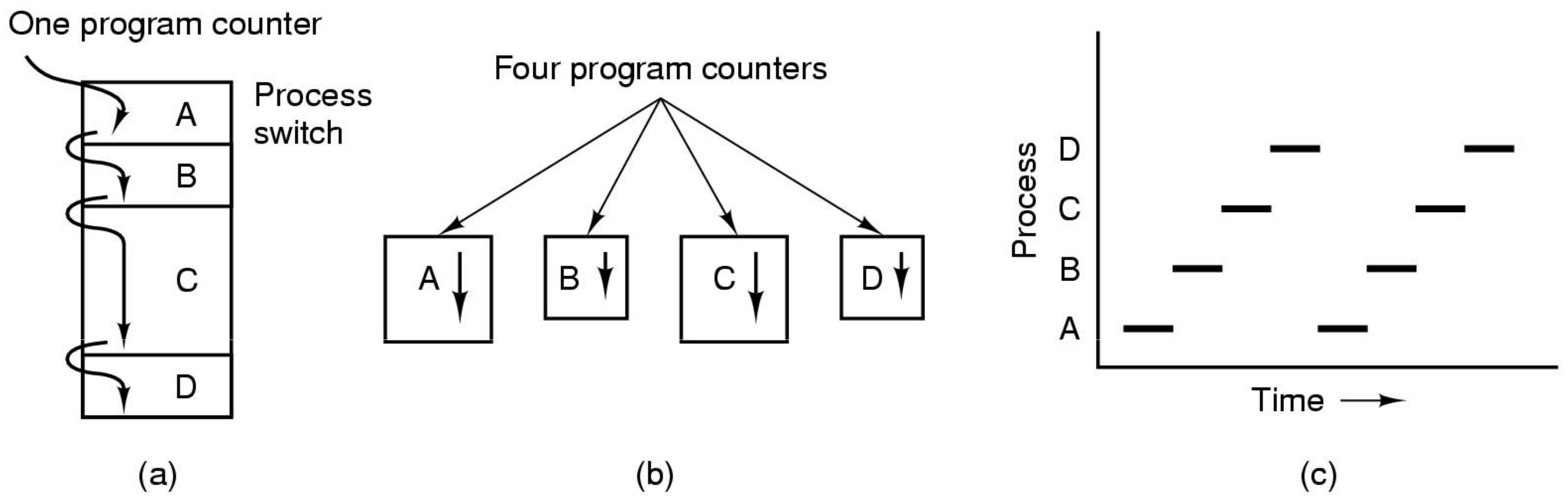
Processes and Threads Implementation

Learning Outcomes

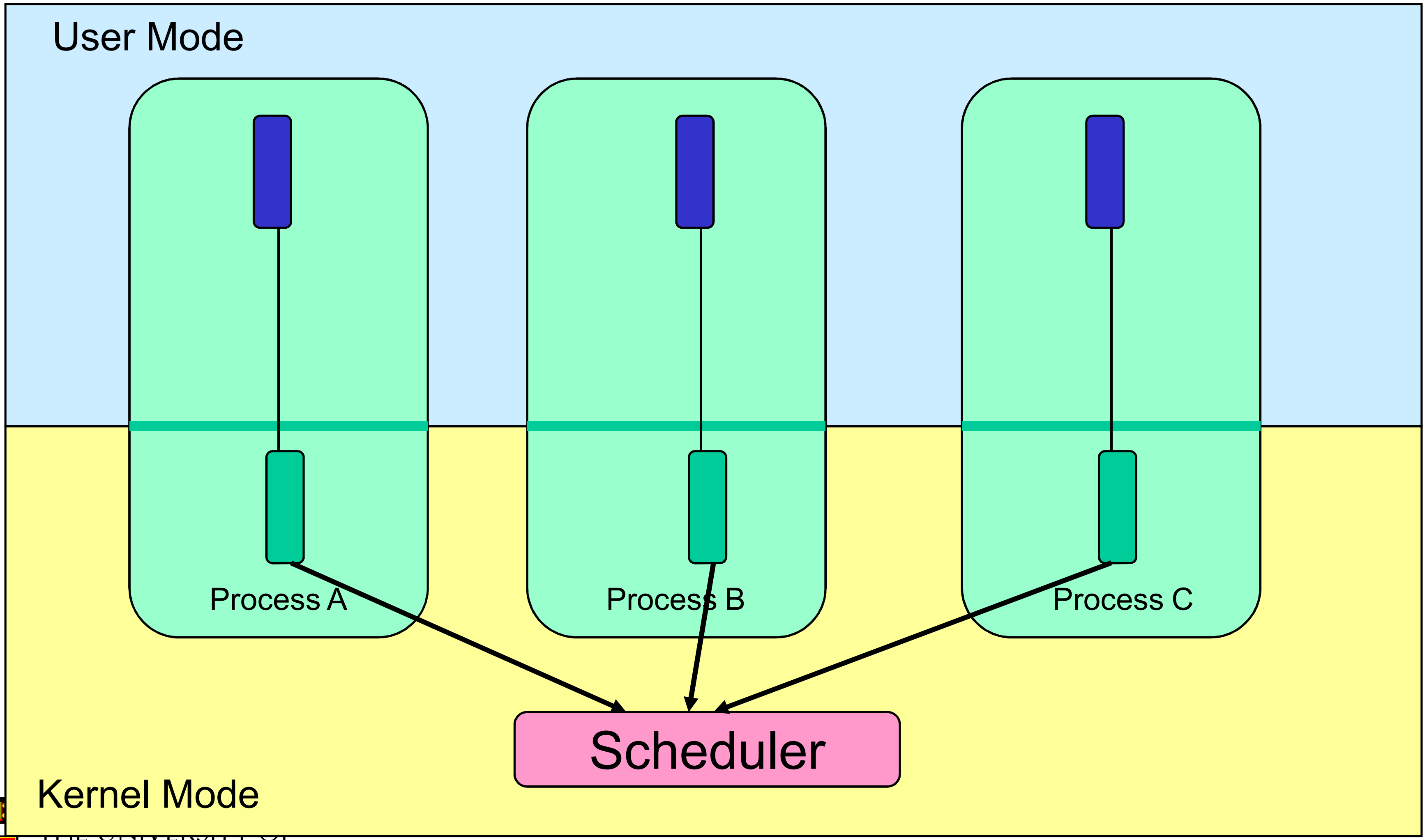
- An understanding of the typical implementation strategies of processes and threads
  - Including an appreciation of the trade-offs between the implementation approaches
    - Kernel-threads versus user-level threads
- A detailed understanding of "context switching"

Summary: The Process Model



- Multiprogramming of four programs
- Conceptual model of 4 independent, sequential processes (with a single thread each)
- Only one program active at any instant

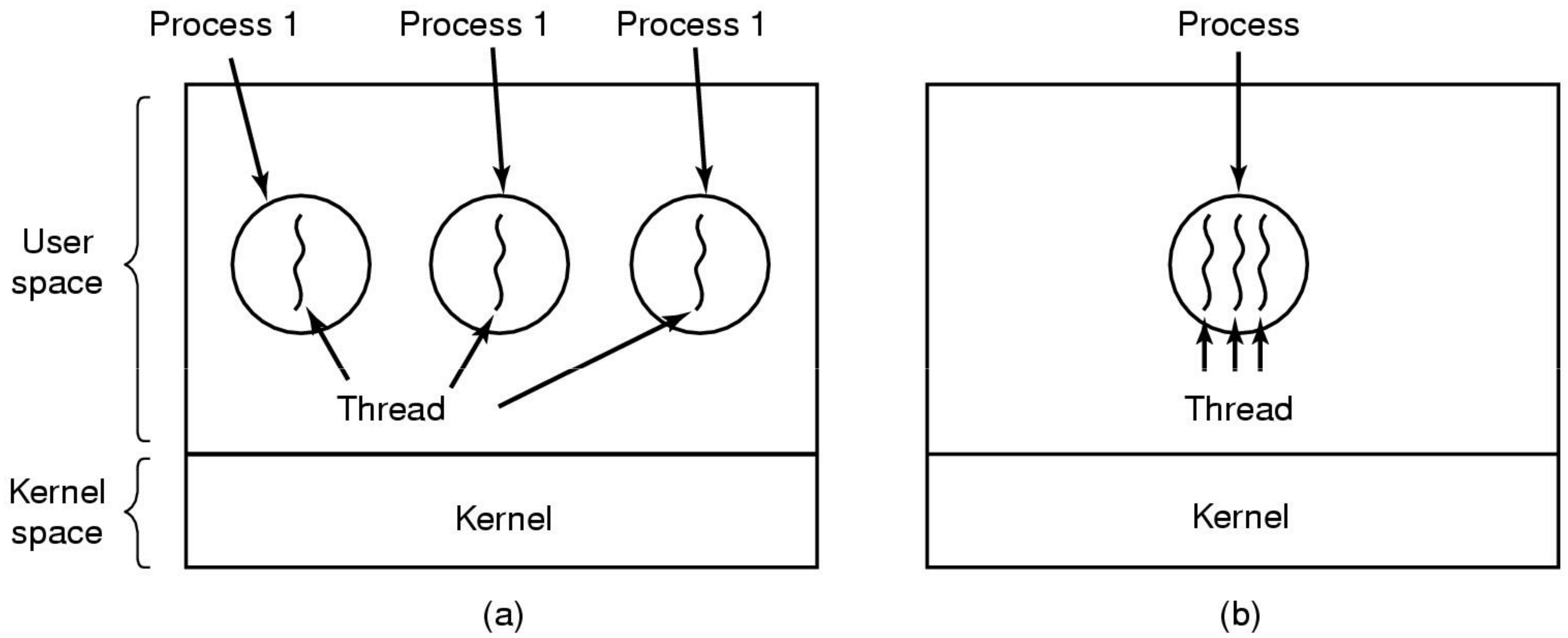
Processes


User Mode
Process A
Process B
Process C
Scheduler
Kernel Mode

Processes

- User-mode
  - Processes (programs) scheduled by the kernel
  - Isolated from each other
  - No concurrency issues between each other
- System-calls transition into and return from the kernel
- Kernel-mode
  - Nearly all activities still associated with a process
  - Kernel memory shared between all processes
  - Concurrency issues exist between processes concurrently executing in a system call

