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/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

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.

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 syscalls` for a list
Some System Calls For Process Management

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getpid()</td>
<td>Create a child process identical to the parent</td>
</tr>
<tr>
<td>waitpid(pid, &amp;status, options)</td>
<td>Wait for a child to terminate</td>
</tr>
<tr>
<td>execl(name, arg, 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>fopen(file, how, ...)</td>
<td>Open a file for reading, writing or both</td>
</tr>
<tr>
<td>close(fd)</td>
<td>Close an open file</td>
</tr>
<tr>
<td>read(fd, buffer, nbyte)</td>
<td>Read data from a file into a buffer</td>
</tr>
<tr>
<td>write(fd, buffer, nbyte)</td>
<td>Write data from a buffer into a file</td>
</tr>
<tr>
<td>lseek(fd, offset, whence)</td>
<td>Move the file pointer</td>
</tr>
<tr>
<td>stat(name, &amp;buf)</td>
<td>Get a 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>mkdir(name, mode)</td>
<td>Create a new directory</td>
</tr>
<tr>
<td>rmdir(name)</td>
<td>Remove an empty directory</td>
</tr>
<tr>
<td>umount(special, name, flag)</td>
<td>Mount a file system</td>
</tr>
<tr>
<td>unmount(name, flag)</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>chdir(dimname)</td>
<td>Change the working directory</td>
</tr>
<tr>
<td>chmod(name, mode)</td>
<td>Change a file's protection bits</td>
</tr>
<tr>
<td>kill(signal, seconds)</td>
<td>Send a signal to a process</td>
</tr>
<tr>
<td>elapsed_time(time)</td>
<td>Get the elapsed time since Jan. 1, 1970</td>
</tr>
</tbody>
</table>

System Calls

*A stripped down shell:*

```
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

**UNIX/Win32**

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateProcess</td>
<td>Create a new process</td>
</tr>
<tr>
<td>ReadFile</td>
<td>Read data from a file</td>
</tr>
<tr>
<td>CloseHandle</td>
<td>Close a handle</td>
</tr>
<tr>
<td>GetFileAttributes</td>
<td>Get file attributes</td>
</tr>
<tr>
<td>GetCurrentDirectory</td>
<td>Change the current working directory</td>
</tr>
<tr>
<td>GetCurrentDirectory</td>
<td>Get current directory</td>
</tr>
</tbody>
</table>

**Win32**

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detach</td>
<td>Detach the thread from the process</td>
</tr>
<tr>
<td>TerminateThread</td>
<td>Terminate a thread</td>
</tr>
<tr>
<td>TerminateProcess</td>
<td>Terminate a process</td>
</tr>
<tr>
<td>ResumeThread</td>
<td>Resume a thread</td>
</tr>
<tr>
<td>SuspendThread</td>
<td>Suspend a thread</td>
</tr>
<tr>
<td>ResumeThread</td>
<td>Resume a thread</td>
</tr>
<tr>
<td>TerminateProcess</td>
<td>Terminate a process</td>
</tr>
<tr>
<td>TerminateProcess</td>
<td>Terminate a process</td>
</tr>
<tr>
<td>CreateSemaphore</td>
<td>Create a semaphore</td>
</tr>
<tr>
<td>GetSemaphore</td>
<td>Get semaphore value</td>
</tr>
<tr>
<td>ReleaseSemaphore</td>
<td>Release a semaphore</td>
</tr>
<tr>
<td>SignalThread</td>
<td>Signal a thread</td>
</tr>
<tr>
<td>Sleep</td>
<td>Sleep for a specified number of milliseconds</td>
</tr>
<tr>
<td>ResumeThread</td>
<td>Resume a thread</td>
</tr>
<tr>
<td>ResumeThread</td>
<td>Resume a thread</td>
</tr>
<tr>
<td>TerminateProcess</td>
<td>Terminate a process</td>
</tr>
<tr>
<td>TerminateProcess</td>
<td>Terminate a process</td>
</tr>
<tr>
<td>CreateEvent</td>
<td>Create an event</td>
</tr>
<tr>
<td>ReleaseSemaphore</td>
<td>Release a semaphore</td>
</tr>
<tr>
<td>ReleaseSemaphore</td>
<td>Release a semaphore</td>
</tr>
<tr>
<td>SignalThread</td>
<td>Signal a thread</td>
</tr>
<tr>
<td>Sleep</td>
<td>Sleep for a specified number of milliseconds</td>
</tr>
<tr>
<td>ResumeThread</td>
<td>Resume a thread</td>
</tr>
<tr>
<td>ResumeThread</td>
<td>Resume a thread</td>
</tr>
<tr>
<td>TerminateProcess</td>
<td>Terminate a process</td>
</tr>
<tr>
<td>TerminateProcess</td>
<td>Terminate a process</td>
</tr>
</tbody>
</table>

