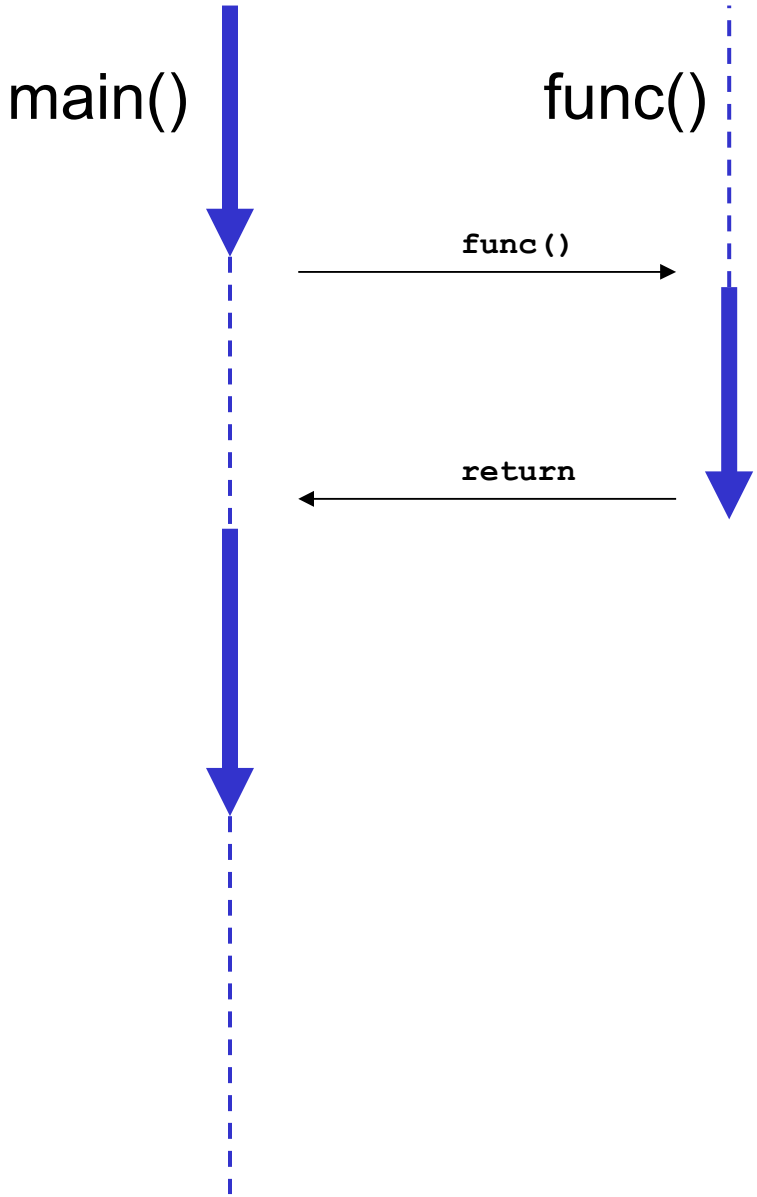
Events, Co-routines, Continuations and Threads
-
OS (and application) Execution Models



Sequential program

- Single flow of control
- Exclusive access to global state
- Execution state is maintained by the language runtime
  - local variables
  - stack
  - control-flow constructs

System Building

- General purpose systems need to deal with
  - Many activities
    - potentially overlapping
    - may be interdependent
  - Activities that depend on external phenomena
    - may require waiting for completion (e.g. disk read)
    - reacting to external triggers (e.g. interrupts)
- Need a systematic approach to system structuring
    - concurrency and synchronisation
    - scheduling
    - shared data

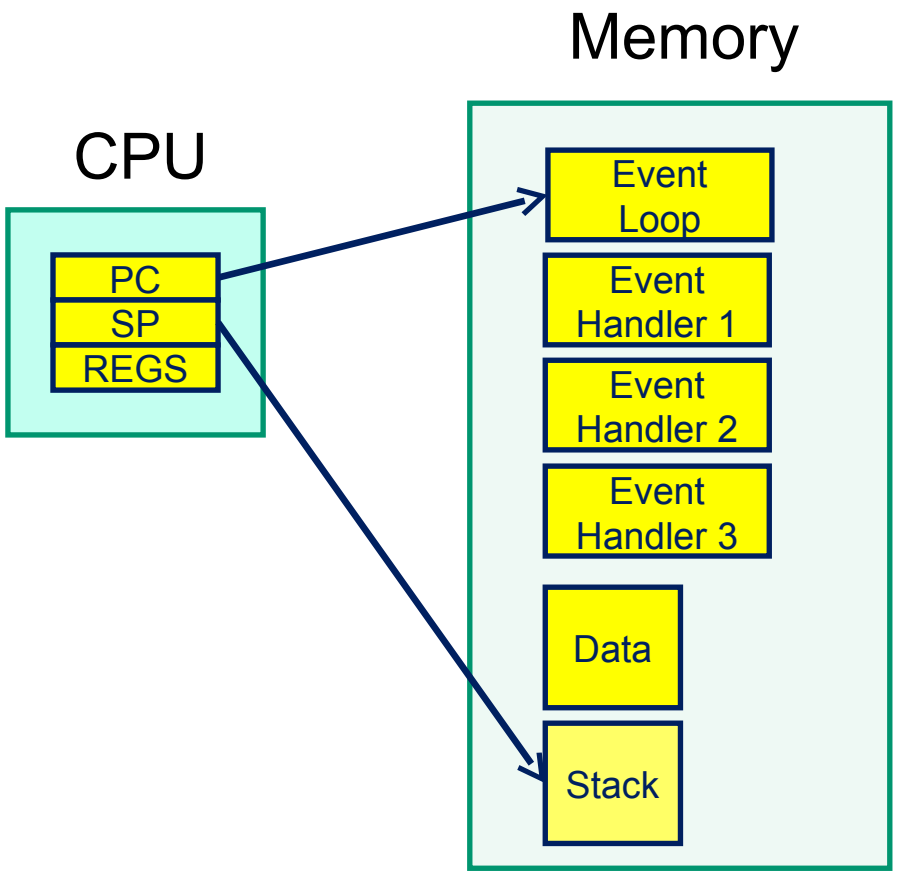
Threads, Events, Continuations

- In the OS course you learned about threads and synchronisation
- Today:
  - Alternative execution models
  - More about threads

Events

- External entities generate (post) events.
  - keyboard presses, mouse clicks, system calls
- *Event loop* waits for events and calls an appropriate *event handler.*
  - common paradigm for GUIs
- *Event handler* is a function that runs until completion and returns to the *event loop.*

Event Model



- The event model only requires a single stack
  - All event handlers must return to the event loop
    - No blocking
    - No yielding
- No preemption of handlers
  - Handlers generally short lived
- No concurrency issues within a handler

Stack ripping

```
example(arg1, arg2) {
    int x;
    P1(arg1, arg2);
    if (need_to_block) {
        perform_io();
        P2(arg2);
    } else {
        P3();
    }
}
```

```
example(arg1, arg2) {
    int x
    P1(arg1, arg2);
    if (need_to_block) {
        save_args_and_locals();
        start_io();
        return;
    } else {
        P3();
    }
}
io_complete() {
    recover_args_and_locals();
    P2(recovered arg2);
    thread_syscall_return(SUCCESS);
}
```

