Processes and Threads Implementation

Learning Outcomes

- An understanding of the typical implementation strategies of processes and threads
  - Including an appreciation of the trade-offs between the implementation approaches
    - Kernel-threads versus user-level threads
- A detailed understanding of “context switching”

Multiprogramming Implementation

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly language procedure starts up new current process.

Skeleton of what lowest level of OS does when an interrupt occurs – a context switch

Context Switch Terminology

- A context switch can refer to
  - A switch between threads
    - Involving saving and restoring of state associated with a thread
  - A switch between processes
    - Involving the above, plus extra state associated with a process.
      - E.g. memory maps

Context Switch Occurrence

- A switch between process/threads can happen any time the OS is invoked
  - On a system call
    - Mandatory if system call blocks or on exit();
  - On an exception
    - Mandatory if offender is killed
  - On an interrupt
    - Triggering a dispatch is the main purpose of the *timer interrupt*

A thread switch can happen between any two instructions

Note instructions do not equal program statements

Context Switch

- Context switch must be *transparent* for processes/threads
  - When dispatched again, process/thread should not notice that something else was running in the meantime (except for elapsed time)

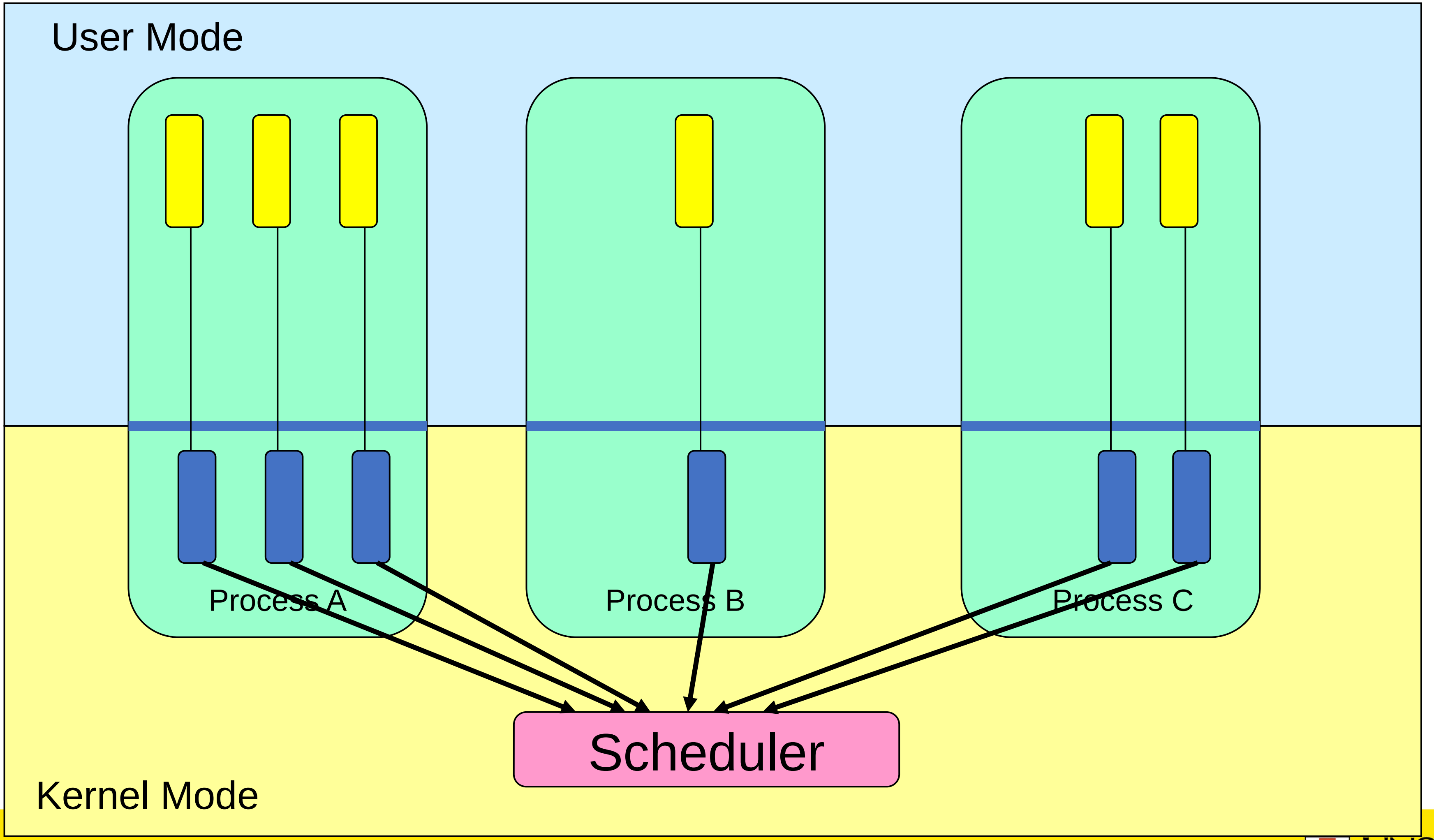
⇒OS must save all state that affects the thread

