JOS 2016 Lab 3 实习报告

陈一茹 1400012976

October 31, 2016

Contents

1 Introduction

之前，lab1, lab2我们只要是在考虑内核相关的事情，现在Lab3, 考虑到了用户进程了。

1.1 Inline Assembly

一开始，指导手册让我们去看看inline assemble, 中文名是内联汇编,，这个语法的基本格式是：

```
1 asm [ volatile ] (
2 assembler template
3 [ : output operands ]                    /* optional */
4 [ : input operands  ]                      /* optional */
5 [ : list of clobbered registers ]      /* optional */
6 );
```

关键字是asm和volatile，也可以是 _asm_ 或者 _volatile_。
以下是一个示例：

```
1 int a=10, b;
2 asm ("movl %1, %%eax;
3 movl %%eax, %0;"
4 :"=r"(b)           /* output
5 :"r"(a)            /* input    */
6 :"%eax"            /* clobbered register */
7 );
```

在内联汇编中，操作数通常用数字来引用，具体的编号规则为：若命令共涉及n个操作数，则第1个输出操作数（the first output operand）被编号为0，第2个output operand编号为1，依次类推，最后1个输入操作数（the last input operand）则被编号为n-1。

具体到上面的示例代码中，根据上下文，涉及到2个操作数变量a、b，这段汇编代码的作用是将a的值赋给b，可见，a是input operand，而b是output operand，那么根据操作数的引用规则，不难推出，a应该用%1来引用，b应该用%0来引用。 还需要说明的是：当命令中同时出现寄存器和以%num来引用的操作数时，会以%%reg来引用寄存器（如上例中的%%eax），以便帮助gcc来区分寄存器和由C语言提供的操作数。

还有一个需要注意的： list of clobbered registers 该字段为可选项，用于列出指令中涉及到的且没出现在output operands字段及input operands字段的那些寄存器。若寄存器被列入clobber-list，则等于是告诉gcc，这些寄存器可能会被内联汇编命令改写。因此，执行内联汇编的过程中，这些寄存器就不会被gcc分配给其它进程或命令使用。

2 Part A: User Environments and Exception Handling

inc/env.h包含了一些用户环境的基本定义。这个env就是课上讲的PCB。

然后看一看inc/env.c：

```
struct Env *envs = NULL;    // All environments
struct Env *curenv = NULL;    // The current env
static struct Env *env_free_list; // Free environment list
          // (linked by Env->env_link)
```

envs是所有进程的env构成的数组，curenv是当前进程的数据结构。env free list是空闲列表。

(NENV is a constant #define'd in incenv.h.)表示最多支持NENV个进程。

在第一个进程运行之前，curenv初始化成null。

2.1 Environment State

这里讲了env的结构，定义在 inc/env.h中, 就是PCB的内容：

```
struct Env {
struct Trapframe env_tf;  // Saved registers
struct Env *env_link;   // Next free Env
envid_t env_id;     // Unique environment identifier
envid_t env_parent_id;    // env_id of this env's parent
enum EnvType env_type;    // Indicates special system
    environments
unsigned env_status;    // Status of the environment
uint32_t env_runs;    // Number of times environment has run

// Address space
pde_t *env_pgdir;   // Kernel virtual address of page dir
};
```

struct Trapframe env_tf; // Saved registers

这个是用来保存一个一个用户进程寄存器信息的结构，当一个用户进程被换下CPU的时候，这个进程的寄存器信息需要被保存。用来保存这个的结构是Trapframe，defined in inc/trap。

```
struct Trapframe {
struct PushRegs tf_regs;
uint16_t tf_es;
uint16_t tf_padding1;
uint16_t tf_ds;
uint16_t tf_padding2;
uint32_t tf_trapno;
/* below here defined by x86 hardware */
uint32_t tf_err;
uintptr_t tf_eip;
```

```
11   uint16_t tf_cs;
12   uint16_t tf_padding3;
13   uint32_t tf_eflags;
14   /* below here only when crossing rings, such as from user to
         kernel */
15   uintptr_t tf_esp;
16   uint16_t tf_ss;
17   uint16_t tf_padding4;
18   } __attribute__((packed));
```

保存了许多相关寄存器的值，padding是用来对齐的。 其中 tf_reg是一个PushRegs 对象，里面也是一写寄存器，但是他们区别于Trapframe中的寄存器，又用了一个数据结构包装了一下。

```
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));
```

如上

struct Env *env_link; // Next free Env

这是一个 指向 env_free_list.的指针， env_free_list第一个指向第一个空闲的进程。

envid_t env_id; // Unique environment identifier

进程号。

envid_t env_parent_id; // env_id of this env's parent

父进程号。

enum EnvType env_type; // Indicates special system environments

标识用户进程，在大多数的进程中是ENV_TYPE_USER，以后还会有其他的，这里先不管了。

unsigned env_status; // Status of the environment

这个表示env的状态，

```
1     // Values of env_status in struct Env
2       enum {
3         ENV_FREE = 0,       // Env structure is inactive, and
              therefore on the env_free_list.
4         ENV_DYING,        //Env structure represents a zombie
              environment.
5         ENV_RUNNABLE,     //Env structure represents an
              environment that is waiting to run on the processor
              .
```

```
6         ENV_RUNNING,     //structure represents the currently
              running environment.
7         ENV_NOT_RUNNABLE   //Env structure represents a
              currently active environment, but it is not
              currently ready to run: for example, because it is
              waiting for an interprocess communication (IPC)
              from another environment.
8       };
```

uint32_t env_runs; // Number of times environment has run
记录这个进程跑过多少次了，用于调度算法。
地址空间：
pde_t *env_pgdir; // Kernel virtual address of page dir
记录进程页目录的虚拟地址。
注意指导中说到，JOS中一次只有一个用户进程活跃，所以只需要一个内核栈，这个和xv6是不同的。

2.2 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 `UENVS` (defined in `inc/memlayout.h`) so user processes can read from this array.

You should run your code and make sure `check_kern_pgdir()` succeeds.

```
1   ////////////////////////////////////////////////////////////////////
2   // Make 'envs' point to an array of size 'NENV' of 'struct
        Env'.
3   // LAB 3: Your code here.
4    envs = (struct Env*)boot_alloc(NENV * sizeof(struct Env));
```

然后我们还要用boot_map_region 函数映射一下, 使得虚拟地址UENVS和和envs指向同一个物理地址，这样，我们知道UENVS是内核不可写的，但是envs是内核可写的，用户不可读的，所以我们就可以完成用户在0xf0000000一下读的要求。

```
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 UENVS  -- kernel R, user R
6   //    - envs itself -- kernel RW, user NONE
```

```
7   // LAB 3: Your code here.
8   boot_map_region(kern_pgdir, UENVS, ROUNDUP(sizeof(struct Env)
        * NENV, PGSIZE), PADDR(envs), PTE_U);
```

检查一下check_kern_pgdir() succeeded! 就可以了。

2.3 Creating and Running Environments

这里需要在kern/env.c 写代码去运行一个用户进程，因为我们没有文件系统，所有我们要载入一些静态的二进制镜像嵌入到JOS系统中，JOS把这些内核中的二进制存成 ELF executable image.

