School of Computer Science & Engineering

COMP9242 Advanced Operating Systems

2025 T3 Week 02 Part 1

OS Execution Models:
Events, Co-routines, Continuations, Threads

Gernot Heiser

Today's Lecture

- Execution models and how they apply to the OS
  - Events
  - Coroutines
  - Threads
  - Continuations
- Trade-offs and relation to SOS

System Building

General purpose OS needs to deal with concurrency

- Many user activities
  - potentially overlapping
  - may be interdependent
    - need to resume after something else happens
- Activities that depend on external events
  - may requiring waiting for completion (e.g. storage read)
  - reacting to external triggers (e.g. interrupts)

OS defines its execution model

- low-level language
- minimal runtime

Need a systematic approach to execution structure

Execution Models

- Events
- Coroutines
- Threads
- Continuations

Note: Focus is on uni-processor for now, multiprocessors later

Events

Events

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

Some Definitions

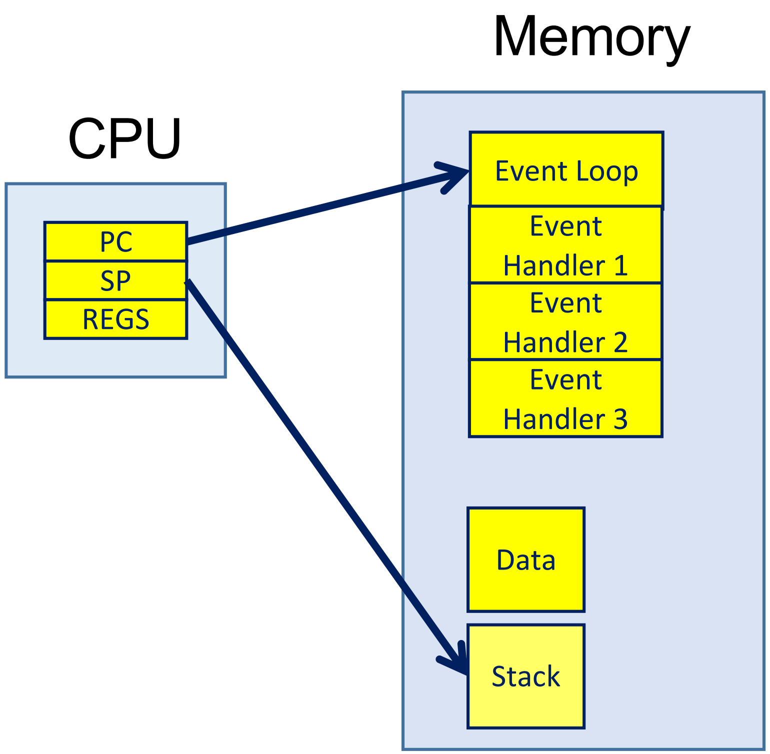
Block:

- Execution state is preserved
- *Marks current execution as blocked*
- *It is no longer considered Ready*
  - *Removed from a Ready Queue*
- *Requires an unblock to mark ready and rejoin the ready queue*
- Resumes from where it blocked

Yield:

- Execution state is preserved
- *The thread relinquishes execution*
- *Immediately placed in the ready queue*
- Resumes from where it yielded

Event Model



Only requires a single stack:

- Event handlers return to the event loop
  - No blocking
  - No yielding
- No preemption of handlers
  - Handler functions should be short!

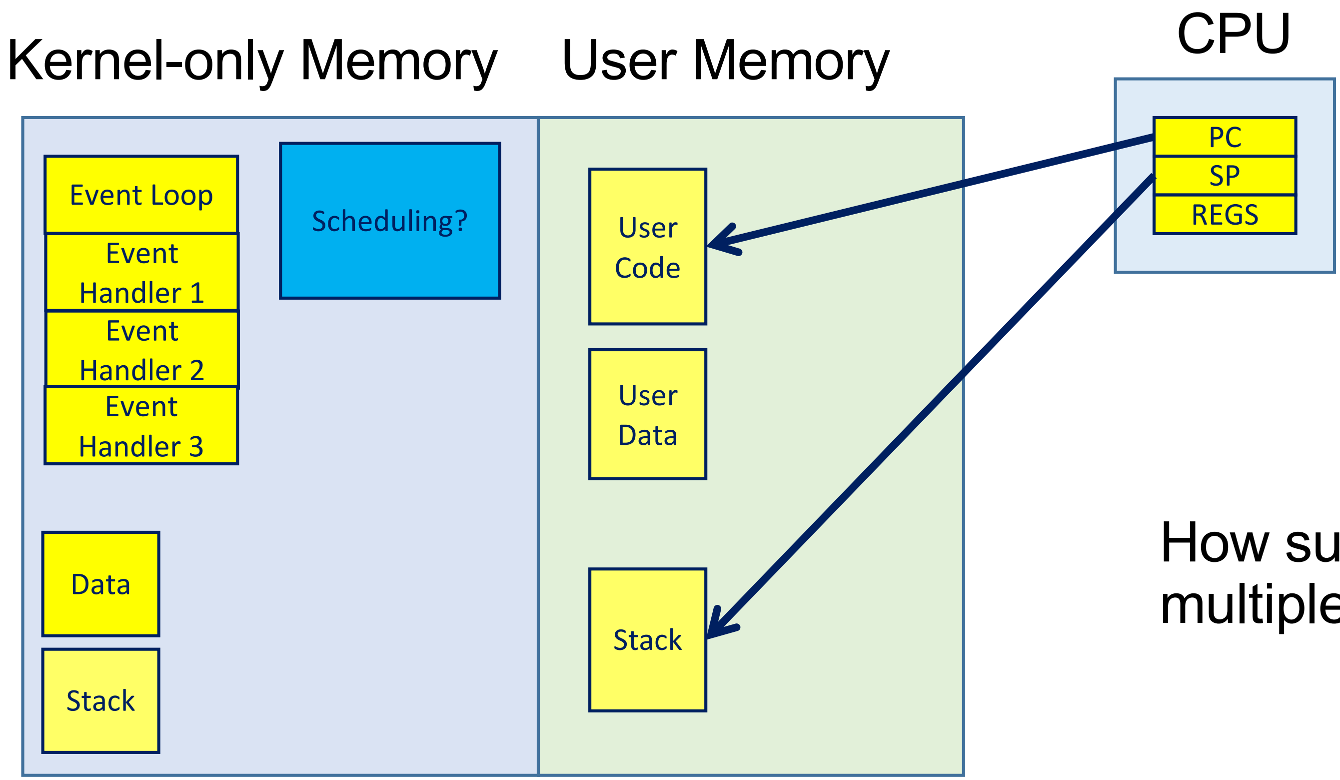
What is ‘a’?

```
int a; /* global */

int func() {
    a = 1;
    if (a == 1) {
        a = 2;
    }
    return a;
}
```

