# 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

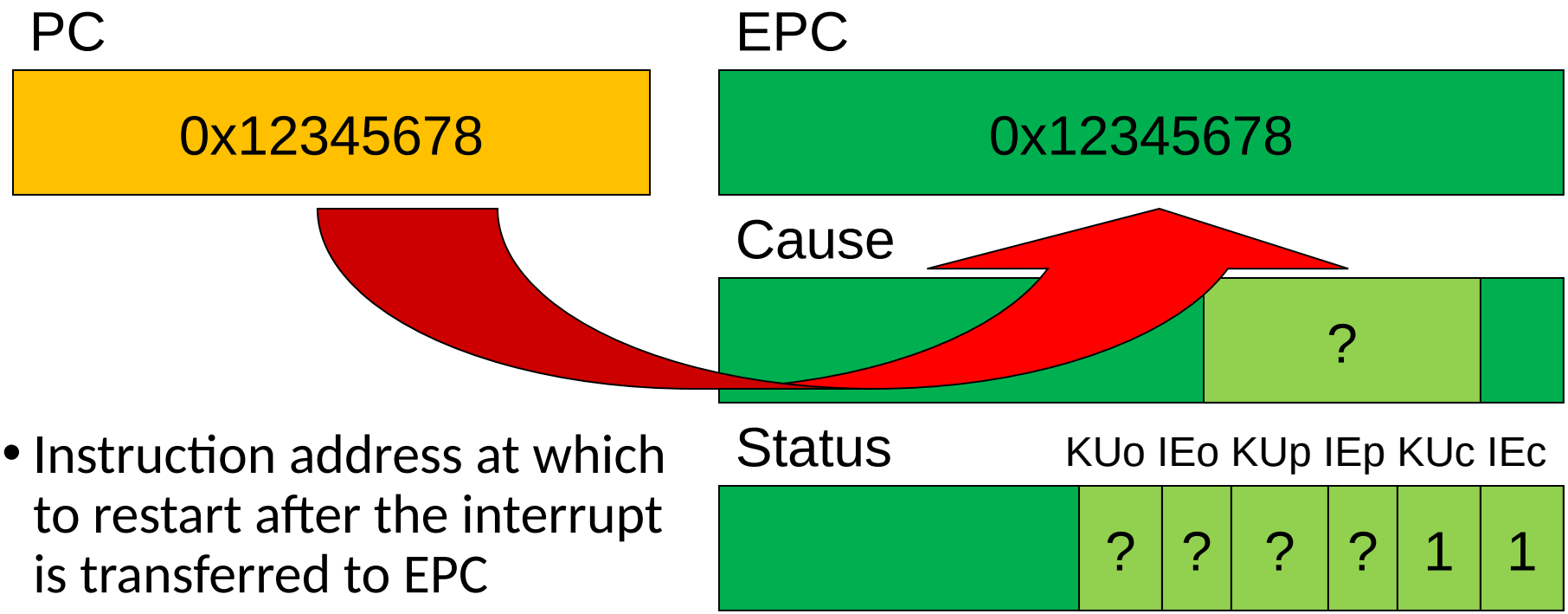
# System Calls Recap

Last lecture, we learned about system calls:

- User Interface: 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

- Implementation:
  - The hardware provides an exception entry mechanism
  - The hardware saves enough state the kernel entry can run
  - The hardware enters the kernel at an **exception vector address**
  - The kernel entry saves additional user state and enters C

