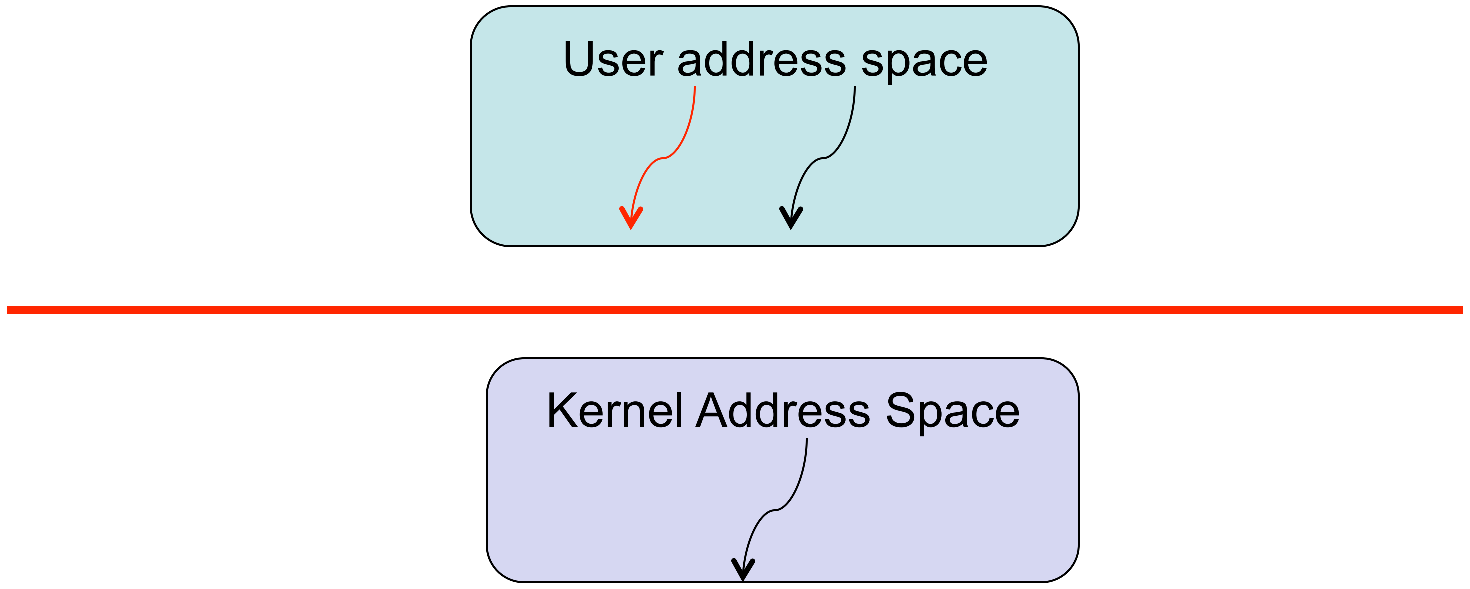
Switching and Crossing

- Topics
  - Terminology.
  - What hardware support is necessary to support multiprogramming?
  - How does all this work on the MIPS?
- Learning Objectives:
  - Be prepared to tackle Assignment 2!

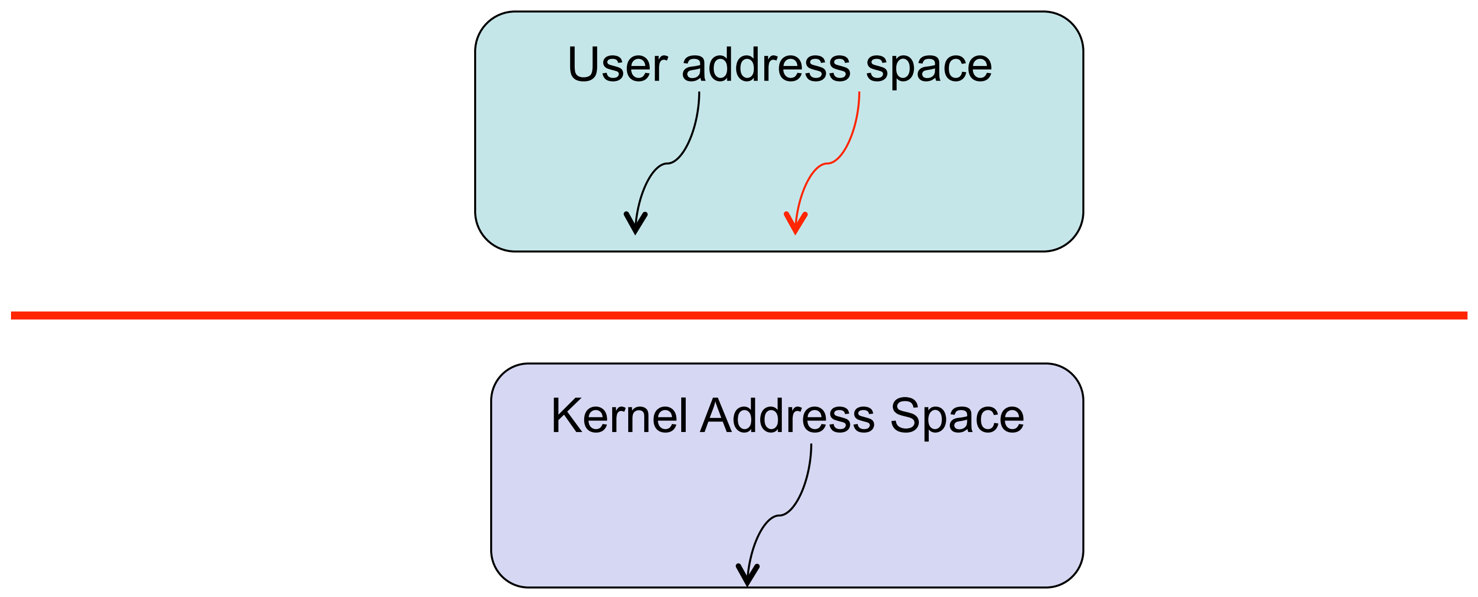
Terminology

- Thread switch: Changes from one thread of execution to another.
  - Does not require a change of protection domain.
  - Continue running in the same address space.
  - Can change threads in user mode or in supervisor mode.
- Domain crossing: Changes the privilege at which the processor is executing.
  - Can change from user to supervisor.
  - Can change from supervisor to user.
  - Requires a trap or return from trap.
  - Requires an address space change (either user to kernel or kernel to user)
- Process switch: Changes from execution in one (user) process to execution in another (user) process.
  - Requires two domain crossings + a thread switch in the kernel.
- Context switch: usage varies
  - Sometimes used for either thread or process switch.
  - Sometimes used to mean only process switch.
  - Every once in a while used to mean domain crossing.

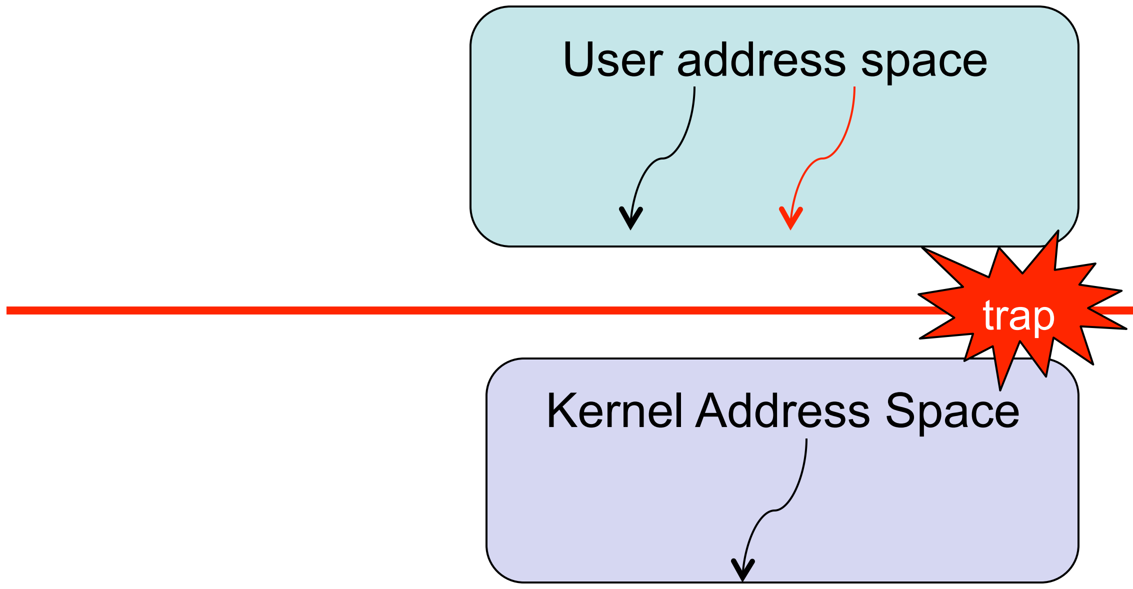
User-Level Thread Switch (1)



User-Level Thread Switch (2)



Domain Crossing



What Causes a Trap?

