System Calls

Learning Outcomes

• A high-level understanding of System Calls
  – Mostly from the user’s perspective
    • From textbook (section 1.6)
  • Exposure 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

• 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.

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>0</td>
<td>create a child process identifier to the parent</td>
</tr>
<tr>
<td>1</td>
<td>fork a child to become a new task</td>
</tr>
<tr>
<td>...</td>
<td>other system calls, e.g., exec, wait, signal...</td>
</tr>
<tr>
<td>100</td>
<td>terminate process execution and return to caller</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(File, mode)</td>
<td>Open a file for reading, writing or both</td>
</tr>
<tr>
<td>f = creat(FileName)</td>
<td>Create an open file</td>
</tr>
<tr>
<td>f = read(Buffer, byte)</td>
<td>Read data from a file into a buffer</td>
</tr>
<tr>
<td>f = write(Buffer)</td>
<td>Write data from a buffer into a file</td>
</tr>
<tr>
<td>f = flush(FileName)</td>
<td>Move the file pointer</td>
</tr>
<tr>
<td>f = stat(FileName)</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(FileName)</td>
<td>Create a new directory</td>
</tr>
<tr>
<td>s = rmdir(FileName)</td>
<td>Remove an empty directory</td>
</tr>
<tr>
<td>s = chdir(FileName)</td>
<td>Change the current directory</td>
</tr>
<tr>
<td>s = getdir(FileName)</td>
<td>Get a directory's contents</td>
</tr>
<tr>
<td>s = readlink(FileName)</td>
<td>Get a file's symbolic 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>p = printf(Format, ...)</td>
<td>Display formatted output</td>
</tr>
<tr>
<td>p = scanf(Format, ...)</td>
<td>Read data from the terminal</td>
</tr>
<tr>
<td>p = gettime(FileName)</td>
<td>Get the file's creation time</td>
</tr>
<tr>
<td>p = puttime(FileName)</td>
<td>Set the file's access time</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 */
        execve (command, parameters, 0); /* execute command */
    }
}
```

System Calls

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

The MIPS R2000/R3000

Some Win32 API calls
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: \texttt{sw r4, (r5)}
    - Store contents of r4 in memory using address contained in register r5
    - Load word: \texttt{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.
  - \texttt{lw, sw, lb, sb, lh, sh,....}

- Arithmetic and logical operations are register to register operations
  - E.g., \texttt{add r3, r2, r1}
  - No arithmetic operations on memory
  - Example
    - \texttt{add r3, r2, r1} \Rightarrow r3 = r2 + r1
  - Some other instructions
    - \texttt{add, sub, and, or, xor, sll, srl}
    - \texttt{move r2, r1} \Rightarrow r2 = r1

- 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: \texttt{addi r2, r1, 2048} \Rightarrow r2 = r1 + 2048
    - Load Immediate: \texttt{li r2, 1234} \Rightarrow r2 = 1234

Example code

Simple code example: \texttt{a = a + 1}

\begin{verbatim}
   lw r4,32(r29)  // r29 = stack pointer
   li r5, 1
   add r4, r4, r5
   sw r4,32(r29)
\end{verbatim}

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 of jump
  - \texttt{li r2, 1}
  - \texttt{sw r0,(r3)}
  - \texttt{j 1f}
  - \texttt{li r2, 2}
  - \texttt{li r2, 3}
  - \texttt{l: sw r2, (r3)}
MIPS R3000

- RISC architecture – 5 stage pipeline
  - Instruction partially through pipeline prior to jmp having an effect

![MIPS 5-stage pipeline](image)

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

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

Coprocessor 0

- 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](image)

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
  - 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_status](image)

- 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
**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](image)

---

**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>TLB</td>
<td>TLB fault/TLB store</td>
</tr>
<tr>
<td>3</td>
<td>TESI</td>
<td>Address error on load/1-fetch or store respectively. Either an attempt to access outsideousing when in user mode, or an attempt to read a word or half word at an unsignaled address.</td>
</tr>
<tr>
<td>4</td>
<td>AdEUC</td>
<td>Address error on load/1-fetch or store respectively. Either an attempt to access outsideousing when in user mode, or an attempt to read a word or half word at an unsignaled address.</td>
</tr>
</tbody>
</table>

![Table 3.2. ExcCode values: different kinds of exceptions](image)

---

**c0_epc**

- The Exception Program Counter
  - Points to address of where to restart execution after handling the exception or interrupt
  - Example
    - Assume `aw r3, (r4)` causes a restartable fault exception
  - Aside: We are ignoring BD bit in c0_cause which is also used in reality on rare occasions.

![Diagram of c0_epc](image)

---

**Exception Vectors**

<table>
<thead>
<tr>
<th>Program Address</th>
<th>“segment”</th>
<th>Physical Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8000 0000</td>
<td>kseg0</td>
<td>0x4000 0000</td>
<td>TLB misses on paging reference only</td>
</tr>
<tr>
<td>0x8000 0080</td>
<td>kseg0</td>
<td>0x4000 0080</td>
<td>All other exceptions.</td>
</tr>
<tr>
<td>0x5f00 0100</td>
<td>kseg1</td>
<td>0x9f0c 0100</td>
<td>Uncharted alternative fetch TLB miss entry point (used if SLE bit BNV set)</td>
</tr>
<tr>
<td>0x5f00 0180</td>
<td>kseg1</td>
<td>0x9f0c 0180</td>
<td>Uncharted alternative for all other exceptions, used if SIB bit BNV set.</td>
</tr>
<tr>
<td>0x8000 0000</td>
<td>kseg1</td>
<td>0x9f0c 0000</td>
<td>The “reset exception”.</td>
</tr>
</tbody>
</table>

![Table 4.1. Reset and exception entry points (vectors) for R3000 family](image)

---

**Simple Exception Walk-through**

User Mode

- Application

Kernel Mode

- Interrupt Handler

![Diagram of Simple Exception Walk-through](image)
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.

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.

Address of general exception vector placed in PC.
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

\[
\text{lw } r27, \text{ saved_epc} \\
\text{nop} \\
\text{jr } r27 \\
\text{rfe}
\]

Store the EPC back in the PC

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

Returning from an exception
• Each function call allocates a new stack frame for local variables, the return
  address, previous frame pointer etc.
  – Frame pointer: start of current stack frame
  – Stack pointer: end of current stack frame
• Example: assume \texttt{f1()} calls \texttt{f2()}, which calls \texttt{f3()}.

\textbf{Function Stack Frames}

\textbf{Stack Pointer} \rightarrow \textbf{Stack Frame} \rightarrow \textbf{f1()} stack frame
Function Stack Frames

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

Compiler 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 i;
  int sixargs(int a, int b, int c, int d, int e, int f)
  {
    return a + b + c + d + e + f;
  }
  i = sixargs(1, 2, 3, 4, 5, 6);
}
```
System Calls

