

**操作系统原理课程设计报告**

|  |  |
| --- | --- |
| 姓 名： |  |
| 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： |  |
| 指导教师： |  |

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

年 月 日

### 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

（记录完成所有Exercise实验过程中的主要步骤内容，或者自己认为重要的部分内容）

##### 2.1 Part 1: PC Bootstrap

###### 2.1.1 Exercise 1

Exercise 1. Familiarize yourself with the assembly language materials available on the 6.828 reference page. 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. It gives a good (and quite brief) description of the AT&T assembly syntax we'll be using with the GNU assembler in JOS.

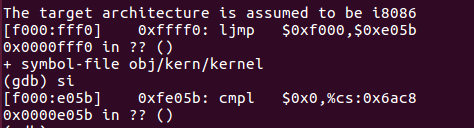
这个Exercise是一些阅读任务，可以花一部分时间复习一下AT&T格式的汇编并学习一下gcc中的内联汇编。

###### 2.1.2 Exercise 2

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, as well as other materials on the 6.828 reference materials page. No need to figure out all the details - just the general idea of what the BIOS is doing first.

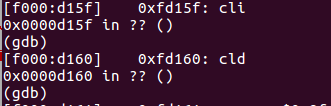
这个任务是通过单步执行程序去体会BIOS大体做了些什么。

Bias前1M空间较高的地址处开始执行（0xffff0)，第一步先做了一个跳转

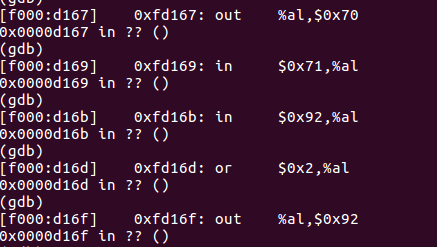


并且才开始CPU运行在实模式下，这么做只是为了实现所谓的向后兼容。

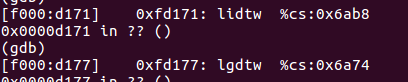
接下来是cli关中断，cld清零方向标志位，这么做是为了之后的串操作。



之后是一连串的IO操作，其中$0x71,$0x92都是表示的外部设备的端口号，可以通过查询相应的手册来了解其端口所代表的内容。应该就是对各个硬件进行检测。



之后是lidtw指令和lgdtw指令



lidt指令加载中断向量表寄存器(IDTR)，这个指令把从偏移地址0x6eb8起始的后面6个字节的数据读入到IDTR中。lgdtw指令把从偏移地址0x6a74起始的6个字节的值加载到GDTR中，为实现保护模式做准备。

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

当BIOS完成自检，初始化设备工作后，就开始进行系统引导，将引导程序装载入物理内存0x7c00处开始执行。

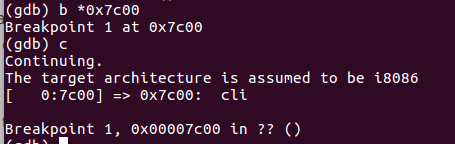
###### 2.2.1 Exercise 3

Exercise 3. Take a look at the lab tools guide, 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.

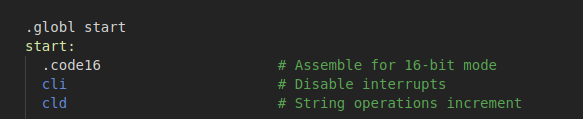
Set a breakpoint at address 0x7c00, which is where the boot sector will be loaded. Continue execution until that breakpoint. Trace through the code in boot/boot.S, using the source code and the disassembly file obj/boot/boot.asm to keep track of where you are. Also use the x/i command in GDB to disassemble 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.

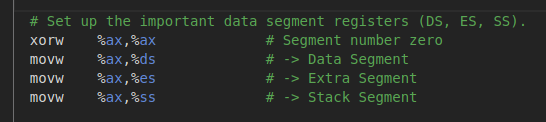
先学习完lab tools guide，在地址0x7c00处打断点，可以看出，这时候还是实模式。



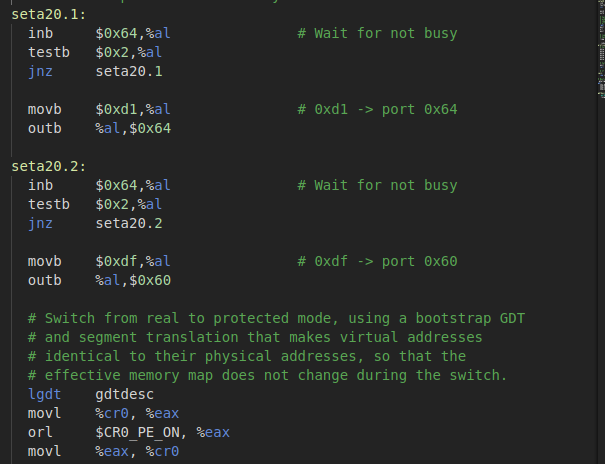
接着便是要我们去读汇编代码。



Cli关中断，cld为串操作连续读取做准备。



接着将三个ds、es、ss寄存器清零。接下来一部分可以说是为跳转到保护模式做准备了。



这个地方可以稍微仔细地进行分析，翻阅手册(http://bochs.sourceforge.net/techspec/PORTS.LST)可以得到相应的端口的含义。

0064 r KB controller read status (ISA, EISA)

bit 7 = 1 parity error on transmission from keyboard

bit 6 = 1 receive timeout

bit 5 = 1 transmit timeout

bit 4 = 0 keyboard inhibit

bit 3 = 1 data in input register is command

0 data in input register is data

bit 2 system flag status: 0=power up or reset 1=selftest OK

bit 1 = 1 input buffer full (input 60/64 has data for 8042)

bit 0 = 1 output buffer full (output 60 has data for system)

所以以上指令通过不断检测bit1，判断缓冲区是否满了，只有这一位为0,程序才能继续向后运行。

0064 w KB controller input buffer (ISA, EISA)

D1 dbl write output port. next byte written to 0060

will be written to the 804x output port; the

original IBM AT and many compatibles use bit 1 of

the output port to control the A20 gate.

Compaq The system speed bits are not set by this command

use commands A1-A6 (!) for speed functions.

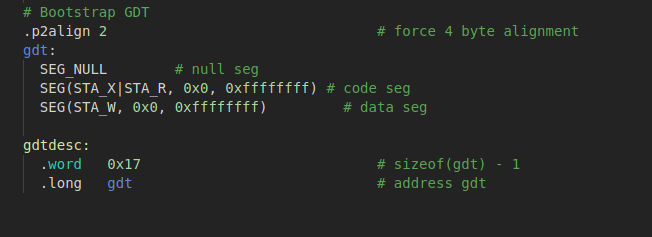
D1大概可以看出是个与0x60端口相关的控制信号。

前面分析了这么多，事实上后面的0x60端口写入0xdf才是关键所在，可以看下df所代表的值的特征：

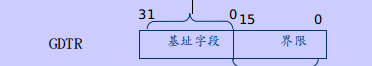
DF sngl enable address line A20

即使能A20线，这大概是进入保护模式的前提所在。

lgdt设置全局描述符，把gdtdesc的值送入GDTR中，文件末尾有GDT表的设置。



GDTR结构如下



可以看出来，gdtdesc的低16个字节正好表示表的长度，高32位表示该表在内存起始地址。可以循着看到我们所关心的gdt表就是gdt:标号之后那三行。

可以看到那三行都是用SEG宏表示的，至于SEG宏定义在文件mmu.h中，可以看出的是在jos系统中事实上是没有采用分段机制的，也就是说数据和代码段的起始地址都是0,大小都是4GB。

接着设置cr0中的使能信号，随后ljmp跳转进入了32位段保护模式。进入32位模式之后，设置段寄存器和栈顶指针，接着便call bootmain进入C代码中。

Boot main所做的工作就是将磁盘的第二个扇区所存放的kernel镜像加载进内存。

其做的第一个工作就是调用readseg函数将一个页的大小读入内存中（从第一块扇区开始读，读入到内存的0x100000处），而后进行检查，判断该文件是否是ELF格式。接着读入程序头表，读入的细节在之后关于loading the kernel的部分进行解释。

###### 2.2.2 Question

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

答：这题问的正是何时从实模式到保护模式切换，在boot.S文件中，计算机首先工作于实模式，ljmp一句话跳转切换到32位段。切换的方法可见2.2.1的分析。

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

答：boot loader执行的最后一条指令是boot/main.c中的最后一条语句”((void (\*)(void)) (ELFHDR->e\_entry))();”跳转到kernel的起始处。

第一条指令位于kernel的entry.S文件中: “movw $0x1234,0x472”;

1. Where is the first instruction of the kernel?

答：第一条指令是”movw $0x1234, 0x472”

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?

答：关于boot loader如何决定它要读入多少个扇区，应该是通过读入的ELF头，定位到程序头表处，然后根据程序头表来决定从kernel磁盘映像什么地方开始装多少数据到内存。

###### 2.2.3 Exercise 4

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) or find one of MIT's 7 copies.

Read 5.1 (Pointers and Addresses) through 5.5 (Character Pointers and Functions) in K&R. Then download the code for 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 that cites K&R heavily), though not as strongly recommended.

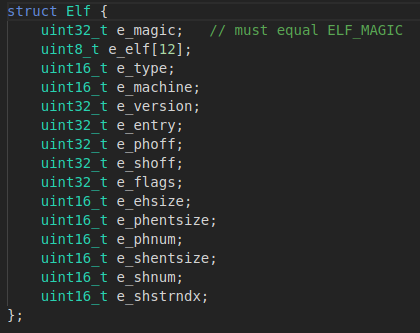
Warning: Unless you are already thoroughly versed in C, do not skip or even skim this reading exercise. If you do not really understand pointers in C, you will suffer untold pain and misery in subsequent labs, and then eventually come to understand them the hard way. Trust us; you don't want to find out what "the hard way" is.

