拷贝的fork函数。主要包括三个方面：实现异常处理函数pgfault、映射关系拷贝函数duppage以及主流程控制函数fork。

|  |
| --- |
| Exercise 12. Implement fork, duppage and pgfault in lib/fork.c.  Test your code with the forktree program. It should produce the following messages, with interspersed 'new env', 'free env', and 'exiting gracefully' messages. The messages may not appear in this order, and the environment IDs may be different.  1000: I am ''  1001: I am '0'  2000: I am '00'  2001: I am '000'  1002: I am '1'  3000: I am '11'  3001: I am '10'  4000: I am '100'  1003: I am '01'  5000: I am '010'  4001: I am '011'  2002: I am '110'  1004: I am '001'  1005: I am '111'  1006: I am '101' |

Pgfault首先检查该页错误是否由写时拷贝引起的，如果不是,直接panic,否则先调用sys\_page\_alloc分配一个物理页并映射到临时使用的虚拟地址PFTEMP,然后将写时拷贝页的内容全部复制过来，最后将该页重新映射到原始页地址。

|  |
| --- |
| static void  pgfault(struct UTrapframe \*utf)  {  void \*addr = (void \*) utf->utf\_fault\_va; //引发页错误的地址  uint32\_t err = utf->utf\_err; //错误号  int r;  if(!((err&FEC\_WR)&&(uvpt[PGNUM(addr)] & PTE\_COW)))  panic("pgfault: VA of %x isn't COW page fault\n",addr);  r = sys\_page\_alloc(0,PFTEMP,PTE\_U|PTE\_W|PTE\_P); //分配一页并映射到PFTEMP  if(r < 0) panic("pgfault:%e\n",r);  memcpy(PFTEMP,(void\*)ROUNDDOWN(addr,PGSIZE),PGSIZE);//复制该页内容  r= sys\_page\_map(0,PFTEMP,0,ROUNDDOWN(addr,PGSIZE),PTE\_U|PTE\_W|PTE\_P);//重新映射到原始页地址  if(r < 0) panic("pgfault:%e\n",r);  r = sys\_page\_unmap(0,PFTEMP); //解除临时映射地址的映射关系  if(r < 0) panic("pgfault:%e\n",r);  } |

duppage将虚拟页pn （地址为pn\*PGSIZE)映射到目标进程envid的相同虚拟地址。如果该页是可写的或者是copy on write的，那么需要将新的映射关系创建为copy on write，同时将原始的映射关系修改为copy on write。否则对于只读的页仅仅复制映射关系即可，无须修改标志位。

|  |
| --- |
| static int  duppage(envid\_t envid, unsigned pn)  {  int r;  // LAB 4: Your code here.  void \*addr = (void\*)(pn\*PGSIZE);  if((uvpt[pn] & PTE\_W) || (uvpt[pn] & PTE\_COW)){  if((r=sys\_page\_map(0,addr,envid,addr,PTE\_COW|PTE\_P|PTE\_U))<0)  panic("duppage: %e\n",r);  if((r=sys\_page\_map(0,addr,0,addr,PTE\_COW|PTE\_P|PTE\_U))<0)  panic("duppage: %e\n",r);  }  else{  if((r=sys\_page\_map(0,addr,0,addr,PTE\_P|PTE\_U))<0){  cprintf("page num is %d\n",pn);  panic("duppage: %e\n",r);  }  }  return 0;  } |

fork函数有几个主要工作：创建子进程、设置好异常处理函数以及将父进程的地址空间复制给子进程。具体而言，先调用set\_pgfault\_handler设置父进程的异常处理函数，接着通过sys\_exofork创建子进程并设置好子进程的地址映射关系。然后添加子进程的入口点并将其状态设为RUNNABLE。

参见如下代码：

|  |
| --- |
| envid\_t  fork(void)  {  // LAB 4: Your code here.  extern void \_pgfault\_upcall(void);  set\_pgfault\_handler(pgfault); //设置异常处理函数  envid\_t pchild = sys\_exofork();  if(pchild<0) return pchild;  else if(pchild == 0){ //子进程  thisenv = &envs[ENVX(sys\_getenvid())];//调整thisenv指针  return 0;  }  else{ //父进程  uint32\_t addr;  for(addr=0x400000;addr <USTACKTOP;addr+=PGSIZE){  if((uvpd[PDX(addr)] & PTE\_P)&&  ((uvpt[PGNUM(addr)] & (PTE\_P|PTE\_U))==(PTE\_P|PTE\_U))){  duppage(pchild,PGNUM(addr)); //复制映射关系  }  }  int r;  if((r= sys\_page\_alloc(pchild,(void\*)(UXSTACKTOP-PGSIZE),PTE\_U|PTE\_W|PTE\_P))<0) //设置用户异常栈  panic("fork:%e\n",r);  if((r = sys\_env\_set\_pgfault\_upcall(pchild,\_pgfault\_upcall))<0)//设置子进程的异常处理入口地址  panic("fork:%e\n",r);  if((r = sys\_env\_set\_status(pchild,ENV\_RUNNABLE))<0)//将子进程标记为可运行状态  panic("fork:%e\n",r);  return pchild;  }  } |

