System Calls

Contents

- A high-level view of System Calls
  - Mostly from the user’s perspective
  - From textbook (section 1.6)
- A look at the R3000
  - A brief overview
  - Mostly focused on exception handling
    - From “Hardware Guide” on class web site
  - Allow me to provide “real” examples of theory
- System Call implementation
  - Case Study: OS/161 system call handling

Operating System

System Calls

- Can be viewed as special procedure 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.

A Brief Overview of Classes

System Calls

- 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>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, argv, envr[0];)</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>f = open(fd, name, ...);</td>
<td>Open a file for reading, writing or both</td>
</tr>
<tr>
<td>l = close(fd);</td>
<td>Close an open file</td>
</tr>
<tr>
<td>r = read(fd, buffer, bytes);</td>
<td>Read data from a file into a buffer</td>
</tr>
<tr>
<td>w = write(fd, buffer, bytes);</td>
<td>Write data from a buffer into a file</td>
</tr>
<tr>
<td>position = seek(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>

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 = unlink(name);</td>
<td>Remove an empty directory</td>
</tr>
<tr>
<td>s = rename(name1, name2);</td>
<td>Create a new entry, rename, pointing to name1</td>
</tr>
<tr>
<td>s = write(name);</td>
<td>Remove a directory entry</td>
</tr>
<tr>
<td>s = mount(spec, name, flags);</td>
<td>Mount a file system</td>
</tr>
<tr>
<td>s = umount(roadspec);</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>a = 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, 1976</td>
</tr>
</tbody>
</table>

System Calls

- A stripped down shell:

  ```c
  while (TRUE) {
    /* repeat forever */
    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 */
    }
  }
  ```

The MIPS R2000/R3000

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

Some Win32 API calls
MIPS R3000

- RISC architecture – 5 stage pipeline

![5-stage pipeline diagram]

- 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)`
    - Load 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`, .....

- 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` $\Rightarrow r3 = r2 + r1$

- Some other instructions
  - `add, sub, and, or, xor, sll, srl`

- 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`
      - Load Immediate: `li r2, 1234`

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

- Example
  - `sw $0, ($3)`
  - `j 1f`
  - `li $2, 1`
  - `1: sw $2, ($3)`
Jump and Link

- JAL is used to implement function calls
  - \( r31 = PC + 8 \)
- Jump Register (JR) is used to return from function call

```
jal 1f
lw $4, ($6)

1:
sw $2, ($3)
jr $31
nop
```

R3000 Address Space Layout

- **kseg:**
  - 2 gigabytes
  - TLB translated (mapped)
  - Cacheable
  - user-mode and kernel mode accessible
  - Page size is 4K

- **kuseg:**

R3000 Address Space Layout

- **kseg0:**
  - 512 megabytes
  - Fixed translation window to physical memory
    - \( 0x80000000 - 0x9fffffff \) virtual
      - \( 0x00000000 - 0x1fffffff \) physical
  - Cacheable
  - Only kernel-mode accessible
  - Usually where the kernel code is placed

- **kseg1:**
  - 512 megabytes
  - Fixed translation window to physical memory
    - \( 0xa0000000 - 0xbfffffff \) virtual
      - \( 0x00000000 - 0x1fffffff \) physical
  - NOT cacheable
  - Only kernel-mode accessible
  - Where devices are accessed (and boot ROM)

- **kseg2:**
  - 1024 megabytes
  - TLB translated (mapped)
  - Cacheable
  - Only kernel-mode accessible

System/161 Aside

- System/161 simulates an R3000 without a cache.
  - You don’t need to worry about cache issues with programming OS161 running on System/161
Coprocesor 0

- The processor control registers are located in CP0
  - Exception 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
    - Note the BD bit in c0_cause
  - c0_badvaddr
    - 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

CP0 Status

- CU0-3
  - Enable access to coprocessors (1 = enable)
    - CU0 never enabled for user mode
    - CU1 is floating point unit (if present, FPU not in sys161)
    - CU2-3 reserved
- RE
  - Reverse endian
- BEV
  - Boot exception vectors
    - 1 = use ROM exception vectors
    - 0 = use RAM exception vectors
- TS
  - TLB shutdown (1 = duplicate entry, need a hardware reset)
- PE
  - Parity error in cache
- CM
  - Cache management
- PZ
  - Cache parity zero
- SwC
  - Access instruction cache as data
- IsC
  - Isolate data cache
- For practical purposes, you can ignore these bits
- IM
  - Individual interrupt mask
    - Bits
- IE
  - 0 = all interrupts masked
  - 1 = interrupts enable
  - Mask determined via IM bits
    - c, p, o = current, previous, old
### c0_cause

- **BD**
  - If set, the instruction that caused the exception was in a branch delay slot
- **IP**
  - Interrupts pending
    - 8 bits indicating current state of interrupt lines
- **CE**
  - Coprocessor error
    - Attempt to access disabled Copro.
  - The code number of the exception taken

