Operating System MIT 6.828 JOS Lab3 Report

Computer Science
ChenHao(1100012776)

2013 年 10 月 31 日

目录

1 Preface

大致使用了 30 小时，完成了所有的 Exercise 和 Challenge

2 Part A: User Environments and Exception Handling

2.1 Exercise 1

分配物理内存和创建虚拟内存映射给 envc，类似 Lab2 即可。

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

// ... ...

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

2.2 Exercise 2

pmap 只对内核进行了内存管理，而对于每个进程，都用有一个独立的内存空间，并且每个进程看起来都拥有整个内存空间，因此我们需要对进程也进行虚拟内存的管理，以及管理如何创建进程和进程的切换的问题。

2.2.1 env_init

env_init 类似 page_init，用来初始化 NENV 个进程管理结构，并且用单向链表来组织空闲的 Env。其中要求 env_free_list 初始指向 &envs[0]。似乎这个的原因是在 init.c 中其会执行 envs[0]。

```
void
env_init(void)
{
    // Set up envs array
    // LAB 3: Your code here.
    uint32_t i;
    env_free_list = envs;
    for (i = 0; i < NENV; i++) {
        envs[i].env_id = 0;
        envs[i].env_status = ENV_FREE;
        if (i + 1 != NENV)
            envs[i].env_link = envs + (i + 1);
        else
            envs[i].env_link = NULL;
    }
```

```
    // Per-CPU part of the initialization
    env_init_percpu();
}
```

2.2.2 env_setup_vm

env_setup_vm 分配进程独立的 Page Directory，即创建该进程的页目录。对于高于 UTOP 的虚拟地址 PDE 应与 Kernel 的页目录，对于低于 UTOP 的位置需要清 0，这部分就是真正用户进程使用的页目录条目。

为什么进程的页目录高于 UTOP 的虚拟地址的映射和 Kernel 的页目录一致？

我觉得原因在于在内核管理进程的时候，在需用对进程使用的内存进行访问或者使用的时候就需要改用进程的 Page Directory，但是同时还需要使用内核的代码或数据，因此保持一直可以保证这一点，不会造成错误和不必要的麻烦。而由于高于 UTOP 的虚拟地址的权限都是 kernel 权限的，因此在用户态的情况可以防止用户进行访问和修改，而且对于 UTOP 以上的内存对于用户进程是不允许访问的，这部分对于用户进程来说是不会使用的。

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

    // Allocate a page for the page directory
    if (!(p = page_alloc(ALLOC_ZERO)))
        return -E_NO_MEM;

    p->pp_ref++;
    e->env_pgdir = (pde_t *)page2kva(p);
    memcpy(e->env_pgdir, kern_pgdir, PGSIZE);
    memset(e->env_pgdir, 0, PDX(UTOP) * sizeof(pde_t));

    // 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;
}
```

2.2.3 region_alloc

region_alloc 用于为进程分配物理内存，因此应该使用对应进程的页目录和页表。

```
static void
region_alloc(struct Env *e, void *va, size_t len)
{
    uint32_t addr = (uint32_t)ROUNDDOWN(va, PGSIZE);
    uint32_t end  = (uint32_t)ROUNDUP(va + len, PGSIZE);
    struct PageInfo *pg;
    // cprintf("region_alloc: %u %u\n", addr, end);
```

```
    int r;
    // cprintf("region_alloc: %u %u\n", addr, end);
    for ( ; addr != end; addr += PGSIZE) {
        pg = page_alloc(1);
        if (pg == NULL) {
            panic("region_alloc : can't alloc page\n");
        } else {
            r = page_insert(e->env_pgdir, pg, (void *)addr, PTE_U | PTE_W);
            if (r != 0) {
                panic("/kern/env.c/region_alloc : %e\n", r);
            }
        }
    }
    return;
}
```

2.2.4 load_icode

load_icode 将目标文件放入内存中，存放的虚拟内存的位置由目标文件指定。这个函数有两个需要注意的地方，第一个是首先使用 region_alloc 分配对应虚拟地址的内存，而这个映射仅在该进程的页表中存在，在内核中是不存在的，因此在 memcpy 和 memset 的时候需要使用的该进程的页目录，而不应该使用内核的页目录。这个地方非常阴险，我一开始就掉进了这个陷阱中。

第二个需要注意的地方就是需要将 elf->e_entry 即目标文件的入口放入进程环境的 eip 中。

```
static void
load_icode(struct Env *e, uint8_t *binary, size_t size)
{
    struct Elf * elf = (struct Elf *)binary;
    if (elf->e_magic != ELF_MAGIC) {
        panic("error elf magic number\n");
    }
    struct Proghdr *ph, *eph;
    ph = (struct Proghdr *) ((uint8_t *) elf + elf->e_phoff);
    eph = ph + elf->e_phnum;

    lcr3(PADDR(e->env_pgdir));
    for (; ph < eph; ph++) {
        if (ph->p_type == ELF_PROG_LOAD) {
            region_alloc(e, (void *)ph->p_va, ph->p_memsz);
            memcpy((void *)ph->p_va, binary + ph->p_offset, ph->p_filesz);
            memset((void *)(ph->p_va) + ph->p_filesz, 0, ph->p_memsz - ph->
                p_filesz);
        }
    }
    e->env_tf.tf_eip = elf->e_entry;

    lcr3(PADDR(kern_pgdir));
    region_alloc(e, (void *)(USTACKTOP - PGSIZE), PGSIZE);

    return;
}
```

2.2.5 env_create

这个函数需要做就是将代码导入内存中，需要分两布：第一创建进程的地址空间的页目录以及设置环境变量，第二是将目标文件的代码导入内存中。