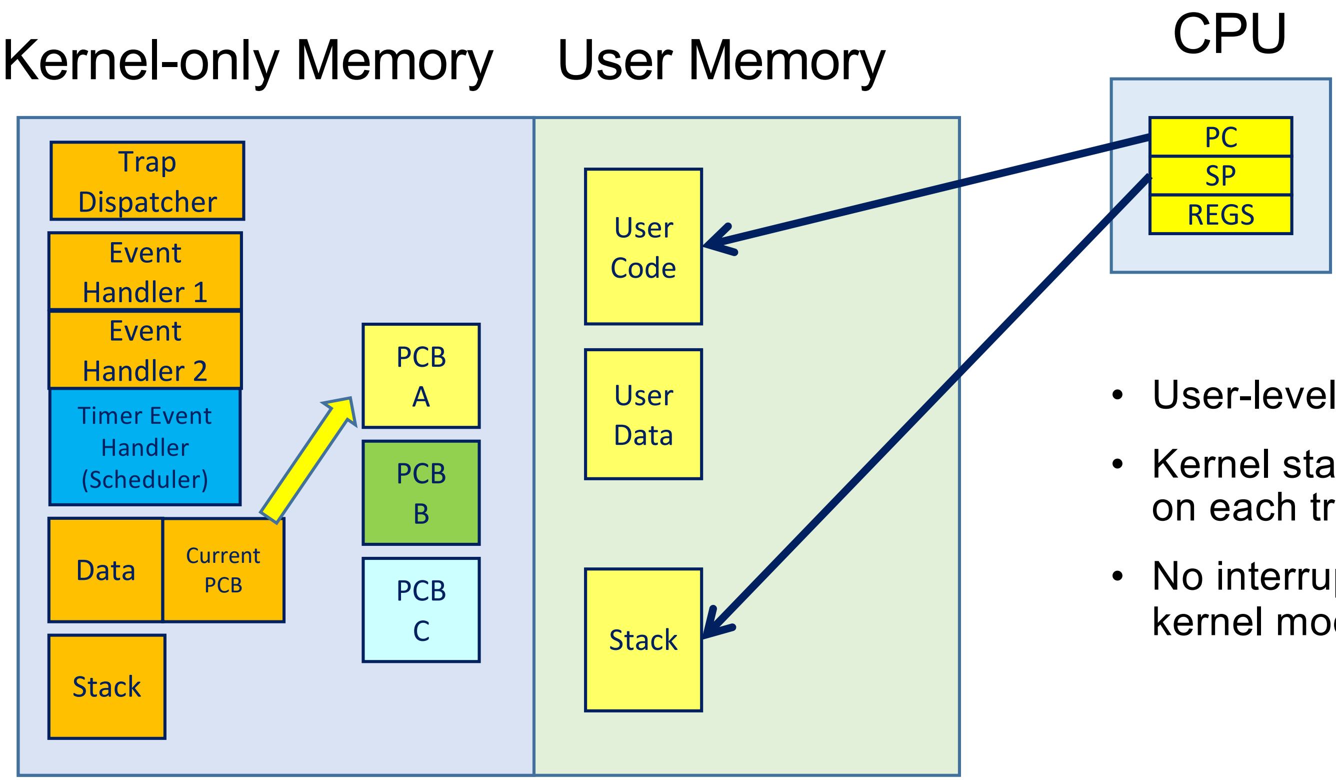
No concurrency issues within a handler

Event-based kernel on CPU with protection



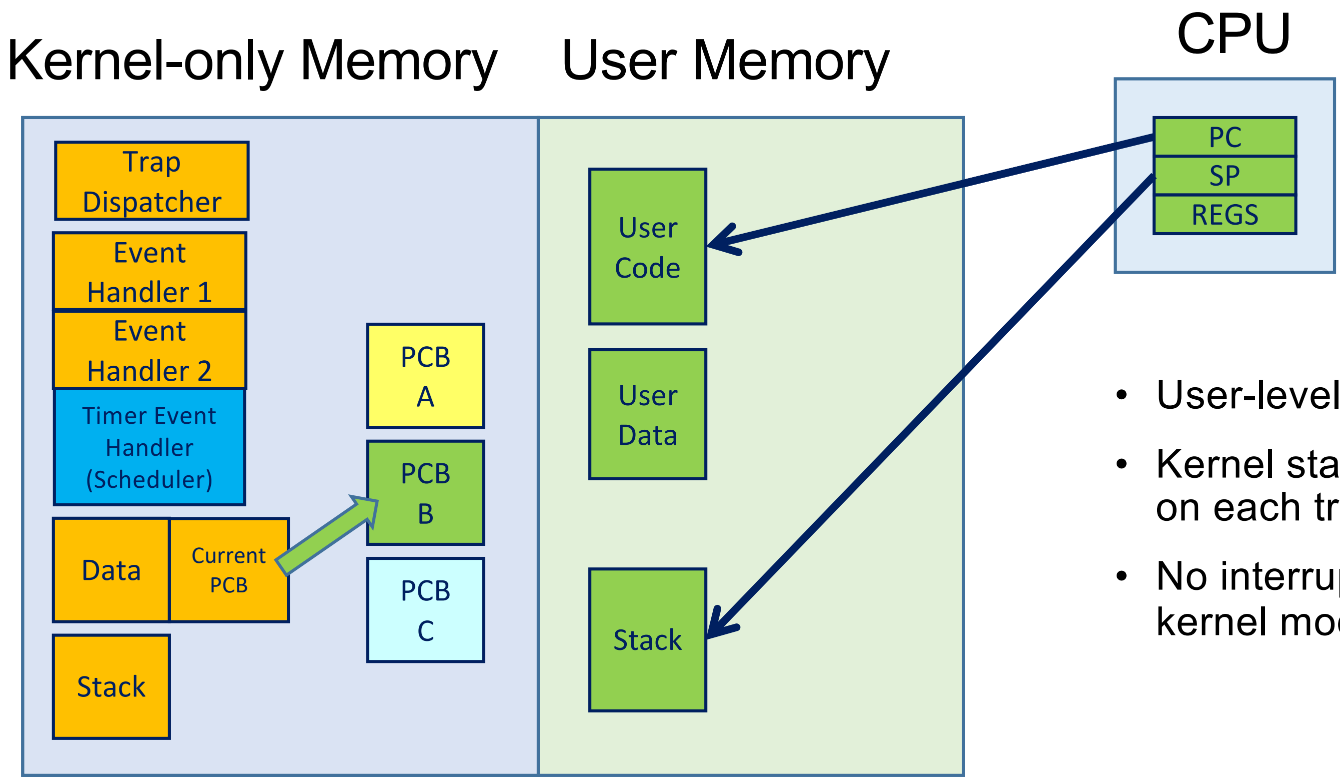
How support multiple processes?

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

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

Coroutines

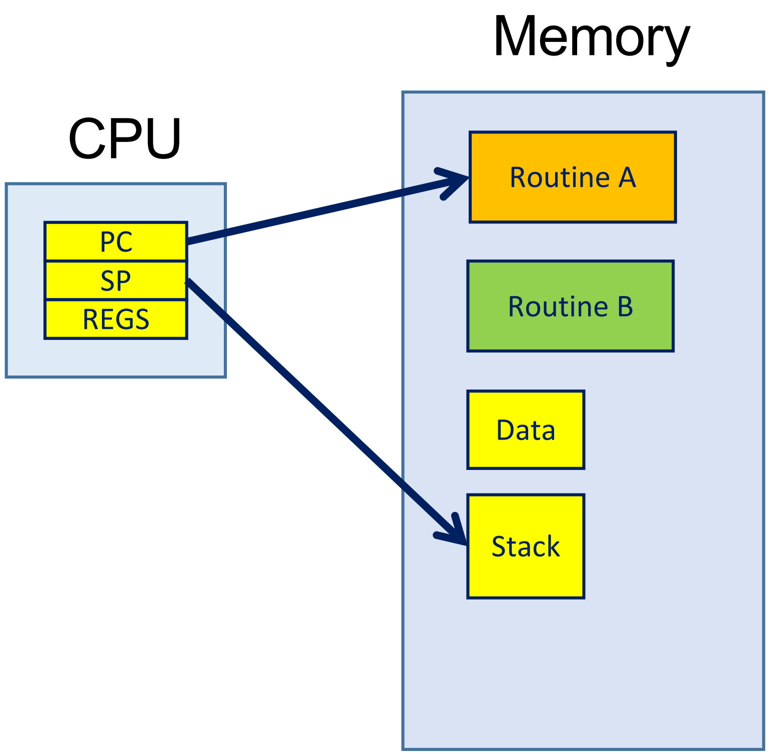
Coroutines

- Old idea:

  Melvin E. Conway. 1963. *Design of a separable transition-diagram compiler*. Commun. ACM 6, 7 (July 1963), 396-408. DOI=http://dx.doi.org/10.1145/366663.366704