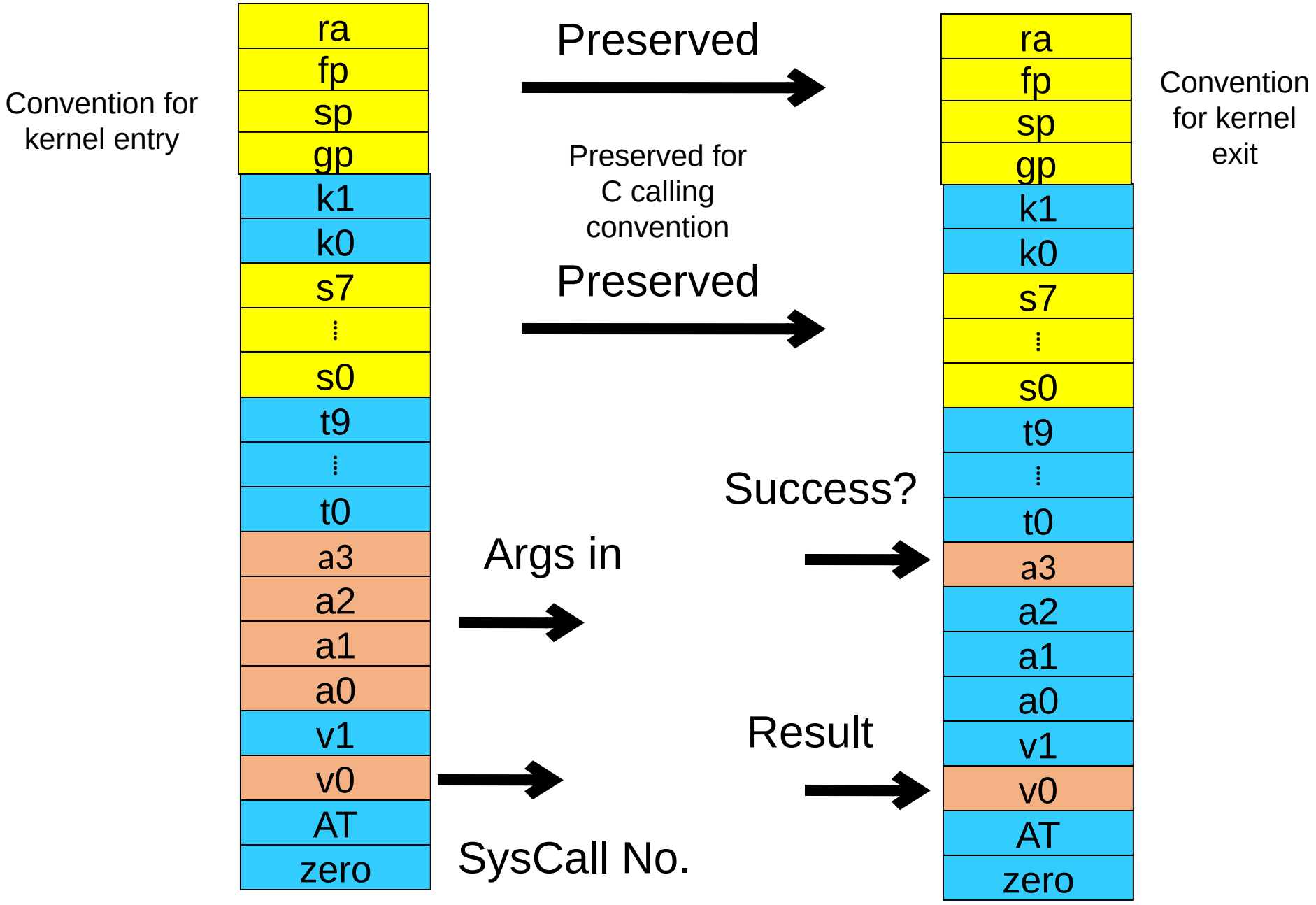
# Hardware exception handling



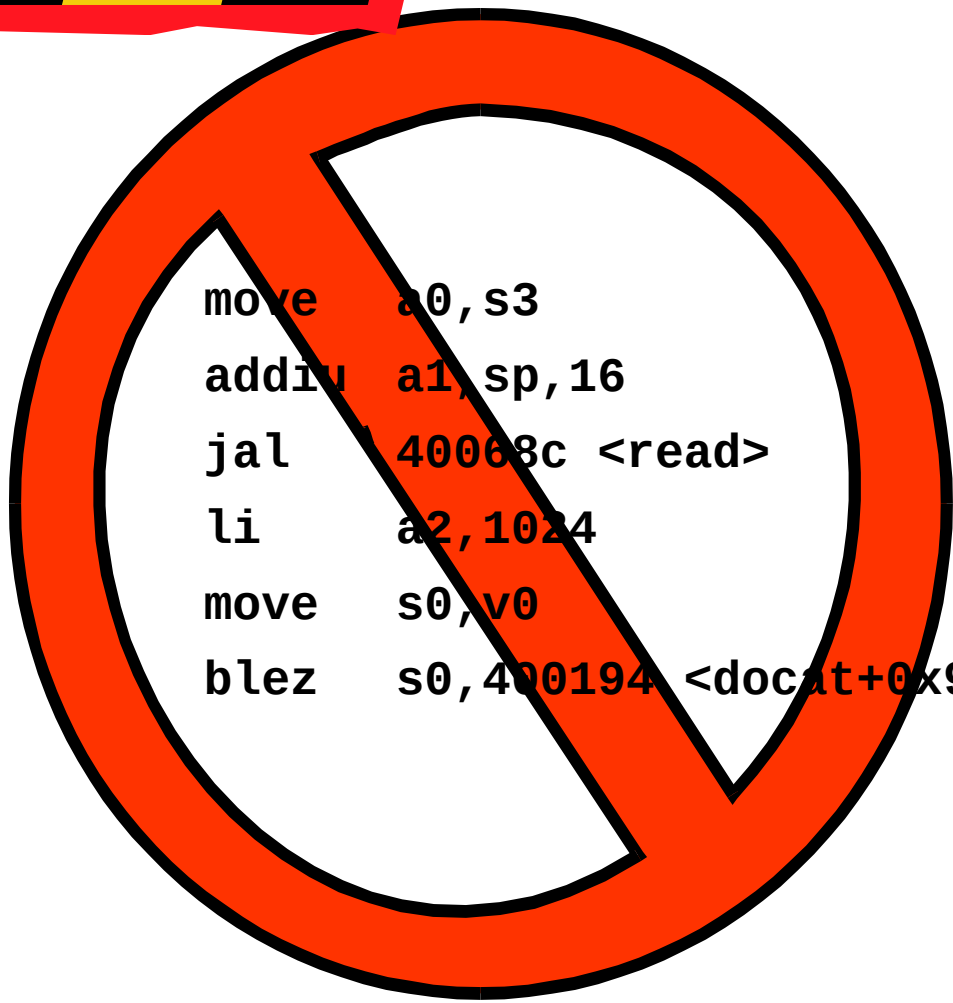

- Instruction address at which to restart after the interrupt is transferred to EPC

# 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
  - Compare this to “man errno”


Convention for kernel entry
ra
fp
sp
gp
k1
k0
s7
⋮
s0
t9
⋮
t0
a3
a2
a1
a0
v1
v0
AT
zero
Preserved
Preserved for C calling convention
Preserved
Args in
SysCall No.
Success?
Result
ra
fp
sp
gp
k1
k0
s7
⋮
s0
t9
⋮
t0
a3
a2
a1
a0
v1
v0
AT
zero
Convention for kernel exit


- Seriously low-level code follows
- This code is not for the faint hearted



# 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>
```

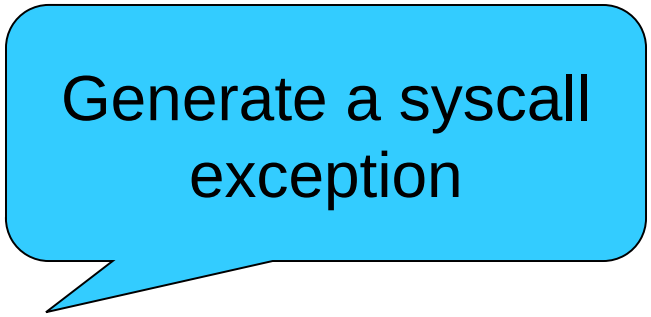
- Args are loaded, return value is tested

# 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