以下这个exercise要我们去补全这些运行二进制镜像的代码。

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` to load an ELF binary into it.

`env_run()`
Start a given environment running in user mode.

As you write these functions, you might find the new cprintf verb `%e` useful -- it prints a description corresponding to an error code. For example,

```
r = -E_NO_MEM;
panic("env_alloc: %e", r);
```

will panic with the message "env_alloc: out of memory".

env_init()

首先是 env_init()，用来初始化 envs 数组，在系统启动时运行一次。将所有进 程初始化为未使用状态，并加入 env_free_list 链表中。这个链表保存所有未 使用的进程，方便快速新建进程。注意 env_free_list 中进程的顺序，小标号 的进程要在前面。最后的env_init_percpu()的是初始化关于CPU的部分。

```
1   // Mark all environments in 'envs' as free, set their env_ids
        to 0,
```

```
2   // and insert them into the env_free_list.
3   // Make sure the environments are in the free list in the
        same order
4   // they are in the envs array (i.e., so that the first call
        to
5   // env_alloc() returns envs[0]).
6   //
7   void
8   env_init(void)
9   {
10    // Set up envs array
11    // LAB 3: Your code here.
12    int i;
13      env_free_list = NULL;
14      for (i = NENV - 1; i >= 0; i--) {
15    envs[i].env_id = 0;
16    envs[i].env_link = env_free_list;
17    env_free_list = &(envs[i]);
18      }
19
20    // Per-CPU part of the initialization
21    env_init_percpu();
22  }
```

env_setup_vm()

```
1   //
2   // Initialize the kernel virtual memory layout for
        environment e.
3   // Allocate a page directory, set e->env_pgdir accordingly,
4   // and initialize the kernel portion of the new environment's
         address space.
5   // Do NOT (yet) map anything into the user portion
6   // of the environment's virtual address space.
7   //
8   // Returns 0 on success, < 0 on error.  Errors include:
9   //  -E_NO_MEM if page directory or table could not be
        allocated.
10  //
11  static int
12  env_setup_vm(struct Env *e)
13  {
14    int i;
15    struct PageInfo *p = NULL;
16
17    // Allocate a page for the page directory
18    if (!(p = page_alloc(ALLOC_ZERO)))
19      return -E_NO_MEM;
20
21    // Now, set e->env_pgdir and initialize the page directory.
22    //
23    // Hint:
24    //    - The VA space of all envs is identical above UTOP
25    //  (except at UVPT, which we've set below).
```

```
26      //  See inc/memlayout.h for permissions and layout.
27      //  Can you use kern_pgdir as a template?  Hint: Yes.
28      //  (Make sure you got the permissions right in Lab 2.)
29      //    - The initial VA below UTOP is empty.
30      //    - You do not need to make any more calls to
            page_alloc.
31      //    - Note: In general, pp_ref is not maintained for
32      //  physical pages mapped only above UTOP, but env_pgdir
33      //  is an exception -- you need to increment env_pgdir's
34      //  pp_ref for env_free to work correctly.
35      //    - The functions in kern/pmap.h are handy.
36
37      // LAB 3: Your code here.
38      p->pp_ref++;
39        e->env_pgdir = (pde_t*)page2kva(p);
40        boot_map_region(e->env_pgdir, UPAGES, ROUNDUP(sizeof(
              struct PageInfo) * npages, PGSIZE), PADDR(pages),
              PTE_U);
41        boot_map_region(e->env_pgdir, UENVS, ROUNDUP(sizeof(
              struct Env) * NENV, PGSIZE), PADDR(envs), PTE_U);
42        boot_map_region(e->env_pgdir, KSTACKTOP - KSTKSIZE,
              ROUNDUP(KSTKSIZE, PGSIZE), PADDR(bootstack), PTE_W);
43        boot_map_region(e->env_pgdir, (uintptr_t)KERNBASE , -
              KERNBASE, (physaddr_t)0, PTE_W);
44
45      // UVPT maps the env's own page table read-only.
46      // Permissions: kernel R, user R
47      e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_P |
            PTE_U;
48
49      return 0;
50    }
```

这个函数完成的就是： 分配一个页目录给一个新的进程，并且在这个新进程的地址空间中配置内核部分，要求和内核虚拟内存相似。并且初始化在UTOP下面的空间都是空的。除此以外，还实现了env_pgdir's的pp_ref，之后要用到的。

由此我们可以看出，在JOS中，在UTOP以上的所有内存空间都是一样的，用户进程只是用UTOP一下的内存，我们只要实现对UTOP一下内存的管理就好了。

另外boot_map_region()函数在pmap.c中是静态函数，所以在本文件中，我需要再复制一遍。

region_alloc() 这个就是在用户进程的虚拟地址空间上va上分配len字节的空间。 这里的思路是先page_alloc 然后map 一下， page_insert就好了，比较简单, 注意一下权限就好了。

```
1   //
2   // Allocate len bytes of physical memory for environment env,
3   // and map it at virtual address va in the environment's
        address space.
4   // Does not zero or otherwise initialize the mapped pages in
```

```
      any way.
// Pages should be writable by user and kernel.
// Panic if any allocation attempt fails.
//
static void
region_alloc(struct Env *e, void *va, size_t len)
{
  // LAB 3: Your code here.
  // (But only if you need it for load_icode.)
  //
  // Hint: It is easier to use region_alloc if the caller can pass
  //   'va' and 'len' values that are not page-aligned.
  //   You should round va down, and round (va + len) up.
  //   (Watch out for corner-cases!)
   int l = 0, va_ = (uintptr_t)va;
    struct PageInfo *p;
    va = (void*)ROUNDDOWN(va_, PGSIZE);
    len = ROUNDUP(va_ + len, PGSIZE) - (uintptr_t)va;
    for (; l < len; l += PGSIZE) {
    p = page_alloc(0);
    if (!p) panic("Panic: region_alloc()\n");
    if (page_insert(e->env_pgdir, p, va + l, PTE_U | PTE_W))
      panic("Panic: region_alloc()\n");
  }
}
```

load_icode()

之前也说了，JOS没有文件系统的，所以为了执行一个用户程序，就要把用户程序的二进制文件直接绑定到“内核”上，作为内核文件的一部分加入到内存中间去。这里就是将JOS内核中的二进制文件拷入到用户进程的地址空间去。

