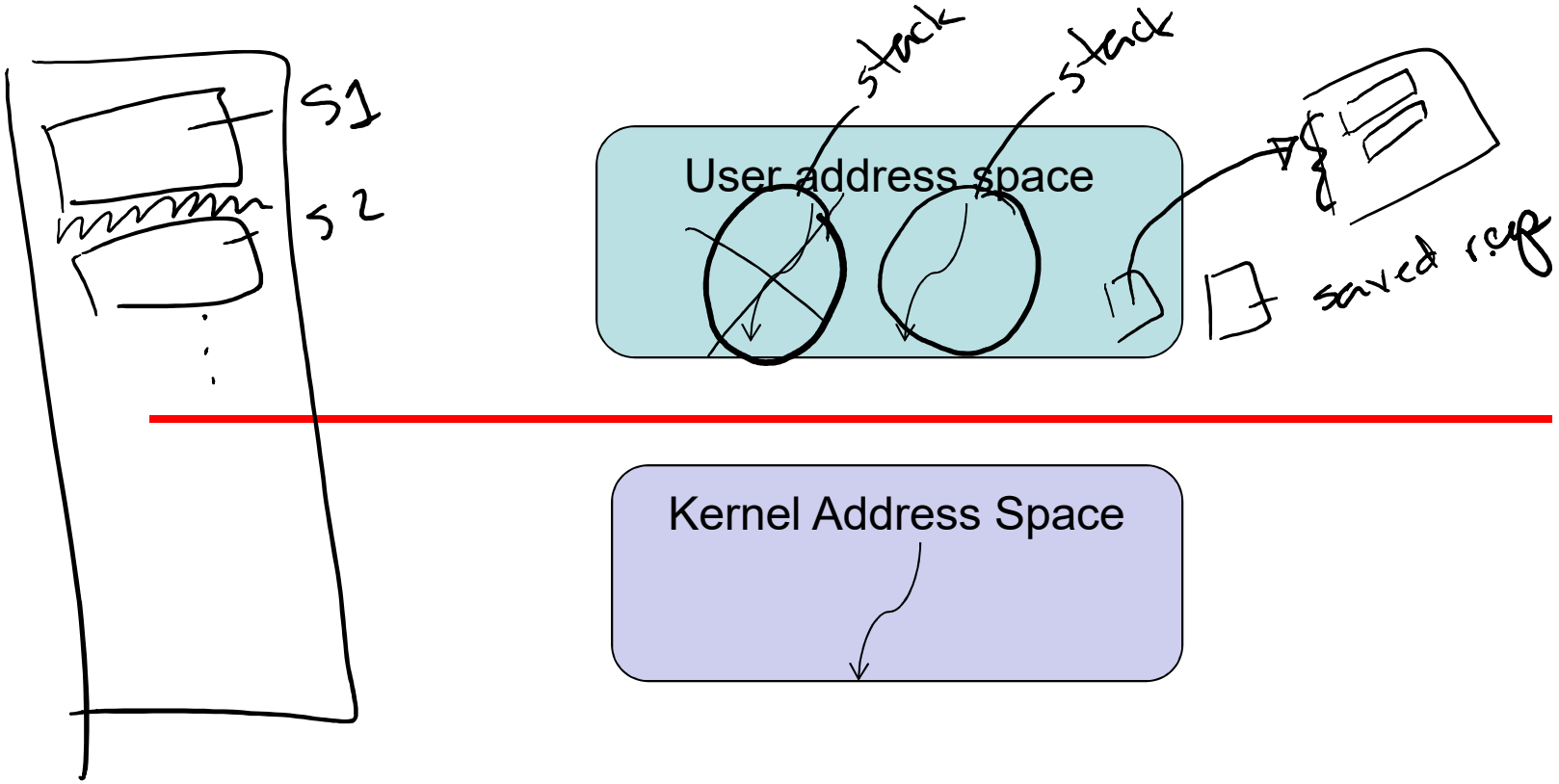
Switching and Crossing

- 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!

