

北京大学操作系统实习(实验班)报告

# **JOS-Lab 3:**

# **User Environments**

黄睿哲 00948265  
huangruizhe@pku.edu.cn

April 15, 2012

# 目录

JOS-Lab 3:.....	1
User Environments .....	1
内容一： 总体概述 .....	4
Part A (I): User Environments .....	4
Part A (II): Exception Handling.....	4
Part B: Page Faults, Breakpoints Exceptions, and System Calls .....	5
内容二： 任务完成情况 .....	6
I: 任务完成列表 .....	6
II: 具体 Exercise 完成情况.....	6
1. (Part A) User Environments .....	6
1.1 Environment State .....	6
1.2 Allocating the Environments Array .....	7
Exercise 1.....	7
1.3 Creating and Running Environments.....	8
Exercise 2.....	8
1.3.1 env_init() .....	9
1.3.2 env_setup_vm().....	9
1.3.3 env_alloc() .....	10
1.3.4 region_alloc().....	11
1.3.5 load_icode().....	12
1.3.6 env_create() .....	13
1.3.7 env_free() & env_destroy() .....	14
1.3.8 env_run() & env_pop_tf() .....	14
2. (Part A) Exception Handling.....	15
2.1 Handling Interrupts and Exceptions .....	15
Exercise 3.....	15
2.2 Basics of Protected Control Transfer .....	19
2.2.1 中断描述符表 IDT .....	19
2.2.2 任务状态段 TSS .....	19
2.3 Types of Exceptions and Interrupts .....	19
2.4 An Example.....	19
2.5 Nested Exceptions and Interrupts .....	19
2.6 Setting Up the IDT .....	20
Exercise 4.....	20
Challenge1.....	24

3. (Part B) Page Faults, Breakpoints Exceptions, and System Calls .....	25
3.1 Handling Page Faults .....	25
Exercise 5 .....	25
3.2 The Breakpoint Exception .....	26
Exercise 6 .....	26
Challenge2 .....	26
3.3 System calls .....	30
Exercise 7 .....	30
3.4 User-mode startup .....	31
Exercise 8 .....	32
3.5 Page faults and memory protection .....	32
Exercise 9 .....	33
Exercise 10 .....	35
内容三：遇到的困难以及解决方法 .....	37
困难一：这次 LAB 的内容比较多，比较难 .....	37
困难二：我 SB 了 .....	37
困难三：报告怎么写了那么久 T.T .....	37
内容四：收获和感想 .....	38
内容五：对课程的意见和建议 .....	39
内容六：参考文献 .....	40

# 内容一：总体概述

JOS-Lab3 实习的主要内容为实现**进程管理**和**中断与异常处理**两个功能：

对于进程管理，我们需要在 JOS 中建立相应的数据结构，用于记录每个用户进程的运行情况；创建和加载用户进程，包括建立进程的虚拟地址空间、加载进程的可执行代码到内存、将控制权交给进程开始执行等。

对于中断管理，我们需要在理解 x86 中断机制的基础上，在 JOS 中建立中断描述符表 IDT，指定相应的中断处理程序，以及实现一些系统调用。

MIT 材料中强调，尽管 JOS 中的“环境（environment）”和 Unix 中的“进程（process）”是同位的概念，但是两者还是不同的。由于这次 Lab 做得比较急，我没有深究这个问题，所以希望到时候听了 Lab3 的小组 Presentation 后，能够解开这个问题。

## Part A (I): User Environments

用户进程管理，首先是在 JOS 中建立和初始化相应的数据结构，用于记录每个用户进程的运行情况。这部分内容涉及 `struct Env` 结构体，`struct Env *envs` 进程管理数组，`struct Env *curenv` 当前运行进程，这些定义出现在 `inc/env.h` 和 `kern/env.h` 中。

其次，要创建和加载用户进程。开机以后，控制权是在 `kernel`，如何将控制权从 `kernel` 转到用户进程？也就是说，如何让 `CS:IP` 指向用户进程的第一条指令，并能够正确往下执行？这部分内容包括建立进程的虚拟地址空间 `env_setup_vm()` 和 `region_alloc()`、加载进程的可执行代码到内存 `load_icode()`、将控制权交给进程开始执行 `env_run()`，这些都在 `kern/env.c` 中。

需要说明的是，直到 Lab3 结束，我们只会涉及单个用户进程，不存在多个进程竞争的情况。这就是说，一开机先是 `Kernel` 运行，然后 `Kernel` 将控制权交给某一个用户进程，然后用户进程运行完了就结束了。当然，用户进程在运行的时候，可以进行系统调用，这时候控制权就在用户进程和 `kernel` 之间切换。

## Part A (II): Exception Handling

中断与异常处理，首先，当然是要了解 x86 体系结构的中断与异常处理机制。

其次，要在 `kernel` 中建立起中断描述符表 IDT。CPU 在遇到一个中断时，就可以根据中断号查找 IDT 表，“跳转”到相应的中断处理程序。如何跳转？在 JOS 中简单来说，这个过程是：CPU 遇到中断->根据中断（向量）号查找 IDT，获得相应 `Handler` 地址->`Handler`（以及 `_alltrap`）开始执行，进行一些压栈操作记录必要的信息->转到 `trap_dispatch`，根据栈中记录的中断向量，将中断分发到相应的中断处理程序。当然，我们也要牢记从中断返回原程序

继续执行的过程。

然后，就是书写各种中断处理程序啦，这就是 **Part B** 的内容，我们将涉及到处理三种中断或异常。

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

我们要为 JOS 书写三个中断处理程序，分别是处理页面错误（从名称来说，“页面错误”比“缺页中断”更加合适）、断点异常、系统调用。通过这几个例子，我们可以对中断处理流程更加熟悉。

在这次 Lab 中的页面错误处理比较简单，对于 kernel 中的 `page fault` 我们只要 `panic` 一下。用户程序出现的 `page fault` 以后将会处理。

断点异常没什么好说的，按部就班。不过里面的 `challenge` 比较有意思，我们要实现 JOS 控制台的 `continue` 和 `single-step` 功能，后者要求我们对“调试异常”有所了解。

系统调用，用户进程如果只在用户态下运行，会比较无趣；然而为了保护 kernel，它也不能够越权做 kernel 能做的事情。因此人们想出了一个办法，那就是让用户进程进行系统调用；kernel 发现系统调用以后，进行相应的处理；然后 kernel 再将控制权交还给用户进程。这样在外面看起来就好像是用户进程做了 kernel 能做的事情，一个很好的包装！

# 内容二：任务完成情况

## I: 任务完成列表

Exercise	1~2	3~4	5~10
第一周			
第二周	Y	Y	Y

Challenge	1	2	3
第三周	Y	Y	

## II: 具体 Exercise 完成情况

### 1. (Part A) User Environments

#### 1.1 Environment State

这部分介绍了 JOS 用户进程管理的数据结构，用于记录系统中每个用户进程的运行情况，它的作用类似于以前学过的进程控制块 PCB。

首先是 struct Env 结构体，定义在 inc/env.h 中。这个结构体是用来记录某一个用户进程的信息，任何一个用户进程都需要一个对应的 Env 结构体。Env 结构体中包含的信息包括：

- env\_tf “CPU 现场保护信息”，这是一个定义在 inc/trap.h 中的 struct Trapframe 结构体类型的变量，用于保存进程切换时此进程的寄存器信息。当当前进程将控制权转交给 kernel 或其他用户进程时，kernel 会帮助当前进程将这些寄存器的值保存到这个结构体中，以便当前进程“苏醒”之后还能够像什么都没发生过一样继续执行。
- env\_link 这是在空闲 Env 链表上用于指向下一个空闲的 Env 结构。
- env\_id 用于记录当前进程的 id。
- env\_parent\_id 用于记录创建当前进程的父进程 id，这样就可以通过一个进程的 family tree 对进程的权限进行一些限制。在 lab3 中我们还没有用到这个属性。
- env\_type 用于记录进程的类型，尤其是一些特殊的进程。目前它的取值可能为 ENV\_TYPE\_USER 或 ENV\_TYPE\_IDLE。在 lab3 中我们还没有用到这个属性。

- `env_status` 用于记录当前进程的状态。在 `inc/env.h` 中定义了进程的四中状态：`ENV_FREE`, `ENV_RUNNABLE`, `ENV_RUNNING`, `ENV_NOT_RUNNABLE`。注意 `ENV_NOT_RUNNABLE` 状态是指，当前进程由于某种原因暂时不能在 CPU 上运行的状态，如等待资源，等待进程通信，并不是消亡状态。

- `env_runs` 用于记录当前进程在 CPU 上运行的次数。

- `env_pgdir` 指向当前进程虚拟地址空间的页目录。而这个指针是在 `kernel` 下的指针，也就是说它是 `kernel` 下的虚拟地址，要通过访问 `kernel` 的页表才能够得到它到底存储在哪里。任何进程的运行，都需要 `kernel` 帮助建立好相应的“保存的寄存器信息”和“正确的地址空间”，两者缺一不可。

说完了 `struct Env` 结构体，我们就要开始使用它了。在 `kern/env.c` 中定义了下面几个变量：

- `struct Env *envs` 这是进程管理数组，通过它能够找到所有的 `Env` 结构体。`envs` 数组等价于 PCB 表。值得一提的是，JOS 支持的进程数量是静态的（`NENV = 1024` 个，定义在 `inc/env.h` 中），`envs` 数组有多少个表项就能够支持多少个并发的进程。这对 `Env` 结构体的管理几乎完全类似于在内存管理中对物理内存页 `Page` 结构体的管理。我们记得有一个 `pages` 数组记录了所有 `Page`，在这里，`envs` 就记录了所有的 `Env`。打个比方，就好像有 1024 个槽，来一个用户进程，就给它分配一个槽一样。

- `struct Env *curenv` 是指向当前运行进程 `Env` 的指针。

- `static struct Env *env_free_list` 是空闲 `Env` 链表，它的作用和内存管理中的 `page_free_list` 空闲页面链表完全一样。细微的差别在于，`env_free_list` 链表要求在最开始时指向 `envs` 中的第一个 `Env` 即 `envs[0]`。不过后来我发现这样做并不是原则性的，在 1.3.1 中再解释。

我们注意到，上面这三个进程管理数据结构是定义在 `kern/env.c` 中，这是 `kernel` 的私有变量，因此显然只有 `kernel` 才能访问。因此我猜测用户进程之间的切换，必经过“`Env1`→`kernel`→`Env2`”的控制权转换过程。

## 1.2 Allocating the Environments Array

**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 `UENV` (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.

这个 exercise 的内容，是给进程管理数据结构 `envs` 数组分配内存空间，方法和我们在 lab2 中为 `pages` 数组分配空间完全一样。然后将这个数组映射到 `UENV` 处，权限是用户只读。代码如下：

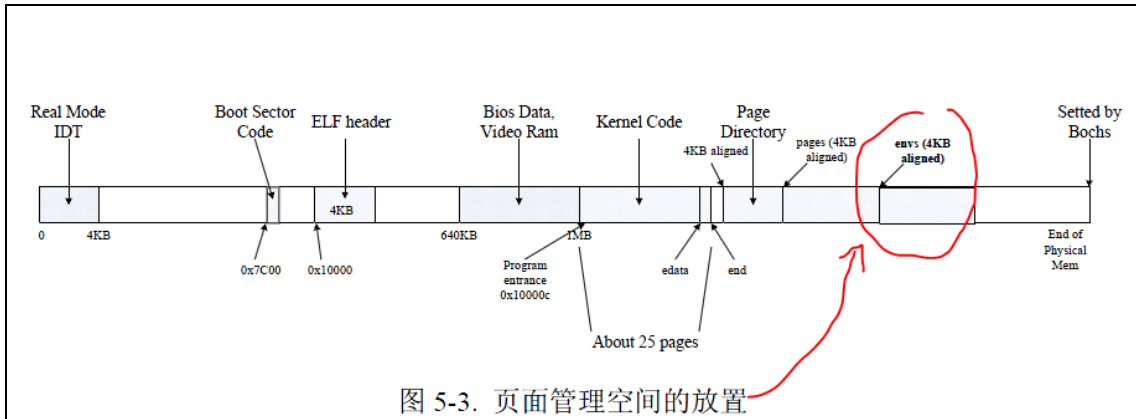
```
// @huangruizhe 2012-4-8
envs = (struct Env *)boot_alloc(NENV * sizeof(struct Env));
memset(envs, 0, NENV * sizeof(struct Env));
```

