Lab3 设计描述

5130309029 沈匡济

PartA

PartA 部分的主要目的是设置 environment 并最终能成功执行用户态代码。lab 一开始便详细介绍了 Env 结构，有了 lab2 的经验，很快就能理解这个结构了，代码就不贴在文档里了。

-Exercise1

为 envs 数组分配物理空间并建立映射，模仿 pages 数组初始化部分，利用 boot_alloc 和 boot_map_region 两个函数搞定。

-Exercise2

接着很快就进入 ex2，要写很多 env 相关的代码，从题目描述中可以看出 jos 把用户态的程序以二进制的形式与内核代码编译在一起了，这一块的目的主要是最终能够通过 iret 跳到用户态执行代码。

env_init：创建一系列的 env 结构，初始化 id 为 0 并且将前后连接起来，虽然之后暂时只会用到一个。

env_setup_vm：已经分配了一个页作为页目录，我们要写的代码就是将它映射到内核地址空间，注释的意思是你可以直接利用 kern_pgdir 第 UTOP 条之后的部分，因为静态映射实际上是不需要分配物理页的，我们不需要再 page_alloc 可以直接整段复制过来。

下面的 env_alloc 函数姑且也自己看了一下，只是很简单的取空闲表的头然后进行一些变量的初始化，不深究。

region_alloc：这里不能跟 alloc 多个 envs 搞混了，我理解为为单个的 env 分配多个页，并把它们写进页表，按照注释对上下界的提示，每次循环 alloc 一个新的页并调用 page_insert 为这个页完成映射即可。

load_icode：这里让我有点头大，因为涉及到了更底层的操作并且和之前代码思路不太一样，先分析注释，它告诉了我们 ph->filesz 和 memsz 的区别，大意是说 filesz 比较小，但是分配空间得分配 memsz 的大小，多出的部分要留白。再分析传递的参数，可以看出 binary 是二进制化的用户态代码入口。于是可以看出首先是用提示的 region_alloc 来分配 memsz 的空间，再把用户态代码赋值过来就行了。注释中还提到页表切换和执行汇编指令地址的问题，前者用 lcr3 解决，后者通过修改 eip 寄存器的值解决。

env_create：两句话就能搞定，alloc 之后执行 load_icode。

env_run：这个函数可以理解为“唤醒一个 env”，由于这个 lab 实际上只在跑一个 env，因此它的作用其实不能完全体现吧，比较重要的是调用 env_pop_tf 进入用户代码。

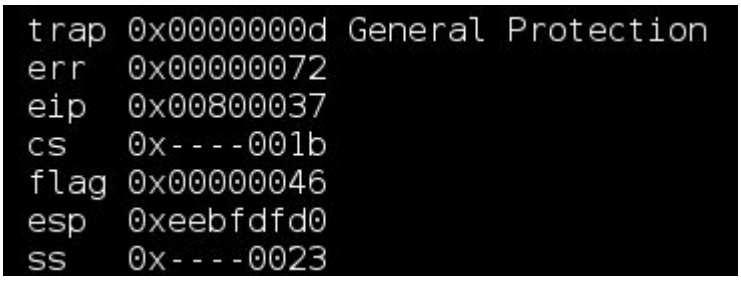
env_pop_tf：这个函数比较难看懂，结合 lab 里的链接，大概能明白它是先将栈上数据取出，然后利用 iret 来进入代码，iret 的详细机理是中断返回，应该是自己已经将各个用于中断恢复的寄存器的值指向了目标代码，然后这里就可以直接跳过去执行了吧。

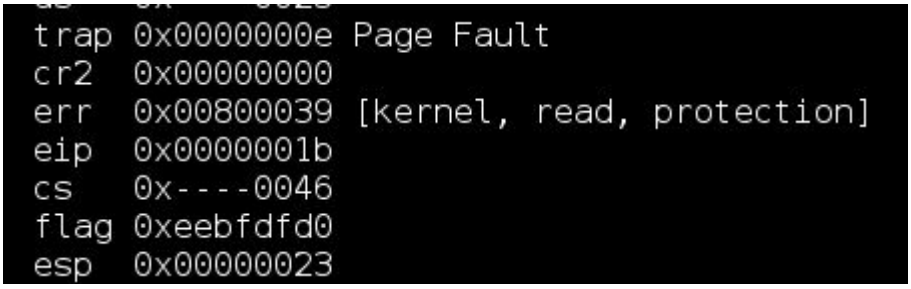
至此 ex2 结束，设置断点的时候一开始直接“break syscall”发现没有进入，仔细阅读意识到不是这个函数，进入 hello.asm 里找到正确的函数地址后进入成功。

-Exercise3

这部分代码非常直观，trap.c 里只需要初始化 idt 表，trapentry.S 里则分为两部分，先为不同的中断号声明处理函数，这里比较坑的是 lab 自带的链接里图都刷不出，好在自己在网上找到了原图。然后是一部分叫 alltrap 的汇编，这里由于 lab 详细的步骤描述也顺利做完了。

ans 详见 txt 文档，这里给出修改用户权限后的效果对比图。





PartB

下面进入 PartB 了，我打算边做边写文档。

开头详细讲述了 system call 的机理，我先来看看 lib/syscall.c 的代码。syscall 函数里一开始将各个寄存器信息压栈，然后取出，这中间应该需要填补代码来进入内核工作。

