JOS Lab 3: User Environments

Introduction

本次lab将实现操作系统的一些基本功能，来实现用户环境下的进程的正常运行。我们需要加强JOS内核的功能，为它增添必要的数据结构来记录用户进程环境的一些信息；创建一个单一的用户环境，加载一个程序并运行它。让JOS内核能够完成用户环境所作出的任何系统调用，以及处理用户环境产生的各种异常。Question的回答请见**answer-lab3.txt**。

Part A

Part A需要我们实现、分配用户环境以及建立初步的异常或中断处理机制，用户环境的具体数据结构在`inc/env.h`中。用户环境是应用程序运行在系统中所需要的上下文环境，操作系统内核使用数据结构`Env`来记录每一个用户环境的信息。本次lab只会创建一个用户环境。

Exercise 1

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

Exercise 1需要修改`mem_init()`函数，为`envs`数组分配空间，将这部分空间设置为用户只读模式，并映射到虚拟地址UENVS处。仿照lab2中`pages`数组空间的分配和映射，来实现`envs`数组的分配和映射。
分配：
`envs=(struct Env *)boot_alloc(NENV*sizeof(struct Env));`
映射：
`boot_map_region(kern_pgdir,UPAGES,ROUNDUP(npages * sizeof(struct Page),PGSIZE),PADDR(pages),PTE_U|PTE_P);`
同时这里我发现了lab2的一个隐藏错误，之前的代码无法通过现在的`check_kern_pgdir()`等几个函数，排查后发现错误出在`page_init()`函数，需要将原来的
`size_t hole_end=PGNUM(end-KERNBASE+PGSIZE+npages*sizeof(struct Page));`
更改为
`size_t hole_end=PGNUM(PADDR(ROUNDUP(boot_alloc(0),PGSIZE)));`

Exercise 2

Exercise 2要我们完成`env_init()`、`env_setup_vm()`、`region_alloc()`、`load_icode()`、`env_create()`及`env_run()`几个函数，用以初始化及运行用户环境。
`env_init()`负责初始化`envs`数组中的`struct Env`，将它们加入`env_free_list`，同时调用`env_init_percpu()`设置访问优先级（privilege level 0 (kernel)及privilege level 3 (user)）。实现方式就是从数组的最后一项向前遍历，一边初始化当前`struct Env`一边将当前Env插在`env_free_list`的表头，这样构建起来的list指向的第一项就是数组的第一项，并依次沿数组里的item指下去。

```
void
env_init(void)
{
    if(NENV<=0)
    {
        env_free_list=NULL;
        env_init_percpu();
        return;
    }

    int i;
    env_free_list = NULL;
    for (i = NENV -1; i >= 0; --i)
    {
         envs[i].env_id = 0;
        envs[i].env_status = ENV_FREE;
        envs[i].env_link = env_free_list;
        env_free_list = &envs[i];
    }
```

```
    env_init_percpu();
}
```

`env_setup_vm`负责为用户环境分配及初始化页目录表。先分配一个页作为Env的`env_pgdir`，将该页的pp_ref加一，然后将UTOP以上的位置从kern_pgdir复制到env_pgdir中以便访问。

```
static int
env_setup_vm(struct Env *e)
{
    int i;
    struct Page *p = NULL;

    if (!(p = page_alloc(ALLOC_ZERO)))
        return -E_NO_MEM;

    e->env_pgdir=page2kva(p);
    p->pp_ref+=1;
    //int i=0;
    for(i=PDX(UTOP);i<NPDENTRIES;++i)
    {
        e->env_pgdir[i]=kern_pgdir[i];
    }

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

    return 0;
}
```

`region_alloc`为用户分配物理空间。将`[va,va+len)`地址空间分配给用户环境，注意这里需要对va及va+len分别进行页对齐，并将访问权限设置为`writable by user and kernel`。

```
void
region_alloc(struct Env *e, void *va, size_t len)
{
    void* i;
    void* begin_addr=ROUNDDOWN(va,PGSIZE);
    void* end_addr=ROUNDUP(va+len,PGSIZE);
    for(i=begin_addr;i<end_addr;i+=PGSIZE)
    {
        struct Page* page1=page_alloc(1);
        if(page1==NULL)panic("kern/env.c:region_alloc:Memory not enough\n");
        page_insert(e->env_pgdir,page1,i,PTE_W|PTE_U);
    }
}
```