Threads
The Thread Model



(a) Three processes each with one thread

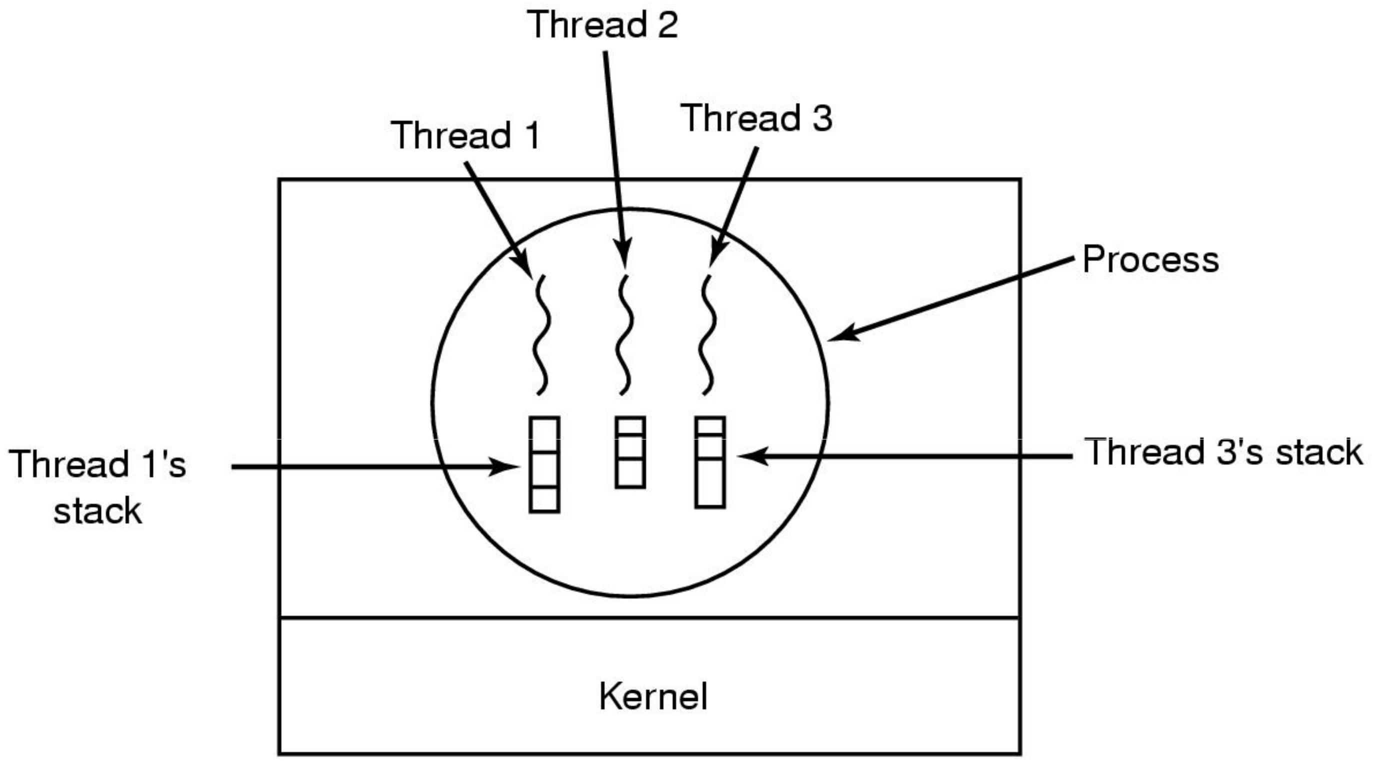
(b) One process with three threads

The Thread Model

Per process items	**Per thread items**
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

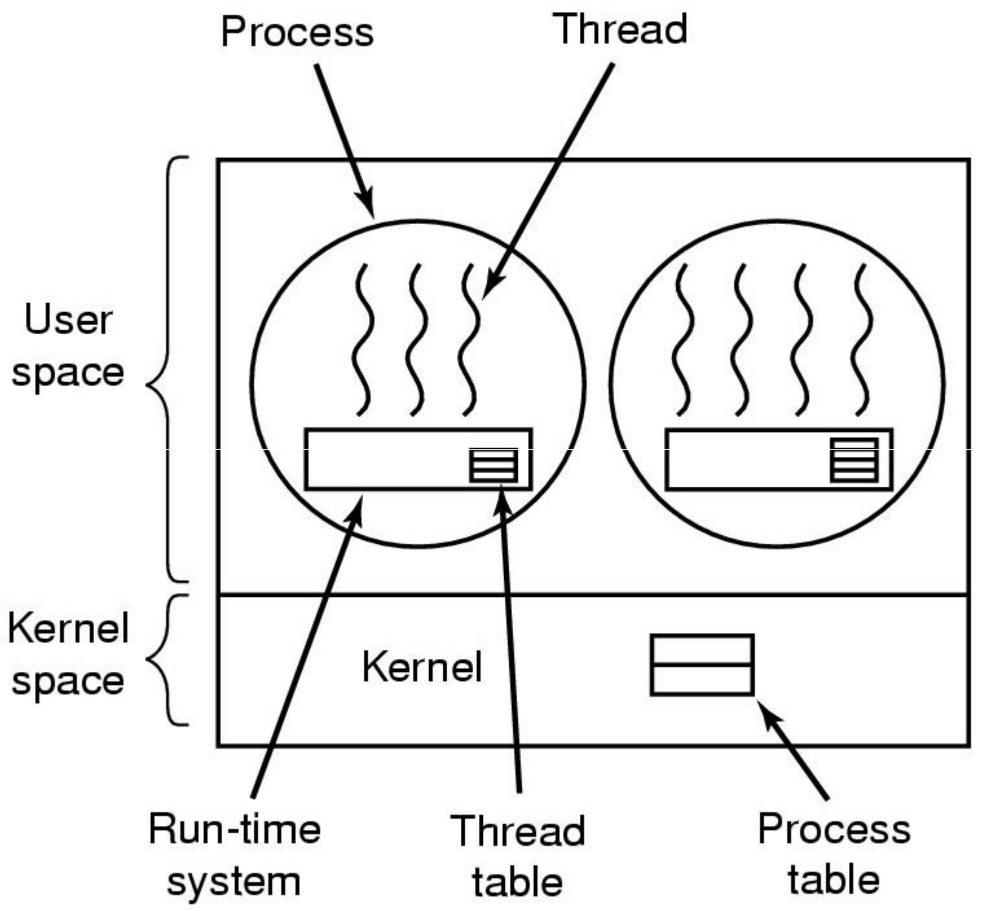
- Items shared by all threads in a process
- Items private to each thread

