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

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

- 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><code>pid = fork();</code></td>
<td>Create a child process identical to the parent</td>
</tr>
<tr>
<td><code>pid = wall(pid, filhandle, options)</code></td>
<td>Wait for a child to terminate</td>
</tr>
<tr>
<td><code>s = execve(name, argv, envir)</code></td>
<td>Replace a process' core image</td>
</tr>
<tr>
<td><code>exit(exit_status)</code></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>open</td>
<td>Open a file for reading, writing or both</td>
</tr>
<tr>
<td>close</td>
<td>Close an open file</td>
</tr>
<tr>
<td>read</td>
<td>Read data from a file into a buffer</td>
</tr>
<tr>
<td>write</td>
<td>Write data from a buffer into a file</td>
</tr>
<tr>
<td>stat</td>
<td>Get 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</td>
<td>Create a new directory</td>
</tr>
<tr>
<td>rmdir</td>
<td>Remove a directory</td>
</tr>
<tr>
<td>link</td>
<td>Create a new entry, rename, pointing to name</td>
</tr>
<tr>
<td>unlink</td>
<td>Remove a directory entry</td>
</tr>
<tr>
<td>mount, umount</td>
<td>Mount a file system, 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</td>
<td>Change the working directory</td>
</tr>
<tr>
<td>chmod</td>
<td>Change file's protection bits</td>
</tr>
<tr>
<td>kill</td>
<td>Send a signal to a process</td>
</tr>
<tr>
<td>time</td>
<td>Get the elapsed time since Jan. 1, 1970</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 */
        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

<table>
<thead>
<tr>
<th>Name</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>WriteFile</td>
<td>Write data to a file</td>
</tr>
<tr>
<td>GetCurrentDirectory</td>
<td>Get the current directory</td>
</tr>
<tr>
<td>GetCurrentTime</td>
<td>Get the current time</td>
</tr>
</tbody>
</table>
MIPS R3000

• RISC architecture – 5 stage pipeline

![MIPS 5-stage pipeline](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, 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` → `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`
        ⇒ `r2 = r1 + 2048`
      • Load Immediate: `li r2, 1234`
        ⇒ `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
  
  ```
  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 if
lw $4, ($6)
sw $2, ($3)
jr $31
```

R3000 Address Space Layout
- kseg:
  - 2 gigabytes
  - TLB translated (mapped)
  - Cacheable
  - user-mode and kernel mode accessible
  - Page size is 4K

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

- Physical Memory
- 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
C0p0cess0r 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_epp
    - 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 the course

### c0_status

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0</td>
<td>Cause of the recent exception</td>
</tr>
<tr>
<td>C1</td>
<td>Current status of the CPU</td>
</tr>
<tr>
<td>C2</td>
<td>Address of the instruction that caused the exception</td>
</tr>
<tr>
<td>C3</td>
<td>Address accessed that caused the exception</td>
</tr>
<tr>
<td>C0_cause</td>
<td>Cause of the recent exception</td>
</tr>
<tr>
<td>C0_epp</td>
<td>Address of the instruction that caused the exception</td>
</tr>
</tbody>
</table>

**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**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RE</td>
<td>Reverse endian</td>
</tr>
</tbody>
</table>
| 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_prid**

- Processor Identifier

**Memory Management**

- c0_index
- c0_random
- c0_entryhi
- c0_entrylo
- c0_context

More about these later in the course.
### c0_cause

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BD</td>
<td>- If set, the instruction that caused the exception was in a branch delay slot</td>
</tr>
<tr>
<td>IP</td>
<td>- Interrupts pending, 8 bits indicating current state of interrupt lines</td>
</tr>
<tr>
<td>CE</td>
<td>- Coprocessor error, attempt to access disabled Copro.</td>
</tr>
</tbody>
</table>

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

### 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>Msh</td>
<td>TLB modification</td>
</tr>
<tr>
<td>2</td>
<td>TLB_L</td>
<td>TLB load/TLB store</td>
</tr>
<tr>
<td>3</td>
<td>TLB_D</td>
<td>Address error (on load/TLB or store respectively)</td>
</tr>
<tr>
<td>4</td>
<td>AdEL</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>