`load_icode`要我们解析ELF文件，将文件内容加载到当前用户环境对应的用户地址空间。参照boot_main读取ELF的方式进行读取。首先在内核态加载ELF文件，然后用`lcr3`函数跳转到用户态获取文件内容，读取完后再用`lcr3`跳转回内核态，设置入口elf->entry，分配一个页作为用户进程的栈，这里只考虑一个用户进程，设置为从`USTACKTOP-PGSIZE`开始的`PGSIZE`大小的地址空间。

```
static void
load_icode(struct Env *e, uint8_t *binary, size_t size)
{
    struct Proghdr *ph,*eph;
    struct Elf* elf=(struct Elf*)binary;
    if(elf->e_magic!=ELF_MAGIC)panic("kern/env.c:load_icode:not a elf\n");
    ph=(struct Proghdr*)(binary+elf->e_phoff);
    eph=ph+elf->e_phnum;

    lcr3(PADDR(e->env_pgdir));
    while(ph<eph)
    {
        if(ph->p_type == ELF_PROG_LOAD)
        {
            if(ph->p_filesz > ph->p_memsz)panic("kern/env.c:load_icode:ph->p_filesz > ph->p_memsz\n");
```

```
            region_alloc(e, (void*)ph->p_va, ph->p_memsz);
            memmove((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));
        }
        ++ph;
    }

    lcr3(PADDR(kern_pgdir));
    e->env_tf.tf_eip = elf->e_entry;

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

`env_create`用`env_alloc`分配用户环境，然后用`load_icode`将ELF文件加载到这个用户环境。

```
void
env_create(uint8_t *binary, size_t size, enum EnvType type)
{
    struct Env* env1;
    if(env_alloc(&env1,0)==0)
    {
        load_icode(env1,binary,size);
        env1->env_type=type;
    }
}
```

`env_run`开始在用户态下运行用户环境，将参数Env赋给curenv，并更改状态为running，用lcr3跳转到要执行的进程的地址空间，然后调用env_pop_tf，读取保存的寄存器信息。

```
void
env_run(struct Env *e)
{
    //cprintf("env_run here!\n");
    if(curenv!=NULL&&curenv->env_status==ENV_RUNNING)
    {
        curenv->env_status=ENV_RUNNABLE;
    }
    curenv=e;
    curenv->env_status=ENV_RUNNING;
    curenv->env_runs+=1;
    lcr3(PADDR(e->env_pgdir));
    env_pop_tf(&(e->env_tf));

    //panic("env_run not yet implemented");
}
```

Exercise 3

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

Exercise 4

Exercise 4需要修改`trapentry.S`和`trap.c`文件。在`trapentry.S`中为每个exception或interrupt创建相应的handler；修改`trap.c`中的`trap_init`函数，用这些handler的地址来初始化IDT。要求每个handler都应在栈上创建一个`struct Trapframe`，并调用trap函数。
首先修改`trapentry.S`

```
/*
 * Lab 3: Your code here for generating entry points for the different traps.
 */
    TRAPHANDLER_NOEC(entry0,T_DIVIDE);
    TRAPHANDLER_NOEC(entry1,T_DEBUG);
    TRAPHANDLER_NOEC(entry2,T_NMI);
    TRAPHANDLER_NOEC(entry3,T_BRKPT);
```

```
    TRAPHANDLER_NOEC(entry4,T_OFLOW);
    TRAPHANDLER_NOEC(entry5,T_BOUND);
    TRAPHANDLER_NOEC(entry6,T_ILLOP);
    TRAPHANDLER_NOEC(entry7,T_DEVICE);
    TRAPHANDLER(entry8,T_DBLFLT);
    TRAPHANDLER(entry10,T_TSS);
    TRAPHANDLER(entry11,T_SEGNP);
    TRAPHANDLER(entry12,T_STACK);
    TRAPHANDLER(entry13,T_GPFLT);
    TRAPHANDLER(entry14,T_PGFLT);
    TRAPHANDLER_NOEC(entry16,T_FPERR);
    TRAPHANDLER(entry17,T_ALIGN);
    TRAPHANDLER_NOEC(entry18,T_MCHK);
    TRAPHANDLER_NOEC(entry19,T_SIMDERR);
```

`_alltraps`参考env_pop_tf的pop方法，将Trapframe结构加入栈中，并设置%ds与%es，最后调用trap函数处理异常或中断。

