System Calls

Interface and Implementation

Learning Outcomes

• A high-level understanding of System Call interface
  – 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.
• 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

The Structure of a Computer System

System Calls

Interface

The System Call Interface: A Brief Overview

• 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.

The System Call Interface: A Brief Overview

• 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>pid = getpid()</td>
<td>Create a child process identical to the parent</td>
</tr>
<tr>
<td>pid = wait3(0, &amp;status, &amp;argpoint)</td>
<td>Wait for a child to terminate</td>
</tr>
<tr>
<td>fork()</td>
<td>Create a new process</td>
</tr>
<tr>
<td>waitpid(pid, &amp;status, &amp;options)</td>
<td>Wait for a process to exit</td>
</tr>
<tr>
<td>write(fd, buffer, nbyte)</td>
<td>Write data to a file</td>
</tr>
<tr>
<td>read(fd, buffer, nbyte)</td>
<td>Read data from a file</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(filename, mode)</td>
<td>Open a file for reading, writing or both</td>
</tr>
<tr>
<td>close(fd)</td>
<td>Close an open file</td>
</tr>
<tr>
<td>lseek(fd, offset, whence)</td>
<td>Move the file pointer</td>
</tr>
<tr>
<td>readlink(name, buffer, size)</td>
<td>Get a link's alias 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>mkdir(name, mode)</td>
<td>Create a new directory</td>
</tr>
<tr>
<td>rmdir(name)</td>
<td>Remove an empty directory</td>
</tr>
<tr>
<td>rename(name1, name2)</td>
<td>Rename a file or directory</td>
</tr>
<tr>
<td>unlink(name)</td>
<td>Remove a directory entry</td>
</tr>
<tr>
<td>mount(name, name, flag)</td>
<td>Mount a file system</td>
</tr>
<tr>
<td>umount(name)</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>execve(command, parameters, 0)</td>
<td>Execute a command</td>
</tr>
</tbody>
</table>

System Calls

- A stripped down shell:

```c
while (TRUE) { /* repeat forever */
    /* some code */

    if (fork() == 0) {
        /* Child code */
        /* Some code */
        execve(command, parameters, 0);
    } else { /* Parent code */
        /* Some code */
        waitpid(-1, &status, 0); /* wait for child to exit */
    }
}
```

Some Win32 API calls

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateFile</td>
<td>Opens a new file</td>
</tr>
<tr>
<td>write</td>
<td>Writes to a file</td>
</tr>
<tr>
<td>lseek</td>
<td>Move the file pointer</td>
</tr>
<tr>
<td>read</td>
<td>Reads from a file</td>
</tr>
<tr>
<td>close</td>
<td>Closes a file</td>
</tr>
</tbody>
</table>

Some Win32 API calls
System Call Implementation
Crossing user-kernel boundary

A Simple Model of CPU Computation

- The fetch-execute cycle
  - Load memory contents from address in program counter (PC)
    - The instruction
    - Execute the instruction
    - Increment PC
    - Repeat

CPU Registers

- Stack Pointer
- Status Register
  - Condition codes
  - Positive result
  - Zero result
  - Negative result
- General Purpose Registers
  - Holds operands of most instructions
  - Enables programmers (compiler) to minimise memory references.

Example Unsafe Instruction

- “cli” instruction on x86 architecture
  - Disables interrupts
- Example exploit
  cli /* disable interrupts */
  while (true)
  /* loop forever */;

Privileged-mode Operation

To protect operating system execution, two or more CPU modes of operation exist

- Privileged mode (system-, kernel-mode)
  - All instructions and registers are available
- User-mode
  - Uses ‘safe’ subset of the instruction set
    - Only affects the state of the application itself
    - They cannot be used to uncontrollably interfere with OS
  - Only ‘safe’ registers are accessible

Example Unsafe Instruction

- “cli” instruction on x86 architecture
  - Disables interrupts
- Example exploit
  cli /* disable interrupts */
  while (true)
  /* loop forever */;

Privileged-mode Operation

Memory Address Space

The accessibility of addresses within an address space changes depending on operating mode

- To protect kernel code and data
- Note: The exact memory ranges are usually configurable, and vary between CPU architectures and/or operating systems.
System Call

User Mode

Application

Kernel Mode

System call mechanism securely transfers from user execution to kernel execution and back.

System Call Mechanism Overview

- System call transitions triggered by special processor instructions
  - User to Kernel
    - System call instruction
  - Kernel to User
    - Return from privileged mode instruction

System Call Mechanism Overview

- 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

System Call Mechanism Overview

- 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

System Call Mechanism Overview

- 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
      - To prevent entering after any security checks

Questions we’ll answer

- There is only one register set
  - How is register use managed?
  - What does an application expect a system call to look like?
- How is the transition to kernel mode triggered?
- Where is the OS entry point (system call handler)?
- How does the OS know what to do?

