

System Calls

Interface and Implementation

1

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

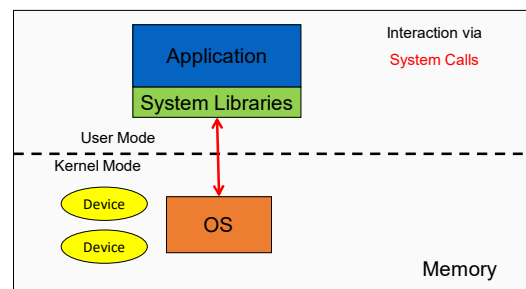
2

System Calls

Interface

3

The Structure of a Computer System



4

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.

5

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

6

Some System Calls For Process Management

Process management	
Call	Description
pid = fork()	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

7 UNSW

7

Some System Calls For File Management

File management	
Call	Description
fd = open(file, how, ...)	Open a file for reading, writing or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

8 UNSW

8

System Calls

• A stripped down shell:

```

while (TRUE) {
    type_prompt( );
    read_command (command, parameters)

    if (fork() != 0) {
        /* Parent code */
        waitpid(-1, &status, 0);
    } else {
        /* Child code */
        execve (command, parameters, 0);
    }
}
    
```

9 UNSW

9

System Calls

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

Some Win32 API calls

10 UNSW

10

System Call Implementation

Crossing user-kernel boundary

11 UNSW

11

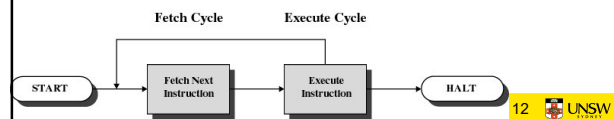
A Simple Model of CPU Computation

• 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



12 UNSW

12

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

13

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

Interrupt Mask
Exception Type
MMU regs
Others
PC: 0x0300
SP: 0xcbf3
Status
R1
↑
Rn

14

Example Unsafe Instruction

- "cli" instruction on x86 architecture
 - Disables interrupts
- Example exploit

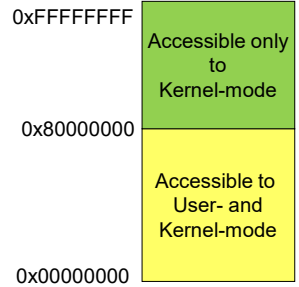

```
cli /* disable interrupts */
while (true)
    /* loop forever */;
```

15

Privileged-mode Operation

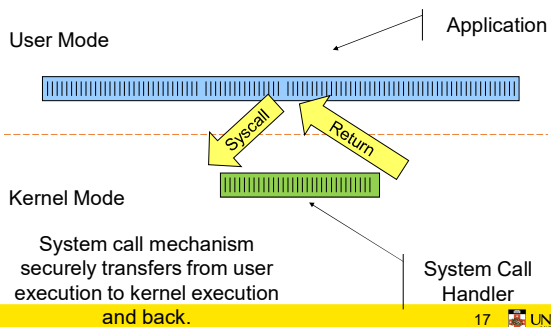
- The accessibility of addresses within an address space changes depending on operating mode
 - To protect kernel code and data
- Note: The exact memory ranges are usually configurable, and vary between CPU architectures and/or operating systems.

Memory Address Space



16

System Call



17

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?

18

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

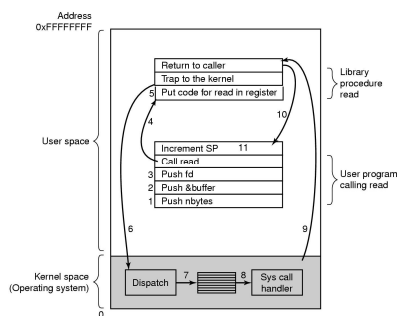
System Call Mechanism Overview

- 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

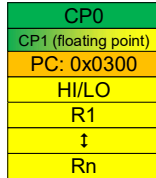


The MIPS R2000/R3000

- 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

c0_status

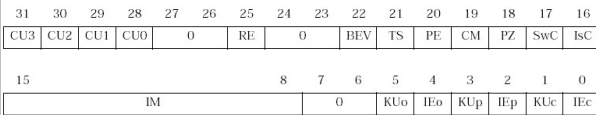


Figure 3.2. Fields in status register (SR)

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

c0_status

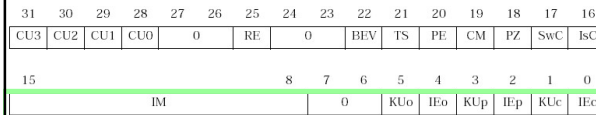


Figure 3.2. Fields in status register (SR)

