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 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

- Can be viewed as special function calls
  - Provides for a controlled entry into the kernel
  - While in kernel, they perform a privileged operation
  - Returns to original caller with the result
- The system call interface represents the abstract machine provided by the operating system.

A Brief Overview of Classes

System Calls

- From the user’s perspective
  - Process Management
  - File I/O
  - Directories management
  - Some other selected Calls
  - There are many more
    - On Linux, see `man syscalls` for a list

Some System Calls For Process Management

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sys fork()</code></td>
<td>Create a child process identical to the parent</td>
</tr>
<tr>
<td><code>sys wait()</code></td>
<td>Wait for a child to terminate</td>
</tr>
<tr>
<td><code>sys exe(0)</code></td>
<td>Replace a process’ core image</td>
</tr>
<tr>
<td><code>sys exit(0)</code></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 = (specfile, how ...)</td>
<td>Open a file for reading, writing or both</td>
</tr>
<tr>
<td>s = (closed)</td>
<td>Close an open file</td>
</tr>
<tr>
<td>n = (readoff, buffer, bytes)</td>
<td>Read data from a file into a buffer</td>
</tr>
<tr>
<td>p = (writeoff, buffer, bytes)</td>
<td>Write data from a buffer into a file</td>
</tr>
<tr>
<td>position = (seekoff, origin, whence)</td>
<td>Move the file pointer</td>
</tr>
<tr>
<td>s = stat(name, buf)</td>
<td>Get a file’s status 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, mode)</td>
<td>Create a new directory</td>
</tr>
<tr>
<td>s = rmdir(name)</td>
<td>Remove an empty directory</td>
</tr>
<tr>
<td>s = ln(name1, name2)</td>
<td>Create a new entry, name1, pointing to name2</td>
</tr>
<tr>
<td>s = unlink(name)</td>
<td>Remove a directory entry</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

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s = chdir(name)</td>
<td>Change the working directory</td>
</tr>
<tr>
<td>s = chmod(name, mode)</td>
<td>Change a file’s protection bits</td>
</tr>
<tr>
<td>s = kill(pid, signal)</td>
<td>Send a signal to a process</td>
</tr>
<tr>
<td>seconds = time(&amp;seconds)</td>
<td>Get the elapsed time since Jan. 1, 1970</td>
</tr>
</tbody>
</table>

Some Win32 API calls

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) {
      /* Parent code */
      waitpid(-1, &status, 0); /* wait for child to exit */
    } else {
      /* Child code */
      execl(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

- 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,....

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

Example code

Simple code example: `a = a + 1`

```
lw r4,32(r29) // r29 = stack pointer
li r5, 1
add r4, r4, r1
sw r4,32(r29)
```

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

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

```
  0x10  jal  1f
  0x14  nop
  0x18  lw  r4, (r6)

  1:  0x2a  sw  r2, (r3)
  0x38  jr  r31
      nop
```

MIPS R3000

- RISC architecture – 5 stage pipeline

```
  Figure 1.1. MIPS 5-stage pipeline
```

Coprocessor 0

- The processor control registers are located in CP0
  - Exception/Interrupt management registers
  - Translation management registers
- CP0 is manipulated using `mtc0` (move to) and `mfc0` (move from) instructions
  - `mtc0/mfc0` are only accessible in kernel mode.

```
  CP0
  CP1 (floating point)
  PC: 0x0300
  H/L/0
  R1
  Rn
```

CP0 Registers

- **Exception Management**
  - `c0_cause`
    - Cause of the recent exception
  - `c0_status`
    - Current status of the CPU
  - `c0_eip`
    - Address of the instruction that caused the exception
  - `c0_baseaddr`
    - 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

  0  1  2  3  4  5  6  7
  0  1  2  3  4  5  6  7
  0  1  2  3  4  5  6  7
  0  1  2  3  4  5  6  7