```
void
env_create(uint8_t *binary, size_t size, enum EnvType type)
{
    struct Env * e;
    int r = env_alloc(&e, 0);
    if (r < 0) {
        panic("env_create: %e\n", r);
    }
    load_icode(e, binary, size);
    e->env_type = type;
    return;
}
```

2.2.6 env_run

只需要进行切换一下即可。遗留问题如果 curenv 的状态为别的状态怎么办？之后回来再来看好了。

```
void
env_run(struct Env *e)
{
    if (curenv != NULL) {
        // context switch
        if (curenv->env_status == ENV_RUNNING) {
            curenv->env_status = ENV_RUNNABLE;
        }
        // how about other env_status ? e.g. like ENV_DYING ?
    }
    curenv = e;
    curenv->env_status = ENV_RUNNING;
    curenv->env_runs++;

    lcr3(PADDR(curenv->env_pgdir));

    env_pop_tf(&curenv->env_tf);
    panic("env_run not yet implemented");
}
```

gdb 得到结果顺利到达 int $0x30 处。

2.3 Exercise 3

2.4 Exercise 4

由 IA-32 手册知是否需要 Error Code 的情况：

Interrupt	ID	Error Code
divide error	0	N
debug exception	1	N
non-maskable interrupt	2	N
breakpoint	3	N
overflow	4	N
bounds check	5	N
illegal opcode	6	N
device not available	7	N
double fault	8	Y
invalid task switch segment	10	Y
segment not present	11	Y
stack exception	12	Y
general protection fault	13	Y
page fault	14	Y
floating point error	16	N
aligment check	17	Y
machine check	18	N
SIMD floating point error	19	N

2.4.1 trapentry.S

trapentry.S 就是设置各种终端的入口，以及进入中断后队进程状态的保护。于是在 trapentry.S 的.text 段中设置对应入口的汇编即可，对于状态的保护即在栈中建 Trapframe, 根据 inc/trap.h 中的 Trapframe 结构，存放相应的寄存器，并将 GD_KD 导入%ds 和%es 中，保存%esp 执行 trap() 函数。

```
.text
 /*
 * Lab 3: Your code here for generating entry points for the different traps.
 */
    TRAPHANDLER_NOEC(vec0, T_DIVIDE)
    TRAPHANDLER_NOEC(vec1, T_DEBUG)
    TRAPHANDLER_NOEC(vec2, T_NMI)
    TRAPHANDLER_NOEC(vec3, T_BRKPT)
    TRAPHANDLER_NOEC(vec4, T_OFLOW)

    TRAPHANDLER_NOEC(vec6, T_BOUND)
    TRAPHANDLER_NOEC(vec7, T_DEVICE)
    TRAPHANDLER(vec8, T_DBLFLT)

    TRAPHANDLER(vec10, T_TSS)
    TRAPHANDLER(vec11, T_SEGNP)
    TRAPHANDLER(vec12, T_STACK)
```

```
    TRAPHANDLER(vec13, T_GPFLT)
    TRAPHANDLER(vec14, T_PGFLT)

    TRAPHANDLER_NOEC(vec16, T_FPERR)
    TRAPHANDLER(vec17, T_ALIGN)
    TRAPHANDLER_NOEC(vec18, T_MCHK)
    TRAPHANDLER_NOEC(vec19, T_SIMDERR)

/*
 * Lab 3: Your code here for _alltraps
 */
_alltraps:
    pushl %ds
    pushl %es
    pushal

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

    pushl %esp
    call trap
```

2.4.2 trap_init

trap_init 为初始化 IDT 表，并将表头导入 IDTR 中。IDT 表中的项如下图所示：

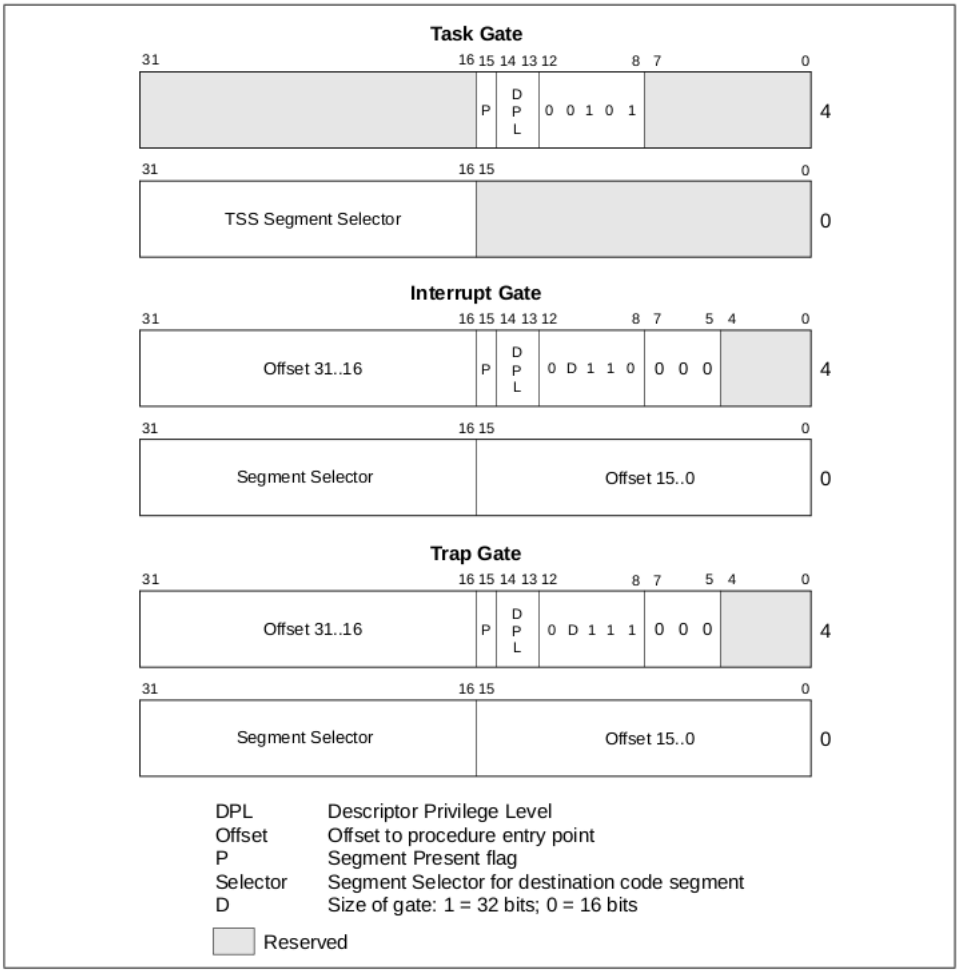


Figure 5-2. IDT Gate Descriptors

因此将对应项传入 SETGATE 即可。