- Analogous to a “subroutine” with extra entry and exit points
  - Exit/enter via yield()
  - Supports long running subroutines
  - Can implement sync primitives that wait for a condition to be true
    - `while (condition != true) yield();`

Coroutines



- yield() saves state of routine A and starts routine B
  - or resumes B's state from its previous yield() point.
- No pre-emption, any switching is explicit via yield() in code

What is ‘a’?

```
int a; /* global */

int func() {
    a = 1;
    if (a == 1) {
       yield();
       a = 2;
    }
    return a;
}
```

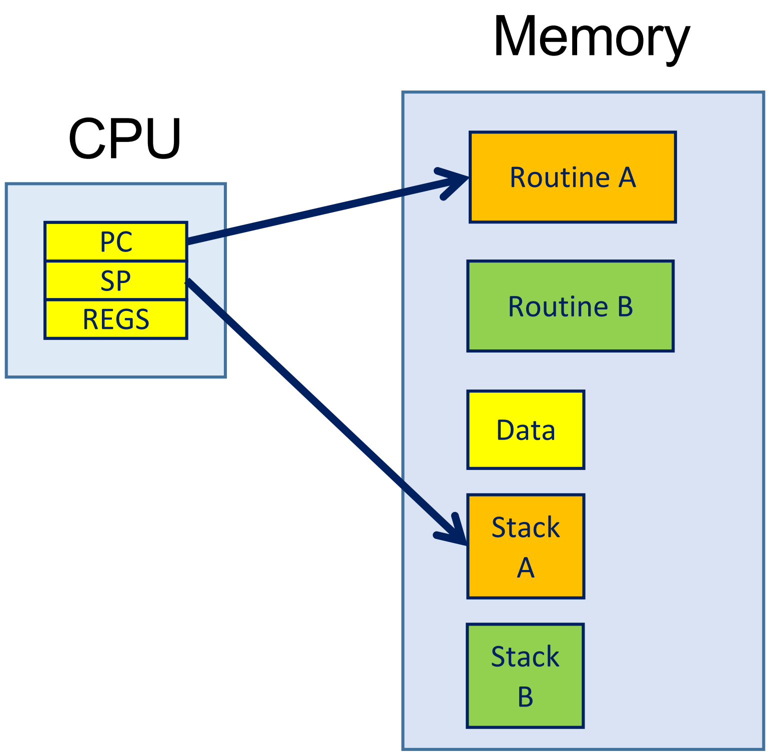
What is ‘a’?

```
int a; /* global */

int func() {
    a = 1;
    yield();
    if (a == 1) {
        a = 2;
    }
    return a;
}
```

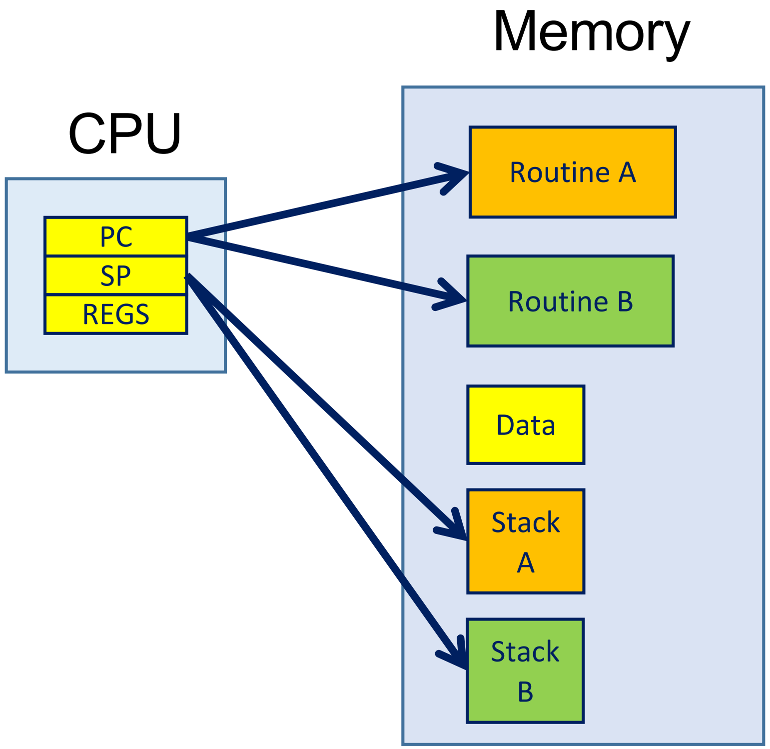
Limited concurrency issues/races as globals are exclusive between yields()

Coroutines Implementation strategy?



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

Coroutines Implementation strategy?



- Routine A state currently loaded
- Routine B state stored on stack
- Routine switch from A → B
  - saving state of A 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
```

A hypothetical yield()

```
/* 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 */
   sw sp, 0(a0)
```

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

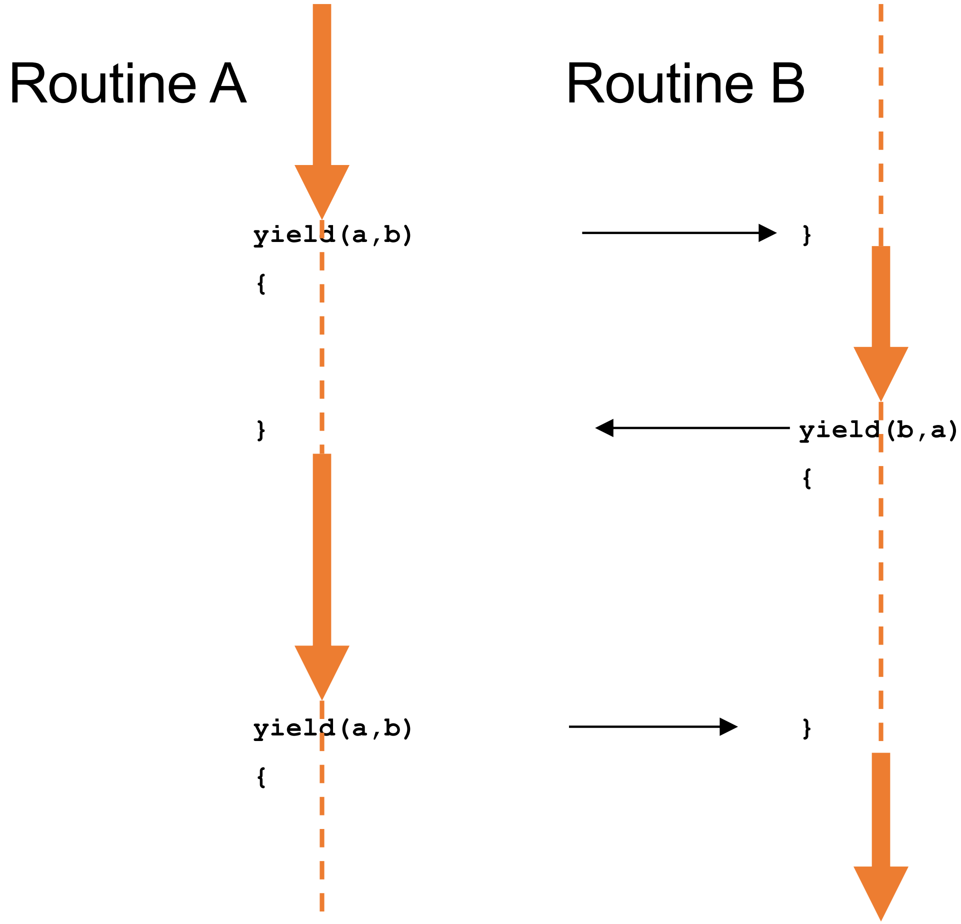
A hypothetical yield()

```
/* 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 gp, 36(sp)
   lw ra, 40(sp)
   nop                      /* delay slot for load */

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

Yield



What is 'a'?

```
int a; /* global */

int func() {
    a = 1;
    func2();
    if (a == 1) {
        a = 2;
    }
    return a;
}
```

Does `func2()` `yield()`?

