

# 操作系统JOS实习第三次报告

张弛 00848231,  
zhangchitc@gmail.com

April 12, 2011

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>User Environments and Exception Handling</b>	<b>2</b>
2.1	Environment State . . . . .	2
2.2	Allocating the Environments Array . . . . .	2
2.3	Creating and Running Environments . . . . .	3
2.4	Handling Interrupts and Exceptions . . . . .	16
2.5	Basics of Protected Control Transfer . . . . .	19
2.6	Types of Exceptions and Interrupts . . . . .	19
2.7	An Example . . . . .	19
2.8	Nested Exceptions and Interrupts . . . . .	19
2.9	Setting Up the IDT . . . . .	19
<b>3</b>	<b>Page Faults, Breakpoints Exceptions, and System Calls</b>	<b>26</b>
3.1	Handling Page Faults . . . . .	26
3.2	The Breakpoint Exception . . . . .	26
3.3	System calls . . . . .	26
3.4	User-mode startup . . . . .	26
3.5	Page faults and memory protection . . . . .	26

## 1 Introduction

我在实验中主要参考了华中科技大学邵志远老师写的JOS实习指导，在邵老师的主页上<http://grid.hust.edu.cn/zyshao/OSEngineering.htm>可以找到。但是这次实验的指导远远不如lab1的指导详尽，所以我这里需要补充的内容会很多。

内联汇编请参考邵老师的第二章讲义，对于语法讲解的很详细。

## 2 User Environments and Exception Handling

### 2.1 Environment State

MIT的材料里对于struct Env的讲解很详细。其中唯一需要注意的就是struct Trapframe的理解，在这里还无法展开叙述。这个我们在后面具体编程的时候会提到。

### 2.2 Allocating the Environments Array

**Exercise 1.** Modify `i386_vm_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_boot_pgdir()` succeeds.

这个练习比较简单，有了前面设置pages数组的经验，对于envs的理解就很顺畅了。我们来看看具体的代码：

```
kern/pmap.c: i386_vm_init ()
1  pages = boot_alloc (npage * sizeof (struct Page), PGSIZE);
2
3  //////////////////////////////////////
4  // Make 'envs' point to an array of size 'NENV' of 'struct Env'.
5  // LAB 3: Your code here.
6
7  envs = boot_alloc (NENV * sizeof (struct Env), PGSIZE);
```

分配了物理空间以后，再在虚拟地址空间为其创建映射：

```
kern/pmap.c: i386_vm_init ()
1  //////////////////////////////////////
2  // Map the 'envs' array read-only by the user at linear address UENVS
3  // (ie. perm = PTE_U | PTE_P).
4  // Permissions:
```

```

5      // - the new image at UENVNS -- kernel R, user R
6      // - envs itself -- kernel RW, user NONE
7      // LAB 3: Your code here.
8      //
9      boot_map_segment (
10         pgdir,
11         UENVNS,
12         ROUNDUP (NENV * sizeof (struct Env), PGSIZE),
13         PADDR ((uintptr_t) envs),
14         PTE_U);

```

## 2.3 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.
env_setup_vm():
    allocate a page directory for a new environment and initialize the
    kernel portion of the new environment's address space.
segment_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".

我们一个一个函数的来看把，首先是`env_init()`:

```

kern/env.c: env_init ()
1 void
2 env_init(void)
3 {
4     int i;
5
6     LIST_INIT(&env_free_list);
7     for (i = NENV - 1; i >= 0; i--) {
8         envs[i].env_id = 0;
9         envs[i].env_status = ENV_FREE;

```

```

10 LIST_INSERT_HEAD(&env_free_list, &envs[i], env_link);
11     }
12 }

```

没有什么好说的，类比pages对应的page\_init()写就行了。接下来看 env\_setup\_vm()

```

                                kern/env.c: env_setup_vm ()
1 static int
2 env_setup_vm(struct Env *e)
3 {
4     int i, r;
5     struct Page *p = NULL;
6
7     // Allocate a page for the page directory
8     if ((r = page_alloc(&p)) < 0)
9         return r;
10
11     e->env_pgdir = page2kva (p);
12     e->env_cr3 = page2pa (p);
13
14     memmove (e->env_pgdir, boot_pgdir, PGSIZE);
15     memset (e->env_pgdir, 0, PDX(UTOP) * sizeof (pde_t));
16
17     p->pp_ref ++;

```

这里主要注意的是第14和15行代码。因为在UTOP之上的所有映射对于任何一个地址空间都是一样的（无论是对于内核地址空间还是对于任意一个用户地址空间而言），他们都和lab2中对于内核地址空间设置的静态映射一样（静态映射就是没有实际分配物理页，即映射是通过boot\_map\_segment()而非page\_insert()），所以这里我们能直接拷贝系统页目录boot.pgdir中的内容。

接下来看看函数segment\_alloc()

```

                                kern/env.c: segment_alloc ()
1 static void
2 segment_alloc(struct Env *e, void *va, size_t len)
3 {
4     va = ROUNDDOWN (va, PGSIZE);
5     len = ROUNDUP (len, PGSIZE);
6
7     struct Page *pp;
8     int r;
9
10    for (; len > 0; len -= PGSIZE, va += PGSIZE) {
11        r = page_alloc (&pp);
12
13        if (r != 0)
14            panic ("segment_alloc: _physical_page_allocation_failed_", r);
15
16        r = page_insert (e->env_pgdir, pp, va, PTE_U|PTE_W);
17
18        if (r != 0)
19            panic ("segment_alloc: _page_mapping_failed_", r);
20    }
21 }

```

这个函数的作用是在e代表的用户虚拟地址空间中从va开始的地址分配出len长度的区域，准备写入数据。

有点类似lab2中的boot\_map\_segment()，但是他们是不一样的。boot\_map\_segment()的操作空间是内核虚拟地址空间boot\_pgdir。它提供的映射是静态映射，不涉及物理页的分配。而segment\_alloc()则是要对实际的物理页面分配映射到当前用户的虚拟地址空间中。

弄清楚这两种映射机制的区别，上面的代码就很好理解了，看下一个函数load\_icode()

```

                                kern/env.c: load_icode ()
1 static void
2 load_icode(struct Env *e, uint8_t *binary, size_t size)
3 {
4     struct Elf *ELFHDR = (struct Elf*) binary;
5     struct Proghdr *ph, *eph;
6
7     // is this a valid ELF?
8     if (ELFHDR->e_magic != ELF_MAGIC)
9         panic ("load_icode: _Not_a_valid_ELF");
10
11     ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);
12     eph = ph + ELFHDR->e_phnum;
13
14     lcr3 (e->env_cr3);
15     for (; ph < eph; ph++) {
16         if (ph->p_type == ELF_PROG_LOAD) {
17             segment_alloc (e, (void*) ph->p_va, ph->p_memsz);
18             memset ((void *)ph->p_va, 0, ph->p_memsz);
19             memmove ((void *)ph->p_va, binary + ph->p_offset, ph->p_filesz);
20         }
21     }
22     lcr3 (boot_cr3);
23
24     e->env_tf.tf_eip = ELFHDR->e_entry;
25
26     segment_alloc (e, (void*) (USTACKTOP - PGSIZE), PGSIZE);
27 }