```
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
```

# The read() syscall function part 2

Test success, if yes, branch to return from function

```
00400640 <__syscall>:
  400640:  0000000c      syscall
  400644:  10e00005      beqz a3,40065c <__syscall+0x1c>
  400648:  00000000      nop
  40064c:  3c011000      lui  at,0x1000
  400650:  ac220000      sw   v0,0(at)
  400654:  2403ffff      li   v1,-1
  400658:  2402ffff      li   v0,-1
  40065c:  03e00008      jr   ra
  400660:  00000000      nop
```

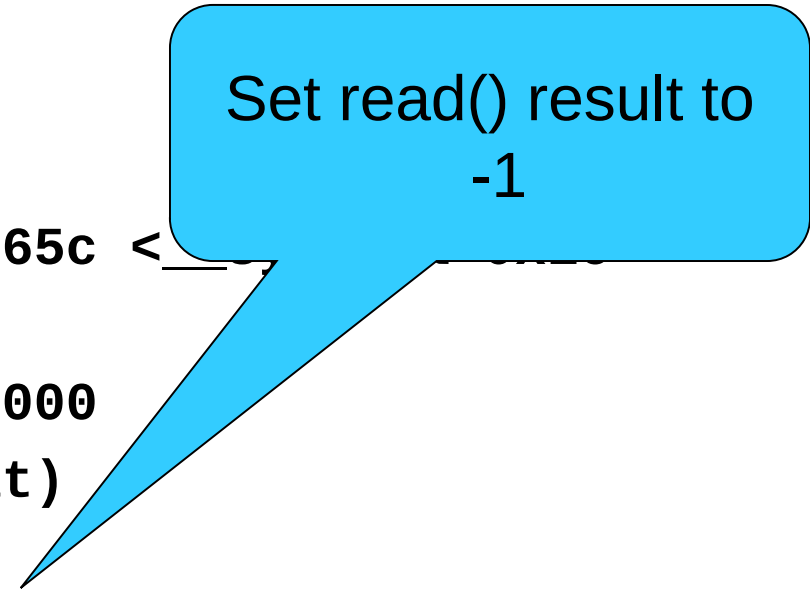
# The read() syscall function part 2

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

If failure, store code in *errno*

# The read() syscall function part 2

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



# The read() syscall function part 2

```
00400640 <__syscall>:
  400640:  0000000c      syscall
  400644:  10e00005      beqz a3,40065c <__
  400648:  00000000      nop
  40064c:  3c011000      lui  at,0x1000
  400650:  ac220000      sw   v0,0(at)
  400654:  2403ffff      li   v1,-1
  400658:  2402ffff      li   v0,-1
  40065c:  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()”
    - This might block or take a while
  - 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_K          /* Check the we-were-in-user-mode bit */
   beq k0, $0, 1f /* If       ar, from kernel, already have stack */
   nop                         /*       y slot */

   /* Coming from user mode                                  to sp */
   la k0, curkstack                                        rkstack" */
   lw sp, 0(k0)
   nop                         /*                                */

