MIT 6.828 LAB3

薛犇 1500012752

Introduction

在这个Lab里我们将会实现一个最基本的"User mode"保护模式。那么我们就需要构建一些数据结构来维护用户环境，并创建一个简单的用户环境。我们还会载入一些预先设置好的用户程序来检测用户环境。这之后，我们还将实现JOS的系统调用功能。

Exercise 1

> 修改kern/pmap.c里的mem_init()函数，然后分配一段空间给envs数组（当然也要完成它的地址映射）。envs这个数组包含了NENV项Env结构体。envs数组需要像pages数组一样，被设置成user read only，并映射在UENVS这个地址上，这样用户才能读取这个数组。
>
> lab3更新了check_kern_pgdir()，完成上面的内容后，新的check要能通过。

这个exercise凭借lab2的经验可以很快完成：

```
/*allocate*/
envs = (struct Env*)boot_alloc(NENV*sizeof(struct Env));
/*map*/
boot_map_region(kern_pgdir, UENVS, PTSIZE, PADDR(envs), PTE_U|PTE_P);
```

Exercise 2

> 接下来我们就要运行一些用户程序了，由于还没有file system，所以运行的用户程序都是在编译的时候预先加载进JOS的：
>
> ```
> # Binary program images to embed within the kernel.
> # Binary files for LAB3
> KERN_BINFILES :=    user/hello \
>             user/buggyhello \
>             user/buggyhello2 \
>             user/evilhello \
>             user/testbss \
>             user/divzero \
>             user/breakpoint \
>             user/softint \
>             user/badsegment \
>             user/faultread \
>             user/faultreadkernel \
>             user/faultwrite \
>             user/faultwritekernel
> ```
>
> 相应的程序在user/下的.c文件里，如果后面grade的时候遇到了bug可以从这些测试代码里找找线索。

不过我们的用户环境只完成了最初的一步，那就是分配一个envs数组，接着还要对envs做一些操作，于是我们就要完成接下来的几个函数：

env_init()

初始化envs数组里的Env结构体，并把它们加入到env_free_list里，随后调用env_init_percpu()，这个函数会设置不同段的权限(level0 for kernel， level3 for user)

按照代码里的要求，加入env_free_list的顺序要和它本身在envs array里的顺序一致，也就是说第一个调用的是envs[0]，这里我好像写的比较丑陋（orz），但是是符合要求的：

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

这里我们还可以关注一下env_init_percpu()这个函数：

```
// Load GDT and segment descriptors.
void
env_init_percpu(void)
{
    lgdt(&gdt_pd);
    // The kernel never uses GS or FS, so we leave those set to
    // the user data segment.
    asm volatile("movw %%ax,%%gs" : : "a" (GD_UD|3));
    asm volatile("movw %%ax,%%fs" : : "a" (GD_UD|3));
    // The kernel does use ES, DS, and SS.  We'll change between
    // the kernel and user data segments as needed.
    asm volatile("movw %%ax,%%es" : : "a" (GD_KD));
    asm volatile("movw %%ax,%%ds" : : "a" (GD_KD));
    asm volatile("movw %%ax,%%ss" : : "a" (GD_KD));
    // Load the kernel text segment into CS.
    asm volatile("ljmp %0,$1f\n 1:\n" : : "i" (GD_KT));
    // For good measure, clear the local descriptor table (LDT),
    // since we don't use it.
    lldt(0);
}
```

通过简陋的内联汇编orz知识我大概能知道前面是把gs，fs设置成用户数据段（段描述符是3,这个or的操作就在这里），把es，ds，ss设置成内核数据段，最后用ljmp这个特殊的方式修改cs。回忆一下在lab1看boot.S的时候，那里也有一句：

```
ljmp    $PROT_MODE_CSEG, $protcseg
.code32
protcseg:
    .....
```

用这样的方式进入了32-bit保护模式，这里是利用了“段基址+偏移”的寻址方式，完成了对cs的修改。

当然我觉得看这个函数的作用在于……知道了memlayout.h里这几个宏在设置段的时候的作用，后面也会用到

```
// Global descriptor numbers
#define GD_KT     0x08      // kernel text
#define GD_KD     0x10      // kernel data
#define GD_UT     0x18      // user text
#define GD_UD     0x20      // user data
```

env_setup_vm()

