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/161 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
Understanding of the existence of compiler function calling conventions
   • Including details of the MIPS ‘C’ compiler calling convention

The Structure of a Computer System

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.
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 = execve(name, args, environ)</td>
<td>Replace a process' core image.</td>
</tr>
<tr>
<td>s = 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, nbys)</td>
<td>Read data from a file into a buffer.</td>
</tr>
<tr>
<td>n = write(fd, buffer, nbys)</td>
<td>Write data from a buffer into 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 the file's status information.</td>
</tr>
</tbody>
</table>

Some System Calls For Directory Management

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s = mkdir(name, mode)</td>
<td>Create a new directory.</td>
</tr>
<tr>
<td>s = link(name, name2)</td>
<td>Create a new entry, name, pointing to name2.</td>
</tr>
<tr>
<td>s =unlink(name)</td>
<td>Remove a directory entry.</td>
</tr>
<tr>
<td>s = mount(special, name, flag)</td>
<td>Mount a file system.</td>
</tr>
<tr>
<td>s = umount(special)</td>
<td>Unmount a file system.</td>
</tr>
</tbody>
</table>

Some System Calls For Miscellaneous Tasks

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s = chdir(name)</td>
<td>Change the working directory.</td>
</tr>
<tr>
<td>s = chmod(name, mode)</td>
<td>Change a file's protection bits.</td>
</tr>
<tr>
<td>s = kill(pid, signal)</td>
<td>Send a signal to a process.</td>
</tr>
<tr>
<td>seconds = time(seconds)</td>
<td>Get the elapsed time since Jan. 1, 1970.</td>
</tr>
</tbody>
</table>

System Calls

A stripped down shell:
```c
while (TRUE) { /* repeat forever */
  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 */
    execve (command, parameters, 0); /* execute command */
  }
}
```

Some Win32 API calls

<table>
<thead>
<tr>
<th>UNIX</th>
<th>Win32</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fork</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>execve</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>waitpid</td>
<td>CreateProcess</td>
<td>Create a new process.</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>lseek</td>
<td>GetFileSizeEx</td>
<td>Get file size.</td>
</tr>
<tr>
<td>close</td>
<td>CloseHandle</td>
<td>Close the file handle.</td>
</tr>
<tr>
<td>remove</td>
<td>RemoveDirectory</td>
<td>Remove an empty directory.</td>
</tr>
<tr>
<td>rmdir</td>
<td>RemoveDirectory</td>
<td>Remove an empty directory.</td>
</tr>
<tr>
<td>rename</td>
<td>MoveFile</td>
<td>Move the file.</td>
</tr>
<tr>
<td>chdir</td>
<td>GetFileAttributes</td>
<td>Get file attributes.</td>
</tr>
<tr>
<td>chmod</td>
<td>CreateFile</td>
<td>Create a new file.</td>
</tr>
<tr>
<td>access</td>
<td>CreateDirectory</td>
<td>Create a new directory.</td>
</tr>
<tr>
<td>setstuid</td>
<td>Change the current working directory.</td>
<td></td>
</tr>
<tr>
<td>setgid</td>
<td>Change the current working directory.</td>
<td></td>
</tr>
<tr>
<td>setuid</td>
<td>Change the current working directory.</td>
<td></td>
</tr>
<tr>
<td>setpriority</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setspwan</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setpgrp</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setvbuf</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</td>
</tr>
<tr>
<td>setrlimit</td>
<td>CreateProcess</td>
<td>Create a new process.</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

CPU Registers

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

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>Interrupt Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exception Type</td>
</tr>
<tr>
<td>MMU regs</td>
</tr>
<tr>
<td>Others</td>
</tr>
<tr>
<td>PC: 0x0300</td>
</tr>
<tr>
<td>SP: 0xcbf3</td>
</tr>
<tr>
<td>Status</td>
</tr>
<tr>
<td>R1</td>
</tr>
<tr>
<td>⌫</td>
</tr>
<tr>
<td>Rn</td>
</tr>
</tbody>
</table>

Example Unsafe Instruction

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

User Mode

Kernel Mode

System call mechanism securely transfers from user execution to kernel execution and back.

