LAB3 用户进程

练习1：

首先要在page\_init()之前为envs分配内存空间。

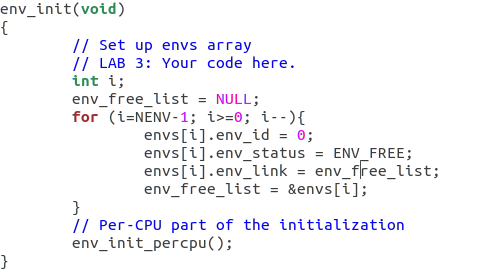


然后要在页表中设置它的映射关系，位于check\_page()函数之后

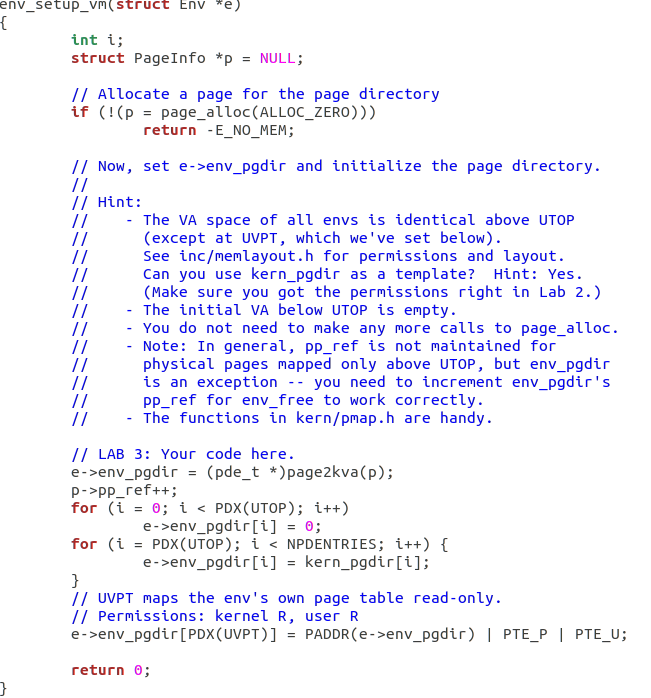


练习2：

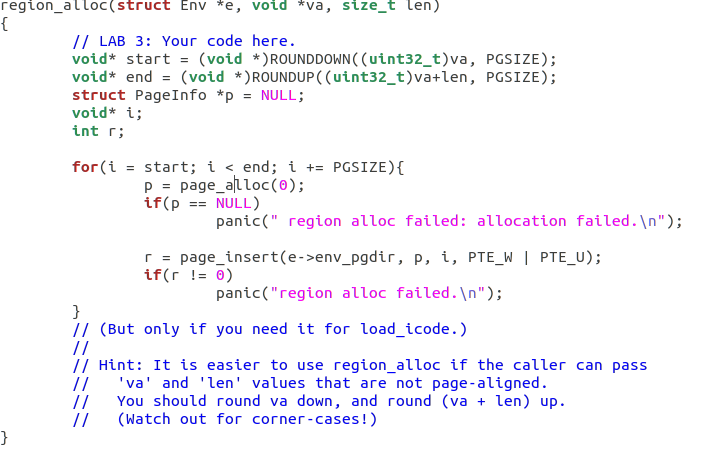
1. env\_init函数很简单，就是遍历 envs 数组中的所有 Env 结构体，把每一个结构体的 env\_id 字段置0，因为要求所有的 Env 在 env\_free\_list 中的顺序，要和它在 envs 中的顺序一致，所以需要采用头插法。