这个练习是要求我们了解C语言及其指针，由于C语言比较熟悉了，故跳过。

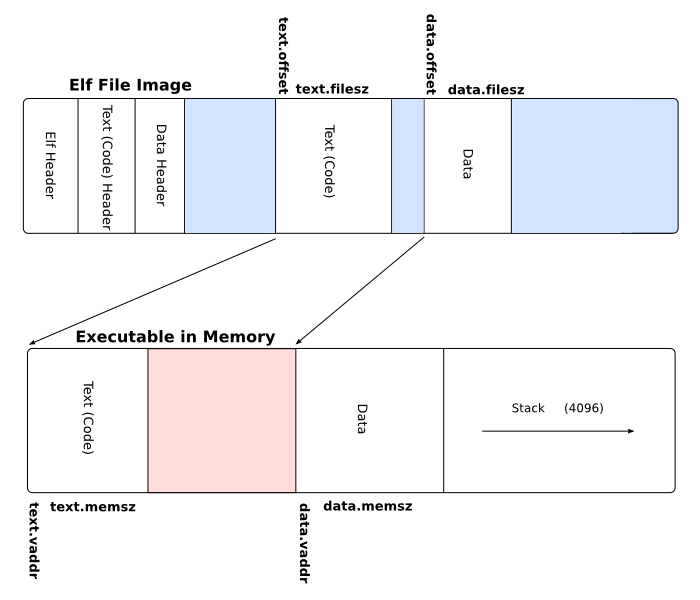
###### 2.2.4 ELF

由于实验做到这一步，涉及到了些ELF及其加载的内容，所以专门开了一个小节进行讲解。

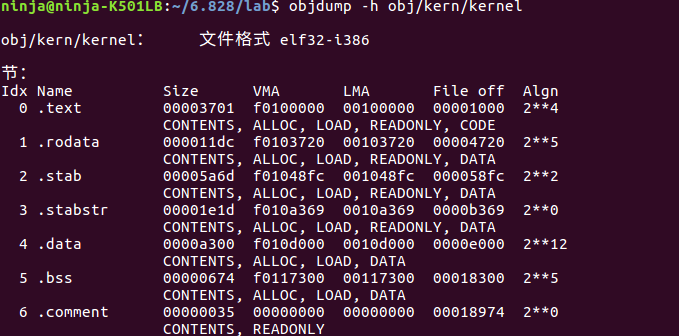
该实验32位系统中的ELF相关结构体定义在elf.h中，ELF头是固定长度的，其结构体定义为



读入与加载ELF文件的步骤是：先读ELF头，ELF头总是在ELF文件的开头，并且有整个文件中其余部分的布局信息。然后便可定位到程序头，这里存放着text和data节的布局（哪部分加载入内存）。一个典型的ELF加载过程如图：



通过objdump -h obj/kern/kernel可以观察kernel的节头表如下

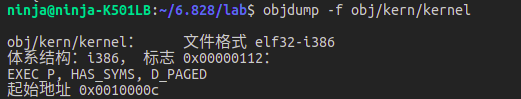


可以发现其VMA（链接地址）与LMA（加载地址）之间差了0xf0000000;

objdump -x查看全部信息



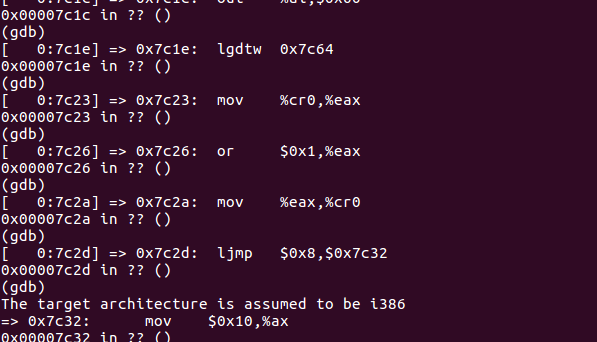
通过objdump -f obj/kern/kernel可以观察到kernel对应的开始执行的链接地址e\_entry = 0x10000c。



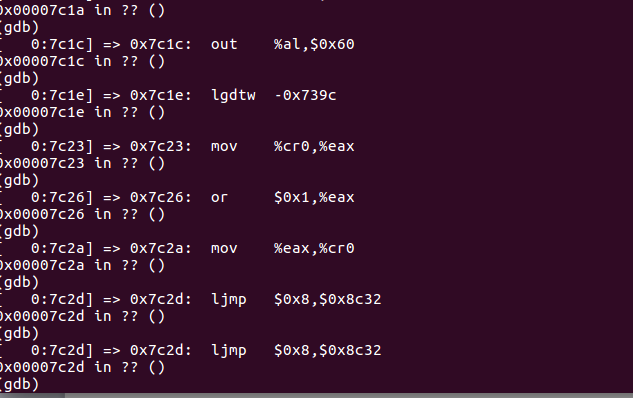
###### 2.2.5 Exercise 5

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!

修改之前，链接地址为0x7c00，结果如图所示：



修改了链接地址之后，即在boot/Makefrag中将0x7c00改为0x8c00

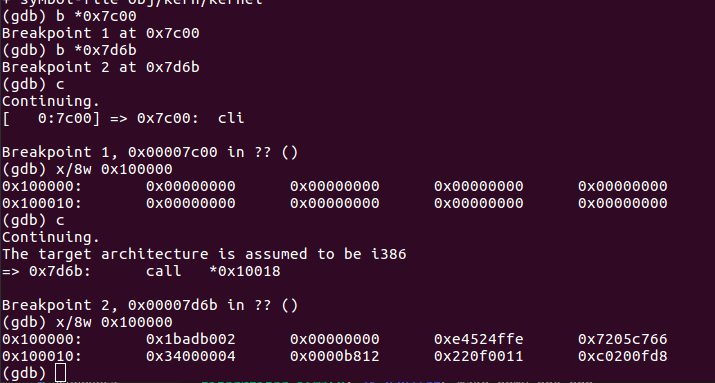


可以看出，执行到跳转这一步的时候，跳转至了别的地方，正是由于我们的链接地址进行了改变。

###### 2.2.5 Exercise 6

Exercise 6. We can examine memory using GDB's x command. The GDB manual has full details, but for now, it is enough to know that the command x/Nx 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.)



刚刚进入boot loader的时候0x100000处的内存的值为0,而加载进内核之后便有了值。

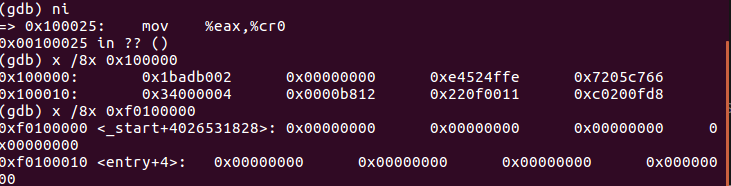
##### **2.3 Part 3: The Kernel**

###### 2.3.1 Exercise 7

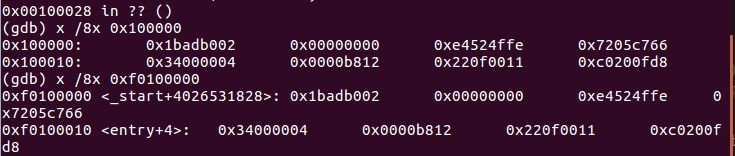
Exercise 7. Use QEMU and GDB to trace into the JOS kernel and stop at the movl %eax, %cr0. Examine memory at 0x00100000 and at 0xf0100000. Now, single step over that instruction using the stepi GDB command. Again, examine memory at 0x00100000 and at 0xf0100000. Make sure you understand what just happened.

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.

执行到mov %eax, %cr0之前，可以看出这个时候还没有进行分页，由于前面的虚拟地址也没有进行分段，所以事实上这个虚拟地址就是物理地址，两个物理地址不同的地方内存内容当然不一样。



继续向后执行一条语句，由于这个时候已经开启了分页，所以虚拟地址0xf0100000便映射到了物理地址0x100000，如图。



###### 2.3.2 Exercise 8

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.

主要是观察文件kern/printf.c, lib/printfmt.c和 kern/console.c，要完成这个练习只需要照着上面的case ‘u’依葫芦画瓢即可。

###### 2.3.3 Question

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?

答：这题是要解释printf.c与console.c之间的接口关系，可以发现printf.c中的函数是通过调用console.c中的putch函数是实现的。

2.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 }

答：这个代码实现的是当屏幕输出内容满了的时候，整体向上移动进行更新。

3.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:

int x = 1, y = 3, z = 4;

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?
* 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.

答：fmt指向格式字符串，ap指向va\_list传入的参数。列举部分进行了跳过。

4.Run the following code.

unsigned int i = 0x00646c72;

cprintf("H%x Wo%s", 57616, &i);

答：机器是小端，会打印He110 World.

5.In the following code, what is going to be printed after 'y='? (note: the answer is not a specific value.) Why does this happen?

cprintf("x=%d y=%d", 3);

答：y是未定的，因为就简单的执行来说，%d取得3这个参数之后，y的值是其后一个内存单元的内容。

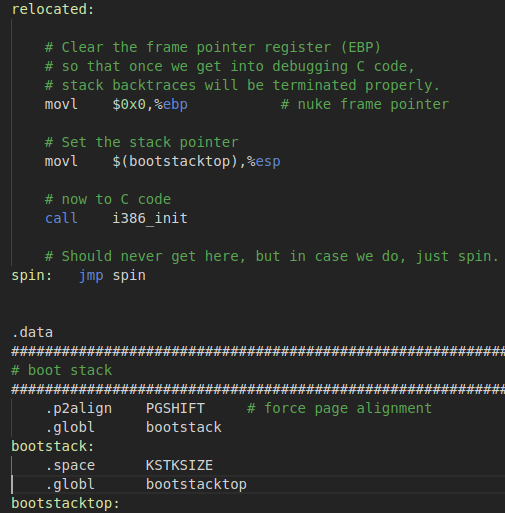
6.Let's say that GCC changed its calling convention so that it pushed arguments on the stack in declaration order, so that the last argument is pushed last. How would you have to change cprintf or its interface so that it would still be possible to pass it a variable number of arguments?