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
      - To prevent entering after any security checks
Steps in Making a System Call

MIPS R3000

- Load/store architecture
  - No instructions that operate on memory except load
  - 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.

- 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` \(\Rightarrow r2 = r1\)

- 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` \(\Rightarrow r2 = 1234\)

The MIPS R2000/R3000

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

Example code

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

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

- User-mode accessible registers
  - 32 general purpose registers
    - r0 hardcoded 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

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

Compiler Register Conventions

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

Compiler Register Conventions Table

<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>2-5</td>
<td>s0-s3</td>
<td>Value (except TPS) returned by subroutine</td>
</tr>
<tr>
<td>4-7</td>
<td>a0-a3</td>
<td>Argument passed to parameters for a subroutine</td>
</tr>
<tr>
<td>8-15</td>
<td>t0-t7</td>
<td>Temporary; subroutines may use without saving</td>
</tr>
<tr>
<td>16-23</td>
<td>s8-s15</td>
<td>Subroutine &quot;register variables&quot;: a subroutine which will save register</td>
</tr>
<tr>
<td></td>
<td></td>
<td>variables and restore them on return</td>
</tr>
<tr>
<td>24-25</td>
<td>t8-t9</td>
<td>Reserved for use by interrupt &quot;trap handler&quot; – stack change and/or stack key</td>
</tr>
<tr>
<td>26</td>
<td>gp</td>
<td>Global pointer – some runtime systems maintain this to give easy access to global scope or extern variables.</td>
</tr>
<tr>
<td>27</td>
<td>sp</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>28</td>
<td>lhr</td>
<td>Link register variable – Subroutines which use one can use this as a &quot;frame pointer&quot;</td>
</tr>
<tr>
<td>29</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().

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); 
    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.

CP0 Registers

- Exception Management
  - c0_cause
    - Cause of the recent exception
  - c0_status
    - Current status of the CPU
  - c0_endpoint
    - Address of the instruction that caused the exception
  - c0_badvaddr
    - Address accessed that caused the exception
- Miscellaneous
  - c0_prid
    - Processor Identifier
- Memory Management
  - c0_index
  - c0_random
  - c0_entryhi
  - c0_entrylo
  - c0_context
  - More about these later in course

C0_status

<table>
<thead>
<tr>
<th>Bit</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</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>
</tr>
</tbody>
</table>

For practical purposes, you can ignore most bits
- Green background is the focus
c0_status

- 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 3.2. Fields in status register (c0_status)

<table>
<thead>
<tr>
<th>IM</th>
<th>KU</th>
<th>IE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Exception Codes

<table>
<thead>
<tr>
<th>Exec 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>“TLB modified”</td>
</tr>
<tr>
<td>2</td>
<td>ERE</td>
<td>“EER fault/TLB store”</td>
</tr>
<tr>
<td>3</td>
<td>ENS</td>
<td>ENS</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 house when in user mode, or an attempt to read a word or half-word at an nonaligned address.</td>
</tr>
<tr>
<td>5</td>
<td>ARES</td>
<td>Address error on load/1-fetch or store respectively. Either an attempt to access outside house when in user mode, or an attempt to read a word or half-word at an nonaligned address.</td>
</tr>
</tbody>
</table>

Table 3.2. Exec Code values: different kinds of exceptions

C0_cause

- IP
  - Interrupts pending
  - 8 bits indicating current state of interrupt lines

- CE
  - Coprocessor error
  - Attempt to access disabled Copro.

- ExcCode
  - The code number of the exception taken

- BD
  - If set, the instruction that caused the exception was in a branch delay slot

Figure 3.3. Fields in the Cause register

<table>
<thead>
<tr>
<th>BD</th>
<th>CE</th>
<th>IP</th>
<th>ExcCode</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Exception Vectors

<table>
<thead>
<tr>
<th>Physical Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000 0x0000</td>
<td>L1DNotOnKorgReferenceOnly</td>
</tr>
<tr>
<td>0x0000 0x0008</td>
<td>All other exceptions.</td>
</tr>
<tr>
<td>0x00c 0x000</td>
<td>Uncached alternative korg L1DNotOnKorgReferenceOnly.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Physical Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00c 0x0100</td>
<td>Uncached alternative korg L1DNotOnKorgReferenceOnly.</td>
</tr>
</tbody>
</table>

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

- The Exception Program Counter
  - Points to address of where to restart execution after handling the exception or interrupt
  - Example: Assume sw z3, (r4) causes a restartable fault exception

Aside: We are ignore BD-bit in C0_cause which is also used in reality on rare occasions.

C0_epc

nop
sw z3, (r4)
nop

<table>
<thead>
<tr>
<th>C0_cause</th>
<th>C0_status</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC: 0x0080</td>
<td>HI/LO</td>
</tr>
<tr>
<td></td>
<td>R1</td>
</tr>
<tr>
<td></td>
<td>Rn</td>
</tr>
</tbody>
</table>

51
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.

