CS 444/544 OS II
Lab Session #6

Page faults, Breakpoint Exceptions, and System Calls

(Lab3 – Part B)

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

```
Physical memory: 131072K available, base = 640K, extended = 130432K
EAX=00000000 EBX=00000000 ECX=00010000 EDX=00000000
ESI=00000000 EDI=00000000 EBP=f0114f78 ESP=f0114f6c
EIP=f0103a83 EFL=00000006 [-----P-] CPL=0 II=0 A20=1 SMM=0 HLT=0
ES =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
CS =0008 00000000 ffffffff 00cf9a00 DPL=0 CS32 [-R-]
SS =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
DS =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
FS =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
GS =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
LDT=0000 00000000 0000ffff 00008200 DPL=0 LDT
TR =0000 00000000 0000ffff 00008b00 DPL=0 TSS32-busy
GDT=     00007c4c 00000017
IDT=     00000000 000003ff
CR0=80010011 CR2=00000000 CR3=00115000 CR4=00000000
DR0=00000000 DR1=00000000 DR2=00000000 DR3=00000000
DR6=ffff0ff0 DR7=00000400
EFER=0000000000000000
Triple fault.  Halting for inspection via QEMU monitor.
```

```
0xf0103a83 in memset (v=0x0, c=0, n=262144) at lib/string.c:131
131                     asm volatile("cld; rep stosl\n"
>>> bt
#0  0xf0103a83 in memset (v=0x0, c=0, n=262144) at lib/string.c:131
#1  0xf0101382 in mem_init () at kern/pmap.c:172
#2  0xf0100086 in i386_init () at kern/init.c:30
#3  0xf010003e in relocated () at kern/entry.S:80
>>>
```

Look at those lines and reason about why it happens..

Trap in JOS

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

```
[00000000] new env 00001000
Found other runnable at: 0
TRAP frame at 0xf02b4000 from CPU 0
  edi  0x00000023
  esi  0x00000023
  ebp  0x00000030
  oesp 0x00000000
  ebx  0x00800e2f
  edx  0x0000001b
  ecx  0x00000286
  eax  0xeebfdfc4
  es   0x----0023
  ds   0x----ff53
  trap 0xf000ff53 (unknown trap)
  err  0xf000e2c3
  eip  0xf000ff53
  cs   0x----ff53
  flag 0xf000ff53
  esp  0xf000ff53
  ss   0x----ff53
[00001000] free env 00001000
No runnable environments in the system!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
```

```
K> backtrace
Stack backtrace:
  ebp efffffc0  eip f0100b0c  args 00000001 efffed8 00000000 00000000 f0230a80
         kern/monitor.c:157: monitor+275
  ebp efffff30  eip f0104f85  args 00000000 003af000 efffff80 f0103da6 f0275dd8
         kern/sched.c:74: sched_halt+74
  ebp efffff50  eip f01050ff  args 000003af 00000ee8 000003bb ee800000 f01085c8
         kern/sched.c:53: sched_yield+238
  ebp efffff80  eip f0103e1b  args f02b4000 0000ff53 f01084d0 00000000 00000000
         kern/env.c:519: env_destroy+88
  ebp efffffa0  eip f0104df1  args f02b4000 00000000 00000000 00000000 00000000
         kern/trap.c:384: trap+547
  ebp efffffd0  eip f0104f3b  args efffffdc 00000023 00000023 00000030 00000000
         kern/sched.c:60: sched_halt+0
```

Look at those lines and reason about why it happens..

Hint: _alltraps

```
_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

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

  - At the end

```
// change cr3 to previous one
lcr3(prev_cr3);
```

Exercise 5: Dispatch Page Fault

- Implement trap_dispatch()

- You may wish to use switch-case

```
// 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

```
case T_BRKPT:
{
    return monitor(tf);
}
```

Exercise 7: System Calls

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

```
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()

```
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

```
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
  - Kernel fault is when fault happens with last two digits of CS register value = 0

```
if ((tf->tf_cs&0x3) == 0) {
```

Exercise 9: Page Fault and Checks

- Implement user_mem_check
  - Look at user_mem_assert first

```
//
// 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) {
        cprintf("[%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…

Check if memory pointed by s is accessible by user

```
// 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);
}
```

Exercise 9: Page Fault and Checks

- Apply checks to
  - kern/syscall.c
  - kern/kdebug.c