  0  1  2  3  4  5  6  7
  0  1  2  3  4  5  6  7
  0  1  2  3  4  5  6  7
  0  1  2  3  4  5  6  7
```

For practical purposes, you can ignore most bits
- Green background is the focus
Figure 3.3: Fields in the Cause register

- **IP**
  - Interrupts 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

### Exception Codes

<table>
<thead>
<tr>
<th>ExcCode 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 miss</td>
</tr>
<tr>
<td>2</td>
<td>TLEN</td>
<td>&quot;TLB load/TLB store&quot;</td>
</tr>
<tr>
<td>3</td>
<td>TIBS</td>
<td>Address error (on load/1-fetch or store respectively)</td>
</tr>
<tr>
<td>4</td>
<td>MREL</td>
<td>Either an attempt to access outside kseg when in reset mode, or an attempt to load a word or half word at a misaligned address.</td>
</tr>
<tr>
<td>5</td>
<td>MRES</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2. ExcCode values: 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>0x8000 0000</td>
<td>kseg0</td>
<td>0x0000 0000</td>
<td>TLB miss on kseg reference only.</td>
</tr>
<tr>
<td>0x8000 0008</td>
<td>kseg0</td>
<td>0x0000 0008</td>
<td>All other exceptions.</td>
</tr>
<tr>
<td>0xd000 0200</td>
<td>kseg1</td>
<td>0x0100 0100</td>
<td>Unrelated alternative kseg TLB miss entry point (used if 16 bit BEV set).</td>
</tr>
<tr>
<td>0xd000 0200</td>
<td>kseg1</td>
<td>0x0100 0100</td>
<td>Unrelated alternative for all other exceptions, used if 32 bit BEV set.</td>
</tr>
<tr>
<td>0xd000 0000</td>
<td>kseg1</td>
<td>0x0100 0000</td>
<td>The &quot;reset exception&quot;.</td>
</tr>
</tbody>
</table>

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

### Simple Exception Walk-through

**User Mode**

- Application

**Kernel Mode**

- Interrupt Handler

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

### Cause and Status

<table>
<thead>
<tr>
<th>PC</th>
<th>EPC</th>
<th>Cause</th>
<th>Status</th>
</tr>
</thead>
</table>

### Interrupted Address

- PC: 0x12345678
- EPC: 0x12345678

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

- Code: 0
- Status: 0 ? 1 1 0 0

### 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
- Address of general exception vector placed in PC

- PC: 0x80000080
- EPC: 0x12345678

### Address of general exception vector placed in PC

- PC: 0x80000080
- EPC: 0x12345678

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

- PC: 0x12345678
- EPC: 0x12345678

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

- Code: 0
- Status: 0 ? 1 1 0 0

### Badvaddr

- 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
- Address of general exception vector placed in PC

- PC: 0x80000080
- EPC: 0x12345678
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

```
\text{This code to return is}
\text{\texttt{lw} } r27, \text{ saved_epc}
\text{\texttt{nop}}
\text{\texttt{jr} } r27
\text{\texttt{rfe}}
```

• This code to return

\text{\texttt{lw} } r27, \text{ saved_epc}
\text{\texttt{nop}}
\text{\texttt{jr} } r27
\text{\texttt{rfe}}

Store the EPC back in
the PC

```
0x80001234
0x2345678
```

\begin{tabular}{|c|c|}
\hline
\text{PC} & 0x12345678 \\
\hline
\text{EPC} & 0x12345678 \\
\hline
\text{Cause} & 0 \\
\hline
\text{Status} & 1 0 0 \\
\hline
\end{tabular}

Returning from an exception

• This code to return

