[TOC]

练习1

mem_init

与lab2初始化pages数组和映射类似

练习2

env_init

初始化全部 envs 数组中的 Env 结构体，并将它们加入到 env_free_list 中。还要调用 env_init_percpu ，这个函数会通过配置段硬件，将其分隔为特权等级 0 (内核) 和特权等级 3（用户）两个不同的段。 通过注释可以得知，第一次调用env_alloc()应该返回envs[0]，所以链表应该倒序存储。

envs

env.h中定义

```
extern struct Env *envs;                // All environments
```

env_setup_vm

为新的进程分配一个页目录，并初始化新进程的地址空间对应的内核部分。 所有envs结构的虚拟地址都是在UTOP上的。 UTOP之上env_pgdir可以拷贝kern_pgdir, 因为所有用户环境的页目录表中和操作系统相关的页目录项都是一样的，但是UVPT处的env_pgdir需要单独设置成用户只读。

> 每个用户进程都需要共享内核空间，所以对于用户进程而言，在UTOP以上的部分，和系统内核的空间是完全一样的。

memcpy

string.h中定义

```
void *  memcpy(void *dst, const void *src, size_t len);
```

memcpy函数的功能是从源src所指的内存地址的起始位置开始拷贝n个字节到目标dest所指的内存地址的起始位置中。

region_alloc

为进程分配和映射物理内存。 利用lab2中的 page_alloc() 完成内存页的分配， page_insert() 完成虚拟地址到物理页的映射。 注意把虚拟地址va，va+len以4k为单位取整，方便后面以页为单位计算。 还需要检查页表是

否申请成功，是否映射到对应的物理地址。

load_icode

将ELF二进制文件读入内存，为每一个用户进程设置它的初始代码区，堆栈以及处理器标识位。因为用户程序是ELF文件，所以要解析ELF文件。

> 函数只在内核初始化且第一个用户进程未运行时被调用，从ELF文件头部指明的虚拟地址开始加载需要加载的字段到用户内存

该段代码需要参考boot/main.c文件来写。 注释提示通过参考env_run和env_pop_tf，修改程序的入口，来确保进程正确开始执行。根据env_run的注释，需要回来修改代码给e->env_tf附上正确的值。

Proghdr

通过查询得知，p_type,p_va,p_memsz是结构Proghdr中的参数。 elf.h中定义

```
struct Proghdr {
        uint32_t p_type;
        uint32_t p_offset;
        uint32_t p_va;
        uint32_t p_pa;
        uint32_t p_filesz;
        uint32_t p_memsz;
        uint32_t p_flags;
        uint32_t p_align;
};
```

通过上网查询可知Proghdr中各个数据类型的意义 uint32_t p_type; // 段类型，说明载入的是代码还是数据，用于动态链接 uint32_t p_offset; // 段相对文件头的偏移值 uint32_t p_va; // 段的第一个字节将被放到内存中的虚拟地址 uint32_t p_pa; // 物理地址，没有使用 uint32_t p_filesz; // 文件中段的长度 uint memsz; // 段在内存中占用的空间(每个程序的大小) uint32_t p_flags; // 读/写/执行位 uint32_t p_align; // 需求队列，总是物理页大小

Elf

elf.h中定义

```
struct Elf {
        uint32_t e_magic;       // must equal ELF_MAGIC
        uint8_t e_elf[12];
        uint16_t e_type;
        uint16_t e_machine;
        uint32_t e_version;
        uint32_t e_entry;
        uint32_t e_phoff;
        uint32_t e_shoff;
        uint32_t e_flags;
        uint16_t e_ehsize;
        uint16_t e_phentsize;
```

```
        uint16_t e_phnum;
        uint16_t e_shentsize;
        uint16_t e_shnum;
        uint16_t e_shstrndx;
};
```

Elf结构具体每个类型的意义可参考下图：

```
/* The ELF file header.  This appears at the start of every ELF file.  */

#define EI_NIDENT (16)

typedef struct
{
  unsigned char e_ident[EI_NIDENT];     /* Magic number and other info */
  Elf32_Half    e_type;                 /* Object file type */
  Elf32_Half    e_machine;              /* Architecture */
  Elf32_Word    e_version;              /* Object file version */
  Elf32_Addr    e_entry;                /* Entry point virtual address */
  Elf32_Off     e_phoff;                /* Program header table file offset */
  Elf32_Off     e_shoff;                /* Section header table file offset */
  Elf32_Word    e_flags;                /* Processor-specific flags */
  Elf32_Half    e_ehsize;               /* ELF header size in bytes */
  Elf32_Half    e_phentsize;            /* Program header table entry size */
  Elf32_Half    e_phnum;                /* Program header table entry count */
  Elf32_Half    e_shentsize;            /* Section header table entry size */
  Elf32_Half    e_shnum;                /* Section header table entry count */
  Elf32_Half    e_shstrndx;             /* Section header string table index */
} Elf32_Ehdr;
```