Coroutines

What about subroutines combined with coroutines

- i.e. what is the issue with calling subroutines?

Subroutine calling might involve an implicit yield()

May creates a race on globals

- either understand where all yields lie, or
- use cooperative multithreading!

Use at your own risk!

- Build has **`libco`** (used by gdb thread):
  - https://github.com/higan-emu/libco
- Tony Finch’s **`picoro`**: https://dotat.at/git/picoro.git/

Threads

Cooperative Multithreading

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

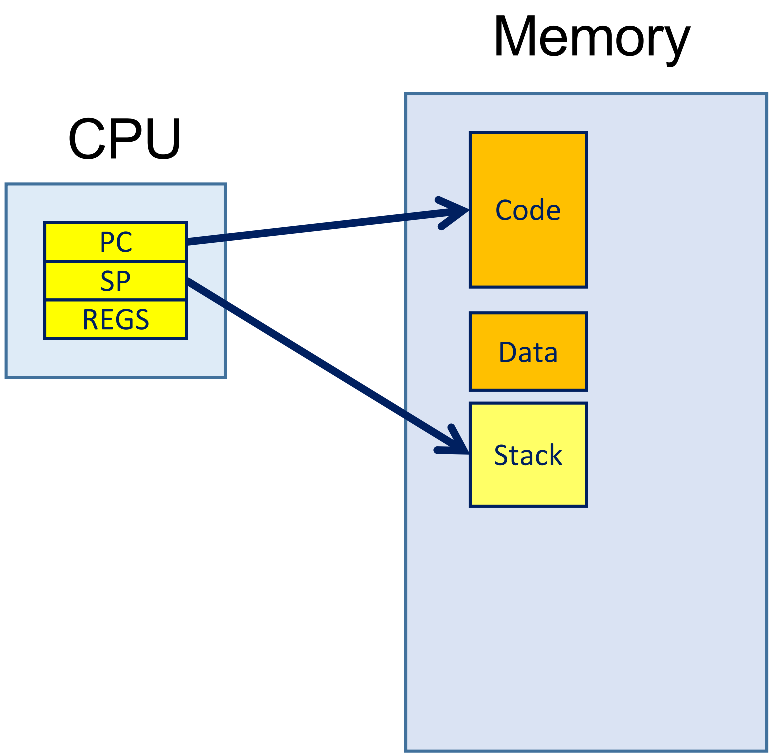
Green Threads

```
int a; /* global */
lock_t a_lock;
int func() {
    int t;
    lock_acquire(a_lock)
    a = 1;
    func2();
    if (a == 1) {
        a = 2;
    }
    t = a;
    lock_release(a_lock);
    return t;
}
```

Pessimistic locking

Deadlocks?

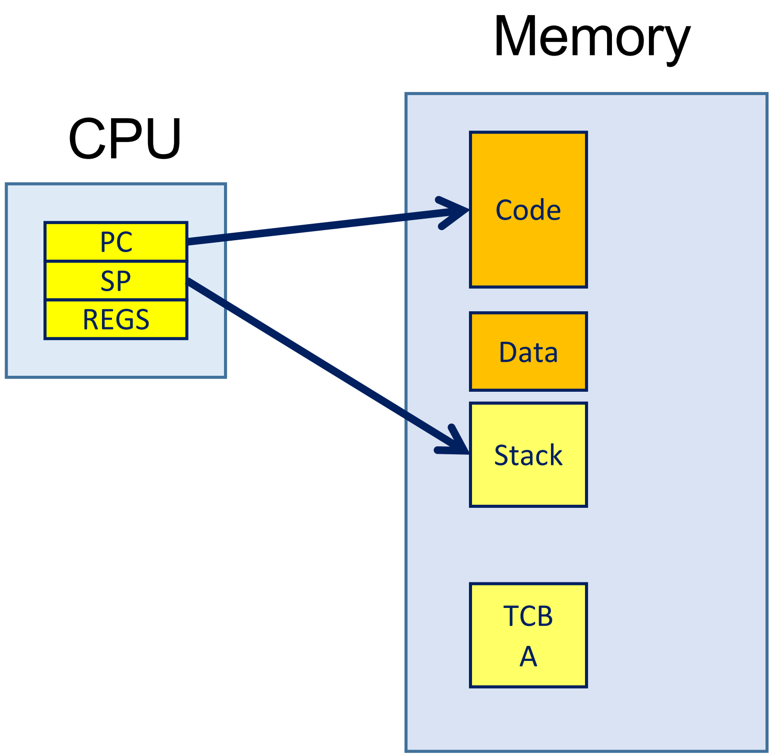
A Thread



Thread attributes

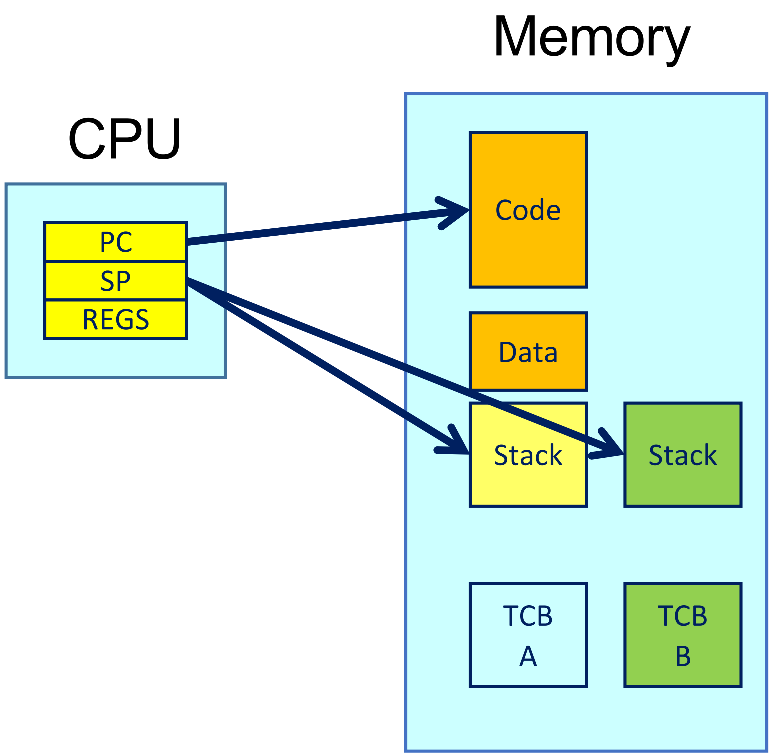
- processor related:
  - memory
  - program counter
  - stack pointer
  - registers (and status)
- OS/package related:
  - state (running/blocked)
  - identity
  - scheduler (queues, priority)
  - etc…

Thread Control Block (TCB)



- To support more than a single thread we to need store thread state and attributes
- Stored in per-thread 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

OS Pseudo-Code

```
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_sp, &next->t_sp);
 /* 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 tcb.
    * a1 contains a pointer to the new thread's struct tcb.
    *
    * The only thing we touch in the tcb is the first word, which
    * we save the stack pointer in. The other 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
```

