

# Context Switching

- Topics
  - What is a context switch?
  - What hardware support is necessary?
  - How does a context switch work on the MIPS?
- Learning Objectives:
  - Be prepared to tackle Assignment 2!

# Changing Protection Domains and Threads of Execution

- **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 and in supervisor mode.
- **Change protection domain:** 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.
- **Context switch:** the process of storing and restoring the execution state (i.e., context) so that multiple processes (or threads) can share a processor. Composed of:
  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. Execute
  5. Restore execution state
  6. Change protection domain (from supervisor[kernel] to user)

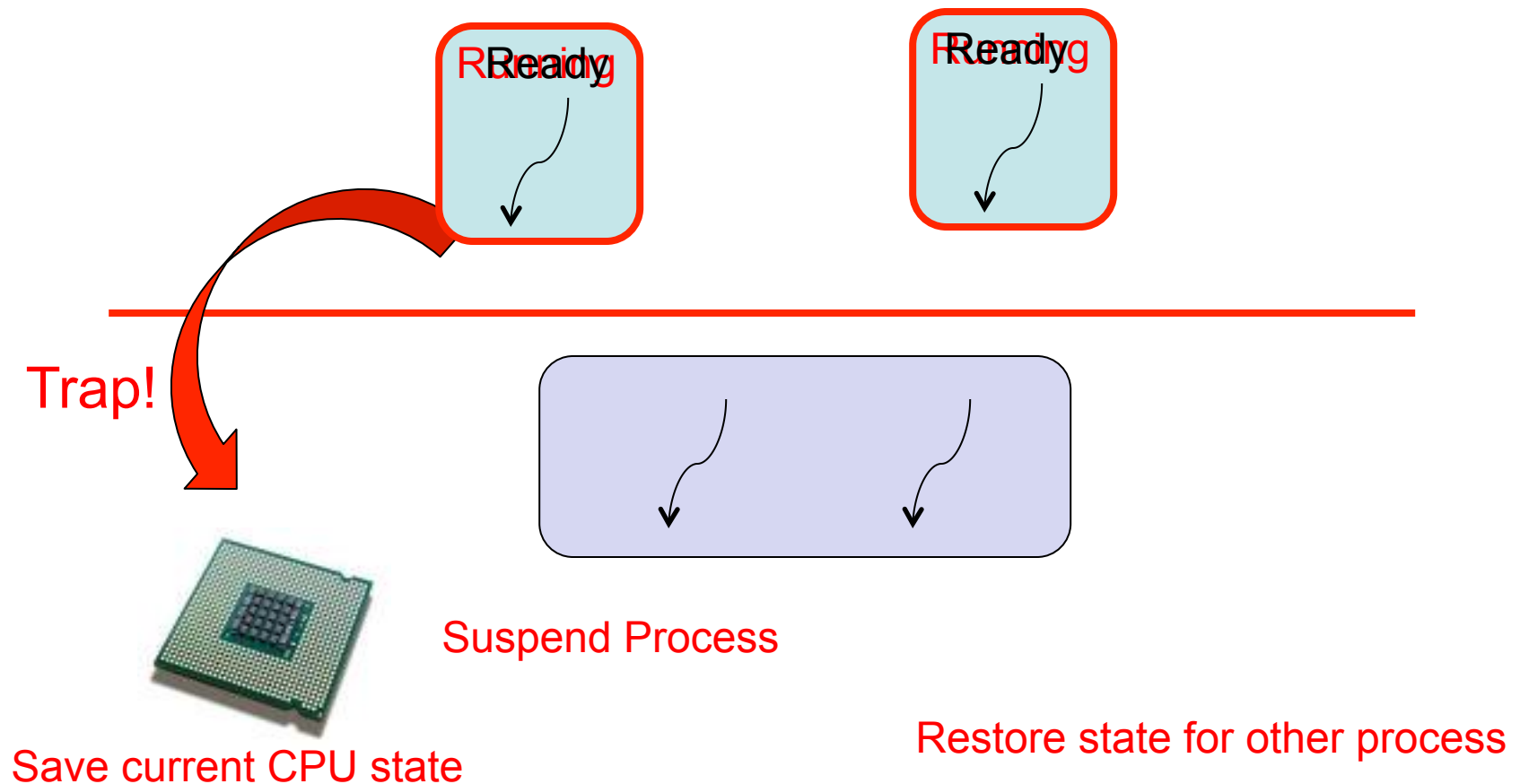
# System Calls

- Without a rescheduling event:
  - Context switch (step 4 is the execution of the kernel code that implements the system call).
- With a rescheduling event:
  - Whenever the OS is invoked, it has the option of running a thread different from the one running immediately before the OS took over.
  - This requires a context switch as in the previous case.
  - It also requires a thread switch:
    - Step 5: rather than restore the state from the original thread/process, we can restore state from some other thread/process.
    - After the protection domain change in step 6, the processor is now executing a different process.

# Interrupt

- Let's assume process P is running in user mode.
- Interrupt happens:
  - Domain crossing into supervisor mode
  - Switch to kernel stack
  - Store execution state of user thread
  - Handle interrupt
  - Restore state of some process
    - If interrupt was a timer for the scheduler, probably restore a different thread/process.
    - If interrupt unblocks a high priority thread/process, probably restore a different thread/process.
    - Else, might restore same thread/process.

# Context Switching w/Change of Process



# Context Switching (implementation)

- There are two basic approaches to implementing context switching
  - Software: carefully coded assembler
  - Hardware: fully implemented in the processor
- Example MIPS (System/161):
  - 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.*

# Intel Context Switch

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

# MIPS R3000 Context Switch (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 exception was generated
  - 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 exception)
- Sets the PC to the address of the appropriate handler (for the specific interrupt type).