映射过程：

```
// @huangruizhe 2012-4-8
boot_map_region(kern_pgdir, UENV, ROUNDUP(NENV*sizeof(struct Env),
```

```
PGSIZE), PADDR(envs), PTE_U | PTE_P);
```

envs 数组空间分配完了以后，我们可以看到当前物理内存布局如下：



### 1.3 Creating and Running Environments

**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".

用户进程要运行，就必须让 **kernel** 为它初始化一个 **Env** 结构（里面的内容），将它的可执行代码读入内存，再经过一些初始化步骤后，将 **CS:IP** 指向用户进程的第一条指令，也就



是将控制权转交给用户进程。这一系列准备工作由下面这些 kernel 中的函数完成，再次提醒，下面的工作都是在 kernel 中进行的。

### 1.3.1 env\_init()

刚刚我们为 envs 数组分配了物理内存空间，现在要将每个 Env 结构初始化，然后添加进 env\_free\_list。这就是 env\_init()要做的事情。

具体来说，要将每个 Env 的 env\_status 置为 ENV\_FREE，将 env\_id 置为 0，然后通过设置 env\_link 添加进 env\_free\_list 链表中。代码如下：

```
void
env_init(void) {
    // Set up envs array
    // LAB 3: Your code here.
    // @huangruizhe 20120408
    int i;
    env_free_list = &envs[0];
    envs[0].env_status = ENV_FREE;
    envs[0].env_id = 0;
    for(i = 1; i < NENV; i ++){
        envs[i].env_status = ENV_FREE;
        envs[i].env_id = 0;
        envs[i - 1].env_link = &envs[i];
    }

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

注释中提醒我们需要注意，要保证第一次调用 env\_alloc()时，其中的 env\_init()返回的是 envs[0]。这是为什么呢？原来，这并不是什么原则性的问题。在 kern/init.c 的 i386\_init()中我们看到，JOS 在初始化工作完成之后，会使用下面的代码执行一个用户进程，来检验我们是不是写对了。JOS 假定这个进程的 Env 结构存放在 envs[0]中。囧。

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

在 env\_init()结束之前，它会调用 env\_init\_percpu()加载新的 GDT。加载的方法和我们之前见过的无异。加载新的 GDT 是因为，引入用户进程以后，我们需要在内核态和用户态下使用不同的代码段和数据段。这涉及所谓的段式内存管理。在 JOS 中，我们并没有使用到太多段式内存管理的映射功能，但是这里设置新的 GDT，是为了能够在内核态及用户态之前切换特权级别 DPL：内核态 DPL=0，用户态 DPL=3。

### 1.3.2 env\_setup\_vm()

这个函数的功能，是为某个用户进程初始化它的 Env 结构中的页目录，包括分配物理页面保存页目录，和将页目录中 kernel 部分的内容初始化两个步骤。

分配物理页面，利用 lab2 中写过的 page\_alloc()来完成，为用户进程的页目录分配一页即可。然后设置 e->env\_pgdir，需要注意的是，赋值时要将 page\_alloc()返回的 Page 结构转化成 kernel 中的虚拟地址，也就是 C 指针的内容。

初始化页目录中的 kernel 部分内容，这一步比较可谈。

首先，为什么要将 `e->env_pgdir` 在 `UTOP` 以上的部分设置成和 `kern_pgdir` 一样呢？这是为了便于用户进程通过某些形式访问内核，毕竟我们也注意到了，在 `UTOP` 以上的内容有一部分是用户不可读的，也有一部分是用户可读不可写的。

在了解了地址空间中 `UTOP` 以上用户没有写权限后，我们完全可以在这一部分将 `e->env_pgdir` 设置得和 `kern_pgdir` 的内容一模一样，反正它也不可能破坏到该虚拟地址所映射的物理地址的内容。页目录和页表的本质就是映射，你只要设置了一个数值，就能够映射到那个地方，所以我们不用担心 `e->env_pgdir` 没有为相应页目录项分配页表的物理页，因为它会指向 `kern_pgdir` 已经分配过的页表。

不过这样做，可能会存在两个问题：①我们在页目录项设置的访问权限是最低权限，也即用户可读可写，也就是说页表是用户可读可写的！那么用户进程有没有可能恶意修改 kernel 的页表呢（通过页目录项找到页表，修改之）？②上面做的映射是静态的，如果 `kern_pgdir` 改变了，缺少 `e->env_pgdir` 的更新机制。

代码实现上，主要使用了 `memmove` 和 `memset` 两个函数，想清楚哪些地方一样，哪些地方要置 0 就好，代码如下：

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

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

    // LAB 3: Your code here.
    // @huangruizhe 20120408
    e->env_pgdir = (pde_t *)page2kva(p);
    memmove(e->env_pgdir, kern_pgdir, PGSIZE);
    memset(e->env_pgdir, 0, PDX(UTOP)*sizeof(pde_t));
    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;
    return 0;
}
```

### 1.3.3 env\_alloc()

这个函数没让我们写，但是很重要。`env_alloc()` 函数的功能，是传入父亲 `env_id`，返回一个初始化好的 `Env` 结构，用于保存我们即将想要创建的用户进程的信息。简单说，就是 kernel 为即将创建的进程分配一个 PCB。

首先，它从 `env_free_list` 中取出一个空闲的 `Env` 结构。然后，调用 `env_setup_vm` 初始化 `Env` 的 `env_pgdir`。随后，为该 `Env` 分配一个 `id`，具体是怎么产生 `id` 的呢？下一段我会解释。此后就是初始化 `Env` 结构中的其他变量，这里面比较重要的是对 `env_tf` 的处理。对于

`env_tf`，先将它清零，防止受到上一个正好用过这个 `Env` 的进程的数据的影响；然后设置一些段寄存器的值，置为 `GDT` 中的相应下标，以及访问权限为 3（噢，原来在段式寻址中的权限，是在段寄存器中指定的啊~那么它应该将被用来和 `GDT` 表项的 `DPL` 比较~）。注意，这里没有设置 `eip`，即用户进程第一条指令的地址，为什么呢？这是因为现在才刚刚初始化 `Env` 嘛，我们还没有加载用户进程的可执行代码呢，我们当然不知道 `eip` 要指向哪里啦，所以待会儿还要记得设置 `eip` 哦。（这一句话太萌了有木有！）

关于 `env_id` 的生成。`env_id` 分为三个部分（见 `inc/env.h`），我将它贴在下面：

```
// An environment ID 'env_id_t' has three parts:
//
// +1+-----21-----+-----10-----+
// |0|           Uniqueifier           | Environment |
// | |           |           |           Index       |
// +-----+-----+-----+
//                                     \--- ENVX(eid) ---/
//
// The environment index ENVX(eid) equals the environment's offset in the
// 'envs[]' array. The uniqueifier distinguishes environments that were
// created at different times, but share the same environment index.
```

由以上可知，`env_id` 的低 10 位，恰为该 `Env` 结构在 `envs` 数组中的下标，所以我们看到在最后一行“或“上(`e - envs`)。那么 `generation` 是个什么东东？它就指定了 `env_id` 的高位。`(e->env_id + (1 << ENVGENSHIFT))`是将上一个 `env_id` 的高位自增 1，然后再“与”上`~(NENV - 1)`将低 10 位置零。这样，如果拿出 `env_id` 的高 22 位来看，我们就看到了它在不断地加 1，而低 10 位对于同一个 `Env` 结构是不会变化的。`if` 的作用就是防止 `env_id` 小于 0，也即最高位为 1，如果发生了这种情况，那么高 22 位就从 1 重新开始计数。

经过上面的步骤，一个初始化后的可用 `Env` 就分配出来了。

### 1.3.4 `region_alloc()`

这个函数是为用户进程分配大小为 `len` 字节的物理页面，并将物理页面在 `env_pgdir` 中映射到指定的虚拟地址 `va`。注意是 `env_pgdir` 而不是 `kern_pgdir`。

这样的事情在 `lab2` 中已经做多次了，只要调用 `page_alloc()`和 `page_insert()`就好。需要注意的就是一个对齐的问题。代码如下：

```
static void
region_alloc(struct Env *e, void *va, size_t len){
    // LAB 3: Your code here.
    // @huangruizhe 20120410
    struct Page *p;
    uint32_t offset = (uint32_t)ROUNDDOWN(va, PGSIZE);
    uint32_t upper_bound = offset + len;
    int r;
    for(; offset < upper_bound; offset += PGSIZE) {
        p = page_alloc(0);
        if(p == NULL)
            panic("kern/env.c/region_alloc: out of memory.\n");
        r = page_insert(e->env_pgdir, p, (void *)offset, PTE_U | PTE_W);
        if(r != 0)
            panic("kern/env.c/region_alloc: %e\n", r);
    }
}
```

```
}  
    return;  
}
```

### 1.3.5 load\_icode()

这个函数呢，就是将用户进程的可执行代码读入内存。由于我们的 JOS 目前还没有文件系统，所以采用的策略是将希望运行的用户程序编译后和 kernel 链接到一起，也即 JOS 将用户程序的可执行代码嵌入到 kernel 中当作一个 ELF 可执行文件。

这样一来，载入用户进程的可执行代码，就是读入 ELF 的过程，这和我们在 Lab1 BootLoader 中做的事情几乎一模一样，具有极大参考价值。传入的参数 `binary` 是 ELF 文件的起始地址，通过它从 ELF 头部找到 Program Header，遍历所有的 Program Header，将标记为 `ELF_PROG_LOAD` 的段读入即可。

关于怎么读入，又有可谈的了。

首先，要小心，我们将可执行代码读入的虚拟地址，是在用户进程中的虚拟地址空间而非 kernel 的！因此，正好可以使用刚刚写好的 `region_alloc()` 来进行分配空间和映射，也就设置了 `env_pgdir` 而非 `kern_pgdir`。

其次，要区分好 `ph->p_va` 和 `ph->p_offset`，`ph->p_memsz` 和 `ph->p_filesz` 这两组概念。`ph->p_va` 是这段可执行代码要加载到的、用户进程虚拟地址空间的地址；`ph->p_offset` 是这段可执行代码相对于 ELF 文件起始地址 `binary` 的字节偏移量。`ph->p_memsz` 是这段可执行代码在内存中占据空间的大小；`ph->p_filesz` 是这段可执行代码在 ELF 文件中占据空间的大小。为什么会有两个大小呢？这是因为 ELF 文件中的 BSS 节，里面的变量不会被分配文件存储空间，而是在实际载入内存时，才分配指定的内存空间，并被初始化为 0。因此可以推断 `ph->p_memsz` 不小于 `ph->p_filesz`。

第三，我们在张驰学长的报告中，看到他在载入可执行代码前后，将 `cr3` 先置为 `env_pgdir` 然后又置回 `kern_pgdir`。事实上，这是不必要也是不正确的。首先我们要明确，现在我们仍然一直在 kernel 中运行，在为即将运行的用户进程进行一系列准备工作。其次，在现在这一步，我们只是要设置 `env_pgdir` 用户地址空间中的内容，而不是真正用它来进行虚拟页式寻址。因此，我们没有必要切换页目录页表。