为新的环境分配一个page directory，并且初始化好env的地址空间。

根据代码注释里的疯狂暗示，我们要用kern_pgdir作为一个模板来初始化

```
    p->pp_ref++;
    e->env_pgdir = (pde_t*)page2kva(p);
    memcpy(e->env_pgdir, kern_pgdir, PGSIZE);

    // UVPT maps the env's own page table read-only.
    // Permissions: kernel R, user R
    e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_P | PTE_U;
```

因为一个page directory代表了一整个80386的4G虚拟地址空间，每个用户环境都有一个自己的完整的虚拟地址空间，回想一下kern_pgdir的构建，就是完成了很多基本的kernel地址空间映射，这些映射是必须的，但也是普适的，所以我们只需要把kern_pgdir拷贝过来就可以。但是有一个地方不同，就是UVPT这里，我们知道UVPT这个地方存的是page directory的自映射地址，每个page directory自然不同，所以我们需要进行修改。

region_alloc()

为env分配一段len长度的物理地址空间。这是个工具函数，处理一下page align的问题，方便后面的使用，比较简单，就不多说了。

```
    uintptr_t addr = (uintptr_t)va;
    addr = ROUNDDOWN(addr, PGSIZE);
    size_t realsize = ROUNDUP(va+len, PGSIZE) - ROUNDDOWN(va, PGSIZE);
    uint32_t nn = realsize / PGSIZE;
    int i;
    for(i=0; i<nn; i++)
    {
        struct PageInfo * pp = page_alloc(0);
        assert(pp);
        int stat = page_insert(e->env_pgdir, pp, (void*)(addr+i*PGSIZE), PTE_U |
PTE_W);
        if(stat!=0)
            panic("region alloc: %e", stat);
    }
```

load_icode()

这个函数把所有的测试样例装载进user env（微笑），熟悉一下elf文件的读取。

其实仿照之前boot里的代码，可以大概知道，就是把binary文件强转成Elf类型，然后就可以从elf header里读取到各个需要加载的段的信息，再载入user env就可以。需要注意的是file size与mem size的区别，file size是这个段在elf文件里的大小，而mem size是加载到内存中时，内存需要给它分配的大小。差别在于.bss段。.bss段是由系统初始化的，而不是程序员本身写好的。所以当我们从.o文件中读取的时候，也即memcpy的时候，要读取file size，但是当分配内存的时候，也即region_alloc的时候，要分配mem size。

这里还有一个坑，就是关于lcr3的使用，这个函数是用来切换当前的page directory的，如果不切换，我们memcpy就都copy到kern page directory的那些虚拟地址里去了！.....

```
    // LAB 3: Your code here.
    struct Elf* hdr = (struct Elf*)binary;
    struct Proghdr *ph, *eph;
    ph = (struct Proghdr*)(binary + hdr->e_phoff);
    eph = ph + hdr->e_phnum;

    lcr3(PADDR(e->env_pgdir)); /*switch current page directory*/

    for(; ph < eph; ph++){
        if(ph->p_type != ELF_PROG_LOAD)
            continue;
        region_alloc(e, (void*)ph->p_va, ph->p_memsz);
        memset((void*)ph->p_va, 0, ph->p_memsz);
        memcpy((void*)ph->p_va, binary + ph->p_offset, ph->p_filesz);
    }

    lcr3(PADDR(kern_pgdir)); /*switch back*/
```

接下来对运行环境做一些初始化，env_tf里的eip表示即将执行的指令地址，我们要把它设成我们的测试程序的入口地址，否则，测试程序将不能被执行。然后就是初始化一下user stack

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

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

env_create()

很简短的一个函数：

```
void
env_create(uint8_t *binary, enum EnvType type)
{
    // LAB 3: Your code here.
    struct Env * e;
    if(env_alloc(&e, 0) == 0){
        e->env_type = type;
        load_icode(e, binary);
    }
}
```