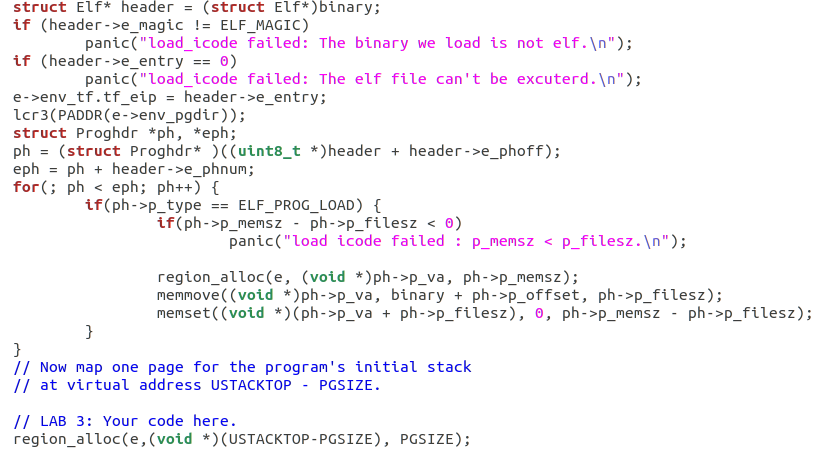
1. env\_setup\_vm 函数主要是初始化新的用户环境的页目录表，不过只设置页目录表中和操作系统内核跟内核相关的页目录项，用户环境的页目录项不要设置，因为所有用户环境的页目录表中和操作系统相关的页目录项都是一样的（除了虚拟地址UVPT，这个也会单独进行设置），所以我们可以参照 kern\_pgdir 中的内容来设置 env\_pgdir 中的内容。

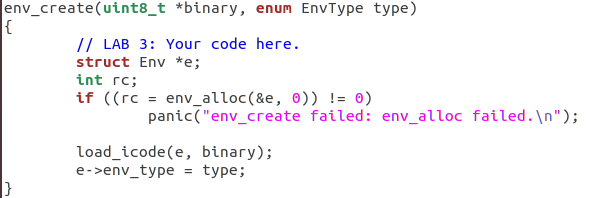
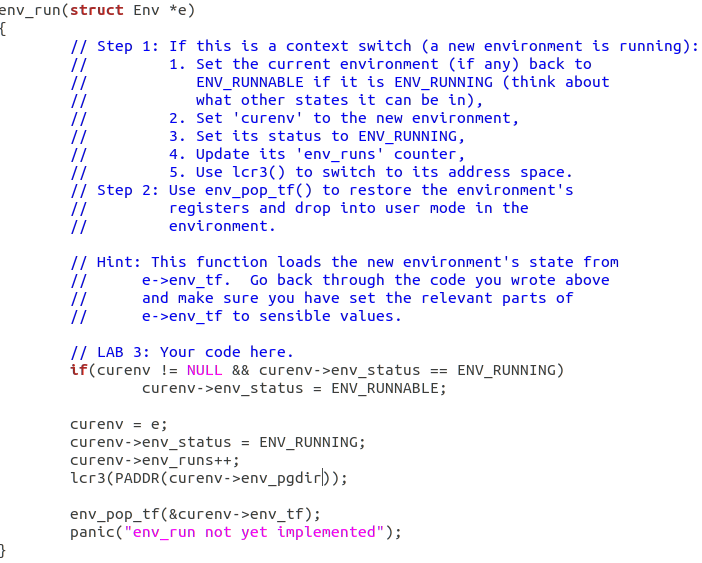


1. region\_alloc 为用户环境分配物理空间，这里注意我们要先把起始地址和终止地址进行页对齐，对其之后我们就可以以页为单位，为其一个页一个页的分配内存，并且修改页目录表和页表。



1. load\_icode 功能是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。每个用户程序都是ELF文件，所以我们要解析该ELF文件。

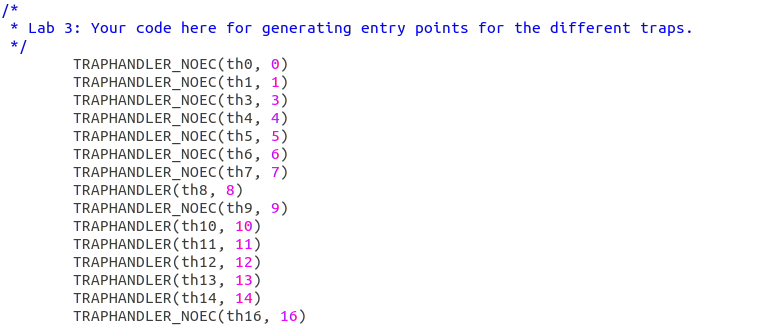


1. env\_create 是利用env\_alloc函数和load\_icode函数，加载一个ELF文件到用户环境中. 
2. env\_run 是真正开始运行一个用户环境

