OS Lab3 文档

515030910211姜子悦

PartA:

Exercise1

修改mem\_init()，在kernel mem中增加env和envs相关内容，并标记为user ro。和lab2中pages的alloc和布局差不多，把size、va、pa换一下即可。

Exercise2

写了用户态环境初始化以及切换、新建、删除环境的内容。

env\_init初始化环境，新建freelist

env\_setup\_vm 建立新env的页表和虚存环境

region\_alloc 将新env的物理内存设置好

load\_icode 模仿boot的过程，加载代码段

env\_create 建立新环境，注意的点是要切换页表基地址

env\_run 切换进程，值得注意的点在于当前进程可能是不存在的(即切换的是第一个进程)，要注意考虑这种情况

Exercise3

提供了一个强力的guide，IA-32手册。很多后面的信息都可以翻这本手册找到，比如lcall、lret、sgbt、syscall、errcode这些

Exercise4

完成exception handler相关代码，对一个exception，过程是在trap.c中定义的idt中找到它的handler以及权限设置，然后到trapentry.S中执行该handler，最后在trap.c的trap中用depatch做后续处理。比较难的部分是汇编的写法，本身不熟悉如何写汇编，在理解的整个过程后，发现这里的汇编做两件事，一件事是通过给的有无error code的模板建好handler，然后在all trap中将余下的参数压栈模拟出一个trap frame结构，最后调音trap来处理。

IA32手册可以给予很多帮助，比如查哪些exception是有error code(主要作用是携带一些标记位，比如user/kernel present/privileged w/b)以及需要什么权限。如breakpoint就需要ring3权限，user态就可以触发，因为断点调试是个user态行为。而page fault这种就需要设置成ring 0才能触发，因为这不应该是user行为并且user如果能触发会出问题。

犯的错：一开始divzero没有跑成功，debug之后发现是之前env创建的时候犯了错误，在region\_alloc中没有考虑清楚边界情况，少alloc一页(< &<=的区别)导致pagefault发生。

Question1：

独立的handler可以做权限设置。如果只用一个handler，所有的exception都能被user态或都不能被user态触发。

Question2:

Softint不会带来问题，因为page fault handler设置成了ring0才能调用，所以会首先触发的是越权异常而不是其他异常。

PartB:

Exercise5

这部分处理了page fault，之前因为跑qemu到hello的时候还没有真正处理 pf，所以会报错误强迫症很难受。由于page fault handler已经给我们了，只需要在trap\_dispatch中加上如果遇到pf的trapno就就给handler处理就行了。

Handler中已经帮我们处理好了（针对grade程序）user pf。

其实这时候exercise还没有让我们做kernel pf的处理，但是我看到在handler里面以为是这部分要写的，于是用error code的 user/kernel bit去判断，由于相关测试没有测这部分所以这里没发现问题。到后面真正写kernel pf的exercise时，发现我理解错了，因为user/kernel bit实际上表示的是触发的页是user还是kernel权限的，并不是触发者自己的身份。

Exercise6

这部分实现syscall，感觉是除了最后一部分之外最难的。好几个syscall.c/h让我有点晕。后来理清楚:lib/syscall是定义了user可以调用的形式和相应代码。Inc/syscall是可以用到了结构和宏，kernel/syscall是我们要实现的主要部分，是kernel处理syscall的调用。

整个过程是，user用一个sysentry来触发syscall，kernel收到这个syscall的要求，wrmsr注册handler状态，trapentry.S中将sysnum和参数压栈，调用kernel中的syscall去处理相应调用。syscall处理函数通过sysnum判断是哪一个系统调用并用对应的函数去处理。

处理完成后，通过sysexit返回user态。

Exercise7

在libmain中，将当前env对应环境加载，这样就可以调用库函数了。

Exercise8

实现sys\_sbrk，这个调用的功能是扩展用户栈，syscall实现很简单，在env中增加一个env\_sbrk记录当前用户栈栈底位置，只需要将env\_sbrk下移并将新增虚存对应的页表和page分配好即可。