```

因为MIT的说明里提到过，因为JOS到现在为止还没有文件系统，所以为了测试我们能运行用户程序，现在的做法是将用户程序编译以后**和内核链接到一起**（即用户程序紧接着内核后面放置）。所以这个函数的作用就是将嵌入在内核中的用户程序取出释放到相应链接器指定好的用户虚拟空间里。这里的binary指针，就是用户程序在内核中的开始位置的虚拟地址。

按照注释的提示，我们可以参照boot/main.c来完成相应的载入，但是有几个地方需要注意

1. 对于用户程序ELF文件的每个程序头ph，ph->p\_memsz和ph->p\_filesz是两个概念，前者是该程序头应在**内存中占用的空间大小**，而后者是实际该程序头**占用的文件大小**。他们俩的区别就是ELF文件中BSS节中那些没有被初始化的静态变量，这些变量不会被分配文件储存空间，但是在实际载入后，需要在内存中给与相应的空间，并且全部初始化为0。所以具体来讲，就是每个程序段ph，总共占用p\_memsz的内存，前面p\_filesz的空间从binary的对应内存复制过来，后面剩下的空间全部清0

2. `ph→p_va`是该程序段应该被放入的虚拟空间地址，但是注意，在这个时候，虚拟地址空间是**用户环境Env的虚拟地址空间**。可是，在进入`load_icode()`时，是内核态进入的，所以虚拟地址空间还是内核的空间。我们要如何对用户的虚拟空间进行操作呢？看到第15行：

```
kern/env.c: load_icode ()
15  lcr3 (e->env_cr3);
```

这个语句在我们进入每个程序头进行具体设置时，将页表切换到用户虚拟地址空间。这样我们就可以方便的在后面使用`memset`和`memmove`等函数对一个虚拟地址进行相应的操作了。其中`e→env_cr3`的值是在前面的`env_setup_vm()`设置好的。

但是仍要小心的是，对于ELF载入完毕以后，我们就不需要对用户空间进行操作了，所以记得在22行重新切回到内核虚拟地址空间来。

3. 注释中还提到了要对程序的入口地址作一定的设置，这里对应的操作是

```
kern/env.c: load_icode ()
24  e->env_tf.tf_eip = ELFHDR->e_entry;
```

这里涉及到对`struct Trapframe`结构的具体介绍，我们留到下一个函数说明`env_create()`的时候进行详细介绍。

继续看下个函数`env_create()`

```
kern/env.c: env_create()
1 void
2 env_create(uint8_t *binary, size_t size)
3 {
4     struct Env *e;
5     int r;
6
7     r = env_alloc (&e, 0);
8
9     if (r < 0)
10         panic ("env_create:_%e", r);
11
12     load_icode (e, binary, size);
13 }
```

到这个函数为止，系统就为一个用户程序的运行做好了一切准备，在这个函数中，接受内核传入的用户程序的所在地址`binary`（内核地址），然后为其创建用户进程空间，并且将其载入到相应的虚拟地址上。接下来的`env_run()`就可以开始真正的运行一个程序了。

这里调用了过程`env_alloc()`来为用户进程分配一个`struct Env`，这个过程是JOS替我们写好的，但是还是有必要好好看看，便于我们对`struct Env`和`struct Trapframe`的理解。

MIT的资料中详细介绍了`struct Env`的结构：

inc/env.h

```

1 struct Env {
2     struct Trapframe env_tf;           // Saved registers
3     LIST_ENTRY(Env) env_link;          // Free list link pointers
4     env_id_t env_id;                   // Unique environment identifier
5     env_id_t env_parent_id;            // env_id of this env's parent
6     unsigned env_status;               // Status of the environment
7     uint32_t env_runs;                 // Number of times environment has run
8
9     // Address space
10    pde_t *env_pgdir;                  // Kernel virtual address of page dir
11    physaddr_t env_cr3;                 // Physical address of page dir
12 };

```

其中env\_tf的说明是保存了用户进程被切换出来时CPU的状态信息。我们去inc/trap.h中找寻其具体定义：

inc/trap.h

```

1 struct PushRegs {
2     /* registers as pushed by pusha */
3     uint32_t reg_edi;
4     uint32_t reg_esi;
5     uint32_t reg_ebp;
6     uint32_t reg_esp;                /* Useless */
7     uint32_t reg_ebx;
8     uint32_t reg_edx;
9     uint32_t reg_ecx;
10    uint32_t reg_eax;
11 } __attribute__((packed));
12
13 struct Trapframe {
14     struct PushRegs tf_regs;
15     uint16_t tf_es;
16     uint16_t tf_padding1;
17     uint16_t tf_ds;
18     uint16_t tf_padding2;
19     uint32_t tf_trapno;
20     /* below here defined by x86 hardware */
21     uint32_t tf_err;
22     uintptr_t tf_eip;
23     uint16_t tf_cs;
24     uint16_t tf_padding3;
25     uint32_t tf_eflags;
26     /* below here only when crossing rings, such as from user to kernel */
27     uintptr_t tf_esp;
28     uint16_t tf_ss;
29     uint16_t tf_padding4;
30 } __attribute__((packed));

```

其他的都很好理解，某些padding开头的变量是为了让数据补齐4Byte。

我们看到，Trapframe保存的都是一些系统关键的寄存器。这里我们只需要特别关注4个寄存器，涉及到程序执行的控制流问题：

- EFLAGS：状态寄存器，这个我们暂时用不到
- EIP：Instruction Pointer，当前执行的汇编指令的地址
- ESP：当前的栈顶地址

- EBP: 辅助用, 当前过程的帧在栈中的开始地址 (高地址) 即EBP到EIP中就是此过程的帧

其中EBP由程序自行操作, 而其他三者都会被在执行汇编指令时被改变。ESP就不说了, push和pop指令都是以ESP指针为操作目标的。至于EIP, 在lab1中的运行栈那一节, 我们看到了C程序编译后压栈的具体信息, 其中就可以看到EIP。现在我们可以来看看在程序调用call时具体是如何修改EIP的。通过查询**IA-32 Intel Architecture Software Developer's Manuals** 中的**Volume 2A: Instruction Set Reference, A-M** 中的CALL指令, 我们可以看到其详细的执行流程:

---

**Algorithm 1: CALL - Call Procedure**

---

```
begin
  tempEIP  $\leftarrow$  EIP + DEST;
  Push(EIP);
  EIP  $\leftarrow$  tempEIP;
end
```

