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。