System Calls
Interface and Implementation

Learning Outcomes

• A high-level understanding of System Call interface
  • Mostly from the user's perspective
  • From textbook (section 1.6)
• Understanding of how the application-kernel boundary is crossed with system calls in general
  • Including an appreciation of the relationship between a case study (OS/163 system call handling) and the general case.
• Exposure architectural details of the MIPS R3000
  • Detailed understanding of the of exception handling mechanism
  • From "Hardware Guide" on class web site

System Calls

• Can be viewed as special function calls
  • Provides for a controlled entry into the kernel
  • While in kernel, they perform a privileged operation
  • Returns to original caller with the result
• The system call interface represents the abstract machine provided by the operating system.

The Structure of a Computer System

The System Call Interface: A Brief Overview

• From the user’s perspective
  • Process Management
  • File I/O
  • Directories management
  • Some other selected Calls
• There are many more
  • On Linux, see `man syscall` for a list
Some System Calls For Process Management

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pid = fork()</td>
<td>Create a child process identical to the parent</td>
</tr>
<tr>
<td>pid = waitpid(pid, &amp;status, options)</td>
<td>Wait for a child to terminate</td>
</tr>
<tr>
<td>s = execlp(name, argv, environ)</td>
<td>Replace a process’ core image</td>
</tr>
<tr>
<td>exit(status)</td>
<td>Terminate process execution and return status</td>
</tr>
</tbody>
</table>

Some System Calls For File Management

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fd = open(file, how, ...)</td>
<td>Open a file for reading, writing or both</td>
</tr>
<tr>
<td>s = close(fd)</td>
<td>Close an open file</td>
</tr>
<tr>
<td>n = read(fd, buffer, nbyte)</td>
<td>Read data from a file into a buffer</td>
</tr>
<tr>
<td>n = write(fd, buffer, nbyte)</td>
<td>Write data from a buffer to a file</td>
</tr>
<tr>
<td>position = lseek(fd, offset, whence)</td>
<td>Move the file pointer</td>
</tr>
<tr>
<td>s = stat(name, &amp;buf)</td>
<td>Get a file’s status information</td>
</tr>
</tbody>
</table>

System Calls

• A stripped down shell:
  ```
  while (TRUE) {
    type_prompt(); /* display prompt */
    read_command(command, parameters); /* input from terminal */
    if (fork()) != 0) { /* fork off child process */
      /* Parent code */
      waitpid(-1, &status, 0); /* wait for child to exit */
    } else {
      /* Child code */
      execute(command, parameters, 0); /* execute command */
    }
  }
  ```

System Call Implementation

Crossing user-kernel boundary

Some Win32 API calls

<table>
<thead>
<tr>
<th>UNIX</th>
<th>Win32</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fork</td>
<td>CreateThread</td>
<td>Create a new thread</td>
</tr>
<tr>
<td>waitpid</td>
<td>WaitForObject</td>
<td>Wait for a process to exit</td>
</tr>
<tr>
<td>exec</td>
<td>CreateProcess</td>
<td>Create a new process</td>
</tr>
<tr>
<td>exit</td>
<td>TerminateProcess</td>
<td>Terminate execution</td>
</tr>
<tr>
<td>open</td>
<td>CreateFile</td>
<td>Create a file or open an existing file</td>
</tr>
<tr>
<td>close</td>
<td>CloseHandle</td>
<td>Close a file</td>
</tr>
<tr>
<td>read</td>
<td>ReadFile</td>
<td>Read data from a file</td>
</tr>
<tr>
<td>write</td>
<td>WriteFile</td>
<td>Write data to a file</td>
</tr>
<tr>
<td>seek</td>
<td>SeekFile</td>
<td>Move the file pointer</td>
</tr>
<tr>
<td>getfilesize</td>
<td>GetFileSize</td>
<td>Get various file attributes</td>
</tr>
<tr>
<td>mkdir</td>
<td>CreateDirectory</td>
<td>Create a new directory</td>
</tr>
<tr>
<td>rmdir</td>
<td>RemoveDirectory</td>
<td>Remove an empty directory</td>
</tr>
<tr>
<td>link</td>
<td>CreateShortcut</td>
<td>Create a new shortcut</td>
</tr>
<tr>
<td>unlink</td>
<td>DeleteFile</td>
<td>Delete a file</td>
</tr>
<tr>
<td>map_file</td>
<td>MapFile</td>
<td>Map file to memory</td>
</tr>
<tr>
<td>must</td>
<td>OpenFileMap</td>
<td>Open file (handles)</td>
</tr>
<tr>
<td>perm</td>
<td>CreateFileMapping</td>
<td>Map file to memory</td>
</tr>
<tr>
<td>shdr</td>
<td>GetServiceDirectory</td>
<td>Get service directory</td>
</tr>
<tr>
<td>sid</td>
<td>GetToken</td>
<td>Get token for current user</td>
</tr>
<tr>
<td>sids</td>
<td>GetToken</td>
<td>Get token for current user</td>
</tr>
<tr>
<td>time</td>
<td>GetSystemTime</td>
<td>Get current time</td>
</tr>
</tbody>
</table>