# MIPS R3000 Context Switch (Software)

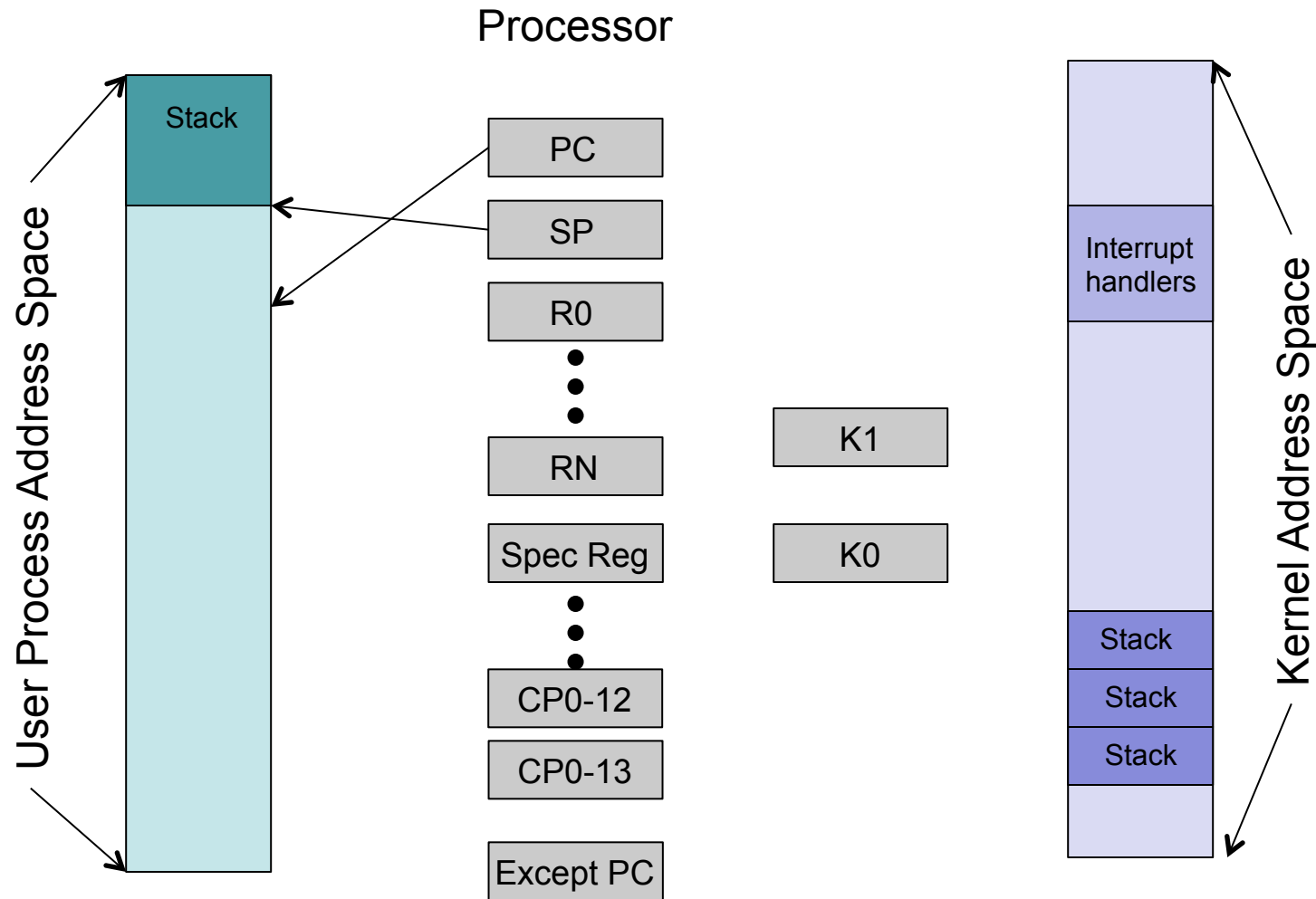
- Save whatever other state that must be saved
- Since you need to be able to save the user registers and you need to manipulate various entries, there are two registers that the kernel is allowed to use in whatever way is necessary (without this, you couldn't do anything).
- In assembly (`kern/arch/mips/locore/exception-mips1.S`)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

# MIPS R3000 Trap Handling

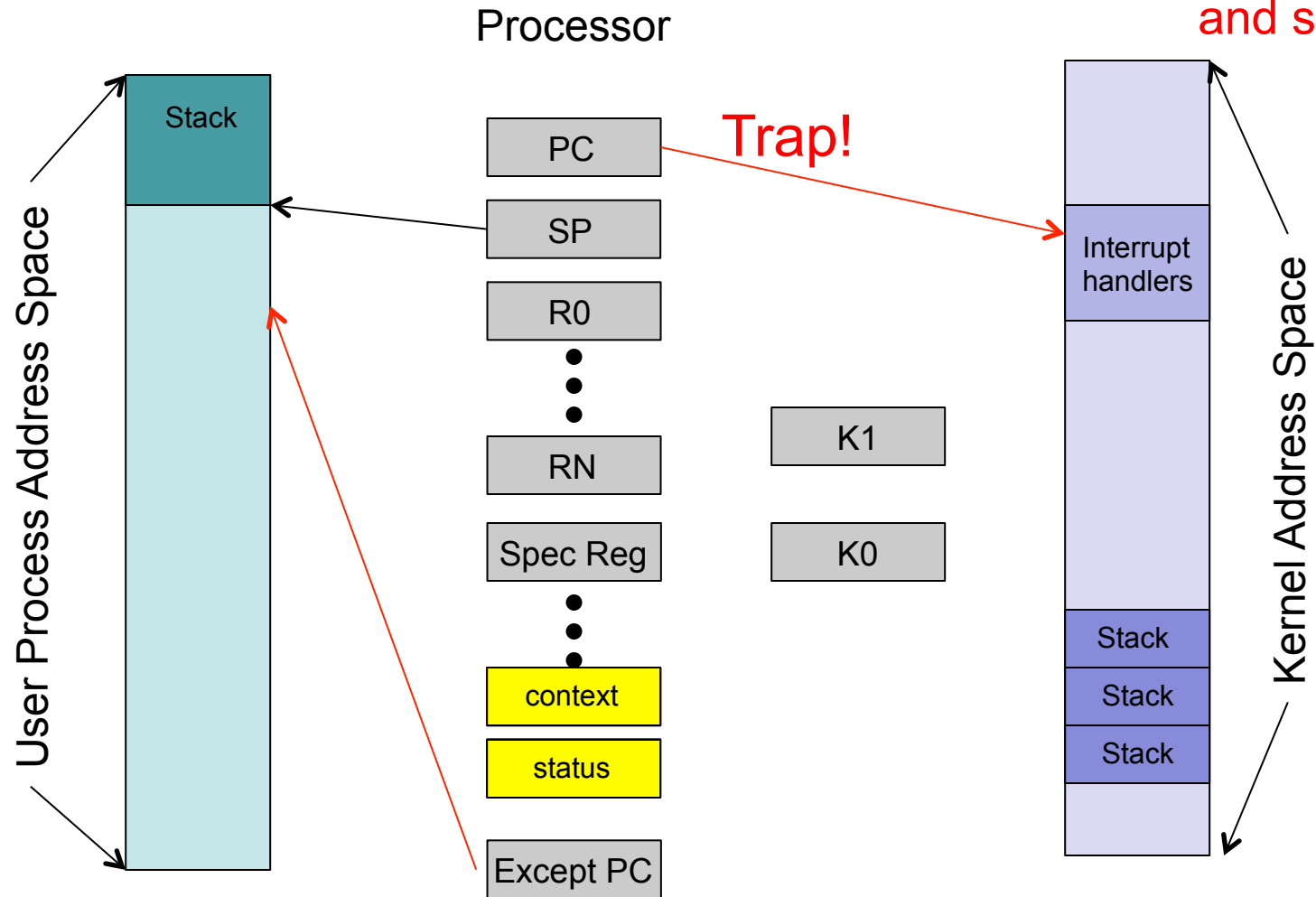
- First we go to a generic trap handler:
  - `kern/arch/mips/locore/trap.c`
- Then, if this is a system call, we go to the system call handler:
  - `kern/arch/mips/syscall/syscall.c`
- Both of these functions assume that all the important information has been stashed away in a trapframe.
- Trap.c:
  - 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.
- Syscall.c:
  - Figure out which system call is needed and dispatch to it.

# Normal Execution (1)

PC and SP  
reference  
addresses in  
user space.



