Lab 3 User Environments

YANG Weikun 1100012442

PartA: 30th September 2013, 17:00 UTC+0800
PartB: 4th October 2013, 22:30 UTC+0800

Ex.1 Allocating the Environments Array

In `kern/pmap.c:mem_init()` , add those two lines:

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

I actually modified more code in `mem_init()` :

```
boot_map_region(kern_pgdir, KERNBASE, npages*PGSIZE, 0, PTE_W);
```

to map only available memory, so accessing above 128MB (default RAM size of QEMU) will cause a page fault instead of obscure behaviours. And in `check_kern_pgdir()` to suit the change:

```
// for (i = 0; i < NPDENTRIES; i++)
for (i = 0; i < npages / NPTENTRIES; i++)
```

Note: from this point, I will try not to quote too much code, for obvious reasons

- **Quoting too much code ruins your report! People don't really care about your exact implementation, they want an general description of your design, something you can't steal. Anyone want to see the code, they head for your code directly.**
- **Isn't it very nice to practice English writing on technical issues?**

The report becomes considerably shorter but nevertheless expressive.

Ex.2 Creating and Running Environments

In `env_init()` , memory occupied by `envs` is wiped clean, then all `struct Env` s are linked.

In `env_setup_vm()` , first a physical page is allocated for the `Env` 's page directory. Page directory entries for VA above `UTOP` is copied from `kern_pgdir` , the rest is unmapped. Then the reference count is incremented.

As for `region_alloc()` , that allocates memory for user ELF image, with user read/write permission. `va` and `len` are rounded appropriately.

Moving on to `load_icode()` . We must parse the ELF structure, and load appropriate segments into the `Env` 's VA, and reserve wiped space if necessary. Then we copy the ELF's entry point to `eip` of the `Env` 's trap frame. Finally we allocate 1 page for the program's stack. I used `lcr3` to switch the page directories, so that I can access user VA directly.

Next we have `env_create()` to sum up all the preparation.

Finally the `env_run()` that fall back to user program.

Now our `joe` should be able to drop to user program. See my debug session: User program was loaded at `0x800020` , call chain is like this (some omitted):

- `0x80002c call libmain<0x800060>`
    - `0x80008d call umain<0x800034>`
        - `0x800041 call cprintf<0x800171>`
            - `0x80018c call vcprintf<0x80010c>`
                - `0x800149 call vprintfmt<0x80035a>`
                    - `0x800fb5 sys_cputs`
                        - `800f4c syscall`

`syscall` emitted a instruction `int $T_SYSCALL` at `0x800f69` , then our CPU went into '*double fault*', without a proper handler, then into '*triple fault*' and halts.

Ex.3 Handling Interrupts and Exceptions

Book reading: done.

In short, how x86 processors handle interrupts is fairly complex. Intel gave interrupt, exceptions, faults and traps a common name '*Protected Control Transfer*' when such transfer happened in user mode. Whatever the cause, the processor uses two important data structures namely **Interrupt Descriptor Table** and **Task Segment Selector**: when an interrupt occurs(including traps etc.), the CPU locates the interrupt descriptor using interrupt number as a subscript to the *IDT*, checks permissions then saves machine state (`SS` , `ESP` , `EIP` etc.) to the location specified by *TSS*, and jump to code pointed by the corresponding interrupt descriptor.

When the interrupt occurred in kernel mode or inside an interrupt handler, the processor does pretty much the same (without changing to the stack specified by *TSS*).

Ex.4 Setting Up the IDT

I've written a little python script, to do the messy work automatically:

1. generate trap vectors in `kern/trapentry.S` using `kern/trapentry.tmpl` as template.
2. setup IDT entries in 'kern/trap.c' using `kern/trap.c.tmpl` as template.

This part of the logic is written in `kern/Makefrag`. Following instruction on MIT's page, we now have the IDT working.

Later I found a alternative way which is pretty elegant as well. A short description:

- Store the interrupt handlers as an array of function pointers, by mixing `.data` and `.text` assembler directives in the handler-generating macro.
- Use the macro to create the handlers, not to leave the unused slots so that the array stays continuous.
- In `trap.c`, use a `for` loop to install the IDT.

Q&A

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?)*
    - if all interrupts were first handled in the same place, there will be no way of telling them part. In the current implementation we relied on the first level handler `_alltraps` to store the interrupt number on stack, then call the generic handler `trap`.
    - Isn't it nice to have all interrupts be dispatched in a same place? So that the design becomes very clear and your code less messy.
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)?*
    - Nope, it happened just like that. Initiating a interrupt requires the '*Current Privilege Level*' be less or equal to the interrupt gate's '*Descriptor Privilege Level*'. In `kern/trap.c` we defined the all interrupt gates' DPL to be 0 (only accessible in kernel mode, by `int` instruction), except for the `breakpoint` gate. So issuing `int $14` will cause general protection fault instead of the requested page fault.
    - If the kernel allows arbitrary use of interrupt gates and trap gates, regardless of CPL, it would be very easy for user programs to disturb the kernel (for example, by repeatedly issuing divide by zero exception through `int $0`). The kernel must forbid such actions.

Ex.5 Handling Page Faults

In `kern/trap.c` `trap_dispatch()` I used `switch-case` to dispatch traps. Trap number is stored in `tf->tf_trapno`, for the case `T_PGFLT` we just need to call `page_fault_handler(tf)`.

The privilege level that the trap's initiator was in, is stored in the last 2 bits of `CS` segment selector. If the page fault was from the kernel (privilege level 0), we will print the trap-frame then panic. Otherwise the fault must be from user (privilege level 3), we handle this by printing the trap-frame then destroy the current environment.

Ex.6 The Breakpoint Exception

This one is dead simple: add a `case` in `trap_dispatch()` for `T_BRKPT`, then invoke `monitor(tf)`.

Q&A

1. *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?*
    - see the answer for question 1, ex. 4. Currently `T_BRKPT` and `T_SYSCALL` are the only two interrupts that can be invoked by `int` instruction in user mode, for their *DPL* being 3. If the *DPL* is 0, then `int $T_BRKPT` will cause a general protection interrupt.
2. *What do you think is the point of these mechanisms, particularly in light of what the user/softint test program does?*
    - I believe my answer for question 2, ex.4 is good for this one as well. The kernel must punish those user programs who abuse the `int` software interrupt instruction to create inconsistencies, by killing them right away without mercy.

Challenge

Break/Continue/Single-Stepping, and a general debugger for user programs

later.

Ex.7 System Calls

In function `syscall()` in `kern/syscall.c`, another `switch-case is` used to dispatch the system calls according to `syscallno`. For every case we convert and pass required arguments, then delegate return values if any. `-E_INVAL` is returned if a non-existing system call is requested.

Challenge

Fast system call by sysenter/sysexit instructions.

later.

Ex.8 User-mode startup

When we don't know where to add code, better search `// LAB 3: Your code here`. So convenient!

First we figure out our env_id by calling user mode helper `sys_getenvid()`, then modify `thisenv` accordingly.

Ex.9/10 Memory Protection

First part: kernel mode page fault is already handled in Ex.5. Second part: sanity checking of user's arguments.

In `kern/pmap.c`, we must implement `user_mem_check()` to verify the existence and permission of a range of memory in user space. `user_mem_assert()` will destroy the user `Env` if memory access violates. `debuginfo_eip` and various system calls use the sanity checks as well.