Document of LAB3@JOS

本次实验中，实现了JOS的进程管理和异常处理机制。

1.记录进程信息的数据结构

JOS使用结构体`struct Env`记录进程的信息。结构体中记录了进程的`id`，状态，页目录地址等基本信息，以及负责在中断或进程切换时保存进程上下文的`struct Trapframe env_tf`，还有在实验的后半部分手动添加的用于记录进程的用户空间中堆的顶部位置，即program break的数据项，我将其命名为`env_pbrk`。

JOS用一段连续的虚拟内存空间`envs`来记录所有的`struct Env`。根据实验文档和注释中的提示，可以用类似于为`pages`分配内存的方式来为`envs`分配内存。于是，使用`boot_alloc`为其分配一段连续的物理内存。为了初始化`envs`时的方便，不妨先把这段内存写成0：

```
envs = (struct Env *)boot_alloc(NENV*sizeof(struct Env));
memset(envs, 0, NENV*sizeof(struct Env));
```

这段物理内存，将会在`men_init()`函数最后一次调用`boot_map_region`（或`boot_map_region_large`)时被映射到内核空间。（当然，它现在已经被映射到了内核空间，但映射关系只存在于临时页表中。）

然后，又将虚拟地址中的`UENVS`区域也映射到这段物理地址，赋予用户读权限:

```
boot_map_region(kern_pgdir, UENVS, NENV*sizeof(struct Env), PADDR(envs), PTE_U | PTE_P);
```

这样一来，当`men_init()`结束后，虚拟地址中`[envs, envs + NENV*sizeof(struct Env) )`和`[UENVS, UENVS + NENV*sizeof(struct Env) )`这两个区域都被线性地映射到了连续的物理地址`[PADDR(envs), PADDR(envs)+NENVS*sizeof(struct Env) )`中。内核一般通过`envs`来访问进程信息。用户程序，在有需要时，可以通过`UENVS`来读取，但不能写入，进程信息。

为`envs`分配好空间后，就可以进行初始化了。初始化时需要把所有的`struct Env`组织成一个链表，并且设置其中的几个数据项。

```
    for (size_t i = 0; i < NENV; i++) {
        envs[i].env_link = envs + (i + 1);
        envs[i].env_id = 0;
        envs[i].env_status = ENV_FREE;
        envs[i].env_pbrk = 0;
    }
    env_free_list = envs;
    envs[NENV - 1].env_link = NULL;
```

其中，将`env_id`和`env_pbrk`设置成0是不必要的，因为在为`envs`分配内存时已经把所有空间写成了0。但为了程序的可读性，这里还是把它们加上了。

2.创建和运行进程

本实验中创建用户进程的大致过程和调用关系是：计算机启动和内核加载后，boot loader将控制传入`i386_init`。在`i386_init`中，内核：

- 调用`men_init`建立好完整的虚拟内存机制，
- 调用`env_init`初始化管理进程的几个数据结构，
- 调用`trap_init`创建中断表描述表IDT并设置CPU的中断环境，
- 通过`ENV_CREATE`宏调用`env_create`创建新进程。在`env_create`中，我们
  - 调用`env_alloc`
    - 为即将创建的新进程在`envs`中分配一个`struct Env`，
    - 产生进程的ID，
    - 设置进程的状态信息，
    - 设置进程的`trap frame`，
    - 调用`env_setup_vm`为该进程分配页目录，建立虚拟内存。
  - 调用`load_icode`
    - 调用`region_alloc`为进程的用户空间分配地址，
    - 把应用程序装在进新进程的用户空间，
    - 设置新进程的入口。
  - 设置进程的种类。
- `env_create`结束后，再调用`env_run`：
  - 改变`curenv`信息，
  - 把新的页目录地址加载进`cr3`，切换虚拟地址空间，
  - 调用`env_pop_tf`完成上下文切换，进入新进程。

2.1 `env_setup_vm()`

要给一个新创建的进程建立虚拟地址空间，首先要分配页目录，这个很好办。JOS作者已经分配好了物理页，只需要把物理页对应的虚拟地址加进`struct Env`中的相应结构就行：

```
    p->pp_ref++;
    e->env_pgdir = (pde_t *)page2kva(p);
```