第四，现在我们已知用户进程第一句指令的地址为 `ELFHDR->e_entry`，所以是时候设置 `Env` 结构中的 `env_tf.tf_eip` 了。如果设置不对或忘了设置的话，CPU 等一下就找不到正确的地址执行用户进程了。

第五，最后要为用户进程分配初始的栈空间，也即分配一页物理内存，映射到用户进程 `Env` 虚拟地址空间的 `USTACKTOP - PGSIZE` 处。小心！不是映射到 kernel 的虚拟地址空间，尽管在这一步 `load_icode()` 仍在 kernel 运行！这样，用户进程自己就有“栈”可以使用喽，这个栈就是用户栈。

终于谈完了。代码如下：

```
static void  
load_icode(struct Env *e, uint8_t *binary, size_t size){  
    // LAB 3: Your code here.  
    // @huangruizhe 20120410  
  
    // the elf file is in the memory already, just point to it.  
    // "binary" is a pointer whose step-length is byte! It's important!  
    struct Elf *ELFHDR = (struct Elf *)binary;
```

```

struct Proghdr *ph, *eph;

// is this a valid ELF
if (ELFHDR->e_magic != ELF_MAGIC)
    panic("kern/env.c/load_icode: ELF not valid\n");

ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);
eph = ph + ELFHDR->e_phnum;

physaddr_t kern_pgdir_cr3 = PADDR(kern_pgdir);
for(; ph < eph; ph++){
    if(ph->p_type == ELF_PROG_LOAD){
        // allocate real physical memory for e
        region_alloc(e, (void *)ph->p_va, ph->p_memsz);
        memset((void *)ph->p_va, 0, ph->p_memsz); // clear memory
        first, the code will be more brief.
        memmove((void *)ph->p_va, binary + ph->p_offset,
ph->p_filesz);
    }
}

// set starting point of environment executing
e->env_tf.tf_eip = ELFHDR->e_entry;

// the stack is in the kernal address space. why?
region_alloc(e, (void *) (USTACKTOP - PGSIZE), PGSIZE);
}

```

### 1.3.6 env\_create()

这个函数的作用，是整合了上面函数所作的一切。它通过调用 `env_alloc()` 申请了一个初始化后的 `Env` 结构，然后调用 `load_icode()` 载入用户进程的可执行代码。也就是说，在 `env_create()` 这个函数执行完以后，就为用户进程的运行做好了一切准备，只欠东风了！

```

void
env_create(uint8_t *binary, size_t size, enum EnvType type){
    // LAB 3: Your code here.
    // @huangruizhe 20120410
    struct Env *e;
    int r = env_alloc(&e, 0);
    if(r < 0)
        panic("kern/env.c/env_create: %e\n", r);
    load_icode(e, binary, size);
}

```

### 1.3.7 env\_free() & env\_destroy()

这两个函数功能也很重要，但没什么好说的，因为能看懂。释放/销毁一个用户进程的工作包括：如果释放/销毁的进程是当前运行的进程则切换到 kernel 页目录，释放当前进程占据的所有物理页面，将 Env 结构清空并还回 env\_free\_list。

若是用 env\_destroy() 进行销毁，在销毁完毕后，就转入内核态的 command line 执行了。

### 1.3.8 env\_run() & env\_pop\_tf()

这个函数用于启动用户进程，完成上下文的切换。这两个函数非常重要，在后面会被多次使用。

我们注意到，在 kern/init.c 中调用了 env\_run()，传入了第一个 Env 即 envs[0]。在调用之前必有对 envs[0] 的初始化。因此我们注意到，它是利用 ENV\_CREATE 宏调用了 env\_create()，将我们在参数中指定的用户进程进行初始化。

在 env\_create() 初始化完成之后，用户进程的运行就箭在弦上，只欠东风了！也就是说我们马上就可以调用 env\_run() 启动用户进程。

在 env\_run() 中，首先会对 curenv 以及相关 Env 结构的 status 进行一些设置；然后，设置 cr3 寄存器，将页目录切换成 Env->env\_pgdir；最后，调用 env\_pop\_tf() 设置寄存器的值，这是在 1.3.3 的 env\_alloc() 中设置的段寄存器和在 1.3.5 的 load\_icode() 中设置的 env\_tf.tf\_eip。env\_run() 的代码如下：

```
void
env_run(struct Env *e) {
    // LAB 3: Your code here.
    // @huangruizhe 20120410
    if(curenv != e) {
        if(curenv != NULL && 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));
    }
    env_pop_tf(&curenv->env_tf);
}
```

我们接着来看一下 env\_pop\_tf()，这个函数真正负责完成用户进程的切换。

```
void
env_pop_tf(struct Trapframe *tf) {
    __asm __volatile("movl %0,%%esp\n"
        "\tpopal\n"
        "\tpopl %%es\n"
        "\tpopl %%ds\n"
        "\taddl $0x8,%%esp\n" /* skip tf_trapno and tf_errcode */
        "\tiret"
        : : "g" (tf) : "memory");
    panic("iret failed"); /* mostly to placate the compiler */
}
```

```
}
```

首先，它将用户进程的 `env_tf`（一个 `Trapframe` 结构类型的指针）赋给 `esp`，这是将内存中的 `Trapframe` 结构当作一个栈，准备往外 `pop` 东西。第二步，`popal` 指令将一些寄存器（参见 Intel 指令手册）设置好，这需要我们在设计 `Trapframe` 结构时注意和 `popal` 指令中寄存器赋值的顺序符合。第三步，设置 `es` 和 `ds` 段寄存器。第四步，通过 `iret` 指令（该指令的功能如下图所示）从当前栈顶设置 `CS`, `IP` 和 `EFLAGS` 寄存器。这样，在设置好 `CS:IP` 之后，下一条执行的指令就是用户进程的指令！这就是 `env_run()` 没有返回的原因！

---

**Algorithm 4: IRET - Interrupt Return**

---

```
begin
  EIP ← Pop();
  CS  ← Pop();
  FLAGS ← Pop();
end
```

---

## 2. (Part A) Exception Handling

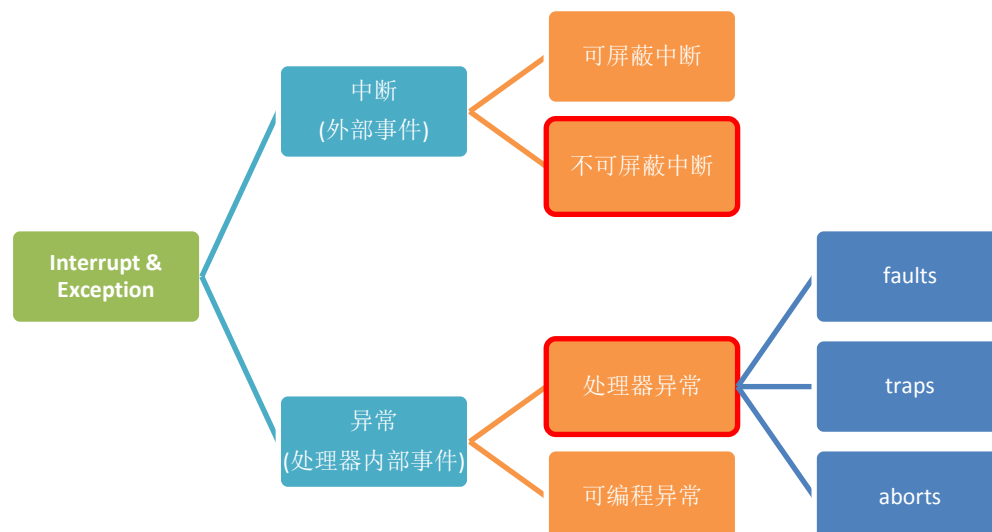
什么是中断（以及异常）？中断就是 CPU 执行到一半，突然发现有个中断请求，然后如果可以的话，就保存现场转而执行中断服务程序，对中断或异常进行响应。举个例子，我现在在写操统实习报告，然后突然打来了一个电话，我检测到这个信号，于是就转而去接电话，在接完电话后再继续回来接着写报告。恩，这个例子让我想起了教我《微机原理》的王克义爷爷，他真是一个和蔼幽默而且又厚道的老师。

### 2.1 Handling Interrupts and Exceptions

**Exercise 3.** Read Chapter 9, Exceptions and Interrupts in the 80386 Programmer's Manual (or Chapter 5 of the IA-32 Developer's Manual), if you haven't already.

这个任务是阅读 Intel 80386 Reference Manual 手册，了解 x86 体系结构中断和异常响应机制相关的内容。我看的是 html 版本的，因为感觉不是很长^^。

中断和异常的分类如下图所示：





其中，“中断”主要是外部事件引起的，“异常”是 CPU 内部的事件引起的。在“异常”的两个子类中，“处理器异常 (Processor detected exception)”我理解为，处理器在执行指令时“自己发现的不对劲”，如除 0 或内存访问越界，要么执行不下去了，要么得先处理不对劲的情况；“可编程异常 (Programmed exception)”是程序员在写程序时，有目的写下的一些指令，如 INT n 等，目的是“引起 CPU 的注意”“故意让 CPU 发现不对劲”，以便它中断手头的事情过来提供一些服务。

不同的事件引起不同的中断或异常，因此得让 CPU 区分开它们，进而才能对症下药。这是通过给不同的中断或异常分配一个 id（也就是我们说的中断向量/中断向量号）来实现的。想法很简单。其中，“不可屏蔽中断”和“处理器异常”（如上图红框所示）的 id 比较特别而固定，因此它们是事先人为规定好的，它们的范围在 0~31，具体的分配参见 intel 手册。此后，32~255 的 id 就可以用来分配给其他中断和异常了，这些分配的 id 可以由“中断控制器”识别并告知 CPU。一般来说，32~255 的 id 被用于识别软件中断 (software interrupts)，它可以是一条 INT n 指令，或非同步的外部硬件中断，如外部的 IO 设备。

CPU 总是在一条指令结束、下一条指令开始前检测并服务中断和异常。手册 9.2 部分介绍了如何使能和屏蔽中断的功能，哪些情况下中断会被屏蔽，以及如何设置 EFLAGS 寄存器中 IF 和 RF 位。手册 9.3 介绍了当多个中断和异常同时发生时，中断和异常的优先级和 CPU 响应顺序。

关于中断描述符表 IDT，这比较重要，涉及到的概念有：中断或异常 id，IDT，IDTR，中断描述符。首先，当 CPU 通过 id 识别出了一个中断或异常，它就会拿着这个 id 到 IDT 去查表，看看该怎么处理这个中断或异常，精确地说，就是要通过 id 获得中断服务程序的入口地址。中断描述符表 IDT 就是一个数组，每个表项是一个中断描述符，每个中断描述符的大小是 8 字节，因此 CPU 将 IDT 首地址加上 id\*8 个字节的偏移量，就可以获得 id 对应的中断描述符。这是不是和我们组织 GDT 的方法一样^^？其次，CPU 是通过 IDTR 寄存器在内存中找到 IDT 的，IDTR 寄存器包括 32 位的 IDT 起始地址和 16 位来记录 IDT 的长度。但是为什么我们只支持 256 个中断和异常 id 呢？我不清楚哦。

中断描述符格式参见手册 9.5.