```
/*
 * Lab 3: Your code here for _alltraps
 */
.func _alltraps
_alltraps:
    pushl %ds
    pushl %es
    pushal
    movw $GD_KD, %ax
    movw %ax, %ds
    movw %ax, %es
    pushl %esp
    call trap
```

然后修改`trap.c`中的`trap_init`，不过先要声明一下入口点函数

```
extern void entry0();
extern void entry1();
extern void entry2();
extern void entry3();
extern void entry4();
extern void entry5();
extern void entry6();
extern void entry7();
extern void entry8();
extern void entry10();
extern void entry11();
extern void entry12();
extern void entry13();
extern void entry14();
extern void entry16();
extern void entry17();
extern void entry18();
extern void entry19();
```

然后在`trap_init`中调用`SETGATE`来建立handler函数和IDT表之间的映射关系，并且设置权限位。

```
    // LAB 3: Your code here.
    SETGATE(idt[T_DIVIDE], 0, GD_KT, entry0, 0);
    SETGATE(idt[T_DEBUG], 0, GD_KT, entry1, 0);
    SETGATE(idt[T_NMI], 0, GD_KT, entry2, 0);
    SETGATE(idt[T_BRKPT], 0, GD_KT, entry3, 3);
    SETGATE(idt[T_OFLOW], 0, GD_KT, entry4, 0);
    SETGATE(idt[T_BOUND], 0, GD_KT, entry5, 0);
    SETGATE(idt[T_ILLOP], 0, GD_KT, entry6, 0);
    SETGATE(idt[T_DEVICE], 0, GD_KT, entry7, 0);
    SETGATE(idt[T_DBLFLT], 0, GD_KT, entry8, 0);
    SETGATE(idt[T_TSS], 0, GD_KT, entry10, 0);
    SETGATE(idt[T_SEGNP], 0, GD_KT, entry11, 0);
```

```
    SETGATE(idt[T_STACK], 0, GD_KT, entry12, 0);
    SETGATE(idt[T_GPFLT], 0, GD_KT, entry13, 0);
    SETGATE(idt[T_PGFLT], 0, GD_KT, entry14, 0);
    SETGATE(idt[T_FPERR], 0, GD_KT, entry16, 0);
    SETGATE(idt[T_ALIGN], 0, GD_KT, entry17, 0);
    SETGATE(idt[T_MCHK], 0, GD_KT, entry18, 0);
    SETGATE(idt[T_SIMDERR], 0, GD_KT, entry19, 0);
```

我理解中的Exercise 4的整体逻辑就是，系统调用`trap_init`函数在IDT中建立各个exception或interrupt与对应的handler函数的映射关系，然后当系统遇到异常或中断时，就会调用对应的handler函数进行处理。调用过程是这样的：IDT中与异常或中断对应的handler函数现在严格上来说还只是入口点函数（trapEntry.S中定义的entry0、entry1等），这些入口点函数将当前异常或中断的type压入栈中以便后续函数识别其类型，然后跳转到`_alltraps`，用GD_KD设置好%ds，%es，表示处理函数是运行在内核代码段，然后调用trap函数真正处理这些异常或中断。

Part B

Part B要求我们继续完善系统对异常或中断的处理能力。

Exercise 5

Exercise 5要求修改`trap_dispatch`函数，让系统在遇到缺页异常时调用`page_fault_handler()`来处理。`trap_dispatch()`函数在`trap()`中被调用，用于解析当前异常或中断是哪种类型，然后调用相应函数处理它。因此只需要在`trap_dispatch()`中添加代码，判断是缺页异常时就调用`page_fault_handler()`函数即可。