然后就卡住了。这个进程现在一穷二白，空有一张页目录，页目录里没有任何页表和物理页。要给给它映射哪些虚拟地址，向这些地址里存进哪些东西，有点无从下手。

```
//  Can you use kern_pgdir as a template?  Hint: Yes.
```

看到注释里的提示，可以借助`kern_pgdir`，然后再想到书上讲的，虚拟地址的一大作用就是使得不同进程的虚拟地址空间可以映射到相同的物理地址，这才发现直接把新进程的内核空间直接映射到和父进程相同的物理地址就行。这样既节省了物理内存，又使得不同进程里对内核数据结构的操作有一致性。考虑到父进程的页目录里所有对应用户空间的`entry`都是0，所以直接把父进程的整张页目录复制过来就行。

```
memcpy(e->env_pgdir, kern_pgdir, PGSIZE);
```

这样就算建立好了新进程的内核空间。用户空间，先根据注释里的提示映射一个页的用户栈。还需要存放程序和堆的地方，可以等到加载程序的时候再说。

```
e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_P | PTE_U;
```

2.2 region_alloc()

一个给虚拟地址分配物理地址的函数，调用 page_alloc 和 page_insert 来实现。

```
while (va < end) {
        if (!(pp = page_alloc(0))) {
            panic("Physical page alloc failed in region alloc.\n");
        }
        if ((page_insert(e->env_pgdir, pp, va, PTE_W | PTE_U)) < 0) {
            panic("Page insert failed in region alloc");
        }
        va += PGSIZE;
    }
```

2.3 load_icode()

把ELF二进制程序加载到新进程中的函数。ELF程序的ELF header地址由参数 binary 提供。实际上，这个binary会在调用 ENV_CREATE 宏时通过应用程序对应的符号（_binary_obj_user_hello 等）由链接器生成，再通过 env_create 传递给 load_icode 。

全部仿照 boot/main.c 中加载内核ELF文件的做法。

先检查是否为EFL格式。

```
    struct Elf *elf_hdr = (struct Elf *)binary;
    // If it's not in ELF format, panic.
    if (elf_hdr->e_magic != ELF_MAGIC) {
        panic("Passing a non-elf binary into load_icode()!");
    }
```

找到program header:

```
    struct Proghdr *ph = (struct Proghdr *) ((uint8_t *) elf_hdr + elf_hdr->e_phoff);
    struct Proghdr *eph = ph + elf_hdr->e_phnum;
```

现在要把文件从父进程的某个空间复制到子进程的用户空间。唯一能想到的办法就是暂时把 cr3 改成新进程的页目录地址了，这样感觉有点不安全，但试了一下发现是可以工作的，毕竟新进程和父进程的内核空间有一样的层次结构，并且映射到完全相同的物理地址，而执行复制任务的代码和栈都在内核空间，所以暂时更改 cr3 理论上是可行的。

```
    // temporarily change cr3 so we can write things
    // into the new user environment's address space.
    // It's not elegant but I didn't found other ways.
    lcr3(PADDR(e->env_pgdir));
```

模仿 boot/main.c 根据program header中的相应信息把程序存入相应位置：

```
    for (; ph < eph; ph++) {
        if (ph->p_type == ELF_PROG_LOAD && ph->p_memsz >= ph->p_filesz) {
            region_alloc(e, (void *)ph->p_va, ph->p_memsz);
            memcpy((void*)ph->p_va, (void *)(binary) + ph->p_offset, ph->p_filesz);
            memset((void *)(ph->p_va + ph->p_filesz), 0, ph->p_memsz - ph->p_filesz);
            // update the program break of the process
            if (ph->p_va + ph->p_memsz > (uint32_t)e->env_pbrk) {
                e->env_pbrk = (void *)(ph->p_va + ph->p_memsz);
            }
        }
    }
    // Change the virtual address space back.
    lcr3(PADDR(kern_pgdir));
```

注意到每次存入一个片段后，更新`pbrk`，保证它指向的是用户空间含有有效数据区域的最高处。

修改`env_tf`，以便上下文转换之后能直接进入程序的起点。

```
    e->env_tf.tf_eip = elf_hdr->e_entry;
```

