Lab 3

简介

总的来说，这个 lab 的难度比前两个要大得多。在写 lab3 的时候，思路还是比较清楚的，主要就是建立 environment（即进程），加载代码，创建 IDT 表，完成系统调用几部分，但是这个 lab 比较烦的地方是你有时候根本不知道怎么写，那么你就需要去查 intel 手册。

从主要的内容来看，这个 lab 可以分成三个部分：1）建立 environment；2）建立 IDT 和 handler；3）进行 system call。

建立 environment

首先谈谈 environment 这个概念，在这个 lab 里面，与进程的概念差不多，什么叫做进程？我们的每个进程都有一个虚拟的地址空间，但是虚拟空间是没有办法 allocated，物理空间才能够 allocated，因此所谓的虚拟空间实际上就是页表，虚构的地址（只要你乐意，你可以随便给虚拟地址），只要不超过 4G 就可以来，然后通过页表映射到物理地址，因此对于一个进程而言，它需要用来表示的只需要 1）页表；2）寄存器和各种状态；3）物理空间。因此这个 lab 中，用 env 这个数据结构表示进程完全 enough。

然后说说建立过程？1）我们建立一系列的进程，用 envs 这个数组来表示，建立好了之后，用 env_init()初始化进程数组，建立空闲的进程链表 env_free_list；2）用 env_create 建立进程；3）用 env_run()加载进程。这样一个进程就加载完成来。

最后我们来说说这些代码是怎样运作的。

1） 创建进程：envs = (struct Env*)boot_alloc(NENV*sizeof(struct Env));

然后 map 一下就可以了：
boot_map_region(kern_pgdir,UENVS,ROUNDUP(NENV*sizeof(struct Env),PGSIZE),PADDR(envs),PTE_U|PTE_P);

之后就是初始化，也比较简单：

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

注意一下倒序建立就可以了。

2） 建立进程。用 env_create()建立，下面我们来看看：

1.

```
struct Env* e;
int result=env_alloc(&e,0);
if(result!=0)
{
        panic("env_create failed\n");
        return;
}
load_icode(e,binary,size);
e->env_type=type;
```

代码比较简单，首先调用 env_alloc 建立一个 environment，然后用 load_icode()加载代码.

2.

env_alloc()已经实现来，但是他用到的 env_setup_vm()并没有实现，它的逻辑很简单，找一个 free 的 env，然后建立 kernel 部分（env_setup_vm()）就可以了。

3.

env_setup_vm()代码如下所示：

```
        p->pp_ref++;
```

```
e->env_pgdir=(pde_t*)page2kva(p);
for(i=PDX(UTOP);i<NPDENTRIES;i++)
{
	e->env_pgdir[i]=kern_pgdir[i];
}
// 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;

return 0;
```

建立一个 pde，然后让 kernel 部分的 pde 中的 pte 指向 kern_pgdir 中对应的 pte（即指向同一个二级页表）就可以了。

4.

至于 load_icode():

```
struct Proghdr* ph,*eph;
struct Elf* prog=(struct Elf*)binary;
if(prog->e_magic!=ELF_MAGIC)
{
	panic("load_icode failed\n");
}
ph=(struct Proghdr*)((uint8_t*)prog+prog->e_phoff);
eph=ph+prog->e_phnum;

lcr3(PADDR(e->env_pgdir));
e->env_tf.tf_eip=prog->e_entry;

for(;ph<eph;ph++)
{
if(ph->p_type == ELF_PROG_LOAD)
{
	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));
}
}
```

```
lcr3(PADDR(kern_pgdir));
// Now map one page for the program's initial stack
// at virtual address USTACKTOP - PGSIZE.
region_alloc(e,(void*)(USTACKTOP-PGSIZE),PGSIZE);
```