- The thread requests a trap: System Call
  - Every system call requires a domain crossing.
- The thread does something bad: Exception
  - E.g., Accessing invalid memory.
- An external event happens: Interrupt
  - E.g., Timer goes off, disk operation completes, network packet arrives, one processor pokes another.

- Regardless of the cause, the kernel handles all traps.
  - A user process that causes a trap causes a domain crossing.
  - A trap that happens while the kernel is already running, does not cause a domain crossing.

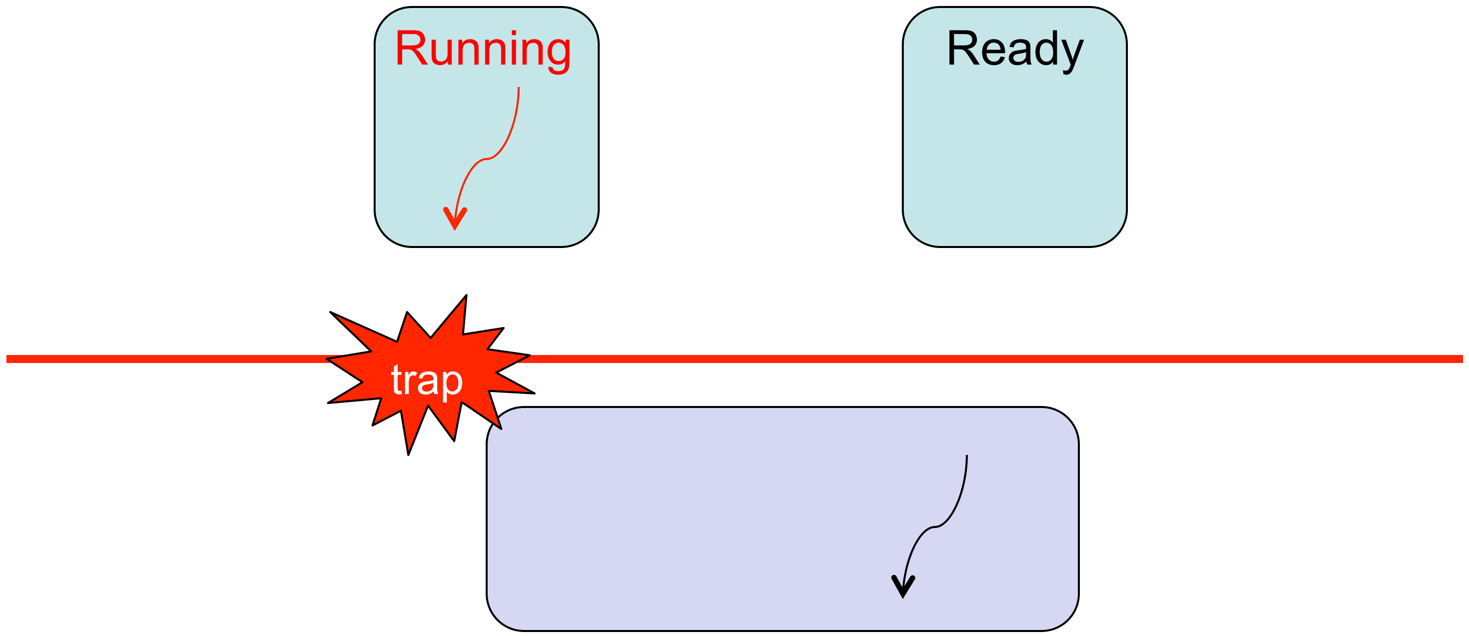
What Does the Kernel do on a Trap?

- The kernel has to find a stack on which to run.
  - If you were already in the kernel, the stack you use is the same as the one on which you were running.
  - If you were running in userland, then you have to find a stack on which to run.
  - Implication: every real* user level thread has a corresponding kernel stack.
- Before doing anything else, the kernel has to save state
  - We'll go through this in detail in a few slides.
- Figure out what caused the trap.

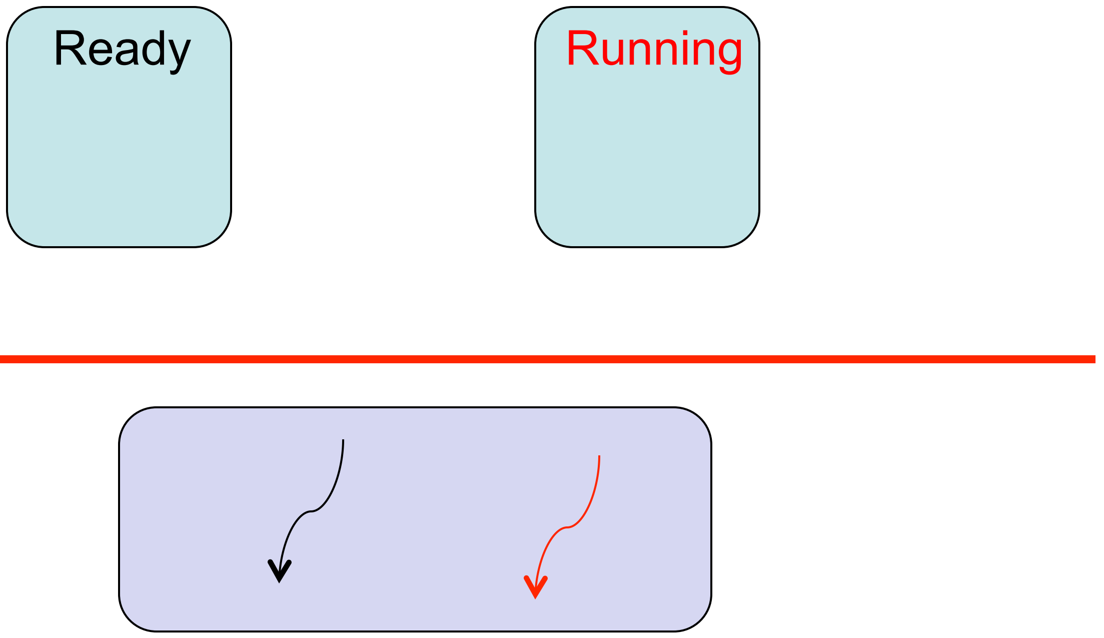
- Note: Whenever the kernel runs, it has the option of changing to another thread.

* You can have purely user-level thread implementations; for now, those aren't "real"

Process Switch (1)



Process Switch (2)



Summarize Process Switch

1. Change protection domain (user to supervisor[kernel]).
2. Change stacks: switch from using the user-level stack to using a kernel stack.
3. Save execution state (on kernel’s stack).
4. Do kernel stuff
5. Kernel thread switch
6. Restore user-level execution state
7. Change protection domain (from supervisor[kernel] to user)

Intel Domain Crossing

- Hardware does it all
  - Saves and restores all the state using a special data segment called the Task State Segment.
- Software assist alternative (used by most modern systems)
  - A cross-protection ring function call saves the EIP (instruction pointer), the EFLAGS (contains user/kernel bit), and code segment (CS) on the stack and saves old value of stack pointer and stack segment.
  - Software does the rest.

MIPS Domain Crossing

- The PC is saved into a special supervisor register.
- The status and cause registers (two other special purpose registers) are set to reflect the details of the trap being processed.
- The processor switches into kernel mode with interrupts disabled.
- *The rest is done in software*.

MIPS R3000 Hardware

- Update status register (CP0 $12) with bits that:
  - turn off interrupts
  - put processor in supervisor mode
  - indicate prior state (interrupts on/off; user/supervisor mode)
- Sets cause register (CP0 $13) with
  - what trap happened
  - bit indicating if you are in a branch delay slot
- Sets the exception PC (CP0 $14) (address where execution is to resume after we handle the trap)
- Sets the PC to the address of the appropriate handler.