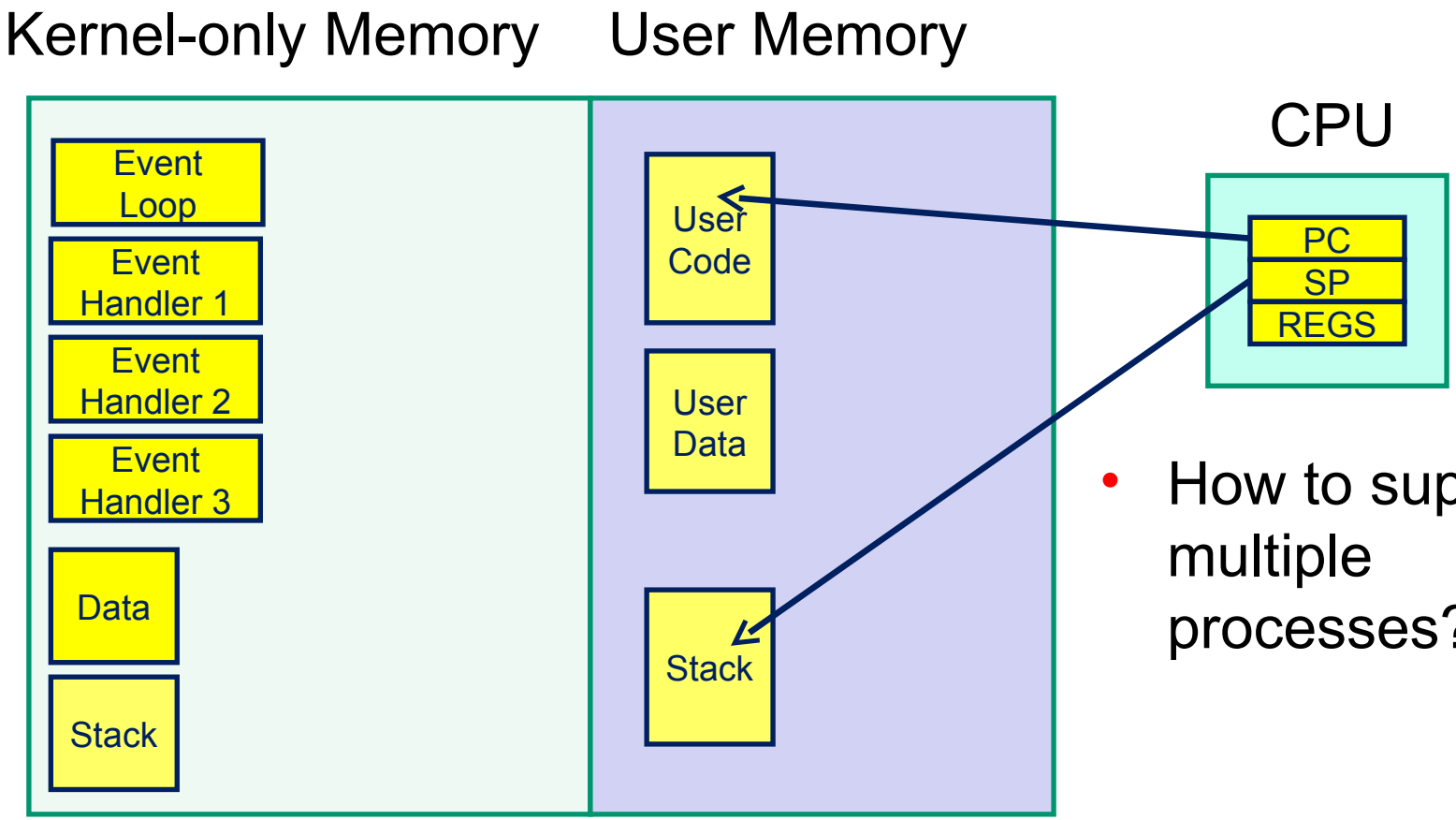
Stack ripping

- No language support for maintaining control flow across event handlers:
  - No stack:
    - what happens if a blocking operation occurs in a library function?
  - No local variables
  - No control flow constructs
    - what if a blocking operation occurs inside a loop?

Event-based design guidelines

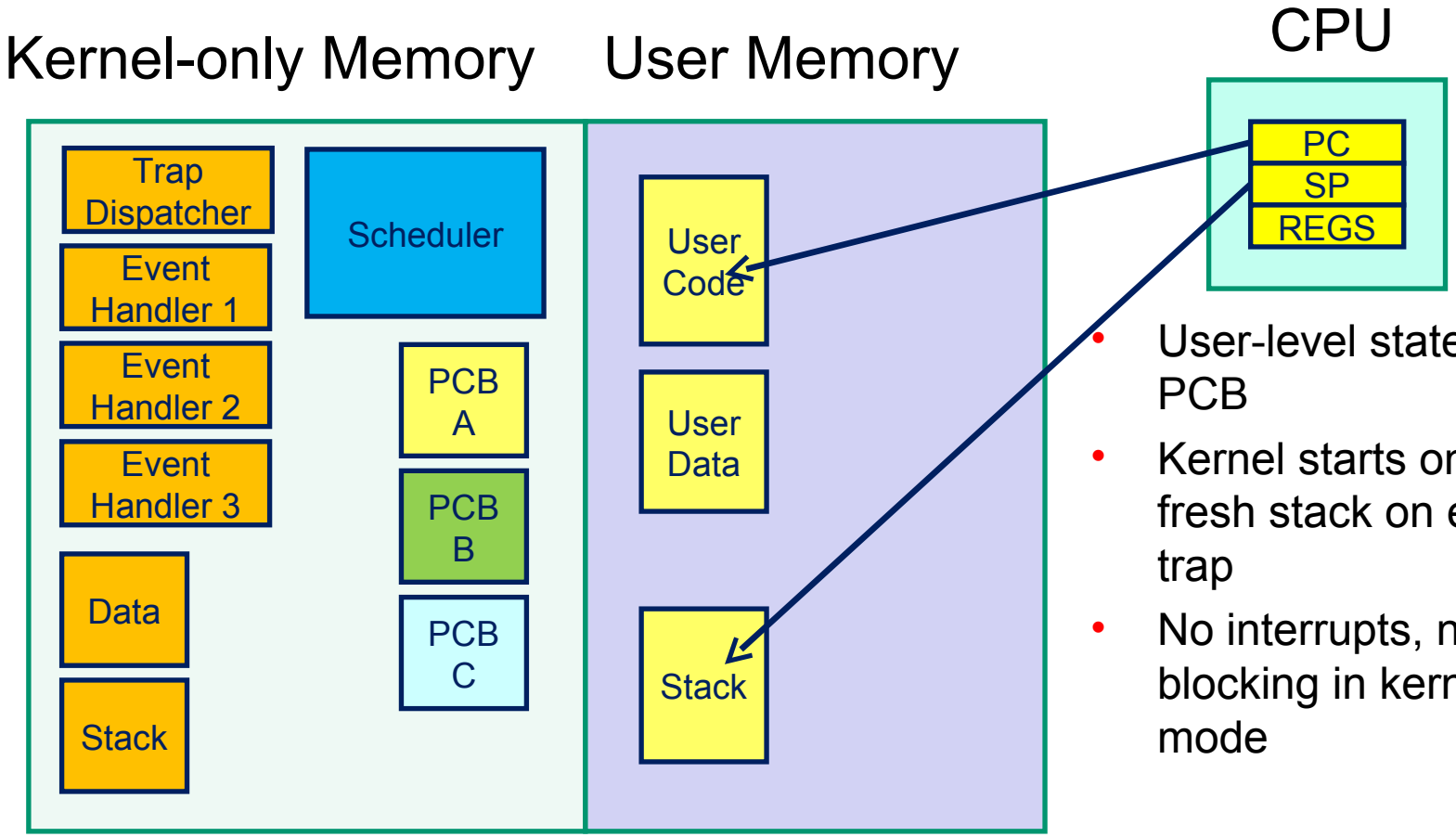
- Events can be simpler and more efficient than threads.
- Use with caution:
  - Usually inappropriate for deep software stacks.
  - Refactoring is hard.
    - Design control flow structure and communication protocols in advance

Event-based kernel on CPU with protection



- How to support multiple processes?

Event-based kernel on CPU with protection

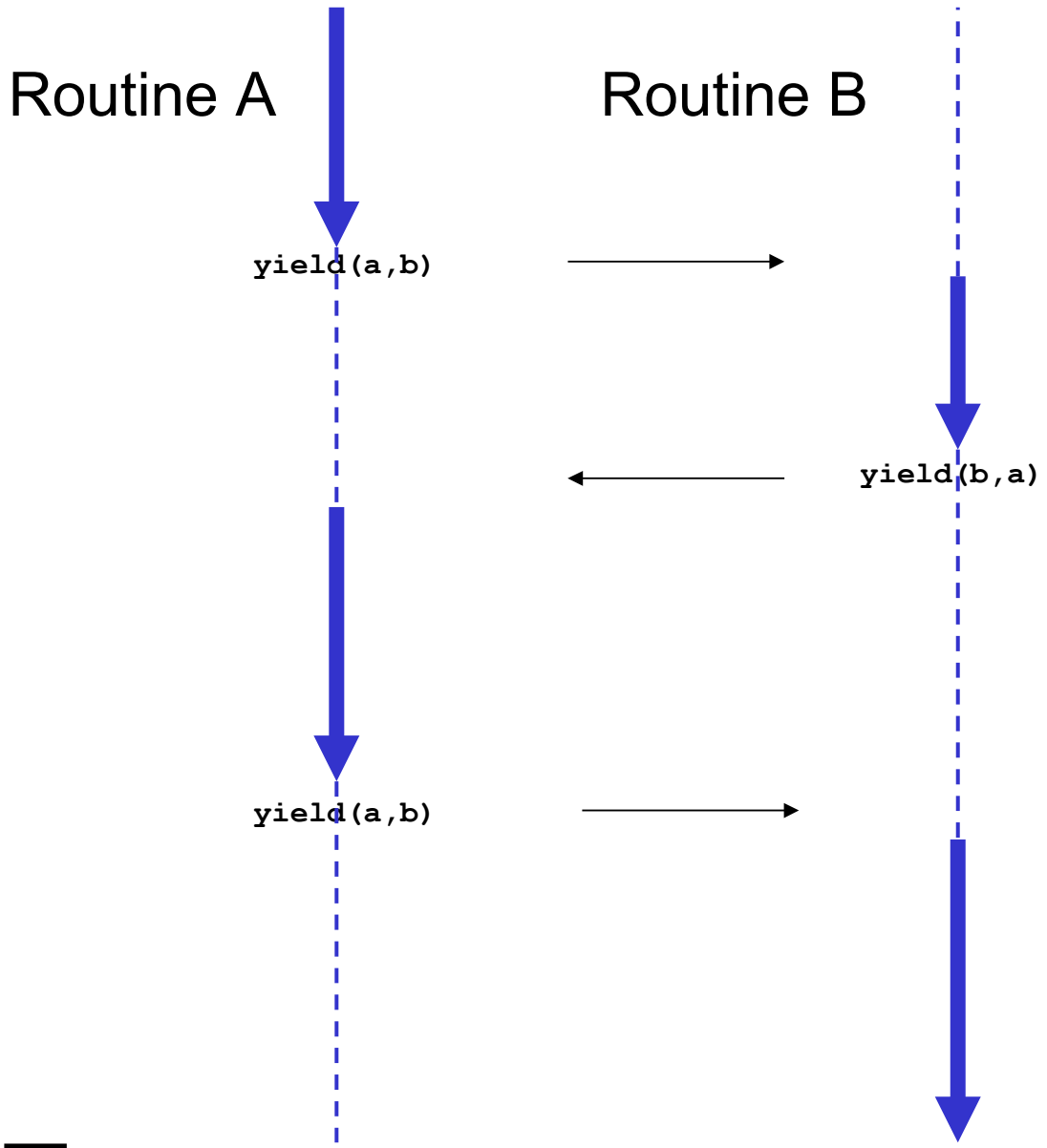


- User-level state in PCB
- Kernel starts on fresh stack on each trap
- No interrupts, no blocking in kernel mode

