

Homework #3

周雨扬

2000013061

1 Challenges

本次作业完成了所有的代码补全任务，做了如下的 Challenge:

- Modify the JOS kernel monitor so that you can 'continue' execution from the current location (e.g., after the `int3`, if the kernel monitor was invoked via the breakpoint exception), and so that you can single-step one instruction at a time. You will need to understand certain bits of the EFLAGS register in order to implement single-stepping.

2 Exercise 1

`mem_init()`

我们只需要声明用于存储页链表 `envs` 的内存即可。因此可以通过直接调用 `boot_alloc` 解决。代码可以见 `kern/pmap.c`, Line 177 ~ 179 ,210。

备注

由于不明原因，我修改了初始化时候的部分内存分配，增加了 `kern/pmap.c`, Line 149。如果不直接添加。其会导致存储 `kern_pgdir` 的地址，恰好落在 `[kern_pgdir, kern_pgdir + PGSIZE)` 区间内，这意味着随着两行后的 `memset` 指令，就会将指针 `kern_pgdir` 指向的地址赋为 0。

产生的原因不是 Lab1 的代码因为 Lab1 几乎没有改动，错误产生的位置在 Lab2 所有改动之前，因此我怀疑是链接器/链接文件出了些 bug。

3 Exercise 2

`env_init()`

我们需要将所有的空环境用链表的形式进行存储，且使得最后的 `env_free_list` 指向 `envs[0]`。因此我们直接按描述建立即可。代码可以见 `kern/env.c`, Line 120 ~ 125。

注意由于我们分配的时候已经赋值为 0，因此不需要设置初始 id 为 0。

`env_setup_vm()`

我们可以利用已经创建好的页表 `kern_pgdir` 作为模板拷贝到新的环境页表中。之后直接将新页表下标赋值给环境即可。代码可以见 `kern/env.c`, Line 189 ~ 191。

注意由于我们需要手动增加新页表的引用数。

region_alloc()

首先我们需要将映射的内存按照整页进行上下取整 (因为我们采用整页管理)。之后, 我们可以直接依序利用 Lab2 中的页声明函数 `page_alloc()` 和页插入函数 `page_insert` 将页插入到当前环境的虚拟内存中即可。代码可以见 `kern/env.c`, Line 280 ~ 288。

load_icode()

首先我们需要检查 ELF 头中的 `e_magic` 项是否正确。如果不正确则无法执行需要退出; 否则我们首先需要切换页表。观察发现下面的代码我们可以使用 `lcr3()` 来实现该操作。

之后我们就可以利用 ELF 头求出程序描述符 `Proghdr`, 并求出代码中每一段的信息。由于我们只需要将属性为 `ELF_PROG_LOAD` 的每一段进行插入, 我们可以直接使用 `region_alloc()` 分配内存后简单的 `memcpy` 即可。多余的部分将其赋值为 0。

最后我们需要将页表切换回内核态的页表, 同时设置插入点。观察发现点为 `env_tf.tf_eip`, 就可以完成代码。代码可以见 `kern/env.c`, Line 346 ~ 361。

env_create()

根据 lab 上面的提示编写即可。注意我们已经在声明存储环境的空间是, 将 `parent ID` 设置为 0 了。代码可以见 `kern/env.c`, Line 381 ~ 386。

env_run()

如果当前存在运行进程, 则我们首先需要将其切出, 之后再当前环境设置为 `ENV_RUNNING`。注意之后的所有寻址都是在当前环境的页表下进行, 因此需要切换页表根节点。

之后根据描述我们只需要调用 `env_pop_tf` 并传入对应参数即可。代码可以见 `kern/env.c`, Line 503 ~ 513。

4 Exercise 4

trapentry.S

观察代码, 发现其引用了 `inc/trap.h`, 其中共有 20 种中断与异常类型。查阅相关材料可以直接确定每一种异常/中断是否需要在栈中压入 `error code`。据此我们可以首先在文件中声明每一种错误的入口函数。代码可以见 `kern/trapentry.S`, Line 50 ~ 70。

之后我们需要完善 `_alltraps`。观察 `Trapframe` 和文档我们发现系统已经帮我们实现压入了 `trap_no` 之后的部分, `trap_no` 在声明异常函数的时候已经被压入, 此时我们只需要压入 `%ds, %es` 和所有的寄存器即可。之后的指令可以直接按照文档里给出的写。代码可以见 `kern/trapentry.S`, Line 76 ~ 87。

trap_init()