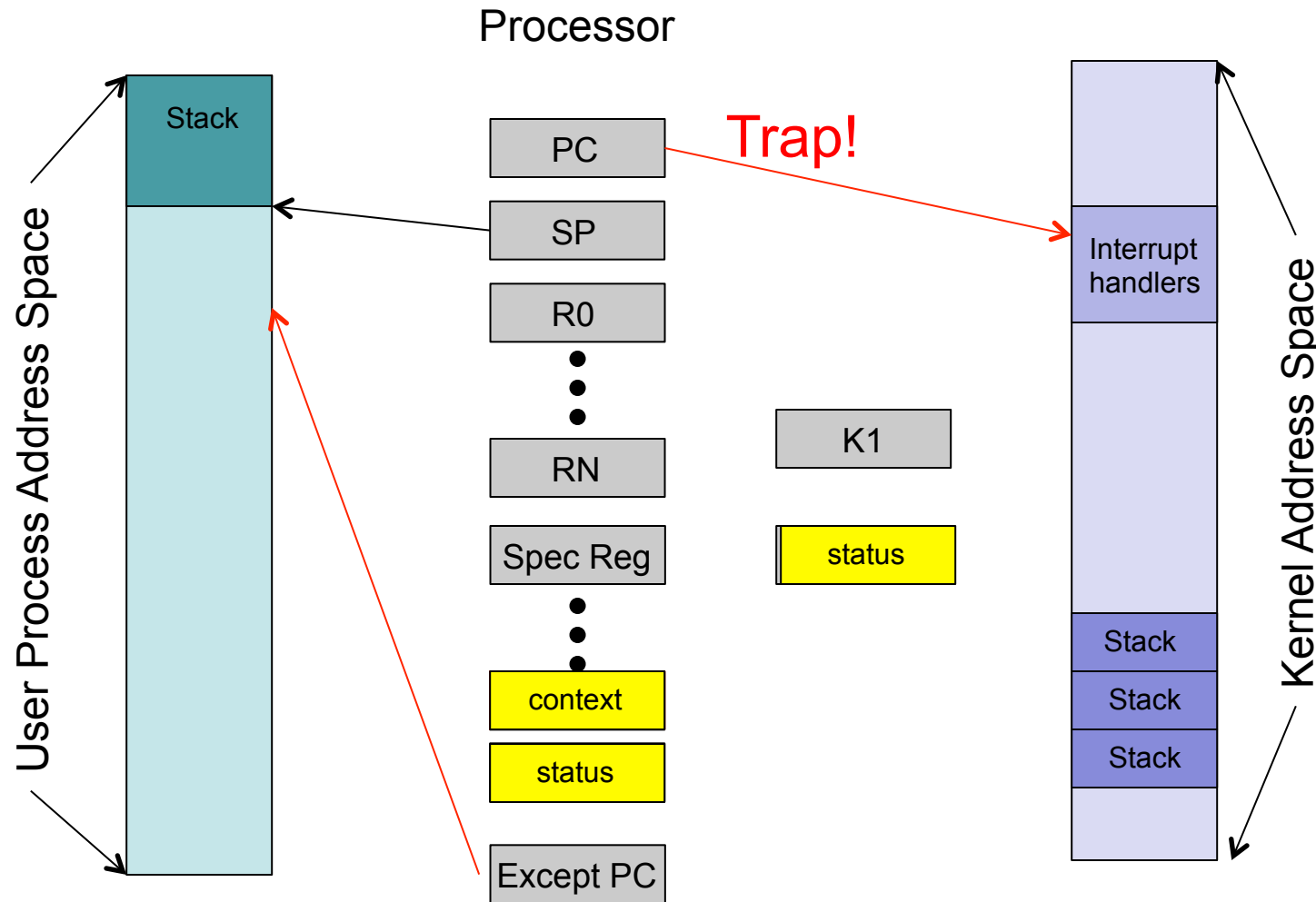
# Trap Happens (2)



HW sets:  
Exception PC  
CPU0-12/13  
get context  
and status

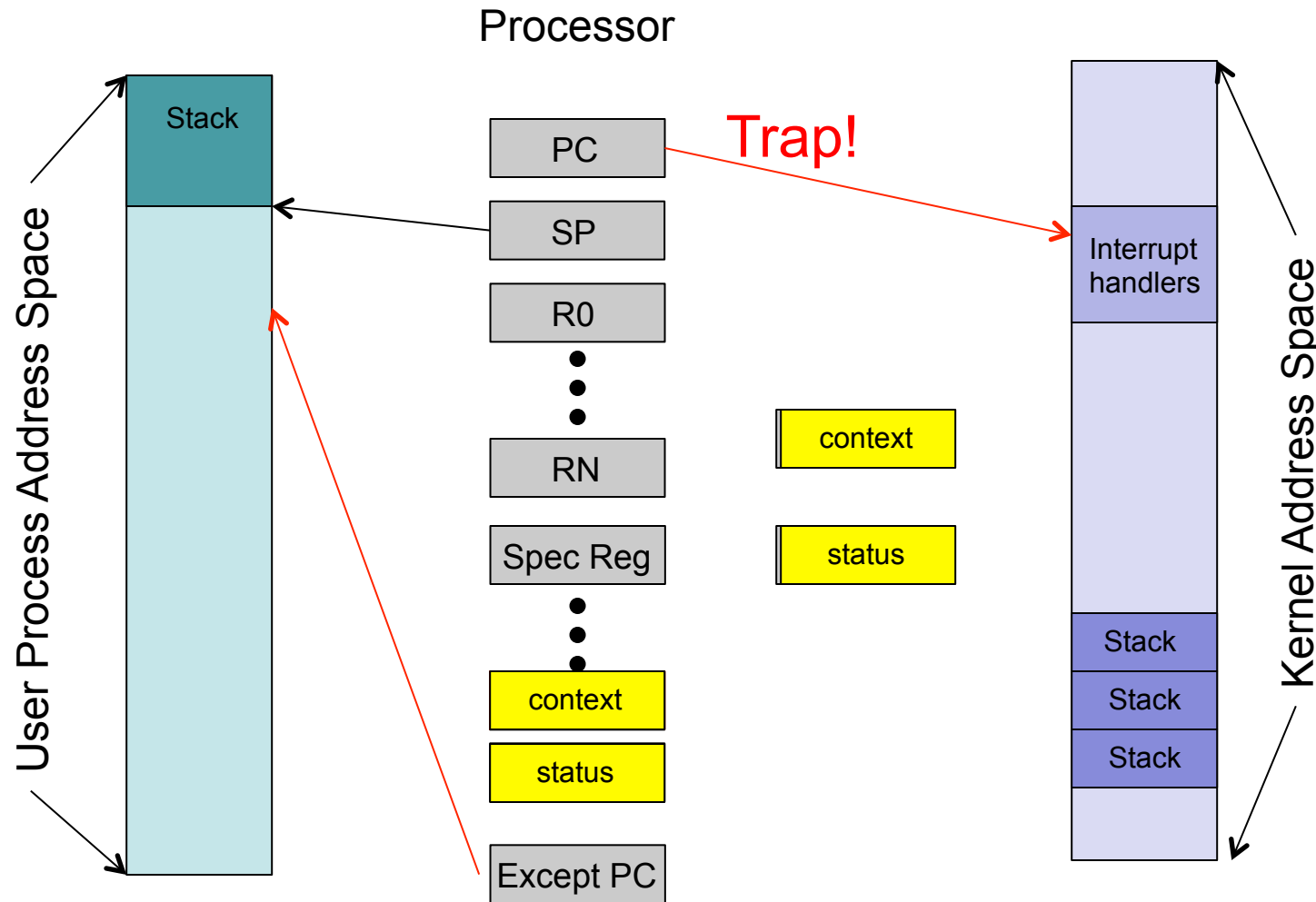
kern/arch/mips/locore/exception-mips1.S(3)  
line 105-107

