JOSLAB3 解题报告

STONE

2015 年 11 月 14 日

Answers to Questions

Q1:

因为不同的 exception 对应有不同的 handle 方法。如果将所有的 exception 都传到一个同一个 handler 处理，第一会出现 exception 不能被正常处理的问题，第二对于一些 nested interrupt 会出现 triple fault 的情况。

Q2:

不需要。因为对于 int14 对应的 Page Fault exception 被设置为 ring0 级别即 Kernel，无法被 user 态访问，所以当 user 调用 int14 时，会出现另一个 exception，也就是 int13(General Protection)。

Q3：

因为对于 breakpoint(T_BRKPT)的优先级为 ring0，所以会出现 general protection，将其修改为 ring3 之后就能正常如题目所说的效果，但是如果访问 ring0 级别的数据/exception 就会出现 general protection。

Q4:

关键明显是 kernel 和 user 拥有不同的优先级，user(ring3)不能访问设置为 kernel 级(ring0)的数据以防止一些不怀好意的 user 进程窃取本不应该看到的 kernel/其他 user 环境的信息。

Brief Description to My Design

User Environment

首先与 lab2 中将 pages、kstack 映射到虚拟内存中操作相同 ,将从物理内存分配的 envs 按要求映射到指定的位置上(UENVS)并设置其权限位。

之后按照要求依次完成/kern/env.c 中的几个函数来实现 user 环境的设置：

env_init() 函数完成 envs[]的初始化，对照/inc/env.h 中的 Env 结构进行一一初始化即可，另外需要将 envs[]按照顺序链接到链表 env_free_list 中（按照要求要第一次 env_alloc()调用 envs[0],那么就要使 envs[0]放在链表的第一个位置，所以按照从后往前的顺序挂链表即可）。

env_setup_vm(struct Env *e) 函数给 e 申请一个 page，设置 e 的 env_pgdir 为这个页的虚拟地址，并将其放到 kernel 虚拟内存中，另外根据要求这个页的 pp_ref 是需要修改的（其他 UTOP 以上的页都不用）。

region_alloc(struct Env *e, void *va, size_t len) 函数是在 e 这个 user 环境下分配一块内存（一个物理页），如果分配成功了就插入到 e->pdgir 中，使用 lab2 完成的 page_alloc()与 page_insert()函数完成即可。

load_icode(struct Env *e, uint8_t *binary, size_t size) 函数是将指定的

binary(ELF 文件)读入 e 这个 user 环境中。先使用 lcr3 切换到 kernel 态，然后仿照 /boot/main.c 中的 bootmain()函数中对于 Elf 的写法完成 Elf 读入，并将多余位置 0，之后根据 env_pop_tf()，将 e->env_tf.tf_eip 置为 Elf 入口，然后返回 user 态使用 region_alloc()完成 user 栈的分配。

env_create(uint8_t *binary, size_t size, enum EnvType type) 函数比较简单，根据提示先用 env_alloc 分配一个新的 env 然后使用 load_icode 读取指定的 elf 即可。

env_run(struct Env *e) 函数即是启动一个 user 环境 e，根据提示的一步一步来就行，先进行状态的切换，然后用 lcr3 切到新的 user 环境对应的页表目录的物理地址位置并使用 env_pop_tf 存储当前寄存器状态即可。

完成之后发现 user 环境能正常启动出现 new env[0x00001000]但是系统不断重启。

Interrupt & Exception Handling

这一部分是根据上一部分完成后由于没有 handle exception 而造成 triple fault 而不断重启的问题，继续完成对不同 exception 的 handler。

首先，**完成 IDT**：根据/inc/trap.h 中的中断类型，在/kern/trapentry.s 中设置 trapentrys，在此处我选择了参考 xv6 相关代码来选择 TRAPHANDLER_NOEC/TRAPHANDLER 的使用(只 push $0 的就用 TRAPHANDLER_NOEC，多 push 一个 error code 的就用 TRAPHANDLER) 而并未查 intel 手册。然后根据设置的 trapentrys 在/kern/trap.c 中的 trap_init()中设置 IDT 表和刚刚设置的 trapentrys 函数的映射关系与权限(ring 级)。

