OS Lab3 Report

陆伟嘉

5140219396

这个Lab实现了操作系统对用户进程的支持，包括创立相应进程并将ELF读入进程地址空间，之后运行进程。而进程的运行会涉及到一系列的中断和系统调用，这也是在这个Lab中实现的。目前，进程只有一个，并在运行完代码之后会被销毁。

Part A: User Environments and Exception Handling

XXX　此处应为对于PartA的一个整体描述　XXX

Understanding and configuration of the Environment:

对于Environment，最关键的便是表示其区别于其他Environment的固有属性，而这些固有属性都放在一个名为Env的struct中，详见下图：

```
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_heapbrk;
    // Address space
    pde_t *env_pgdir;       // Kernel virtual address of page dir
};
```

原Env

```
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
    enum EnvStatus 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_heapbrk;
    // Address space
    pde_t *env_pgdir;       // Kernel virtual address of page dir
};
```

新Env

为了使结构体看上去更一致，我将env_status的类型也改为了enum类型。

这些结构体放在一个数组envs中，同时通过链表记录其中未分配的部分，这和之前lab的pages管理非常相似。

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

在编译时，envs变量是没有分配地址空间的，而在env.c中是在envs已经在内存中分配好的基础上进行的，所以envs的内存分配就需要在mem_init()中进行。

分配的过程只要仿照pages即可，申请相应空间（初始化在env.c中进行），同时将该内核段对应的物理空间映射到UENV位置以便用户态访问。

在做这部分的时候，发现了之前lab的一个问题。

```
page_free_list=0;
size_t i;
for (i = 1; i < IOPHYSMEM/PGSIZE; i++) {
    pages[i].pp_ref = 0;
    pages[i].pp_link = page_free_list;
    page_free_list = &pages[i];
}
for(i=PADDR(boot_alloc(0)) / PGSIZE; i < npages; i++){
    pages[i].pp_ref = 0;
    pages[i].pp_link = page_free_list;
    page_free_list = &pages[i];
}
```

在之前的lab中，计算后一部分可分配的页的时候是通过end+kern_pgdir(size)+pages(size)计算的，这在当时是正确的。但是这种写法并没有考虑到未来的拓展性，以至于在这次又新分配了envs之后，这个值就不再正确了，所以应该通过boot_alloc(0)动态确定这个界限值。

在分配好内存空间后就要对环境进行初始化使其能够运行用户程序，这就包括env结构体对初始化，环境页表的设置，环境寄存器的设置还有用户程序的载入等。

首先是对env的初始化，这在env_init()函数中实现。

```
env_init(void)
{
    // Set up envs array
    // LAB 3: Your code here.
    uint32_t i;
    struct Env *e = NULL;
    e = &envs[0];
    e->env_id = 0;
    env_free_list = e;
    for(i = 1; i < NENV; i ++){
        e = &envs[i];
        e->env_id = 0;
        env_free_list->env_link = e;
        env_free_list = e;
    }
    env_free_list = envs;
    // Per-CPU part of the initialization
    env_init_percpu();
}
```

这个过程比较简单，不再赘述。

前面是整个env系统的整体设置，现在要开始针对一个具体环境进行设置，首先是env_setup_vm()（关键）。

```
// LAB 3: Your code here.
e->env_pgdir = page2kva(p);
for(i = PDX(UTOP); i < NPDENTRIES; i++){
    e->env_pgdir[i] = kern_pgdir[i];
}
p->pp_ref ++;
// 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;
// UTOP maps first(last from memory's perspective) page of the user stack
// as described in OS assignment page, thish should be done in load_icode
// but i think it should be done here
p = page_alloc(0);
page_insert(e->env_pgdir, p, (void*)USTACKTOP - PGSIZE, PTE_U|PTE_W);

return 0;
```

