System Calls

Operating System
System Calls

Kernel Level

Operating System

Requests
(System Calls)

Applications

Applications

User Level

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

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 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 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.

Some System Calls For Process Management

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sysopen()</code></td>
<td>Create a new process with the parent.</td>
</tr>
<tr>
<td><code>write()</code></td>
<td>Writes with a specified offset.</td>
</tr>
</tbody>
</table>

Examples:
- `sysopen()` creates a new process with the parent.
- `write()` writes with a specified offset.
Some System Calls For File Management

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Open a file for reading, writing or both</td>
</tr>
<tr>
<td>b</td>
<td>Close a file</td>
</tr>
<tr>
<td>r</td>
<td>Read or group TTY</td>
</tr>
<tr>
<td>n</td>
<td>Read from stdin in bytes, not bytes</td>
</tr>
<tr>
<td>w</td>
<td>Write a file</td>
</tr>
<tr>
<td>p</td>
<td>Write to a file</td>
</tr>
<tr>
<td>s</td>
<td>Write to a file, write error 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>c</td>
<td>Create a new directory</td>
</tr>
<tr>
<td>l</td>
<td>Link a file to a file</td>
</tr>
<tr>
<td>r</td>
<td>Rename or move a file</td>
</tr>
<tr>
<td>d</td>
<td>Display file system information</td>
</tr>
<tr>
<td>s</td>
<td>Set the file system status</td>
</tr>
</tbody>
</table>

Some System Calls For Miscellaneous Tasks

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>Input from terminal</td>
</tr>
<tr>
<td>o</td>
<td>Output to terminal</td>
</tr>
<tr>
<td>d</td>
<td>Display prompt</td>
</tr>
<tr>
<td>s</td>
<td>Send a signal to a process</td>
</tr>
</tbody>
</table>

System Calls

- A stripped down shell:

  ```
  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 */
    }
  }
  ```

The MIPS R2000/R3000

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

- RISC architecture – 5 stage pipeline
- 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 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
  - Examples:
    - `li r2, 1`
    - `sw r0, (r3)`
    - `j 1f`
    - `li r2, 2`
    - `li r3, 3`
    - `lw r2, (r3)`
Jump and Link

- JAL is used to implement function calls
  - r31 = PC+8
- Return Address register (RA) is used to return from function call

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

R3000 Address Space Layout

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

```
io00000000
    kseg2
    kseg1
    kseg0
    kuseg
```

- kseg0:
  - 512 megabytes
  - Fixed translation window to physical memory
    - 0x80000000 - 0xbfffffff virtual = 0x00000000 - 0x1fffffff physical
    - MMU not used
  - Cacheable
  - Only kernel-mode accessible
  - Usually where the kernel code is placed

```
io00000000
    kseg2
    kseg1
    kseg0
    kuseg
```

- kseg1:
  - 512 megabytes
  - Fixed translation window to physical memory
    - 0xa0000000 - 0xbfffffff virtual = 0x00000000 - 0x1fffffff physical
    - MMU not used
  - NOT cacheable
  - Only kernel-mode accessible
  - Where devices are accessed (and boot ROM)

```
io00000000
    kseg2
    kseg1
    kseg0
    kuseg
```

- kseg2:
  - 1024 megabytes
  - MMU translated (mapped)
  - Cacheable
  - Only kernel-mode accessible

```
io00000000
    kseg2
    kseg1
    kseg0
    kuseg
```

System161 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 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_spc
    – 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 course

C0_status

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0</td>
<td>C1</td>
<td>C2</td>
<td>C3</td>
<td>C0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For practical purposes, you can ignore these bits
– Green background is the focus

CU0-3

• 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

- PE  - Parity error in cache
- CM  - Cache management
- PZ  - Cache parity zero
  
  Figure 3.2. Fields in status register (c0_status)

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PE</td>
<td>Parity error in cache</td>
</tr>
<tr>
<td>CM</td>
<td>Cache management</td>
</tr>
<tr>
<td>PZ</td>
<td>Cache parity zero</td>
</tr>
</tbody>
</table>