Co-routines

- A subroutine with extra entry and exit points
- Via yield()
  - supports long running subroutines
  - subtle variations (yieldto, asymmetric and symmetric)
- Also called *Fibers*

Yield



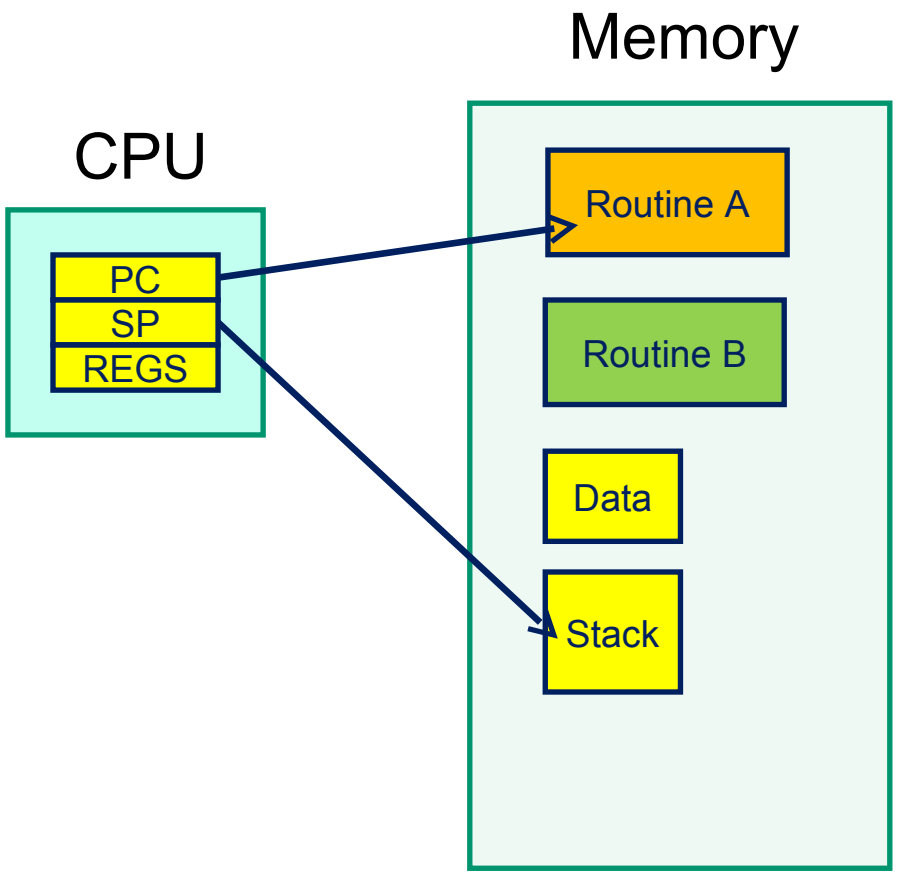
Co-routine example

```
var q := new queue
coroutine produce
    loop
        while q is not full
            create some new items
            add the items to q
        yield to consume

coroutine consume
    loop
        while q is not empty
            remove some items from q
            use the items
        yield to produce
```

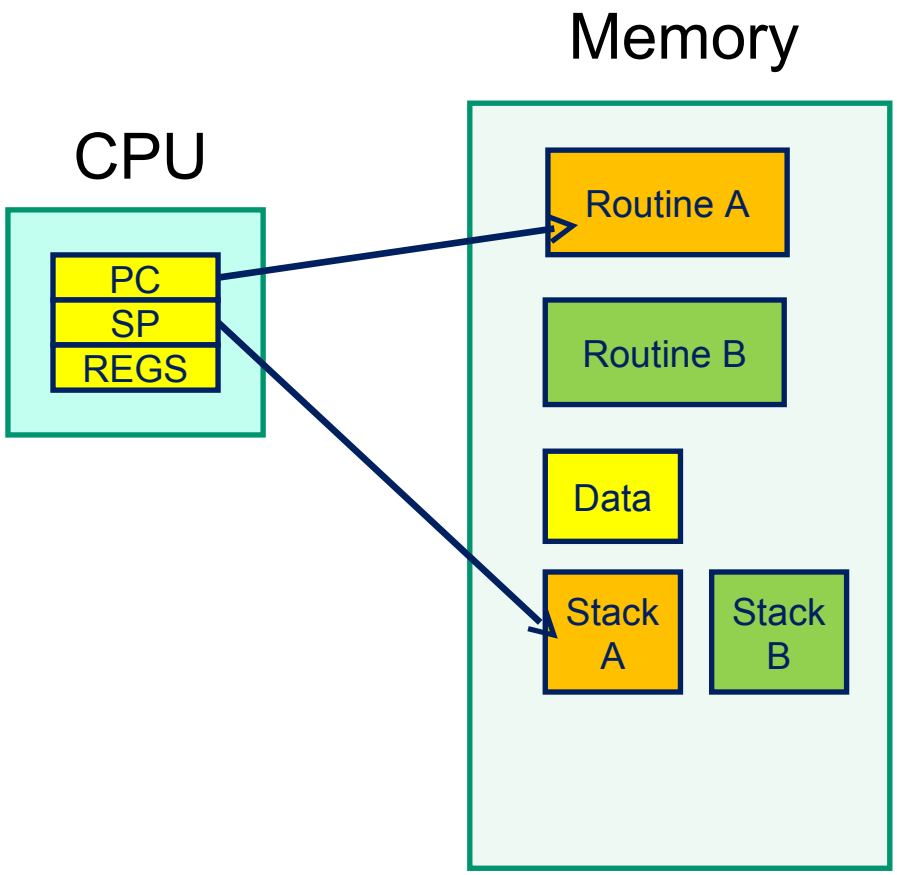
Derived from http://en.wikipedia.org/wiki/Coroutine

Co-routines



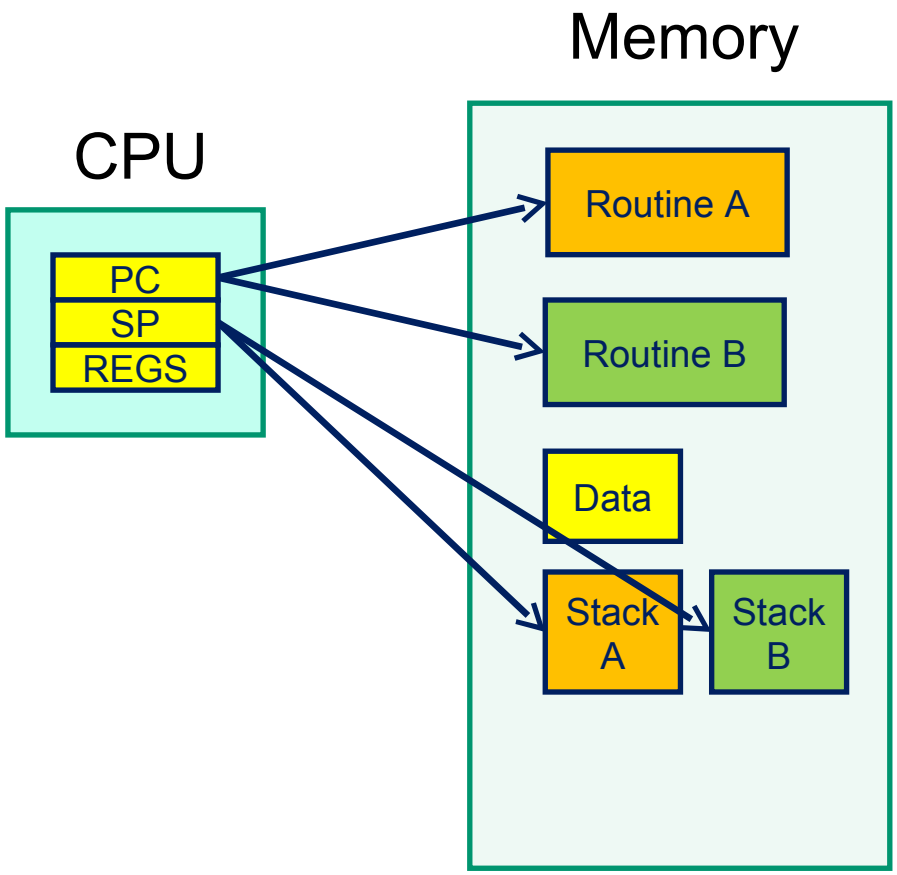
- yield() saves state of routine A and starts routine B
  - or resumes B's state from its previous yield() point.
- No preemption
- No concurrency issues/races as globals are exclusive between yields()
- Implementation strategy?

