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 of exception handling mechanism
    • From “Hardware Guide” on class web site

• 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.

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>getpid()</td>
<td>Obtain a child process identifier to the parent</td>
</tr>
<tr>
<td>waitpid((pid, status, options)</td>
<td>Wait for a child to terminate</td>
</tr>
<tr>
<td>writev()</td>
<td>Replace a process' own image</td>
</tr>
<tr>
<td>exit(EXIT)</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>move</td>
<td>Move the file pointer</td>
</tr>
<tr>
<td>link</td>
<td>Create a hard link to a file</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 an empty directory</td>
</tr>
<tr>
<td>rename</td>
<td>Rename an entry, named <code>name1</code> to <code>name2</code></td>
</tr>
<tr>
<td>chown</td>
<td>Change the owner and group of a file</td>
</tr>
<tr>
<td>umount</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>chdir</td>
<td>Change the working directory</td>
</tr>
<tr>
<td>chdev</td>
<td>Change the device number</td>
</tr>
<tr>
<td>getcurtime</td>
<td>Get the clock time since January 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

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 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 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

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

```
li r2, 1
sw r0, (r3)
j 1f
li r2, 2
li r2, 3
1: sw r2, (r3)
```

```
0x10 jal 1f
0x14 nop
0x18 lw r4, (r6)
1:
0x2a sw r2, (r3)
0x38 jr r31
0x3a nop
```
**MIPS R3000**

- RISC architecture – 5 stage pipeline

**Figure 1.1: MIPS 5-stage pipeline**

**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**

- **Exception Management**
  - `c0_cause`
    - Cause of the recent exception
  - `c0_status`
    - Current status of the CPU
- **Memory Management**
  - `c0_index`
  - `c0_random`
  - `c0_entryhi`
  - `c0_entrylo`
  - `c0_context`
  - More about these later in course

**c0_status**

- For practical purposes, you can ignore most bits
  - Green background is the focus

**Figure 3.2. Fields in status register (SE)**

- **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**

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

- **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>Exception Code</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>TLB</td>
<td>TLB modification</td>
</tr>
<tr>
<td>2</td>
<td>EML</td>
<td>TLB fault/TLB source</td>
</tr>
<tr>
<td>3</td>
<td>MEB</td>
<td>Address error on load/1 fetch or store respectively. Either an attempt to access outside kbase when in user mode, or an attempt to read a word or half-word at an unaligned address.</td>
</tr>
<tr>
<td>4</td>
<td>MEB</td>
<td>Address error on load/1 fetch or store respectively. Either an attempt to access outside kbase 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. Exception codes: different kinds of exceptions

### Exception Vectors

<table>
<thead>
<tr>
<th>Program Address</th>
<th>Physical</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000 0000</td>
<td>0x0000 0000</td>
<td>TLB miss on kseg reference only</td>
</tr>
<tr>
<td>0x0000 0000</td>
<td>0x0000 0000</td>
<td>All other exceptions.</td>
</tr>
<tr>
<td>0x0f00 0100</td>
<td>0x0f00 0100</td>
<td>Uncaught alternative kseg TLB miss entry point (used if 32-bit R3000 only)</td>
</tr>
<tr>
<td>0x0f00 0100</td>
<td>0x0f00 0100</td>
<td>Uncaught alternative for all other exceptions, used if 64-bit R3000 only.</td>
</tr>
<tr>
<td>0x0f00 0000</td>
<td>0x0f00 0000</td>
<td>The &quot;reset exception&quot;</td>
</tr>
</tbody>
</table>

Table 4.1. Reset and exception entry points (vectors) for R3000 family

### Simple Exception Walk-through

**User Mode**

```
PC 0x12345678
```

**Kernel Mode**

```
EPC 0x12345678
```

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.

### Hardware exception handling

- **PC**
  - User Mode: 0x12345678
  - Kernel Mode: ?

- **EPC**
  - User Mode: ?
  - Kernel Mode: ?