- This state is called the *process/thread context*
- Switching between process/threads consequently results in a *context switch*.

Simplified Explicit Thread Switch

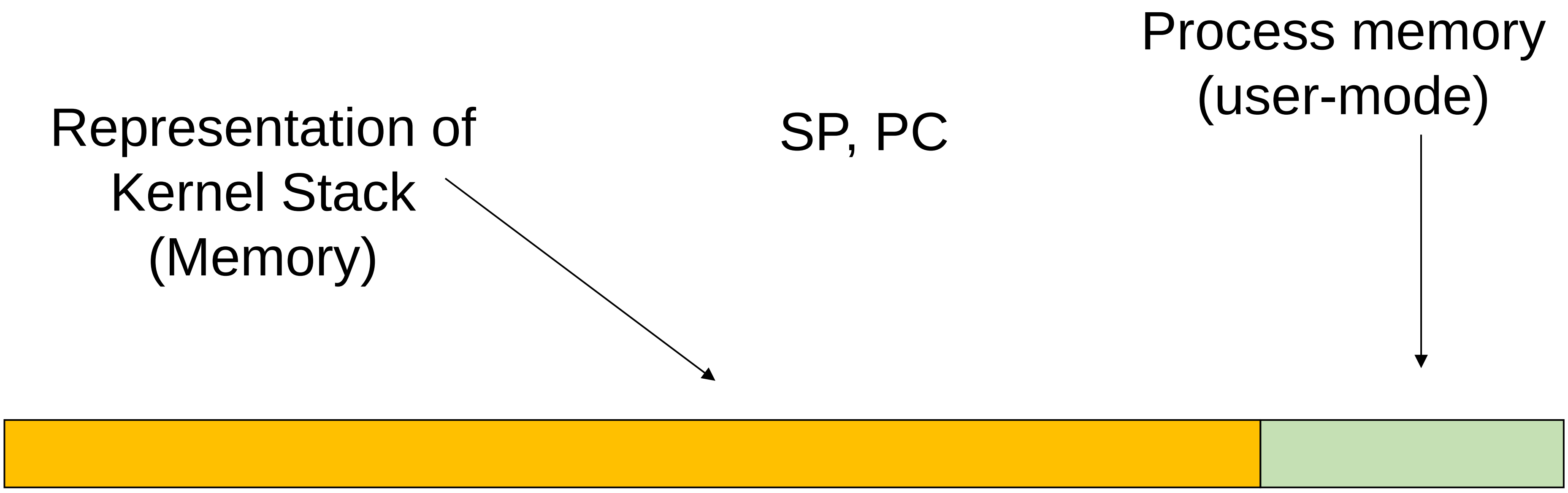
Assume Kernel-Level Threads

Lets focus on user->kernel – switch – kernel -> user



Example Context Switch

- Running in user mode, SP points to user-level stack (not shown on slide)