- 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

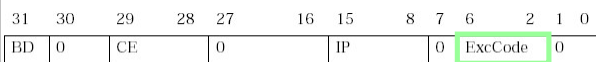


Figure 3.3. Fields in the Cause register

- 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

ExcCode Value	Mnemonic	Description
0	Int	Interrupt
1	Mod	"TLB modification"
2	TLBL	"TLB load/TLB store"
3	TLBS	
4	AdEL	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.
5	AdES	

Table 3.2. ExcCode values: different kinds of exceptions

Exception Codes

ExcCode Value	Mnemonic	Description
6	IBE	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".
7	DBE	
8	Syscall	Generated unconditionally by a <code>syscall</code> instruction.
9	Bp	Breakpoint - a <code>break</code> instruction.
10	RI	"reserved instruction"
11	CpU	"Co-Processor unusable"
12	Ov	"arithmetic overflow". Note that "unsigned" versions of instructions (e.g. <code>addu</code>) never cause this exception.
13-31	-	reserved. Some are already defined for MIPS CPUs such as the R6000 and R4xxx

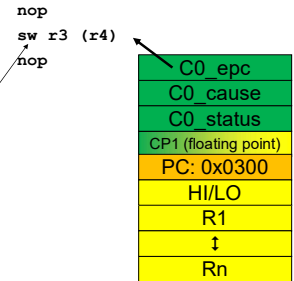
Table 3.2. ExcCode values: different kinds of exceptions

c0_etc

The Exception Program Counter

- Points to address of where to restart execution after handling the exception or interrupt
- Example
 - Assume `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.

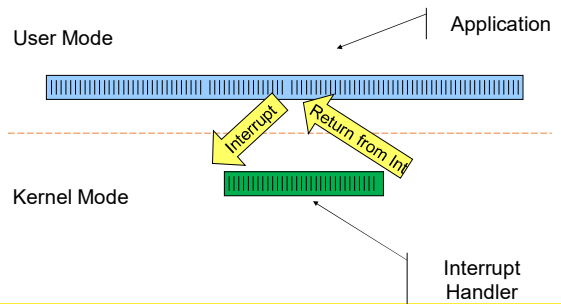


Exception Vectors

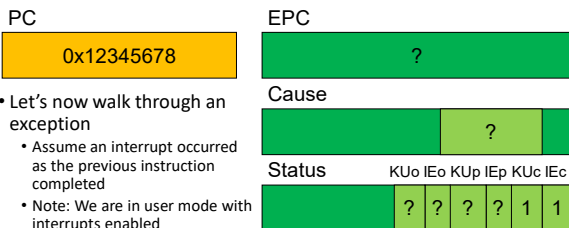
Program address	"segment"	Physical Address	Description
0x8000 0000	kseg0	0x0000 0000	TLB miss on <code>kuseg</code> reference only.
0x8000 0080	kseg0	0x0000 0080	All other exceptions.
0xbfc0 0100	kseg1	0x1fc0 0100	Uncached alternative <code>kuseg</code> TLB miss entry point (used if <code>SR</code> bit BEV set).
0xbfc0 0180	kseg1	0x1fc0 0180	Uncached alternative for all other exceptions, used if <code>SR</code> bit BEV set).
0xbfc0 0000	kseg1	0x1fc0 0000	The "reset exception".

Table 4.1. Reset and exception entry points (vectors) for R30xx family

Simple Exception Walk-through

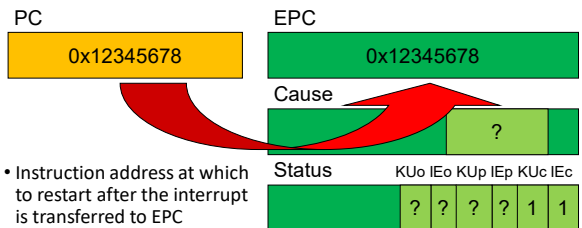


Hardware exception handling

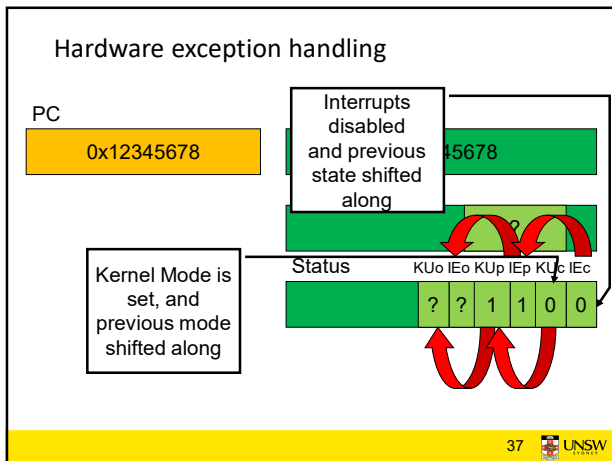


- Let's now walk through an exception
 - Assume an interrupt occurred as the previous instruction completed
 - Note: We are in user mode with interrupts enabled