然后，将所有的**中断信息组织起来压栈**并调用 trap()函数进行处理：查看/kern/trap.c 的 trap()函数，发现其参数为一个 TrapFrame 结构指针，在/kern/trapentry.s 中根据/inc/trap.h 中的 TrapFrame 结构处理组织参数压栈并调用 call trap 即可。

Page Fault

首先先梳理一下 **trap()函数结构**：先检查是否关闭了中断，然后判断是否是 user 态触发的中断，如果是，就重设定 curenv 的 tf 以保证 handler 返回时能让 user 环境切在在中断触发位置，然后进入 trap_dispatch 来根据中断种类进行调度，处理完之后返回之前的 user 环境。

此处需要添加的 page fault 处理应当在 trap_dispatch 中完成，那么在/kern/trap.c trap_dispatch 函数中添加对 PGFLT 的检测并使用提供的 page_fault_handler 函数进行处理即可。

System Call

使用 sysenter 和 sysexit 来完成 syscall，根据 reference，这一操作主要是跳过了 IDT 直接使用 syscallno 来进行 syscall 的处理。这一部分对我来说难度很大，参考 reference 也有很多不理解之处，很多汇编直接复制 reference 中的代码，并参考 wiki 等其他资料勉强完成了，但仍有不理解之处。（另外由于这个检查机制太差，报错全 tm 是 Page Fault，导致我不得不在各种地方输出一些奇怪的东西，最后这些地方写的太多了+完成这个部分时间太久不连续，有一些我的输出代码仍保留着没删完，请忽略）。

仍旧是首先梳理一下流程：首先在/kern/trapentry.s 中的 sysenter_handler 进入 syscall ,这里首先要修改/lib/syscall.h，完成其对 sysenter 的接口，并判断是否进入 sysenter，然后将 wrmsr 方法写入/inc/x86.h 并使用 wrmsr 完成/kern/trap.c 中 trap_init()函数中修改 CS(0x174),ESP(0x175)，EIP(0x176)三个 msr，并将其优先级置为 0（kernel）。这样就能顺利进入 syscall()函数中，然后根据 syscallno 对不同的 syscall 请求进行对应处理即可，之后会通过 sysexit 之后进入 trap 函数返回 user 环境。

User-mode Startup

Hello--这个很简单了，在/lib/libmain.c 中将 thisenv 设置成当前 user 环境的 env 就行，使用 envs[]与 sys_getendid()函数即可。

Sbrk--这个也比较简单，需要一个额外指针记录当前用户分配到的虚拟地址位置，然后使用一个类似 region_alloc 的 sbrk 函数（我在/kern/syscall.c 中实现）进行分配即可。不过这个额外指针是在涉及到页操作时候都要更改（包括 region_alloc）。

Breakpoint Exception

这个同样也很简单，首先与 Page Fault 处理方式相同，现在 trap_dispatch 中加入处理 BRKPT 与 DEBUG 的检测，然后根据提示在/kern/monitor.c 实现的 monitor 中完成 si/x/c 三个操作的实现即可。要注意的是需要在 Commands 添加这三个新命令。

Page faults and memory protection

目前的 page fault 都是统一在 page_fault_handler 中处理为 panic，而实际上 user 态的 page fault 并不需要直接 panic，那么就在 page_fault_handler 中先做判断，若为 user 态的 page fault，就改为摧毁这个 user env 而不是直接 panic。

而后进入/kern/pmap.c 查看所需要修改和使用的函数：

user_mem_check(struct Env *env, const void *va, size_t len, int perm)函数判断给定的 va 是否超过了给定 env 的界。这个判断主要分两个部分，第一个是看超没超过 ULIM，第二个是看超没超过权限（即看 va 是否在访问这个 env 无权看到的位置）。

user_mem_assert(struct Env *env, const void *va, size_t len, int perm)函数则是调用刚刚完成的 user_mem_check 函数来检查有没有 user page fault 出现，如果有就报错并摧毁当

