MIT6.828 2011fall JOS LAB

刘道琛

ldaochen@gmail.com

June 11, 2015

目录

1 LAB

1.1 LAB1: Booting a PC

1.1.1 Part 1: PC Bootstrap

1.1.2 Part 2: The Boot Loader

Exercise 3 Q1: At what point does the processor start executing 32-bit code? What exactly causes the switch from 16- to 32-bit mode?

A: boot/boot.S 中如下指令

```
movl %eax %cr0
```

开始使得处理器开始执行 32-bit 代码.

```
ljmp    $PROT_MODE_CSEG, $protcseg
```

引起从 16 位模式切换到 32 位模式.ljmp 将 cs 寄存器的值指向一个 32bit 代码段, 所以处理器开始执行 32bit 代码这条命令本身就是执行 32 位代码, 因为此时保护模式已经开启 (是执行完这条语句才开始执行 32 位代码, 还是这条语句本身就是执行 32 位代码?). stackoverflow 上的一段回答http://stackoverflow.com/questions/5211541/bootloader-switching-processor-to-protected-mode. 节选一段:When you looked at the .asm file, the instruction was interpretted as if the address size was 32 bits, but GDB interpretted it as if the address size was 16 bits. The data at the address of the instruction would be 0xEA 32 7C 08 00 66 B8. EA is the long jump opcode. In a 32 bit address space, the address would be specified using the next four bytes, for an address of 0x87C32, but in a 16 bit address space, only 2 bytes are used, for an address of 0x7C32. The 2 bytes after the address specify the requested code segment, which would be 0xB866 in 32 bit mode and 0x0008 in 16 bit mode. The 0x66 B8 is the start of the next instruction, which is moving a 16 bit immediate value into the ax register, probably to set up the data segments for protected mode.(为什么 ljmp 在.asm 文件中地址和 gdb 调试时候显示的地址不同)

Q2 What is the last instruction of the boot loader executed, and what is the first instruction of the kernel it just loaded?

A: Bootloader 最后的一条指令

```
((void (*)(void)) (ELFHDR->e_entry))()
```

对应的汇编指令是

```
7d61:call *0x10018
```

使用 p/x *0x10018 命令查看 0x10018 处存放的内容, 知地址 0x10018 内存放的值是 0x10000c. 使用 objdump -f obj/kern/kernel 查询知道 jos 的入口地址是 0x10000c. 在该地址处设置断点:b *0x10000c, 执行到下面的指令处

```
0x10000c:movw $0x1234,0x472,
```

内核的第一条指令是 movw $0x1234,0x472, 对应 kern/entry.S 中的 entry 符号处 .

Q3: Where is the first instruction of the kernel?

A:0x10000c 处, entry.S 中 entry 符号处 movw $0x1234,0x472

Q4: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?

A: elf 头中包含该信息,ELFHDR->e_phnum 表示 section 的个数,ph->p_memsz 表示 section 的大小.

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!

A: 一些命令和说明:

```
objdump -h:Display the contents of the section headers -See VMA/LMA.
objdump -f: Display the contents of the section headers -See entry\_point
       for execution start address
Change boo/Makefrag的-Ttext : 0x7c000 for link address
VMA: Link address. Expected address for it to rum.
LMA: Load address. Expected address for it to load into memory.
```

将 boot/Makefrag -Ttext 从 0x7c00 修改为 0x7c04,bios 仍然将 bootloader 加载到 0x7c00 处.gdtdesc 在内存 0x7c64 处, 当执行 lgdt gdtdesc 使用却是 lgdt 0x7c68, 因此加载了错误的 gdt. 接下来执行到 ljmp 会出现错误. 因为 ljmp 会使得 processor 跳转到 32-bit protected mode, 但是此时使用的 gdt 却是错误的. 所以会出错.the first instruction that would "break":

```
ljmp    $PROT_MODE_CSEG, $protcseg   @boot/boot.S
```

问题：当修改 -Ttext 为 0x7c04 后, 使用 objdump -h boot.out 。输出如下

```
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         0000017e  00007c04  00007c04  00000074  2**2
```

vma =lma =0x7c04. 为什么会修改两个. 不是只修改 vma 吗?

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**

A:

```
set breakpoint at 0x7c00. Continue. x/8x 0x00100000
0x100000: 0x00000000  0x00000000  0x00000000  0x00000000
0x100010: 0x00000000  0x00000000  0x00000000  0x00000000

set breakpoint at kernel entry(0x10000c). Continue x/8x 0x00100000
0x100000: 0x1badb002  0x00000000  0xe4524ffe  0x7205c766
0x100010: 0x34000004  0x0000b812  0x220f0011  0xc0200fd8
```

当 bios 进入 bootloader 时, 内核并没有加载进入内存 0x100000, 所以从 0x100000 处开始的 8 个字节是 0. 但是当 bootloader 进入内核, 内核已经载入到内存 0x100000, 所在在那里有数据.

1.1.3 Part 3: The Kernel

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.**

A:

```
before  movl %eax, %cr0
x/8x 0x100000
0x100000: 0x1badb002  0x00000000  0xe4524ffe  0x7205c766
0x100010: 0x34000004  0x0000b812  0x220f0011  0xc0200fd8

x/8x 0xf0100000
```

```
0xf0100000: 0x00000000  0x00000000  0x00000000  0x00000000
0xf0100010: 0x00000000  0x00000000  0x00000000  0x00000000


stepi  movl %eax, %cr0

x/8x 0x100000
0x100000: 0x1badb002  0x00000000  0xe4524ffe  0x7205c766
0x100010: 0x34000004  0x0000b812  0x220f0011  0xc0200fd8

x/8x 0xf0100000
0xf0100000: 0x1badb002  0x00000000  0xe4524ffe  0x7205c766
0xf0100010: 0x34000004  0x0000b812  0x220f0011  0xc0200fd8
```

因为 movl %eax %cr0 之后开启了分页. 映射虚地址 [KERNBASE, KERNBASE+4MB) 到物理地址 [0,4MB).

注释掉 movl %eax, %cr0, 重新启动系统执行出现下面的错误. fatal: Trying to execute code outside RAM or ROM at 0xf010002c, 第一个 failed 的指令是 movl $0x0 %ebp. 如图.

```
=> 0x10001a:    mov    %eax,%cr3
0x0010001a in ?? ()
(gdb) si
=> 0x10001d:    mov    %cr0,%eax
0x0010001d in ?? ()
(gdb) si
=> 0x100020:    or     $0x80010001,%eax
0x00100020 in ?? ()
(gdb) si
=> 0x100025:    mov    $0xf010002c,%eax
0x00100025 in ?? ()
(gdb) si
=> 0x10002a:    jmp    *%eax
0x0010002a in ?? ()
(gdb) si
=> 0xf010002c <relocated>:      add    %al,(%eax)
relocated () at kern/entry.S:73
73              movl    $0x0,%ebp                       # nuke frame pointer
(gdb)
```

图 1: ljmp *%eax 前

```
=> 0x10001a:    mov    %eax,%cr3
0x0010001a in ?? ()
(gdb) si
=> 0x10001d:    mov    %cr0,%eax
0x0010001d in ?? ()
(gdb) si
=> 0x100020:    or     $0x80010001,%eax
0x00100020 in ?? ()
(gdb) si
=> 0x100025:    mov    $0xf010002c,%eax
0x00100025 in ?? ()
(gdb) si
=> 0x10002a:    jmp    *%eax
0x0010002a in ?? ()
(gdb) si
=> 0xf010002c <relocated>:      add    %al,(%eax)
relocated () at kern/entry.S:73
73              movl    $0x0,%ebp                       # nuke frame pointer
(gdb) si
Remote connection closed
(gdb) si
```

图 2: 执行 ljimp *%eax 后

```
=> 0x10001d:    mov    %cr0,%eax
0x0010001d in ?? ()
(gdb) si
=> 0x100020:    or     $0x80010001,%eax
0x00100020 in ?? ()
(gdb) si
=> 0x100025:    mov    $0xf010002c,%eax
0x00100025 in ?? ()
(gdb) si
=> 0x10002a:    jmp    *%eax
0x0010002a in ?? ()
(gdb) si
=> 0xf010002c <relocated>:      add    %al,(%eax)
relocated () at kern/entry.S:73
73              movl    $0x0,%ebp                       # nuke frame pointer
(gdb) p/x $eip
$1 = 0xf010002c
(gdb) p/x $cs
$2 = 0x8
(gdb)
```

图 3: 打印%cs,%eip 的值

这条指令执行在高地址执行, 但是分页机制没有开启, 所以不能够映射高地址到低地址, 在执行 ljmp *%eax 指令后%cs 寄存器的值 0x8, %eip 的的值为 0xf010002c. 该地址经过分页应该对应低地址处的 movl $0x0 %ebp, 但是由于分页没有开启, 所以%eip(0xf010002c) 指向高地址的指令 add %al,(%eax), 使用 x/Ni 命令查看该处的指令. 如图:

```
=> 0x100020:    or     $0x80010001,%eax
0x00100020 in ?? ()
(gdb) si
=> 0x100025:    mov    $0xf010002c,%eax
0x00100025 in ?? ()
(gdb) si
=> 0x10002a:    jmp    *%eax
0x0010002a in ?? ()
(gdb) si
=> 0xf010002c <relocated>:      add    %al,(%eax)
relocated () at kern/entry.S:73
73              movl    $0x0,%ebp                       # nuke frame pointer
(gdb) p/x $eip
$1 = 0xf010002c
(gdb) p/x $cs
$2 = 0x8
(gdb) x/i 0xf010002c
=> 0xf010002c <relocated>:      add    %al,(%eax)
(gdb)
```

图 4: 查看 0xf010002c

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.

A: 在 lib/printfmt.c 中 vprintfmt 函数添加如下代码

```
case 'o':
    // Replace this with your code.
    num = getuint(&ap, lflag);
    base = 8;
    goto number;
```

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?

A: 接口是 cputchar()

```
High-level console I/O.  Used by readline and cprintf.
void cputchar(int c)
{
        cons_putc(c);
}
```

cputchar 可以被 readline 等 lib 目录下的函数使用, console.c 导出 cputchar,getchar 函数, printf.c 中 cprintf 函数最终会使用 cputchar 进行输出

2 Explain the following from console.c:

```
1       if (crt_pos >= CRT_SIZE) {
2               int i;
3               memcpy(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       }
```

A: crt_pos 是当前的输出位置.CRT_SIZE = CRT_COLS * CRT_ROWS, 当输出屏幕满时. 向上滚动一行. 所以冲掉第一行的数据. 之后的所有数据向前移动一行. 每行的字节数为 CRT_COLS 个.crt_pos 也对应的减小 CRT_COLS 个. 因为向上滚动了一行表示多了一行的空间.

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.

A: 添加如下代码到/kern/init.c

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

重新编译 JOS, 在 init.c 的 42 行处设置断点 (就是我添加上面的代码的地方)
b init.c:42 单步执行, 查看传递给 cprintf 函数的参数.

```
Mov 4  0xc(%esp)
Mov 3  0x8(%esp)
Mov 1  0x4(%esp)
```

然后将字符串"x %d, y %x, z %d\n" 的地址移动到 (%esp), 即使放到栈上, 之后调用 cprintf 函数, 在/kern/printf.c:32 处设置断点. 查看 ap 的值

```
lea    0xc(%ebp),%eax
mov    %eax,0x4(%esp)
```

ap 存放在 0x4(%esp) 处, 即 ap 是一个指针, 指向地址 0xc(%ebp), 该地址是 x 在栈中的位置, 查看其中的内容 ap 指向的地址中存放的值为 1, 即 x 的值

4 Run the following code.

```
   unsigned int i = 0x00646c72;
   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. Here's an ASCII table that

maps bytes to characters. 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?

A: He110 World 57616 的十六进制表示是 0xell0, i 为 0x00646c72, 在 ASCII 中 0x64='d', 0x6c='l', 0x72='r' 而且 x86 是小端存储,r 存放在低地址处. 使用 GDB 可以查看 rld'\0' 在内存中的是如何存放的.

```
(gdb) p/c *0xf010ffec
$9 = 114 'r'
(gdb) p/c *0xf010ffed
$10 = 108 'l'
(gdb) p/c *0xf010ffee
$11 = 100 'd'
(gdb) p/c *0xf010ffef
$12 = 0 '\000'
```

如果 x86 采用大端 i 应该改为 0x726c6400, e110 的打印和 57616 的存储方式没有关系, 没有必要修改 57616 的值.

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);
```

A:x=3 y=-267321364 参数 x 的值存放在栈中 0x4(%esp) 即 0xf010ffd4 处, 使用 GDB 输出该值:

(gdb) p/x *0xf010ffd4

$7 = 0x3

参数 y 的值应该存放在栈中 0x8(%esp), 即 0Xf010ffd8 处:

用 GDB 输出该值

(gdb) p *0xf010ffd8

$10 = -267321364

所以输出时 y=-267321364, 虽然 cprintf 没有传入 y 的值, 但是当 cprintf 执行时会将栈中 x 的下一个位置当做参数 y 的位置.

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?

A: 从上面可以看到,va_arg 每次是以地址往后增长取出下一参数变量的地址的. 而这个实现方式就默认假设了编译器是以从右往左的顺序将参数入栈的. 因为栈是以从高往低的方向增长的. 后压栈的参数放在了内存地址的低位置. 所以如果要以从左到右的顺序依次取出每个变量. 那么编译器必须以相反的顺序即从右往左将参数压栈。如果编译器更改了压栈的顺序. 那么为了仍然能正确

取出所有的参数, 那么需要修改上面代码中的 va_start 和 va_arg 两个宏, 将其改成用减法得到新地址即可. 下面给出一个 va_arg 的实现

```
/*  $NetBSD: stdarg.h,v 1.12 1995/12/25 23:15:31 mycroft Exp $  */

#ifndef JOS_INC_STDARG_H
#define JOS_INC_STDARG_H

typedef char *va_list;

#define __va_size(type) \
  (((sizeof(type) + sizeof(long) - 1) / sizeof(long)) * sizeof(long))

#define va_start(ap, last) \
  ((ap) = (va_list)&(last) + __va_size(last))

#define va_arg(ap, type) \
  (*(type *)((ap) += __va_size(type), (ap) - __va_size(type)))

#define va_end(ap) ((void)0)

#endif /* !JOS_INC_STDARG_H */
```

现在 xv6 的 va_arg 等宏采用了 gcc 内置的实现.

7 Challenge Enhance the console to allow text to be printed in different colors. The traditional way to do this is to make it interpret ANSI escape sequences embedded in the text strings printed to the console, but you may use any mechanism you like. There is plenty of information on the 6.828 reference page and elsewhere on the web on programming the VGA display hardware. If you're feeling really adventurous, you could try switching the VGA hardware into a graphics mode and making the console draw text onto the graphical frame buffer.

可以先看下颜色的控制机制, 在 kern/console.c 文件中函数 cga_putc, 我们可以看到关于颜色控制的相关机制:

```
if (!(c & ~0xFF))
      c |= 0x0700;
```

在 c 中, 低 8 位是为了指定需要打印的字符的 ASCII 码, 而高八位则是指定打印的颜色, 可以看到代码中如果不设置颜色则默认是黑底白字, 高八位为 07, 其高 8 位具体含义如下:

表 1: Properties of Different Paging Modes.

15	14	13	12	11	10	9	8
B_I	B_R	B_G	B_B	F_I	F_R	F_G	F_B

C 的 15 到 12 位用于指定字符的背景颜色, 颜色由 3 位 RGB 颜色代码 +1 位是否高亮显示.11 到 8 位用于指定字符的前景颜色.

可以将输出字符修改为红色, 对应的代码修改为:

```
if (!(c & ~0xFF))
        c |= 0x0400; //红色
```

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?

A: 在/kern/entry.S 中内核初始化它的的栈, 对应的代码如下

```
  movl    $0x0,%ebp           # nuke frame pointer
  # Set the stack pointer
  movl    $(bootstacktop),%esp
```

内核初始作的工作主要是将寄存器%ebp 初始为 0,%esp 初始化为 bootstacktop，我们进入 bootstacktop 的定义位置进行查看:

```
 .p2align    PGSHIFT     # force page alignment
    .globl      bootstack
bootstack:
     .space      KSTKSIZE
     .globl      bootstacktop
bootstacktop:
```

其中 KSTKSIZE 的大小是 8 页大小, 即 32K. 使用 GDB 查看栈的底和顶地址:

```
p  &bootstack
$1 = (<data variable, no debug info> *) 0xf0108000

p &bootstacktop
$2 = (<data variable, no debug info> *) 0xf0110000
```

即 bootstack 地址 0xf0108000,bootstacktop 地址 0xf0110000. 两地址差恰好是 stack 的大小:8 页. 即 esp 值为 0xf0110000. 朝低地址方向增长. 栈底为 bootstack 即 0xf0108000. 栈的空间定义在 ELF 文件中的.data 段, 载入内核时根据 data 段在 ELF 文件中的相对位置被载入内存.

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.

A:push ebp,push ebx 归调用时的 call 会自动 push eip 总个 3 个 32bit 的 word 压栈

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.

A:x86 调用栈如下:

```

                       +------------+   |
                       | arg 2      |   \
                       +------------+    >- previous function's stack frame
                       | arg 1      |   /
                       +------------+   |
                       | ret\%eip   |   /
                       +============+
                       | saved\%ebp |   \
               \%ebp-> +------------+   |
                       |            |   |
                       |   local    |   \
                       | variables, |    >- current function's stack frame
                       |    etc.    |   /
                       |            |   |
                       |            |   |
               \%esp-> +------------+   /
```

熟悉理解调用栈后, 对 kern/monitor.c 中的 mon_backtrace 函数进行修改, 最终代码如下:

```
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
    // Your code here.
    uintptr_t *ebp, *eip;
    int i;
    struct Eipdebuginfo info;
    uint32_t args[5];
    cprintf("Stack backtrace:\n");
    ebp =(uintptr_t*) read_ebp();
    while(ebp != 0)
    {
        eip = ebp+1;
```

```
14          for(i = 0; i < 5; i++)
15              args[i] = *(ebp+2+i);
16
17          cprintf("␣ebp␣%08x␣eip␣%08x␣args␣%08x␣%08x␣%08x␣%08x␣%08x\n", ebp,
            *eip,
18              args[0], args[1], args[2], args[3], args[4]);
19          debuginfo_eip(*eip, &info);
20          cprintf("␣␣␣␣␣%s:%d:␣%.*s+%d\n",info.eip_file, info.eip_line,
21              info.eip_fn_namelen, info.eip_fn_name, (*eip)-info.eip_fn_addr)
            ;
22          ebp = (uintptr_t*)(*ebp);
23      }
24      return 0;
25  }
```

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 i386-jos-elf-objdump -h obj/kern/kernel
run i386-jos-elf-objdump -G obj/kern/kernel
run i386-jos-elf-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:

```
1  K> backtrace
2  Stack backtrace:
3    ebp f010ff78  eip f01008ae  args 00000001 f010ff8c 00000000 f0110580
         00000000
4           kern/monitor.c:143: monitor+106
5    ebp f010ffd8  eip f0100193  args 00000000 00001aac 00000660 00000000
         00000000
6           kern/init.c:49: i386\_init+59
7    ebp f010fff8  eip f010003d  args 00000000 00000000 0000ffff 10cf9a00 0000
         ffff
8           kern/entry.S:70: <unknown>+0
9  K>
```

Each line gives the file name and line within that file of the stack frame's eip, followed by the name of the function and the offset of the eip from the first

instruction of the function (e.g., monitor+106 means the return eip is 106 bytes past the beginning of monitor).

Be sure to print the file and function names on a separate line, to avoid confusing the grading script.

Tip: printf format strings provide an easy, albeit obscure, way to print non-null-terminated strings like those in STABS tables. printf("%.*s", length, string) prints at most length characters of string. Take a look at the printf man page to find out why this works.

You may find that the some functions are missing from the backtrace. For example, you will probably see a call to monitor() but not to runcmd(). This is because the compiler in-lines some function calls. Other optimizations may cause you to see unexpected line numbers. If you get rid of the -O2 from GNUMakefile, the backtraces may make more sense (but your kernel will run more slowly).

A: 首先在 https://sourceware.org/gdb/onlinedocs/stabs.html 查阅 Line Numbers 信息, An N_SLINE symbol represents the start of a source line.The desc field contains the line number. 在 inc/stab.h 中 struct Stab 定义如下:

```
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
 };
```

1. n_strx: 该项在 stabstr section 中的索引
2. n_type: 该项类型
3. n_other: 一般为空
4. n_desc: 表示源文件中的行号
5. n_value: 该项对应的符号值

我们使用 objdump -G 命令查看一个 ELF 文件的 stab section 信息, 即符号表信息.

使用 objdump -G obj/kern/kernel 命令输出如下

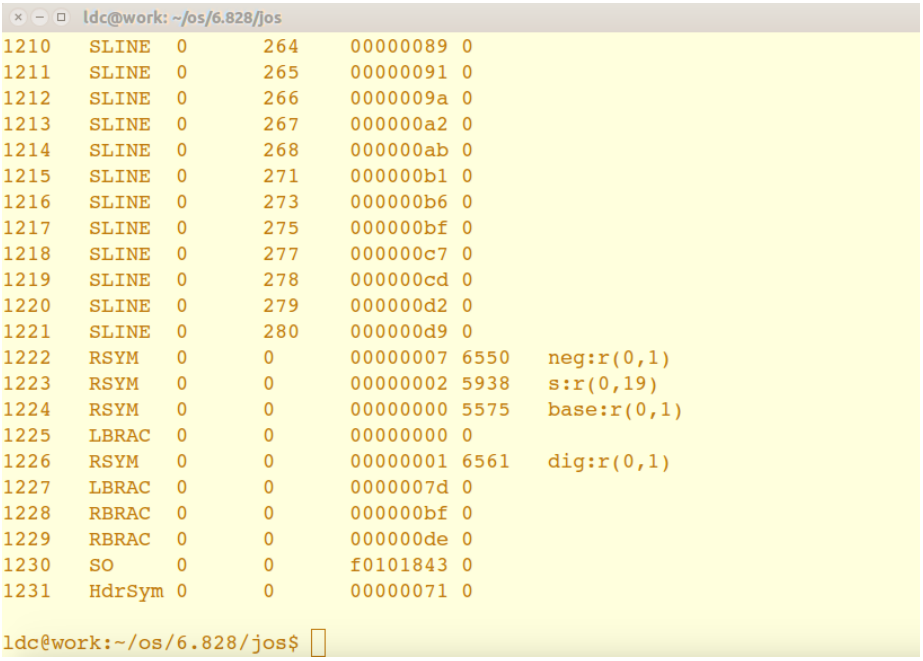


图 5: 输出内核符号表信息

在这里可以看到一些类型名字,SO, SLINE, FUN 等等, 上面提到的 n_type 就是这些值之一.SO 表示源文件, FUN 表示函数, SLINE 表示行号.

运行 objdump -G obj/kern/kernel | grep 'SO' 查看所有的文件. 输出如下, 输出很长只截取了最后一部分.

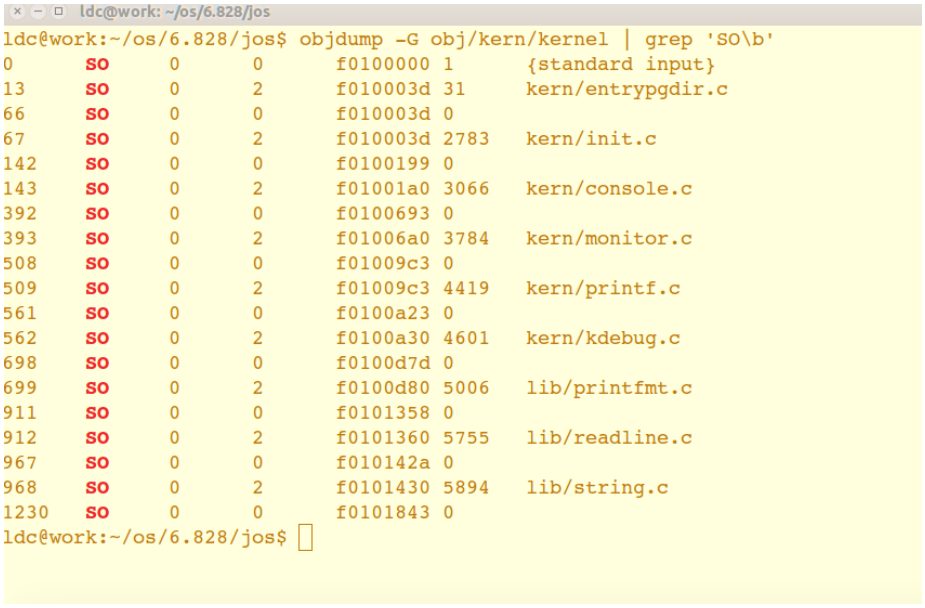


图 6: 内核中所有的文件

可以看出每个文件在编译链接后在 ELF 文件中的链接地址递增排列. 在观察所有的函数, 运行 objdump -G obj/kern/kernel | grep FUN, 输出如下:

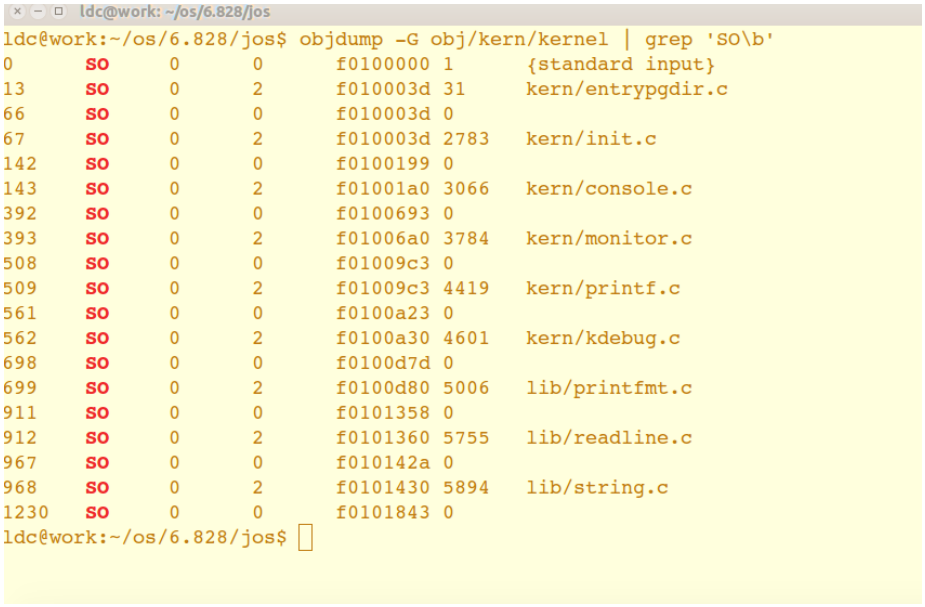


图 7: 内核中的函数

可以发现 test_backtrace,i386_init, _painc, _warn 这 4 个函数都属于 kern/init.c. 而 serial_pro_data, cons_intr,.... 一直到都是属于文件 kern/con-

sole.c, 可以看到这些函数也是按照他们属于的文件的顺序依次排列的. 所以我们查找一个符号的文件信息, 所在函数, 以及所在行数的的思路就很清晰了, 如果要查找所在的文件, 根据该符号的虚地址, 只要查找两个相邻的 SO 符号表项里前者地址和后者地址一起包含了该符号的地址就行了. 查找行数也类似. 但是这种方法在查找行号时出现了问题, 使用命令 objdump -G obj/kern/kernel | grep SLINE. 输出如下:

```
ldc@work: ~/os/6.828/jos

ldc@work:~/os/6.828/jos$ objdump -G obj/kern/kernel | grep FUN
95      FUN     0       0       f010003d 2867   test_backtrace:F(0,18)
105     FUN     0       0       f010009a 2908   i386_init:F(0,18)
112     FUN     0       0       f01000f5 2926   _panic:F(0,18)
128     FUN     0       0       f0100154 3022   _warn:F(0,18)
172     FUN     0       0       f01001a0 3108   serial_proc_data:f(0,1)
184     FUN     0       0       f01001bc 3132   cons_intr:f(0,18)
197     FUN     0       0       f0100200 3187   kbd_proc_data:f(0,1)
242     FUN     0       0       f0100309 3233   cons_putc:f(0,18)
300     FUN     0       0       f0100502 3269   serial_intr:F(0,18)
306     FUN     0       0       f010051e 3289   kbd_intr:F(0,18)
310     FUN     0       0       f0100530 3306   cons_getc:F(0,1)
324     FUN     0       0       f010057a 3323   cons_init:F(0,18)
365     FUN     0       0       f0100668 3370   cputchar:F(0,18)
370     FUN     0       0       f0100678 3396   getchar:F(0,1)
377     FUN     0       0       f0100689 3411   iscons:F(0,1)
423     FUN     0       0       f01006a0 4077   mon_help:F(0,1)
430     FUN     0       0       f0100701 4157   mon_kerninfo:F(0,1)
443     FUN     0       0       f01007b2 4201   mon_backtrace:F(0,1)
469     FUN     0       0       f0100881 4306   monitor:F(0,18)
503     FUN     0       0       f01009bb 4366   read_eip:F(0,4)
532     FUN     0       0       f01009c3 4433   putch:f(0,18)
539     FUN     0       0       f01009d6 4486   vcprintf:F(0,1)
```

图 8: 行号输出

可以看到有点符号的地址变成了很小的整数, 这样我们在以 eip 这样的指针进行查询是肯定队不上号, 不过前面我们提到的哪种包含关系, 我们可以通过一次地址转换后做到, 具体来看代码, 在 kern/kdebug.c 文件 stab_binsearch 函数中:

```
1   lfun = lfile;
2      rfun = rfile;
3      stab_binsearch(stabs, &lfun, &rfun, N_FUN, addr);
4
5      if (lfun <= rfun) {
6          // stabs[lfun] points to the function name
7          // in the string table, but check bounds just in case.
8          if (stabs[lfun].n_strx < stabstr_end - stabstr)
9              info->eip_fn_name = stabstr + stabs[lfun].n_strx;
10         info->eip_fn_addr = stabs[lfun].n_value;
11         addr -= info->eip_fn_addr;
12         // Search within the function definition for the line number.
13         lline = lfun;
14         rline = rfun;
15     } else {
```

```
        // Couldn't find function stab!  Maybe we're in an assembly
        // file.  Search the whole file for the line number.
        info->eip_fn_addr = addr;
        lline = lfile;
        rline = rfile;
    }
```

注意 addr -= info->eip_fn_addr, 这就是我们刚提到的如果查找行号时, 表项里的地址很小的话可以进行地址转换. 为什么转换? 在/kern/kdebug.c 文件 debuginfo_eip () 函数中添加的代码如下:

```
stab_binsearch(stabs, &lline, &rline, N_SLINE, addr);
     if(lline <= rline) {
         info->eip_line = stabs[lline].n_desc;
     } else {
         return -1;
     }
```

最后对 kern/monitor.c 中的 mon_backtrace 函数进行修改, 添加 debuginfo_eip 函数, 以及相应的 cprinf, 最终的 mon_backtrace 如下:

```
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
    // Your code here.
    uintptr_t *ebp, *eip;
    int i;
    struct Eipdebuginfo info;
    uint32_t args[5];
    cprintf("Stack␣backtrace:\n");
    ebp =(uintptr_t*) read_ebp();
    while(ebp != 0)
    {
        eip = ebp+1;
        for(i = 0; i < 5; i++)
            args[i] = *(ebp+2+i);
        cprintf("␣ebp␣%08x␣eip␣%08x␣args␣%08x␣%08x␣%08x␣%08x␣%08x\n", ebp,
        *eip,
            args[0], args[1], args[2], args[3], args[4]);
        debuginfo_eip(*eip, &info);
        cprintf("␣␣␣␣␣%s:%d:␣%.*s+%d\n",info.eip_file, info.eip_line,
            info.eip_fn_namelen, info.eip_fn_name, (*eip)-info.eip_fn_addr)
        ;
        ebp = (uintptr_t*)(*ebp);
    }
    return 0;
}
```

添加 backtrace 命令到 kernel monitor 中, 只需要在 struct Command commands 数组中多加一项 backtrace, 代码如下

```
static struct Command commands[] = {
```

```
2       { "help", "Display␣this␣list␣of␣commands", mon_help },
3       { "kerninfo", "Display␣information␣about␣the␣kernel", mon_kerninfo },
4       { "backtrace", "backtrace␣the␣stack", mon_backtrace},
5   }
```

1.2 LAB2: Memory Management

1.2.1 Part 1: Physical Page Management

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.

这几个函数主要是对物理内存进行管理，初始化和分配，下面给出详细的实现

boot_alloc

```
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.
  if(n == 0)
    return nextfree;
    if((uint32_t)(nextfree + n) > KERNBASE + PTSIZE)
    panic("boo_alloc: out of memory\n");
  else {
    result = nextfree;
```

```
    nextfree = nextfree + n;
    nextfree = ROUNDUP(nextfree, PGSIZE);
    return result;
  }
  return NULL;
}
```

boot_alloc 只使用 [KERNBASER, KERNBASE+PTSIZE) 这 4MB 的空间.