---

这个是我个人简化后最关键的部分, 实际上指令的流程涉及到32位、64位、访问权限、以及长跳转和短跳转的各种问题, 不过那不是我们关心的。我们只需要知道**它对EIP和ESP做了什么**就好了。

从上面简单的三条语句我们可以知道, 在进入新的过程体之前, 老的EIP就被系统压入了堆栈以便后面返回时使用, 然后将新的执行地址放入了EIP。CALL执行完以后, ESP和EIP都发生了改变。

同样的有调用就有返回, 我们去看看RET指令的详细手册:

---

**Algorithm 2: RET - Return from Procedure**

---

```
begin
  EIP  $\leftarrow$  Pop();
end
```

---

看着很简单, 如果涉及保护模式和实模式的切换, 那么还有相应段寄存器CS的保存切换问题, 在IRET中我们就可以看到相应的逻辑, 现在我们先可以不管。

好了, 看完Trapframe的内部结构和关键寄存器以后, 我们回到Trapframe的讨论。既然这里保存了程序执行所需要的状态, 那么刚才在load\_icode() 中是如何设置的呢, 在env\_create() 调用load\_icode()之前分配用户环境env\_alloc()。我们进这里看看:



```

kern/env.c: env_alloc()

1 int
2 env_alloc(struct Env **newenv_store, env_id_t parent_id)
3 {
4     int32_t generation;
5     int r;
6     struct Env *e;
7
8     if (!(e = LIST_FIRST(&env_free_list)))
9         return -E_NO_FREE_ENV;
10
11     // Allocate and set up the page directory for this environment.
12     if ((r = env_setup_vm(e)) < 0)
13         return r;
14
15     // Generate an env_id for this environment.
16     generation = (e->env_id + (1 << ENVGENSHIFT)) & ~(NENV - 1);
17     if (generation <= 0) // Don't create a negative env_id.
18         generation = 1 << ENVGENSHIFT;
19     e->env_id = generation | (e - envs);
20
21     // Set the basic status variables.
22     e->env_parent_id = parent_id;
23     e->env_status = ENV_RUNNABLE;
24     e->env_runs = 0;
25
26     // Clear out all the saved register state,
27     // to prevent the register values
28     // of a prior environment inhabiting this Env structure
29     // from "leaking" into our new environment.
30     memset(&e->env_tf, 0, sizeof(e->env_tf));
31
32     // Set up appropriate initial values for the segment registers.
33     // GD_UD is the user data segment selector in the GDT, and
34     // GD_UT is the user text segment selector (see inc/memlayout.h).
35     // The low 2 bits of each segment register contains the
36     // Requestor Privilege Level (RPL); 3 means user mode.
37     e->env_tf.tf_ds = GD_UD | 3;
38     e->env_tf.tf_es = GD_UD | 3;
39     e->env_tf.tf_ss = GD_UD | 3;
40     e->env_tf.tf_esp = USTACKTOP;
41     e->env_tf.tf_cs = GD_UT | 3;
42     // You will set e->env_tf.tf_eip later.
43
44     // commit the allocation
45     LIST_REMOVE(e, env_link);
46     *newenv_store = e;
47
48     cprintf("[%08x]_new_env_%08x\n", curenv ? curenv->env_id : 0, e->env_id);
49     return 0;
50 }

```

我们只需要关注从26行开始以后的内容，这里开始对`e->env_tf`进行设置。有几个关键点：

- `tf_esp`: 初始化为`USTACKTOP`，表示当前用户栈为空
- `tf_cs`: 初始化为user text segment selector，权限为用户可访问
- `tf_eip`: 这里没有设置，但是注释告诉了我们该由我们设置，很显然，这里`eip`的值就是我们在`load_icode()`里应该设置的用户程序入口地址

这样梳理一遍以后，我们就可以对load\_icode() 里那行设置入口地址代码完全理解了。

看到最后一个要完成的过程env\_run()

```
kern/env.c: env_run()

1 void
2 env_run(struct Env *e)
3 {
4     if (curenv != e) {
5         curenv = e;
6         curenv->env_runs ++;
7         lcr3 (curenv->env_cr3);
8     }
9
10    env_pop_tf (&curenv->env_tf);
11
12    panic("env_run_not_yet_implemented");
13 }
```

这里的一个问题就是处理重复切换到当前用户环境的判断，只有是切换到一个新的用户环境时，才需要启用新的用户页面。这个过程里最主要的任务是理解env\_pop\_tf()，这个过程是真正负责切换到用户程序的过程：

```
kern/env.c: env_pop_tf()

1 void
2 env_pop_tf(struct Trapframe *tf)
3 {
4     __asm __volatile("movl_%0,%%esp\n"
5                      "\tpopal\n"
6                      "\tpopl_%%es\n"
7                      "\tpopl_%%ds\n"
8                      "\taddl_$0x8,%%esp\n" /* skip tf_trapno and tf_errcode */
9                      "\tiret"
10                      : : "g" (tf) : "memory");
11    panic("iret_failed"); /* mostly to placate the compiler */
12 }
```

我们来尝试理解这段内联汇编：

```
4     movl %0,%%esp
```

这里出现了占位符%0，通过后面的参数可以看到这里的占位符代表的是memory中的变量tf，即Trapframe的指针地址。这里把它传给esp是什么意思？看到后面的各种pop命令，就可以知道，这里的想法是把Trapframe看作一个存储了很多内容的栈，然后利用pop命令一个一个输出到我们想要重置的寄存器里。因为我们知道弹栈的时候栈指针是不断加的过程（栈的生长是栈指针不断减），所以将ESP设置为Trapframe所在内存的首地址，就可以以内存中的排布顺序释放出所有的内容了。非常的巧妙！

```
5     popal
```

通过查询手册，可以得到popal的执行明细：

**Algorithm 3: POPA - Pop All General Purpose Registers**

```

begin
    EDI ← Pop();
    ESI ← Pop();
    EBP ← Pop();
    ESP ← ESP + 4 ;(* Skip next 4 bytes of stack *)
    EBX ← Pop();
    EDX ← Pop();
    ECX ← Pop();
    EAX ← Pop();
end

```

第一句就输出了这么多寄存器，这里每一次Pop ()，就是从ESP指向的Trapframe里拿出4个Byte，我们来看看Trapframe的前8个DWORD是什么：

```

inc/trap.h

1 struct PushRegs {
2     /* registers as pushed by pusha */
3     uint32_t reg_edi;
4     uint32_t reg_esi;
5     uint32_t reg_ebp;
6     uint32_t reg_oesp;           /* Useless */
7     uint32_t reg_ebx;
8     uint32_t reg_edx;
9     uint32_t reg_ecx;
10    uint32_t reg_eax;
11 } __attribute__((packed));
12
13 struct Trapframe {
14     struct PushRegs tf_regs;
15     uint16_t tf_es;
16     uint16_t tf_padding1;
17     uint16_t tf_ds;
18     uint16_t tf_padding2;
19     uint32_t tf_trapno;
20     /* below here defined by x86 hardware */
21     uint32_t tf_err;
22     uintptr_t tf_eip;
23     uint16_t tf_cs;
24     uint16_t tf_padding3;
25     uint32_t tf_eflags;
26     /* below here only when crossing rings, such as from user to kernel */
27     uintptr_t tf_esp;
28     uint16_t tf_ss;
29     uint16_t tf_padding4;
30 } __attribute__((packed));