```
// In inc/mmu.h
// Set up a normal interrupt/trap gate descriptor.
// - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
    //   see section 9.6.1.3 of the i386 reference: "The difference between
    //   an interrupt gate and a trap gate is in the effect on IF (the
    //   interrupt-enable flag). An interrupt that vectors through an
    //   interrupt gate resets IF, thereby preventing other interrupts from
    //   interfering with the current interrupt handler. A subsequent IRET
    //   instruction restores IF to the value in the EFLAGS image on the
    //   stack. An interrupt through a trap gate does not change IF."
// - sel: Code segment selector for interrupt/trap handler
// - off: Offset in code segment for interrupt/trap handler
```

```
// - dpl: Descriptor Privilege Level -
//     the privilege level required for software to invoke
//     this interrupt/trap gate explicitly using an int instruction.
#define SETGATE(gate, istrap, sel, off, dpl) ...
// ------------------------------------------------------------------------------

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

    // LAB 3: Your code here.
    void vec0();
    void vec1();
    void vec2();
    void vec3();
    void vec4();
    void vec6();
    void vec7();
    void vec8();
    void vec10();
    void vec11();
    void vec12();
    void vec13();
    void vec14();
    void vec16();
    void vec17();
    void vec18();
    void vec19();

    SETGATE(idt[0], 0, GD_KT, vec0, 0);
    SETGATE(idt[1], 0, GD_KT, vec1, 0);
    SETGATE(idt[2], 0, GD_KT, vec2, 0);
    SETGATE(idt[3], 0, GD_KT, vec3, 0);
    SETGATE(idt[4], 0, GD_KT, vec4, 0);

    SETGATE(idt[6], 0, GD_KT, vec6, 0);
    SETGATE(idt[7], 0, GD_KT, vec7, 0);
    SETGATE(idt[8], 0, GD_KT, vec8, 0);
    SETGATE(idt[10], 0, GD_KT, vec10, 0);
    SETGATE(idt[11], 0, GD_KT, vec11, 0);
    SETGATE(idt[12], 0, GD_KT, vec12, 0);
    SETGATE(idt[13], 0, GD_KT, vec13, 0);
    SETGATE(idt[14], 0, GD_KT, vec14, 0);

    SETGATE(idt[16], 0, GD_KT, vec16, 0);
    SETGATE(idt[17], 0, GD_KT, vec17, 0);
    SETGATE(idt[18], 0, GD_KT, vec18, 0);
    SETGATE(idt[19], 0, GD_KT, vec19, 0);

    // Per-CPU setup
    trap_init_percpu();
}
```

执行，成功了！

2.5 Challenge 1

好吧，看来 Exercise4 写挫了。。。得重写了

实际上就是用 data 构造一个数组，每个数组都指向入口的地址，这样就可以完成了，我重新构建了三个宏，一个是对无 error code 的，一个是对有 error code 的，还有一个是对空着的无入口的中断。

下面代码中'at' 表示 @，我的 latex 貌似显示不出来，查资料也解决不了。

```
#define MYTH(name, num)         \
.text;      \
    .globl name;                \
    .type name, 'at'function;       \
    .align 2;           \
name:               \
    pushl $(num);       \
    jmp _alltraps;      \
.data;              \
    .long name


#define MYTH_NOEC(name, num)    \
.text;      \
    .globl name;                \
    .type name, 'at'function;       \
    .align 2;           \
name:               \
    pushl $0;           \
    pushl $(num);       \
    jmp _alltraps;      \
.data;              \
    .long name