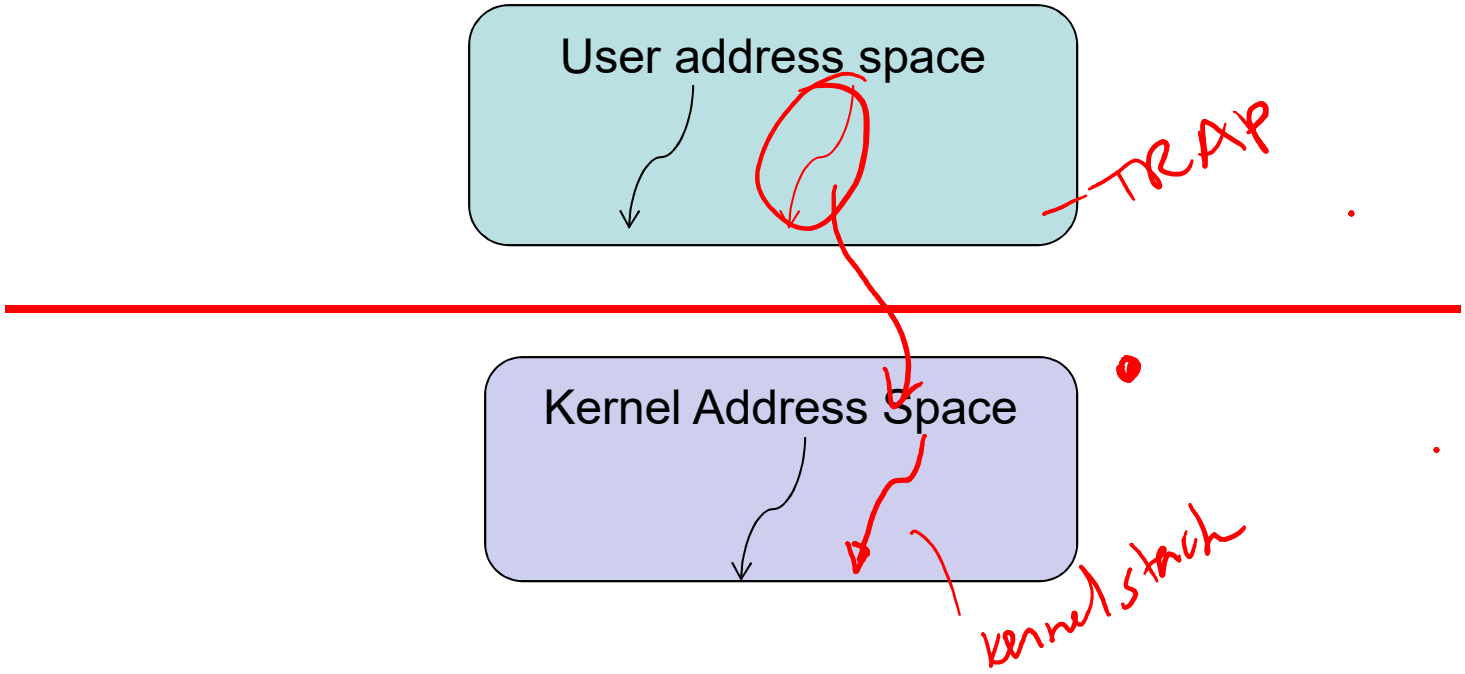
Terminology

- 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



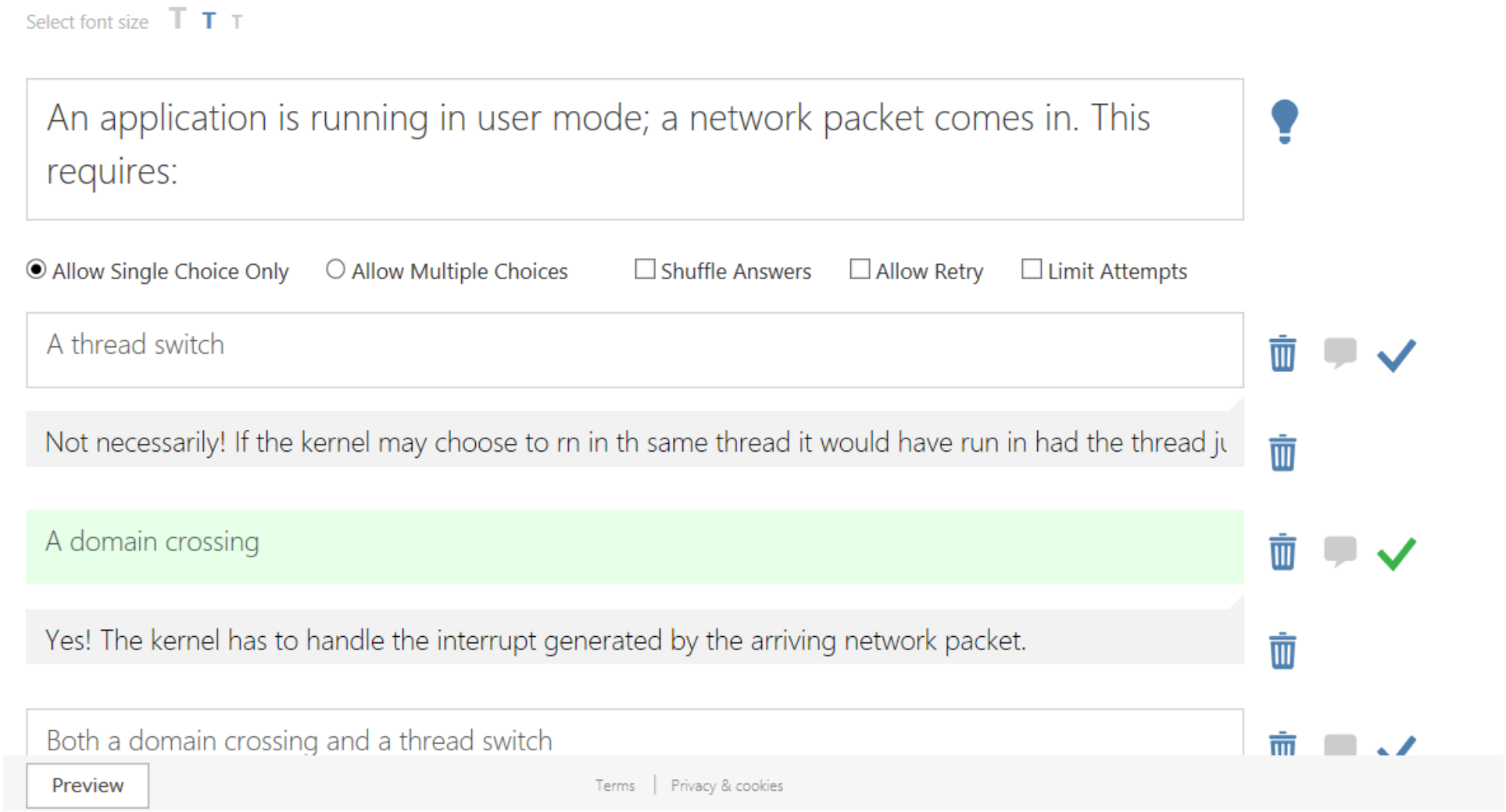
Domain Crossing



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.


Select font size
An application is running in user mode; a network packet comes in. This requires:
Allow Single Choice Only
Allow Multiple Choices
Shuffle Answers
Allow Retry
Limit Attempts
A thread switch
Not necessarily! If the kernel may choose to rn in th same thread it would have run in had the thread ju
A domain crossing
Yes! The kernel has to handle the interrupt generated by the arriving network packet.
Both a domain crossing and a thread switch
Preview
Terms
Privacy & cookies

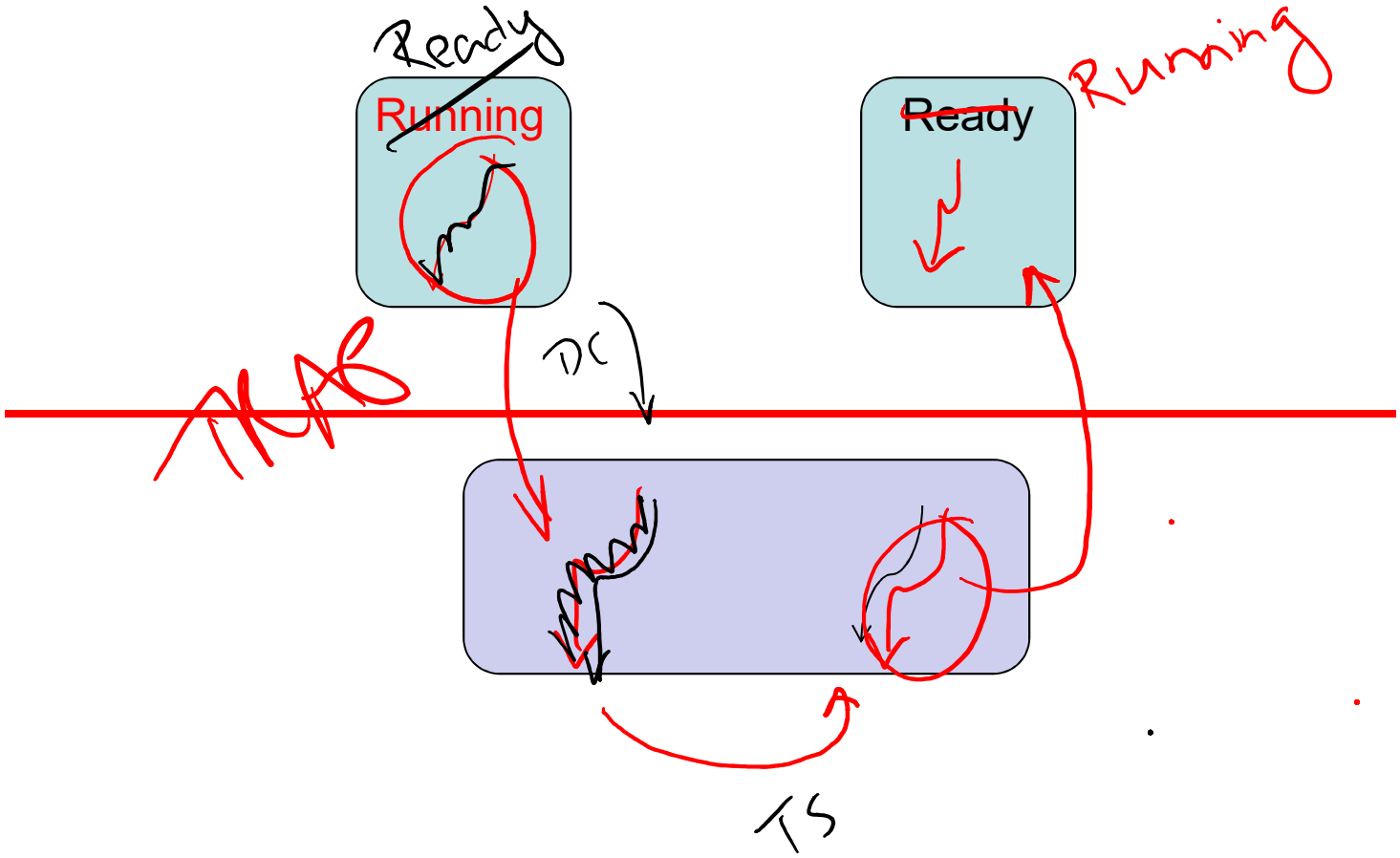
What Does the Kernel do on a Trap?

- 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 user mode, then you have to find a kernel 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



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)

Select font size T T T

A user level process calls the system call getpid; the kernel immediately schedules a different process. Is this OK?

☑ Allow Retry

True

False

Preview

Terms | Privacy & cookies

Select font size T T T

Why must the kernel run on a stack different from that of the user level process?

