Learning Outcomes

- 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
- A high-level understanding of System Calls
  - Mostly from the user's perspective
    - From textbook (section 1.6)
- 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.

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.
      - \( \text{lw, sw, lb, sb, lh, sh,...} \)

MIPS R3000

- 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 \Rightarrow r3 = r2 + r1`
- Some other instructions
  - `add, sub, and, or, xor, sll, srl`
  - `move r2, r1 \Rightarrow r2=r1`

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`\( \Rightarrow r2 = r1 + 2048 \)
    - Load Immediate: `li r2, 1234`\( \Rightarrow r2 = 1234 \)
Example code

Simple code example: \( a = a + 1 \)

\[
\begin{align*}
\text{lw} & \ r4, 32(r29) & // r29 = \text{stack pointer} \\
\text{li} & \ r5, 1 \\
\text{add} & \ r4, r4, r5 \\
\text{sw} & \ r4, 32(r29)
\end{align*}
\]

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

\[
\begin{align*}
\text{li} & \ r2, 1 \\
\text{sw} & \ r0, (r3) \\
\text{i} & \ 1f \\
\text{li} & \ r2, 2 \\
\text{li} & \ r2, 3 \\
\text{sw} & \ r2, (r3)
\end{align*}
\]

MIPS R3000

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

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

\[
\begin{align*}
0x10 & \text{jal} 1f \\
0x14 & \text{nop} \\
0x18 & \text{lw} r4, (r6) \\
1: & \text{sw} r2, (r3) \\
0x2a & \text{jr} r31 \\
0x3a & \text{nop}
\end{align*}
\]

Compiler Register Conventions

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

<table>
<thead>
<tr>
<th>Reg No</th>
<th>Name</th>
<th>Used for</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>zero</td>
<td>Always returns 0</td>
</tr>
<tr>
<td>1</td>
<td>at</td>
<td>Assembly temporary; Reserved for use by assembler</td>
</tr>
<tr>
<td>2-3</td>
<td>$t0-$t1</td>
<td>Stack locations; FPRs returned by subroutine</td>
</tr>
<tr>
<td>4</td>
<td>$s0</td>
<td>Frame pointer of current frame</td>
</tr>
<tr>
<td>8-15</td>
<td>$t2-$t7</td>
<td>Temporary subroutines may use without saving</td>
</tr>
<tr>
<td>16-23</td>
<td>$s8-$s15</td>
<td>Temporary subroutines; Saved on return from subroutine</td>
</tr>
<tr>
<td>24-27</td>
<td>$s16-$s19</td>
<td>Saved for use by interrupt/top handler—stay change stack frame if valid first argument called on stack frame</td>
</tr>
<tr>
<td>28</td>
<td>gp</td>
<td>Global pointer; some runtime systems maintain this to give long access to pointers “static” or “external” variables</td>
</tr>
<tr>
<td>29</td>
<td>sp</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>30</td>
<td>$s10</td>
<td>80 register variable; system registers which need one may use this on or “top” pointer</td>
</tr>
<tr>
<td>31</td>
<td>ra</td>
<td>Return address for subroutine</td>
</tr>
</tbody>
</table>

Simple factorial

```c
int fact(int n) {
    int r = 1;
    int i;
    for (i = 1; i < n+1; i++) {
        r = r * i;
    }
    return r;
}
```

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().

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().

MIPS calling convention for gcc

- Args 1-4 have space reserved for them
Example Code

```c
main () {
    int i;
    i = sixargs(1, 2, 3, 4, 5, 6);
}
```

```c
int sixargs(int a, int b, int c, int d, int e, int f) {
    return a + b + c + d + e + f;
}
```

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.

- 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

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

### c0_cause

- **IP**
  - Intermits pending
- **CE**
  - Coprocessor error
  - Attempt to access disabled Copro.