#define MYTH_NULL()     \
.data;              \
    .long 0


.data
.align 2
.globl vectors
vectors:
.text
    MYTH_NOEC(vec0, T_DIVIDE)
    MYTH_NOEC(vec1, T_DEBUG)
    MYTH_NOEC(vec2, T_NMI)
    MYTH_NOEC(vec3, T_BRKPT)
    MYTH_NOEC(vec4, T_OFLOW)
    MYTH_NULL()
    MYTH_NOEC(vec6, T_BOUND)
    MYTH_NOEC(vec7, T_DEVICE)
    MYTH(vec8, T_DBLFLT)
    MYTH_NULL()
    MYTH(vec10, T_TSS)
    MYTH(vec11, T_SEGNP)
```

```
    MYTH(vec12, T_STACK)
    MYTH(vec13, T_GPFLT)
    MYTH(vec14, T_PGFLT)
    MYTH_NULL()
    MYTH_NOEC(vec16, T_FPERR)
    MYTH(vec17, T_ALIGN)
    MYTH_NOEC(vec18, T_MCHK)
    MYTH_NOEC(vec19, T_SIMDERR)

    TRAPHANDLER_NOEC(vec48, T_SYSCALL)
```

对于初始化 trap_init，只需要按照数组的方式处理即可。方便了很多。代码如下：

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

    extern uint32_t vectors[];
    extern void vec48();
    int i;
    for (i = 0; i != 20; i++) {
        if (i == T_BRKPT) {
            SETGATE(idt[i], 0, GD_KT, vectors[i], 3);
        } else {
            SETGATE(idt[i], 0, GD_KT, vectors[i], 0);
        }
    }
    SETGATE(idt[48], 0, GD_KT, vec48, 3);

    // Per-CPU setup
    trap_init_percpu();
}
```

2.6 Question

1. 有的中断需要 error code，有的中断不需要。同时无法保存对应的中断号。因此需要分开处理。

2. 因为 IDT 中设置 page fault 只能允许内核产生这种终端，如果在用户态产生 page fault 则会触发 general protection fault。

如果用户可以随意产生 page fault，则可能有恶意的进程疯狂产生 page fault 将整个内存空间占满。因此 page fault 只能在内核中处理。

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

3.1 Exercise 5 & Exercise 6

根据 trapno 来分配即可

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

3.2 Challenge 2

这个 challenge 就是需要在 breakpoint 的情况下增加 continue 和 single-step 的指令来进行调试。

对于 continue 非常简单，增加一个 mon 的函数，其直接恢复 curenv 的 enviroment 即可。

```
int
mon_continue(int argc, char **argv, struct Trapframe *tf)
{
    if (tf == NULL) {
        cprintf("Error: you only can use continue in breakpoint.\n");
        return -1;
    }

    tf->tf_eflags &= (~FL_TF);
    env_run(curenv);    // usually it won't return;
    panic("mon_continue : env_run return");
    return 0;
}
```

为了方便之后的测试我改变了一下 usr/breakpoint.c 的程序：

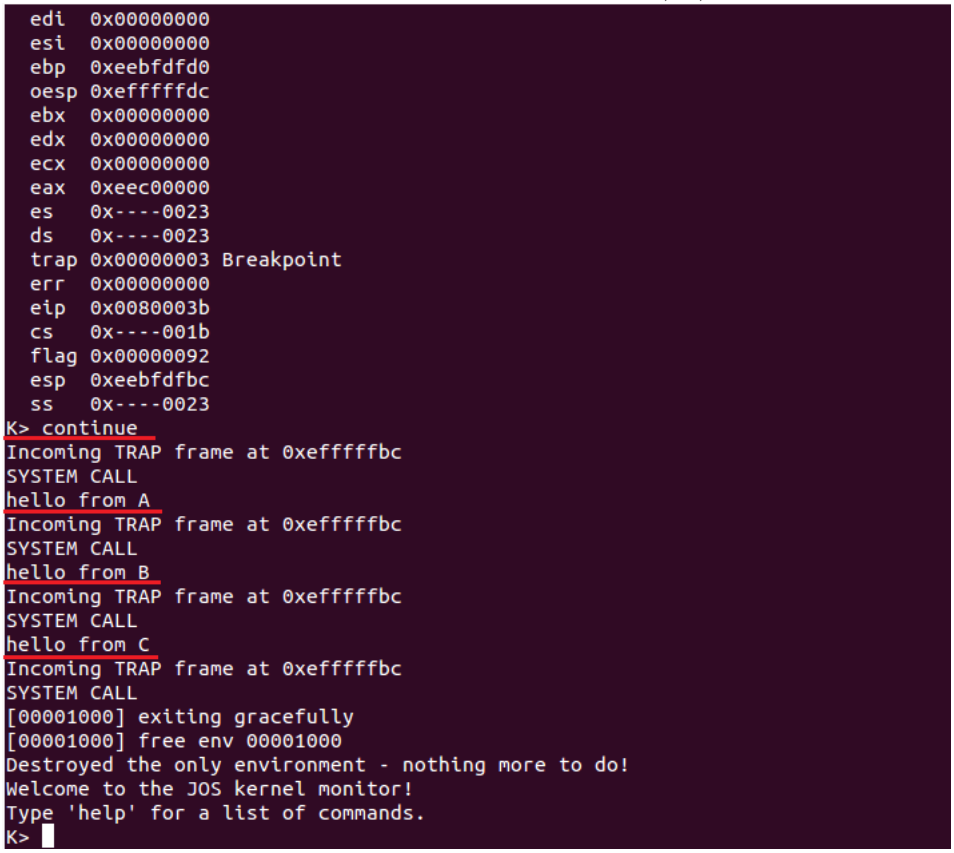
```
#include <inc/lib.h>

void
```

```
umain(int argc, char **argv)
{
    asm volatile("int $3");

    // my code:
    cprintf("hello from A\n");
    cprintf("hello from B\n");
    cprintf("hello from C\n");
}
```