Continued

User and Kernel Execution

- Simplistically, CPU 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 kernel entry point
  - 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
    - 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 – Calling `read()`**

- **Three arguments, one return value**
- **Code fragment calling the `read()` function**

```assembly
400124: 02602021 move a0, a3
400128: 27a50010 addiu al, 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**

---

**Inside the `read()` syscall function part 1**

```assembly
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**

```assembly
00400640 <__syscall>:
400640: 0000000c syscall
400644: 10a00005 beqz a3, 40065c <__syscall+0x1c>
400648: 00000000 nop
40064c: 3c011000 lui at, 0x1000
400650: ac220000 sw v0, 0(at)
400654: 2402ffff li v1, -1
400658: 2402ffff li v0, -1
40065c: 03a00008 jr ra
400660: 00000000 nop
```

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

---

**Inside the `read()` syscall function part 2**

```assembly
0400640: 0000000c syscall
400644: 10a00005 beqz a3, 40065c <__syscall+0x1c>
400648: 00000000 nop
40064c: 3c011000 lui at, 0x1000
400650: ac220000 sw v0, 0(at)
400654: 2402ffff li v1, -1
400658: 2402ffff li v0, -1
40065c: 03a00008 jr ra
400660: 00000000 nop
```

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

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 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

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

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
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;
    u_int32_t tf_t7;
    u_int32_t tf_s0; 
    u_int32_t tf_s7; 
    u_int32_t tf_t8;
    u_int32_t tf_t9;
    u_int32_t tf_k0;/* dummy (see exception.S comments) */
    u_int32_t tf_k1;/* dummy */
    u_int32_t tf_gp;
    u_int32_t tf_sp;
    u_int32_t tf_s8;
    u_int32_t tf_epc; /* coprocessor 0 epc register */
};

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

Kernel Stack

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

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, 28(sp)
    lw t0, 32(sp)
    mthi t1
    mtlo 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)
    lw t8, 132(sp)
    lw t9, 136(sp)
/* 140(sp) */
/* saved k1 was dummy garbage anyway */
/* 144(sp) */
/* saved k1 was dummy garbage anyway */