使用make run-forktree测试fork的正确性，结果如下图所示，与标准输出相同，fork函数正确。

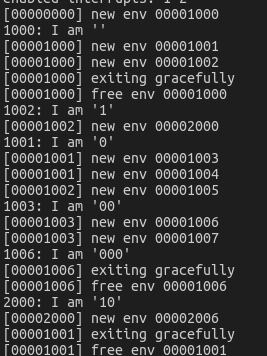


图4-4 forktree运行结果

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

###### Clock Interrupts and Preemption

外部中断又被称为IRQ,总共存在16个IRQ,分别编号为0-15。这些IRQ编号与IDT的入口的对应关系不是固定的，取决于不同系统的实现，在本实验中我们要把0-15的编号映射到IDT的入口点IRQ\_OFFSET-IRQ\_OFFSET+15。这样在发生外部中断时能根据IRQ号顺利索引到正确的中断处理函数。

到目前为止，JOS屏蔽了所有外部中断，外部中断受eflag寄存器的FL\_IF位影响，当它设为1时，外部中断被开启,反之则会被屏蔽掉。要实现抢占式多任务，一个重要的实现机制是时钟中断，而该中断属于外部中断，因此必须开启FL\_IF位。

|  |
| --- |
| Exercise 13. Modify kern/trapentry.S and kern/trap.c to initialize the appropriate entries in the IDT and provide handlers for IRQs 0 through 15. Then modify the code in env\_alloc() in kern/env.c to ensure that user environments are always run with interrupts enabled.  Also uncomment the sti instruction in sched\_halt() so that idle CPUs unmask interrupts.  The processor never pushes an error code when invoking a hardware interrupt handler. You might want to re-read section 9.2 of the [80386 Reference Manual](https://pdos.csail.mit.edu/6.828/2018/readings/i386/toc.htm), or section 5.8 of the [IA-32 Intel Architecture Software Developer's Manual, Volume 3](https://pdos.csail.mit.edu/6.828/2018/readings/ia32/IA32-3A.pdf), at this time.  After doing this exercise, if you run your kernel with any test program that runs for a non-trivial length of time (e.g., spin), you should see the kernel print trap frames for hardware interrupts. While interrupts are now enabled in the processor, JOS isn't yet handling them, so you should see it misattribute each interrupt to the currently running user environment and destroy it. Eventually it should run out of environments to destroy and drop into the monitor. |

类似内部中断的安装过程，我们需要在trapentry.s中添加中断汇编入口并在trap.c中设置中断向量表的相关表项。如下是两者的配置：

|  |
| --- |
| TRAPHANDLER\_NOEC(irq0,IRQ\_OFFSET+0)  TRAPHANDLER\_NOEC(irq1,IRQ\_OFFSET+1)  ... ... ...  TRAPHANDLER\_NOEC(irq15,IRQ\_OFFSET+15)  extern void irq0();  extern void irq1();  ... ... ...  extern void irq15();  SETGATE(idt[IRQ\_OFFSET+0],0,GD\_KT,irq0,3);  SETGATE(idt[IRQ\_OFFSET+1],0,GD\_KT,irq1,3);  ... ... ...  SETGATE(idt[IRQ\_OFFSET+15],0,GD\_KT,irq15,3); |

为开启外部中断我们在env\_alloc中添加如下一行代码.并在sched\_halt中取消sti的注释。

|  |
| --- |
| e->env\_tf.tf\_eflags |= FL\_IF; |

现在已经为响应外部中断做好了相关准备。接下来就是设置外部中断处理函数。

|  |
| --- |
| Exercise 14. Modify the kernel's trap\_dispatch() function so that it calls sched\_yield() to find and run a different environment whenever a clock interrupt takes place.  You should now be able to get the user/spin test to work: the parent environment should fork off the child, sys\_yield() to it a couple times but in each case regain control of the CPU after one time slice, and finally kill the child environment and terminate gracefully. |

此部分实现简单，只需要在trap\_dispatch()中增加一个case分支用来处理时钟中断即可。根据要求，当产生时钟中断时首先要向lapic发送一个ACK,然后转进程调度。

|  |
| --- |
| case IRQ\_OFFSET+IRQ\_TIMER:  lapic\_eoi(); //发送ack  sched\_yield();  return ; |

###### Inter-Process communication (IPC)

为了接受一个消息，进程将调用函数sys\_ipc\_recv,该系统调用对当前进程解调度停止其运行直至消息得到正确接收。当一个进程正在等待消息时，任何其它进程可以向它发送消息。

类似地，当一个进程想发送一个数值时，会调用sys\_ipc\_try\_send。该函数携带有接收进程ID以及要发送的值，如果目标进程正在等待消息，那么就将该值递交给目标进程否则返回-E\_IPC\_NOT\_RECV.