SW:  
Save status  
register to k0  
Check mode



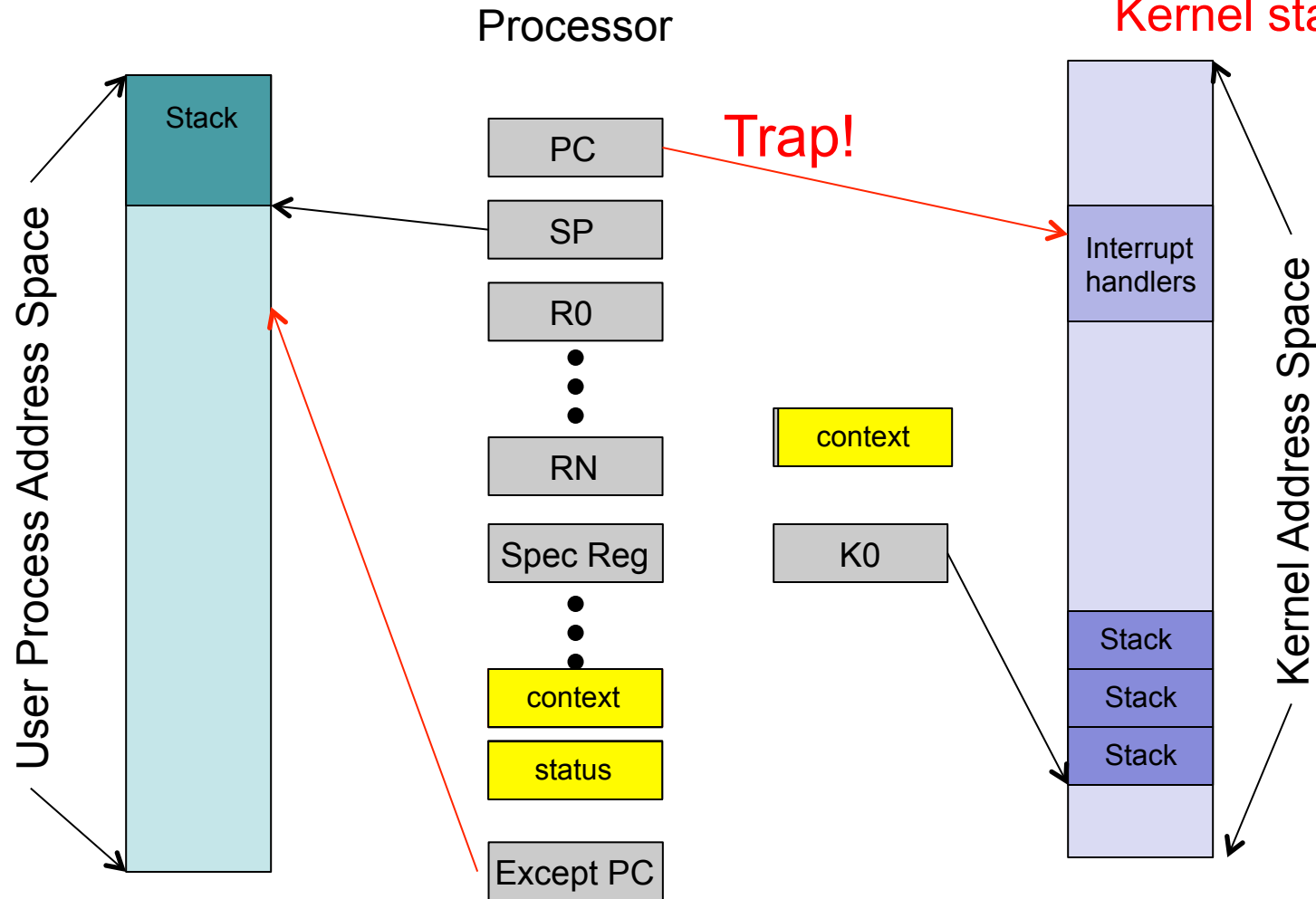
kern/arch/mips/locore/exception-mips1.S(4)  
line 111 (assume we came from user mode)