```
System Call Implementation

Crossing user-kernel boundary

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

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

Example Unsafe Instruction

• “cli” instruction on x86 architecture
  • Disables interrupts

• Example exploit
  ```c
  int / 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>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFFF000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Accessible to User- and Kernel-mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Accessible only to Kernel-mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x800000000</td>
</tr>
</tbody>
</table>

CPU Registers

PC: 0x0300
SP: 0xcbf3
R1
Status
Rn

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.

CPU Registers

PC: 0x0300
SP: 0xcbf3
R1
Status
Rn

CPU Registers

PC: 0x0300
SP: 0xcbf3
R1
Status
Rn

CPU Registers

PC: 0x0300
SP: 0xcbf3
R1
Status
Rn

CPU Registers

PC: 0x0300
SP: 0xcbf3
R1
Status
Rn
System Call

User Mode

Kernel Mode

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

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?

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

System Call Mechanism Overview

• 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

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

System Call

System Call Mechanism Overview

• 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
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.

Coproprocessors

• 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_eip`
    - Address of the instruction that caused the exception
  • `c0_trace`
    - 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

For practical purposes, you can ignore most bits
• Green background is the focus

• IM
  • Individual interrupt mask bits
  • 6 external
  • 2 software
• KU
  • 0 = kernel
  • 1 = user mode
• IE
  • 0 = all interrupts masked
  • 1 = interrupts enable
  • Mask determined via IM bits
• c, p, o = current, previous, old
### Exception Codes

<table>
<thead>
<tr>
<th>Exception Code 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 modification</td>
</tr>
<tr>
<td>2</td>
<td>TLB</td>
<td>TLB fault/TLB store</td>
</tr>
<tr>
<td>3</td>
<td>TLBES</td>
<td>Address error (on load/1-fetch or store respectively). Either an attempt to access outside kseg when in user mode, or an attempt to read a word or half-word at a misaligned address.</td>
</tr>
</tbody>
</table>

### 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 misses on kseg reference only</td>
</tr>
<tr>
<td>0x0000 0000</td>
<td>kseg0</td>
<td>0x0000 0000</td>
<td>All other exceptions.</td>
</tr>
<tr>
<td>0x010c 0100</td>
<td>kseg1</td>
<td>0x100c 0100</td>
<td>Uncaused alternative kseg TLB misses entry point (used if SR bit: BEV set)</td>
</tr>
<tr>
<td>0x010c 0100</td>
<td>kseg1</td>
<td>0x100c 0100</td>
<td>Uncaused alternative for all other exceptions, used if SR bit: BEV set.</td>
</tr>
<tr>
<td>0x010c 0000</td>
<td>kseg1</td>
<td>0x100c 0000</td>
<td>The “reset exception”.</td>
</tr>
</tbody>
</table>

**Table 4.1. Reset and exception entry points (vectors) for R3000 family**

### c0_cause

- **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

**Figure 3.3. Fields in the Cause register**

### 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

**Notes:** We are ignoring the BD-bit in c0_cause which is also used in reality on rare occasions.

### Simple Exception Walk-through

**User Mode**

Application

**Kernel Mode**

Interrupt Handler

**DIAGRAM:**

- Process flow diagram showing the transition from User Mode to Kernel Mode.
Hardware exception handling

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.

Interrupts disabled and previous state shifted along.

Kernel Mode is set, and previous mode shifted along.

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

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:

```assembly
lw r27, saved_epc
nop
jr r27
rfe
```

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`

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

- Three arguments, one return value
- Code fragment calling the read function

Inside the `read()` syscall function part 1

```
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

```
move a0,a3
addiu a1,sp,16
jal 40068c <read>
li a2,1024
move s0,v0
blez s0,400194 <docat+0x94>
```

- 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
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

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.

```
exception:
    move k1, sp       /* Save previous stack pointer in k1 */
    mfio 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:
    mfio 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:
    * Interrupts 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
```

```
/* 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 */
w x k1, 152(sp) /* real saved sp */
sp /* delay slot for store */
```

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

```
/* 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 */
w x k1, 152(sp) /* real saved sp */
sp /* delay slot for store */
```

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

Create a pointer to the base of the saved registers and state in the first argument register

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

Now we arrive in the 'C' kernel

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 */
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)
addi t1
tbhi 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)
/* 140(sp) "saved" k0 was dummy garbage anyway */
/* 144(sp) "saved" k1 was dummy garbage anyway */
/* 152(sp) stack pointer - below */
lw s8, 156(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