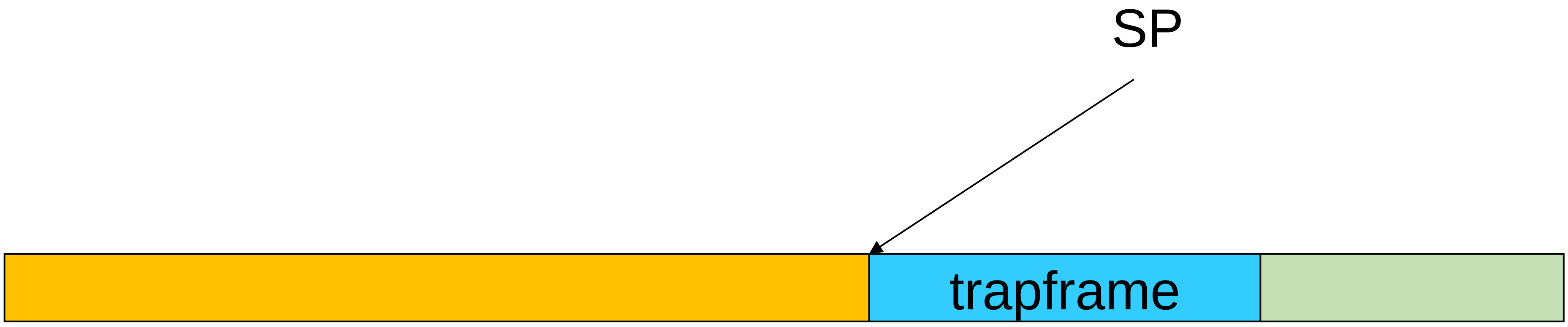


Example Context Switch

- Take an exception, syscall, or interrupt, and we switch to the kernel stack

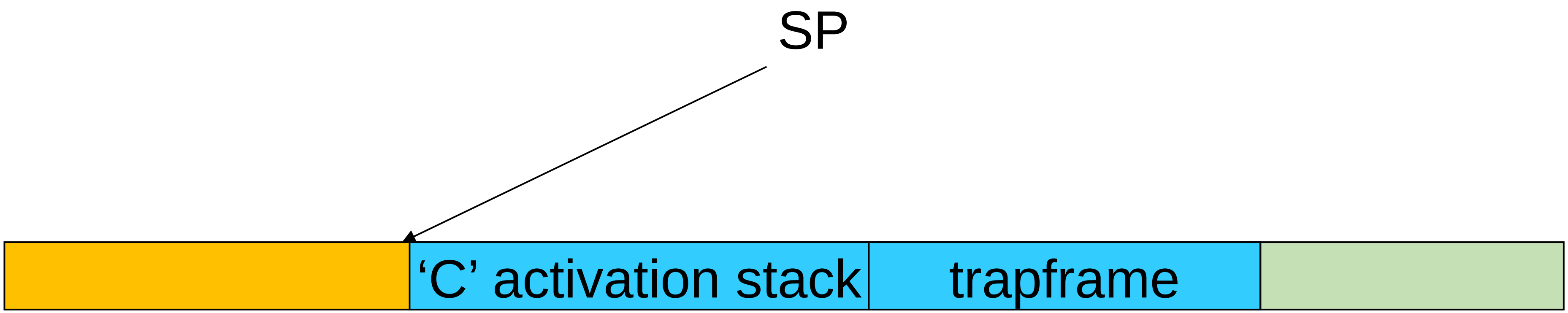
Example Context Switch

- We push a *trapframe* on the stack
  - Also called *exception frame, user-level context….*
  - Includes the user-level PC and SP



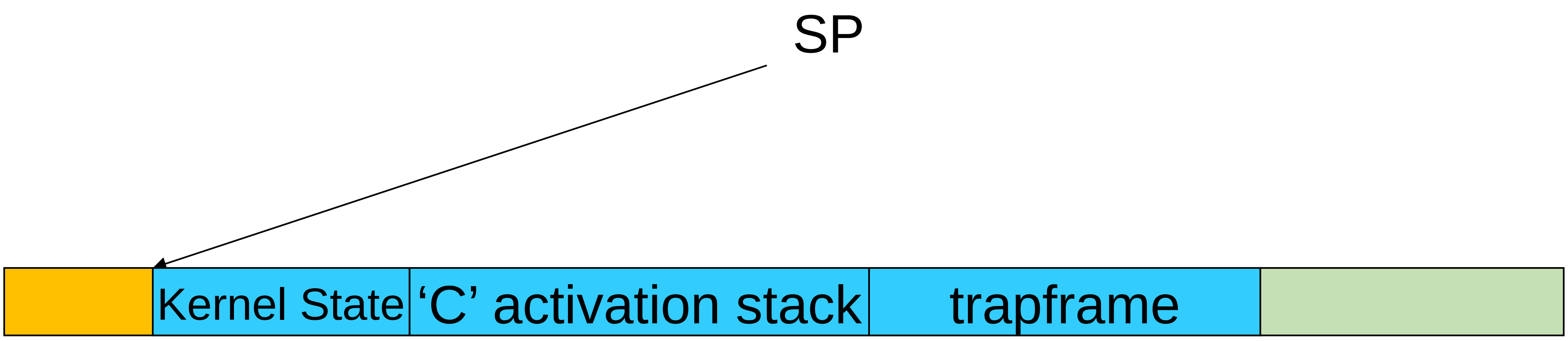
Example Context Switch

- Call ‘C’ code to process syscall, exception, or interrupt
  - Results in a ‘C’ activation stack building up