The Thread Model



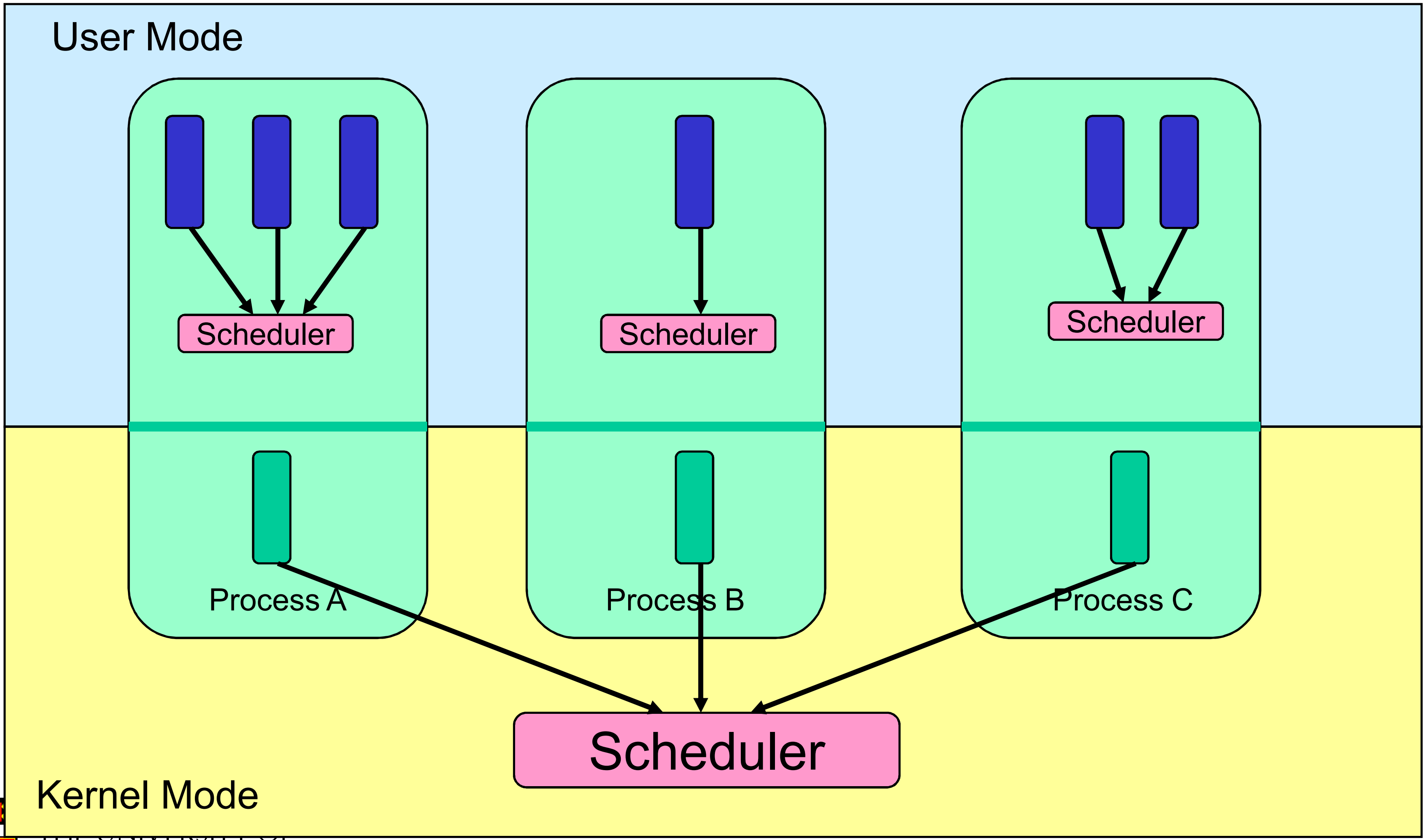
Each thread has its own stack

Implementing Threads in User Space



A user-level threads package

User-level Threads



User-level Threads

- Implementation at user-level
  - User-level Thread Control Block (TCB), ready queue, blocked queue, and dispatcher
  - Kernel has no knowledge of the threads (it only sees a single process)
  - If a thread blocks waiting for a resource held by another thread, its state is saved and the dispatcher switches to another ready thread
  - Thread management (create, exit, yield, wait) are implemented in a runtime support library

User-Level Threads

- Pros
  - Thread management and switching at user level is much faster than doing it in kernel level
    - No need to trap (take syscall exception) into kernel and back to switch
  - Dispatcher algorithm can be tuned to the application
    - E.g. use priorities
  - Can be implemented on any OS (thread or non-thread aware)
  - Can easily support massive numbers of threads on a per-application basis
    - Use normal application virtual memory
    - Kernel memory more constrained. Difficult to efficiently support wildly differing numbers of threads for different applications.

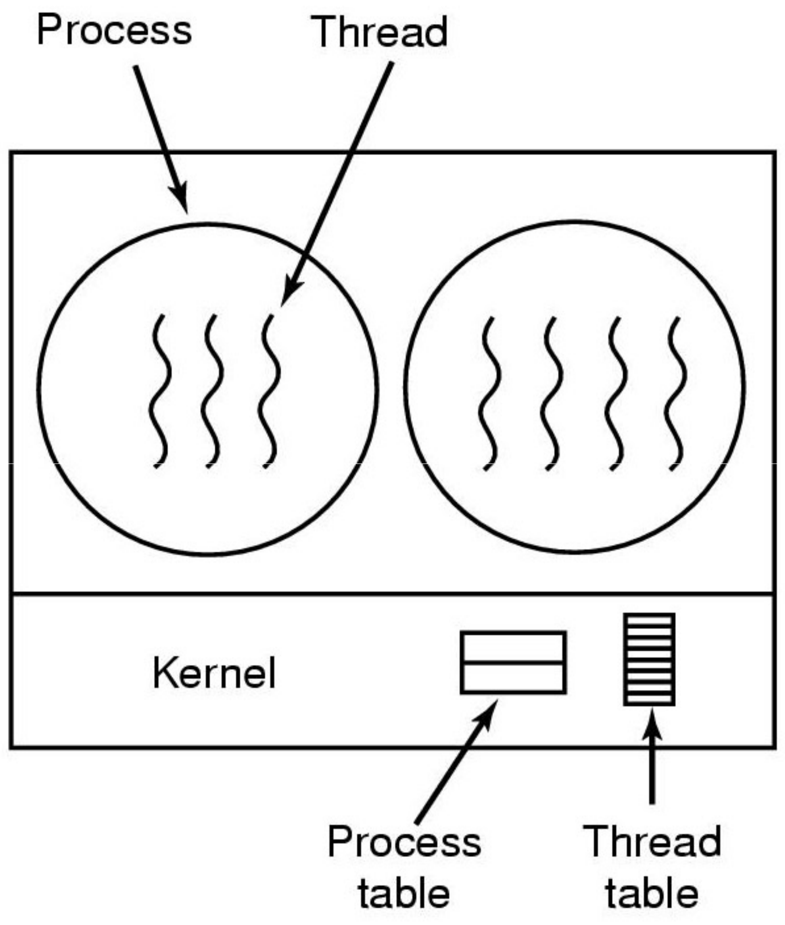
User-level Threads

- Cons
  - Threads have to yield() manually (no timer interrupt delivery to user-level)
    - Co-operative multithreading
      - A single poorly design/implemented thread can monopolise the available CPU time
    - There are work-arounds (e.g. a timer signal per second to enable pre-emptive multithreading), they are course grain and a kludge.
  - Does not take advantage of multiple CPUs (in reality, we still have a single threaded process as far as the kernel is concerned)

User-Level Threads

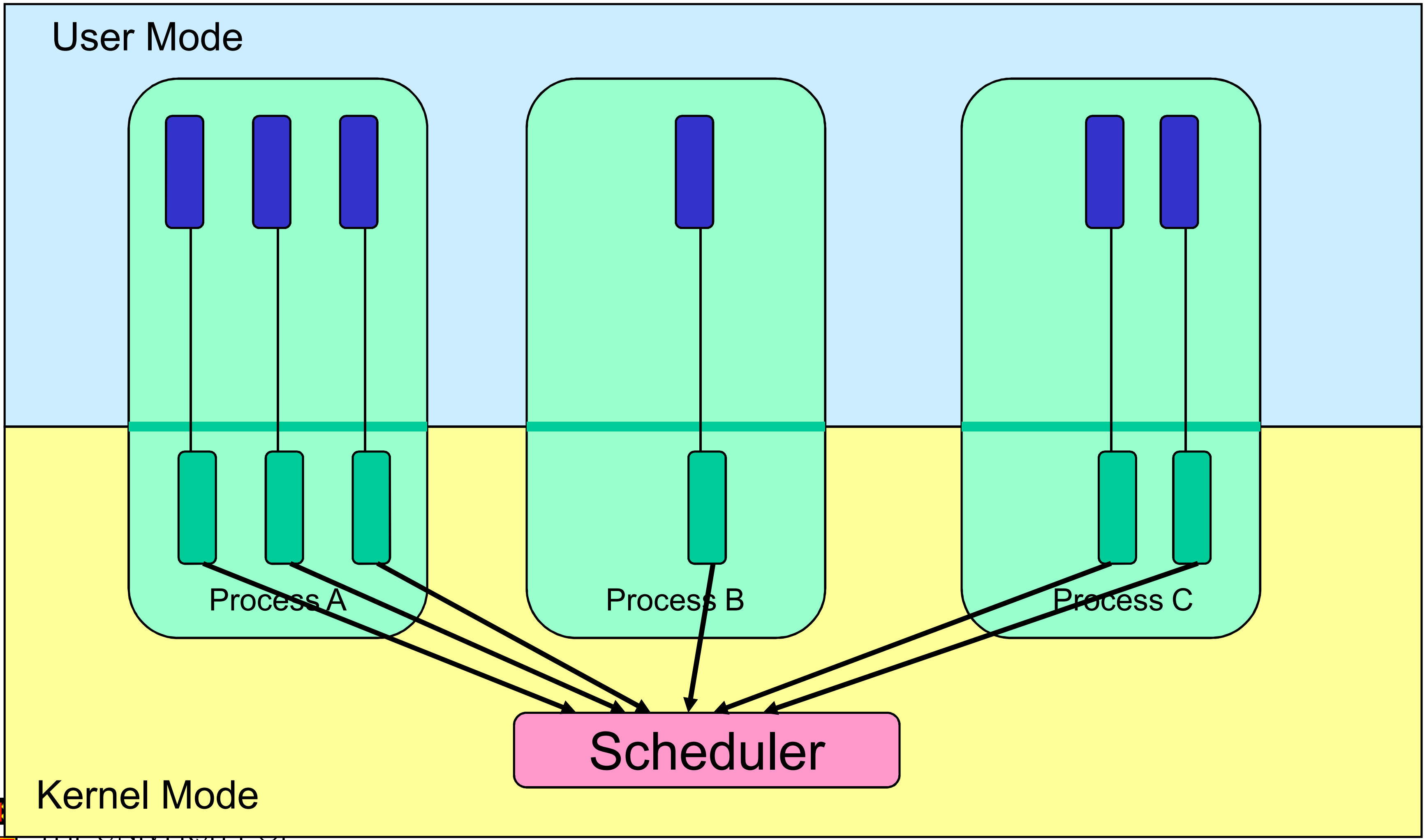
- Cons
  - If a thread makes a blocking system call (or takes a page fault), the process (and all the internal threads) blocks
    - Can't overlap I/O with computation
    - Can use wrappers as a work around
      - Example: wrap the **`read()`** call
      - Use **`select()`** to test if read system call would block
        - » **`select()`** then **`read()`**
        - » Only call **`read()`** if it won't block
        - » Otherwise schedule another thread
      - Wrapper requires 2 system calls instead of one
        - » Wrappers are needed for environments doing lots of blocking system calls – exactly when efficiency matters!
    - Can change to kernel to support non-blocking system call
      - Lose "on any system" advantage, page faults still a problem.

