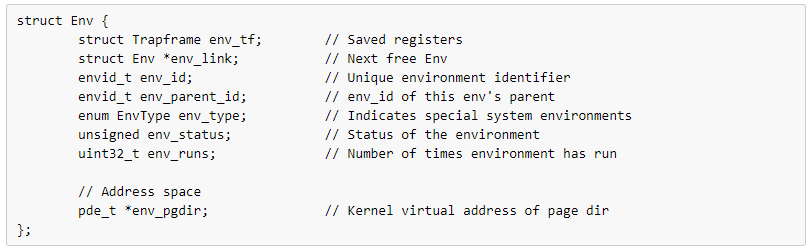
# Lab3

**Part A: User Environments and Exception Handling**

struct Env \*envs = NULL; // All environments

struct Env \*curenv = NULL; // The current env

static struct Env \*env\_free\_list; // Free environment list



Exercise1:

为envs声明NENV个env所需要的空间，然后映射到内存结构中UENVS的位置上。

### **Creating and Running Environments**

Exercise2:

Env\_init()遍历envs，将其中每一项的env\_id设置为0，env\_link设置为下一项，最后一项设置为NULL。然后根据要求，将env\_free\_list指向envs的第一项。

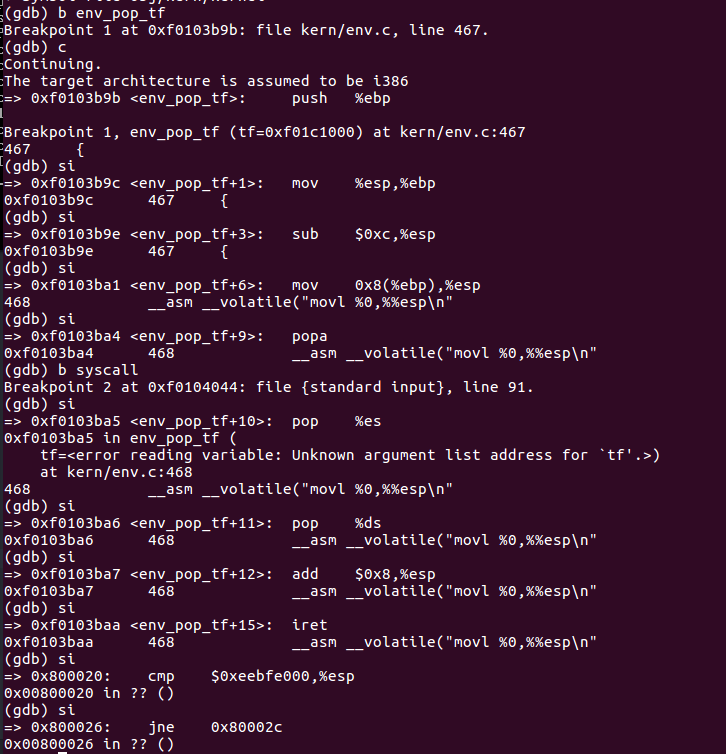
Env\_setup\_vm()申请了一个page directory，赋给环境e的env\_pgdir。为了使env\_free能够正常工作，将该页的pp\_ref加1。然后将UTOP以上的pgdir复制到e的env\_pgdir，最后设置UVPT的PDX对应的PDE。

Region\_alloc()申请了环境需要的物理空间。首先将输入的虚拟地址段开始地址round down，将结束地址round up，然后申请相应大小的页，每一页不做初始化，如果申请失败就panic，否则就把这一夜映射到对应的PDE，标记位可以被用户写。

Load\_icode()加载一个ELF文件，将内容加载到新的用户地址空间中。首先根据boot/main.c中的代码，获得ph和eph，唯一的区别是加载elf文件不需要从磁盘中读取文件。修改cr3，使用新的用户的地址空间。然后遍历所有的ph，只处理其中type为ELF\_PROG\_LOAD的ph。首先申请一块从ph->p\_va开始，大小为ph\_memsz的空间，全部置为零，然后将ph从binary + ph->p\_offset处开始，大小为ph\_filesz大小的数据复制到va处。复制完成后，将cr3修改回kernel，并为新的用户栈申请初始的一页空间，从USTACKTOP - PGSIZE开始。在最后将e的trap frame中的eip设置为elf中的entry。