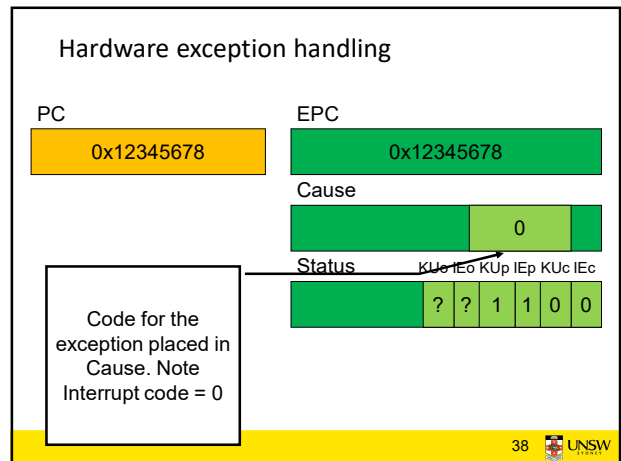
Hardware exception handling



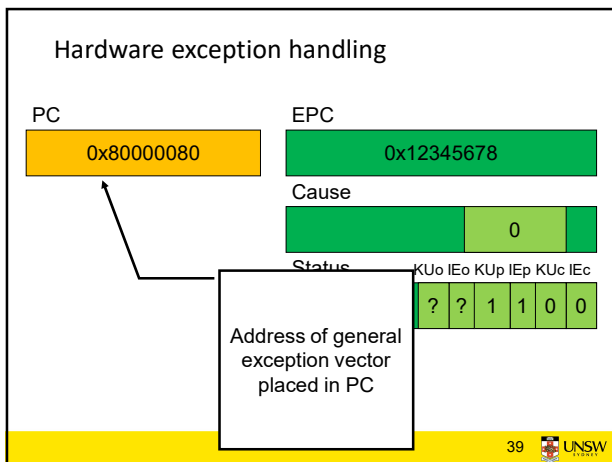
- Instruction address at which to restart after the interrupt is transferred to EPC



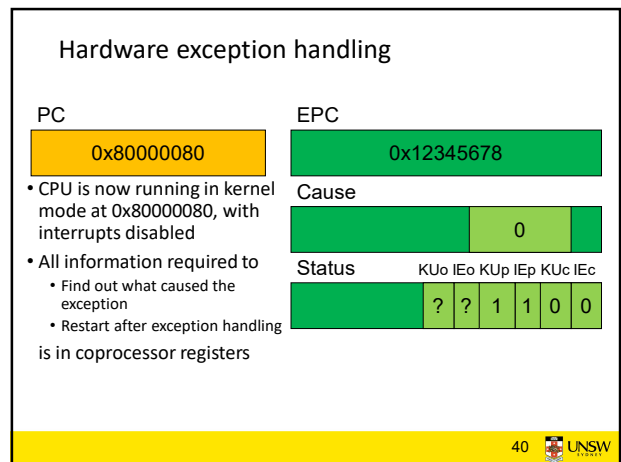
37



38



39



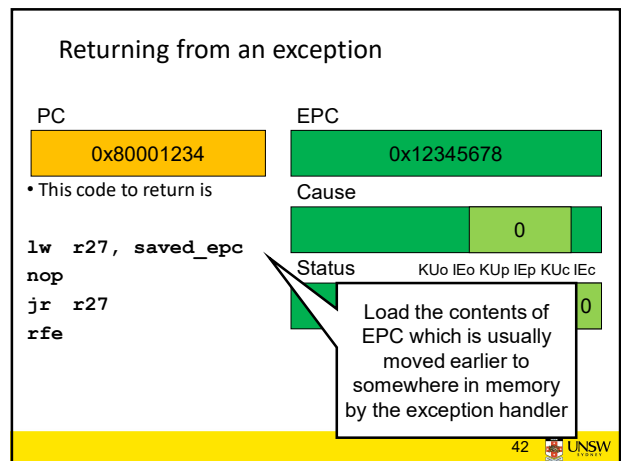
40

Returning from an exception

- For now, lets ignore
 - how the exception is actually handled
 - how user-level registers are preserved
- Let's simply look at how we return from the exception