因为此时的页表只有这段地址空间的映射，高于 KERNBASE+PTSIZE 的空间未被页表映射，无法使用所以限制 nextfree 的范围在 [KERNBASER, KERNBASE+PTSIZE) 内.

page_init

```
void
page_init(void)
{
  // The example code here marks all physical pages as free.
  // However this is not truly the case.  What memory is free?
  //  1) Mark physical page 0 as in use.
  //     This way we preserve the real-mode IDT and BIOS structures
  //     in case we ever need them.  (Currently we don't, but...)
  //  2) The rest of base memory, [PGSIZE, npages_basemem * PGSIZE)
  //     is free.
  //  3) Then comes the IO hole [IOPHYSMEM, EXTPHYSMEM), which must
  //     never be allocated.
  //  4) Then extended memory [EXTPHYSMEM, ...).
  //     Some of it is in use, some is free. Where is the kernel
  //     in physical memory?  Which pages are already in use for
  //     page tables and other data structures?
  //
  // Change the code to reflect this.
  // NB: DO NOT actually touch the physical memory corresponding to
  // free pages!
  /*size_t i;
  for (i = 0; i < npages; i++) {
    pages[i].pp_ref = 0;
    pages[i].pp_link = page_free_list;
    page_free_list = &pages[i];
  } */
  // my code:
  size_t i;
  extern char end[];
  for(i = 0; i < npages; i++) {
    if(i == 0) {
      pages[i].pp_ref = 0;
      continue;
    }
    if(i < IOPHYSMEM / PGSIZE) {
      pages[i].pp_ref = 0;
      pages[i].pp_link = page_free_list;
      page_free_list = &pages[i];
      continue;
```

```
    }
    if(i < EXTPHYSMEM / PGSIZE) {
      pages[i].pp_ref = 0;
      continue;
    }
    if(i < (PADDR(ROUNDUP((char*) end, PGSIZE)) / PGSIZE)) {
      pages[i].pp_ref = 0;
      continue;
    }
    if(i < PADDR(boot_alloc(0))/ PGSIZE) {
      pages[i].pp_ref = 0;
      continue;
    }
    pages[i].pp_ref = 0;
    pages[i].pp_link = page_free_list;
    page_free_list = &pages[i];
  }
}
```

page_init 执行时，内核，页目录表 kern_pgdir, 以及 pages 数组所占的物理内存已经分配使用，所以不加入 page_free_list.

page_alloc

```
struct Page *
page_alloc(int alloc_flags)
{
  // Fill this function in
  struct Page *result;
  if(page_free_list != NULL) {
    result = page_free_list;
    result->pp_ref = 0;
    page_free_list = page_free_list->pp_link;
    if(alloc_flags & ALLOC_ZERO)
      memset(page2kva(result), 0, PGSIZE);
    return result;
  }
  return 0;
}
```

page_alloc 每次分配一页, 返回该页对应的 struct Page 结构指针. 如果 alloc_flags 中 ALLOC_ZERO 置位，对应的物理页清 0.

page_free

```
void
page_free(struct Page *pp)
{
  // Fill this function in
  assert(pp->pp_ref == 0);
  pp->pp_link = page_free_list;
  page_free_list = pp;
}
```

采用头插法将对应的空闲 struct page 结构插入到 page_free_list 链表头.

1.2.2 Part 2: Virtual Memory

> 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 memory are mapped and with what permissions.

使用 xp 命令检查从物理地址 0x100000 起的 20 个字大小的内存内容.

```
+ ld boot/boot
boot block is 380 bytes (max 510)
+ mk obj/kern/kernel.img
sed "s/localhost:1234/localhost:26000/" < .gdbinit.tmpl > .gdbinit
***
*** Now run 'gdb'.
***
/usr/local/bin/qemu-system-i386 -nographic -hda obj/kern/kernel.img -serial mo
QEMU 1.7.0 monitor - type 'help' for more information
(qemu) xp/20x 0x100000
0000000000100000: 0x1badb002 0x00000000 0xe4524ffe 0x7205c766
0000000000100010: 0x34000004 0x1000b812 0x220f0011 0xc0200fd8
0000000000100020: 0x0100010d 0xc0220f80 0x10002fb8 0xbde0fff0
0000000000100030: 0x00000000 0x111000bc 0x0002e8f0 0xfeeb0000
0000000000100040: 0x83e58955 0x8cb818ec 0x2df01139 0xf0113300
(qemu)
```

图 9: Qemu monitor xp 命令输出

使用 GDB 的 x 命令检查从虚拟地址 0xf0100000 起的 20 个字大小的内存内容.

```
(gdb) x/20x 0xf0100000
0xf0100000 <_start+4026531828>: 0x1badb002      0x00000000      0xe4524ffe      0x7205c766
0xf0100010 <entry+4>:   0x34000004      0x1000b812      0x220f0011      0xc0200fd8
0xf0100020 <entry+20>:  0x0100010d      0xc0220f80      0x10002fb8      0xbde0fff0
0xf0100030 <relocated+1>:       0x00000000      0x111000bc      0x0002e8f0      0xfeeb0000
0xf0100040 <i386_init>: 0x83e58955      0x8cb818ec      0x2df01139      0xf0113300
(gdb)
```

图 10: GDB x 命令输出

可以看到两命令的输出结果相同.

下图是 info mem 的输出

```
K> QEMU 1.7.0 monitor - type 'help' for more informatior
(qemu) info mem
00000000ef000000-00000000ef021000 0000000000021000 ur-
00000000ef7bc000-00000000ef7bf000 0000000000003000 ur-
00000000ef7c0000-00000000ef800000 0000000000040000 ur-
00000000efbf8000-00000000efc00000 0000000000008000 -rw
00000000f0000000-0000000100000000 0000000010000000 -rw
(qemu) info pg
```

图 11: info mem 输出

1. 地址 0xef000000-0xef021000 映射到 PADDR(pages).
2. 地址 0xefbf8000-0xefc00000 映射到栈 PADDR(bootstack).
3. 地址 0xf0000000-0x100000000 映射到物理地址 [0, 2^{32}-KERNBASE).
4. 0xef7bc000-0xef7bf000 与 0xef7c0000-0xef800000 的映射我们下面再分析.

下面是 info pg 的输出

```
(qemu) info pg
VPN range     Entry         Flags        Physical page
[ef000-ef3ff]  PDE[3bc]     -------UWP
  [ef000-ef020]  PTE[000-020] -------U-P 00119-00139
[ef400-ef7ff]  PDE[3bd]     -------U-P
  [ef7bc-ef7bc]  PTE[3bc]     -------UWP 003fd
  [ef7bd-ef7bd]  PTE[3bd]     -------U-P 00118
  [ef7be-ef7be]  PTE[3be]     -------UWP 003fe
  [ef7c0-ef7d0]  PTE[3c0-3d0] ----A--UWP 003ff 003fc 003fb 003fa 003f9
  [ef7d1-ef7ff]  PTE[3d1-3ff] -------UWP 003ec 003eb 003ea 003e9 003e8
[ef800-efbff]  PDE[3be]     -------UWP
  [efbf8-efbff]  PTE[3f8-3ff] --------WP 0010d-00114
[f0000-f03ff]  PDE[3c0]     ----A--UWP
  [f0000-f0000]  PTE[000]     --------WP 00000
  [f0001-f009f]  PTE[001-09f] ---DA---WP 00001-0009f
  [f00a0-f00b7]  PTE[0a0-0b7] --------WP 000a0-000b7
  [f00b8-f00b8]  PTE[0b8]     ---DA---WP 000b8
  [f00b9-f00ff]  PTE[0b9-0ff] --------WP 000b9-000ff
  [f0100-f0104]  PTE[100-104] ----A---WP 00100-00104
  [f0105-f0113]  PTE[105-113] --------WP 00105-00113
  [f0114-f0114]  PTE[114]     ---DA---WP 00114
  [f0115-f0116]  PTE[115-116] --------WP 00115-00116
  [f0117-f011a]  PTE[117-11a] ---DA---WP 00117-0011a
  [f011b-f0139]  PTE[11b-139] ----A---WP 0011b-00139
  [f013a-f03bd]  PTE[13a-3bd] ---DA---WP 0013a-003bd
  [f03be-f03ff]  PTE[3be-3ff] --------WP 003be-003ff
[f0400-f3fff]  PDE[3c1-3cf] ----A--UWP
  [f0400-f3fff]  PTE[000-3ff] ---DA---WP 00400-03fff
[f4000-f43ff]  PDE[3d0]     ----A--UWP
  [f4000-f40fe]  PTE[000-0fe] ---DA---WP 04000-040fe
  [f40ff-f43ff]  PTE[0ff-3ff] --------WP 040ff-043ff
[f4400-fffff]  PDE[3d1-3ff] -------UWP
  [f4400-fffff]  PTE[000-3ff] --------WP 04400-0ffff
(qemu)
```

图 12: info pg 输出

ef000-ef3ff PDE[3bc] ——-UWP 对应的是 pages 的映射

ef800-efbff PDE[3be] ——-UWP 对应的是栈 bootstack 的映射

f0000-f03ff PDE[3c0] —-A–UWP

f0400-f3fff PDE[3c1-3cf] —-A–UWP

f4000-f43ff PDE[3d0] —-A–UWP 后三个都是对应 [KERNBASE, 2^{32}) 到 [0, 2^{32}-KERNBASE) 的映射.

ef400-ef7ff PDE[3bd] ——-U-P

ef7bc-ef7bc PTE[3bc] ——-UWP 003fd

ef7bd-ef7bd PTE[3bd] ——-U-P 00118

ef7be-ef7be PTE[3be] ——-UWP 003fe

ef7c0-ef7d0 PTE[3c0-3d0] —-A–UWP 003ff 003fc 003fb 003fa 003f9 003f8 ..

ef7d1-ef7ff PTE[3d1-3ff] ——-UWP 003ec 003eb 003ea 003e9 003e8 003e7

..

PDE[3BD] 是 UVPT 对应的目录项，该项指向页目录表 kern_pgdir. 所以页目录表中的 3BD 项指向自身. 3BD«22 | 3BD «12 形成的地址是 0xef7bd000. 故 PTE[3BD] 存在.

在图中 [ef000-ef3ff] PDE[3bc] ——-UWP 存在, 所以存在 PTE[3bc], 因为

ef800-efbff PDE[3be] ——-UWP

f0000-f03ff PDE[3c0] —-A–UWP

f0400-f3fff PDE[3c1-3cf] —-A–UWP

f4000-f43ff PDE[3d0] —-A–UWP

f4400-fffff PDE[3d1-3ff] ——-UWP

存在，故 PTE[3be], PTE[3c03d0], PTE[3d13ff] 均存在. 所以在 info mem 中存在 0xef7bc000-0xef7bf000 与 0xef7c0000-0xef800000 的映射.

Question

Question

1Assuming 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;
```

这里是对虚拟地址进行操作的,x 的类型是 uintptr_t. x 是指向值为 10 的指针.

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 的实现如下, 从 pgdir 中找到虚地址 va 所在的 page table 的地址. 如果 page table 不存在并且 create == 1 则分配一页空间作为 va 所在的 page table. 最后从 page table 中返回 va 的 pte(page table entry).

pgdir_walk

```
1 pte_t *
2 pgdir_walk(pde_t *pgdir, const void *va, int create)
3 {
4   // Fill this function in
5   pte_t *pde;
6   pte_t *pgtab;
```

```
  struct Page *pginfo;
  pde = &pgdir[PDX(va)];
  if(*pde & PTE_P) {
    pgtab = (pte_t*)KADDR(PTE_ADDR(*pde));
    return &pgtab[PTX(va)];
  } else {
    if(create && (pginfo = page_alloc(ALLOC_ZERO))!= NULL) {
      pginfo->pp_ref++;
      pgtab = (pte_t*)page2kva(pginfo);
      pgdir[PDX(va)] = PADDR(pgtab) | PTE_P | PTE_U | PTE_W;
      return &pgtab[PTX(va)];
    } else {
      return NULL;
    }
  }
//  return NULL;
}
```

boot_map_region 的实现如下, 将虚地址 [va, va+size) 映射到物理地址 [pa, pa +size). pte 上的权限为 perm|PTE_P. 首先通过 pgdir_walk 找到虚地址 va 对应的 pte, 然后设置 pte 指向物理地址 pa 和设置权限位.

```
static void
boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa, int
        perm)
{
  // Fill this function in
  uint32_t last;
  pte_t* pte;
  va = ROUNDDOWN(va, PGSIZE);
  pa = ROUNDDOWN(pa, PGSIZE);
  last = 0;
  while(last < size) {
    pte = pgdir_walk(pgdir, (void*)va, 1);
    assert(pte != NULL);
    *pte = pa | perm | PTE_P;
    last += PGSIZE;
    pa += PGSIZE;
    va += PGSIZE;
  }
}
```

page_lookup 的实现如下, 返回 va 映射到的 page, 如果 pte_store 不为 NULL, 则在 *pte_store = pte.

```
struct Page *
page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
{
  // Fill this function in
  pte_t *pte;
  pte = pgdir_walk(pgdir, va, 0);
  if(pte != NULL) {
```

```
8       if(pte_store != NULL)
9         *pte_store = pte;
10      return pa2page(PTE_ADDR(*pte));
11    }
12
13    return NULL;
14  }
```

page_remove 的实现如下

page_remove

```
1   void
2   page_remove(pde_t *pgdir, void *va)
3   {
4     // Fill this function in
5     struct Page *pg;
6     pte_t *pte;
7     pg = page_lookup(pgdir, va, &pte);
8     if(pg != NULL) {
9       page_decref(pg);
10        *pte = 0;
11      tlb_invalidate(pgdir, va);
12    }
13  }
```

page_insert 的实现如下, 在 page_remove 之前对页引用计数增加, 否则如果这个操作放在 page_remove 之后，那么插入相同页时会出现先将这样释放，然后将这个已经 free 掉的页的引用计数加 1，那么会在空闲链表中存在引用计数不为 0 的页，并且我们认为该页已经映射，并不是空闲, 显然这是错误的.

page_insert

```
1   int
2   page_insert(pde_t *pgdir, struct Page *pp, void *va, int perm)
3   {
4     // Fill this function in
5     pte_t *pte;
6     pte = pgdir_walk(pgdir, va, 1);
7     if(pte == NULL)
8       return -E_NO_MEM;
9     assert(pte != NULL);
10    pp->pp_ref++;
11    // should be there, can not put this operation after
12    // page_remove, because if put after page_remove,
13    // when the same pp is re_inserted, the pp maybe free in page_remove.
14    // then we pp->ref++, so the the free pp have pp_ref > 0. this wrong.
15    if(*pte | PTE_P)
16      page_remove(pgdir, va);
17      *pte = page2pa(pp) | perm | PTE_P;
18    tlb_invalidate(pgdir, va);
```

```
  return 0;
}
```

1.2.3 Part 3: Kernel Address Space

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.

```
// Allocate an array of npages 'struct Page's and store it in 'pages'.
// The kernel uses this array to keep track of physical pages: for
// each physical page, there is a corresponding struct Page in this
// array.  'npages' is the number of physical pages in memory.
// Your code goes here:
pages = (struct Page *)boot_alloc(npages * sizeof(struct Page));
```

```
// Map 'pages' read-only by the user at linear address UPAGES
// Permissions:
//    - the new image at UPAGES -- kernel R, user R
//      (ie. perm = PTE_U | PTE_P)
//    - pages itself -- kernel RW, user NONE
// Your code goes here:
size_t size = ROUNDUP(npages*sizeof(struct Page), PGSIZE);
boot_map_region(kern_pgdir, (uintptr_t)UPAGES, size, PADDR(pages), PTE_U |
       PTE_P);
```

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

```
// Map all of physical memory at KERNBASE.
// Ie.  the VA range [KERNBASE, 2^32) should map to
//      the PA range [0, 2^32 - KERNBASE)
// We might not have 2^32 - KERNBASE bytes of physical memory, but
// we just set up the mapping anyway.
// Permissions: kernel RW, user NONE
// Your code goes here:
size = 0xffffffff - KERNBASE + 1;
boot_map_region(kern_pgdir, (uintptr_t)KERNBASE, size, 0, PTE_W | PTE_P);
```

Question

Question2 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 Virtual Address	Points to (logically):
1023	?	Page table for top 4MB of phys memory
1022	?	?
.	?	?
.	?	?
.	?	?
2	0x00800000	?
1	0x00400000	?
0	0x00000000	[see next question]

表 2: entry in page directory

Entry	BASE Virtual Address	Point to (logically)
0x3c0-3ff	0xf0000000	Page table for 4MB of phys memory[0 2^{32}-KERNBASE)
0x3be	0xef800000	page table for stack
0x3bd	0xef400000	cur page table kern_pgdir(user R)
0x3bc	0xef000000	page table for pages
0-3bb	Not map	Not map

Question3 (From Lecture 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?

因为内核地址空间的页表标志位设为用户不可访问，如果用户可以随意读写内核很可能导致系统崩溃.

Question4 What is the maximum amount of physical memory that this operating system can support? Why?

256MB，因为系统的地址空间是 [0xf0000000, 2^{32}). 最多支持 256M, 还有一种算法是 pages 映射到 PTSIZE 大小的地址空间, 可以根据 pages 数组中 page 的个数来算算支持的内存大小, pages 数组最大 PTSIZE.　可支持的内存 1.3G 左右, .

Question5 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?

管理物理页面需要的总开销为:1 个页目录,1024 个页表, 每个页面为 4KB 大小, 那么 256MB 的物理页面共有 64k 个页面，而每个物理页面需要一个 Page 数组来管理, 那么总的开销为:(1 + 1024) * 4KB + 64k * sizeof(struct Page) = 4740 KB, 我认为　因为页目录中有一项是指向自身, 所以页表实际只有 1023 个, 所以总的开销是 (1 + 1023)*4KB + 64 * sizeof(struct Page), 但是只有 256MB 内存，只需要 64 个页表就行了吧? 不需要 1024 个吧？　(1+64)*4kb + 64k * sizeof(struct Page)

Question6 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?

因为我们在 entry_pgdir 中设置了 [0, 4MB) 到 [0,4MB) 的映射，所以在低地址执行是可以的, 如果没有这个映射，只有 [KERNBASE, KERBASE+4MB) 到 [0, 4MB) 的映射，那么在开启分页之后系统将启用分页机制, 但是在跳转到高地址前 eip 任然在低地址处, 不在 [KERNBASR, KERNBASE+4MB) 这个范围内, 执行会出错. 所以必须设立 [0, 4MB) 到 [0,4MB) 的映射.

Challenge! We consumed many physical pages to hold the page tables for the KERNBASE mapping. Do a more space-efficient job using the PTE_PS ("Page Size") bit in the page directory entries. This bit was not supported in the original 80386, but is supported on more recent x86 processors. You will therefore have to refer to Volume 3 of the current Intel manuals. Make sure you design the kernel to use this optimization only on processors that support it!

首先需要检查机器是否支持 PSE, 可以使用 CPUID 命令.EAX 寄存器存放调用号, CPUID 根据 EAX 寄存器中的值返回对应的信息，当 EAX=1，CPUID 命令执行后在 EDX 寄存器中的第三位 (从 0 开始计算) 表明是否支持 PSE. 新建分支 lab2challenge1 用于 PSE 所做的修改, 具体代码在该分支上，只对 KERNBASE 以上地址空间使用了 PSE, 在加载 kern_pgdir 到 cr3 寄存器之前必须将 cr4 寄存器的 CR4_PSE 位置 1, 实验结果如图,0xf0000000 以上空间只有 PDE 没有 PTE，且权限位 PTE_PS 已经设置，显然 0xf0000000 以上空间使用了 4MB 页.

```
Type 'help' for a list of commands.
K> QEMU 1.7.0 monitor - type 'help' for more information
(qemu) info mem
00000000ef000000-00000000ef021000 0000000000021000 ur-
00000000ef7bc000-00000000ef7bf000 0000000000003000 ur-
00000000ef7c0000-00000000ef800000 0000000000040000 -r-
00000000efbf8000-00000000efc00000 0000000000008000 -rw
00000000f0000000-0000000100000000 0000000010000000 -rw
(qemu) info pg
VPN range     Entry         Flags        Physical page
[ef000-ef3ff]  PDE[3bc]     -------UWP
  [ef000-ef020]  PTE[000-020] -------U-P 00119-00139
[ef400-ef7ff]  PDE[3bd]     -------U-P
  [ef7bc-ef7bc]  PTE[3bc]     -------UWP 003fd
  [ef7bd-ef7bd]  PTE[3bd]     -------U-P 00118
  [ef7be-ef7be]  PTE[3be]     -------UWP 003fe
  [ef7c0-ef7d0]  PTE[3c0-3d0] --SDA---WP 00000 00400 00800 00c00 01000 01400 ..
  [ef7d1-ef7ff]  PTE[3d1-3ff] --S-----WP 04400 04800 04c00 05000 05400 05800 ..
[ef800-efbff]  PDE[3be]     -------UWP
  [efbf8-efbff]  PTE[3f8-3ff] --------WP 0010d-00114
[f0000-f43ff]  PDE[3c0-3d0] --SDA---WP 00000-043ff
[f4400-fffff]  PDE[3d1-3ff] --S-----WP 04400-0ffff
(qemu)
```

图 13: 使用 PSE

> Challenge! Extend the JOS kernel monitor with commands to:
> Display in a useful and easy-to-read format all of the physical page mappings (or lack thereof) that apply to a particular range of virtual/linear addresses in the currently active address space. For example, you might enter 'showmappings 0x3000 0x5000' to display the physical page mappings and corresponding permission bits that apply to the pages at virtual addresses 0x3000, 0x4000, and 0x5000. Explicitly set, clear, or change the permissions of any mapping in the current address space. Dump the contents of a range of memory given either a virtual or physical address range. Be sure the dump code behaves correctly when the range extends across page boundaries! Do anything else that you think might be useful later for debugging the kernel. (There's a good chance it will be!)

在 kern/monitor.c 中 showmapping 的实现，如果虚地址没有映射到物理内存则显示 not mapped 使用方法 show_map lva uva

清单:

```
1  int mon_showmappings(int argc, char** argv, struct Trapframe *tf)
2  {
3    if(argc != 3) {
4      cprintf("Usage:␣showmappings␣LOWADDR,␣UPPERADDR\n");
5      cprintf("Both␣address␣must␣be␣aligned␣in␣4KB\n");
6      return 0;
7    }
8    uintptr_t lva = strtol(argv[1], 0, 0);
9    uintptr_t uva = strtol(argv[2], 0, 0);
10   if(lva > uva) {
```

```
11      cprintf("LOWADDR␣must␣be␣less␣than␣UPPERADDR\n");
12      return 0;
13    }
14    showmappings(lva, uva);
15    return 0;
16  }
17
18  void showmappings(uintptr_t lva, uintptr_t uva)
19  {
20    pte_t *pte;
21    while(lva <= uva) {
22      cprintf("␣␣0x%08x␣␣", lva);
23      pte = pgdir_walk(kern_pgdir, (void*)lva, 0);
24      if(pte == NULL || !(*pte & PTE_P)) {
25        cprintf("not␣mapped\n");
26        lva += PGSIZE;
27        continue;
28      }
29      lva += PGSIZE;
30      cprintf("␣␣0x%08x", PTE_ADDR(*pte));
31
32      if(*pte & PTE_G)
33        cprintf("␣␣G");
34      else
35        cprintf("␣␣-");
36
37      if(*pte & PTE_PS)
38        cprintf("S");
39      else
40        cprintf("-");
41
42      if(*pte & PTE_D)
43        cprintf("D");
44      else
45        cprintf("-");
46
47      if(*pte & PTE_A)
48        cprintf("A");
49      else
50        cprintf("-");
51
52      if(*pte & PTE_PCD)
53        cprintf("C");
54      else
55        cprintf("-");
56
57      if(*pte & PTE_PWT)
58        cprintf("T");
59      else
60        cprintf("-");
61
62      if(*pte & PTE_U)
63        cprintf("U");
```

```
64      else
65        cprintf("-");
66
67
68      if(*pte & PTE_W)
69        cprintf("W");
70      else
71        cprintf("-");
72      if(*pte & PTE_P)
73        cprintf("P\n");
74      else
75        cprintf("-\n");
76    }
77  }
```

实验结果如图

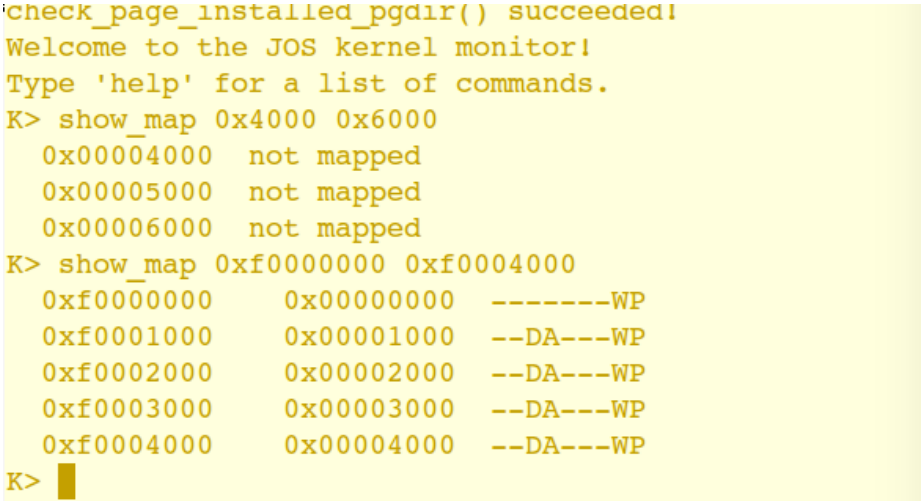


图 14: show_map

chperm 的实现, 使用方法如下: set_perm +/-perm addr, perm 可以是 k(即 kernel),u,p,w,r 的组合，+ 表示设置该位，-表示清楚对应的位. 其他的位不影响. 如 set_perm +uwp 0xf0000000.

set_perm

```
1   int mon_chperm(int argc, char **argv, struct Trapframe *tf)
2   {
3     pte_t *pte;
4     uint32_t flag;
5     char *option = argv[1];
6     uintptr_t va = strtol(argv[2], 0, 0);
7   #define CLEAR 0
8   #define SET 1
9     if(argc != 3) {
```

```
10      cprintf("Usage: chperm +/-perm addr\n");
11      return 0;
12    }
13    if(option[0] == '+')
14      flag = SET;
15    else if(option[0] == '-')
16      flag = CLEAR;
17    else {
18      cprintf("Usage: chperm +/-perm addr\n");
19      return 0;
20    }
21
22    option++;
23    if(PGOFF(va) != 0) {
24      cprintf("addr shoud be aligned at 4KB\n");
25      return 0;
26    }
27    pte = pgdir_walk(kern_pgdir, (void*)va, 0);
28    if(pte == NULL)
29      return 0;
30
31    while(*option != '\0') {
32      switch(*option) {
33      case 'k':
34        if(flag == SET)
35          *pte &= ~PTE_U;
36        else // CLEAR k
37          *pte |= PTE_U;
38        break;
39      case 'u':
40        if(flag == SET)
41          *pte |= PTE_U;
42        else // CLEAR
43          *pte &= ~PTE_U;
44        break;
45      case 'p':
46        if(flag == SET)
47          *pte |= PTE_P;
48        else
49          *pte &= ~PTE_P;
50        break;
51      case 'w':
52        if(flag == SET)
53          *pte |= PTE_W;
54        else
55          *pte &= ~PTE_W;
56        break;
57      case 'r':
58        if(flag == SET)
59          *pte &= ~PTE_W;
60        else //CLEAR
61          *pte |= PTE_W;
62        break;
```

```
    default:
      break;
    }
    option++;
  }
  return 0;
}
```

实验结果如图

```
check_page_installed_pgdir() succeeded!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> show_map 0xf0000000 0xf0000000
  0xf0000000    0x00000000  -------WP
K> set_perm +ur 0xf0000000
K> show_map 0xf0000000 0xf0000000
  0xf0000000    0x00000000  ------U-P
K> set_map -ur 0xf0000000
Unknown command 'set_map'
K> set_perm -ur 0xf0000000
K> show_map 0xf0000000 0xf0000000
  0xf0000000    0x00000000  -------WP
K>
```

图 15: 修改权限

mem_dump 的实现如下，使用方法 mem_dump -p/v la size, 例如 mem_dump -v 0xf0000000 20, 显示从虚地址 0xf0000000 起始的 20 个字的内存内容，p 表示 la 是物理地址，v 表示 la 是虚拟地址，当需要 dump 指定的物理地址时需要将物理地址 pa 转换为虚拟地址处理. 即 pa+KERNBASE, 如果对应的地址为被映射则会报错, 所以使用时确保指定的地址已经映射到物理内存. 实现中直接对虚地址进行 * 操作取内容 (即 *va) 就可读出内存中的值.

```
int mon_dump(int argc, char **argv, struct Trapframe *tf)
{
  uintptr_t la;
  uint32_t size, i;
  char flag;
  if(argc != 4)
  {
    cprintf("Usage: dump -p/v la size\n");
    cprintf("-p specify la  physical address\n");
    cprintf("-v specify la  virtual address\n");
    cprintf("the range [la, la+size] shoud be mapped, otherwise crashed\n")
        ;
    return 0;
  }
  la = strtol(argv[2], 0, 0);
  size = strtol(argv[3], 0, 0);
```

```
16    if(*argv[1] != '-')
17    {
18      cprintf("Usage:␣dump␣-p/v␣la␣size\n");
19      cprintf("-p␣specify␣la␣␣physical␣address\n");
20      cprintf("-v␣specify␣la␣␣virtual␣address\n");
21      return 0;
22
23    }
24
25    flag = *(++argv[1]);
26    if(flag == 'p')
27      la = (uintptr_t)KADDR(la);
28
29    if(la + size < la) {
30      cprintf("size␣too␣large\n");
31      return 0;
32    }
33
34    for(i = 0; i < size; i++) {
35      if(i % 4 == 0) {
36        cprintf("\n0x");
37        if(flag == 'p')
38          cprintf("%08x:", PADDR((void*)la));
39        else
40          cprintf("%08x:", la);
41      }
42      cprintf("␣␣0x");
43      cprintf("%08x", *(uintptr_t*)(la));
44      la += 4;
45    }
46    cprintf("\n");
47    return 0;
48  }
```

实验结果如下使用 mem_dump 查看

```
Type 'help' for a list of commands.
K> mem_dump -p 0x0 20

0x00000000:  0xf000ff53  0xf000ff53  0xf000e2c3  0xf000ff53
0x00000010:  0xf000ff53  0xf000ff53  0xf000ff53  0xf000ff53
0x00000020:  0xf000fea5  0xf000e987  0xf000d67b  0xf000d67b
0x00000030:  0xf000d67b  0xf000d67b  0xf000ef57  0xf000d67b
0x00000040:  0xc00083f9  0xf000f84d  0xf000f841  0xf000e3fe
K> mem_dump -v 0xf0000000 20

0xf0000000:  0xf000ff53  0xf000ff53  0xf000e2c3  0xf000ff53
0xf0000010:  0xf000ff53  0xf000ff53  0xf000ff53  0xf000ff53
0xf0000020:  0xf000fea5  0xf000e987  0xf000d67b  0xf000d67b
0xf0000030:  0xf000d67b  0xf000d67b  0xf000ef57  0xf000d67b
0xf0000040:  0xc00083f9  0xf000f84d  0xf000f841  0xf000e3fe
K>
```

图 16: mem_dump -p 0x0 和 mem_dump -v 0xf0000000

使用 GDB　x 命令查看

```
+ symbol-file obj/kern/kernel
(gdb) b monitor
Breakpoint 1 at 0xf0100d22: file kern/monitor.c, line 354.
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0xf0100d22 <monitor>:        push   %ebp

Breakpoint 1, monitor (tf=0x0) at kern/monitor.c:354
354     {
(gdb) x/20x 0xf0000000
0xf0000000:     0xf000ff53      0xf000ff53      0xf000e2c3      0xf000ff53
0xf0000010:     0xf000ff53      0xf000ff53      0xf000ff53      0xf000ff53
0xf0000020:     0xf000fea5      0xf000e987      0xf000d67b      0xf000d67b
0xf0000030:     0xf000d67b      0xf000d67b      0xf000ef57      0xf000d67b
0xf0000040:     0xc00083f9      0xf000f84d      0xf000f841      0xf000e3fe
(gdb) c
Continuing.
```

图 17: GDB x/20x 0xf0000000

可以看出使用 -p 查看地址 0x0 和 -v 查看 0xf0000000 和 GDB x 命令查看 0xf0000000 所得结果相同.

> Challenge! Write up an outline of how a kernel could be designed to allow user environments unrestricted use of the full 4GB virtual and linear address space. Hint: the technique is sometimes known as "follow the bouncing kernel." In your design, be sure to address exactly what has to happen when the processor transitions between kernel and user modes, and how the kernel would accomplish such transitions. Also describe how the kernel would access physical memory and I/O devices in this scheme, and how the kernel would access a user environment's virtual address space during system calls and the like. Finally, think about and describe the advantages and disadvantages of such a scheme in terms of flexibility, performance, kernel complexity, and other factors you can think of.

这个不会，也没找到相关资料.

Challenge! Since our JOS kernel's memory management system only allocates and frees memory on page granularity, we do not have anything comparable to a general-purpose malloc/free facility that we can use within the kernel. This could be a problem if we want to support certain types of I/O devices that require physically contiguous buffers larger than 4KB in size, or if we want user-level environments, and not just the kernel, to be able to allocate and map 4MB superpages for maximum processor efficiency. (See the earlier challenge problem about PTE_PS.) Generalize the kernel's memory allocation system to support pages of a variety of power-of-two allocation unit sizes from 4KB up to some reasonable maximum of your choice. Be sure you have some way to divide larger allocation units into smaller ones on demand, and to coalesce multiple small allocation units back into larger units when possible. Think about the issues that might arise in such a system.

类似 linux 中的伙伴系统，未做实现.

1.3 LAB3:User Environments

1.3.1 Part A: User Environments and Exception Handling

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.

从启动内存分配 NENV 个 struct Env 结构

```
1 envs = (struct Env *)boot_alloc(NENV *sizeof(struct Env));
```

映射到 UENVS, 用户/内核可读

```
1 size = ROUNDUP(NENV*sizeof(struct Env), PGSIZE);
2 boot_map_region(kern_pgdir, (uintptr_t)UENVS, size, PADDR(envs), PTE_U |
        PTE_P);
```

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 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"

env_int 必须要保证 env 在空闲链表中的次序和数组中的次序相同, 首次分配 env 时必须返回 envs[i],env_int 代码如下

```
void
env_init(void)
{
    // Set up envs array
    // LAB 3: Your code here.
    int32_t i;
    for(i = NENV - 1; i >= 0; i--) {
        envs[i].env_status = ENV_FREE;
        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();
}
```

env_setup_vm 初始化 Env e 页目录的内核虚地址空间部分. 首先分配一页作为 e 的页目录表. 然后直接将内核映射部分拷贝到 e 的页目录上，UVPT 映射到 env e 的页目录.