```
\text{This code to return is}
\text{\texttt{lw} } r27, \text{ saved_epc}
\text{\texttt{nop}}
\text{\texttt{jr} } r27
\text{\texttt{rfe}}
```

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

\begin{tabular}{|c|c|}
\hline
\text{PC} & 0x12345678 \\
\hline
\text{EPC} & 0x12345678 \\
\hline
\text{Cause} & 0 \\
\hline
\text{Status} & 0 0 0 1 1 1 1 \\
\hline
\end{tabular}

Returning from an exception

• We are now back in the
same state we were in
when the exception
happened

\begin{tabular}{|c|c|}
\hline
\text{PC} & 0x12345678 \\
\hline
\text{EPC} & 0x12345678 \\
\hline
\text{Cause} & 0 \\
\hline
\text{Status} & 1 0 0 1 1 1 1 \\
\hline
\end{tabular}

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().

\begin{tabular}{|c|c|}
\hline
\text{Frame Pointer} & \text{f1() stack frame} \\
\hline
\text{Stack Pointer} & \text{Stack} \\
\hline
\end{tabular}
Function Stack Frames

- Each function call allocates a new stack frame for local variables, the return address, previous frame pointer etc.
  - Frame pointer: start of current stack frame
  - Stack pointer: end of current stack frame
- Example: assume f1() calls f2(), which calls f3().

Compiler Register Conventions

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

Stack Frame

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

Example Code

```
main ()
{
  int sixargs(int a, int b, int c, int d, int e, int f);