Application

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?

Questions we’ll answer

- 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

System Call Mechanism Overview

Processor mode

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

SP

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

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

System Call Mechanism Overview

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

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)`.

MIPS R3000

Load/store architecture
- No instructions that operate on memory except load and store
- Simple load/stores to/from memory from/to registers
  - Store word: `sw r4, (r5)`
    - Store contents of r4 in memory using address contained in register r5
  - Load word: `lw r3, (r7)`
    - Load contents of memory into r3 using address contained in r7
    - Delay of one instruction after load before data available in destination register
    - Must always an instruction between a load from memory and the subsequent use of the register.
- `lw, sw, lb, sb, lh, sh, ...`

MIPS R3000

Arithmetic and logical operations are register to register operations
- E.g., `add r3, r2, r1`
- No arithmetic operations on memory

Example
- `add r3, r2, r1 => r3 = r2 + r1`

Some other instructions
- `add, sub, and, or, xor, sll, srl`
- `move r2, r1 => r2 = r1`

MIPS R3000

All instructions are encoded in 32-bit
Some instructions have immediate operands
- Immediate values are constants encoded in the instruction itself
- Only 16-bit value
- Examples
  - Add Immediate: `addi r2, r1, 2048`
    - r2 = r1 + 2048
  - Load Immediate: `li r2, 1234`
    - r2 = 1234

Example code
Simple code example: `a = a + 1`
```
lw   r4,32(r29) // r29 = stack pointer
li    r5, 1
add   r4, r4, r5
sw    r4,32(r29)
```
MIPS Registers

User-mode accessible registers
• 32 general purpose registers
  – r0 hardwired to zero
  – r31 the link register for jump-and-link (JAL) instruction
• HI/LO
  – 2 * 32-bits for multiply and divide
• PC
  – Not directly visible
  – Modified implicitly by jump and branch instructions

Branching and Jumping

Branching and jumping have a branch delay slot.
The instruction following a branch or jump is always executed prior to destination of jump.

31
r2, 1
sw r0, (r3)
j
r2, 2
r2, 3
l:
sw r2, (r3)

MIPS R3000

RISC architecture – 5 stage pipeline
• Instruction partially through pipeline prior to jmp having an effect

Jump and Link Instruction

JAL is used to implement function calls
- r31 = PC+8
Return Address register (RA) is used to return from function call

Compiler Register Conventions

Given 32 registers, which registers are used for
• Local variables?
• Argument passing?
• Function call results?
• Stack Pointer?

Compiler Register Conventions

<table>
<thead>
<tr>
<th>Reg No</th>
<th>Name</th>
<th>Used for</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>zero</td>
<td>Always stores 0.</td>
</tr>
<tr>
<td>1-2</td>
<td>r0-r1</td>
<td>Reserved for use by assembler.</td>
</tr>
<tr>
<td>3-10</td>
<td>r3-r10</td>
<td>Value of register E positioned by subroutine.</td>
</tr>
<tr>
<td>11-15</td>
<td>r11-r15</td>
<td>Arguments for parameters for a subroutine.</td>
</tr>
<tr>
<td>16-19</td>
<td>r16-r19</td>
<td>Temporary: subroutines may use without saving.</td>
</tr>
<tr>
<td>20-31</td>
<td>r20-r31</td>
<td>Subroutine “register variables”: a subroutine which will write one of these must save the old value and restore it before it exits, so the calling routine uses their values preserved.</td>
</tr>
<tr>
<td>32</td>
<td>sp</td>
<td>Storage for use by interrupt/stack handler – stack change must be valid stack.</td>
</tr>
<tr>
<td>33</td>
<td>gp</td>
<td>Global pointer – some machine systems maintain this to give easy access to “global” static or external variables.</td>
</tr>
<tr>
<td>34</td>
<td>tp</td>
<td>Task pointer.</td>
</tr>
<tr>
<td>35</td>
<td>t0-t1</td>
<td>This register variable. Subroutines which need one can use this as a “temp” variable.</td>
</tr>
<tr>
<td>36</td>
<td>ra</td>
<td>Return address of subroutine.</td>
</tr>
</tbody>
</table>
Simple factorial

```c
int fact(int n)
{
    int r = 1;
    int i;
    for (i = 1; i < n+1; i++)
        r = r * i;
    return r;
}
```

