Jos Lab 3: User Environments
Lab Report

王勇杰 5120379068

2014.11.8

注：**lab** 简介以及 **questions** 都在此文档中，可通过目录直接跳转。

Contexts

1. User Environments and Exception Handling

1.1 Environment State

在这一部分介绍了 Env 这个存储环境的变量。

1.2 Allocating the Environments Array

Exercise 1

> Modify mem_init() in kern/pmap.c to allocate and map the envs array. This array consists of exactly NENV instances of the Env structure allocated much like how you allocated the pages array. Also like the pages array, the memory backing envs should also be mapped user read-only at UENVS (defined in inc/memlayout.h) so user processes can read from this array.

这个练习和上个 lab 中的练习类似，给 envs 分配内存空间，并进行映射，添加的代码如下：

```
envs = boot_alloc(NENV * sizeof(struct Env));
boot_map_region(kern_pgdir, UENVS, ROUNDUP(NENV * sizeof(struct Env), PGSIZE),
    PADDR(envs), PTE_U | PTE_P);
```

1.3 Creating and Running Environments

Exercise 2

> In the file env.c, finish coding the following functions:
>
> | | | |
> |---|---|---|
> | ***env_init()*** | ***env_setup_vm()*** | ***region_alloc()*** |
> | ***load_icode()*** | ***env_create()*** | ***env_run()*** |

A. env_init()

```
119     int i;
120     for(i = 0; i < NENV - 1; ++i){
121         envs[i].env_id = 0;
122         envs[i].env_status = ENV_FREE;
123         envs[i].env_link = &envs[i+1];
124     }
125
126     envs[NENV-1].env_id = 0;
127     envs[NENV-1].env_link = NULL;
128     env_free_list = envs;
```

这段代码为初始化 envs 链表，首先在 120-124 行将 envs 中每项设置初始值并将其指向下一项，

126-127 行初始化最后一项，128 行初始化链表头。

B. env_setup_vm()

```
193     e->env_pgdir = page2kva(p);
194
195     for(i = 0; i < PDX(UTOP); i++)
196         e->env_pgdir[i] = 0;
197     for(i = PDX(UTOP); i < NPDENTRIES; i++)
198         e->env_pgdir[i] = kern_pgdir[i];
199     p->pp_ref++;
```

这个函数初始化了环境内核的虚拟空间。

C. region_alloc()

```
288     va = ROUNDDOWN(va, PGSIZE);
289     len = ROUNDUP(va + len, PGSIZE) - va;
290
291     struct Page *pp;
292     uintptr_t vap;
293     for (vap = (uintptr_t)va; vap < (uintptr_t)va + len; vap += PGSIZE)
294     {
295         pp = page_alloc(0);
296         if (pp == NULL) panic("[env.c]region_alloc error: page_alloc failed");
297         page_insert(e->env_pgdir, pp, (void*) vap, PTE_U | PTE_W);
298     }
```

这个函数就是为环境分配长度为 len 的物理空间，将其与虚拟地址 va 做映射。在 288-289 行求得所需内存长度之后在 293-298 行一页一页分配内存。

D. load_icode()

这个函数是将 ELF 格式的可执行代码载入内存，具体代码可见 kern/env.c 文件中。

E. env_create()

```
401     struct Env* env;
402     if(env_alloc(&env, 0) < 0)
403         panic("[env.c]env_create error: env_alloc failed");
404
405     load_icode(env, binary, size);
406     env->env_type = type;
```

这个函数用于初始化一个环境。这个函数首先调用 env_alloc 来分配一个环境，再通过 load_inode 将可执行代码载入。

F. env_run()

```
522     if (e != curenv)
523     {
524         if (curenv && curenv->env_status == ENV_RUNNING)
525             curenv->env_status = ENV_RUNNABLE;
526
527         curenv = e;
528         curenv->env_status = ENV_RUNNING;
529         curenv->env_runs++;
530         lcr3(PADDR(curenv->env_pgdir));
531     }
532
533
534     env_pop_tf(&e->env_tf);
```

这个函数是将 e 设置为当前执行的环境，主要在于 522 行对当前环境的判断，只有在切换到一个新的用户环境时，才需要开启新的用户页。

1.4 Setting Up the IDT

Exercise 4

> Edit trapentry.S and trap.c and implement the features described above. The macros TRAPHANDLER and TRAPHANDLER_NOEC in trapentry.S should help you, as well as the T_* defines in inc/trap.h. You will need to add an entry point intrapentry.S (using those macros) for each trap defined in inc/trap.h, and you'll have to provide _alltraps which the TRAPHANDLER macros refer to. You will also need to modify trap_init() to initialize the idt to point to each of these entry points defined in trapentry.S; the SETGATE macro will be helpful here.
>
> Your _alltraps should:
>
> 1. push values to make the stack look like a struct Trapframe
> 2. load GD_KD into %ds and %es
> 3. pushl %esp to pass a pointer to the Trapframe as an argument to trap()
> 4. call trap (can trap ever return?)
>
> Consider using the pushal instruction; it fits nicely with the layout of the struct Trapframe.