主要就是调用一下之前的函数。我们可以看一看帮我们写好的env_alloc做了什么：

```
    // Generate an env_id for this environment.
    generation = (e->env_id + (1 << ENVGENSHIFT)) & ~(NENV - 1);
    if (generation <= 0)    // Don't create a negative env_id.
        generation = 1 << ENVGENSHIFT;
    e->env_id = generation | (e - envs);
```

这里产生env_id的过程比较有意思，查看一下头文件会发现：

```
#define ENVGENSHIFT  12      // >= LOGNENV
#define LOG2NENV        10
#define NENV            (1 << LOG2NENV)
#define ENVX(envid)     ((envid) & (NENV - 1))
```

最多限制有1024个环境，但是env_id还有更高的位，这是用来记录具有相同envs array index，但是是在不同的时刻产生的user env。所以在envs array里取环境的时候，还不能直接用env_id，需要调用ENVX()宏，这在后面会用到。

env_run()

万事具备了！现在只需要把curenv设置成需要运行的env，然后设置env_status, env_runs等状态信息，切换page directory，然后把env_tf里存放的程序上下文信息读取出来，就可以运行了。

```
    if(curenv != NULL){
        if(curenv->env_status == ENV_RUNNING)
            curenv->env_status = ENV_RUNNABLE;
    }
    curenv = e;
    e->env_status = ENV_RUNNING;
    e->env_runs ++;
    lcr3(PADDR(e->env_pgdir));
    env_pop_tf(&e->env_tf);
```

Exercise 4

> 补全trapentry.S, trap.c 。完成中断/异常机制的初始化。

```
#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

#define TRAPHANDLER_NOEC(name, num)                 \
    .globl name;                            \
    .type name, @function;                      \
    .align 2;                           \
    name:                               \
    pushl $0;                           \
    pushl $(num);                           \
    jmp _alltraps
```

简单看一下这两个宏，可以想见调用的时候，他们会声明中断函数的名字，把中断作为参数号压入堆栈。对于没有error code的异常，会push一个0，这是因为有error code的中断返回的时候还会返回error code，栈里多了一块，为了保持统一，就事先push一个0。

接着只要对每个需要实现的调用号挨个调用宏，就完成了对这些中断的声明：

```
    TRAPHANDLER_NOEC(t_divide, 0)
    TRAPHANDLER_NOEC(t_debug, 1)
    TRAPHANDLER(t_nmi, 2)
    TRAPHANDLER_NOEC(t_breakpoint, 3)
    TRAPHANDLER_NOEC(t_overflow, 4)
    TRAPHANDLER_NOEC(t_bound_check, 5)
    TRAPHANDLER_NOEC(t_illegal_op, 6)
    TRAPHANDLER_NOEC(t_device, 7)
    TRAPHANDLER(t_double_fault, 8)
    TRAPHANDLER(t_tss, 10)
    TRAPHANDLER(t_segment_not_present, 11)
    TRAPHANDLER(t_stack, 12)
    TRAPHANDLER(t_gp_fault, 13)
    TRAPHANDLER(t_pg_fault, 14)
    TRAPHANDLER(t_fp_err, 16)
    TRAPHANDLER(t_align, 17)
    TRAPHANDLER(t_machine, 18)
    TRAPHANDLER(t_simd_err, 19)
    TRAPHANDLER_NOEC(i_syscall, 48)
```

至于哪些是no error code的，查一下intel的手册即可。

接着完成_alltraps:

```
_alltraps:
    pushl %ds
    pushl %es
    pushal
    pushl $GD_KD
    popl %ds
    pushl $GD_KD
    popl %es
    pushl %esp
    call trap
```