41 UNSW

41



42

Returning from an exception

PC: 0x12345678

EPC: 0x12345678

• This code to return is

```
lw r27, saved_epc
nop
jr r27
rfe
```

Cause: 0

Status: KUo IEo KU_p IE_p KU_c IE_c (bits: 1 1 0 0)

Store the EPC back in the PC

43 UNSW

43

Returning from an exception

PC: 0x12345678

EPC: 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

Status: KUo IEo KU_p IE_p KU_c IE_c (bits: ? ? ? ? 1 1)

44 UNSW

44

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: KUo IEo KU_p IE_p KU_c IE_c (bits: ? ? ? ? 1 1)

45 UNSW

45

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

46 UNSW

46

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

47 UNSW

47

Convention for kernel entry

Preserved → Preserved for C calling convention

Preserved →

Success? →

Args in →

Result →

SysCall No. →

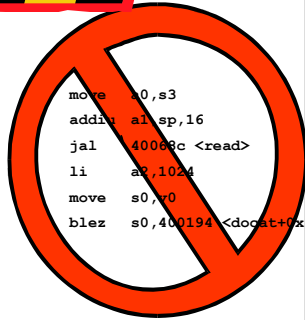
Convention for kernel exit

48 UNSW

48

CAUTION

- 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    s0,400194 <docat+0x94>

```

49

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,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

50

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

51

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

```

Generate a syscall exception

52

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

53

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

```

If failure, store code in errno

54

The read() syscall function part 2

```

00400640 <__syscall>:
400640: 0000000c syscall
400644: 10e00005 beqz a3,400658
400648: 00000000 nop
40064c: 3c011000 lui at,0x1000
400650: ac220000 sw v0,0(at)
400654: 2403ffff li v1,-1
400658: 2402ffff li v0,-1
40065c: 03e00008 jr ra
400660: 00000000 nop
    
```

Set read() result to -1

55 UNSW

55

The read() syscall function part 2

```

00400640 <__syscall>:
400640: 0000000c syscall
400644: 10e00005 beqz a3,400658
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

56 UNSW

56

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.

57 UNSW

57

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

58 UNSW

58

OS/161 Exception Handling

- Note: The following code is from the uniprocessor variant of OS/161 (v1.x).
 - Simpler, but broadly similar to current version.

59 UNSW

59

```

exception:
    move k1, sp /* Save previous stack pointer in k1 */
    mfc0 k0, c0_status /* Get status register */
    andi k0, k0, CST_USR /* Check the we-were-in-user-mode bit */
    beq k0, $0, if_clear /* Clear, from kernel, already have stack */
    nop /* delay slot */

    /* Coming from user mode, restore to sp */
    la k0, curkstack /* kernel stack */
    lw sp, 0(k0)
    nop /* delay slot */

1:
    mfc0 k0, c0_cause /* Note cause. */
    j common_exception /*
    nop
    
```

Note k0, k1 registers available for kernel use

60 UNSW

60

```

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 */

```

61 UNSW

61

```

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

```

62 UNSW

62

/* 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.

63 UNSW

63

/* 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 */

```

The real work starts here

64 UNSW

64

```

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 a0, 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

65 UNSW

65

```

/*
 * 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)

```

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

66 UNSW

66

```

/*
 * 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

67

```

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;
    u_int32_t tf_hi;
    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) */
    /* etc. */
    u_int32_t tf_v1;
    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_t1;
    u_int32_t tf_t2;
    u_int32_t tf_t3;
    u_int32_t tf_t4;
    u_int32_t tf_t5;
    u_int32_t tf_t6;
    u_int32_t tf_t7;
    u_int32_t tf_s0;
    u_int32_t tf_s1;
    u_int32_t tf_s2;
    u_int32_t tf_s3;
    u_int32_t tf_s4;
    u_int32_t tf_s5;
    u_int32_t tf_s6;
    u_int32_t tf_s7;
    u_int32_t tf_s8;
    u_int32_t tf_s9;
    u_int32_t tf_k0;
    u_int32_t tf_k1;
    u_int32_t tf_gp;
    u_int32_t tf_sp;
    u_int32_t tf_s8;
    u_int32_t tf_epc; /* coprocessor 0 epc register */
};

```

Kernel Stack

epc
s8
sp
gp
k1
k0
t9
t8
...
at
ra
hi
lo
cause
status
vaddr

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

68

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;

```

69

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

70

```

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)

```

71

```

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 */

```

72

```
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