Preview

Terms | Privacy & cookies

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

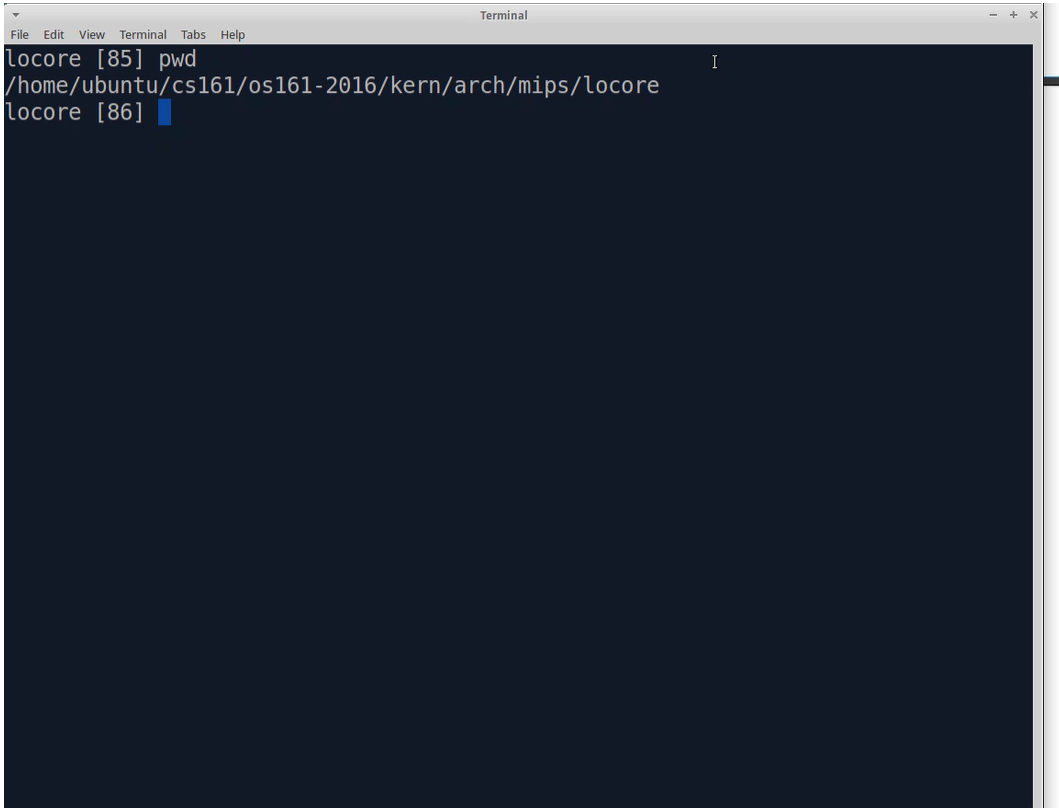
- The PC is saved into a special supervisor register.
- The status and cause registers (two other special purpose registers) are set to reflect the details of the trap being processed.
- The processor switches into kernel mode with interrupts disabled.
- *The rest is done in software*.

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).

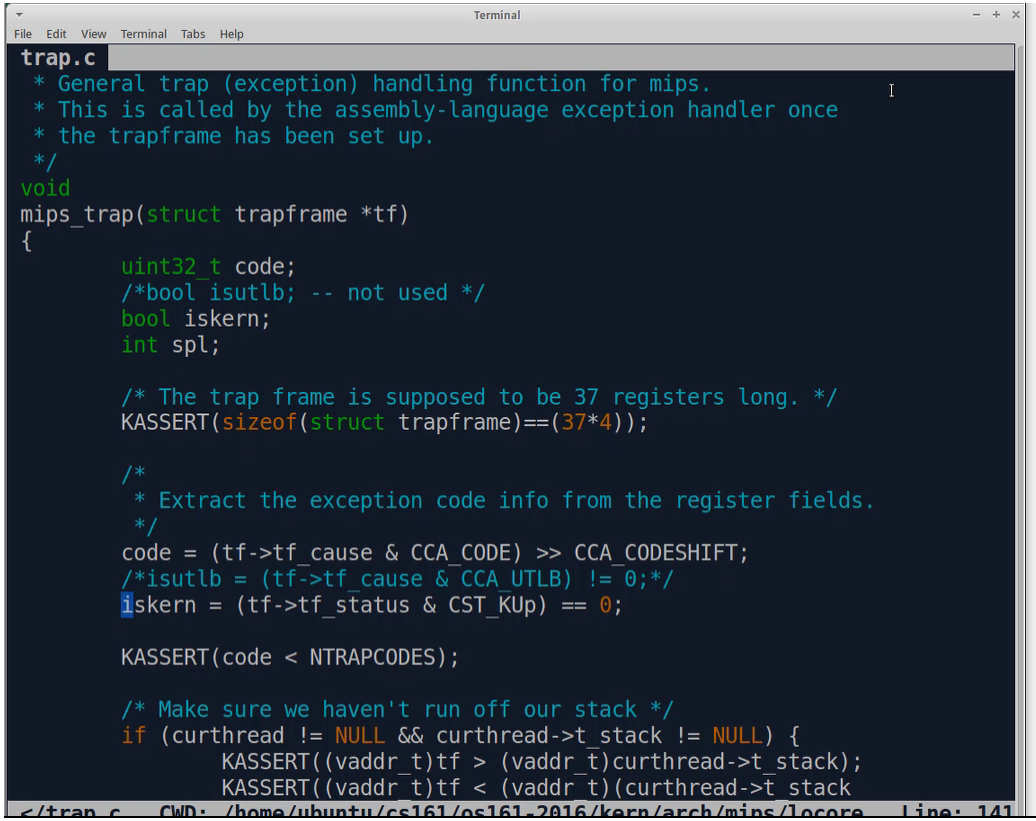


MIPS R3000 Software

- 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).

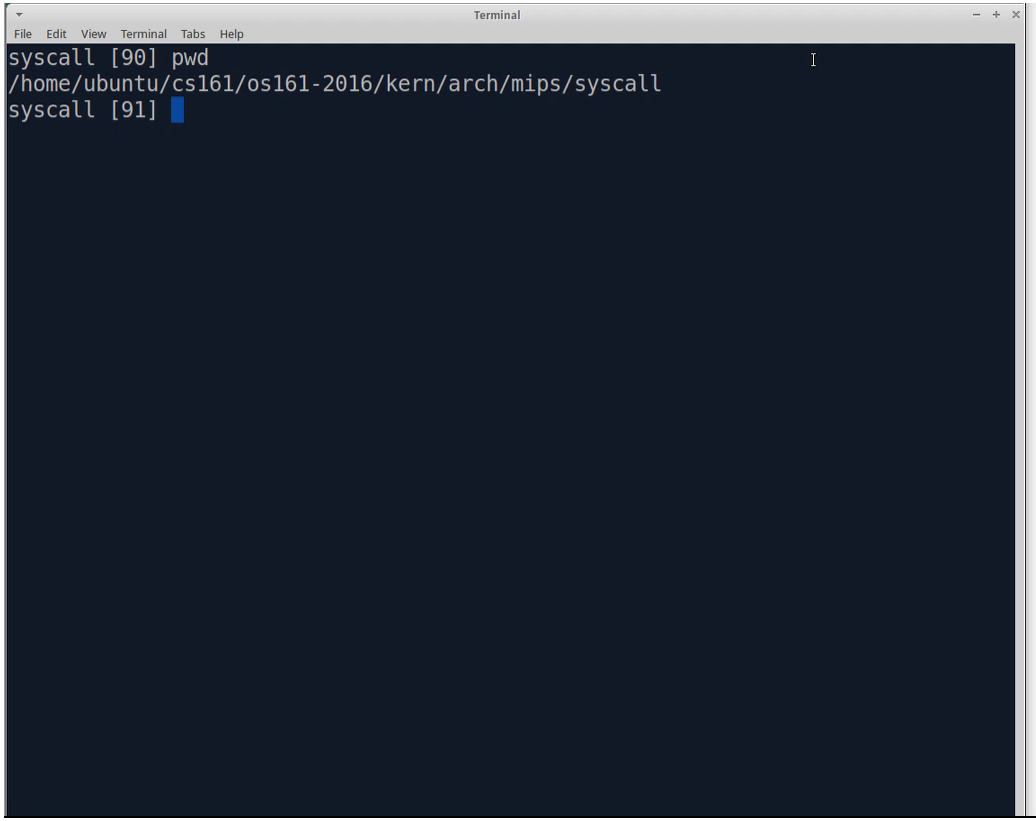
MIPS R3000 Trap Handling

- 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.