Implementing Threads in the Kernel



A threads package managed by the kernel

Kernel-Level Threads



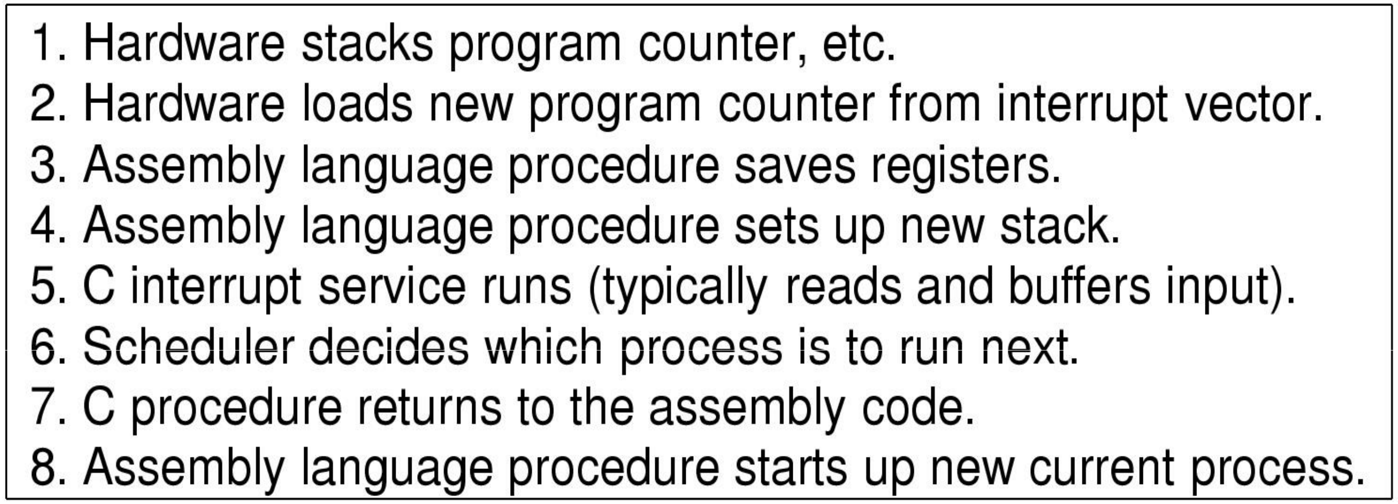
Kernel Threads

- Threads are implemented in the kernel
  - TCBs are stored in the kernel
    - A subset of information in a traditional PCB
      - The subset related to execution context
    - TCBs have a PCB associated with them
      - Resources associated with the group of threads (the process)
  - Thread management calls are implemented as system calls
    - E.g. create, wait, exit

Kernel Threads

- Cons
  - Thread creation and destruction, and blocking and unblocking threads requires kernel entry and exit.
    - More expensive than user-level equivalent
- Pros
  - Preemptive multithreading
  - Parallelism
    - Can overlap blocking I/O with computation
    - Can take advantage of a multiprocessor

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 thread/context switch

Thread Switch

- A switch between 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

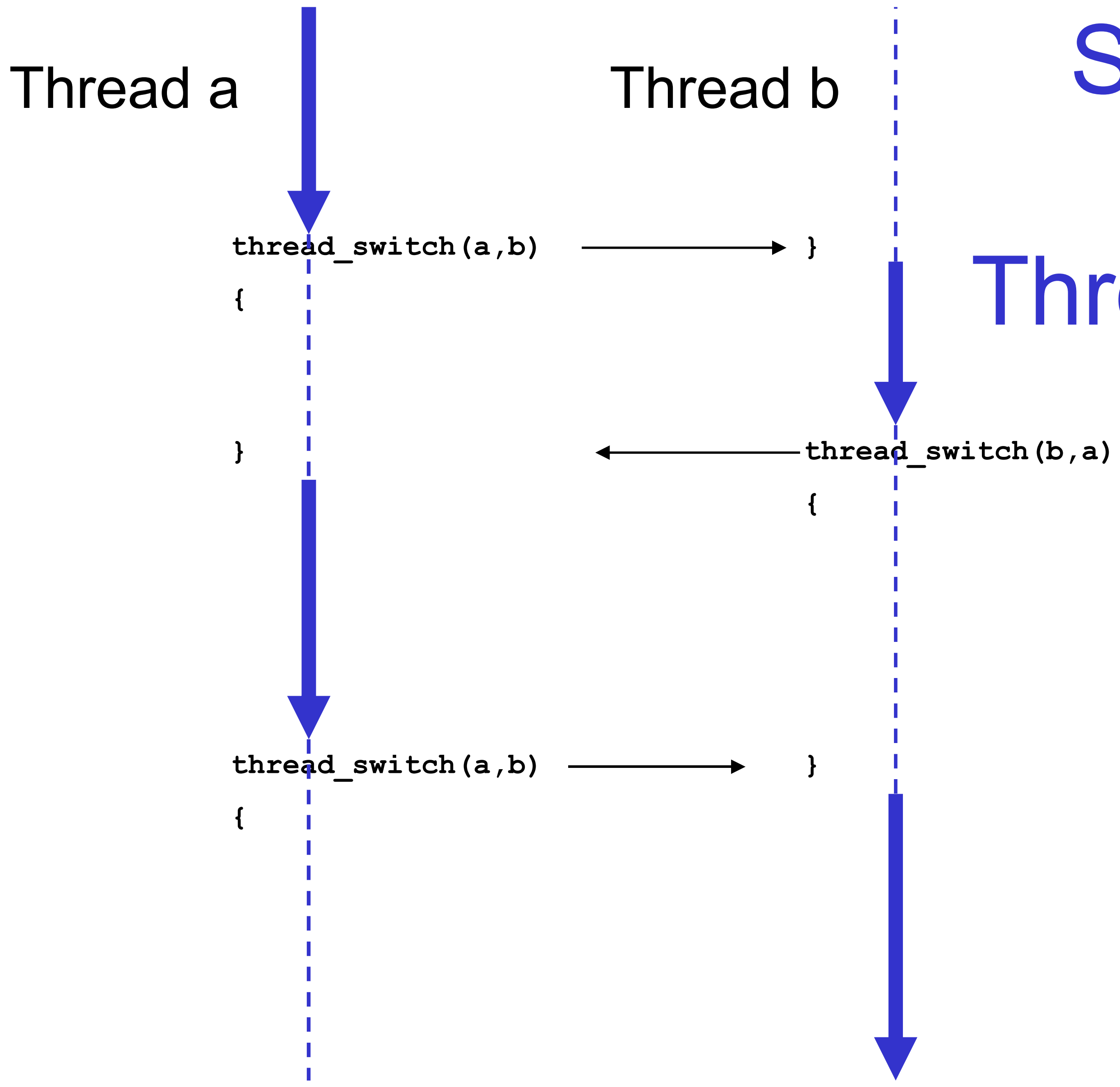
Context Switch

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

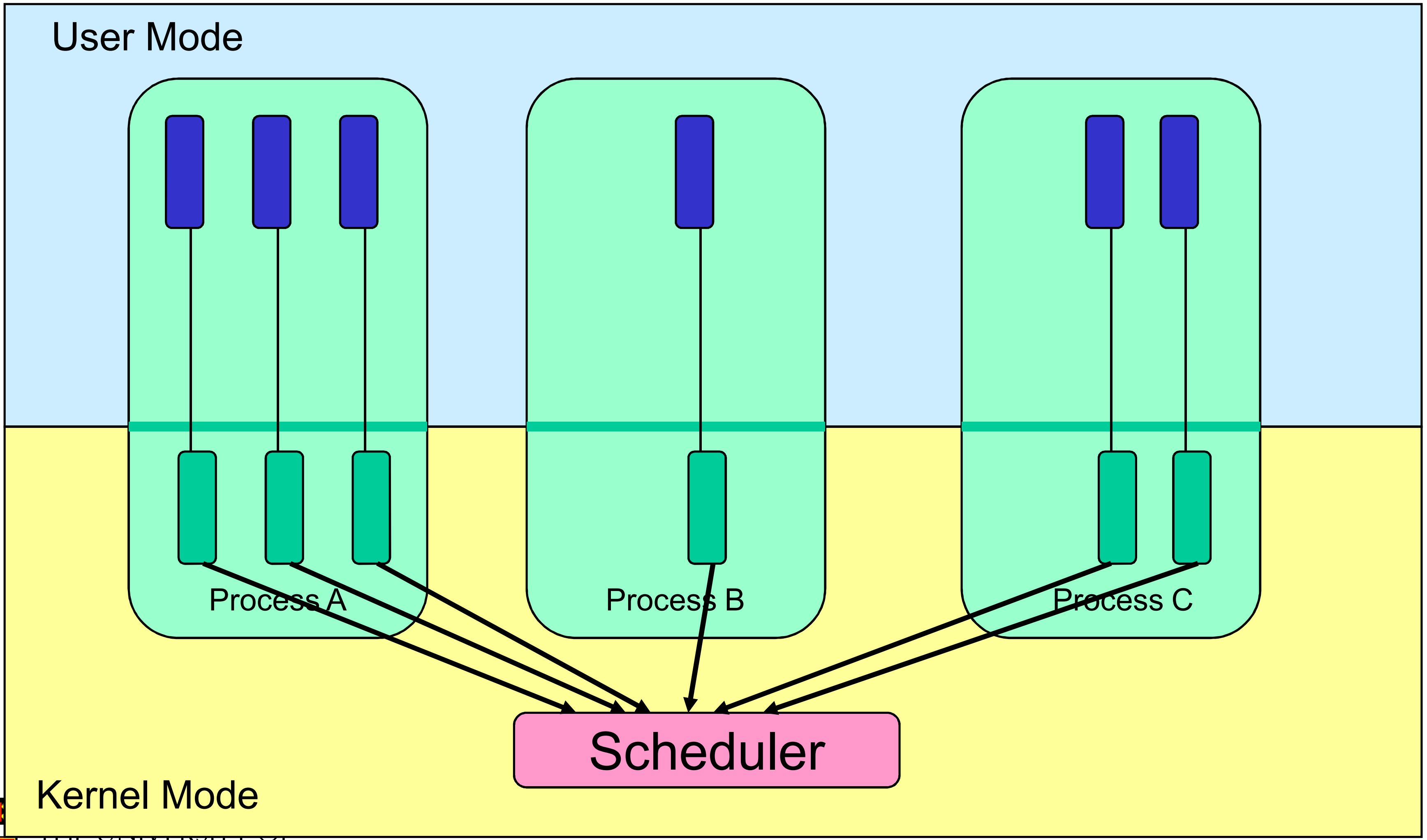
⇒OS must save all state that affects the thread