答：gcc调用方式如果改成了顺序压栈，由于ap是递增的，所以cprintf中格式匹配应该反过来，从后开始。

###### 2.3.4 Exercise 9

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?

栈的初始分配代码



Kernel在movl $(bootstacktop), %esp这一句初始化栈，esp初始指向栈的高地址(bootstacktop)，栈的保留空间大小为KSTKSIZE。

###### 2.3.5 Exercise 10

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 page or on Athena. Otherwise, you'll have to manually translate all breakpoint and memory addresses to linear addresses.

该部分考察的是C语言栈帧结构，在csapp实验中已经相当熟悉了，故跳过。

###### 2.3.6 Exercise 11、12

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.

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.

Backtrace过程可以通过回溯ebp实现：ebp = (uint32\_t\*)ebp[0]，根据当前eip，结合stab内容可以输出调试的字符信息。将其挂载到命令行菜单，只需要在commands结构体数组中添加相应项即可。

### Lab2 Memory Management

#### **1 实验目的**

1.熟悉MITJOS的内存组织结构

2.实现JOS的内存分配

3.理解虚拟内存和物理内存，理解两者的映射关系

4.实现JOS的页式内存管理

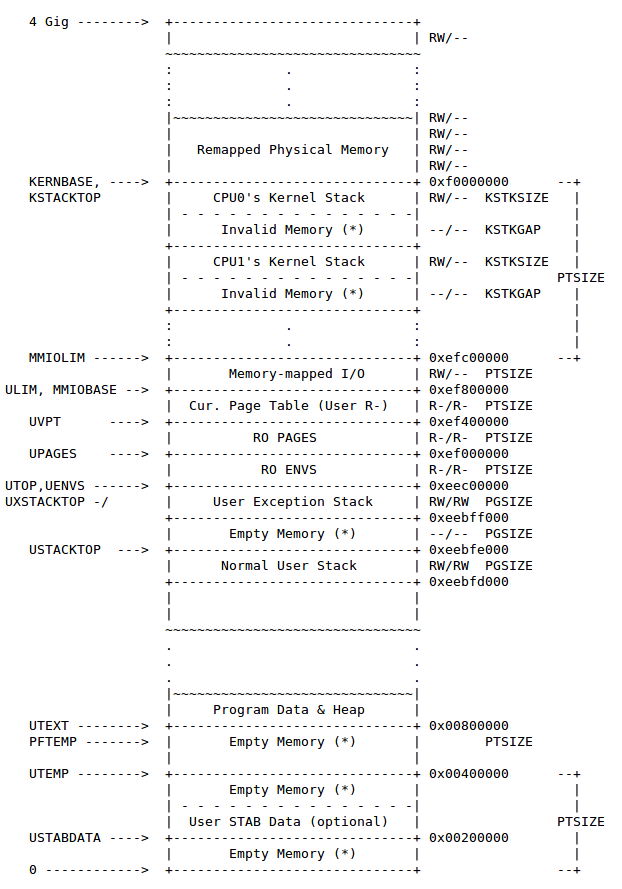
#### **2 实验内容**

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

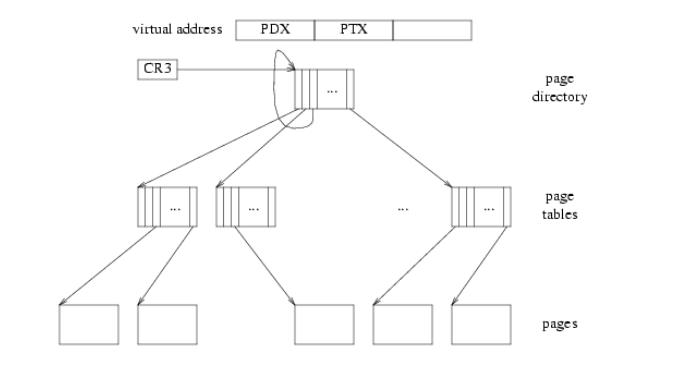
##### 2.1.1 引入

JOS在boot loader中实现了分段，在kernel中实现了分页，在lab1中采用的分页是极其简单的分页，只是个单级的页表结构，所以在lab2中主要目的就是构建二级页表结构。

其中JOS的虚拟内存分布如下：



其分页机制如下



###### 2.1.2 Exercise 1

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

boot\_alloc()

mem\_init() (only up to the call to check\_page\_free\_list(1))

page\_init()

page\_alloc()

page\_free()

check\_page\_free\_list() and check\_page\_alloc() test your physical page allocator. You should boot JOS and see whether check\_page\_alloc() reports success. Fix your code so that it passes. You may find it helpful to add your own assert()s to verify that your assumptions are correct.

下面就各个函数的实现细节进行讲解。

2.1.2.1 boot\_alloc()

该函数的nextfree为static变量，只在函数第一次进入时值为0,代表的含义为指向下一个空内存的虚拟地址指针。end为链接器通过ld文件所定义的kernel的bss段的尾后指针，事实上就指向kernel加载入内存后的最高地址的后一字节地址。通过ROUNDUP函数进行按页对齐，值得注意的是每次分配是按页进行分配，即每次都要保证按页对齐。如果超过最大分配区域0x400000则报错提示。其中得注意0x400000代表的是物理地址，得进行转换。

2.1.2.2 mem\_init()

该函数的作用是初始建立内核部分的地址空间，即从UTOP往上的空间。该函数第一步是先检测物理内存大小和基础内存大小。然后分配第一个页面作为页目录表，并为该表建立相应的访问权限。接下来分配区域用来记录空闲页面，分配npages个struct PageInfo结构体。接下来调用page\_init函数对这些结构体进行初始化操作。至此，part 1的mem\_init部分完结。

2.1.2.3 page\_init()

用链表的结构将对应空闲页的那些结构体串起来，引用计数为0。

2.1.2.4 page\_alloc()

page\_alloc的作用为从空闲page中分配一个page，该函数事实上就是简单的取链表首元素。

2.1.2.5 page\_free()

page\_free的作用是回收page，要求待回收的page的引用计数为0且指针域为空，之后将其插入空闲页表中。

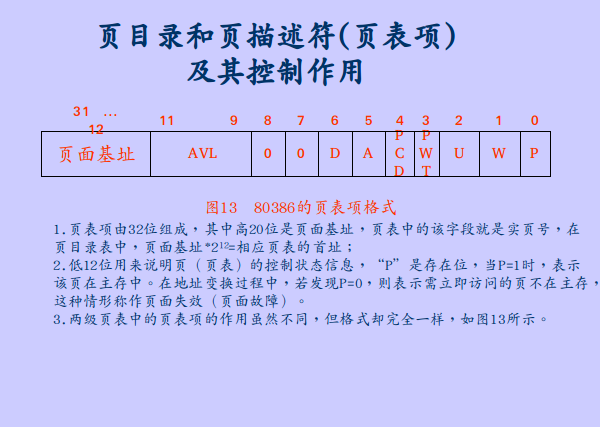
##### **2.2 Part 2: Virtual Memory**

###### 2.2.1 Excercise 2

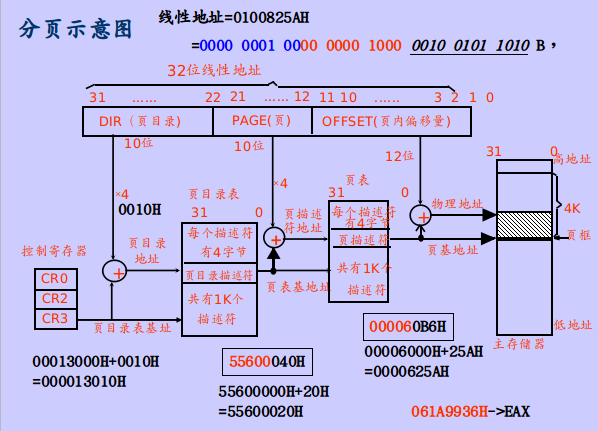
Exercise 2. Look at chapters 5 and 6 of the Intel 80386 Reference Manual, if you haven't done so already. Read the sections about page translation and page-based protection closely (5.2 and 6.4). We recommend that you also skim the sections about segmentation; while JOS uses the paging hardware for virtual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

这个练习要我们熟悉分页的基本方式，只需注意两个内容：

页描述符



分页变换方式



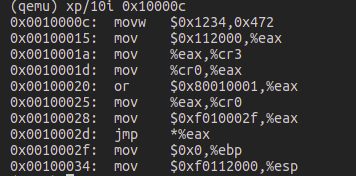
###### 2.2.2 Exercise 3

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

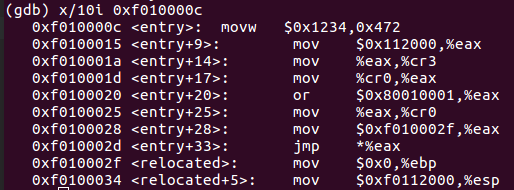
Use the xp command in the QEMU monitor and the x command in GDB to inspect memory at corresponding physical and virtual addresses and make sure you see the same data.

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

由于qemu中xp显示的是实地址，gdb中x显示的是虚拟地址，所以xp显示0x10000c处内存内容应该与x显示0xf010000c处内容相等，经过检验确实一样。



qemu xp



gdb x

###### 2.2.3 Question 1

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

mystery\_t x;

char\* value = return\_a\_pointer();

\*value = 10;

x = (mystery\_t) value;

这个其实前文已经提示了很多，开启分页之后代码中出现指针的值，不进行转换的情况下一般指的是虚拟地址。

###### 2.2.4 Exercise 4

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

pgdir\_walk()

boot\_map\_region()

page\_lookup()

page\_remove()

page\_insert()

check\_page(), called from mem\_init(), tests your page table management routines. You should make sure it reports success before proceeding.

