JOS 实习 Lab3 报告

1100012836 崔治丞

1. Part A: User Environments and Exception Handling

1.1 Exercise1
1.2 Exercise 2
1.3 Exercise 4
1.3.1 Question

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

2.1 Exercise 5
2.2 Exercise 6
2.2.1 Question
2.3 Exercise 7
2.4 Exercise 8
2.5 Exercise 9
2.6 Exercise 10

3. Challenge

3.1 Challenge 1
3.2 Challenge 2

Part A: User Environments and Exception Handling

1.1 Exericse 1:

> **Exercise 1.** Modify mem_init() in kern/pmap.c to allocate and map the envs array. This array consists of exactly NENV instances of the Env structure allocated much like how you allocated the pages array. Also like the pages array, the memory backing envs should also be mapped user read-only at UENVS (defined in inc/memlayout.h) so user processes can read from this array.
>
> You should run your code and make sure check_kern_pgdir() succeeds.

类似于 lab2 中的分配

```
//------------------------------------ Lab3 ------------------------------
  envs = (struct Env *)boot_alloc(sizeof(struct Env) * NENV);
//------------------------------------ Lab3 ------------------------------
```

```
//------------------------------------ Lab3 ------------------------------
  boot_map_region(kern_pgdir, UENVS, PTSIZE, PADDR(envs),  PTE_U);
//------------------------------------ Lab3 ------------------------------
```

1.2 Exercise 2:

> **Exercise 2.** In the file env.c, finish coding the following functions:
>
> env_init()
> Initialize all of the Env structures in the envs array and add them to the env_free_list. Also calls env_init_percpu, which configures the segmentation hardware with separate segments for privilege level 0 (kernel) and privilege level 3 (user).
> env_setup_vm()
> Allocate a page directory for a new environment and initialize the kernel portion of the new environment's address space.
> region_alloc()
> Allocates and maps physical memory for an environment
> load_icode()
> You will need to parse an ELF binary image, much like the boot loader already does, and load its contents into the user address space of a new environment.
> env_create()
> Allocate an environment with env_alloc and call load_icode load an ELF binary into it.
> env_run()
> Start a given environment running in user mode.
>
> As you write these functions, you might find the new cprintf verb %e useful -- it prints a description corresponding to an error code. For example,
>
> ```
> r = -E_NO_MEM;
> panic("env_alloc: %e", r);
> ```
>
> will panic with the message "env_alloc: out of memory".

env_init:

该函数初始化 envs 数组，设定其 env_id 为 0、env_status 为 FREE、env_link 指向 envs 数组中下一个成员。并设定 env_free_list 指向&envs[0]。这是因为 envs[0]稍后会作为内核环境来使用，而且在 kern/init.c 文件中也可以看到在 i386_init 函数的最后有这样一条指令：

```
// We only have one user environment for now, so just run it.
env_run(&envs[0]);
```

所以 env_free_list 必须指向&envs[0]。

函数实现如下：

```
//------------------------------------- Lab3 ----------------
    uint32_t i;
    env_free_list = &envs[0];
    for (i = 0; i < NENV - 1; i++)
    {
        envs[i].env_id = 0;
        envs[i].env_status = ENV_FREE;
        envs[i].env_link = &envs[i + 1];
    }
    envs[NENV - 1].env_id = 0;
    envs[NENV - 1].env_status = ENV_FREE;
    envs[NENV - 1].env_link = NULL;
//------------------------------------- Lab3 ----------------
```

env_setup_vm：

env_setup_vm 为给定环境分配页目录。对于高于 UTOP 的部分和 kernel 的页目录保持一致，而低于 UTOP 的部分要清零，这部分就是用户进程所能使用的页目录。将高于 UTOP 的部分保持与 kernel 页目录一致是为了在用户进程引发中断或者调用时，可以直接通过这部分的页目录进入 kernel 进行处理，而不用再进行整体切换，节省了系统资源。