以及博客： https://www.cnblogs.com/dengxiaojun/p/4279407.html

具体ELF文件格式可参考下面这篇博客： https://blog.csdn.net/fang92/article/details/48092165

Trapframe

trap.h中定义

```
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;
        uint16_t tf_cs;
        uint16_t tf_padding3;
        uint32_t tf_eflags;
        /* below here only when crossing rings, such as from user to kernel
*/
```

```
        uintptr_t tf_esp;
        uint16_t tf_ss;
        uint16_t tf_padding4;
} __attribute__((packed));
```

env_create

调用env_alloc，从env_free_list中取出一个env结构体，再通过 env_setup_vm为其初始化，申请新的页目录;然后执行load_icode,这个函数加载elf文件(二进制文件)，它会调用region_alloc为其分配页，并将虚拟地址和物理地址作出映射，load_icon之后分配进程栈，以及，将env->env_tf.tf_eip指向将执行进程函数的入口(等待env_pop_tf的调用)

env_run

启动进程，curenv结构体指向当前运行的进程env,改变curenv结构体中运行状态等信息,通过env_pop_tf函数，将env结构体中保存的寄存器中的信息加在到真正的寄存器中。

curenv

env.h中定义

```
extern struct Env *curenv;              // Current environment
```

env_pop_tf

使用'iret'指令复原Trapframe中的寄存器值，退出内核，开始运行一些进程的代码。 ENV结构中提到Trapframe结构的寄存器 `struct Trapframe env_tf; // 保存的寄存器` env.h中声明

```
void    env_pop_tf(struct Trapframe *tf) __attribute__((noreturn));
```

env.c中定义

```
// Restores the register values in the Trapframe with the 'iret'
instruction.
// This exits the kernel and starts executing some environment's code.
//
// This function does not return.
//
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 */
}
```

lcr3

x86.h中定义

```
static __inline void
lcr3(uint32_t val)
{
        __asm __volatile("movl %0,%%cr3" : : "r" (val));
}
```

汇编代码，将地址装入cr3寄存器，而cr3中装的都是页目录的起始地址。

debug

memmove error

运行gdb的时候发现报了memmove错误 然后按照错误提示去lib/string.c中查找memmove，发现自己原本写的memcpy在这里的string.c中竟然直接调用memmove,就顺手查了关于这两个函数的区别

```
void *
memmove(void *dst, const void *src, size_t n)
{
        const char *s;
        char *d;

        s = src;
        d = dst;
        if (s < d && s + n > d) {
                s += n;
                d += n;
                while (n-- > 0)
                        *--d = *--s;
        } else
                while (n-- > 0)
                        *d++ = *s++;

        return dst;
}

void *
memcpy(void *dst, const void *src, size_t n)
{
```

```
        return memmove(dst, src, n);
}
```

具体这两个函数的区分可以参考下面的博客，解析的十分详细
https://blog.csdn.net/li_ning_/article/details/51418400

发现报错的memmove是发生在调用汇编代码的memmove时

```
void *
memmove(void *dst, const void *src, size_t n)
{
        const char *s;
        char *d;

        s = src;
        d = dst;
        if (s < d && s + n > d) {
                s += n;
                d += n;
                if ((int)s%4 == 0 && (int)d%4 == 0 && n%4 == 0)
                        asm volatile("std; rep movsl\n"
                                :: "D" (d-4), "S" (s-4), "c" (n/4) : "cc",
"memory");
                else
                        asm volatile("std; rep movsb\n"
                                :: "D" (d-1), "S" (s-1), "c" (n) : "cc",
"memory");
                // Some versions of GCC rely on DF being clear
                asm volatile("cld" ::: "cc");
        } else {
                if ((int)s%4 == 0 && (int)d%4 == 0 && n%4 == 0)
                        asm volatile("cld; rep movsl\n"
                                :: "D" (d), "S" (s), "c" (n/4) : "cc",
"memory");
                else
                        asm volatile("cld; rep movsb\n"
                                :: "D" (d), "S" (s), "c" (n) : "cc",
"memory");
        }
        return dst;
}
```

