CS 444/544 OS II
Lab Tutorial #5
User Environments and Exception Handling
(Lab3 – Part A)
Before Start

• Lab 3 is for creating User environment
  • Will run your code in Ring 3!

• You will mainly edit kern/env.c

• Also,
  • kern/pmap.c
  • kern/trapentry.S
  • kern/syscall.c
  • lib/syscall.c
  • inc/syscall.h
Before Start

• If you are suffering an infinite loop of JOS, then try to check your path
  • It must have $HOME/bin as the path
    • Otherwise, it will execute OS’s default QEMU
  • We have to use a modified version of QEMU
    • Which is at under $HOME/bin/qemu-system-i386

• Easy solution: use BASH! (CS444/544 dotfiles includes that for you)
Checkout & Merge Lab2 into Lab3

- Follow instructions on lab3 document
  - $ git status
  - ... commit all changes since submitting lab2
  - $ git checkout lab3
  - $ git merge lab2
  - ...
  - Start Lab3!
Exercise 1: ENVs

Exercise 1. Modify `mem_init()` in `kern/pmap.c` to allocate and map the `envs` array. This array consists of exactly `NENV` instances of the `Env` structure allocated much like how you allocated the `pages` array. Also like the `pages` array, the memory backing `envs` should also be mapped user read-only at `UENVS` (defined in `inc/memlayout.h`) so user processes can read from this array.

You should run your code and make sure `check_kern_pgdir()` succeeds.

- Use `boot_alloc` to allocate ENVS (as we do for pages)

- Use `boot_map_region` to make
  - `envs` RW for kernel,
  - UENV R for both kernel and user
One tip

• Fix the line below if you have a weird memory error

```c
if (!nextfree) {
    extern char end[];
    nextfree = ROUNDUP((char *) end + 1, PGSIZE);
}
```

• In boot_alloc(),
  • Add +1 to the end...
  • This is for making the area for ‘envs’ array in-use
Exercise 2: Implement funcs for ENV

• env_init()

```c
// Mark all environments in 'envs' as free, set their env_ids to 0, // and insert them into the env_free_list. // Make sure the environments are in the free list in the same order // they are in the envs array (i.e., so that the first call to // env_alloc() returns envs[0]). //
void
env_init(void)
```

• Building linked-list (similar to page_free_list)
  • But, we need to keep the order (envs[0] is the first free one)
Exercise 2: Implement funcs for ENV

• env_setup_vm()

• Create a new page directory for an ENV
• Copy all kernel mappings above UTOP, and set UVPT

• Check pp_ref..
  • p->pp_ref += 1

```c
for(int i=PDX(UTOP); i<NPDESTRIES; ++i) {
  e->env_pgdire[i] = kern_pgdire[i];
}

// UVPT maps the env's own page table read-only.
// Permissions: kernel R, user R
e->env_pgdire[PDX(UVPT)] = PADDR(e->env_pgdire) | PTE_P | PTE_U;
```
Exercise 2: Implement funcs for ENV

• region_alloc(struct Env *e, void *va, size_t len)

• Similar to boot_map_region, but it is only virtually contiguous
  • boot_map_region allocates both physically and virtually contiguous memory

• Use functions wisely
  • Page_lookup()
  • Page_alloc()
  • Page_insert()
Exercise 2: Implement funcs for ENV

• load_icode()
  • Load an application program to memory space
  • Load to the environment’s memory space

• Now we are using kern_pgdir in running kernel
  • If we load the code in current memory space, it will be loaded to kernel

• We want to load the program to the memory space of ENV
  • We need to switch the page directory
Exercise 2: Implement funcs for ENV

• Change page directory from kern_pgdir to env’s pgdir

```c
// LAB 3: Your code here.
uint32_t prev_cr3 = rcr3();
lcr3(PADDR(e->env_pgdir));
```

• CR3 points to the current page directory
  • store previous cr3 (kern_pgdir) to prev_cr3
  • Load the page directory of the environment to CR3
    • cr3 = PADDR(e->env_pgdir)

• After that, we can access virtual memory space of the ENV
Exercise 2: Implement funcs for ENV

• load_icode()
• Get ELF Header:
  • Understand how ELF file is formatted...
    • Refer to how bootmain() in boot/main.c read the code
    • Use virtual address (from the header) for mapping
    • Set the entry point as
  ```
  e->env_tf.tf_eip = elf->e_entry;
  ```

• Use
  • memset, memcpy
  • region_alloc
Exercise 2: Implement funcs for ENV

• Don’t forget to restore the cr3 when returning from load_icode

```c
// LAB 3: Your code here.
uint32_t prev_cr3 = rcr3();
lcr3(PADDR(e->env_pgdir));
```

• Restore the cr3 to the previous value before returning from load_icode

```c
// change cr3 to previous one
lcr3(prev_cr3);
```
Exercise 2: Implement funcs for ENV

• env_create()
  • Allocate a new env, set type, and load binary

• Use
  • env_alloc()
  • load_icode()
Exercise 2: Implement funcs for ENV

- env_run()
  - Follow the comment...

```plaintext
// Step 1: If this is a context switch (a new environment is running):
//   1. Set the current environment (if any) back to
//      ENV_RUNNABLE if it is ENV_RUNNING (think about
//      what other states it can be in),
//   2. Set 'curenv' to the new environment,
//   3. Set its status to ENV_RUNNING,
//   4. Update its 'env_runs' counter,
//   5. Use lcr3() to switch to its address space.
// Step 2: Use env_pop_tf() to restore the environment's
//         registers and drop into user mode in the
//         environment.
```
Exercise 3: Read Intel Manual Chapter 6