函数实现如下：

```
//------------------------------------- Lab3 ----------
    p->pp_ref++;
    e->env_pgdir = (pde_t *)page2kva(p);
    memcpy(e->env_pgdir, kern_pgdir, PGSIZE);
    memset(e->env_pgdir, 0, sizeof(pde_t) * PDX(UTOP));
//------------------------------------- Lab3 ----------
```

region_alloc：

region_alloc 分配物理页，并对应设定该进程的页目录。

函数实现如下：

```
//------------------------------------- Lab3 ----------------------
    uint32_t left = (uint32_t)ROUNDDOWN(va, PGSIZE);
    uint32_t right = (uint32_t)ROUNDUP(va + len, PGSIZE);
    struct PageInfo *p;
    for (; left < right; left += PGSIZE)
    {
        p = page_alloc(0);
        if (!p)
            panic("Allocation attempt failed!");
        page_insert(e->env_pgdir, p, (void *)left, PTE_U | PTE_W);
    }
//------------------------------------- Lab3 ----------------------
```

load_icode：

load_icode 将目标文件加载到给定的地址上。根据 ELF 中的信息进行内存的分配和数据的加载。需要注意的是，在进行内存分配之前，需要先切换到进程的页目录。在分配加载结束后，还需要切换回 kernel 的页目录。最后还需要将程序的 entry point 赋给 eip。

函数实现如下：

```
//-------------------------------------  Lab3  ----------------------------------------
    struct Elf *elf = (struct Elf *)binary;
    if (elf->e_magic != ELF_MAGIC)
        panic("Elf magic number is wrong!");
    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);
        }
    lcr3(PADDR(kern_pgdir));
    e->env_tf.tf_eip = elf->e_entry;        //information from boot/main.c and env_alloc();
//-------------------------------------  Lab3  ----------------------------------------
```

env_create：

env_create 将申请一个新的进程环境，加载给定程序，并将状态设定为给定状态。

函数实现如下：

```
//-------------------------------------  Lab3  ----------
    struct Env *env;
    int result;
    result = env_alloc(&env, 0);
    if (result < 0)
    {
        panic("env_create failed: %e\n", result);
    }
    load_icode(env, binary, size);
    env->env_type = type;
    return ;
//-------------------------------------  Lab3  ----------
```

env_run：

env_run 函数将当前运行的进程从 RUNNING 状态切换到 RUNNABLE 状态，将 curenv 设为给定进程，修改其状态，并加载当前进程的页目录

函数实现如下：

```
    // LAB 3: Your code here.
//-------------------------------------  Lab3  ----------
    //step1
    if (curenv != NULL)
        if (curenv->env_status == ENV_RUNNING)
            curenv->env_status = ENV_RUNNABLE;
    curenv = e;
    curenv->env_status = ENV_RUNNING;
    curenv->env_runs++;
    lcr3(PADDR(curenv->env_pgdir));
    //step2
    env_pop_tf(&curenv->env_tf);
//-------------------------------------  Lab3  ----------
```

1.3 Exercise 4

> **Exercise 4.** Edit trapentry.S and trap.c and implement the features described above. The macros TRAPHANDLER and TRAPHANDLER_NOEC in trapentry.S should help you, as well as the T_* defines in inc/trap.h. You will need to add an entry point in trapentry.S (using those macros) for each trap defined in inc/trap.h, and you'll have to provide _alltraps which the TRAPHANDLER macros refer to. You will also need to modify trap_init() to initialize the idt to point to each of these entry points defined in trapentry.S; the SETGATE macro will be helpful here.
>
> Your _alltraps should:
>
> 1. push values to make the stack look like a struct Trapframe
> 2. load GD_KD into %ds and %es
> 3. pushl %esp to pass a pointer to the Trapframe as an argument to trap()
> 4. call trap (can trap ever return?)
>
> Consider using the pushal instruction; it fits nicely with the layout of the struct Trapframe.
>
> Test your trap handling code using some of the test programs in the user directory that cause exceptions before making any system calls, such as user/divzero. You should be able to get **make grade** to succeed on the divzero, softint, and badsegment tests at this point.

