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 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.

Operating System
System Calls

Kernel Level

Operating System

User Level

Requests
(System Calls)

Applications

Applications

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

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>Process management</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>kill (pid, signal)</code></td>
<td></td>
<td>Send a signal to a process.</td>
</tr>
<tr>
<td><code>waitpid (pid, status, options)</code></td>
<td></td>
<td>Wait for a child to terminate.</td>
</tr>
<tr>
<td><code>mprotect (addr, size, prot)</code></td>
<td></td>
<td>Protect a portion of an image.</td>
</tr>
<tr>
<td>`ptrace (request, pid, /*address</td>
<td>data*/)`</td>
<td></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(...)</td>
<td>Open a file for reading, writing or both</td>
</tr>
<tr>
<td>s = close(...)</td>
<td>Close an open file</td>
</tr>
<tr>
<td>i = readfile(...)</td>
<td>Read file contents into a buffer</td>
</tr>
<tr>
<td>w = writefile(...)</td>
<td>Write data to a file from a buffer</td>
</tr>
<tr>
<td>f = linkfile(...)</td>
<td>Move the file to another location</td>
</tr>
<tr>
<td>r =unlinkfile(...)</td>
<td>Unlink a file from a directory</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 = mkdir(...)</td>
<td>Create an empty directory</td>
</tr>
<tr>
<td>c = rmdir(...)</td>
<td>Remove a directory</td>
</tr>
<tr>
<td>c = chdir(...)</td>
<td>Change the current directory</td>
</tr>
<tr>
<td>c = renamefile(...)</td>
<td>Rename a file to another name</td>
</tr>
<tr>
<td>c = symlink(...)</td>
<td>Make a symbolic link</td>
</tr>
<tr>
<td>c = unmount(...)</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>s = chdir(...)</td>
<td>Change the working directory to a new one</td>
</tr>
<tr>
<td>s = chown(...)</td>
<td>Change the owner of a file in the directory</td>
</tr>
<tr>
<td>s = chtime(...)</td>
<td>Change the timestamp of a file in the directory</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 */
    }
}
```

- 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: `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
    - `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 R3000

- 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
  - `lw r4, (r6)`
  - `jr r31`
MIPS R3000

- RISC architecture – 5 stage pipeline

![MIPS 5-stage pipeline](image)

- 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_epcs
    - Address of the instruction that caused the exception
  - c0_badvaddr
    - Address accessed that caused the exception
- **Memory Management**
  - c0_index
  - c0_random
  - c0_entryhi
  - c0_entrylo
  - c0_context
- **Miscellaneous**
  - c0_prid

**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>
</tr>
</thead>
<tbody>
<tr>
<td><strong>C11</strong></td>
<td>C12</td>
<td>C13</td>
<td>C14</td>
<td>C15</td>
<td>C16</td>
<td>C17</td>
<td>C18</td>
<td>C19</td>
<td>C20</td>
<td>C21</td>
<td>C22</td>
<td>C23</td>
<td>C24</td>
<td>C25</td>
<td>C26</td>
</tr>
<tr>
<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>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</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>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>IM</strong></td>
<td><strong>KU</strong></td>
<td><strong>IE</strong></td>
<td><strong>ExcCode</strong></td>
<td><strong>BD</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<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>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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

**c0_cause**

<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>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BD</strong></td>
<td><strong>CE</strong></td>
<td><strong>IP</strong></td>
<td><strong>ExcCode</strong></td>
<td><strong>BP</strong></td>
<td><strong>BP</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<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>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- IP
  - Intermits 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>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>Mod</td>
<td>&quot;TLB modification&quot;</td>
</tr>
<tr>
<td>2</td>
<td>TLBL</td>
<td>&quot;TLB load/TLB store&quot;</td>
</tr>
<tr>
<td>3</td>
<td>TSB</td>
<td>Address error (on load/1-fetch or store respectively). Either an attempt to access outside kseg 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>ABE</td>
<td>Address error on load/1-fetch or store respectively.</td>
</tr>
<tr>
<td>5</td>
<td>AUS</td>
<td>Address error on load/1-fetch or store respectively.</td>
</tr>
</tbody>
</table>

Table 3.2. Exception codes: different kinds of exceptions

Exception Vectors

<table>
<thead>
<tr>
<th>Program address</th>
<th>&quot;segment&quot;</th>
<th>Physical address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000 0000</td>
<td>0x0000</td>
<td>0x0000</td>
<td>TLB miss on kseg reference only.</td>
</tr>
<tr>
<td>0x0000 0080</td>
<td>0x0000</td>
<td>0x0080</td>
<td>All other exceptions.</td>
</tr>
<tr>
<td>0x07e0 0100</td>
<td>0x07e0</td>
<td>0x0100</td>
<td>Uncached alternative to seg TLB miss entry point (used if 32-bit ELI set).</td>
</tr>
<tr>
<td>0x07e0 0180</td>
<td>0x07e0</td>
<td>0x0180</td>
<td>Uncached alternative for all other exceptions, used if 32-bit ELI set.</td>
</tr>
<tr>
<td>0x07e0 0000</td>
<td>0x07e0</td>
<td>0x0000</td>
<td>The &quot;reset exception.&quot;</td>
</tr>
</tbody>
</table>

Table 4.1. Reset and exception entry points (vectors) for R3000x family

Simple Interrupt Walk-through

User Mode

Kernel Mode

Interrupt Handler

Hardware exception handling

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

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

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

- 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

PC: 0x80001234
EPC: 0x12345678

- 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.

Returning from an exception

PC: 0x12345678
EPC: 0x12345678

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

Store the EPC back in the PC.

Returning from an exception

PC: 0x12345678
EPC: 0x12345678

- We are now back in the same state we were in when the exception happened.
  
  ```
  Cause 0
  Status KU IE IE IU IE IC IE
  ```

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().

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().
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().

Software Register Conventions

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

Software 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 return 0</td>
</tr>
<tr>
<td>1</td>
<td>at</td>
<td>assembler temporary reserved for use by assembler</td>
</tr>
<tr>
<td>2-3</td>
<td>s0-s1</td>
<td>Value from interrupt PSW returned by subroutine</td>
</tr>
<tr>
<td>4-7</td>
<td>a0-a3</td>
<td>Argument first four parameters for a subroutine</td>
</tr>
<tr>
<td>8-15</td>
<td>a4-a17</td>
<td>Temporary subroutines may use without saving</td>
</tr>
<tr>
<td>16-25</td>
<td>s2-s31</td>
<td>Subroutine &quot;register variables&quot;, a subroutine which will write one of these must save the old value and restore it before it exits, to the calling routine using their values preserved</td>
</tr>
<tr>
<td>26-27</td>
<td>hi-lo</td>
<td>Reserved for use by interrupt or trap handler - may change during run</td>
</tr>
<tr>
<td>28</td>
<td>gp</td>
<td>Global pointer - some machine systems maintain this to give easy access to &quot;static&quot; or &quot;external&quot; variables.</td>
</tr>
<tr>
<td>29</td>
<td>sp</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>30</td>
<td>a3</td>
<td>Old register variable. Subroutines which need one can use this as a &quot;frame pointer&quot;.</td>
</tr>
<tr>
<td>32</td>
<td>ra</td>
<td>Return address for subroutine</td>
</tr>
</tbody>
</table>

Stack Frame

- MIPS calling convention for gcc
  - Args 1-4 have space reserved for them
  - Local variables
  - ... dynamic area

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;
    }
}
```