```
static int
env_setup_vm(struct Env *e)
{
    int i;
    struct Page *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++;
    memmove(page2kva(p), kern_pgdir, PGSIZE);
    e->env_pgdir = (pde_t*)page2kva(p);
    e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_P | PTE_U;
    return 0;
}
```

region_alloc 为 Env e 分配和映射物理内存, 为 env e 分配 len 字节的物理内存, 并且映射到 env 虚地址空间的地址 va，不需要清 0 或者初始化映射到的物理页，分配的页用户/内个均可写

```
static void
region_alloc(struct Env *e, void *va, size_t len)
{
    struct Page *p;
    pte_t *pte;
    assert((uintptr_t)va < KERNBASE);
    if((uintptr_t)va + len < (uintptr_t)va)
        panic("len␣too␣large");
    if(len <= 0)
        return ;
    uintptr_t upper = ROUNDUP((uintptr_t)va+len, PGSIZE);
    assert(upper <= KERNBASE);
    va = ROUNDDOWN(va, PGSIZE);
    while((uintptr_t)va < upper) {
        p = page_alloc(0);
        if(p == NULL)
            panic("page_alloc␣failed");
        assert(p != NULL);
//can use page_insert(e->env_pgdir, va, PTE_U | PTE_W)
        p->pp_ref++;
//      boot_map_region(e->env_pgdir, va, PGSIZE, page2pa(p), PTE_U | PTE_W
        | PTE_P);
//      can use page_insert
        pte = pgdir_walk(e->env_pgdir, va, 1);
        if(pte == NULL)
```

```
25              panic("pgdir_walk failed");
26          *pte = page2pa(p) | PTE_U | PTE_W | PTE_P;
27          tlb_invalidate(e->env_pgdir, va);
28          va += PGSIZE;
29      }
30  }
```

load_icode 初始化进程的代码段，栈，数据等等和相应的处理器标志. 从 elf 程序头地址 binary 处加载所有可以加载的段到 env e 的虚地址空间. 标志需要映射到内存却在 elf 文件中不存在的空间全部清 0.

1. 扫描 elf 文件中的各个段
2. 各个段的 p_va 字段记录了该段的起始虚地址, p_memsz 记录了该段在内存中的大小. 调用 region_alloc 为 env e 分配 p_memsz 大小的内存并映射到虚地址 p_va.
3. 将 elf 文件中的数据复制到刚刚分配的物理页中. 即拷贝到进程的地址空间中. 第一个进程的数据和代码内核链接到了一起，在 boot 的时候一起载入了内存 binary 处所以不需要访问文件去读取对应的 elf 格式文件.
4. p_memsz 总是大于等于 p_filesz. 大于的部分清 0
5. e_entry 字段表示 elf 格式文件中程序的入口地址. 即从该地址开始执行
6. 最后映射 PGSIZE 大小的栈，栈是朝低地址增长的. 所以从 USTACKTOP-PGSIZE 开始映射
7. 在开始地方切换到 env e 的页表 e->env_pgdir 是为了方便. 因为 p_va 是用户空间的地址. 内核的页表是 kern_pgdir 访问不了该虚地址. 必须将 env e 的页表 e->env_pgdir 得到物理地址, 然后转换位内核空间虚地址 kva，内核才能访问, 才能将 elf 格式文件中的各个段 memmove 到 kva 中. 内核是无法直接 memmove 到 p_va 的, 在结尾处切换回内核的页表 kern_pgdir.

```
1   static void
2   load_icode(struct Env *e, uint8_t *binary, size_t size)
3   {
4
5
6       struct Elf *elf;
7       struct Proghdr *ph, *eph;
8       elf = (struct Elf *) binary;
9       ph = (struct Proghdr *)((uint8_t*)binary + elf->e_phoff);
10      eph = ph + elf->e_phnum;
11      lcr3(PADDR(e->env_pgdir));
12      for(; ph < eph; ph++) {
13          if(ph->p_type != ELF_PROG_LOAD)
14              continue;
15          assert(ph->p_memsz >= ph->p_filesz);
16          region_alloc(e, (void *)ph->p_va, ph->p_memsz);
17          memmove((uint8_t*)ph->p_va, (uint8_t*)binary+ph->p_offset, ph->
            p_filesz);
```

```
18          memset((uint8_t*)ph->p_va + ph->p_filesz, 0, ph->p_memsz - ph->
            p_filesz);
19      }
20      e->env_tf.tf_eip = elf->e_entry;
21       region_alloc(e, (void*)(USTACKTOP-PGSIZE), PGSIZE);
22   lcr3(PADDR(kern_pgdir));
23  }
```

env_create 创建一个 env, 执行 elf binary 处的代码数据.

1. env_alloc 从 env_free_list 中分配一个 Env e. 并且 parent id 置为 0. 在 env_alloc 中已经为 Env e 建立好了内核部分的映射.
2. 调用 load_icode 为这个 Env e 分配内存, 建立地址空间, 将 elf binary 的数据拷贝到地址空间中.
3. 设置 env_type

```
1   void
2   env_create(uint8_t *binary, size_t size, enum EnvType type)
3   {
4       // LAB 3: Your code here.
5       struct Env *e;
6       int32_t err;
7       err = env_alloc(&e,0);
8       if(err != 0)
9           panic("env_alloc␣failed:␣%e", err);
10      load_icode(e, binary, size);
11      e->env_type = type;
12  }
```

env_run 从当前 Env curenv 切换到 Env e. 如果第一次调用 env_run, curenv 为 NULL.

1. 如果不是第一次调用, 当前进程状态变为 ENV_RUNNABLE
2. 将当前进程设为 e, 状态设为了 ENV_RUNNING
3. 更新 env_runs, 每调度一次加 1.
4. 切换到 Env e 的页表. env_tf 中恢复 e 的执行上下文.
5. 这个函数不会返回, 并且不会执行到 panic 处.

```
1   void
2   env_run(struct Env *e)
3   {
4    if(curenv != NULL) {
5           if(curenv->env_status == ENV_RUNNING)
6               curenv->env_status = ENV_RUNNABLE;
7       }
8       curenv = e;
9       curenv->env_status = ENV_RUNNING;
10      curenv->env_runs++;
11      lcr3(PADDR(curenv->env_pgdir));
12      env_pop_tf(&(curenv->env_tf));
13
14      panic("env_run␣not␣yet␣implemented");
```

```
15 }
```

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
3. pushl
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.

在 trapentry.S 中创建了 256 个 handler. 对应 handler 地址是 vector0-vector255, 存储在 vectors 数组中. 这部分重复代码很多, 可以用脚本自动生成. 每个 handler 首先将 error_code 压栈, 再将中断/trap 号压栈. 最后投跳转到 _alltraps 处. _alltraps 在栈上构造处一个 struct Trapframe. 最后调用 trap 函数.

```
1  .text
2  .global _alltraps
3  _alltraps:
4      pushl %ds
5      pushl %es
6      pushal;
7      movw $(GD_KD), %ax
8      movw %ax, %ds
9      movw %ax, %es
10     pushl %esp
11     call trap
12     call env_pop_tf
```

在 trap_init 中设置中断描述符表, 其中共 256 个描述符. 系统调用的中断 DPL 为 3.

```
void
trap_init(void)
{
    extern struct Segdesc gdt[];

    // LAB 3: Your code here.
    extern uint32_t vectors[];
    uint32_t i;
    for(i = 0; i < 256; i++)
        SETGATE(idt[i], 0, GD_KT, vectors[i], 0);
    SETGATE(idt[T_BRKPT], 0, GD_KT, vectors[T_BRKPT], 3);
    SETGATE(idt[T_SYSCALL], 0, GD_KT, vectors[T_SYSCALL], 3)

    // Per-CPU setup
    trap_init_percpu();
}
```

Challenge! You probably have a lot of very similar code right now, between the lists of TRAPHANDLER in trapentry.S and their installations in trap.c. Clean this up. Change the macros in trapentry.S to automatically generate a table for trap.c to use. Note that you can switch between laying down code and data in the assembler by using the directives .text and .data.

Challenge! kern 目录下 vectors.pl 脚本可以自动生成各个 handler. 自动生成的代码先重定向到一个文件, 然后从该文件复制到 trapentry.S 中.

```
#!/usr/bin/perl -w
# Based on xv6's␣vectors.pl
# Generate the trap/interrupt entry points,
# which needs to be pasted into kern/trapentry.S
# This isn't␣a␣nice␣hack,␣but␣does␣work␣in␣a␣simple␣way.

for(my $i = 0; $i < 256; $i++){
    if(($i < 8 || $i > 14) && $i != 17) {
        print "TRAPHANDLER_NOEC(vector$i,$i)\n";
    }
    else {
        print "TRAPHANDLER(vector$i,$i)\n";
    }
}

print "\n.data\n";
print ".globl␣vectors\n";
print "vectors:\n";
for(my $i = 0; $i < 256; $i++){
    print "␣␣.long␣vector$i\n";
}
```

Answer the following questions in your answers-lab3.txt:

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?)
2. 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)?

Question

1. 因为在 handler 中要将中断号压栈, 如果系统只有一个 handler 并且所有进程中断都跳到这一个处理程序, 那么就会正确设置对应的中断号. 本来应该是一个 handler 压栈对应的中断号的. 现在只有一个中断 handler. 就无法判断是那个正确设定中断号
2. 因为在 trap.c 中把 14 号中断 (pagefault) 的中断描述符 DPL 设为 0. 用户进程在执行时 CPL=3. CPL >= DPL. 所以权限检查不会通过, 产生 GP 中断 (General protection).GP 为 13 号中断. 用户用 int 指令调用中断是不会压入错误代码的?

1.3.2 Part B: Page Faults, Breakpoints Exceptions, and System Calls

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.

修改 trap_dispatch 函数如下, 检查中断号, 进入对应的 case 进行处理.

```
static void
trap_dispatch(struct Trapframe *tf)
{
    // Handle processor exceptions.
    // LAB 3: Your code here.
    switch(tf->tf_trapno) {
        case T_PGFLT:
            page_fault_handler(tf);
            break;
```

```
        case T_BRKPT:
            monitor(tf);
            break;
        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);
            break;
        default:
            print_trapframe(tf);
            if (tf->tf_cs == GD_KT)
                panic("unhandled␣trap␣in␣kernel");
            else {
                env_destroy(curenv);
                return;
            }
            break;
    }
    // Unexpected trap: The user process or the kernel has a bug.
}
```

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.

Exercise 5 给出了 lab3 trap_dispatch 函数的完整代码. 其中处理 break point 异常的代码是

```
case T_BRKPT:
            monitor(tf);
            break;
```

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

Question

1. 在 trap.c 的 trap_init 函数中 break point exception 对应的中断描述符 DPL 设为 3. 因此可以在用户空间触发该中断. 如果 DPL 设为 0, 则权限检查不会通过. 会产生 GP(General protection) 中断.
2. 权限检查. 用户不是什么都可以访问的.

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. You may also find it helpful to read inc/syscall.h.
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.

trap_init 函数中对系统调用对应的中断描述符进行设置,DPL 为 3

```
SETGATE(idt[T_SYSCALL], 0, GD_KT, vectors[T_SYSCALL], 3)
```

在 trap_dispatch 中添加代码处理系统调用的情形, 返回值存放在 eax 寄存器中

```
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);
            break;
```

在 syscall.c 中 syscall 函数代码如下. 如果系统调用号无效则返回 -E_INVAL. 根据系统调用号调用不同的函数.

```
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_cputs:
```

```
            sys_cputs((char*)a1, a2);
            return 0;
        case SYS_cgetc:
            return sys_cgetc();
        case SYS_getenvid:
            return sys_getenvid();
        case SYS_env_destroy:
            return sys_env_destroy(a1);
        default:
            return -E_INVAL;
    }
    panic("syscall␣not␣implemented");
}
```

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.

lib/libmain.c/libmain 函数中 thisenv 的初始化如下

```
    thisenv = 0;
    envid_t eid;
    eid = sys_getenvid();
    thisenv = (struct Env *)envs + ENVX(eid);
```

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.

在 page_fault_handler 中检查 pagefault 是否发生在内核态. 如果发生在内核态直接 panic, 在 page_fault_handler 中代码如下

```
if((tf->tf_cs & 3) == 0) {
        panic("page␣fault␣happens␣in␣kernel␣mode");
    }
```

user_mem_check 检查 env 具有访问虚地址 [va, va+len) 的权限 perm|PTE_P

```
int
user_mem_check(struct Env *env, const void *va, size_t len, int perm)
{
    // LAB 3: Your code here.
    uint8_t* low;
    assert((uintptr_t)va < (uintptr_t)va + len);
    uint8_t* upper = (uint8_t*)va + len;
    low = (uint8_t*)ROUNDDOWN((uintptr_t)va, PGSIZE);
    upper = (uint8_t*)ROUNDUP((uintptr_t)upper, PGSIZE);
    pte_t *pte;
    perm = perm | PTE_P;
    for(; low < upper; low += PGSIZE){
        if((uintptr_t)low > ULIM) {
            user_mem_check_addr = (uintptr_t)va;
            return -E_FAULT;
        }
        pte = pgdir_walk(env->env_pgdir, low, 0);
        if(pte == NULL) {
            user_mem_check_addr = (uintptr_t)va;
            return -E_FAULT;
        }
```

```
        if(((*pte) & perm) != perm) {
            user_mem_check_addr = (uintptr_t)va;
            return -E_FAULT;
        } else {
            va = va - PGOFF(va) + PGSIZE;
        }
    }
    return 0;
}
```

在 syscall.c/sys_cputs() 函数中添加权限检查代码

```
static void
sys_cputs(const char *s, size_t len)
{
    // Check that the user has permission to read memory [s, s+len).
    // Destroy the environment if not.

    // LAB 3: Your code here.
    user_mem_assert(curenv, s, len, PTE_U);

    // Print the string supplied by the user.
    cprintf("%.*s", len, s);
}
```

最后修改 kern/kdebug.c 中的 debuginfo_eip, 调用 user_mem_check 检查 usd, stabs. stabstr.

```
  if(user_mem_check(curenv, usd, sizeof(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.
        int len = (stab_end - stabs) * sizeof(struct Stab);
        if(user_mem_check(curenv, stabs, len, PTE_U) < 0)
            return -1;
        len = stabstr_end - stabstr;
        if(user_mem_check(curenv, stabstr, len, PTE_U) < 0)
            return -1;
```

运行 breakpoint 然后 backtace 会产生 pagefault. 在 kernel panic 之前输出 trapframe 信息, 在 page_fault_handler 函数中 int perm 之前添加 print_trapframe(tf), 输出 trapframe 的信息用于调试 (调试完后删除这行代码). 输出结果如下

```
Incoming TRAP frame at 0xefbffe7c
TRAP frame at 0xefbffe7c
  edi  0x00000001
  esi  0xefbffef0
  ebp  0xefbfff10
  oesp 0xefbffe9c
  ebx  0xeebfdff0
  edx  0x00000000
  ecx  0x000003d4
  eax  0x00000002
  es   0x----0010
  ds   0x----0010
  trap 0x0000000e Page Fault
  cr2  0xeebfe000
  err  0x00000000 [kernel, read, not-present]
  eip  0xf0100797
  cs   0x----0008
  flag 0x00000093
```

图 18: trapframe 输出信息

根据 cr2 寄存器的值可以确定发生 pagefault 的地址是 0xeebfe000, 这个地址就是 USTACKTOP. 根据 err 可以看出这是一个内核读不存在页产生的 fault. 用户程序 breakpoint 从/lib/entry.S 中开始执行, 先将两个参数 0 进栈 (8 字节). 然后调用 call libmain 将返回地址 pc 压栈. 进入 libmain 后将 ebp 压栈 (x86 函数调用规则). 最后 movl %esp %ebp. 此时总共将 4 个字 (16byte) 的内容放入栈 USTACKTOP 中. 所以 esp 为 USTACKTOP-16 即 0xeebfdff0. 所以 ebp 初始为 0xeebfdff0. 当 backtrace 进入到 libmain 的栈帧 (stack frame, 就是编译里面的活动记录), 而且从栈中取 libmain 的参数

```
\\kern/monitor.c
 eip = ebp+1;
 for(i = 0; i < 5; i++)
  args[i] = *(ebp+2+i);
```

但是栈中只有两个参数 0. 且第二个参数的地址是 0xeebfdffc. 第三个参数的地址是 0xeebfe000(USTACKTOP). 但是这个地址在 JOS 的内存布局 (layout) 中没有映射到任何物理页是一个 Empty memory, Jos memory layout:

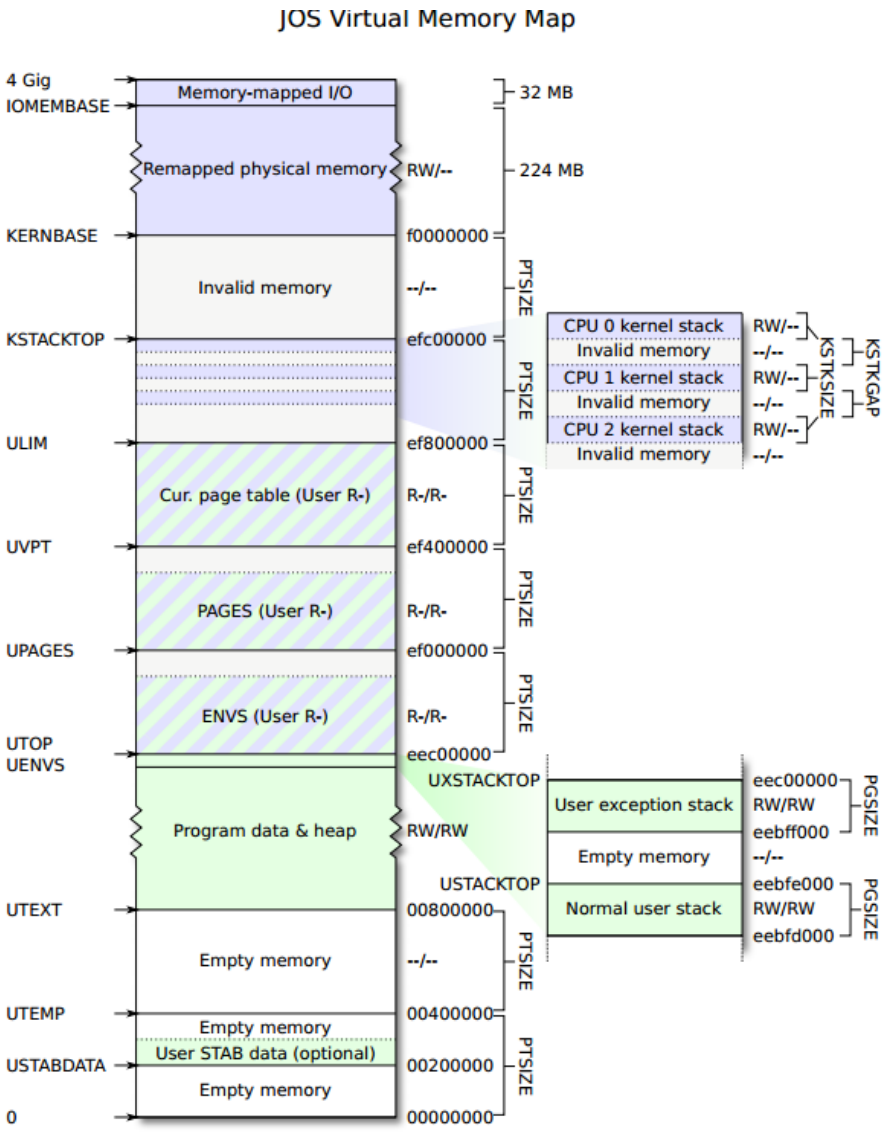


图 19: Jos memory layout

所以最终发生 pagefault. 而且系统处于内核态, 所以 panic.

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 f0100020 [00001000] free env 00001000

1.4 LAB4: Preemptive Multitasking

第四个实验在 make grade 允许到最后一个 primes 程序时出现下面的错误 missing 'CPU .: 1877 .00001128. new env 00001129' 解决办法: grade_functions.sh 换成 MIT-6.828-Adventure 中的 grade_functions.sh

env_init_percpu 中下面代码意思是?

```
asm volatile("ljmp␣\%0,$1f\n␣1:\n" :: "i" (GD_KT));
```

1.4.1 Part A: Multiprocessor Support and Cooperative Multitasking

Exercise 1. 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).

在 kern/pmap.c 中 page_init 函数添加下面的代码

```
if(i == 0 || i ==(MPENTRY_PADDR / PGSIZE)) {
                pages[i].pp_ref = 0;
                continue;
}
```

如果物理页好是 MPENTRY_PADDR/PGSIZE, 则将该页不加如空闲页链表.

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.

Question 1 因为 boot.S 是 bootloader,boot.S 中的代码直接被 BIOS 加载到了物理内存 0x7c00 处, 并且链接时使用 -Ttext 0x7c00 将链接地址设为 7c00. 因此 boot.S 中所使用的符号的地址也就是对应的物理地址, 链接地址和加载地址一样. mpentry.S 中的代码却是链接到 KERNBASE 以上, 而且在加载时也加载到了 1M(物理地址) 以上, 这段代码复制到物理内存 0x7000 处执

行.MPBOOTPHYS 就是为了计算 mpentry.S 中符号在 0x7000 处对应的物理地址.

Exercise 2. Modify mem_init_mp() (in kern/pmap.c) to map per-CPU stacks starting at KSTACKTOP, as shown in our revised 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().

添加的代码如下

```
int i;
    for(i = 0; i < NCPU; i++) {
        uintptr_t kstacktop_i = KSTACKTOP - i * (KSTKSIZE + KSTKGAP);
        boot_map_region(kern_pgdir, kstacktop_i - KSTKSIZE, KSTKSIZE, PADDR
        (percpu_kstacks[i]), PTE_W | PTE_P);
    }
```

从 KSTACKTOP 开始, 每个 CPU 栈大小位 KSTKSIZE(8 页), 依次往下, 为了防止栈溢出污染, 每两个栈直接有 KSTKGAP(8 页) 大小的 guard page. 栈映射到物理内存 PADDR(percpu_kstacks[i]), 内核可读写, 用户无权限访问, 而 guard page 未映射到内存.

Exercise 3. 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.)

代码如下

```
thiscpu->cpu_ts.ts_esp0 = (uintptr_t)percpu_kstacks[thiscpu->cpu_id];
    thiscpu->cpu_ts.ts_ss0 = GD_KD;
    gdt[(GD_TSS0 >> 3) + cpunum()] = SEG16(STS_T32A, (uint32_t) &(thiscpu->
        cpu_ts),
                                          sizeof(struct Taskstate), 0);
    gdt[(GD_TSS0 >> 3) + cpunum()].sd_s = 0;
    ltr(GD_TSS0 + (cpunum() << 3));
    lidt(&idt_pd);
```

首先获取当前 cpu 的内核栈地址 percpu_kstacks[thiscpu->cpu_id], 将当前 cpu 的 TSS 中 esp0,ss0 分别设置成内核栈地址, GD_KD. 这样当在这个 cpu 上发生中断或者 trap 时, 用户栈将自动切换到内核栈 (GD_KD:percpu_kstacks[thiscpu->cpu_id]). thiscpu->cpu_ts 作为当前 cpu 的 TSS. sd_s 置为 0 表示系统段. 最后加载 TSS selector 到 TR 寄存器中. 这样硬件就能自动从 TSS 中获取 ss0:esp0. ss0:esp0 在发生栈切换时作为内核栈地址.

Exercise 4. Apply the big kernel lock as described above, by calling lock_kernel() and unlock_kernel() at the proper locations.

在如下的地方添加 lock_kernel, 具体见代码.

1. i386_init 中, 在 BSP 唤醒其他 CPU 前需要获取内核锁, 即在 boot_aps 添加 lock_kernel.
2. mp_main 中, 初始化 AP 后, 需要获取内核锁, 再调用 sched_yield 开始在 AP 上运行用户进程. 所以在 sched_yield 前添加 lock_kernel.
3. 在 trap 函数中, 当从用户模式 trap 进入到内核模式时, 需要获取内核锁. 在 trap 函数中添加 lock_kernel
4. 在 env_run 函数中, 在从内核返回到用户时, 释放内核锁. 在 env_run 中添加 unlock_kernel

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.

Question 2 例如进程 A 发生中断, 在获取内核锁之前将会自动的使用内核栈, 假设此时进程 B 也发生中断, 在获取内核锁前也会自动的使用内核栈, 如果使用一个内核栈, 那么栈中的内容就有可能被污染.

Exercise 5. Implement round-robin scheduling in sched_yield() as described above. Don't forget to modify syscall() to dispatch sys_yield().
Modify kern/init.c to create three (or more!) environments that all run the program user/yield.c. You should see the environments switch back and forth between each other five times before terminating, like this:
... Hello, I am environment 00001008. Hello, I am environment 00001009. Hello, I am environment 0000100a. Back in environment 00001008, iteration 0. Back in environment 00001009, iteration 0. Back in environment 0000100a, iteration 0. Back in environment 00001008, iteration 1. Back in environment 00001009, iteration 1. Back in environment 0000100a, iteration 1. ... After the yield programs exit, when only idle environments are runnable, the scheduler should invoke the JOS kernel monitor. If any of this does not happen, then fix your code before proceeding.

round-robin 的实现如下

```
1  int  cur;
2    if(curenv)
```

```
        cur = ENVX(curenv->env_id) + 1;
    else
        cur = 0;
```

如果存在前一个进程存在 cur 置为当前进程的下一个进程索引 (即在 envs 数组中的下标), 否在 cur 置为 0. 接下来从索引 cur 进行搜索.

```
for (i = 0; i < NENV; i++)
{
        if(envs[cur].env_type != ENV_TYPE_IDLE &&
            envs[cur].env_status == ENV_RUNNABLE)
            env_run(&envs[cur]);

        cur = (cur + 1) % NENV;
}
```

从 cur 开始搜索，如果找到第一个可以运行的进程 (ENV_RUNNABLE && ! ENV_TYPE_IDLE), 如果找到则从停止搜索 · 如果没有其他的进程可以运行，并且之前的进程可以运行. 则运行之前的进程. 代码如下.

```
if(curenv && curenv->env_type != ENV_TYPE_IDLE &&
        curenv->env_status == ENV_RUNNING &&
        curenv->env_cpunum == cpunum())
        env_run(curenv);
```

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?

Question 3 因为 e(指向 envs 数组中的一个元素) 是属于内核地址空间，所有进程的地址空间内核映射部分是相同的. 所以尽管发生了地址空间的切换，但是内核部分映射没有改变,virtual address (namely e) 任然映射到相同的物理地址.

Add a less trivial scheduling policy to the kernel, such as a fixed-priority scheduler that allows each environment to be assigned a priority and ensures that higher-priority environments are always chosen in preference to lower-priority environments. If you're feeling really adventurous, try implementing a Unix-style adjustable-priority scheduler or even a lottery or stride scheduler. (Look up "lottery scheduling" and "stride scheduling" in Google.) Write a test program or two that verifies that your scheduling algorithm is working correctly (i.e., the right environments get run in the right order). It may be easier to write these test programs once you have implemented fork() and IPC in parts B and C of this lab.

Challenge! 清华大学操作系统实验 ucore lab 中要求实现 stride scheduling 调度策略，因此这里不给出实现而在 ucore 的实验中给出.

Implement the system calls described above in kern/syscall.c. 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.

sys_exofork 函数原型如下

```
1 static envid_t sys_exofork(void)
```

sys_exfork 分配一个新的环境 (进程)，并返回环境 (进程) 号.

1. 调用 env_alloc 分配一个新的进程控制块 (也可以认为是进程), 当前的进程作为父进程. 新分配的为子进程
2. 将子进程的状态设为 ENV_NOT_RANNABLE，表示还可以被调度运行.
3. 复制父进程的上下文 (curenv->env_tf) 给子进程 (child->env_tf)
4. 将子进程的上下文 (child->env_tf) 中 eax 寄存器的值设置为 0. 这样子进程运行后就从 sys_exofork 中返回 0.
5. 返回子进程号

sys_env_set_status 函数原型如下

```
1 static int sys_env_set_status(envid_t envid, int status)
```

sys_env_set_status 设置一个特定的进程的状态为 ENV_RUNNABLE 或者 ENV_NOT_RUNNABLE. 并且这个特定的进程为当前进程, 或者当前进程的直接子进程.

1. 检查参数中传入的 status 是否为 ENV_RUNNABLE,ENV_NOT_RUNNABLE 之一. 如果不是则报错.
2. 调用 envid2env 获取特定的进程控制块. 并且 envid2env 的第三个参数设为 1, 用来检查该进程是否是当前进程或者是当前进程的直接子进程.
3. 修改进程的状态

sys_page_alloc 下面是 sys_page_alloc 的原型.

```
1 static int sys_page_alloc(envid_t envid, void *va, int perm)
```

分配一个新物理页映射到进程的 envid 的地址空间中 va 处, 映射的权限为 perm. 如果之前就有一页已经映射到 va. 则自动覆盖这一映射. 这一旧页被释放. perm 中 PTE_P 和 PTE_U 必须被设置. PTE_AVAIL 和 PTE_W 可以设置也可以不设置. 其余的位都不允许被设置. 并且 va 必须小于 UTOP.

1. 检查 va 是否小于 UTOP,va 是否按页对齐
2. 检查 perm 的权限位设置.
3. 调用 envid2env 获取 envid 的进程控制块
4. 调用 page_alloc 分配一新页, 并且 page_alloc 的参数为 ALLOC_ZERO 表示这一页清 0
5. 调用 page_insert 将这一新页映射到进程 envid 的地址空间中, 如果 va 之前就已经映射到一页, 那么 page_insert 会释放这个旧页. 覆盖旧的映射.

sys_page_map 函数原型如下

```
1 static int
2 sys_page_map(envid_t srcenvid, void *srcva, envid_t dstenvid, void *dstva,
        int perm)
```

将进程 screnvid 地址空间中 srcva 处的映射的页, 映射到 dstenvid 进程地址空间中 dstva 处. 映射的权限为 perm. srcva 和 dstva 必须是低于 UTOP 并且都是按页对齐. perm 的权限要求与 sys_page_alloc 一样. 如页是只读的那么 perm 不能有 PTE_W.

1. 检查 srcva,dstva 必须低于 UTOP 并且都是按页对齐
2. 检查 perm 的权限位.
3. 调用 envid2env 获取源进程的进程元数据结构和目的进程的元数据结构.envid2env 的第三个参数为 1.
4. 调用 page_lookup 获取进程源进程地址 srcva 映射到的页.
5. 调用 page_insert 将该页映射到目的进程地址空间 dstva 处.

sys_page_unmap 函数原型如下

```
1 static int sys_page_unmap(envid_t envid, void *va)
```

sys_pahe_unmap 解除进程 envid 地址空间中 va 处的映射. 如果 va 未映射任何页, 那么该函数 silently succeeds.

1. 检查 va 必须小于 UTOP, 并且按页对齐.
2. 调用 envid2env 获取 envid 对应的进程控制块.envid2env 的第三个参数为 1
3. 调用 page_remove 解除 va 处的页映射, 如果事先 va 并没有映射到任务页. 那么 page_remove 将不会产生任何影响, 并且成功返回.

1.4.2 Part B: Copy-on-Write Fork

Exercise 7. 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.

JOS 缺页处理是在用户空间实现. 每个 Env 增加一个新的成员 env_pgfault_upcall. 这个成员就是该 Env 的缺页处理程序. sys_env_set_pgfault 的原型如下.