为用户栈分配物理地址。

```
    region_alloc(e, (void *)USTACKTOP - PGSIZE, PGSIZE);
```

2.4 `env_create()`

根据注释里的提示写`env_create`。

调用`env_alloc()`分配`Env`结构体并完成一些初始化，检查分配结果:

```
    struct Env *newenv;
    int flag = env_alloc(&newenv, 0);
    if (flag == -E_NO_FREE_ENV) {
        panic("All environments have been alocated.");
    }
    if (flag == -E_NO_MEM) {
        panic("Memory exhaustion when creating environment.");
    }
```

加载用户程序，设置进程种类:

```
    load_icode(newenv, binary);
    newenv->env_type = type;
```

2.5 `env_run`

这个函数将当前进程切换到进程`e`，首先改变当前进程的状态:

```
        curenv->env_status = ENV_RUNNABLE;
```

理论上，在改变`curenv`之前，需要把当前进程的上下文保存起来。但在实验三中只需要完成从初始进程到用户进程这一个动作，并且也没有找到类似`env_push_tf`这样的函数，所以暂时不保存当前上下文。想必这个功能会在实验四中被完成。

然后把当前进程改成需要运行的进程，并设置几个状态：

```
    curenv = e;
    curenv->env_status = ENV_RUNNING;
    curenv->env_runs++;
```

加载`cr3`切换虚拟地址空间，然后上下文转换进入新进程：

```
    lcr3(PADDR(curenv->env_pgdir));
        env_pop_tf(&(curenv->env_tf));
```

3.实现异常处理机制

当CPU检测到异常时，硬件通过寄存器`IDTR`和异常编号，跳转到异常处理代码处并传递相应的信息。现实现这一机制。

3.1 trapentry.S

每个编号不同的异常在`trapentry.S`中拥有一段相应的handler代码。异常发生时，硬件应该能够自动跳转到相应的handler处。这段代码最重要的功能是把相应的异常编号存入栈中，然后跳转至`_alltraps`统一处理。

```
#define TRAPHANDLER(name, num)                      \
    .globl name;        /* define global symbol for 'name' */   \
    .type name, @function;  /* symbol type is function */       \
    .align 2;       /* align function definition */     \
    name:           /* function starts here */      \
    pushl $(num);                           \
    jmp _alltraps
```

这里已经定义好了一个宏来统一生成handler。上课讲的xv6操作系统是通过python脚本来生成一大段相似重复的汇编代码来处理不同的异常和中断，但JOS先把每个重复单元定义成宏，直接调用宏来handle不同的异常，非常地整洁。

有两种不同的宏，分别对应于是否会使CPU把error code压入栈中的两种异常。从wiki中差到了哪些异常会push error code, 调用相应的宏生成handler:

```
    TRAPHANDLER_NOEC(DIVIDE, T_DIVIDE)  ;
    TRAPHANDLER_NOEC(DEBUG, T_DEBUG)    ;
    TRAPHANDLER_NOEC(NMI, T_NMI)        ;
    TRAPHANDLER_NOEC(BRKPT, T_BRKPT)    ;
    TRAPHANDLER_NOEC(OFLOW, T_OFLOW)    ;
    TRAPHANDLER_NOEC(BOUND, T_BOUND)    ;
    TRAPHANDLER_NOEC(ILLOP, T_ILLOP)    ;
    TRAPHANDLER_NOEC(DEVICE, T_DEVICE)  ;
    TRAPHANDLER(DBLFLT, T_DBLFLT)               ;
    TRAPHANDLER(TSS, T_TSS)                     ;
```

```
    TRAPHANDLER(SEGNP, T_SEGNP)                ;
    TRAPHANDLER(STACK, T_STACK)                ;
    TRAPHANDLER(GPFLT, T_GPFLT)                ;
    TRAPHANDLER(PGFLT, T_PGFLT)                ;
    TRAPHANDLER_NOEC(FPERR, T_FPERR)    ;
    TRAPHANDLER(ALIGN, T_ALIGN)                ;
    TRAPHANDLER_NOEC(MCHK, T_MCHK)      ;
    TRAPHANDLER_NOEC(SIMDERR, T_SIMDERR);
```

宏的第二个参数是异常编号，会被压入栈中。第一个参数是这段handler代码的一个名字。汇编器会通过这个名字生成一个`symbol`。在其它文件中引用这个名字，链接器就会通过这个symbol在相应的地方填入这段代码的入口地址。