首先查询 IA32 手册来确定哪些中断需要压入错误码。结果如下：

Interrupt	ID	Error Code
T_DIVIDE	0	N
T_DEBUG	1	N
T_NMI	2	N
T_BRKPT	3	N
T_OFLOW	4	N
T_BOUND	5	N
T_ILLOP	6	N
T_DEVICE	7	N
T_DBLFLT	8	Y
T_TSS	10	Y
T_SEGNP	11	Y
T_STACK	12	Y
T_GPFLT	13	Y
T_PGFLT	14	Y
T_FPERR	16	N
T_ALIGN	17	Y
T_MCHK	18	N
T_SIMDERR	19	N
T_SYSCALL	48	N

entry.S 设定中断处理函数的入口

```
TRAPHANDLER_NOEC(tid0, 0)
TRAPHANDLER_NOEC(tid1, 1)
TRAPHANDLER_NOEC(tid2, 2)
TRAPHANDLER_NOEC(tid3, 3)
TRAPHANDLER_NOEC(tid4, 4)
TRAPHANDLER_NOEC(tid5, 5)
TRAPHANDLER_NOEC(tid6, 6)
TRAPHANDLER_NOEC(tid7, 7)
TRAPHANDLER(tid8, 8)

TRAPHANDLER(tid10, 10)
TRAPHANDLER(tid11, 11)
TRAPHANDLER(tid12, 12)
TRAPHANDLER(tid13, 13)
TRAPHANDLER(tid14, 14)

TRAPHANDLER_NOEC(tid16, 16)
TRAPHANDLER(tid17, 17)
TRAPHANDLER_NOEC(tid18, 18)
TRAPHANDLER_NOEC(tid19, 19)
```

_alltraps：

首先在栈中建立 TrapFrame 结构，将 GD_KD 赋给%ds 和%es，pushl %eax 将 TrapFrame 作为 trap 函数的一个参数压栈，最后调用 trap 函数。

```
//-----------------------
_alltraps:
  pushl %ds
  pushl %es
  pushal
  movl $GD_KD, %eax
  movw %ax, %ds
  movw %ax, %es
  pushl %esp
  call trap
//-----------------------
```

在 kern/trap.c 文件中，需要对 trap_init 函数进行修改。利用 inc/mmu.h 中提供的 SETGATE 宏，和 xv6 系统中 trap.c 中的代码，就可以实现对于中断的初始化。

```
void tid0();
void tid1();
void tid2();
void tid3();
void tid4();
void tid5();
void tid6();
void tid7();
void tid8();

void tid10();
void tid11();
void tid12();
void tid13();
void tid14();

void tid16();
void tid17();
void tid18();
void tid19();
SETGATE(idt[0], 0, GD_KT, tid0, 0);
SETGATE(idt[1], 0, GD_KT, tid1, 0);
SETGATE(idt[2], 0, GD_KT, tid2, 0);
SETGATE(idt[3], 0, GD_KT, tid3, 0);
SETGATE(idt[4], 0, GD_KT, tid4, 0);
SETGATE(idt[5], 0, GD_KT, tid5, 0);
SETGATE(idt[6], 0, GD_KT, tid6, 0);
SETGATE(idt[7], 0, GD_KT, tid7, 0);
SETGATE(idt[8], 0, GD_KT, tid8, 0);
SETGATE(idt[10], 0, GD_KT, tid10, 0);
SETGATE(idt[11], 0, GD_KT, tid11, 0);
SETGATE(idt[12], 0, GD_KT, tid12, 0);
SETGATE(idt[13], 0, GD_KT, tid13, 0);
SETGATE(idt[14], 0, GD_KT, tid14, 0);
SETGATE(idt[16], 0, GD_KT, tid16, 0);
SETGATE(idt[17], 0, GD_KT, tid17, 0);
SETGATE(idt[18], 0, GD_KT, tid18, 0);
SETGATE(idt[19], 0, GD_KT, tid19, 0);
```

1.4.1 Question

Questions

Answer the following questions in your answers-lab3.txt:

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?)
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)?

1.

因为有些中断需要压入错误码，有些不需要，所以需要利用两种不同的handler来进行处理。

2.

int $14 引发 page fault，但是根据上面初始化的设定。page fault 只能由内核触发，用户程序触发 page fault 会引起 general protection fault 来保护内核。所以实际上产生的中断时编号为 13 的 general protection fault。

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

2.1 Exercise 5 & 2.2 Exercise 6

Exercise 5. Modify `trap_dispatch()` to dispatch page fault exceptions to `page_fault_handler()`. You should now be able to get `make grade` to succeed on the `faultread`, `faultreadkernel`, `faultwrite`, and `faultwritekernel` tests. If any of them don't work, figure out why and fix them. Remember that you can boot JOS into a particular user program using `make run-x` or `make run-x-nox`.