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 system call handler:
  - `kern/arch/mips/syscall/syscall.c`
    - Figure out which system call is needed and dispatch
- Both of these functions assume that all th information has been stashed away in a t

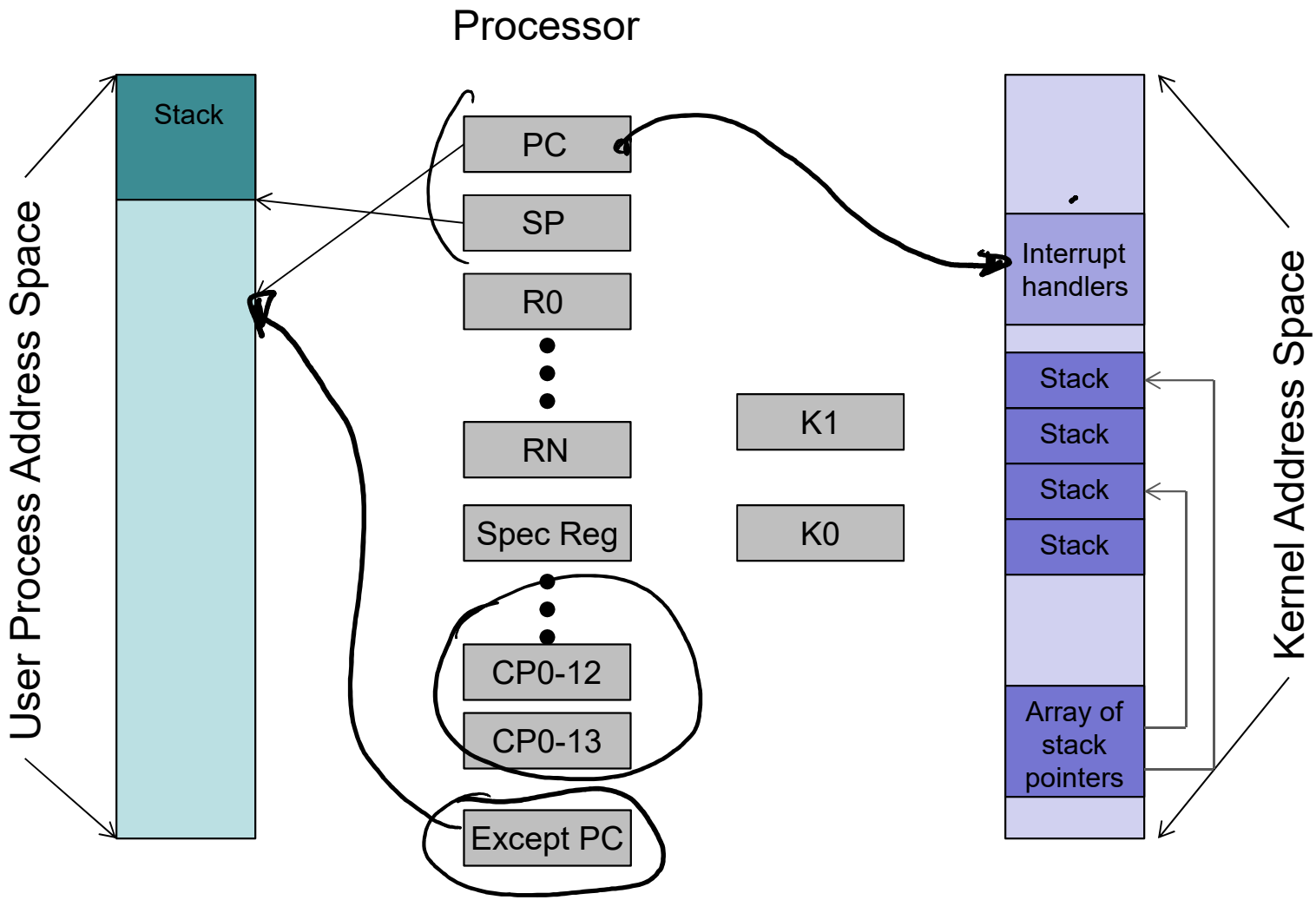


MIPS R3000 Trap Handling

- 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 system call handler:
  - `kern/arch/mips/syscall/syscall.c`
    - Figure out which system call is needed and dispatch
- Both of these functions assume that all th information has been stashed away in a t

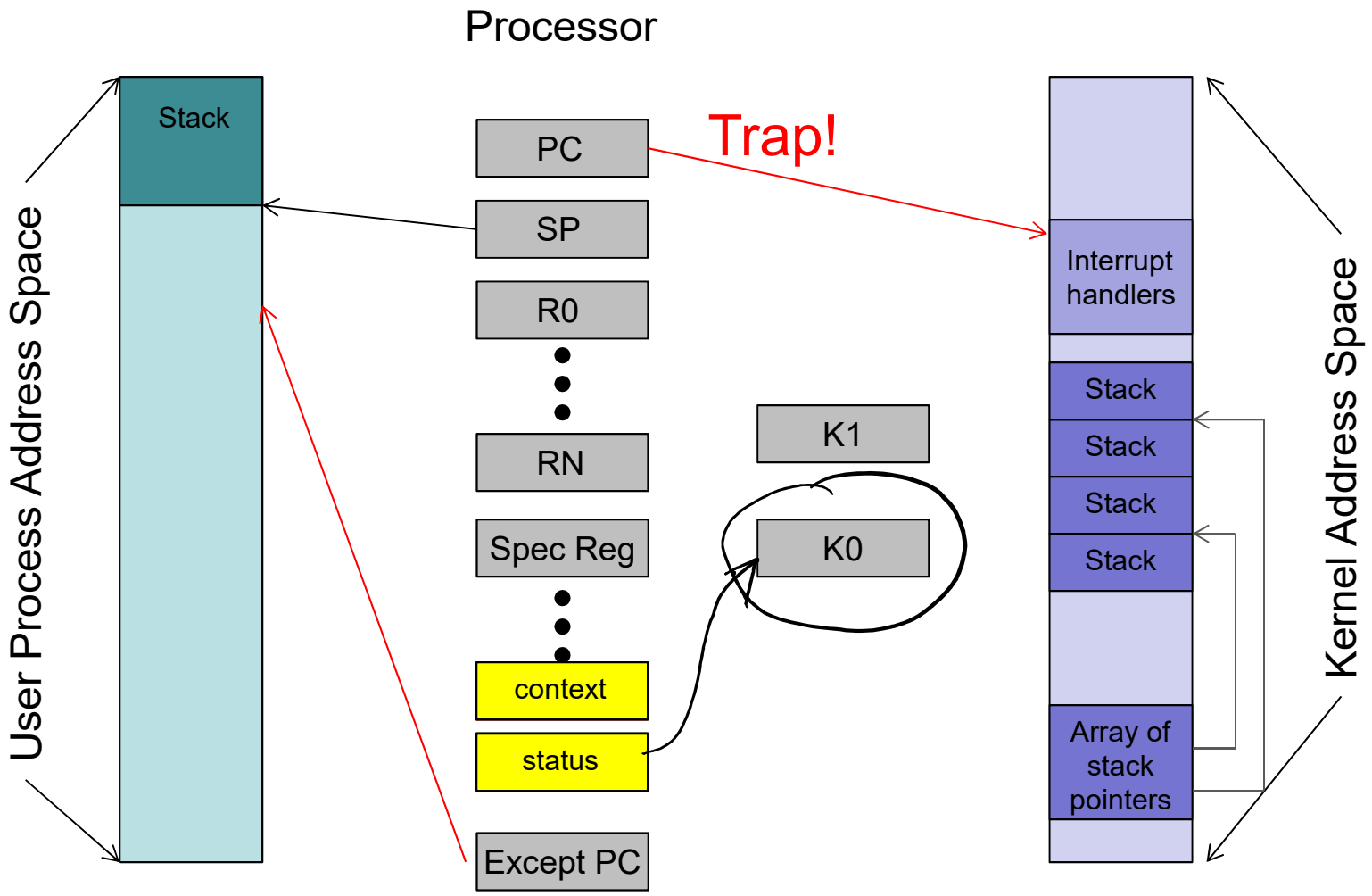
PC and SP reference addresses in user space.