A Simple Model of CPU Computation

• The fetch-execute cycle
  • Load memory contents from address in program counter (PC)
  • The instruction
  • Execute the instruction
  • Increment PC
  • Repeat
A Simple Model of CPU Computation

- Stack Pointer (SP)
- Status Register
  - Condition codes
    - Positive result
    - Zero result
    - Negative result
- General Purpose Registers
  - Holds operands of most instructions
  - Enables programmers (compiler) to minimise memory references.

Privileged-mode Operation

- To protect operating system execution, two or more CPU modes of operation exist
  - Privileged mode (system-, kernel-mode)
    - All instructions and registers are available
  - User mode
    - Uses ‘safe’ subset of the instruction set
      - Only affects the state of the application itself
      - They cannot be used to uncontrollably interfere with OS
    - Only ‘safe’ registers are accessible

CPU Registers

<table>
<thead>
<tr>
<th>PC</th>
<th>0x0300</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
<td>0xcbf3</td>
</tr>
<tr>
<td>Status</td>
<td>R1</td>
</tr>
<tr>
<td></td>
<td>i</td>
</tr>
<tr>
<td></td>
<td>Rn</td>
</tr>
</tbody>
</table>

Example Unsafe Instruction

- “cli” instruction on x86 architecture
  - Disables interrupts
- Example exploit
  
```c
cli /* disable interrupts */
while (true)
  /* loop forever */;
```

Privileged-mode Operation

- The accessibility of addresses within an address space changes depending on operating mode
  - To protect kernel code and data
  - Note: The exact memory ranges are usually configurable, and vary between CPU architectures and/or operating systems.

Memory Address Space

<table>
<thead>
<tr>
<th>Accessible only to Kernel-mode</th>
<th>0xFFFFFFFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessible to User- and Kernel-mode</td>
<td>0x80000000</td>
</tr>
<tr>
<td>0x00000000</td>
<td>16</td>
</tr>
</tbody>
</table>

Questions we’ll answer

- There is only one register set
  - How is register use managed?
  - What does an application expect a system call to look like?
- How is the transition to kernel mode triggered?
- Where is the OS entry point (system call handler)?
- How does the OS know what to do?
System Call Mechanism Overview

- System call transitions triggered by special processor instructions
  - User to Kernel
  - System call instruction
  - Kernel to User
  - Return from privileged mode instruction

- Processor mode
  - Switched from user-mode to kernel-mode
  - Switched back when returning to user-mode

- Stack Pointer (SP)
  - User-level SP is saved and a kernel SP is initialised
  - User-level SP restored when returning to user-mode

- Program Counter (PC)
  - User-level PC is saved and PC set to kernel entry point
  - User-level PC restored when returning to user-level
  - Kernel entry via the designated entry point must be strictly enforced

- Registers
  - Set at user-level to indicate system call type and its arguments
  - A convention between applications and the kernel
  - Some registers are preserved at user-level or kernel-level in order to restart user-level execution
    - Depends on language calling convention etc.
    - Result of system call placed in registers when returning to user-level
    - Another convention
  - A convention between applications and the kernel
    - Some registers are preserved at user-level or kernel-level in order to restart user-level execution
    - Result of system call placed in registers when returning to user-level
    - Another convention

- Why do we need system calls?
  - Why not simply jump into the kernel via a function call???
    - Function calls do not
      - Change from user to kernel mode
      - and eventually back again
      - Restrict possible entry points to secure locations
        - To prevent entering after any security checks

Steps in Making a System Call

There are 11 steps in making the system call `read(fd, buffer, nbytes)`

- The MIPS R2000/R3000
  - Before looking at system call mechanics in some detail, we need a basic understanding of the MIPS R3000
Coprocessor 0

- The processor control registers are located in CP0
  - Exception/Interrupt management registers
  - Translation management registers
- CP0 is manipulated using mtc0 (move to) and mfc0 (move from) instructions
  - mtc0/mfc0 are only accessible in kernel mode.