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 tcb */
   sw sp, 0(a0)
```

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

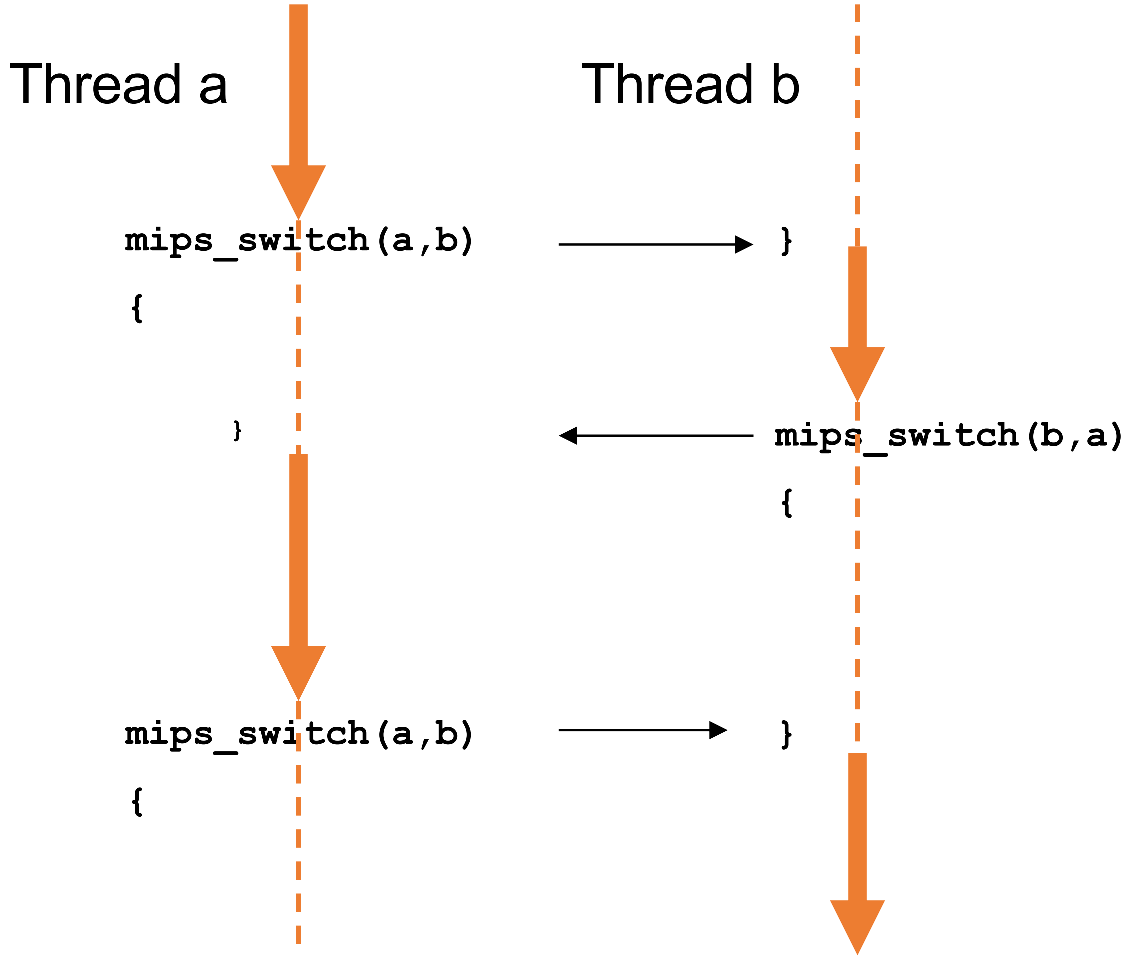
OS/161 mips_switch

