OSLab 3 Report

November 2, 2024

1 Overview

OSLab 3 focuses on implementing the fundamental kernel facilities needed to run a protected user-mode environment. In this lab, we will establish the necessary data structures to track user environments, create a single user environment, load a program image into it, and initiate its execution. Additionally, we will enable the JOS kernel to handle system calls made by the user environment and manage any exceptions that may occur.

2 Part A: User Environments and Exception Handling

2.1 Exercise 1

Allocating and mapping the `envs` array is straightforward, similar to what we did in Lab 2:

```
envs = (struct Env *) boot_alloc(NENV * sizeof(struct Env));
...
boot_map_region(kern_pgdir, UENVS, sizeof(struct Env)*NENV,
                PADDR(envs), PTE_U|PTE_P, 0);
```

2.2 Exercise 2

In this exercise, we need to implement the following functions:

- `env_init()`: Just reverse the order as suggested to maintain consistency.
- `env_setup_vm`: Based on the hint, we can simply use `memcpy` to copy the page directory from the kernel. This works because the VA space of all environments is identical above `UTOP` (except for `UVPT`, which we set separately). Finally, we map the environment's own page table as read-only at `UVPT`.

  ```
  e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_P | PTE_U;
  ```

- `region_alloc()`: Simply call `page_alloc()` and `page_insert()`.
- `load_icode()`: Follow the function in `boot/main.c`. To set the entry point, assign `e->env_tf.tf_eip` to the entry point. However, before loading the program and setting the entry point, switch to the current environment's page directory:

  ```
  lcr3(PADDR(e->env_pgdir));
  ```

  Otherwise, the JOS kernel will crash and continuously reboot.
- `env_create()`: Simply call `env_alloc()`, `load_icode()`, and set the environment type.
- `env_run()`: Follow the comments in the code.

2.3 Exercise 3 & 4 & Challenge 1

We first implement _alltraps in trapentry.S to handle all exceptions:

```
.text
.globl _alltraps
.align 2
_alltraps:
    # save %ds, %es, and pushal
    pushl %ds
    pushl %es
    pushal
    # load GD_KD into %ds and %es
    movw $GD_KD, %ax
    movw %ax, %ds
    movw %ax, %es
    # push %esp to pass a pointer
    pushl %esp
    # call trap
    call trap
    # a trap handler should not return
my_trap_spin:
    jmp my_trap_spin
```

To simplify the code, I wrote the trap names and their corresponding numbers in a table. I also defined macros to generate the trap handlers:

```
#define MY_NE(name, num) \
TRAPHANDLER_NOEC(name, num)
#define MY_E(name, num) \
TRAPHANDLER(name, num)
#include <kern/my_trap_data.txt>
#undef MY_NE
#undef MY_E
```

```
.data
#define MY_NE(name, num) .long name
#define MY_E(name, num) .long name
.globl trap_table
trap_table:
#include <kern/my_trap_data.txt>
#undef MY_NE
#undef MY_E
```

I had to use a txt file because directly defining:

```
#define table_list \
    MY_NE(th_divide, 0) \
    MY_NE(th_debug, 1)
```

would result in an error:

```
kern/trapentry.S:68: Error: junk '.globl th_debug' after expression.
```

I couldn't resolve this issue, so I opted to use a txt file to handle it.

Finally, we implemented trap_init() and set the privilege levels properly.

2.3.1 Question 1

We use different handlers for different exceptions because they push different values onto the stack.

2.3.2 Question 2

We set the privilege level of exception 14 to 0 so that when a user tries to trigger it, the kernel will handle it by calling exception 13.

3 Part B: Page Faults, Breakpoints Exceptions, and System Calls

3.1 Exercise 5 & 6 & Challenge 2

We simply choose the corresponding handler based on the value of tf->tf_trapno. The implementation of continue and step involves setting the FL_TF flag and rerunning the program.

3.1.1 Question 3

We set the privilege level of breakpoints to 3 so that when a user triggers it, the kernel will handle it directly via the breakpoints handler instead of raising a general protection fault.

3.1.2 Question 4

The goal is to allow the user to make system calls while restricting their privileges, ensuring they do not gain excessive control over the system.

3.2 Exercise 7 & Challenge 3

This exercise is straightforward: follow the requirements closely. One thing to note is that the error code for system calls is `0x30`, which is not consecutive with the other error codes. Initially, I mistakenly set it to 20, which led to considerable debugging effort. For the callenge 3, I implemented the **system_enter** following the instructions. However, I failed to revise the inline assembly code to use it. It's too late to solve it before the deadline, so I will leave it as is.

3.3 Exercise 8

This is a simple task of following the instructions and adding the code `thisenv = envs` as required.

3.4 Exercise 9 & 10

To handle kernel page faults, add the following code to `trap.c`:

```
if ((tf->tf_cs & 3) == 0) {
    panic("Kernel page fault at %08x\n", fault_va);
}
```

Additionally, follow the instructions to implement the `user_mem_check` function and revise the `debuginfo_eip` function as needed.