CS 444/544 OS II
Lab Tutorial #6
Page faults, Breakpoint Exceptions, and System Calls
(Lab3 – Part B)

Acknowledgement: Slides drawn heavily from Yeongjin Jiang
Before Start

• Triple Fault
  • Please attach GDB and trace where the error happens

• Commands
  • [terminal 1] make qemu-nox-gdb
  • [terminal 2] gdb
  • [terminal 2] c
  • Crashes...
  • [terminal 2] bt
  • Prints stack trace
Triple Fault – Use GDB

Look at those lines and reason about why it happens.
Trap in JOS

- Printing Trap Frame
  - Run ‘backtrace’ to see what’s happening

Look at those lines and reason about why it happens..
Hint: `_alltraps`

```plaintext
_alltraps:
pushl %ds
pushl %es
pushal
movl $GD_KD, %eax
movw %ax, %ds
movw %ax, %es
pushl %esp
call trap
```

Your `_alltraps` should:

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?)

Consider using the `pushal` instruction; it fits nicely with the layout of the `struct Trapframe`.
load_icode()

• Change your CR3 to env_pgdir
  • This will allow you to freely access env’s virtual memory space

• Do not forget to get the previous pgdir back to CR3

• At start

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

• At the end

```c
// change cr3 to previous one
lcr3(prev_cr3);
```
Writing Trap Handlers

• Implement handlers for
  • 0—8, 10—14, 16—19 and 48.
  • inc/trap.h

• Declare entries as functions
  • In kern/trap.c

```c
void t_divide();
void t_debug();
void t_nmi();
void t_brkpt();
void t_oflow();
void t_bound();
void t_illop();
void t_device();
void t_dblflt();

TRAPHANDLER(t_divide, T_DIVIDE);     // 8
TRAPHANDLER(t_debug, T_DEBUG);
TRAPHANDLER(t_nmi, T_NMI);
TRAPHANDLER(t_brkpt, T_BRKPT);
TRAPHANDLER(t_oflow, T_OFLOW);
TRAPHANDLER(t_bound, T_BOUND);
TRAPHANDLER(t_illop, T_ILLOP);
TRAPHANDLER(t_device, T_DEVICE);
TRAPHANDLER(t_dblflt, T_DBLFLT);     // 8

TRAPHANDLER(t_tss, T_TSS);          // 10
TRAPHANDLER(t_segnp, T_SEGNP);
TRAPHANDLER(t_stack, T_STACK);
TRAPHANDLER(t_gpflt, T_GPFLT);
TRAPHANDLER(t_pgflt, T_PGFILT);
TRAPHANDLER(t_fperr, T_FPE);        // 14

TRAPHANDLER(t_align, T_ALIGN);      // 17
TRAPHANDLER(t_mchk, T_MCHK);
TRAPHANDLER(t_simderr, T_STMDERR);

SETGATE(idt[T_DIVIDE], 0, GD_KT, t_divide, 0);
```
Exercise 5: Dispatch Page Fault

• Implement `trap_dispatch()

• You may wish to use `switch-case`

```c
// dispatch page_fault
switch (tf->tf_trapno) {
    case T_PGFLT:
    {
        return page_fault_handler(tf);
    }
}```
Exercise 6: Dispatch Breakpoint

• Implement trap_dispatch()

• You may wish to use switch-case

```c
    case T_BRKPT:
        {
            return monitor(tf);
        }
```
Exercise 7: System Calls

• syscall() in kern/syscall.c will invoke kernel functions

\[
\text{syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t a4, uint32_t a5)}
\]

• Arguments
  • syscallno = eax
  • a1 = edx
  • a2 = ecx
  • a3 = ebx
  • a4 = edi
  • a5 = esi

The system call number will go in %eax, %edx, %ecx, %ebx, %edi, and %esi.
Exercise 7: System Calls

• How to dispatch system call trap

• Read all register values from
  • Trapframe

• Invoke syscall()

```c
case T_SYSCALL:
{
    int32_t ret = syscall(tf->tf_regs.reg_eax,
                           tf->tf_regs.reg_edx,
                           tf->tf_regs.reg_ecx,
                           tf->tf_regs.reg_ebx,
                           tf->tf_regs.reg_edi,
                           tf->tf_regs.reg_esi);
    tf->tf_regs.reg_eax = ret;
    return;
}
```
Exercise 7: System Calls

- In syscall() kern/syscall.c
  - Dispatch system calls by eax and argument values

```c
switch (syscallno) {
  case SYS_cputs:
    {
      sys_cputs((const char *)a1, (size_t) a2);
      return 0;
    }
}
```
Exercise 9: Page Fault and Checks

• Panic at kernel page fault (in `page_fault_handler()`)
  • Kernel fault is when fault happens with last two digits of CS register value = 0

```c
if ((tf->tf_cs&0x3) == 0) {
```
Exercise 9: Page Fault and Checks

• Implement user_mem_check
  • Look at user_mem_assert first

```c
// Checks that environment 'env' is allowed to access the range // of memory [va, va+len) with permissions 'perm | PTE_U | PTE_P'. // If it can, then the function simply returns. // If it cannot, 'env' is destroyed and, if env is the current // environment, this function will not return.

void
user_mem_assert(struct Env *env, const void *va, size_t len, int perm)
{
    if (user_mem_check(env, va, len, perm | PTE_U) < 0) {
        printf("[\%08x] user_mem_check assertion failure for "
               "va %08x\n", env->env_id, user_mem_check_addr);
        env_destroy(env); // may not return
    }
}
```
Exercise 9: Page Fault and Checks

- Why do we implement user_mem_check?
  - Prevent user to access kernel memory...

```c
// Print a string to the system console.
// The string is exactly 'len' characters long.
// Destroys the environment on memory errors.
static void
sys_cputs(const char *s, size_t len)
{
// Check that the user has permission to read memory [s, s+len].
// Destroy the environment if not.

// LAB 3: Your code here.
user_mem_assert(curenv, s, len, PTE_U|PTE_P);

// Print the string supplied by the user.
cprintf("%.s", len, s);
}
```

Check if memory pointed by s is accessible by user
Exercise 9: Page Fault and Checks

• Apply user_mem_assert to
  • kern/sySCALL.c (in sys_cputs)
  • kern/kdebug.c (in debuginfo_eip)