```
    switch(tf->tf_trapno)
    {
        case T_PGFLT:
            page_fault_handler(tf);
            break;
```

Exercise 6

Exercise 6要我们实现system calls，并且要求只能通过使用`sysenter`和`sysexit`的方式来实现。
首先是在`lib/syscall.c`中添加汇编代码，让程序执行sysenter，进入内核态。这里虽然文档上要求只为syscall提供4个参数，但在syscall的定义里面参数却有五个，所以这里我用了%esi来作为最后一个参数。

```
    asm volatile("pushl %%ecx\n\t"
         "pushl %%edx\n\t"
             "pushl %%ebx\n\t"
         "pushl %%esp\n\t"
         "pushl %%ebp\n\t"
         "pushl %%esi\n\t"
         "pushl %%edi\n\t"

                 //Lab 3: Your code here
        "pushl $1f\n\t"
        "movl %%esp, %%ebp\n\t"
        "sysenter\n\t"
        "1: addl $0x4, %%esp\n\t"

                 "popl %%edi\n\t"
                 "popl %%esi\n\t"
                 "popl %%ebp\n\t"
                 "popl %%esp\n\t"
                 "popl %%ebx\n\t"
                 "popl %%edx\n\t"
                 "popl %%ecx\n\t"

                 : "=a" (ret)
                 : "a" (num),
                   "d" (a1),
                   "c" (a2),
                   "b" (a3),
                   "D" (a4),
```

```
                  "S" (a5)
                : "cc", "memory");
```

然后在`trapEntry.S`中实现`sysenter_handler`。首先将syscall需要的参数逐个压栈，压栈的顺序与参数顺序相反，然后设置%ds、%es表示运行在内核代码段，调用syscall，调用完毕后重新设置%ds、%es，回到用户代码段，最后调用sysexit结束系统调用。

```
.globl sysenter_handler;
.type sysenter_handler, @function;
.align 2;
sysenter_handler:
/*
 * Lab 3: Your code here for system call handling
 */
    pushl %esi
    pushl %edi
    pushl %ebx
    pushl %ecx
    pushl %edx
    pushl %eax
    movw $GD_KD, %ax
    movw %ax, %ds
    movw %ax, %es
    call syscall
    movw $GD_UD, %dx
    movw %dx, %ds
    movw %dx, %es
    movl %ebp, %ecx
    movl (%ebp), %edx
    sysexit
```

然后修改`kern/syscall.c`，对于不同类型的系统调用采用不同的函数来处理

```
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.
    int32_t ret = -E_INVAL;
    switch(syscallno)
    {
        case SYS_cputs:
            sys_cputs((char*)a1, a2);
            break;
        case SYS_cgetc:
            ret = sys_cgetc();
            break;
        case SYS_getenvid:
            ret = sys_getenvid();
            break;
        case SYS_env_destroy:
            ret = sys_env_destroy(a1);
            break;
        case SYS_map_kernel_page:
            ret = sys_map_kernel_page((void *) a1, (void *)a2);
            break;
        case SYS_sbrk:
            cprintf("kern/syscall.c: syscall(): SYS_sbrk\n");
            ret = sys_sbrk(a1);
            break;
        default: break;
    }
    return ret;

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

最后还需要在`kern/init.c`中添加代码来设置特定寄存器。但是在`kern/init.c`中并没有发现可以添加代码的地方(没有找到Lab3: your code here的提示)，i386_init函数又不敢随便乱改，不过它调用了之前用来绑定异常与异常处理函数的trap_init()函数，于是决定将代码添加在这里。文档所给链接只有`wrmsr`的使用示例，但是并没有给`wrmsr`的定义，最后在网上找到其定义，添加后为