**CP0 Registers**

- **Exception Management**
  - c0_cause
    - Cause of the recent exception
  - c0_status
    - Current status of the CPU
  - c0_epc
    - Address of the instruction that caused the exception
  - c0_linkaddr
    - Address accessed that caused the exception
- **Miscellaneous**
  - c0_prid
    - Processor identifier
- **Memory Management**
  - c0_index
  - c0_random
  - c0_entryhi
  - c0_entrylo
  - c0_context
  - More about these later in course

### c0_status

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>IM</td>
</tr>
<tr>
<td>32</td>
<td>KU</td>
</tr>
<tr>
<td>33</td>
<td>IE</td>
</tr>
<tr>
<td>34</td>
<td>c, p, o</td>
</tr>
<tr>
<td></td>
<td>Current, previous, old</td>
</tr>
</tbody>
</table>

**IM**
- Individual interrupt mask bits
  - 0 = kernel
  - 6 external
  - 2 software

**KU**
- 0 = all interrupts masked
  - 1 = interrupts enabled
  - Mask determined via IM bits

**IE**
- 0 = all interrupts masked
  - 1 = interrupts enabled

### c0_cause

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>IP</td>
</tr>
<tr>
<td>32</td>
<td>BD</td>
</tr>
<tr>
<td>33</td>
<td>BD</td>
</tr>
<tr>
<td>34</td>
<td>CE</td>
</tr>
<tr>
<td>35</td>
<td>BD</td>
</tr>
</tbody>
</table>

**IP**
- Interrupts pending
  - 8 bits indicating current state of interrupt lines

**CE**
- Coprocessor error
  - Attempt to access disabled Copro.

**BD**
- If set, the instruction that caused the exception was in a branch delay slot

**ExcCode**
- The code number of the exception taken

### Exception Codes

<table>
<thead>
<tr>
<th>ExcCode Value</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Int</td>
<td>Interrupt</td>
</tr>
<tr>
<td>1</td>
<td>Mod</td>
<td>TLB modify</td>
</tr>
<tr>
<td>2</td>
<td>TLBE</td>
<td>TLB hardware fault</td>
</tr>
<tr>
<td>3</td>
<td>TID</td>
<td>TLB dirty</td>
</tr>
<tr>
<td>4</td>
<td>MRU</td>
<td>Address error on load/store. 1 fetch or store 2</td>
</tr>
<tr>
<td>5</td>
<td>MRS</td>
<td>Either an attempt to access outside/::keng when in user mode, or an attempt to read a word or half word at an unaligned address.</td>
</tr>
</tbody>
</table>

Table 3.2. ExcCode values: different kinds of exceptions
### Exception Codes

<table>
<thead>
<tr>
<th>Exception Value</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>SIEE</td>
<td>Bus error (instruction or data fault), respectively.</td>
</tr>
<tr>
<td>7</td>
<td>DBUR</td>
<td>External hardware has signaled an error of some kind.</td>
</tr>
<tr>
<td>8</td>
<td>Syscall</td>
<td>Generated unconditionally by a system instruction.</td>
</tr>
<tr>
<td>9</td>
<td>Hbp</td>
<td>Breakpoint - a break instruction.</td>
</tr>
<tr>
<td>10</td>
<td>RI</td>
<td>&quot;reserved instruction&quot;.</td>
</tr>
<tr>
<td>11</td>
<td>Cpu</td>
<td>&quot;CoProcessor unavailable&quot;.</td>
</tr>
<tr>
<td>12</td>
<td>Ox</td>
<td>Arithmetic overflow. Note that &quot;unexpanded&quot; version of instructions e.g., add h never cause this exception.</td>
</tr>
<tr>
<td>13:31</td>
<td>Reserved</td>
<td>Some are already defined for MIPS CPUs such as the R10000 and R12KX.</td>
</tr>
</tbody>
</table>

