# Lab3

## Part A: User Environments and Exception Handling

struct Env \*envs = NULL; //所有的 Env 结构体

struct Env \*curenv = NULL; //目前正在运行的用户环境

static struct Env \*env\_free\_list; //还没有被使用的 Env 结构体链表

### Environment Status

我们要看一下，Env结构体每一个字段的具体含义是什么，Env结构体定义在 inc/env.h 文件中

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; //envid of this env's parent

enum EnvType env\_type;　　//Indicates special system environment

unsigned env\_status;　　 //Status of the environment

uint32\_t env\_runs; //Number of the times environment has run

pde\_t \*env\_pgdir;　　　　//Kernel virtual address of page dir.

};

env\_tf:

这个类型的结构体在inc/trap.h文件中被定义，里面存放着当用户环境暂停运行时，所有重要寄存器的值。内核也会在系统从用户态切换到内核态时保存这些值，这样的话用户环境可以在之后被恢复，继续执行。

env\_link:

这个指针指向在env\_free\_list中，该结构体的后一个free的Env结构体。当然前提是这个结构体还没有被分配给任意一个用户环境时，该域才有用。

env\_id:

这个值可以唯一的确定使用这个结构体的用户环境是什么。当这个用户环境终止，内核会把这个结构体分配给另外一个不同的环境，这个新的环境会有不同的env\_id值。

env\_parent\_id:

创建这个用户环境的父用户环境的env\_id

env\_type:

用于区别出来某个特定的用户环境。对于大多数环境来说，它的值都是 ENV\_TYPE\_USER.

env\_status:

这个变量存放以下可能的值

ENV\_FREE: 代表这个结构体是不活跃的，应该在链表env\_free\_list中。

ENV\_RUNNABLE: 代表这个结构体对应的用户环境已经就绪，等待被分配处理机。

ENV\_RUNNING: 代表这个结构体对应的用户环境正在运行。

ENV\_NOT\_RUNNABLE: 代表这个结构体所代表的是一个活跃的用户环境，但是它不能被调度运行，因为它在等待其他环境传递给它的消息。

ENV\_DYING: 代表这个结构体对应的是一个僵尸环境。一个僵尸环境在下一次陷入内核时会被释放回收。

env\_pgdir:

这个变量存放着这个环境的页目录的虚拟地址

### Allocating the Environments Array

#### Exercise 1.

修改一下mem\_init()的代码，让它能够分配envs数组。这个数组是由NENV个Env结构体组成的。envs数组所在的这部分内存空间也应该是用户模式只读的。被映射到虚拟地址UENVS处。

### Creating and Running Environments

#### Exercise 2.

在文件kern/env.c中，实现下列函数

void

env\_init(void)

初始化所有的在envs数组中的 Env结构体，并把它们加入到 env\_free\_list中。 还要调用 env\_init\_percpu，这个函数要配置段式内存管理系统，让它所管理的段，可能具有两种访问优先级其中的一种，一个是内核运行时的0优先级，以及用户运行时的3优先级

static int

env\_setup\_vm(struct Env \*e)

为一个新的用户环境分配一个页目录表，构造自己的地址空间，并且初始化这个用户环境的地址空间中的和内核相关的部分。

static void

region\_alloc(struct Env \*e, void \*va, size\_t len)

填充页表，将虚拟地址空间中[va, va+len )这段空间建立映射，为用户环境分配物理地址空间

static void

load\_icode(struct Env \*e, uint8\_t \*binary)

为用户进程e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文件分析一个ELF文件，类似于boot loader做的那样，我们可以把它的内容加载到用户环境下。

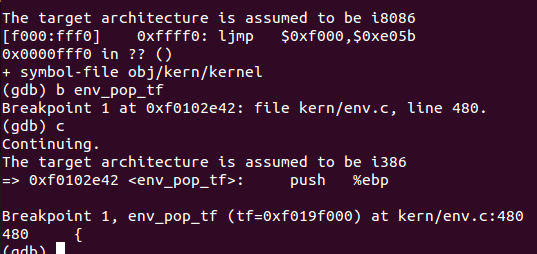
void

env\_create(uint8\_t \*binary, enum EnvType type)