在用户态封装了两个库函数，ipc\_recv以及ipc\_send.前者通过调用sys\_ipc\_ recv来接收信息，并读取当前进程的struct Env获取接收值。后一个函数则通过重复调用sys\_ipc\_try\_send直至信息被正确接收。

当进程调用含有参数dstva的sys\_ipc\_recv时，表明该进程想要接受一个页面映射，如果恰好发送方发送了一个页那么就将该页插入接收进程的虚拟地址dstva处。

当发送进程调用带有有效参数srcva的sys\_ipc\_try\_send时，表明发送方想要发送在其地址空间srcva处的页给接收方。在一次成功的IPC之后发送方的页面映射不变，而接收方在其虚拟地址dstva处获得了发送方的一个页面。这样实现了该页在发送方与接收方之间的共享。

|  |
| --- |
| Exercise 15. Implement sys\_ipc\_recv and sys\_ipc\_try\_send in kern/syscall.c. Read the comments on both before implementing them, since they have to work together. When you call envid2env in these routines, you should set the checkperm flag to 0, meaning that any environment is allowed to send IPC messages to any other environment, and the kernel does no special permission checking other than verifying that the target envid is valid.  Then implement the ipc\_recv and ipc\_send functions in lib/ipc.c.  Use the user/pingpong and user/primes functions to test your IPC mechanism. user/primes will generate for each prime number a new environment until JOS runs out of environments. You might find it interesting to read user/primes.c to see all the forking and IPC going on behind the scenes. |

sys\_ipc\_try\_send的核心思想是先通过page\_lookup找到srcva对应的物理页，然后使用envid2env找到目标进程，设置其与ipc相关的信息，若目标进程dstva与发送进程的srcva均是有效的，则将srcva对应的物理页用page\_insert插入到目标进程的dstva处。

Sys\_ipc\_recv则是调整当前进程的IPC参数,如env\_ipc\_recving、env\_ipc\_ dstva等。然后将其状态设置为ENV\_NOT\_RUNNABLE,并转入进程调度。

如下是这两个函数的实现：

|  |
| --- |
| static int  sys\_ipc\_try\_send(envid\_t envid, uint32\_t value, void \*srcva, unsigned perm)  {  int r;  struct Env \*e;  if((r=envid2env(envid,&e,false))<0)//根据id得到目标进程的env结构  return r;  if(e->env\_ipc\_recving == false) //参数非法  return -E\_IPC\_NOT\_RECV;  if(srcva < (void\*)UTOP &&(uint32\_t)srcva%PGSIZE)//参数非法  return -E\_INVAL;  if(srcva < (void\*)UTOP && ((perm&PTE\_U)==0 || (perm&PTE\_P)==0  || (perm&(~(PTE\_AVAIL|PTE\_W|PTE\_U|PTE\_P))))) //参数非法  return -E\_INVAL;  pte\_t \*pte\_ptr;  struct PageInfo \*pg = page\_lookup(curenv->env\_pgdir,srcva,&pte\_ptr);//查找src va所在的物理页信息  if(srcva < (void\*)UTOP && pg==NULL)  return -E\_INVAL;  if((perm&PTE\_W) && (\*pte\_ptr&PTE\_W)==0 )  return -E\_INVAL;  //now comes to the core code  e->env\_ipc\_value = value; //ipc 通信的值  e->env\_ipc\_recving = 0; //将接收进程的接收状态关闭  e->env\_ipc\_from = curenv->env\_id;  e->env\_ipc\_perm = 0;  if(srcva < (void\*)UTOP && e->env\_ipc\_dstva < (void\*)UTOP){//要传递页  int r;  r = page\_insert(e->env\_pgdir,pg,e->env\_ipc\_dstva,perm);//插入到目标进程的dstva处  if(r < 0) return r;  e->env\_ipc\_perm = perm;  }  e->env\_tf.tf\_regs.reg\_eax = 0; //接收进程的返回值设为0  e->env\_status = ENV\_RUNNABLE;//恢复接收进程的状态  return 0;  }  static int  sys\_ipc\_recv(void \*dstva)  {  if(dstva < (void\*)UTOP && (uint32\_t)dstva%PGSIZE)//参数无效  return -E\_INVAL;  curenv->env\_ipc\_recving = 1; //接收进程等待标记打开  curenv->env\_ipc\_dstva = dstva;  curenv->env\_status = ENV\_NOT\_RUNNABLE;//停止接收进程的运行  sched\_yield(); //转系统调度  return 0;  } |