手册 9.6 介绍了 Interrupt Tasks 和 Interrupt Procedures，JOS 不涉及 Interrupt Tasks 因此我们只需要涉及后者，即中断服务程序。至于为什么我知道不涉及呢，这是在参考了前人资料和做完了之后才知道的(ToT)。对这一部分的概述我觉得手册中写得非常非常到位（我是认真的），翻译灌水如下。

---

正如一条 CALL 指令，能够调用一个 procedure 或 task；一个 interrupt 或 exception 也能“调用”一个 interrupt 的 handler，这个 handler 也可以是一个 procedure 或 task。当 CPU 对 interrupt 或 exception 做出响应，CPU 将会用对应的 id 作为索引查询 IDT 表获得描述符。如果这个描述符是一个 trap gate（我们就把描述符当作是一个“门”吧，你不觉得很生动形象么？通过描述符，我们获得中断服务程序的起始地址，就能够“进入”中断服务程序，就好像经过了一个“门”一样，当然，“门”有很多种了，比如木门，防盗门，校门，登机口，安检的那种门等等，你进不同的门，目的和动作都是不一样的。——译者注），那么 CPU 调用的 handler 将和 CALL 经过一个 call gate 的机制类似；如果描述符是一个 task gate，那么 CPU 将引起一个和 CALL 经过一个 task gate 机制类似的任务切换。

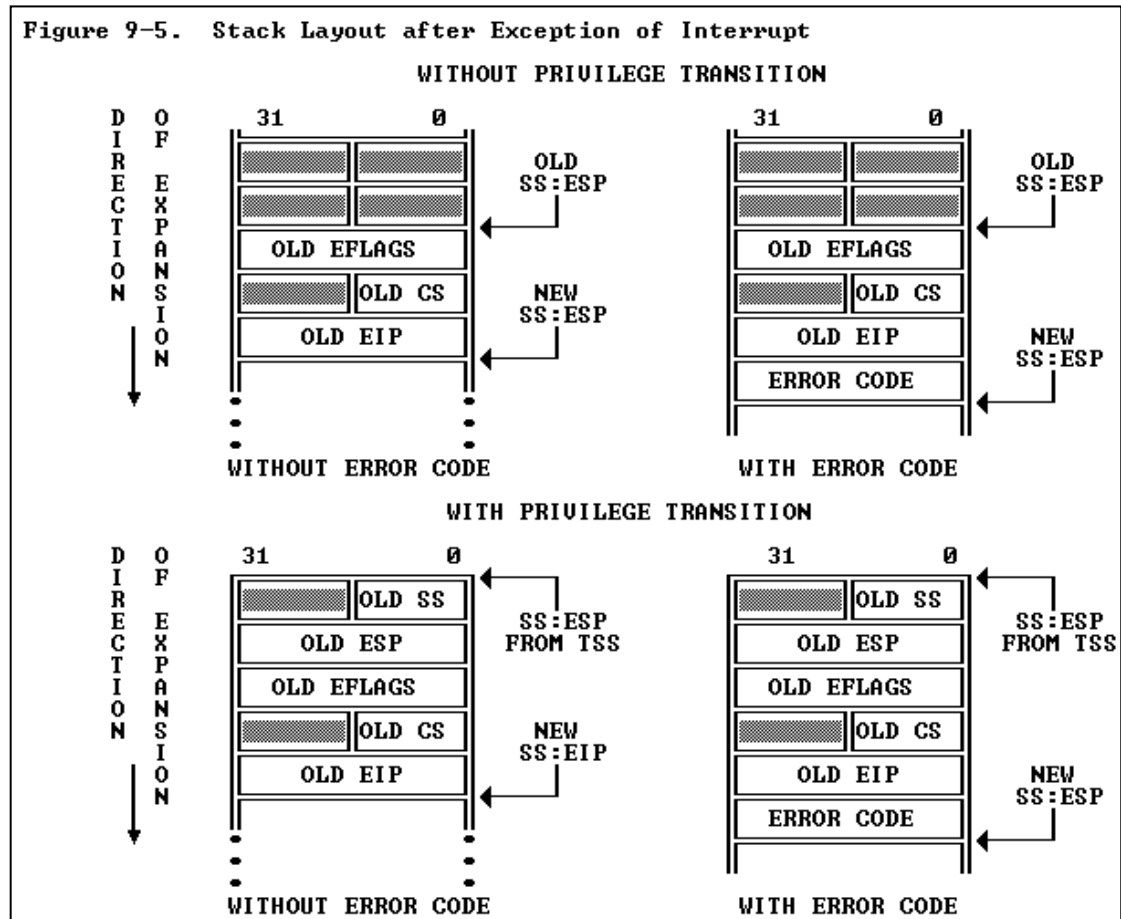
---

通过 id 查询 IDT 得到描述符（对描述符分类，可能是三种“门”），我们就得到了中断服务程序的起始地址；中断服务程序要做好哪些准备工作，要提供哪些服务，这是系统程序员编



写的。

首先，要跳转到中断服务程序，我们得为当前运行的程序保存现场，这就需要一个 **Stack of Interrupt Procedure**。它的作用是保存必要的、从中断服务程序返回原程序的信息。栈是在内存中的，可是你有没有想过它具体在哪里呢？访问它的权限是什么？我们将在 2.2 再介绍。在这里我们只要注意，这个栈不是用户栈，而是切换到一个新的、系统定义的栈。当中断发生时，CPU 首先**自动地**将当前运行程序的 EFLAGS 寄存器压入栈中，然后压入 CS 和 IP，然后对于一些特殊的异常 CPU 还会压入 error code，然后才会转到中断服务程序的第一条指令。中断或异常发生后，栈的布局将会这样变化（仔细看还是能看懂的!）：



应当注意的是，上图所示的栈布局是中断发生后，CPU 自动帮我们做好的：①左上角是“未发生特权级别转换、没有 error code”的情况，我们看到栈是自顶向下生长，OLD SS:ESP 指向中断发生前的栈顶，中断发生后，CPU 自动将原程序的 EFLAGS、原程序的 CS:IP 依次压入栈中，NEW SS:ESP 将指向新的栈顶。②右上角是“未发生特权级别转换，有 error code”的情况，我们注意到，和左上角的情况一样，就是 CPU 自动地在 OLD EIP 上额外压入了 ERROR CODE。③左下角是“要发生特权级别转换、没有 error code”的情况，和左上角的不同在于，它的初始栈底是 TSS 中指定的 SS:ESP，且它在 OLD EFLAGS 前面压入了 OLD SS 和 OLD ESP。这是干什么用的呢？别急，我们在 2.2 会详细解答这个问题。这里我只要告诉你，由于特权级别发生转换，要求“栈”空间也要进行切换，这里的 OLD SS 和 OLD ESP 记录了转换前的栈空间地址，以便恢复。④右下角是“要发生特权级别转换、有 error code”的情况，不再赘述。

其次，考虑中断服务程序执行完以后的返回。这是通过一条 IRET 指令来完成，前面我贴过了张驰学长总结的 IRET 指令的工作原理，在 html 版本的手册中，我们也可以点开 IRET

的超链接，哇，手册解释得好详细。IRET 和 RET 的不同在于，IRET 会从栈中多弹出一个 EFLAGS 寄存器的值，而 RET 不需要。

第三，关于 TF，IF 标志位。手册告诉我们中断（either interrupt gates or trap gates）发生以后 CPU 会自己重置 TF 寄存器的值，防止单步跟踪调试对中断服务的影响。只有当中断服务完成以后，CPU 才会将 TF 寄存器重新设置成原来的值。其次，分别从 interrupt gate 和 trap gate 进入的中断服务程序的区别在于，前者将重新设置 IF 使得其他中断不能打断当前中断服务，后者不改变 IF 的值。

---

首先复习一下预备知识，通用寄存器 EFLAGS 保存的是 CPU 的执行状态和控制信息，如下图，其中我们只需要关注两个寄存器：IF 和 TF。

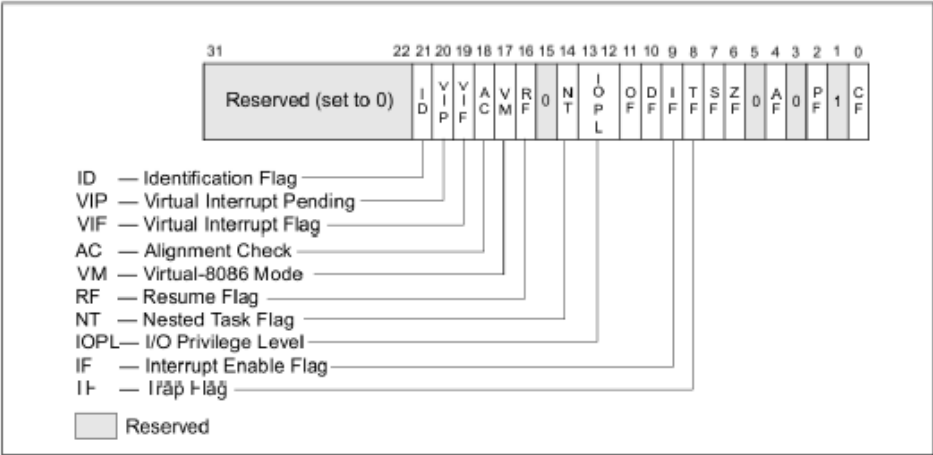


Figure 2-4. System Flags in the EFLAGS Register

**TF(Trap Flag) :**

跟踪标志。置1则开启单步执行调试模式，置0则关闭。在单步执行模式下，处理器在每条指令后产生一个调试异常，这样在每条指令执行后都可以查看执行程序的状态。

**IF (Interrupt enable) :**

中断许可标志。控制处理器对可屏蔽硬件中断请求的响应。置1则开启可屏蔽硬件中断响应，置0则关闭可屏蔽硬件中断响应。IF 标志不影响异常和不可屏蔽中断（NMI）的产生。

---

门用来实现从一段代码跳转到另一段代码（可能在不同的代码段，不同特权级）时的保护机制问题。

---

第四，中断服务程序中的特权级以及保护。我没细看，不过这里面提到任何试图越权的行为都会导致 general protection exception，这个知识还是挺 general 的。

上面我详细总结了中断服务程序运行前后的机制。接下来，手册 9.7 介绍了 error code 及其格式。手册 9.8 详细介绍了各种异常 exception，以及异常中 faults, traps, aborts 三个类别的区别。这一部分可以等到我们需要的时候再查。

啊，手册看完了，好开心（其实是写报告的时候累死了）！

## 2.2 Basics of Protected Control Transfer

什么是“Protected Control Transfer（受保护的转移机制）”？

中断和异常都是 Protected Control Transfer，意思是它们都可能引起 CPU 从用户态切换到内核态（CPL=0），并且绝不会让用户态的进程对 kernel 或其他进程产生干扰。怎么保证这件事情，即保证 protected？这就是“受保护的转移机制”的内容。简单地说，当用户进程需要执行特权指令或者内核功能时，或者中断/异常事件发生时，用户进程不能够自己决定它要进入 kernel 的哪里去执行，怎么进入；而是 CPU 提供两个机制一起保证从用户态进入内核态是受约束的：

### 2.2.1 中断描述符表 IDT

这决定了用户进程进入 kernel 中的哪个位置去执行，即中断服务程序的起始地址。我在 2.1 中已经介绍过。

### 2.2.2 任务状态段 TSS

中断或异常发生后，CPU 要先去处理它们，稍后再回来继续执行当前的事情，因此 CPU 要保存现场，记住刚刚它执行到哪里了（CS:IP），以及当时执行的状态。那么这些保存现场的信息记录在哪里呢？在 2.1 中，我们说过，这需要一个栈来保存。显然，如果用户有权限访问和修改这个栈的内容，那么恶意代码将会导致程序崩溃，也即我们刚才从哪里来，现在却回不去了(ToT)；因此这个栈需要被保护，即不能让没有特权的用户代码随便访问。

因此，当 CPU 发现一个将要引起特权级别从用户态切换到内核态的 interrupt 或 trap（一种 exception）时，CPU 将会自动地切换到 kernel 空间中的一个栈，切换动作也即设置 ESP 和 SS 寄存器。这时，就由 TSS 来指定这个栈在 kernel 地址空间中的位置，即 ESP 和 SS 寄存器的值。然后 CPU 将会如我们在 2.1 介绍的，把 SS, ESP, EFLAGS, CS, EIP, 以及可选的 error code 压入这个切换后的、kernel 中的栈中。

TSS 结构主要用于描述代码切换之间权限的转换，里面保存了很多有用的信息。但是 JOS 只用它来指定当从用户态切换到内核态时，需要使用的、kernel 中的、栈的地址。仅此而已。