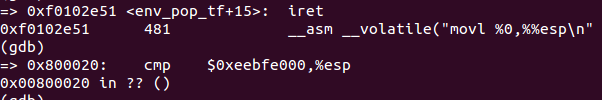
利用env\_alloc函数和load\_icode函数，加载一个ELF文件到用户环境中

完成上述子函数的代码，并且在QEMU下编译运行，系统会进入用户空间，并且开始执行hello程序，直到它做出一个系统调用指令int。但是这个系统调用指令不能成功运行，因为到目前为止，JOS还没有设置相关硬件来实现从用户态向内核态的转换功能。当CPU发现，它没有被设置成能够处理这种系统调用中断时，它会触发一个保护异常，然后发现这个保护异常也无法处理，从而又产生一个错误异常，然后又发现仍旧无法解决问题，所以最后放弃，我们把这个叫做"triple fault"（三重错误）。通常来说，接下来CPU会复位，系统会重启。

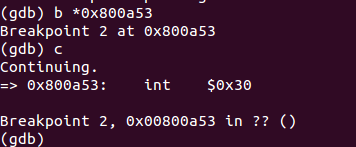
接下来调试。使用make qemu-gdb 并且在 env\_pop\_tf 处设置断点，这条指令应该是即将进入用户模式之前的最后一条指令。



然后进行单步调试，处理会在执行完 iret 指令后进入用户模式。然后依旧可以看到进入用户态后执行的第一条指令了，该指令是一个cmp指令，开始于文件 lib/entry.S 中。



现在使用 b \*0x... 设置一个断点在hello文件（obj/user/hello.asm）中的sys\_cputs函数中的 int $0x30 指令处。这个int指令是一个系统调用，用来展示一个字符到控制台。



如果程序运行不到这个int指令，说明有错误。

### Handling Interrupts and Exceptions

### Basics of Protected Control Transfer

### Setting Up the IDT

#### Exercise 4.

编辑一下trapentry.S 和 trap.c 文件，并且实现上面所说的功能。宏定义 TRAPHANDLER 和 TRAPHANDLER\_NOEC 会对你有帮助。你将会在 trapentry.S文件中为在inc/trap.h文件中的每一个trap加入一个入口指， 你也将会提供\_alttraps的值。

你需要修改trap\_init()函数来初始化idt表，使表中每一项指向定义在trapentry.S中的入口指针，SETGATE宏定义在这里用得上。

你所实现的 \_alltraps 应该：

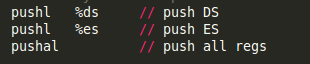
1. 把值压入堆栈使堆栈看起来像一个结构体 Trapframe

2. 加载 GD\_KD 的值到 %ds, %es寄存器中

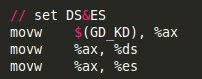
3. 把%esp的值压入，并且传递一个指向Trapframe的指针到trap()函数中。

4. 调用trap

考虑使用pushal指令，他会很好的和结构体 Trapframe 的布局配合好。



压入栈之后（esp）可以当做一个Trapframe结构体的起始地址，其布局是一致的



切换至内核模式



以esp的值为参，调用trap

Question:

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?)

答：

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. 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)?

答：

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。

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

### Handling Page Fault

#### Exercise 5

修改一下/kern/trap.c文件中trap\_dispatch 函数，使系统能够把缺页异常引导到 page\_fault\_handler() 上执行。

#### The Breakpoint Exception

#### Exercise 6

修改trap\_dispatch()使断点异常发生时，能够触发kernel monitor。

这个练习其实和上一个练习是类似的，只不过是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，

Question

1、在上面的break point exception测试程序中，如果你在设置IDT时，对break point exception采用不同的方式进行设置，可能会产生触发不同的异常，有可能是break point exception，有可能是 general protection exception。这是为什么？你应该怎么做才能得到一个我们想要的breakpoint exception，而不是general protection exception？

答：

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。

2、What do you think is the point of these mechanisms, particularly in light of what the user/softint test program does?