然后编写`_alltraps`。这段代码需要在栈中压入一个看起来像`struct Trapframe`的结构，然后跳转到真正处理异常的函数`trap`，这个函数通过这个手工产生的`struct Trapframe`结构来获知和异常有关的各种信息。

实际上，我用生命发现了，`struct Trapframe`结构可以被分成三个部分存进栈里：

```
struct Trapframe {
    struct PushRegs tf_regs;
    uint16_t tf_es;
    uint16_t tf_padding1;
    uint16_t tf_ds;
    uint16_t tf_padding2;
    uint32_t tf_trapno;
    /* below here defined by x86 hardware */
    uint32_t tf_err;
    uintptr_t tf_eip;
    uint16_t tf_cs;
    uint16_t tf_padding3;
    uint32_t tf_eflags;
    /* below here only when crossing rings, such as from user to kernel */
    uintptr_t tf_esp;
    uint16_t tf_ss;
    uint16_t tf_padding4;
} __attribute__((packed));
```

第一个部分是`tf_padding 4`至`tf_eip`。在异常刚刚产生时，硬件立即通过IDT改变`cs`和`eip`，跳转至trap handler处；同时，进程即刻变成内核态，并改变`esp`，开始使用内核栈。同时，TSS机制把`ss`,`esp`，...`eip`等信息存入栈中，正好和`struct Trapframe`的相应部分相吻合。

第二部分时`tf_err`至`tf_es`，这一部分通过`trapentry.S`里的汇编指令手动被存进栈里。（实际上，有一部分异常的`tf_err`会被硬件存进栈里，它们就是之前使用宏`TRAPHANDLER`的那部分异常。）其中，`tf_trapno`和部分异常的`tf_err（其实存进的都是0）`是在handler里被存进的，而`tf_padding2`至`tf_es`要在`_alltraps`里手动存入：

```
    pushw $(0);
    pushw %ds;
    pushw $(0);
    pushw %es;
```

第三个部分，`tf_regs`，所用通用寄存器，根据题目里的提示，正好可以用一条`pushal`指令全部存入！

```
    pushal
```

这是JOS里一个极具美感的设计。硬件的TSS机制存入的信息形成的结构，和直接使用`phshal`指令把当前通用寄存器压栈形成的结构，正好和`struct Trapframe`相吻合，可以看出JOS的作者非常有大局观，在设计`struct Trapframe`的时候就考虑到了这一层，从而可以非常优雅地实现`_alltrap`。

在这之后，把寄存器`%esp`压入栈，然后调用`trap`函数。

```
    pushl %esp;
    call trap;
```

根据调用协议，编译器把调用前的栈顶值当作`trap`的第一个参数，它正好指向之前存入的一段和`Trapframe`结构相同的数据，所以`trap`函数可以把他当成一个`Trapframe`的指针，根据它来确定异常的种类、发生时的环境、以及其他信息，从而对其进行处理。

3.2 `trap_init()`

要在IDT中生成相应的条目，首先需要取得异常的`handler`的地址。注意到在`trapentry.S`中为每个handler都产生了一个`name`标记，如果在其他地方将这个标记当作函数名进行声明，链接器在程序的链接阶段就会在引用了这些`name`的地方生成相应的地址。

```
    void DIVIDE();
    void DEBUG();
    void NMI();
    // ... ...
    void SYSCALL();
```

根据题目里的提示，可以使用宏`SETGATE`填写IDT里的条目。为了调试时的方便，把这个宏包裹起来了：

```
void
load_idt(size_t i, bool is_trap, void *p) {
    // idt[i].gd_off_15_0 = (unsigned)p & 0x00FF;
    // idt[i].gd_off_31_16 = (unsigned)p >> 16;
    // idt[i].gd_sel = GD_KT;

    // User should have the permission to use breakpoint and system call.
    if (i == T_BRKPT || i == T_SYSCALL) {
        // Spent a century to find this micro.
        SETGATE(idt[i], is_trap, GD_KT, (uint32_t)p, 3);
        return;
    }
    SETGATE(idt[i], is_trap, GD_KT, (uint32_t)p, 0);
}
```