注意一个技巧，在拷贝内存之前切换成 e 的页目录，这样可以方便使用 memset 等函数，之后再切换回内核的。否则由于 region_alloc 申请的内存在内核空间 中不一定是连续的，只能一页一页拷贝，复杂很多。 最后为 e 分配一页的栈空间，并设置好程序入口，即在 tf_eip 中。整体可以参 考 boot/main.c 中的做法。

```
  // LAB 3: Your code here.
  struct Elf *elf = (struct Elf*)binary;
    struct Proghdr *ph, *eph;
    struct PageInfo *pp;
    unsigned i, va, sz, delta;

    if (elf->e_magic != ELF_MAGIC)
  panic("Panic: load_icode() ELF_MAGIC\n");
    ph = (struct Proghdr*)(binary + elf->e_phoff);
    eph = ph + elf->e_phnum;
    lcr3(PADDR(e->env_pgdir));
    for (; ph < eph; ph++) {
  if (ph->p_type != ELF_PROG_LOAD) continue;
  region_alloc(e, (void*)ph->p_va, ph->p_memsz);
```

```
15      memset((void*)ph->p_va, 0, ph->p_memsz);
16      memcpy((void*)ph->p_va, binary + ph->p_offset, ph->p_filesz
            );
17          }
18        lcr3(PADDR(kern_pgdir));
19        e->env_tf.tf_eip = elf->e_entry;
```

```
1     // Now map one page for the program's initial stack
2     // at virtual address USTACKTOP - PGSIZE.
3     // LAB 3: Your code here.
4       region_alloc(e, (void*)(USTACKTOP - PGSIZE), PGSIZE);
5   }
```

env_create() 创建第一个进程，用刚刚写的env_alloc()(这个函数已经写好了)和load_icode()函数，这个进程是在内核初始化的时候被创建的，进程id是0，比第一个用户进程早。

```
1   void
2   env_create(uint8_t *binary, enum EnvType type)
3   {
4     // LAB 3: Your code here.
5      struct Env *e;
6       env_alloc(&e, 0);
7       load_icode(e, binary, size);
8       e->env_type = type;
9   }
```

env_run()

```
1   // LAB 3: Your code here.
2   if (curenv) {
3     if (curenv->env_status == ENV_RUNNING)
4       curenv->env_status = ENV_RUNNABLE;
5     }
6     curenv = e;
7     curenv->env_status = ENV_RUNNING;
8     curenv->env_runs++;
9     lcr3(PADDR(e->env_pgdir));
10
11    env_pop_tf(&e->env_tf);
```

按照注释改的，最后修改一下cr3寄存器的值。env_run就写好了。 然后在来关注一下env_pop_tf函数：

```
1   //
2   // Restores the register values in the Trapframe with the '
        iret' instruction.
3   // This exits the kernel and starts executing some
        environment's code.
4   //
5   // This function does not return.
```

```
6   //
7   void
8   env_pop_tf(struct Trapframe *tf)
9   {
10    asm volatile(
11      "\tmovl %0,%%esp\n"
12      "\tpopal\n"
13      "\tpopl %%es\n"
14      "\tpopl %%ds\n"
15      "\taddl $0x8,%%esp\n" /* skip tf_trapno and tf_errcode */
16      "\tiret\n"
17      : : "g" (tf) : "memory");
18    panic("iret failed");  /* mostly to placate the compiler */
19  }
```

这个是之前学习的内嵌式汇编，第11行，我们可以看到，先将tf的值放进esp中，然后不断地pop，由于栈是从高向低增长的，所以，pop就是不断地将tf中的值，存进相应的寄存器。第12行，一下子弹出8个，是PushRegs中的值。然后对应的寄存器值进行保存，跳掉了8字节，tf_trapno and tf_errcode。 最后执行，iret , 查询intel 手册，是Return from interrupt的意思。intel 手册还说了，iret 弹栈三次，存入EIP, CS，EFLAGS的值。

这里要注意的是，即使没有发生新的进程间的切换，我们也要做env_pop_tf操作，因为这里发生过用户进程到内核进程的切换，所以寄存器发生过变化，需要重新切换回去。

”Triple fault“，然后按指示进行gdb调试：

操作流程：

```
1  make qemu-nox-gdb
2  make gdb
3  b env_pop_tf
4  c
5  si .... si
```

```
0xf01039d0      481             asm volatile(
(gdb)
=> 0xf01039d1 <env_pop_tf+12>:  add    $0x8,%esp
0xf01039d1      481             asm volatile(
(gdb)
=> 0xf01039d4 <env_pop_tf+15>:  iret
0xf01039d4      481             asm volatile(
(gdb) info refister
Undefined info command: "refister".  Try "help info".
(gdb) info registers
eax            0x0      0
ecx            0x0      0
edx            0x0      0
ebx            0x0      0
esp            0xf01c1030       0xf01c1030
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0xf01039d4       0xf01039d4 <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) x/x 0xf01c1030
0xf01c1030:     0x00800020
(gdb)
0xf01c1034:     0x0000001b
(gdb) si
=> 0x800020:    cmp    $0xeebfe000,%esp
0x00800020 in ?? ()
(gdb)
=> 0x800026:    jne    0x80002c
```

再参考， obj/user/hello.asm ，发现正确地进入了用户进程。 继续lab。

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.

虽然学过了，但是还是读一下，我想我学过的可能和lab中的实现有点不同。

Interrupts

Maskable interrupts, which are signalled via the INTR pin.

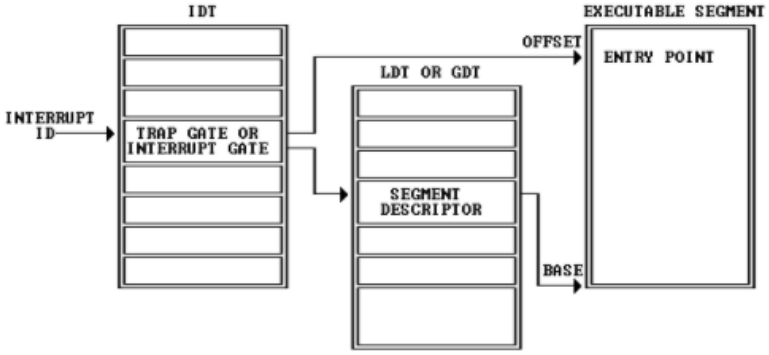
Nonmaskable interrupts, which are signalled via the NMI (Non-Maskable Interrupt) pin.

Exceptions

Processor detected. These are further classified as faults, traps, and aborts.

Programmed. The instructions INTO, INT 3, INT n, and BOUND can trigger exceptions. These instructions are often called "software interrupts", but the processor handles them as exceptions.