## 2.3 Types of Exceptions and Interrupts

这就是我们在 2.1 中介绍的中断或异常 id（中断向量）的分配。

### 2.4 An Example

现在 CPU 正在欢乐地执行一个用户程序，它突然发现有一条指令试图除以 0，于是：

1. CPU 切换栈空间，这个栈空间的地址在 TSS 中指定；
2. CPU 往新的栈空间压入数据，依次是 SS, ESP, EFLAGS, CS, EIP，栈的布局如我们在 2.1 图中左下角“要发生特权级别转换、没有 error code”的情况；
3. 除 0 异常的中断向量 id 是 0，因此 CPU 读取 IDT 中第一个描述符，并按照描述符设置了 CS:IP，这就指向了除 0 异常相应的中断服务程序的起始地址；
4. 中断服务程序执行，并处理异常。

### 2.5 Nested Exceptions and Interrupts

这是我们在 2.1 图中上方“没有发生特权级别转换”的中断或异常的栈布局情况。如果 CPU 本身已经在 kernel 中，这时 kernel 中的指令又引起了新的中断或异常，这时候，我们就需要进行特权级别的转换，CPU 直接在 kernel 当前栈上压入保存现场的信息。这时我们就

只需要压入 EFLAGS, CS, IP 和可选的 error code 了。

## 2.6 Setting Up the IDT

**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.

在这个 Exercise 我们要把 IDT 建立起来，IDT 本质是在 `kernel` 中的一个数组，是中断向量 `id` 到中断描述符（也即中断服务程序起始地址）的映射。

下图显示了 JOS 中断/异常处理的整个过程的控制流，包括“陷入过程”和“返回过程”，其中黄色线条为中断返回的过程，重要的节点用蓝色点标出，并加以注释。我觉得在这个 lab，以及后续的操作系统的课程中，将这个示意图牢记于心，非常重要。

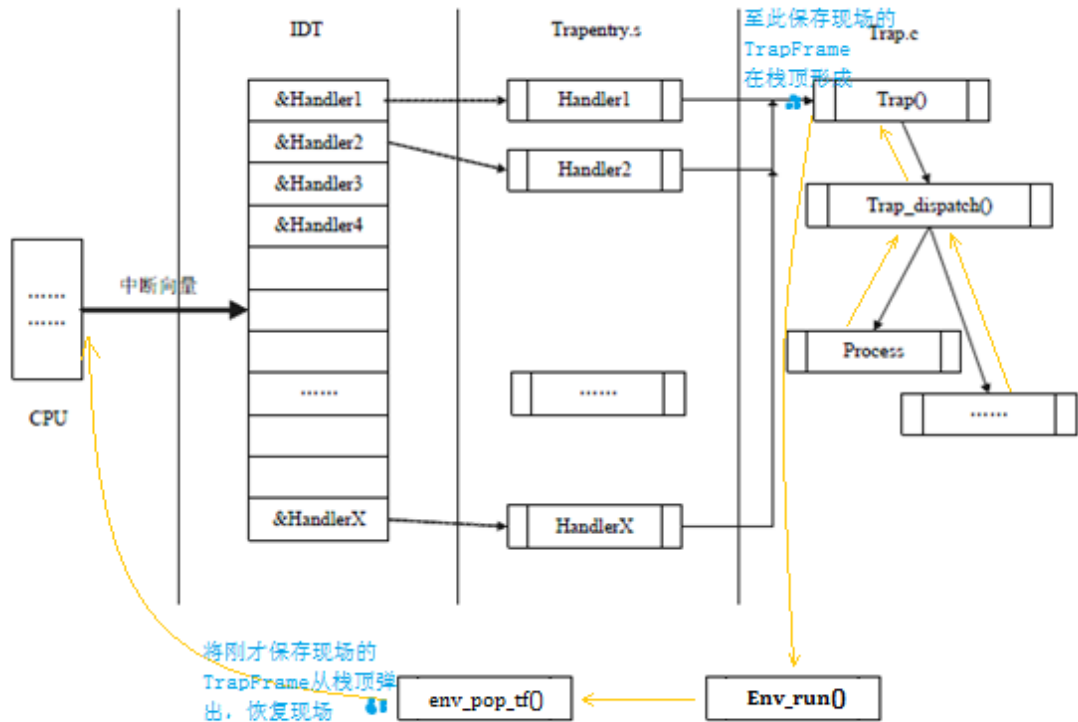


图 5-10. JOS 系统中断过程的控制流

在这个 Exercise 中我们要填写 IDT, 就要知道每种中断或异常各自的 handler 的地址是什么, 这个地址是 kernel 中的虚拟地址。事实上, 这个 handler 也是我们来写的, handler 作为中断或异常跳转的第一步, 是要能够让 CPU 将一个 Trapframe 结构 (就是“保存现场”的信息) 在栈顶补充完整, 以便后续处理步骤能够从栈顶正确 pop 出必要的信息 (取中断号以及“恢复现场”)。

我们要获得 handler 的内核虚拟地址, 然而, 这些 handler 的地址在最终代码生成之前是无法确定的, 也不会是固定不变的, 毕竟你怎么知道它最终链接到 kernel 地址中的哪里? 因此, 在填写 IDT 的时候, 在 C 代码中我们可以用“函数名(Label)”来指定 handler 的地址, 这样在链接的时候就能相应替换成虚拟地址了。关于这个问题, 在 challenge 1 还会遇到。

具体实现步骤如下:

首先, 我们利用 TRAPHANDLER 和 TRAPHANDLER\_NOEC 宏来实现 handler。这两个宏所生成的代码的作用是, 根据传入参数定义 handler 的“标签”, 然后进行一些压栈操作。压什么东西? 在 2.1 我们已经知道, 中断发生后, CPU 自动切换到一个新的栈, 然后自动地将当前运行程序的 EFLAGS 寄存器压入栈中, 然后压入 CS 和 IP, 然后对于一些特殊的异常 CPU 还会压入 error code, 然后才会转到中断服务程序的第一条指令——这第一条指令就是在 handler 中。

然后, handler 就开始压栈了。为了保证 Trapframe 结构的一致性和完整性, 对于 CPU 没有压入 error code 的情况, TRAPHANDLER\_NOEC 定义的 handler 首先压入一个 0 用于占位。

接着, TRAPHANDLER 和 TRAPHANDLER\_NOEC 都压入了 trap number。

最后, 两者都跳转到 \_alltraps, 这是每个 handler 都一样的代码, 它的作用依然是继续补充 Trapframe 结构, 注意压栈顺序要和 Trapframe 结构中的顺序一致。我们稍后就会发现, 经过上述一系列操作之后, 一个完整的 Trapframe 结构诞生了, 也就意味着保存现场的工作

完成了（随时在脑海里将这两个概念等价：**TrapFrame=保存现场**）！

代码如下。接下来 CPU 又即将跳转到哪里去呢？

用宏定义 handler:

```
.text
// @huangruizhe 20120411
//note: the followings are macros, not functions!
TRAPHANDLER_NOEC(Entry_Divide_Error, T_DIVIDE);
TRAPHANDLER_NOEC(Entry_Debug_Exception, T_DEBUG);
TRAPHANDLER_NOEC(Entry_NMI_Exception, T_NMI);
TRAPHANDLER_NOEC(Entry_Breakpoint, T_BRKPT);
TRAPHANDLER_NOEC(Entry_Overflow, T_OFLOW);
TRAPHANDLER_NOEC(Entry_Bound_Check, T_BOUND);
TRAPHANDLER_NOEC(Entry_Illegal_Opcode, T_ILLOP);
TRAPHANDLER_NOEC(Entry_Device_Not_Available, T_DEVICE);
TRAPHANDLER(Entry_Double_Fault, T_DBLFLT);
TRAPHANDLER(Entry_T_COPROC, -1); // Reserved, 9, 占位
TRAPHANDLER(Entry_Invalid_TSS, T_TSS);
TRAPHANDLER(Entry_Segment_Not_Present, T_SEGNP);
TRAPHANDLER(Entry_Stack_Exception, T_STACK);
TRAPHANDLER(Entry_General_Protection_Fault, T_GPFLT);
TRAPHANDLER(Entry_Page_Fault, T_PGFLT);
TRAPHANDLER(Entry_T_RES, -1); // Reserved, 15, 占位
TRAPHANDLER_NOEC(Entry_Floating_Point_Error, T_FPERR);
TRAPHANDLER(Entry_Alignment_Check, T_ALIGN);
TRAPHANDLER_NOEC(Entry_Machine_Check, T_MCHK);
TRAPHANDLER_NOEC(Entry_Simd_Floating_Point_Error, T_SIMDERR);
```

\_alltraps 是所有 handler 共有的部分:

```
// @huangruizhe 20120411
_alltraps:
    pushw $0x0
    pushw %ds
    pushw $0x0
    pushw %es
    pushal

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

    pushl %esp
    call trap
```

在 kern/trap.c 的 trap\_init() 函数对 IDT 初始化，也即填写 IDT:

```
extern void Entry_Divide_Error();
extern void Entry_Debug_Exception();
```

```

extern void Entry_NMI_Exception();
extern void Entry_Breakpoint();
extern void Entry_Overflow();
extern void Entry_Bound_Check();
extern void Entry_Illegal_Opcode();
extern void Entry_Device_Not_Available();
extern void Entry_Double_Fault();
extern void Entry_Invalid_TSS();
extern void Entry_Segment_Not_Present();
extern void Entry_Stack_Exception();
extern void Entry_General_Protection_Fault();
extern void Entry_Page_Fault();
extern void Entry_Floating_Point_Error();
extern void Entry_Alignment_Check();
extern void Entry_Machine_Check();
extern void Entry_Simd_Floating_Point_Error();
SETGATE(idt[T_DIVIDE], 0, GD_KT, Entry_Divide_Error, 0);
SETGATE(idt[T_DEBUG], 0, GD_KT, Entry_Debug_Exception, 0);
SETGATE(idt[T_NMI], 0, GD_KT, Entry_NMI_Exception, 0);
SETGATE(idt[T_BRKPT], 0, GD_KT, Entry_Breakpoint, 3);    //break
point needs no kernel mode privilege
SETGATE(idt[T_OFLOW], 0, GD_KT, Entry_Overflow, 0);
SETGATE(idt[T_BOUND], 0, GD_KT, Entry_Bound_Check, 0);
SETGATE(idt[T_ILLOP], 0, GD_KT, Entry_Illegal_Opcode, 0);
SETGATE(idt[T_DEVICE], 0, GD_KT, Entry_Device_Not_Available, 0);
SETGATE(idt[T_DBLFLT], 0, GD_KT, Entry_Double_Fault, 0);
SETGATE(idt[T_TSS], 0, GD_KT, Entry_Invalid_TSS, 0);
SETGATE(idt[T_SEGNP], 0, GD_KT, Entry_Segment_Not_Present, 0);
SETGATE(idt[T_STACK], 0, GD_KT, Entry_Stack_Exception, 0);
SETGATE(idt[T_GPFLT], 0, GD_KT, Entry_General_Protection_Fault, 0);
SETGATE(idt[T_PGFLT], 0, GD_KT, Entry_Page_Fault, 0);
SETGATE(idt[T_FPERR], 0, GD_KT, Entry_Floating_Point_Error, 0);
SETGATE(idt[T_ALIGN], 0, GD_KT, Entry_Alignment_Check, 0);
SETGATE(idt[T_MCHK], 0, GD_KT, Entry_Machine_Check, 0);
SETGATE(idt[T_SIMDERR], 0, GD_KT, Entry_Simd_Floating_Point_Error,
0);

```

接下来，在 `_alltraps` 中调用了定义在 `kern/trap.c` 中的 `trap()` 函数。

我们进入 `trap()` 函数中，它的参数正是当前栈顶的 `Trapframe` 结构。`trap()` 函数首先做了一些杂七杂八的工作，这不用管，然后调用 `trap_dispatch()` 函数进行“派发”。