-Exercise6

提到了关键的函数 sysenter，它用来取代之前用的 iret，通过寄存器传值而非压栈传值，因此当前任务是在 trapentry.S 中设置这个函数的处理函数。逻辑倒很简单，但是照 TrapFrame 的格式压栈让我觉得挺麻烦。

这里我觉得不如理一理思路，先是用户代码 lib/syscall.c 调用 sysenter 函数，在 trap_init() 里设置好 handler 函数，在 trapentry.S 里完成 handler 函数对 syscall 的调用，即将寄存器压栈并调用 kern/syscall.c 中的 syscall 函数，这个函数里你还要根据 syscall number 来进入对应的函数。

于是先去看了 wrmsr 函数，这里的链接又打不开了，只好直接自己去查，发现这个函数主要就是给寄存器赋值，那么具体是怎么做？好在之前 lab 的另一个关于 sysenter 函数的链接里有这么一段：

These must be accessed through rdmsr and wrmsr:

IA32_SYSENTER_CS (0x174) - base ring 0 code segment. Ring 0 data = CS + 8. If REX.W prefix is used with SYSEXIT, ring 3 code = CS + 32 and ring 3 data = CS + 40. Otherwise, ring 3 code = CS + 16 and ring 3 data = CS + 24.

These values cannot be changed, therefore your GDT must be structured as such.

IA32_SYSENTER_ESP (0x175) - The kernel's ESP for SYSENTER.

IA32_SYSENTER_EIP (0x176) - The kernel's EIP for SYSENTER. This is the address of your

SYSENTER entry point.

于是很快在 trap_init 中添加了三句 wrmsr，完成了这块，这里有点想吐槽的是一开始不知道 wrmsr 要写 64 位，仔细查了资料才知道原来 EDX 寄存器内容拷贝至选定的 MSR 的高 32 位，EAX 内容拷贝至选定的 MSR 的低 32 位，接下来可以去 lib/syscall.c 里调用 sysenter 了。

lab 这里很友好的把每个寄存器的意义写了出来，这里一开始自己想仿照 ret 将 ret address 压栈，但是 lab 说得把它存在%esi 里，这里如果还要传满 6 个参数寄存器数量是不够的，我决定先用后者试试。在 trapentry.S 里，逻辑和 alltrap 有点相似，这里已经在内核态下了，函数调用只要用跟普通的汇编一样的逻辑就行了，于是照 lab 中的功能说明压了 5 个寄存器的值传参，然后调用 syscall 函数。在返回后将栈指针存在 ecx 寄存器中，将 retaddress 存在 edx 中，就可以调用 sysexit 返回到用户态代码啦。

最后是完成内核执行 syscall 的代码，这个就是个分发的逻辑，这时我发现其实一共只用到了 a1，a2 和 sysnumber3 个参数的值，实测甚至只压 ecx，edx，eax 也可以通过测试，所以之前 lab 提到的那个如何传 5 个参数的方法就不管了。

至此，ex6 结束，这一块因为上课听的不认真，现在学的十分痛苦，好在完整的用 gdb 跟踪了一遍，掌握了整个流程。

-Exercise7

一句话完成，在初始化过的 envs 数组里取出对应 id 的一个即可。

-Exercise8

完成 sys_sbrk()函数，它的作用是将一个函数的 data 段增长 inc，题目提示 alloc 很多页然后插到页表里面，这和前面的 region_alloc 如出一辙。

唯一的区别就是它还得返回增长后的位置，修改 Env 结构增加 env_sbrk 来追踪它，现在还要初始化这个值，根据提示去 load_icode 里面找答案。用户代码被加载进 ph->va 后，之后 filesz 大小是代码占用空间，filesz 到 memsz 之间是用于一些未被初始化的全局变量，那么 data 段自然是从 memsz 处开始了，至此，Ex8 结束。

-Exercise9

这里比较坑的一点是自己没有把 T_BRKPT 设置为 Ring3 级别，导致测试时候一直报 Protection Fault 然后无限重启。

函数本身逻辑很简单，x 是打印内存，写了一个很简单的进制转换的函数就解决了，另外两个函数则只要修改 EFLAGS，现在运行 make grade 可以看到已经 70/90。

-Exercise10、11

文章提到 OS 通常依赖硬件支持来实现内存保护，而对于非法内存读写分为可处理和不可处理两种，可处理的例子是程序请求额外栈空间的时候内核会实时分配。而 syscall 带来这方面的影响是，内核必须小心处理用户态程序传递的指针，因为它们可能导致更严重的内核 Page Fault 或是指向用户不可访问的地址。

这部分的核心就是实现 user_mem_check 函数，它的作用是给出一个内存段，你得一页一页去判断是否超出了 ULIM，以及页表项中是否给予了 perm 权限，这里注释写的很清晰，自己一开始采用了 region_alloc 中的写法，从 va 所在的页开始判断，但是如果该页出错，实际上应该返回 va 而不是像后面的页一样返回页地址。完成这个函数之后，需要在 kern/kdebug.c 的 debuginfo 中调用它来判断几个值，至此，EX10、11 结束。

-Exercise12

这个练习的要求是在用户态代码里利用 sys_map_kernel_page()获取 Ring0 权限，在我们把 gdt 用这个函数映射到用户态的内存空间后再进行修改就可以了，这里注释写的很详细。

-结语

这个 lab 花了自己很多时间，不过也学习了很多，对夏老师上课讲的内容理解更深了。