Table 3.2. Encoder values: different kinds of exceptions

### Exception Vectors

<table>
<thead>
<tr>
<th>Program Address</th>
<th>“segment”</th>
<th>Physical Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000 0000</td>
<td>kseg0</td>
<td>0x0000 0000</td>
<td>TLB miss on kseg reference only.</td>
</tr>
<tr>
<td>0x0000 0080</td>
<td>kseg0</td>
<td>0x0000 0080</td>
<td>All other exceptions.</td>
</tr>
<tr>
<td>0x60c0 0100</td>
<td>kseg1</td>
<td>0x60c0 0100</td>
<td>Uncoached alternative kseg TLB miss entry point used if 32-bit EIM set.</td>
</tr>
<tr>
<td>0x60c0 0180</td>
<td>kseg1</td>
<td>0x60c0 0180</td>
<td>Uncoached alternative for all other exceptions, used if 50-bit EIM set.</td>
</tr>
<tr>
<td>0x60c0 0000</td>
<td>kseg0</td>
<td>0x60c0 0000</td>
<td>The &quot;reset&quot; exception.</td>
</tr>
</tbody>
</table>

Table 4.1. Reset and exception entry points (vectors) for R10000 family

### Hardware Exception Handling

**User Mode**

- **PC**: 0x12345678
- **EPC**: ?
- **Cause**: ?
- **Status**: KUo IEo KUp IEp KUc IEc

**Kernel Mode**

- **PC**: 0x12345678
- **EPC**: ?
- **Cause**: ?
- **Status**: KUo IEo KUp IEp KUc IEc

### c0_epc

- **The Exception Program Counter**
  - Points to address of where to restart execution after handling the exception or interrupt

- **Example**
  - Assume `sw r3, (r4)` causes a restartable fault exception

Aside: We are ignore BD-bit in c0_cause which is also used in reality on rare occasions.

### Simple Exception Walk-through

- **User Mode**
  - Instruction address at which to restart after the interrupt is transferred to EPC
  - **PC**: 0x12345678
  - **EPC**: ?
  - **Cause**: ?
  - **Status**: KUo IEo KUp IEp KUc IEc

- **Kernel Mode**
  - Instruction address at which to restart after the interrupt is transferred to EPC
  - **PC**: 0x12345678
  - **EPC**: ?
  - **Cause**: ?
  - **Status**: KUo IEo KUp IEp KUc IEc
Hardware exception handling

Kernel Mode is set, and previous mode shifted along

Interrupts disabled and previous state shifted along

PC 0x12345678

Cause

Status

KUo IEo KUp IEp KUc IEc

0 KUo KUp KUc IEc

0

1

1


PC 0x80000080

Address of general exception vector placed in PC

CPU is now running in kernel mode at 0x80000080, with interrupts disabled

All information required to find out what caused the exception

Restart after exception handling is in coprocessor registers

Code for the exception placed in Cause. Note Interrupt code = 0

Returning from an exception

For now, let’s ignore

• how the exception is actually handled
• how user-level registers are preserved

Let’s simply look at how we return from the exception

This code to return is

lw r27, saved_epc
nop
jr r27
rfe

Load the contents of EPC which is usually moved earlier to somewhere in memory by the exception handler
Returning from an exception

• This code to return is:
  ```assembly
  lw r27, saved_epc
  nop
  jr r27
  rfe
  ```

Store the EPC back in the PC

MIPS System Calls

• System calls are invoked via a syscall instruction.
  • The syscall instruction causes an exception and transfers control to the general exception handler
  • A convention (an agreement between the kernel and applications) is required as to how user-level software indicates:
    - Which system call is required
    - Where its arguments are
    - Where the result should go

OS/161 Systems Calls

• OS/161 uses the following conventions
  • Arguments are passed and returned via the normal C function calling convention
  • Additionally:
    - Reg v0 contains the system call number
    - On return, reg a3 contains:
      - 0: if success, v0 contains successful result
      - not 0: if failure, v0 has the error
      - v0 stored in error
      - -1 returned is valid

Preserved for kernel entry

Preserved for C calling convention

Success?