Env\_create()首先调用env\_alloc创建一个新的env，处理返回值后，将他的type设置为输入值，将parent id设置为0，然后加载指定的elf文件。

Env\_run()按照注释，如果需要进程切换的话，首先将存在的并且running的进程的状态设置为runnable，然后修改当前进程为新的进程，其状态改为running，计数器增加1，然后修改cr3到相应的pgdir。然后调用env\_pop\_tf储存进程的寄存器，进入user mode。



### **Handling Interrupts and Exceptions**

kernel to recover control of the processor from user-mode code

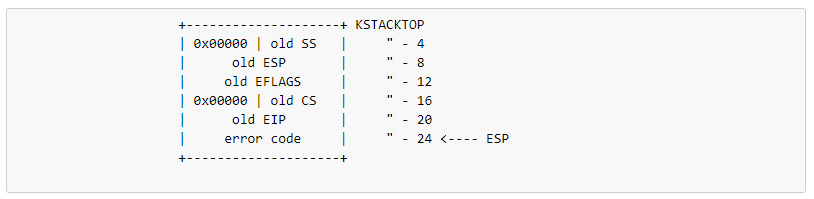
### **Basics of Protected Control Transfer**

the processor ensures that the kernel can be entered only under carefully controlled conditions

The Interrupt Descriptor Table:eip + cs(privilege level,x86 -> 0)

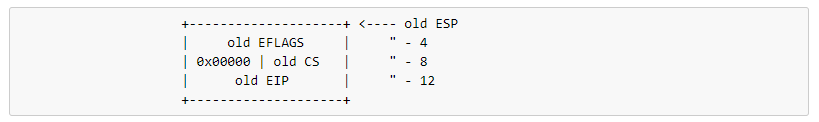
The Task State Segment: For this reason (protect user code in unprivileged level), when an x86 processor takes an interrupt or trap that causes a privilege level change from user to kernel mode, it also switches to a stack in the kernel's memory. A structure called the task state segment (TSS) specifies the segment selector and address where this stack lives. The processor pushes (on this new stack) SS, ESP, EFLAGS, CS, EIP, and an optional error code. Then it loads the CS and EIP from the interrupt descriptor, and sets the ESP and SS to refer to the new stack.

### **Types of Exceptions and Interrupts**

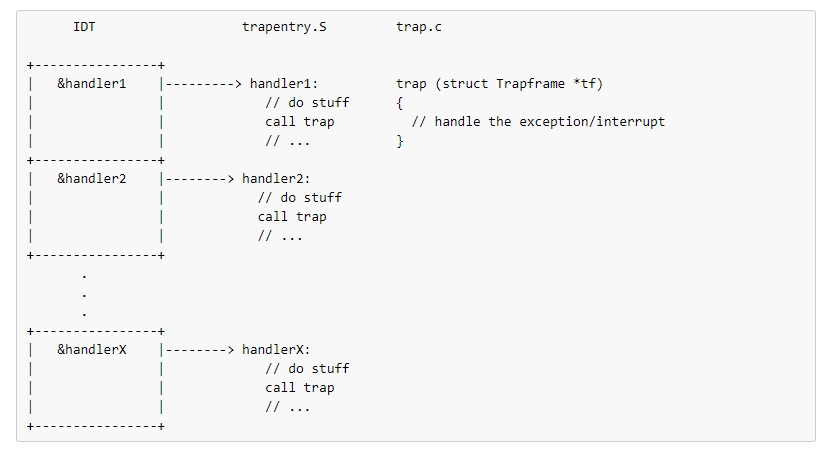


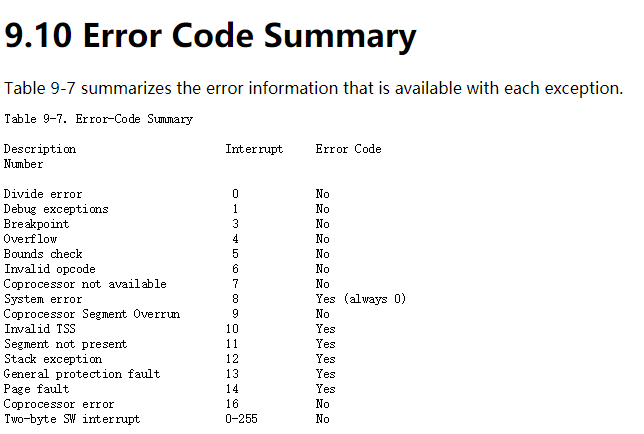
### **Nested Exceptions and Interrupts**