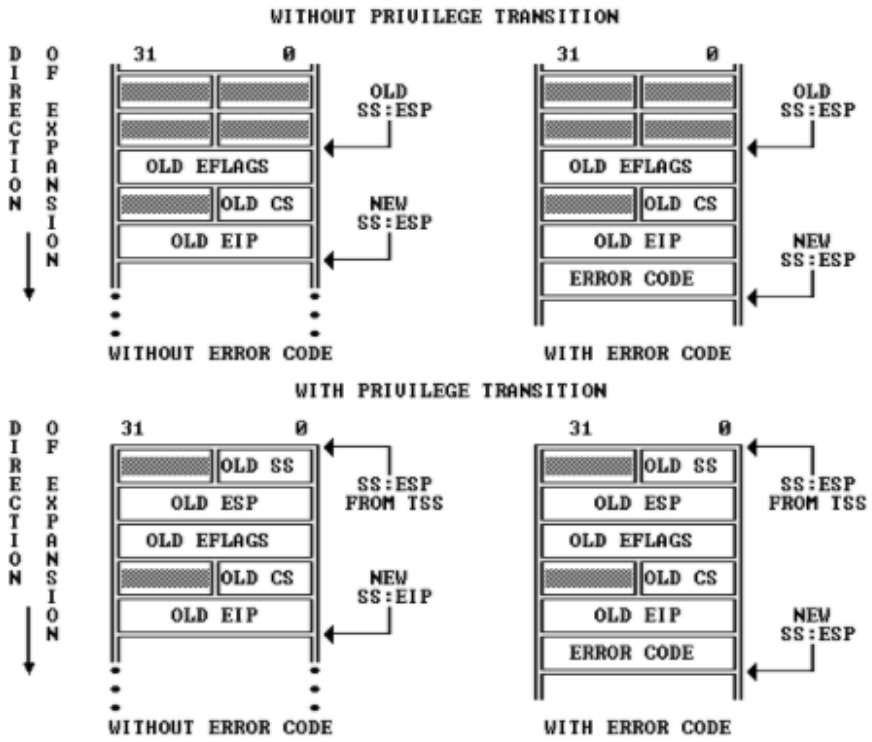
Figure 9-4. Interrupt Vectoring for Procedures



这个是中断的流程图。

中断的返回和正常程序的返回是不同的，中断的返回用的是iret指令，这个比正常的返回指令多4字节, 压入EFLAGS，CS， EIP三个值。

Figure 9-5. Stack Layout after Exception of Interrupt



这两个不同是因为，这其中一个是带error code的，一个是不带error code的。

细节方便可以查询intel手册。这里贴上Intel手册中的EFLAGS位。

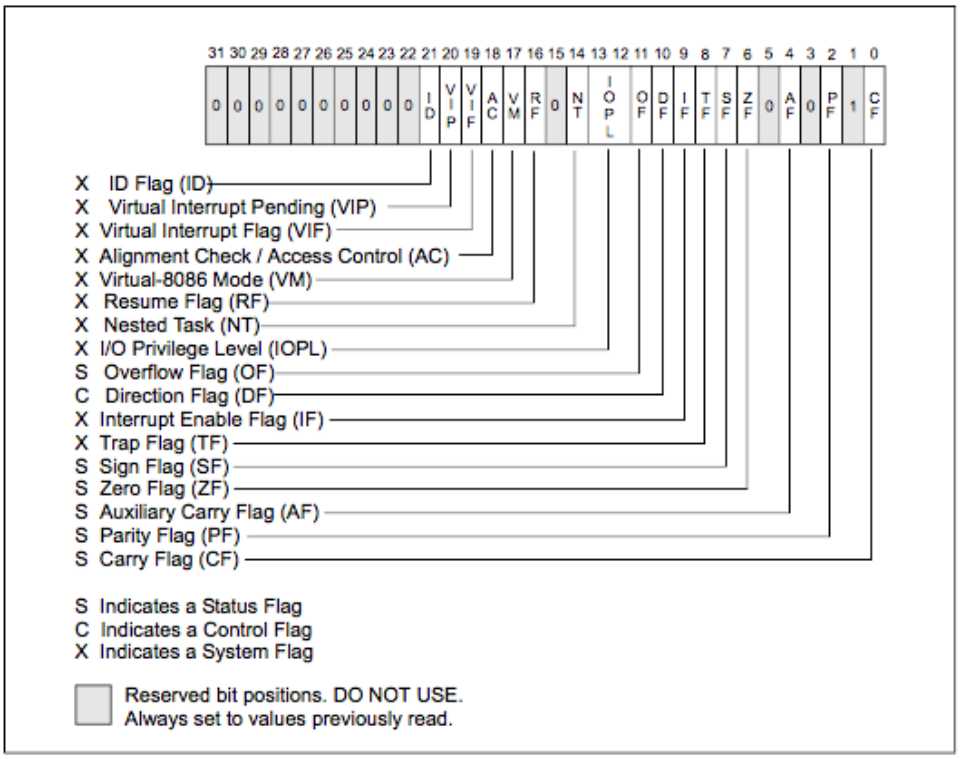


Figure 3-8. EFLAGS Register

2.5 Basics of Protected Control Transfer

中断和异常都是Protected Control Transfer, 为了达成这个目标，JOS采用了两种机制：中断描述符表和TSS段：

中断描述符表

中断描述符表中允许有256种不同中断类型，0-255，这些都是由内核确定的。CPU用中断向量在在中断描述符表中找到对应的条目。从中断描述符中，我们可以找到相应的中断处理程序，并且将相应的值加载到eip和CS中， 在JOS中，这里的Cs都是0，所有的异常都在内核模式中处理。

TSS 任务状态段

在跳转到中断处理程序之前，我们要保存一下旧的CPU状态，（比如EIP,CS寄存器值）。并且这部分空间也要保存起来，防止用户不小心损坏。

所以，当x86cpu处理一次从用户态向内核态的陷入时，需要把旧的cpu状态保存到一个内核地址空间中的栈去，而这个栈的位置就保存在TSS中。**A**

structure called the task state segment (TSS) specifies the segment selector and address where this stack lives. 每次陷入的时候，CPU先把当前的使用的栈从用户进程的栈切换到这个内核栈，把旧的CPU信息（比如各个寄存器的值）压到这个新的内核状态的栈中，然后再进行中断处理程序的执行。

TSS的功能是非常丰富的，但是在JOS中，他只是被用来存放这个内核栈的信息，由于这个内核栈的访问级别的是0，所有他的信息保存在TSS段的SS0和ESP0域中，而且JOS不在其他地方存放TSS。