Co-routines



- Usually implemented with a stack per routine
- Preserves current state of execution of the routine

Co-routines



- Routine A state currently loaded
- Routine B state stored on stack
- Routine switch from A → B
  - saving state of A
    - regs, sp, pc
  - restoring the state of B
    - regs, sp, pc

A hypothetical yield()

```
yield:
   /*
    * a0 contains a pointer to the previous routine's struct.
    * a1 contains a pointer to the new routine's struct.
    *
    * The registers get saved on the stack, namely:
    *
    *      s0-s8
    *      gp, ra
    *
    *
    */

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

```
/* 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

```
/* 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
```

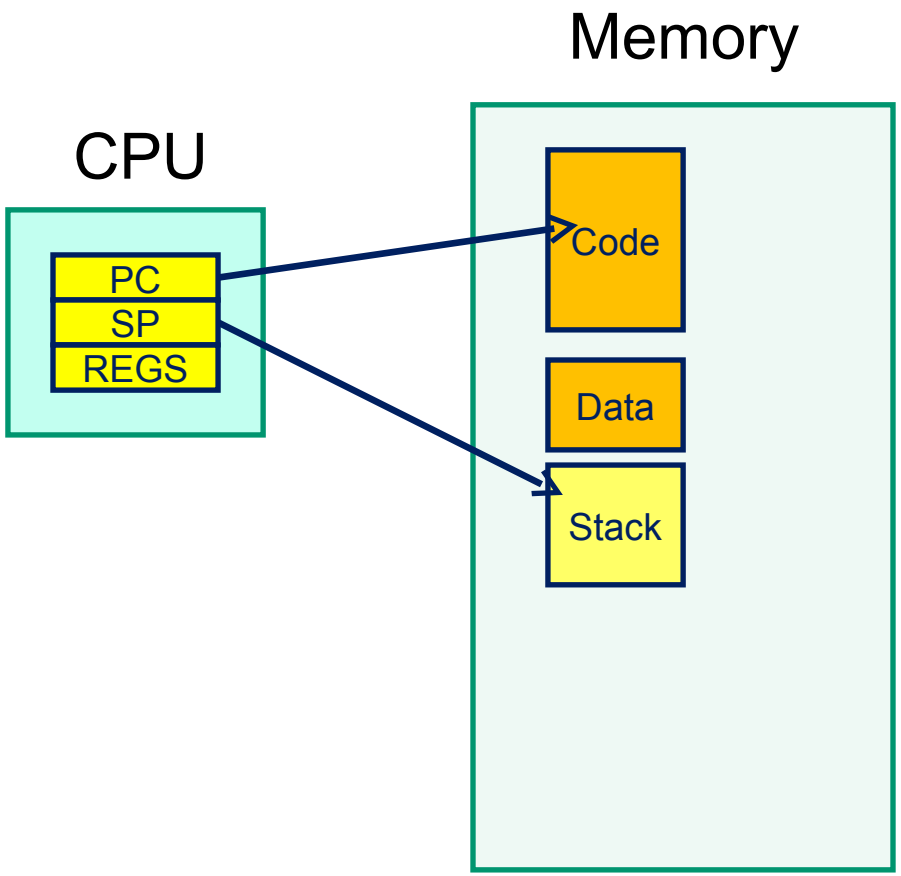
Coroutines

- What about subroutines combined with coroutines
  - i.e. what is the issue with calling subroutines?
- Subroutine calling might involve an implicit yield()
  - potentially creates a race on globals
    - either understand where all yields lie, or
    - cooperative multithreading

Cooperative Multithreading

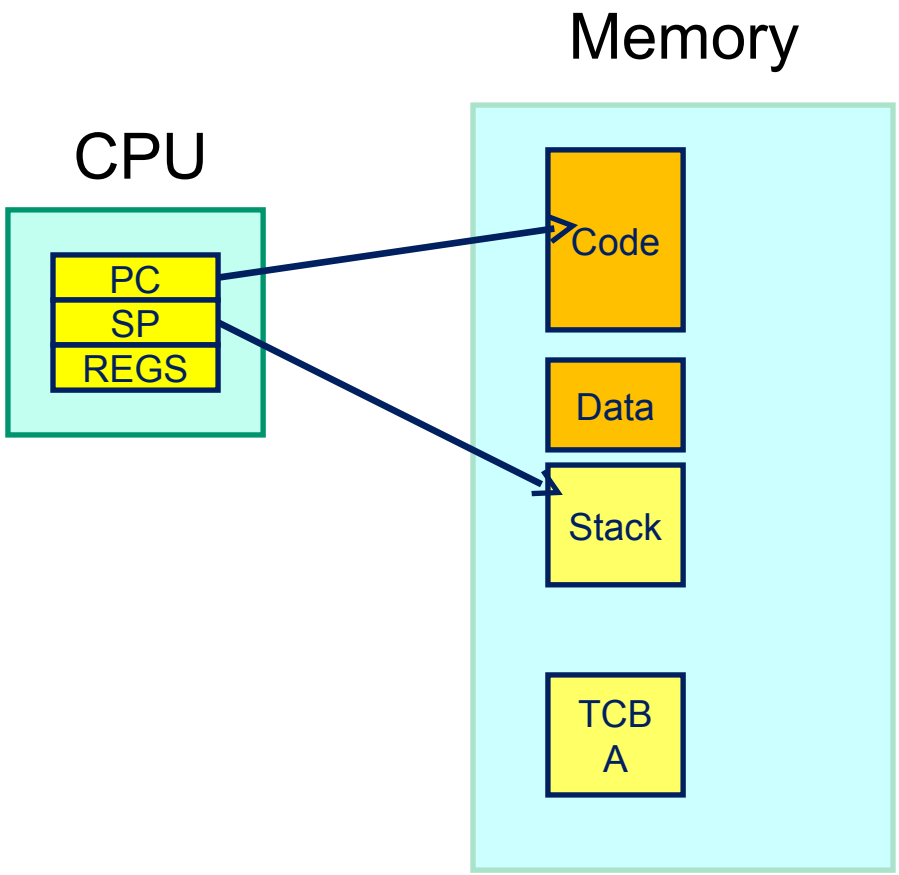
- Also called green threads
- Conservative assumes a multithreading model
  - i.e. uses synchronisation to avoid races,
  - and makes no assumption about subroutine behaviour
    - it can potentially yield()

A Thread



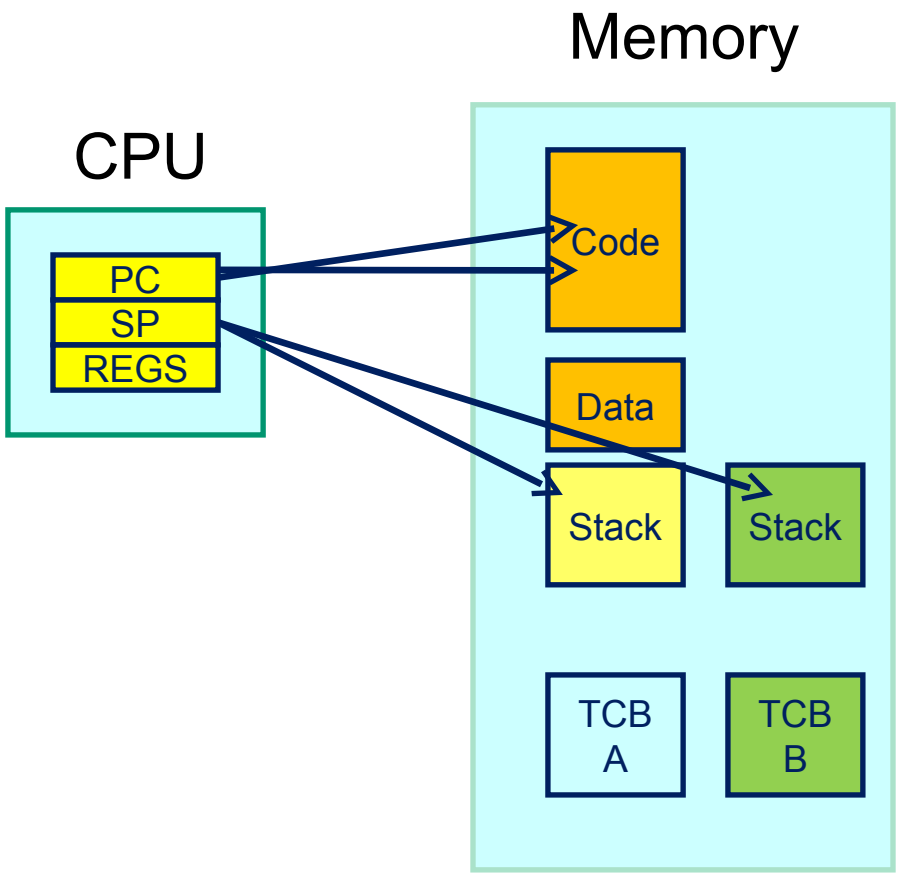
- Thread attributes
  - processor related
    - memory
    - program counter
    - stack pointer
    - registers (and status)
  - OS/package related
    - state (running/blocked)
    - identity
    - scheduler (queues, priority)
    - etc…

Thread Control Block



- To support more than a single thread we need to store thread state and attributes
- Stored in thread control block
  - also indirectly in stack

Thread A and Thread B



- Thread A state currently loaded
- Thread B state stored in TCB B
- Thread switch from A → B
  - saving state of thread a
    - regs, sp, pc
  - restoring the state of thread B
    - regs, sp, pc
- Note: registers and PC can be stored on the stack, and only SP stored in TCB

Approx OS

```
mi_switch()
{
  struct thread *cur, *next;
  next = scheduler();

/* update curthread */
  cur = curthread;
  curthread = next;

/*
 * Call the machine-dependent code that actually does the
 * context switch.
 */
  md_switch(&cur->t_pcb, &next->t_pcb);

/* back running in same thread */

}
```