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

    // LAB 3: Your code here.
    SETGATE(idt[T_DIVIDE], 0, GD_KT, entry0, 0);
    SETGATE(idt[T_DEBUG], 0, GD_KT, entry1, 0);
    SETGATE(idt[T_NMI], 0, GD_KT, entry2, 0);
    SETGATE(idt[T_BRKPT], 0, GD_KT, entry3, 3);
    SETGATE(idt[T_OFLOW], 0, GD_KT, entry4, 0);
    SETGATE(idt[T_BOUND], 0, GD_KT, entry5, 0);
    SETGATE(idt[T_ILLOP], 0, GD_KT, entry6, 0);
    SETGATE(idt[T_DEVICE], 0, GD_KT, entry7, 0);
    SETGATE(idt[T_DBLFLT], 0, GD_KT, entry8, 0);
    SETGATE(idt[T_TSS], 0, GD_KT, entry10, 0);
    SETGATE(idt[T_SEGNP], 0, GD_KT, entry11, 0);
    SETGATE(idt[T_STACK], 0, GD_KT, entry12, 0);
    SETGATE(idt[T_GPFLT], 0, GD_KT, entry13, 0);
     SETGATE(idt[T_PGFLT], 0, GD_KT, entry14, 0);
      SETGATE(idt[T_FPERR], 0, GD_KT, entry16, 0);
    SETGATE(idt[T_ALIGN], 0, GD_KT, entry17, 0);
    SETGATE(idt[T_MCHK], 0, GD_KT, entry18, 0);
    SETGATE(idt[T_SIMDERR], 0, GD_KT, entry19, 0);

    // Per-CPU setup
    __asm__ __volatile__("wrmsr"
        : : "c" (0x174), "a" (GD_KT), "d" (0));
    __asm__ __volatile__("wrmsr"
        : : "c" (0x175), "a" (KSTACKTOP), "d" (0));
    __asm__ __volatile__("wrmsr"
        : : "c" (0x176), "a" (sysenter_handler), "d" (0));

    trap_init_percpu();
}
```

这里还是要先对sysenter_handler函数添加外部引用声明

```
extern void sysenter_handler();
```

Exercise 7

按照提示在`libmain()`中初始化`thisenv`指针即可

```
thisenv=&envs[ENVX(sys_getenvid())];
```

Exercise 8

Exercise 8要求我们实现系统调用sbrk，具体来说就是实现`kern/syscall.c`中的sys_sbrk函数，拓展用户环境的data段，只需调用一下region_alloc函数即可。

```
extern void region_alloc(struct Env *e, void *va, size_t len);
static int
sys_sbrk(uint32_t inc)
{
    // LAB3: your code sbrk here...
    region_alloc(curenv, (void*)(curenv->env_va_end - inc), inc);
    return curenv->env_va_end;
    //return 0;
```

```
}
```

参数里面的curenv->env_va_end是在Env定义中新加的属性，用于保存当前用户环境的虚拟空间的栈的地址的底。

```
struct Env {
    struct Trapframe env_tf;    // Saved registers
    struct Env *env_link;        // Next free Env
    envid_t env_id;            // Unique environment identifier
    envid_t env_parent_id;        // env_id of this env's parent
    enum EnvType env_type;        // Indicates special system environments
    unsigned env_status;        // Status of the environment
    uint32_t env_runs;        // Number of times environment has run

    // LAB3: might need code here for implementation of sbrk
    uint32_t env_va_end;
```

region_alloc函数中需要添加代码，按照实际分配的空间对env_va_end进行更改

```
void
region_alloc(struct Env *e, void *va, size_t len)
{
    uint32_t env_va_end = (uint32_t)ROUNDDOWN(va, PGSIZE);
    va = ROUNDDOWN(va, PGSIZE);
    len = ROUNDUP(len, PGSIZE);
    struct Page* pp;
    for(; len > 0; len -= PGSIZE, va += PGSIZE){
        pp = page_alloc(1);
        if (!pp)
            panic("region alloc: page_alloc(1) failed\n");
        if (page_insert(e->env_pgdir, pp, va, PTE_U | PTE_W))
            panic("region alloc: page_insert_failed\n");
    }
    e->env_va_end = (uint32_t)env_va_end;
}
```

Exercise 9

Exercise 9要求我们修改`trap_dispatch()`函数，让breakpoint异常触发kernel monitor。还要修改kernel monitor来支持GDB式的调试`c`, `si`及`x`。
首先是修改`trap_dispatch()`，在其中添加异常情况为breakpoint情况时的处理（即调用monitor函数）。

```
static void
trap_dispatch(struct Trapframe *tf)
{
    // Handle processor exceptions.
    // LAB 3: Your code here.