然后进行运行，输入 continue，发现输出了 hello from A, B, C 并正常结束，说明成功了！



对于 single-stepping 则需要在 tf_eflags 中设置 TF 位，然后恢复原进程的 enviroment，当原进程执行完一条语句后会检查 TF 位，如果为 1，则会自动触发一个 int 1 即 debug exception。然后我们还需要在 trap_dispatch 中处理 T_DEBUG, 为了偷懒，我直接使用了 breakpoint 的处理过程 monitor(tf)。:)

```
\\ in trap_dispatch:
    switch (tf->tf_trapno) {
            case T_DEBUG:
                monitor(tf);
                break;
            ...
        }
```

```
\\ in monitor.c
int
mon_si(int argc, char **argv, struct Trapframe *tf)
{
    if (tf == NULL) {
        cprintf("Error: you only can use si in breakpoint.\n");
        return -1;
    }

    // next step also cause breakpoint interrupt
    tf->tf_eflags |= FL_TF;

    env_run(curenv);
    panic("mon_si : env_run return");
    return 0;
}
```

我又改写了 breakpoing.c 的程序来测试

```
void
umain(int argc, char **argv)
{
    asm volatile("int $3");

    // my test for singal stepping
    asm volatile("movl $0x1, %eax");
    asm volatile("movl $0x2, %eax");
}
```

我们来通过测试来观察%eax 的变化，为了方便看，我暂时注释掉 print_trapframe 中的一些信息，只保留%eax 的值。结果如下：

```
Incoming TRAP frame at 0xeffffffc
TRAP NUM : 3
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
  eax  0xeec00000
K> si
tfno: 3
Incoming TRAP frame at 0xeffffffc
TRAP NUM : 1
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
  eax  0x00000001
K> si
tfno: 1
Incoming TRAP frame at 0xeffffffc
TRAP NUM : 1
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
  eax  0x00000002
K> si
tfno: 1
Incoming TRAP frame at 0xeffffffc
TRAP NUM : 1
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
  eax  0x00000002
K> continue
Incoming TRAP frame at 0xeffffffc
TRAP NUM : 48
[00001000] exiting gracefully
[00001000] free env 00001000
Destroyed the only environment - nothing more to do!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

完全符合预期，哈哈，搞定了！

3.3 Question

3. 由于 breakpoint 的陷阱是可以面向用户态的进程的，因此需要将 IDT 的特权位置设 3，允许用户态进程使用。如果不设置则用户无法使用断点中断，当 int 3 的时候就会因为权限不足造成 general protection fault。

4. 这种机制有效限制了用户可以使用所有的中断而造成的问题，因此可以通过设置 IDT 的特权位来控制用户的中断操作。

3.4 Exercise 7

系统调用的系统调用号会存在寄存器%eax 上，而接下来 5 个参数会存放在%edx, %ecx, %ebx, %edi, %esi 中，当系统调用结束后，其返回值会保存在寄存器%eax 上。

lib/syscall.c 中是用户调用系统调用的接口，里面也显示了不同系统调用的参数存放的位置，通过这个我们再根据系统调用号分开处理好系统调用即可。

需要更改的地方有：

(1) 增加系统调用中断入口程序和将其地址加入 IDT 表中，注意特权位置要置为 3，因为系统调用是提供给用户态进程使用的

(2) 在中断的 trap_dispatch() 中增加 T_SYSTEM 的分配 (3) 实现根据 system call 号来分发给不同的执行程序。

```
// trapentry.S:
    TRAPHANDLER_NOEC(vec48, T_SYSCALL)

// trap.c/trap_init
    void vec48();
    SETGATE(idt[48], 0, GD_KT, vec48, 3);

// trap.c/trap_dispatch:
    switch (tf->tf_trapno) {
        case T_PGFLT:
            if (tf->tf_cs == GD_KT)
                panic("page fault in kernel");
            else
                page_fault_handler(tf);
            break;
        case T_BRKPT:
            monitor(tf);
            break;
        case T_SYSCALL:
            r = 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);
            if (r < 0)
                panic("trap.c/syscall : %e\n", r);
            else
                tf->tf_regs.reg_eax = r;
            break;
        ...
    }

// syscall.c/syscall
// Dispatches to the correct kernel function, passing the arguments.
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.

    switch (syscallno) {
        case SYS_cputs:
            sys_cputs((char *)a1, (size_t)a2);
            return 0;
            break;
        case SYS_cgetc:
            return sys_cgetc();
            return 0;
            break;
```