补充，SS是堆栈寄存器，CS是代码段寄存器。ESP是栈顶寄存器。

2.6 Types of Exceptions and Interrupts

这里有一个例子阐述了整个流程：

- 假设当前CPU正在执行一段用户程序，发现一条指令试图进行除零操作。
- 处理器会跳转到被TSS数据结构定义的内核态的栈上去，这里SS0和ESP0的值分别是GD _KD 和 KSTACKTOP。
- 找到内核态的栈之后， CPU将当前CPU的一些状态压栈，（其他寄存器信息需要由系统来维护）

```
1     +--------------------+ KSTACKTOP
2     |0x00000|  oldSS      |               "-4
3     | old ESP                    |              "-8
4         | old EFLAGS                |               "-12
5         |0x00000|oldCS         |              "-16
6         | old EIP                       |              "-20
                <----ESP +---------------------+
```

- 因为我们处理的是一个除零异常，这个下表就是零。因此，CPU从IDT中找到第0项，从中读出中断处理程序入口处的CS和EIP值。并且设置CS和EIP寄存器的值。接下来，执行中断处理程序。
- 中断处理程序在内核态，由内核态接管。
- 如果有error code，那么需要将error code加进去。

2.7 Nested Exceptions and Interrupts

这里还需要注意的是： CPU能够处理来自内核态和用户态两个状态下的中断和异常。 只有在中断发生在用户态陷入内核态的时候才会进行栈的切换。 如果发生中断的时候，已经处于内核态了，那么就没有必要进行栈的切换了。所以旧的SS和ESP就不用压栈了。

这样的话，如果有嵌套的中断或者系统调用，就可以很优雅的执行，因为我们已经在内核态了。不需要进行栈的切换。因此，SS和ESP寄存器是不用压栈的。

栈的状况如下：

```
1   +--------------------+ <---- old ESP
2   | old EFLAGS              |                "-4
3   |0x00000|oldCS         |               "-8
4   | oldEIP                       |                "-12
```

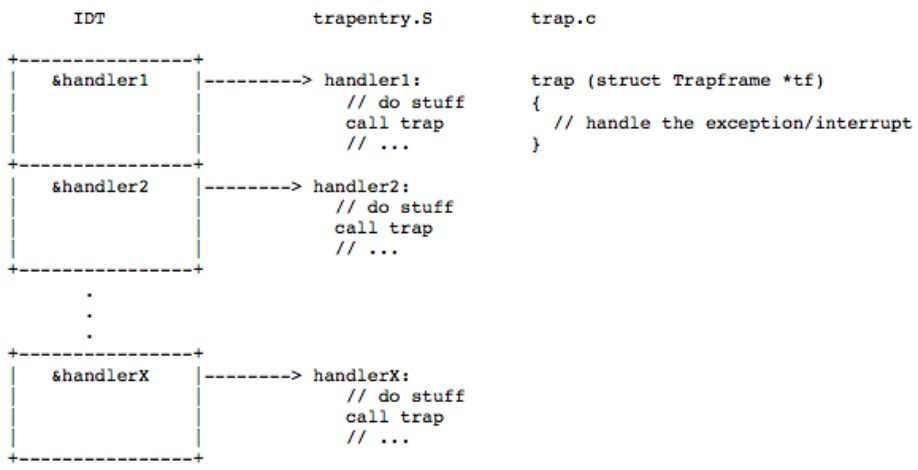
如果有error code， 那么就在EIP的后面接上。

2.8 Setting Up the IDT

设置0-31号的中断向量。

之前我们说的保存现场都是由CPU完成的，现在我们要看系统在遇到中断的时候做什么。

控制的整体流程图：



中断代码层面上的流程

- 中断发生后，CPU响应，从IDT中取出中断向量，找到中断门或者陷阱门。
- 通过中断门或者陷阱门跳转到内核态中指定的程序入口，JOS 中是 kern / trapentry.S。
- 在kern/trapentry.s中， 先往栈中压入一个中断号，然后跳转到_alltraps，保存当前用户进程信息。
- _alltrap完成之后， 调用kern/trap.c中的trap()
- trap, 更新curenv的trapframe信息，以保证进程恢复的时候可以继续从断点处执行，然后调用trap_dispatch()选择对应的中断处理程序进行处理。

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.

这个是调用这两种宏定义

```
1   #define TRAPHANDLER_NOEC(name, num)
2   #define TRAPHANDLER(name, num)
```

来定义出我们需要的中断处理程序，name是随意的，所以我简单点，跟中断号一样。这两个宏的区别是一个压入error code, 一个不压入error code。查找了一下资料，我完善了前面18个函数，后面的现在还没有用到，所以先这样写。

```
1 /*
2  * Lab 3: Your code here for generating entry points for the
       different traps.
3  */
4 .data
5 .global vectors
6 vectors:
7 .long vector0
8 .long vector1
9 .long vector2
10 .long vector3
11 .long vector4
12 .long vector5
13 .long vector6
14 .long vector7
15 .long vector8
16 .long vector9
17 .long vector10
```

```
18 .long vector11
19 .long vector12
20 .long vector13
21 .long vector14
22 .long vector15
23 .long vector16
24
25 TRAPHANDLER_NOEC(vector0, 0)
26 TRAPHANDLER_NOEC(vector1, 1)
27 TRAPHANDLER_NOEC(vector2, 2)
28 TRAPHANDLER_NOEC(vector3, 3)
29 TRAPHANDLER_NOEC(vector4, 4)
30 TRAPHANDLER_NOEC(vector5, 5)
31 TRAPHANDLER_NOEC(vector6, 6)
32 TRAPHANDLER_NOEC(vector7, 7)
33 TRAPHANDLER(vector8, 8)
34 TRAPHANDLER(vector9, 9)
35 TRAPHANDLER(vector10, 10)
36 TRAPHANDLER(vector11, 11)
37 TRAPHANDLER(vector12, 12)
38 TRAPHANDLER(vector13, 13)
39 TRAPHANDLER(vector14, 14)
40 TRAPHANDLER(vector15, 15)
41 TRAPHANDLER_NOEC(vector16, 16)
42 TRAPHANDLER(vector17, 17)
43 TRAPHANDLER_NOEC(vector18, 18)
44 TRAPHANDLER_NOEC(vector19, 19)
```

_alltraps() 这个函数需要：

- 由于有个一部分寄存器的值已经被保存了，所以这个函数就是要保存剩下的寄存器的值
- GD_KD放入%ds和%es中。
- 将ESP压栈，压栈是想作为一个参数传给trap 函数，这样，就相当于把一个已经存好的struct Trapframe （因为上面是按照这个格式存的），传给了trap函数。
- 调用trap函数。

```
1  /*
2   * Lab 3: Your code here for _alltraps
3   */
4  .text
5  _alltraps:
6  pushw $0x0
7  pushw %ds
8  pushw $0x0
9  pushl %es
10 pushal
11
```

```
12   movw $GD_KD, %ax
13   movw %ax, %ds
14   movw %ax, %es
15   pushl %esp
16   call trap
```

e 指导里说了 pushal 完美的匹配了Trapframe,具体我就不深究了。这个和Trapframe正好相反的从后往前push，因为后面部分的值已经由CPU保存了，所以从中断号开始push。

