OS Lab3 Report

Author: Chengzu Ou (14302010037)

Description

There are two global variable in `kern/env.c`.

```
struct Env *envs = NULL; // All environments
struct Env *curenv = NULL; // The current env
static struct Env *env_free_list; // Free environment list
```

After JOS bootup, `envs` points to a list of `Env` which stores all environments. It uses `env_free_list` to manage free environments and `curenv` to represent current running environment.

Note that when implementing `env_init()`, we should add items in `envs` to `env_free_list` in reverse order so that the first we call `env_alloc()`, we will get `envs[0]`.

In `env_setup_vm`, we can use `kern_pgdir` as a template to set `env_pgdir`. But to better understand the memory layout, I choose to manually set it up.

In `region_alloc`, we should round down the start address and round up the end address of the region to allocate.

In `load_icode`, we use `lcr3()` to load the physical address of the starting point of environment directory table. And set `e_entry` to `e->env_tf.tf_eip` so that the environment start executing here.

When we are at kernel mode and we need a system call, we just have to directly call the function. But when we are at user mode, we need a system interrupt to do a system call.

To handle system call interrupt, we need to add a switch case in `trap_dispatch()`. Note that we need to call `syscall()` function in `kern/syscall.c`.

`syscall()` in `kern/syscall.c` serve as a dispatch to call other functions in the same file based on the `syscallno` argument passed by.

Answers

Q1: 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?)

Different exceptions/interrupts need different handler function because we handle them in different ways.

An *interrupt* is a protected control transfer that is caused by an asynchronous event usually external to the processor, such as notification of external device I/O activity. An *exception*, in contrast, is a protected control transfer caused synchronously by the currently running code, for example due to a divide by zero or an invalid memory access.

If all exceptions/interrupts were delivered to the same handler, we will not be able to differenciate between those exceptions that are caused by errors and those that are cause by interruptions and need to go back to the program after.

Q2: 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)?

This is because current program is running under user mode and at privilege level 3. Instruction INT is a system instruction and at privilege level 0. Program with privilege level 3 can't directly call program with privilege level 0. Otherwise, it will cause a general protection exception, a.k.a. trap 13.

Q3: 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?

If we set DPL of breakpoint exception entry in IDT to 3, it will generate a break point exception. It set it to 0, it will generate a general protection exception.

This is because when executing handler, we need current privileged level less than DPL. Otherwise, there will be a general protection exception. So we have to set it to 3 so that DPL is equal to CPL.

Q4: What do you think is the point of these mechanisms, particularly in light of what the `user/softint` test program does?

These machanisms is used to make sure that user programs do not directly call kernel programs to prevent unsafe calling.