MIPS R3000 Software

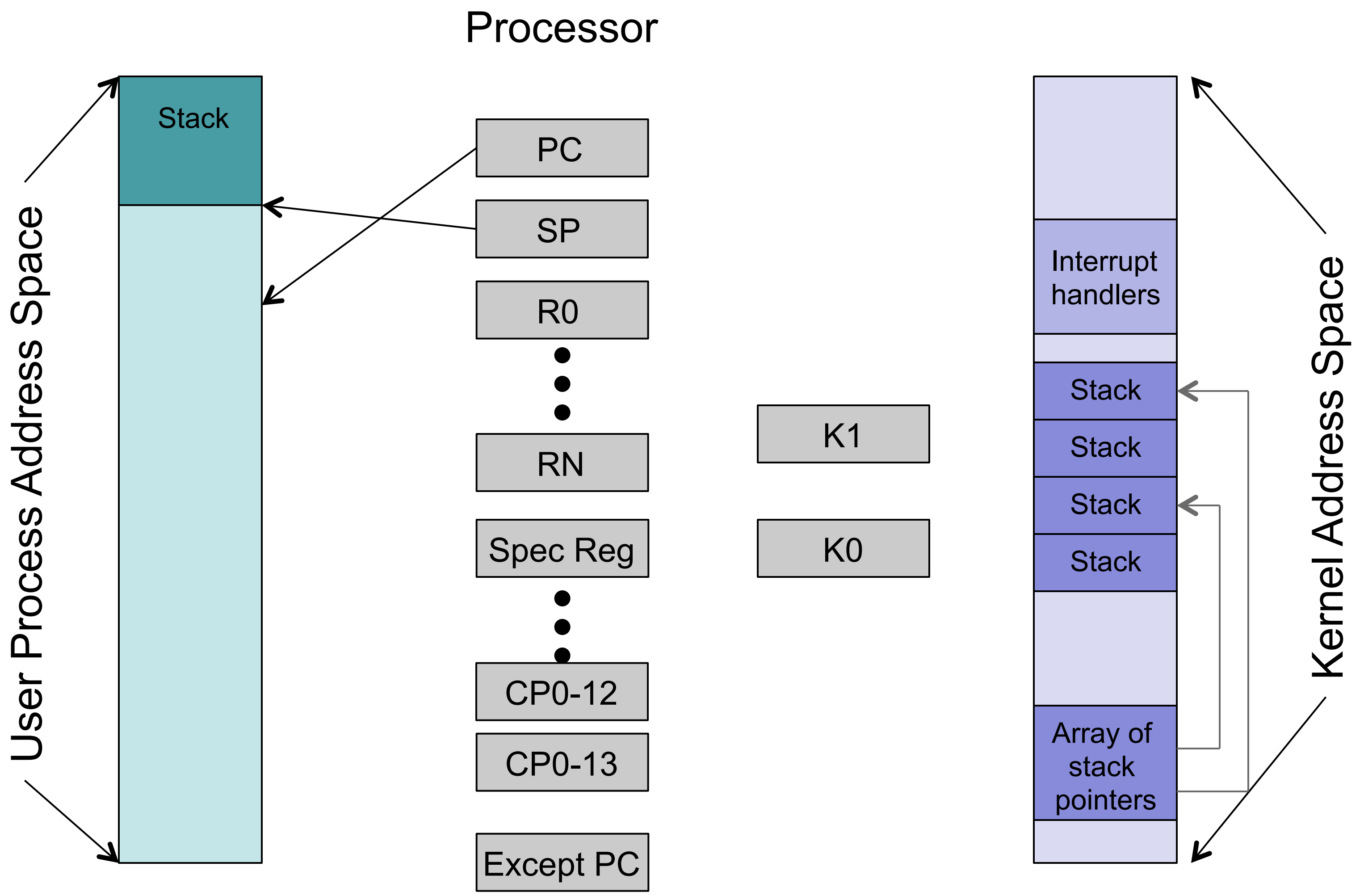
- Save whatever other state that must be saved!
- Since you need to be able to save the user registers and you need to manipulate various entries, there are two registers that the kernel is allowed to use in whatever way is necessary (without this, you couldn't do anything).
- In assembly (`kern/arch/mips/locore/exception-mips1.S`)
  - Save the previous stack pointer
  - Get the status register
  - If we were in user mode:
    - Find the appropriate kernel stack
  - Get the cause of the current trap
  - Create a trap frame (on the kernel stack) that will contain
    - General registers
    - Special registers (status, cause)
  - Now, call the trap handling code (in C).

MIPS R3000 Trap Handling

- First we go to a generic trap handler:
  - `kern/arch/mips/locore/trap.c`:
    - Does a bunch of error handling
    - If this was an interrupt, handle it.
    - If this was a system call, call the system call dispatch.
    - Otherwise, handle other exception cases.
- Then, if this is a system call (215), we go to the system call handler:
  - `kern/arch/mips/syscall/syscall.c`
    - Figure out which system call is needed and dispatch to it.
- Both of these functions assume that all the important information has been stashed away in a trapframe.

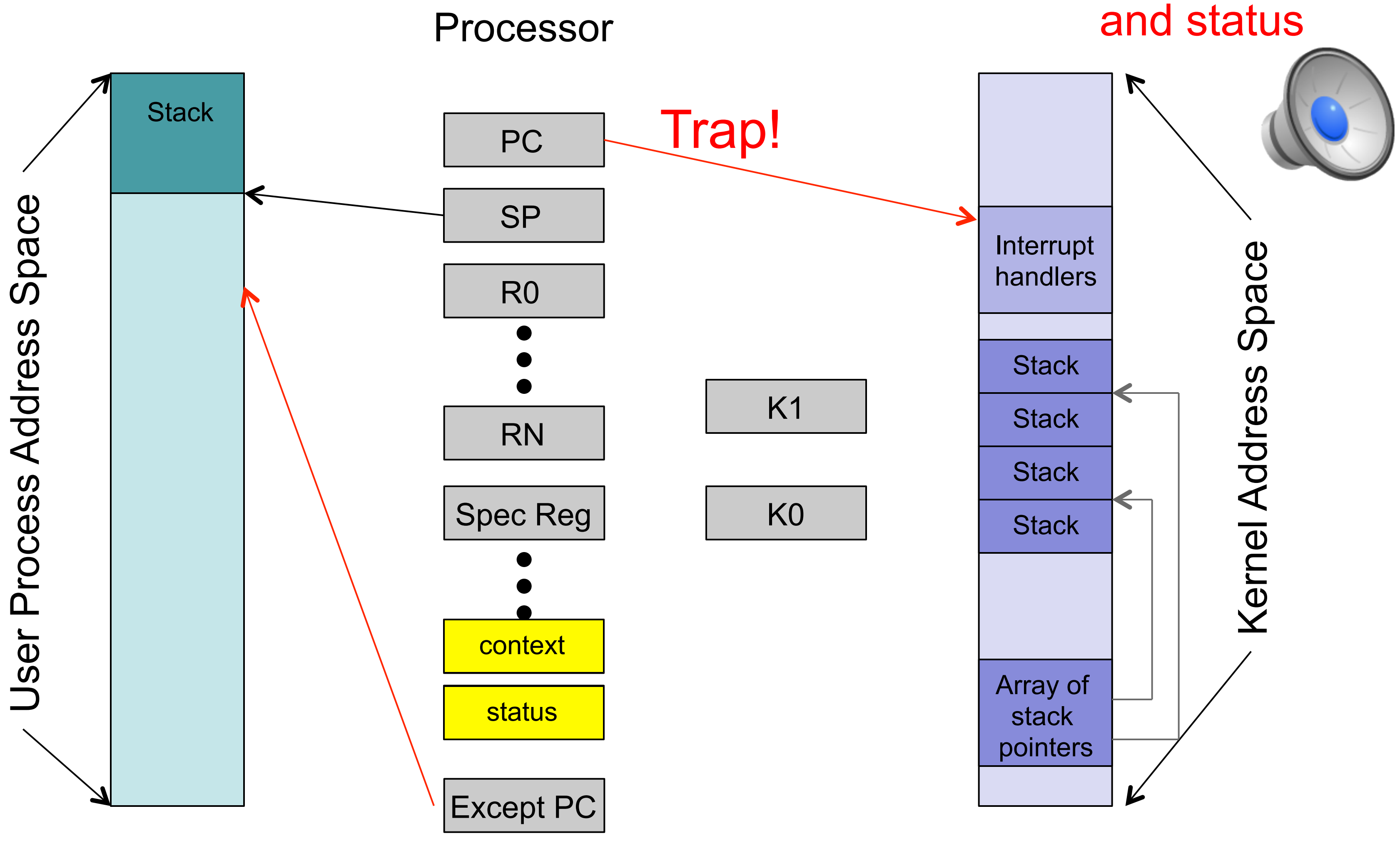
PC and SP reference addresses in user space.