这里申请了一个页作为页表，并将其与环境页表变量相关联，同时将当前环境页表映射到UVPT。而UVPT以上的部分，根据jos的设计理念，各个环境是相同的，所以只要将kern的该部分页表复制过来就可以。另外，本来应该在load_icode中实现的用户栈分配我认为应该在此处就实现，所以在最后申请了一个页用来作为用户栈。

region_alloc()函数是一个工具函数，不再赘述。

接下来是load_icode()函数：

```

// LAB 3: Your code here.
struct Proghdr *ph, *eph;
if(((struct Elf *)binary)->e_magic != ELF_MAGIC){
    panic("binary is not ELF!");
}
ph = (struct Proghdr *)(binary + ((struct Elf *)binary)->e_phoff);
eph = ph + ((struct Elf *)binary)->e_phnum;
lcr3(PADDR(e->env_pgdir));
for(; ph<eph; ph++){
    if(ph->p_type == ELF_PROG_LOAD){
        //assume that no 2 segments will hit, region_alloc will not re
        region_alloc(e, (void*)ph->p_va, ph->p_memsz);
        uint32_t i;
        for(i = 0; i < ph->p_filesz; i++){
            *((uint8_t*)ph->p_va + i) = *(binary + ph->p_offset + i);
        }
        for(i = ph->p_filesz; i < ph->p_memsz; i++){
            *((uint8_t*)ph->p_va + i) = 0;
        }
    e->env_heapbrk = ph->p_va + ph->p_memsz;
    }
}
e->env_tf.tf_eip = ((struct Elf *)binary)->e_entry;
// Now map one page for the program's initial stack
// at virtual address USTACKTOP - PGSIZE.
// LAB 3: Your code here.
// Frank : i write in env_setup_vm
lcr3(PADDR(kern_pgdir));
```

这里对于ELF文件的解析和解读主要参考了boot/main.c中的代码和wikipidea。其中 比较关键的是要切换页表，由于region_alloc()插入在了用户环境的页表中，所以在对虚拟地址进行操作时也要对应的改为该用户环境的页表，这样才能在运行时正确读到用户代码。另外，栈的申请放在了前面，所以这里就没有了。

env_create()和env_run()就是真正使用之前编写的函数的地方，照着逻辑编写即可，最后加载相应环境页表并将tf pop给硬件就可以正确运行用户代码了，在此不再赘述。

Handling Interrupts and Exceptions:

中断的部分主要通过trap.c和trapentry.s的相互配合，使得各种Exception（interrupt在这个lab中还没用到，syscall在这个lab中不通过idt途径）能够顺利进入到kernel态的处理函数，同时在内核栈（应该是ss0,esp0）中压入了正确对应的各种信息。

首先是trap_init():

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

    // LAB 3: Your code here.
    int i;
    for(i = 0;i < 20; i++){
        bool istrap = 0;//close interrupt
        struct Gatedesc *gate=&idt[i];
        uint16_t sel = GD_KT;
        uint16_t dpl;
        uint32_t off = trap_entry[i];
        if(i == 3){
            dpl = 3;
        }
        else{
            dpl = 0;
        }
        if(i!=9 && i!=15){
            SETGATE(*gate, istrap, sel, off, dpl);
        }
    }
```

通过设置门的方式，使得硬件可以从用户态切换到内核态。其中，跳转的位置和门所要求的特权级别是我们重点要设置的。就特权级别来说，系统调用和调试是用户可以直接触发的，但由于系统调用在lab中不经过idt，所以只要考虑调试即可。而跳转位置则复杂许多，需要精心设计，我们先将视线转到trapentry.s中。

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

我们要用到上图提供的macro来完成trap的路径。从中可以看出，所有trap都汇到_alltraps，而_alltraps最终由我们实现跳转到trap()，再进而分发和最终处理。

所以需要完成的分成了两部分，怎么让不同的trap都汇到_alltraps和之后该如何编写。

先看第一部分，通过调用相应的macro，可以正确地压入错误码和系统调用号，硬件也已经将旧状态压入栈中，我们只要负责让idt找到这里就可以了，也就是找到name对应的function。最简单的方式是在trap_init()中为每个trap写一遍SETGATE，但这太冗长。所以我们不把函数名直接填入idt中，而是将函数地址依次排列，组成一个函数指针数组，进而可以在idt初始化中通过数组填入，详情见下图：