Function Stack Frames

Each function call allocates a new stack frame for local variables, the return address, previous frame pointer etc.
- Frame pointer: start of current stack frame
- Stack pointer: end of current stack frame

Example: assume f1() calls f2(), which calls f3().

Function Stack Frames

- Each function call allocates a new stack frame for local variables, the return address, previous frame pointer etc.
  - Frame pointer: start of current stack frame
  - Stack pointer: end of current stack frame
- Example: assume f1() calls f2(), which calls f3().

Example Code

```c
int sixargs(int a, int b, int c, int d, int e, int f)
{
    int i;
    return a + b + c + d + e + f;
}
```
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.
**C0_status**

- IM
  - Individual interrupt mask bits
  - 0 = kernel
  - 1 = user mode
- IE
  - 0 = all interrupts masked
  - 1 = interrupts enabled
- KU
  - 0 = kernel
  - 1 = user mode
- BD
  - If set, the instruction that caused the exception was in a branch delay slot

**C0_cause**

- IP
  - Interrupts pending
  - 8 bits indicating current state of interrupt lines
- CE
  - Coprocessor error
  - Attempt to access disabled Copro.
- 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>&quot;TLB modified&quot;</td>
</tr>
<tr>
<td>2</td>
<td>TLBI</td>
<td>&quot;TLB bust/TLB store&quot;</td>
</tr>
<tr>
<td>3</td>
<td>TLBS</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>MEL</td>
<td>Address error (on load/1-fetch or store respectively). Either an attempt to access outside kuser when in user mode, or an attempt to read a word or half-word at an unaligned address.</td>
</tr>
<tr>
<td>5</td>
<td>MES</td>
<td></td>
</tr>
</tbody>
</table>

**Exception Vectors**

<table>
<thead>
<tr>
<th>Program Address</th>
<th>&quot;segment&quot;</th>
<th>Physical Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3000 0000</td>
<td>0x3000</td>
<td>0x3000</td>
<td>TLI mis in kuser reference only</td>
</tr>
<tr>
<td>0x3000 0080</td>
<td>0x3000</td>
<td>0x3000</td>
<td>All other exceptions</td>
</tr>
<tr>
<td>0x3efc 0200</td>
<td>0x3efc</td>
<td>0x3efc 0100</td>
<td>Unaligned alternative kuser TLI mis entry point (used if SFI bit in IEV set)</td>
</tr>
<tr>
<td>0x3efc 0280</td>
<td>0x3efc</td>
<td>0x3efc 0180</td>
<td>Unaligned alternative for all other exceptions, used if SFI bit in IEV set</td>
</tr>
<tr>
<td>0x3efc 0000</td>
<td>0x3efc</td>
<td>0x3efc 0000</td>
<td>The &quot;reset exception&quot;</td>
</tr>
</tbody>
</table>

Aside: We ignore BD-bit in C0_cause which is also used in reality on rare occasions.
Let’s now walk through an exception.

• Assume an interrupt occurred as the previous instruction completed.
• Note: We are in user mode with interrupts enabled.

Instruction address at which to restart after the interrupt is transferred to EPC.

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

Address of general exception vector placed in PC.
Hardware exception handling

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

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:

\[
\begin{align*}
    \text{lw } r27, \text{ saved}_\text{epc} \\
    \text{nop} \\
    \text{jr } r27 \\
    \text{rfe}
\end{align*}
\]

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:

\[
\begin{align*}
    \text{lw } r27, \text{ saved}_\text{epc} \\
    \text{nop} \\
    \text{jr } r27 \\
    \text{rfe}
\end{align*}
\]

Store the EPC back in the PC.