这个部分需要实现中断的映射。首先现在 trapentry.S 中定义各个中断的处理函数，而后在 trap.c 中将中断号与处理函数想关联，最后在 trapentry.S 中完成 alltraps，用于将中断跳转至 trap 函数处理。

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?)

因为在处理中断的时候，如果公用一个处理函数，首先对 errorcode 的处理会有难度，其次处理程序无法得之是哪个中断调用而来，也就无法很好地设置中断号了。

Question 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)?

因为每个中断都有权限，虽然 user/softint 调用用的是 int 14，但我们在 IDT 中设置的权限为 0，即这是一个内核中断，因此当用户程序触发这个中断时，CPU 产生int 13 的中断来进行保护。

2. Page Faults, Breakpoints Exceptions, and System Calls

2.1 Handling Page Faults

Exercise 5

> Modify `trap_dispatch()` to dispatch page fault exceptions to `page_fault_handler()`. You should now be able to get **make grade** to succeed on the `faultread`, `faultreadkernel`, `faultwrite`, and `faultwritekernel` tests. If any of them don't work, figure out why and fix them. Remember that you can boot JOS into a particular user program using **make run-*x*** or **make run-*x*-nox**.

这里主要需要根据不同的中断号进行判断，如果需要进行处理则调用 handler 函数。

2.2 System calls

Exercise 6

> Implement system calls using the sysenter and sysexit instructions.

在这个部分需要实现 sysenter/sysexit 的系统调用。

首先在 trap.c 的 trap_init 中将 sysenter 指令与 sysenter_handler 进行映射，并在 lib.syscall.c 中的 syscall 函数中调用这个指令，使其触发 handler。

而后在 sysenter_handler 中调用用 syscall_wrapper，并在设置好各个参数后调用syscall，而在 syscall 中根据不同的中断号，调用不同的中断函数来处理这些中断即可。

2.3 User-mode startup

Exercise 7

> Add the required code to the user library, then boot your kernel. You should see user/hello print "hello, world" and then print "i am environment 00001000". user/hello then attempts to "exit" by calling sys_env_destroy() (see lib/libmain.c and lib/exit.c). Since the kernel currently only supports one user environment, it should report that it has destroyed the only environment and then drop into the kernel monitor. You should be able to get **make grade** to succeed on the hello test.

这个练习需要设置用户态的环境，只需要在 libmain 中对 thisenv 进行设置即可。

```
16      thisenv = envs + ENVX(sys_getenvid());
```

Exercise 8

You need to write syscall *sbrk*. The *sbrk()*, as described in the manual page (man sbrk), extends the size of a process's data segment (heap). It dynamically allocates memory for a program. Actually, the famous *malloc* allocates memory in the heap using this syscall.
As

```
int sys_sbrk(uint32_t increment);
```

shows, *sbrk()* increase current program' data space by increment bytes. On success,*sbrk()* returns the current program 's break after being increased.

这个函数要拓展用户堆，因此先要在环境变量中增加存储用户堆大小的变量，而后将用户堆进行加大，并在 syscall 中增加对这个指令的系统调用。具体代码可见 kern/syscall.c 的 77-84 行代码。

2.4 The Breakpoint Exception

Exercise 9

Modify trap_dispatch() to make breakpoint exceptions invoke the kernel monitor. You should now be able to get **make grade** to succeed on the breakpoint test. After that , you should modify the JOS kernel monitor to support GDB-style debugging commands c, si and x.
c tells GDB to continue execution from the current location , si means executing the code instruction by instruction , and x means display the memory. You will need to understand certain bits of the EFLAGS register. Your si should call the debuginfo_eip() to print some information about current eip , and your x should print consequent 4 byte data .

首先需要修改 trap_dispatch 这个函数使其对 breakpoint 的中断增加处理方式，并且引导至 monitor.c 增加 c，si 与 x 这几个指令。（在执行 si 时会有错误，没有找到原因）

Question 3

> The break point test case will either generate a break point exception or a general protection fault depending on how you initialized the break point entry in the IDT (i.e., your call to `SETGATE` from `trap_init`). Why? How do you need to set it up in order to get the breakpoint exception to work as specified above and what incorrect setup would cause it to trigger a general protection fault?