答：General protection exception

### System calls

Exercise7

修改/kern/syscall.c /kern/trap.c

给中断向量T\_SYSCALL编写一个中断处理函数。编辑kern/trapentry.S和kern/trap.c中的trap\_init()函数。修改trap\_dispatch()函数，使他能够通过调用syscall()（在kern/syscall.c中定义的）函数处理系统调用中断。

最终需要去实现kern/syscall.c中的syscall()函数。确保这个函数会在系统调用号为非法值时返回-E\_INVAL。你应该充分理解lib/syscall.c文件。我们要处理在inc/syscall.h文件中定义的所有系统调用。

通过make run-hello指令来运行 user/hello 程序，它应该在控制台上输出 "hello, world" 然后出发一个页中断。如果没有发生的话，代表你编写的系统调用处理函数是不正确的

答：

此时当系统调用中断发生时，系统就可以捕捉到这个中断了，中断发生时，系统会调用 \_alltraps 代码块，并且最终来到 trap() 函数处，进入trap函数后，经过一系列处理进入 trap\_dispatch 函数。题目中要求此时我们需要去调用 kern/syscall.c 中的syscall函数，这里注意，这个函数可不是 lib/syscall.c 中的 syscall 函数，但是通过阅读 kern/syscall.c 中的 syscall 程序我们发现，它的输入和 lib/syscall.c 中的 syscall 很像，如下

kern/syscall.c 中的 syscall ：

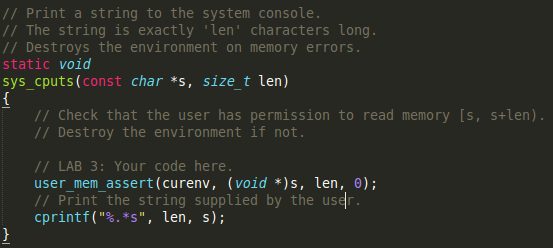
syscall(uint32\_t syscallno, uint32\_t a1, uint32\_t a2, uint32\_t a3, uint32\_t a4, uint32\_t a5)

lib/syscall.c 中的 syscall ：

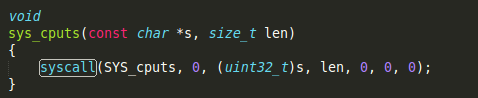
syscall(int num, int check, uint32\_t a1, uint32\_t a2, uint32\_t a3, uint32\_t a4, uint32\_t a5)

继续观察 kern/syscall.c 中的其他函数， kern/syscall.c 中的所有函数和 lib/syscall.c 中的所有函数都是一样的！！比如 在这两个文件中都有 sys\_cputs 函数，实现方式却不一样。拿 sys\_cputs 函数举例

在 kern/syscall.c 中的 sys\_cputs 是这样的：



而在 lib/syscall.c 中的 sys\_cputs 是这样的

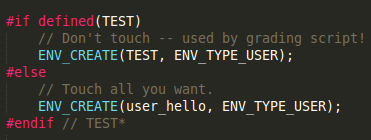


可见在用户态调用 lib/syscall.c 中sys\_cputs，后调用lib/syscall.c中的 syscall 函数，其汇编指令产生访管中断，进入内核态，调用kern/syscall.c中的syscall函数，该函数根据系统功能调用号调用kern/syscall.c 中的 sys\_cputs，它调用了 cprintf，完成输出的功能。

### User-mode startup

用户程序真正开始运行的地方是在lib/entry.S文件中。该文件中，首先会进行一些设置，然后就会调用lib/libmain.c 文件中的 libmain() 函数。你首先要修改一下 libmain() 函数，使它能够初始化全局指针 thisenv ，让它指向当前用户环境的 Env 结构体。

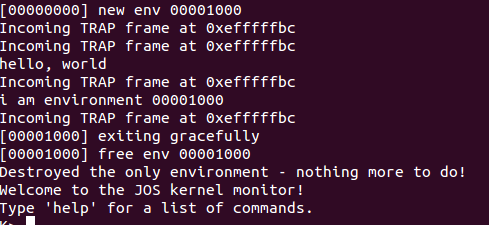
然后 libmain() 函数就会调用 umain，这个 umain 程序具体哪个函数在/kern/init.c中的i386\_init()函数中声明，如下图，此处是 user/hello.c 中被调用的函数。