```

可以看到前8个DWORD为一个struct PushRegs，**这里面的定义顺序和popal里设置的顺序是完全对应的！**可见PushRegs的定义也是经过了缜密的思考的，非常的巧妙，利用一句汇编指令就完成了这么多寄存器的设置。

后面几句汇编代码就很好理解了，直到这句：

```

5     iret

```

再次求助INTEL的指令手册，可以看到IRET和RET的不同：

---

**Algorithm 4: IRET - Interrupt Return**


---

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

---

因为IRET涉及到中断返回的各种控制，所以在保护模式以及实模式切换中会涉及段寄存器切换以及访问控制的问题，实际的控制流非常非常非常复杂，有兴趣的同学可以参考手册里的详细说明。

这个时候执行的IRET语句，会把Trapframe里的下面三个成员放入相应的寄存器

```
uintptr_t tf_eip;
uint16_t tf_cs;
uint16_t tf_padding3;
uint32_t tf_eflags;
```

这些成员我们在env\_alloc() 以及load\_icode()中都设置好了，其中EIP为用户程序入口地址，CS为用户程序代码段段基址。

那么执行完这条语句以后，CPU再往下执行的第一条语句，应该就是用户程序的第一条指令了。

所以说env\_run()和env\_pop\_tf()都是没有返回的。

到这里，我们的Exercise 2就算做完了，但是编译启动JOS发现它给出了Triple fault的错误信息。在MIT的课程材料上解释了这样的原因。是因为我们没有对中断表进行相应的设置，以至于用户程序在调用系统终端输出字符时产生了错误。但是我们需要认为的确认一下是否真的错误是由中断而不是其他设置造成的，所以我们启动GDB调试，选择在env\_pop\_tf()函数停下：

```
The target architecture is assumed to be i8086
[f000:fff0] 0xffff0: ljmp $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file obj/kern/kernel
(gdb) b env_pop_tf
Breakpoint 1 at 0xf0103128: file kern/env.c, line 523.
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0xf0103128 <env_pop_tf>: push %ebp

Breakpoint 1, env_pop_tf (tf=0xf01af000) at kern/env.c:523
523 {
(gdb)
```

从这里开始单步跟踪，在IRET指令之前停下来，我们在这里查看寄存器的信息看是否都被设置好了：

```
(gdb) next
=> 0xf010312e <env_pop_tf+6>: mov     0x8(%ebp),%esp
525      __asm __volatile("movl_%0,%esp\n"
(gdb) si
=> 0xf0103131 <env_pop_tf+9>: popa
0xf0103131      525      __asm __volatile("movl_%0,%esp\n"
(gdb) si
=> 0xf0103132 <env_pop_tf+10>: pop     %es
0xf0103132 in env_pop_tf (tf=???) at kern/env.c:525
525      __asm __volatile("movl_%0,%esp\n"
(gdb) si
=> 0xf0103133 <env_pop_tf+11>: pop     %ds
0xf0103133      525      __asm __volatile("movl_%0,%esp\n"
(gdb) si
=> 0xf0103134 <env_pop_tf+12>: add     $0x8,%esp
0xf0103134      525      __asm __volatile("movl_%0,%esp\n"
(gdb) si
=> 0xf0103137 <env_pop_tf+15>: iret
0xf0103137      525      __asm __volatile("movl_%0,%esp\n"
(gdb) info registers
eax            0x0          0
ecx            0x0          0
edx            0x0          0
ebx            0x0          0
esp            0xf01af030    0xf01af030
ebp            0x0          0x0
esi            0x0          0
edi            0x0          0
eip            0xf0103137    0xf0103137 <env_pop_tf+15>
eflags         0x96          [ PF AF SF ]
cs             0x8          8
ss             0x10         16
ds             0x23         35
es             0x23         35
fs             0x23         35
gs             0x23         35
(gdb)
```

从EAX、ECX等寄存器中看到都被清0了，这个和我们在env\_alloc()中看到的设置是一致的，但是在IRET执行之前CS和EIP两个寄存器都还看不到，不过没有关系，我们知道栈顶的接下来三个DWORD分别为EIP、CS和EFLAGS，我们查看一下栈顶的这三个DWORD：

```
(gdb) x/3x 0xf01af030
0xf01af030:    0x00800020    0x0000001b    0x00000000
(gdb)
```

可以看到EIP的值为0x00800020即用户程序的入口地址，我们可以打开user/user.ld文件查看一下：

```
user/user.ld

4 OUTPUT_FORMAT("elf32-i386", "elf32-i386", "elf32-i386")
5 OUTPUT_ARCH(i386)
6 ENTRY(_start)
7
8 SECTIONS
9 {
10     /* Load programs at this address: "." means the current address */
11     . = 0x800020;
```

```

12     .text : {
13         *(.text .stub .text.* .gnu.linkonce.t.*)
14     }
15

```

可以看到第11行链接器对于程序入口地址的设置，和我们看到的调试结果是符合的。这就说明我们正确的将入口地址加载进来了，接下来我们看看是否正确载入了用户程序的ELF文件：

```

(gdb) si
=> 0x800020:    cmp     $0xeebfe000,%esp
0x00800020 in ?? ()
(gdb) x/6i 0x800020
=> 0x800020:    cmp     $0xeebfe000,%esp
0x800026:    jne     0x80002c
0x800028:    push   $0x0
0x80002a:    push   $0x0
0x80002c:    call   0x800060
0x800031:    jmp     0x800031
(gdb)

```

实际的用户程序hello的汇编代码可以在obj/user/hello.asm中找到：

```

                                obj/user/hello.asm
5  Disassembly of section .text:
6
7  00800020 <_start>:
8  // starts us running when we are initially loaded into a new environment.
9  .text
10 .globl _start
11 _start:
12     // See if we were started with arguments on the stack
13     cmpl $USTACKTOP, %esp
14     800020:    81 fc 00 e0 bf ee    cmp     $0xeebfe000,%esp
15     jne args_exist
16     800026:    75 04              jne     80002c <args_exist>
17
18     // If not, push dummy argc/argv arguments.
19     // This happens when we are loaded by the kernel,
20     // because the kernel does not know about passing arguments.
21     pushl $0
22     800028:    6a 00              push    $0x0
23     pushl $0
24     80002a:    6a 00              push    $0x0
25
26 0080002c <args_exist>:
27
28 args_exist:
29     call libmain
30     80002c:    e8 2f 00 00 00    call   800060 <libmain>
31 1:    jmp 1b
32     800031:    eb fe              jmp     800031 <args_exist+0x5>
33     ...