```
/* Get the new stack pointer from the new tcb */
   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 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
```

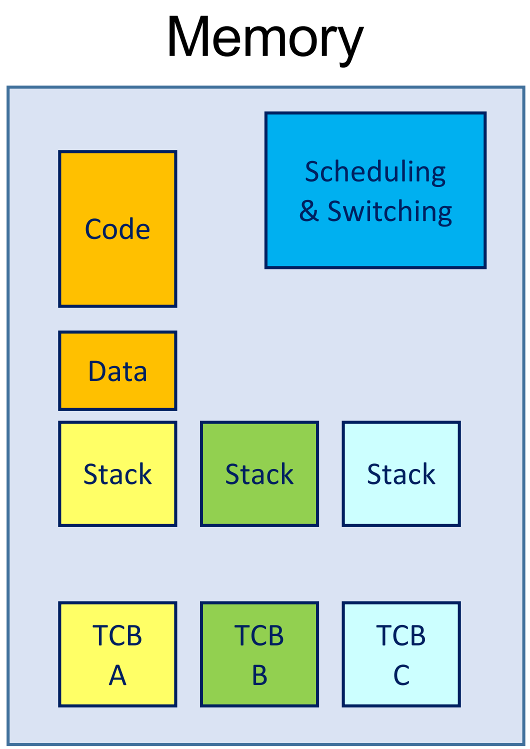
Thread Switch



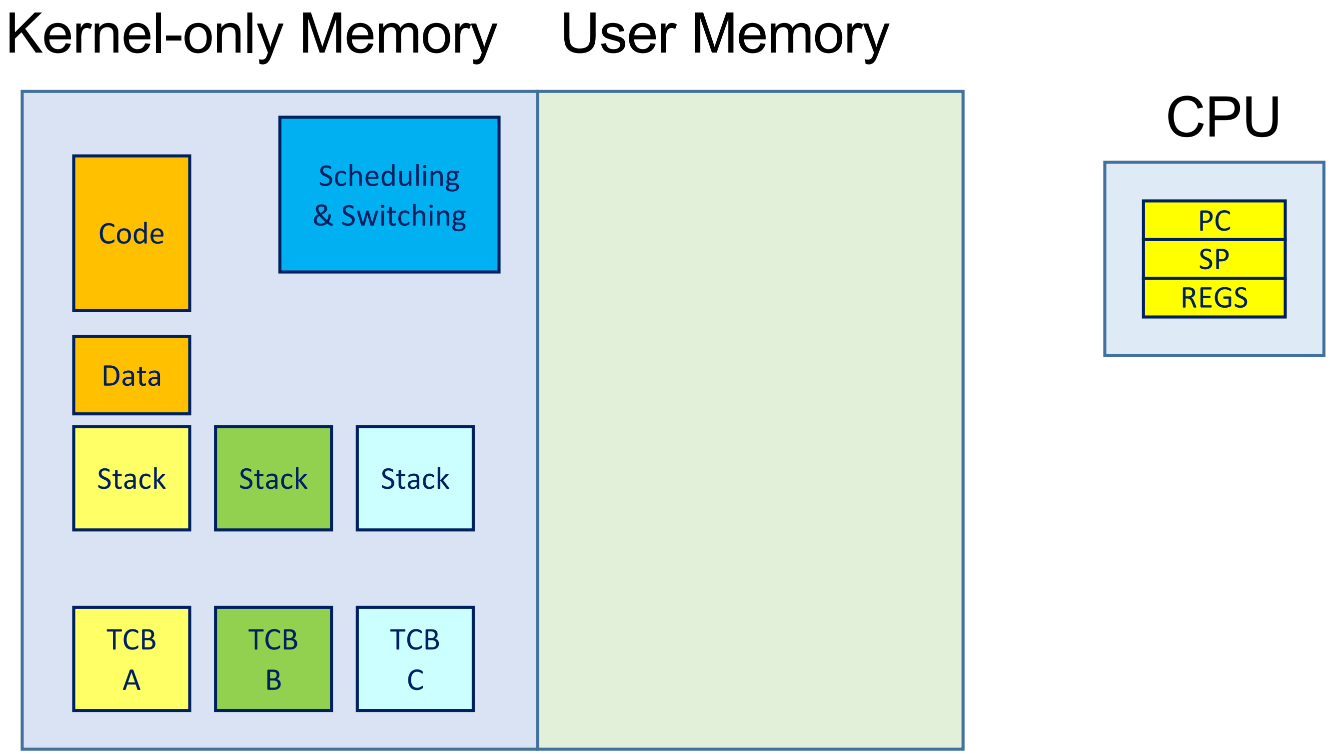
Preemptive Multithreading

- Switch can be triggered by asynchronous external event
  - eg. timer interrupt
- Asynchronous interrupt triggers saving 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

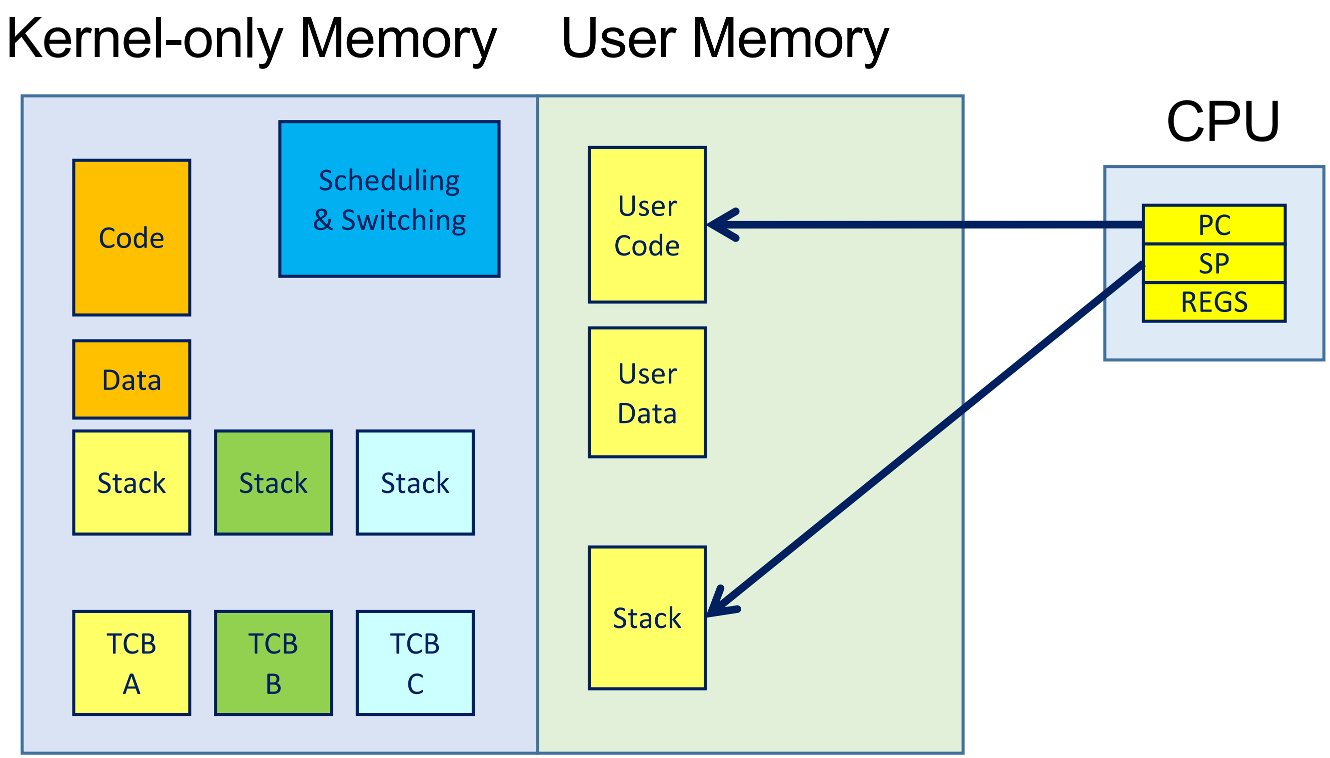


Threads on CPU with protection



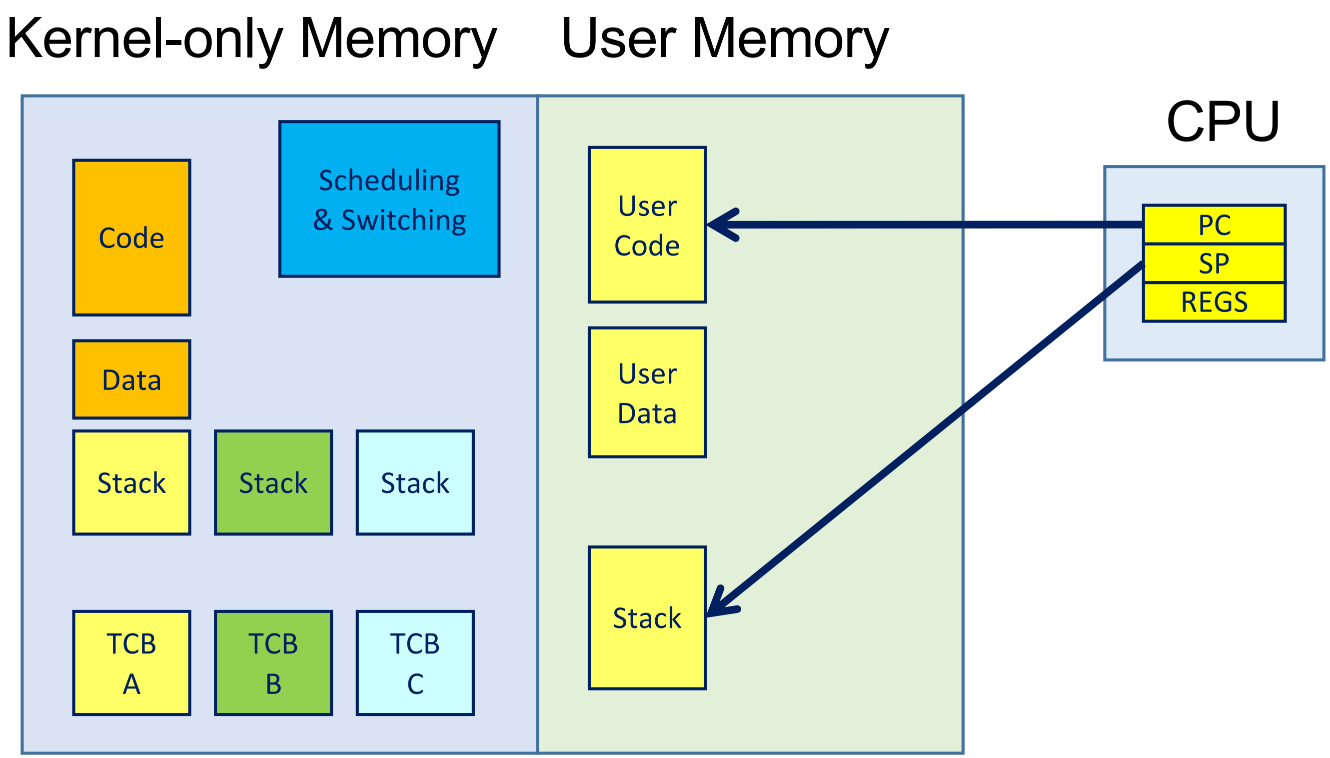
What is missing?

Threads on CPU with protection

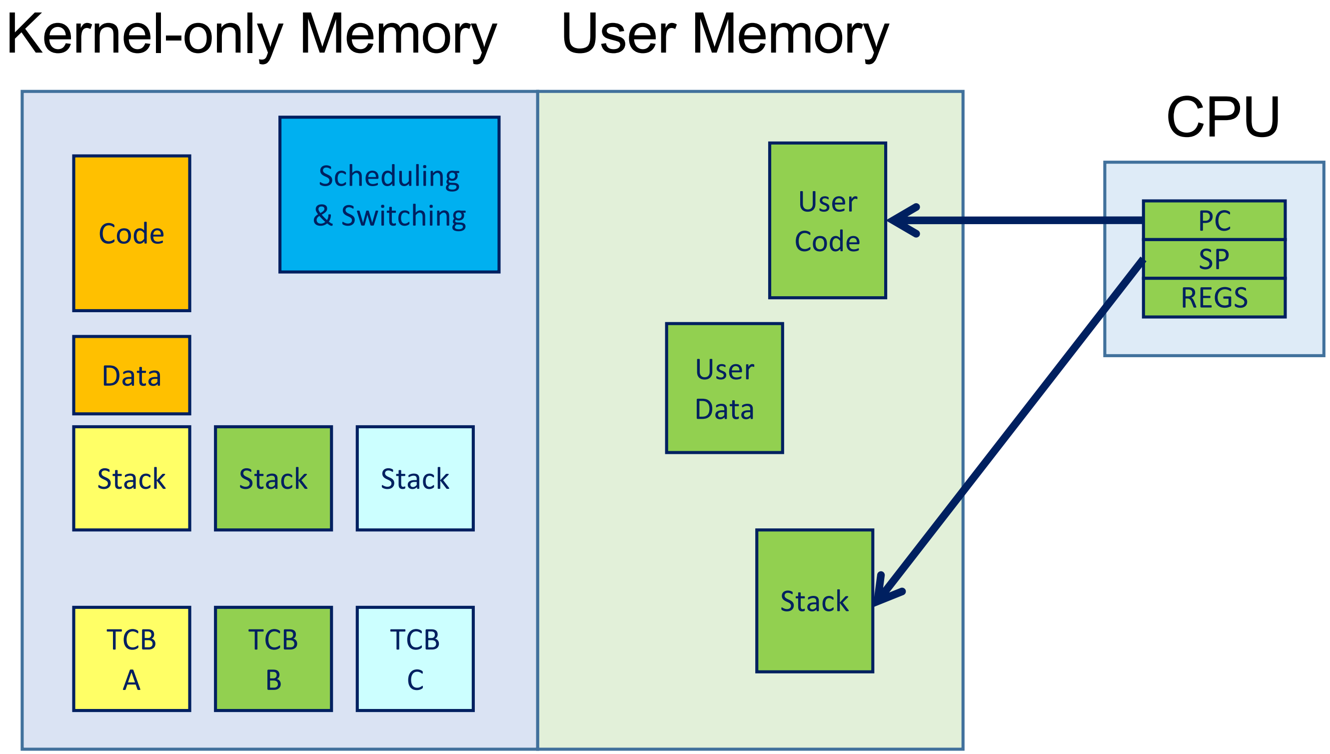


- What happens on kernel entry and exit?

