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

Kernel Level

Applications

Operating System

Requests (System Calls)

User Level

Applications

Applications

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

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.

Some System Calls For Process Management
Some System Calls For File Management

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f = open(a file, mode, ...)</td>
<td>Open a file for reading, writing or both</td>
</tr>
<tr>
<td>f = create()</td>
<td>Create a new file</td>
</tr>
<tr>
<td>r = read(a buffer, position)</td>
<td>Read data from the file into a buffer</td>
</tr>
<tr>
<td>w = write(a buffer)</td>
<td>Write data from a buffer into a file</td>
</tr>
<tr>
<td>p = seek(a file position, whence)</td>
<td>Move the file position</td>
</tr>
<tr>
<td>a = stat(file, &amp;status)</td>
<td>Get a file 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>d = read(a directory)</td>
<td>Read a file descriptor</td>
</tr>
<tr>
<td>d = mkdir(a directory)</td>
<td>Create a new directory</td>
</tr>
<tr>
<td>d = rmdir(a directory)</td>
<td>Remove a directory</td>
</tr>
<tr>
<td>d = stat(a directory)</td>
<td>Get a directory status</td>
</tr>
<tr>
<td>d = list(a directory)</td>
<td>List the files in a directory</td>
</tr>
<tr>
<td>d = link(a file)</td>
<td>Create a new file link</td>
</tr>
<tr>
<td>d = unlink(a file)</td>
<td>Remove a file link</td>
</tr>
</tbody>
</table>

Some System Calls For Miscellaneous Tasks

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c = connect()</td>
<td>Connect to a server</td>
</tr>
<tr>
<td>c = disconnect()</td>
<td>Disconnect from a server</td>
</tr>
<tr>
<td>c = listen()</td>
<td>Listen for a connection</td>
</tr>
<tr>
<td>c = accept()</td>
<td>Accept a connection</td>
</tr>
<tr>
<td>c = send()</td>
<td>Send data to a client</td>
</tr>
<tr>
<td>c = recv()</td>
<td>Receive data from a client</td>
</tr>
</tbody>
</table>

System Calls

- A stripped down shell:

```c
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 */
        execve(command, parameters, 0); /* execute command */
    }
}
```

System Calls

Some Win32 API calls

The MIPS R2000/R3000

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

- RISC architecture – 5 stage pipeline

![MIPS R3000 Pipeline Diagram](image)

- 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, lw, lb, sb, sh, 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` ⇒ `r3 = r2 + r1`

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

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
  - `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

R3000 Address Space Layout

- kseg0:
  - 512 megabytes
  - Fixed translation window to physical memory
    - 0x80000000 - 0xffffffff virtual = 0x00000000 - 0x1fffffff physical
    - MMU not used
  - Cacheable
  - Only kernel-mode accessible
  - Where the kernel code is placed

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

- kseg2:
  - 1024 megabytes
  - MMU 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
Coprocessor 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

C0_registers

- For practical purposes, you can ignore these bits
  - Green background is the focus
- CU0-3
  - Enable access to coprocessors (1 = enable)
    - CU0 never enabled for user mode
    - Always accessible in kernel mode regardless of setting
    - CU1 is floating point unit (if present, FPU not in sys161)
    - CU2-3 reserved

C0_status

- 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)

C0_status

- PE
  - Parity error in cache
- CM
  - Cache management
- PZ
  - Cache parity zero
- SwC
  - Access instruction cache as data
- IsC
  - Isolate data cache

C0_status

- 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
c0_cause

<table>
<thead>
<tr>
<th>ED</th>
<th>CE</th>
<th>IP</th>
<th>IP</th>
<th>ExcCode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>15</td>
<td>15</td>
</tr>
</tbody>
</table>

Figure 3.3. Fields in the Cause register

- **IP**: Intermits 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</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Int</td>
<td>Interrupt</td>
</tr>
</tbody>
</table>
| 1       | SI:EB    | TLB modified
| 2       | TL:LE    | TLB load/TLB store
| 4       | Ad:EL    | Address error (on load/4-bit or store)
| 5       | MEM:ES   | Attempt to read a word or half word at an unaligned address

Table 3.2. ExcCode values: different kinds of exceptions

### 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

```
sw r3, (r4)
nop
sw r3, (r4)
```

### 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 Addresses</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8000 0000</td>
<td>kseg0</td>
<td>0x8000 0000</td>
<td>TLB misses on ECX reference only</td>
</tr>
<tr>
<td>0x8000 0000</td>
<td>kseg0</td>
<td>0x8000 0000</td>
<td>All other exceptions</td>
</tr>
<tr>
<td>0x0000 0000</td>
<td>kseg0</td>
<td>0x0000 0000</td>
<td>Unachieved alternative to TLB miss entry point used if SR bit is set</td>
</tr>
<tr>
<td>0x0000 0000</td>
<td>kseg0</td>
<td>0x0000 0000</td>
<td>Unachieved alternative for all other exceptions, used if SC bit is set</td>
</tr>
<tr>
<td>0x0000 0000</td>
<td>kseg0</td>
<td>0x0000 0000</td>
<td>The &quot;reset exception&quot;</td>
</tr>
</tbody>
</table>

Table 4.1. Reset and exception entry points (vectors) for x86 family
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

<table>
<thead>
<tr>
<th>PC</th>
<th>EPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x12345678</td>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cause</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>KU IE K Up IE K Uc IE c</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Badvaddr</th>
<th></th>
</tr>
</thead>
</table>