下面就各个函数的细节进行分析

2.2.4.1 pgdir\_walk()

该函数接受一个指向页目录的指针，返回指向页表项的指针。

还是和之前一样，该指针得是虚拟地址，即得用KADDR进行转换。可以通过pde & PTE\_P进行判断，如果为1,代表所在页在内存，返回对应的page table entry所在地址；否则进行页表分配，再进行返回。

2.2.4.2 boot\_map\_region()

该函数的作用是将[va, va+size)虚拟地址区域映射到[pa, pa+size)物理地址区域，其中size,pa,va都是按页对齐。其关键是调用pgdir\_walk()函数完成相关映射功能。

2.2.4.3 page\_lookup()

page\_lookup实现根据虚拟地址查找对应的struct PageInfo结构体所在地址，如果没有找到返回NULL。需要注意的是调用pgdir\_walk没有对应的应该直接返回，所以create应该为false。

2.2.4.4 page\_remove()

page\_remove函数先根据虚拟地址进行对应的page查找，如果该page是存在的，则引用计数减1,减为0则释放，同时将对应的pte置为0.

2.2.4.5 page\_insert()

page\_insert函数实现增加一组虚拟地址与物理地址的映射。如果该虚拟地址已经映射到了某个物理地址，则执行page\_remove操作将之前的映射关系移除，然后重新定义映射关系。

2.2.4.6 mem\_init()

可以继续完成mem\_init的检测部分,主要是调用boot\_map\_region进行虚拟地址空间与物理地址空间的映射。

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

###### 2.3.1 Exercise 5

已经在part1 和 part2中将其完成。

###### 2.3.2 Question 2

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

在mem\_init函数中进行了分页方式的切换，从小页表切换到大页表，这里问的是切换之后的页表映射的虚拟地址。

|  |  |  |
| --- | --- | --- |
| Entry | Base Virtual Address | Points to (logically): |
| 1023 | 0xffc00000 | Page table for top 4MB of phys memory |
| ... | ... | ... |
| 960-1023 | 0xf0000000-0xffffffff | kernel |
| 959 | efff8000 | Kern stack |
| 957 | 0xef400000 | Kern\_pgdir |
| 956 | 0xef000000 | PageInfo struct array |
| 1 | 0x00400000 | ? |
| 0 | 0x00000000 | [see next question] |

###### 2.3.4 Question 3

We have placed the kernel and user environment in the same address space. Why will user programs not be able to read or write the kernel's memory? What specific mechanisms protect the kernel memory?

答：一般操作系统可以进行段保护也可以进行页保护，但是JOS中事实上是没有进行分段的，所以是通过分页的机制来实现的，通过设置权限位可以限制用户的权限。

###### 2.3.5 Question 4

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

答：JOS操作系统利用一个大小为4MB的空间UPAGES来存放PageInfo结构体，而这个结构体与页是一一对应的关系，由于每个结构体大小为8B，所以一共可以存放512K个这样的结构体，即512K个物理页，每个物理页大小为4KB，总的物理内存大小为4KB \* 512K = 2GB。

###### 2.3.6 Question 5

How much space overhead is there for managing memory, if we actually had the maximum amount of physical memory? How is this overhead broken down?

答：这问得是事实上需要多少的物理内存开销用于管理，4MB描述器，512个页表需要2MB空间，还有一个4K大小的页目录，所以总共需要6MB+4KB空间。

###### 2.3.7 Question 6

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

答：在entry.S文件中执行完jmp \*%eax，这个指令执行跳转，重新设定EIP的值于KERNBASE之上。在entry\_pgdir页表中，将虚拟地址

### Lab3 User **Environments**

#### **1 实验目的**

1. 了解建立用户环境的条件；
2. 了解中断检测，中断处理的细节；
3. 实现用户进程运行需要的受保护环境以及相关基础例程和数据结构；
4. 了解异常和中断，在JOS中实现中断机制。

#### **2 实验内容**

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

用户环境管理事实上就是用户进程管理，有三个重要的全局变量：

struct Env \*envs = NULL; // All environments

struct Env \*curenv = NULL; // The current env

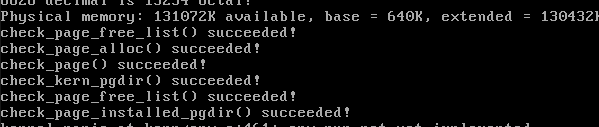
static struct Env \*env\_free\_list; // Free environment list

###### 2.1.1 Exercise 1

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.

Exercise 1较为简单，基本步骤和lab2差不多，assert成功截图：



###### 2.1.2 Exercise 2

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".

这部分和一般的链接不一样，利用-b参数直接将文件原生的进行链接，这些.o文件事实上没有不是ELF镜像。下面就完成的各个函数细节进行讲解：

2.1.2.1 env\_init()

这个函数主要是将所有的环境结构体加入空闲链表，注意的一点是要求ENV结构数组的顺序要和链表的顺序一致，所以采用先进后出方式构造链表的话，应该从数组尾部开始链接进链表。

2.1.2.2 env\_setup\_vm()

该函数主要是为用户环境分配页目录表，建立分页，该页目录表分配的时候注意需要增加其引用计数，这在用户的分配中相当重要，因为内核之前的分配一开始分配之后不会释放，而用户分配的很有可能会经常分配与释放，所以需调整正确的引用计数。需要注意得自己指向自己，这已经帮我们完成了：

e->env\_pgdir[PDX(UVPT)] = PADDR(e->env\_pgdir) | PTE\_P | PTE\_U;

2.1.2.3 region\_alloc()

用户进程分配完成页目录之后，便可以分配物理页面，并建立虚拟页面与物理页面的映射关系。这个时候，之前写的page\_insert就可以起到作用了，利用page\_insert将虚拟地址与pageinfo对插入到页表中。要注意，待分配的虚拟地址不一定是页面对齐的参数，所以需要进行稍微处理。该函数相当于用户请求给某个虚拟地址分配相应的物理空间。

2.1.2.4 load\_icode()

这正是之前所提到的，直接从ELF镜像中加载一些片段来初始化用户环境，加载到的虚拟内存地址所在位置（该位置由程序头表中的p\_va决定）。大概分为几个步骤，首先lcr3(PADDR(e->env\_pgdir));切换到当前环境的页目录表，然后加载对应的p\_type = ELF\_PROG\_LOAD的segment进入内存，最后分配一个页面大小的用户栈空间。这中间保留env\_tf.tf\_eip为e\_entry要注意不能漏掉。

2.1.2.5 env\_create()

这个函数的作用是分配一个新的用户环境，作用的时间是在内核初始化阶段。

其中调用env\_alloc函数进行分配，可以看看env\_alloc函数做了些什么。

env\_alloc函数从空闲环境列表中取下首元素：

if (!(e = env\_free\_list))

return -E\_NO\_FREE\_ENV;

并且为该环境初始化页目录：

if ((r = env\_setup\_vm(e)) < 0)

return r;

并初始化了很多环境属性。

env\_create函数也是只在内核初始化的时候调用，并利用load\_icode将用户程序载入。

2.1.2.6 env\_run()

env\_run函数和上一个函数env\_create不同，env\_create是在内核初始化的时候运行来加载用户环境的，而env\_run是在用户环境中切换用的，如果不是首次执行的用户进程，那么将之前的进程设置为RUNNABLE状态，而将e设为RUNNING，这事实上是就是UNIX中进程就绪态与运行态的切换。并且切换环境地址空间。其中寄存器的切换需要调用env\_pop\_tf函数，由于涉及到寄存器，得用汇编语言级别才能直接调用：

asm volatile(

"\tmovl %0,%%esp\n"

"\tpopal\n"

"\tpopl %%es\n"

"\tpopl %%ds\n"

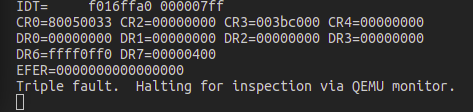
"\taddl $0x8,%%esp\n" /\* skip tf\_trapno and tf\_errcode \*/

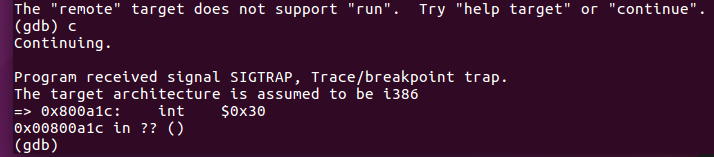
"\tiret\n"

: : "g" (tf) : "memory");

可以大概看下这条内联汇编语句的作用，先是将tf赋值给esp，tf中事实上储存这寄存器的值，于是进行pop操作将寄存器进行切换，最后iret返回，利用iret可以切换标志寄存器的值。

完成了这些函数之后，可以看到已经切换到了用户环境，但是当系统还没有建立硬件措施让其从用户环境切换到内核环境，所以会发生错误，引发三重异常，如图：





###### 2.1.3 Exercise 3

Exercise 3. Read Chapter 9, Exceptions and Interrupts in the 80386 Programmer's Manual (or Chapter 5 of the IA-32 Developer's Manual), if you haven't already.