Exercise 6. Modify `trap_dispatch()` to make breakpoint exceptions invoke the kernel monitor. You should now be able to get `make grade` to succeed on the `breakpoint` test.

根据 trapno 来分配即可，注意处理完成后要 return，不能执行最后的错误处理代码。

函数实现如下：

```
//------------------------------------ Lab3 ----------------------------------------------
    if (tf->tf_trapno == T_PGFLT) {
        //cprintf("pagefault!\n");
        page_fault_handler(tf);
        return;
    }
    if (tf->tf_trapno == T_BRKPT) {
        //cprintf("brkpt!\n");
        monitor(tf);
        return;
    }
    if (tf->tf_trapno == T_DEBUG) {
        my_monitor(tf);
        return;
    }
    if (tf->tf_trapno == T_SYSCALL) {
        //cprintf("Syscall!\n");
        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);
        if (tf->tf_regs.reg_eax < 0)
            panic("syscall failed: %e\n", tf->tf_regs.reg_eax);
        return;
    }
//------------------------------------ Lab3 ----------------------------------------------
```

2.2.1 Question

Questions

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

3. 产生何种中断取决于 SETGATE 中对于 T_BRKPT 的特权级设定。如果设定为 3，则用户进程也可以引发 breakpoint 中断，否则会产生 general protection fault.

4. 通过设定特权级来限制用户所能够引发的中断，用以保护 kernel。

2.3 Exercise 7

Exercise 7. Add a handler in the kernel for interrupt vector T_SYSCALL. You will have to edit kern/trapentry.S and kern/trap.c's trap_init(). You also need to change trap_dispatch() to handle the system call interrupt by calling syscall() (defined in kern/syscall.c) with the appropriate arguments, and then arranging for the return value to be passed back to the user process in %eax. Finally, you need to implement syscall() in kern/syscall.c. Make sure syscall() returns -E_INVAL if the system call number is invalid. You should read and understand lib/syscall.c (especially the inline assembly routine) in order to confirm your understanding of the system call interface. You may also find it helpful to read inc/syscall.h.

Run the user/hello program under your kernel (**make run-hello**). It should print "hello, world" on the console and then cause a page fault in user mode. If this does not happen, it probably means your system call handler isn't quite right. You should also now be able to get **make grade** to succeed on the testbss test.

首先需要修改 trap_dispatch 函数，增加 SYSCALL 的操作，然后修改 kern/syscall.c 中的 syscall 函数，根据不同的 syscallno 进行不同的系统调用

实现如下：

```
if (tf->tf_trapno == T_SYSCALL) {
    //cprintf("Syscall!\n");
    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);
    if (tf->tf_regs.reg_eax < 0)
        panic("syscall failed: %e\n", tf->tf_regs.reg_eax);
    return;
}
```

```
//------------------------------------- Lab3 ----------
    if (syscallno == SYS_cputs){
        //cprintf("sys_cputs!\n");
        sys_cputs((char *)a1, (size_t)a2);
        return 0;
    }
    if (syscallno == SYS_cgetc) {
        //cprintf("sys_cgetc!\n");
        return sys_cgetc();
        return 0;
    }
    if (syscallno == SYS_getenvid) {
        //cprintf("sys_getenvid!\n");
        return sys_getenvid();
        return 0;
    }
    if (syscallno == SYS_env_destroy) {
        //cprintf("sys_env_destroy!\n");
        return sys_env_destroy((envid_t)a1);
    }
    return -E_INVAL;
//------------------------------------- Lab3 ----------
```

2.4 Exercise 8

> **Exercise 8.** Add the required code to the user library, then boot your kernel. You should see user/hello print "hello, world" and then print "i am environment 00001000". user/hello then attempts to "exit" by calling sys_env_destroy() (see lib/libmain.c and lib/exit.c). Since the kernel currently only supports one user environment, it should report that it has destroyed the only environment and then drop into the kernel monitor. You should be able to get **make grade** to succeed on the hello test.