接着是**trap_init()**函数

这个函数要用SETGATE macro ，介绍一下SETGATE macro ：

```
1   #define SETGATE(gate, istrap, sel, off, dpl)      \
2   {                 \
3     (gate).gd_off_15_0 = (uint32_t) (off) & 0xffff;   \
4     (gate).gd_sel = (sel);          \
5     (gate).gd_args = 0;         \
6     (gate).gd_rsv1 = 0;         \
7     (gate).gd_type = (istrap) ? STS_TG32 : STS_IG32;  \
8     (gate).gd_s = 0;          \
9     (gate).gd_dpl = (dpl);          \
10    (gate).gd_p = 1;          \
11    (gate).gd_off_31_16 = (uint32_t) (off) >> 16;   \
12  }
```

- gate : 门
- istrap : 门类型，1是陷阱门，0是中断门
- sel : 所在的代码段
- off : 段内偏移
- dpl: 触发的权限，0代表只有内核可以触发，3代表用户也可以触发

接着可以完成代码

```
1   void
2   trap_init(void)
3   {
4     extern struct Segdesc gdt[];
5     extern long* vectors;
6     int i;
7     // LAB 3: Your code here.
8     for (i = 0; i <= 0x30; ++i) {
9         switch (i) {
10      case T_BRKPT:
11      case T_SYSCALL:
12          SETGATE(idt[i], 0, GD_KT, vectors[i], 3);
13          break;
14      default:
```

```
15            SETGATE(idt[i], 0, GD_KT, vectors[i], 0);
16          }
17      }
18      // Per-CPU setup
19      trap_init_percpu();
20  }
```

补充一点解释：

GD_KT，这个是内核代码段，所以这里的所在代码段就是GD_KT。然后在中断向量表中偏移就是中断号。

// Per-CPU setup trap_init_percpu(); 这个函数设置了TSS和IDTR，这样就可以找到内核栈和中断向量表了。

这里make grade， part A 已经完成了。

再来看一下kern/trap.c中的trap()函数：

```
1   void
2   trap(struct Trapframe *tf)
3   {
4           // The environment may have set DF and some versions
5           // of GCC rely on DF being clear
6           asm volatile("cld" ::: "cc");
7
8           // Check that interrupts are disabled.  If this
                assertion
9           // fails, DO NOT be tempted to fix it by inserting a
                "cli" in
10          // the interrupt path.
11          assert(!(read_eflags() & FL_IF));
12
13          cprintf("Incoming TRAP frame at %p\n", tf);
14
15          if ((tf->tf_cs & 3) == 3) {
16                  // Trapped from user mode.
17                  assert(curenv);
18
19                  // Copy trap frame (which is currently on the
                        stack)
20                  // into 'curenv->env_tf', so that running the
                        environment
21                  // will restart at the trap point.
22                  curenv->env_tf = *tf;
23                  // The trapframe on the stack should be
                        ignored from here on.
24                  tf = &curenv->env_tf;
25          }
26
27          // Record that tf is the last real trapframe so
28          // print_trapframe can print some additional
                information.
29          last_tf = tf;
30
31          // Dispatch based on what type of trap occurred
```

```
32              trap_dispatch(tf);
33
34              // Return to the current environment, which should be
                    running.
35              assert(curenv && curenv->env_status == ENV_RUNNING);
36              env_run(curenv);
37      }
```

Questions

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

1. What is the purpose of having an individual handler function for each exception/interrupt? (i.e., if all exceptions/interrupts were delivered to the same handler, what feature that exists in the current implementation could not be provided?)
2. Did you have to do anything to make the `user/softint` program behave correctly? The grade script expects it to produce a general protection fault (trap 13), but `softint`'s code says `int $14`. *Why* should this produce interrupt vector 13? What happens if the kernel actually allows `softint`'s `int $14` instruction to invoke the kernel's page fault handler (which is interrupt vector 14)?

- 由于中断号需要手动保存，而入口函数得不到中断号的信息，所以需要为每 个中断创建一个函数，压入自己的中断号。
- softint产生的是缺页中断，即14号中断，但是14中断的描述符权限是只有内核能够调用，所以CPU产生了一个13号中断，即权限错误。

  可以把14号中断的门描述符改成3，这样用户就能够触发这个中断了。

  但是这样会产生一个问题就是，就是用户触发这个中断是int $14是不会产生错误码的，而同时CPU也不会产生这里的错误码，所以，这样将得不到错误码，和int $14的handler函数的格式不一样，之后也会再次出错。

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

实现几种中断处理例程

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`.

很简单：

```
1     switch (tf->tf_trapno) {
2       case T_PGFLT:
3       page_fault_handler(tf);
4       return;
5     }
```

3.2 The Breakpoint Exception

类似与上面的

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.

```
1   case T_BRKPT:
2   monitor(tf);
3   return ;
```

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?

- 也是因为中断描述符的 dpl 位的设置问题，设置成 3 即用户可引发中断即可，以前已经讲过了。
- 这种机制重要的地方是中断描述符的信息，很关键。有softint的例子也可以看出，这种机制具有较健壮，更灵活。

3.3 System calls

用户需要内核做一些事情的时候，会产生一个系统调用。内核执行相应的代码去执行系统调用，然后继续用户进程。实现的细节每个系统都会有一点不一样。

JOS内核中，T_SYSCALL to 48 (0x30)，用int $0x30来实现系统调用，参数从%eax, %edx, %ecx, %ebx, %edi, %esi,中获取。

内核把返回值放到%eax中。触发系统调用的汇编代码已经实现了syscall() in lib/syscall.c。

在inc/syscall.h中可以找到支持的系统调用：

```
1   /* system call numbers */
2   enum {
3           SYS_cputs = 0,
4           SYS_cgetc,
5           SYS_getenvid,
6           SYS_env_destroy,
7           NSYSCALLS
8   };
```

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. Handle all the system calls listed in `inc/syscall.h` by invoking the corresponding kernel function for each call.