后面发现是前面region_alloc中对页表的分配出错了，导致后面内存异常。

PADDR called with invalid kva 00000000

改完上面的错误，然后出现了如下图的错误 找到半天panic出处最后发现是PADDR出问题了，看PADDR源码可知是因为转换的地址小于KERNBASE，也就是说不是在内核所映射的空间中。

```
#define PADDR(kva) _paddr(__FILE__, __LINE__, kva)

static inline physaddr_t
_paddr(const char *file, int line, void *kva)
{
        if ((uint32_t)kva < KERNBASE)
                _panic(file, line, "PADDR called with invalid kva %08lx",
kva);
        return (physaddr_t)kva - KERNBASE;
}
```

= =最后发现是env_init出错了。。

some

start (kern/entry.S) 由实模式进入保护模式 i386_init (kern/init.c) 初始化内核 cons_init(); 初始化显示屏 mem_init(); 内存管理初始化 env_init(); 进程初始化 trap_init(); 中断初始化

gbd调试

通过观察指令地址发现指令iret后进入了实模式 在obj/user/hello.asm中查找int $0x30的地址 在int $0x30的地址0x800a1c处设置断点，成功运行到该代码处说明前面写的代码没有出错

练习4

trapentry.S

.global/.globl:用来定义一个全局的符号 使用.global/.globl将函数声明为全局函数以后就可以被其他文件调用。 .type:用来指定一个符号的类型是函数类型或者是对象类型,对象类型一般是数据 .align:用来指定内存对齐方式

在CPU返回error code的时候调用TRAPHANDLER，没有则调用TRAPHANDLER_NOEC

关于内联汇编的部分参考了博客https://blog.csdn.net/slvher/article/details/8864996

first

关于CPU何时返回error code，可参考Intel手册
https://pdos.csail.mit.edu/6.828/2014/readings/i386/s09_10.htm

> 手册中没有说明是否返回error code的中断暂时没加上

second

根据提示可知就是将Trapframe结构中的所有数据压入栈中，文档告诉我们 tf_ss,tf_esp,tf_eflags,tf_cs,tf_eip,tf_err在中断发生时由CPU压栈，所以只需要将剩下的寄存器tf_es,tf_ds压入栈中即可。因为需要将栈模仿成Trapframe结构，而且CPU已经将其余寄存器的值倒序压入，所以需要注意先压入ds，后压入es。

pushal

pushal指令会按顺序将eax到edi压入栈中,具体参考下面网站
http://www.fermimn.gov.it/linux/quarta/x86/pusha.htm

GD_KD

memlayout.h中定义 GD_KD内核数据段的偏移量

```
/*
 * This file contains definitions for memory management in our OS,
 * which are relevant to both the kernel and user-mode software.
 */

// Global descriptor numbers
#define GD_KT     0x08     // kernel text
#define GD_KD     0x10     // kernel data
#define GD_UT     0x18     // user text
#define GD_UD     0x20     // user data
#define GD_TSS0   0x28     // Task segment selector for CPU 0
```

需要注意的是内联汇编需要通过使用 $ 指定操作立即数

> 这里需要注意不能直接用立即数给段寄存器赋值，至于为什么还是不太懂，可以参考下面的讨论 https://bbs.csdn.net/topics/340215235 所以应该先给通用寄存器赋值，或者入栈出栈来实现赋值。

trap_init

根据trapentery.S中的提示，声明函数，同时也使用SETGATE设置IDT。

设置IDT

在kern/trap.h中找到Gatedesc结构的idt数组

```
/* The kernel's interrupt descriptor table */
extern struct Gatedesc idt[];
extern struct Pseudodesc idt_pd;
```

Gatedesc

mmu.h中定义

```
// Gate descriptors for interrupts and traps
struct Gatedesc {
        unsigned gd_off_15_0 : 16;   // low 16 bits of offset in segment
        unsigned gd_sel : 16;        // segment selector
        unsigned gd_args : 5;        // # args, 0 for interrupt/trap gates
```

```
        unsigned gd_rsv1 : 3;        // reserved(should be zero I guess)
        unsigned gd_type : 4;        // type(STS_{TG,IG32,TG32})
        unsigned gd_s : 1;           // must be 0 (system)
        unsigned gd_dpl : 2;         // descriptor(meaning new) privilege
level
        unsigned gd_p : 1;           // Present
        unsigned gd_off_31_16 : 16;  // high bits of offset in segment
};
```

