JOS Lab3 实验报告

朱汉峰
1120379059

Part A(I): User Environments

这部分内容主要和系统进程相关，首先从磁盘载入ELF格式的二进制文件，然后加载到指定的内存位置，并为进程分配相应的环境，然后开始执行代码。除了进程的调度外，其它和进程相关的内容这部分都有些涉及，所以工作量还是不小的。当然如果原来就对linux系统的底层代码很熟悉的话，这个部分做起来还是挺快的。

Exercise 1

这个练习和lab 2中为页表分配空间非常类似，主要就是给envs分配空间，并进行映射，关键代码如下：

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

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

Exercise 2

和环境相关的内容主要都包含在这个练习中，从环境的初始化(类似与lab 2中页表的初始化)到ELF格式文件的载入已经执行代码都有所涉及。

env_init()

这个函数唯一需要注意的地方就是env链表和page链表的插入方向是相反的。在lab2中，我们在初始化page链表时，总是把新的page插在链表的头部，而这

里要求，新的env总是插入在链表的尾部，相关代码如下:

```
for(i = NENV-1; i >= 0; --i){
    ...

    // insert before head
    envs[i].env_link = env_free_list;
    env_free_list = envs+i;
}
```

env_setup_vm(struct Env *e)

这个函数的作用就是初始化环境的内核虚拟空间，主要和lab2中页表内容相关性比较大。涉及环境页表以及页目录的初始化。初始化页表的代码如下:

```
e->env_pgdir = page2kva(p);
memset(e->env_pgdir, 0, PGSIZE);
memmove(e->env_pgdir, kern_pgdir, PGSIZE);
```

页目录的初始化以及权限分配的相关代码其实也就一行:

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

region_alloc(struct Env *e, void *va, size_t len)

这个函数的作用是为环境分配长度为len的物理内存，并将这块内存映射到环境的虚拟地址空间va所指向的地方。这个函数没太大难度，主要还是lab2中的一些东西，就是一页一页的分配并插入就可以了。

load_icode(struct Env *e, uint8_t *binary, sizt_t size)

这个函数的作用就是将ELF格式的可执行代码载入内存，实现这个函数需要对ELF格式的文件比较了解，不过好在JOS代码里提供了相关信息，只需要阅读inc/elf.h中关于 `struct Elf` 的定义即可。如果还是不太理解的话，网上也可以搜到很多ELF相关的资料。了解了ELF格式(主要是ELF的头部信息)后，根据这个函数的相关注释，基本就可以实现它了。这里需要注意的是在载入可执行代码的时候需要把cr3指向当前环境的页目录，在载入完成后，再将cr3指向内核的页目录，相关代码如下:

```
lcr3(PADDR(e->env_pgdir));
while(ph < eph){
    region_alloc(e, (void*)ph->p_va, ph->p_memsz);
    memset((void*)ph->p_va, 0, ph->p_memsz);
    memmove((void*)ph->p_va, binary+ph->p_offset, ph->p_fil
esz);
    ph += 1;
}
lcr3(PADDR(kern_pgdir));
```

最后还需要为这个环境初始化栈空间，只需要调用上面提到的 `region_alloc` 即可，相关代码如下：

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

env_create(uint8_t *binary, size_t size, enum EnvType type)

这个函数的作用就是先初始化一个环境，然后将可执行的ELF文件载入到这个环境中。实现这个函数还是比较简单的，主要有两个步骤，首先就是调用现成的函数 `env_alloc` 来分配一个环境；然后调用上面提到 `load_icode` 将可执行文件载入到新分配的环境中即可。

env_run(struct Env *e)

这个函数的作用就是将环境e设置为当前的执行环境，由于注释写得比较全面，所以实现起来并不难。不过需要注意的是要确保e和当前的环境 `curenv` 不是同一个环境，否则会出错，相关代码如下：

```
if(e && curenv != e){
    // 根据注释实现环境切换代码
    // ...
}
```

Part A(II): Exception Handling

这部分主要和系统的中断和异常处理相关，主要涉及中断向量的建立，相关知识在这里有所介绍。

Exercise 4

这个练习的目录就是建立中断向量，并实现_alltraps函数，使得所有中断和异常都交给 `trap` 函数来处理。所谓中断向量也就是为各个中断定义相关的处理函数，并将中断号和中断处理函数关联起来。这主要包含两部分代码，第一部分代码位于kern/trapentry.S中，这里面主要定义各种中断的处理函数，它的处理逻辑很简单，直接跳转到 `_alltraps` 进行处理，定义处理函数的代码如下： `t_divide` 为函数名， `T_DIVIDE` 则是中断号。

```
TRAPHANDLER_NOEC(t_divide, T_DIVIDE);
...
```

将中断号和处理函数进行关联的代码位于kern/trap.c的 `trap_init` 中，相关代码如下：

```
extern void t_divide();
...

SETGATE(idt[T_DIVIDE], 0, GD_KT, t_divide, 0);
...
```

`_alltraps` 的作用就是将中断交给kern/trap.c中的 `trap` 函数来处理，相关代码如下：

```
_alltraps:
    pushl   %ds
    pushl   %es
    pushal

    movw    $GD_KD, %ax
    movw    %ax, %ds
    movw    %ax, %es

    pushl   %esp
    call    trap
```

Question 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?)

我觉得这些中断不能共用一个处理函数，如果共用一个处理函数的话，对于error code的处理会比较麻烦，因为并不是所有的中断都有error code；此外，如果共用的话，当前的中断号将无法被简单记录，可能会影响后续对中断号的使用。

Question 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)?