```
1 static int sys_env_set_pgfault_upcall(envid_t envid, void *func)
```

这个系统调用将 func 设为进程 envid 的缺页处理程序. 处理流程如下.

1. 调用 envid2env 获取 envid 对应的进程控制块.envid2env 的第三个参数为 1.
2. 将 envid 对应的进程控制块 env_pgfault_upcall 成员设置为 func.

Exercise 8. 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?)

JOS 中应用程序正常运行时使用的栈是 [USTACKTOP-PGSIZE, USTACKTOP). 当发生缺页中断时系统转到用户异常栈 [UXSTACKTOP-PGSIZE, UXSTACKTOP) 上执行缺页处理程序. 修改 trap.c 中的 page_fault_handler 使得用户程序能切换到异常栈执行缺页处理程序.

1. 如果进程没有注册缺页处理程序 (即 Env 的 env_pgfault_upcall 成员为空), 那么销毁这个 env.
2. 如果进程注册有缺页处理程序. 那么设置进程的 esp, 使得 esp 转到用户异常栈

```
1     if(tf->tf_esp >= UXSTACKTOP - PGSIZE && tf->tf_esp <= UXSTACKTOP
          - 1) {
2             esp = tf->tf_esp - 4 - sizeof(struct UTrapframe);
3         } else {
```

```
4               esp = UXSTACKTOP - sizeof(struct UTrapframe);
5           }
```

当发生 pagefault 时如果 env 的 esp 已经在用户异常栈上说明这是一个嵌套的缺页 (即在缺页处理程序中发生了缺页), 否则不是嵌套的缺页. 我们需要在异常栈中压入一个 struct UTrapframe. 因此在这里为 struct UTrapframe 预留空间. 对于嵌套的缺页我们还需要压入一个 4 字节的空数据 (后面 pfentry.s 中需要 4 字节的空间来保存 eip).

3. 调用 user_mem_assert 检查进程是否拥有对栈的读写权限.
4. 然后在 esp 处构造 struct UTrapframe 结构.

```
1       utf->utf_esp = tf->tf_esp;
2           utf->utf_eflags = tf->tf_eflags;
3           utf->utf_eip = tf->tf_eip;
4           utf->utf_regs = tf->tf_regs;
5           utf->utf_err = tf->tf_err;
6           utf->utf_fault_va = fault_va;
```

异常栈中保存有发生 pagefualt 时进程的上下文状态信息.

5. 修改进程的栈地址 esp 和指令指针 eip，使得进程恢复执行时处在异常栈上开始执行缺页处理程序

```
1       tf->tf_esp = esp;
2       tf->tf_eip = (uintptr_t)(curenv->env_pgfault_upcall);
```

6. 调用 env_run 恢复当前进程的执行. 进程就会恢复到用户态开始在异常栈上执行缺页处理程序.

我们分析下 struct UTrapframe 和 struct Trapframe 的不同. struct UTrapframe 是用户异常栈中的结构. 而 struct Trapframe 是内核栈中的结构. 可以看出他们有下面几个不同

1. UTrapframe 是特别设计给用户自定义的中断错误处理程序的（不一定是页错误中断处理程序）的，用以保存错误发生之前的环境信息，所以 UTrapframe 有了一个特殊的成员 fault_va，表示访问出错的指令涉及的具体地址. Trapframe 没有这个结构, 在内核中可以读取 cr2 寄存器的值获取这个地址. 在用户态则无法读取这个结构.
2. UTrapframe 和 Trapframe 相比少了 es, ds, ss 等段寄存器信息。因为根据刚才提到栈切换的不同，无论是两种情况的哪种，都是从用户态 $\longrightarrow$ 内核态 $\longrightarrow$ 用户态，因为两个用户态程序实际上是同一个用户进程，所以我们从后面的中断错误处理程序切换到前面的触发程序就不会涉及到段的切换，自然也不需要保存它们.

Exercise 9. 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.

当缺页处理程序处理完成后我们需要恢复进程的执行或这恢复到上一层缺页处理程序中执行.

```
1  movl 48(%esp), %eax
2    subl $4, %eax
3    movl 40(%esp), %ebx;
4    movl %ebx, (%eax);
5    movl %eax, 48(%esp)
```

struct Utrapframe 总共 52 个字节.48(%esp) 就是地址 esp+48 中的数据, 即 Utrapframe 的 esp, 就是发生缺页时的 esp. 该值赋值给 eax, 然后减 4 是为了在栈 (发生缺页时的栈,trap-time stack) 上预留 4 字节的空间. 然后 40(%esp) 是发生缺页时 eip. 将 eip 的值放入到刚刚预留出的 4 字节中，这里已经将 eip 保存, 所以后面恢复寄存器时会忽略 eip. 因为栈 (发生缺页时的栈 trap-time stack) 已经增大了 4 字节. 所以这个更新要反应到异常栈中. 即 eax 的值 (trap-time stack pointer) 移动到 48(%esp). 这样当把异常栈中的用户状态恢复时可以恢复正确的 trap-time stack 指针. 接下来恢复通用寄存器.

```
1  addl $8, %esp
2  popal;
```

给 esp 加 8 忽略 fault_va 和 tf_err. popal 恢复所有通用寄存器 (eax, ebx, ecx, edx,esp,ebp,esi,edi). 然后恢复 eflags 寄存器

```
1  addl $4, %esp
2  popf
```

给 esp 加 4 忽略 eip，因为 eip 已经在前面已经保存到了 trap-time stack 中.popf 指令从栈中恢复 eflags. 最后恢复栈指针并返回执行

```
1  popl %esp;
2  ret;
```

popl 恢复 esp 的值. 这个 esp 是 trap-time 栈指针. 这样栈就发生了切换. 最后调用 ret 指令.ret 指令会从栈顶中弹出一个值作为 eip. 因为 ret 前已经将栈切换到 trap-time stack. 我们之前将 eip 的值已经压入到了 trap-time stack 中. 所以 ret 从栈顶中取到的值就是正确的 eip. 现在即恢复了栈又恢复了 eip. 进程可以继续运行.

Exercise 10. Finish set_pgfault_handler() in lib/pgfault.c.

该函数的原型如下.

```
void set_pgfault_handler(void (*handler)(struct UTrapframe *utf))
```

1. 调用 sys_page_alloc 分配一页并映射到 UXSTACKTOP-PGSIZE 作为用户异常栈
2. 调用 sys_env_set_pgfault_upcall 将缺页处理程序设为 _pgfault_upcall(lib/pfentry.s 中定义).
3. 将 _pgfault_handler(lib/pfentry.s 中定义) 设为 handler.

Exercise 11. 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.
1008: I am " 1009: I am '0' 2008: I am '00' 2009: I am '000' 100a: I am '1' 3008: I am '11' 3009: I am '10' 200a: I am '110' 4008: I am '100' 100b: I am '01' 5008: I am '011' 4009: I am '010' 100c: I am '001' 100d: I am '111' 100e: I am '101'

pgfault 是缺页处理程序. 当发生 fault 的页是 cow. 则新分配一个私有的页. 父进程会调用 set_pgfault_handle 将 pgfault 设置为缺页的 handler.pgfault 的原型如下

```
static void pgfault(struct UTrapframe *utf)
```

pgfualt 的处理流程如下:

```
if(!(err & FEC_WR) || !(vpt[(uintptr_t)addr >> PGSHIFT] & PTE_COW))
        panic("pgfault:␣not␣write␣or␣not␣cow");
```

addr 为发生 fualt 的地址. 如果不是写 fualt 并且不是 COW 页那么 panic. 如果通过这个检查

1. 调用 sys_page_alloc 分配一页并映射到地址 PFTEMP.
2. 调用 memmove 将 addr 起始一页大小的内容复制到地址 PFTEMP.
3. 调用 sys_page_map 使得 addr 映射到 PFTEMP 映射的页处.
4. 调用 sys_page_unmap 解除 PFTEMP 处的映射. 现在只有 addr 映射到这一页了.

函数 duppage 负责将页号 pn 对应的页映射到进程 envid 地址 pn * PGSIZE 处. 函数原型如下

```
static int duppage(envid_t envid, unsigned pn)
```

该函数处理流程如下:

1. 如果页可写或者是 COW. 那么建立的映射是 COW. 所以调用 sys_page_map 使得当前进程和进程 envid 的地址 pn*PGZIE 都映射到同一页. 移除 envid 进程的写权限. 标记上 PTE_COW. 再次调用 sys_page_map 使得当前进程也移除写权限, 并设置 PTE_COW 位. 这样两个进程就会读共享页 pn.
2. 如果不是写或者 COW. 调用 sys_page_map 建立当前进程和 envid 进程读共享页 pn, 但不会设置 PTE_COW 位.

最后是 fork 的实现,fork 负责创建一个子进程并返回进程号. fork 需要使用 vpt 和 vpd. 我们首先看下这两个变量的定义.lib/entry.s 中

```
.globl vpt
.set vpt, UVPT
.globl vpd
.set vpd, (UVPT+(UVPT>>12)*4)
```

在看 kern/pmap.c 中

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

假设需要查询的虚地址是 va.

1. 其对应的页目录表项为:vpd[va » PDXSHIFT];
2. 其对应的页表项为: vpt[va » PTXSHIFT];

fork 的原型如下

```
envid_t fork(void)
```

fork 的实现如下:

1. 调用 set_pgfault_handler(pgfault) 设置 pgfault 为 page fualt handler. 用来处理 COW
2. 调用 sys_exofork 分配一个进程控制块 env(创建一个子进程). 在子进程中需要修正 thisenv. thisenv 在 lib/libmain.c 中定义.
3. 复制父进程地址空间 [0, UTOP-PGSZIE) 到子进程. 具体的每一页的复制由 duppage 负责. UTOP-PGSIZE 这一页不复制. 因为这是用户异常栈的起始地址. 子进程有自己的用户异常栈并且不是 COW.
4. 调用 sys_page_alloc 分配一页作为异常栈映射到子进程 UXSTACKTOP-PGSIZE(UTOP-PGSIZE) 处.
5. 调用 sys_env_set_pgfault_upcall 将父进程的 env_pgfault_upcall 成员赋值给子进程.
6. 调用 sys_env_set_status 将子进程的状态设置为 ENV_RUNNABLE. 这样子进程才能从 sys_exofork 中返回然后才能修正自己的 thisenv.

1.4.3 Part C: Preemptive Multitasking and Inter-Process communication (IPC)

Exercise 12. 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.

trap.c/trap_inic 函数中添加如下代码

```
entry = IRQ_OFFSET + IRQ_TIMER;
   SETGATE(idt[entry], 0, GD_KT, vectors[entry], 0);
   entry = IRQ_OFFSET + IRQ_KBD;
   SETGATE(idt[entry], 0, GD_KT, vectors[entry], 0);
   entry = IRQ_OFFSET + IRQ_SERIAL;
   SETGATE(idt[entry], 0, GD_KT, vectors[entry], 0);
   entry = IRQ_OFFSET + IRQ_SPURIOUS;
   SETGATE(idt[entry], 0, GD_KT, vectors[entry], 0);
   entry = IRQ_OFFSET + IRQ_IDE;
   SETGATE(idt[entry], 0, GD_KT, vectors[entry], 0);
   entry = IRQ_OFFSET + IRQ_ERROR;
   SETGATE(idt[entry], 0, GD_KT, vectors[entry], 0);
```

外部中断指 IRQ. 总共 16 个 IRQ. 这 16 个 IRQ 并不是固定的映射到一个中断号 (即 IDT 的索引号). JOS 中将 IRQ 0-15 映射到 IDT 的 IRQ_OFFSET-IRQ_OFFSET+15. IRQ_OFFSET 定义为 32. 所以 IDT 的 32-47 映射到 IRQ 0-15. kern/env.c/env_alloc 函数中添加如下代码:

```
e->env_tf.tf_eflags |= FL_IF;
```

当 eflags 寄存 FL_IF 位置为 1 时, 可以接收到外部中断.

Exercise 13. 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.

在 kern/trap.c/trap_dispatch 中添加如下代码.

```
if(tf->tf_trapno == IRQ_OFFSET + IRQ_TIMER) {
        lapic_eoi();
        sched_yield();
}
```

调用 lapic_eoi 确认中断, 然后进入调度器进行调度.

> Exercise 14. 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. You might find it interesting to read user/primes.c to see all the forking and IPC going on behind the scenes.

为了实现 IPC, struct Env 结构中添加了结构成员:

env_ipc_recving 当进程使用 sys_ipc_recv() 等待信息时，会将这个成员置为 1，然后阻塞等待；当一个进程像它发消息解除阻塞后，发送进程将此成员修改为 0.

env_ipc_dstva 如果进程要接受消息，并且是传送页，则该地址 UTOP.

env_ipc_value 若等待消息的进程接受到了消息，发送方将接受方此成员置为消息值.

env_ipc_from 发送方负责设置该成员为自己的 envid 号.

env_ipc_perm 如果进程要接受消息，并且传送页，那么发送方发送页以后将传送的页权限传给这个成员.
sys_ipc_recv 的原型如下. 该函数挂起直到接收到一个数值. 如果 dstva 小于 UTOP 还表示它接收一个页.

```
1 static int sys_ipc_recv(void *dstva)
```

1. 检查 dstva 是否大于 UTOP 和页对齐
2. 将当前进程控制块的 env_ipc_dstva 成员设为 dstva,env_ipc_recving 成员设为 1, 表示希望接收.
3. 将当前进程的状态设为 ENV_NOT_RUNNBALE. 让后调用 sched_yield 使得当前进程挂起.

sys_ipc_try_send 发送一个数值, 或者还发送一个页给目的进程如果 srcva 小于 UTOP. 该函数的原型如下

```
1 static int sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva,
        unsigned perm)
```

sys_ipc_try_send 的基本流程如下

1. 检查目的进程的 env_ipc_recving 字段是否为 1, 如果为 0 表示目的进程并没有在请求发送数据. 返回 -E_IPC_NOT_RECV.
2. 检查 srcva 是否大于 0 或者页对齐. 否则返回 -E_INVAL
3. srcva 虽然小于 UTOP 且页对齐. 但并不代表 srcva 是个有效的值. 例如当不想发送一个页时我们可能会把 srcva 设置成 NULL(也就是 0). 但是地址 0 在 JOS 中是有效的. 所以我们需要另外一个地址来充当 NULL, 表明 srcva 无效. 我们选择 USTACKTOP 作为这个地址.
4. 如果 srcva 不等于 USTACKTOP(即 srcva 是有效的). 获取 srcva 对应的 pte, 检查 pte 的权限设置是否和 perm 一致, 防止页是只读但传入的 perm 要求写映射.
   (a) 目的进程的 env_ipc_dstva 如果不等于 USTACKTOP(表示有效). 则调用 sys_page_map 将发送进程地址 srcva 映射的页映射到目的进程地址 env_ipc_dstva 处. 并将目的进程 env_ipc_perm 设置为参数 perm.
   (b) 如果目的进程的 env_ipc_dstva 如果等于 USTACKTOP(表示无效, 目的进程不想接受一页). 则把目的进程 env_ipc_perm 设置为 0.
5. 如果 srcva 等于 USTACKTOP(表示无效, 目的进程不发送页). 则把目的进程 env_ipc_perm 设置为 0
6. 最后将目的进程的 env_ipc_from 设为当前进程 (即发送进程) 的 env_id. env_ipc_vale 设为参数 value. 目的进程的上下文 eax 寄存器置为 0. 这样目的进程从 sys_ipc_recv 调用中能返回 0. 将目的进程的 env_ipc_recving 设为 0. 最后将目的进程的状态设置为 ENV_RUNNABLE. 返回.

1.5 LAB5: File System

这是关于 JOS 文件系统的实验,jos 中没有实现 inode, 所以在磁盘上没有对应的类似 xv6 的 inode 区域, 仅仅有 superblock. free block bitmap 和 data block. 由于没有实现 inode, 所以使用一个 struct File 来实现 inode 类似的功能. 原有的 unix-like 系统中目录项保存一个 < 文件名,inode 指针 >, 但是 jos 没有实现 inode, 所以 jos 目录项仅仅是一个 struct File 结构. 在 jos 中文件系统在用户空间实现, 由一个单独的进程来实现文件系统. 文件系统需要访问磁盘,jos 没有采用中断的方式去访问磁盘. 而是更改 eflags 的 IOPL, 使得文件系统进程有权限取访问磁盘.

1.5.1 JOS File system Introduction

JOS 单独维护了一个文件系统进程作为 server. 其他的进程通过 RPC 请求文件系统进程 server 的服务.

磁盘缓冲区 这里描述了磁盘块缓冲的具体机制,JOS 将整个磁盘映射到进程的虚地址空间. 最大能支持 3GB 的磁盘.

1. 用文件系统服务进程的虚拟地址空间 (0x10000000-0xD0000000) 对应到磁盘的地址空间上（3GB）
2. 初始文件系统服务进程里什么页面都没映射, 如果要访问一个磁盘的地址空间, 则发生页错误
3. 在页错误处理程序中, 在内存中申请一个块的空间映射到相应的文件系统虚拟地址上, 然后去实际的物理磁盘上读取这个区域的东西到这个内存区域上, 然后恢复文件系统服务进程

因为 32 机器上进程地址空间最大也就 4GB. 所以支持的磁盘大小也就 3GB. 如果在 64 位机器上能支持的磁盘空间将会更大.

文件系统结构 JOS 文件系统块大小为 4KB. 磁盘扇区大小为 512KB

1. 磁盘上的第一个块, 即块 0, 是启动块. 其中包含 bootloader 和分区表
2. 磁盘上第二个块, 即块 1, 是超级块. 超级快包含的信息有: 磁盘的总块数, 文件系统根目录等等.
3. 超级块后的几个块是 free block bitmap. 用一位表示一个块是否空闲.1 表示空闲,0 表示已分配. 当需要分配空闲块时需要搜索这个位图寻找空闲块. 虽然这比搜索空闲块链表费时间. 但是文件系统费时主要在磁盘 IO 上. 所以搜索空闲块的时间不是主要的.
4. free block bitmap 之后就是数据块.

JOS 的每一个文件都有一个 struct File 结构描述. 一般 UNIX-like 系统文件都是由一个 inode 描述, 但是 jos 做了简化. 我们看看 struct File 的结构.

```
1 struct File {
2     char f_name[MAXNAMELEN];    // filename
```

```
    off_t f_size;           // file size in bytes
    uint32_t f_type;        // file type

    // Block pointers.
    // A block is allocated iff its value is != 0.
    uint32_t f_direct[NDIRECT]; // direct blocks
    uint32_t f_indirect;        // indirect block

    // Pad out to 256 bytes; must do arithmetic in case we're compiling
    // fsformat on a 64-bit machine.
    uint8_t f_pad[256 - MAXNAMELEN - 8 - 4*NDIRECT - 4];
} __attribute__((packed));  // required only on some 64-bit machines
```

其中包含的信息有文件名, 文件大小, 类型. 直接块和间接块. struct File 占据 256 个字节. 因为 JOS 没有实现 inode 结构所以目录结构也和其他 UNIX-like 系统不同, 一般 UNIX-like 系统目录结构包含文件名和指向文件名对应的 inode 的指针 (inode 号).JOS 中目录只是包含 struct File 结构.

磁盘映像的生成　fs/Makefrag 中包含磁盘镜像生成的详细过程.

```
OBJDIRS += fs

FSOFILES :=         $(OBJDIR)/fs/ide.o \
            $(OBJDIR)/fs/bc.o \
            $(OBJDIR)/fs/fs.o \
            $(OBJDIR)/fs/serv.o \
            $(OBJDIR)/fs/test.o \

USERAPPS :=         $(OBJDIR)/user/init

FSIMGTXTFILES :=    fs/newmotd \
            fs/motd



FSIMGFILES := $(FSIMGTXTFILES) $(USERAPPS)

$(OBJDIR)/fs/%.o: fs/%.c fs/fs.h inc/lib.h
    @echo + cc[USER] $<
    @mkdir -p $(@D)
    $(V)$(CC) -nostdinc $(USER_CFLAGS) -c -o $@ $<

$(OBJDIR)/fs/fs: $(FSOFILES) $(OBJDIR)/lib/entry.o $(OBJDIR)/lib/libjos.a
        user/user.ld
    @echo + ld $@
    $(V)mkdir -p $(@D)
    $(V)$(LD) -o $@ $(ULDFLAGS) $(LDFLAGS) -nostdlib \
        $(OBJDIR)/lib/entry.o $(FSOFILES) \
        -L$(OBJDIR)/lib -ljos $(GCC_LIB)
    $(V)$(OBJDUMP) -S $@ >$@.asm

# How to build the file system image
```

```
32 $(OBJDIR)/fs/fsformat: fs/fsformat.c
33     @echo + mk $(OBJDIR)/fs/fsformat
34     $(V)mkdir -p $(@D)
35     $(V)$(NCC) $(NATIVE_CFLAGS) -o $(OBJDIR)/fs/fsformat fs/fsformat.c
36
37 $(OBJDIR)/fs/clean-fs.img: $(OBJDIR)/fs/fsformat $(FSIMGFILES)
38     @echo + mk $(OBJDIR)/fs/clean-fs.img
39     $(V)mkdir -p $(@D)
40     $(V)$(OBJDIR)/fs/fsformat $(OBJDIR)/fs/clean-fs.img 1024 $(FSIMGFILES)
41
42 $(OBJDIR)/fs/fs.img: $(OBJDIR)/fs/clean-fs.img
43     @echo + cp $(OBJDIR)/fs/clean-fs.img $@
44     $(V)cp $(OBJDIR)/fs/clean-fs.img $@
45
46 all: $(OBJDIR)/fs/fs.img
47
48 #all: $(addsuffix .sym, $(USERAPPS))
49
50 #all: $(addsuffix .asm, $(USERAPPS))
```

从 32 行开始:

1. 第 32 行: 将 fs/fsformat.c 编译成可执行文件
2. 第 38 行: 使用 fs/fsformat 可执行文件产生镜像文件 fs/clean-fs.img
3. 第 43 行: 将 fs/clean-fs.img 复制成眞正的磁盘镜像文件 fs/fs.img

我们主要关注第二步是如何产生 fs/clean-fs.img. 可以看到它使用的命令为:

```
1 $(V)$(OBJDIR)/fs/fsformat $(OBJDIR)/fs/clean-fs.img 1024 $(FSIMGFILES)
```

fs/foformat 接受了一个参数 1024, 和 FSIMGFILES.1024 表示所创建磁盘映像的块数, FSIMGFILES 包含创建磁盘映像时需要写入这个磁盘映射的文件. fs/fsformat 由 fs/fsformat.c 编译生成. 这是一个小的磁盘映像创建工具. 我们看看它的主要功能

```
1  int main(int argc, char **argv)
2  {
3      int i;
4      char *s;
5      struct Dir root;
6
7      assert(BLKSIZE % sizeof(struct File) == 0);
8
9      if (argc < 3)
10         usage();
11
12     nblocks = strtol(argv[2], &s, 0);
13     if (*s || s == argv[2] || nblocks < 2 || nblocks > 1024)
14         usage();
15
16     opendisk(argv[1]);
17
18     startdir(&super->s_root, &root);
```

```
19      for (i = 3; i < argc; i++)
20          writefile(&root, argv[i]);
21      finishdir(&root);
22
23      finishdisk();
24      return 0;
25  }
```

nblocks 是总的块数.

1. 使用 opendisk 创建一个磁盘文件, 超级块在这里被初始化. 实际的操作是创建一个 nblocks 大小的文件, 将此文件映射到系统地址 diskmap 处. 然后该文件的第 0 块保留作为启动块. 第一块作为超级块. 初始化 super 结构. 接下俩将 free block bitmap 初始化. 并且全部置为 1. 表示全部块都是空闲的. 因为这个磁盘映像文件已经映射到内存, 所以上面的操作与修改都是在内存中进行. 这些修改之后会写回到磁盘映像文件中. 这样磁盘映像文件就包含了 super,free block bitmap 等.
2. startdir 开始创建根目录, 在这个函数中只是为 struct Dir 结构分配了 128 个目录项, 就是 128 个 struct File 结构. 初始磁盘映像需要的文件全部放入到这个目录中.
3. 使用 writefile 将目标文件写入磁盘映像. 该函数逐个读取 FSIMGFILES 中的文件的 stat 信息. 然后将文件名, 类型, 保存到 struct Dir 的目中. 让后将文件的内容读入到磁盘映像所映射的内存区域. 调用 finishdir 初始化目录中文件的数据块, 具体就是设置直接块和间接块的数值. 指向对应的块即可.
4. 使用 finishdisk 将 r 将根目录写入磁盘映像. 根目录还保存在 struct Dir 结构中. 将这些目录结构作为根目录文件的内容.
5. finishdisk 将块位图设置为正确的值, 将所做的修改写回到磁盘映像文件. 完成磁盘映像的创建

   不在做过多的介绍, 详细过程见代码.

文件操作 在进行下面的工作之前, 先了解一下 fs/fs.c 中提供的各个函数的功能:

```
1   /* public*/
2   void    fs_init(void);
3   int file_get_block(struct File *f, uint32_t file_blockno, char **pblk);
4   int file_create(const char *path, struct File **f);
5   int file_open(const char *path, struct File **f);
6   ssize_t file_read(struct File *f, void *buf, size_t count, off_t offset);
7   int file_write(struct File *f, const void *buf, size_t count, off_t offset)
        ;
8   int file_set_size(struct File *f, off_t newsize);
9   void    file_flush(struct File *f);
10  int file_remove(const char *path);
11  void    fs_sync(void);
12
```

```
13 /* int  map_block(uint32_t); */
14 bool    block_is_free(uint32_t blockno);
15 int alloc_block(void);
16
17 /* static */
18 static int file_block_walk(struct File *f, uint32_t filebno, uint32_t **
       ppdiskbno, bool alloc)
19 static int dir_lookup(struct File *dir, const char *name, struct File **
       file)
20 static int dir_alloc_file(struct File *dir, struct File **file)
21 static const char* skip_slash(const char *p)
22 static int walk_path(const char *path, struct File **pdir, struct File **pf
       , char *lastelem)
23 static int file_free_block(struct File *f, uint32_t filebno)
24 static void file_truncate_blocks(struct File *f, off_t newsize)
```

这里前面一部分 public 的函数应该无论是从函数名还是参数上都是比较明确的, 就不再解释了, 主要需要解释的是下面一部分静态函数:

- file_block_walk(*f, filebno, ppdiskbno, alloc) 寻找一个文件结构 f 中的第 filebno 个块指向的硬盘块编号放入 ppdiskbno, 即如果 filebno 小于 NDIRECT, 则返回属于 f direct[NDIRECT] 中的相应链接, 否则返回 f indirect 中查找的块. 如果 alloc 为真且相应硬盘块不存在, 则分配一个. 当我们要将一个修改后的文件 flush 回硬盘, 就需要使用这个函数找一个文件中链接的所有磁盘块, 将他们都 flush block
- dir_lookup(*dir, *name, **file) 在目录 dir 中查找名为 name 的文件. 使得 file 指向这个文件.
- dir_alloc_file(*dir, **file) 在目录 dir 中新分配一个 slot.file 指向这个 slot, 这是用于添加文件的操作.
- skip_slash(*p) 用于路径中的字符串处理, 跳过斜杠
- walk_path(*path, **pdir, **pf, *lastlem) *path 为从根目录开始描述的文件名, 如果成功找到了文件, 则把相应的文件 File 结构赋值给 *pf, 其所在目录的 File 结构赋值给 **pdir,lastlem 为失败时最后剩下的文件名字
- file_free_block(*f, filebno) 释放一个文件中的第 filebno 个磁盘块. 此函数在 file truncate blocks 中被调用
- file_truncate_blocks(*f, newsize) 将文件设置为缩小后的新大小, 清空那些被释放的物理块.

JOS C/S 文件系统访问 我们首先关注服务器端程序的架构. 首先有几个特别重要的结构需要了解, 看到 inc/fs.h.

```
1 union Fsipc {
2     struct Fsreq_open {
3         char req_path[MAXPATHLEN];
4         int req_omode;
5     } open;
6     struct Fsreq_set_size {
```

```
7           int req_fileid;
8           off_t req_size;
9       } set_size;
10      struct Fsreq_read {
11          int req_fileid;
12          size_t req_n;
13      } read;
14      struct Fsret_read {
15          char ret_buf[PGSIZE];
16      } readRet;
17      struct Fsreq_write {
18          int req_fileid;
19          size_t req_n;
20          char req_buf[PGSIZE - (sizeof(int) + sizeof(size_t))];
21      } write;
22      struct Fsreq_stat {
23          int req_fileid;
24      } stat;
25      struct Fsret_stat {
26          char ret_name[MAXNAMELEN];
27          off_t ret_size;
28          int ret_isdir;
29      } statRet;
30      struct Fsreq_flush {
31          int req_fileid;
32      } flush;
33      struct Fsreq_remove {
34          char req_path[MAXPATHLEN];
35      } remove;
36
37      // Ensure Fsipc is one page
38      char _pad[PGSIZE];
39  };
```

这里需要了解 union Fsipc, 文件系统中客户端和服务器端通过 IPC 进行通信, 那么通信的数据格式就是 union Fsipc, 它里面的每一个成员对应一种文件系统的操作请求. 每次客户端发来请求, 都会将参数放入一个 union Fsipc 映射到一个物理页传递给服务器端, 同时有时候服务器还会将处理以后的结果放入 Fsipc 内, 传递给用户程序. 这里就涉及到文件服务器程序的地址空间的布局:

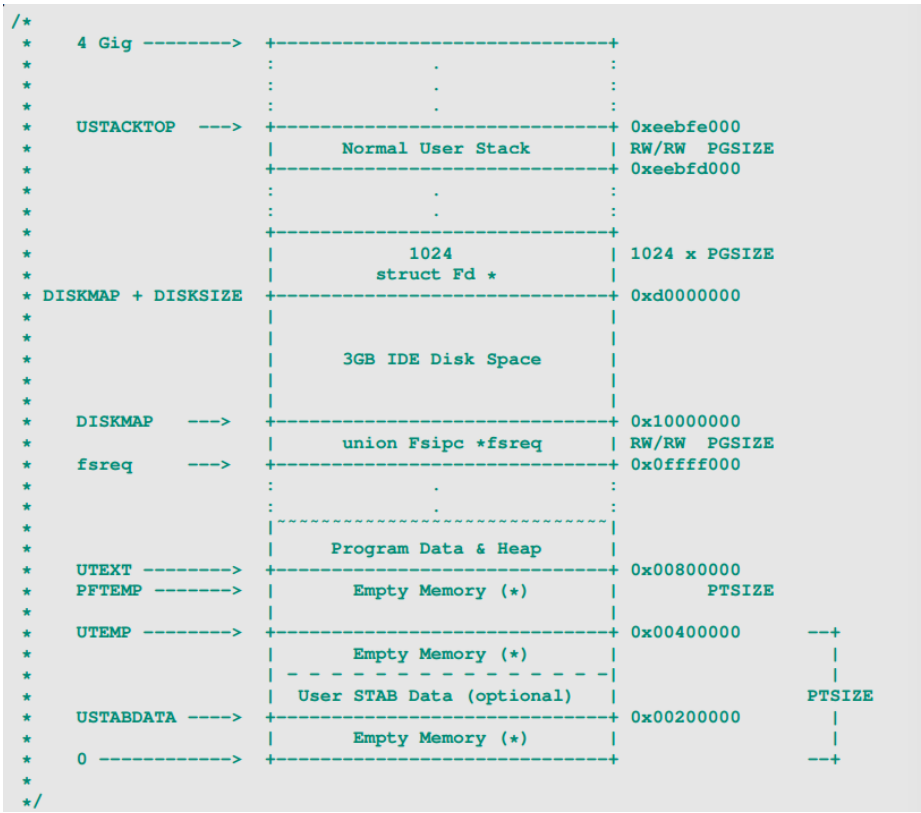


图 20: 文件系统进程地址空间布局

文件系统进程和普通进程地址空间布局有些不同. 其中

- 地址空间 DISKMAP – DISKMAP + DISKSIZE, 这部分空间映射了 3GB 的对应 IDE 磁盘空间.
- 地址空间 0x0ffff00 – DISKMAP, 一次 IPC 请求的 union Fsipc 放置的地址空间.
- 这片空间有 1024 个物理页, 每个物理页对应一个 struct Fd. 即系统最多只能打开 1024 个文件. 一个文件被不同的两进程各打开一次, 则占两个 FD.

打开 fs/serv.c, 可以看到它定义的唯一一个全局变量 opentab

```
1  struct OpenFile {
2      uint32_t o_fileid;  // file id
3      struct File *o_file;    // mapped descriptor for open file
4      int o_mode;     // open mode
5      struct Fd *o_fd;    // Fd page
6  };
7  // Max number of open files in the file system at once
8  #define MAXOPEN     1024
9  #define FILEVA      0xD0000000
10 // initialize to force into data section
```

```
struct OpenFile opentab[MAXOPEN] = {
    { 0, 0, 1, 0 }
};
// Virtual address at which to receive page mappings containing client
       requests.
union Fsipc *fsreq = (union Fsipc *)0x0ffff000;
```

OpenFile 结构是服务器程序维护的一个映射, 它将一个真实文件 struct File 和用户客户端打开的文件描述符 struct Fd 对应到一起（具体 struct Fd 代表的意义见下段）. 每个被打开的文件对应的 struct Fd 都被映射到 FILEVA 上往上的一个物理页, 服务器程序和打开这个文件的客户程序共享这个物理页. 客户端程序和文件系统服务器通信时使用 o fileid 来指定要操作的文件. 文件系统默认最大同时可以打开的文件个数为 1024, 所以有 1024 个 struct Openfile. 所以对应着在服务器地址空间 0xd0000000 往上留出了 1024 个物理页用于映射这些对应的 struct Fd. 它是一个抽象层, 因为 JOS 和 Linux 一样, 所有的 IO 都是文件, 所以用户看到的都是 Fd 代表的文件, 但是 Fd 会记录其对应的具体对象, 比如真实文件、Socket 和管道等等, 因为我们现在只有文件, 所以看到 union 里只有一个 FdFile, 后面如果有其他类型的对象加入, 那么 union 里会有其他的内容.

'inc/fd.h'

```
struct FdFile {
    int id;
};

struct Fd {
    int fd_dev_id;
    off_t fd_offset;
    int fd_omode;
    union {
        // File server files
        struct FdFile fd_file;
    };
};
```

至此我们已经搞明白了服务器段程序的内存结构, 然后我们来看看它会作一些什么工作:

'fs/serv.c'

```
void serve(void)
{
    uint32_t req, whom;
    int perm, r;
    void *pg;

    while (1) {
        perm = 0;
        req = ipc_recv((int32_t *) &whom, fsreq, &perm);
        if (debug)
```

```
            cprintf("fs req %d from %08x [page %08x: %s]\n",
                req, whom, vpt[PGNUM(fsreq)], fsreq);

        // All requests must contain an argument page
        if (!(perm & PTE_P)) {
            cprintf("Invalid request from %08x: no argument page\n",
                whom);
            continue; // just leave it hanging...
        }

        pg = NULL;
        if (req == FSREQ_OPEN) {
            r = serve_open(whom, (struct Fsreq_open*)fsreq, &pg, &perm);
        } else if (req < NHANDLERS && handlers[req]) {
            r = handlers[req](whom, fsreq);
        } else {
            cprintf("Invalid request code %d from %08x\n", whom, req);
            r = -E_INVAL;
        }
        ipc_send(whom, r, pg, perm);
        sys_page_unmap(0, fsreq);
    }
}
```

服务器主循环会使用轮询的方式接受客户端程序的文件请求, 每次

1. 从 IPC 接受一个请求类型 req 以及数据页 fsreq
2. 然后根据 req 来执行相应的服务程序
3. 将相应服务程序的执行结果（如果产生了数据页则有 pg）通过 IPC 发送回调用进程
4. 将映射好的物理页 fsreq 取消映射

'fs/serv.c'

```
typedef int (*fshandler)(envid_t envid, union Fsipc *req);

fshandler handlers[] = {
    // Open is handled specially because it passes pages
    /* [FSREQ_OPEN] =   (fshandler)serve_open, */
    [FSREQ_SET_SIZE] =  (fshandler)serve_set_size,
    [FSREQ_READ] =      serve_read,
    [FSREQ_WRITE] =     (fshandler)serve_write,
    [FSREQ_STAT] =      serve_stat,
    [FSREQ_FLUSH] =     (fshandler)serve_flush,
    [FSREQ_REMOVE] =    (fshandler)serve_remove,
    [FSREQ_SYNC] =      serve_sync
};
#define NHANDLERS (sizeof(handlers)/sizeof(handlers[0]))
```

服务程序被定义在了 handler 数组里, 通过请求号进行调用, 具体定义在

'inc/fs.h'

Exercise 1. i386_init identifies the file system environment by passing the type ENV_TYPE_FS to your environment creation function, env_create. Modify env_create in env.c, so that it gives the file system environment I/O privilege, but never gives that privilege to any other environment. Make sure you can start the file environment without causing a General Protection fault. You should pass the "fs i/o" test in make grade.