  int i;
  return a + b + c + d + e + f;
}
```
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
System Call Mechanism in Principle

• Registers
  – Set at user-level to indicate system call type and its arguments
    • A convention between applications and the kernel
    – Some registers are preserved at user-level or kernel-level in order to restart user-level execution
    • Depends on language calling convention etc.
  – 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

Convention for kernel entry
Preserved
Success?
SysCall No.

Convention for kernel exit
Preserved
Result
Args in
• Seriously low-level code follows
• This code is not for the faint hearted

inside 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

User-Level System Call Walk Through – Calling read()

int read(int filehandle, void *buffer, size_t size)
• Three arguments, one return value
• Code fragment calling the read function
  400124: 02602021 move a0,a3
  400128: 27a50010 addiu a1,sp,16
  40012c: 0c1001a3 jal 40068c <read>
  400130: 24060400 li a2,1024
  400134: 4006821 move a0,v0
  400138: 1a000016 blez a0,400194 <docat+0x94>
  • Args are loaded, return value is tested

inside the read() syscall function
part 2

The read() syscall function
part 2

Generate a syscall exception

Test success, if yes, branch to return from function

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

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
common_exception:

/*  
 * At this point:  
 *      Interrupts are off. (The processor did this for us.)  
 *      k0 contains the exception cause value.  
 *      k1 contains the old stack pointer.  
 *      sp points into the kernel stack.  
 *      All other registers are untouched.  
 */

/*  
 * Allocate stack space for 37 words to hold the trap frame,  
 * plus four more words for a minimal argument block.  
 */
addi sp, sp, -164

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

sw ra, 140(sp) /* dummy for gdb */
sw k0, 156(sp) /* save k0 */
sw sp, 152(sp) /* dummy for gdb */
sw sp, 148(sp) /* save sp */
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 */
mfc0 k1, c0_epc /* Copr.0 reg 13 == PC for exception */
sw k1, 160(sp) /* real saved PC */

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

The real work starts here

sw t9, 136(sp)
sw t8, 132(sp)
sw s7, 128(sp)
sw s6, 124(sp)
sw s5, 120(sp)
sw s4, 116(sp)
sw s3, 112(sp)
sw s2, 108(sp)
sw s1, 104(sp)
sw s0, 100(sp)
sw t7, 96(sp)
sw t6, 92(sp)
sw t5, 88(sp)
sw t4, 84(sp)
sw t3, 80(sp)
sw t2, 76(sp)
sw t1, 72(sp)
sw t0, 68(sp)
sw a3, 64(sp)
sw a2, 60(sp)
sw a1, 56(sp)
sw a0, 52(sp)
v1, 48(sp)
v0, 44(sp)
v AT, 40(sp)
ra, 36(sp)
mfc0 t1, c0_status /* Copr.0 reg 11 == status */
sw t2, 16(sp) /* real saved sp */
nop /* delay slot for store */

Save all the registers on the kernel stack

mfc0 t1, c0_status /* Copr.0 reg 11 == status */
sw k0, 24(sp) /* k0 was loaded with cause earlier */
mfc0 t1, k0.status /* Copr.0 reg 11 == status */
sw t1, 20(sp)
mfc0 t2, t0.vaddr /* Copr.0 reg 8 == faulting vaddr */
sw t2, 16(sp)

/* Pretend to save $0 for gdb's benefit. */
sw $0, 12(sp)

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

/* Prepare to call mips_trap(struct trapframe *) */
addiu a0, sp, 16 /* set argument */
jal mips_trap /* call it */
nop /* delay slot */

Create a pointer to the base of the saved registers and state in the first argument register.
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; /* Saved register 2 (v0) */
    u_int32_t tf_ra; /* Saved register 1 (AT) */
    u_int32_t tf_v0; /* Saved register 0 (v0) */
    u_int32_t tf_at; /* Saved register 1 (AT) */
    u_int32_t tf_s0; /* Saved register 2 (v0) */
    u_int32_t tf_s1; /* Saved register 3 (a1) */
    u_int32_t tf_t1; /* Saved register 4 (a2) */
    u_int32_t tf_s2; /* Saved register 5 (a3) */
    u_int32_t tf_t2; /* Saved register 6 (t0) */
    u_int32_t tf_s3; /* Saved register 7 (t1) */
    u_int32_t tf_t3; /* Saved register 8 (t2) */
    u_int32_t tf_s4; /* Saved register 9 (t3) */
    u_int32_t tf_t4; /* Saved register 10 (t4) */
    u_int32_t tf_s5; /* Saved register 11 (t5) */
    u_int32_t tf_t5; /* Saved register 12 (t6) */
    u_int32_t tf_s6; /* Saved register 13 (t7) */
    u_int32_t tf_t6; /* Saved register 14 (k0) */
    u_int32_t tf_s7; /* Saved register 15 (k1) */
    u_int32_t tf_t8; /* Saved register 16 (gp) */
    u_int32_t tf_t9; /* Saved register 17 (sp) */
    u_int32_t tf_k0; /* dummy */
    u_int32_t tf_k1; /* dummy */
    u_int32_t tf_gp; /* Saved register 18 (gp) */
    u_int32_t tf_sp; /* Saved register 19 (sp) */
    u_int32_t tf_s8; /* Saved register 20 (s8) */
    u_int32_t tf_epc; /* coprocessor 0 epc register */
    u_int32_t tf_nonce; /* dummy */
    u_int32_t tf_nxt; /* dummy */
    u_int32_t tf_dpc; /* dummy */
    u_int32_t tf_nxs; /* dummy */
    u_int32_t tf_dxs; /* dummy */
    u_int32_t tf_nxt; /* dummy */
};

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:
    /* 16(sp) no need to restore tf_vaddr */
    lw t0, 20(sp) /* load status register value into t0 */
    nop /* load delay slot */
    mtc0 t0, c0_status /* store it back to coprocessor 0 */
    /* 24(sp) no need to restore tf_cause */
    /* restore special registers */
    lw t1, 28(sp)
    lw t0, 32(sp)
    mthi t1
    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 t8, 100(sp)
    lw t9, 104(sp)
    lw s0, 108(sp)
    lw s1, 112(sp)
    lw s2, 116(sp)
    lw s3, 120(sp)
    lw s4, 124(sp)
    lw s5, 128(sp)
    lw s6, 132(sp)
    lw s7, 136(sp)
    /* 140(sp) "saved" k0 was dummy garbage anyway */
    /* 144(sp) "saved" k1 was dummy garbage anyway */
    lw gp, 148(sp) /* restore gp */
}

Note again that only k0, k1 have been trashed

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