System Calls

A high-level understanding of System Calls
- Mostly from the user’s perspective
  - From textbook (section 1.6)
- Exposure to architectural details of the MIPS R3000
  - Detailed understanding of the exception handling mechanism
    - From “Hardware Guide” on class website
- Understanding of the existence of compiler function calling conventions
  - Including details of the MIPS ‘C’ compiler calling convention
- 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.

Operating System

System Calls

Kernel Level

Operating System

Requests
(System Calls)

User Level

Applications

Applications

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>Process management</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fork</td>
<td>Create a child process identical to the parent.</td>
<td>Create a new process and return its Process ID.</td>
</tr>
<tr>
<td>kill</td>
<td>Send a signal to a process.</td>
<td>Sends a signal to a process.</td>
</tr>
<tr>
<td>execve</td>
<td>Replace the process’s address space with a new process image.</td>
<td>Replaces the process address space 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(name, mode)</td>
<td>Opens a file for reading, writing or both</td>
</tr>
<tr>
<td>s = close(f)</td>
<td>Closes an open file</td>
</tr>
<tr>
<td>read(buf, size, file)</td>
<td>Reads data from a file into a buffer</td>
</tr>
<tr>
<td>write(buf, size, file)</td>
<td>Writes data to a buffer into a file</td>
</tr>
<tr>
<td>ioctl(file, request, arg)</td>
<td>Calls the ioctl function</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)</td>
<td>Creates an empty directory</td>
</tr>
<tr>
<td>rename(name1, name2)</td>
<td>Creates a new entry, name2, pointing to name1</td>
</tr>
<tr>
<td>rmdir(name)</td>
<td>Removes a directory</td>
</tr>
<tr>
<td>umount(name)</td>
<td>Unmounts 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(name)</td>
<td>Changes the working directory</td>
</tr>
<tr>
<td>umount(name)</td>
<td>Unmounts a file system</td>
</tr>
<tr>
<td>times = time(NULL)</td>
<td>Gets 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 */
        /* child code */
        execve(command, parameters, 0); /* execute command */
    }
}
```

Some Win32 API calls

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

The MIPS R2000/R3000

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

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`

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

MIPS R3000

- RISC architecture – 5 stage pipeline
  - 32-bit instructions
  - 5 stages: Instruction Fetch (IF), Instruction Decode (ID), ALU/Accumulator (ALU), Memory Read (MEM), Write Back (WB)

Figure 1.1. MIPS 5-stage pipeline
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