Exception Codes

<table>
<thead>
<tr>
<th>Exception Code 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 modified”</td>
</tr>
<tr>
<td>2</td>
<td>TLB</td>
<td>“TLB modified TLB reset”</td>
</tr>
<tr>
<td>3</td>
<td>IES</td>
<td>Address error (int load/3 fetch or store respectively)</td>
</tr>
<tr>
<td>4</td>
<td>AdES</td>
<td>Address error (int load/3 fetch or store respectively)</td>
</tr>
<tr>
<td>5</td>
<td>AdIES</td>
<td>Address error (int load/3 fetch or store respectively)</td>
</tr>
</tbody>
</table>

Table 3.2. Exception Code values: different kinds of exceptions

c0_cause

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

Exception Codes

<table>
<thead>
<tr>
<th>Exception Code Value</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>BPE</td>
<td>Bus error (instruction or data fault, respectively)</td>
</tr>
<tr>
<td>1</td>
<td>TBR</td>
<td>TLB miss (TLB miss)</td>
</tr>
<tr>
<td>2</td>
<td>TLB</td>
<td>“TLB modified TLB reset”</td>
</tr>
<tr>
<td>3</td>
<td>IES</td>
<td>Address error (int load/3 fetch or store respectively)</td>
</tr>
<tr>
<td>4</td>
<td>AdES</td>
<td>Address error (int load/3 fetch or store respectively)</td>
</tr>
<tr>
<td>5</td>
<td>AdIES</td>
<td>Address error (int load/3 fetch or store respectively)</td>
</tr>
<tr>
<td>6</td>
<td>CE</td>
<td>Coprocessor error</td>
</tr>
</tbody>
</table>
| 7                    | BD       | Branch delay slot

Table 3.2. Exception Code 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
  - BD-bit in c0_cause is used on rare occasions when one needs to identify the actual exception-causing instruction
  - Example
    - Assume sw r3, (r4) causes a page fault exception

```
 nop
 sw r3, (r4)  # sw r3, (r4) causes page fault exception
```

```
c0_epc BD = 0
```

```
nop
 sw r3, (r4)
```

```
c0_epc BD = 1
```

```
} printf
 sw r3, (r4)
```

```
c0_epc
```

```
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

```
 nop
 sw r3, (r4)  # sw r3, (r4) causes page fault exception
```

```
} printf
 sw r3, (r4)
```

```
```
### Exception Vectors

<table>
<thead>
<tr>
<th>Program address</th>
<th>“segment”</th>
<th>Physical Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0800 0000</td>
<td>kseg0</td>
<td>0x0800 0000</td>
<td>TLB misses an memory reference only.</td>
</tr>
<tr>
<td>0x0800 0080</td>
<td>kseg0</td>
<td>0x0800 0080</td>
<td>All other exceptions.</td>
</tr>
<tr>
<td>0x8000 0000</td>
<td>kseg1</td>
<td>0x8000 0000</td>
<td>Unrelated alternative kseg TLB miss entry point timed if SE bit (BEV set).</td>
</tr>
<tr>
<td>0x8000 0080</td>
<td>kseg1</td>
<td>0x8000 0080</td>
<td>Unrelated alternative for all other exceptions, used if SE bit (BEV set).</td>
</tr>
<tr>
<td>0x0800 0000</td>
<td>kseg1</td>
<td>0x0800 0000</td>
<td>The “reset exception”.</td>
</tr>
</tbody>
</table>

Table 4.1: Reset and exception entry points (vectors) for 130xx 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>

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

<table>
<thead>
<tr>
<th>Cause</th>
<th>Status</th>
<th>Badvaddr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x12345678</td>
<td>KUo IEo KUp IEp KUc IEc</td>
<td>?</td>
</tr>
</tbody>
</table>

- Kernel Mode is set, and previous mode shifted along

<table>
<thead>
<tr>
<th>EPC</th>
<th>Interrupts disabled and previous state shifted along</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>
<th>Badvaddr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x12345678</td>
<td>KUo IEo KUp IEp KUc IEc</td>
<td>?</td>
</tr>
</tbody>
</table>

- Address of general exception vector placed in PC
Hardware exception handling

- 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

Returning from an exception

- This code to return is:
  \- lw r27, saved_epc
  \- nop
  \- jr r27
  \- rfe

Returning from an exception

- This code to return is:
  \- lw r27, saved_epc
  \- nop
  \- jr r27
  \- rfe

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

Returning from an exception

- 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().

Stack Frame

- MIPS calling convention for gcc
  - Args 1-4 have space reserved for them
  - Local variables
  - Static space

Software Register Conventions

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

<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 return 0</td>
</tr>
<tr>
<td>1</td>
<td>at</td>
<td>(reserverd for use by assembler)</td>
</tr>
<tr>
<td>2-21</td>
<td>t0-t31</td>
<td>Value except FP returned by subroutine</td>
</tr>
<tr>
<td>27</td>
<td>s0-s15</td>
<td>Arguments to functions that have parameters</td>
</tr>
<tr>
<td>25-29</td>
<td>t0-t4</td>
<td>Subroutine register variables; a subroutine which calls another one of these must save the old value and restore it before a return, so the calling context saves these values temporarily.</td>
</tr>
<tr>
<td>24:31</td>
<td>t5-t12</td>
<td>Saved for use by the interrupt handler; may change without notice.</td>
</tr>
<tr>
<td>14</td>
<td>sp</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>28</td>
<td>$fp</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>31</td>
<td>$ra</td>
<td>31 registers variable. Subroutines which need one can use this as a 'frame pointer'.</td>
</tr>
<tr>
<td>8</td>
<td>$a0</td>
<td>Return address for subroutine.</td>
</tr>
</tbody>
</table>
**Example Code**

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

int sixargs(int a, int b, int c, int d, int e, int f) {
    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
- 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
  - 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

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

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: 0408021 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

Generate a syscall exception

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
  400660: 00000000 nop
  ```