```
        case SYS_getenvid:
            return sys_getenvid();
            break;
        case SYS_env_destroy:
            return sys_env_destroy(a1);
            break;
        dafult:
            return -E_INVAL;
    }
    panic("syscall not implemented");
}
```

3.5 Challenge 3: sysenter/sysexit

让我们重新看看 JOS 中用户进程调用系统调用的情景：

用户调用 lib/syscall，syscall 会根据参数生成汇编代码，通过 int 0x30 进入系统调用的中断，通过中断进入内核态的对应中断的入口，跳转到 trap 函数中，根据中断号为 0x30，则进入 syscall 的处理程序，再根据 syscall_no 来调用对应的系统调用执行程序，执行完后若无错误，则返回至 trap 部分直接通过恢复 env 来恢复到原来的运行状况。

sysenter 和 sysexit 这两个指令，是 Intel 专门用来加速 int 和 iret 的。其通过寄存器传参快速到达 syscall 中，而不需要通过中断进入内核态再通过 trap 的分发来到达 syscall 的执行程序。其中会用到 3 个特殊的寄存器称为 model specific registers(MSRs)。这三个寄存器分别是：

(1) IA32_SYSENTER_CS(174H): 保存着内核态的 CS 段值

(2) IA32_SYSENTER_ESP(175H): 保存着内核态的栈，在 JOS 就是 KSTACKTOP

(3) IA32_SYSENTER_EIP(176H): 保存着 sysenter_handler 的代码的地址

我们再来看看 sysenter 和 sysexit 具体会干什么：

当 sysenter 执行时，处理器会依次执行：

1. 将 IA32_SYSENTER_CS 的值导入 CS 寄存器中
2. 将 IA32_SYSENTER_EIP 导入 EIP 寄存器中
3. 将 IA32_SYSENTER_CS + 8 导入 SS 寄存器中
4. 将 IA32_SYSENTER_ESP 导入 ESP 寄存器中
5. 将权限为设置为 0, 即最高权限
6. 清理 EFLAGS 寄存器中的 VM flag
7. 开始按照 CS 和 EIP 进行执行

而当 sysexit 执行时，处理器会以此执行：

1. 将 16+IA32_SYSENTER_CS 导入 CS 寄存器中
2. 将 EDX 导入 EIP 中
3. 将 24+IA32_SYSENTER_CS 导入 SS 寄存器中
4. 将 ECX 导入 ESP 中
5. 将权限位设置为 3，即最低权限

6. 根据 EIP 执行，即回到用户程序中

通过这个我想到了一个方式来解决实现 sysenter 和 sysexit，即通过合理分配好寄存器来实现系统调用：
0. 初始化 IA32_SYSENTER_CS, IA32_SYSENTER_ESP, IA32_SYSENTER_EIP 在内核启动时，IA32_SYSENTER_EIP 指向 sysenter_handler。
1. 当用户调用系统调用的时候，先保存现场，将所有寄存器进行备份，并且将返回 sysenter 的地址放入%esi 中，%esp 保存在%ebp 中，并将系统调用的所有参数都保存在相应的寄存器中。
2. 调用 sysenter，则会跳到 sysenter_handler 上（这个时候已经在内核态了）
3.sysenter_handler 保存寄存器，并将参数保存在栈中，调用 syscall。
4. 待 syscall 回来后，恢复除%eax 外的所有寄存器
5. 将%esi 导入%edx，将%ebp 导入%ecx 中，这两个寄存器保存着进入返回用户进程的 eip 和 esp。
6. 调用 sysexit。

具体实现如下：

3.5.1 Step. 0 implementation

在 inc/types 增加宏

```
// in inc/types:
// MSRs(model specific registers) SYSENTER
#define IA32_SYSENTER_CS    0x174
#define IA32_SYSENTER_ESP   0x175
#define IA32_SYSENTER_EIP   0x176
```

在 inc/x86.h 中增加 wrmsr 的宏

```
// in inc/x86.h:
#define wrmsr(msr, val1, val2) \
    __asm__ __volatile__("wrmsr" \
    : /* no outputs */ \
    : "c" (msr), "a" (val1), "d" (val2))
```

在 kern/init.c 增加初始化函数，并在 i386_init 中调用

```
void msrs_init()
{
    // set up model specific registers
    extern void sysenter_handler();

    // GD_KT is kernel code segment, is also CS register
    wrmsr(IA32_SYSENTER_CS, GD_KT, 0);
    wrmsr(IA32_SYSENTER_ESP, KSTACKTOP, 0);
    wrmsr(IA32_SYSENTER_EIP, (uint32_t)(sysenter_handler), 0);      // entry of
        sysenter
```

```
}
```

3.5.2 Step. 1 & 2 implementation

在 lib/system 中增加函数 my_sysenter 提供给用户进程通过 sysenter 来系统调用的接口：

对于 ESP 的保存很简单，如何保存 EIP，从 Challenge 的材料中得知非常技巧的实现方式，通过 leal 一个标记代码的地址即可。

```
// Use my_sysenter, a5 must be 0.
// Attention: it will not update trapframe
static int32_t
my_sysenter(int num, int check, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t
    a4, uint32_t a5)
{
    assert(a5 == 0);
    int32_t ret;

    asm volatile(
        "pushl %%ebp\n\t"
        "pushl %%esp\n\t"
        "popl %%ebp\n\t"
        "leal after_sysenter_label, %%esi\n\t"
        "sysenter\n\t"
        "after_sysenter_label:\n\t"
        "popl %%ebp"
        : "=a" (ret)
        : "a" (num),
          "d" (a1),
          "c" (a2),
          "b" (a3),
          "D" (a4),
          "S" (a5)
        : "cc", "memory");

    if(check && ret > 0)
        panic("my_sysenter %d returned %d (> 0)", num, ret);

    return ret;
}
```

3.5.3 Step. 3 - 6 implementation

在 trapentry.S 中增加一个函数 sysenter_handler 即可，里面参数保存在栈中，然后调用 syscall 即可。记得%eax 不要恢复值，因为其保存着返回值。由于 sysenter 调用导致参数传递不足，但是我发现 JOS 提供的系统调用中最后一个参数都是 0，因此我就直接扔 0 在最后一个参数了。