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

Simplified Explicit Thread Switch

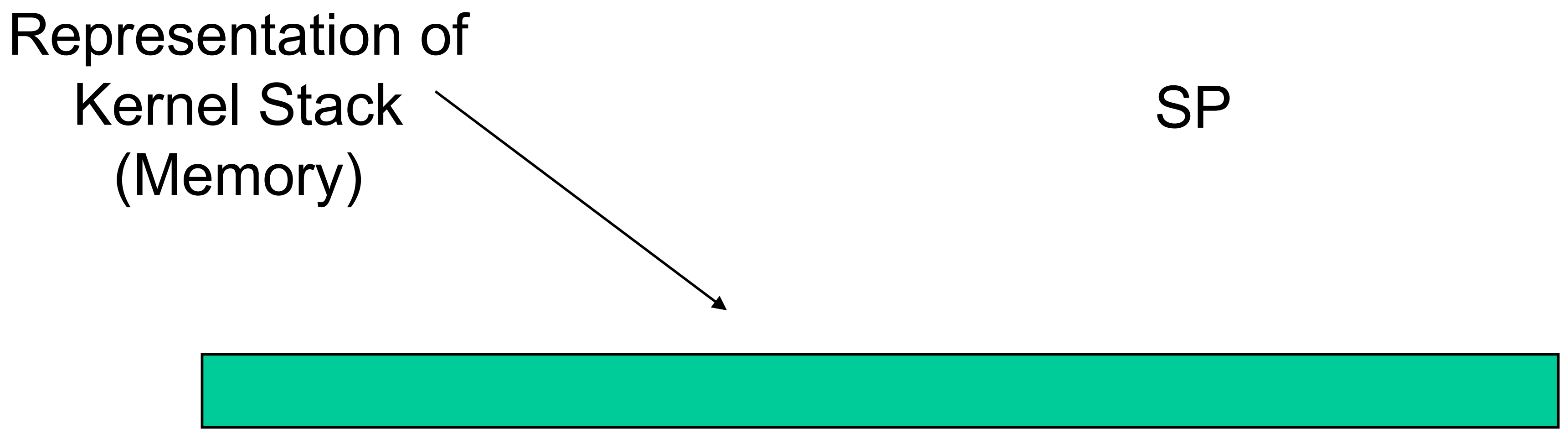


Kernel-Level Threads



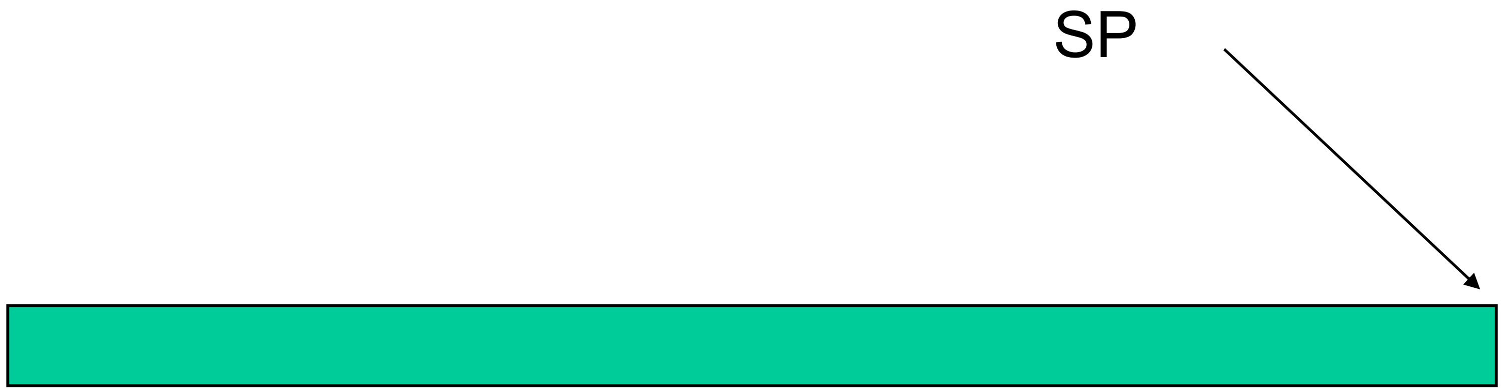
Example Context Switch

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



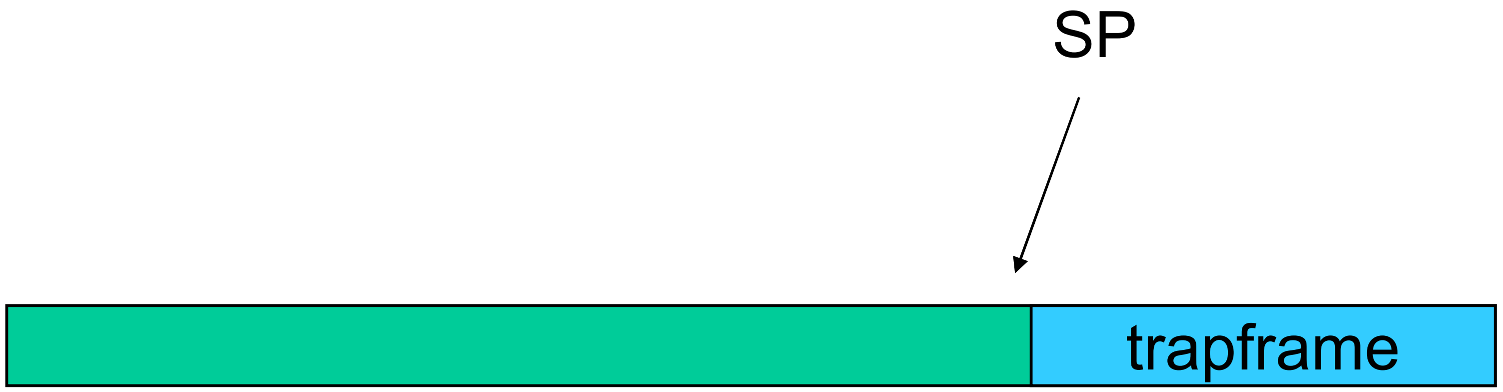
Example Context Switch

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



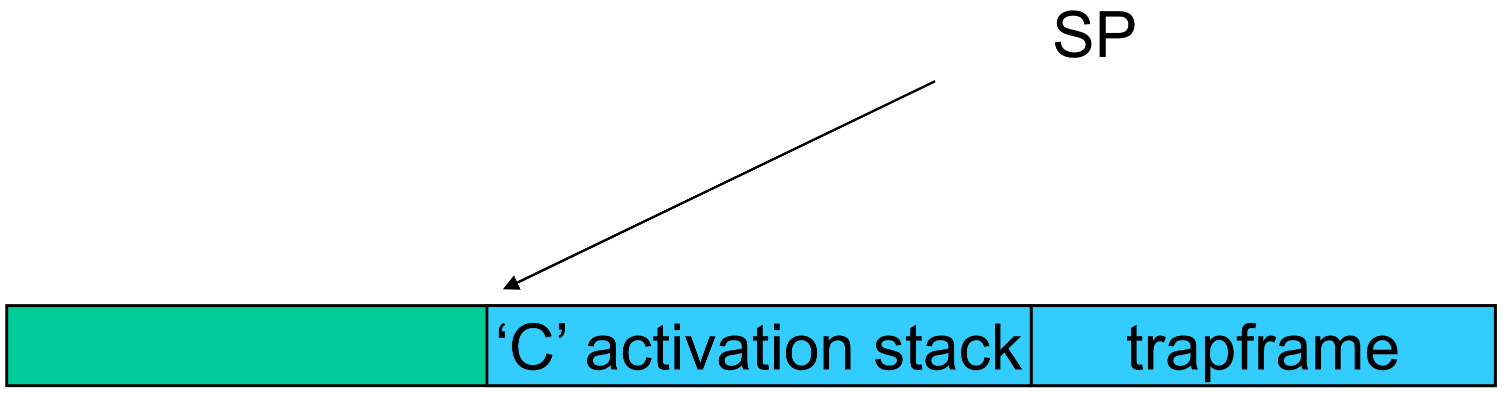
Example Context Switch

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



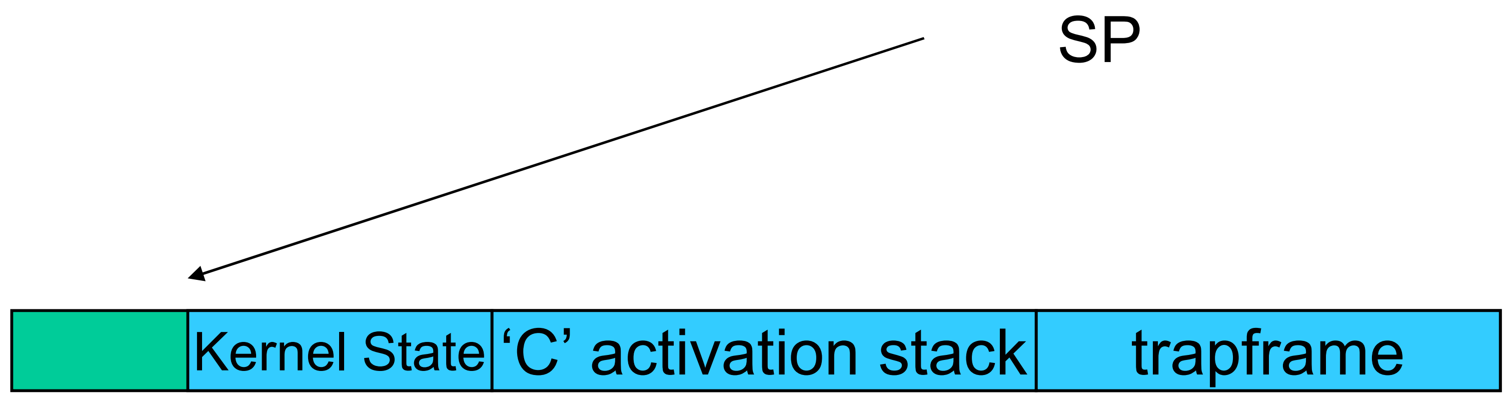
Example Context Switch

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



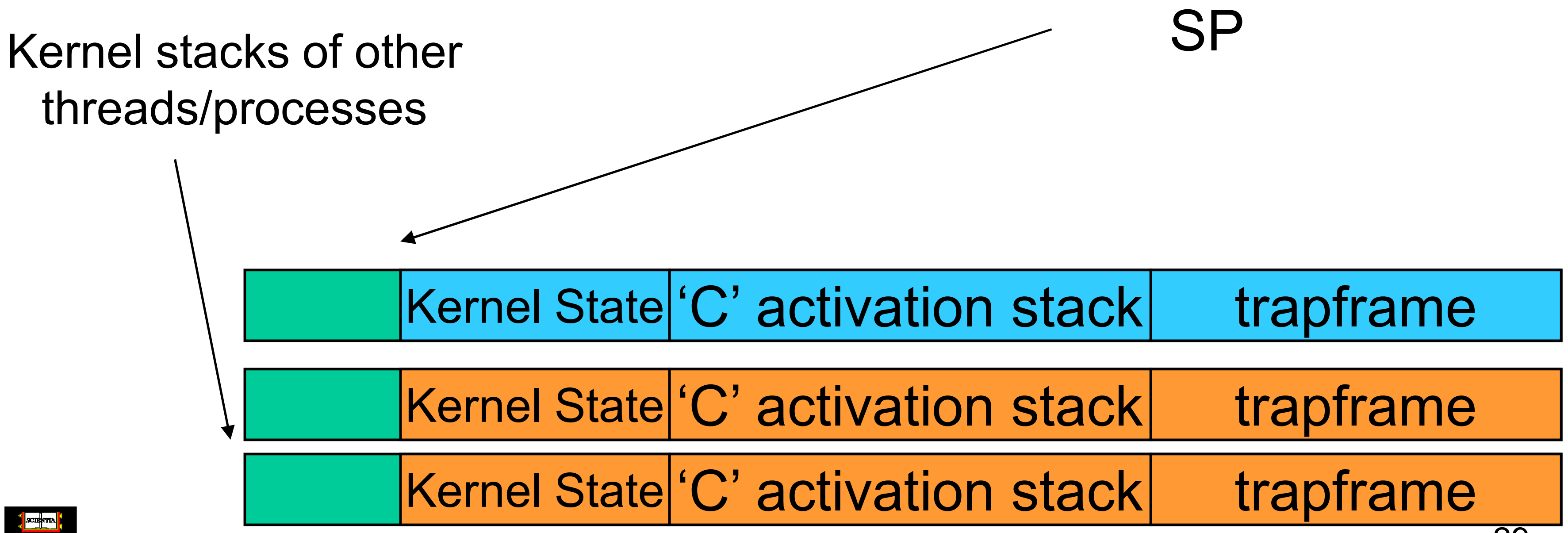
Example Context Switch

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



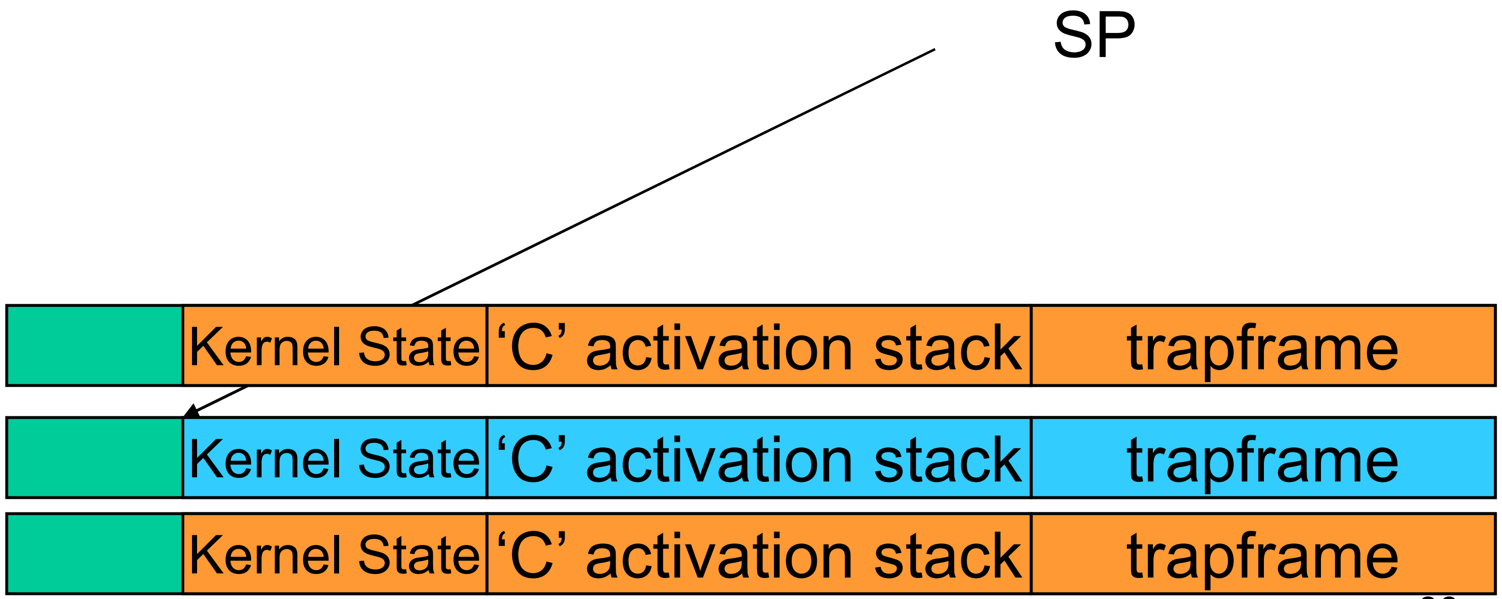
Example Context Switch