- **Cause**

- **Status**
  - User Mode: KUo Ile KUp Ile KUc Ile
  - Kernel Mode: ?? ?? ?? ?? 1 1
Hardware exception handling

- 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>

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

<table>
<thead>
<tr>
<th>Cause</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>KUo IEo KUp IEp KUc IEc</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PC</th>
<th>EPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x080000080</td>
<td>0x12345678</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>Cause</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>KUo IEo KUp IEp KUc IEc</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PC</th>
<th>EPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08000080</td>
<td>0x12345678</td>
</tr>
</tbody>
</table>

Returning from an exception

- For now, lets ignore
  - how the exception is actually handled
  - how user-level registers are preserved
- Let’s simply look at how we return from the exception

<table>
<thead>
<tr>
<th>Cause</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>KUo IEo KUp IEp KUc IEc</td>
</tr>
</tbody>
</table>

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

• Interrupts disabled and previous state shifted along

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

Kernel Mode is set, and previous mode shifted along

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

Address of general exception vector placed in PC

<table>
<thead>
<tr>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>? 1 1 0 0</td>
</tr>
</tbody>
</table>
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 moved earlier to somewhere in memory by the exception handler.

Store the EPC back in the PC.

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.
  - Frame pointer: start of current stack frame
  - Stack pointer: end of current stack frame

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.
  - 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?

Example Code

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

Continued

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

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.
- 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 – Calling read()

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: 00400201 move a0,v0
400134: la000016 bnez a0,400194 <docat+0x94>

• Args are loaded, return value is tested

Inside the read() syscall function part 1

0040068c <read>:
40068c: 0000000c 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

Generate a syscall exception

00400640 <__syscall>:
400640: 0000000c syscall
400644: 10a00005 bgeq a3,40065c <__syscall+0xlc>
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: 03a00008 jr ra
400660: 00000000 nop

Test success, if yes, branch to return from function

00400640 <__syscall>:
400640: 0000000c syscall
400644: 10a00005 bgeq a3,40065c <__syscall+0xlc>
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: 03a00008 jr ra
400660: 00000000 nop

If failure, store code in errno

The read() syscall function part 2

00400640 <__syscall>:
400640: 0000000c syscall
400644: 10a00005 bgeq a3,40065c
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: 03a00008 jr ra
400660: 00000000 nop

Set read() result to -1

The read() syscall function part 2

00400640 <__syscall>:
400640: 0000000c syscall
400644: 10a00005 bgeq a3,40065c
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: 03a00008 jr ra
400660: 00000000 nop
The read() syscall function part 2

00400640 <__syscall>:
400640: 0000000c syscall
400644: 10e00005 breq a3,40065c
400648: 00000000 nop
40064c: 3e011000 lui a0,0x1000
400650: a8230000 sw v0,0(at)
400654: 2403ffff li v1,-1
400658: 2402ffff li v0,-1
40065c: 03e00008 jr ra
400660: 00000000 nop

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 */
sf0 k0, c0_status /* Get status register */
andi k0, k0, CST_Kup /* Check the we-were-in-user-mode bit */
beg 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)     /* Use */
1:
    sf0 k0, c0_cause /* Now, load the exception cause. */
} common_exception
    /* skip to common code */
    nop /* delay slot */

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

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

sw ra, 160(sp) /* dummy for gdb */
sw s8, 156(sp) /* save s8 */
sw gp, 148(sp) /* save gp */
sw k0, 140(sp) /* dummy for gdb */
sw k1, 152(sp) /* real saved sp */
nop /* delay slot for store */

mfc0 k1, c0_epc /* Copr.0 reg 13 == PC for exception */
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.

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'.

The real work starts here.

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'.

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.
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
    • ‘mips_trap’ eventually returns

    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
    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 sp, 72(sp)
    /* need to restore sp_vaddr */
    lw gp, 76(sp)
    /* 144(sp) stack pointer - below */
    lw s8, 156(sp)
    lw k0, 160(sp)
    lw sp, 164(sp)
    jr k0 /* jump back */
    rfe /* in delay slot */
    .end common_exception

Note again that only k0, k1 have been trashed