```
.globl sysenter_handler
.type sysenter_handler, 'at' function
sysenter_handler:
```

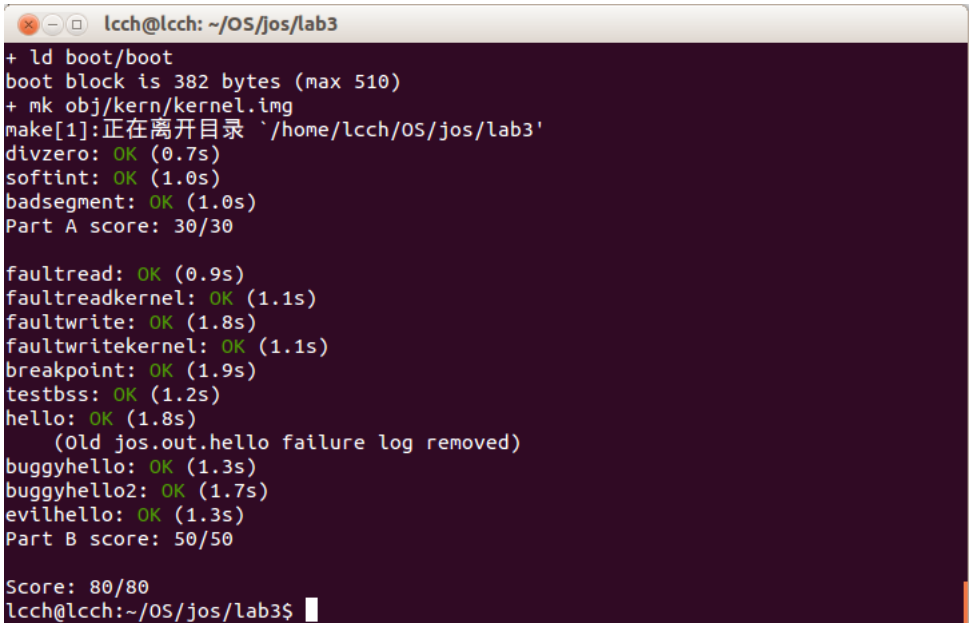
```
    pushl %esi;
    pushl %ebp;
    # pushl %oesp  (Useless)
    pushl %ebx;

    pushl $0     # in most of syscall, last argument will be 0
    pushl %edi
    pushl %ebx
    pushl %ecx
    pushl %edx
    pushl %eax
    call syscall
    popl %edx
    popl %edx
    popl %ecx
    popl %ebx
    popl %edi
    popl %ecx   # %ecx will be covered by %ebp, so it will be ok

    popl %ebx;
    #popl %oesp  (Useless)
    popl %ebp;
    popl %esi;
    movl %esi, %edx
    movl %ebp, %ecx
    sysexit
```

这个 Challenge 开始完全一头雾水，看了 IA32 的手册再 wiki 了很多东西才大概清楚，在实现过程中因为很多寄存器的使用不当，例如忘记保存之类的造成了很多次的错误。还有一次保存了所有的寄存器，还恢复了所有的寄存器，连%eax 这个返回值也恢复了，然后就瞎了。不过历经多次 gdb 调试终于成功了。我将 lib/system.c 中的所有系统调用都使用 my_sysenter 来调用，进行 make run-hello 和 make grade 的测试：

在 make run-hello 没有进入 trap 的中断中也将 hello world 打印了出来。并且通过了所有的 make grade 的测试。大功告成!!!



注意：材料也说了，这种调用方式不使用，因为不会更新 curenv 的 trapframe。

3.6 Exercise 8

lib/entry.S 是为创建的进程的入口，之后会跳转到 libmain，之后会跳到相应的进程。在 libmain 需要记录该进程的环境。

而我们有 sys_getenvid 返回当前进程的 envid 号，ENVX(eid) 可以得知其在 envs 的下标，这样就可以得到 thisenv 了。

```
    // set thisenv to point at our Env structure in envs[].
    // LAB 3: Your code here.
    thisenv = envs + ENVX(sys_getenvid());
```

3.7 Exercise 9

如果在 kernel 下发生了 page fault 则说明这是一个 bug，因此需要 panic 掉。思索一下加入 kernel 下发生 page fault，而不 panic 会发生什么？

因此需要在发生 page fault 的时候判断一下是用户态还是内核态下产生的 page fault。

```
// in trap.c/page_fault_handler

    // LAB 3: Your code here.
    if ((tf->tf_cs & 3) == 0)
        panic("page_fault_handler : page fault in kernel\n");
```

用户可能会通过中断访问不属于其空间的地址，例如 printf 一个内核的地址，或者创建超过字符串定义大小的内容等等。如果用户进程进行了这些操作，则直接杀掉它。

填写检测所访问的虚拟地址是否在其内存空间内。

```
int
user_mem_check(struct Env *env, const void *va, size_t len, int perm)
{
    // LAB 3: Your code here.
    if (len == 0) return 0;

    perm |= PTE_P;
    pte_t * pte;
    uint32_t va_now = (uint32_t)va;
    uint32_t va_last = ROUNDUP((uint32_t)va + len, PGSIZE);
    for (; ROUNDDOWN(va_now, PGSIZE) != va_last; va_now = ROUNDDOWN(va_now +
        PGSIZE, PGSIZE)) {
        if (va_now >= ULIM) {
            user_mem_check_addr = va_now;
            return -E_FAULT;
        }
        pte = pgdir_walk(env->env_pgdir, (void *)va_now, false);
        if (pte == NULL || ((*pte & perm ) != perm)) {
            user_mem_check_addr = va_now;
            return -E_FAULT;
        }
    }
    return 0;
}
```

在 sys_cputs 中加入：