SW:  
Save context  
register into K1

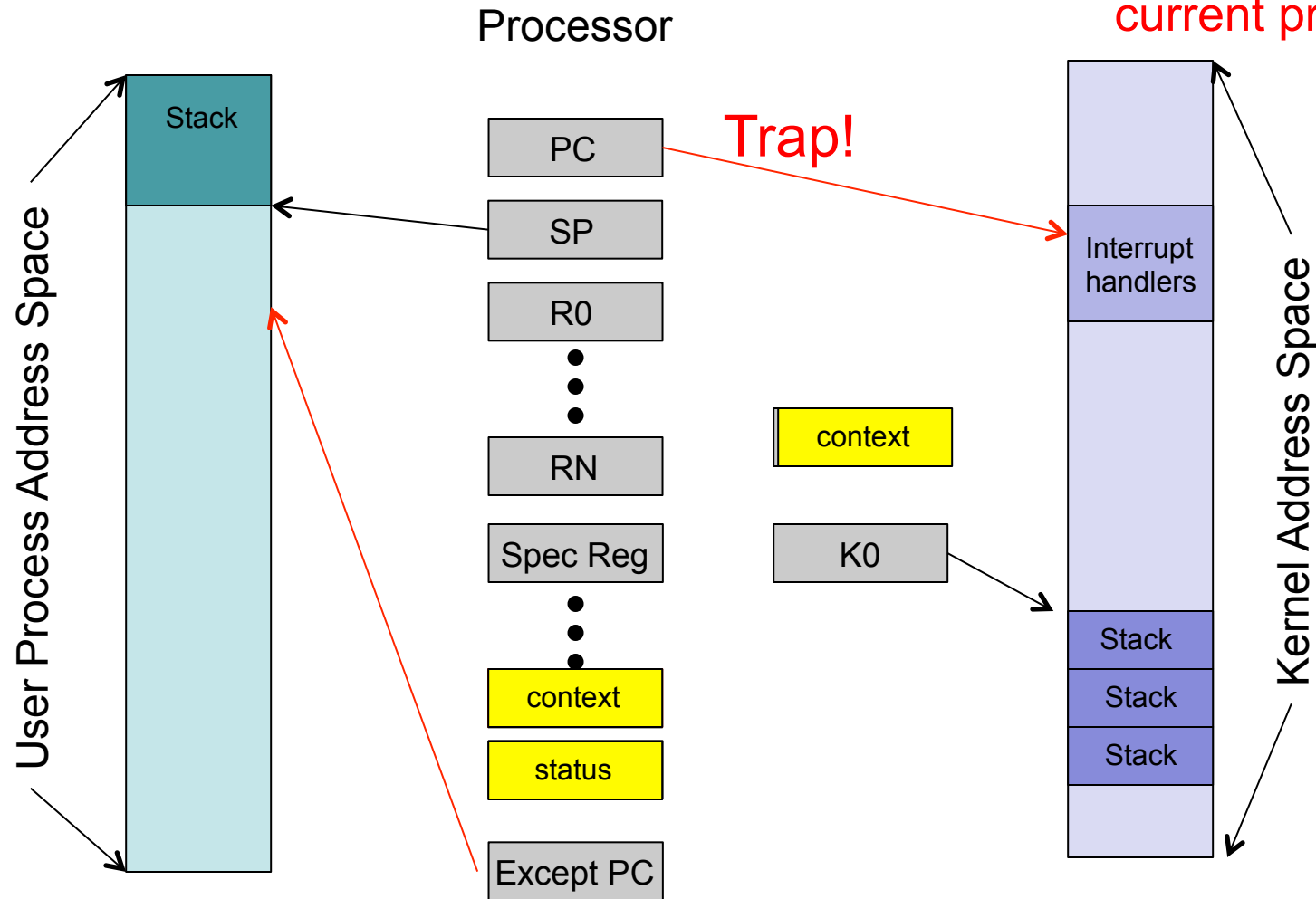


kern/arch/mips/locore/exception-mips1.S(5)  
line 112-114 (find kernel stack)

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)

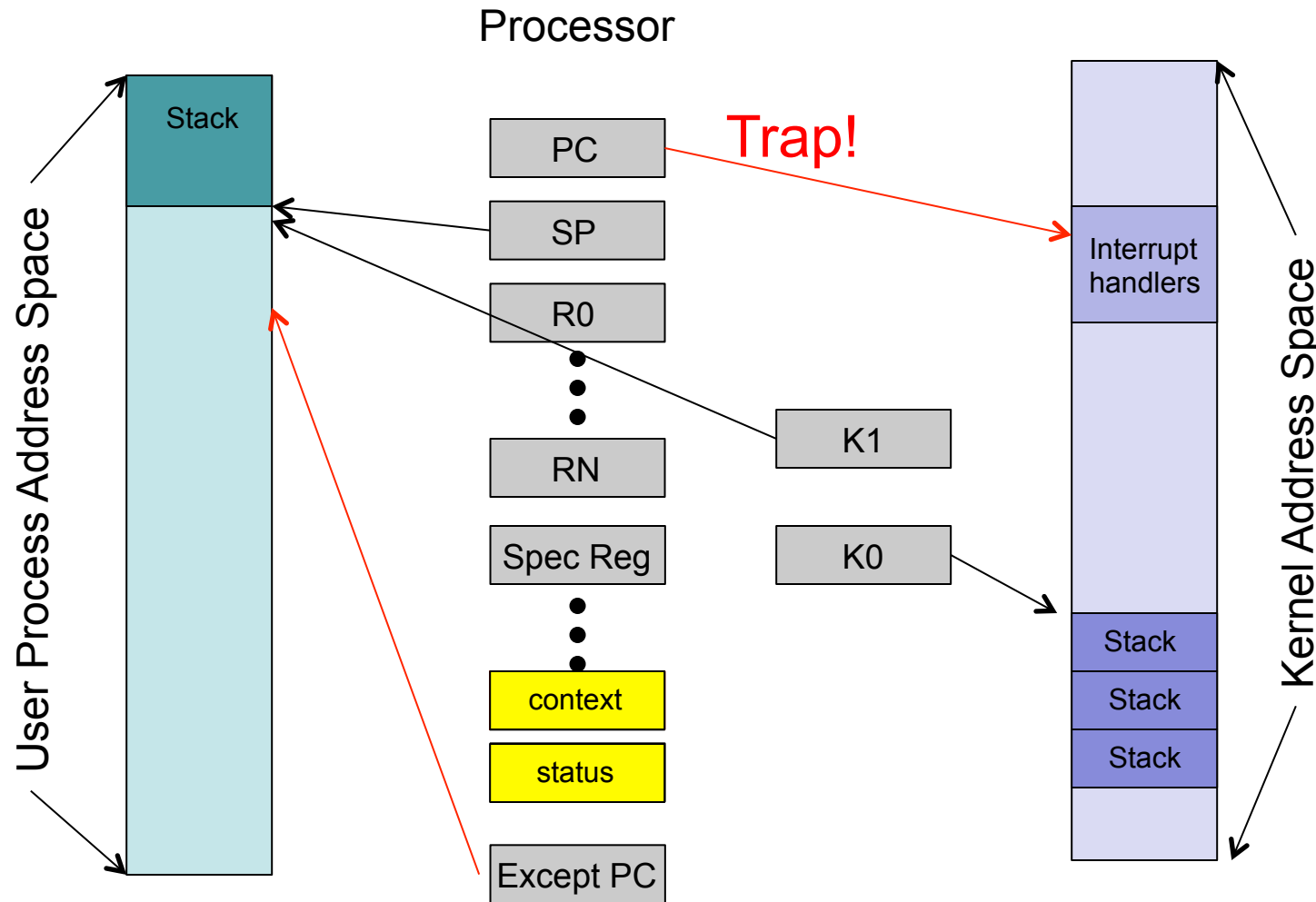


SW:  
Add K1 to K0 so  
that K0 points to  
the stack for the  
current processor