进入 `trap_dispatch()` 函数，这里将根据参数 `Trapframe` 中的 `tf_trapno` 域，调用不同的、真正执行特定功能的中断服务程序。

中断服务程序开始执行，它们和平时我们所见到的一般的函数没什么两样，都是完成特定的功能。待中断服务程序执行完毕，好，现在它要返回原程序刚才执行到、被迫中断的那条指令的地址了。怎么返回？我们看到，程序首先返回 `trap_dispatch()`，然后返回 `trap()` 函数，

然后，调用了 `env_run(curenv)`，在里面调用了 `env_pop_tf()`。通过这几个步骤，程序指针就能够置为刚才我们保存过的 `TrapFrame` 里面的中断地址，原来的程序就可以接着执行了！

**Challenge1.** 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`.

题目写得很清楚，当然，令人遗憾的是这种感觉只有在做完之后才会恍然大悟：原来题目是这个意思啊 T.T~~

题目大意：我们在 `trapentry.S` 中定义了不少 TRAPHANDLER，以及在 `trap.c` 中将这些 TRAPHANDLER 注册到 IDT，然而这其中存在着大量相似代码。我们能不能对 `trapentry.S` 中的 TRAPHANDLER 稍加修改，使之能够自动形成一个 entry point 的表格，使得在 `trap.c` 就能够统一地使用了（而不用写那么多重复的代码用于把 TRAPHANDLER 一个一个地注册）？

做这个 challenge 需要了解两件事情：

第一，TRAPHANDLER 的地址在最终代码生成之前是无法确定的，也不会是固定不变，因此我们要用“函数名(Label)”来指定 TRAPHANDLER 的地址，这样在链接的时候就能相应替换成虚拟地址。

第二，怎么在汇编中形成表格？自然联想到“数据段”。题目中提示：“我们可以通过 `.text` 和 `.data` 指令告诉汇编器，我们现在正在代码段和数据段之间切换哦，你要把这些数据和代码分别放入相应的地方呢亲~”

对什么内容建立表格？当然是导致我们在 `trap.c` 中注册 TRAPHANDLER 时不得不一个一个分别处理的那些信息，主要是起始地址和特权级别。这些信息能否在定义 TRAPHANDLER 时确定？答案是可以。用 `.data` 切换到数据段，起始地址我们只要用 `name` 标签就可以了，声明数据类型为 `long`；至于特权级别，我们可以给 TRAPHANDLER 和 TRAPHANDLER\_NOEC 宏多传入一个参数来指定。在我的实现中，我直接在注册时再设置特权级别，代码量差别不大：

用宏定义 `handler`（另一个宏类似，只是多了 `.data` 段），其中包含数据段的“地址”信息，用 `label` 表示：

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

.data
.globl _idt_entry_code
.align 2
_idt_entry_code:
```

初始化 IDT:



```

extern uint32_t _idt_entry_code[];
int i;
for(i = 0; i <= T_SIMDERR; i++){
    SETGATE(idt[i], 0, GD_KT, _idt_entry_code[i], 0);
}
// system call of course will be invoked in user mode, 特殊处理
extern void Entry_Breakpoint();
SETGATE(idt[T_BRKPT], 0, GD_KT, Entry_Breakpoint, 3);

```

### Question

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. 的确，现在对于每个中断或异常，我们都通过 TRAPHANDLER 或 TRAPHANDLER\_NOEC 宏来实现了一个它自己的 handler。这个 handler 是不能共用的。若共用，首先无法处理有/没有 error code 的情况，其次无法记录当前发生的中断号，供后续处理。

2. 虽然 user/softint 调用的是 int 14 (page fault)，但是我们在 IDT 中设置的 page fault 的特权级别是 0，也就是说只有内核才能产生该中断。因此，这时的用户就“越权”了，CPU 产生一个 general protection fault (trap 13) 进行保护。如果 kernel 允许用户自己调用 int 14 即 page fault 指令，那么恶意的用户程序可能会不断地调用 page fault，然后将物理内存占满。所以 page fault 只能交给 kernel 代为处理。

## 3. (Part B) Page Faults, Breakpoints Exceptions, and System Calls

在以上的中断和异常处理流程全部实现以后，剩下的任务就是书写“真正的”中断处理程序，以及在 trap\_dispatch() 函数中根据 tf\_trapno 域 dispatch 出去。

### 3.1 Handling Page Faults

**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.

“页面错误 page fault”的中断向量号为 14，当处理器遇到一个 page fault，它将会把引

起该 page fault 的线性地址，保存到 CR2 寄存器中。这个 exercise 比较简单，我们只需要将中断流程 dispatch 到 page\_fault\_handler()，代码如下；之后我们将进一步完善 page fault handler 的功能。

```
// @huangruizhe 20120411
int r;
switch(tf->tf_trapno) {
case T_PGFLT:
    page_fault_handler(tf);
    break;
...
default:
    break;
}
```

### 3.2 The Breakpoint Exception

“断点异常 breakpoint exception”的中断向量号为 13，在 JOS 中，我们用类似“系统调用”的方式来在用户程序中设置断点，一旦发生 breakpoint exception，则进入 JOS kernel monitor.

**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.

这个 exercise 很简单，类似 exercise 5，只要在 trap\_dispatch() 中加入相应分支语句即可，代码如下：

```
case T_BRKPT:
    monitor(tf);
    break;
```

**Challenge2.** 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 是让我们在 JOS kernel monitor 加入下面两条命令，当用户程序由 breakpoint exception 陷入内核进入 JOS kernel monitor 后：

- continue: 通过 continue 命令，可以让用户程序从“陷入地址 current location”重新

继续运行起来。

- **single-step**: 通过 **single-step** 命令实现“单步调试”，可以让用户程序从“陷入地址”执行一条指令后，然后发生“调试异常”陷入内核并进入 JOS kernel monitor.

在了解中断处理（及其返回）流程之后，这两个命令还是很好实现的。**continue** 的实现，就是调用 **env\_run()** 函数实现中断返回，相关代码如下：

在 **kern/monitor.c** 的 **static struct Command commands[]** 数组中添加调用接口：

```
{ "continue", "Continue execution from the current location", mon_continue },
```

**continue** 命令相应的处理代码：

```
int mon_continue(int argc, char **argv, struct Trapframe *tf){
    extern struct Env * curenv;
    if ( tf == NULL ){
        cprintf("Cannot Continue: (tf == NULL)\n");
        return -1;
    }
    else if ( tf->tf_trapno != T_BRKPT && tf->tf_trapno != T_DEBUG ){
        cprintf("Cannot Continue: wrong trap number\n");
        return -1;
    }
    cprintf("Continue execution from the current location...\n");
    tf->tf_eflags &= ~FL_TF; //将TF位置0，小心“与”！
    env_run(curenv);
    return 0;
}
```

实现 **single-step** 需要了解“调试异常”及其相应的控制场位。我们需要明确，单步调试功能是如何实现的。在通用寄存器 **EFLAGS** 中的 **TF** 位（跟踪标志），将其置 1 则开启单步执行调试模式，置 0 则关闭。在单步执行模式下，处理器在执行每条指令后将产生一个“调试异常”，调用相应的异常处理程序。

首先，需要在 **trap\_dispatch()** 函数中添加对 **T\_DEBUG** 的处理，同样是 **dispatch** 到控制台：

```
case T_DEBUG:
    monitor(tf);
    break;
```

在 **kern/monitor.c** 的 **static struct Command commands[]** 数组中添加调用接口：

```
{ "si", "Single-step one instruction at a time", mon_si },
```

**single-step** 命令相应的处理代码：

```
int mon_si(int argc, char **argv, struct Trapframe *tf){
    extern struct Env * curenv;
    if ( tf == NULL ){
        cprintf("Cannot Continue: (tf == NULL)\n");
        return -1;
    }
    else if ( tf->tf_trapno != T_BRKPT && tf->tf_trapno != T_DEBUG ){
        cprintf("Cannot Continue: wrong trap number\n");
        return -1;
    }
}
```

```

}
cprintf("Single-step one instruction at a time...");
tf->tf_eflags |= FL_TF; //将TF值1, 单步调试, 小心"或"!

struct Eipdebuginfo info;
debuginfo_eip(tf->tf_eip, &info); // See mon_backtrace() in Lab1
cprintf("\nSi\ information:\ntf_eip=%08x\ns:%d: %.*s+%d\n",
        tf->tf_eip, info.eip_file, info.eip_line,
info.eip_fn_namelen, info.eip_fn_name, tf->tf_eip-info.eip_fn_addr);

env_run(curenv);
return 0;
}

```

我们对上面的两条命令进行测试。首先模仿 user/breakpoint.c 写一个测试的用户程序:

```

#include <inc/lib.h>
void
umain(int argc, char **argv){
    cprintf("this is before int 3\n");
    asm volatile("int $3");
    cprintf("this is after int 3 (1)\n");
    cprintf("this is after int 3 (2)\n");
}

```

运行截图, continue 命令:

```

eraser@ubuntu: ~/eclipseWorkspace/lab
[00000000] new env 00001008
this is before int 3
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
TRAP frame at 0xf02903e0 from CPU 0
edi 0x00000000
esi 0x00000000
ebp 0xeebdfd0
oesp 0xefbfffdc
ebx 0x00000000
edx 0xeebfde88
ecx 0x00000015
eax 0x00000015
es 0x---0023
ds 0x---0023
trap 0x00000003 Breakpoint
err 0x00000000
eip 0x00800047
cs 0x---001b
flag 0x00000292
esp 0xeebdfdb8
ss 0x---0023
K> continue
Continue execution from the current location...
this is after int 3 (1)
this is after int 3 (2)
[00001000] exiting gracefully
[00001008] free env 00001008
No more runnable environments!
Welcome to the JOS kernel monitor!

```

运行 si 命令:

```
K> si
Single-step one instruction at a time..."Si" information:
tf_eip=00800047
user/breakpoint.c:16: umain+19
TRAP frame at 0xf02903e0 from CPU 0
edi 0x00000000
esi 0x00000000
ebp 0xeebdfd0
oesp 0xefbfffdc
ebx 0x00000000
edx 0xeebfde88
ecx 0x00000015
eax 0x00000015
es 0x---0023
ds 0x---0023
trap 0x00000001 Debug
err 0x00000000
eip 0x0080004e
cs 0x---001b
flag 0x00000392
esp 0xeebdfdb8
ss 0x---0023
[00001008] free env 00001008
No more runnable environments!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

### 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. 注意到我们实现的 break point 很像一个系统调用，用户通过 int 3 指令便产生一个 breakpoint exception. 因此，我们需要考虑权限的问题，也就是说用户有没有权限执行 int 3 呢？这取决于 IDT 的设置。如果在初始化 IDT 时把断点异常对应中断描述符的 DPL 权限级别设为 3（我们在 trap\_init()中就是这样实现的），那么用户态下可以执行 int 3，这时产生的一个 breakpoint exception 是符合我们的预期的。如果在初始化 IDT 时把断点异常对应中断描述符的 DPL 权限级别设为 0，那么用户无权执行 int 3 指令，这时将会产生一个 general protection fault.

4. “Protected Control Transfer（受保护的控制转移机制）”问题，回到我的报告的 2.2 部分。当中断和异常发生时（任何引起 CPU 从用户态切换到内核态的情形），必须保证绝不会让用户态的进程对 kernel 或其他进程产生干扰。因此，当用户程序在进入内核时，只能通过少数几个经过事先精心设计好的入口，进行权限检查，并由 kernel 指定跳转到的目标地址。我们在初始化 IDT 时，就设置了“用户级中断/异常”和“内核级中断/异常”。在 user/softint 程序中试图自己产生一个 page fault，即运行 int 14 指令，这是一个“内核级中断/异常”，将产生一个 general protection fault.