这是由中断的权限引起的，虽然user/softint调用的是int 14，但我们在IDT中设置的int 14的特权界别是0，也就是说只有内核才能产生该中断，所以，当用户程序触发这个中断时会使得CPU产生另一个中断进行保护，这个保护中断就是int 13, general protection。

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

这部分主要是利用Part A写好的中断处理函数来处理Page Faults和Breakpoint；此外这部分还涉及到利用 `sysenter/sysexit` 来调用system call。

Exercise 5 & 6

这两个练习比较简单，只需要在kern/trap.c的 `trap_dispatch` 函数中，对中断号进行判断，如果中断号是 `T_PGFLT` 则调用 `page_fault_handler` ，如果是 `T_BRKPT` 则调用 `monitor` ，否则不进行特殊处理。相关代码如下：

```
switch(tf->tf_trapno){
case T_PGFLT:
    page_fault_handler(tf);
    return;
case T_BRKPT:
    monitor(tf);
    return;
}
```

Question 3

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?

这个问题和Question 2有点类似，到底是产生break point exception 还是general protection fault取决于你给int 3在IDT中所设置的权限。如果你给int 3设置的权限是0，那么只有内核才能产生该中断，如果用户程序触发int 3，就会产生general protection fault来进行保护；如果设置的权限是3，用户程序就会有权限来产生这个中断，那么就会产生break point exception。所以为了让用户程序能够产生break point exception，我将int 3的权限设置为3。

Question 4

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

我觉得这个机制的关键在于对内核的保护，只有足够的权限才能触发相应的中断，这样可以使得系统更加健壮，减少潜在的风险。在user/softint中，由于用户程序没有足够的权限，所以不能触发page fault中断，从而使得CPU产生general protection fault来进行保护。如果没有这个机制，那么任何用户程序都可以很容易就将内核搞坏。

Exercise 7

这个练习应该算是这个lab中最难的部分，它涉及的东西比较多，不过在理清楚执行顺序之后，会容易不少。从函数调用的角度来看，当用户程序调用某个

system call时，大致的函数调用流程如下：

> user program --> lib/syscall.c syscall --> kern/trapentry.S
> sysenter_handler-->
> kern/syscall.c syscall_wrapper --> kern/syscall.c syscall

由上面的流程我们可以看到，用sysenter/sysexit来实现系统调用还是比较复杂的，不过我们可以对这个流程进行拆分，各个击破。首先要解决的问题就是，如何让lib/syscall.c中的 `syscall` 调用 kern/trapentry.S中的 `sysenter_handler` 。首先需要将 `sysenter` 指令和 `sysenter_handler` 函数进行绑定，这部分代码位于kern/trap.c的 `trap_init` 函数中，如下：

```
wrmsr(0x174, GD_KT, 0);
wrmsr(0x175, KSTACKTOP, 0);
wrmsr(0x176, (uint32_t)&sysenter_handler, 0);
```

此外，我们还需要在lib/syscall.c的 `syscall` 函数中调用 `sysenter` 指令，从而触发 `sysenter_handler` 的执行，相关代码如下：

```
"movl %%esp, %%ebp\n\t"
"leal after_sysenter%=, %%esi\n\t"
"sysenter\n\t"
"after_sysenter%=:\n\t"
```

接下来，只需要在 `sysenter_handler` 中调用 `syscall_wrapper` , 然后在 `syscall_wrapper` 中调用 `syscall` 就可以了， `syscall_wrapper` 的定义如下：

```
void syscall_wrapper(struct Trapframe *tf){
    curenv->env_tf = *tf;
    tf->tf_flags.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,
                0);
}
```

进入kern/syscall.c中的 `syscall` 函数就比较容易处理了，在这个函数中，只需要根据中断号，调用对应的处理函数即可，如果无法处理，则返回 `-E_INVAL`；

Exercise 8

这个练习比较简单，只需要在lib/libmain.c的 `libmain` 中添加一行代码来设置 `thisenv` 就可以了，相关代码如下：

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

Exercise 9

这个练习可以分成两部分，第一部分就是如果在kernel mode下发生page fault，直接让系统崩溃，这部分代码位于kern/trap.c 的 `page_fault_handler` 中，相关代码如下：

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

user_mem_check(struct Env *env, const void *va, size_t len, int perm)

这就是这个练习的第二部分，这个函数的作用就是环境env时候有权限可以访问[va, va+len)范围之内的虚拟地址空间。一个用户程序可以访问的虚拟地址，需要满足两个条件：

1. 该虚拟地址低于 `ULIM`
2. 有足够的权限访问相应的页目录

实现的这个函数的时候只要循环遍历需要访问的虚拟空间就可以了，只要违反上面任意一个条件，则立即返回 `-E_FAULT`。在具体写代码时，有一个地方需要注意，那就是va的对其问题，相关代码如下：

```
for(idx = lva; idx <= rva; idx += PGSIZE){
    // 检查 idx 是否低于 ULIM
    // ...

    pre = pgdir_walk(env->env_pgdir, (void*)idx, 0);

    // 检查pte是否有足够的权限
    // ...

    idx = ROUNDDOWN(idx, PGSIZE);
}
```

Exercise 11

这个练习刚看到lab3网页上的介绍时，感觉挺难，不过看到具体的函数时，由于有很详细的解释，所以并不是太难。这个练习的关键就是使用了一个wrapper函数，相关的代码主要有两部分，一部分位于函数 `ring0_call` 中，如下：

```
SETCALLGATE(*((struct Gatedesc *)gdt_entry), GD_KT, evil_wr
apper, 3);
```

还有一部分就是 `evil_wrapper` 函数的定义，具体代码如下：

```
void evil_wrapper(){
    evil();
    *gdt_entry = saved_gdtdesc;

    asm volatile("popl %ebp");
    asm volatile("lret");
}
```