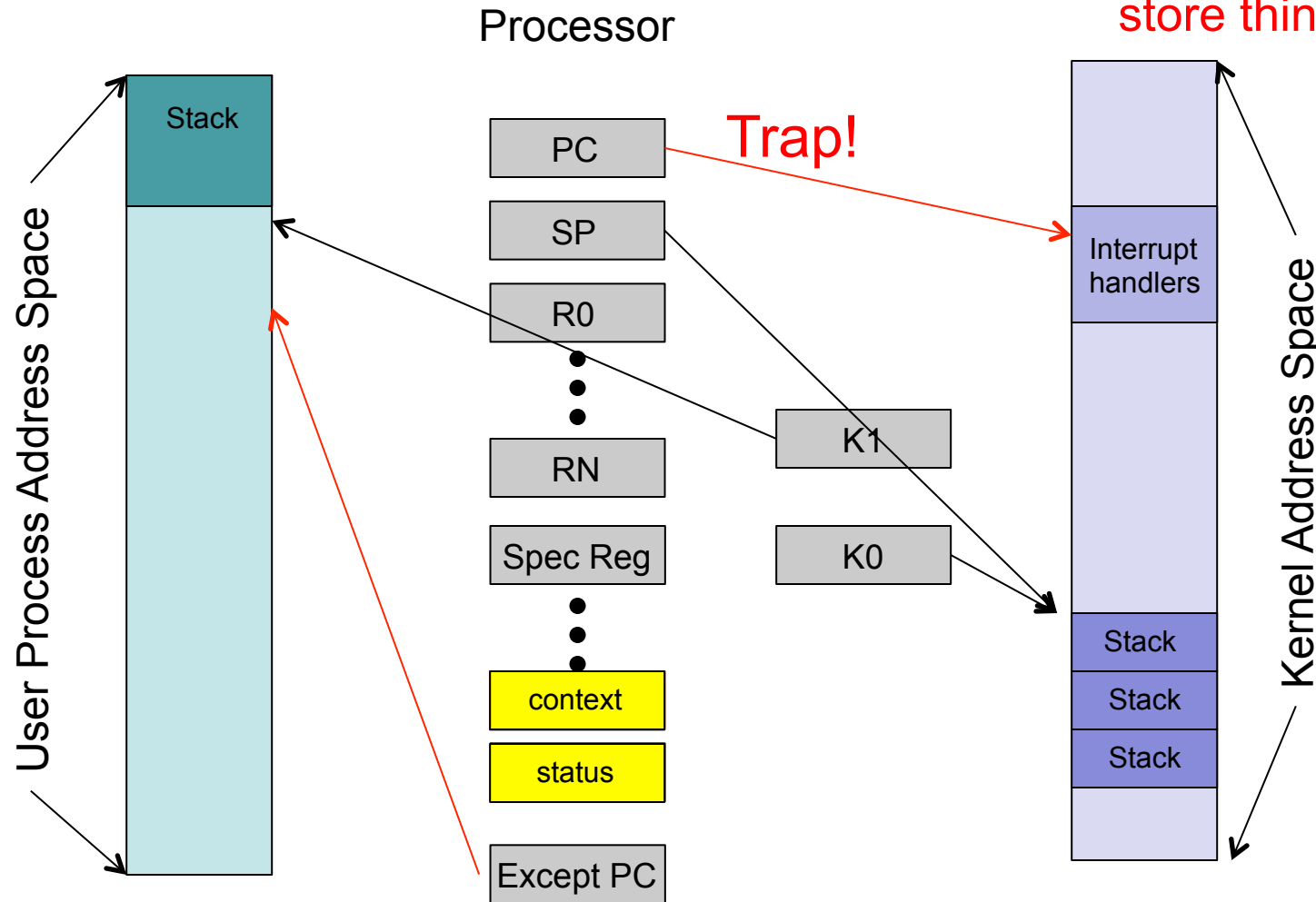
SW:  
Save old SP in K1

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



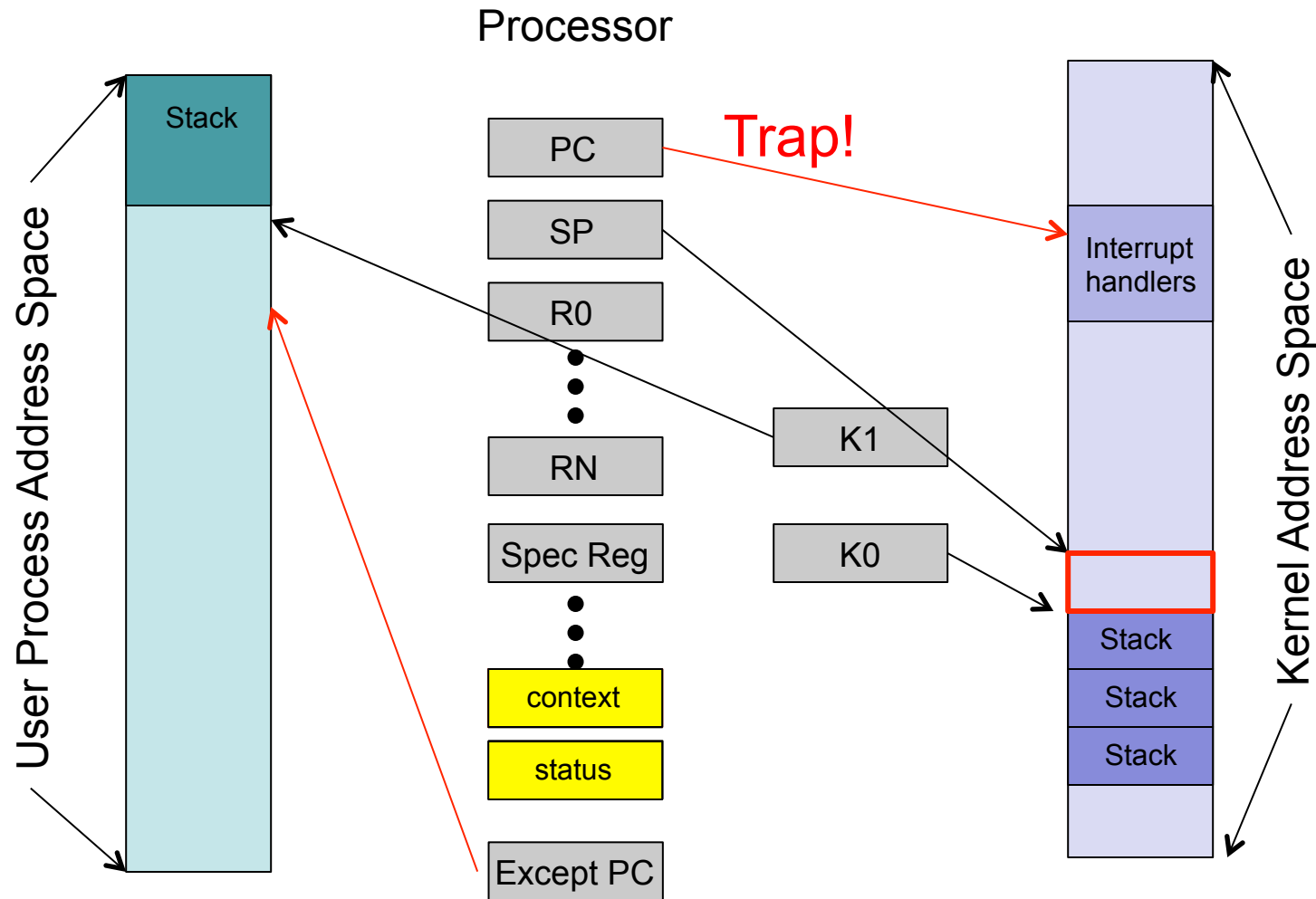
kern/arch/mips/locore/exception-mips1.S(8)  
line 117 (set SP to kernel stack)