### 3.3 System calls

用户程序通过“系统调用”来请求内核为其提供服务。系统调用时，将发生权限级别从用户态到内核态的转换，处理器进入内核态。处理器和 kernel 共同合作保存好用户程序陷入前的执行状态后，kernel 便开始执行相应的系统服务程序，执行完后便返回用户程序。

这里有两个问题：①用户程序怎么引起 kernel 的注意，为其提供服务？②用户程序如何告诉 kernel 为它提供什么服务？

对于第一个问题，JOS kernel 通过让用户执行 int 指令产生一个处理器中断。具体来说，是 int 0x30（十进制的 48）。显然，在做过 breakpoint exception 之后，我们知道系统调用的中断也应在 IDT 中设置为“用户级”。

对于第二个问题，用户程序是通过寄存器来传递系统调用号和系统调用参数的。具体来说，用户程序在寄存器 %eax 中存放系统调用号，在寄存器 %edx, %ecx, %ebx, %edi, 和 %esi 中分别存放（最多 5 个）参数。同时，kernel 将系统调用返回值放在寄存器 %eax 中。见 lib/syscall.c 中的 syscall() 函数，这一机制已经为我们实现好了。

**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.

这个 exercise 是让我们在 JOS 中，将系统调用的流程补充完整。

首先，在 kern/trapentry.S 中添加相应的 handler：

```
TRAPHANDLER_NOEC(Entry_System_Call, T_SYSCALL);
```

第二步，在 kern/trap.c's trap\_init() 中初始化 IDT，注意特权级别设置为 3（用户级）：

```
SETGATE(idt[T_SYSCALL], 0, GD_KT, Entry_System_Call, 3);
```

第三步，在 trap\_dispatch() 增加对系统调用的 dispatch，即调用 syscall() 函数，并通过 TrapFrame 中寄存器的值传入参数：

```
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("kern/trap.c/trap_dispatch: %e\n", r);
tf->tf_regs.reg_eax = r;
return;
```

第四步，在 `kern/syscall.c` 中实现 `syscall()`。我们注意到，`syscall()` 函数实际上也是一个 dispatcher，也就是说，处理系统调用需要经过两次分派。在这里，根据 `syscallno` 来确定调用哪个函数即可。关于参数问题，我们约定用户调用系统调用时，按照一定的顺序来传入参数，这里，我们也是按顺序将参数传递进去，注意类型强制转换。代码如下：

```
int32_t
syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3,
uint32_t a4, uint32_t a5){
    // LAB 3: Your code here.
    // @huangruizhe 20120411
    int32_t r = 0;
    switch(syscallno){
    case SYS_cputs:
        sys_cputs((const char *)a1, (size_t)a2);
        break;
    case SYS_cgetc:
        r = sys_cgetc();
        break;
    case SYS_getenvid:
        r = sys_getenvid();
        break;
    case SYS_env_destroy:
        r = sys_env_destroy((envid_t)a1);
        break;
    case NSYSCALLS:
    default:
        return -E_INVALID;
    }
    return r;
}
```

要对系统调用有更深刻的理解，我们还应该阅读 `lib/syscall.c`（这里面定义了用户可以当作函数调用的系统调用接口，包括 `syscall()` 函数及其包装），`inc/syscall.h`（这里定义了系统调用号），不再赘述。

### 3.4 User-mode startup

用户程序从 `lib/entry.S` 中的代码开始执行（具体为什么这样执行我没看明白，估计和链接什么的有关系），注意里面定义了 `envs`, `pages`, `vpd`, 和 `vpt` 几个变量，分别指向 `kernel` 中的数据结构提供给用户只读，我们将会在 `lab 4` 中使用到。然后在里面调用 `libmain()` 函数，这是定义在 `lib/libmain.c` 中。在 `libmain()` 函数中我们需要让全局指针 `thisenv` 指向当前进程在 `envs` 数组中对应的 `Env` 结构体，然后调用用户程序的主函数 `umain`。



**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.

需要添加的代码就一行，如下：

```
// LAB 3: Your code here.  
// @huangruizhe 20120411  
thisenv = envs + ENVX(sys_getenvid());
```

### 3.5 Page faults and memory protection

内存保护是操作系统的一个至关重要的特性，要能够保证当一个用户程序出错时，其他用户程序，以及操作系统本身不至于崩溃。

一般来说，操作系统依赖硬件来实现内存的保护机制，不过，操作系统需要告诉硬件，对于当前程序来说哪些虚拟/线性地址是可以访问的，哪些是无效的、越界的。当一个程序试图访问非法地址（地址不存在 or 越权）时，处理器将阻止用户程序继续运行，陷入 **kernel** 并进入相应的处理程序。如果该错误是可以被修复的（也许用户程序具有什么目的，故意出错让 **kernel** 过来处理），**kernel** 将会修复该错误/异常，接着让用户程序重新运行；如果该错误不可修复，那么这真是一个 **bug**，用户程序不能再执行下去。

关于这种机制的一个例子，就是“写时复制技术”。例如，实现用户程序可以自增长的用户栈，刚开始时，**kernel** 只分配给这个栈一个物理页框，用户栈也只映射到一页。当用户程序需要的栈空间越来越大时，它将必然产生一个超过已有范围的地址访问。这时，就产生了一个页面错误 **page fault**。此时 **kernel** 将立刻为用户程序再分配一块物理页并建立好映射，那么用户程序便可以继续执行。从用户程序的角度，它似乎感觉栈空间是超大的，事实上，这时 **kernel** 动态分配的。

在内存保护机制中，“系统调用”存在一个有趣的问题。许多系统调用的接口是让用户传入一个指针，这个指针指向用户的内存空间用于用户的读写。于是 **kernel** 将在处理系统调用时访问这个指针，需要注意下面两点：

① **kernel** 中的 **page fault** 比用户程序中出现的 **page fault** 要严重得多，如果 **kernel** 在运行自己的代码时出现 **page fault**，那应该是一个 **bug**，这时 **page fault handler** 应该 **panic** 表示出错了。但是，如果 **kernel** 中出现的 **page fault** 是访问用户程序传入的用户空间的内存指针，那么需要进行不同的处理，而不是马上 **panic** 那么简单。因此，**kernel** 要能够识别 **page fault** 是谁产生的。

② 一般来说，**kernel** 必然比用户程序内存访问的权限级别更高。有时候用户程序可能会通过系统调用，传入一个 **kernel** 能够读写的地址的指针，但是这个地址用户程序根本无权访问！这时候，**kernel** 就要小心不要被用户程序“欺骗”去访问这类地址。

处于上面这两个原因，**kernel** 在访问用户程序提供的内存指针时，要格外小心，做好判断，分别对待处理。

因此，现在需要我们实现这样的功能，即让 **kernel** 检查一个内存地址是不是在用户空间，检查页目录/页表是否允许对该地址的操作。这样就能够做到不同的 **page fault** 分开对待。



**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.

这个 exercise 第一步，是让我们在 kern/trap.c 中的 page fault handler 中添加代码处理 kernel 中发生的 page fault，也就是 panic 一下。问题的关键是如何确定 page fault 是发生在 kernel 中。其实，我们只要检查一下 tf->tf\_cs 是否等于 GD\_KT 就可以啦。代码如下：

```
// Handle kernel-mode page faults.
// LAB 3: Your code here.
// @huangruizhe 20120412
if(tf->tf_cs == GD_KT){
    panic("kern/trap.c/page_fault_handler: kernel page fault!\n");
}
```

第二步，我们要实现 kern/pmap.c 中的 user\_mem\_check 函数。这一函数的作用是，检查用户程序是否有权访问某一范围的虚拟地址空间。一个用户程序可以访问一个虚拟地址，需要满足两个条件：①该虚拟地址低于 ULIM，②相应页目录项标示了足够的权限。

实现起来，写一个循环调用 pgdir\_walk 获得 va 对应的 pte，比较一下就行啦。不过我实现的时候出现了一些波折。首先，注意到这个 va 是不对齐的，因此，我们要将它对齐。可是，JOS 的检查函数要求保存的出错地址，是第一个不合法的 va。因此，出现这种情况，如果你将 va 向下对齐，那么如果该访问不合法，不是输出对齐后的地址，而是输出 va！因此，最好的方法是，不是将 va 向下对齐，而是将 va+len 向上对齐。代码如下：

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

```

// @huangruizhe 20120412
pte_t * pte;
perm = perm | PTE_P;

uint32_t offset = (uint32_t)va;
uint32_t offset_limit = ROUNDUP((uint32_t)va + len, PGSIZE);
for(; offset < offset_limit; offset += PGSIZE){
    if(offset >= ULIM){
        user_mem_check_addr = (uintptr_t)offset;
        return -E_FAULT;
    }
    pte = pgdir_walk(env->env_pgdir, (void *)offset, 0);
    if(pte == NULL || !(*pte & perm)){
        user_mem_check_addr = (uintptr_t)offset;
        return -E_FAULT;
    }
}
return 0;
}

```

第三步，我们要在 kern/syscall.c 中的系统调用函数中，对用户传入的地址，调用刚刚写的 user\_mem\_check 进行检查。涉及到内存访问的系统调用只有 sys\_cputs()函数一个，添加的代码如下：

```

// LAB 3: Your code here.
// @huangruizhe 20120412
user_mem_assert(curenv, (void *)s, len, PTE_U);

```

第四步，运行 JOS 和 user/buggyhello 用户程序，输出情况如下，和预期相符合：

```

QEMU
SeaBIOS (version pre-0.6.3-20110315_112143-titi)

iPXE v1.0.0-591-g7aee315
iPXE (http://ipxe.org) 00:03.0 C900 PCI2.10 PnP PMM+07FC8D60+07F88D60 C900

Booting from Hard Disk...
6828 decimal is 15254 octal!
Physical memory: 66556K available, base = 640K, extended = 65532K
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
[00000000] new env 00001000
Incoming TRAP frame at 0xefbffffc
Incoming TRAP frame at 0xefbffffc
[00001000] user_mem_check assertion failure for va 00000001
[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> _

```

第五步，修改 kern/kdebug.c 中的 debuginfo\_eip()函数，里面有两处 lab3: your code here，

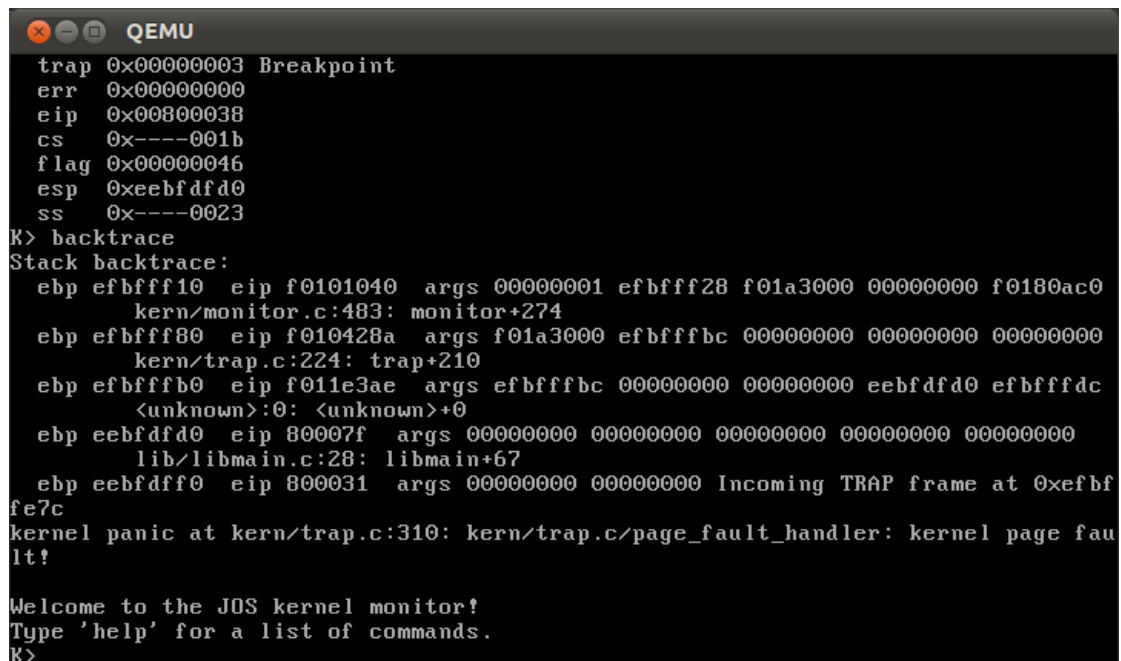
我们需要对 `usr`, `stabs`, and `stabstr` 进行权限检查:

```
// Make sure this memory is valid.
// Return -1 if it is not. Hint: Call user_mem_check.
// LAB 3: Your code here.
// @huangruizhe 20120412
if(user_mem_check(curenv, usr, sizeof(struct UserStabData),
PTE_U) < 0)
    return -1;

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

// Make sure the STABS and string table memory is valid.
// LAB 3: Your code here.
// @huangruizhe 20120412
if(user_mem_check(curenv, stabs, stab_end - stabs, PTE_U) < 0
|| user_mem_check(curenv, stabstr, stabstr_end - stabstr,
PTE_U) < 0)
    return -1;
```