其中涉及到位域的概念可参考下面博客https://blog.csdn.net/ai2000ai/article/details/56489667

SETGATE

mmu.h中定义

```
#define SETGATE(gate, istrap, sel, off, dpl)                    \
{                                                               \
        (gate).gd_off_15_0 = (uint32_t) (off) & 0xffff;         \
        (gate).gd_sel = (sel);                                  \
        (gate).gd_args = 0;                                     \
        (gate).gd_rsv1 = 0;                                     \
        (gate).gd_type = (istrap) ? STS_TG32 : STS_IG32;        \
        (gate).gd_s = 0;                                        \
        (gate).gd_dpl = (dpl);                                  \
        (gate).gd_p = 1;                                        \
        (gate).gd_off_31_16 = (uint32_t) (off) >> 16;           \
}
```

根据注释可知 SETGATE是用来设置正常的中断/陷阱向量表的。

- istrap：1表示陷阱（异常），0表示中断。 中断和陷阱影响中断使能标志(IF)。 通过向量的中断复位IF，从而防止其他中断干扰当前的中断处理程序。随后通过IRET指令将IF恢复为堆栈上EFLAGS映像中的值。通过陷阱的中断不会改变IF。
- sel：中断/陷阱处理程序的代码段选择器
- off：中断/陷阱处理程序的代码段偏移量
- dpl：Descriptor Privilege Level 描述符权限级别 软件调用所需的权限级别。 中断/陷阱显式地使用int指令。

> 不太懂为什么off直接写上函数名,对istrap和dpl的设置不太确定 可能是函数名对应着函数的地址，可以作为相对于段的偏移?

问题

1

> 对每一个中断/异常都分别给出中断处理函数的目的是什么？换句话说，如果所有的中断都交给同一个中断处理函数处理，现在我们实现的哪些功能就没办法实现了？

因为不同的中断/异常之间所需要的处理并不完全一样，例如中断/异常结束之后是需要从中断/异常继续执行，还是跳过这条指令，或者是其他的处理方式。而且不同的中断/异常需要的权限也是不一样，也不全都会返回error code。全使用一个中断处理函数还需要用switch或者if来进行分类处理。 如果都交给一个中断处理函数处理，函数写得合理应该还能实现中断处理。

2

> 你有没有额外做什么事情让 user/softint 这个程序按预期运行？打分脚本希望它产生一个一般保护错(陷阱 13)，可是 softint 的代码却发送的是 int $14。为什么 这个产生了中断向量 13？ 如果内核允许 softint 的 int $14 指令去调用内核中断向量 14 所对应的的缺页处理函数，会发生什么?

首先查看user/softint代码

```
void
umain(int argc, char **argv)
{
        asm volatile("int $14");        // page fault
}
```

grade-lab3代码

```
@test(10)
def test_softint():
    r.user_test("softint")
    r.match('Welcome to the JOS kernel monitor!',
            'Incoming TRAP frame at 0xeffffbc',
            'TRAP frame at 0xf.......',
            '  trap 0x0000000d General Protection',
            '  eip  0x008.....',
            '  ss   0x----0023',
            '.00001000. free env 0000100')
```

因为当前系统运行在用户态模式下，权限级别为3，而INT指令是系统指令，权限级别为0，因此会首先引发Gerneral Protection Fault，根据前面可知define T_GPFLT 13。

练习5

trap_dispatch

根据trap number对不同的中断/异常进行处理分配

page_fault_handler

trap.h中定义

```
void page_fault_handler(struct Trapframe *);
```

练习6

make grade失败，发现 `MISSING ' trap 0x00000003 Breakpoint'` 回头把break_point的权限修改为3就解决了

monitor

monitor.h中定义

```
// Activate the kernel monitor,
// optionally providing a trap frame indicating the current state
// (NULL if none).
void monitor(struct Trapframe *tf);
```

问题

1

> 断点那个测试样例可能会生成一个断点异常，或者生成一个一般保护错，这取决你是怎样在 IDT 中初始化它的入口的（换句话说，你是怎样在 trap_init 中调用 SETGATE 方法的）。为什么？你应该做什么才能让断点异常像上面所说的那样工作？怎样的错误配置会导致一般保护错？

在trap_init中调用SETGATE方法，是对中断/异常向量表进行设置。 在 trap_init 中设置好中断的入口，并且设置中断的权限，使得用户进程或者系统内核能够产生这个中断。 权限不够的时候会导致一般保护错误，比如运行在用户模式下，想要调用系统中断的时候。