Normal Execution (1)



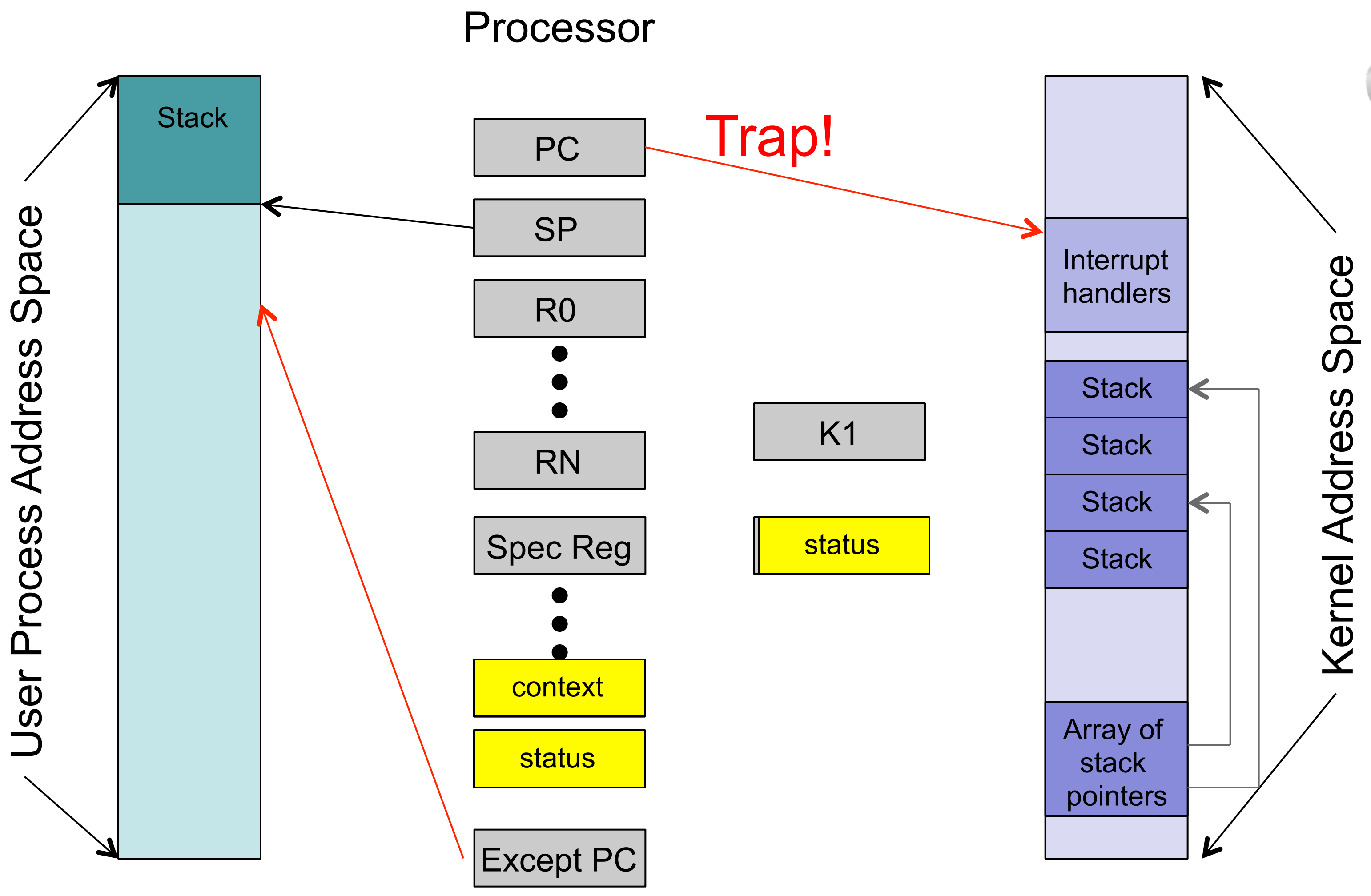
Trap Happens (2)

HW sets:
Exception PC
C0P0 $12/13
get context
and status



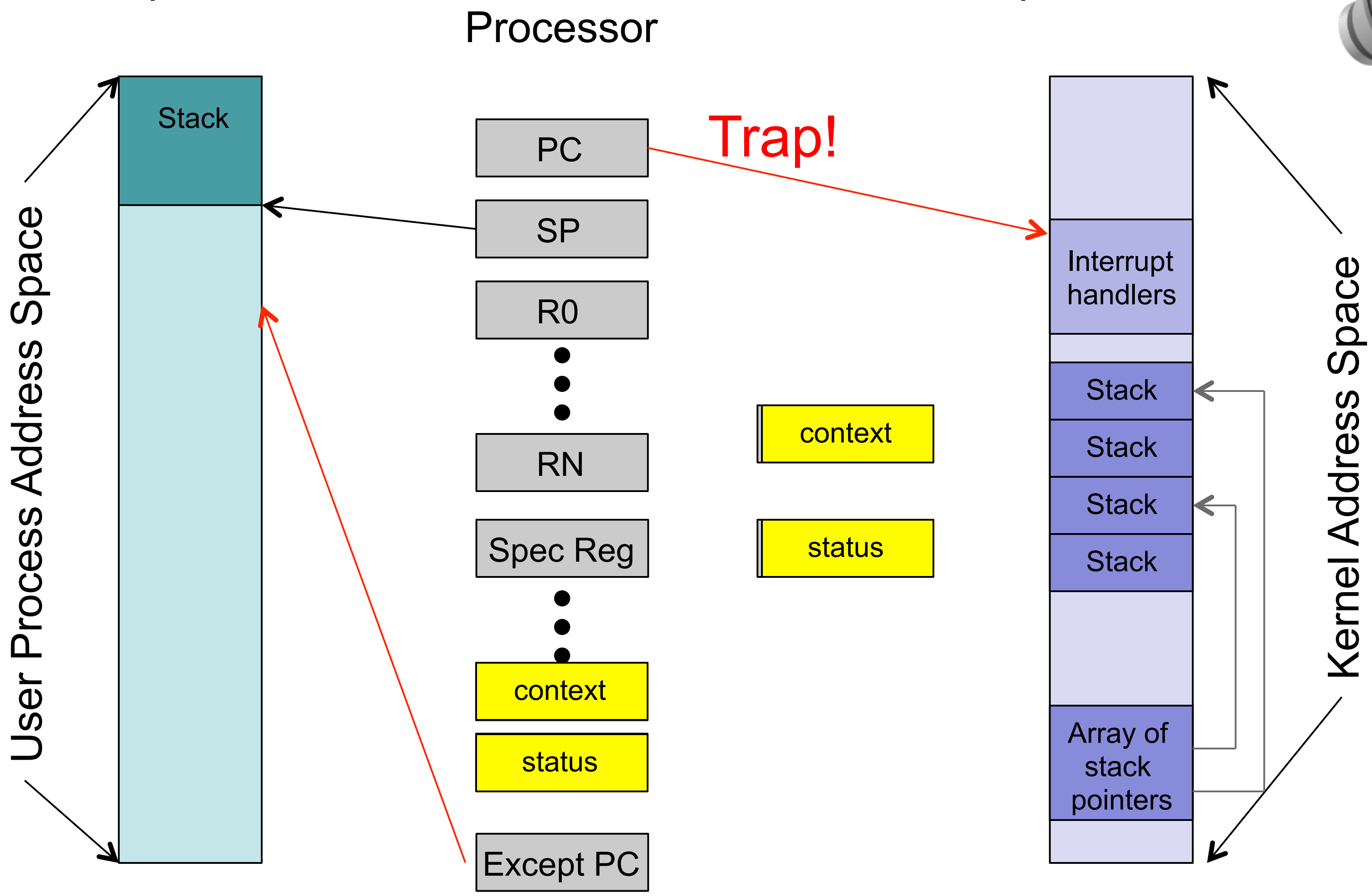
kern/arch/mips/locore/exception-mips1.S(3) line 105-107