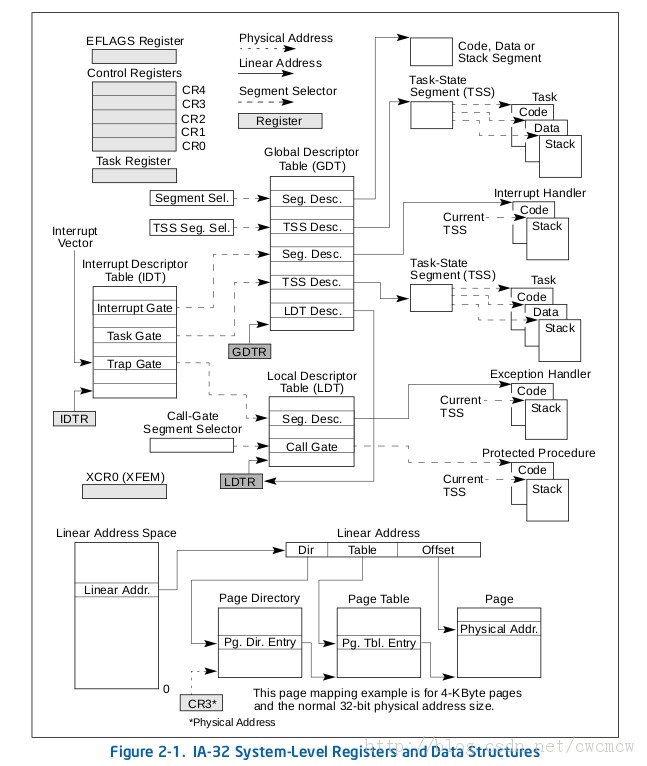
这个练习可以读一下关于中断与异常的知识，为后面实验做好准备。该实验是按照intel的那一套来模拟实现的。

Intel关于中断的定义：an interrupt is a protected control transfer that is caused by an asynchronous event usually external to the processor, such as notification of external device I/O activity.

关于异常的定义：An exception, in contrast, is a protected control transfer caused synchronously by the currently running code, for example due to a divide by zero or an invalid memory access.

这个定义和课内的有点小的区别，之前学的是中断包括异常，而这里中断只包括外部中断。而且我们知道，用户态与核心态的切换由x86硬件完成，这里是模拟了一个虚拟的环境。

实验文中提出了两个重要的机制：中断描述符表和任务状态段(TSS)，并着重讨论了TSS的存放结构。



###### 2.1.4 Exercise 4

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

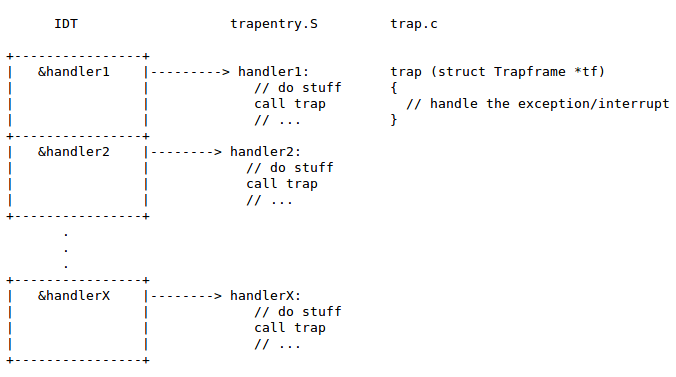
3.pushl %esp to pass a pointer to the Trapframe as an argument to trap()

4.call trap (can trap ever return?)

Consider using the pushal instruction; it fits nicely with the layout of the struct Trapframe.

这个练习首先要求我们在trapentry.S中增加中断的入口点，这需要查阅相应的intel手册，根据是否需要error\_code来决定需要使用哪个宏。

Trapentry.S构造目标如图：



一开始trapentry.S的压栈方式没太看明白（比如说没有压入error code等等东西），这点还是比较重要的，在这个部分得区分硬件压栈和软件压栈的区别，事实上，文中说的那些保护压栈都是由x86的硬件自动是实现的，而不是我们在中断程序中进行软件的实现。可以结合Trapframe结构体一看，这很重要：

struct Trapframe {

struct PushRegs tf\_regs;

uint16\_t tf\_es;

uint16\_t tf\_padding1;

uint16\_t tf\_ds;

uint16\_t tf\_padding2;

uint32\_t tf\_trapno;

/\* below here defined by x86 hardware \*/

uint32\_t tf\_err;

uintptr\_t tf\_eip;

uint16\_t tf\_cs;

uint16\_t tf\_padding3;

uint32\_t tf\_eflags;

/\* below here only when crossing rings, such as from user to kernel \*/

uintptr\_t tf\_esp;

uint16\_t tf\_ss;

uint16\_t tf\_padding4;

} \_\_attribute\_\_((packed));

注释中就已经说明清楚了哪部分是硬件完成，哪部分是软件完成，当然我们所要写的就是Trapframe结构中软件部分。需要注意的是，Trapframe结构体中总有tf\_err域，而硬件并不是都会压入error code，所以这就是TRAPHANDLER\_NOEC中pushl $0的作用，作为无用的占位符。

接下来便可按照说明一句一句的完成，最后

pushl %esp

call trap

程序跳转到trap函数入口处，并且传递参数%esp，这就是中断向量表的内容。

trap\_init用来初始化中断向量表（指定中断向量表的位置），要完成trap\_init函数，得先对gdt的工作原理有一定的了解。

gdt表的定义可以在env.c中看到，jos中gdt表只用来进行权限控制：

struct Segdesc gdt[] =

{

// 0x0 - unused (always faults -- for trapping NULL far pointers)

SEG\_NULL,

// 0x8 - kernel code segment

[GD\_KT >> 3] = SEG(STA\_X | STA\_R, 0x0, 0xffffffff, 0),

// 0x10 - kernel data segment

[GD\_KD >> 3] = SEG(STA\_W, 0x0, 0xffffffff, 0),

// 0x18 - user code segment

[GD\_UT >> 3] = SEG(STA\_X | STA\_R, 0x0, 0xffffffff, 3),

// 0x20 - user data segment

[GD\_UD >> 3] = SEG(STA\_W, 0x0, 0xffffffff, 3),

// 0x28 - tss, initialized in trap\_init\_percpu()

[GD\_TSS0 >> 3] = SEG\_NULL

};

其中SEG传入的4个参数为：

SEG(type, base, lim, dpl)

总的中断流程应该是：

文中对这方面讲解不是很详细，课上也只学了中断向量表，感到有点困惑，于是查阅了相关资料：

系统在一开始实模式中由BIOS设置IVT，即中断向量表，这便是课上学的：在实模式下，DITR.base 指向的表格项直接给出中断服务例程（Interrupt Service Routine）的入口地址。而这个地方讨论的是保护模式下的，保护模式下不再用IVT，而是用IDT：在保护模式下，并不直接给出入口地址，而是门描述符（Interrupt/Trap/Task gate），从这些门描述符间接取得中断服务例程入口。jos模拟的x86，有256个中断存在，可见jos代码：

struct Gatedesc idt[256] = { { 0 } };

这里重点讨论的保护模式，定位并初始化的过程为：

称IDTR.base为IDT表首地址（即为JOS中idt数组首址）

从IDTR.base+vector \* 8处读取8bytes宽的description，即idt[vector]，将其初始化（调用SETGATE宏），SETGATE对于中断传入参数基本是类似的，段选择子(sel)传入内核段--GD\_KT，off即为handler偏移;中断与异常会有区别：The difference between an interrupt gate and a trap gate is in the effect on IF (the interrupt-enable flag).这对于interrupt来说是很重要的，因为其要在切换到handler的过程中关中断，防止打断，所以要istrap进行区分，而这全是内部异常，所以is\_trap字段都填1。DPL这一项相当于是做了一个检查，只有Max(CPL,RPL) <= DPL的才能通过这个gate。

弄明白了之后，填好就行了。

###### 2.1.5 Questions

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?)

答：如果所有的异常和中断处理都使用相同的处理函数，会使得这个函数很复杂，而且这样中断扩充相对更麻烦。

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)?

答：因为这个中断的DPL是0,用户态通过软中断进行调用的话会产生页保护异常。

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

###### 2.2.1 Exercise 5 & 6

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.

这两个exercise较简单，对tf->tf\_trapno进行判断调用相应的中断处理函数即可。

###### 2.2.2 Questions

3. 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?

答：在转到IDT的时候，会进行DPL检验，用户的CPL是3，如果DPL是0，切换就会产生保护错误，只有DPL是3才能通过。

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

答：这些机制的关键就是将用户态和核心态进行区分，实现权限保护的作用。

###### 2.2.3 Exercise 7

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.

要完成这一问，题目建议我们先看懂lib/syscall.c/syscall()函数是做了什么。

这个很简单，看一下主体的内联汇编代码：

asm volatile("int %1\n"

: "=a" (ret)

: "i" (T\_SYSCALL),

"a" (num),

"d" (a1),

"c" (a2),

"b" (a3),

"D" (a4),

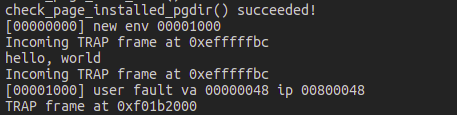
"S" (a5)

: "cc", "memory");

这段代码的作用是int $(T\_SYSCALL)，并且将num(%eax)是系统调用号;a1,a2,a3,a4,a5放入相应的寄存器中作为输入参数，并且利用最后一条声明”cc”,”memory”来维护之前用的寄存器内存和状态寄存器。最后返回ax%的值。其实，用户调用的syscall接口（lib中)，也是通过执行int 0x30指令，然后转到内核态，调用内核的syscall，并在%eax中传递相应的中断号。

看明白了这些之后，只需要按照参数传递的约束，在trap\_dispatch中增加对应的T\_SYSCALL对应的中断项即可。同时在syscall函数中增加对中断号的判断调用相应的系统功能函数。

调用user/hello.c中的hello函数之后只会打印一条hello world，接着触发page fault，如图：



产生这个的原因在接下来的实验有讲解：

libmain() then calls umain, which, in the case of the hello program, is in user/hello.c. Note that after printing "hello, world", it tries to access thisenv->env\_id. This is why it faulted earlier.可以看的出是程序返回切换回用户环境的时候发生了权限错误。

###### 2.2.4 Exercise 8

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.