在之前的实验中我们发现，hello.c程序只会打印 "hello, world" 这句话，然后就会报出 page fault 异常，原因就是 thisenv->env\_id 这条语句。现在你已经正确初始化了这个 thisenv的值，再次运行就应该不会报错了。

#### Exercise 8.

把我们刚刚提到的应该补全的代码补全，然后重新启动内核，此时你应该看到 user/hello 程序会打印 "hello, world", 然后在打印出来 "i am environment 00001000"。user/hello 然后就会尝试退出，通过调用 sys\_env\_destroy()。由于内核目前仅仅支持一个用户运行环境，所以它应该汇报 “已经销毁用户环境”的消息，然后退回内核监控器(kernel monitor)。



### Page faults and memory protection

#### Exercise9

1、

trap.c

void

page\_fault\_handler(struct Trapframe \*tf)

修改kern/trap.c文件，使其能够实现：当在内核模式下发现页错，trap.c 文件会panic。

提示：为了能够判断这个page fault是出现在内核模式下还是用户模式下，我们应该检查 tf\_cs 的低几位。

2、

kern/pmap.c

user\_mem\_check()

阅读 user\_mem\_assert （在 kern/pmap.c），并且实现 user\_mem\_check;

3、

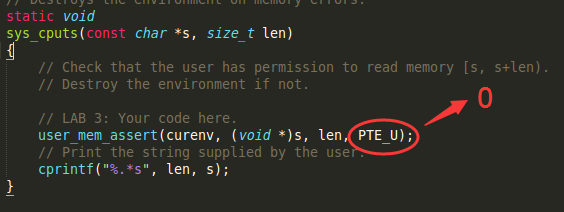
kern/syscall.c

static void

sys\_cputs(const char \*s, size\_t len)

修改一下 kern/syscall.c 去检查输入参数。即 sys\_cputs 函数，这个函数要求检查用户程序对虚拟地指空间 [s, s+len] 是否有访问权限，所以我们恰好可以使用刚刚写好的函数 user\_mem\_assert() 来实现。

此处权限值perm换做0也不会出错？

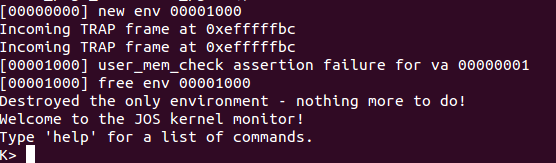


启动内核后，运行 user/buggyhello 程序，用户环境可以被销毁，内核不可以panic，你应该看到：

[00001000] user\_mem\_check assertion failure for va 00000001

[00001000] free env 00001000

Destroyed the only environment - nothing more to do!



4、

kern/kdebug.

debuginfo\_eip()

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.

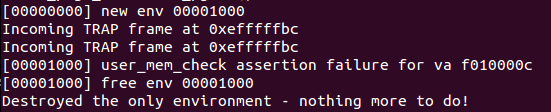
#### Exercise 10

重新启动内核，并且运行 user/evilhello。内核应该不能 panic，并且输出如下信息：

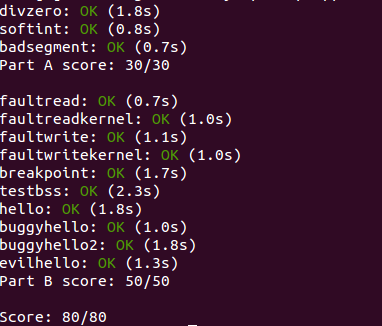
[00000000] new env 00001000

[00001000] user\_mem\_check assertion failure for va f010000c

[00001000] free env 00001000



#### Make grade



查阅知识：

lcr3(PADDR(pgdir))

CR3是页目录基址寄存器，保存页目录表的物理地址，页目录表总是放在以4K字节为单位的存储器边界上，因此，它的地址的低12位总为0，不起作用，即使写上内容，也不会被理会。