```
enum {
    FSREQ_OPEN = 1,
    FSREQ_SET_SIZE,
    // Read returns a Fsret_read on the request page
    FSREQ_READ,
    FSREQ_WRITE,
    // Stat returns a Fsret_stat on the request page
    FSREQ_STAT,
    FSREQ_FLUSH,
    FSREQ_REMOVE,
    FSREQ_SYNC
};
```

1.5.2 The File System

这个很简单, 修改 env_create. 如果传入的 type 是 ENV_TYPE_FS. 表示这是文件系统进程, 修改该进程 trapframe 的 eflags. 使得文件系统进程有权限访问磁盘 IO. 只要当前进程 CPL 小于等于 eflags 的 IOPL, 就可以直接访问 IO.

```
  if (type == ENV_TYPE_FS) {
        e->env_tf.tf_eflags |= FL_IOPL_3;
    }
```

Exercise 2. Implement the bc_pgfault and flush_block functions in fs/bc.c. bc_pgfault is a page fault handler, just like the one your wrote in the previous lab for copy-on-write fork, except that its job is to load pages in from the disk in response to a page fault. When writing this, keep in mind that (1) addr may not be aligned to a block boundary and (2) ide_read operates in sectors, not blocks.

The flush_block function should write a block out to disk if necessary. flush_block shouldn't do anything if the block isn't even in the block cache (that is, the page isn't mapped) or if it's not dirty. We will use the VM hardware to keep track of whether a disk block has been modified since it was last read from or written to disk. To see whether a block needs writing, we can just look to see if the PTE_D "dirty" bit is set in the vpt entry. (The PTE_D bit is set by the processor in response to a write to that page; see 5.2.4.3 in chapter 5 of the 386 reference manual.) After writing the block to disk, flush_block should clear the PTE_D bit using sys_page_map.

Use make grade to test your code. Your code should pass "check_bc", "check_super", and "check_bitmap".

```
bc_pgfault(struct UTrapframe *utf)
{
    void *addr = (void *) utf->utf_fault_va;
    uint32_t blockno = ((uint32_t)addr - DISKMAP) / BLKSIZE;
    int r;

    // Check that the fault was within the block cache region
    if (addr < (void*)DISKMAP || addr >= (void*)(DISKMAP + DISKSIZE))
        panic("page fault in FS: eip %08x, va %08x, err %04x",
              utf->utf_eip, addr, utf->utf_err);

    // Sanity check the block number.
    if (super && blockno >= super->s_nblocks)
        panic("reading non-existent block %08x\n", blockno);

    // Allocate a page in the disk map region, read the contents
    // of the block from the disk into that page, and mark the
    // page not-dirty (since reading the data from disk will mark
    // the page dirty).
    //
    // LAB 5: Your code here
    //int perm;
    if ((r = sys_page_alloc(0, ROUNDDOWN(addr, PGSIZE), PTE_U|PTE_W|PTE_P))
         < 0)
        panic("sys_page_alloc: %e", r);
    ide_read(blockno * BLKSECTS, ROUNDDOWN(addr, PGSIZE), BLKSECTS);
    //perm = vpt[PGNUM(addr)] & (PGSIZE - 1);
    //perm &= ~PTE_D;
```

```
28      if ((r = sys_page_map(0, ROUNDDOWN(addr, PGSIZE), 0, ROUNDDOWN(addr,
            PGSIZE), PTE_SYSCALL)) < 0)
29          panic("sys_page_map:␣%e", r);
30
31  //  panic("bc_pgfault not implemented");
32
33      // Check that the block we read was allocated. (exercise for
34      // the reader: why do we do this *after* reading the block
35      // in?)
36      if (bitmap && block_is_free(blockno))
37          panic("reading␣free␣block␣%08x\n", blockno);
38  }
```

addr 并不不是按页对齐. 首先分配一页映射到地址 ROUNDDOWN(addr, PGSIZE) 处. 然后调用 ide_read 从磁盘读取数据.ide_read 是以 sector 为单位读取数据的, 表示从扇区号 sectno 开始读取 nsecs 个扇区的内容到地址 dst. 然后再次调用 sys_page_map 清除 pte 上的 dirty 标志.

```
1   void
2   flush_block(void *addr)
3   {
4       uint32_t blockno = ((uint32_t)addr - DISKMAP) / BLKSIZE;
5
6       if (addr < (void*)DISKMAP || addr >= (void*)(DISKMAP + DISKSIZE))
7           panic("flush_block␣of␣bad␣va␣%08x", addr);
8       // LAB 5: Your code here.
9       int r;
10      if (va_is_mapped(addr) && va_is_dirty(addr)) {
11          if ((r = ide_write(blockno * BLKSECTS, ROUNDDOWN(addr, PGSIZE),
            BLKSECTS)) < 0)
12              panic("ide_write:␣%e",r);
13          r = sys_page_map(0, ROUNDDOWN(addr, PGSIZE), 0,
14              ROUNDDOWN(addr, PGSIZE), PTE_SYSCALL);
15          if (r < 0)
16              panic("sys_page_map:␣%e", r);
17      }
18  //  panic("flush_block not implemented");
19  }
```

如果 addr 映射到一页并且这一页已经是 dirty. 调用 ide_write 将这一页更新到磁盘.ide_write 以 sector 为单位进行处理. 将地址 addr 处 nsecs 个扇区的内容写入到从 secno 起始的 nsecs 个扇区上. 调用 sys_page_map 清除 pte 上的 dirty 标志, 因为这一页的内容和磁盘上的是一致的, 不在是 dirty(不一致).

Exercise 3. Use free_block as a model to implement alloc_block, which should find a free disk block in the bitmap, mark it used, and return the number of that block. When you allocate a block, you should immediately flush the changed bitmap block to disk with flush_block, to help file system consistency.
Use make grade to test your code. Your code should now pass "alloc_block".

```
int
alloc_block(void)
{
    // The bitmap consists of one or more blocks.  A single bitmap block
    // contains the in-use bits for BLKBITSIZE blocks.  There are
    // super->s_nblocks blocks in the disk altogether.

    // LAB 5: Your code here.
    uint32_t blockno, bi;
    for (blockno = 0; blockno < super->s_nblocks; blockno++) {
        if (block_is_free(blockno)) {
            bitmap[blockno/32] ^= 1<<(blockno%32);
            //bitmap begin from block 2
            bi = blockno / BLKBITSIZE + 2;
            flush_block(diskaddr(bi));
            return blockno;
        }
    }
//  panic("alloc_block not implemented");
    return -E_NO_DISK;
}
```

遍历整个磁盘, 找到一个空闲的块. 修改 bitmap 表示这一块被使用. 因为 bitmap 已经被修改, 为了维护一致性, 所以 bitmap 需要被写回磁盘. 最后返回盘块号.

Exercise 4. Implement file_block_walk and file_get_block. file_block_walk maps from a block offset within a file to the pointer for that block in the struct File or the indirect block, very much like what pgdir_walk did for page tables. file_get_block goes one step further and maps to the actual disk block, allocating a new one if necessary.
Use make grade to test your code. Your code should pass "file_open", "file_get_block", and "file_flush/file_truncated/file rewrite".

```
static int
file_block_walk(struct File *f, uint32_t filebno, uint32_t **ppdiskbno,
        bool alloc)
{
    // LAB 5: Your code here.
    uint32_t blockno, *indirect;
    if(filebno >= NDIRECT + NINDIRECT)
        return -E_INVAL;
    if (filebno < NDIRECT) {
            *ppdiskbno = &f->f_direct[filebno];
            return 0;
    } else {
        if (f->f_indirect) {
            indirect = diskaddr(f->f_indirect);
            *ppdiskbno = &indirect[filebno - NDIRECT];
```

```
            return 0;
        } else if (alloc) {
            if ((blockno = alloc_block()) < 0)
                return blockno; //-E_NO_DISK;
            f->f_indirect = blockno;
            memset(diskaddr(blockno), 0, BLKSIZE);
            flush_block(diskaddr(blockno));
    //      flush_block(f);
            indirect = diskaddr(f->f_indirect);
            *ppdiskbno = &indirect[filebno - NDIRECT];
            return 0;
        } else // alloc is 0
            return -E_NOT_FOUND;
    }

    //panic("file_block_walk not implemented");
}
```

这里需要注意的是要将申请的 INDIRECT 块清空, 并写回磁盘.

```
int
file_get_block(struct File *f, uint32_t filebno, char **blk)
{
    // LAB 5: Your code here.
    uint32_t r, *ppdiskbno;
    if ((r = file_block_walk(f, filebno, &ppdiskbno, 1)) < 0)
        return r;
    if (!*ppdiskbno) {
        if ((r = alloc_block()) < 0)
            return r; //-E_NO_DISK
        *ppdiskbno = r;
        memset(diskaddr(r), 0, BLKSIZE);
        flush_block(diskaddr(r));
    //  flush_block(f);
        *blk = diskaddr(r);
        return 0;
    } else {
        *blk = diskaddr(*ppdiskbno);
        return 0;
    }
//  panic("file_get_block not implemented");
}
```

新申请的块需要清 0 并且写回磁盘.

Exercise 5. Implement serve_read in fs/serv.c and devfile_read in lib/file.c.
serve_read's heavy lifting will be done by the already-implemented file_read in fs/fs.c (which, in turn, is just a bunch of calls to file_get_block). serve_read just has to provide the RPC interface for file reading. Look at the comments and code in serve_set_size to get a general idea of how the server functions should be structured.
Likewise, devfile_read should pack its arguments into fsipcbuf for serve_read, call fsipc, and handle the result.
Use make grade to test your code. Your code should pass "lib/file.c" and "file_read".

```
int
serve_read(envid_t envid, union Fsipc *ipc)
{
    struct Fsreq_read *req = &ipc->read;
    struct Fsret_read *ret = &ipc->readRet;

    if (debug)
        cprintf("serve_read␣%08x␣%08x␣%08x\n", envid, req->req_fileid, req
        ->req_n);

    // Look up the file id, read the bytes into 'ret', and update
    // the seek position.  Be careful if req->req_n > PGSIZE
    // (remember that read is always allowed to return fewer bytes
    // than requested).  Also, be careful because ipc is a union,
    // so filling in ret will overwrite req.
    //
    // Hint: Use file_read.
    // Hint: The seek position is stored in the struct Fd.
    // LAB 5: Your code here
    int r, count;
    struct OpenFile *o;
    if((r = openfile_lookup(envid, req->req_fileid, &o)) < 0)
        return r;
    count = (req->req_n > PGSIZE) ? PGSIZE : req->req_n;
    if((r = file_read(o->o_file, ret->ret_buf, count, o->o_fd->fd_offset))
        < 0)
        return r;
    o->o_fd->fd_offset += r;
    return r;
    //panic("serve_read not implemented");
}
```

serve_read 从文件服务器端读取文件, 参数由用户端通过 IPC 传送到服务端地址 Fsipc *ipc 处. Fsipc 中包含要去读的文件 id. 和要读取的字节数. 首先在打开文件表中查找文件. 然后 file_read 读取文件. 最后更新读写偏移.

```
static ssize_t
```

```
devfile_read(struct Fd *fd, void *buf, size_t n)
{
    // Make an FSREQ_READ request to the file system server after
    // filling fsipcbuf.read with the request arguments.  The
    // bytes read will be written back to fsipcbuf by the file
    // system server.
    // LAB 5: Your code here
    int r;
    fsipcbuf.read.req_fileid = fd->fd_file.id;
    fsipcbuf.read.req_n = n;
    if((r = fsipc(FSREQ_READ, &fsipcbuf)) < 0)
        return r;
    memmove(buf, fsipcbuf.readRet.ret_buf, r);
    return r;
//  panic("devfile_read not implemented");
}
```

客户端读操作, 将参数放在 fsipcbuf 中, 然后发送到服务器端, 服务器端将读取的数据同样也放在 fsipcbuf, 然后发送回客户端. 最后 memmove 至 buf 中.

Exercise 6. Implement serve_write in fs/serv.c and devfile_write in lib/file.c. Use make grade to test your code. Your code should pass "file_write" and "file_read after file_write".

file_write,devfile_write 和上面的 file_read,devfile_read 架构类似. 不做其他的解释.

```
int
serve_write(envid_t envid, struct Fsreq_write *req)
{
    if (debug)
        cprintf("serve_write␣%08x␣%08x␣%08x\n", envid, req->req_fileid, req
        ->req_n);

    // LAB 5: Your code here.
    int r, count;
    struct OpenFile *o;
    if((r = openfile_lookup(envid, req->req_fileid, &o)) < 0)
        return r;
//  count = (req->req_n > PGSIZE) ? PGSIZE : req->req_n;
    count = req->req_n;
//  in client side we have check count <= req_buf size;
    if((r = file_write(o->o_file, req->req_buf, count, o->o_fd->fd_offset))
         < 0)
        return r;
    o->o_fd->fd_offset += count;
    return r;
//  panic("serve_write not implemented");
}
```

```
static ssize_t
devfile_write(struct Fd *fd, const void *buf, size_t n)
{
    // Make an FSREQ_WRITE request to the file system server.  Be
    // careful: fsipcbuf.write.req_buf is only so large, but
    // remember that write is always allowed to write *fewer*
    // bytes than requested.
    // LAB 5: Your code here
    int r, count, bufsize;
    bufsize = PGSIZE - sizeof(int) - sizeof(size_t);
    count = (bufsize > n) ? n : bufsize;
    fsipcbuf.write.req_fileid = fd->fd_file.id;
    fsipcbuf.write.req_n = count;
    memmove(fsipcbuf.write.req_buf, buf, count);
    if((r = fsipc(FSREQ_WRITE, NULL)) < 0)
        return r;
    return r;
//  panic("devfile_write not implemented");
}
```

Exercise 7. Implement open. The open function must find an unused file descriptor using the fd_alloc() function we have provided, make an IPC request to the file system environment to open the file, and return the number of the allocated file descriptor. Be sure your code fails gracefully if the maximum number of files are already open, or if any of the IPC requests to the file system environment fail. Use make grade to test your code. Your code should pass "open", "large file", "motd display", and "motd change".

```
open(const char *path, int mode)
{
    // Find an unused file descriptor page using fd_alloc.
    // Then send a file-open request to the file server.
    // Include 'path' and 'omode' in request,
    // and map the returned file descriptor page
    // at the appropriate fd address.
    // FSREQ_OPEN returns 0 on success, < 0 on failure.
    //
    // (fd_alloc does not allocate a page, it just returns an
    // unused fd address.  Do you need to allocate a page?)
    //
    // Return the file descriptor index.
    // If any step after fd_alloc fails, use fd_close to free the
    // file descriptor.

    // LAB 5: Your code here.
    struct Fd *fd;
    int r;
    if((r = fd_alloc(&fd)) < 0)
        return r;
```

Exercise 8. spawn relies on the new syscall sys_env_set_trapframe to initialize the state of the newly created environment. Implement sys_env_set_trapframe. Test your code by running the user/icode program from kern/init.c, which will attempt to spawn /init from the file system.

```
22      if (strlen(path) > MAXPATHLEN) {
23          r = -E_BAD_PATH;
24          goto err;
25      }
26      strcpy(fsipcbuf.open.req_path, path);
27      fsipcbuf.open.req_omode = mode;
28      if ((r = fsipc(FSREQ_OPEN, (void*)fd)) < 0)
29          goto err;
30      return fd2num(fd);
31  err:
32      fd_close(fd, 1);
33      return r;
34  }
```

首先分配一个文件描述符结构 struct Fd. 然后将参数放入 fsipcbuf 发送给服务器端. 服务器端会在打开文件表中分配一项. 客户端和服务器端的 struct Fd 结构映射到一个物理页. open 最后调用 fd2num 返回文件描述符.

```
1   static int
2   sys_env_set_trapframe(envid_t envid, struct Trapframe *tf)
3   {
4       // LAB 5: Your code here.
5       // Remember to check whether the user has supplied us with a good
6       // address!
7       struct Env *env;
8       int r;
9       if ((r = envid2env(envid, &env, 1)) < 0)
10          return r; /*-E_BAD_ENV*/
11      if ((r = user_mem_check(curenv, tf, sizeof(struct Trapframe), PTE_U |
            PTE_P)) < 0)
12          return r;
13
14      env->env_tf = *tf;
15      env->env_tf.tf_eflags |= FL_IF;
16      return 0;
17  //  panic("sys_env_set_trapframe not implemented");
18  }
```

1.6 Lab 6: Network Driver

1.6.1 Introduction

本次实验主要实现两个部分

1. e1000(Intel 82540EM) 网卡驱动
2. 输入进程和输出进程代码.

JOS 的网络栈是网卡驱动以及输入和输出进程框架图如下

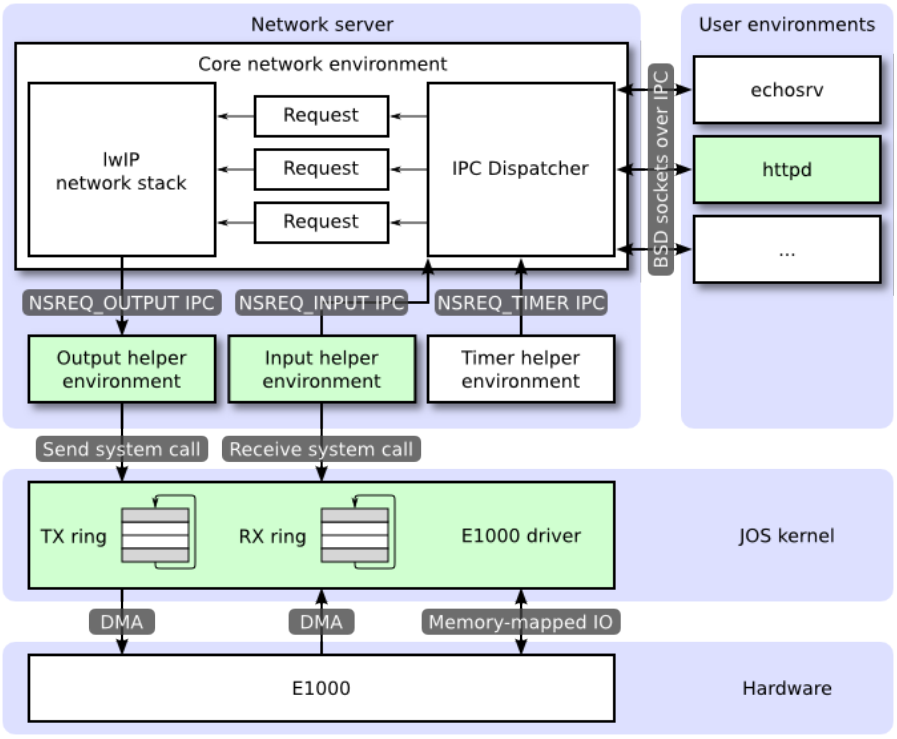


图 21: 系统框架图

这图中包含用户进程 (user enviroments), 网络栈 (core network environment), 输入和输出进程 (input helper enviroment 和 output helper enviroment), 网卡驱动 (E1000 driver). 其中绿色部分是需要在实验中实现的. 用户进程如 httpd,echosrc 为了接受数据和发送数据, 通过 socket 访问网络栈. 但是 JOS 中用户程序并不直接使用 socket 结构. 而是将网络看成一个文件, 通过文件描述符从网络中读写数据. 例如文件描述符号结构添加了 struct FdSock fd_sock 结构:

```
1 struct Fd {
2     int fd_dev_id;
3     off_t fd_offset;
4     int fd_omode;
5     union {
```

```
        // File server files
        struct FdFile fd_file;
        // Network sockets
        struct FdSock fd_sock;
    };
};
```

并且在文件系统的设备层添加了一个 devsock 层.

```
static struct Dev *devtab[] =
{
    &devfile,
    &devsock,
    0
};
```

而 devsock 的结构如下:

```
struct Dev devsock =
{
    .dev_id =   's',
    .dev_name = "sock",
    .dev_read = devsock_read,
    .dev_write =    devsock_write,
    .dev_close =    devsock_close,
    .dev_stat = devsock_stat,
};
```

从这里我们可以看到对应的设备读写函数 dev_read,dev_write 是与 socket 相关的函数 devsock_read,devsock_write. 这样在文件描述符层将文件系统和网络做了统一, 都通过文件描述符来访问, 根据 struct Fd 结构中 fd_dev_id 来找到对应的 dev. 然后调用 dev 相关的读写函数 dev_read 和 dev_write. 对于文件系统这两个函数就是文件系统相关的操作, 具体查看 lab6. 而对于网络, dev 相关的读写函数则会对应到 socket 相关的函数 devsock_write,devsock_read 等函数. 从图中我们可以看到网络栈也是一个单独的进程. 虽然用户进程使用文件描述符来访问网络栈, 但是具体的通信方式采用类似 lab6 中实现的文件服务器, 网络栈作为服务器端接受用户进程的请求, 然后返回结果. 通信方式采用 RPC(远程过程调用), 在 JOS 中采用 IPC(进程间通信) 来实现 RPC. IPC dispatcher 接受用户的请求, 根据请求的类型调用 lwIP 网络栈提供的接口.lwIP 是一个轻量级的 TCP/IP 协议族. lwip 通过 NSREQ_OUTPUT 类型的 IPC 将数据发送到输出进程 (output helper enviroment) , 通过 NSREQ_INPUT 类型的 IPC 从输入进程 (input helper enviroment) 中获取数据. 输入和输出进程通过系统调用访问 e1000 驱动, 从而获取数据和发送数据.

PCI 总线配置 该部分主要调研了《The Linux Kernel》by David A Rusling, 中关于 PCI 的介绍和网络资源 http://wiki.osdev.org/PCI. 下图是一个基于 PCI 的系统示意图.

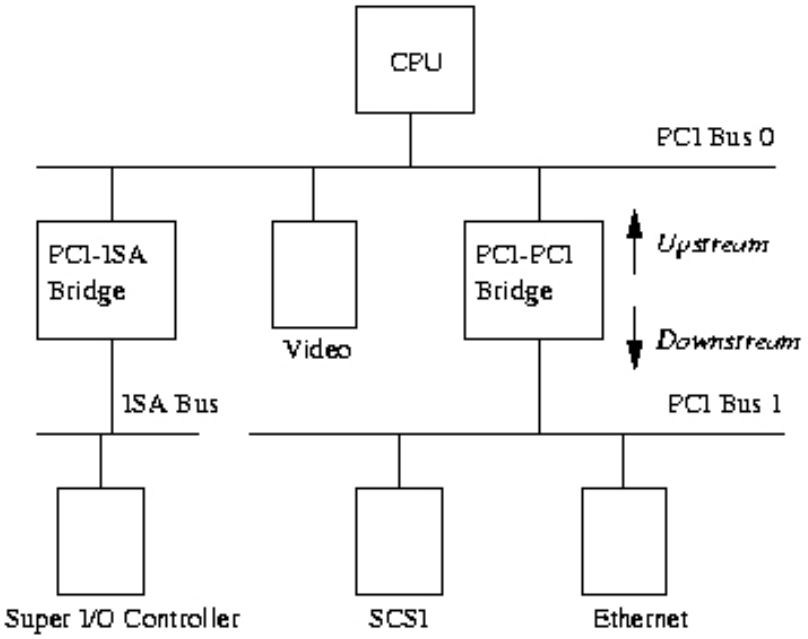


图 22: 一个基于 PCI 的系统示意图

上图是一个基于 PCI 的系统示意图。PCI 总线和 PCI-PCI 桥接器在连接系统中设备到上起关键作用，在这个系统中 CPU 和视频设备被连到 PCI bus 0 上，它是系统中的主干 PCI 总线。而 PCI-PCI 桥接器这个特殊 PCI 设备将主干总线 PCI bus 0 与下级总线 PCI bus 1 连接到一起。PCI 标准术语中，PCI bus 1 是 PCI-PCI 桥接器的 downstream 而 PCI bus 0 是此桥接器的 up-stream。SCSI 和以太网设备通过二级 PCI 总线连接到这个系统中。而在物理实现上，桥接器和二级 PCI 总线被集成到一块 PCI 卡上。而 PCI-ISA 桥接器用来支持古老的 ISA 设备，图中有一个高级 I/O 控制芯片来控制键盘、鼠标及软盘设备. 外设有其自身的内存空间.CPU 可以自由存取此空间, 但设备对系统主存的访问将处于 DMA（直接内存访问）通道的严格控制下.PCI 设备有三种地址空间：PCI I/O、PCI 内存和 PCI 配置空间。CPU 则可以访问所有这些地址空间.PCI I/O 和 PCI 内存由设备驱动程序使用而 PCI 配置空间被中的 PCI 初始化代码使用. 系统初始化和配置设备时需要访问该设备的配置空间.

PCI 配置空间　PCI 规范提供了 PCI 配置空间来使得软件能够初始化个配置每个设备. 系统中每个 PCI 设备，包括 PCI-PCI 桥接器在内，都有一个 64 字节的配置数据结构，它通常位于 PCI 配置地址空间中. 为了访问这个数据结构需要两个 32 位的 I/O 位置 (端口). 第一个是 CONFIG_ADDRESS(数值是 0xCF8), 第二个是 CONFIG_DATA(0xCFC).CONFIG_ADDRESS 是一个 32

位的寄存器, 它的结构如下:

31	30 - 24	23 - 16	15 - 11	10 - 8	7 - 2	1 - 0
Enable Bit	Reserved	Bus Number	Device Number	Function Number	Register Number	00

图 23: CONFIG_ADDRESS

第 31 位是使能位,16-23 是指定 bus 号, 选择特定的 bus, 例如图 1 中的 bus 0 和 bus 1, 11-15 在特定的 bus 上选择设被, 这个 5 位是设备号. 一个设备可能有多个模块, 各个模块有不同的功能. 8-10 制定了设备中的功能号. 2-7 选择该设备配置空间数据结构 (64 字节) 中的特定寄存器. 因为读写都是按 32 位进行, 所以最低两位 0-1 全部是 0. 初始化和配置设备时为了访问配置空间数据结构中某个寄存器时, 按照上面介绍的格式将地址写入到端口 CONFIG_ADDRESS, 然后从 CONFIG_DATA 端口就可以读取到所需的数据.

PCI 配置数据结构　PCI bus 上有两种两种设备, 一种是普通的设备如网卡, 磁盘等, 另一个是连接下一级 bus 的 PCI-TO-PCI 桥, 这里主要介绍第一种设备的配置结构. 后一种可以查阅上面提到的网址 http://wiki.osdev.org/PCI.
第一种配置结构 (Header Type 是 0x00h) 如图, 第二种配置结构的 Header Type 是 0x01h, 这里不做介绍.

register (offset)	bits 31-24	bits 23-16	bits 15-8	bits 7-0
00	Device ID		Vendor ID	
04	Status		Command	
08	Class code	Subclass	Prog IF	Revision ID
0C	BIST	Header type	Latency Timer	Cache Line Size
10	Base address #0 (BAR0)			
14	Base address #1 (BAR1)			
18	Base address #2 (BAR2)			
1C	Base address #3 (BAR3)			
20	Base address #4 (BAR4)			
24	Base address #5 (BAR5)			
28	Cardbus CIS Pointer			
2C	Subsystem ID		Subsystem Vendor ID	
30	Expansion ROM base address			
34	Reserved			Capabilities Pointer
38	Reserved			
3C	Max latency	Min Grant	Interrupt PIN	Interrupt Line

图 24: Configuration Space

这里只做简单的部分介绍, 后面用到时在做详细的解释

1. Device ID 识别特定的设备, 由制造商指定.
2. Vendor ID 制造商 ID. 表示这是哪一个制造商.0xFFFF 表示是一个无效值
3. status 记录状态信息
4. Header Type 表示是设备的类型和是否是多功能的设备. 如果值是 0x00 表示是一个普通设备，0x01 表示是一个 PCT-TO-PCI 桥. 如果这个寄存器的第 7 位是 1 表示是一个多功能的设备. 否则是一个单功能设备.CONFIG_ADDRESS 中就介绍了有些设备需要选择对应的功能号.
5. Base address, 配置空间中总共有 6 个这样的寄存器, 分别是 BAR0-BAR6(Base Address Register) 但不一定全都用的上. 每个设备内部都有很多寄存器, 通过将这些寄存器 (或者一个 EEPROM) 映射到物理内存, 然后通过访问物理内存将实现了对这些寄存器的访问. 这片物理物理内存就是 MMIO(memory mapped IO) 这 6 个寄存器就存了该设备内存寄存

器应该映射到的物理地址. 除了映射地址外, 还需要映射区域的大小. 映射区域的大小也可以从 BAR 寄存器中读出, 具体的操作是先保存原有 BAR 的值, 然后写入全 1 到 BAR, 再从 BAR 读取值, 对读取的值所有位 (不包括非地址位, 对于内存映射是低 4 位是非地址位, 非地址位全部当成 0) 取反 (操作) 加 1. 那么这个值就是映射区域的大小, 具体可以查阅 PCI Local Bus Specification. 最后恢复 BAR 的值. JOS 实验中用到的 E1000 网卡只用到了 BAR0.

设备的初始化　PCI 设备在使用前必须被发现和初始化. 发现设备的过程就是遍历 PCI 总线的过程. 初始化设备包括为设备分配前面提到内存空间和 IO 空间, 以及 IRQ. JOS 中 pci_init 函数用来进行设备的发现和初始化, 该函数遍历整个 PCI 总线, 当发现一个设备, 就该设备的配置空间中读取制造商 ID(vendor ID) 和设备 ID(device ID). 然后用这两个值去搜索 pci_attch_vendor 数组, 该数组中的元素都是如下的类型

```
struct pci_driver {
    uint32_t key1, key2;
    int (*attachfn) (struct pci_func *pcif);
};
```

如果发现的设备其制造商 ID 和设备 ID 与 key1,key2 匹配. 然后就调用 attachfn 函数初始化该设备. 该函数的参数是一个 pci 功能 (pci function), 有的设备有多个功能, 但是 e1000 仅有一个功能.