然后, 我们运行一下 `user/breakpoint` 用户程序和 `backtrace`, 输出结果如下:



```
QEMU
trap 0x00000003 Breakpoint
err  0x00000000
eip  0x00800038
cs   0x----001b
flag 0x00000046
esp  0xeebdfd0
ss   0x----0023
K> backtrace
Stack backtrace:
ebp efbfff10 eip f0101040 args 00000001 efbfff28 f01a3000 00000000 f0180ac0
    kern/monitor.c:483: monitor+274
ebp efbfff80 eip f010428a args f01a3000 efbfffb0 00000000 00000000 00000000
    kern/trap.c:224: trap+210
ebp efbfffb0 eip f011e3ae args efbfffb0 00000000 00000000 eebdfd0 efbfffdc
    <unknown>:0: <unknown>+0
ebp eebdfd0 eip 80007f args 00000000 00000000 00000000 00000000 00000000
    lib/libmain.c:28: libmain+67
ebp eebdff0 eip 800031 args 00000000 00000000 Incoming TRAP frame at 0xeebf
fe7c
kernel panic at kern/trap.c:310: kern/trap.c/page_fault_handler: kernel page fau
lt!

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

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

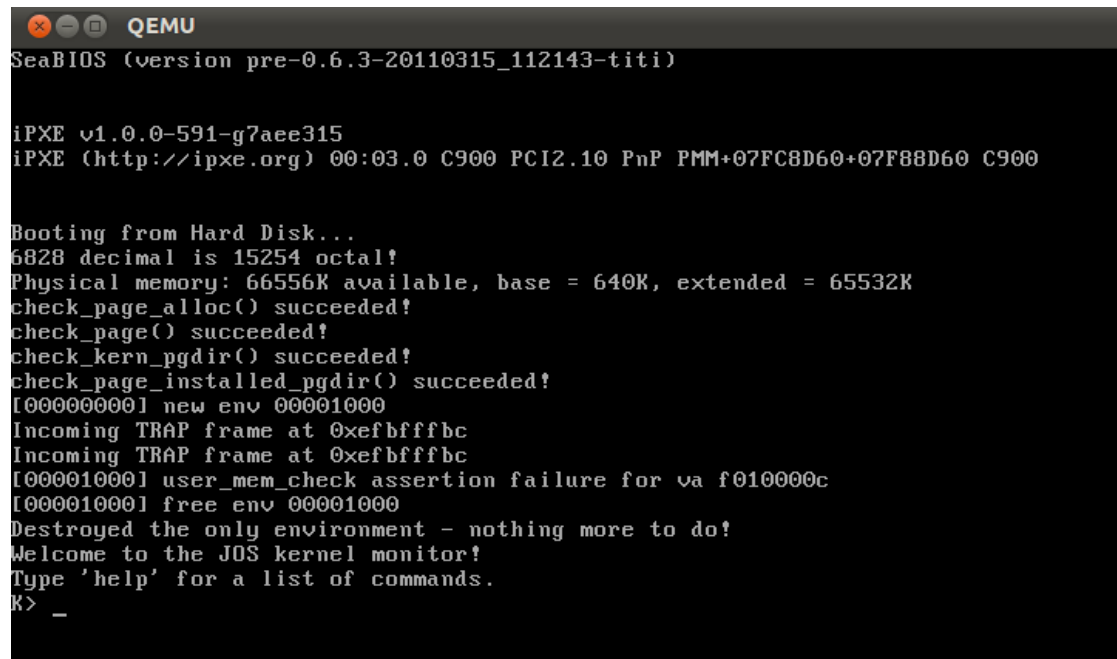
```
[00000000] new env 00001000
[00001000] user_mem_check assertion failure for va f0100020
```

```
[00001000] free env 00001000
```

运行 user/evilhello 用户程序，这个程序萌爆了！如下：

```
void
umain(int argc, char **argv)
{
    // try to print the kernel entry point as a string! mua ha ha!
    sys_cputs((char*)0xf010000c, 100);
}
```

它很得意地调用 sys\_cputs() 函数，妄图将一个 kernel 地址指向的内容输出，当然这是不会得逞的，输出如下：

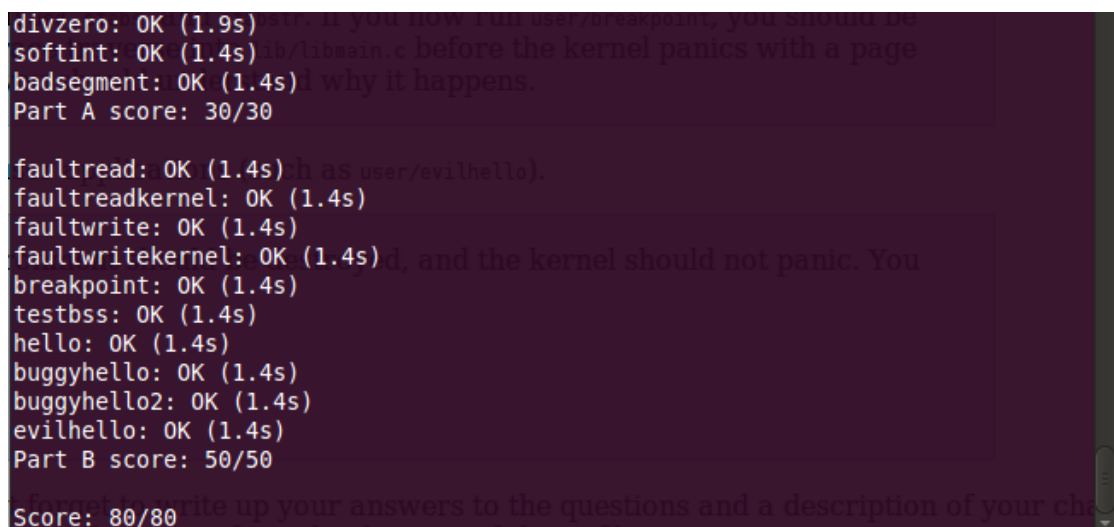


```
QEMU
SeaBIOS (version pre-0.6.3-20110315_112143-titi)

iPXE v1.0.0-591-g7aee315
iPXE (http://ipxe.org) 00:03.0 C900 PCI2.10 PnP PMM+07FC8D60+07F88D60 C900

Booting from Hard Disk...
6828 decimal is 15254 octal!
Physical memory: 66556K available, base = 640K, extended = 65532K
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
[00000000] new env 00001000
Incoming TRAP frame at 0xefbffffbc
Incoming TRAP frame at 0xefbffffbc
[00001000] user_mem_check assertion failure for va f010000c
[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> _
```

到此，Lab 3 全部完成，make grade 纪念：



```
divzero: OK (1.9s)
softint: OK (1.4s)
badsegment: OK (1.4s)
Part A score: 30/30

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

Score: 80/80
```

## 内容三：遇到的困难以及解决方法

### 困难一：这次 LAB 的内容比较多，比较难

主要是理解和学习的问题。User Environment 那一部分还可以，但是我在理解 Exception & Interrupt handling 花了很多功夫，概念又多又杂，手册又臭又长（ms 我的报告也是 T\_T），什么 Protected Control Transfer 啊，TSS 啊，到后来发现这些概念对写代码来说影响不大，尤其是 TSS。我参考了邵老师的讲义和张驰学长的报告，受益颇多。

在我写完报告之后，再回过头来看这些概念，现在感觉清晰了很多。遇到困难，硬着头皮做下去，看看能做到哪里。我想这就是学习的过程吧。

### 困难二：我 SB 了

经常有那种很 SB 的错，这种错不包含任何技术含量。调半天，发现是粗心。

另外一种错，就没有那么 SB 了，我觉得是和“写代码的经验和习惯”有关，这是一种能力。比如说，有时别人一写下来，就对了；而我写下来，咋一看和别人的功能是一样的，但是因为具体实现有差异，而这种差异就会导致错误，例如我在 exercise 9 说的那个 bug。

这个困难，需要经常阅读大牛同学们的良好风格和高效的代码解决。

### 困难三：报告怎么写了那么久 T.T

和上一次一样。

## 内容四：收获和感想

主要的收获是，①深入理解了用户进程（环境）的概念，及其上下文切换过程；②理解了 x86 体系结构以及操作系统处理中断的流程！回过头来看，没什么神秘的，就是控制你的 eip 指向哪里而已，当然在这个过程中会涉及很多东西，什么保护现场啊，权限啊等等。

话说回来，eip 真是一个很重要的概念，eip 指向哪里，处理器就会从哪里执行下去。这让我想起了程序的本质就是一系列指令。

感觉 lab3 比 lab2 难。

这次心得体会不想写那么多，因为我累了。

最后，我要感谢刘驰同学，他耐心地指导了我很多不明白的地方。同时通过讨论，他纠正了我很多我本来理解错题目的地方，让我少走弯路。他写作业写得好快啊，一直是我追赶的目标和努力的方向。

## 内容五：对课程的意见和建议

助教很 nice，本次暂无。

## 内容六：参考文献

- [1] MIT 6.828 课程资料, MIT, 2011;
- [2] 邵志远, JOS 实验讲义第五章, 华中科技大学, 2010;
- [3] 王仲禹, JOS Lab3 实验报告, 北京大学, 2011;
- [4] 张驰, 操作系统 JOS 第三次实习报告, 北京大学, 2011;
- [5] 卓达城, JOS 实验室 3 实验记录,  
<http://wenku.baidu.com/view/3b71cf232f60ddccda38a024.html?from=rec&pos=1&weight=15&astweight=11&count=5>
- [6] 其他:  
<http://wenku.baidu.com/view/358425bcc77da26925c5b042.html>  
[http://hi.baidu.com/chengyongyuan\\_/blog/item/e121e6a503054f9ed143582e.html](http://hi.baidu.com/chengyongyuan_/blog/item/e121e6a503054f9ed143582e.html)  
[http://hi.baidu.com/chengyongyuan\\_/blog/item/57176cfe789e44f6fc037f00.html](http://hi.baidu.com/chengyongyuan_/blog/item/57176cfe789e44f6fc037f00.html)  
[http://hi.baidu.com/chengyongyuan\\_/blog/item/6c8e021c0da67814413417ed.html](http://hi.baidu.com/chengyongyuan_/blog/item/6c8e021c0da67814413417ed.html)  
<http://www.cnblogs.com/hydd/articles/1389487.html>  
<http://www.doc88.com/p-01844588389.html>