此时我们需要声明 `idt` 数组。我们发现所有的中断都会重设 `IF` 位, 因此 `istrap = 0`; `kernel code segment` 在 `kern/env.h` 出现过, 因此其值位 `GD_KT`; 最后两项依次位函数入口和权限级别。

函数接口我们在 `trapentry` 已经声明过，其类型为 `void()`；权限目前可以统一设置为内核级 (0)。

据此我们完成了其初始化，代码可以见 `kern/trap.c`, line 66 ~ 106。

question 1

如果所有异常都使用同一个 handler，一会出现安全问题，因为不同异常操作权限不同；二也不利于多个异常同时抛出情况下内核的处理。

question 2

因为页错误的权限位是 0，我们不希望用户能够抛出页错误。

如果我们将权限位设为 3(用户态)，则用户就可以抛出该错误，但这不是我们所希望的：用户不能也不应该干涉内存管理。

5 Exercise 5,6

trap_dispatch()

观察到页错误和端点分别对应了 `T_PGFLT` 和 `T_BRPKT`, 对应需要执行的函数是 `page_fault_handler()` 和 `moniter`, 因此直接实现即可。但是这样断点会错误，因为其是在用户态下被发送的，我们需要额外在 `idt` 种修改其权限位为 3。代码可以见 `kern/trap.c`, line 186 ~ 192。

question 3

类似于 question 2，如果我们设置只有系统能够产生该中断，则会产生 `general protection fault`，因为用户权限不足。

为此我们才需要额外在 `idt` 种修改其权限位为 3。这样用户才能产生端点错误，系统便能够正确捕捉了。

question 4

为了保护系统，不让用户干涉某些重要过程。

6 Exercise 7

trap_dispatch()

由于用户可以发起系统调用，因此我们需要将其权限位设置为 3。查阅资料发现系统调用的参数依次为 `%edx`, `%ecx`, `%ebx`, `%edi`, `%esi`, 因此我们直接调用 `syscall` 函数并将其返回值存储在 `%eax` 中即可。该部分代码可以见 `kern/trap.c`, line 193 ~ 200

syscall()

根据描述我们只需要利用一个 `switch` 语句判断其执行的命令即可。注意参数是按照 `a1, a2` 的顺序传输，在命令错误时只需要返回 `-E_INVALID`. 该部分代码可以见 `kern/syscall.c`, line 75 ~ 87

7 Exercise 8

libmain()

由于我们已经声明了存储环境信息的内存 `envs`，因此我们只需要获取环境编号即可。获取编号可以使用 `sys_getenvid`，利用编号获取偏移可以使用 `ENVX`，简单实现即可。该部分代码可以见 `lib/libmain.c`, line 16.

8 Exercise 9

page_fault_handler()

观察代码发现, `tf_cs` 的最低两位恰好代表环境的权限位。基于此可以简单地实现，代码可以见 `lib/libmain.c`, line 265 ~ 266.

user_mem_assert()

类似于 `region_alloc()`，我们可以首先获取需要检查的页，对于每一个页，我们利用 `pgdir_walk()` 获取该页的信息，并检查该页的信息是否存在，是否越界，权限是否合法。注意最后生成的 `user_mem_check_addr` 需要和 `va` 取较大者。代码可以见 `kern/pmap.c`, line 577 ~ 586.

sys_cputs()

利用 `user_mem_assert` 检查对应内存区间权限即可。代码可以见 `kern/syscall.c`, line 24.

debug_eip_info()

类似的利用 `user_mem_check` 检查对应内存区间权限即可。代码可以见 `kern/kdebug.c`, line 145, 154, 155.

Challenge 2

再出现 `int $3` 之后我们会进入内核态，不再进行当前进程。但是我们实际上可以利用陷入时候的 `struct Trapframe *tf` 恢复原先状态。当然在恢复之前，我们需要检查传入的参数 `tf`，判定其是否属于可以继续执行的状态。

复原状态的时候，我们直接修改当前状态的 `env_tf` 值为传入值即可。针对单步执行和继续执行两种不同的情况，查阅资料发现 `env_tf` 中 `tf_eflags` 的第 9 位 (0x100) 恰好可以控制该性质，其值位 1 的时候会进入单步状态。据此我们针对不同的任务对应修改即可。

代码可以见 `kern/moniter.c`, line 182 ~ 216.