```
struct pci_func {
    struct pci_bus *bus;

    uint32_t dev;
    uint32_t func;

    uint32_t dev_id;
    uint32_t dev_class;

    uint32_t reg_base[6];
    uint32_t reg_size[6];
    uint8_t irq_line;
};
```

1. bus 表示该设备链接到的 bus
2. dev 表示设备号, 前面图 3 中用 5 位来寻址设备号
3. func 表示功能号, 图 3 中用 3 位来寻址功能号
4. dev_id 识别特定的设备, 由制造商指定, 就是设备配置空间中的 Device ID
5. dev_class 表示设备类型, 就是配置空间的中偏移 0x8 处.
6. reg_base[6], 就是该设备配置空间中 BAR0-BAR6 的值, 记录了内存映射的基地址.

7. reg_size[6], 记录了该设备内存映射区域的大小, 前面已经介绍了如何从 BAR 中获取该值.
8. irq_line 表示中断请求 line.

我们接下来分析下 JOS 中设备发现和初始化的代码. 首先用两个数据结构来描述 PCI bus 和 PCI 设备. 前面提到的 struct pci_func 用来描述 PCI 设备, 下面的 struct pci_bus 用来描述 PCI BUS.

```
struct pci_bus {
    struct pci_func *parent_bridge;
    uint32_t busno;
};
```

1. parent_bridge 表示该总线通过哪一个桥链接到上一级总线, PCI 桥也是一个设备,parent_bridge 表示该桥
2. busno 表示 bus 号

JOS 中设备的发现和初始化从 pci_init 函数开始, 采用递归扫描各级 bus, 发现各级 bus 上的设备并初始化

```
int
pci_init(void)
{
    static struct pci_bus root_bus;
    memset(&root_bus, 0, sizeof(root_bus));

    return pci_scan_bus(&root_bus);
}
```

root_bus 是如图 1 中的 bus 0, 从这个 bus 调用 pci_scan_bus 开始递归搜索.

```
static int
pci_scan_bus(struct pci_bus *bus)
{
    int totaldev = 0;
    struct pci_func df;
    memset(&df, 0, sizeof(df));
    df.bus = bus;

    for (df.dev = 0; df.dev < 32; df.dev++) {
        uint32_t bhlc = pci_conf_read(&df, PCI_BHLC_REG);
        if (PCI_HDRTYPE_TYPE(bhlc) > 1)     // Unsupported or no device
            continue;

        totaldev++;

        struct pci_func f = df;
        for (f.func = 0; f.func < (PCI_HDRTYPE_MULTIFN(bhlc) ? 8 : 1);
             f.func++) {
            struct pci_func af = f;

            af.dev_id = pci_conf_read(&f, PCI_ID_REG);
```

```
22              if (PCI_VENDOR(af.dev_id) == 0xffff)
23                  continue;
24
25              uint32_t intr = pci_conf_read(&af, PCI_INTERRUPT_REG);
26              af.irq_line = PCI_INTERRUPT_LINE(intr);
27
28              af.dev_class = pci_conf_read(&af, PCI_CLASS_REG);
29              if (pci_show_devs)
30                  pci_print_func(&af);
31              pci_attach(&af);
32          }
33      }
34
35      return totaldev;
36  }
```

pci_scan_bus 以参数 bus 为根, 搜索和初始化该 bus 上的设备, 如果该 bus 上桥, 那么递归到下一级 bus 上搜索和初始化. 因为前面图 3 中设备号只有 5 位寻址, 所以一个 bus 上最多 32 个设备. 并且功能号用 3 位寻址, 所以最多 8 个功能号.

```
1   struct pci_func df;
2  for (df.dev = 0; df.dev < 32; df.dev++) {
3    ...
4     struct pci_func f = df;
5    for (f.func = 0; f.func < (PCI_HDRTYPE_MULTIFN(bhlc) ? 8 : 1);   f.func
         ++) {
6      struct pci_func af = f;
7     pci_attach(&af);
8    }
9  }
```

这就是搜索的主要框架, 依次搜索该 bus 上的 32 个设备 (最多 32 个设备). 每个设备最多 8 个功能号 (如果是多功能设备), 单功能设备显然只有 1 个功能. 然后搜索各个功能号, 调用 pci_attach 初始该功能 (设备). 我们来看具体的细节

```
1     uint32_t bhlc = pci_conf_read(&df, PCI_BHLC_REG);
2           if (PCI_HDRTYPE_TYPE(bhlc) > 1)     // Unsupported or no device
3               continue;
```

读取该该设备配置空间中的 header type 部分 (偏移 0xC) 从而判断该设备的类型,0x00 表示普通设备,0x01 表示 PCI 桥. 大于 1JOS 不做处理. 所以 continue.

```
1   struct pci_func af = f;
2
3               af.dev_id = pci_conf_read(&f, PCI_ID_REG);
4               if (PCI_VENDOR(af.dev_id) == 0xffff)
5                   continue;
```

读取设备配置空间中的 vendor ID 和 device id. 如果 vendor id 是 0xffff 表示无效.

```
uint32_t intr = pci_conf_read(&af, PCI_INTERRUPT_REG);
           af.irq_line = PCI_INTERRUPT_LINE(intr);
```

读取设备配置空间中的中断请求 line 号.

```
 af.dev_class = pci_conf_read(&af, PCI_CLASS_REG);
```

读取设备配置空间中的关于设备 class 的部分 (偏移 0x08 处). 在分析 pci_attach 之前我们分析下 pci_conf_read 函数

```
static uint32_t
pci_conf_read(struct pci_func *f, uint32_t off)
{
    pci_conf1_set_addr(f->bus->busno, f->dev, f->func, off);
    return inl(pci_conf1_data_ioport);
}
```

参数 off 表示配置空间中的寄存器偏移.pci_conf1_set_addr 形成一个图 3 中格式的地址,off 就是配置空间中的 register 号 (偏移). 并输出到 pci_conf1_addr_ioport. 该符号的定义如下

```
static uint32_t pci_conf1_addr_ioport = 0x0cf8;
```

这就是我们前面提到的 CONFIG_ADDRESS. 设置好地址后, 我们从 pci_conf1_data_ioport 读取数据, 该符号的定义如下

```
static uint32_t pci_conf1_data_ioport = 0x0cfc;
```

这就是我们前面提到的 CONFIG_DATA.
接下来我们分析 pci_attach 函数.

```
static int
pci_attach(struct pci_func *f)
{
    return
        pci_attach_match(PCI_CLASS(f->dev_class),
                 PCI_SUBCLASS(f->dev_class),
                 &pci_attach_class[0], f) ||
        pci_attach_match(PCI_VENDOR(f->dev_id),
                 PCI_PRODUCT(f->dev_id),
                 &pci_attach_vendor[0], f);
}
```

该函数很简单, 如果是桥, 就调用第一个 pci_attch_match, 如果是普通设备就调用第二个 pci_attach_match.

对于桥 ,pci_attch_match 使用该桥的 class code 和 subclass 去数组 pci_attch_class 中搜索, 如果匹配就调用对应的 attachfn 函数. JOS 中 pci_attach_class 只有一个有效元素.

```
struct pci_driver pci_attach_class[] = {
    { PCI_CLASS_BRIDGE, PCI_SUBCLASS_BRIDGE_PCI, &pci_bridge_attach },
    { 0, 0, 0 },
};
```

所以会调用 pci_bridge_attach. 我们来分析这个函数.

```
static int
pci_bridge_attach(struct pci_func *pcif)
{
    uint32_t ioreg  = pci_conf_read(pcif, PCI_BRIDGE_STATIO_REG);
    uint32_t busreg = pci_conf_read(pcif, PCI_BRIDGE_BUS_REG);

    if (PCI_BRIDGE_IO_32BITS(ioreg)) {
        cprintf("PCI: %02x:%02x.%d: 32-bit bridge IO not supported.\n",
            pcif->bus->busno, pcif->dev, pcif->func);
        return 0;
    }

    struct pci_bus nbus;
    memset(&nbus, 0, sizeof(nbus));
    nbus.parent_bridge = pcif;
    nbus.busno = (busreg >> PCI_BRIDGE_BUS_SECONDARY_SHIFT) & 0xff;

    if (pci_show_devs)
        cprintf("PCI: %02x:%02x.%d: bridge to PCI bus %d--%d\n",
            pcif->bus->busno, pcif->dev, pcif->func,
            nbus.busno,
            (busreg >> PCI_BRIDGE_BUS_SUBORDINATE_SHIFT) & 0xff);

    pci_scan_bus(&nbus);
    return 1;
}
```

首先读取 PCI 桥配置空间中的 IO/BASE 寄存器. 判断是否支持 32 位. I/O Base 和 I/O Limit 的低 4 位包含相同的值, 如果是值是 00 表示 16 位 IO 寻址, 0x01 表示 32 位 IO 寻址. 详细可以查看 PCI-to-PCI Bridge Architecture Specification. 然后读取 bus 号寄存器. 因为桥链接连个 bus. 一个是上一级 bus, 一个是下一级 bus. 接下来 nbus 描述这个桥链接的下一级 bus. 设置好该 nbus 的 parent_bridge 为 pcif. 表示 nbus 通过该桥链接到上一级 bus. busno 表示该 nbus 的总线号, 如图 1 中的 bus1. 桥配置空间 bus 号寄存器中有 Primary Bus Number 和 Secondary Bus Number 分别表示桥链接的两个 bus 的 bus 号. 因为 nbus 是下一级 bus, 所以 busno 的值是 Secondary Bus Number. 接下来调用 pci_scan_bus 递归搜索 nbus 上的设备.

对于普通设备 ,pci_attch_match 使用该设备的 vendor id 和 device id(不是设备号, 这其实是 product id, 由厂商指定) 去数组 pci_attch_vendor 中搜索, 如果匹配就调用对应的 attachfn 函数. JOS 中 pci_attach_vendor 只有一个有效

元素.attachfn 对应的是 attach_e1000

```
struct pci_driver pci_attach_vendor[] = {
    { 0x8086, 0x100E, &attach_e1000},
    { 0, 0, 0 },
};
```

其中 attach_e1000 就是 e1000 的初始化函数. 这个元素是需要在实验中添加的. PCI 设备的搜索有初始化基本框架介绍到此结束,attachfn 是设备相关的函数, 与具体每个设备细节有关. 接下来在实验以 e1000 为例子说明.

1.6.2 Part A: Initialization and transmitting packets

Exercise 1. Add a call to time_tick for every clock interrupt in kern/trap.c. Implement sys_time_msec and add it to syscall in kern/syscall.c so that user space has access to the time.

系统中维护了一个全局变量 ticks, 表示时间, 系统启动时初始化为 0. time_tick 函数将该值加 1. 系统的时钟中断是 100 次每秒, 即每 10 毫秒 1 次时钟中断. 所以发生时钟中断时, 在中断处理函数中调用 time_tick 将值加 1 即可. 表示滴答了一下. 但是 SMP 系统中, 每个核都会发生时钟中断. 如果每个中断都直接将 ticks 加 1. 那么时间就混乱了. 所以只能由一个核给该值加 1.xv6 中是由 cpu 0 来维护这个变量的. 我也参照这种方法.

```
    if(tf->tf_trapno == IRQ_OFFSET + IRQ_TIMER) {
        if (thiscpu->cpu_id == 0) {
            lapic_eoi();
            time_tick();
            sched_yield();
        }
    }
```

Exercise 3. Implement an attach function to initialize the E1000. Add an entry to the pci_attach_vendor array in kern/pci.c to trigger your function if a matching PCI device is found (be sure to put it before the 0, 0, 0 entry that mark the end of the table). You can find the vendor ID and device ID of the 82540EM that QEMU emulates in section 5.2. You should also see these listed when JOS scans the PCI bus while booting.
For now, just enable the E1000 device via pci_func_enable. We'll add more initialization throughout the lab.
We have provided the kern/e1000.c and kern/e1000.h files for you so that you do not need to mess with the build system. You may still need to include the e1000.h file in other places in the kernel.
When you boot your kernel, you should see it print that the PCI function of the E1000 card was enabled. Your code should now pass the pci attach test of make grade.

查阅手册, 获得 e1000 的 device id(product id) 和 vendor id. 在 pci_attach_vendor 添加一项

```
struct pci_driver pci_attach_vendor[] = {
    { 0x8086, 0x100E, &attach_e1000},
    { 0, 0, 0 },
};
```

然后直接调用 pci_func_enable 使能 e1000. 该函数首先向配置空间中 command&status register 写入一个值: PCI_COMMAND_IO_ENABLE | PCI_COMMAND_MEM_ENABLE | PCI_COMMAND_MASTER_ENABLE. 表示 I/O Access Enable, Memory Access Enable 和 Enable Mastering(不太明白这是什么意思). 然后读入配置空间中的 BAR0-BAR6 到 reg_base[0-5], 读取 BAR0-BAR6 对应的映射区域大小到 reg_size[0-5]. 采用的是前面提到的方法. 不过这里的取反 (操作) 加 1 操作用的是宏 PCI_MAPREG_MEM_SIZE.

```
#define PCI_MAPREG_MEM_SIZE(mr)                 \
        (PCI_MAPREG_MEM_ADDR(mr) & -PCI_MAPREG_MEM_ADDR(mr))
```

就是原值和取负号的值做 & 操作. 这和取反 (操作) 加 1 的结果有点不同. 不知道为什么可以这么做. 貌似是 mr 的格式是 1*10*0 这种, 即前面全是 1, 后面全是 0. 不存在 0 和 1 交替这种状况. 所以最后这两种计算结果相等.

Exercise 4. In your attach function, create a virtual memory mapping for the E1000's BAR 0. Since this is device memory and not regular DRAM, you'll have to tell the CPU that it isn't safe to cache access to this memory. Luckily, the page tables provide bits for this purpose; simply create the mapping with PTE_PCD|PTE_PWT (cache-disable and write-through). (If you're interested in more details on this, see section 10.5 of IA32 volume 3A.) You'll want to record where you made this mapping in a variable so you can later access the registers you just mapped. Take a look at the lapic variable in kern/lapic.c for an example of one way to do this. If you do use a pointer to the device register mapping, be sure to declare it volatile; otherwise, the compiler is allowed to cache values and reorder accesses to this memory.
To test your mapping, try printing out the device status register (section 13.4.2). This is a 4 byte register that starts at byte 8 of the register space. You should get 0x80080783, which indicates a full duplex link is up at 1000 MB/s, among other things.

BAR0(即 reg_base[0],reg_size[0]) 存放了 MMIO 的基地址和大小. 对这个地址的读写不会对内存寻址, 而是直接寻址到设备.

```
addr = pcif->reg_base[0];
  size = pcif->reg_size[0];
  size = ROUNDUP(size, PGSIZE);
  perm = PTE_P | PTE_W | PTE_PCD | PTE_PWT;

  // Memory map I/O for PCI device
  boot_map_region(kern_pgdir, KSTACKTOP, size, addr, perm);
  e100 = (uint32_t *)KSTACKTOP;
  cprintf("e1000 status register: %x\n", e100[E1000_STATUS/sizeof(
     uint32_t)]);
```

这个 BAR0 映射到 KSTACKTOP, 大小是 size.BAR0 中保存的是物理地址. 因为 size 小于 4MB 所以可以映射到这区域 [KSTACKTOP, KERNBASE), 另外因为这是设备内存不是常规的 DRAM, 所以不用缓存, 所以在页表对应的权限中需要设置 PTE_PCD(cache-disable) 和 PTE_PWT(write-through). 我们将映射的虚基地址保存在变量 e100 中. 此后对设备的访问就可以通过 e100 来进行.

Exercise 5. Perform the initialization steps described in section 14.5 (but not its subsections). Use section 13 as a reference for the registers the initialization process refers to and sections 3.3.3 and 3.4 for reference to the transmit descriptors and transmit descriptor array.
Be mindful of the alignment requirements on the transmit descriptor array and the restrictions on length of this array. Since TDLEN must be 128-byte aligned and each transmit descriptor is 16 bytes, your transmit descriptor array will need some multiple of 8 transmit descriptors. However, don't use more than 64 descriptors or our tests won't be able to test transmit ring overflow.
For the TCTL.COLD, you can assume full-duplex operation. For TIPG, refer to the default values described in table 13-77 of section 13.4.34 for the IEEE 802.3 standard IPG (don't use the values in the table in section 14.5).

网络包在内存和网卡间的传送是采用 DMA 的方式. 下面介绍下内存和网卡间传递数据包的过程.

发送数据包　系统在内存中维护一个传送描述符数组, 有的地方也成为发送描述符环形缓冲区. 数组中的元素是发送描述符. 该描述符中包含一个地址域, 表明数据包在内存中的位置, 还有长度域表示包的大小, 还有状态域表示该报是否发送到网卡, 该描述符是否可以复用了. 网卡中包含有发送描述符数组基地址寄存器和该数组的长度寄存器, 这两个寄存器分别用数组的基地址和长度初始化. 同时网卡中还包含发送描述符 header 和 tail 指针. 其实就是两个寄存器. 分别作为发送描述符数组的索引. 当用户进程发送包时先将数据从用户缓冲区拷贝到内存中发送描述符地址域所指向的地址. 具体是哪个发送描述符呢? 就是 tail 指向的那个描述符. 拷贝完成后将 tail 加 1. 然后硬件系统会自动的将包从该缓冲区采用 DMA 方式传送到网卡. 然后设置描述符号的状态域. 表明传输完毕. 将 header 加 1. 之所以成为唤醒缓冲区, 是因为 tail 和 header 都是取模加 1 的. 当这两值达到数组中最后一个索引后, 再加 1 就又回到 0 了. 从面的描述可见用户的数据包每拷贝到内存的缓冲区中.tail 就加 1. 硬件每将一个包发送到网卡,header 就加 1. 所以当 tail 等于 header 时系统中就没有包要发送了. 下面来看一下发送描述符的具体结构, 该结构是按照特定的规范给出的.

```
1  struct tx_desc
2  {
3      uint64_t addr;
4      uint16_t length;
5      uint8_t cso;
6      uint8_t cmd;
7      uint8_t status;
8      uint8_t css;
9      uint16_t special;
10 }__attribute__((__packed__));
```

该结构占 16 字节.

1. addr 该描述符对应的数据缓冲区
2. length 该数据缓冲区的大小
3. status 包含一些状态信息, 例如该描述符是否完成传送, 是否可以复用

其他的可以查看 e1000 手册. 在 kern/e1000.c 中定了发送描述符环形缓冲区 tx_fifo

```
struct tx_desc tx_fifo[TX_SIZE]__attribute__((aligned(16)));
```

该描述符缓冲区起始地址结构必须对齐到 16 字节的边界上 (可以查阅手册 PCI/PCI-X Family of Gigabit Ethernet Controllers Software Developer's Manual 中 section3.4). 该缓冲区中有 TX_SIZE(32, 可以自定义但必须要 8 的倍数) 个描述符. 实际数据包的存放区域是 tx_buffer

```
uint8_t *tx_buffer[TX_SIZE * MAX_PACKET_SIZE];
```

在 JOS 中我们默认一个数据包对应一个描述符 (有的包很大可能对应多个描述符). 并且每个包最大是 MAX_PACKET_SIZE(largest possible standard Ethernet packet (1518 bytes)). 所以缓冲区的大小是 TX_SIZE * MAX_PACKET_SIZE 字节. 描述符数组中第 i 个描述符的 addr 域指向 tx_buffer+MAX_PACKET_SIZE*i; 下面给出发送初始化相关代码

```
// transmit initialization
  // Program the Transmit Descriptor Base Address Registers
  e100[E1000_TDBAL/sizeof(uint32_t)] = PADDR((void*)tx_fifo);
  e100[E1000_TDBAH/sizeof(uint32_t)] = 0x0;

  // Set the Transmit Descriptor Length Register
  e100[E1000_TDLEN/sizeof(uint32_t)] = TX_SIZE * sizeof(struct tx_desc);

  // Set the Transmit Descriptor Head and Tail Registers
  e100[E1000_TDH/sizeof(uint32_t)] = 0;
  e100[E1000_TDT/sizeof(uint32_t)] = 0;

  // Initialize the Transmit Control Register
  // Transmit Enable
  e100[E1000_TCTL/sizeof(uint32_t)] |= E1000_TCTL_EN;
  //Padding short packets, makes the packet 64 bytes long.
  e100[E1000_TCTL/sizeof(uint32_t)] |= E1000_TCTL_PSP;
  //Configure the Collision Threshold (TCTL.CT) to the desired value.
  //Ethernet standard is 10h.
  //This setting only has meaning in half duplex mode.--by 8254x manual
  e100[E1000_TCTL/sizeof(uint32_t)] |= 0x10 << E1000_TCTL_CT_SHIFT;
  //Configure the Collision Distance (TCTL.COLD) to its expected value.
  //For full duplex operation, this value should be set to 40h.
  //For gigabit half duplex, this value should be set to 200h.
  //For 10/100 half duplex, this value should be set to 40h.--by 8254
     manual
  e100[E1000_TCTL/sizeof(uint32_t)] |= 0x40 << E1000_TCTL_COLD_SHIFT;
```

```
27
28      // Program the Transmit IPG Register
29      e100[E1000_TIPG/sizeof(uint32_t)] |= 10 << E1000_IPGT_SHIFT; // IPGR
30      e100[E1000_TIPG/sizeof(uint32_t)] |= 4 << E1000_IPGR1_SHIFT; //IPGR1
31      e100[E1000_TIPG/sizeof(uint32_t)] |= 6 << E1000_IPGR2_SHIFT; //IPGR2
32      e100[E1000_TIPG/sizeof(uint32_t)] |= 0 << E1000_RS_SHIFT;
33
34      // Initialize tx_fifo and packet buffer array
35      memset((void*)tx_fifo, 0x0, TX_SIZE * sizeof(struct tx_desc));
36     memset(tx_buffer, 0x0, TX_SIZE * MAX_PACKET_SIZE);
37      for ( i = 0; i < TX_SIZE; i++) {
38          tx_fifo[i].status |= E1000_TXD_STAT_DD;
39          tx_fifo[i].addr = PADDR(tx_buffer + i * MAX_PACKET_SIZE);
40      }
```

按照手册 PCI/PCI-X Family of Gigabit Ethernet Controllers Software Developer’ s Manual 中 section 14.5 的描述

1. 首先应该是分配一区域作为发送描述符 (transmit descriptor) 数组并且保证 16 字节对齐, 我们分配了 txi_fifo.
2. 将这个数组的地址写入到网卡发送描述符基地址寄存器 Transmit Descriptor Base Address (TDBAL/TDBAH) register(s) 中.
3. 将该数组的大小写入到 Transmit Descriptor Length (TDLEN) register. 数组大小必须是 128 字节对齐的.
4. 将 head 而和 tail 指针 (Transmit Descriptor Head and Tail (TDH/TDT) registers) 设为 0
5. 初始化发送控制寄存器 (Transmit Control Register (TCTL)), 设置 TCTL.EN 位为 1 使能发送操作, 设置短包补全 (Pad Short Packets (TCTL.PSP)) 位为 1. 设置碰撞阈值 Collision Threshold (TCTL.CT), 以太网标准是 0x10, 这个设置仅仅在半双工模式下有效. 设置碰撞距离寄存器 (Collision Distance (TCTL.COLD)), 全双工模式下是 0x40. 设置 Transmit IPG 寄存器, 因为网络包的发送是有间隔的.
6. 最后初始化发送描述符数组中各个描述符的状态域和对应的数据缓冲区地址.E1000_TXD_STAT_DD 表示这个描述符是空闲 (或者其中的数据包已经发送完成可以复用).

Exercise 6. Write a function to transmit a packet by checking that the next descriptor is free, copying the packet data into the next descriptor, and updating TDT. Make sure you handle the transmit queue being full.

具体的函数实现是 kern/e1000.c 中的 transmit_e1000 函数.

```
1  // return 0 on success
2  //  < 0 on error
3  int
4  transmit_e1000(void* src, size_t len)
```

```
{
    int next = e100[E1000_TDT/sizeof(uint32_t)];
    if (len > MAX_PACKET_SIZE)
        return -E_OUT_BUFF;
    // Check if next tx desc is free
    if (tx_fifo[next].status & E1000_TXD_STAT_DD) {
        memmove(tx_buffer + MAX_PACKET_SIZE * next, src, len);
        tx_fifo[next].length = len;

        tx_fifo[next].status &= ~E1000_TXD_STAT_DD;
        tx_fifo[next].cmd |=  E1000_TXD_CMD_RS;
        tx_fifo[next].cmd |= E1000_TXD_CMD_EOP;

        e100[E1000_TDT/sizeof(uint32_t)] = (next + 1) % TX_SIZE;
        return 0;
    } else {
        return -E_DESC_FULL;
    }
}
```

TDT 是 transmit descriptor tail, 即前面提到的 tail 指针, 它是发送描述符缓冲区的索引, 也是下一个描述符. 该函数将用户空间 src 起始的 len 个字节发送到网络上. 首先将检查 len 是否比最大的网络还大, 过大的话返回 -E_OUT_BUFF 然后测试 next 所指向的发送描述符的状态, 看能够利用这个描述符. 如果 E1000_TXD_STAT_DD 位设置, 表示可以利用. 在前面初始化的时候我们将所有的发送描述符都是设为可用的. 然后 memmove 将数据拷贝到对应的数据缓冲区. 设置发送描述符的长度域为 len. 清除该描述符的 E1000_TXD_STAT_DD 位. 设置该描述符的 RS 位 (E1000_TXD_CMD_RS), 表示需要报告该描述符的状态, 那么当网卡已经发送这个包时, 网卡会设置描述符状态域的 DD 位 (E1000_TXD_STAT_DD), 表示发送完成这个描述符可以再利用. 最后是设置该描述符的 EOP 位 (E1000_TXD_CMD_EOP) 表示是该数据包的最后一个描述符, 因为有的数据包很大会拆分对应到多个发送描述符, 但 JOS 中默认一个包只有一个描述符对应, 过大的包直接返回 E_OUT_BUFF. 最后更新 TDT, 加 1. 指向下一个描述符. 如果 TDT 对应的描述符其 DD 位没有设置, 表示不能利用这个描述符, 则返回 -E_DESC_FULL 表示发送缓存区已满.

Exercise 7. Add a system call that lets you transmit packets from user space. The exact interface is up to you. Don't forget to check any pointers passed to the kernel from user space.

具体上代码, 调用上面的 transmit_e1000 即可.

```
static int
sys_net_transmit(void *src, size_t len)
{
    user_mem_assert(curenv, src, len, PTE_P | PTE_U);
```

```
5       return transmit_e1000(src, len);
6   }
```

别忘了添加对应的系统调用号.

Implement net/output.c.

网络栈 (核心网络环境) 将数据 struct jif_pkt 放到一页上, 将这一页发送到 output 环境 (进程),output 接受这个数据然后发送到网卡.

```
1   struct jif_pkt {
2       int jp_len;
3       char jp_data[0];
4   };
```

1. jp_len 数据的长度
2. jp_data 是一个长度为 0 的数组. 这里采用了一个 C 语言的技巧. 在结构体的结尾使用一个零长的数组表示一个预先未定义大小的缓冲区. 该缓冲区中存放数据. 该缓冲区就从该结构体的结束处开始, 起始这个结构体的大小是 sizeof(int). 只有 jp_len 占了一个 int 的空间.jp_data 指向该结构的结束处. 即这一页中前面是存放这个结构体, 紧接下来存放的就是缓冲区,jp_data 指向缓冲区的基地址.

```
1   extern union Nsipc nsipcbuf;
2
3   void
4   output(envid_t ns_envid)
5   {
6       binaryname = "ns_output";
7
8       // LAB 6: Your code here:
9       //   - read a packet from the network server
10      //   - send the packet to the device driver
11      int ret;
12      envid_t envid;
13      while(1) {
14          if ((ret = ipc_recv(&envid, &nsipcbuf, NULL)) < 0)
15              panic("ipc_recv:␣%e", ret);
16          if ((envid != ns_envid) || (ret != NSREQ_OUTPUT))
17              panic("Invalid␣Request");
18
19          while ((ret = sys_net_transmit(nsipcbuf.pkt.jp_data, nsipcbuf.pkt.
            jp_len)) == -E_DESC_FULL)
20              sys_yield();
21          }
22          if (ret < 0)
23              panic("sys_net_transmit:␣%e", ret);
24      }
25  }
```

首先从网络栈中接收数据, 然后调用系统调用 sys_net_transmit 将数据发送给网卡驱动. 如果返回值是 -E_DESC_FULL 表示发送描述符缓冲区已经满了, 那么 output 唤醒就调用 sys_yield() 让出处理器. 等待一会儿.

1.6.3 Part B: Receiving packets and the web server

Exercise 10. Set up the receive queue and configure the E1000 by following the process in section 14.4. You don't have to support "long packets" or multicast. For now, don't configure the card to use interrupts; you can change that later if you decide to use receive interrupts. Also, configure the E1000 to strip the Ethernet CRC, since the grade script expects it to be stripped.
By default, the card will filter out all packets. You have to configure the Receive Address Registers (RAL and RAH) with the card's own MAC address in order to accept packets addressed to that card. You can simply hard-code QEMU's default MAC address of 52:54:00:12:34:56 (we already hard-code this in lwIP, so doing it here too doesn't make things any worse). Be very careful with the byte order; MAC addresses are written from lowest-order byte to highest-order byte, so 52:54:00:12 are the low-order 32 bits of the MAC address and 34:56 are the high-order 16 bits.
The E1000 only supports a specific set of receive buffer sizes (given in the description of RCTL.BSIZE in 13.4.22). If you make your receive packet buffers large enough and disable long packets, you won't have to worry about packets spanning multiple receive buffers. Also, remember that, just like for transmit, the receive queue and the packet buffers must be contiguous in physical memory.

接收数据包　类似发送操作, 系统也维护一个接收描述符数组 (接收描述符环形缓冲区), 接收符也有地址域, 长度, 状态等信息. 网卡中包含有接收描述符数组基地址寄存器和该数组的长度寄存器, 这两个寄存器分别用数组的基地址和长度初始化. 同时网卡中还包含发送描述符 header 和 tail 指针. 其实就是两个寄存器. 分别作为接收描述符数组的索引. 当一个数据包到达网卡时, 查看 MAC 地址是否匹配, 不匹配则丢弃数据包, 对于匹配的数据包, 将该数据包通过 DMA 方式放入到内存中. 具体的操作是获取 header 指向的接收描述符, 将该数据包放到该描述符地址域所指向的内存中. 设置该描述符的状态位表示接收完成, 然后将 header 加 1, 如果 header 和 tail 相等 (也不能说相等, 因为初始时 header 和 tail 都是 0, 更精确的说法是 if the head (RDH) has caught up with the tail (RDT), then the receive queue is out of free descriptors). 表示没有空闲的描述符了. 我们来看看接收描述符的结构.

```
1 struct rx_desc
2 {
3     uint64_t addr;
```

```
    uint16_t length;
    uint16_t checksum;
    uint8_t status;
    uint8_t errors;
    uint16_t special;
}__attribute__((__packed__));
```

1. addr 表示接收到数据包应放在内存中的位置, 也是该接收描述符对应的接收数据缓冲区.
2. length 该描述符对应的数据包的长度
3. status 表示状态信息, 例如该描述符对应的数据缓冲区中的数据是否可用?

类似发送操作, 接收操作在 kern/e1000.c 中定了发送描述符环形缓冲区 rx_fifo

```
struct rx_desc tx_fifo[rX_SIZE]__attribute__((aligned(16)));
```

该描述符缓冲区起始地址结构必须对齐到 16 字节的边界上 (可以查阅手册 PCI/PCI-X Family of Gigabit Ethernet Controllers Software Developer's Manual 中 section3.2.6). 该缓冲区中有 RX_SIZE(32, 可以自定义但必须要 8 的倍数) 个描述符. 实际数据包的存放区域是 rx_buffer

```
uint8_t *rx_buffer[RX_SIZE * BUFFER_SIZE];
```

在 JOS 中我们默认一个数据包对应一个描述符 (有的包很大可能对应多个描述符). 在接收时实际的数据缓冲区的大小是固定的, 更具初始化网卡时的配置, 可以有几种的不同的固定选择, 这里选的是 BUFFER_SIZE(2048 bytes). 所以缓冲区的大小是 RX_SIZE * BUFFER_SIZE 字节. 描述符数组中第 i 个描述符的 addr 域指向 rx_buffer+BUFFER_SIZE*i; 接下来看下初始化接收的具体代码.