If the processor is already in kernel mode when the interrupt or exception occurs (the low 2 bits of the CS register are already zero), then the CPU just pushes more values on the same kernel stack.



### **Setting Up the IDT**





Exercise4

在trapentry.S中，为在trap.h中的每一个宏增加了entry point。其中，产生error code的8、10-14号trap使用宏TRAPHANDLER，其余使用TRAPHANDLER\_NOEC。此外，宏内还保存了IDT的索引和对应的entry point。

然后实现了\_alltraps，按照文档，通过push和pushal构建trapframe，加载GD\_KD到%es、%ds中，push %esp，call trap。（TSS由硬件push， trapno由handler的宏push）

在trap\_init中，初始化了IDT。

出现中断后，硬件首先push TSS，然后搜索IDT，得到对应的enter point，调用handler宏，push对应的num。之后调用alltraps，补足剩下的trapframe。最后调用trap，找到指定的处理函数。

Challenge:

在trapentry.S中，在.data之后建立一个全局数组，保存每一个interrupt需要的entry、IDT索引、优先级（0或3）。然后在trap.c中，遍历这个数组，使用其中储存的数据生成IDT表，缩短代码。

Questions：

1. 每个exception/interrupt的处理方式都是不同的。有的会产生error code，而有的不会，他们的trapframe是不同的。因此trapframe会很难控制。此外，有的中断处理结束后会返回用户进程，而有的会直接停止用户进程。但是我认为，以上的问题都可以通过一个handler内部不同的分支判断解决。
2. Softint在用户态下产生了page fault的中断。由于page fault的特权级是0，当前状态不具有权限，因此不会触发相应的page fault的handler。而是产生了general potection fault（13）。如果用户态可以调用系统级的函数，则整个系统会变得非常不安全，危险的用户代码可以更轻易地进行buffer overflow攻击，访问与修改系统内部代码与数据。

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

### **Handling Page Faults**

Exercise5

在trap.c的trap\_dispatch中判断trapframe的trap number，如果是page fault，就运行相应的handler。目前只是用了handler的已实现部分就可以通过测试了。

### **System calls**

Exercise6

首先在trapentry.S中增加sysenter\_handler，按照文档中的register layout，将相应的寄存器push。然后调用syscall，结束后返回。之后实现kern/syscall.c，根据syscall的syscallno调用已实现的syscall。

然后再kern/init.c中设置MSRs。首先在参考资料中找到wrmsr的实现，将他加入到inc/x86.h。然后查看enable\_sep\_cpu的实现，用kernel text段、kernel栈顶、handler位置（即eip）配置MSRs。

最后修改lib/syscall.c，支持sysenter。按照文档，使用“leal after\_sysenter\_label, %%esi”把返回地址保存到%esi中，然后调用sysenter。

### **User-mode startup**

Exercise7

修改libmain（），初始化全局指针thisenv。使用sys\_getenvid()获得当前env的id，然后调用ENVX，用索引在envs中找到对应的env。

Exercise8

首先修改Env的结构，记录当前程序的heap底。然后在load\_icode初始化用户进程的时候，对该值也进行初始化。

之后在内核实现system call - sbrk。首先声明一块从当前环境的heap底开始的空间。因为heap是由大地址向小地址增长，而申请空间要从小地址到大地址，因此申请的空间实际上是以当前的heap底为结束的。之后将heap底的位置更新，并返回。（根据文档，JOS的sbrk返回的是增长之后的地址空间的heap底）

### **The Breakpoint Exception**

Exercise9

首先修改trap\_dispatch，增加T\_BRKPT的处理，调用monitor，参数为trapframe。

然后修改monitor代码，使JOS支持c，ci，x的指令。

c指令从kernel返回user mode继续执行，只需要返回就可以。注意到monitor跳出循环的条件是返回负值，因此返回-1。如果没有用户进程，trapframe为NULL，需要另外处理。由于接下来的si指令可能设置了FL\_TF，需要在这里把他置回0.