Stack Frame

```
0040011c <main>:
4000011c: 27bdff8    addiu sp,sp,-40
40000120: afbf0024    sw ra,36(s8)
40000124: afbe0020    sw s8,32(sp)
40000128: 03a0f021    move s8,sp
4000012c: 24020005    li v0,5
40000130: afa20010    sw v0,16(sp)
40000134: 24020006    li v0,6
40000138: af20014    sw v0,20(sp)
4000013c: 24040001    li a0,1
40000140: 24050002    li a1,2
40000144: 24060003    li a2,3
40000148: 0132000c    jal 4000b0 <sixargs>
4000014c: 24070004    li a3,4
40000150: af20018    sw v0,24(s8)
40000154: 03be021    move sp,a8
40000158: ebf0024    lw ra,36(s8)
4000015c: af2e020    lw s8,32(sp)
40000160: 03e00008    jr ra
40000164: 27bf00b    addiu sp,sp,40
...```

040011c <main>:
```
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.
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

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 System 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

Steps in Making a System Call

There are 11 steps in making the system call
read (fd, buffer, nbytes)
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: 02060201 move a0,s3
400128: 27a50010 addiu a1,sp,16
40012c: 0c1001a3 jal 40068c <read>
400130: 24060400 li a2,1024
400134: 00400201 move s0,v0
400138: la000016 blz 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: 03a00008 jr ra
400660: 00000000 nop

The read() syscall function part 2

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: 03a00008 jr ra
400660: 00000000 nop

The read() syscall function part 2

If failure, store code in errno

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: 03a00008 jr ra
400660: 00000000 nop

The read() syscall function part 2

Set read() result to -1

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: 03a00008 jr ra
400660: 00000000 nop
The read() syscall function part 2

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>00400640</td>
<td><code>_syscall</code></td>
</tr>
<tr>
<td>0040640</td>
<td>0000000c  syscall</td>
</tr>
<tr>
<td>0040644</td>
<td>10e00005  beqz a3,40065c</td>
</tr>
<tr>
<td>0040648</td>
<td>00000000  nop</td>
</tr>
<tr>
<td>004064c</td>
<td>3c011000  lui at,0x01000</td>
</tr>
<tr>
<td>0040650</td>
<td>ac120000  sw v0,0(at)</td>
</tr>
<tr>
<td>0040654</td>
<td>2403ffff  li v1,-1</td>
</tr>
<tr>
<td>0040658</td>
<td>2402ffff  li v0,-1</td>
</tr>
<tr>
<td>004065c</td>
<td>03e00008  jr ra</td>
</tr>
<tr>
<td>0040660</td>
<td>00000000  nop</td>
</tr>
</tbody>
</table>

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, c_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 */
   /* delay slot */
/* Coming from user mode - load kernel stack into sp */
la k0, curkstack /* get address of "curkstack" */
sw sp, 0(k0) /* Bus */
   /* delay slot for the load */
1:
sf0 k0, c_cause /* Now, load the exception cause. */
} common_exception /* Skip to common code */
   /* 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
```
These six stores are a "hack" to avoid confusing GDB. You can ignore the details of why and how.

The real work starts here.

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.

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

Save all the registers on the kernel stack.

The kernel stack by creating a pointer to the base of the saved registers and state in the first argument register.

The real work starts here.

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.

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

Save all the registers on the kernel 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;
  exception_return:
  /* 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, 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)
  /* "saved" k0 was dummy garbage anyway */
  /* "saved" k1 was dummy garbage anyway */
  /* fetch exception return PC into k0 */
  lw sp, 140(sp)
  lw k0, 140(sp)
  /* fetch saved sp (must be last) */
  /* done */
  jr k0
  /* jump back */
  rfe  /* in delay slot */
.end common_exception

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

Note again that only
\textcolor{red}{k0, k1 have been trashed}