Result

SysCall No.
User-Level System Call Walk Through – Calling read()

```c
int read(int filehandle, void *buffer, size_t size)
```

• Three arguments, one return value

Code fragment calling the read function

```assembly
move a0, s3
addiu a1, sp, 16
jal 40068c <read>
li a2, 1024
move s0, v0
blez s0, 400194 <docat+0x94>
```

*Args are loaded, return value is tested*
The read() syscall function
part 2

00400640 <__syscall>:
400640: 0000000c syscall
400644: 10e00005 beqz a3,40065c <__syscall+0x1c>
400648: 00000000 nop
40064c: 3c011000 lui at,0x1000
400650: ac220000 sw v0,0(at)
400654: 2403ffff li v1,-1
400658: 2402ffff li v0,-1
40065c: 03e00008 jr ra
400660: 00000000 nop

Set read() result to -1
Return to location after where read() was called

Summary

- From the caller’s perspective, the read() system call behaves like a normal function call
- It preserves the calling convention of the language
- However, the actual function implements its own convention by agreement with the kernel
- Our OS/161 example assumes the kernel preserves appropriate registers(s0-s8, sp, gp, ra).
- Most languages have similar libraries that interface with the operating system.

System Calls - Kernel Side

- Things left to do
  - Change to kernel stack
  - Preserve registers by saving to memory (on the kernel stack)
  - Leave saved registers somewhere accessible to
  - Read arguments
  - Store return values
  - Do the “read()”
  - Restore registers
  - Switch back to user stack
  - Return to application

OS/161 Exception Handling

- Note: The following code is from the uniprocessor variant of OS161 (v1.x).
- Simpler, but broadly similar to current version.

```assembly
exception:
    move k1, sp /* Save previous stack pointer in k1 */
    mflo k0, cp_always /* Get status register */
    andi k0, k0, CST_Kup /* Check the we-were-in-user-mode bit */
    beq k0, $0, 1f /* If clear, from kernel, already have stack */
    nop /* delay slot */

1: /* Coming from user mode */
    la k0, curkstack /* load k0, curkstack */
    lw sp, 0(k0) /* Get its value */
    nop /* delay slot */

mflo k0, cp_cause /* Load the exception cause */
    j common_exception /* Skip to common code */
    nop
```
move k1, sp  /* Save previous stack pointer in k1 */
mfc0 k0, c0_status  /* Get status register */
andi k0, k0, CST_Kup  /* Check the we-were-in-user-mode bit */
beq k0, $0, 1f  /* If clear, from kernel, already have stack */
nop  /* Delay slot */
/* Coming from user mode - load kernel stack into sp */
la k0, curkstack  /* get address of "curkstack" */
lw sp, 0(k0)  /* get its value */
nop  /* Delay slot for the load */
1:
mfc0 k0, c0_cause  /* Now, load the exception cause. */
j common_exception  /* Skip to common code */
nop  /* Delay slot */

The order here must match mips/include/trapframe.h. */
sw ra, 140(sp)  /* Dummy for gdb */
sw s8, 156(sp)  /* Save s8 */
sw sp, 148(sp)  /* Save gp */
sw k1, 144(sp)  /* Dummy for gdb */
sw k0, 140(sp)  /* Dummy for gdb */
sw k1, 152(sp)  /* Real saved sp */
nop  /* Delay slot for store */
mfc0 k0, c0_mpc  /* Core.0 reg 13 == PC for exception */
sw k1, 140(sp)  /* Real saved pc */

The real work starts here
sw ra, 140(sp)  /* Dummy for gdb */
sw s8, 156(sp)  /* Save s8 */
sw sp, 152(sp)  /* Dummy for gdb */
sw gp, 148(sp)  /* Save gp */
sw k0, 140(sp)  /* Dummy for gdb */
sw k1, 144(sp)  /* Dummy for gdb */
sw k0, 140(sp)  /* Dummy for gdb */
sw k1, 152(sp)  /* Real saved sp */
nop  /* Delay slot for store */
mfc0 k1, c0_mpc  /* Core.0 reg 13 == PC for exception */
sw k1, 140(sp)  /* Real saved pc */

The order here must match mips/include/trapframe.h. */
sw ra, 140(sp)  /* Dummy for gdb */
sw s8, 156(sp)  /* Save s8 */
sw sp, 152(sp)  /* Dummy for gdb */
sw gp, 148(sp)  /* Save gp */
sw k0, 140(sp)  /* Dummy for gdb */
sw k1, 144(sp)  /* Dummy for gdb */
sw k0, 140(sp)  /* Dummy for gdb */
sw k1, 152(sp)  /* Real saved sp */
nop  /* Delay slot for store */
mfc0 k1, c0_mpc  /* Core.0 reg 13 == PC for exception */
sw k1, 140(sp)  /* Real saved pc */

The real work starts here

Save all the registers on the kernel stack

These six stores are a "hack" to avoid confusing GDB. You can ignore the details of why and how.