```
* Lab 3: Your code here for generating entry points for the
 */
TRAPHANDLER_NOEC(tp_en0,T_DIVIDE);
TRAPHANDLER_NOEC(tp_en1,T_DEBUG);
TRAPHANDLER_NOEC(tp_en2,T_NMI);
TRAPHANDLER_NOEC(tp_en3,T_BRKPT);
TRAPHANDLER_NOEC(tp_en4,T_OFLOW);
TRAPHANDLER_NOEC(tp_en5,T_BOUND);
TRAPHANDLER_NOEC(tp_en6,T_ILLOP);
TRAPHANDLER_NOEC(tp_en7,T_DEVICE);
TRAPHANDLER(tp_en8,T_DBLFLT);
TRAPHANDLER_NOEC(tp_en9,9);
TRAPHANDLER(tp_en10,T_TSS);
TRAPHANDLER(tp_en11,T_SEGNP);
TRAPHANDLER(tp_en12,T_STACK);
TRAPHANDLER(tp_en13,T_GPFLT);
TRAPHANDLER(tp_en14,T_PGFLT);
TRAPHANDLER_NOEC(tp_en15,15);
TRAPHANDLER_NOEC(tp_en16,T_FPERR);
TRAPHANDLER(tp_en17,T_ALIGN);
TRAPHANDLER_NOEC(tp_en18,T_MCHK);
TRAPHANDLER_NOEC(tp_en19,T_SIMDERR);


.data
.globl trap_entry
trap_entry:
    .long tp_en0
    .long tp_en1
    .long tp_en2
    .long tp_en3
    .long tp_en4
    .long tp_en5
    .long tp_en6
    .long tp_en7
    .long tp_en8
    .long tp_en9
    .long tp_en10
    .long tp_en11
    .long tp_en12
    .long tp_en13
    .long tp_en14
    .long tp_en15
    .long tp_en16
    .long tp_en17
    .long tp_en18
    .long tp_en19
```

而第二部分则照着os assignment上的描述一步一步完成即可，不再赘述。这样我们就顺利地进入到了trap.c中的trap()函数。（trap()不会返回，而是通过env_run()切换回用户态）

Questions:

1. 如果采用统一的处理方式，那就无法将中断号和对应的error code信息传给内核（gate提供不同的跳转地址来区分trap，这是唯一可以灵活设置的地方，如果这里进行了统一，就没有其他参数位置能够做到这件事了）
2. 要将page fault对应的特权级别设置为0，这样用户调用会由于权限原因出现general protection exception，而不是page fault exception。如果允许用户调用这个trap，那么系统会由于没有压入error code（page fault默认是硬件压入的，但是用户调用不会压入错误码）而出错。且由于之后的参数全部错位，可能会导致page fault甚至直接宕机。

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

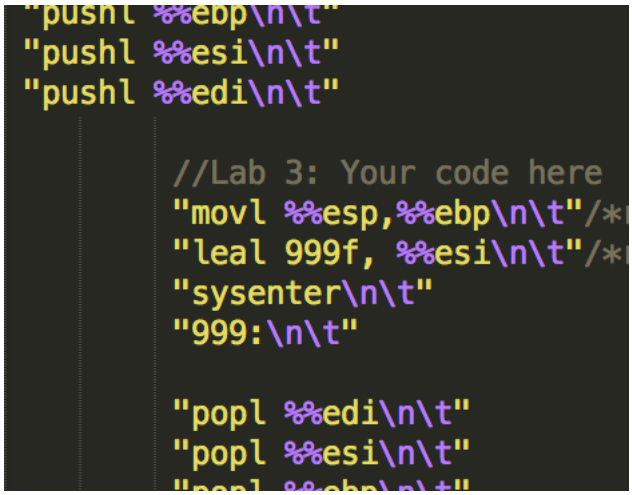
XXX　此处应为对于PartB的一个整体描述　XXX

Handling Page Faults:

安装要求的程序逻辑编写分发以及处理代码即可，不再赘述。

System calls:

jos的syscall是通过user/syscall.c调到kern/syscall.c来实现用户态到内核态到切换。先来观察user/syscall.c：



在调用sysenter之前，7个通用寄存器被压入栈中以保证syscall前后的连续（可恢复性），在代码执行流回到user之后再通过pop恢复这些寄存器。要想让前一句话所说的事得到保证，就要确保执行流可以恢复到pop且栈要一致，即记录eip和esp。通过标号和leal将恢复的eip赋给了esi，而esp赋给了ebp。这是由于只有这两个寄存器可以使用。esp在处理过程中会变化，而其他寄存器被用来传参。

接着要通过设置MSR来使执行流来到sysenter_handler：

```
wrmsr(0x174, GD_KT, 0);                 /* SYSENTER_CS_MSR  */
wrmsr(0x175, KSTACKTOP, 0);             /* SYSENTER_ESP_MSR */
wrmsr(0x176, (uint32_t)sysenter_handler, 0);    /* SYSENTER_EIP
```