```
// receive initialization

    // Program the Receive Descriptor Base Address Registers
    e100[E1000_RDBAL / sizeof(uint32_t)] = PADDR((void*)rx_fifo);
    e100[E1000_RDBAH / sizeof(uint32_t)] = 0x0;

    // Set the Receive Descriptor Length Register
    e100[E1000_RDLEN / sizeof(uint32_t)] = RX_SIZE * sizeof(struct rx_desc)
        ;

    // Set the Receive Descriptor Head and Tail Registers
    e100[E1000_RDH / sizeof(uint32_t)] = 0x0;
    e100[E1000_RDT / sizeof(uint32_t)] = 0x0;

    // configure NIC MAC Address 52:54:00:12:34:56
    // 52:54:00:12:34:56 is from lowest-order byte to highest-order
    // so 52:54:00:12 are the low-order 32 bits of the MAC address
    // and 34:56 are the high-order 16 bits
    e100[E1000_RAL/sizeof(uint32_t)] = (0x12 << 24) | (0x0 << 16) | (0x54
        << 8) | (0x52 << 0);
```

```
19      e100[E1000_RAH/sizeof(uint32_t)] = (0x56 << 8) | (0x34 << 0);
20
21
22    //address valid
23      e100[E1000_RAH/sizeof(uint32_t)] |= E1000_RAH_AV;
24
25      //initialize Multicast Table Array to 0b, total 4096 bit in MTA.
26      memset(((uint8_t*)e100) + E1000_MTA, 0, 4096/8);
27
28      //mask all interrupts
29      //e100[E1000_IMC/sizeof(uint32_t)] |= 0xFFFFFFFF;
30
31
32      // Buffer Size 2048, because largest possible standard Ethernet
33      // packet (1518 bytes), we use one descriptor  for one packet.
34      e100[E1000_RCTL/sizeof(uint32_t)] = 0;
35      e100[E1000_RCTL/sizeof(uint32_t)] |= E1000_RCTL_SECRC; /* Strip
            Ethernet CRC */
36      e100[E1000_RCTL/sizeof(uint32_t)] |= E1000_RCTL_SZ_2048;
37
38      // Initialize rx_fifo and packet buffer array
39      memset((void*)rx_fifo, 0x0, RX_SIZE * sizeof(struct tx_desc));
40      memset(rx_buffer, 0x0, RX_SIZE * BUFFER_SIZE);
41      for (i = 0; i < RX_SIZE; i++) {
42          rx_fifo[i].addr = PADDR(rx_buffer + i * BUFFER_SIZE);
43      }
44
45     //Initialize the Receive Control Register
46      e100[E1000_RCTL/sizeof(uint32_t)] &= ~E1000_RCTL_LPE; // Long Packet
            Reception disable
47      e100[E1000_RCTL/sizeof(uint32_t)] |= E1000_RCTL_BAM; // broadcast
            enable.
48      //Loopback Mode (RCTL.LBM) should be set to 00b for normal operation.
49      e100[E1000_RCTL/sizeof(uint32_t)] &= ~E1000_RCTL_LBM_NO;
50      e100[E1000_RCTL/sizeof(uint32_t)] &= ~E1000_RCTL_MPE; // multicast
            promiscuous disable
51      e100[E1000_RCTL/sizeof(uint32_t)] &= ~E1000_RCTL_BSEX; // Buffer size
            extension
52      /* broadcast enable */
53      //it is best to leave the Ethernet controller receive logic disabled (
            RCTL.EN = 0b)
54      //until after the receive descriptor ring has been initialized and
            software is ready
55      //to process received packets --by 8254x manual
56      e100[E1000_RCTL/sizeof(uint32_t)] |= E1000_RCTL_EN;  //enable receiver
```

按照手册 PCI/PCI-X Family of Gigabit Ethernet Controllers Software Developer’s Manual 中 section 14.4 中的描述

1. 设置正确的 MAC 地址, 将该地址写入到接收地址寄存器 (Receive Address Register(s) (RAL/RAH)) JOS 中给的 MAC 地址是 52:54:00:12:34:56. 注意 MAC 地址的顺序, 注释中已经给出.
2. 首先应该是分配一区域作为接收描述符 (Receive Descriptor) 数组并且保

证 16 字节对齐, 我们分配了 rx_fifo.

3. 将这个数组的基地址写入到网卡接收描述符基地址寄存器 Receive Descriptor Base Address (RDBAL/RDBAH) register(s) 中.
4. 将该数组的大小写入到 Receive Descriptor Length (RDLEN) register. 数组大小必须是 128 字节对齐的.
5. 将 head 和 tail 指针 (Receive Descriptor Head and Tail registers ((RDH/RDT)) 设为 0
6. 初始化接收控制寄存器 (Receive Control (RCTL) register), 包括设置除去 CRC 校验 (因为 JOS 成绩测试脚本中除去了 CRC). 设置每个接收数据缓冲区大小为 2048 字节. 具体可以查看手册和上面的代码注释.
7. 初始化接收描述符数组中各个描述符对应的数据缓冲区地址. 注意在上面的代码中描述符对应的状态域全部在 memset 中清 0 了.
8. 设置接收控制寄存器 (Receive Control (RCTL) register) 的使能位, 使能接收. 对于这个操作, 手册上说最好最后做.

Exercise 11. Write a function to receive a packet from the E1000 and expose it to user space by adding a system call. Make sure you handle the receive queue being empty.
If you decide to use interrupts to detect when packets are received, you'll need to write code to handle these interrupts. If you do use interrupts, note that, once an interrupt is asserted, it will remain asserted until the driver clears the interrupt. In your interrupt handler make sure to clear the interrupt handled as soon as you handle it. If you don't, after returning from your interrupt handler, the CPU will jump back into it again. In addition to clearing the interrupts on the E1000 card, interrupts also need to be cleared on the LAPIC. Use lapic_eoi to do so.

这里的核心是如何检测 tail 指向的描述符对应的缓冲区中的数据是可用的. 接收描述符状态域中 DD(E1000_RXD_STAT_DD) 位为 1, 那么该数据就可用, 可以拷贝到应用程序的空间中. 如果 DD 位是 0, 则对应的数据还没有到内存中, 数据还不可用.

```
int
receive_e1000(void* dst)
{
    int len;
    int tail = e100[E1000_RDT/sizeof(uint32_t)];
    if (rx_fifo[tail].status & E1000_RXD_STAT_DD) {
            len = rx_fifo[tail].length;
            memmove(dst, rx_buffer + BUFFER_SIZE * tail, len);
            rx_fifo[tail].status &= ~E1000_RXD_STAT_DD;
            rx_fifo[tail].status &= ~E1000_RXD_STAT_EOP;
            e100[E1000_RDT/sizeof(uint32_t)] = (tail + 1) % RX_SIZE;
            return len;
```

```
    } else {
        return -E_DESC_EMPTY;
    }
}
```

首先检查数据包是否已经传输到内存, 是否可用. 然后将这个数据拷贝到用户空间 dst 中. 清除相应的状态位,tail 加 1, 返回值是数据包的长度. 接收描述符中的长度域是由硬件设置的. 如果接收描述符数组空的话就返回 -E_DESC_EMPTY. 然后添加对应的系统调用. 别忘了添加系统调用号

```
static int
sys_net_receive(void* dst)
{
    if((uintptr_t)dst >= UTOP)
        return -E_INVAL;
    return receive_e1000(dst);
}
```

该调用接收一个数据包, 存放到用户空间的 dst 处.

Exercise 12. Implement net/input.c.

input 环境 (进程) 负责调用前面的 sys_net_receive 接收数据, 然后利用 IPC 发送给网络栈.

```
extern union Nsipc nsipcbuf;

void
input(envid_t ns_envid)
{
    binaryname = "ns_input";

    // LAB 6: Your code here:
    //   - read a packet from the device driver
    //   - send it to the network server
    // Hint: When you IPC a page to the network server, it will be
    // reading from it for a while, so don't immediately receive
    // another packet in to the same physical page.
    int ret;
    int perm = PTE_P | PTE_W | PTE_U;
    uint8_t buf[BUFFER_SIZE];
    while(1) {
        if ((ret = sys_page_alloc(0, &nsipcbuf, perm)) < 0)
            panic("sys_pape_alloc:␣%e", ret);

        while ((ret = sys_net_receive(buf)) == -E_DESC_EMPTY) {
            sys_yield();
        }
        memmove(nsipcbuf.pkt.jp_data, buf, ret);
        nsipcbuf.pkt.jp_len = ret;
        ipc_send(ns_envid, NSREQ_INPUT, &nsipcbuf, perm);
```

```
27      }
28  }
```

调用 sys_net_receive 接收数据到 buf 中. 为 nsipcbuf 分配一页. 将 buf 中的数据拷贝到该页中. 然后调用 ipc_send 将该页发送给网络栈 (核心网络环境). 当接收描述符数组空的话, 那么调用 sys _yield 让出处理器, 等待描述符数组中有数据, 这种处理方式不是太好. 实验指导中给出了说明.

> You can simply return a "try again" error and require the caller to retry. While this approach works well for full transmit queues because that's a transient condition, it is less justifiable for empty receive queues because the receive queue may remain empty for long stretches of time.

实验指导中也给出了另一种方法

> A second approach is to suspend the calling environment until there are packets in the receive queue to process. This tactic is very similar to sys_ipc_recv. Just like in the IPC case, since we have only one kernel stack per CPU, as soon as we leave the kernel the state on the stack will be lost. We need to set a flag indicating that an environment has been suspended by receive queue underflow and record the system call arguments. The drawback of this approach is complexity: the E1000 must be instructed to generate receive interrupts and the driver must handle them in order to resume the environment blocked waiting for a packet.

这里的意思就是如果数据包还未到的话就将当前进程的状态设为不可运行 (参考 sys_ipc_recv), 因为 jos 每个 cpu 对应一个内核栈 (xv6 是每个进程一个内核栈), 所以当前的接收进程让出处理器后意味着接收系统调用的参数信息也将丢失. 所以需要设置一个标志表示该进程是因为数据包未到而挂起, 同时要保存参数信息 (如用于接收的用户空间缓冲区), 我们看到 sys_ipc_recv 也是这样处理的

```
1   curenv->env_ipc_dstva = dstva;
2   curenv->env_ipc_recving = 1;
3   curenv->env_status = ENV_NOT_RUNNABLE;
4  sched_yield();
```

首先保存参数信息 dstva, 然后设置一个标志表示要接收数据, 然后将进程状态设为不可运行, 让出处理器. 我们网络包的接收也可以采用类似的方式. 当一个网络包来后产生一个中断, 然后恢复接收进程的执行就可以了.

Exercise 13. The web server is missing the code that deals with sending the contents of a file back to the client. Finish the web server by implementing send_file and send_data.

该 http 服务器将一个客户端请求的文件返回给客户端, 首先 http 服务需要访问文件服务, 读取文件, 然后将文件的内容放到 http 响应报文的响应体中 (http 响应报文分三个部分. 响应行, 响应头, 响应体. 具体信息请 google). 然后将该 http 报文发送给客户端. send_file 的部分实现如下

```
    if ((r = send_header(req, 200)) < 0)
        goto end;

    if ((r = send_size(req, file_size)) < 0)
        goto end;

    if ((r = send_content_type(req)) < 0)
        goto end;

    if ((r = send_header_fin(req)) < 0)
        goto end;

    r = send_data(req, fd);
```

send_data 的实现如下

```
static int
send_data(struct http_request *req, int fd)
{
    // LAB 6: Your code here.
    int r, len,file_size;
    struct Stat stat;
    uint8_t *buf;

    if ((r = fstat(fd, &stat)) < 0) {
        die("Failed to stat");
    }
    file_size = stat.st_size;

    buf = (uint8_t*) malloc(file_size * sizeof(uint8_t));
    if(!buf) {
        die("Failed to malloc");
    }

    if ((len = read(fd, buf, file_size)) != file_size) {
        die("Failed to read entire file");
    }

    if (write(req->sock, buf, len) != len) {
        die("Failed to send file content");
    }
    return 0;
```

```
27 //  panic("send_data not implemented");
28 }
```

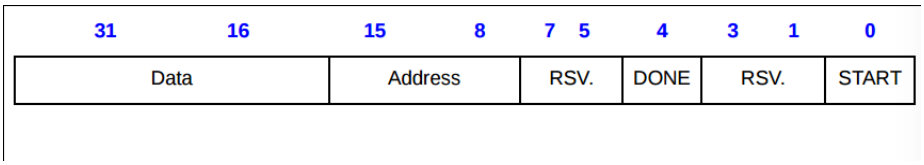
这部分的理解参考前面关于文件系统的介绍. 文件的设备层添加了关于套接字的设备. 测试这个 http 服务的步骤如下. 首先在终端中输入 make run-httpd. 然后在浏览器中输入 http://host:port/index.html. 其中 host 是主机名, 如果 qemu 在运行在本机上, 那么直接用 localhost 就行了. port 是端口号, 通过 make which-ports 命令可以得到. 在浏览器中输入 http://localhost:26002/index.html, 就可以得到如下的页面

图 25: http server

Challenge!

Challenge! Read about the EEPROM in the developer's manual and write the code to load the E1000's MAC address out of the EEPROM. Currently, QEMU's default MAC address is hard-coded into both your receive initialization and lwIP. Fix your initialization to use the MAC address you read from the EEPROM, add a system call to pass the MAC address to lwIP, and modify lwIP to the MAC address read from the card. Test your change by configuring QEMU to use a different MAC address.

JOS 中 e10000 的 MAC 是 hard-coded 的, 但是我们也可以在该网卡的 EEPROM 中设置好 MAC 地址. 然后从其中读出来即可. 查看手册 PCI/PCI-X Family of Gigabit Ethernet Controllers Software Developer's Manual 中的 section5.6.1. EEPROM 区域前 6 个字节是存储了 MAC 地址. 接收地址寄存器 (Receive Address Register 0 (RAL0/RAH0)) 就是利用该 MAC 地址初始化. 然后我们关注的重点就是如何读出这 6 个字节的 MAC 地址. 我们看下 section13.4.4 中介绍的 EEPROM Read Register.



31 16	15 8	7 5	4	3 1	0
Data	Address	RSV.	DONE	RSV.	START

图 26: EEPROM Read Register

当要读 EPPROM 区域时, 首先要向该寄存器 address 字段写入地址.MAC 地址是 EPPRROM 区域的前 6 个字节. 地址是 0x00,0x01,0x02. 得到的数据是在该寄存器的 data 字段中. 下面的是该寄存器中各个字段的解释

Field	Bit(s)	Initial Value	Description
START	0	0b	Start Read Writing a 1b to this bit causes the EEPROM to read a (16-bit) word at the address stored in the EE_ADDR field and then storing the result in the EE_DATA field. This bit is self-clearing.
Reserved	3:1	0b	Reserved. Reads as 0b.
DONE	4	0b	Read Done Set to 1b when the EEPROM read completes. Set to 0b when the EEPROM read is in progress. Writes by software are ignored.
Reserved	7:5	0b	Reserved. Reads as 0b.
ADDR	15:8	X	Read Address This field is written by software along with *Start Read* to indicate the word to read.
DATA	31:16	X	Read Data. Data returned from the EEPROM read.

图 27: EEPROM Read Register

然后我们来看下具体的代码.

```
// Challenge. read mac address from EEPROM
   // first 16-bit
   e100[E1000_EERD/sizeof(uint32_t)] = 0x0;
   e100[E1000_EERD/sizeof(uint32_t)] |= E1000_EEPROM_RW_REG_START;
   while(!(e100[E1000_EERD/sizeof(uint32_t)] & E1000_EEPROM_RW_REG_DONE));
   e100[E1000_RAL/sizeof(uint32_t)] =
       e100[E1000_EERD/sizeof(uint32_t)] >> E1000_EEPROM_RW_REG_DATA;
   // second 16-bit
   e100[E1000_EERD/sizeof(uint32_t)] = 0x1 << E1000_EEPROM_RW_ADDR_SHIFT;
   e100[E1000_EERD/sizeof(uint32_t)] |= E1000_EEPROM_RW_REG_START;
   while(!(e100[E1000_EERD/sizeof(uint32_t)] & E1000_EEPROM_RW_REG_DONE));
   e100[E1000_RAL/sizeof(uint32_t)] |= e100[E1000_EERD/sizeof(uint32_t)] &
       0xffff0000;
   // third 16-bit
   e100[E1000_EERD/sizeof(uint32_t)] = 0x2 << E1000_EEPROM_RW_ADDR_SHIFT;
   e100[E1000_EERD/sizeof(uint32_t)] |= E1000_EEPROM_RW_REG_START;
```

```
16      while(!(e100[E1000_EERD/sizeof(uint32_t)] & E1000_EEPROM_RW_REG_DONE));
17      e100[E1000_RAH/sizeof(uint32_t)] = e100[E1000_EERD/sizeof(uint32_t)] >>
            E1000_EEPROM_RW_REG_DATA;
18      //end challenge
```

首先分别写入地址 0x00,0x01,0x02 到 EEPROM Read Register 中. 然后检查读是否完成, 然后再从 EEPROM Read Register 的 data 字段中读出数据. 接下来把读出的 MAC 地址写入到接收地址寄存器 (Receive Address Register 0 (RAL0/RAH0)), 这个和原来一样, 就没列出了. 接下来我们添加了一个系统调用 MAC 地址.

```
1  static int
2  sys_get_mac(uint32_t *low, uint32_t *high)
3  {
4      user_mem_assert(curenv, low, sizeof(uint32_t), PTE_P | PTE_U);
5      user_mem_assert(curenv, high, sizeof(uint32_t), PTE_P | PTE_U);
6      *low = e100[E1000_RAL/sizeof(uint32_t)];
7      *high = e100[E1000_RAH/sizeof(uint32_t)] & 0x0000ffff;
8      return 0;
9  }
```

其中参数 low 存放 MAC 地址的低 32 位,high 存放高 16 位. 这个 MAC 地址从接收地址寄存器 (Receive Address Register 0 (RAL0/RAH0)) 中读出即可. 因为 EEPROM 中的 MAC 地址已经读入到接收地址寄存器 (Receive Address Register 0 (RAL0/RAH0)) 中. 为了测试这个 Challenge, 还需要做一些修改,net/lwip/jos/jif/jif.c 中 low_level_init 函数需要读取 MAC 地址保存下来. 现在不在使用 hard-coded. 而是调用上面提供的系统调用读取地址. 为了测试新的地址, 我们给 e1000 配一个新的 mac 地址. 修改 GNUMakefile. 原来是

```
QEMUOPTS += -net user -net nic,model=e1000 -redir tcp:$(PORT7)::7  -redir
tcp:$(PORT80)::80 -redir udp:$(PORT7)::7 -net dump,file=qemu.pcap
```

修改为

```
QEMUOPTS += -net user -net nic,macaddr=52:54:00:12:34:60,model=e1000
-redir tcp:$(PORT7)::7  -redir tcp:$(PORT80)::80 -redir udp:$(PORT7)::7 -net
dump,file=qemu.pcap
```

这里给新配了一个 MAC 地址 52:54:00:12:34:60. 为了测试通过 testinput,net/testinput.c 中对应的 mac 地址也需要修改为

```
1  uint8_t mac[6] = {0x52, 0x54, 0x00, 0x12, 0x34, 0x60};
```

最后 make grade 通过所有测试.

1.7 Lab 7: Final Project

本次实验主要是实现父子进程共享打开文件表. 系统依赖父子进程之间共享文件表来实现 shell. 下面依次介绍 unix-like 系统文件和进程之间的关系,shell 的基本实现思路, 以及进程的 spawn. JOS 的文件系统的实现虽然和 unix-like 系统文件系统实现有区别, 但是一些语义是一样的.

1.7.1 UNIX-LIKE File system

这部分主要介绍文件系统的一些调用和与进程的交互关系. 在系统内部使用 inode 来表示一个文件.inode 一般包含文件的数据存放地址 (磁盘块号), 文件类型 (目录或者一般文件), 访问权限, 访问时间等等. 每个文件都有一个 inode. 但是一个 inode 可以有多个文件名 (如硬链接存在的情形). 内核同时维护其他两个数据结构. 文件表 (file table) 和用户文件描述符表 (user file descriptor table). 文件表是一个全局的内核数据结构. 用户文件描述符表是一个每个进程私有的数据结构. 当进程打开一个文件, 内核为每个表都创建一项, 三个数据结构 (file table, user file descriptor table, inode table) 中的一项维护了一个访问. 文件表 (file table) 维护了读写偏移和进程的访问权限, 用户文件描述符表 (user file descriptor table) 记录了该进程所有的打开文件, 该表的偏移就是文件描述符号. 当进程使用文件描述符作为参数调用 read,write 等系统调用时, 首先使用文件描述符作为索引在 user file descriptor table 查找对应的项, 然后找到 file table 中对应的项, 取出读写偏移和访问权限等. 最后找到 inode 进行读写. 关系图如下.

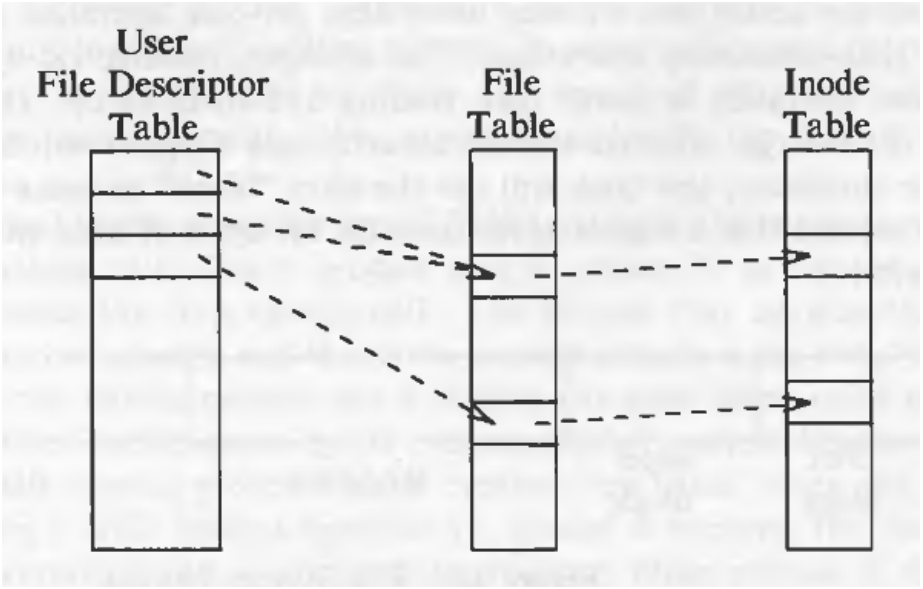


图 28: File Descriptors, File Table, and Inode Table

每打开一个文件, 就会在描述符表和文件表中各分配一项. 例如当前进程调用如下的系统调用

```
fdl = open ("etc/passwd", O_RDONLY);
fd2 = open ("local", O_RDWR);
fd3 = open ("/etc/passwd", O_WRONLY);
```

其中打开文件/etc/passwd 两次, 返回两个不同的文件描述符号 3 和 5(0,1,2 是标准输入, 输出, 错误) , 我们看到在同一个进程中, 同一个文件打开两次, 返回的文件描述符都是不同的, 读写偏移也不是共享的. 读写权限存储在文件表 (file table) 中, 也不共享. 打开文件 local 一次返回文件描述符号 4. 如下图所示.

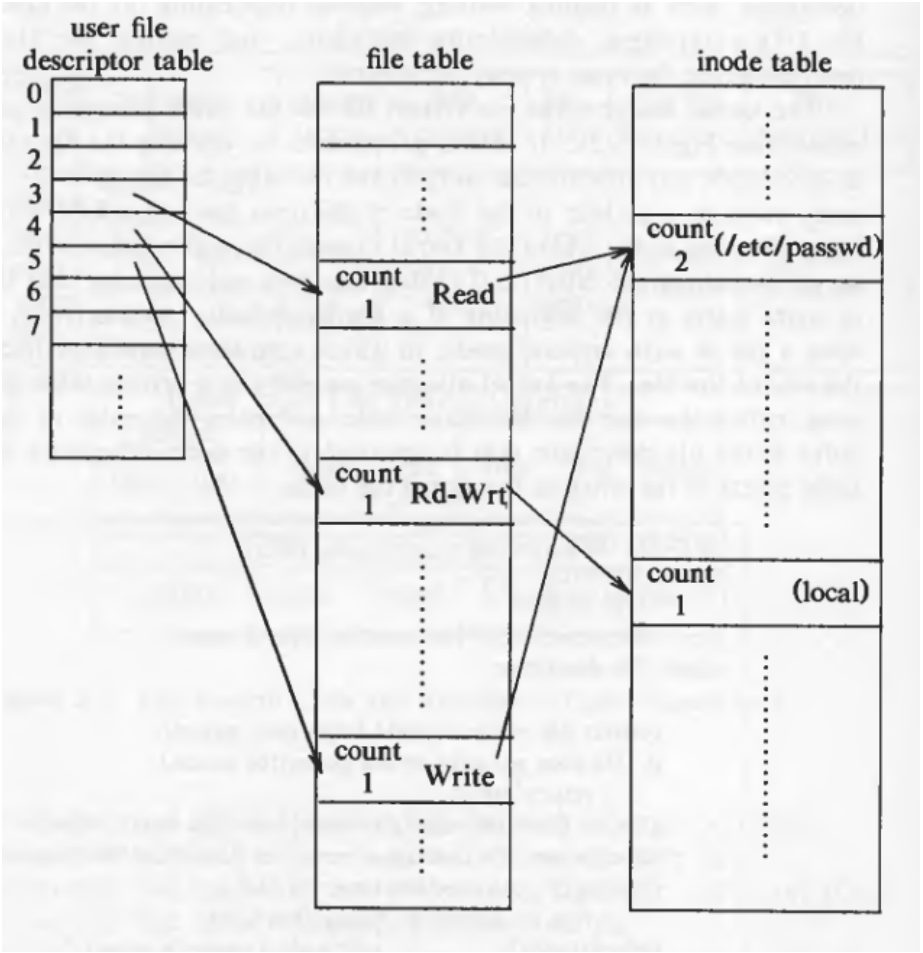


图 29: Data Structures after Open

如果另外一个进程下面的代码, 也打开文件

```
fdl = open ("/etc/passwd", O_RDONLY);
fd2 = open ("private", O_RDONLY);
```

在另外一个进程中页打开了/etc/passwd 文件. 关系图如下:

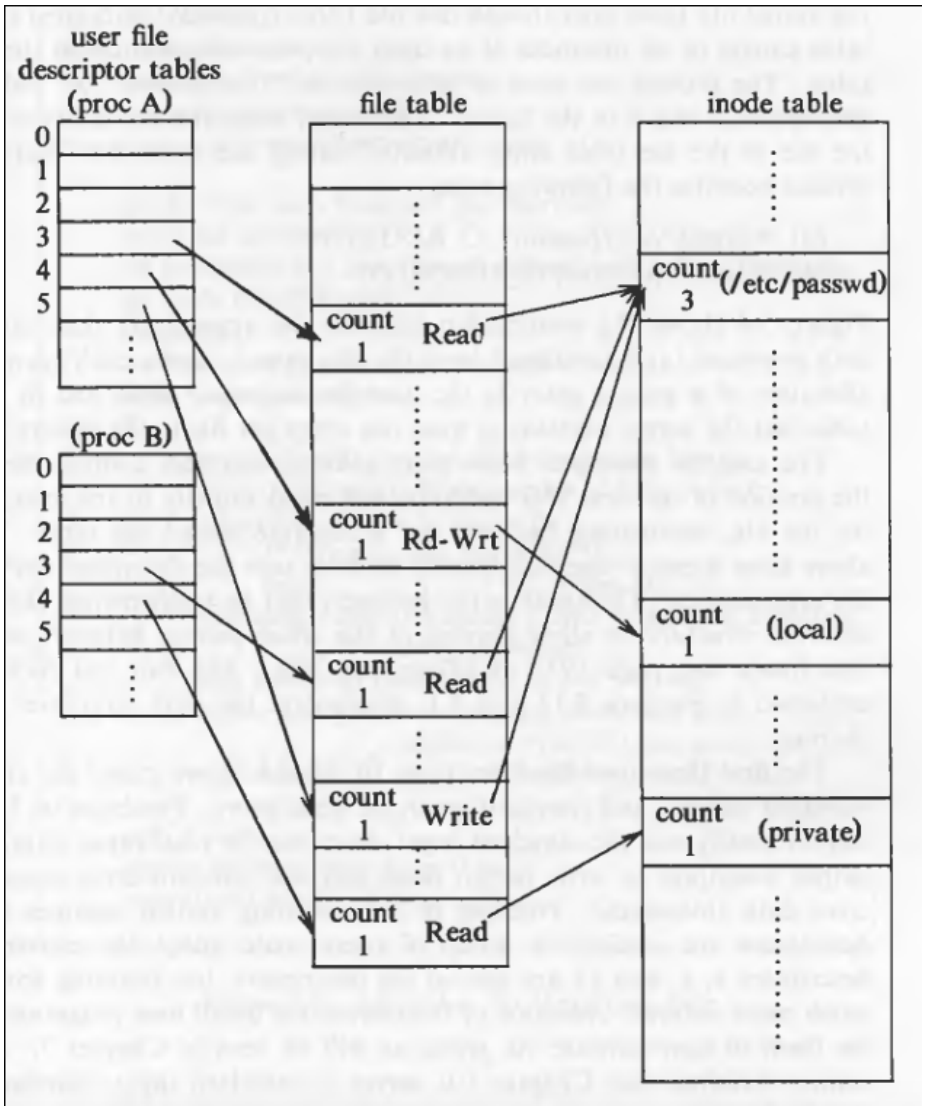


图 30: Data Structures after Two Processes Open Files

如图所示, 进程 B 中打开文件/etc/passwd, 返回的描述号是 3. 文件描述符的分配规则是:" 从头开始搜索文件描述符表, 找到第一个未使用的表项, 将该表项标记为已使用, 返回该表项对应的索引, 这个索引就是文件描述符号". 如果关闭文件, 则将描述符表和文件表 (file table) 中对应的项释放 (计数减 1, 减到 0 则释放). 如果进程 B 执行关闭操作:

```
1 close(fd1);
2 close(fd2);
```

那么得到的关系图如下.

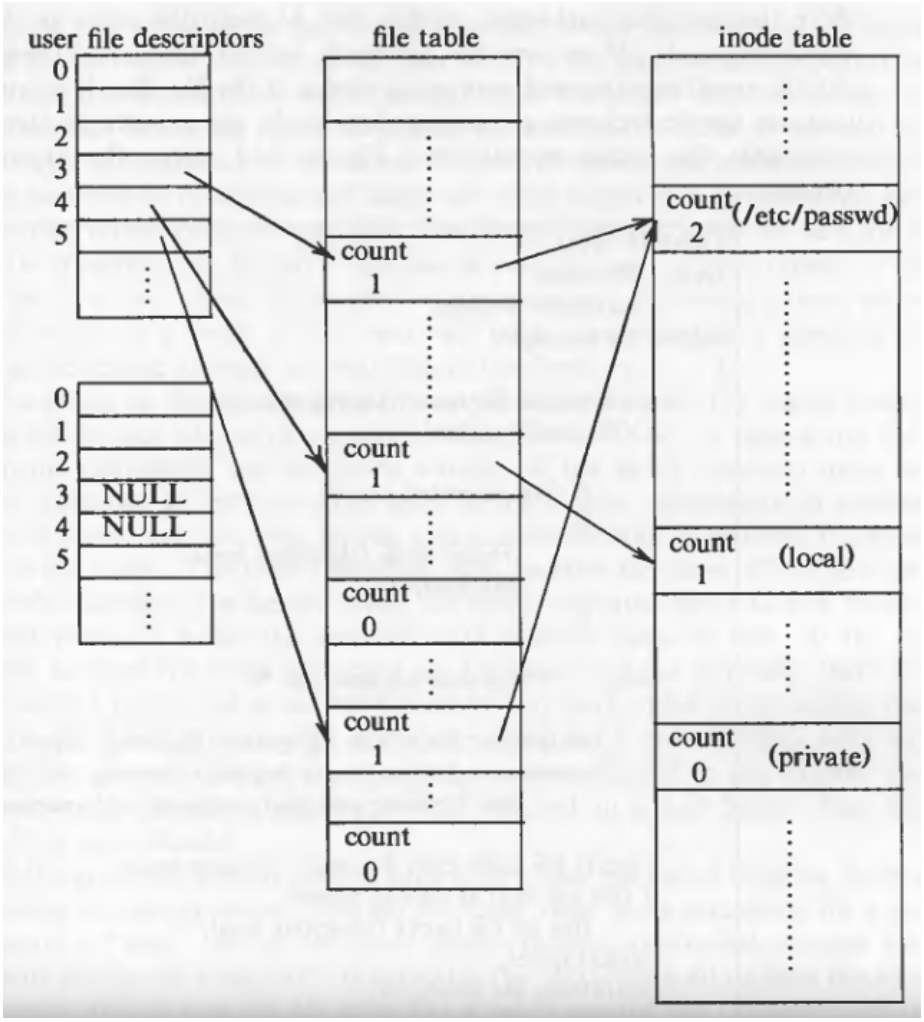


图 31: Tables after Closing a File

其中 fd1 和 fd2 对应的项都被释放, 而文件表 (file table) 中只有计数减到 0 的才释放. 如果我们 dup 一个文件. 则只是在文件描述符表中新分配一个一项, 并返回新的文件描述符号. 例如我们调用