Note: global variable curthread

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

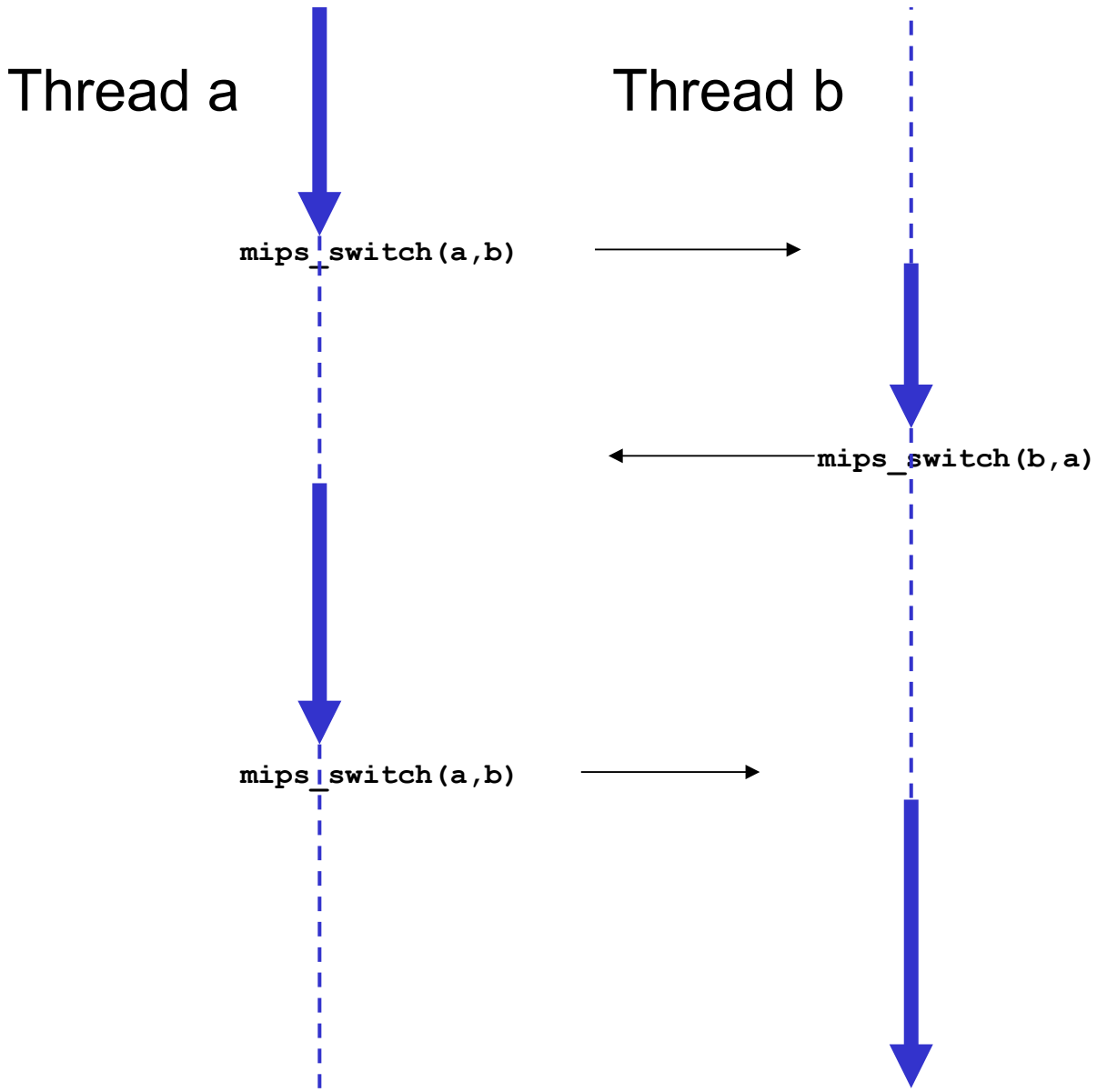
OS/161 mips_switch

```
/* 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
```

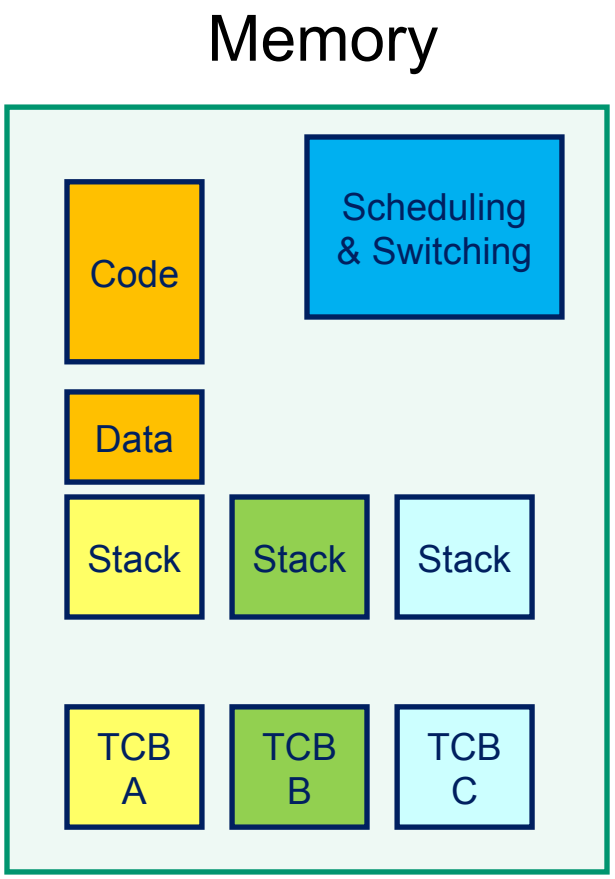
Thread Switch



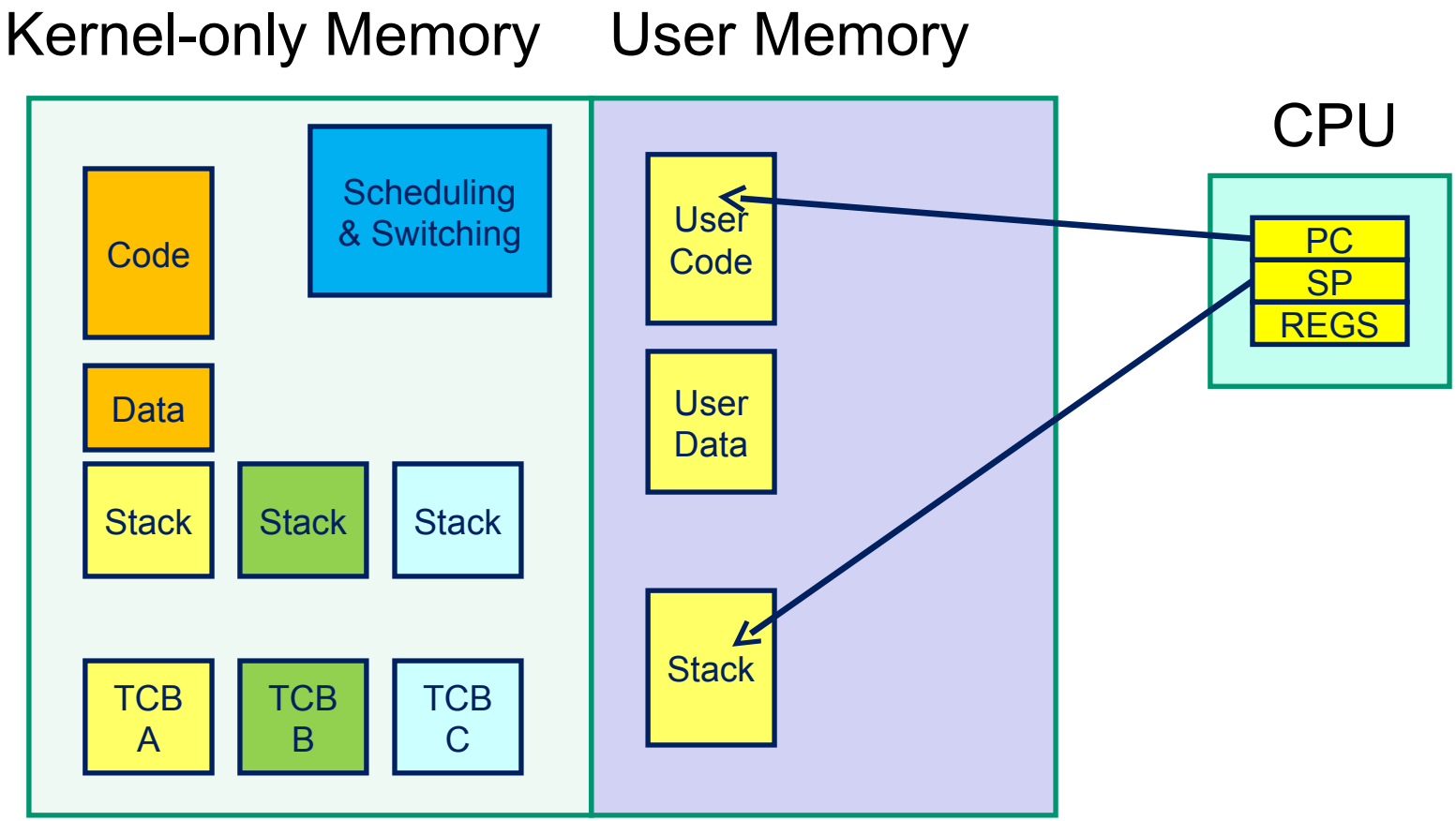
Preemptive Multithreading

- Switch can be triggered by asynchronous external event
  - timer interrupt
- Asynch event saves current state
  - on current stack, if in kernel (nesting)
  - on kernel stack or in TCB if coming from user-level
- call thread_switch()