指令si逐指令执行用户代码。首先修改eflags，把FL\_TF置为1.然后调用debuginfo\_eip，并按照规定格式输出返回结果。

指令x输出指定地址的4B的数据，使用汇编代码，将指定地址出的数据mov到输出，然后按照格式进行print。

由于首次执行gdb相关操作使用用户程序产生T\_BRKPT执行，可以正常运作。之后如果运行了si，则会由kernel产生trap，发现产生的trap为T\_DEBUG，于是在dispatch中加入相应的处理，即调用monitor。

Question3

该测试正确情况下会由用户产生T\_BRKPT，然后执行相应的处理函数。这是因为在设置IDT的时候，将该trap的权限设置为了3。如果错误地设置为0，那么当用户需要设置断点的时候，就不会触发T\_BRKPT，因为用户不具有权限，而会触发T\_GPFLT了。

Question4

以上的机制防止用户恶意地产生trap，但是breakpoint本身为程序员提供了便利，且只涉及暂停、读内存等有限的功能，因此是安全的，可以允许用户触发。

而softint触发了page fault。如果用户拥有权限触发page fault的话，会干扰kernel对于内存的管理，是有害的。因此通过权限设置，禁止了用户的这种行为。

### **Page faults and memory protection**

Exercise10

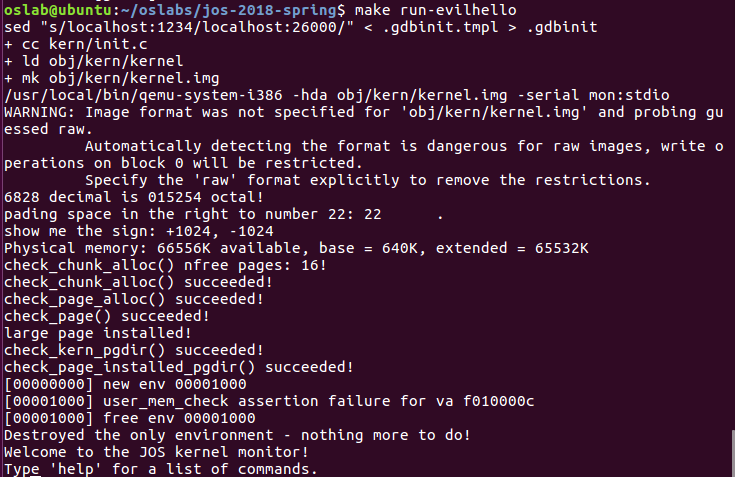
首先在page fault handler中检查cs中的低位，如果发现page fault是内核引起的，则panic。这是因为截止到目前的实现，内核的地址空间都是按照规则映射好的，正确运行是不会由page fault的。如果出现了page fault，只能说明代码存在bug。

然后实现user\_mem\_check，遍历所给的地址空间覆盖的页，为每一页检查地址是否小于ULIM，以及页表中是否有对应的权限。如果出错，就修改user\_check\_num，并且返回-E\_FAULT。根据测试代码，虽然需要进行round检查每个整页，但是第一页的错误输出信息需要从实际开始的va输出，因此需要进行特殊判断。

在sys\_cputs中检查用户是否拥有读取s开始，长度为len的地址的权限，调用user\_mem\_assert。

然后修改kdebug.c，在usd，stabs，stabstr中加入地址检查，user\_mem\_check。

Exercise11



Exercise12

根据注释修改evilhello2.c。首先把GTD descriptor利用sgtd储存在内存中，然后使用sys\_map\_kernel\_page映射到用户空间。之后将需要执行的函数（fun\_ptr、leave、lret)封装在一个函数中，作为call gate的参数插入GDT。执行lcall。执行后利用之前存储好的备份恢复GDT。