SW:
Save status register to k0
Check mode



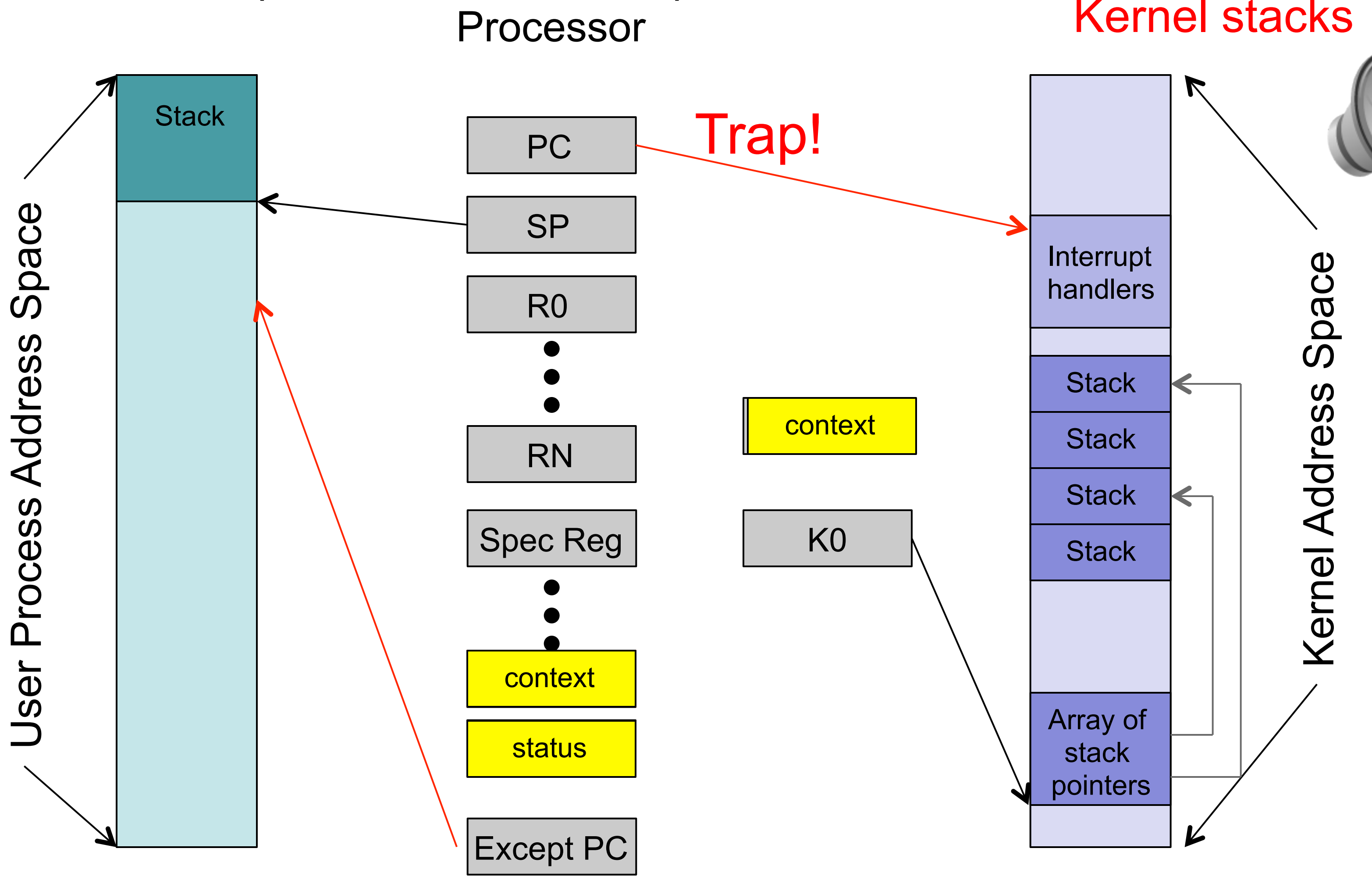
SW:
Save context register into K1

kern/arch/mips/locore/exception-mips1.S(4) line 111 (assume we came from user mode)



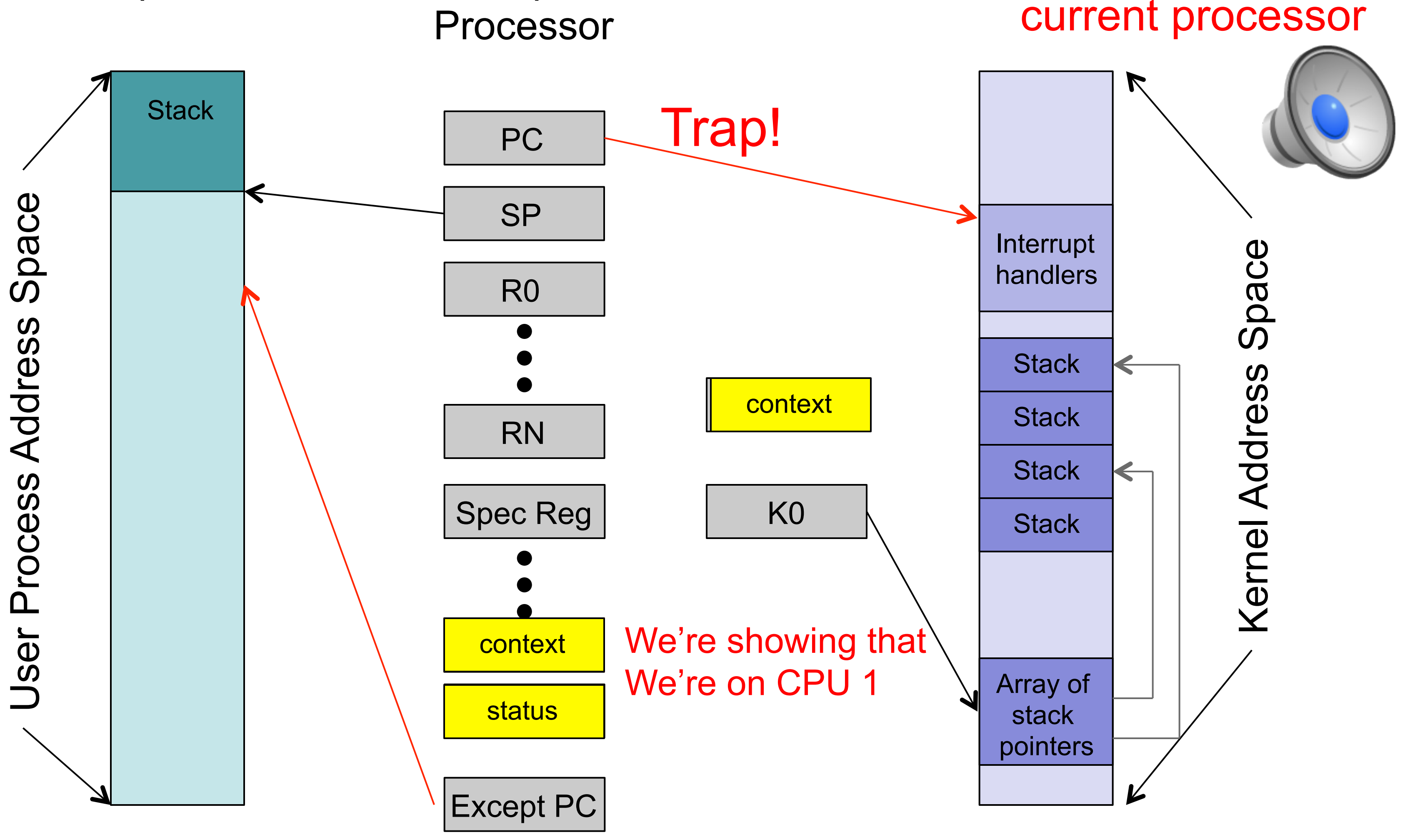
kern/arch/mips/locore/exception-mips1.S(5) line 112-114 (find kernel stack)