注意到对于break point异常和系统调用，应该将调用的权限设置为ring3，否则将触发通用保护异常，无法达成想要的目的。

然后依次设置IDT条目：

```
load_idt(0, 1, DIVIDE);
load_idt(1, 1, DEBUG);
load_idt(2, 0, NMI);
// others
```

4.处理具体的异常

这里处理几个具体异常。

4.1缺页异常

在`trap_dispath`中，根据传入的`struct Trapframe tf`中记录的异常编号`tf_trapno`确定异常种类。若为缺页异常，则跳转至缺页处理函数`page_fault_handler`。

```
if (tf->tf_trapno == T_PGFLT) {
    page_fault_handler(tf);
}
```

4.2断点异常

如果异常种类为断点异常，则跳转至函数`monitor`，便于用户调试程序。

```
if (tf->tf_trapno == T_BRKPT) {
    monitor(tf);
}
```

在IDT中要将设置该异常的权限使得`ring3`可调用，否则在侦测到的一瞬间硬件触发通用保护故障。

4.3系统调用

用户程序使用系统调用的调用过程如下：

- 用户调用lib库中的输入输出等接口函数
- 这些接口函数调用lib库中的sys系列函数
- lib中的sys系列函数调用lib中的`syscall`
- `syscall`函数把寄存器设置成相关的参数，以便让TSS机制将这些参数存入栈中，并产生相应的汇编指令`int x30`，这个指令触发系统调用异常
- 触发异常后，通过IDT和`trapentry.S`进入`trap`函数，再进入`trap_dispatch`函数
- `trap_dispatch`调用kern中的`syscall`函数
- `syscall`函数调用kern中的sys系列函数，完成具体的系统调用任务

```
    if (tf->tf_trapno == 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);
       // Without this, user process will be destroyed.
       return;
    }
```

如上按照题目里的提示跳转至`syscall`。把返回值存入`reg_eax`，这样这个返回值在返回用户程序的时候就会被存进`%eax`供用户程序使用。

`syscall`没啥好说的，按次序填入参数调用`sys_.*`：

```
        switch (syscallno) {
        case SYS_cputs:
            sys_cputs((const char *)a1, (size_t)a2);
            // Return the length of the output string.
            return (int32_t)a2;

            ... ...
```

4.4动态内存分配

用于动态分配的堆内存区在用户程序ELF文件的上方，栈的下方。要有一个数据结构记录堆顶的位置，好像没有现成的，试着自己加一个：

```
    void *env_pbrk;         // Program break, the top of the process's data segment
```

在`load_icode`里加载应用程序的时候要记得维护这个数据结构。然后再`sbrk`里就可以用它分配内存了：

```
static int
sys_sbrk(uint32_t inc)
{
    // LAB3: your code here.
    // Added on April 8.
    void *begin = (void *)ROUNDDOWN(curenv->env_pbrk, PGSIZE);
    void *end = (void *)ROUNDUP(curenv->env_pbrk + inc, PGSIZE);
    pte_t *pte_ptr;
    struct PageInfo *pp;

    while (begin < end) {
        // If virtual address begin is not mapped
        if(!page_lookup(curenv->env_pgdir, begin, &pte_ptr)) {
            // Allocate a physical page
            pp = page_alloc(ALLOC_ZERO);
            if(!pp) {
                panic("Page alloc failed at sys_sbrk().");
            }
```

```
            pp->pp_ref++;
            // Permission should be writable
            page_insert(curenv->env_pgdir, pp, begin, PTE_U | PTE_W);
        }
        begin += PGSIZE;
    }

    return (int)(curenv->env_pbrk += inc);
}
```

用`page_lookup`查查需要分配的地方有没有被映射，没有的话，用`page_alloc`分配一个页框，用`page_insert`映射过去。更新`env_pbrk`，返回更新的值。

4.5检查地址合法性

用户传给内核想让内核处理的地址，内核要检查它是否在用户空间，以及用户是否有权限接触这个地址。

```
    if ((uint32_t)va >= ULIM) {
        user_mem_check_addr = (uintptr_t)va;
        return -E_FAULT;
    }

    if ((uint32_t)(va + len) >= ULIM) {
        user_mem_check_addr = (uintptr_t)ULIM;
        return -E_FAULT;
    }
```