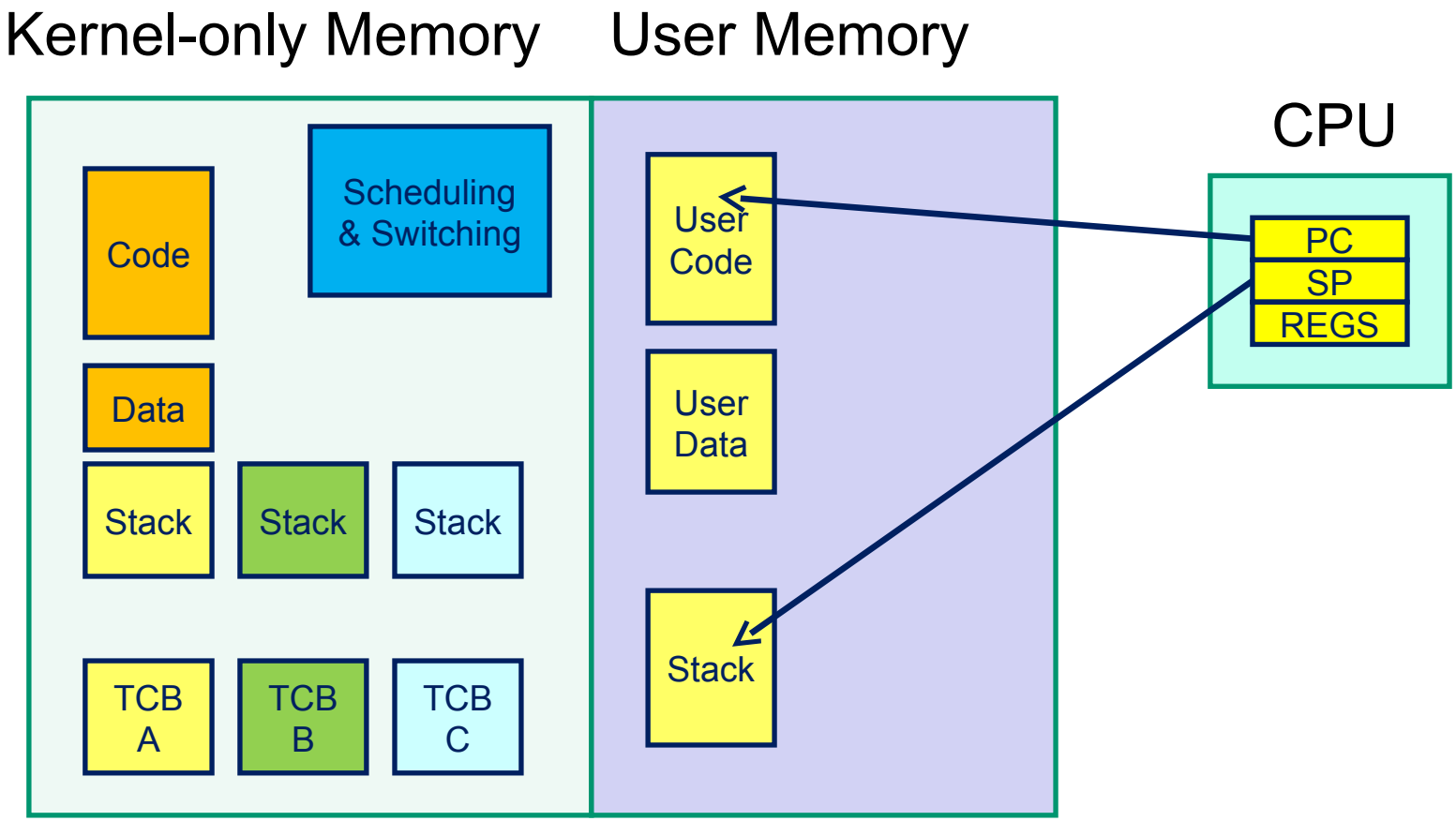
Threads on simple CPU



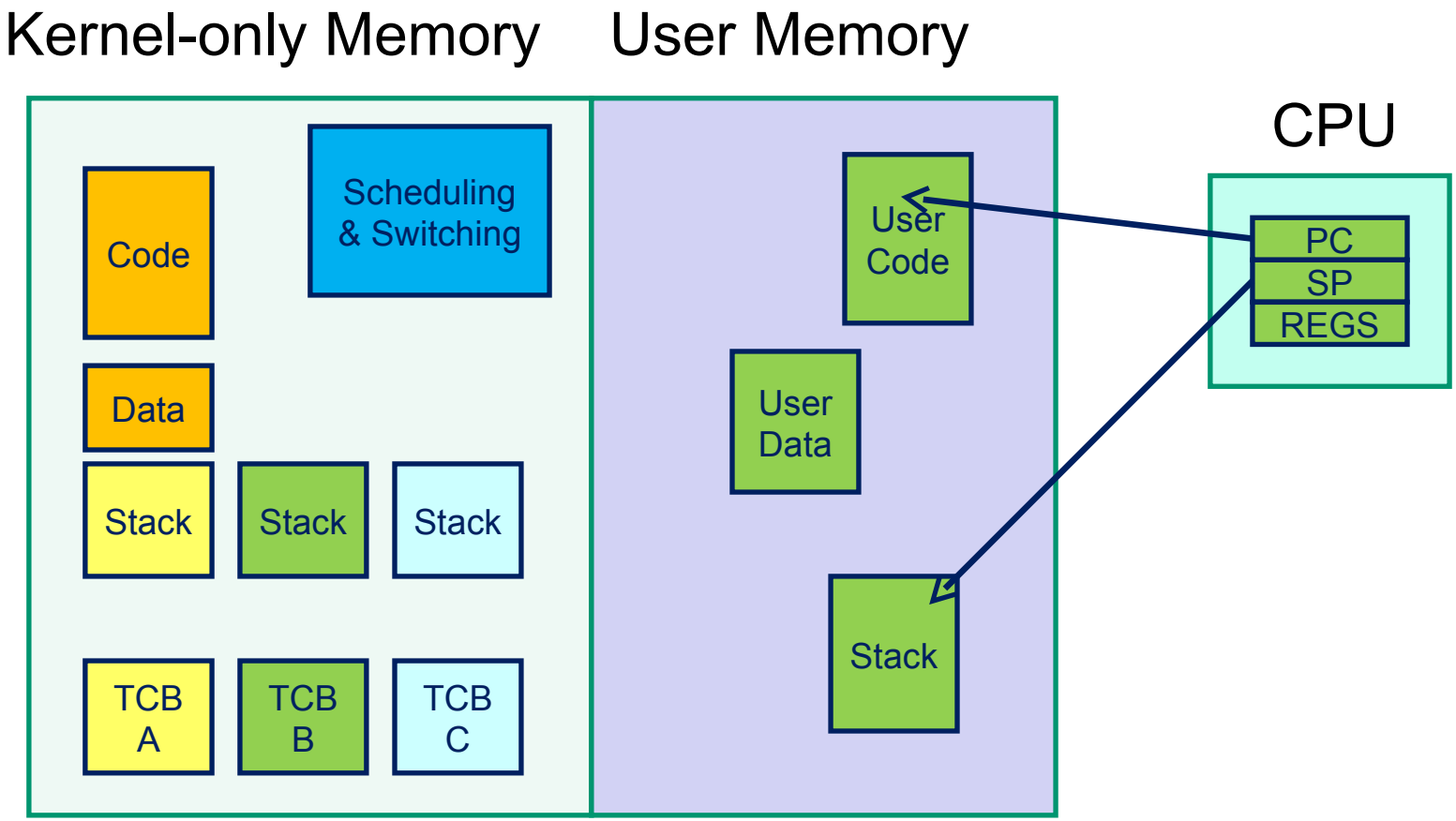
Threads on CPU with protection



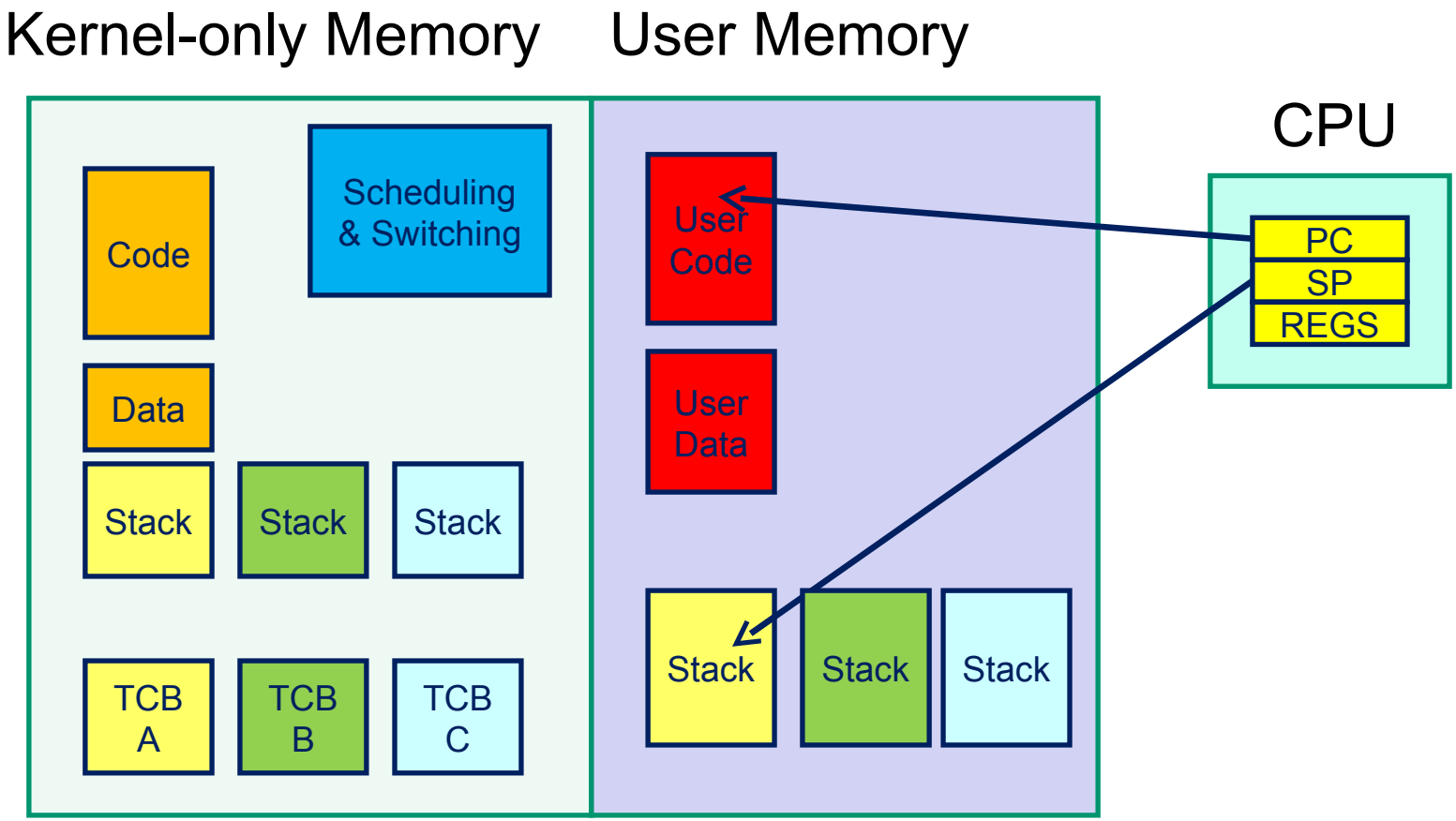