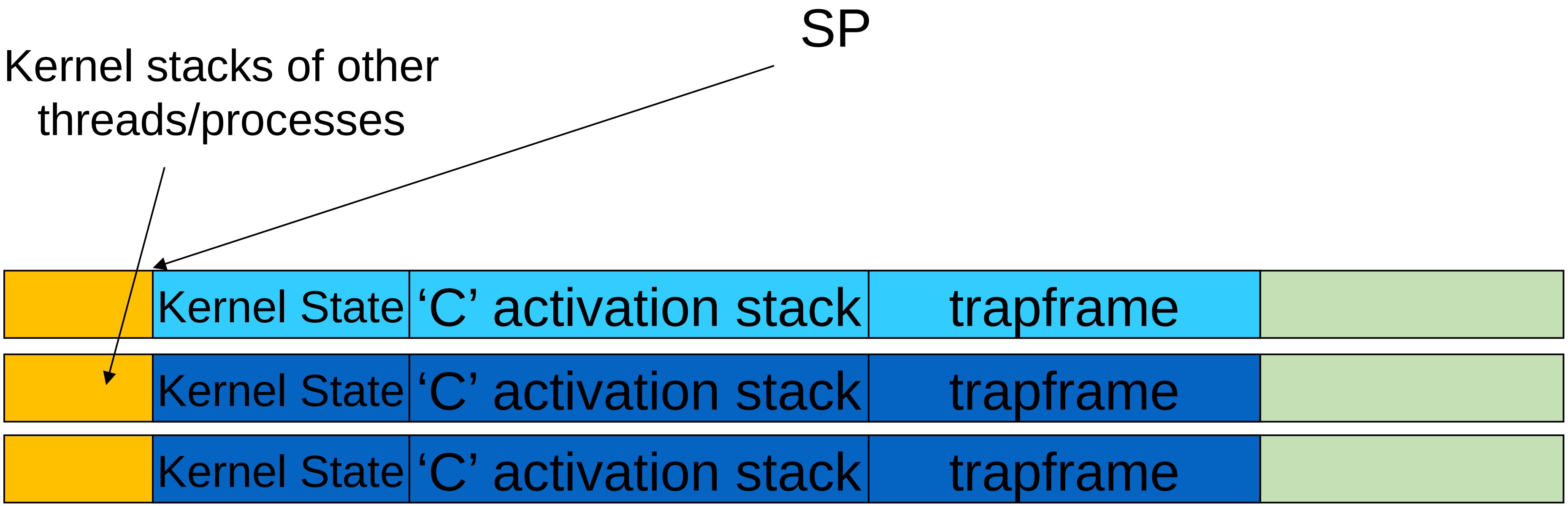
Example Context Switch

- The kernel decides to perform a context switch
  - It chooses a target thread (or process)
  - It pushes remaining kernel context onto the stack