用户进程通过 lib/entry.S 开始运行，之后进入 lib/libmain.c，执行 libmain 函数。前面说到过，envs[0]是作为内核进程的，所以在这里要对 thisenv 进行修改以进入用户进程。根据题目中的提示，利用 sys_getenvid 函数得到当前进程 id，再通过 ENVX 宏得到其在 envs 数组中的下标，即可得到正确的 thisenv。

函数实现如下：

```
//------------------------------------- Lab3 --------
    thisenv = envs + ENVX(sys_getenvid());
//------------------------------------- Lab3 --------
```

2.5 Exercise 9

> **Exercise 9.** Change kern/trap.c to panic if a page fault happens in kernel mode.
>
> Hint: to determine whether a fault happened in user mode or in kernel mode, check the low bits of the tf_cs.
>
> Read user_mem_assert in kern/pmap.c and implement user_mem_check in that same file.
>
> Change kern/syscall.c to sanity check arguments to system calls.
>
> Boot your kernel, running user/buggyhello. The environment should be destroyed, and the kernel should *not* panic. You should see:
>
> ```
> [00001000] user_mem_check assertion failure for va 00000001
> [00001000] free env 00001000
> Destroyed the only environment - nothing more to do!
> ```
>
> Finally, change debuginfo_eip in kern/kdebug.c to call user_mem_check on usd, stabs, and stabstr. If you now run user/breakpoint, you should be able to run **backtrace** from the kernel monitor and see the backtrace traverse into lib/libmain.c before the kernel panics with a page fault. What causes this page fault? You don't need to fix it, but you should understand why it happens.

page fault 不能够由内核进程引发，所以在 kern/trap.c 中的 page_fault_handler 中添加如下代码：

```
//----------------------------------------- Lab3 --------------------
    if ((tf->tf_cs & 3) == 0)
            panic("page_fault_handler : page fault in kernel\n");
//----------------------------------------- Lab3 --------------------
```

kern/pmap.c 函数中的 user_mem_check 将检查给定地址的内容是否符合给定的权限。由于给定的 va 并不一定是按 PGSIZE 对齐的，所以需要单独处理一次。

函数实现如下：

```
//------------------------------------------ Lab3 ----------------
    if (len == 0) return 0;
    perm |= PTE_P;
    pte_t * pte;
    uint32_t addr = (uint32_t)va;
    if (addr >= ULIM) {
        user_mem_check_addr = addr;
        return -E_FAULT;
    }
    pte = pgdir_walk(env->env_pgdir, (void *)addr, false);
    if (pte == NULL || (*pte & perm) != perm) {
        user_mem_check_addr = addr;
        return -E_FAULT;
    }
    uint32_t left = ROUNDDOWN(addr + PGSIZE, PGSIZE);
    uint32_t right = ROUNDUP(addr + len, PGSIZE);
    for (; left < right; left += PGSIZE, addr += PGSIZE) {
        if (left >= ULIM) {
            user_mem_check_addr = left;
            return -E_FAULT;
        }
        pte = pgdir_walk(env->env_pgdir, (void *)left, false);
        if (pte == NULL || (*pte & perm) != perm) {
            user_mem_check_addr = left;
            return -E_FAULT;
        }
    }
//------------------------------------------ Lab3 ----------------
```

对应的，需要在 kern/syscall.c 中修改 sys_cputs 函数，检查给定字符串是否是用户可用。代码如下：

```
//------------------------------------------ Lab3 --------
    user_mem_assert(curenv, (void *)s, len, PTE_U);
//------------------------------------------ Lab3 --------
```