Switching Address Spaces on Thread Switch = Processes



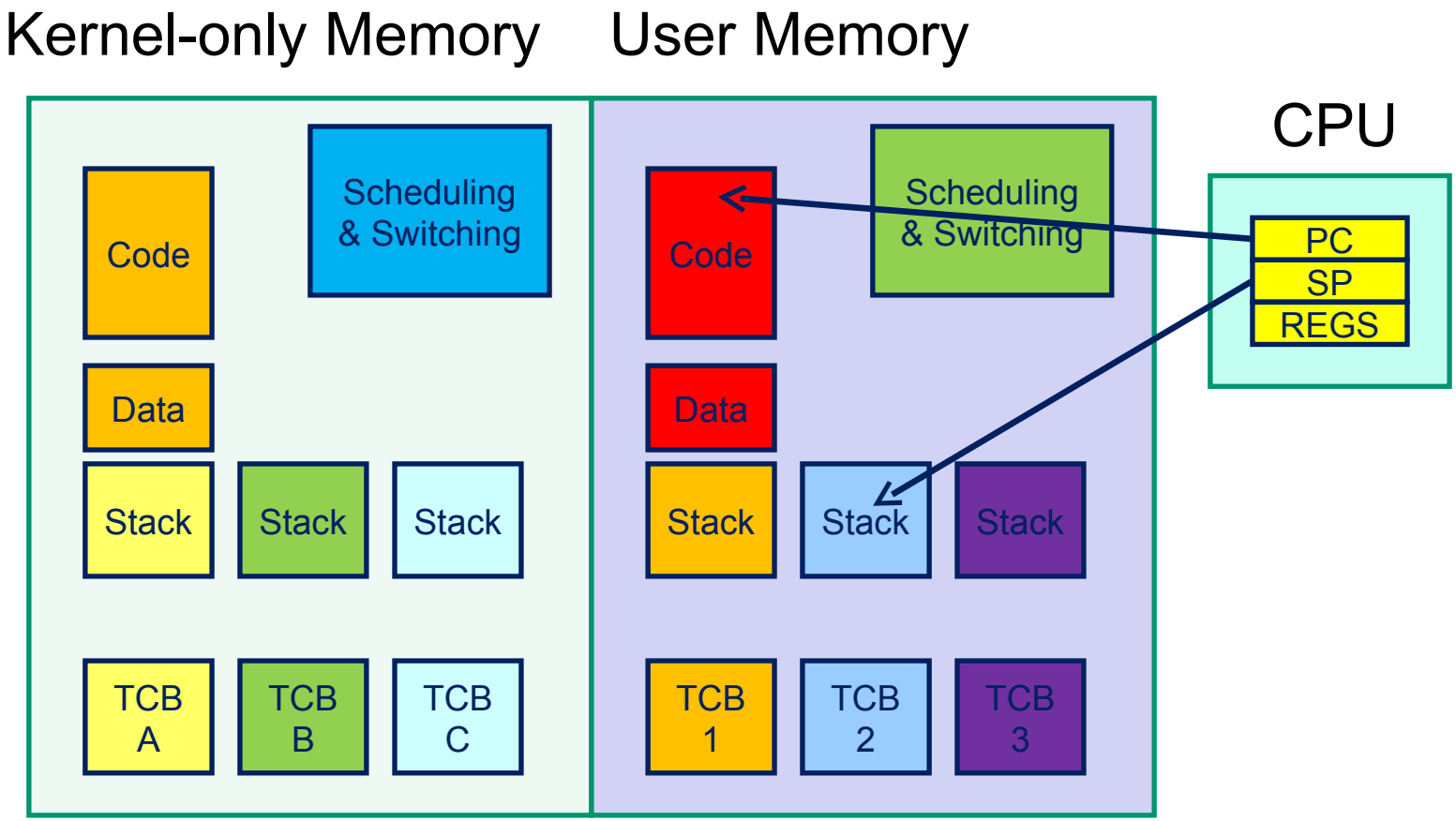
Switching Address Spaces on Thread Switch = Processes



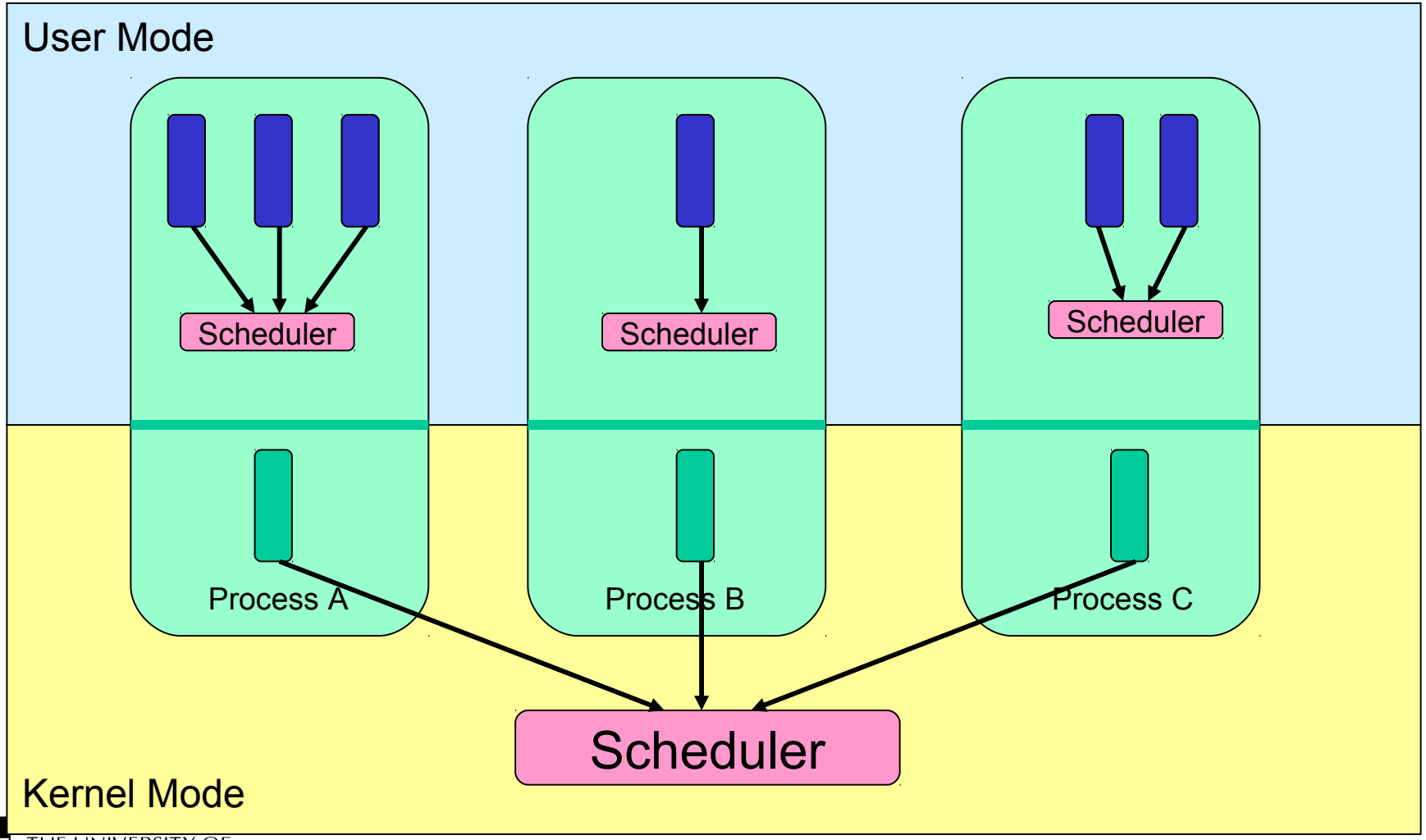
What is this?



