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
System Calls

Interface
The Structure of a Computer System

Memory

The Structure of a Computer System

Interaction via
System Calls

Application
System Libraries

User Mode
Kernel Mode

Device
Device

OS

Memory
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.
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 = fork()</td>
<td>Create a child process identical to the parent</td>
</tr>
<tr>
<td>pid = waitpid(pid, &amp;statloc, options)</td>
<td>Wait for a child to terminate</td>
</tr>
<tr>
<td>s = execve(name, argv, environp)</td>
<td>Replace a process’ core image</td>
</tr>
<tr>
<td>exit(status)</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><code>fd = open(file, how, ...)</code></td>
<td>Open a file for reading, writing or both</td>
</tr>
<tr>
<td><code>s = close(fd)</code></td>
<td>Close an open file</td>
</tr>
<tr>
<td><code>n = read(fd, buffer, nbytes)</code></td>
<td>Read data from a file into a buffer</td>
</tr>
<tr>
<td><code>n = write(fd, buffer, nbytes)</code></td>
<td>Write data from a buffer into a file</td>
</tr>
<tr>
<td><code>position = lseek(fd, offset, whence)</code></td>
<td>Move the file pointer</td>
</tr>
<tr>
<td><code>s = stat(name, &amp;buf)</code></td>
<td>Get a file’s status information</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) {
        /* Parent code */
        waitpid( -1, &status, 0); /* wait for child to exit */
    } else {
        /* Child code */
        execve (command, parameters, 0); /* execute command */
    }
}
```
## System Calls

<table>
<thead>
<tr>
<th>UNIX</th>
<th>Win32</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fork</td>
<td>CreateProcess</td>
<td>Create a new process</td>
</tr>
<tr>
<td>waitpid</td>
<td>WaitForSingleObject</td>
<td>Can wait for a process to exit</td>
</tr>
<tr>
<td>execve</td>
<td>(none)</td>
<td>CreateProcess = fork + execve</td>
</tr>
<tr>
<td>exit</td>
<td>ExitProcess</td>
<td>Terminate execution</td>
</tr>
<tr>
<td>open</td>
<td>CreateFile</td>
<td>Create a file or open an existing file</td>
</tr>
<tr>
<td>close</td>
<td>CloseHandle</td>
<td>Close a file</td>
</tr>
<tr>
<td>read</td>
<td>ReadFile</td>
<td>Read data from a file</td>
</tr>
<tr>
<td>write</td>
<td>WriteFile</td>
<td>Write data to a file</td>
</tr>
<tr>
<td>lseek</td>
<td>SetFilePointer</td>
<td>Move the file pointer</td>
</tr>
<tr>
<td>stat</td>
<td>GetFileAttributesEx</td>
<td>Get various file attributes</td>
</tr>
<tr>
<td>mkdir</td>
<td>CreateDirectory</td>
<td>Create a new directory</td>
</tr>
<tr>
<td>rmdir</td>
<td>RemoveDirectory</td>
<td>Remove an empty directory</td>
</tr>
<tr>
<td>link</td>
<td>(none)</td>
<td>Win32 does not support links</td>
</tr>
<tr>
<td>unlink</td>
<td>DeleteFile</td>
<td>Destroy an existing file</td>
</tr>
<tr>
<td>mount</td>
<td>(none)</td>
<td>Win32 does not support mount</td>
</tr>
<tr>
<td>umount</td>
<td>(none)</td>
<td>Win32 does not support umount</td>
</tr>
<tr>
<td>chdir</td>
<td>SetCurrentDirectory</td>
<td>Change the current working directory</td>
</tr>
<tr>
<td>chmod</td>
<td>(none)</td>
<td>Win32 does not support security (although NT does)</td>
</tr>
<tr>
<td>kill</td>
<td>(none)</td>
<td>Win32 does not support signals</td>
</tr>
<tr>
<td>time</td>
<td>GetLocalTime</td>
<td>Get the current time</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

PC: 0x0300
A Simple Model of CPU Computation

• Stack Pointer (SP)
• 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.

CPU Registers

| PC: 0x0300 |
| SP: 0xcbf3 |
| Status     |
| R1         |
| ↓          |
| Rn         |
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

CPU Registers

<table>
<thead>
<tr>
<th>Interrupt Mask</th>
<th>Exception Type</th>
<th>MMU regs</th>
<th>Others</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PC: 0x0300</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SP: 0xcbf3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Status</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>R1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>↓</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Rn</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example Unsafe Instruction

• “cli” instruction on x86 architecture
  • Disables interrupts

• Example exploit
  
  cli /* disable interrupts */
  while (true)
  /* loop forever */;
Privileged-mode Operation

- 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.

<table>
<thead>
<tr>
<th>Memory Address Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessible to User- and Kernel-mode</td>
</tr>
<tr>
<td>Accessible only to Kernel-mode</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000 to 0x80000000</td>
</tr>
<tr>
<td>0x80000000 to 0xFFFFFFF</td>
</tr>
<tr>
<td>0xFFFFFFF to 0xFFFFFFFF</td>
</tr>
</tbody>
</table>
System Call

User Mode

Kernel Mode

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

• Stack Pointer (SP)
  • User-level SP is saved and a kernel SP is initialised
    • User-level SP restored when returning to user mode

• Program Counter (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
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

There are 11 steps in making the system call read (fd, buffer, nbytes)
The MIPS R2000/R3000

- Before looking at system call mechanics in some detail, we need a basic understanding of the MIPS R3000.
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_epc
    • 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