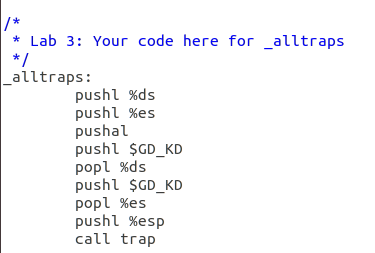


练习4：

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。

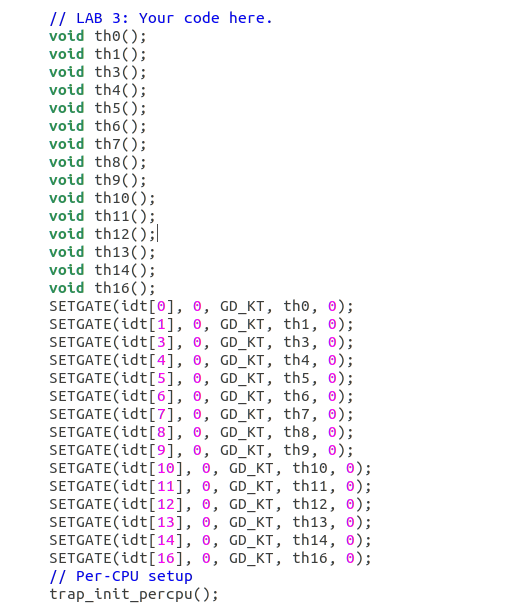


最后在trap.c中实现trap\_init函数，即在idt表中插入中断向量描述符，可以使用SETGATE宏实现：

　　SETGATE宏的定义：

　　#define SETGATE(gate, istrap, sel, off, dpl)

其中gate是idt表的index入口，istrap判断是异常还是中断，sel为代码段选择符，off表示对应的处理函数地址，dpl表示触发该异常或中断的用户权限。

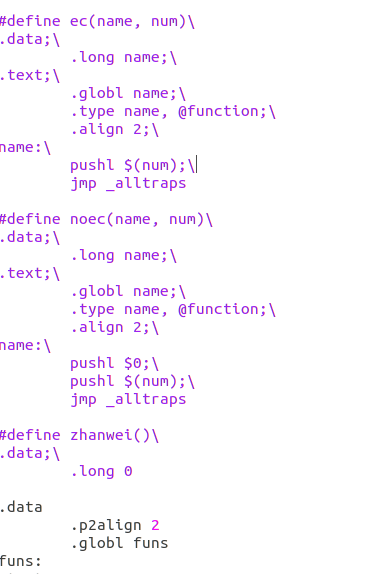


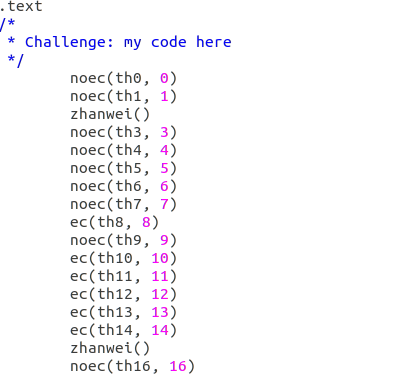
问题：

1. 不同的中断或者异常当然需要不同的中断处理函数，因为不同的异常/中断可能需要不同的处理方式，比如有些异常是代表指令有错误，则不会返回被中断的命令。而有些中断可能只是为了处理外部IO事件，此时执行完中断函数还要返回到被中断的程序中继续运行。
2. 因为当前的系统正在运行在用户态下，特权级为3，而INT指令为系统指令，特权级为0。特权级为3的程序不能直接调用特权级为0的程序，会引发一个General Protection Exception，即trap 13。

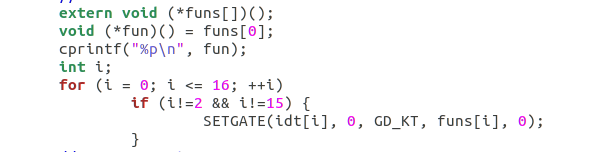
挑战：

trapentry.S



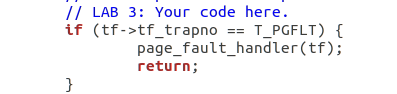


trap.c

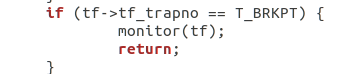


练习5：

根据 trapentry.S 文件中的 TRAPHANDLER 函数可知，这个函数会把当前中断的中断码压入堆栈中，再根据 inc/trap.h 文件中的 Trapframe 结构体我们可以知道，Trapframe 中的 tf\_trapno 成员代表这个中断的中断码。所以在 trap\_dispatch 函数中我们需要根据输入的 Trapframe 指针 tf 中的 tf\_trapno 成员来判断到来的中断是什么中断，这里我们需要判断是否是缺页中断，如果是则执行 page\_fault\_handler 函数