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

<table>
<thead>
<tr>
<th>PC</th>
<th>EPC</th>
<th>Cause</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x80000080</td>
<td>0x12345678</td>
<td>0</td>
<td>? ? 1 1 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>? ? ? ?</td>
</tr>
</tbody>
</table>

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`

- Load the contents of EPC which is usually moved earlier to somewhere in memory by the exception handler

Returning from an exception

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

- Store the EPC back in the PC

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

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

- We are now back in the same state we were in when the exception happened
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()

```c
int read(int filehandle, void *buffer, size_t size);
```

- Three arguments, one return value
- Code segment calling the `read` function
  - Args are loaded, return value is tested

```assembly
400124: 02602021 move a0, a3
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>
```

- 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

### 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

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

Generate a syscall exception

Test success, if yes, branch to return from function

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

OS/161 Exception Handling

- Note: The following code is from the uniprocessor variant of OS161 (v1.x).
  - Simpler, but broadly similar.

```assembly
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 */
  beq 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: /* Now, load the exception cause. */
  mfc0 k0, c0_cause /* Note k0, k1 registers available for kernel use */
  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

  sw ra, 140(sp) /* dummy for gdb */
  sw s8, 156(sp) /* save s8 */
  sw sp, 152(sp) /* dummy for gdb */
  sw gp, 144(sp) /* save gp */
  sw k1, 144(sp) /* dummy for gdb */
  sw k0, 140(sp) /* dummy for gdb */
  sw k1, 152(sp) /* real saved sp */
  nop /* delay slot for store */
  sw k0, 160(sp) /* real saved PC */
  mfc0 k0, c0_epc /* Copr.0 reg 13 == PC for */
  sw k1, 160(sp) /* real saved PC */
```

Note k0, k1 registers available for kernel use

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.

Now we arrive in the `C` kernel.

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

Kernel Stack

```
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_hi;
    u_int32_t tf_lo;
    u_int32_t tf_ra; /* Saved register 31 */
    u_int32_t tf_at; /* Saved register 1 (AT) */
    u_int32_t tf_v0; /* Saved register 2 (v0) */
    u_int32_t tf_v1; /* etc. */
    u_int32_t tf_a0;
    u_int32_t tf_a1;
    u_int32_t tf_a2;
    u_int32_t tf_a3;
    u_int32_t tf_t0;
    u_int32_t tf_t7;
    u_int32_t tf_s0;
    u_int32_t tf_s7;
    u_int32_t tf_t8;
    u_int32_t tf_t9;
    u_int32_t tf_k0; /* dummy */
    u_int32_t tf_k1; /* dummy */
    u_int32_t tf_gp;
    u_int32_t tf_sp; /* Saved register 30 */
    u_int32_t tf_s8;
    u_int32_t tf_epc; /* coprocessor 0 epc register */
    u_int32_t tf_s9;
    u_int32_t tf_s10;
    u_int32_t tf_s11;
    u_int32_t tf_s12;
    u_int32_t tf_s13;
    u_int32_t tf_s14;
    u_int32_t tf_s15;
    u_int32_t tf_s16;
    u_int32_t tf_s17;
    u_int32_t tf_s18;
    u_int32_t tf_s19;
    u_int32_t tf_s20;
    u_int32_t tf_s21;
    u_int32_t tf_s22;
    u_int32_t tf_s23;
    u_int32_t tf_s24;
    u_int32_t tf_s25;
    u_int32_t tf_s26;
    u_int32_t tf_s27;
    u_int32_t tf_s28;
    u_int32_t tf_s29;
    u_int32_t tf_s30; /* Saved register 30 */
};
```

Kernel Stack

```
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. */
    savepl = curspl;
    /* Right now, interrupts should be off. */
    int savepl = curspl;
    /* The trap frame has been set up. */
    mips_trap_call();
    /* General trap (exception) handling function for mips. */
    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. */
        savepl = curspl;
        /* Right now, interrupts should be off. */
    }
```
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

```assembly
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, 36(sp)
lw t0, 32(sp)
mtlo t1
mthi t0
/* load the general registers */
lw t8, 68(sp)
lw t7, 72(sp)
lw t6, 76(sp)
lw t5, 80(sp)
lw t4, 84(sp)
lw t3, 88(sp)
lw t2, 92(sp)
lw t1, 96(sp)
lw t0, 100(sp)
lw x1, 104(sp)
lw x2, 108(sp)
lw x3, 112(sp)
lw x4, 116(sp)
lw x5, 120(sp)
lw x6, 124(sp)
lw x7, 128(sp)
lw x8, 132(sp)
lw x9, 136(sp)
/*     140(sp) "saved" k0 was dummy garbage anyway */
/*     144(sp) "saved" k1 was dummy garbage anyway */
```

Note again that only k0, k1 have been trashed