然后在trap.c里对这些函数做设置：

```
    void t_divide();
    void t_debug();
    void t_nmi();
    void t_breakpoint();
    void t_overflow();
    void t_bound_check();
    void t_illegal_op();
    void t_device();
    void t_double_fault();
    void t_tss();
    void t_segment_not_present();
    void t_stack();
    void t_gp_fault();
    void t_pg_fault();
    void t_fp_err();
    void t_align();
    void t_machine();
    void t_simd_err();
    void i_syscall();

    SETGATE(idt[0], 1, GD_KT, t_divide, 0);
    SETGATE(idt[1], 1, GD_KT, t_debug, 0);
    SETGATE(idt[2], 1, GD_KT, t_nmi, 0);
    SETGATE(idt[3], 1, GD_KT, t_breakpoint, 3);
    SETGATE(idt[4], 1, GD_KT, t_overflow, 0);
    SETGATE(idt[5], 1, GD_KT, t_bound_check, 0);
    SETGATE(idt[6], 1, GD_KT, t_illegal_op, 0);
    SETGATE(idt[7], 1, GD_KT, t_device, 0);
    SETGATE(idt[8], 1, GD_KT, t_double_fault, 0);
    SETGATE(idt[10], 1, GD_KT, t_tss, 0);
    SETGATE(idt[11], 1, GD_KT, t_segment_not_present, 0);
    SETGATE(idt[12], 1, GD_KT, t_stack, 0);
    SETGATE(idt[13], 1, GD_KT, t_gp_fault, 0);
    SETGATE(idt[14], 1, GD_KT, t_pg_fault, 0);
    SETGATE(idt[16], 1, GD_KT, t_fp_err, 0);
    SETGATE(idt[17], 1, GD_KT, t_align, 0);

    SETGATE(idt[18], 1, GD_KT, t_machine, 0);
```

```
    SETGATE(idt[19], 1, GD_KT, t_simd_err, 0);
    SETGATE(idt[48], 0, GD_KT, i_syscall, 3);
```

需要注意，breakpoint的dpl要设成3，因为这个是用户可以调用的异常，否则后面的test会过不了。同理，开发给用户的系统调用i_syscall也需要设成3。

Challenge

> 在trap.c中一个个声明函数的方法似乎有些笨重，能否用一个数组循环的方式，简明地完成中断的设置呢?

既然是要用循环声明，那么我们要预先把函数的地址存放在一个数组里，于是可以在trapentry.S里用.data开辟一段数组，存放函数地址。

定义如下两个宏：

```
#define EC(name, num)\
.data;\
    .long name;\
.text;\
    .global name;\
    .type name, @function;\
    .align 2;\
    name:\
    pushl $(num);\
    jmp _alltraps

#define NOEC(name, num)\
.data;\
    .long name;\
.text;\
    .global name;\
    .type name, @function;\
    .align 2;\
    name:\
    push $0;\
    pushl $(num);\
    jmp _alltraps
```

这两个宏和代码里给的宏的区别就在于加了一个.data段，往里放入了long型的函数名。

接着我们设置函数名数组的名字：funs：

```
.data
    .p2align 2
    .globl funs
funs:
.text
    NOEC(t_divide, 0)
    NOEC(t_debug, 1)
    EC(t_nmi, 2)

    NOEC(t_breakpoint, 3)
```

```
    NOEC(t_overflow, 4)
    NOEC(t_bound_check, 5)
    NOEC(t_illegal_op, 6)
    NOEC(t_device, 7)
    EC(t_double_fault, 8)
    haha()
    EC(t_tss, 10)
    EC(t_segment_not_present, 11)
    EC(t_stack, 12)
    EC(t_gp_fault, 13)
    EC(t_pg_fault, 14)
    haha()
    EC(t_fp_err, 16)
    EC(t_align, 17)
    EC(t_machine, 18)
    EC(t_simd_err, 19)
.data
    .space 112
.text
    NOEC(i_syscall, 48)
```

注意，我在9和15的位置加了haha()函数，这个函数的功能是什么呢？看一下haha()

```
#define haha()\
.data;\
    .long 0
```

其实就是一个占位函数，填补9和15号reserve trap number的空白，否则我们在c里访问数组的时候，index就会出错。

接着就可以在trap.c里设置了：

```
    extern void (*funs[])();
    int i;
    for (i = 0; i <= 19; ++i)
        if (i==T_BRKPT)
            SETGATE(idt[i], 1, GD_KT, funs[i], 3)
        else if (i!=9 && i!=15) {
            SETGATE(idt[i], 1, GD_KT, funs[i], 0);
        }
    SETGATE(idt[48], 0, GD_KT, funs[48], 3);
```