SW:
Extract proc # K1
Convert to index
Set K0 to base of
Kernel stacks



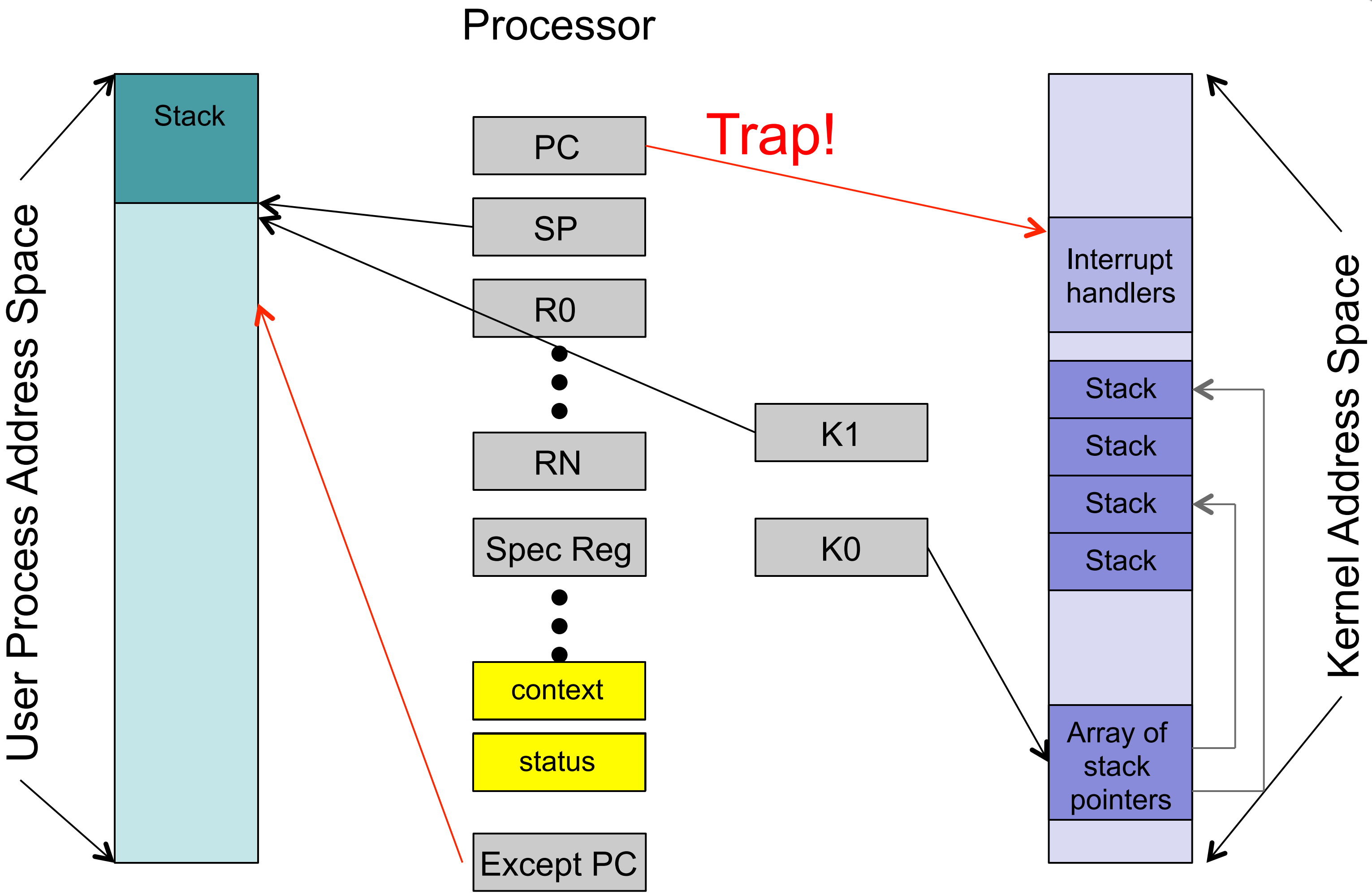
kern/arch/mips/locore/exception-mips1.S(6) line 115 (find kernel stack)

SW:
Add K1 to K0 so that K0 points to the stack for the current processor



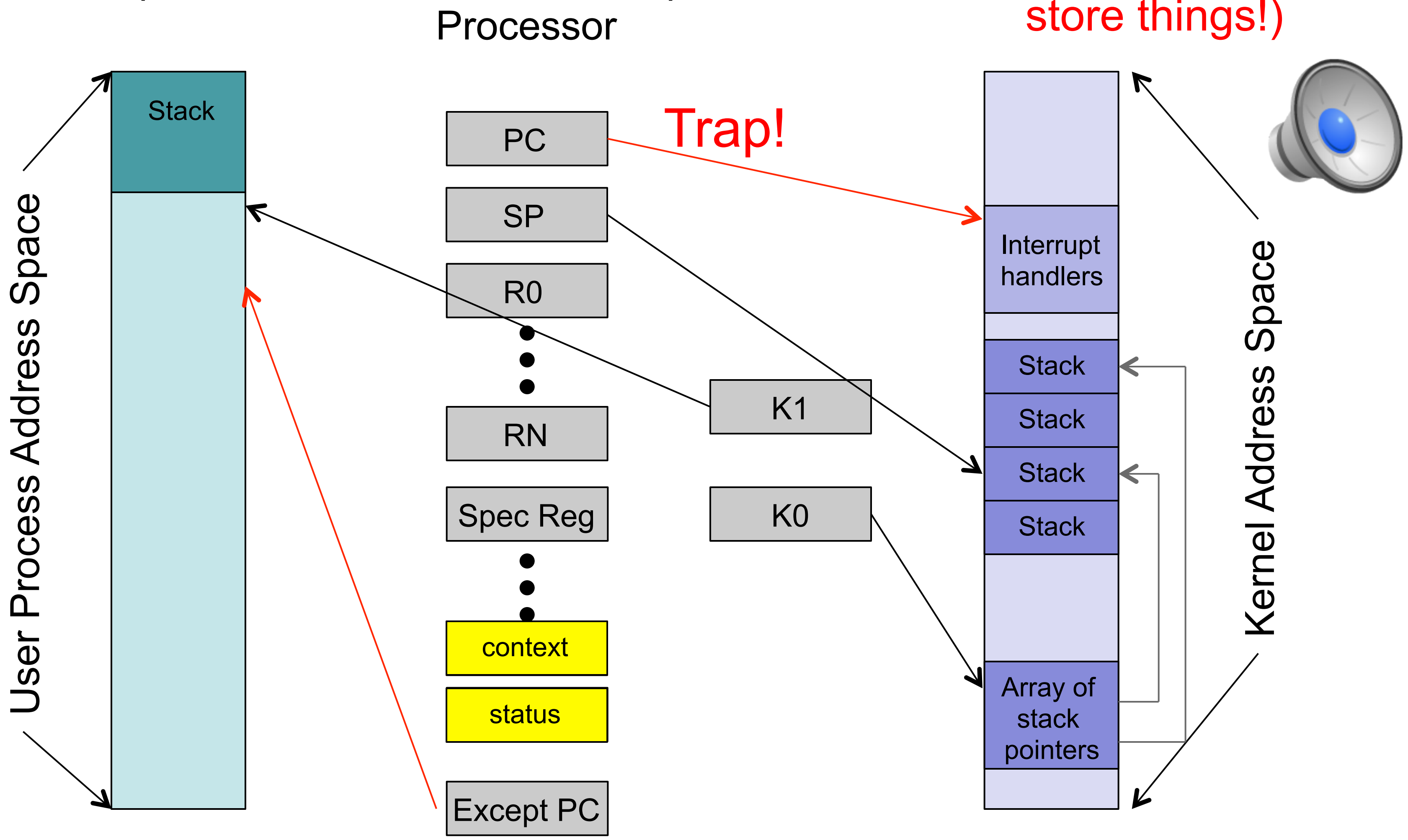
SW:
Save old SP in K1

kern/arch/mips/locore/exception-mips1.S(7) line 116 (save user SP)



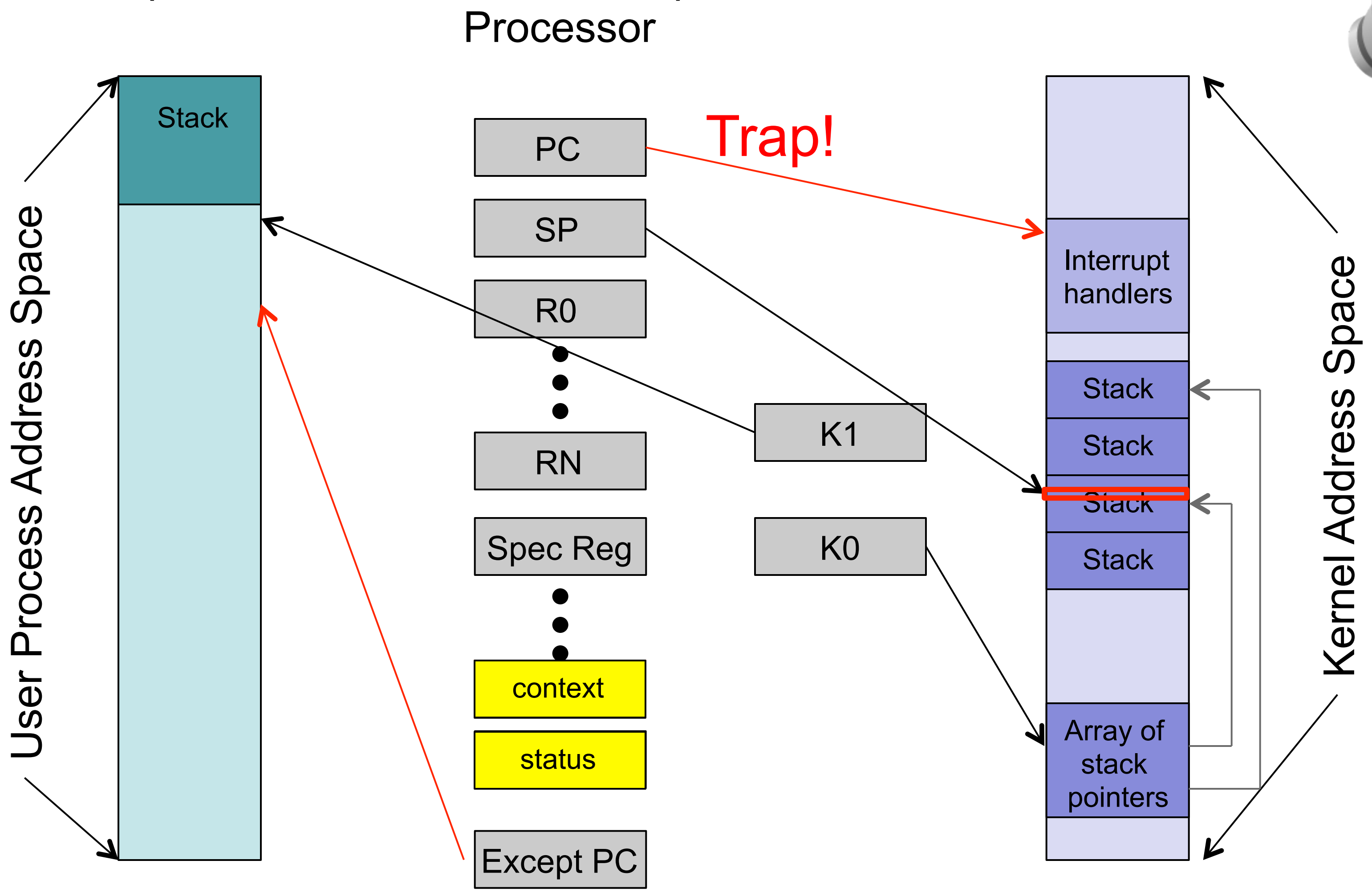
SW:
Set SP to kernel stack (we now have a place to store things!)