### Exception Codes

#### | ExecCode | Mnemonic | Description |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Int</td>
<td>Interrupt</td>
</tr>
<tr>
<td>1</td>
<td>Mod</td>
<td>“ILB modified”</td>
</tr>
<tr>
<td>2</td>
<td>ILB</td>
<td>“ILB fault/TLB store”</td>
</tr>
<tr>
<td>3</td>
<td>ILIS</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>AdEL</td>
<td>Address error (on load/1-fetch or store respectively). Either an attempt to access outside known when in user mode, an attempt to read a word or half-word at a nonaligned address.</td>
</tr>
<tr>
<td>5</td>
<td>XRES</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2: ExecCode values: different kinds of exceptions

### c0_epc

- The Exception Program Counter
  - Points to address of where to restart execution after handling the exception or interrupt
  - Example
    - Assume `sw r3,(r4)` causes a restartable fault exception

Aside: We are ignore BD-bit in c0_cause which is also used in reality only on rare occasions.
### Exception Vectors

<table>
<thead>
<tr>
<th>Program Address</th>
<th>“segment”</th>
<th>Physical Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000 0000</td>
<td>kseg0</td>
<td>0x0000 0000</td>
<td>TLB miss on data reference only</td>
</tr>
<tr>
<td>0x8000 0000</td>
<td>kseg0</td>
<td>0x0000 0000</td>
<td>All other exceptions.</td>
</tr>
<tr>
<td>0x4000 0100</td>
<td>kseg1</td>
<td>0x4000 0100</td>
<td>Uncached alternative kseg TLB miss entry point (used if S bit of E bit set).</td>
</tr>
<tr>
<td>0x4000 0100</td>
<td>kseg1</td>
<td>0x4000 0100</td>
<td>Uncached alternative for all other exceptions, used if S bit E bit set.</td>
</tr>
<tr>
<td>0x4000 0000</td>
<td>kseg0</td>
<td>0x4000 0000</td>
<td>The “reset exception”.</td>
</tr>
</tbody>
</table>

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

### Simple Exception Walk-through

**User Mode**

- Application

**Kernel Mode**

- Interrupt Handler

### Hardware Exception Handling

- **PC** 0x12345678
- **EPC**
- **Cause**
- **Status** KUo IEo KUp IEp KUc IEc

- 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

- **PC** 0x12345678
- **EPC**
- **Cause**
- **Status** KUo IEo KUp IEp KUc IEc

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

- **PC** 0x12345678
- **EPC**
- **Cause**
- **Status** KUo IEo KUp IEp KUc IEc

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

- **PC** 0x12345678
- **EPC**
- **Cause**
- **Status** KUo IEo KUp IEp KUc IEc

- Kernel Mode is set, and previous mode shifted along
Hardware exception handling

**PC** 0x80000080

**EPC** 0x12345678

- 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

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

```
0x80000080
0x12345678
PC EPC
Cause
Status
0
KUo IEo KUp IEp KUc IEc
0
0
1
1
?
?

This code to return is:
```
```w
lw r27, saved_epc
nop
jr r27
rfe
```

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

```
0x12345678
0x12345678
PC EPC
Cause
Status
0
KUo IEo KUp IEp KUc IEc
0
0
1
1
?
?

This code to return is:
```
```w
lw r27, saved_epc
nop
jr r27
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

<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>0</td>
<td>KUc IEc</td>
</tr>
</tbody>
</table>

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.

Operating System

- 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

A Brief Overview of Classes

<table>
<thead>
<tr>
<th>System Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requests (System Calls)</td>
</tr>
<tr>
<td>Applications</td>
</tr>
<tr>
<td>Applications</td>
</tr>
</tbody>
</table>

Some System Calls For Process Management

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sys = waitpid</td>
<td>Wait for a child to terminate</td>
</tr>
<tr>
<td>fork</td>
<td>Create a child process identical to the parent</td>
</tr>
<tr>
<td>execve</td>
<td>Replace a process’s code image</td>
</tr>
<tr>
<td>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>f = open(filename, mode)</td>
<td>Open a file for reading, writing or both</td>
</tr>
<tr>
<td>c = close(f)</td>
<td>Close an open file</td>
</tr>
<tr>
<td>t = read(f, buffer, size)</td>
<td>Read data from a file into a buffer</td>
</tr>
<tr>
<td>t = write(f, buffer, size)</td>
<td>Write data from a buffer into a file</td>
</tr>
<tr>
<td>t = tell(f)</td>
<td>Move the file pointer</td>
</tr>
<tr>
<td>t = stat(name, &amp;st)</td>
<td>Get a file's various 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(name)</td>
<td>Create a new directory</td>
</tr>
<tr>
<td>s = rmdir(name)</td>
<td>Remove an empty directory</td>
</tr>
<tr>
<td>s = lnkname()</td>
<td>Remove a directory entry</td>
</tr>
<tr>
<td>s = lnkname(name, name)</td>
<td>Create a new entry, name, pointing to name</td>
</tr>
<tr>
<td>s = mount(special, name, flag)</td>
<td>Mount a file system</td>
</tr>
<tr>
<td>s = umount(special)</td>
<td>Unmount a file system</td>
</tr>
</tbody>
</table>

Some System Calls For Miscellaneous Tasks

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

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 user-level
  - Kernel entry via the designated entry point must be strictly enforced

- 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 – Calling read()**

- **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,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**

**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: 10e00005 beqz a3,40065c <__syscall+0x1c>
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: 03e00008 jr ra
```