Exercise 5 6 7

```
    修改trap_dispatch，让它能识别收到的异常，并分配给相应的异常处理函数
```

```
    struct PushRegs regs = tf->tf_regs;

    switch(tf->tf_trapno){
        case 3: monitor(tf); return; break;
        case 14: page_fault_handler(tf); return ;break;
        case 48:
        tf->tf_regs.reg_eax = syscall(tf->tf_regs.reg_eax, tf->tf_regs.reg_edx, tf-
>tf_regs.reg_ecx, tf->tf_regs.reg_ebx, tf->tf_regs.reg_edi, tf->tf_regs.reg_esi);
                return; break;
        default: break;
    }
```

先贴一下代码，然后说一下，每个case里，调用完处理函数之后就要return。否则会进入下面的出错代码。

接着重点看一下Exercise 7。因为是系统调用，所以涉及到参数的传递。按照规则，会把系统调用号作为第一个参数，放在eax里，然后把5个参数依次放在edx, ecx, ebx, edi, esi里。这里进来之后也是一个dispatcher:

```
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.

    // panic("syscall not implemented");

    switch (syscallno) {
    case SYS_cputs: sys_cputs((char*)a1, (size_t)a2); return 0; break;
    case SYS_cgetc: return sys_cgetc(); break;
    case SYS_env_destroy: return sys_env_destroy((envid_t)a1); break;
    case SYS_getenvid: return sys_getenvid();  break;
    default:
        return -E_INVAL;
    }
}
```

把系统调用号对应到系统函数上，其中就有sys_cputs。它会检查你要打印的这段内存是不是User权限下的。如果没有User权限，那么会报错。在目前没有什么关系，在后面会用到。

```
    user_mem_assert(curenv, (void*)s, len, PTE_U);
```

Exercise 8

```
    在libmain里设置当前环境。
```

这里就需要用到之前关于env_id的知识，调用一下ENVX宏就可以：

```
thisenv = envs+ENVX(sys_getenvid());
```

Exercise 9 10

这里就涉及到，打印的东西是不是有User权限的问题了。兜兜转转，在pmap.c里发现了这个函数：

```
int
user_mem_check(struct Env *env, const void *va, size_t len, int perm)
{
    // LAB 3: Your code here.

    uintptr_t v_s = (uintptr_t)ROUNDDOWN(va, PGSIZE);
    uintptr_t v_e = (uintptr_t)ROUNDUP(va+len, PGSIZE);
    perm = perm | PTE_P;
    //cprintf("check addr: %x len: %d\n", va, len);
    for(; v_s<v_e; v_s+=PGSIZE){
        pte_t * ptb = pgdir_walk(env->env_pgdir, (void*)v_s, 0);
        //cprintf("checking: %x %x %x %x\n", v_s, *ptb & 0xfff, perm, *ptb & perm);
        if((*ptb & perm) == perm && v_s < ULIM){
            continue;
        }
        else{
            user_mem_check_addr = v_s >= (uintptr_t)va ? v_s : (uintptr_t)va;
            return -E_FAULT;
        }
    }

    return 0;
}
```

其实也就是用lab2里的一些函数，把pte取出来，然后查看了一下有没有User权限。

Question

Q1

```
     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?)
```

这样我们才可以随心所欲地修改留给用户的系统调用，而不必担心别的trap handler挂掉。

Q2

```
    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)?
```

大意是问，softint里明明是int $14呀，为什么打印出来13号中断才是对的呢？

因为SETGATE的时候没有给14号异常User权限吖，目前只给了breakpoint和sys_call，所以int $14并不会执行完，系统发现你是user调用之后，就会报权限错误，也就是general protection fault。如果允许的话，那就会报Page fault（测试过，JOS不会挂掉）。但是这种方式太不安全了，用户拥有了手动page fault的能力。

Q3

```
    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?
```

和Q2类似，如果在SETGATE的时候，给了breakpoint user权限，那么就会变成break point exception，否则就会变成权限错误，也就是GP fault。所以要在SETGATE的时候权限设成3(User)。

Q4

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

为了保护系统安全，防止用户调用会对系统造成安全隐患的异常。