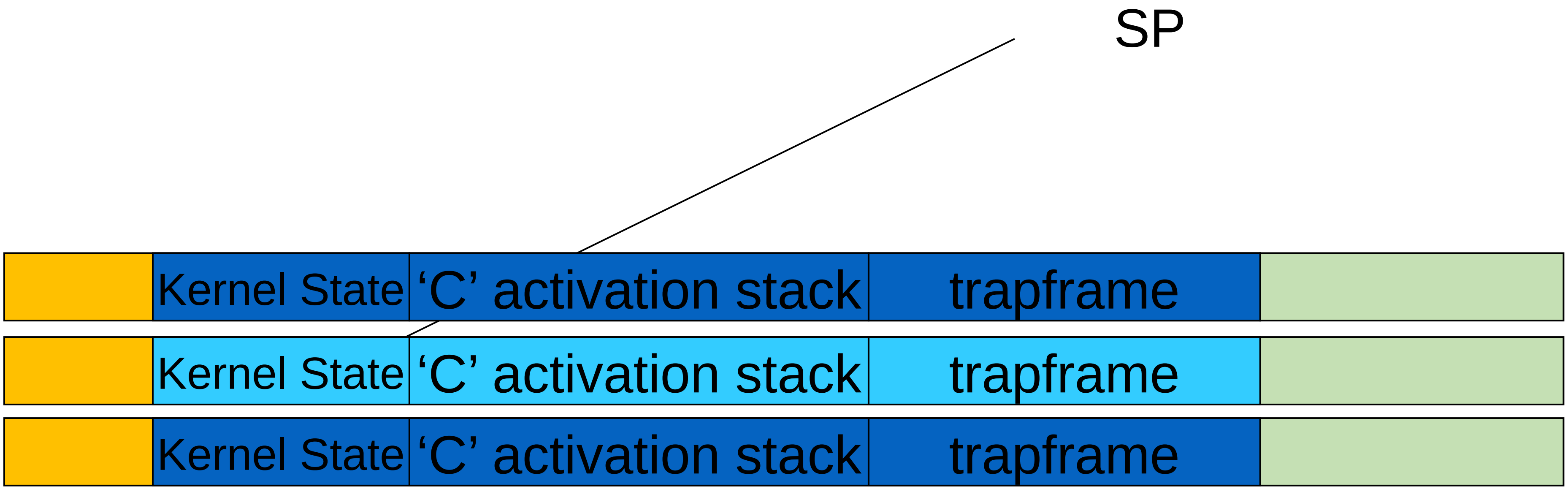
Example Context Switch

- Any other existing thread must
  - be in kernel mode (on a uni processor),
  - and have a similar stack layout to the stack we are currently using



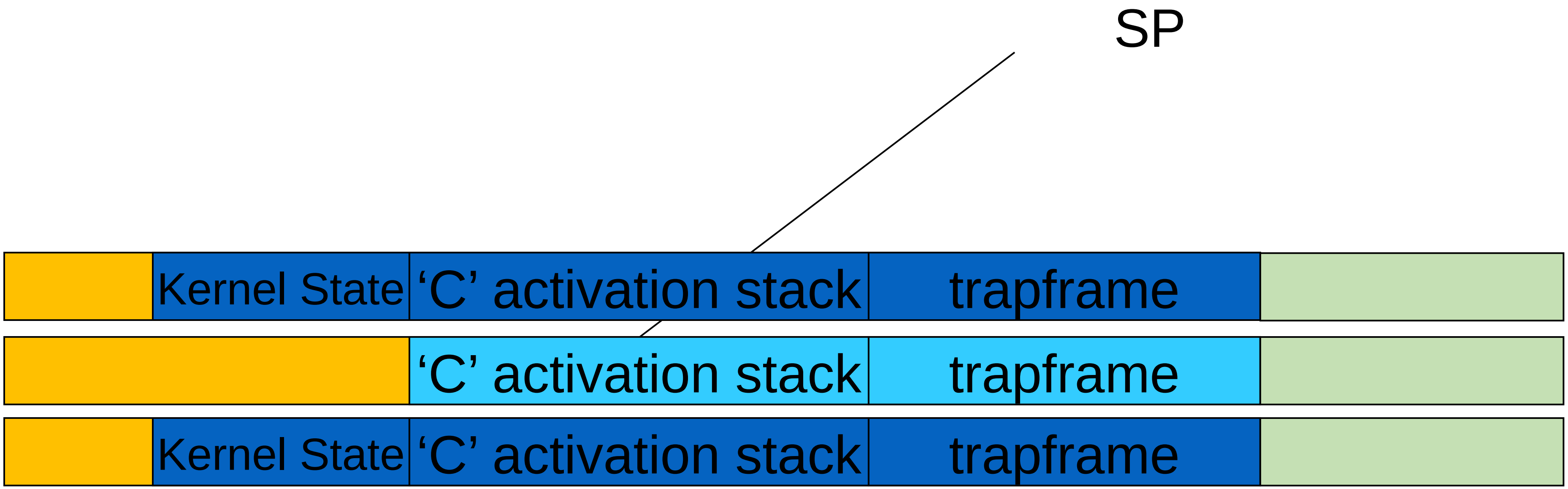
Example Context Switch

- We save the current SP in the PCB (or TCB), and load the SP of the target thread.
  - Thus we have *switched contexts*