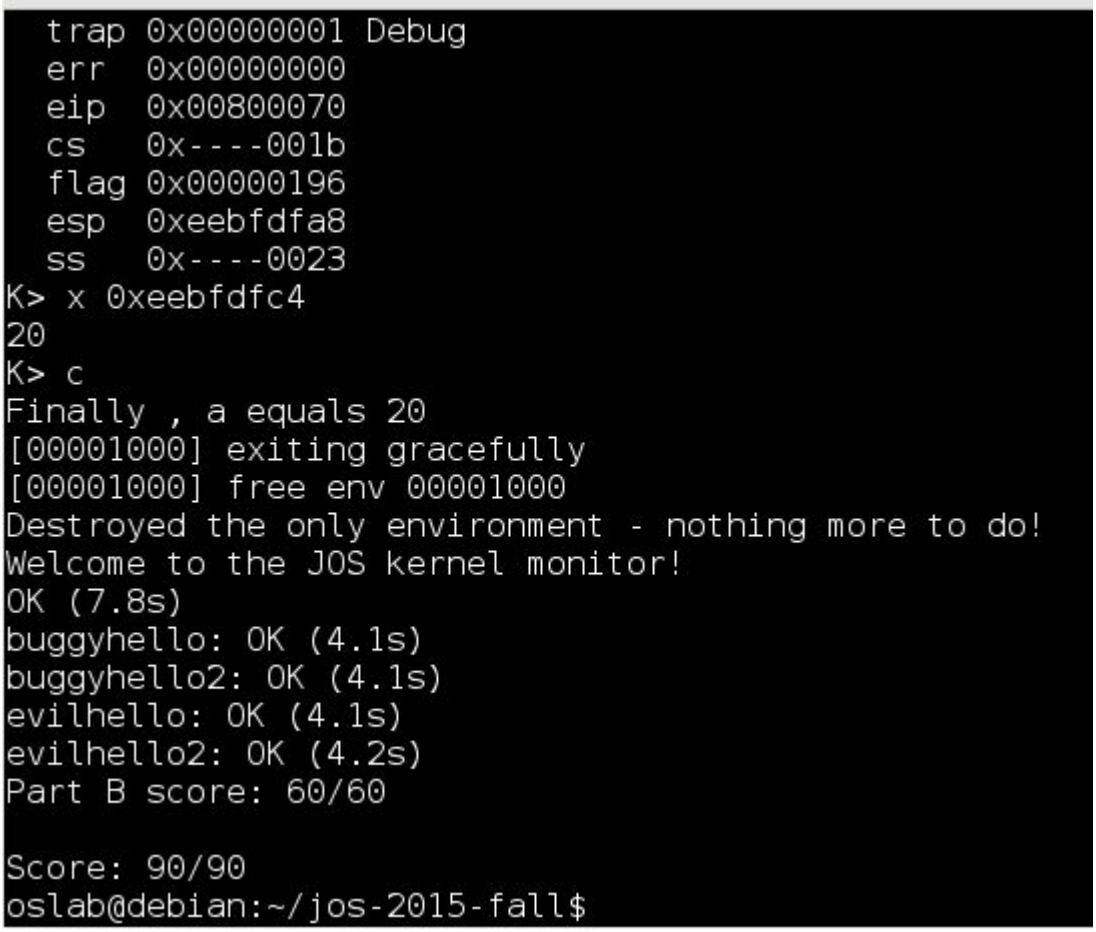
前环境。

那么很明显，在/kern/syscall.c 中我们只需要调用后者来完成 sys_cputs 就可以了。

同理，对于/kern/kdebug.c 中 debuginfo_eip()函数的修改也只需要相应调用 user_mem_check 就可以了。

最后一个 exercise 原理也比较好懂，就是在 user 态伪装成自己是 kernel 态调用 evil 函数来获取(这个在 user 态调用会 page fault)，根据提示，通过 sys_map_kernel_page（这玩意怎么会让 user 态用啊摔！太假了吧摔！）读取 gdt 并映射到 user 态创建的内存中，然后将这块内存的位置设置为 entry，之后通过 SETGATE 将这个 entry 映射到需要进入的 evil 函数中，就可以实现伪装成 kernel 调用的 evil，之后再将 gdt 恢复成之前的样子就能恢复成正常的 user 态。

Result



NO HACKER.

如果有任何问题请联系我 wy30123@163.com 石嘉昊