```

可以看到和输出是一致的，从这里可以知道我们的loadicode()的载入是正常工作的。

我们找到MIT教材中提到的sys\_cputs()函数中的中断指令在用户程序中的位置：

```

obj/user/hello.asm

2074 void
2075 sys_cputs(const char *s, size_t len)
2076 {
2077     800d3c:    55                push    %ebp
2078     800d3d:    89 e5            mov     %esp,%ebp
2079     800d3f:    83 ec 0c         sub     $0xc,%esp
2080     800d42:    89 1c 24         mov     %ebx, (%esp)
2081     800d45:    89 74 24 04      mov     %esi, 0x4(%esp)
2082     800d49:    89 7c 24 08      mov     %edi, 0x8(%esp)
2083     //
2084     // The last clause tells the assembler that this can
2085     // potentially change the condition codes and arbitrary
2086     // memory locations.
2087     asm volatile("int %1\n"
2088     800d4d:    b8 00 00 00 00   mov     $0x0,%eax
2089     800d52:    8b 4d 0c         mov     0xc(%ebp),%ecx
2090     800d55:    8b 55 08         mov     0x8(%ebp),%edx
2091     800d58:    89 c3            mov     %eax,%ebx
2092     800d5a:    89 c7            mov     %eax,%edi
2093     800d5c:    89 c6            mov     %eax,%esi
2094     800d5e:    cd 30           int     $0x30
2095
2096 void
2097 sys_cputs(const char *s, size_t len)
2098 {
2099     syscall(SYS_cputs, 0, (uint32_t)s, len, 0, 0, 0);
2100 }
2101
2102     800d60:    8b 1c 24         mov     (%esp),%ebx
2103     800d63:    8b 74 24 04      mov     0x4(%esp),%esi
2104     800d67:    8b 7c 24 08      mov     0x8(%esp),%edi
2105     800d6b:    89 ec            mov     %ebp,%esp
2106     800d6d:    5d              pop     %ebp
2107     800d6e:    c3              ret

```

可以看到中断调用的地址为0x800d5e，我们尝试着在这里设下断点，看JOS能否运行到这里：

```

(gdb) b *0x800d5e
Breakpoint 2 at 0x800d5e
(gdb) c
Continuing.
=> 0x800d5e:    int     $0x30

Breakpoint 2, 0x00800d5e in ?? ()
(gdb) si
=> 0x800d5e:    int     $0x30

Breakpoint 2, 0x00800d5e in ?? ()
(gdb)

```

可以看到JOS成功运行到了该断点，再执行一条指令，EIP没有发生变化，这个时候看QEMU的输出信息，发现已经产生Triple fault：

```

zhangchi@zhangchi-vostro1400:~/lab$ make qemu-gdb
sed "s/localhost:1234/localhost:26000/" < .gdbinit.tmpl > .gdbinit
***
*** Now run 'gdb'.
***
qemu -hda obj/kern/kernel.img -serial mon:stdio -S -gdb tcp::26000
6828 decimal is 15254 octal!
Hooray! Passed all test cases for stdlib!!
Physical memory: 66556K available, base = 640K, extended = 65532K

```

```

check_page_alloc() succeeded!
page_check() succeeded!
check_boot_pgdir() succeeded!
[00000000] new env 00001000
EAX=00000000 EBX=00000000 ECX=0000000d EDX=eebfde88
ESI=00000000 EDI=00000000 EBP=eebfde60 ESP=eebfde54
EIP=00800d5e EFL=00000092 [--S-A--] CPL=3 II=0 A20=1 SMM=0 HLT=0
ES =0023 00000000 ffffffff 00cff300 DPL=3 DS [-WA]
CS =001b 00000000 ffffffff 00cffa00 DPL=3 CS32 [-R-]
SS =0023 00000000 ffffffff 00cff300 DPL=3 DS [-WA]
DS =0023 00000000 ffffffff 00cff300 DPL=3 DS [-WA]
FS =0023 00000000 ffffffff 00cff300 DPL=3 DS [-WA]
GS =0023 00000000 ffffffff 00cff300 DPL=3 DS [-WA]
LDT=0000 00000000 00000000 00008200 DPL=0 LDT
TR =0028 f017c8e0 00000068 f0408917 DPL=0 TSS32-avl
GDT=      f011a320 0000002f
IDT=      f017c0e0 000007ff
CR0=80050033 CR2=00000000 CR3=0005c000 CR4=00000000
DR0=00000000 DR1=00000000 DR2=00000000 DR3=00000000
DR6=ffff0ff0 DR7=00000400
Triple fault. Halting for inspection via QEMU monitor.

```

所以到目前为止，我们写出的JOS的运行一切正常。

## 2.4 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.

一定要读！尤其是**Chapter 9, Exceptions and Interrupts**，因为IA-32的开发手册实在是太长没法看，但是前面这个HTML的说明不长不短，刚好能大致完整的介绍一下中断的机制。

在**9.5 IDT Descriptors**中提到了IDT中一共有三种类型的门描述符

- Task gates
- Interrupt gates
- Trap gates

后来我在查找资料时在IA32-3A.pdf里看到了x86中关于门描述符的综合介绍（详见**4.8.2 Gate Descriptors**），发现还有一个Call gates，然后我就晕了，不知道这4种门都是干什么的，然后为什么IDT只有其中三种。

尽管在**9.6 Interrupt Tasks and Interrupt Procedures**中稍微解释了一下，但是我觉得它没有讲清楚**为什么要区别开来这四种门，它们各自的不同分别对应什么样的应用场景等等**。我找到了一个中文版的说明：[http://www.mouseos.com/arch/gate\\_descriptor.html](http://www.mouseos.com/arch/gate_descriptor.html)，比前面那个网页版要详细一点，但是仍没有说到点子上。后来我终于找到StackOverflow上一个相关的问题，我觉得讲的很好：<http://stackoverflow.com/questions/>



[3425085/the-difference-between-call-gate-interrupt-gate-trap-gate](#)。这里我就按照我的理解把上面的资料总结一下：

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**。

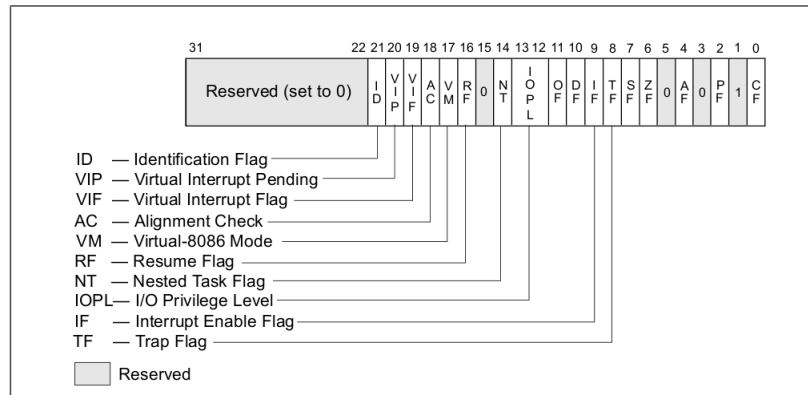


Figure 2-4. System Flags in the EFLAGS Register

#### 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。

其中**Interrupt gate**和**trap gate**和另外两个的区别是，他们用来专门处理处理器异常或者中断(Exception or interrupt)，而另外两种一般处理用户的软件切换。