来分析一下执行的流程，按代码中注释的解释，lib/entry.S是用户环境切换之后(env\_run)，程序开始运行的地方(类似kernel从加载进的elf的entry开始运行)。在代码段开始，先有个判断：

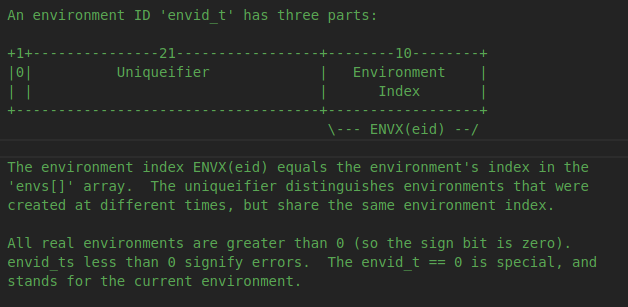
cmpl $USTACKTOP, %esp

jne args\_exist

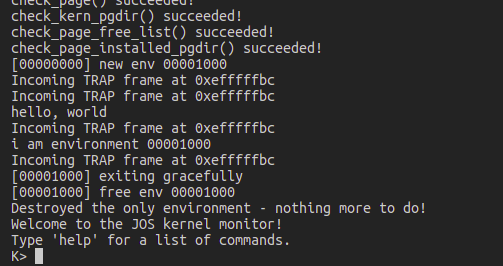
如果当前%esp在栈顶，代表才开始进入环境（从内核加载），则传入两个dummy arguments，进入libmain.c中。添加一行：

thisenv = envs + ENVX(sys\_getenvid());

这是由envid的layout决定的：



添加完毕后便可以继续运行了，从运行结果可以看出是先创立了环境0x00001000，然后打印了hello.c中的两句话，之后销毁用户环境，回到内核的monitor。



###### 2.2.5 Exercise 9

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.

这个练习讨论的是内存的保护问题，是操作系统中相当重要的问题。之前的执行中事实上我们向内核执行程序没有传递指针之类的参数，如果传递了得特别小心。或者说，要求我们解决两个问题：

1. 如果内核自身发生了page fault，这就是bug了，这个时候内核就得panic。但是如果只是在解引用用户指针的发生page fault，内核就需要知道这是用户指针造成的page fault，并且进行相应处理。
2. 另一方面，内核对用户传来的指针要严格检查，防止其泄漏内核信息或者毁坏内核组成。

内核不会受到用户指针导致的page fault的影响，如果内核自身发生了page fault，它应该终止执行。

检验是来自用户还是kernel的panic可以通过tss中存放的tf\_cs来判断，如果其cpl是0,则直接panic内核。

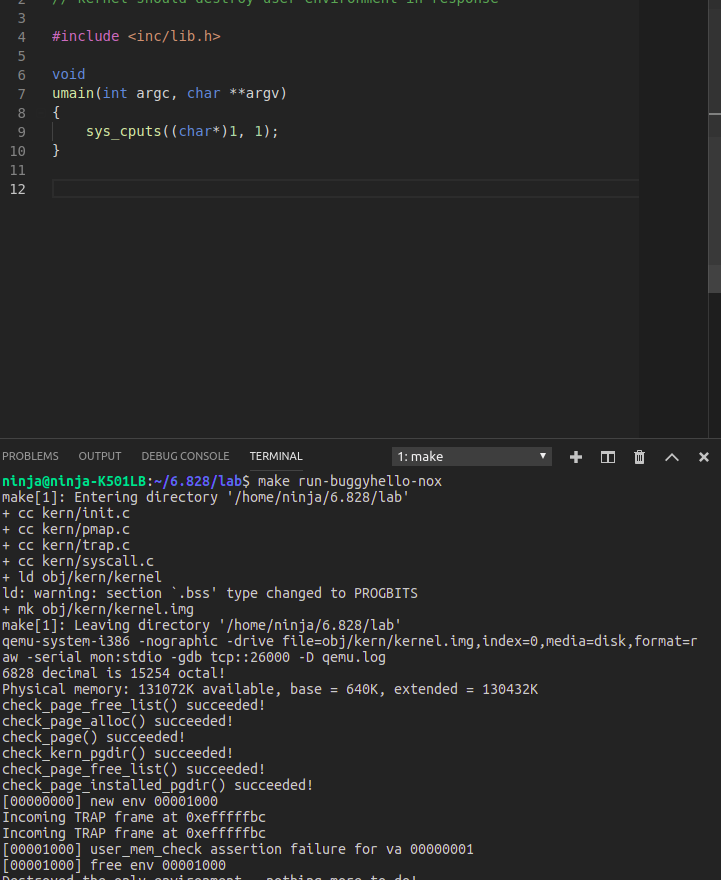
完成user\_mem\_check函数时判断是否超出ULIM界限：

if(start > ULIM)//check (1)

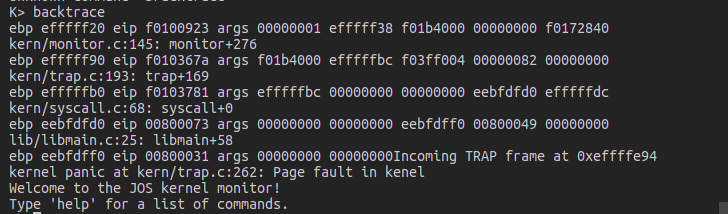
判断页面是否存在并有足够权限访问：

if((\*pte & (perm | PTE\_P)) != (perm | PTE\_P) || !pte)//check (2)

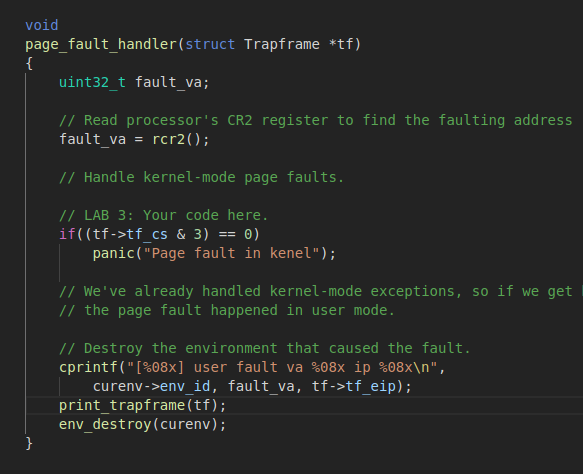
完成了检查指针函数之后，最后只需要将该函数用在传递系统调用的指针处即可。修改之后，run一下buggyhello，由于用户对0x1号内存没有读权限，触发page fault，如图



题目最后还有一个要求，为debuginfo\_eip函数添加内存访问权限检查，然后在执行int 3的breakpoint用户程序之后，backtrace查看栈，为什么内核会发生page fault错误？



可以发现其输出了page fault in kernel，这正是在函数中加入的检验内核是否发生page fault错误的代码片段。在此之前，还有用户的cprintf打印也出现了权限问题。cprintf的问题很好理解，无非是访问到了其不能访问的内存区：0xeffffe94，



###### 2.2.6 Exercise 10

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

这个结果和上述run buggyhello类似，都会触发内存检查错误。

### Lab4 **Preemptive** Multitasking

#### **1 实验目的**

1.在多个用户进程下实现可抢占式(preemptive)的多任务处理；

2.为JOS添加多处理器支持，实现轮转调度(round-robin)并添加一些基本的进程控制系统调用；

3.了解并实现一个和Unix类似的fork函数；

4.了解并实现进程间通信IPC，过程中需要添加对时钟中断的支持。

#### **2 实验内容**

多处理器结构的概念在之前没有遇见过，这里摘自百度百科：

（1） 包含两台或多台功能相近的处理器， 且彼此可交换数据；

（2） 所有处理器共享内存；

（3） 所有处理器都共享I/O通道、 控制器和外部设备；

（4） 整个系统由统一的操作系统控制， 在处理器和程序之间实现作业、 任务、 程序段、 数组及其元素各级的全面并行。

JOS中其实也给出了其要完成的多处理机的概念---SMP模型，该模型是CPU有同样的方式去获得系统资源（例如内存和IO总线），虽然所有的CPU最终地位是等价的，但是启动的时候会有不同，有这样两种类型的CPU——BSP和APs。BSP是用来引导并初始化操作系统的，APs由BSP引导并初始化操作系统之后激活。哪个处理机是BSP取决于硬件和BIOS。而且，还指出了目前为止的代码都是运行在BSP上的。

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

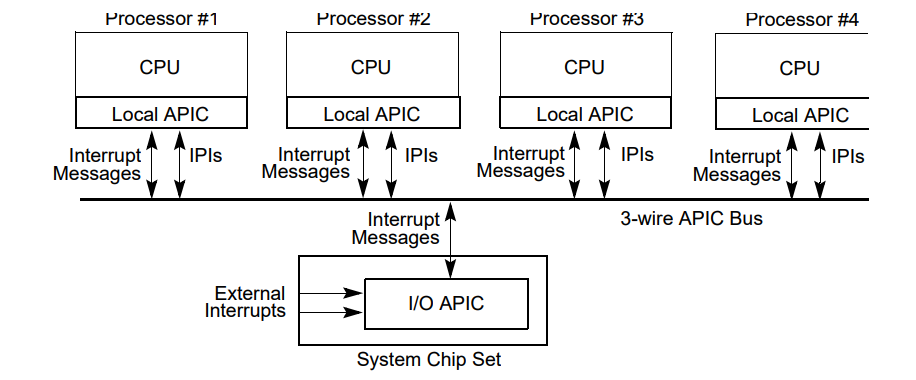
###### 2.1.1 Exercise 1

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.

这里涉及到CPU与LAPIC的通信，其实就是与外设的通信，每个LAPIC都有一个唯一的identifier。实验中用到的LAPIC功能(lapic.c)：

* 通过cpunum()函数读取LAPIC ID获取run当前代码的cpu。
* 通过lapic\_startap()函数从BSP发送STARTUP interprocessor interrupt(IPI)至APs启动相应的别的CPU。

Local APIC如图，总的来说，其可以接收来自其他LAPIC的中断和外部的I/O APIC中断并发送中断。



练习1简单的完成mmio\_map\_regoin函数，将从虚拟地址MMIOBASE开始的size大小个地址与物理地址pa建立映射，直接调用boot\_map\_region函数即可，其中注意对齐方法。

###### 2.1.2 Exercise 2

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).