```
jal 1f
nop
lw r4, (r6)
sw r2, (r3)
jr r31
nop
```

R3000 Virtual Memory Addressing

• MMU
  \[ \text{address translation in hardware} \]
  \[ \text{management of translation is software} \]

R3000 Translation

Unprivileged (User) Mode
\[ A_{phys} = \begin{cases} 
    f_{mmu}(A_{virt}) : & A_{virt} < 0x80000000 \\
    \end{cases} \]

Privileged (Kernel) Mode
\[ A_{phys} = \begin{cases} 
    f_{mmu}(A_{virt}) : & A_{virt} < 0x80000000 \\
    A_{virt} - 0x80000000 : & 0x80000000 < A_{virt} < 0xA0000000 \\
    A_{virt} - 0xA0000000 : & 0xA0000000 < A_{virt} < 0xC0000000 \\
    f_{mmu}(A_{virt}) : & A_{virt} \geq 0xC0000000 \\
    \end{cases} \]

R3000 Address Space Layout

• kuseg:
  \[ \text{2 gigabytes} \]
  \[ \text{MMU translated} \]
  \[ \text{Cacheable} \]
  \[ \text{Only kernel-mode accessible} \]
  \[ \text{Usually where the kernel code is placed} \]

R3000 Address Space Layout

• kseg0:
  \[ \text{512 megabytes} \]
  \[ \text{Fixed translation window to physical memory} \]
  \[ \text{MMU not used} \]
  \[ \text{Cacheable} \]
  \[ \text{Only kernel-mode accessible} \]

• kseg1:
  \[ \text{512 megabytes} \]
  \[ \text{Fixed translation window to physical memory} \]
  \[ \text{MMU not used} \]
  \[ \text{Cacheable} \]
  \[ \text{Only kernel-mode accessible} \]

0x00000000

0x80000000

0xA0000000

0xC0000000

0xFFFFF000

0xFFFFF3FF

0xA0000000

0xBFFFF000

0x80000000

0x1FFFFFFF
R3000 Address Space Layout

- kseg2:
  - 1024 megabytes
  - MMU translated
  - 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 coprocessor 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.

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

C0 Registers
- Exception Management
  - c0_cause
    - Cause of the recent exception
  - c0_status
    - Current status of the CPU
  - c0_spn
    - Address of the instruction that caused the exception
    - Note the 80 bit in c0_cause
  - c0_badvaddr
    - Address accessed that caused the exception
- Miscellaneous
  - c0_prd
    - 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
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

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>TLB modification</td>
</tr>
<tr>
<td>2</td>
<td>TLBE</td>
<td>“TLB load/TLBS store”</td>
</tr>
<tr>
<td>3</td>
<td>TLBS</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>MREL</td>
<td>Address error (on load/1-fetch or store respectively). Either an attempt to access outside kernel when in user mode, or an attempt to read a word or half word at a misaligned address.</td>
</tr>
</tbody>
</table>

Table 3.2. ExcCode values: different kinds of exceptions

Exception Vectors

<table>
<thead>
<tr>
<th>Program address</th>
<th>“segment”</th>
<th>Physical Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0a0000 0000</td>
<td>kseg0</td>
<td>0a0000 0000</td>
<td>TLB misses on kernel reference only.</td>
</tr>
<tr>
<td>0a0000 0080</td>
<td>kseg0</td>
<td>0a0000 0080</td>
<td>All exceptions.</td>
</tr>
<tr>
<td>0a00c0 0100</td>
<td>kseg1</td>
<td>0a10c0 0100</td>
<td>Unreached alternative kernel TLB miss entry point (used if 8M-bit BEV set).</td>
</tr>
<tr>
<td>0a00c0 0180</td>
<td>kseg1</td>
<td>0a10c0 0180</td>
<td>Unreached alternative for all other exceptions, used if 64-bit BEV set.</td>
</tr>
<tr>
<td>0a00c0 0000</td>
<td>kseg1</td>
<td>0a10c0 0000</td>
<td>The “reset exception”.</td>
</tr>
</tbody>
</table>

Table 4.1. Reset and exception entry points (vectors) for R3000 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>
</table>

Aside: We have ignored BD-bit in c0_cause which is also used in reality on rare occasions.
Hardware exception handling

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

PC: 0x12345678
EPC: 0x12345678

• Cause
Status: KU I E o K U p I E p K U c I E c

? ? ? ? 1 1

Hardware exception handling

Interrupts disabled and previous state shifted along

Status: K U o I E o K U p I E p K U c I E c

? ? 1 1 0 0

Kernelp Mode is set, and previous mode shifted along

Hardware exception handling

0x12345678

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

PC: 0x12345678
EPC: 0x12345678

• Status

K U o I E o K U p I E p K U c I E c

? ? 1 1 0 0

Address of general exception vector placed in PC

Hardware exception handling

0x80000080

• 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

PC: 0x80000080
EPC: 0x12345678

• Cause

Status: K U o I E o K U p I E p K U c I E c

? ? 1 1 0 0

Badvaddr

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

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

Store the EPC back in the PC.

Returning from an exception

- lw r27, saved_epc
- nop
- jr r27
- rfe

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

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().

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().

Stack Frame

- f1() stack frame
- f2() stack frame
- f3() stack frame

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 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 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

- 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

```c
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

Caution

- Seriously low-level code follows
- This code is not for the faint hearted
The `read()` syscall function part 1

0040064c <read>:
- 08100190 j 400640 <__syscall>
- 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>:
- 0000000c syscall
- 00000005 beqz a3,40065c <__syscall+0x1c>
- 00000000 nop
- 3c011000 lui at,0x1000
- 2403fff7 li v1,-1
- 03a00008 jr ra
- 00000000 nop

Generate a syscall exception

Test success, if yes, branch to return from function

If failure, store code in errno

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:

```assembly
move k1, sp /* Save previous stack pointer in k1 */
sf0 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 sp, 0(k0) /* Get its value */
nop /* delay slot for the load */
1:
sf0 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:

```assembly
/*
 * 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
```

exception:

```assembly
move k1, sp /* Save previous stack pointer in k1 */
sf0 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 sp, 0(k0) /* Get address of "curkstack" */
lw sp, 0(k0) /* Get its value */
 nop /* delay slot for the load */
1:
sf0 k0, c0_cause /* Now, load the exception cause. */
j common_exception /* Skip to common code */
 nop /* delay slot */
```

/* The order here must match mips/include/trapframe.h. */

```assembly
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 */
sf0 k0, c0_dp /* Copr.0 reg 13 == PC for */
sw k1, 160(sp) /* real saved PC */
```

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

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'
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

```assembly
/* 14(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 */
/* no need to restore tf_cause */
/* restore special registers */
lw t1, 24(sp)
lw t0, 13(sp)
mtlo t1
mtih t0
/* load the general registers */
lw ra, 36(sp)
lw s8, 156(sp)
lw sp, 152(sp)
/* done */
jr k0 /* jump back */
rfe /* in delay slot */
.end common_exception
```

Note again that only k0, k1 have been trashed.