#### Interrupt gate 和 Trap gate 的区别 :

我们可以看到这两者的描述符基本没有区别，而实际上它们的执行工作也基本类似，除了一点！就是**Interrupt gate**会修改**IF**，会对中断响应屏蔽，即不再响应接下来的中断了

这个不同在处理什么情况时候会导致差别呢？上面链接中的网友提供了一个很好的实例，比如操作系统捕获了一个硬件中断正在处理，又来了另一个。如果用trap gate，那么第一个处理就被打断了，这样会造成数

据崩溃。所以必须屏蔽掉第二个处理。这样的话可以使一次硬件操作成为**原子操作**，保证处理的正确和完整。（当然，如果是NMI，那就另说了）

还有一个是断点异常的处理，这个中断必须暴露给用户程序进行调用。但是又要区别于中断，所以使用trap gate来处理。所以一般情况下，interrupt gate用于处理意外发生的错误，而trap gate用来处理人为制造的软件中断比如page fault、调试中断等等

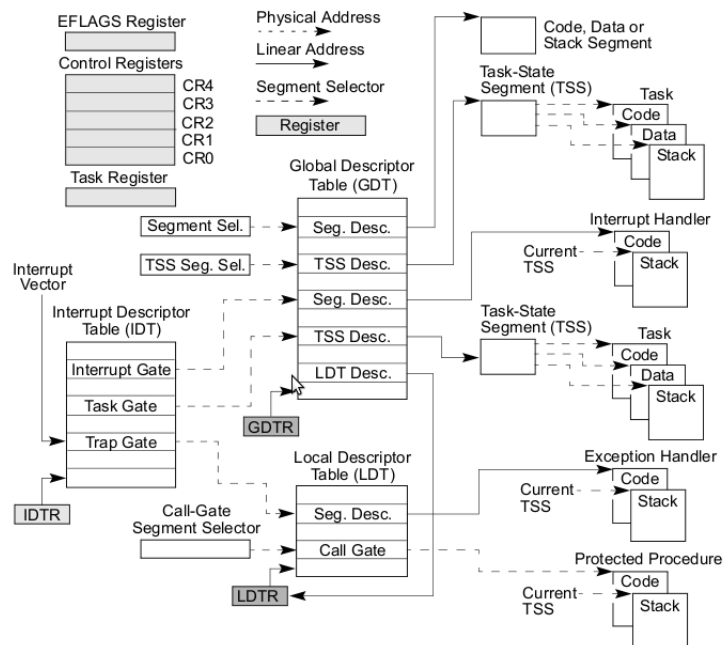
### Call gate 和 Task gate 的区别：

首先x86中引入了Task的概念，具体请看IA32-3A.pdf的**CHAPTER 6 TASK MANAGEMENT** 一章，简单来说，task就是一个具体的可运行的单位，可以运行，挂起，重启等等，然后在这个单位上可以保存其状态到TSS(Task-state segment)去。我们可以通过一个CALL或者JMP指令来具体的调用一个task程序。

call gate和task gate都可以用来切换到一个task，**但是task gate的寻址需要经过一个TSS找到code selector**（具体见task gate descriptor）。当然这样显然比call gate麻烦，但是这样带来的好处是：

- 切换到时候原来task的上下文环境被自动保存(TSS)
- 如果使用task gate来处理中断例程，那么可以使程序和其他例程分开，使其具有独立的地址空间比如页表等等

x86手册第一章有幅图能大致总结一下以上四种切换机制的具体细节：



## 2.5 Basics of Protected Control Transfer

这里开始提到IDT和TSS的具体构造，这里我主要想谈一谈TSS。

在IA32-3A.pdf的6.2.1 Task-State Segment (TSS) 有对其的详细描述，简而言之，为了能保存和恢复一个task的执行状态而引入了TSS的概念。TSS描述了一个task在执行中的状态信息。

但是我看到TSS的就想起了前面的struct Env结构，这两个东西不都是保存状态信息的么，有什么区别？

Env对应的是一个用户进程的状态，这里的进程是一个抽象程度较高的概念，而不仅仅是一段程序代码而已。它有独立的PCB，地址空间等等。所以对比struct Env和struct Taskstate的详细结构就可以知道，Env中有env\_pgdir和env\_cr3 这种页表相关的成员，表示它不仅要保存CPU的即时运行状态，还要保存其空间页表的信息。

相比之下，TSS对应的则不是进程，而是比较底层的概念，是一段汇编代码中的过程。它比struct Env多出的像ESP0，SS0，ESP1，SS1这样的结构，重在描述代码切换之间权限的转换。

**综上，TSS侧重权限，主要用于保护机制，而Env则主要用于保持用户进程的独立。**两者适用的对象不同，目的和侧重点更不同。

## 2.6 Types of Exceptions and Interrupts

### 2.7 An Example

主要注意弄清处理器在捕获中断以后的执行过程。注意当前如何切换到内核栈以及切换过去后处理器在内核栈中放了一些什么东西。这个MIT的材料里已经讲的非常清楚，不再赘述。

## 2.8 Nested Exceptions and Interrupts

### 2.9 Setting Up the IDT

具体实现的部分涉及到几个MIT材料里没有提到的东西：

首先IDT的数据结构，定义在kern/trap.c中：

```
kern/trap.c
1 struct Gatedesc idt[256] = { { 0 } };
2 struct Pseudodesc idt_pd = {
3     sizeof(idt) - 1, (uint32_t) idt
4 };
```