- **• Generate a syscall exception**

**The read() syscall function part 2**

```assembly
00400640 <__syscall>:
400640: 0000000c syscall
400644: 10e00005 beqz a3,40065c <__syscall+0x1c>
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: 03e00008 jr ra
```

- **Test success, if yes, branch to return from function**
The read() syscall function part 2

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 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
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 */
beg k0, $0, 1f /* If clear, from kernel, already have stack */
nop /* delay slot */

/* Coming from user mode - load kernel stack into sp */
la k0, curkstack /* get address of "curkstack" */
lw sp, 0(k0) /* get its value */
nop /* delay slot for the load */

1:
mfc0 k0, c0_cause /* Now, load the exception cause. */
j common_exception /* Skip to common code */
nop /* delay slot */

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

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

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

/*
* Save special registers.
*/
mfhi t0
mflo t1
sw t0, 32(sp)
w t1, 20(sp)

/*
* Save remaining exception context information.
*/
sw k0, 24(sp) /* k0 was loaded with cause earlier */
mfc0 t1, c0_status /* Copr 0 reg 11 == status */
w t1, 20(sp)
mfc0 t2, c0_vaddr /* Copr 0 reg 8 == faulting vaddr */
w t2, 14(sp)
/*
* Pretend to save $0 for gdb's benefit.
*/
sw $0, 12(sp)
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)
{

/* 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 trapframe structure contains the saved registers and state */

/* The trapframe 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:
/* Need to restore tf_vaddr */
lw t0, 20(sp) /* load status register value into t0 */
mtc0 t0, c0_status /* store it back to coprocessor 0 */
/* Need to restore tf_cause */
lw t1, 28(sp) /* load the cause register */
mtc0 t1, c1_status /* store it back to coprocessor 1 */
/* Need to restore the special registers */
lw t2, 32(sp)
mtc0 t2, c2_status
/* Need to restore the general registers */
lw t3, 36(sp)
lw s0, 40(sp)
lw s1, 44(sp)
lw s2, 48(sp)
lw s3, 52(sp)
lw a0, 56(sp)
lw a1, 60(sp)
lw a2, 64(sp)
lw a3, 68(sp)
lw t4, 72(sp)
lw t5, 76(sp)
lw t6, 80(sp)
lw t7, 84(sp)
lw k0, 88(sp)
lw k1, 92(sp)
lw k2, 96(sp)
lw k3, 100(sp)
lw k4, 104(sp)
lw k5, 108(sp)
lw k6, 112(sp)
lw k7, 116(sp)
lw a0, 120(sp)
lw a1, 124(sp)
lw a2, 128(sp)
lw a3, 132(sp)
lw a4, 136(sp)
lw a5, 140(sp)
lw a6, 144(sp)
/* Saved k0 was dummy garbage anyway */
/* Saved k1 was dummy garbage anyway */
lw gp, 148(sp) /* restore gp */
/* 152(sp) */ stack pointer - below */
lw s8, 152(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 */
sfe /* in delay slot */
.end common_exception

Note again that only k0, k1 have been trashed