```
    // LAB 3: Your code here.
    user_mem_assert(curenv, (void *)s, len, PTE_U);
```

在 kern/kdebug.c 加入对 usd, stabs, stabstr 的检查。

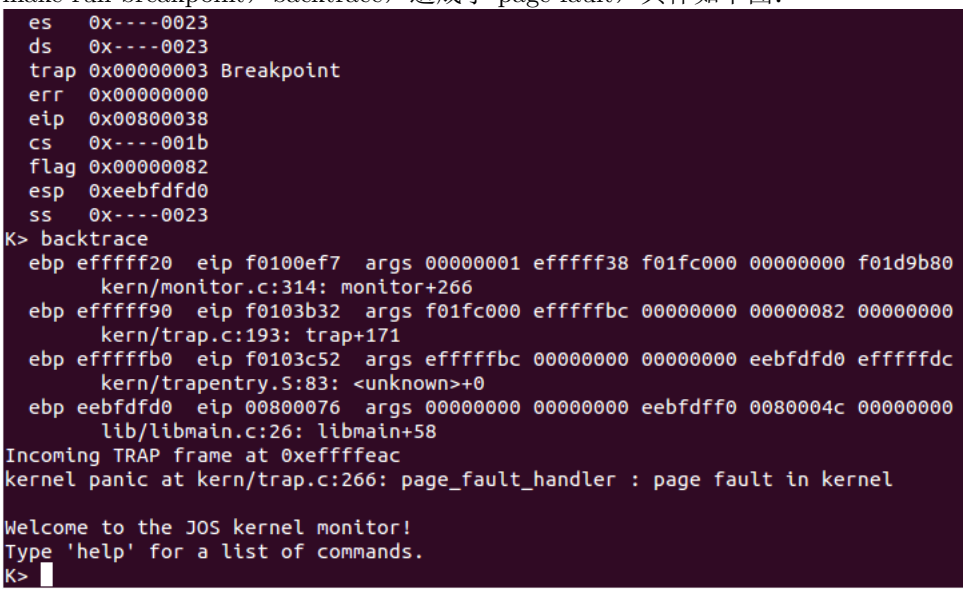
```
        // Make sure this memory is valid.
        // Return -1 if it is not.  Hint: Call user_mem_check.
        // LAB 3: Your code here.
        if (user_mem_check(curenv, (void *)usd, sizeof(struct UserStabData),
            PTE_U) < 0) {
            return -1;
        }

        stabs = usd->stabs;
        stab_end = usd->stab_end;
        stabstr = usd->stabstr;
        stabstr_end = usd->stabstr_end;

        // Make sure the STABS and string table memory is valid.
        // LAB 3: Your code here.
        if (user_mem_check(curenv, (void *)stabs, (uint32_t)stab_end - (uint32_t
            )stabs, PTE_U) < 0) {
            return -1;
        }
        if (user_mem_check(curenv, (void *)stabstr, (uint32_t)stabstr_end - (
            uint32_t)stabstr, PTE_U) < 0) {
            return -1;
```

```
        }
```

make run-breakpoint，backtrace，造成了 page fault，具体如下图：



产生 page fault 的原因是其尝试访问了超过 0xeebfe000 的内存，这段内存是不属于用户栈上的，因此产生了 Page Fault。

在哪里超过的 0xeebfe000，由于 libmain 已经处于 USTACKTOP 的位置，其两个参数就处于 USTACKTOP 的开头 8 个字节，而 backtrace 输出 5 个产生，因此在输出第 3 个参数的时候就访问到了用户栈外的内存空间，造成了 Page fault。

3.8 Exercise 10

似乎什么都没写就直接通过了。。通过看 evilhello 的代码，发现其向内核段写数据，我们确实在上面已经处理过了这种非法访问了。

4 Summary

这个 Lab 我前后做了大概 30 个小时，总体来说还是比较顺利的，特别是中断处理的部分，由于在大班中学习了理论只是以及 xv6 中的代码，所以做起来比较清晰。

这个 Lab 还有一个有意思的地方就是用户进程的执行以及上下文切换的过程，一开始觉得挺神秘的东西，做完这个 Lab 后，我发现其实在上下文切换就是使用了一个 Trapframe 来保存现场，并保存在其相应进程的 env_tf 御中。在之后回到进程时就只需要按照其保存的 trapframe 中的状态进行恢复即可。

这个 Lab 涉及到了比较多的汇编代码，这对一开始并不熟悉汇编的我是一个挑战，特别是 env_pop_tf 那段代码一开始看到之后完全就蒙了，一上来就 movl %0, %%esp，之后找了资料才明白这是在表示 input/output 中的第 1 个参数。之后的 trapentry.S 以及实现 sysenter 都让

我对汇编代码有了更好的了解。写完之后才发现写汇编真是比看汇编费劲太多了。有一个地方 debug 了很长时间，后来才发现我把 movl $GD_KD, %eax, 写成了 movl GD_KD, %eax。虽然是宏，但是也是立即数，要加 $，这个真的是很不小心呀。