因为加载的是elf格式的，所以照抄boot时加载kernel时的代码就可以了，先用region_alloc()开辟一块内存，然后memmove直接加载进去就可以了，中间的for循环就是干这事的，要注意的是，因为现在是在用户进程开辟内存，因此要lcr3()加载该进程的页表，同时把程序的入口用 `e->env_tf.tf_eip=prog->e_entry;` 拿到就可以了。

5.

region_alloc():

```
uint32_t start=(uint32_t)ROUNDDOWN(va,PGSIZE);
uint32_t end=(uint32_t)ROUNDUP(va+len,PGSIZE);
struct Page* cur=NULL;
uint32_t i=0;
for(i=start;i<end;i+=PGSIZE)
{
cur=page_alloc(ALLOC_ZERO);
if(cur==NULL)
{
	panic("region_alloc failed\n");
}
else
{
	if(page_insert(e->env_pgdir,cur,(void*)i,
	PTE_U|PTE_W))
		panic("region_alloc failed\n");
}
}
e->brk=end;
```

主要就是先开辟物理页（用page_alloc）,然后映射到虚拟内存区域就可以了，代码比较简单，都是lab2的函数。brk是空间top位置的指针。

3） 加载进程。用env_run()就可以了：

```
if(curenv!=e)
{
if(curenv&&curenv->env_status==ENV_RUNNING)
	curenv->env_status=ENV_RUNNABLE;  //更改原进程状态
curenv=e;                //更改
curenv->env_status=ENV_RUNNING; //更改新进程状态
```

```
curenv->env_runs++;
lcr3(PADDR(curenv->env_pgdir)); //切换页表
}
env_pop_tf(&curenv->env_tf);
```

主要就是将之前进程的状态进行更改，切换新的进程，记住，要更改页表。

讲到这里，environment 部分就讲完了，接下来就是 IDT 部分了。

IDT 和 handler

1）建立 IDT 表比较简单，在 trap_init()中建立 IDT 表的 entry 就可以了。代码如下：

```
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();

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);
```

代码简单用 SETGATE 建立就可以了。

2）建立好了 IDT 表之后，怎么让调用呢？在硬件或者软件发出 interrupt 或者 exception 后，系统会根据相应的 entry 调用相应的函数，即为上面的 entry0—entry19，它们的函数体：

```
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 );
```

TRAPHANDLER_NOEC 和 TRAPHANDLER 的区别是前者会 push 0，至于到底用哪个查看 intel 手册（太坑了），他们都会执行 jmp _alltraps

它的代码是：

```
_alltraps:
pushw $0    #uint16_t padding
pushw %ds
pushw $0    #uint16_t padding
```

```
pushw %es
pushal

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

pushl %esp
call trap
```

前面 push 压入栈其实是一个 trapframe，将该结构的第一部分倒过来 push 就是上面的顺序，后面的 mov 按照 hint 写就可以了，最后 pushl %esp 实际上就是 trapframe 的指针 tf，然后调用 trap 函数，正好这个函数的参数就是 trapframe 的指针，与上面一致，在这个函数中它会调用 trap_dispath()函数，这个函数是我们要补全的，补全部分为：

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

}
```

主要处理上面 3 个中断，在 exercise 中要实现 T_BPKPT 中断，然而 debug 的时候 T_DEBUG 信号会产生，如果不写的话，就会漏掉一部分处理（题目没说，调试半天才发现，(▼-▼)）。首先对于 page fault 的处理：

```
if(!(tf->tf_cs&0x3))
{
	panic("kern page fault\n");
}
```

只需要判断是不是 kernel 的 page fault，如果不是，接下来的部分已经写好了，是的话，报错就可以了。

对于 debug，我们调用 monitor（ ）函数，这个函数在 lab1 中就出现了，主要是敲命令行执行，要求实现的 si，x，c 就是属于这一部分，要实现这部分，首先注册这些命令：