文中介绍了几个给出的函数的作用：

mp\_init()函数用于BSP收集多处理机系统信息（cpu总数，它们的APIC IDs和LAPIC单元的MMIO地址。

boot\_aps()函数驱动AP的启动引导进程，APs也是从实模式开始工作和实验1中看过的boot.S过程很像，boot\_aps()在实模式下拷贝AP entry code(kern/mpentry.S)到一个内存可以寻址的地方，并且可以控制从某个地方开始执行。

boot\_aps的具体功能也可以从代码中看出来：

code = KADDR(MPENTRY\_PADDR);

memmove(code, mpentry\_start, mpentry\_end - mpentry\_start);

这是和前面启动类似，将对应的代码放置在MPENTRY\_PADDR处，MPENTRY\_PADDR宏的值为0x7000。

for (c = cpus; c < cpus + ncpu; c++)

for循环是一个个地激活AP

if (c == cpus + cpunum()) // We've started already.

continue;

跳过已经激活的cpu。

cpu c的栈地址为percpu\_kstacks[c - cpus]，其加上KSTKSIZE为栈顶地址。设立好栈之后，lapic\_startap(c->cpu\_id, PADDR(code));

总的来说control flow是boot\_aps()->lapic\_startup()，然后lapic\_startup(id, addr)然后发送IPI到APIC bus上，让相应AP的LAPIC收到这个中断，开始执行mpentry.S中的内容，开启保护模式，进行分页；加载数据段，代码段等。接着mpentry.S->mp\_main()，转到main的时候要间接跳转，如图，因为这样才能远跳。

接下来题目要我们修改page\_init()函数，因为注意到这时，要为MPENTRY\_PADDR处的其他cpu的内核启动程序分配空间，只需要将这个区间的内存简单的排除在free list外就好了。

###### 2.1.3 Question 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.

答：事实上MPBOOTPHYS的作用在代码段中有说：

# - it uses MPBOOTPHYS to calculate absolute addresses of its

# symbols, rather than relying on the linker to fill them

其定义：#define MPBOOTPHYS(s) ((s) - mpentry\_start + MPENTRY\_PADDR)

具体点说是在boot.S的LMA与VMA都是0x7c00，因此链接地址就是物理地址，这没有问题。但是mpentry.S中的代码片段是属于内核中的，也就是说VMA在高地址，所以需要将其转换为对应的物理地址。

###### 2.1.4 Exercise 3

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().

CPU的状态分为私有状态和共享状态。每个CPU的私有状态主要有：

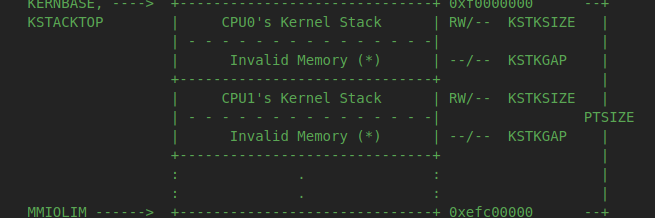
Kernel stack：存放在percpu\_kstacks[NCPU][KSTKSIZE]数组中

TSS和TSS descriptor：每个CPU的TSS存放在cpus[i].cpu\_ts中，并且对应的TSS descriptor在gdt[(GD\_TSS0 >> 3) + i]中定义。不再用之前实验的ts全局变量。

Current environment pointer：不再使用之前的全局变量curenv，而是保存在cpus[cpunum()].cpu\_env，即保存每个cpu执行的各自的当前环境。

System registers：**所有**的寄存器都是CPU私有的。

这个Exercise也比较简单，只需根据memlayout得出的虚拟地址



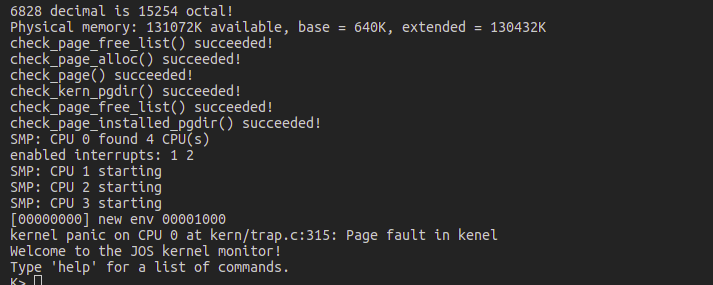
与物理地址栈percpu\_kstacks[i]做映射即可。

###### 2.1.5 Exercise 4

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.)

这个Exercise只需要对lab 3对应的代码进行改写就好了，主要是将之前的单ts改写成对应的thiscpu->cpu\_ts就好了。

完成之后，输入make qemu CPUS=4，发现CPU1,2,3都被成功启动：



###### 2.1.6 Exercise 5

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

这个Exercise主要是禁止CPU同时进入内核模式，当一个CPU进入内核模式便上锁，返回用户模式开锁。

上锁和开锁可以用kern/spinlock.h中提供的函数：lock\_kernel()和unlock\_kernel，在四个位置需要进行锁控制：

1. 在i386\_init()中，需要在BSP唤醒其他CPU(APs)的时候上锁，于是我们在boot\_aps()之前调用lock\_kernel()函数。
2. 在mp\_main()中，需要在AP初始化之后进行上锁。

AP分段、分页完成之后，一句话 xchg(&thiscpu->cpu\_status, CPU\_STARTED); // tell boot\_aps() we're up

这句话将thiscpu->cpu\_status设置为CPU\_STARTED，这是给BSP传递信息，告知已完成。同时，该AP也请求执行内核程序，同样上锁在sched\_yield()之前： lock\_kernel();

sched\_yield();

1. 在env\_run的时候，从核心态切换回用户态，这个时候可以解锁，释放核心态执行资源，供其他CPU使用。

###### 2.1.7 Question 2

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.

答：考虑这种情况，CPU0在执行用户态程序的时候发生了中断，然后转移到核心态的过程中进行压栈为Trapframe操作，然后才进入trap函数，而此时还没有加锁（进入trap之后才加的锁），如果这时候别的CPU也同样有个用户到内核的中断，栈结构就会被破坏，造成CPU返回错误。

###### 2.1.8 Exercise 6

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.

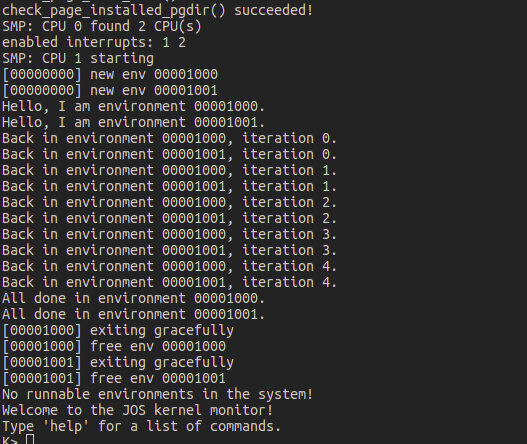
Modify kern/init.c to create three (or more!) environments that all run the program user/yield.c.

我们之前所完成的内核都只是简单的运行完了用户程序便退出，接下来要实现一个轮转调度策略——Round-Robin Scheduling。主要完成的函数是sched\_yield()，分为以下几步：

1.先按照循环的方式搜索envs[]数组，从上一次运行完的程序开始搜索，搜索到第一个ENV\_RUNNABLE的状态的env，可以看出这就是一个简单的先进先出轮转策略，ENV\_RUNNABLE可以作为unix中的就绪状态理解。

2.如果没有找到ENV\_RUNNABLE的程序，接着执行原来的程序。

完成之后，打印CPU运行结果图：



###### 2.1.9 Question 3

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?

答：因为e在创建的时候，有memcpy(e->env\_pgdir, kern\_pgdir, PGSIZE);

###### 2.1.10 Question 4

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?

答：在env\_run()函数中 env\_pop\_tf(&(e->env\_tf));//switch register

就是从trapframe中将该环境中的寄存器恢复出来。其作用很明显，就是保留现场。

###### 2.1.11 Exercise 7

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.

这一问要完成的内容比较多，是要我们添加多个系统调用，总的来说，是为了实现类似UNIX fork调用的功能，分为几个部分分步完成：

2.1.11.1 sys\_exofork()

该系统调用分配了一个ENV\_NOT\_RUNNABLE的环境，将相关寄存器一律设定为与父寄存器一致：

new\_env->env\_tf = curenv->env\_tf;

其中得注意的一点是，类似于fork，其对父函数返回的值就是新分配的这个环境，但是由于这时候子环境还没有开始run，将其相对于子环境的返回值存放在trapframe的eax寄存器中，供子环境运行的时候使用。

2.1.11.2 sys\_env\_set\_status

其作用是将环境的状态设置为ENV\_RUNNABLE或ENV\_NOT\_RUNNABLE。函数部分主要是做两个检查：!(status == ENV\_RUNNABLE || status == ENV\_NOT\_RUNNABLE)与envid2env(envid, &e, 1) < 0都是传入的错误的不满足条件的参数，将检验完e->env\_status设置为status即可。事实上ENV\_RUNNABLE对应的是就是UNIX中的就绪状态，而ENV\_NOT\_RUNNABLE可以理解为等待状态。

2.1.11.3 sys\_page\_alloc

其作用是为之前分配的还没有run的环境设置虚拟地址空间，先对传入的参数进行异常检验，然后在虚拟地址va处分配一个页面。

2.1.11.4 sys\_page\_map

sys\_page\_map是将srcenvid环境的虚拟地址srcva与对应的物理地址的映射关系，拷贝到distenvid环境的虚拟地址dstva与对应的物理地址的映射关系中。

2.1.11.5 sys\_page\_unmap

sys\_page\_unmap即取消envid环境中的虚拟地址va与其对应的物理地址之间的映射。

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

###### 2.2.1 Exercise 8

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.

这个exercise没有什么内容，主要是为后面做铺垫的。

###### 2.2.2 Exercise 9

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?)