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

Note k0, k1 registers available for kernel use

```
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 exce
   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

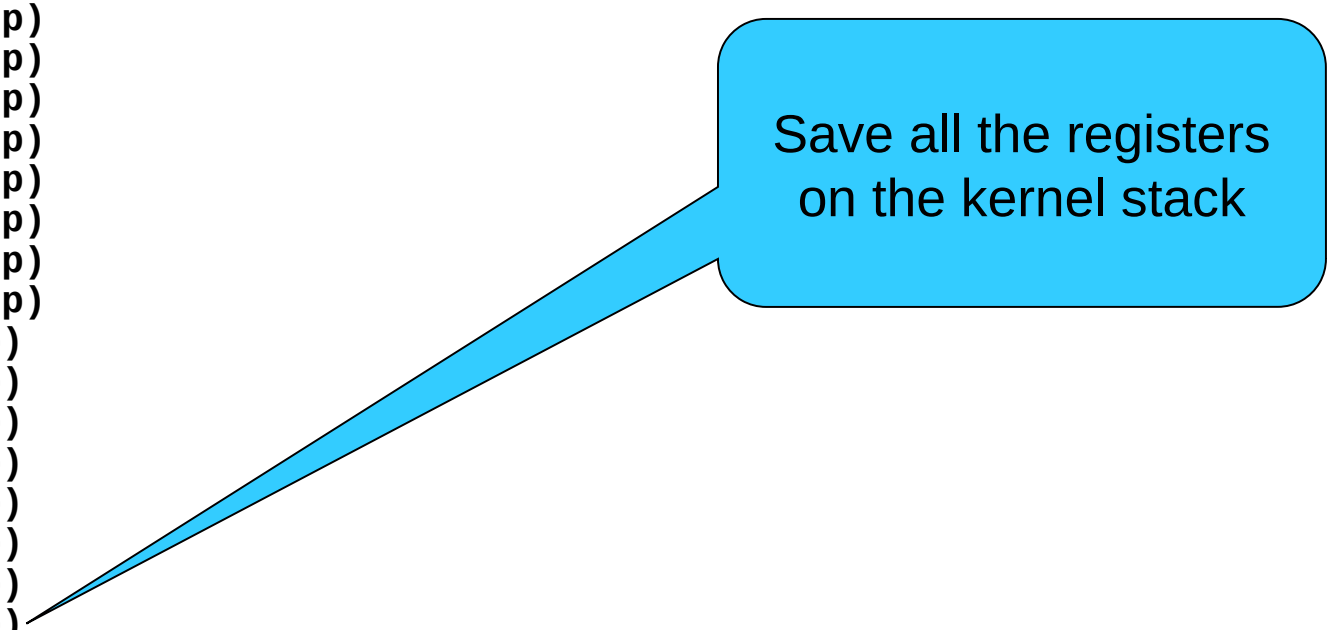
```
/* The order here must match mips/include/trapframe.h. */

 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 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

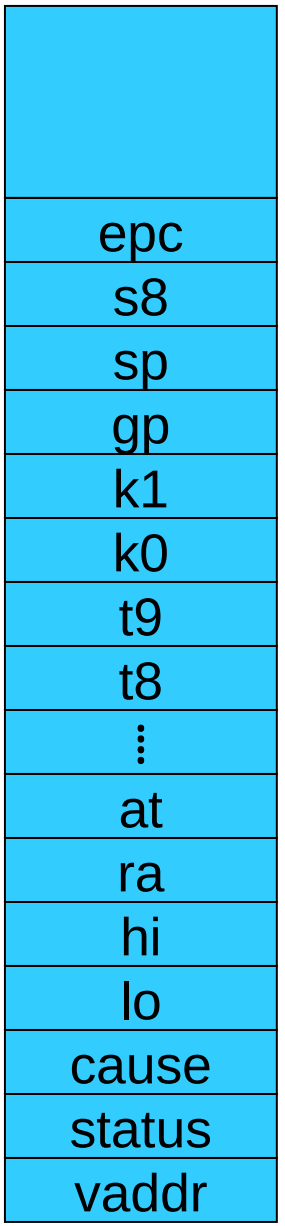
```
 /*
 * Prepare to call mips_trap(struct trapframe *)
 */

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;
  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) */
  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;
  */
  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 regi
};
```

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 C code 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_exceptio
```

Note again that only k0, k1 have been trashed

# System Calls Implementation

- User Interface: system calls can be viewed as special function calls

- Implementation:
  - The hardware provides an exception entry mechanism
  - The hardware saves enough state the kernel can get started
  - The hardware enters the kernel at an **exception vector address**
  - The kernel entry saves additional user state and enters C

  - The kernel exit restores user register values
  - The kernel exit restores user-mode and user PC