- Do not have to read all but focus on Error Code and Interrupt numbers
  - IA-32 Developer's Manual

<table>
<thead>
<tr>
<th>Vector</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Type</th>
<th>Error Code</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#DE</td>
<td>Divide Error</td>
<td>Fault</td>
<td>No</td>
<td>DIV and IDIV instructions.</td>
</tr>
<tr>
<td>1</td>
<td>#DB</td>
<td>Debug Exception</td>
<td>Fault/ Trap</td>
<td>No</td>
<td>Instruction, data, and I/O breakpoints; single-step; and others.</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>NMI Interrupt</td>
<td>Interrupt</td>
<td>No</td>
<td>Nonmaskable external interrupt.</td>
</tr>
<tr>
<td>3</td>
<td>#BP</td>
<td>Breakpoint</td>
<td>Trap</td>
<td>No</td>
<td>INT 3 instruction.</td>
</tr>
<tr>
<td>4</td>
<td>#OF</td>
<td>Overflow</td>
<td>Trap</td>
<td>No</td>
<td>INTO instruction.</td>
</tr>
<tr>
<td>5</td>
<td>#BR</td>
<td>BOUND Range Exceeded</td>
<td>Fault</td>
<td>No</td>
<td>BOUND instruction.</td>
</tr>
<tr>
<td>6</td>
<td>#UD</td>
<td>Invalid Opcode (Undefined Opcode)</td>
<td>Fault</td>
<td>No</td>
<td>UD2 instruction or reserved opcode.</td>
</tr>
<tr>
<td>7</td>
<td>#NM</td>
<td>Device Not Available (No Math Coprocessor)</td>
<td>Fault</td>
<td>No</td>
<td>Floating-point or WAIT/FWAIT instruction.</td>
</tr>
<tr>
<td>8</td>
<td>#DF</td>
<td>Double Fault</td>
<td>Abort</td>
<td>Yes (zero)</td>
<td>Any instruction that can generate an exception, an NMI, or an INTR.</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>Coprocessor Segment Overrun (reserved)</td>
<td>Fault</td>
<td>No</td>
<td>Floating-point instruction.</td>
</tr>
<tr>
<td>10</td>
<td>#TS</td>
<td>Invalid TSS</td>
<td>Fault</td>
<td>Yes</td>
<td>Task switch or TSS access.</td>
</tr>
<tr>
<td>11</td>
<td>#NP</td>
<td>Segment Not Present</td>
<td>Fault</td>
<td>Yes</td>
<td>Loading segment registers or accessing system segments.</td>
</tr>
</tbody>
</table>
Exercise 4: Implement Trap Handlers

- kern/trapentry.S
  - Use MACROs to define handlers, depending on their error code existence

- If error code does not exist:

  \[
  \text{TRAPHANDLER}_\text{NOEC}(t\_\text{divide}, \text{T\_DIVIDE});
  \]

- If error code exists:

  \[
  \text{TRAPHANDLER}(t\_\text{dblflt}, \text{T\_DBLFLT});
  \]
Exercise 4: Implement Trap Handlers

- kern/trapentry.S

- Implement _alltraps:
  - Both TRAPHANDER_EC and TRAPHANDER_NOEC runs _alltraps
  - Push
    - ds
    - es
    - All general purpose registers
  - Change DS and ES to kernel DS ($GD_KD)
  - Push esp
  - Call trap() (kern/trap.c)

```
_alloptraps:
pushl %ds
pushl %es
pushal
movl $GD_KD, %eax
movw %ax, %ds
movw %ax, %es
```

1. push values to make the stack look like a struct Trapframe
2. load GD_KD into %ds and %es
3. pushl %esp to pass a pointer to the Trapframe as an argument to trap()
4. call trap (can trap ever return?)
Exercise 4: Implement Trap Handlers

• Please think about how we store trap context and use
  • struct TrapFrame

```c
struct PushRegs {
    /* registers as pushed by pusha */
    uint32_t reg_edi;
    uint32_t reg_esi;
    uint32_t reg_ebp;
    uint32_t reg_esp;    /* Useless */
    uint32_t reg_ebx;
    uint32_t reg_edx;
    uint32_t reg_ecx;
    uint32_t reg_eax;
} __attribute__((packed));
```
Exercise 4: Implement Trap Handlers

• kern/trap.c
  • Implement trap_init()
  • Use reference in comment, e.g.,

\[
\text{SETGATE(idt[T\_DIVIDE], 0, GD\_KT, t\_divide, 0);}
\]

• You must define t\_divide (for the above case) in trap.c

\[
\text{void t\_divide();}
\]

• Do this for all traps that you would like to handle...
Exercise 4: Implement Trap Handlers

- kern/trap.c
  - Implement trap_init()

- T_BRKPT and T_SYSCALL must be available to Ring 3
  - E.g.,

  ```c
  SETGATE(idt[T_SYSCALL], 0, GD_KT, t_syscall, 3);
  ```
Tips

• How to get the current pgdir?
  • `physaddr_t pgdir_addr = rcr3()`

• Virtual address of the current pgdir?
  • `KADDR(rcr3())`

• How to set the page directory to CR3?
  • `lcr3(PADDR(e->env_pgdir))`
How to Run USER Program?

```
make run-[NAME]-nox
    Run the program, e.g., make run-divzero-nox
make run-[NAME]-nox-gdb
    Run the program with gdb, e.g., make run-divzero-nox-gdb
```
make run-divzero-nox
How to debug USER program?

• Set a breakpoint at env_pop_tf
  • b env_pop_tf
  • c

• Then, trace it with ‘si’ upto iret
  • After iret, user execution starts!

• How to know about the semantics of the user program?
  • Open obj/user/program_name.asm