如果地址超出用户空间范围，返回错误，将标记设置为第一个不合法地址。

```
    while (va < end) {
        // Get the page tabel entry!
        // If the page is not mapped, return!
        if (!(page_lookup(env->env_pgdir, (void *)va, &pte_ptr))) {
            user_mem_check_addr = first_loop ? (uintptr_t)va : (uintptr_t)ROUNDDOWN(va,
PGSIZE);
            return -E_FAULT;
        }

        // Check the permission! (a little dirty)
        if (((*pte_ptr) & (perm | PTE_P)) != (perm | PTE_P)) {
            user_mem_check_addr = first_loop ? (uintptr_t)va : (uintptr_t)ROUNDDOWN(va,
PGSIZE);
            return -E_FAULT;
        }

        va += PGSIZE;
        first_loop = false;
    }
```

再检查地址是否被映射以及用户是否有权限读写。检查失败时将标记设为第一个不合法地址。

4.5 callgate机制

`callgate`机制，使得用户程序可以在自身的代码区将权限暂时提高，执行一些原本只有内核有权限执行的指令，而不必使用系统调用。

在GDT里设置好一个`callgate`，再调用`lcall`，硬件会根据GDT和`lcall`指令设置`cs`和`eip`，跳转至需要执行的代码处并暂时改变权限，再通过`lret`返回。

代码基本都已经提供了。**仔细调试**后发现包装函数`call_fun_ptr`少了一条把`ebx`push出来的指令，导致无法正常返回，进行修改：

```
void call_fun_ptr()
{
    evil();
    *entry = old;
    asm volatile("popl %ebx");
    asm volatile("popl %ebp");
    asm volatile("lret");
}
```

执行成功。

5.总结

本次实验建立了基本的进程管理机制和异常处理机制，内容比较多，理解和调试也比较苦难。实验的过程碰到很多困难，比如

- 合并LAB2分支之后系统不能正常启动。一个Assertion error告诉我某处对宏`PADDR`传入了非法值，然后发现是一个值为0的`kern_pgdir`，接着发现`kern_pgdir`在初始化它自己所指空间的`memset()`调用之后变成0，进而发现是`boot_alloc()`传回的`kern_pgdir`的值竟然在`kern_pgdir`的地址前面。再然后想到未初始化的全局变量是放在`.bss segment`里的，且在程序运行前不会给其分配空间。而`boot_alloc()`最先返回的值是`.bss`之后的第一个可用空间，由于此时程序还没有给全局变量分配空间，导致`boot_alloc()`返回的值竟然在几个全局变量之前，几个全局变量都被`memset()`清空了。所以系统生成的magic number `end`可能并不是安全的(当其已经对齐PGSIZE时)。加上一个PGSIZE可以解决。
- 建立好创建进程机制后一直不能输出hello，而且整个系统无限循环。我以为是进程切换的函数没写好或者LAB2里的内存管理函数有问题，一直查到晚上7：30才发现是因为输出字符需要系统调用，但现在还没有实现系统调用的环境，所以一个中断直接导致系统宕机重启无限循环。进程切换的实现是正确的，浪费了很多时间，比较愚蠢。这是对系统调用的理解和对JOS现在的进度把握不充分导致的。
- 设置好异常机制后发现传进`trap()`的`tf`没有正常维护异常进入点的`cs`等值，后来发现是因为DIVIDE异常会导致CPU存入错误码，而在handler处选择了不存入错误码时对应的宏，使栈结构变化，不能正常读出`tf`结构。修复好之后又发现不能正常显示trap的种类，发现原因是不能正常读取`tf->trapno`，再发现原因是`_alltrap`处人工在栈上构建的`Trapframe`结构少了一段，这一段不会由`pushal`生成。修好之后又发现`cs`的值有问题。找到真正原因：实际上最开始用的宏没有问题，`cs`显示错误是因为少了第二个BUG里的那一段，改完第一个bug后宏用错了导致多了一段padding，同时又少了第二个Bug里的一段，负负得正`cs`显示正确。这说明下手之前要把整个原理想清楚，不然会犯拆东墙补西墙的错误。

等等。

在这次实验里熟悉了进程和异常的知识，也锻炼了代码读写和调试的能力，对Unix-like内核的架构有了初步了解 。这次实验是很有意义的。