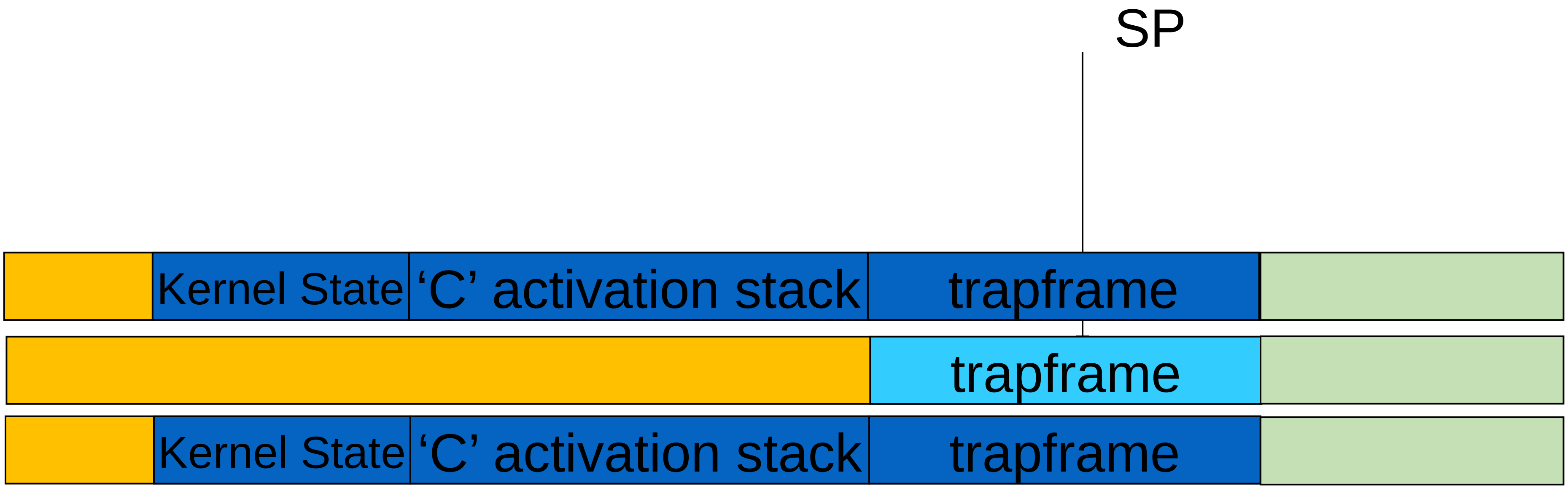
Example Context Switch

- Load the target thread's previous context, and return to C



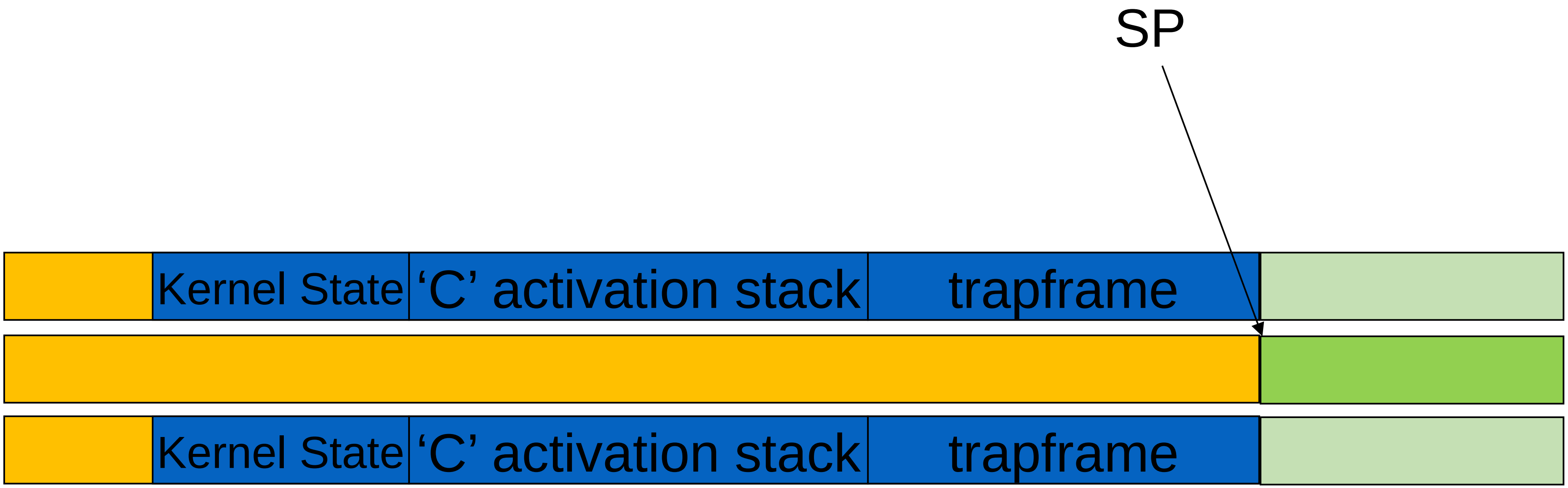
Example Context Switch