kern/arch/mips/locore/exception-mips1.S(8) line 117 (set SP to kernel stack)



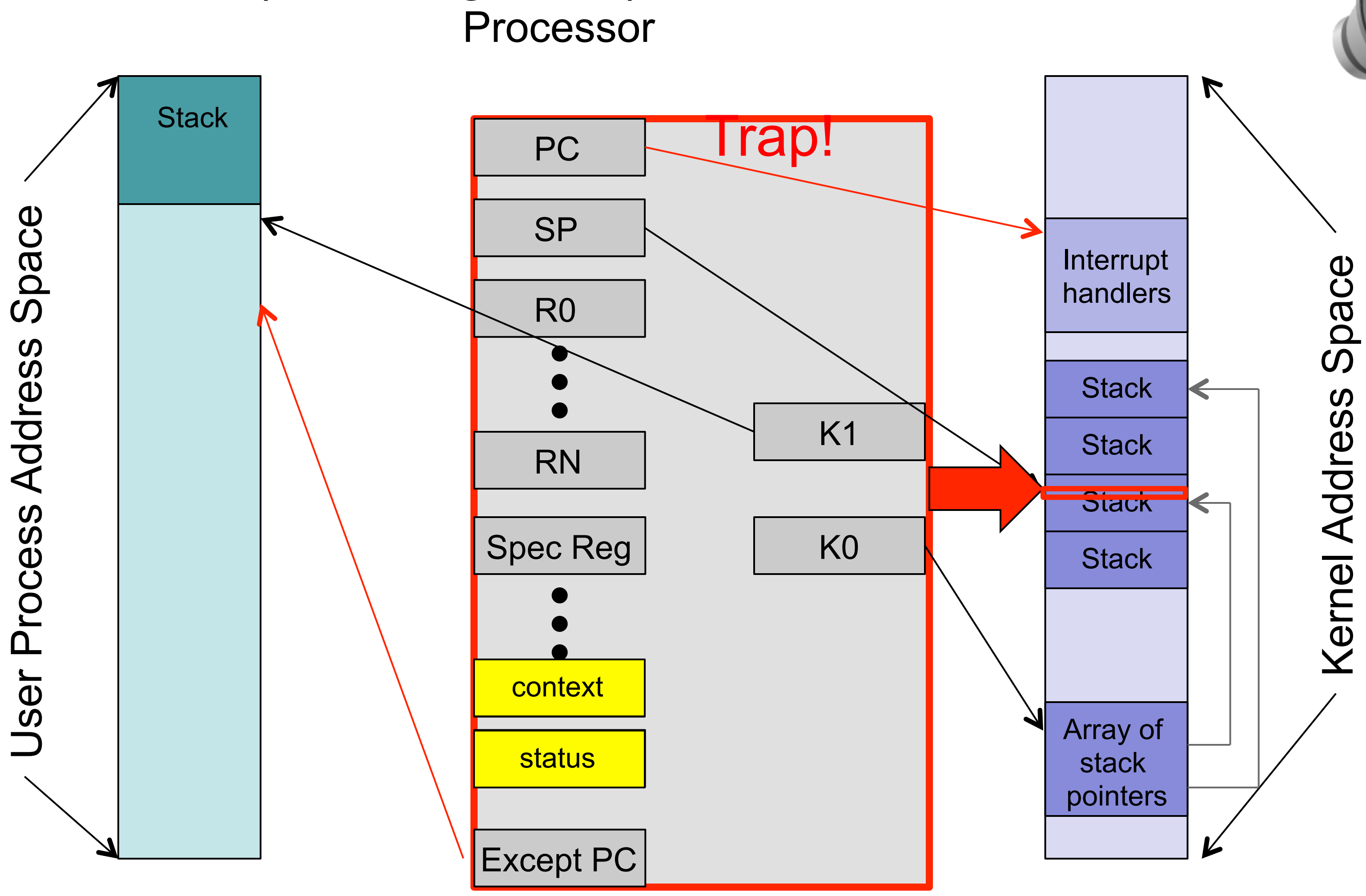
SW:
Allocate trap stack frame (bump SP)

kern/arch/mips/locore/exception-mips1.S(9) line 137 (create a stack frame)



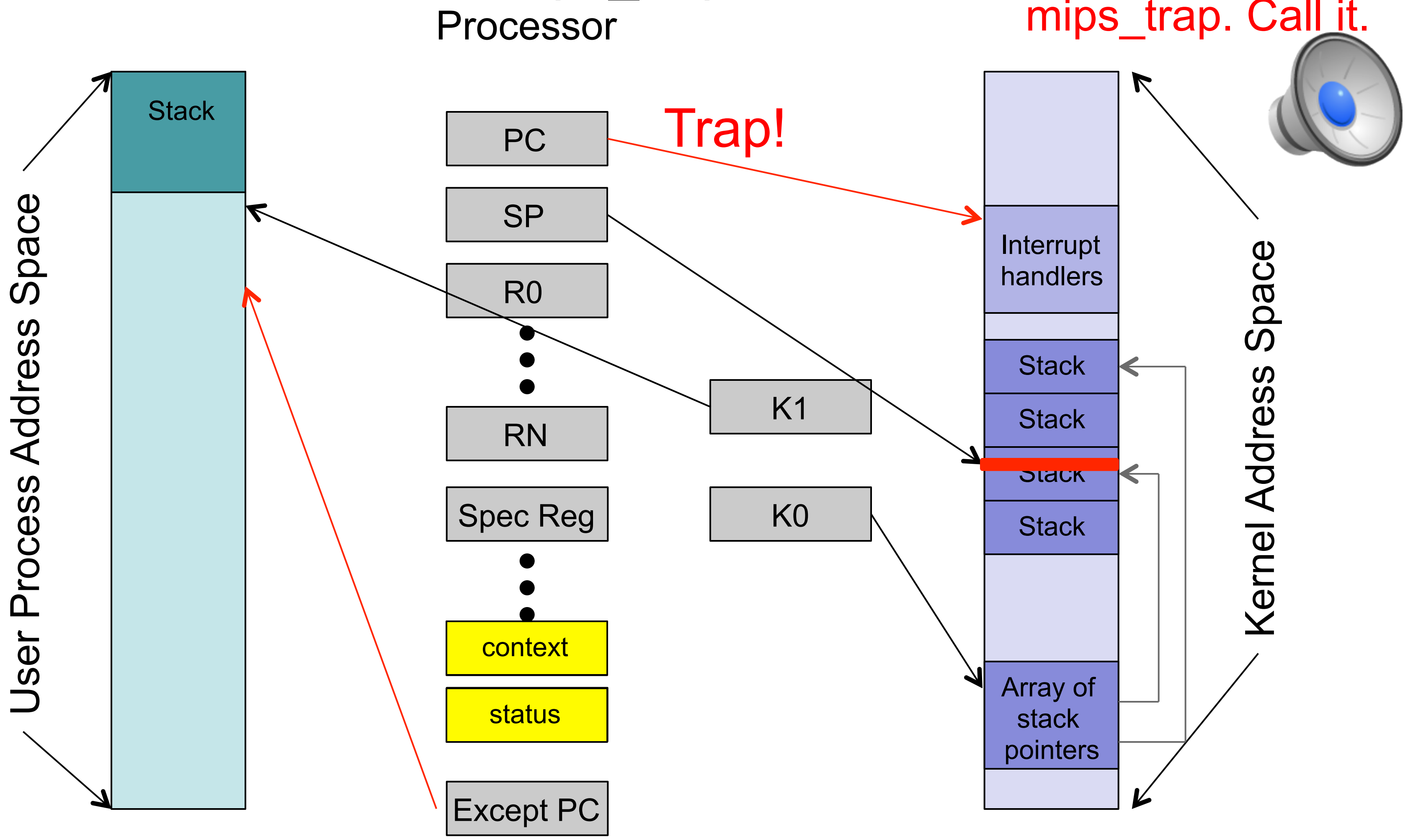
kern/arch/mips/locore/exception-mips1.S(10) line 170-233 (save registers)