• For practical purposes, you can ignore most bits
  • Green background is the focus
## 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>CU3</td>
<td>CU2</td>
<td>CU1</td>
<td>CU0</td>
<td>0</td>
<td>RE</td>
<td>0</td>
<td>BEV</td>
<td>TS</td>
<td>PE</td>
<td>CM</td>
<td>PZ</td>
<td>SwC</td>
<td>IsC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Figure 3.2. Fields in status register (SR)

- **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
### c0_cause

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>BD</td>
<td>0</td>
<td>CE</td>
<td>0</td>
<td>IP</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>ExcCode</strong></td>
</tr>
</tbody>
</table>

*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
- **ExcCode**
  - The code number of the exception taken
## 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 modification”</td>
</tr>
<tr>
<td>2</td>
<td>TLBL</td>
<td>“TLB load/TLB store”</td>
</tr>
<tr>
<td>3</td>
<td>TLBS</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>AdEL</td>
<td>Address error (on load/I-fetch or store respectively). Either an attempt to access outside kuseg when in user mode, or an attempt to read a word or half-word at a misaligned address.</td>
</tr>
<tr>
<td>5</td>
<td>AdES</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2. ExcCode values: different kinds of exceptions
### Exception Codes

<table>
<thead>
<tr>
<th>ExcCode Value</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>IBE</td>
<td>Bus error (instruction fetch or data load, respectively). External hardware has signalled an error of some kind; proper exception handling is system-dependent. The R30xx family CPUs can’t take a bus error on a store; the write buffer would make such an exception “imprecise”.</td>
</tr>
<tr>
<td>7</td>
<td>DBE</td>
<td>Generated unconditionally by a syscall instruction.</td>
</tr>
<tr>
<td>8</td>
<td>Syscall</td>
<td>Generated unconditionally by a syscall instruction.</td>
</tr>
<tr>
<td>9</td>
<td>Bp</td>
<td>Breakpoint - a break instruction.</td>
</tr>
<tr>
<td>10</td>
<td>RI</td>
<td>“reserved instruction”</td>
</tr>
<tr>
<td>11</td>
<td>CpU</td>
<td>“Co-Processor unusable”</td>
</tr>
<tr>
<td>12</td>
<td>Ov</td>
<td>“arithmetic overflow”. Note that “unsigned” versions of instructions (e.g. addu) never cause this exception.</td>
</tr>
<tr>
<td>13-31</td>
<td>-</td>
<td>reserved. Some are already defined for MIPS CPUs such as the R6000 and R4xxx</td>
</tr>
</tbody>
</table>

Table 3.2. ExcCode values: different kinds of exceptions
c0_epc

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

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

<table>
<thead>
<tr>
<th>Program address</th>
<th>“segment”</th>
<th>Physical Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8000 0000</td>
<td>kseg0</td>
<td>0x0000 0000</td>
<td>TLB miss on kseg reference only.</td>
</tr>
<tr>
<td><strong>0x8000 0080</strong></td>
<td>kseg0</td>
<td><strong>0x0000 0080</strong></td>
<td>All other exceptions.</td>
</tr>
<tr>
<td>0xbfc0 0100</td>
<td>kseg1</td>
<td>0x1fc0 0100</td>
<td>Uncached alternative kseg TLB miss entry point (used if SR bit BEV set).</td>
</tr>
<tr>
<td>0xbfc0 0180</td>
<td>kseg1</td>
<td>0x1fc0 0180</td>
<td>Uncached alternative for all other exceptions, used if SR bit BEV set.</td>
</tr>
<tr>
<td>0xbfc0 0000</td>
<td>kseg1</td>
<td>0x1fc0 0000</td>
<td>The “reset exception”.</td>
</tr>
</tbody>
</table>

Table 4.1. Reset and exception entry points (vectors) for R30xx family
Simple Exception Walk-through

User Mode

Kernel Mode

Application

Interrupt

Return from Int

Interrupt Handler
Let’s now walk through an exception.

- Assume an interrupt occurred as the previous instruction completed.
- Note: We are in user mode with interrupts enabled.

PC: 0x12345678

EPC: ?

Cause: ?

Status: KUo IEo KUp IEp KUc IEc

? ? ? ? 1 1
Hardware exception handling

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

PC: 0x12345678

EPC: 0x12345678

Cause

Status: KUo IEo KUp IEp KUc IEc

Values: ? ? ? ? 1 1
Hardware exception handling

PC

0x12345678

Interrupts disabled and previous state shifted along

Status

Kernel Mode is set, and previous mode shifted along

KUo IEo KUp IEp KUc IEc

? ? 1 1 0 0
Hardware exception handling

- **PC**: 0x12345678
- **EPC**: 0x12345678

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

**Status**
- KUo lEo KUp lEp KUc lEc
- ? ? 1 1 0 0
Hardware exception handling

PC: 0x80000080

EPC: 0x12345678

Cause: 0

Status: KUo IEo KUp IEp KUc IEc

Address of general exception vector placed in PC: EPC
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>
</tbody>
</table>

- Cause: 0
- Status: KUo IEo KUp IEp KUc IEc
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

\[
\begin{align*}
\text{lw} & \ r27, \ \text{saved}_{-}\text{epc} \\
\text{nop} & \\
\text{jr} & \ r27 \\
\text{rfe} &
\end{align*}
\]

Store the EPC back in the PC
Returning from an exception

PC

0x12345678

• This code to return is

lw  r27, saved_epc
nop
jr  r27
rfe

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

KU0  IE0  KUP  IEP  KUC  IEC

Status

0  1  1

PC  EPC
Returning from an exception

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

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

<table>
<thead>
<tr>
<th>Cause</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>? ? ? ? 1 1</td>
</tr>
</tbody>
</table>

KUo IEo KUp IEp KUc IEc
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
• Seriously low-level code follows
• This code is not for the faint hearted