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

<table>
<thead>
<tr>
<th>PC</th>
<th>EPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x12345678</td>
<td>0x12345678</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cause</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>KU IE K Up IE K Uc IE c</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Badvaddr</th>
<th></th>
</tr>
</thead>
</table>

- 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

<table>
<thead>
<tr>
<th>PC</th>
<th>EPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x80000080</td>
<td>0x12345678</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cause</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x12345678</td>
<td>KU IE K Up IE K Uc IE c</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Badvaddr</th>
<th></th>
</tr>
</thead>
</table>
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

PC 0x80001234
EPC 0x12345678

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

Load the contents of EPC which is usually saved somewhere when the exception was originally taken

Returning from an exception

PC 0x12345678
EPC 0x12345678

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

Store the EPC back in the PC

Returning from an exception

PC 0x12345678
EPC 0x12345678

• We are now back in the same state we were in when the exception happened

Returning from an exception

PC 0x12345678
EPC 0x12345678

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

Load the contents of EPC which is usually saved somewhere when the exception was originally taken

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

Returning from an exception

PC 0x12345678
EPC 0x12345678

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

Badvaddr

Store the EPC back in the PC

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

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().
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

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

Stack Frame

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

Example Code

```c
main ()
{
    int sixargs(int a, int b, int c, int d, int e, int f);
    int i;
    i = sixargs(1, 2, 3, 4, 5, 6);
    return a + b + c + d + e + f;
}
```
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
System Call Mechanism in Principle

- 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

CAUTION

- Seriously low-level code follows
- This code is not for the faint hearted
User-Level System Call Walk Through

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

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

```assembly
400124: 02602021 move a0,a3
400128: 27a50010 addiu a1,sp,16
40012c: 0c1001a3 jal 40068c <read>
400130: 24060201 move s0,v0
400134: 00400016 blez s0,400194 <docat+0x94>
```

- Args are loaded, return value is tested

The read() syscall function part 1

```assembly
0040068c <read>:
40068c: 08100190 j 400640 <__syscall>
```

- 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

The read() syscall function part 2

```assembly
00400640 <__syscall>:
400640: 0000000c syscall
```

- Test success, if yes, branch to return from function

The read() syscall function part 2

```assembly
00400640 <__syscall>:
400644: 0e000005 beqz a3,40065c <__syscall+0x1c>
```

- If failure, store code in errno

The read() syscall function part 2

```assembly
00400640 <__syscall>:
400654: 2403ffff li v1,-1
```

- Set read() result to -1

```assembly
400658: 2402ffff li v0,-1
40065c: 03e00008 jr ra
400660: 00000000 nop
400664: 00000000 nop
400668: 00000000 nop
40066c: 3c011000 lui at,0x1000
400670: ac220000 sw v0,0(at)
400674: 400658: 2402ffff li v0,-1
40065c: 03e00008 jr ra
400660: 00000000 nop
```

- If failure, store code in errno
The read() syscall function part 2

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

Note k0, k1 registers available for kernel use

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 */
lw k0, curkstack /* get address of "curkstack" */
w sp, 0(k0) /* load */
1:
mfc0 k0, c0_cause /* Now, load the exception cause. */
j common_exception /* Skip to common code */
 nop /* delay slot for the load */
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.

Save all the registers on the kernel stack.

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

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

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

Struct trapframe {
  u_int32_t tf_vaddr; /* vaddr register */
  u_int32_t tf_status; /* status register */
  u_int32_t tf_cause; /* cause register */
  u_int32_t tf_lo;
  u_int32_t tf_hi;
  u_int32_t tf_ra;/* Saved register 31 */
  u_int32_t tf_at;/* Saved register 1 (AT) */
  u_int32_t tf_v0;/* Saved register 2 (v0) */
  u_int32_t tf_v1;/* etc. */
  u_int32_t tf_a0;
  u_int32_t tf_a1;
  u_int32_t tf_a2;
  u_int32_t tf_a3;
  u_int32_t tf_t0; /* dummy */
  u_int32_t tf_t1; /* dummy */
  u_int32_t tf_t2; /* dummy */
  u_int32_t tf_t3; /* dummy */
  u_int32_t tf_t4; /* dummy */
  u_int32_t tf_t5; /* dummy */
  u_int32_t tf_t6; /* dummy */
  u_int32_t tf_t7;  
  u_int32_t tf_s0; /* dummy (see exception.S comments) */
  u_int32_t tf_s1; /* dummy */
  u_int32_t tf_s2; /* dummy */
  u_int32_t tf_s3; /* dummy */
  u_int32_t tf_s4; /* dummy */
  u_int32_t tf_s5; /* dummy */
  u_int32_t tf_s6; /* dummy */
  u_int32_t tf_s7; /* dummy */
  u_int32_t tf_epc; /* coprocessor 0 epc register */
};
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;

    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, 24(sp)
    lw t0, 32(sp)
    mtlc 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 gp, 68(sp)
    lw t0, 72(sp)
    lw t1, 76(sp)
    lw t2, 80(sp)
    lw t3, 84(sp)
    lw t4, 88(sp)
    lw t5, 92(sp)
    lw t6, 96(sp)
    lw t7, 100(sp)
    lw t8, 104(sp)
    lw t9, 108(sp)
    lw s0, 112(sp)
    lw s1, 116(sp)
    lw s2, 120(sp)
    lw s3, 124(sp)
    lw s4, 128(sp)
    lw s5, 132(sp)
    lw s6, 136(sp)
    lw s7, 140(sp)
    lw sp, 144(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 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

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