Normal Execution (1)



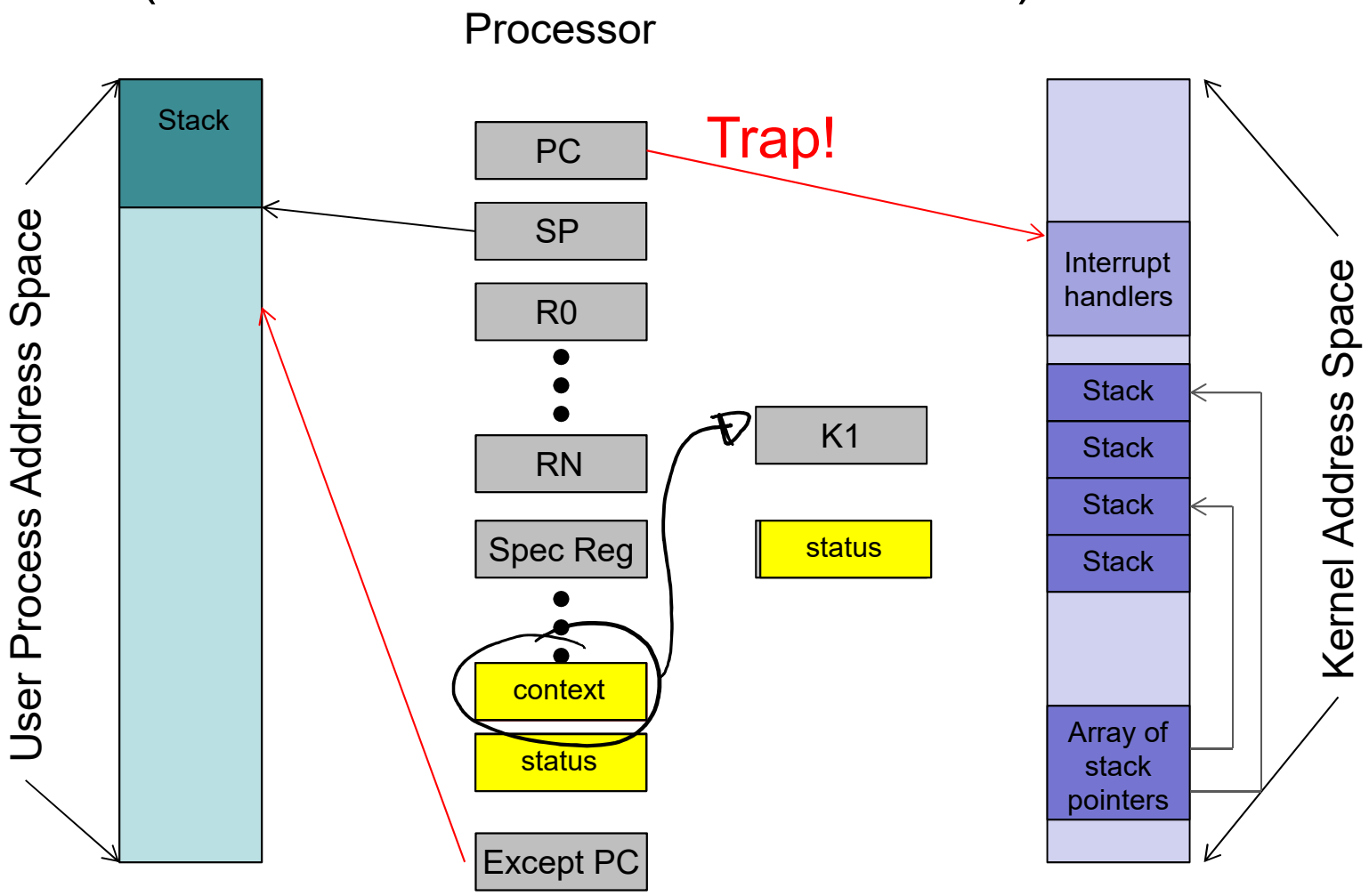
kern/arch/mips/locore/exception-mips1.S(2) line 105-107

SW:
Save status register to k0
Check mode



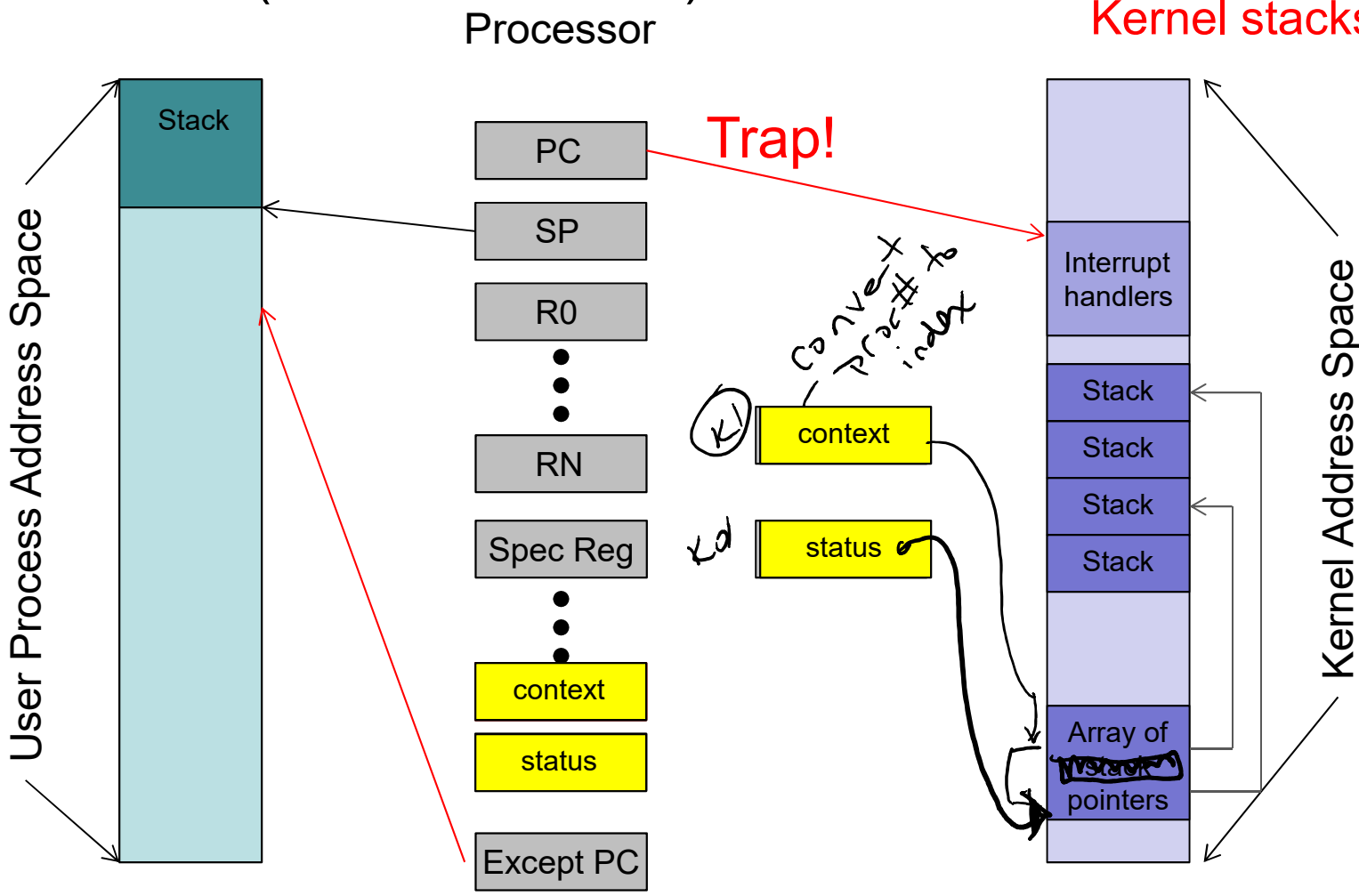
SW:
Save context register into K1

kern/arch/mips/locore/exception-mips1.S(3) line 111 (assume we came from user mode)