运行结果如下图所示：

```

return 0;
check_page_installed_pgdir() succeeded!
[00000000] new env 00001000
Incoming TRAP frame at 0xeffffbfc
Incoming TRAP frame at 0xeffffbfc
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
TRAP frame at 0xf01d4000
edi 0x00000000
esi 0x00000000
ebp 0xeebdfdc0
oesp 0xefffffdc
ebx 0x00002000
edx 0x0000202c
ecx 0x00000000
eax 0xeec00000
es 0x---0023
ds 0x---0023
trap 0x00000003 Breakpoint
err 0x00000000
eip 0x00000037
cs 0x---001b
flag 0x00000082
esp 0xeebdfdc0
ss 0x---0023
K> _

```

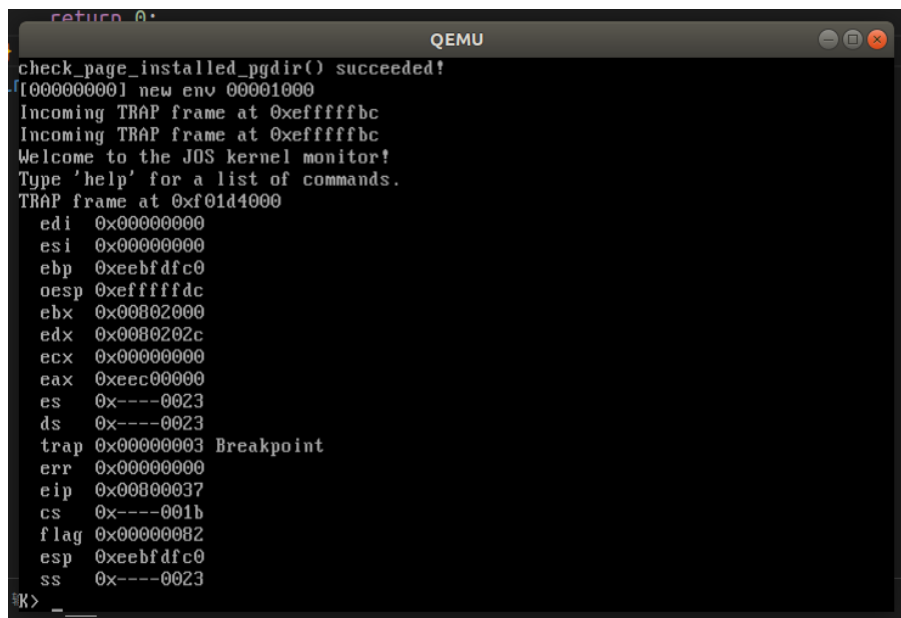
Figure 1: 运行 make run_breakpoint 的结果

```

ebp 0xeebdfdc0
oesp 0xefffffdc
ebx 0x00002000
edx 0x0000202c
ecx 0x00000000
eax 0xeec00000
es 0x---0023
ds 0x---0023
trap 0x00000003 Breakpoint
err 0x00000000
eip 0x00000037
cs 0x---001b
flag 0x00000082
esp 0xeebdfdc0
ss 0x---0023
K> ci
Unknown command 'ci'
K> continue
Incoming TRAP frame at 0xeffffbfc
[00001000] exiting gracefully
[00001000] free env 00001000
Destroyed the only environment - nothing more to do!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> s_

```

Figure 2: 运行 ci/continue 的结果



```
return 0;
check_page_installed_pgdir() succeeded!
[00000000] new env 00001000
Incoming TRAP frame at 0xeffffbfc
Incoming TRAP frame at 0xeffffbfc
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
TRAP frame at 0xf01d4000
  edi 0x00000000
  esi 0x00000000
  ebp 0xeebdfdc0
  oesp 0xefffffdc
  ebx 0x00002000
  edx 0x0000202c
  ecx 0x00000000
  eax 0xeec00000
  es  0x----0023
  ds  0x----0023
  trap 0x00000003 Breakpoint
  err 0x00000000
  eip 0x00000037
  cs  0x----001b
  flag 0x00000082
  esp 0xeebdfdc0
  ss  0x----0023
K> _
```

Figure 3: 运行 s/si 的结果