Save special registers.

Save remaining exception context information.

/*
 * Pretend to save $0 for gdb's benefit.
 */
sw $0, 12(sp)
Now we arrive in the ‘C’ kernel

/* General trap (exception) handling function for MIPS.
 * This is called by the assembly-language exception handler once
 * the trapframe has been set up.
 */

void mips_trap(struct trapframe *tf)
{
    u_int32_t code, isutlb, iskern;
    int savespl;
    /* The trap frame is supposed to be 37 registers long. */
    assert(sizeof(struct trapframe)==(37*4));
    /* Save the value of curspl, which belongs to the old context. */
    savespl = curspl;
    /* Right now, interrupts should be off. */
    curspl = SPL_HIGH;

    /* 16(sp) no need to restore tf_vaddr */
    lw t0, 20(sp) /* load status register value into t0 */
    nop /* load delay slot */
    mtc0 t0, c0_status /* store it back to coprocessor 0 */
    /* 24(sp) no need to restore tf_cause */
    /* restore special registers */
    lw t1, 28(sp)
    lw t0, 32(sp)
    mtlo t1
    mthi t0
    /* load the general registers */
    lw ra, 36(sp)
    lw AT, 40(sp)
    lw v0, 44(sp)
    lw v1, 48(sp)
    lw a0, 52(sp)
    lw a1, 56(sp)
    lw a2, 60(sp)
    lw a3, 64(sp)
    lw t0, 68(sp)
    lw t1, 72(sp)
    lw t2, 76(sp)
    lw t3, 80(sp)
    lw t4, 84(sp)
    lw t5, 88(sp)
    lw t6, 92(sp)
    lw t7, 96(sp)
    lw s0, 100(sp)
    lw s1, 104(sp)
    lw s2, 108(sp)
    lw s3, 112(sp)
    lw s4, 116(sp)
    lw s5, 120(sp)
    lw s6, 124(sp)
    lw s7, 128(sp)
    lw t8, 132(sp)
    lw t9, 136(sp)
    /* 140(sp) “saved” k0 was dummy garbage anyway */
    /* 144(sp) “saved” k1 was dummy garbage anyway */
    /* exception_return:
     * 16(sp) no need to restore tf_vaddr */
    lw t0, 20(sp) /* load status register value into t0 */
    nop /* load delay slot */
    /* 24(sp) no need to restore tf_cause */
    /* restore special registers */
    lw t1, 28(sp)
    lw t0, 32(sp)
    mtlo t1
    mthi t0
    /* load the general registers */
    lw ra, 36(sp)
    lw AT, 40(sp)
    lw v0, 44(sp)
    lw v1, 48(sp)
    lw a0, 52(sp)
    lw a1, 56(sp)
    lw a2, 60(sp)
    lw a3, 64(sp)
    lw t0, 68(sp)
    lw t1, 72(sp)
    lw t2, 76(sp)
    lw t3, 80(sp)
    lw t4, 84(sp)
    lw t5, 88(sp)
    lw t6, 92(sp)
    lw t7, 96(sp)
    lw s0, 100(sp)
    lw s1, 104(sp)
    lw s2, 108(sp)
    lw s3, 112(sp)
    lw s4, 116(sp)
    lw s5, 120(sp)
    lw s6, 124(sp)
    lw s7, 128(sp)
    lw t8, 132(sp)
    lw t9, 136(sp)
    /* 140(sp) “saved” k0 was dummy garbage anyway */
    /* 144(sp) “saved” k1 was dummy garbage anyway */

What happens next?

• The kernel deals with whatever caused the exception
  • Syscall
  • Interrupt
  • Page fault
  • It potentially modifies the trapframe, etc
    • E.g., Store return code in v0, zero in a3
  • ’mips_trap’ eventually returns
lw gp, 148(sp) /* restore gp */
  /* 152(sp)  stack pointer - below */
lw s8, 154(sp) /* restore s8 */
lw k0, 160(sp) /* fetch exception return PC into k0 */
lw sp, 152(sp) /* fetch saved sp (must be last) */
/* done */
jr k0 /* jump back */
rfe /* in delay slot */
.end common_exception

Note again that only k0, k1 have been trashed