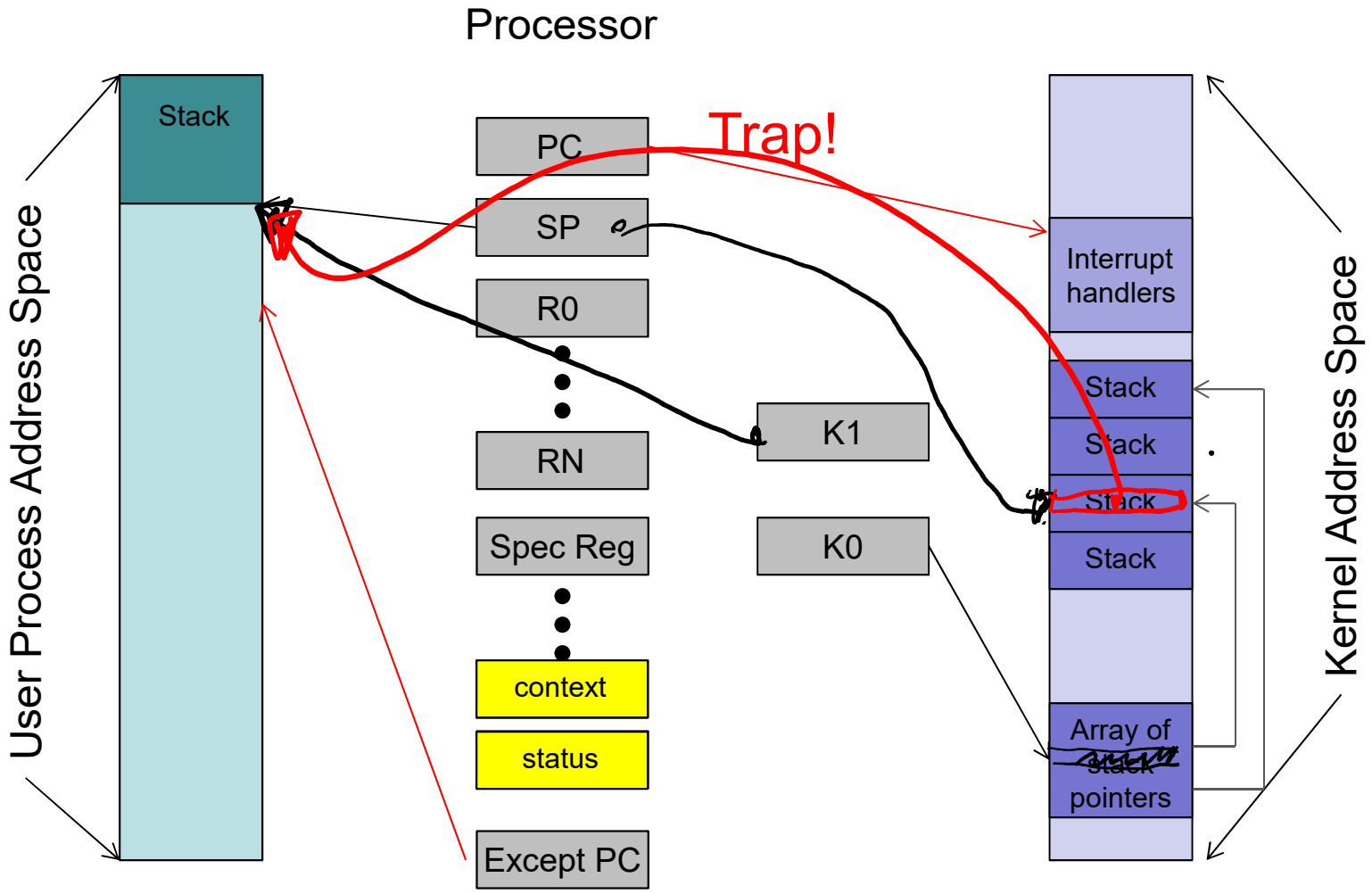
kern/arch/mips/locore/exception-mips1.S(4) line 112-115 (find kernel stack)

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



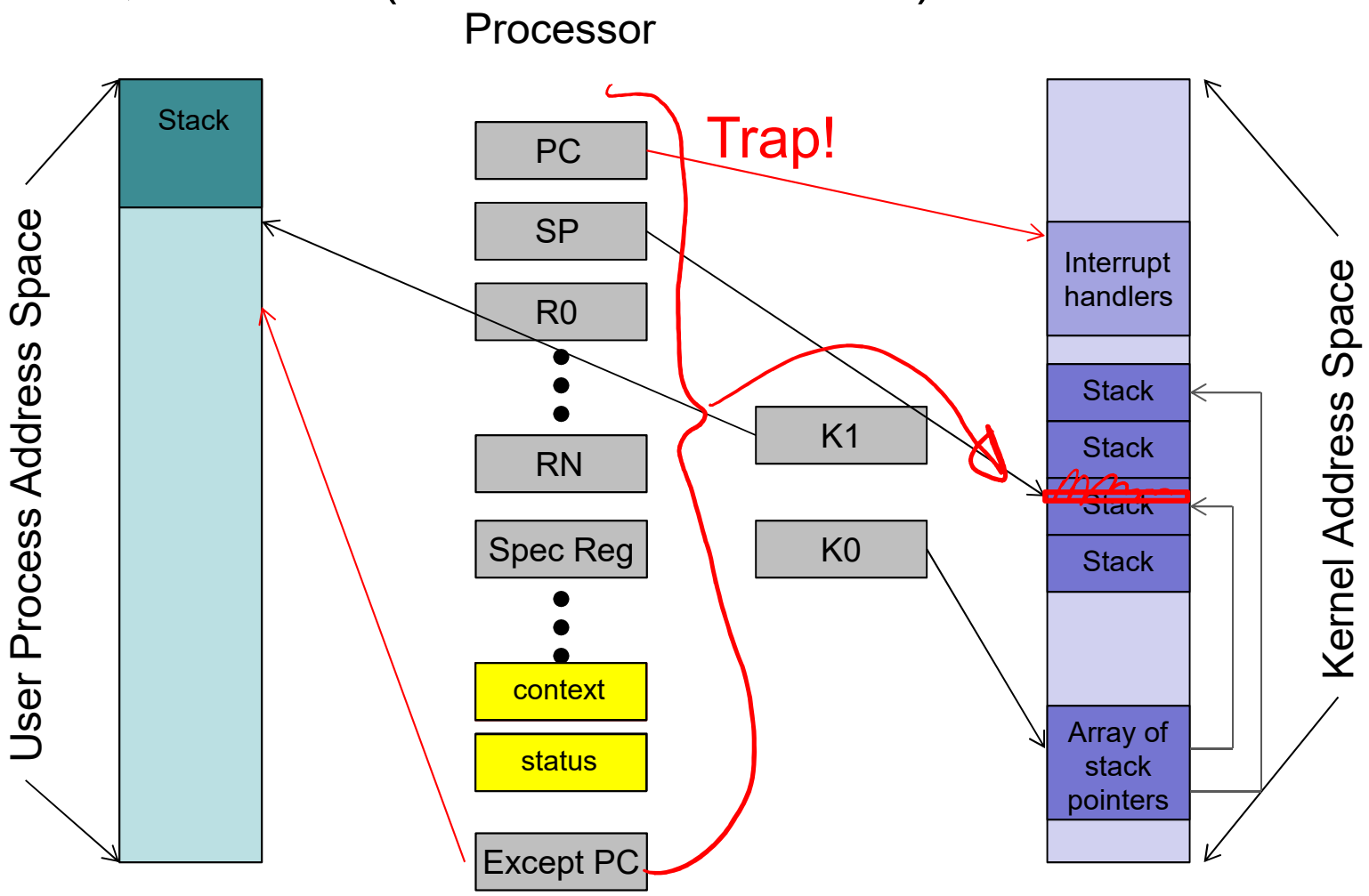
SW:
Save old SP in K1

kern/arch/mips/locore/exception-mips1.S(5) line 116 (save user SP)