练习6：在这里我们需要处理断点中断 (T\_BRKPT)，kernel monitor 就是定义在 kern/monitor.c 文件中的 monitor 函数



问题：

通过实验发现出现这个现象的问题就是在设置IDT表中的breakpoint exception的表项时，如果我们把表项中的DPL字段设置为3，则会触发break point exception，如果设置为0，则会触发general protection exception。DPL字段代表的含义是段描述符优先级（Descriptor Privileged Level），如果我们想要当前执行的程序能够跳转到这个描述符所指向的程序哪里继续执行的话，有个要求，就是要求当前运行程序的CPL，RPL的最大值需要小于等于DPL，否则就会出现优先级低的代码试图去访问优先级高的代码的情况，就会触发general protection exception。那么我们的测试程序首先运行于用户态，它的CPL为3，当异常发生时，它希望去执行 int 3指令，这是一个系统级别的指令，用户态命令的CPL一定大于 int 3 的DPL，所以就会触发general protection exception，但是如果把IDT这个表项的DPL设置为3时，就不会出现这样的现象了，这时如果再出现异常，肯定是因为我们还没有编写处理break point exception的程序所引起的，所以是break point exception。

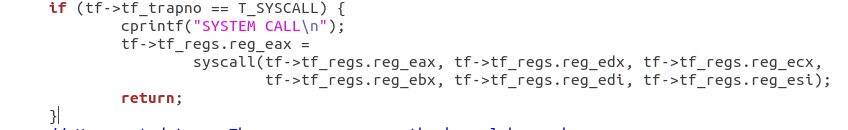
练习7：

步骤一：为中断号T\_SYSCALL添加一个中断处理函数

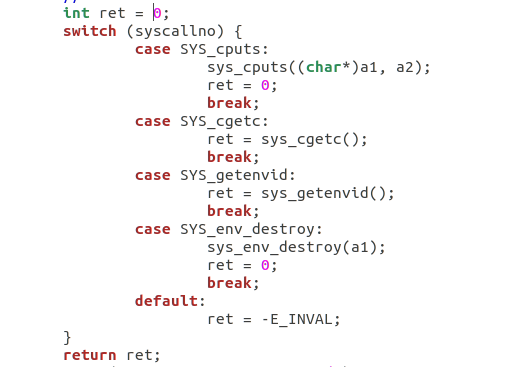




步骤二：在trap\_dispatch()中判断中断号如果是T\_SYSCALL，调用定义在kern/syscall.c中的syscall()函数，并将syscall()保存的返回值保存到tf->tf\_regs.reg\_eax等将来恢复到eax寄存器中。



步骤三：修改kern/syscall.c中的syscall()函数，使能处理定义在inc/syscall.h中的所有系统调用。



练习8：

用户程序执行后都会走到lib/libmain.c中的libmain()，需要修改该函数初始化其中的const volatile struct Env \*thisenv;变量。

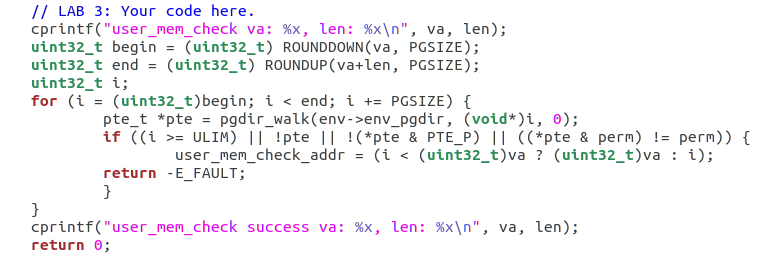


练习9：

第一步：在page\_fault\_handler()中添加如下代码：



第二步：修改kern/pmap.c中的user\_mem\_check()，进行检查



有了工具函数，我们看kern/syscall.c中的系统调用函数只有sys\_cputs()参数中有指针，所以需要对其进行检测：