- The C continues and (in this example) returns to user mode.



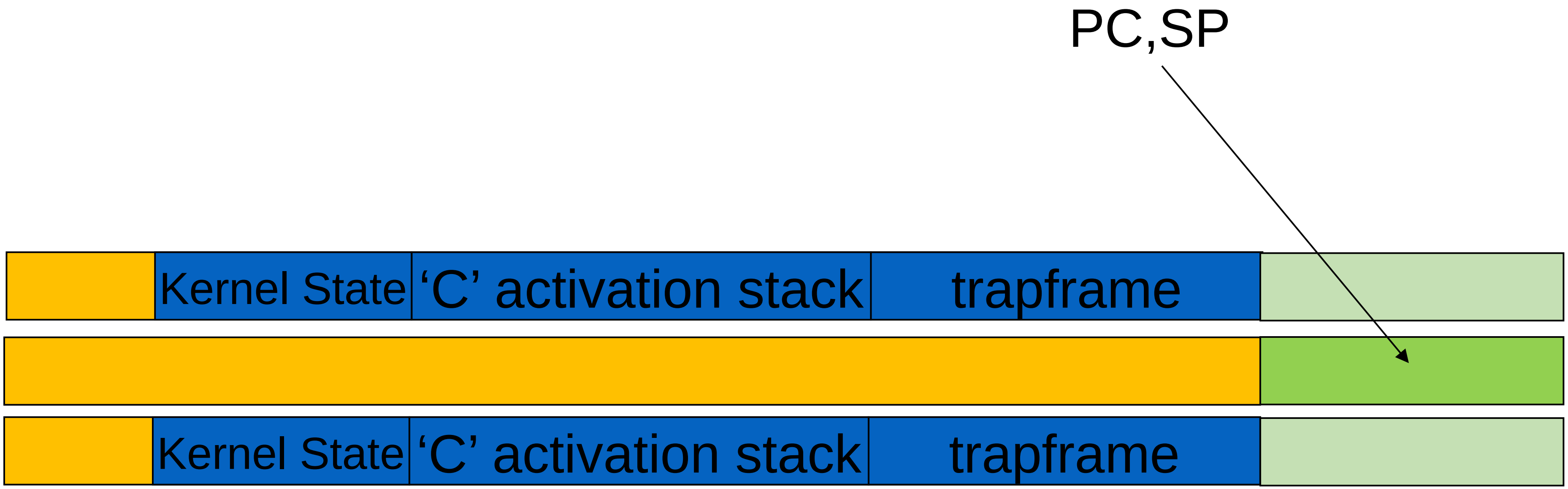
Example Context Switch

- The user-level context is restored
  - The registers load with that processes previous content