最大的难点在于需要修改很早之前写的env.c中的load\_icode，加入env\_sbrk初始化的代码。这个bug相当难以考虑到。多亏群里有同学说过这个问题才找到这个bug。

Exercise9

处理一个独特的breakpoint，它的行为是调出monitor。

这个过程很简单，在trap中捕获brpk异常，其处理行为是调用monitor函数即可。

第二部分增加x，si，c比较困难。

x addr，show某一个地址的值，属于之前做过的show栈的简化版，唯一值得注意的点是接受进来的参数是字符串而不是hex的数，所以需要注意转化。

si，行为类似gdb中step i命令，单步调式。一开始觉得完全没有思路，还试过每次通过eip去找下一条指令然后手动中断。后来google到了原来eflags中有一个bit专门控制单步执行模式，它的行为就是每执行一步发一次中断。

所以si只需要把这一位打开，然后像之前lab写过的show debug info同样的方式，显示相应debug即可。

c 和gdb中continue类似，让程序继续执行。有了eflags就方便多了，将单步执行bit关闭即可。

还有需要注意的是，si和c的return val应该是个负数，因为0会导致monitor不退出继续等到执行。需要用负数告诉monitor退出以符合grade需要的行为。

Question3：

依然是之前说的权限位的问题。把breakpoint的handler设置为ring3可以用，就能正常使用。如果设置成ring0可用，会触发一个越权异常。

Question4:

这个机制主要用来防止kernel被打断、triple fault这些因素导致出问题。严格控制一些exception只能被kernel/硬件触发以防止user态的一些攻击行为。

Exercise10

实现user\_mem\_insert防止user访问没有权限的va。

就是便利va到va+len，判断每个页在不在或者user有没有访问权限，没有就记录下来第一个不能访问的页然后返回错误码。否则就返回可以访问。

难点在于对齐，va和len都有可能是不对齐的，又需要返回第一个不能访问的地址。考虑到我们是一个page一个page的控制权限。所以采取如下做法：

先将va下对齐，va+len上对齐，以保证所有完整的page被访问到。除了第一页之外，所有应该被记录的无权限va都应该是page的第一个va。特殊处理第一个page，如果出现错误，将错误va设置成va。

Exercise11

做完10，11也一起过关了，美滋滋

Exercise12

实现一些evil的事情，crack自己的os……

超级超级难的一个part，一度想放弃，查了很久google也才明白原理

是这样的：

我们用段机制来控制不通权限的访问，如kernel data、kernel text、user data、user text，通过在gdt中给相应段设置权限位(ring 0，ring3)即可实现权限的划分。如果我们能通过某种方法修改它，把ring0 ring3改一改，就可以为所欲为。

而这张记录着信息的gdt是存在kernel某个地方的，由于它不是固定在某个地址上，我们需要像页基地址一样，有一个gdt基地址来表明它所在的位置。这个信息在我们的x86中，存在gdtr这个寄存器里，通过sgdt汇编指令，我们拿到了gdt基地址和limit。

现在，我们就有了gdt所在的地址和limit，并且我们的目的是去修改它。现在我们遇到的问题是这张表在kernel data中，我们正常访问会触发越权异常。

好在这个os特意提供了一个funny的syscall，能将一个kernel page以user可读写的形式拉到一个user va上。通过这个syscall，我们将gdt所在的物理页拿到了user可读写的vm上，现在我们可以放心修改它了。

一般gdt中就写了kernel data、kernel text、user data、user text四条entry，我们在后面本来是NULL的tss上新建一条entry即可。它参照kernel text，将ring 0 bit改成ring3即可，然后指向evil函数，这样我们user态就可以访问这个段。然后通过lcall，我们通过gdt的这个callgate，实现跨段call，来到了evil函数中，此时最大的区别是我们处于os认为的ring0段，即我们拥有了kernel级别的权限。现在我们可以随意调用kernel态函数，修改kernel数据了。

在完成这个evil操作后，我们把tss恢复成NULL然后直接通过leave+lret回到user态原来的位置。此时，gdt也恢复了原来的样子所以后续程序会正常的继续执行。