这个主要原因在于在设置 IDT 的时候权限为 0 或者是 3。如果权限为 0，说明只有内核能处理中断，此时用户出发 breakpoint 会产生 general protection fault。而如果权限为 3，说明用户有权限处理 breakpoint 的中断，就会产生 break point exception。在这个 lab 中需要用户能处理 breakpoint，因此将其权限设置为 3。

Question 4

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

主要在于在 IDT 中对于每个中断都设置的权限，如果用户出发了一个只有内核能处理的中断就会触发 general protection fault，以此来保护内核不被任意修改。

2.5 Page faults and memory protection

Exercise 10

> Change kern/trap.c to panic if a page fault happens in kernel mode.
>
> Read user_mem_assert in kern/pmap.c and implement user_mem_check in that same file.
>
> Change kern/syscall.c to sanity check arguments to system calls.
>
> Boot your kernel, running user/buggyhello. The environment should be destroyed, and the kernel should *not* panic. You should see:
>
> ```
> [00001000] user_mem_check assertion failure for va 00000001
> [00001000] free env 00001000
> Destroyed the only environment - nothing more to do!
> ```
>
> 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.

首先当这个 page fault 发生在 kernel mode 时直接停止系统运行，代码如下：

```
310     if((tf->tf_cs & 3) == 0){
311         panic("Kernel model page fault: %08x", fault_va);
312     }
```

接下来在 user_mem_check 中对用户内存进行检查：

```
722 user_mem_check(struct Env *env, const void *va, size_t len, int perm)
723 {
724     // LAB 3: Your code here.
725     unsigned int* p;
726     bool breakFlag = 0;
727     perm = perm | PTE_U | PTE_P;
728     for(p = ROUNDDOWN((unsigned int*)va, PGSIZE); p < ROUNDUP((unsigned int*)(va + le
729         pte_t* pte_p = pgdir_walk(env->env_pgdir, p, 0);
730         if(p > (unsigned int*)ULIM ||    //out of range
731             !pte_p ||    //page doesn't exist
732             ((*pte_p) & perm) != perm)  //less perm than expected.
733         {
734             //sth is wrong.
735             user_mem_check_addr = ((uintptr_t)va>(uintptr_t)p)?(uintptr_t)va:(uintptr
736             return -E_FAULT;
737         }
738     }
739     return 0;
```

首先检查该虚拟地址是不是低于 ULIM（730 行），而后对每一页都检查其是否对相应的页有访问权限（731-732 行），如果不满足则返回-E_FAULT。

Exercise 12

> evilhello2.c want to perform some privileged operations in function evil(). Function ring0_call() takes an function pointer as argument. It calls the provided function pointer with ring0 privilege and then return to ring3. There's few ways to achieve it. You should follow the instructions in the comments to enter ring0.
>
> sgdt is an unprivileged instruction in x86 architecture. It stores the GDT descripter into a provided memory location. After mapping the page contains GDT into user space, we could setup an callgate in GDT. Callgate is one of the cross-privilege level control transfer mechanisms in x86 architecture. After setting up the call gate. Applications may use lcall (far call) instruction to call into the segment specified in callgate entry (For example, kernel code segment). After that, lret instruction could be used to return to the original segment. For more information on Callgate. Please refer to intel documents.
>
> Finish ring0_call() and run evilhello2.c, you should see IN RING0!!! followed by a page fault. (the function evil() is called twice, one in ring0 and one in ring3).

这里就跟着注释进行编写完成 ring0_call 的函数。

```
38  void evil_wrapper(){
39      evil();
40      *gdt_entry = saved_gdtdesc;
41      asm volatile("popl %ebp");
42      asm volatile("lret");
43  }
```

```
61      //step1
62      struct Pseudodesc gdtd;
63      struct Segdesc *gdt;
64      sgdt(&gdtd);
65      //step2
66      if (sys_map_kernel_page((void*)gdtd.pd_base, (void*)va) < 0)
67          cprintf("map kernel page error\n");
68      //step3
69      gdt = (struct Segdesc*) ((uint32_t)(PGNUM(va) << PTXSHIFT) + PGOFF(gdtd.pd_base));
70      gdt_entry = gdt + (uint32_t)(GD_UD >> 3);
71      saved_gdtdesc = *gdt_entry;
72      SETCALLGATE(*((struct Gatedesc*)gdt_entry), GD_KT, evil_wrapper, 3);
73
74      //step 7
75      asm volatile("lcall $0x20, $0");
```