**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 aw x3, (x4) 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>0x0000 0000</td>
<td>lsigO</td>
<td>0x0000 0000</td>
<td>TLB miss on locking reference only.</td>
</tr>
<tr>
<td>0x0000 0000</td>
<td>lsigO</td>
<td>0x0000 0000</td>
<td>All other exceptions.</td>
</tr>
<tr>
<td>0x0000 0100</td>
<td>lsigO</td>
<td>0x0000 0100</td>
<td>Unaligned alternative for TEB entry point (used if SR bit BEV set).</td>
</tr>
<tr>
<td>0x0000 0100</td>
<td>lsigO</td>
<td>0x0000 0100</td>
<td>Unaligned alternative for all other exceptions, used if SR bit BEV set.</td>
</tr>
<tr>
<td>0x0000 0000</td>
<td>lsigO</td>
<td>0x0000 0000</td>
<td>The 'reset exception'.</td>
</tr>
</tbody>
</table>

**Table 4.1._Reset and exception entry points (vectors) for R3900x 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>
<th>Badvaddr</th>
</tr>
</thead>
</table>

<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>
<th>Badvaddr</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>0x80000080</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cause</th>
<th>Status</th>
<th>Badvaddr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>? ? ? ? 1 1</td>
<td>?</td>
</tr>
</tbody>
</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>
<th>Badvaddr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>? ? ? ? 1 1</td>
<td>?</td>
</tr>
</tbody>
</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

This code to return is:

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

Store the EPC back in the PC:

```
0x12345678
```

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

```
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
    - 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

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: 0c1001a3 jal 40068c <read>
  400130: 24060400 li a2,1024
  400134: 00408021 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>:
  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

The read() syscall function part 2

00400640 <__syscall>:
  0040640: 0000000c syscall
  0040644: 10a00005 beqz a3,40065c <__syscall+0x1c>
  0040648: 00000000 nop
  004064c: 3c011000 lui at,0x1000
  0040650: ac220000 sw v0,0(at)
  0040654: 2403ffff li v1,-1
  0040658: 2402ffff li v0,-1
  004065c: 03e00008 jr ra
  0040660: 00000000 nop
• 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

Test success, if yes, branch to return from function

If failure, store code in errno

Set read() result to -1
The read() syscall function part 2

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

<table>
<thead>
<tr>
<th>Exception:</th>
<th>Note k0, k1 registers available for kernel use</th>
</tr>
</thead>
<tbody>
<tr>
<td>move k1, sp</td>
<td>/* Save previous stack pointer in k1 */</td>
</tr>
<tr>
<td>mflo k0, c0_status</td>
<td>/* Get status register */</td>
</tr>
<tr>
<td>andi k0, k0, CST_Kup</td>
<td>/* Check we-were-in-user-mode bit */</td>
</tr>
<tr>
<td>beq k0, $0, 1f</td>
<td>/* If clear, from kernel, already have stack */</td>
</tr>
<tr>
<td></td>
<td>/* delay slot */</td>
</tr>
<tr>
<td>/* Coming from user mode - load kernel stack into sp */</td>
<td></td>
</tr>
<tr>
<td>lw k0, curkstack</td>
<td>/* Get address of &quot;curkstack&quot; */</td>
</tr>
<tr>
<td>lw sp, 0(k0)</td>
<td>/* Get its value */</td>
</tr>
<tr>
<td>nop</td>
<td>/* delay slot for the load */</td>
</tr>
<tr>
<td>1:</td>
<td></td>
</tr>
<tr>
<td>mflo k0, c0_cause</td>
<td>/* Now, load the exception cause */</td>
</tr>
<tr>
<td>j common_exception</td>
<td>/* Skip to common code */</td>
</tr>
<tr>
<td>nop</td>
<td>/* delay slot */</td>
</tr>
</tbody>
</table>

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 real work starts here.

Save all the registers on the kernel stack.

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

These six stores are a “hack” to avoid confusing GDB. You can ignore the details of why and how.
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;
    /* 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
    */
    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 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)
        /* 140(sp) "saved" k0 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