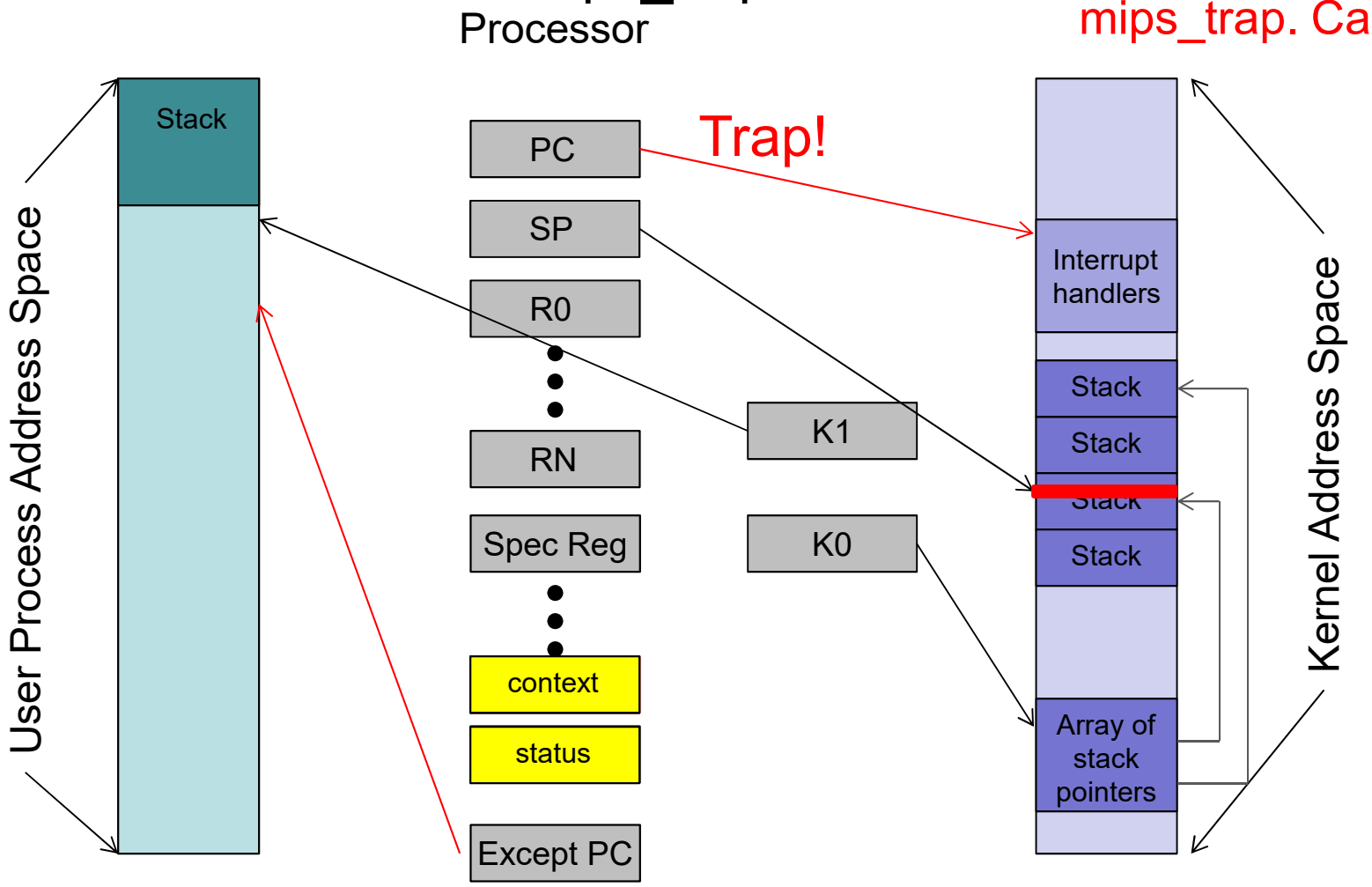
SW:
Allocate trap stack frame (bump SP)
Save registers

kern/arch/mips/locore/exception-mips1.S(6) line 137, 170-233 (create a stack frame)



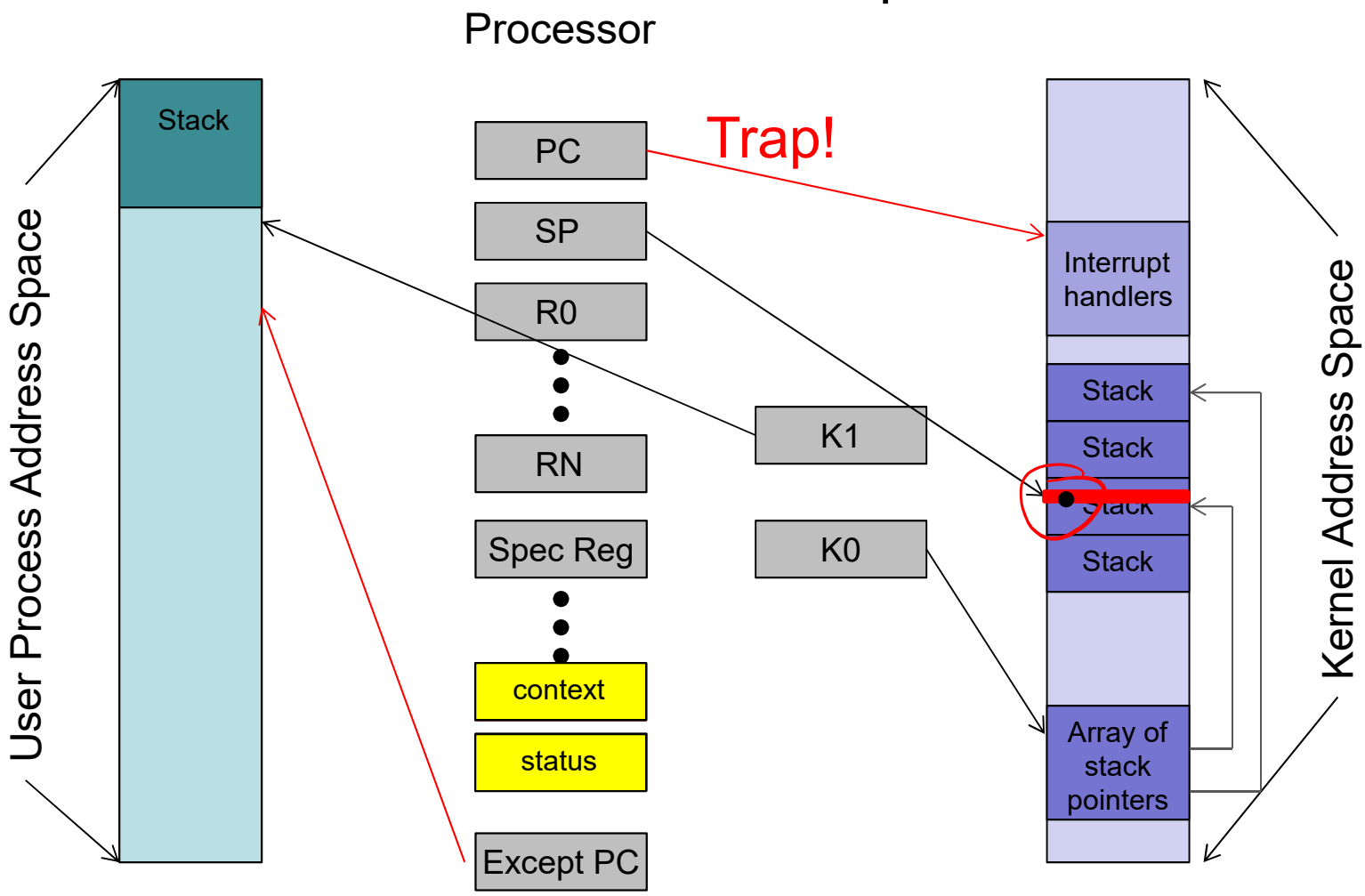
SW:
Trapframe on the stack is the argument to mips_trap. Call it.