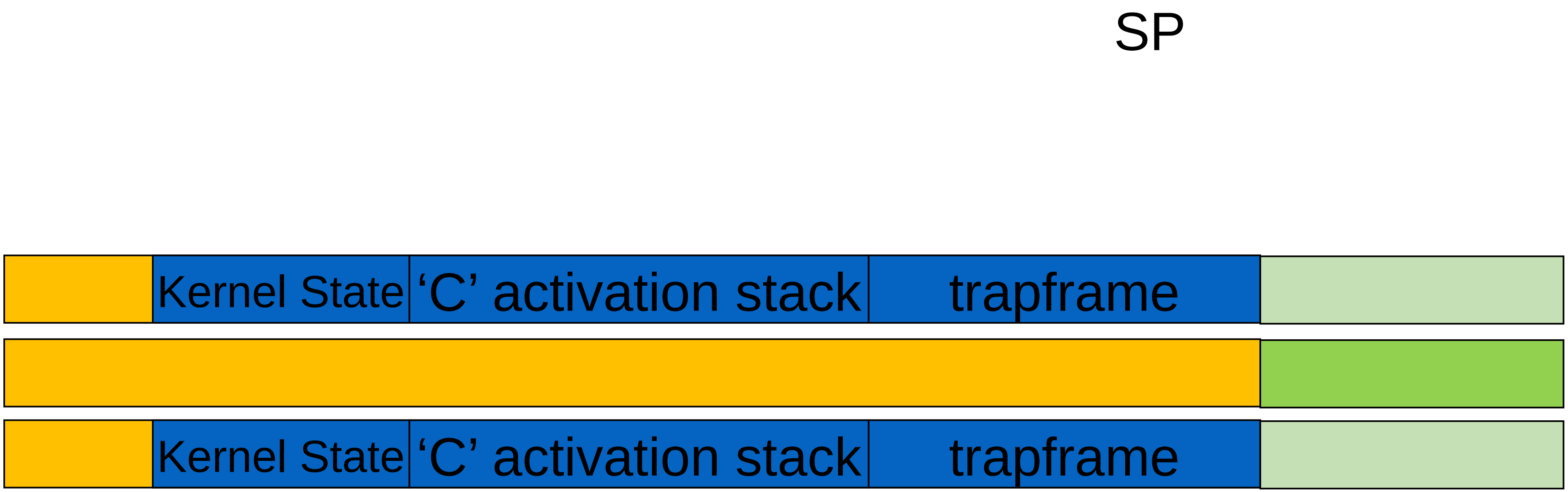
Example Context Switch

- The user-level SP and PC is restored



The Interesting Part of a Thread Switch

- What does the "push kernel state" part do???



Simplified OS/161 thread_switch

```
static
void
thread_switch(threadstate_t newstate, struct wchan *wc)
{
struct thread *cur, *next;

cur = curthread;
do {
  next = threadlist_remhead(&curcpu->c_runqueue);
  if (next == NULL) {
        cpu_idle();
  }
} while (next == NULL);

/* do the switch (in assembler in switch.S) */
switchframe_switch(&cur->t_context, &next->t_cont

}
```

Lots of code removed – only basics of pick next thread and switch to it remain

OS/161 switchframe_switch

```
switchframe_switch:
  /*
   * a0 contains the address of the switchframe pointer in the old thread.
   * a1 contains the address of the switchframe pointer in the new thread.
   *
   * The switchframe pointer is really the stack pointer. The other
   * registers get saved on the stack, namely:
   *
   *      s0-s6, s8
   *      gp, ra
   *
   * The order must match <mips/switchframe.h>.
   *
   * Note that while we'd ordinarily need to save s7 too, because we
   * use it to hold curthread saving it would interfere with the way
   * curthread is managed by thread.c. So we'll just let thread.c
   * manage it.
   */
```

OS/161 switchframe_switch

```
/* Allocate stack space for saving 10 registers. 10*4 = 40 */
addi sp, sp, -40

/* Save the registers */
sw   ra, 36(sp)
sw   gp, 32(sp)
sw   s8, 28(sp)
sw   s6, 24(sp)
sw   s5, 20(sp)
sw   s4, 16(sp)
sw   s3, 12(sp)
sw   s2, 8(sp)
sw   s1, 4(sp)
sw   s0, 0(sp)

/* Store the old stack pointer in the old thread */
sw   sp, 0(a0)
```

Save the registers that the ‘C’ procedure calling convention expects preserved

OS/161 switchframe_switch

```
/* Get the new stack pointer from the new thread */
lw   sp, 0(a1)
nop          /* delay slot for load */

/* Now, restore the registers */
lw   s0, 0(sp)
lw   s1, 4(sp)
lw   s2, 8(sp)
lw   s3, 12(sp)
lw   s4, 16(sp)
lw   s5, 20(sp)
lw   s6, 24(sp)
lw   s8, 28(sp)
lw   gp, 32(sp)
lw   ra, 36(sp)
nop              /* delay slot for load */
```

OS/161 switchframe_switch

```
/* and return. */
j ra
addi sp, sp, 40      /* in delay slot */
```

Simplified Explicit Thread Switch