在 kern/kdebug.c 中的 debug_info 函数中也需要检查 usd，stabs 和 stabstr 是否是用户可用的。代码如下：

```
//------------------------------------ Lab3 ----------------------------------------
        if (user_mem_check(curenv, (void *)usd, sizeof(struct UserStabData), PTE_U) < 0) {
            return -1;
        }
//------------------------------------ Lab3 ----------------------------------------
```

```
//------------------------------------ Lab3 ----------------------------------------
    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;
    }
//------------------------------------ Lab3 ----------------------------------------
```

backtrace 的结果如下：

```
K> backtrace
Stack backtrace:
  ebp efffffe0  eip f0100905  args 00000001 efffff08 f01a2000 f01001cc f0103d48
         kern/monitor.c:314: runcmd+230
  ebp efffff60  eip f0100f3b  args f010601d efffff8c f014a541 f011af88 00000000
         kern/monitor.c:338: monitor+69
  ebp efffff80  eip f0104215  args f01a2000 effffffc 00000000 00000000 00000000
         kern/trap.c:206: trap+186
  ebp effffff0  eip f011d3c0  args effffffc 00000000 00000000 eebfdfd0 effffffc
         <unknown>:0: <unknown>+0
  ebp eebfdfd0  eip 008000b3  args 00000000 00000000 00000000 00000000 00000000
         lib/libmain.c:29: libmain+67
  ebp eebfdff0  eip 00800031Incoming TRAP frame at 0xeffffe5c
kernel panic at kern/trap.c:284: page_fault_handler : page fault in kernel

Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

2.6 Exercise 10

Exercise 10. Boot your kernel, running user/evilhello. The environment should be destroyed, and the kernel should not panic. You should see:

```
[00000000] new env 00001000
[00001000] user_mem_check assertion failure for va f0100020
[00001000] free env 00001000
```

所有需要完成的部分都已经完成，这个 exercise 不需要写任何东西了。

make grade 结果如下：

```
divzero: OK (1.9s)
softint: OK (1.7s)
badsegment: OK (1.2s)
Part A score: 30/30

faultread: OK (1.2s)
faultreadkernel: OK (2.3s)
faultwrite: OK (2.3s)
faultwritekernel: OK (1.1s)
breakpoint: OK (1.5s)
testbss: OK (1.5s)
hello: OK (1.5s)
buggyhello: OK (2.0s)
buggyhello2: OK (1.4s)
evilhello: OK (1.5s)
Part B score: 50/50

Score: 80/80
czc@ubuntu:~/test/lab$
```

Challenge

3.1 Challenge 1

> *Challenge!* You probably have a lot of very similar code right now, between the lists of TRAPHANDLER in trapentry.S and their installations in trap.c. Clean this up. Change the macros in trapentry.S to automatically generate a table for trap.c to use. Note that you can switch between laying down code and data in the assembler by using the directives .text and .data.

这个 challenge 需要重写 exercise4 的内容。参考 xv6 中对于 trapentry 的生成方式，先定义三个宏，如下：

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

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

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

然后用自己的宏完成对 idt 的初始化。这里需要注意的是由于第 9 项和第 15 项没有定义，但必须为这两项留出空间，以保证后面的项偏移量的正确，所以设置 MyNull 宏占用空间。

```
//-----------------------------------Lab3-Challenge-----
    MyFun_N(tid0, 0)
    MyFun_N(tid1, 1)
    MyFun_N(tid2, 2)
    MyFun_N(tid3, 3)
    MyFun_N(tid4, 4)
    MyFun_N(tid5, 5)
    MyFun_N(tid6, 6)
    MyFun_N(tid7, 7)
    MyFun(tid8, 8)
    MyNull()
    MyFun(tid10, 10)
    MyFun(tid11, 11)
    MyFun(tid12, 12)
    MyFun(tid13, 13)
    MyFun(tid14, 14)
    MyNull()
    MyFun_N(tid16, 16)
    MyFun(tid17, 17)
    MyFun_N(tid18, 18)
    MyFun_N(tid19, 19)
//-----------------------------------Lab3-Challenge-----
```

对应 kern/trap.c 中也需要修改 trap_init 函数。同样借鉴于 xv6 中的 trap.c 文件中的代码。