- 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



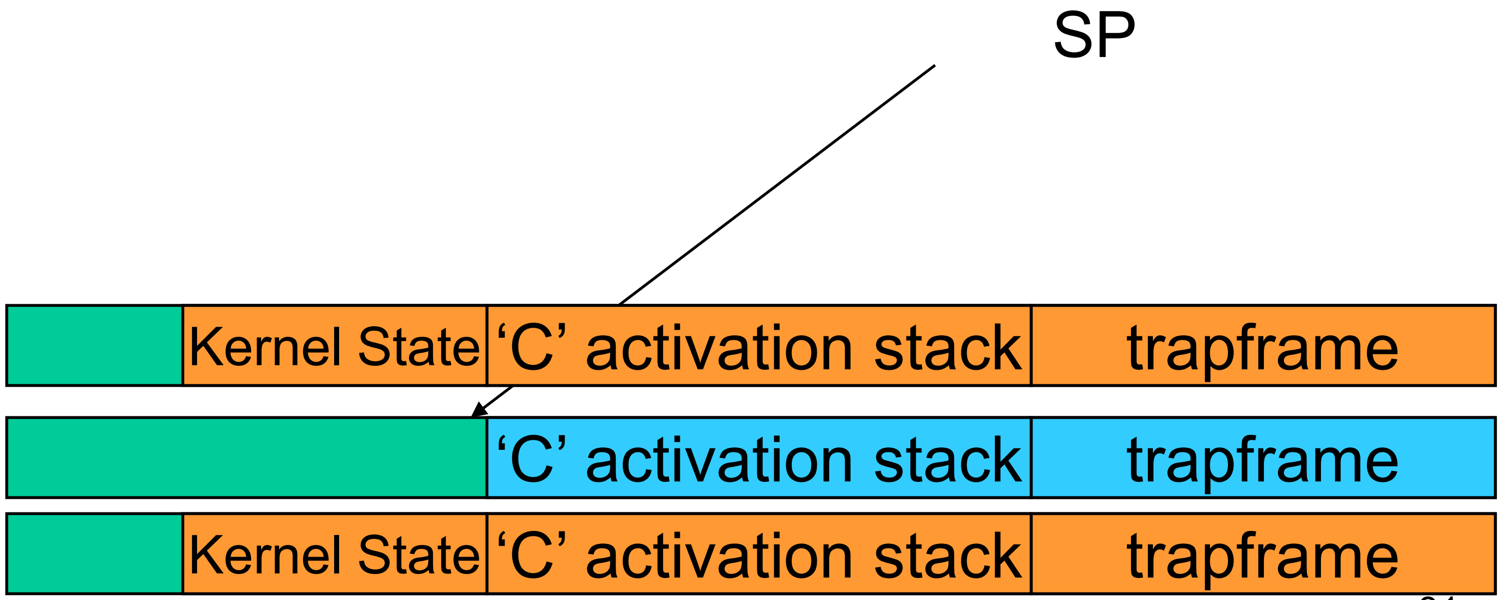
Example Context Switch

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



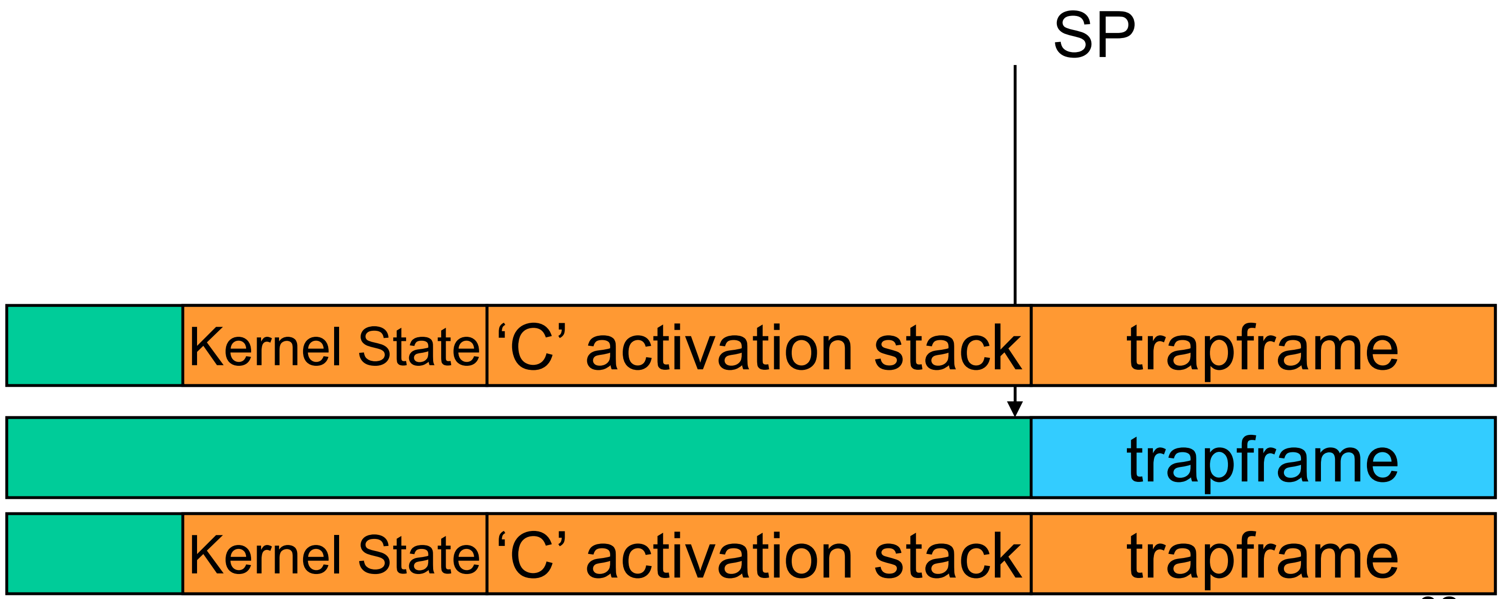
Example Context Switch

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



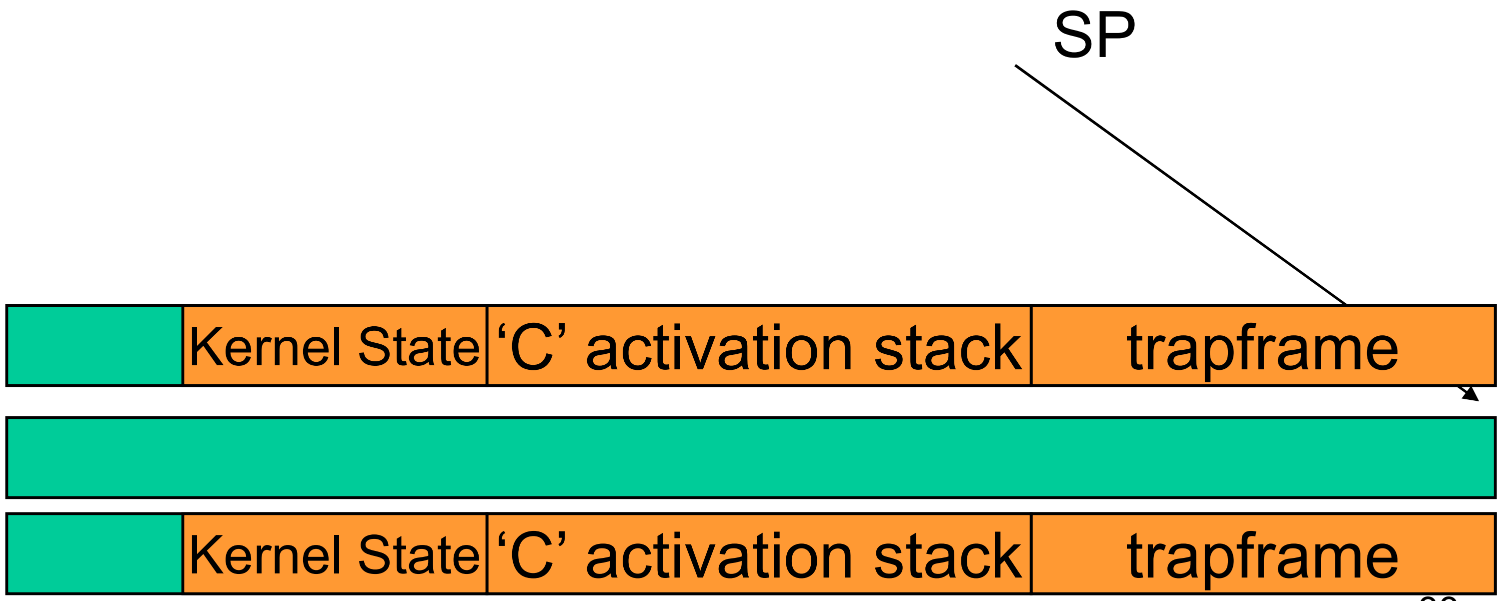
Example Context Switch

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



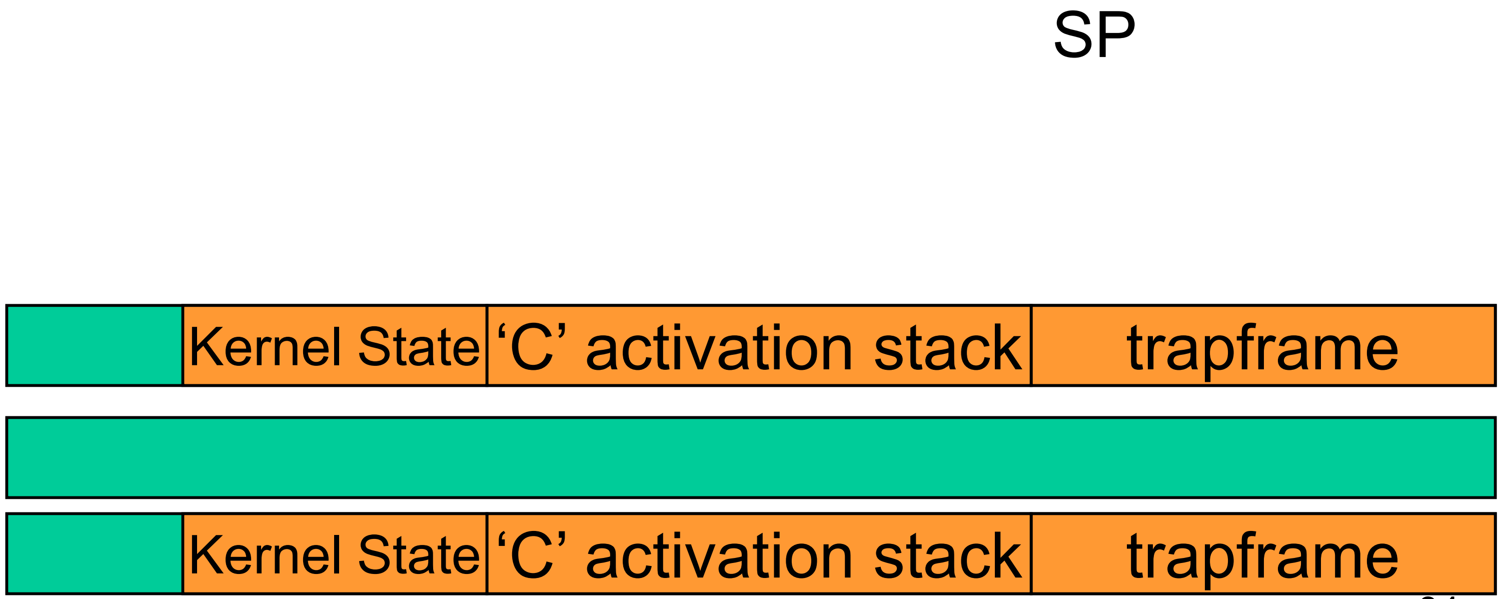
Example Context Switch

- The user-level context is restored



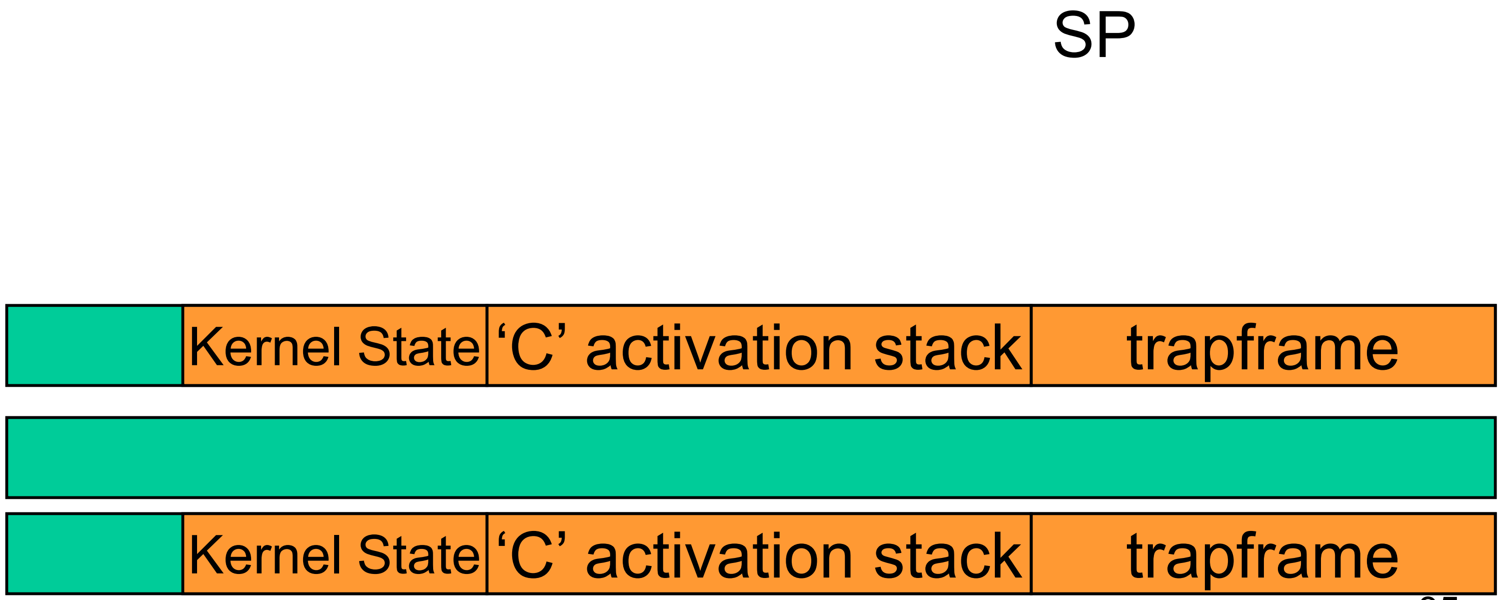
Example Context Switch

- The user-level SP is restored



The Interesting Part of a Thread Switch

- What does the “push kernel state” part do???



OS/161 md_switch

```
md_switch(struct pcb *old, struct pcb *nu)
{
   if (old==nu) {
        return;