SW:
Copy all our registers onto the stack



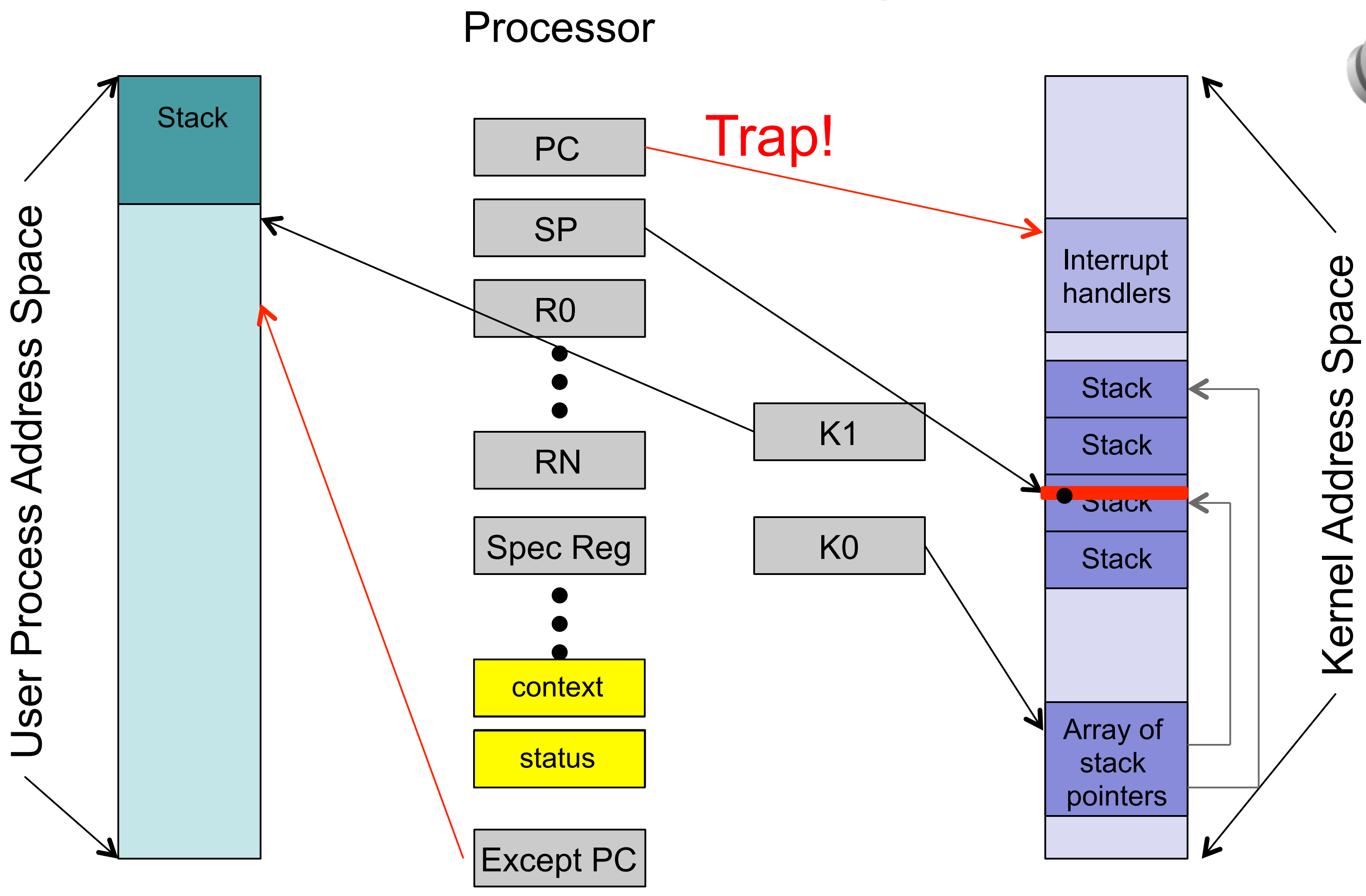
kern/arch/mips/locore/trap.c(11) line 125: We called into mips_trap

SW:
Trapframe on the stack is the argument to mips_trap. Call it.



SW:
Extract exception type from the trap frame.

kern/arch/mips/locore/trap.c(12)
line 431-433: Call routine for this exception



Handling the Syscall (Dispatch)

`kern/arch/mips/syscall.c`

```
void
syscall(struct trapframe *tf)
{
        int callno;
        int32_t retval;
        int err;

        KASSERT(curthread != NULL);
        KASSERT(curthread->t_curspl == 0);
        KASSERT(curthread->t_iplhigh_count == 0);

        callno = tf->tf_v0;

        retval = 0;

        switch (callno) {
            case SYS_reboot:
                err = sys_reboot(tf->tf_a0);
                break;

            case SYS___time:
                err = sys___time((userptr_t)tf->tf_a0,
                                 (userptr_t)tf->tf_a1);
                break;

            /* Add stuff here */
```

Handling the Syscall (Error handling)

`kern/arch/mips/syscall.c`

```
if (err) {
        /*
         * Return the error code. This gets converted at
         * userlevel to a return value of -1 and the error
         * code in errno.
         */
        tf->tf_v0 = err;
        tf->tf_a3 = 1;      /* signal an error */
}
else {
        /* Success. */
        tf->tf_v0 = retval;
        tf->tf_a3 = 0;      /* signal no error */
}

/*
 * Now, advance the program counter, to avoid restarting
 * the syscall over and over again.
 */

tf->tf_epc += 4;

/* Make sure the syscall code didn't forget to lower spl */
KASSERT(curthread->t_curspl == 0);
/* ...or leak any spinlocks */
KASSERT(curthread->t_iplhigh_count == 0);
```

Syscall Details

- Upon entry into our syscall handler, we:
  - Are in supervisor mode
  - Have saved away the process's state
- System call details
  - Where did we leave the arguments?
  - How do we know which system call to execute?
  - Where do we return an error?
- Do we need to do anything special with the arguments?
  - Where does data referenced by an argument live?
  - How do we get to it?

Syscall Details

- Upon entry into our syscall handler, we:
  - Are in supervisor mode
  - Have saved away the process's state
- System call details
  - Where did we leave the arguments?
  - How do we know which system call to execute?
  - Where do we return an error?
- Do we need to do anything special with the arguments?
  - Where does data referenced by an argument live?
  - How do we get to it?

First four are in a0-a3; rest are on the stack.

Syscall Details

- Upon entry into our syscall handler, we:
  - Are in supervisor mode
  - Have saved away the process's state
- System call details
  - Where did we leave the arguments?
  - How do we know which system call to execute?
  - Where do we return an error?
- Do we need to do anything special with the arguments?
  - Where does data referenced by an argument live?
  - How do we get to it?

Syscall
# is in v0

Syscall Details

- Upon entry into our syscall handler, we:
  - Are in supervisor mode
  - Have saved away the process's state
- System call details
  - Where did we leave the arguments?
  - How do we know which system call to execute?
  - Where do we return an error?
- Do we need to do anything special with the arguments?
  - Where does data referenced by an argument live?
  - How do we get to it?

A3 indicates success/failure; v0 contains errno or return value

Copyin/Copyout

- Processes that issue system calls with pointer arguments pose two problems:
  - The items referenced reside in the process address space.
  - Those pointers could be bad addresses.
- Most kernels have some pair of routines that perform both of these functions.
- In OS/161 they are called: copyin, copyout
  - Copyin: verifies that the pointer is valid and then copies data from a user process address space into the kernel’s address space.
  - Copyout: verifies that the address provided by the user process is valid and then copies data from the kernel back into a user process.

Creating User Processes

- Once we have one user process, creating new ones is easy: `fork`:
  - OS makes sure forking process is **not** running at user-level
    - That is, if the process is multi-threaded, no other threads are currently active. (Why?)
  - Save all state from forking process.
  - Make a copy of the code, data, and stack.
  - Copy trap frame of the original process into the new one.
  - Make the new process known to the dispatcher.
- `Exec`:
  - replace the current program image with the code and data of a new program.