2

> 你认为这样的机制意义是什么？尤其要想想测试程序 user/softint 的所作所为

通过不同的权限设置来保护操作系统，避免用户程序能通过调用系统中断来进行内核空间。

练习7

模仿前面中断/异常调用，在trapentry.S和trap.c中为syscall添加调用。 需要注意区分kern和lib中的syscall，在kern/trap.c中调用的实际上就是kern/syscall.c的syscall函数，而不是lib/syscall.c中的函数。

> lib里应该都是用户所需要的依赖文件？

syscall

lib/syscall.c中定义

```
static inline int32_t
syscall(int num, int check, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t
a4, uint32_t a5)
{
        int32_t ret;

        // Generic system call: pass system call number in AX,
        // up to five parameters in DX, CX, BX, DI, SI.
        // Interrupt kernel with T_SYSCALL.
        //
        // The "volatile" tells the assembler not to optimize
        // this instruction away just because we don't use the
        // return value.
        //
        // The last clause tells the assembler that this can
        // potentially change the condition codes and arbitrary
        // memory locations.

        asm volatile("int %1\n"
                : "=a" (ret)
                : "i" (T_SYSCALL),
                  "a" (num),
                  "d" (a1),
                  "c" (a2),
                  "b" (a3),
                  "D" (a4),
                  "S" (a5)
                : "cc", "memory");

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

        return ret;
}
```

根据注释可知 这段代码是通用系统调用，通过返回AX来传递系统调用号，最多DX，CX，BX，DI，SI五个参数。 用T_SYSCALL中断内核。 “volatile”告诉汇编器不优化该指令。 最后一个子句告诉汇编器可能会改变条件代码和任意内存的位置。

GCC内联汇编

GCC内联汇编语法固定为：

```
 asm [ volatile ] (
         assembler template
         [ : output operands ] /* optional */
         [ : input operands  ] /* optional */
         [ : list of clobbered registers ] /* optional */
         );
```

关于内联汇编的部分参考了博客https://blog.csdn.net/slvher/article/details/8864996

system call numbers

inc/syscall.h中声明

```
/* system call numbers */
enum {
        SYS_cputs = 0,
        SYS_cgetc,
        SYS_getenvid,
        SYS_env_destroy,
        NSYSCALLS
};
```

lib/syscall.c中定义

```
void
sys_cputs(const char *s, size_t len)
{
        syscall(SYS_cputs, 0, (uint32_t)s, len, 0, 0, 0);
}

int
sys_cgetc(void)
{
        return syscall(SYS_cgetc, 0, 0, 0, 0, 0, 0);
}

int
sys_env_destroy(envid_t envid)
{
        return syscall(SYS_env_destroy, 1, envid, 0, 0, 0, 0);
}

envid_t
sys_getenvid(void)
{
         return syscall(SYS_getenvid, 0, 0, 0, 0, 0, 0);
}
```

JOS系统调用的步骤

1.用户进程使用 inc/ 目录下暴露的接口 2.lib/syscall.c 中的函数将系统调用号及必要参数传给寄存器，并引起一次 int $0x30 中断 3.kern/trap.c 捕捉到这个中断，并将 TrapFrame 记录的寄存器状态作为参数，调用处理中断的函数 4.kern/syscall.c 处理中断

练习8

libmain

使用sys_getenvid，初始化thisenv全局指针，使它指向envs[]中的当前Env结构

```
const volatile struct Env *thisenv;
```

ENVX

inc/env.h中定义

```
#define LOG2NENV                10
#define NENV                    (1 << LOG2NENV)
#define ENVX(envid)             ((envid) & (NENV - 1))
```

结合之前的注释可知，envid_t有三部分，最后10位是Environment Index(eid)，使用ENVX(envid)可以截取后10位的值。

练习9

修改page_fault_handler

参考已有的其他代码，例如

```
        if ((tf->tf_cs & 3) == 3) {
                // Trapped from user mode.
```

得知tf_cs等于3时是用户态，等于0时是内核态。

user_mem_check

该函数主要是检查对应的用户进程是否能访问对应的内存空间 利用lab2中完成的pgdir_walk函数，填入进程的pgdir地址，返回va虚拟地址对应的页表项。 根据注释可知如果地址低于ULIM，并且页表给予了权限，则用户程序可以访问虚拟地址。