Returning from an exception

In the branch delay slot, execute a restore from exception instruction.

Returning from an exception

We are now back in the same state we were in when the exception happened.
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 errno.
  - v0 stored in errno
  - -1 returned in v0

User-Level System Call Walk Through – Calling `read()`

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

Three arguments, one return value

Code fragment calling the read function

```
00124: 02602021 move a0,a3
00128: 27a50010 addiu a1,sp,16
0012c: 0c1001a3 jal 40068c <read>
00130: 24060400 li a2,1024
00134: 00408021 move s0,v0
00138: 1a000016 blez s0,400194 <docat+0x94>
```

Args are loaded, return value is tested

Inside the `read()` syscall function

```
part 1
```

```
0040068c <read>:
40068c: 08100190 j 400640 __syscall
400690: 24020005 li v0,5
```

Appropriate registers are preserved:
- Arguments (a0-a3), return address (ra), etc.
- The syscall number (5) is loaded into v0
- Jump (not jump and link) to the common syscall routine

```
0040068c <read>:
400690: 24020005 li v0,5
```

Seriously low-level code follows

This code is not for the faint-hearted
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

Test success, if yes,
branch to return
from function

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

If failure, store code
in errno

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

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

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 */
    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 */
```

Note k0, k1 registers available for kernel use

```
common_exception:
    /*
    * At this point:
    * Intercepts are off. (The processor did this for us.)
    * k0 contains the exception cause value.
    * k1 contains the old stack pointer.
    * sp points into the kernel stack.
    * All other registers are untouched.
    */
    /*
    * Allocate stack space for 37 words to hold the trap frame,
    * plus four more words for a minimal argument block.
    */
    addi sp, sp, -164
```

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

```
    sw sa, 160(sp) /* dummy for gdb */
    sw s8, 156(sp) /* save s8 */
    sw sp, 152(sp) /* dummy for gdb */
    sw fp, 148(sp) /* save fp */
    sw k1, 144(sp) /* dummy for gdb */
    sw k0, 140(sp) /* dummy for gdb */
    sw k1, 152(sp) /* real saved sp */
    addi sp, sp, -164 /* dummy for gdb */
    sw fp, 146(sp) /* save fp */
```
The real work starts here

Save all the registers on the kernel stack

We can now use the other registers (t0, t1) that we have preserved on the stack

Kernel Stack

Now we arrive in the ‘C’ kernel

By creating a pointer to here of type struct trapframe *, we can access the user’s saved registers as normal variables within ‘C’

/* The order here must match mips/include/trapframe.h. */
sw ra, 160(sp) /* dummy for gdb */
sw s8, 156(sp) /* save s8 */
sw sp, 152(sp) /* dummy for gdb */
sw gp, 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 k1, c0_epc /* Copr.0 reg 13 == PC for exception */
sw k1, 160(sp) /* real saved PC */

Save all the registers on the kernel stack

We can now use the other registers (t0, t1) that we have preserved on the stack

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;
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

```
exception_return:
 /*     16(sp) no need to restore tf_vaddr */
 lw t0, 20(sp) /* load status register value into t0 */
 nop /* load delay slot */
 mtsc 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)
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 sp, 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 sp, 132(sp)
 /*     140(sp) "saved" k0 was dummy garbage anyway */
 /*     144(sp) "saved" k1 was dummy garbage anyway */

lw gp, 148(sp) /* restore gp */
 /*     152(sp) stack pointer - below */
 lw sp, 156(sp) /* restore sp */
 lw k0, 160(sp) /* fetch exception return PC into k0 */
 /* done */
 jr k0 /* jump back */
 rfe /* in delay slot */
.end common_exception
```

Note again that only k0, k1 have been trashed