最后就是实现函数ipc\_recv以及ipc\_send了。这两个函数是上面系统调用函数的封装，直接给出代码，详情见注释

|  |
| --- |
| int32\_t  ipc\_recv(envid\_t \*from\_env\_store, void \*pg, int \*perm\_store)  {  int r;  if(pg == NULL) pg = (void\*)UTOP;  if((r = sys\_ipc\_recv(pg) < 0)){ //接收失败  if(from\_env\_store) \*from\_env\_store = 0; //将参数置0  if(perm\_store) \*perm\_store = 0;  return r;  }  //接收成功，填入适当参数  if(from\_env\_store) \*from\_env\_store = thisenv->env\_ipc\_from;  if(perm\_store) \*perm\_store = thisenv->env\_ipc\_perm;  return thisenv->env\_ipc\_value;//返回接收的value  }  void  ipc\_send(envid\_t to\_env, uint32\_t val, void \*pg, int perm)  {  if(pg==NULL) pg=(void\*)UTOP;  int ok=0,r;  while(!ok){  r = sys\_ipc\_try\_send(to\_env,val,pg,perm);  if(r==0) ok=1; //发送成功  else if(r==-E\_IPC\_NOT\_RECV){//接收进程还未运行  sys\_yield();//系统调度运行目标进程  }  else //出错  panic("ipc\_send:%e %d\n",r,r);  }  } |

最后运行本实验的评分脚本：

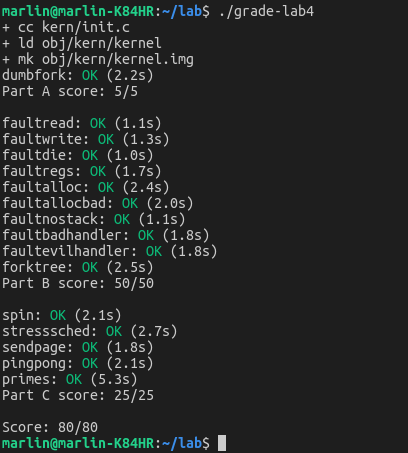


图4-5 实验四测试结果

#### **3 实验问题**

1. Compare kern/mpentry.S side by side with boot/boot.S. Bearing in mind that kern/mpentry.S is compiled and linked to run above KERNBASE just like everything else in the kernel, what is the purpose of macro MPBOOTPHYS? Why is it necessary in kern/mpentry.S but not in boot/boot.S? In other words, what could go wrong if it were omitted in kern/mpentry.S?

Hint: recall the differences between the link address and the load address that we have discussed in Lab 1.

宏MPBOOTPHYS用来处理LVM与RVM不同所带来的问题，由于汇编指令运行在虚拟地址空间，但是又被装载到低物理地址处，造成两者不同，因此为了使汇编指令能顺利运行，需要将指定中所有的地址改写为物理地址否则将会导致地址相关指令（如ljump)运行出错。

1. It seems that using the big kernel lock guarantees that only one CPU can run the kernel code at a time. Why do we still need separate kernel stacks for each CPU? Describe a scenario in which using a shared kernel stack will go wrong, even with the protection of the big kernel lock.

当中断产生时，在执行中断入口过程中尚未加锁，正式加锁的地方是在trap函数中。因此，如果共用内核栈，当两个进程同时发生中断时，一个进程可能先将现场寄存器信息压人内核栈然后执行中断汇编入口跳到trap,但此时尚未加锁，另一个进程也会将现场压栈导致第一个进程的现场被破坏。

1. In your implementation of env\_run() you should have called lcr3(). Before and after the call to lcr3(), your code makes references (at least it should) to the variable e, the argument to env\_run. Upon loading the %cr3 register, the addressing context used by the MMU is instantly changed. But a virtual address (namely e) has meaning relative to a given address context--the address context specifies the physical address to which the virtual address maps. Why can the pointer e be dereferenced both before and after the addressing switch?

回答这个问题需要回顾用户页目录的映射关系，在env\_setup\_vm中，我们以kern\_pgdir为模板直接复制了其地址空间[UTOP,0xffffffff]的页目录项，而参考memlayout.h可知UENV位于此空间中，所以用户页目录与kernel页目录对指针e有相同的映射关系，即在切换页目录后e还是指向同一个物理地址。

1. Whenever the kernel switches from one environment to another, it must ensure the old environment's registers are saved so they can be restored properly later. Why? Where does this happen?

进程切换依赖于函数sys\_yield，此函数是系统功能调用，会先进入内核态执行调度函数选择其它进程运行。陷入内核时会将当前进程的现场信息（各寄存器值）压入内核栈，执行完系统调用后会将现场重新恢复到env程序控制块中并执行env\_run回到用户态。