```
1 newfd = dup(fd)
```

newfd 和 fd 现在都是表示同一个文件, 他们在文件描述符表中的项都指向同一个在文件表 (file table) 中的项. 例如执行下面的代码

```
1 fdl = open ("etc/passwd", O_RDONLY);
2 fd2 = open ("local", O_RDWR);
3 fd3 = open ("/etc/passwd", O_WRONLY);
```

```
4 fd4 = dup(fd1);
```

得到的关系图如下.

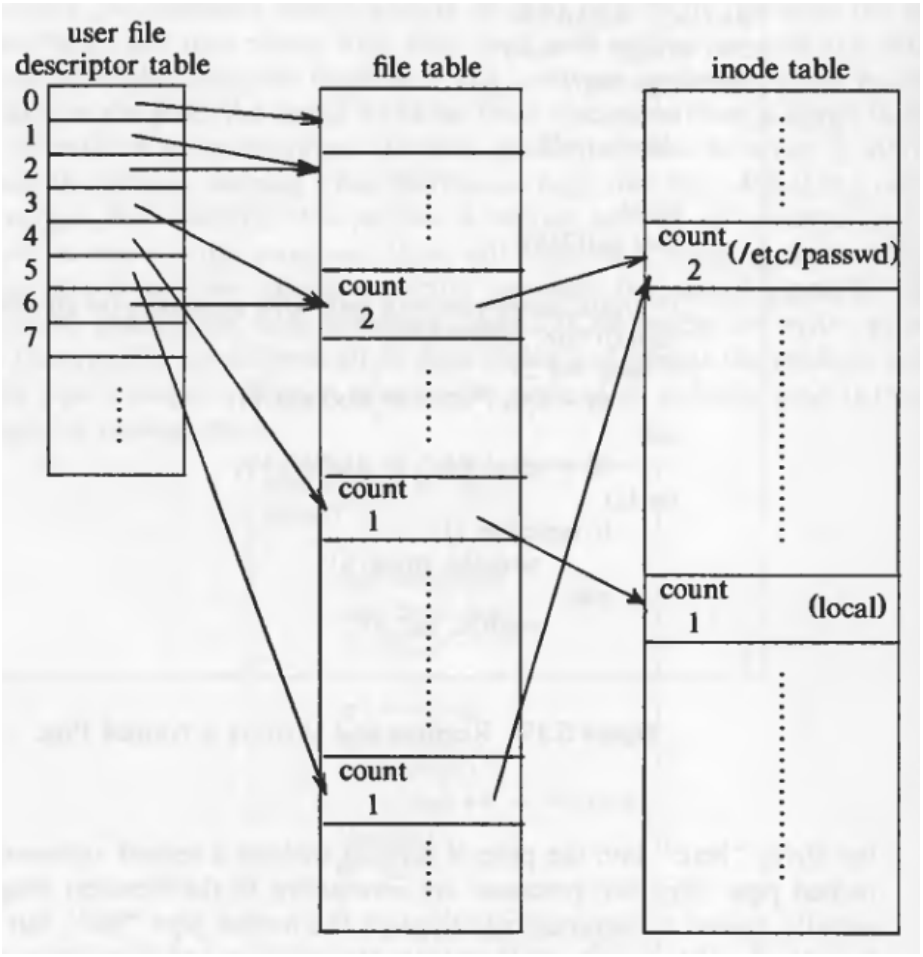


图 32: Data Structures after Dup

从中可以看到文件描述符 3 和 6 指向文件表 (file table) 文件表中的一项. 进程在 fork 子进程后, 父子进程共享打开的文件, 读写偏移, 访问权限, 这是因为子进程的打开文件表是父进程的拷贝. 他们都指向相同的文件表 (file table) 项如下图.

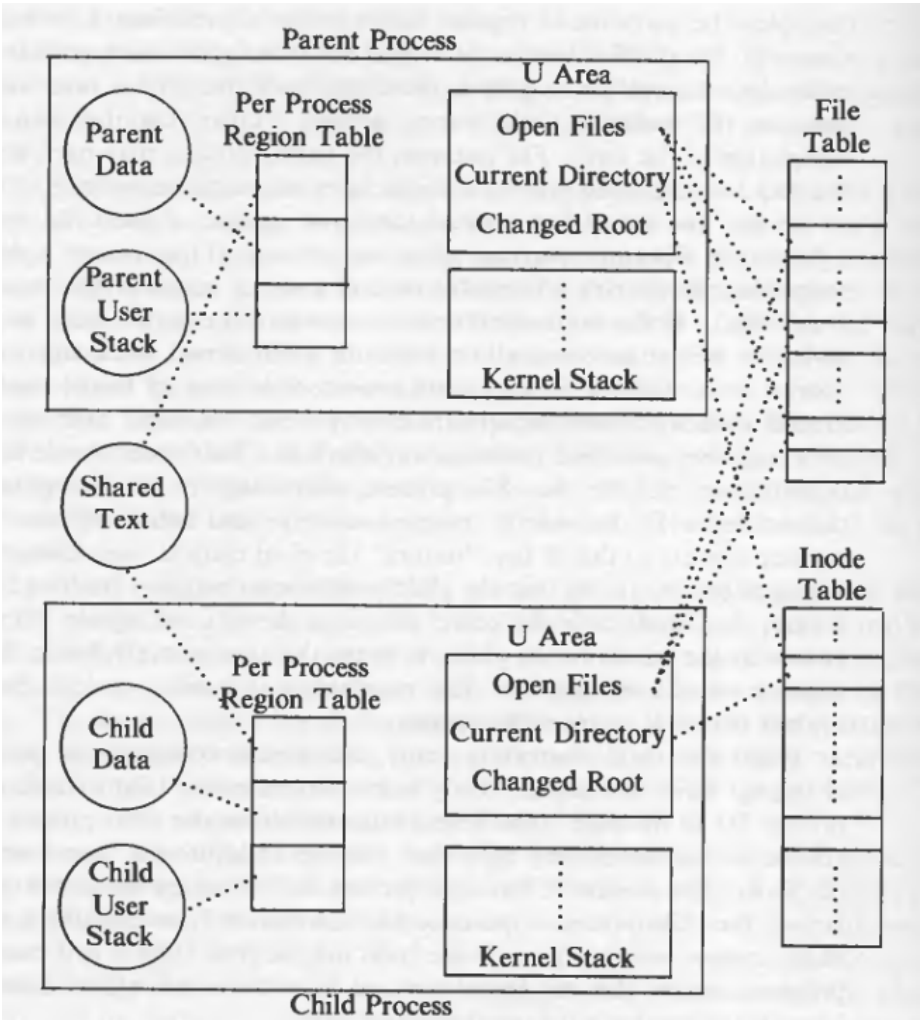


图 33: Data Structures after Dup

1.7.2 JOS Shell Implemention

shell 作为一个用户程序, 在 user/sh.c 中做了实现. 在 umain 函数中, 调用 readline 读取一整行的 shell 命令. 然后 fork 一个子进程. 在子进程中调用 runcmd 函数依次处理读入的命令, 主进程调用 wait 等待该子进程. 在 runcmd 为每个命令创建一个进程进行处理. 例如输入 ls -l | cat | num. 那么会创建 3 个进程分别处理 ls,cat, num 三个命令. 下面介绍重定向 (<,>),pipe 的实现. 如果命令之间使用 pipe 进行通信, 基本的框架是这样的.

```
1       int p[2];
2       pipe(p);
3             r = fork()
4             if (r == 0) {
```

```
5                   if (p[0] != 0) {
6                       dup(p[0], 0);
7                       close(p[0]);
8                   }
9                   close(p[1]);
10                  goto again;
11              } else {
12                  pipe_child = r;
13                  if (p[1] != 1) {
14                      dup(p[1], 1);
15                      close(p[1]);
16                  }
17                  close(p[0]);
18                  goto runit;
19              }
```

首先创建一个子进程, 在子进程中将文件描述符 0 指向到 p[0] 所指的文件, 然后关闭 p[0],[1], 子进程如果进行读操作会从文件描述符 0 中读入. 但是 0 现在不在是标准输入了, 而是一个 pipe 的读端口. 对应的主进程中需要修改标准输入文件描述符 1. 将 1 指向 p[1] 所指的文件. 然后关闭 p[0],p[1]. 那么主进程写时会往文件描述符 1 写, 但是 1 现在不是标准输出了, 而是一个 pipe 的写端口. 例如 ls -l | cat | num, 进程的关系图如下.

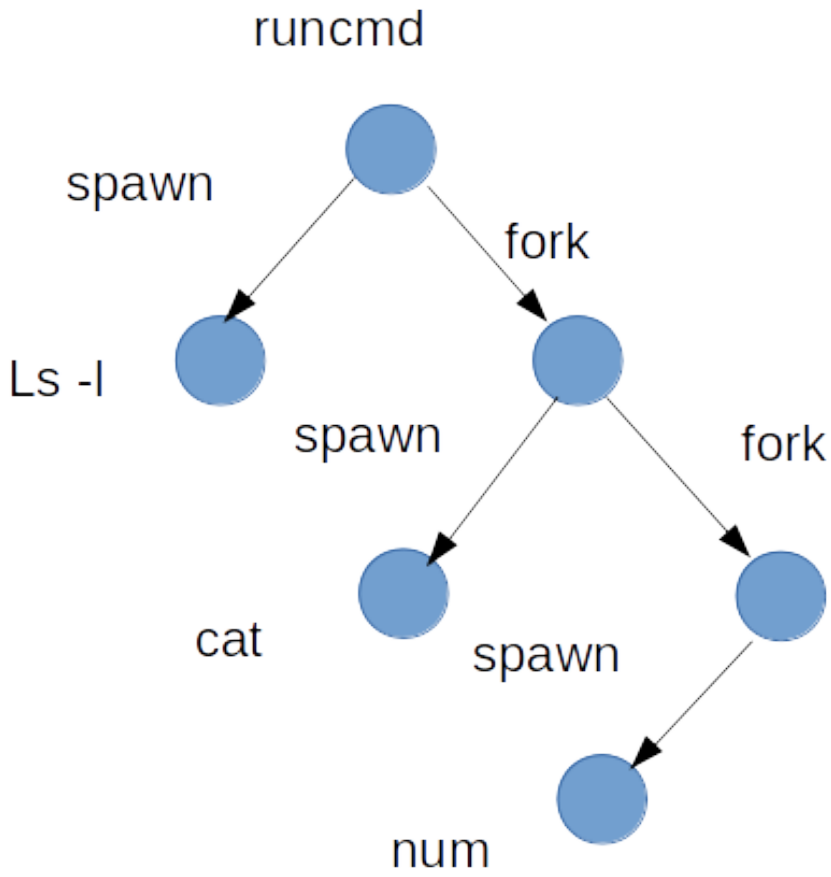


图 34: ls -l | cat | num

因为前面提到 fork 或者 spawn 子进程, 父子进程都会共享打开的文件, 所以设置好的 pipe 关系, 在执行命令的子进程中被保存.
shell 中另一个常用的操作是重定向. 下面是一个基本的框架.

```
case '<':   // Input redirection

        if (gettoken(0, &t) != 'w') {
            cprintf("syntax␣error:␣<␣not␣followed␣by␣word\n");
            exit();
        }
        if ((fd = open(t, O_RDONLY)) < 0) {
            cprintf("open␣%s␣for␣read:␣%e", t, fd);
            exit();
        }
        if (fd != 0) {
            dup(fd, 0);
            close(fd);
        }
```

```
            break;

        case '>':   // Output redirection
            // Grab the filename from the argument list
            if (gettoken(0, &t) != 'w') {
                cprintf("syntax␣error:␣>␣not␣followed␣by␣word\n");
                exit();
            }
            if ((fd = open(t, O_WRONLY|O_CREAT|O_TRUNC)) < 0) {
                cprintf("open␣%s␣for␣write:␣%e", t, fd);
                exit();
            }
            if (fd != 1) {
                dup(fd, 1);
                close(fd);
            }
            break;
```

对于输入重定向 <, 取 < 后面的参数. 如 cat < readme.txt. 那么 readme.txt 就是参数. 并且认为该参数是一个文件, 调用 open 以读的方式打开该文件, 然后将标准输入 0 指向该文件. 这样从 0 读取输入, 就是重定向到从该文件中读取数据. 输出重定向类似. 只是以写的方式打开该文件, 然后将标准输出 1 指向该文件, 这样从 1 输出数据, 就是重定向输出到该文件中.

1.7.3 JOS Pipe Implemention

pipe 可以通过文件描述符进行访问, 为了实现文件描述符到 pipe 的映射, 在设备存又添加了一种新的设备: devpipe.

```
struct Dev devpipe =
{
    .dev_id =   'p',
    .dev_name = "pipe",
    .dev_read = devpipe_read,
    .dev_write =    devpipe_write,
    .dev_close =    devpipe_close,
    .dev_stat = devpipe_stat,
};
```

该设备的读写函数是

```
.dev_read = devpipe_read,
.dev_write =    devpipe_write,
```

所以调用 read,write 调用读写 pipe 时最后会调用该设备的

devpipe_read,devpipe_write 进行读写. 我们看下 pipe 的结构.

```
#define PIPEBUFSIZ 32
struct Pipe {
    off_t p_rpos;       // read position
    off_t p_wpos;       // write position
```

```
    uint8_t p_buf[PIPEBUFSIZ];  // data buffer
};
```

其中包含读偏移和写偏移, 以及存放数据缓冲区.pipe(int p[2]) 函数建立 pipe 过程如下

1. 分配两个文件描述符:fd0,fd1 作为读写描述符. 并分配对应的 fd 页.
2. 分配一个 fd 数据页. 将 fd0,fd1 对应的描述符数据页虚地址映射到该页. lab5 中有文件描述符数据页介绍.
3. 将 fd0,fd1 所以对应的设备设置位 devpipe. 权限分别设为读和写. 这样读写操作就会调用设备相关的读写函数进行. 其中 struct pipe 结构就是存放在文件描述符数据页上.

我们来看看 pipe 读的基本过程

```
for (i = 0; i < n; i++) {
       while (p->p_rpos == p->p_wpos) {
           // pipe is empty
           // if we got any data, return it
           if (i > 0)
               return i;
           // if all the writers are gone, note eof
           if (_pipeisclosed(fd, p))
               return 0;
           // yield and see what happens
           if (debug)
               cprintf("devpipe_read␣yield\n");
           sys_yield();
       }
       // there's a byte.  take it.
       // wait to increment rpos until the byte is taken!
       buf[i] = p->p_buf[p->p_rpos % PIPEBUFSIZ];
       p->p_rpos++;
   }
```

如果缓冲区不为空, 那么读出数据, 更新读计数. 如果缓冲区读的过程中满了, 则返回已经读到的字符个数. 如果已经没有写者了, 那么返回 0. 否则让出处理器. 等待 pipe 缓冲区中有数据.

下面是 pipe 写的基本过程

```
for (i = 0; i < n; i++) {
       while (p->p_wpos >= p->p_rpos + sizeof(p->p_buf)) {
           // pipe is full
           // if all the readers are gone
           // (it's only writers like us now),
           // note eof
           if (_pipeisclosed(fd, p))
               return 0;
           // yield and see what happens
           if (debug)
               cprintf("devpipe_write␣yield\n");
           sys_yield();
```

```
        }
        // there's room for a byte.  store it.
        // wait to increment wpos until the byte is stored!
        p->p_buf[p->p_wpos % PIPEBUFSIZ] = buf[i];
        p->p_wpos++;
    }
```

如果 pipe 缓冲区没有满则直接写入数据, 并更新写计数. 如果缓冲区满, 并且现在没有读者了. 那么返回 0. 否则让出处理器, 等待缓冲区有空间.
在读和写操作中都有一个检查是否还有读或者写者的函数 _pipeisclosed. 这个函数如下:

```
static int
_pipeisclosed(struct Fd *fd, struct Pipe *p)
{
    int n, nn, ret;

    while (1) {
        n = thisenv->env_runs;
        ret = pageref(fd) == pageref(p);
        nn = thisenv->env_runs;
        if (n == nn)
            return ret;
        if (n != nn && ret == 1)
            cprintf("pipe␣race␣avoided\n", n, thisenv->env_runs, ret);
    }
}
```

其中 pageref(fd) == pageref(p) 是判断的核心.fd 是 pipe 的读或写文件描述符. p 是描述符数据页即 pipe 缓冲区. 这两个计数引用相等, 则说明当前只有文件描述 fd 端口存在,pipe 的另一端口已经关闭. 这是因为 dup 或者 fork 时,fd 的计数加 1, 对应的描述符数据页也会计数也会加 1. 所以此时说明只有一个端口在使用 pipe, 另一个已经关闭. 还有就是

```
  n = thisenv->env_runs;
    ....
  nn = thisenv->env_runs;
```

不知道为什么需要统计两次运行的次数, 难道在这之间如果发生了时钟中断, 运行的次数发生了改变难道有什么影响吗?

1.7.4 Sharing library code across fork and spawn

Exercise 1. Change duppage in lib/fork.c to follow the new convention. If the page table entry has the PTE_SHARE bit set, just copy the mapping directly. (You should use PTE_SYSCALL, not 0xfff, to mask out the relevant bits from the page table entry. 0xfff picks up the accessed and dirty bits as well.) Likewise, implement copy_shared_pages in lib/spawn.c. It should loop through all page table entries in the current process (just like fork did), copying any page mappings that have the PTE_SHARE bit set into the child process.

具体代码如下 lib/fork.c, 如果有 PTE_SHARE 标志, 则父子进程共享这一页.

```
1  if (vpt[pn] & PTE_SHARE) {
2          if ((ret = sys_page_map(0, (void*)addr, envid, (void*)addr, vpt[pn]
           & PTE_SYSCALL)) < 0)
3              panic("sys_page_map:␣%e", ret);
4          return 0;
5      }
```

另一个是 lib/spawn.c

```
1  static int
2  copy_shared_pages(envid_t child)
3  {
4      // LAB 7: Your code here.
5      uintptr_t addr;
6      uint32_t pn, pd;
7      int r;
8      for (addr = 0; addr < UTOP - PGSIZE; addr += PGSIZE) {
9          pd = PDX(addr);
10         if (vpd[pd] & PTE_P) {
11             pn = addr >> PGSHIFT;
12             if(vpt[pn] & PTE_SHARE) {
13                 if ((r = sys_page_map(0, (void*)addr, child, (void*)addr,
           vpt[pn] & PTE_SYSCALL)) <
14                     panic("sys_page_map:␣%e", r);
15             }
16         }
17     }
18     return 0;
19 }
```

共享区域不采用 COW(copy on write). 父子进程直接共享.

Exercise 2. Change the file server so that all the file descriptor pages get mapped with PTE_SHARE.

这个主要是因为 fork 或者 spawn 时子进程需要共享父进程已经打开的文件. 对应在 JOS 中就是子进程需要共享父进程已经存在的 fd 页. 这个 fd 页由 open 操作时由文件服务器分配, 并通过 IPC 返回给用户进程. 所以在服务端返回该 fd 页时标记该 fd 页需要共享, 即标明权限 PTE_SHARE. 在 fs/serv.c serve_open 函数中修改 fd 页的权限. 添加 PTE_SHARE.

```
*perm_store = PTE_P | PTE_U | PTE_W | PTE_SHARE;
```

1.7.5 The keyboard interface

Exercise 3. In your kern/trap.c, call kbd_intr to handle trap IRQ_OFFSET +IRQ_KBD and serial_intr to handle trap IRQ_OFFSET+IRQ_SERIAL.

这个很简单, 添加对应的中断处理函数就行了. 在 kern/trap.c 中:

```
if (tf->tf_trapno == IRQ_OFFSET + IRQ_KBD) {
        kbd_intr();
        return;
    }
    if (tf->tf_trapno == IRQ_OFFSET + IRQ_SERIAL) {
        serial_intr();
        return;
    }
```

汇编跳转　调研地址:http://docs.oracle.com/cd/E19253-01/817-5477/ennab/index.html

Numeric Labels A numeric label consists of a single digit in the range zero (0) through nine (9) followed by a colon (:). Numeric labels are used only for local reference and are not included in the object file's symbol table. Numeric labels have limited scope and can be redefined repeatedly.

When a numeric label is used as a reference (as an instruction operand, for example), the suffixes b ("backward") or f ("forward") should be added to the numeric label. For numeric label N, the reference Nb refers to the nearest label N defined before the reference, and the reference Nf refers to the nearest label N defined after the reference. The following example illustrates the use of numeric labels:

```
1:          / define numeric label "1"
one:        / define symbolic label "one"

/ ... assembler code ...

```

```
6  jmp   1f    / jump to first numeric label "1" defined
7              / after this instruction
8              / (this reference is equivalent to label "two")
9
10 jmp   1b    / jump to last numeric label "1" defined
11             / before this instruction
12             / (this reference is equivalent to label "one")
13
14 1:          / redefine label "1"
15 two:        / define symbolic label "two"
16
17 jmp   1b    / jump to last numeric label "1" defined
18             / before this instruction
19             / (this reference is equivalent to label "two")
```

2 Survey

2.1 GDB Tips

1. 打印变量地址: p &var
2. 打印寄存器的值: p $esp
3. 打印地址 addr 存储的值: p/x *addr, x 表示 16 进制输出, p/c 表示以字符输出.
4. 设置断点:
   1) b (break) addr (eg.. b *0x7c00)
   2) 可以使用 c 或者 si 命令继续执行, c causes QEMU to continue execution until the next breakpoint (or until you press Ctrl-C in GDB), and si N steps through the instructions N at a time
   3) b function
   4) b file:linenum
   5) b file:functionname
5. finish 跳出当前函数
6. next 下一条指令 (一个函数也当成一条指令)
7. x/Nx ADDR prints N words of memory at ADDR.
8. x/Ni:To examine instructions in memory (besides the immediate next one to be executed, which GDB prints automatically), you use the x/i command. This command has the syntax x/Ni ADDR, where N is the number of consecutive instructions to disassemble and ADDR is the memory address at which to start disassembling.

2.2 GIT Tips

2.2.1 git 修改远程仓库

git remote add origin git@github.com:ldaochen/jos2011.git
出现 fatal:remote origin already exists. 错误
输入 git remote rm origin
再次 git remote add origin git@github.com:ldaochen/jos2011.git
使用 git remove -v 可是显示 remote origin 信息
git remote set-url origin git@github.com:ldaochen/jos2011.git
git push origin lab1 删除远程 master 分支 git push origin –delete master

2.3 X86 硬件以及一个规则

2.3.1 一些规则

%eax 包含返回值, %ecx,%edx 的值可能被冲掉, 这三个寄存器有调用函数保存 (caller saved) %ebp,%ebx,%esi, %edi 在函数被调用前后任然保持不变, 这四个寄存器必须由被调用函数保存.

调用函数 (caller) 保存: eax: 包含返回值 ecx: 值可能被冲掉 edx: 值可能被冲掉

被调用函数 (callee) 保存: ebp ebx esi edi

例如在 bootmain 函数中

```
1 push   %ebp     #x86 gcc calling convention
2 mov    %esp,%ebp    #x86 gcc calling convention
3 push   %edi
4 push   %ebx
```

在 bootmain.c/readseg 函数中

```
1 push   %ebp     #x86 gcc calling convention
2 mov    %esp,%ebp    #x86 gcc calling convention
3 push   %edi
```

jmp 指令可以直接跳转, 也可以间接跳转, 所谓直接跳转就是直接给出跳转目标的标号. 例如:jmp .L1. 所谓间接跳转就是跳转目标的有效地址存储在寄存器或内存中. 例如，jmp *%eax. 目标地址在 eax 寄存器中 jmp *(%eax), 目标地址在 eax 指向的内存双字中. 此外,call 指令也有直接调用和间接调用两种.

2.3.2 VGA 相关调研

The CGA (Color Graphics Adapter) and MDA (Monochrome Display Adapter) cards used different regions. One (color) is at 0xB8000; but monochrome starts at 0xB0000. Remember, you could have both displays active on the machine at once. ypically, the Video Memory is mapped as the following: 0xA0000 -

0xBFFFF Video Memory used for graphics modes 0xB0000 - 0xB7777 Monochrome Text mode 0xB8000 - 0xBFFFF Color text mode and CGA compatible graphics modes

2.3.3 键盘调研

在 lab1 中, 建立了几个数组，以扫描码为下标, 对应的元素就是相应的字符, 数组 normalmap[] 是单独按某键的扫描码对应的字符,shiftmap[] 是 Shift + 某键对应的字符,ctlmap[] 是 ctrl + 某键对应的字符. 另外, 有部分扫描码要区别对待. 它们是:CTL, SHIFT, ALT, CAPSLOCK, NUMLOCK, SCROLLLOCK，以 0xE0 开头的扫描码 (扫描码有 2 个字节). 区别对待的扫描码的赋值方法比较奇特, 有关扫描码资料可在课程主页 reference 中找到.Jos 使用第一套扫描码 断码 = 通码 OR 0x80

2.3.4 IDE 驱动调研

同时参考http://wiki.osdev.org/ATA_PIO_Mode#Polling_the_Status_vs._IRQs Most motherboards today have two IDE controllers built-in, designated as the primary and the secondary IDE controller. These two IDE controllers use the following standard I/O addresses: Primary IDE controller: 1F0h to 1F7h and 3F6h to 3F7h Secondary IDE controller: 170h to 177h and 376h to 377h Each I/O address corresponds to a register on the IDE controller. The following is a list of each I/O address used by ATA controllers and the corresponding regiuster. (I/O addys given are for the primary IDE controller, obviously, but they correspond to the same secondary IDE controller addresses. Thus, for example, the secondary IDE controller's data register is at 170h, the secondary controller's error and features register is at 171h, and so on): 1F0 (Read and Write): Data Register 1F1 (Read): Error Register 1F1 (Write): Features Register 1F2 (Read and Write): Sector Count Register 1F3 (Read and Write): LBA Low Register 1F4 (Read and Write): LBA Mid Register 1F5 (Read and Write): LBA High Register 1F6 (Read and Write): Drive/Head Register 1F7 (Read): Status Register 1F7 (Write): Command Register 3F6 (Read): Alternate Status Register 3F6 (Write): Device Control Register The standard IRQ for the Primary bus is IRQ14, and IRQ15 for the Secondary bus. the "drive select" IO port. On the Primary bus, this would be port 0x1F6 The status register is an 8-bit register which contains the following bits, listed in order from left to right: BSY (busy) DRDY (device ready) DF (Device Fault) DSC (seek complete) DRQ (Data Transfer Requested) CORR (data corrected) IDX (index mark) ERR (error) The error register is also an 8-bit register, and contains the following bits, again listed in order from left to right: BBK (Bad Block) UNC (Uncorrectable data error) MC (Media Changed) IDNF (ID mark Not Found)

MCR (Media Change Requested) ABRT (command aborted) TK0NF (Track 0 Not Found) AMNF (Address Mark Not Found)

ATA commands are issued by writing the commands to the command register. More specifically, ATA commands are issued using the following steps: 1. Poll the status register until it indicates the device is not busy (BUSY will be set to 0) 2. Disable interrupts (assembler "cli" command) 3. Poll the status register until it indicates the device is ready (DRDY will be set to 1) 4. Issue the command by outputting the command opcode to the command register 5. Re-enable interrupts (assembler "sti" command) The following program is a relatively simple assembler program to run the ATA "IDENTIFY DRIVE" command, and print out the results of this command to the screen.

IDENTIFY command To use the IDENTIFY command, select a target drive by sending 0xA0 for the master drive, or 0xB0 for the slave, to the "drive select" IO port. On the Primary bus, this would be port 0x1F6. Then set the Sectorcount, LBAlo, LBAmid, and LBAhi IO ports to 0 (port 0x1F2 to 0x1F5). Then send the IDENTIFY command (0xEC) to the Command IO port (0x1F7). Then read the Status port (0x1F7) again. If the value read is 0, the drive does not exist. For any other value: poll the Status port (0x1F7) until bit 7 (BSY, value = 0x80) clears. Because of some ATAPI drives that do not follow spec, at this point you need to check the LBAmid and LBAhi ports (0x1F4 and 0x1F5) to see if they are non-zero. If so, the drive is not ATA, and you should stop polling. Otherwise, continue polling one of the Status ports until bit 3 (DRQ, value = 8) sets, or until bit 0 (ERR, value = 1) sets. At that point, if ERR is clear, the data is ready to read from the Data port (0x1F0). Read 256 words, and store them.

```
1  MOV DX, 1F7h ;status register
2  LOOP1:
3  IN AL, DX ;sets AL to status register (which is 8 bits)
4
5  ;If the first bit of the status register (BUSY) isn't␣0,␣the␣device␣is␣busy
       ,
6  ;so␣keep␣looping␣until␣it␣isn't.
7
8  AND AL, 10000000xB
9  JNE LOOP1
10
11 ;--------------------------------------------------------------------------
12
13 ;Clear interrupts so something doesn't␣interrupt␣the␣drive␣or␣controller
14 ;while␣this␣program␣is␣working.
15 CLI
16
17 ;--------------------------------------------------------------------------
```

```
18
19  MOV␣DX,␣1F7h␣;status␣register␣again
20  LOOP2:
21  IN␣AL,␣DX␣;sets␣AL␣to␣status␣register␣again
22
23  ;If␣the␣second␣bit␣of␣the␣status␣register␣(DRDY)␣isn't 1, the device isn't
24  ;ready,␣so␣keep␣looping␣until␣it␣is.
25
26  AND␣AL,␣01000000xB
27  JE␣LOOP2
28
29  ;--------------------------------------------------------------------------------
30
31  MOV␣DX,␣1F6h␣;device/head␣register
32  MOV␣AL,␣0␣;0␣selects␣device␣0␣(master).␣10h␣would␣select␣device␣1␣(slave).
33  OUT␣DX,␣AL␣;selects␣master␣device
34
35  MOV␣DX,␣1F7h␣;command␣register
36  MOV␣AL,␣0ECh␣;"IDENTIFY␣DRIVE"␣command
37  OUT␣DX,␣AL␣;sends␣the␣command!
38
39  ;--------------------------------------------------------------------------------
40
41  MOV␣DX,␣1F7h␣;status␣register
42  LOOP3:
43  IN␣AL,␣DX
44
45  AND␣AL,␣00001000xB␣;if␣DRQ␣is␣not␣high,␣the␣device␣doesn't have data for us
46  JE LOOP3 ;yet, so keep looking until it does!
47
48  ;--------------------------------------------------------------------------------
49
50  MOV DX, 1F0h ;data register
51  MOV DI, OFFSET buff ;points DI to the buffer we're␣using
52  MOV␣CX,␣256␣;256␣decimal.␣This␣controls␣the␣REP␣command.
53  CLD␣;clear␣the␣direction␣flag␣so␣INSW␣increments␣DI␣(not␣decrements␣it)
54  REP␣INSW
```

xv6 从 IDE 读 bootloader 代码

```
1  readsect(void *dst, uint32_t offset)
2  {
3      // wait for disk to be ready
4      waitdisk();
5
6      outb(0x1F2, 1);     // count = 1
7      outb(0x1F3, offset);
8      outb(0x1F4, offset >> 8);
9      outb(0x1F5, offset >> 16);
```

```
10      outb(0x1F6, (offset >> 24) | 0xE0);
11      outb(0x1F7, 0x20);  // cmd 0x20 - read sectors
12
13      // wait for disk to be ready
14      waitdisk();
15
16      // read a sector
17      insl(0x1F0, dst, SECTSIZE/4);
18  }
19
20  waitdisk(void)
21  {
22      // wait for disk ready
23      while ((inb(0x1F7) & 0xC0) != 0x40)
24          /* do nothing */;
25  }
```

wait 读取磁盘状态寄存即 1F7. 只有值为 40 时才表示磁盘就绪 (即 BSY 位为 0, RDY 位为 1), 表示磁盘 not busy (BSY=0), 且 ready(RDY=1), 发送扇区计数到 1f2, 发送低 LBA 的低 8 位至 1f3, 发送 LBA 的接下来 8 位至 1f4, 发送 LBA 的再接下来 8 位至 1f5,1f0 包含 LBA 的最高 4 位和选择 master/slave 设备,E0 表示选择 master, F0 表示选择 slave 设备,1F7 同时作为命令寄存器即 20 表示读命令, 然后等待磁盘完成. 相应的数据在数据寄存器 1f0 中.