> 需要注意的是，因为以页为单位来考虑问题，所以需要对地址取整 !就算出错的是va之前的地址，需要返回的也应该是va的地址

修改 kern/syscall.c

验证系统调用的参数 发现kern/syscall.c中系统调用函数里只有sys_cputs()参数中存在指针，所以对其进行检测。

修改kdebug.c

对 usd, stabs, stabstr 都要调用 user_mem_check

补充

> 依旧忘记在哪个博客看到的了

before

由于每个用户进程都需要共享内核空间，所以对于用户进程而言，在UTOP以上的部分，和系统内核的空间是完全一样的。因此在pgdir开始设置的时候，只需要在一级页表目录上，把共享部分的一级页表目录部分复制进用户进程的地址空间就可以了，这样，就实现了页面的共享。因为一级页目录里面存储的是二级页表目录的物理地址，其直接映射到物理内存部分，而共享的内核部分的二级页目录在前期的内核操作中，已经完成了映射，所以二级页目录是不需要初始化的。简单来说，不需要映射二级页表的原因是，用户进程可以和内核共用这些二级页表。

load_icode()

作用是将 ELF 二进制文件读入内存，由于 JOS 暂时还没有自己的文件系统，实际就是从 *binary 这个内存地址读取。 大概需要做的事：

```
根据 ELF header 得出 Programm header。
遍历所有 Programm header，分配好内存，加载类型为 ELF_PROG_LOAD 的段。
分配用户栈。
```

需要思考的问题：

```
怎么切换页目录?
lcr3([页目录物理地址]) 将地址加载到 cr3 寄存器。
怎么更改函数入口?
将 env->env_tf.tf_eip 设置为 elf->e_entry，等待之后的 env_pop_tf() 调用。
```

memset((void *)ph[i].p_va, 0, ph[i].p_memsz); //因为这里需要访问刚分配的内存，所以之前需要切换页目录
memcpy((void *)ph[i].p_va, binary + ph[i].p_offset, ph[i].p_filesz); //应该有如下关系：ph->p_filesz <= ph->p_memsz。搜索BSS段

trapentry.S

首先看一下 trapentry.S 文件，里面定义了两个宏定义，TRAPHANDLER，TRAPHANDLER_NOEC。他们的功能从汇编代码中可以看出：声明了一个全局符号name，并且这个符号是函数类型的，代表它是一个中断处理函数名。其实这里就是两个宏定义的函数。这两个函数就是当系统检测到一个中断/异常时，需要首先完成的一部分操作，包括：中断异常码，中断错误码(error code)。正是因为有些中断有中断错误码，有些没有，所以我们采用利用两个宏定义函数。

然后就会调用 _alltraps，_alltraps函数其实就是为了能够让程序在之后调用trap.c中的trap函数时，能够正确的访问到输入的参数，即Trapframe指针类型的输入参数tf。

所以在trapentry.S中，我们要根据这个中断是否有中断错误码，来选择调用TRAPHANDLER，还是TRAPHANDLER_NOEC，然后再统一调用_alltraps，其实目的就是为了能够让系统在正式运行中断处理程序之前完成必要的准备工作，比如保存现场等等。

练习3

需要加载到代码段寄存器(CS)的值，其中最低两位表示优先级（这也是为什么说可以寻址 2^46 的空间而不是2^48)。 在JOS 中，所有的异常都在内核模式处理，优先级为0 (用户模式为3)。

练习7

现在回顾一下系统调用的完成流程：以user/hello.c为例，其中调用了cprintf()，注意这是lib/print.c中的cprintf，该cprintf()最终会调用lib/syscall.c中的sys_cputs()，sys_cputs()又会调用lib/syscall.c中的syscall()，该函数将系统调用号放入%eax寄存器，五个参数依次放入in DX, CX, BX, DI, SI，然后执行指令int 0x30，发生中断后，去IDT中查找中断处理函数，最终会走到kern/trap.c的trap_dispatch()中，我们根据中断号0x30，又会调用kern/syscall.c中的syscall()函数（注意这时候我们已经进入了内核模式CPL=0），在该函数中根据系统调用号调用kern/print.c中的cprintf()函数，该函数最终调用kern/console.c中的cputchar()将字符串打印到控制台。当trap_dispatch()返回后，trap()会调用env_run(curenv);，该函数前面讲过，会将curenv->env_tf结构中保存的寄存器快照重新恢复到寄存器中，这样又会回到用户程序系统调用之后的那条指令运行，只是这时候已经执行了系统调用并且寄存器eax中保存着系统调用的返回值。任务完成重新回到用户模式CPL=3。