What is this?



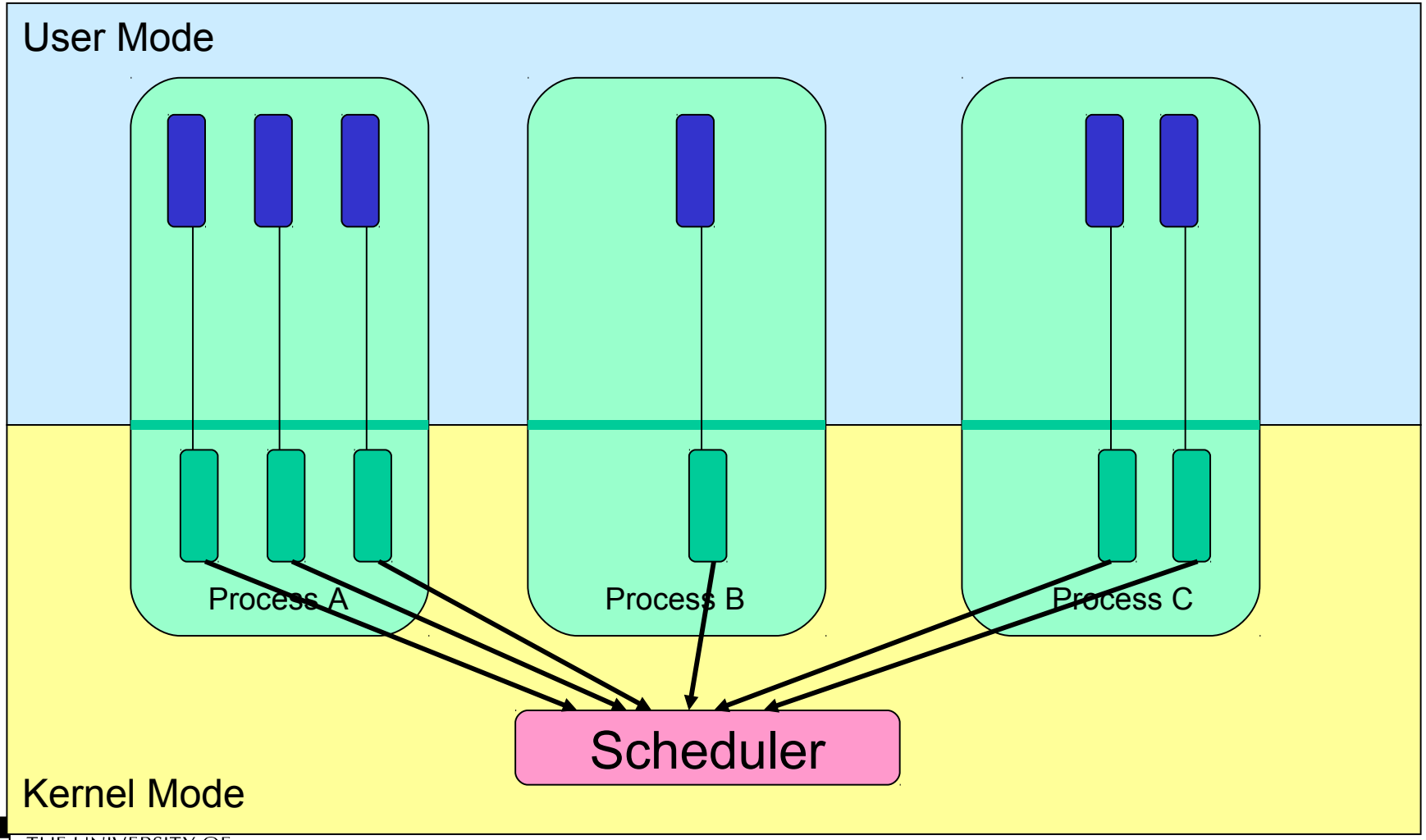
User-level Threads



User-level Threads

- ✓ Fast thread management (creation, deletion, switching, synchronisation…)
- ✘ Blocking blocks all threads in a process
  - Syscalls
  - Page faults
- ✘ No thread-level parallelism on multiprocessor

Kernel-Level Threads



Kernel-level Threads

- ✘ Slow thread management (creation, deletion, switching, synchronisation…)
  - System calls
- ✓ Blocking blocks only the appropriate thread in a process
- ✓ Thread-level parallelism on multiprocessor

What should be a kernel's execution model?

Note that the same question can be asked of applications

The two alternatives

No one correct answer

From the view of the designer there are two alternatives.

Single Kernel Stack

Only one stack is used all the time to support all user threads.

Per-Thread Kernel Stack

Every user thread has a kernel stack.

Per-Thread Kernel Stack

Processes Model

- A thread's kernel state is implicitly encoded in the kernel activation stack
  - If the thread must block in-kernel, we can simply switch from the current stack, to another threads stack until thread is resumed
  - Resuming is simply switching back to the original stack
  - Preemption is easy
  - no conceptual difference between kernel mode and user mode

```
example(arg1, arg2) {
    P1(arg1, arg2);
    if (need_to_block) {
        thread_block();
        P2(arg2);
    } else {
        P3();
    }
    /* return control to user */
    return SUCCESS;
}
```

Single Kernel Stack

"Event" or "Interrupt" Model

- How do we use a single kernel stack to support many threads?
  - Issue: How are system calls that block handled?

⇒ either *continuations*

- Using Continuations to Implement Thread Management and Communication in Operating Systems. [Draves *et al.*, 1991]

⇒ or *stateless kernel* (interrupt model)

- Interface and Execution Models in the Fluke Kernel. [Ford *et al.*, 1999]

Continuations

- A method to snapshot current state and return to the computation in the future
- In the general case, as many times as we like
- Variations and language environments (e.g. in C) result in less general continuations
  - e.g. one shot continuations

Continuations

- State required to resume a blocked thread is explicitly saved in a TCB
  - A function pointer
  - Variables
- Stack can be discarded and reused to support new thread
- Resuming involves discarding current stack, restoring the continuation, and continuing

```
example(arg1, arg2) {
    P1(arg1, arg2);
    if (need_to_block) {
        save_arg_in_TCB;
        thread_block(example_continue);
        /* NOT REACHED */
    } else {
        P3();
    }
    thread_syscall_return(SUCCESS);
}
example_continue() {
    recover_arg2_from_TCB;
    P2(recovered arg2);
    thread_syscall_return(SUCCESS);
}
```

Stateless Kernel

- System calls can not block within the kernel
  - If syscall must block (resource unavailable)
    - Modify user-state such that syscall is restarted when resources become available
    - Stack content is discarded (functions all return)
- Preemption within kernel difficult to achieve.
  - ⇒ Must (partially) roll syscall back to a restart point
- Avoid page faults within kernel code
  - ⇒ Syscall arguments in registers
    - Page fault during roll-back to restart (due to a page fault) is fatal.

IPC examples – Per thread stack

```
msg_send_rcv(msg, option,
        send_size, rcv_size, ...) {

   rc = msg_send(msg, option,
        send_size, ...);

   if (rc != SUCCESS)
   return rc;

   rc = msg_rcv(msg, option, rcv_size, ...);
   return rc;
}
```

Send and Receive system call implemented by a non-blocking send part and a blocking receive part.

Block inside msg_rcv if no message available

IPC examples - Continuations

```
msg_send_rcv(msg, option, send_size, rcv_size, ...) {
   rc = msg_send(msg, option, send_size, ...);
   if (rc != SUCCESS)
        return rc;
   cur_thread->continuation.msg = msg;
   cur_thread->continuation.option = option;
   cur_thread->continuation.rcv_size = rcv_size;
        ...
   rc = msg_rcv(msg, option, rcv_size, ...,
        msg_rcv_continue);
   return rc;
}
msg_rcv_continue() {
   msg = cur_thread->continuation.msg;
   option = cur_thread->continuation.option;
   rcv_size = cur_thread->continuation.rcv_size;
        ...
   rc = msg_rcv(msg, option, rcv_size, ...,
        msg_rcv_continue);
   return rc;
}
```

The function to continue with if blocked

IPC Examples – stateless kernel

```
msg_send_rcv(cur_thread) {
   rc = msg_send(cur_thread);
   if (rc != SUCCESS)
        return rc;

   rc = msg_rcv(cur_thread);
   if (rc == WOULD_BLOCK) {
       set_pc(cur_thread, msg_rcv_entry);
       return RESCHEDULE;
   }
   return rc;
}
```

Set user-level PC to restart msg_rcv only

RESCHEDULE changes curthread on exiting the kernel

Single Kernel Stack

per Processor, event model

- either *continuations*
  - complex to program
  - must be conservative in state saved (any state that *might* be needed)
  - Mach (Draves), L4Ka::Strawberry, NICTA Pistachio, OKL4, seL4
- or *stateless kernel*
  - no kernel threads, kernel not interruptible, difficult to program
  - request all potentially required resources prior to execution
  - blocking syscalls must always be re-startable
  - Processor-provided stack management can get in the way
  - system calls need to be kept simple “atomic”.
  - e.g. the fluke kernel from Utah
- low cache footprint
  - always the same stack is used !
  - reduced memory footprint

Per-Thread Kernel Stack

- simple, flexible
  - kernel can always use threads, no special techniques required for keeping state while interrupted / blocked
  - no conceptual difference between kernel mode and user mode
  - e.g. traditional L4, Linux, Windows, OS/161
- but larger cache footprint
- and larger memory consumption