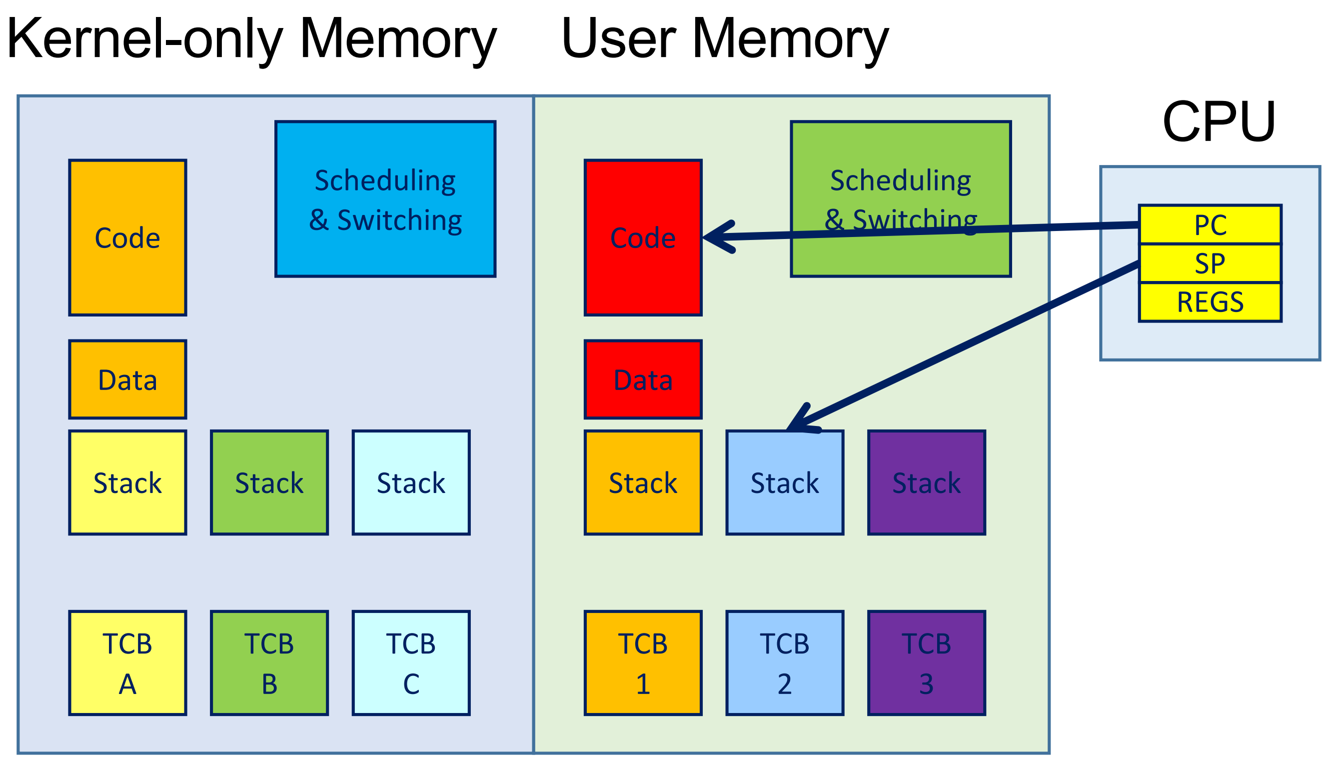
Thread Switch Switching Address Space: Process



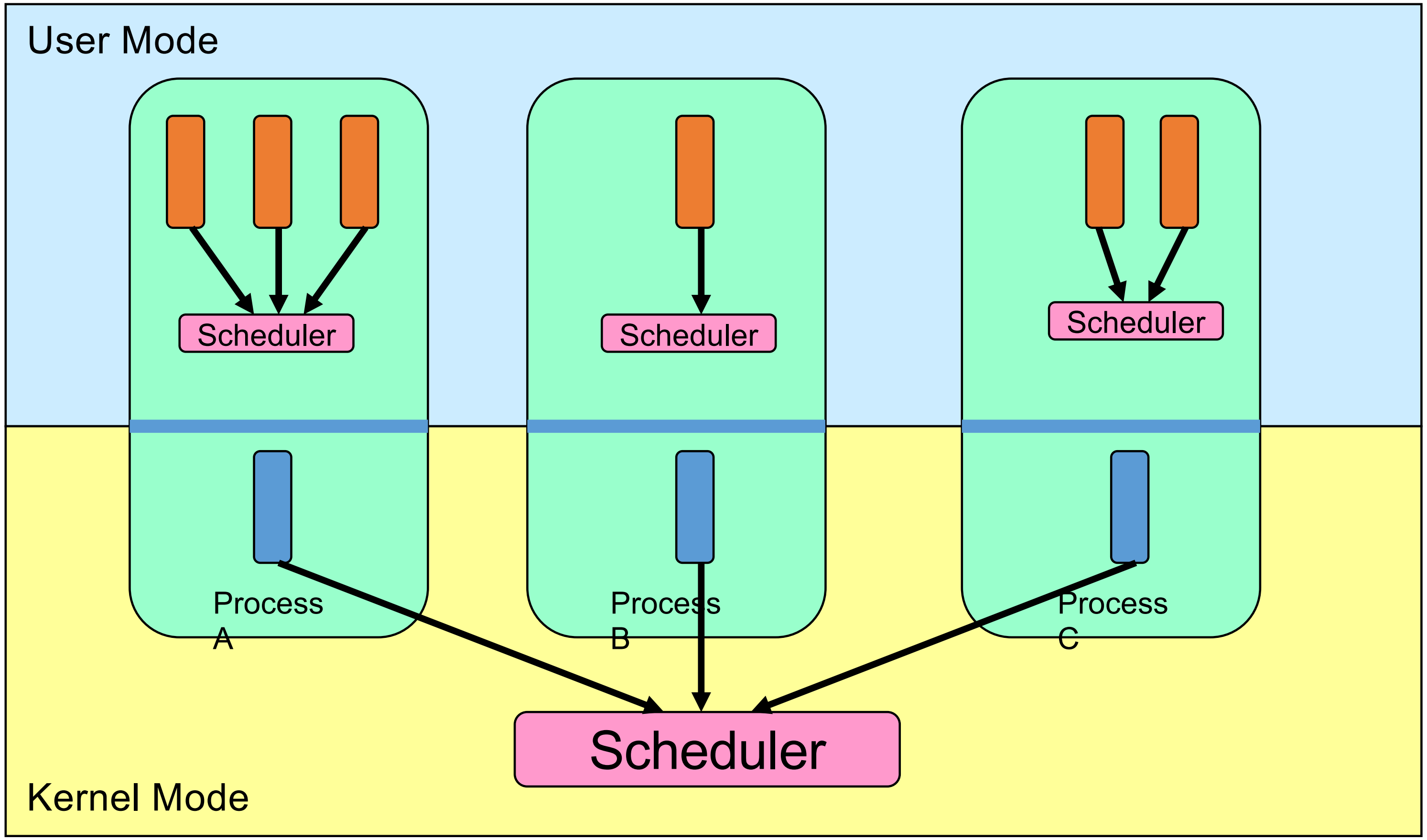
Thread Switch Switching Address Space: Process



What is this?


Kernel-only Memory
User Memory
CPU
Code
Scheduling & Switching
Data
Stack
Stack
Stack
TCB A
TCB B
TCB C
Code
Scheduling & Switching
Data
Stack
Stack
Stack
TCB 1
TCB 2
TCB 3
PC
SP
REGS

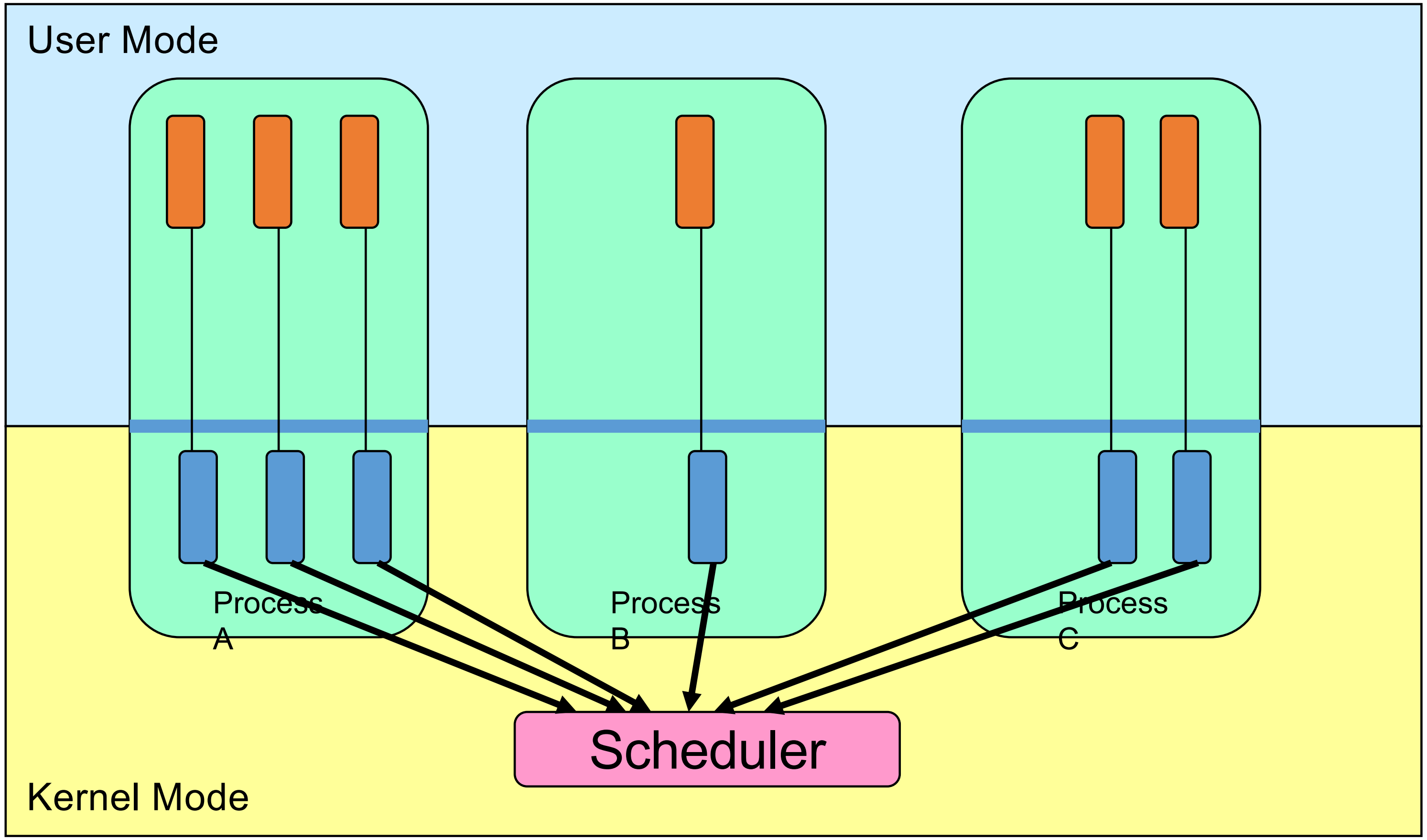
User-level Threads



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

Continuations

Continuations

Continuation:

- representation of an instance of a computation at a point in time
- the state and code where to *continue* from

Continuations in PLs: Python

- Traditional function that returns:

```
def func(x):
    return x+1
```

- Function with a continuation indicating where to continue

```
def func_cps(x,c):
    c(x+1)
```

Continuation invoked with function's result

Continuations

The concept of capturing current (stack) state to continue the computation in the future

- In the general case can restore same state repeatedly
- C has one-shot continuations: **`setjmp()/longjump()`**

OS Execution Models

OS Execution Model Alternatives

Single Kernel Stack

- One stack supports all user threads
- “Event model” / “interrupt model”

Per-Thread Kernel Stack

- Every user threads has a separate kernel stack (besides its user-level stack)
- “Process model”

Per-Thread Kernel Stack

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

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

- Dump registers on stack
- Switch stack
- Restore registers

Single Kernel Stack

How do we use a single kernel stack to support many threads?

- Issue: How are system calls that block handled?

⇒ Use *continuations*

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

⇒ Use *stateless kernel* (event model)

  - Used in Fluke: *Interface and Execution Models in the Fluke Kernel.* [Ford *et al.,* 1999]
  - Also used seL4

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);
}
```

Logically, **`p2 (arg2)`** exceuted here

Stateless Kernel

System calls cannot 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.

Example Implementations – IPC

IPC implementation – 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 implementation – Continuations

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

```
msg_rcv_continue() {
  msg = cur_thread->contin.msg;
  option = cur_thread->
        contin.option;
  rcv_size = cur_thread->
        contin.rcv_size;
        ...
  rc = msg_rcv(msg, option,
        rcv_size,
        ..., msg_rcv_continue);
  return rc;
}
```

Save state

The function to continue with if blocked

IPC Implementation – 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

Summary

Single Kernel Stack

- Either *continuations*
  - complex to program
  - must be conservative in state saved (any state that *might* be needed)
  - Mach (Draves), L4Ka::Strawberry, NICTA Pistachio, OKL4
- 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, seL4
- 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
- … and more concurrency issues