JOS_Lab3_Report

姓名：方嘉聪，学号：2200017849

I. Challenge

由于时间限制，我只实现了第二个challenge

主要在breakpoint中实现了单步调试的功能，即 `step` 和 `continue` 。注意在进入断点后会切换到 `qemu monitor` 中，所以我添加了两个 monitor function 分别用来处理 `step` 和 `continue` 功能。

1. 首先需要判断是否是在 user mode 在 breakpoint trap 后调用了 monitor function，下列代码依次判断 `Trapframe` 是否存在，是否在 user mode 下及是否为正确的 `trap` 类型。

```
if (!(tf && ((tf->tf_cs & 0x3) == 0x3) && (tf->tf_trapno == T_BRKPT || tf->tf_trapno == T_DEBUG)))
        return 0;
```

2. 根据文档所给的提示， `Trapframe->eflags` 寄存器包含了控制和状态的标志，其中 `TF/Trap Flag` 用于单步调试，在执行每条指令之后会产生一个调试异常 `T_DEBUG` 。因此对于 `step` 和 `continue` 分别设置和清空 `TF` 位即可。

```
tf->tf_eflags |= FL_TF; // For `step`
tf->tf_eflags &= ~FL_TF; // For `continue`
```

3. 注意为了使 `step` 可以正常运作，还需要修改 `kern/trap.c/trap_dispatch()` 将 `T_DEBUG` 的handler也设置为 `monitor()`

```
case T_DEBUG:
        cprintf("In Debug trap: Single Step at 0x%08x\n", tf->tf_eip);
        monitor(tf);
        return;
```

结果如下：

```
K> step
Step the kernel
Incoming TRAP frame at 0xeffffffc
In Debug trap: Single Step at 0x00800081
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
TRAP frame at 0xf01c7000
  edi  0x00000000
  esi  0x00000000
  ebp  0xeebfdff0
  oesp 0xefffffdc
  ebx  0x00802000
  edx  0x00000000
  ecx  0x00000000
  eax  0xeec00000
  es   0x----0023
  ds   0x----0023
  trap 0x00000001 Debug
  err  0x00000000
  eip  0x00800081
  cs   0x----001b
  flag 0x00000182
  esp  0xeebfdfc8
  ss   0x----0023
K> continue
Incoming TRAP frame at 0xeffffffc
[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.
```

II. Exercise

Exercise 1:

类似 Lab 2 实现 `env` array 分配即可，

```
envs = (struct Env *)boot_alloc(NENV * sizeof(struct Env));
memset(envs, 0, NENV * sizeof(struct Env));
```

Exercise 2:

依次实现下列函数

1. `env_init()`：难度不大，注意插入 `env_free_list` 的顺序即可
2. `env_setup_vm()`：模仿 `kern_pgdir`，在环境中初始化内核的虚拟内存排布，需要增加 `pp_ref`，见下

```
for (size_t i = PDX(UTOP); i < NPDENTRIES; ++i)
        e->env_pgdir[i] = kern_pgdir[i];
```

3. `region_alloc()`：分配 `len` bytes 的物理内存到 `env` 中的虚拟地址 `va`。注意虚拟地址需要与 `PGSIZE` 对齐。使用 lab2 中实现的 `page_alloc` 和 `page_insert` 即可。
4. `load_icode()`：为一个用户进程设置初始程序二进制数据、栈以及 `processor flags`。使用已经实现的 `region_alloc` 为相应的段分配内存内存，注意需要将二进制文件导入其中并将其他内存区域初始化为0，此外需设计程序入口：

```
// In a for loop:
// ...
region_alloc(e, (void *)ph->p_va, ph->p_memsz);
memset((void *)ph->p_va, 0, ph->p_memsz);
memcpy((void *)ph->p_va, binary + ph->p_offset, ph->p_filesz);
// ...

// Set reigster $eip to the entry point of the binary
e->env_tf.tf_eip = elf->e_entry;
```

5. `env_create`：创建一个新环境。调用 `env_alloc` 与 `load_icode` 即可。
6. `env_run`：切换环境并运行。不是太难，设置好 `curenv` 的相关参数，并调用 `lcr3` 设置对应寄存器实现地址空间的切换。

```
lcr3(PADDR(e->env_pgdir));
env_pop_tf(&e->env_tf);
```

Exercise 3&4

需要阅读 80836 Programmer's Manual 了解对中断和异常的处理方式。

- **The Interrupt Descriptor Table(IDT).** 内核将 IDT 设置在内核专用内存中，用于存储指向相应 handler 的相关值：**The instruction pointer register** `$eip`，**the code segment register** `CS`。在JOS中所有 exceptions 都在内核中处理，$\text{privilege level} = 0$;
- **The Task State Segment.** 用于存放保存中断或异常发生前的旧处理器状态，

The processor pushes (on this new stack) SS, ESP, EFLAGS, CS, EIP, and an optional error code. Then it loads the CS and EIP from the interrupt descriptor, and sets the ESP and SS to refer to the new stack.