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



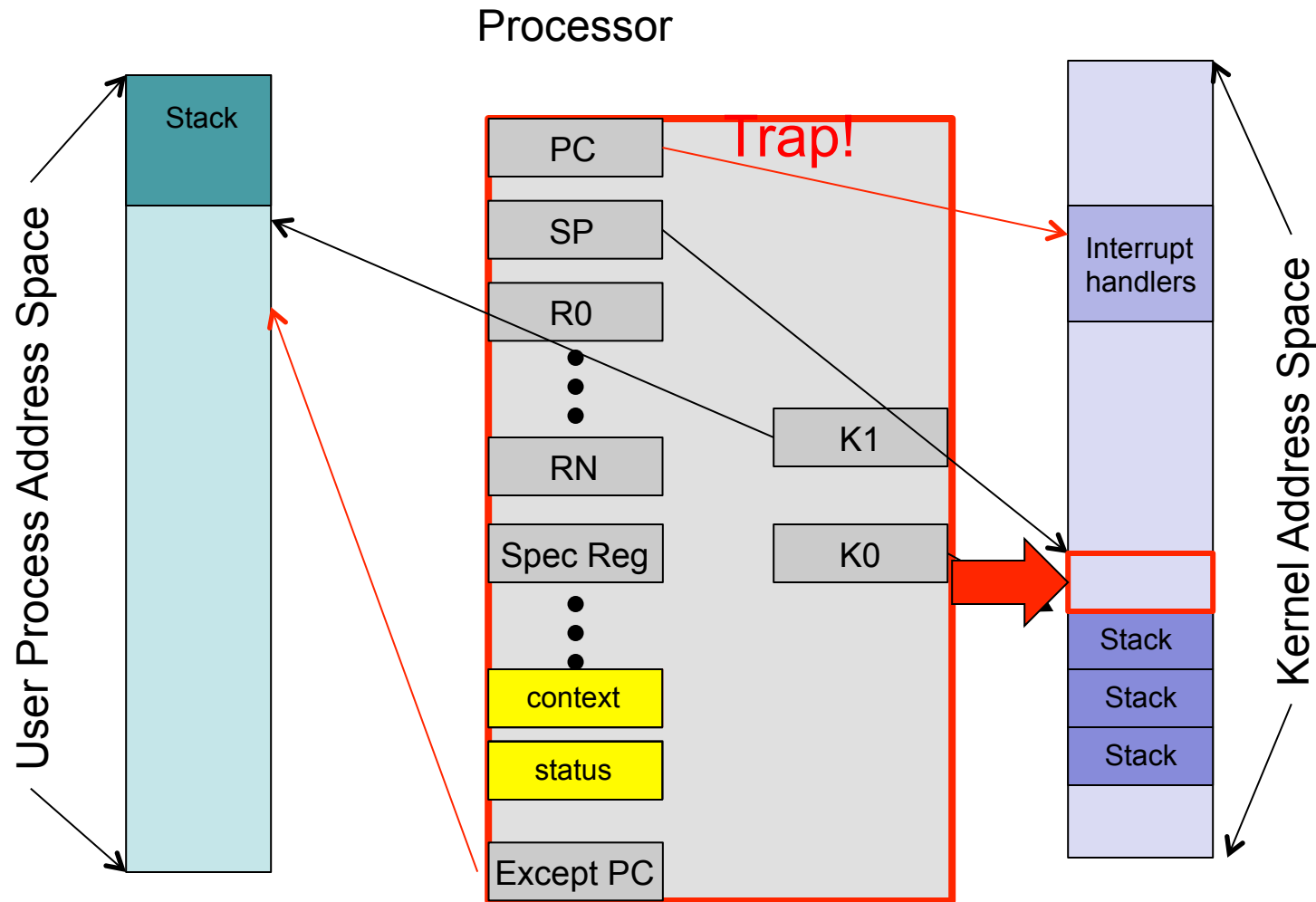
kern/arch/mips/locore/exception-mips1.S(9)  
line 137 (create a stack frame)

SW:  
Allocate stack  
frame (bump SP)



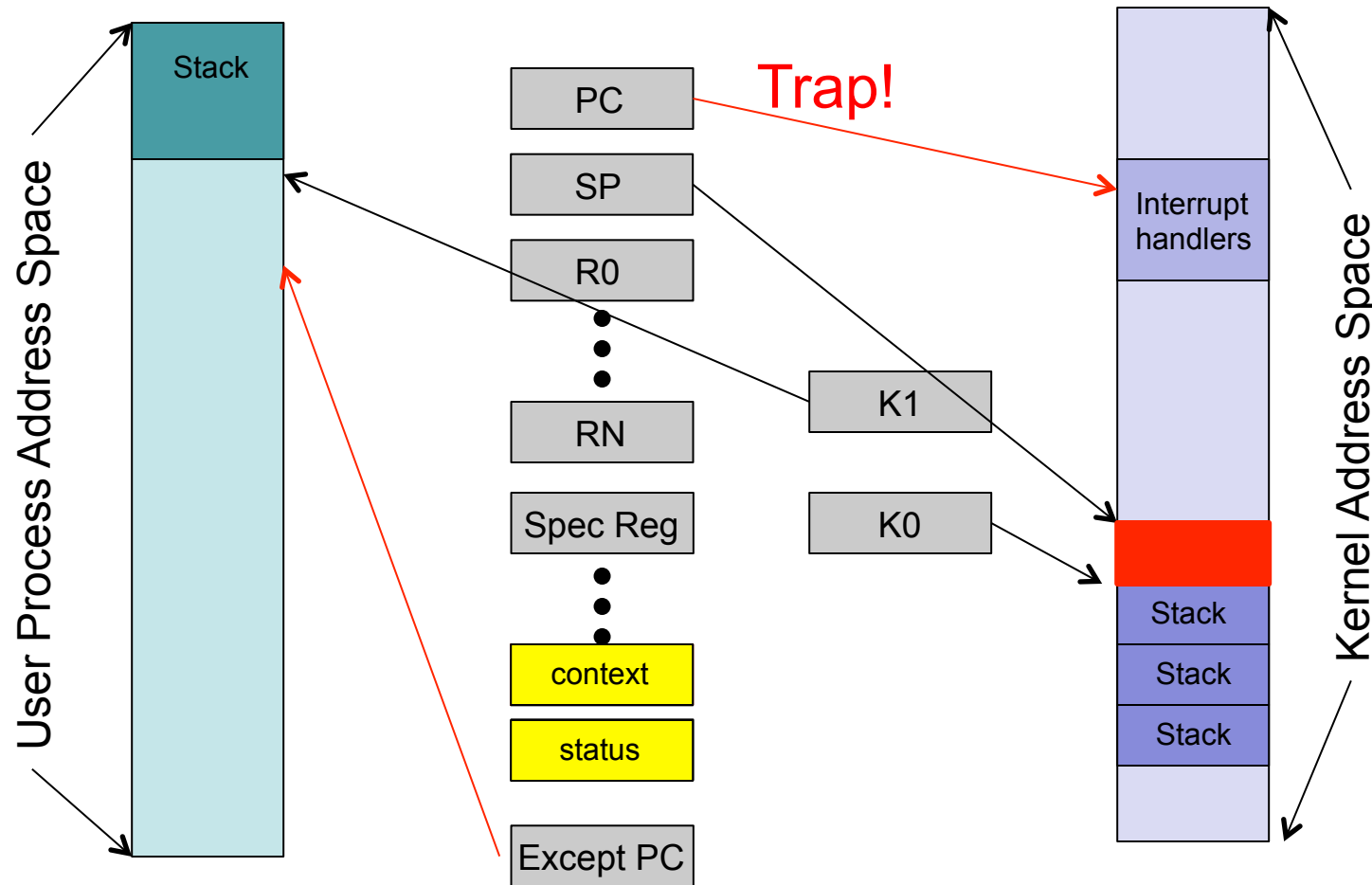
kern/arch/mips/locore/exception-mips1.S(10)  
line 170-233 (save registers)