```
//-----------------------------------Lab3-Challenge---
    extern uint32_t vec[];
    void tid48();
    int i;
    for (i = 0; i != 20; i++) {
        if (i == 3){
            SETGATE(idt[i], 0, GD_KT, vec[i], 3);
        }
        else
            SETGATE(idt[i], 0, GD_KT, vec[i], 0);
    }
//-----------------------------------Lab3-Challenge---
```

3.2 Challenge 2

Challenge! Modify the JOS kernel monitor so that you can 'continue' execution from the current location (e.g., after the int3, if the kernel monitor was invoked via the breakpoint exception), and so that you can single-step one instruction at a time. You will need to understand certain bits of the EFLAGS register in order to implement single-stepping.

Optional: If you're feeling really adventurous, find some x86 disassembler source code - e.g., by ripping it out of QEMU, or out of GNU binutils, or just write it yourself - and extend the JOS kernel monitor to be able to disassemble and display instructions as you are stepping through them. Combined with the symbol table loading from lab 2, this is the stuff of which real kernel debuggers are made.

这个 Challenge 需要我们能够在触发 breakpoint 中断的情况下，实现继续运行（continue）和单步运行（single—step）这两种操作。

根据提示，查询 EFLAGS 各位的意义，发现 FL_TF 位用来设定单步调试。所以可以通过修改该位的值来实现单步运行。进一步查询，发现单步运行的实质是通过 DEBUG 中断来实现的，所以还需要增加 DEBUG 中断的 handler。于是代码如下：

monito.c 中：

```
int
mon_c(int argc, char **argv, struct Trapframe *tf) {
    if (tf == NULL){
        cprintf("Continue Error!\n");
        return -1;
    }

    tf->tf_eflags &= (~FL_TF);
    env_run(curenv);
    cprintf("This should never be printed!\n");
    return 0;
}

int mon_si(int argc, char **argv, struct Trapframe *tf) {
    if (tf == NULL){
        cprintf("Continue Error!\n");
        return -1;
    }

    tf->tf_eflags |= FL_TF;
    env_run(curenv);
    cprintf("This should never be printed!\n");
    return 0;
}
```

trap.c 中 trap_dispatch 增加：

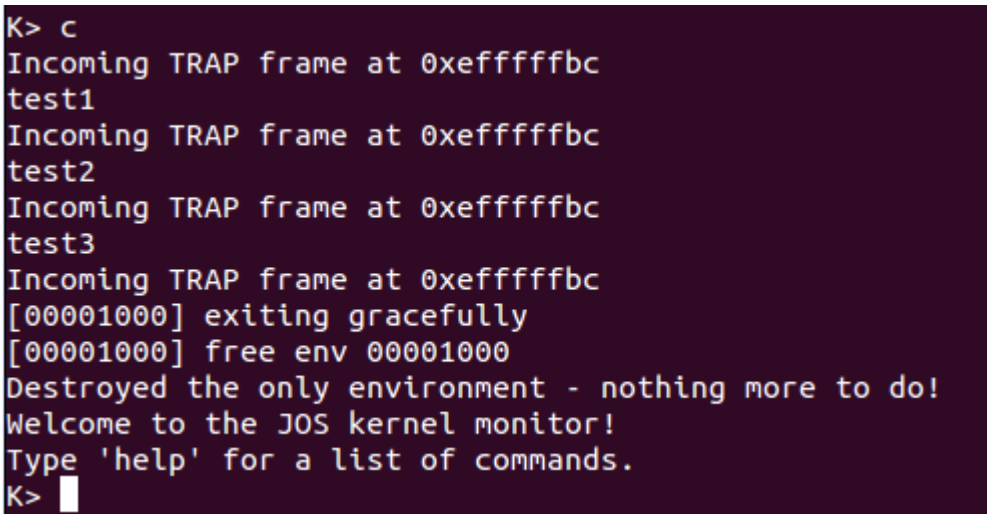
```
if (tf->tf_trapno == T_DEBUG) {
    my_monitor(tf);
    return;
}
```

my_monitor 和 monitor 是一样的，这里为了区分就重新定义了一个函数而已。

通过修改 breakpoint.c 文件，可以查看上述修改是否正确。

```
void
umain(int argc, char **argv)
{
    asm volatile("int $3");
    asm volatile("movl $0x1, %ebx");
    asm volatile("movl $0x2, %ebx");
    asm volatile("movl $0x3, %ebx");
    cprintf("test1\n");
    cprintf("test2\n");
    cprintf("test3\n");
}
```

结果如下：



可以看到达到了预期的效果。