在 **exercise 4** 中依次实现了下述内容：

- 在 `trapentry.S` 中实现了各个exception对应的 handler 与 `_alltraps:`

```
TRAPHANDLER_NOEC(handler_simderr, T_SIMDERR);
```

```
TRAPHANDLER(handler_dblft, T_DBLFLT);
```

- `kern/trap_init()` 中初始化 IDT，注意需要在这里声明 `trapentry.S` 中会用到的handler。在这里我直接依次使用 `SETGATE(idt[T_*], 0, GD_KT, &handler_*, 0);` 进行初始化。注意handler都在内核代码段中，selector 选择 `GD_KT`。
- 可以考虑将所有 handler 的声明写成一个数组形式，用 `for-loop` 简化代码。
- 这里我并未实现 Challenge 1 简化代码，需要额外引入一些宏。

Question 1

1. 由于有的异常中断（如 `double fault`）需要额外将一个 error code 压入栈中，为每个异常分别设置一个handler更方便处理。
2. 此外可以提供权限管理或隔离（Isolation）可以定义每个 handler 是否可以被用户程序触发，设置限制防止用户程序干扰或威胁内核运行。

Question 2

1. 不需要额外设置使得 `user/softint` 正常运行
2. trap 13为 general protection fault，trap 14 为 page fault，都需要内核权限 (privilege level = 0)，因此在用户程序尝试使用 `int $14`，会触发 `$14` 即通用的保护异常。
3. 如果允许用户程序可以触发 page_fault，那么用户可以借此干预虚拟内存，可能导致内核和整个程序的安全问题。

Exercise 5&6

实现Page Fault 和 Breakpoints Exceptions 的处理，修改 `trap_dispatch()` 即可实现，这里我用了一个 `switch`

```
switch (tf->tf_trapno){
        case T_PGFLT:
                page_fault_handler(tf);
                return;
        case T_BRKPT:
                monitor(tf);
                return;
        default:
                ...
}
```

注意这里在每个case后直接return就行，注意最好将 `unexpected trap` 放入 `default` 中。
此外需要在 `kern/trap_init()` 中修改 `breakpoint` 的 privilege level = 3，否则 `make grade`（用户态）无法正常触发。

Question 3

在 `breakpoint.asm` 中使用 `int $3` 触发 breakpoint exceptions，因此我们需要在 `kern/trap_init()` 中修改 `breakpoint` 的 privilege level = 3（即用户态），否则会触发 `$14` 通用保护异常 general protection fault。

Question 4

主要是为了实现隔离（Isolation），防止用户随意触发异常而产生的内核安全风险。

Exercise 7

实现对 `syscall` 的支持。按照文档的指引一步步实现即可。`kern/syscall.c/syscall()` 的的参数为 `syscalllno` 和最多五个参数，用一个 `switch` 分别处理四种 `syscall` 即可。
这里要注意 `T_SYSCALL` 是没有 error code的，在第一次实现时错误使用了 `TRAPHANDLER`，导致在测试 `hello` 是会无限循环输出 hello。

Exercise 8

很简单，在 `lib/libmain.c/libmain()` 设置 `thisenv` 即可

```
thisenv = &envs[ENVX(sys_getenvid())];
```

Exercise 9 & 10

进一步完善 page fault，确保内存之间的相互隔离。

1. 修改 `kern/trap.c`，使得如果在 kernel mode 中出现 page fault 则 panic，具体如下

```
if ((tf->tf_cs & 0x3) == 0)
        panic("Page fault in kernel mode at va %08x", fault_va);
```

2. 仿照 `kern/pmap.c/user_mem_assert` 实现 `user_mem_check` 即可。这里值得注意的是需要将 `user_mem_check_addr` 设置正确，我最初将其设置为

```
uintptr_t start_va = ROUNDDOWN((uintptr_t)va, PGSIZE);
for (uintptr_t cur_va = start_va; cur_va < end_va; cur_va += PGSIZE)
        ...
        user_mem_check_addr = cur_va;
        ...
```

事实上，需要判断一下 是否 `cur_va < (uintptr_t)va` 即正确代码为

```
if (cur_va < (uintptr_t)va)
        user_mem_check_addr = (uintptr_t)va;
else
        user_mem_check_addr = cur_va;
```

3. 按照助教给的blog修改了一个bug，这里不再赘述。

III. Result

顺利通过所有测试 见下：

```
make[1]: Leaving directory '/home/ubuntu/6.828/lab'
divzero: OK (1.3s)
softint: OK (1.0s)
badsegment: OK (1.0s)
Part A score: 30/30

faultread: OK (1.0s)
faultreadkernel: OK (1.0s)
faultwrite: OK (1.1s)
faultwritekernel: OK (1.0s)
breakpoint: OK (1.0s)
testbss: OK (1.0s)
hello: OK (1.0s)
buggyhello: OK (1.0s)
buggyhello2: OK (1.0s)
evilhello: OK (1.0s)
Part B score: 50/50

Score: 80/80
```