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

```
}
```

然后在`monitor.c`里的commands数组中添加`c`，`si`及`x`指令

```
    { "c", "Continue execution from the current location", mon_c},
    { "si", "Execute the code instruction by instruction", mon_si},
    { "x", "Display the memory", mon_x}
```

然后实现以上三个指令对应的函数（同时在`monitor.h`中添加它们的声明）

```
int mon_c(int argc, char **argv, struct Trapframe *tf)
{
    if(argc!=1)
    {
        cprintf("Usage: c\n");
    }
    tf->tf_eflags &= ~FL_TF;
    env_run(curenv);
    return 0;
}

int mon_si(int argc, char **argv, struct Trapframe *tf)
{
    if(argc!=1)
    {
        cprintf("Usage: si\n");
    }
    tf->tf_eflags |= FL_TF;
    cprintf("tf_eip=0x%x\n", tf->tf_eip);
    env_run(curenv);
     return 0;
}

int mon_x(int argc, char **argv, struct Trapframe *tf)
{
    if(argc != 2)
    {
        cprintf("Usage: x [address]\n");
        return 0;
    }
    uint32_t addr = strtol(argv[1], NULL, 16);
    cprintf("%08p:\t%u\n", addr, *((uint32_t *)addr));
    return 0;
}
```

Exercise 10

Exercise10要求添加检测用户程序，防止有攻击性的用户程序破坏系统。
首先是更改`kern/trap.c`中的page_fault_handler函数，检测page fault是否发生在kernel态，是的话则要引发panic

```
    if (!(tf->tf_cs & 0x3))
        panic("PGFLT trapped in kernel\n");
```

然后是实现`kern/pmap.c`中的`user_mem_check`函数，用以检查va是否由越界情况，若果有越界就抛出-E_FAULT。

```
int
user_mem_check(struct Env *env, const void *va, size_t len, int perm)
{
    // LAB 3: Your code here.
    pte_t *pte;
    uintptr_t begin_addr = (uintptr_t)va;
    uintptr_t end_addr = (uintptr_t)(begin_addr + len-1);
    uintptr_t i=begin_addr;
    perm|=PTE_U|PTE_P;
```

```
    while(i<=end_addr)
    {
        if (i >= ULIM)
        {
            user_mem_check_addr = i;
            return -E_FAULT;
        }

        pte = pgdir_walk(env->env_pgdir, (void*)i, 0);
        if (!pte || (*pte & perm) != perm)
        {
            user_mem_check_addr = i;
            return -E_FAULT;
        }
        i=ROUNDDOWN(i+PGSIZE,PGSIZE);
    }
    return 0;
    //return 0;
}
```

最后在`sys_cputs`函数以及`kdebug.c`的`debuginfo_eip`中调用`user_mem_check`函数检查用户程序的安全性。

Exercise 11

用于检查之前的代码的正确性，希望之后的lab都能多几个这样的exercise来凑数。

Exercise 12

Exercise12 要求我们hack自己的系统。系统会分别在0和3权限下执行含有特权指令的evil函数，要求我们实现的是自主进入0权限然后调用evil函数。首先通过sgdt指令读取gdt，并映射到用户态创建的内存空间中，然后根据这块内存空间修改gdt，然后调用SETCALLGATE进入0权限并调用evil_helper函数，在evil_helper函数中会调用evil函数。

```
char addr[PGSIZE];
struct Segdesc* gdt;
struct Segdesc* entry;
struct Segdesc old;
void evil_helper(){
    evil();
    *entry = old;
    asm volatile("leave\n\t");
    asm volatile("lret");
}
void ring0_call(void (*fun_ptr)(void)) {
    // Lab3 : Your Code Here
    struct Pseudodesc p1;
    sgdt(&p1);
    int ret = sys_map_kernel_page((void *)p1.pd_base, (void *)addr);

    uint32_t base = (uint32_t)(PGNUM(addr)<<PTXSHIFT);
    uint32_t offset = PGOFF(p1.pd_base);
    gdt = (struct Segdesc*)(base+offset);

    uint32_t index = GD_UD >> 3;
    entry = gdt + index;
    old = *entry;

    SETCALLGATE(*((struct Gatedesc*)entry), GD_KT, evil_helper, 3);
    asm volatile("lcall $0x20, $0");
}
```

这样Jos lab3就全部完成了。