```
static struct Command commands[] = {
{ "help", "Display this list of commands", mon_help },
{ "kerninfo", "Display information about the kernel", mon_kerninfo },
{ "backtrace","stack backtrace",mon_backtrace },
{ "time help","Display this list of commands",time_help },
{ "time kerninfo","Display information about the kernel",time_kerninfo },
{ "time backtrace","stack backtrace",time_backtrace },
{"c","continue in debug",mon_continue},
{"si","step by step in debug",mon_step},
{"x","display memory",mon_x}
};
```

加入要执行的指令就可以了，他们会分别执行注册的函数。

```
int mon_continue(int argc, char **argv, struct Trapframe *tf)
{
tf->tf_eflags = (tf->tf_eflags)&(~FL_TF);
env_run(curenv);
return 0;
}
```

将 eflags 中的 Trap Flag 标识为设置为 0，cpu 就不会进入单步执行模式。

```
int mon_step(int argc, char **argv, struct Trapframe *tf)
{
tf->tf_eflags = (tf->tf_eflags)|(FL_TF);

cprintf("tf_eip=0x%x\n", tf->tf_eip);
env_run(curenv);
return 0;
}
```

将 eflags 中的 Trap Flag 标识为设置为 1，cpu 就会进入单步执行模式。

```
int mon_x(int argc, char **argv, struct Trapframe *tf)
{
int addr = strtol(argv[1], NULL, 16);

int get_val;
asm volatile("movl (%0), %0" : "=r" (get_val) : "r" (addr));

cprintf("%d\n", get_val);
return 0;
}
```

根据传入的参数获取地址，用内联汇编获取地址的值，打印出来就可以了。

讲到这里，第二部分就完成了，第二部分总的来说难度不是很大，主要就是trapEntry.S 中相应的编写，其他部分很好理解。下面第三部分系统调用。

System call

与 trap 相似，首先在 trap_init()里面

```
extern void sysenter_handler();
wrmsr(0x174, GD_KT, 0);
wrmsr(0x175, KSTACKTOP, 0);
wrmsr(0x176, sysenter_handler, 0);
```

wrmsr的源码看x86.h文件就可以了（为什么这么写，看wiki手册去吧），在inc/syscall.c中有一个syscall，当发生system call时，就会转到该函数中，该函数

```
"movl %%esp,%%ebp\n\t"
          "leal 1f, %%esi\n\t"
          "sysenter\n\t"
          "1:\n\t"
```

他会执行 sysenter 指令，就会 sysenter_handler 中，代码如下：

```
pushl $GD_UD|3
pushl %ebp
pushfl
pushl $GD_UT|3
pushl %esi
pushl $0
pushl $0
pushl %ds
pushl %es
pushal

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

pushl %esp
call my_syscall

popl %esp
popal
popl %es
popl %ds
movl %ebp, %ecx
movl %esi, %edx
```

```
sysexit
```

hint 说 push 一个 trapframe 的结构，所以我这样做了，最后 push %esp 压入一个 trapframe 的指针，但是我发现要 call 的函数根本就没有这样的参数，所以我自己写了一个 my_syscall:

```
void
my_syscall(struct Trapframe *tf){
curenv->env_tf = *tf;
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,
                        0);
return;
}
```

其实就是调用 syscall 函数，按照它传参寄存器的要求写入参数，调用 syscall。

```
switch (syscallno){
       case SYS_cputs:
              sys_cputs((const char*) a1,a2);
              return 0;
       case SYS_cgetc:
              return sys_cgetc();
       case SYS_getenvid:
              return sys_getenvid();
       case SYS_env_destroy:
              return sys_env_destroy(a1);
       case SYS_map_kernel_page:
              return sys_map_kernel_page((void *)a1, (void *)a2);
       case SYS_sbrk:
              return sys_sbrk(a1);
       case NSYSCALLS:
              return 0;
       default:
              return -E_INVAL;
}
```

按照定义好的（inc/syscall.h）信号类型进行处理。

```
sys_cputs(const char *s, size_t len)
{
```

```
// Check that the user has permission to read memory [s, s+len).
// Destroy the environment if not.