If failure, store code in errno

Test success, if yes, branch to return from function

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
  400660: 00000000 nop
  ```

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

00400640 <__syscall>:  
400640: 0000000c syscall  
400644: 10e00005 beqz 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: 03e00008 jr ra  
400660: 00000000 nop

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

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

/*
 * Prepare to call mips_trap(struct trapframe *)
 */
addiu a0, sp, 16
jal mips_trap
/* call it */

*/

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

sw ra, 160(sp) /* dummy for gdb */
sw s8, 156(sp) /* save s8 */
sw sp, 152(sp) /* dummy for gdb */
sw gp, 148(sp) /* dummy for gdb */
sw k1, 144(sp) /* real saved sp */
sw k0, 140(sp) /* real saved 9C */

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 */

sw t9, 156(sp)
sw t8, 152(sp)
sw s8, 156(sp)
sw sp, 152(sp)
sw gp, 148(sp)
sw k1, 144(sp)
sw k0, 140(sp)
sw k1, 152(sp)

nop /* delay slot for store */
mfc0 k1, c0_epc /* Copr.0 reg 13 == PC for exception */
sw k1, 160(sp) /* real saved 9C */

sw k0, 24(sp) /* k0 was loaded with cause earlier */
mfc0 t0, c0_status /* Copr.0 reg 11 == status */
sw t0, 20(sp)
mfc0 t2, c0_vaddr /* Copr.0 reg 8 == faulting vaddr */
sw t2, 16(sp)

/*
 * Pretend to save $0 for gdb's benefit.
 */
sw $0, 12(sp)
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;       /* Saved register 31 */
    u_int32_t tf_hi;
    u_int32_t tf_ra;       /* 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;       /* Saved register 32 */
    u_int32_t tf_t7;       /* dummy (see exception.S comments */
    u_int32_t tf_s0;       /* Saved register 33 */
    u_int32_t tf_s7;       /* dummy */
    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'.

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;

    /* 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

    exception_return:
    /* 16(sp) no need to restore tf_vaddr */
    lw t0, 20(sp) /* load status register value into t0 */
    op /* load delay slot */
    addi t0, t0, -status /* store it back to coprocessor 0 */
    /* 24(sp) no need to restore tf_cause */
    /* restore special registers */
    lw t1, 32(sp)
    slli t1
    ori 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" k0 was dummy garbage anyway */
    /* 144(sp) "saved" k1 was dummy garbage anyway */
    lw gp, 148(sp) /* restore gp */
    /* 152(sp) stack pointer - below */
    lw s8, 156(sp) /* restore a7 */
    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