idt\_pd是系统寄存器IDTR的对应结构，门描述符数据结构struct Gatedesc定义在inc/mmu.h中：

```

inc/mmu.h

257 // Gate descriptors for interrupts and traps
258 struct Gatedesc {
259     unsigned gd_off_15_0 : 16;    // low 16 bits of offset in segment
260     unsigned gd_ss : 16;          // segment selector
261     unsigned gd_args : 5;         // # args, 0 for interrupt/trap gates
262     unsigned gd_rsv1 : 3;         // reserved(should be zero I guess)
263     unsigned gd_type : 4;         // type(STS_{TG,IG32,TG32})
264     unsigned gd_s : 1;           // must be 0 (system)
265     unsigned gd_dpl : 2;         // descriptor(meaning new) privilege level
266     unsigned gd_p : 1;          // Present
267     unsigned gd_off_31_16 : 16;   // high bits of offset in segment
268 };
269
270 // Set up a normal interrupt/trap gate descriptor.
271 // - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
272 //   see section 9.6.1.3 of the i386 reference: "The difference between
273 //   an interrupt gate and a trap gate is in the effect on IF (the
274 //   interrupt-enable flag). An interrupt that vectors through an
275 //   interrupt gate resets IF, thereby preventing other interrupts from
276 //   interfering with the current interrupt handler. A subsequent IRET
277 //   instruction restores IF to the value in the EFLAGS image on the
278 //   stack. An interrupt through a trap gate does not change IF."
279 // - sel: Code segment selector for interrupt/trap handler
280 // - off: Offset in code segment for interrupt/trap handler
281 // - dpl: Descriptor Privilege Level -
282 //   the privilege level required for software to invoke
283 //   this interrupt/trap gate explicitly using an int instruction.
284 #define SETGATE(gate, istrap, sel, off, dpl) \
285 { \
286     (gate).gd_off_15_0 = (uint32_t) (off) & 0xffff; \
287     (gate).gd_ss = (sel); \
288     (gate).gd_args = 0; \
289     (gate).gd_rsv1 = 0; \
290     (gate).gd_type = (istrap) ? STS_TG32 : STS_IG32; \
291     (gate).gd_s = 0; \
292     (gate).gd_dpl = (dpl); \
293     (gate).gd_p = 1; \
294     (gate).gd_off_31_16 = (uint32_t) (off) >> 16; \
295 }

```

其中提供一个很好用的宏SETGATE用来设置一个特定的描述符。其中dpl参数是我们要重点注意的，这个在后面程序中会提到：

好了，这个exercise的流程分为两步

1. 在kern/trapentry.S中定义好每个中断对应的中断处理程序
2. 在kern/trap.c的idt\_init() 中将那些第一步定义好的中断处理程序安装进IDT

首先开始定义中断处理程序，根据MIT的材料，每个interrupt handler都必须要做的事就是在内核栈中设置好一个Trapframe的布局结构，然后将这个结

构传给trap()进行进一步处理，最后在trap\_dispatch()中进行具体中断处理程序的分发。

在kern/trapentry.S中JOS提供了两个很好用的宏给我们：

```
kern/trapentry.S

1  /* The TRAPHANDLER macro defines a globally-visible function for handling
2  * a trap. It pushes a trap number onto the stack, then jumps to _alltraps.
3  * Use TRAPHANDLER for traps where the CPU automatically pushes an error code.
4  */
5  #define TRAPHANDLER(name, num)                                \
6      .globl name;                                              /* define global symbol for 'name' */ \
7      .type name, @function; /* symbol type is function */    \
8      .align 2;                                                 /* align function definition */      \
9      name:                                                      /* function starts here */           \
10     pushl $(num);                                              \
11     jmp _alltraps
12
13 /* Use TRAPHANDLER_NOEC for traps where the CPU doesn't push an error code.
14 * It pushes a 0 in place of the error code, so the trap frame has the same
15 * format in either case.
16 */
17 #define TRAPHANDLER_NOEC(name, num)                            \
18     .globl name;                                              \
19     .type name, @function;                                    \
20     .align 2;                                                  \
21     name:                                                      \
22     pushl $0;                                                  \
23     pushl $(num);                                              \
24     jmp _alltraps
25
26 .text
```

他们的功能就是接受一个函数名和对应处理的中断向量编号，然后定义出一个相应的以该函数名命名的中断处理程序。这样的中断向量程序的执行流程就是向栈里压入相关错误码和中断号，然后跳转到\_alltraps来执行共有的部分（把Trapframe剩下的那些结构在栈中设置好）

具体代码如下：

```
kern/trapentry.S

1  .text
2
3  /*
4  * Lab 3: Your code here for generating entry points for the different traps.
5  */
6
7  TRAPHANDLER_NOEC(routine_divide, T_DIVIDE)
8  TRAPHANDLER_NOEC(routine_debug, T_DEBUG)
9  TRAPHANDLER_NOEC(routine_nmi, T_NMI)
10 TRAPHANDLER_NOEC(routine_brkpt, T_BRKPT)
11 TRAPHANDLER_NOEC(routine_oflow, T_OFLOW)
12 TRAPHANDLER_NOEC(routine_bound, T_BOUND)
13 TRAPHANDLER_NOEC(routine_illop, T_ILLOP)
14 TRAPHANDLER_NOEC(routine_device, T_DEVICE)
15 TRAPHANDLER(routine_dblflt, T_DBLFLT)
16 TRAPHANDLER(routine_tss, T_TSS)
17 TRAPHANDLER(routine_segnp, T_SEGNP)
18 TRAPHANDLER(routine_stack, T_STACK)
19 TRAPHANDLER(routine_gpflt, T_GPFLT)
20 TRAPHANDLER(routine_pgflt, T_PGFLT)
21 TRAPHANDLER_NOEC(routine_fperr, T_FPERR)
```

```

22 TRAPHANDLER(routine_align, T_ALIGN)
23 TRAPHANDLER_NOEC(routine_mchk, T_MCHK)
24 TRAPHANDLER_NOEC(routine_simderr, T_SIMDERR)
25
26
27 /*
28  * Lab 3: Your code here for _alltraps
29  */
30 _alltraps:
31
32     pushw    $0x0
33     pushw    %ds
34     pushw    $0x0
35     pushw    %es
36     pushal
37
38     movl     $GD_KD, %eax
39
40     movw     %ax, %ds
41     movw     %ax, %es
42
43     pushl    %esp
44
45     call trap

```

按照注释和材料的提示即可完成，关于每个中断是否有错误码请参考邵老师的讲义Chapter 05中的第5.4.2中的那张图，其他的话在压栈时注意数据的长度大小选择对应的指令，其他就没有什么需要注意的。

把中断服务程序定义好以后，我们开始安装IDT表，看到kern/trap.c中的idt\_init()

```

                                kern/trap.c: idt_init()
1 void
2 idt_init(void)
3 {
4     extern struct Segdesc gdt[];
5
6     // LAB 3: Your code here.
7
8     extern void routine_divide ();
9     extern void routine_debug ();
10    extern void routine_nmi ();
11    extern void routine_brkpt ();
12    extern void routine_oflow ();
13    extern void routine_bound ();
14    extern void routine_illop ();
15    extern void routine_device ();
16    extern void routine_dblflt ();
17    extern void routine_tss ();
18    extern void routine_segnp ();
19    extern void routine_stack ();
20    extern void routine_gpflt ();
21    extern void routine_pgflt ();
22    extern void routine_fperr ();
23    extern void routine_align ();
24    extern void routine_mchk ();
25    extern void routine_simderr ();
26
27
28    SETGATE (idt[T_DIVIDE], 0, GD_KT, routine_divide, 0);
29    SETGATE (idt[T_DEBUG], 0, GD_KT, routine_debug, 0);
30    SETGATE (idt[T_NMI], 0, GD_KT, routine_nmi, 0);
31