```
move    a0,s3
addiu   a1,sp,16
jal    $40068c <read>
li     a2,1024
move    s0,v0
blez    s0,$0x400194 <docat+0x94>
```
User-Level System Call Walk Through – Calling read()

```c
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: 00408021 move s0,v0
  400138: 1a000016 blez s0,400194
  <docat+0x94>
  ```

• Args are loaded, return value is tested
Inside the read() syscall function
part 1
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

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>00400640</td>
<td>0000000c</td>
<td>syscall</td>
</tr>
<tr>
<td>00400644</td>
<td>10e00005</td>
<td>beqz a3,40065c</td>
</tr>
<tr>
<td>00400648</td>
<td>00000000</td>
<td>nop</td>
</tr>
<tr>
<td>0040064c</td>
<td>3c011000</td>
<td>lui at,0x1000</td>
</tr>
<tr>
<td>00400650</td>
<td>ac220000</td>
<td>sw v0,0(at)</td>
</tr>
<tr>
<td>00400654</td>
<td>2403ffff</td>
<td>li v1,-1</td>
</tr>
<tr>
<td>00400658</td>
<td>2402ffff</td>
<td>li v0,-1</td>
</tr>
<tr>
<td>0040065c</td>
<td>03e00008</td>
<td>jr ra</td>
</tr>
<tr>
<td>00400660</td>
<td>00000000</td>
<td>nop</td>
</tr>
</tbody>
</table>

Generate a syscall exception
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:  03e00008  jr  ra 
  400660:  00000000  nop 

Test success, if yes, branch to return from function
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,0x100  
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>:
  0000000c      syscall
  10e00005      beqz a3,40065c <__syscall+0x1c>
  00000000      nop
  3c011000      lui   at,0x1000
  ac220000      sw    v0,0(at)
  2403ffff      li    v1,-1
  2402ffff      li    v0,-1
  03e00008      jr     ra
  00000000      nop

Set read() result to -1
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:  03e00008     jr ra
400660:  00000000     nop

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 to current version.
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, we must the stack to sp */
  la k0, curkstack /* load the address of "curkstack" */
  lw sp, 0(k0)
  nop /* delay slot for the load */

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

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

/*@*/

/*@*/

* Allocate stack space for 37 words to hold the trap frame,
 * plus four more words for a minimal argument block.
 */
addi sp, sp, -164
/* The 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)  /* 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 */
mfc0 k1, c0_epc /* Copr.0 reg 13 == PC for
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 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) /* 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 */
mfc0 k1, c0_epc /* Copr.0 reg 13 == PC for exception */
sw k1, 160(sp) /* real saved PC */
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)
sw v1, 48(sp)
sw v0, 44(sp)
sw AT, 40(sp)
sw ra, 36(sp)

Save all the registers on the kernel stack
/*
 * Save special registers.
 */
mfhi t0
mflo t1
sw t0, 32(sp)
sw t1, 28(sp)

/*
 * Save remaining exception context information.
 */
sw k0, 24(sp)               /* k0 was loaded with cause earlier */
mfc0 t1, c0_status          /* Copr.0 reg 11 == status */
sw t1, 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)
/*
* 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
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 */
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)
mtlo t1
mthi 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 s8 */
lw k0, 160(sp)        /* fetch exception return PC into k0 */

lw sp, 152(sp)        /* fetch saved sp (must be last) */

/* done */
jr k0                  /* jump back */
rfe                   /* in delay slot */
.end common_exception

Note again that only k0, k1 have been trashed