   }
   /*
    * Note: we don't need to switch curspl, because splhigh()
    * should always be in effect when we get here and when we
    * leave here.
    */

   old->pcb_kstack = curkstack;
   old->pcb_ininterrupt = in_interrupt;

   curkstack = nu->pcb_kstack;
   in_interrupt = nu->pcb_ininterrupt;

   mips_switch(old, nu);

   }
```

OS/161 mips_switch

```
mips_switch:
   /*
    * a0 contains a pointer to the old thread's struct pcb.
    * a1 contains a pointer to the new thread's struct pcb.
    *
    * The only thing we touch in the pcb is the first word, which
    * we save the stack pointer in. The other registers get saved
    * on the stack, namely:
    *
    *      s0-s8
    *      gp, ra
    *
    * The order must match arch/mips/include/switchframe.h.
    */

   /* Allocate stack space for saving 11 registers. 11*4 = 44 */
   addi sp, sp, -44
```

OS/161 mips_switch

```
/* Save the registers */
   sw    ra, 40(sp)
   sw    gp, 36(sp)
   sw    s8, 32(sp)
   sw    s7, 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 pcb */
   sw    sp, 0(a0)
```

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

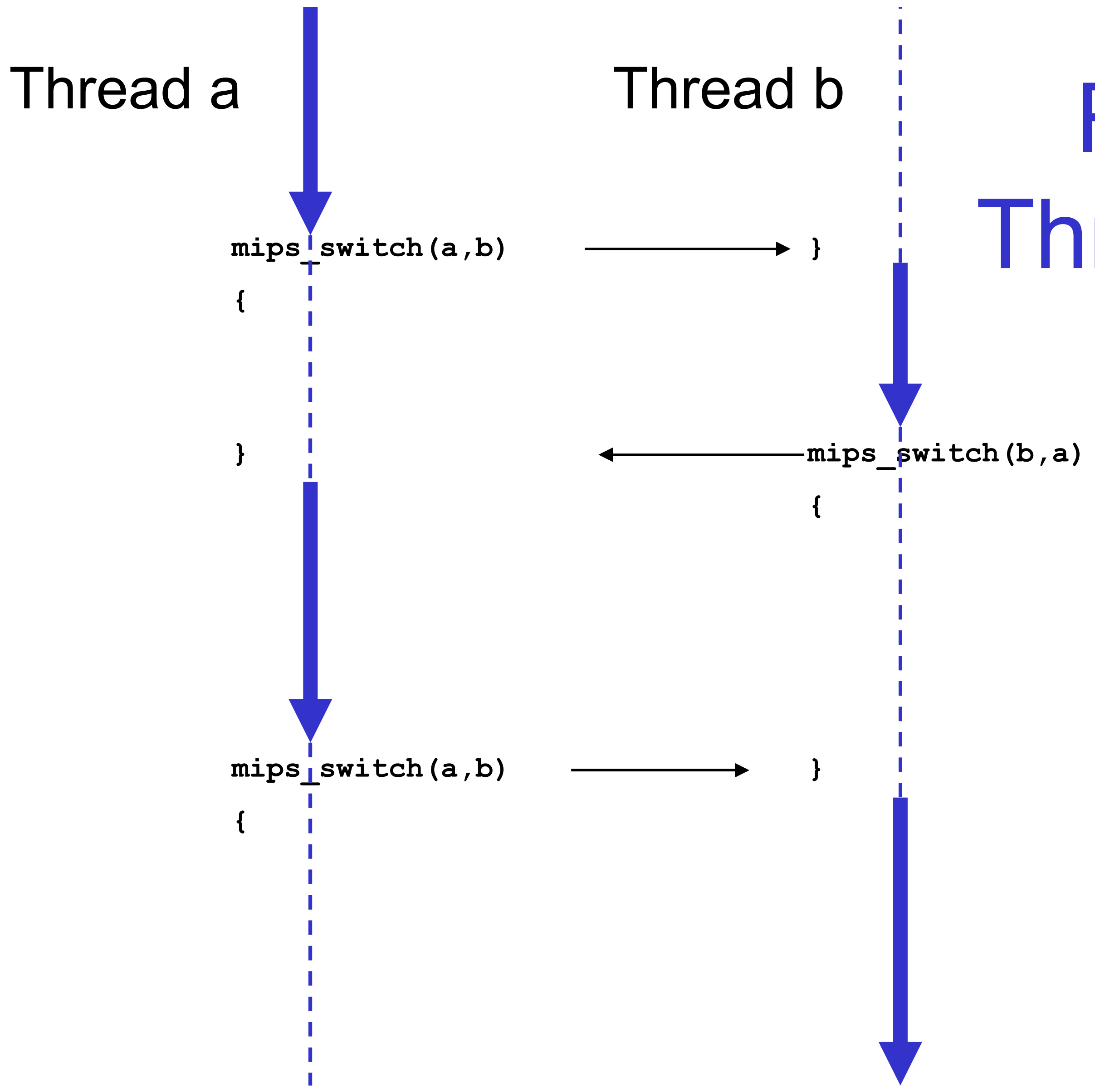
OS/161 mips_switch

```
/* Get the new stack pointer from the new pcb */
   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     s7, 28(sp)
   lw     s8, 32(sp)
   lw     gp, 36(sp)
   lw     ra, 40(sp)
   nop                         /* delay slot for load */

   /* and return. */
   j ra
   addi   sp, sp, 44           /* in delay slot */
   .end mips_switch
```

Revisiting Thread Switch



Thread a
Thread b
mips_switch(a,b)
{
}
}
mips_switch(b,a)
{
mips_switch(a,b)
{
}