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.

kern/trapentry.S添加一个条目, kern/trap.c's trap_init()多加一个条目，很简单，就注意下，dpl是3，并且这个不带错误码就行了。

trap_dispatch()需要修改一下，使他调用syscall()函数：

```
1    static void
2   trap_dispatch(struct Trapframe *tf)
3   {
4       // Handle processor exceptions.
5       // LAB 3: Your code here.
6       switch (tf->tf_trapno) {
7           case T_DEBUG:
8           case T_BRKPT:
9               monitor(tf);
10              break;
```

```
11          case T_PGFLT:
12              page_fault_handler(tf);
13              break;
14          case T_SYSCALL:
15              tf->tf_regs.reg_eax =
16                  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);
17              break;
18          default:
19              // Unexpected trap: The user process or the
                    kernel has a bug.
20              print_trapframe(tf);
21              if (tf->tf_cs == GD_KT)
22                  panic("unhandled trap in kernel");
23              else {
24                  env_destroy(curenv);
25                  return;
26              }
27      }
28  }
```

将寄存器的值作为syscall()的参数，然后返回的值存放到eax中。

最后去实现kern/syscall.c中的syscall()

参照lib/syscall.c 中内联汇编，知道参数是从前往后读的， inc/syscall.h 中支持的系统调用都要实现。

```
1   // Dispatches to the correct kernel function, passing the
        arguments.
2     int32_t
3   syscall(uint32_t syscallno, uint32_t a1, uint32_t a2,
        uint32_t a3, uint32_t a4, uint32_t a5)
4   {
5       // Call the function corresponding to the 'syscallno'
            parameter.
6       // Return any appropriate return value.
7       // LAB 3: Your code here.
8
9       //panic("syscall not implemented");
10
11      switch (syscallno) {
12          case SYS_cputs:
13              sys_cputs((const char*)a1, (size_t)a2);
14              return 0;
15          case SYS_cgetc:
16              return sys_cgetc();
17          case SYS_getenvid:
18              return sys_getenvid();
19          case SYS_env_destroy:
20              return sys_env_destroy((envid_t)a1);
21          default:
22              return -E_INVAL;
```

```
23          }
24      }
```

然后按照指导上跑了一下，确实输出了hello world但是page fault了。

Make grade 60/80。

3.4 User-mode startup

用户进程的启动，从lib/entry.S开始运行，一些初始化之后，跳转到lib/libmain.c中的libmain()函数，并且初始化全局的指针thisenv指向这个环境的Env。

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.

很简单，一句话：

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

按要求测试一下，符合。

3.5 Page faults and memory protection

用户可以传递指针给内核，但是内核不能随意解指针。所以有了缺页处理和内存保护机制，内核必须很小心的处理指针。

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.

如果发生在内核态，就panic。

```
1   // Handle kernel-mode page faults.
2   // LAB 3: Your code here.
3   if (!(tf->tf_cs & 3))
4   panic("a page fault happens in kernel mode\n");
```

接下来实现user_mem_check(),check一个进程是否被允许访问相应的内存。

```
1     int
2 user_mem_check(struct Env *env, const void *va, size_t len, int
       perm)
3 {
4     // LAB 3: Your code here.
5
6     ntptr_t va1 = (uintptr_t)va, va2 = va1 + len;
7     pte_t *pte;
8     for (; va1 < va2; va1 = ROUNDDOWN(va1 + PGSIZE, PGSIZE)) {
9         if (va1 >= ULIM) {
10            user_mem_check_addr = va1;
11            return -E_FAULT;
12        }
13        pte = pgdir_walk(env->env_pgdir, (void*)va1, 0);
14        if ((*pte & (perm | PTE_P)) != (perm | PTE_P)) {
15            user_mem_check_addr = va1;
16            return -E_FAULT;
17        }
18    }
19    return 0;
```

```
20 }
```

为syscall.c增加页面检查, 是sys_cputs()函数。

```
1       // LAB 3: Your code here.
2     if (curenv->env_tf.tf_cs & 3)
3         user_mem_assert(curenv, s, len, 0);
```

然后按要求 make run-buggyhello, 得到了和期望一样的输出。
最后改一下debuginfo_eip in kern/kdebug.c

```
1      // Make sure this memory is valid.
2         // Return -1 if it is not.  Hint: Call user_mem_check.
3         // LAB 3: Your code here.
4         if (user_mem_check(curenv, usd, sizeof(struct
              UserStabData), PTE_U) < 0)
5             return -1;
6
7         stabs = usd->stabs;
8         stab_end = usd->stab_end;
9         stabstr = usd->stabstr;
10        stabstr_end = usd->stabstr_end;
11
12        // Make sure the STABS and string table memory is valid
              .
13        // LAB 3: Your code here.
14        if (user_mem_check(curenv, stabs, stab_end - stabs,
              PTE_U) < 0) return -1;
15        if (user_mem_check(curenv, stabstr, stabstr_end -
              stabstr, PTE_U) < 0) return -1;
```

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 f010000c
[00001000] free env 00001000
```

检查，和指导中一样。

然后make run-breakpoint-nox

```
Stack backtrace:
ebp efffe30 eip f0100b3b args f0106b72 effffea8 efffe90 f0105ae9 0000000a
            kern/monitor.c:7: mon_backtrace+60
ebp effffea0 eip f010112a args 00000001 effffec0 f01d1000 0000000d f0108098
            kern/monitor.c:27: runcmd+306
ebp efffff10 eip f01011a8 args f018f569 f01d1000 efffff50 f010490a f01048c0
            kern/monitor.c:9: monitor+86
ebp efffff40 eip f0104ef5 args f01d1000 00000000 efffff80 f010492d f0107c10
            kern/trap.c:5: trap_dispatch+55
ebp efffff80 eip f010505e args f01d1000 effffffbc 80050033 f0152db4 f0152d00
            kern/trap.c:14: trap+193
ebp effffffb0 eip f010513c args f01d1000 00000000 00000000 eebfdfd0 effffffdc
            kern/syscall.c:3: sys_cputs+0
ebp eebfdfd0 eip 00800080 args 00000000 00000000 00000000 00000000 00000000
            lib/libmain.c:8: libmain+71
Incoming TRAP frame at 0xefffffdfc
kernel panic at kern/trap.c:232: a page fault happens in kernel mode

Welcome to the JOS kernel monitor!
```

这是backtrace输出的， 在这里ary[i]=*(ebp+j); 出现了page fault, 应该是不能访问，所以报错了，但是之中有一些syscall,我就不知道了。这算一个疑问。

4 Challenge 3.1 clean up similar code

上课说统一交，所以这里就不写了。

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

```
1   #define TRAPHANDLER(name, num)                  \
2   .text;                  \
3   .globl name;    /* define global symbol for 'name' */ \
4   .type name, @function;  /* symbol type is function */   \
5   .align 2;   /* align function definition */   \
6   name:     /* function starts here */    \
7   .if !(num == 8 || num >= 10 && num <= 14 || num == 17); \
8   pushl $0; \
9   .endif; \
10  pushl $(num);               \
11  jmp _alltraps;                \
12  .data;                  \
13  .long name;
```

这里将TRAPHANDLER_NOEC和TRAPHANDLER合并起来，判断一下他是否有错误码。

然后将之后的统一起来：