// LAB 3: Your code here.
user_mem_assert(curenv,(void*)s,len,PTE_U|PTE_P);
// Print the string supplied by the user.
cprintf("%.*s", len, s);
}
```

调用 user_mem_assert()判断传进来的地址是不是合法的，合法的打印即可。对于 user_mem_assert()它调用 user_mem_check():

```
user_mem_check(struct Env *env, const void *va, size_t len, int perm)
{
// LAB 3: Your code here.
uint32_t start=(uint32_t)va;
uint32_t end=(uint32_t)(va+len);
perm=perm|PTE_U|PTE_P;

uint32_t i=0;
pte_t* pte;
for(i=start;i<end;i++)
{
	if(i>=(uint32_t)ULIM)
	{
		user_mem_check_addr=i;
		return -E_FAULT;
	}
	pte = pgdir_walk (env->env_pgdir,(void*)i, 0);
	if(pte==NULL||((*pte&perm)!=perm))
	{
		user_mem_check_addr=i;
		return -E_FAULT;
	}
}
return 0;
}
```

主要就是判断两个方面：1）是否越界，即>=ULIM，第二个判断该虚拟地址对应的 pte 上的权限是否与传进来的一样。代码逻辑很简单。

接下来的那些 system call 中要求实现的只有 sys_brk():

```
sys_sbrk(uint32_t inc)
{
// LAB3: your code sbrk here...
```

```
region_alloc(curenv,(void*)(curenv->brk), inc);
return curenv->brk;
}
```

这个函数主要是扩展用户空间，所以在 brk（用户空间的 top）上 alloc inc 长度就可以了，返回 brk（brk 在 region_alloc 中会被改成 top 位置）；

到这里位置，system call 的部分就完成了，最后是一个 hack kernel 的小代码：

```
static void my_evil()
{
call_func();
asm volatile("leave\n\t"
           "lret");
}
// Invoke a given function pointer with ring0 privilege, then return to ring3
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;
struct Gatedesc* gdt,entry;
char* gdt_addr=(char*)0x80000000;

sgdt(&gdtd);
sys_map_kernel_page((char*)gdtd.pd_base,gdt_addr);
gdt=(struct Gatedesc*)(gdt_addr+(gdtd.pd_base%PGSIZE));

call_func=fun_ptr;
entry=gdt[5];
SETCALLGATE(((struct Gatedesc volatile*)gdt)[5],GD_KT,my_evil,
3);

asm volatile("lcall %0,$0": : "i"(GD_TSS0));
gdt[5]=entry;
```

```
}
```

主要逻辑是1）将gdt表映射到用户虚拟内存地址

```
sys_map_kernel_page((char*)gdtd.pd_base,gdt_addr);
```

2）找到 gdt 表的入口：

```
gdt=(struct Gatedesc*)(gdt_addr+(gdtd.pd_base%PGSIZE));
```

3）将 gdt 表的某一项 entry 设置成要执行的函数

```
SETCALLGATE(((struct Gatedesc volatile*)gdt)[5],GD_KT,my_evil,
3);
```

你会问我，为什么是第 5 项？我也不知道为什么，我从 0，1，。。。一个个换。

4）lcall跳转就可以了`asm volatile("lcall %0,$0": : "i"(GD_TSS0));`

记住执行完了之后，将第 5 项还原。

总结

Lab3 到这里就结束了，说实话还是有些不懂得地方，但是不妨碍写代码，感觉要看的东西不少，至于最后那个 hack kernel 为什么是第 5 项，我还是不清楚，助教能解答就再好不过来。总的来说，这个 lab 好花时间，说起花时间，就想起网站上面说 environment 配好之后就可以运行到进入 user 代码那个位置，我写完那个部分之后到不了，于是我调试发现无缘无故的 load_icode 会跳回去，但是我发现代码并没有问题，所以我就做下去，结果写好 IDT 之后就可以正常了，我说这尼玛也太坑了，浪费我不少时间(▼-▼)。

Lab 就说到这里吧，作业还有好多，还有个编译器。。。。。