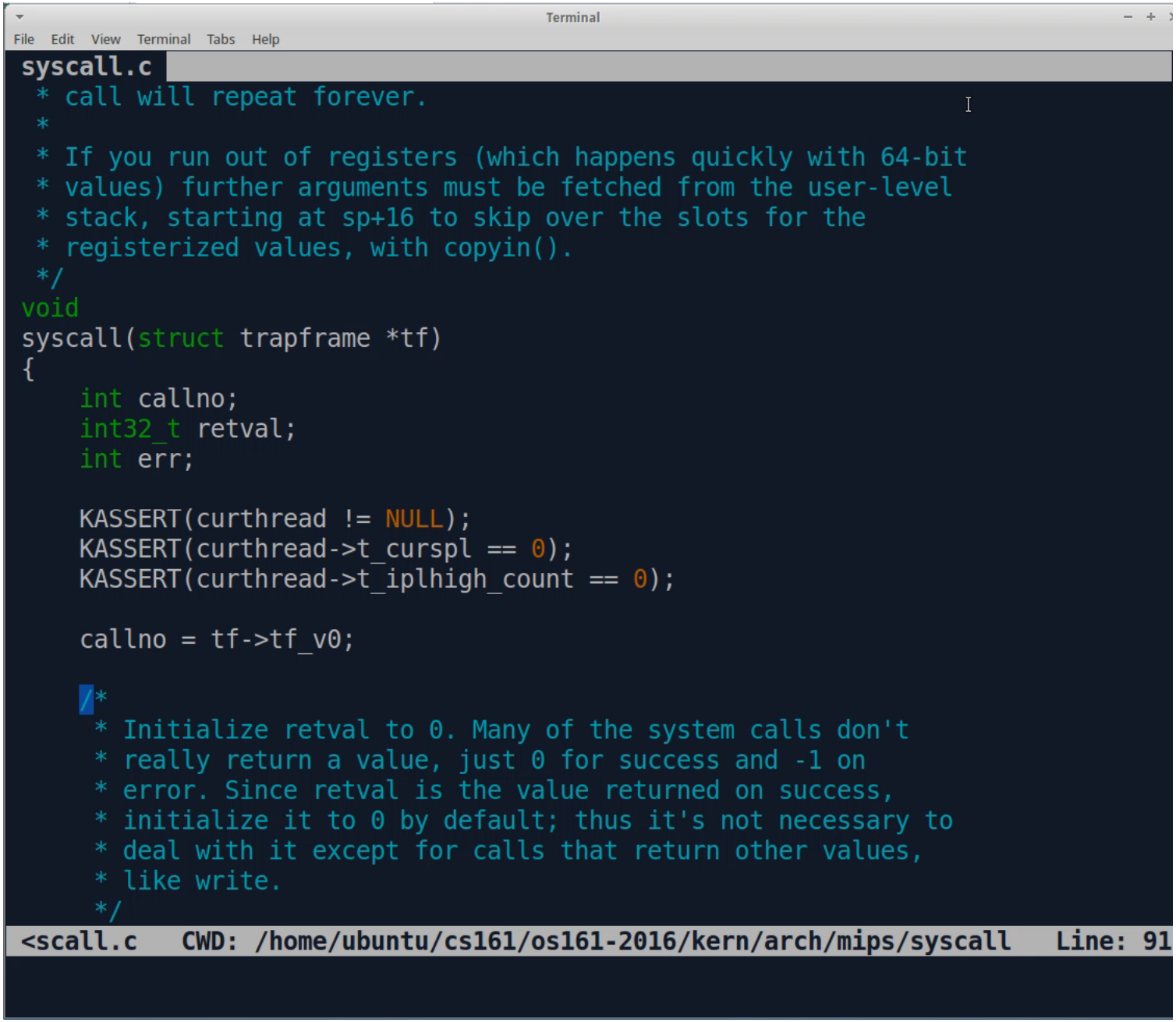
kern/arch/mips/locore/trap.c(7)
line 125: We called into mips_trap



SW:
Extract exception type from the trap frame.

kern/arch/mips/locore/trap.c(8)
line 431-433: Call routine for this exception




Terminal
File Edit View Terminal Tabs Help
syscall.c
 * call will repeat forever.
 *
 * If you run out of registers (which happens quickly with 64-bit
 * values) further arguments must be fetched from the user-level
 * stack, starting at sp+16 to skip over the slots for the
 * registerized values, with copyin().
 */
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);

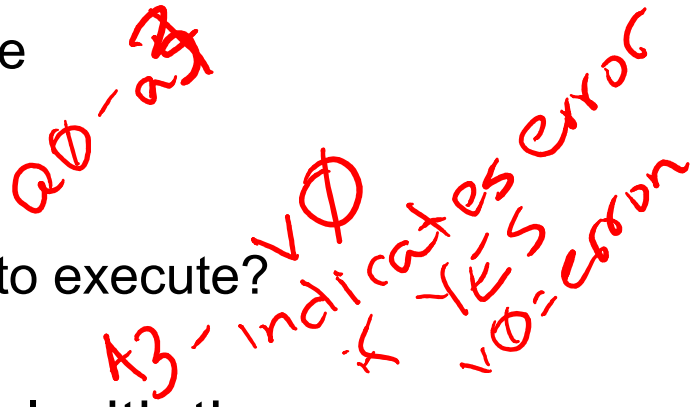
    callno = tf->tf_v0;

    /*
     * Initialize retval to 0. Many of the system calls don't
     * really return a value, just 0 for success and -1 on
     * error. Since retval is the value returned on success,
     * initialize it to 0 by default; thus it's not necessary to
     * deal with it except for calls that return other values,
     * like write.
     */
<scall.c CWD: /home/ubuntu/cs161/os161-2016/kern/arch/mips/syscall Line: 91

Given what you now know about how the kernel sets up return values from system calls, what must the library code that invokes the trap for the system call do upon return?

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?



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 using `fork`. But how do we create an initial user process? And what might that process be?
- On Unix™ systems, the first process is called `init`.
- The kernel hand crafts this process during startup.
  - If you took CS61, think about the `process_setup` function in Weensy OS (you might even find reviewing it useful).
  - In os161, you will find it useful to read and understand `kern/proc/proc.c`.
- All other processes are descendants of `init`.

Implementing Fork

- Things to think about when implementing `fork`:
  - The forking process must **not** be running at user-level
    - That is, in a multi-threaded process, no other threads may be active. (Why?)
  - Before copying a process, you need to save its state!
  - Copying a process requires copies of the code and the data (including the stack).
  - The processor state for the copy must be identical to that of the parent (with one exception). Where do you find the parent's processor state?
  - What is the one difference between the original (parent) and new (child) process?
  - Finally, make sure that your dispatcher/scheduler knows about this new process.

Fork's Friends

- Exec:
  - Replace the current process with a new one.
  - This involves changing the contents of the address space and then starting execution at the "right" place. Where is that place?
- Wait:
  - Parent must be able to wait on one child or any child.
  - Child state must stick around until it has been reaped (i.e., until someone has waited on it).
  - Parent must get child's exit status.
- Pipes and file descriptors:
  - On a fork, which parts of the file descriptor table (and the things it references) are shared and which are copied?
  - What happens on exec?