```
1   .data
2   .global vectors
```

```
3   vectors:
4   TRAPHANDLER(vector0, 0)
5   TRAPHANDLER(vector1, 1)
6   TRAPHANDLER(vector2, 2)
7   TRAPHANDLER(vector3, 3)
8   TRAPHANDLER(vector4, 4)
9   TRAPHANDLER(vector5, 5)
10  TRAPHANDLER(vector6, 6)
11  TRAPHANDLER(vector7, 7)
12  TRAPHANDLER(vector8, 8)
13  TRAPHANDLER(vector9, 9)
14  TRAPHANDLER(vector10, 10)
15  TRAPHANDLER(vector11, 11)
16  TRAPHANDLER(vector12, 12)
17  TRAPHANDLER(vector13, 13)
18  TRAPHANDLER(vector14, 14)
19  TRAPHANDLER(vector15, 15)
20  TRAPHANDLER(vector16, 16)
21  TRAPHANDLER(vector17, 17)
22  TRAPHANDLER(vector18, 18)
23  TRAPHANDLER(vector19, 19)
24  TRAPHANDLER(vector20, 20)
25  TRAPHANDLER(vector21, 21)
26  TRAPHANDLER(vector22, 22)
27  TRAPHANDLER(vector23, 23)
28  TRAPHANDLER(vector24, 24)
29  TRAPHANDLER(vector25, 25)
30  TRAPHANDLER(vector26, 26)
31  TRAPHANDLER(vector27, 27)
32  TRAPHANDLER(vector28, 28)
33  TRAPHANDLER(vector29, 29)
34  TRAPHANDLER(vector30, 30)
35  TRAPHANDLER(vector31, 31)
36  TRAPHANDLER(vector32, 32)
37  TRAPHANDLER(vector33, 33)
38  TRAPHANDLER(vector34, 34)
39  TRAPHANDLER(vector35, 35)
40  TRAPHANDLER(vector36, 36)
41  TRAPHANDLER(vector37, 37)
42  TRAPHANDLER(vector38, 38)
43  TRAPHANDLER(vector39, 39)
44  TRAPHANDLER(vector40, 40)
45  TRAPHANDLER(vector41, 41)
46  TRAPHANDLER(vector42, 42)
47  TRAPHANDLER(vector43, 43)
48  TRAPHANDLER(vector44, 44)
49  TRAPHANDLER(vector45, 45)
50  TRAPHANDLER(vector46, 46)
51  TRAPHANDLER(vector47, 47)
52  TRAPHANDLER(vector48, 48)
```

这个研究了一下，C语言中并不能循环写。 其余的不变，trap.c这样写

```
1     // LAB 3: Your code here.
```

```
2     for (i = 0; i <= 0x30; ++i) {
3         switch (i) {
4             case T_BRKPT:
5             case T_SYSCALL:
6                 SETGATE(idt[i], 0, GD_KT, vectors[i], 3);
7                 break;
8             default:
9                 SETGATE(idt[i], 0, GD_KT, vectors[i], 0);
10        }
11    }
```

5 Challenge 3.2 breakpoint and disassembler

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

这个通过修改tf中的eflags的陷入位FL_TF实现stepi和continue功能，使用env_pop_tf(tf)返回程序继续运行。

代码如下

kern/monitor.c

```
1   int
2   mon_continue(int argc, char **argv, struct Trapframe *tf)
3   {
4   if (tf->tf_trapno == T_BRKPT || tf->tf_trapno == T_DEBUG) {
5   tf->tf_eflags &= ~FL_TF;
6   env_pop_tf(tf);
7   }
8   return 0;
9   }
10
11  int
12  mon_stepins(int argc, char **argv, struct Trapframe *tf)
13  {
14  if (tf->tf_trapno == T_BRKPT || tf->tf_trapno == T_DEBUG) {
15  tf->tf_eflags |= FL_TF;
16  env_pop_tf(tf);
17  }
18  return 0;
19  }
```

添加commands[]

```
1     { "si", "single-step one instruction at a time", mon_stepins
          },
2     { "c", "continue", mon_continue },
```

kern/trap.c/trap_dispatch()中也加上：

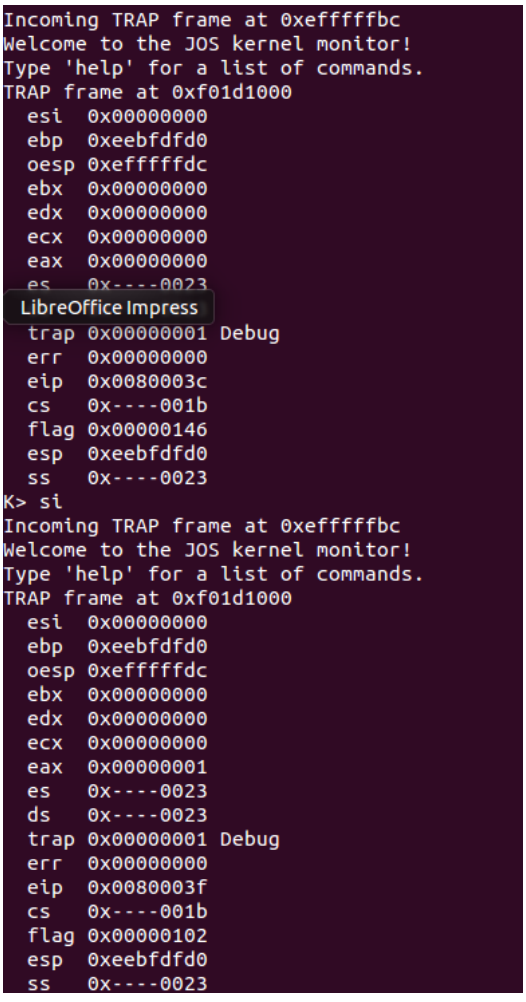
```
1       case T_DEBUG:
2          case T_BRKPT:
3              monitor(tf);
4              break;
```

这样就好了，测试一下：

在user/breakpoint.c进行测试，实测每次%eax会增加1，并且continue功能也正常。

```
1   asm volatile("movl $0, %eax");
2   asm volatile("addl $1, %eax");
3   asm volatile("addl $1, %eax");
4   asm volatile("addl $1, %eax");
5   asm volatile("addl $1, %eax");
6   asm volatile("addl $1, %eax");
```

这是成果图：



6 Complete

This completes the lab.

```
divzero: OK (1.3s)
softint: OK (0.9s)
badsegment: OK (1.0s)
Part A score: 30/30

faultread: OK (1.1s)
faultreadkernel: OK (1.1s)
faultwrite: OK (0.8s)
faultwritekernel: OK (1.0s)
breakpoint: OK (0.9s)
testbss: OK (1.0s)
hello: OK (1.0s)
buggyhello: OK (1.0s)
    (Old jos.out.buggyhello failure log removed)
buggyhello2: OK (1.0s)
    (Old jos.out.buggyhello2 failure log removed)
evilhello: OK (1.9s)
    (Old jos.out.evilhello failure log removed)
Part B score: 50/50

Score: 80/80
```