```
static __inline void
wrmsr(uint32_t msr, uint32_t eax, uint32_t edx)
{
    __asm __volatile("wrmsr" : : "c" (msr), "a" (eax), "d" (edx));
}
```

设置好CS:EIP后（同时将栈切换为内核栈），执行流顺利进入到sysenter_handler，在设置好参数和段描述符后，调用syscall就进入到kern/syscall()了，而syscall()中只要根据参数进行dispatch即可。syscall返回后，换回段描述符，将要恢复到eip和esp放到指定的寄存器中调用sysexit就完成这一整个调用过程了。

```
.globl sysenter_handler;
.type sysenter_handler, @function;
.align 2;
sysenter_handler:
/*
 * Lab 3: Your code here for system call handling
 */
    pushl $0
    pushl %edi
    pushl %ebx
    pushl %ecx
    pushl %edx
    pushl %eax
    pushw $GD_KD
    pushw $GD_KD
    popw %ds
    popw %es
    call syscall
    pushw $GD_UD
    pushw $GD_UD
    popw %ds
    popw %es
    movl %ebp,%ecx
    movl %esi,%edx
    sysexit
```

在user/entry.s设置好了envs之后，在libmain()中，将第一个env赋给thisenv，从而有了对应的env之后，用户程序便可执行了。

Sbrk:

在env中加入了记录堆位置的变量后，首先要确定该变量的初始化地址，heap应该在ELF文件之后，所以在读入elf的同时，也记录所占的内存大小从而确定heap的初始地址。

```
static int
sys_sbrk(uint32_t inc)
{
    // LAB3: your code sbrk here...
    //Frank:how to deal with the collision of heap and stack (or li
    uint32_t start,end,i;
    start = curenv->env_heapbrk;
    end = start + inc;
    for(i = ROUNDUP(start,PGSIZE); i < end; i += PGSIZE){
        struct Page *p = page_alloc(0);
        page_insert(curenv->env_pgdir, p, (void*)i, PTE_U|PTE_W);
    }
    curenv->env_heapbrk = end;
    return end;
}
```

断点：

在trap_dispatch中调用monitor(),而monitor则实现对应指令的逻辑过程，值得注意的是，c指令不能直接return，由于si指令的存在，c要将FL_TF位恢复。除此之外，这个过程比较简单，不再赘述。

Questions:

3. 产生断点异常而不是保护异常，这由于gate的权限级别为3而不是0，如果是0则会产生保护异常。

4. 权限问题，有些异常不应该被软调用。

最后是利用call gate实现ring0_call()，代码如下：

```
void my_evil(){
    evil();
    gdt[GD_UD>>3] = ori;
    asm volatile("pop %ebp");
    asm volatile("lret");
}

// Invoke a given function pointer with ring0 privilege, then return to
void ring0_call(void (*fun_ptr)(void)) {
    // Here's some hints on how to achieve this.
    // 1. Store the GDT descripter to memory (sgdt instruction)
    // 2. Map GDT in user space (sys_map_kernel_page)
    // 3. Setup a CALLGATE in GDT (SETCALLGATE macro)
    // 4. Enter ring0 (lcall instruction)
    // 5. Call the function pointer
    // 6. Recover GDT entry modified in step 3 (if any)
    // 7. Leave ring0 (lret instruction)

    // Hint : use a wrapper function to call fun_ptr. Feel free
    //        to add any functions or global variables in this
    //        file if necessary.

    // Lab3 : Your Code Here
    struct Pseudodesc gdtd;
    char respage[PGSIZE];//avoid conflict between user original page an
    sgdt(&gdtd);
    sys_map_kernel_page((void*) gdtd.pd_base, (void*)gdtmap);
    uint32_t gdtpginit = ROUNDDOWN((uint32_t)gdtmap, PGSIZE);
    uint32_t gdtpgoff = gdtd.pd_base - ROUNDDOWN(gdtd.pd_base, PGSIZE);
    uint32_t gdtaddr = gdtpginit + gdtpgoff;
    gdt = (struct Segdesc *)gdtaddr;
    ori = gdt[GD_UD>>3];
    SETCALLGATE(*((struct Gatedesc*)(&gdt[GD_UD>>3])), GD_KT, my_evil,
    asm volatile("lcall $0x20, $0");
}
```

其中比较重要的是先用一个PGSIZE大小的变量用作缓冲，防止有效数据在map时被误覆盖。