```

```

32 // break point needs no kernel mode privilege
33 SETGATE (idt[T_BRKPT], 0, GD_KT, routine_brkpt, 3);
34
35 SETGATE (idt[T_OFLOW], 0, GD_KT, routine_oflow, 0);
36 SETGATE (idt[T_BOUND], 0, GD_KT, routine_bound, 0);
37 SETGATE (idt[T_ILLOP], 0, GD_KT, routine_illop, 0);
38 SETGATE (idt[T_DEVICE], 0, GD_KT, routine_device, 0);
39 SETGATE (idt[T_DBLFLT], 0, GD_KT, routine_dblflt, 0);
40 SETGATE (idt[T_TSS], 0, GD_KT, routine_tss, 0);
41 SETGATE (idt[T_SEGNP], 0, GD_KT, routine_segnp, 0);
42 SETGATE (idt[T_STACK], 0, GD_KT, routine_stack, 0);
43 SETGATE (idt[T_GPFLT], 0, GD_KT, routine_gpflt, 0);
44 SETGATE (idt[T_PGFLT], 0, GD_KT, routine_pgflt, 0);
45 SETGATE (idt[T_FPERR], 0, GD_KT, routine_fperr, 0);
46 SETGATE (idt[T_ALIGN], 0, GD_KT, routine_align, 0);
47 SETGATE (idt[T_MCHK], 0, GD_KT, routine_mchk, 0);
48 SETGATE (idt[T_SIMDERR], 0, GD_KT, routine_simderr, 0);
49
50 // Setup a TSS so that we get the right stack
51 // when we trap to the kernel.
52 ts.ts_esp0 = KSTACKTOP;
53 ts.ts_ss0 = GD_KD;
54
55 // Initialize the TSS field of the gdt.
56 gdt[GD_TSS >> 3] = SEG16(STS_T32A, (uint32_t) (&ts),
57                          sizeof(struct Taskstate), 0);
58 gdt[GD_TSS >> 3].sd_s = 0;
59
60 // Load the TSS
61 ltr(GD_TSS);
62
63 // Load the IDT
64 asm volatile("lidt_idt_pd");
65 }

```

这段程序中唯一值得留意的地方就是SETGATE宏里两个参数的设置：

1. 第三个参数cs，记得设置为内核的代码段GD.KT
2. 最后一个用户特权级的设置，一开始我就搞错了。怎么设置？就拿除零为例，你肯定不想让用户int 0这么毫无意义的调用。所以0号中断只能由level 0的内核产生（运行时抛出），但是调试是例外，应该能让用户自发调用

到这里，我们的中断响应机制就建立起来了。根据代码，如果一个除零中断被捕获，会转到kern/trapentry.S中的routine\_divide()，然后跳转到\_alltraps，然后是kern/trap.c中的trap()：

```

                                kern/trap.c
1 static void
2 trap_dispatch(struct Trapframe *tf)
3 {
4     // Handle processor exceptions.
5     // LAB 3: Your code here.
6

```

```

7
8     // Unexpected trap: The user process or the kernel has a bug.
9     print_trapframe(tf);
10    if (tf->tf_cs == GD_KT)
11        panic("unhandled_trap_in_kernel");
12    else {
13        env_destroy(curenv);
14        return;
15    }
16 }
17
18 void
19 trap(struct Trapframe *tf)
20 {
21     // The environment may have set DF and some versions
22     // of GCC rely on DF being clear
23     asm volatile("cld" ::: "cc");
24
25     // Check that interrupts are disabled. If this assertion
26     // fails, DO NOT be tempted to fix it by inserting a "cli" in
27     // the interrupt path.
28     assert(!(read_eflags() & FL_IF));
29
30     cprintf("Incoming_TRAP_frame_at_%p\n", tf);
31
32     if ((tf->tf_cs & 3) == 3) {
33         // Trapped from user mode.
34         // Copy trap frame (which is currently on the stack)
35         // into 'curenv->env_tf', so that running the environment
36         // will restart at the trap point.
37         assert(curenv);
38         curenv->env_tf = *tf;
39         // The trapframe on the stack should be ignored from here on.
40         tf = &curenv->env_tf;
41     }
42
43     // Dispatch based on what type of trap occurred
44     trap_dispatch(tf);
45
46     // Return to the current environment, which should be runnable.
47     assert(curenv && curenv->env_status == ENV_RUNNABLE);
48     env_run(curenv);
49 }

```

从代码中可以看到，最终程序会进入trap\_dispatch() 打印出寄存器信息，那么我们尝试着运行一个有除零错误的用户程序试试，将kern/init.c中载入的第一个程序设置为user\_divzero:

```

kern/init.c: i386_init()
44     // Temporary test code specific to LAB 3
45 #if defined(TEST)
46     // Don't touch -- used by grading script!
47     ENV_CREATE2(TEST, TESTSIZE);
48 #else
49     // Touch all you want.
50     //ENV_CREATE(user_hello);
51     ENV_CREATE(user_divzero);
52 #endif // TEST*
53
54     // We only have one user environment for now, so just run it.
55     env_run(&envs[0]);

```

接下来编译启动QEMU，可以看到正确的处理画面：

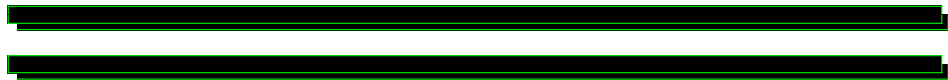


```

[00000000] new env 00001000
Incoming TRAP frame at 0xefbfffbc
TRAP frame at 0xf01af000
edi 0x00000000
esi 0x00000000
ebp 0xeebffd0
oesp 0xefbfffdc
ebx 0x00000000
edx 0x00000000
ecx 0x00000000
eax 0x00000001
es 0x---0023
ds 0x---0023
trap 0x00000000 Divide error
err 0x00000000
eip 0x00000044
cs 0x---001b
flag 0x00000046
esp 0xeebffb8
ss 0x---0023
[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>

```

至此Part A就全部完成了，欢呼一下！



kern/env.c: env\_create()



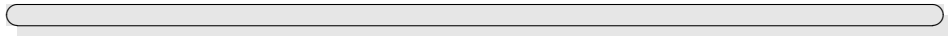
kern/env.c: env\_create()



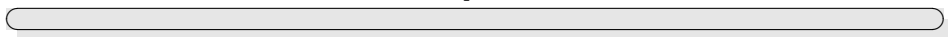
kern/env.c: env\_create()



kern/env.c: env\_create()



inc/queue.h



inc/queue.h



inc/queue.h

inc/queue.h



### 3 Page Faults, Breakpoints Exceptions, and System Calls

#### 3.1 Handling Page Faults

#### 3.2 The Breakpoint Exception

#### 3.3 System calls

#### 3.4 User-mode startup

#### 3.5 Page faults and memory protection