SW:  
Copy all our  
registers onto the  
stack



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

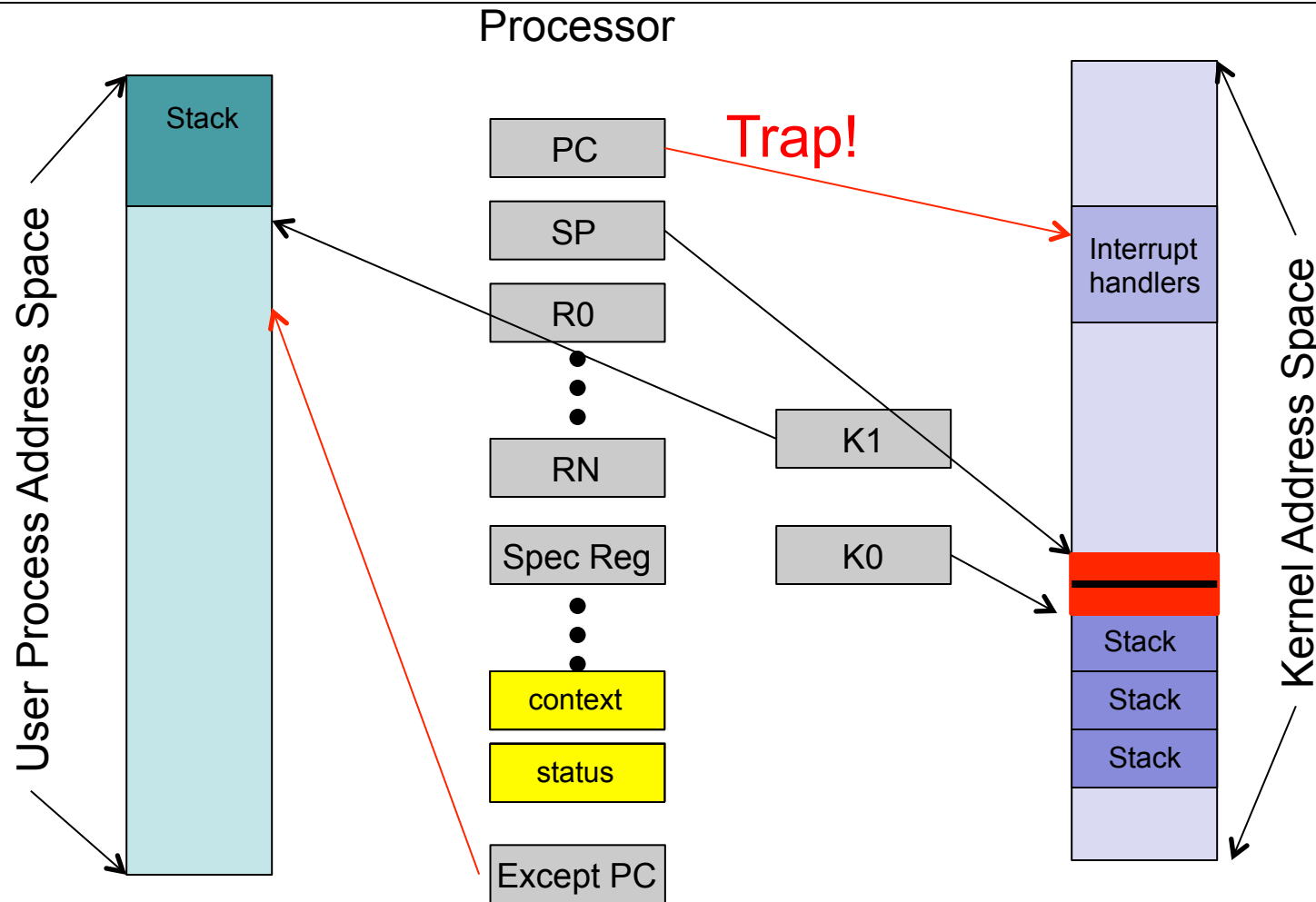
Processor



SW:  
Trapframe on the  
stack is the  
argument to  
mips\_trap. Call it.

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

SW:  
Extract exception  
type from the  
trap frame.



# 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_iphigh_count == 0);
```

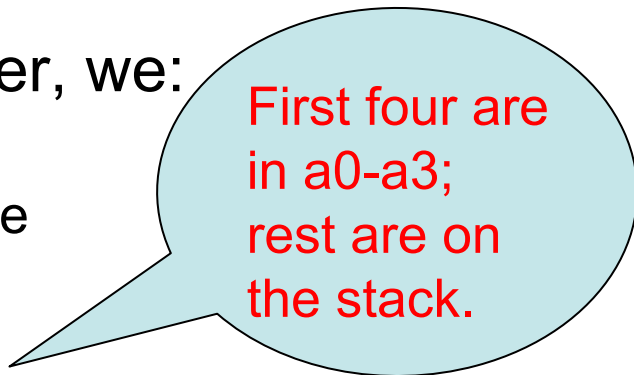


# Syscall Details

- 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

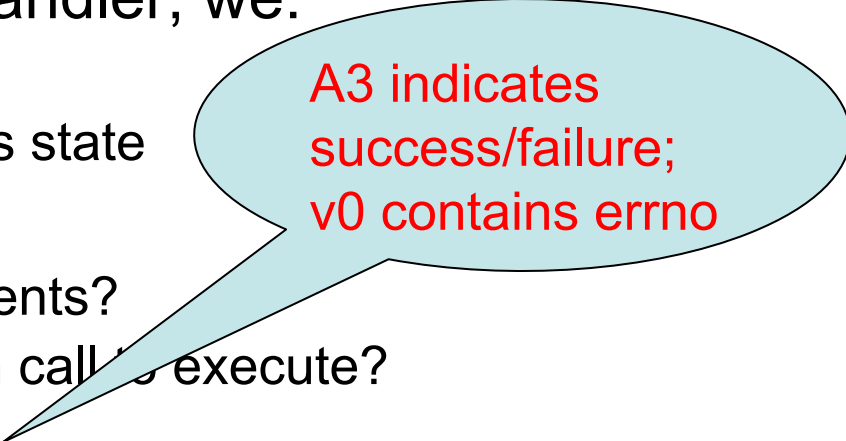
- Upon entry into our syscall handler, we:
  - Are in supervisor mode
  - Have saved away the process's state
- System call details
  - Where did we leave the arguments?
  - How do we know which system call to execute?
  - Where do we return an error?
- Do we need to do anything special with the arguments?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

# 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` makes a copy of the calling process.
    - Example, the shell makes a copy of itself each time you want to execute a program.
    - One copy waits while the other executes the command.
  - 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 PCB of the original process into the new one.
  - Make the new process known to the dispatcher.
- Challenges
  - The two processes are identical.
  - How do you get useful work done?
  - `exec`: replace the current program image with the code and data of a new program.
- One problem left; what is it?

# What about that first process?

- How do you get started?
  - Hand crafted by the operating system when it begins running.
  - Allocate space for the process.
  - Load the code and data into memory from disk.
  - Create (empty) call stack (procedures, local variables).
  - Create and initialize a PCB.
  - Make the process known to the dispatcher.
- Alternative model: Windows
  - CreateProc: creates a single-threaded process.
- What are the tradeoffs?