### Exception Codes

<table>
<thead>
<tr>
<th>ExeCode 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>Stbl</td>
<td>&quot;TLB modification&quot;</td>
</tr>
<tr>
<td>2</td>
<td>TLBL</td>
<td>TLB load/TLB store</td>
</tr>
<tr>
<td>3</td>
<td>TLBSN</td>
<td>TLB steal/TLB store</td>
</tr>
<tr>
<td>4</td>
<td>AdEL</td>
<td>Address error (on load/32-bit or store respectively)</td>
</tr>
<tr>
<td>5</td>
<td>AdES</td>
<td>Either an attempt to access outside kernel when in user mode, or an attempt to read a word or half-word at an unaligned address</td>
</tr>
</tbody>
</table>

### c0_epc

- The Exception Program Counter
  - The address of where to restart execution after handling the exception or interrupt
  - BD-bit in c0_cause is used on rare occasions when one needs to identify the actual exception-causing instruction
  - Example
    - Assume `sw r3,(r4)` causes a page fault exception

### c0_badvaddr

- The address access that caused the exception
  - Set if exception is
    - MMU related
    - Access to kernel space from user-mode
    - Unaligned memory access
      - 4-byte words must be aligned on a 4-byte boundary

### 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>0x4000 0000</td>
<td>kseg0</td>
<td>0x4000 0000</td>
<td>TLB misses on cache reference only</td>
</tr>
<tr>
<td>0x4000 0000</td>
<td>kseg0</td>
<td>0x4000 0000</td>
<td>All other exceptions</td>
</tr>
<tr>
<td>0x0000 0000</td>
<td>kseg1</td>
<td>0x0000 0100</td>
<td>Unreached alternative kseg TLB entry point (used if SE bit in SEV set)</td>
</tr>
<tr>
<td>0x0000 0000</td>
<td>kseg1</td>
<td>0x0000 0100</td>
<td>Unreached alternative for all other exceptions, used if SE bit in SEV set</td>
</tr>
<tr>
<td>0x0000 0000</td>
<td>kseg1</td>
<td>0x0000 0000</td>
<td>The 'reset exception'</td>
</tr>
</tbody>
</table>
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.

![Diagram of PC and EPC values with instruction address and status]

The labs are now:
- Thursday 4-5pm
- Friday 1-3pm
- In the “spoons” lab
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

Returning from an exception

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

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
Function Stack Frames

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

Software 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 returns 0</td>
</tr>
<tr>
<td>1</td>
<td>at</td>
<td>Pointer temporarily reserved for use by assembler</td>
</tr>
<tr>
<td>2-3</td>
<td>sp/v1</td>
<td>Value typically reserved for subroutine</td>
</tr>
<tr>
<td>4-15</td>
<td>a0-v0</td>
<td>Arguments first parameter for a subroutine</td>
</tr>
<tr>
<td>16-25</td>
<td>a1-v7</td>
<td>Temporary; subroutine may use without saving</td>
</tr>
<tr>
<td>26-27</td>
<td>v8-v9</td>
<td>Reserved for use by interrupt trap handler; may change under your feet</td>
</tr>
<tr>
<td>28</td>
<td>gp</td>
<td>Global pointer; some runtime systems maintain this to give access to pointed-at static or external variables</td>
</tr>
<tr>
<td>29</td>
<td>sp</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>30</td>
<td>sb/s8</td>
<td>32-bit register variable; Subroutines which need one can use this as a “frame pointer”</td>
</tr>
<tr>
<td>31</td>
<td>ra</td>
<td>Return address for subroutine</td>
</tr>
</tbody>
</table>

Stack Frame

- MIPS calling convention for gcc
  - Args 1-4 have space reserved for them

• Each function call allocates a new stack frame for local variables, the return address, previous frame pointer etc.
• Example: assume f1() calls f2(), which calls f3().

Software Register Conventions

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

Continued

User and Kernel Execution

- Simplistically, execution state consists of
  - Registers, processor mode, PC, SP
- User applications and the kernel have their
  own execution state.
- System call mechanism safely transfers
  from user execution to kernel execution and back.
**System Call Mechanism in Principle**

- 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

**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

**Steps in Making a System Call**

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

**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

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

• Three arguments, one return value
• Code fragment calling the read function
  
  400124: 02602021 move a0,s3
  400128: 27a50010 addiu a1,sp,16
  40012c: e1 a1,1024
  400130: 0040068c <read>
  400134: 0040068c move s0,v0
  400138: 1a000016 blez s0,400194 <docat+0x94>

• Args are loaded, return value is tested

The read() syscall function

part 1

0040068c <read>:
  4006b: 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

part 2

00400640 <__syscall>:
  400640: 0000000c syscall
  400644: 10e00005 beqz a3,40065c <__syscall+0x1c>
  400648: 00000000 nop
  40064c: 3c001000 lui at,0x1000
  400650: ac220000 sw v0,0(at)
  400654: 240ffffff li v1,-1
  400658: 240ffffff li v0,-1
  40065c: 03e00008 jr ra
  400660: 00000000 nop

Test success, if yes, branch to return from function

If failure, store code in errno
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 support 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 (the 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

```
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) /* load */  
1:  
    mfc0 k0, c0_cause /* Now, load the exception cause. */  
    jr ra /* Skip to common code */  
    nop /* delay slot */
```
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

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

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’.

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

Note again that only k0, k1 have been trashed