如果内核发生page fault将会直接panic，而如果是用户程序发生中断就需要我们进行处理了，基本过程如下：

page\_fault\_handler()函数处于内核态，先通过

fault\_va = rcr2();

这条语句得到发生错误的地址，通过传入page\_fault\_handler函数的trapframe可以判断是从内核态还是从用户态陷入进来的，page\_fault是个较特别的中断，因为如果是内核发生缺页中断，那么就要panic，如果是发生在用户，就需要我们对后续过程进行处理了。

如果是从异常栈转过来，就空4个字节再将UTrapframe结构（设置）压栈；如果是从用户栈过来，就在异常栈栈顶进行（设置）压栈。结构体包含如下内容：

struct UTrapframe {

/\* information about the fault \*/

uint32\_t utf\_fault\_va; /\* va for T\_PGFLT, 0 otherwise \*/

uint32\_t utf\_err;

/\* trap-time return state \*/

struct PushRegs utf\_regs;

uintptr\_t utf\_eip;

uint32\_t utf\_eflags;

/\* the trap-time stack to return to \*/

uintptr\_t utf\_esp;

} \_\_attribute\_\_((packed));

栈的初始化步骤：

if(curenv->env\_pgfault\_upcall)//page fault upcall exists

{

// check permision

user\_mem\_assert(curenv, (void \*)uxesp, sizeof(struct UTrapframe), PTE\_U | PTE\_P | PTE\_W);

utf = (struct UTrapframe \*)uxesp;

utf->utf\_fault\_va = fault\_va;

utf->utf\_err = tf->tf\_err;

utf->utf\_regs = tf->tf\_regs;

utf->utf\_eflags = tf->tf\_eflags;

utf->utf\_eip = tf->tf\_eip;

utf->utf\_esp = tf->tf\_esp;

tf->tf\_esp = uxesp;

tf->tf\_eip = (uint32\_t)curenv->env\_pgfault\_upcall;

env\_run(curenv);

}

根据我们对tf的esp和eip的修改，env\_run接下来执行的程序事实上就是curenv->env\_pgfault\_upcall，执行程序时候需要的堆栈也切换到用户异常栈。env\_run中关键的调用env\_pop\_tf之前已经分析过了，其转移到对应的由trapframe结构体所代表的环境继续执行，也就是之前说的env\_fault\_upcall。

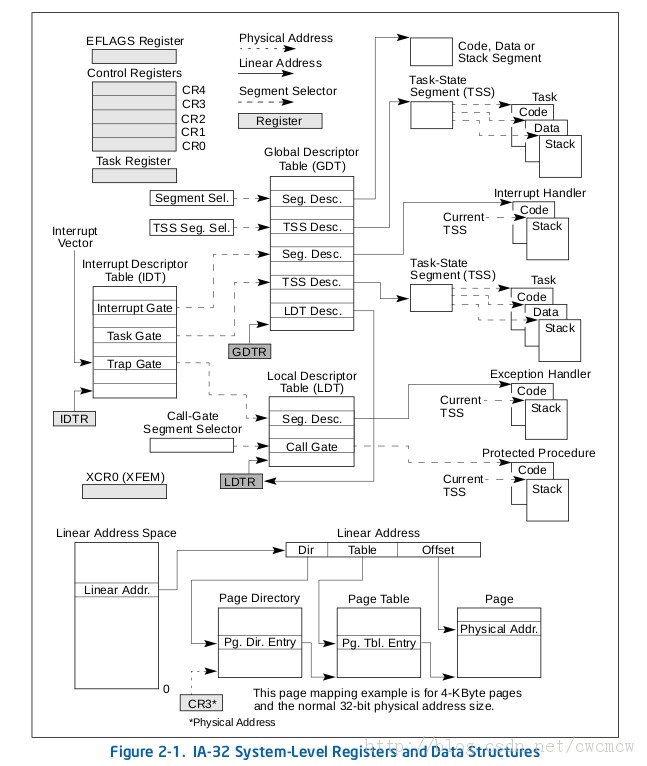
###### 2.2.3 Exercise 10 & 11

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.

注意到\_pgfault\_upcall在lib目录下，所以可以判定这是用户态执行的程序，他是如何装载以及执行的，应该结合Exercise 11一起看

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

从这个练习，就可以看到， 该函数将



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

###### 2.3.1 Exercise 13

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, or section 5.8 of the IA-32 Intel Architecture Software Developer's Manual, Volume 3, at this time.

这里要实现IPC，实际上是要JOS先支持外部中断，在JOS系统中IRQ 0-15对应的中断号32-47。而对应的函数handler已经编制完成，不需要我们去编写，这个任务就是将该函数handler放置到对应的位置。

所以这里只要重复之前的工作即可。

这里还要做额外做的一个事情是增加一个sti。可以看一下sched\_halt的源码：

for (i = 0; i < NENV; i++) {

if ((envs[i].env\_status == ENV\_RUNNABLE ||

envs[i].env\_status == ENV\_RUNNING ||

envs[i].env\_status == ENV\_DYING))

break;

}

if (i == NENV) {

cprintf("No runnable environments in the system!\n");

while (1)

monitor(NULL);

}

这里如果没有RUNNABLE,RUNNING,DYING的进程，就直接进入monitor中。如果有，主体部分呢就是如下内联汇编代码

asm volatile (

"movl $0, %%ebp\n"

"movl %0, %%esp\n"

"pushl $0\n"

"pushl $0\n"

"sti\n"

"1:\n"

"hlt\n"

"jmp 1b\n"

: : "a" (thiscpu->cpu\_ts.ts\_esp0));

sti开启了中断，使得CPU可以接收外部中断，再来看看HLT指令的作用：

HLT Description

Stops instruction execution and places the processor in a HALT state. An enabled interrupt (including NMI and SMI), a debug exception, the BINIT# signal, the INIT# signal, or the RESET# signal will resume execution. If an interrupt (including NMI) is used to resume execution after a HLT instruction, the saved instruction pointer (CS:EIP) points to the instruction following the HLT instruction.

When a HLT instruction is executed on an Intel 64 or IA-32 processor supporting Intel Hyper-Threading Technology, only the logical processor that executes the instruction is halted. The other logical processors in the physical processor remain active, unless they are each individually halted by executing a HLT instruction.

The HLT instruction is a privileged instruction. When the processor is running in protected or virtual-8086 mode, the privilege level of a program or procedure must be 0 to execute the HLT instruction.

在这个实验里要明白的作用大概就是第一段，挂起当前CPU，停止当前的指令执行，遇到（时钟）中断之后才继续执行。

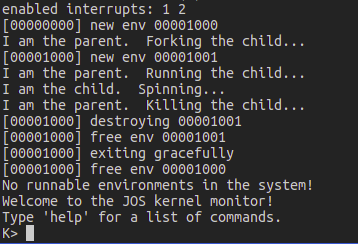
###### 2.3.2 Exercise 14

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.

这一练习就是简单的往trap\_dispatch中增加一个中断处理，时钟中断对应的中断号是IRQ\_OFFSET + 0。

运行run-spin-nox结果如下



###### 2.3.3 Exercise 15

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.

这一练习的要求很明确，就是在kern中添加系统调用处理函数，然后在lib中添加给用户的系统调用接口。

在kern中要完成的部分是sys\_ipc\_recv和sys\_ipc\_try\_send函数。

主要就是从src往dst进行通信，传输一个页的信息。

2.3.3.1 sys\_ipc\_recv()

该函数接收四个参数，envid作为发送目的环境，value作为发送给目的地址的32位的值，srcva是发送源虚拟地址内存中的某一页地址，perm为对应的虚拟页权限参数。大概分为以下几步：

1. 首先判断目的环境是否处于接收状态，如果不处于，则sys\_ipc\_try\_send失败返回，事实上之后可以看到，sys\_ipc\_try\_send位于一个循环中，所以其尝试失败会接着循环尝试。
2. 如果srcva < UTOP，则发送一个起始地址为srcva的页面。

这里的发送步骤事实上就是在目的环境创建一个虚拟内存，然后源环境与目的环境共享一个物理块来通信。

源环境通过page\_lookup找到对应的物理块

if((pp = page\_lookup(curenv->env\_pgdir, srcva, &pte)) == NULL)

return -E\_INVAL;

然后page\_insert插入目的虚拟地址空间

if((uintptr\_t)(dstenv->env\_ipc\_dstva) < UTOP)

{

if(page\_insert(dstenv->env\_pgdir, pp, dstenv->env\_ipc\_dstva, perm) < 0)

return -E\_NO\_MEM;

dstenv->env\_ipc\_perm = perm;

}

1. 最后无论srcva是否在UTOP之下，都传递一个32位的值value给目的环境，并且目的环境接收标记dstenv->env\_ipc\_recving设置为false。
2. 唤醒受阻塞的进程

2.3.3.2 sys\_ipc\_recv()

这个函数实现接收其他进程信号的功能，如果没有收到信号，进程（环境）表现为阻塞，即该环境状态设置为ENV\_NOT\_RUNNABLE，并且标记为请求接收，即curenv->env\_ipc\_recving = true;，然后接着调度其他进程。

在lib中要完成的两个函数是ipc\_send和ipc\_find\_env。这两个函数完成的主要工作就是调用kern中的通信函数，在这里不再赘述。

### Reporting experience

不愧是MIT有名的操作系统实验，做起来感觉很吃力，但是收获也是很大的。做的时候也遇到了不少bug，在网上参考了一些博客，也查看了些资料。做的过程中感觉比较困难的地方其实是函数之间的互相调用关系，以及很多共享的变量，模块与模块之间的界限不是太清楚，而且其布置任务的时候会要求我们先完成被调函数，再完成调用函数，这样子经常会在写被调函数的时候还不清楚其使用的场合、对作用的理解也不大明确。事实上，完成了之后，再把函数关系梳理一下、把基本流程走一遍还是很有必要的。