Processes and Threads Implementation

Learning Outcomes

- A basic understanding of the MIPS R3000 assembly and compiler generated code.
- 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"

UNSW SYDNEY

MIPS R3000

- Load/store architecture
  - No instructions that operate on memory except load and store
  - Simple load/stores to/from memory from/to registers
    - Store word: `sw r4, (r5)` sw from r4 to r5
      - Store contents of r4 in memory using address contained in register r5
    - Load word: `lw r3,(r7)` load from r7 to r3
      - Load contents of memory into r3 using address contained in r7
      - Delay of one instruction after load before data available in destination register
        - Must always an instruction between a load from memory and the subsequent use of the register.
  - `lw, sw, lb, sb, lh, sh`,....

32 32 8 8

MIPS R3000

- Arithmetic and logical operations are register to register operations
    - E.g., add r3, r2, r1
    - No arithmetic operations on memory
- Example
    - `add r3, r2, r1` ⇒ r3 = r2 + r1
- Some other instructions
    - `add, sub, and, or, xor, sll, srl`
    - `move r2, r1` ⇒ r2 = r1

D S

MIPS R3000

- All instructions are encoded in 32-bit
- Some instructions have *immediate* operands
  - Immediate values are constants encoded in the instruction itself
  - Only 16-bit value
  - Examples
    - Add Immediate: `addi r2, r1, 2048`
      ⇒ r2 = r1 + 2048
    - Load Immediate : `li r2, 1234`
      ⇒ r2 = 1234

D S

D S

UNSW SYDNEY

Example code

Simple code example: **`a = a + 1`**

load value of a into r4

local variable is located on the stack

```
lw  r4,32(r29)  // r29 = stack pointer
li  r5, 1
add r4, r4, r5
sw  r4,32(r29)
```

store r4 to stack pointer

Offset(Address)

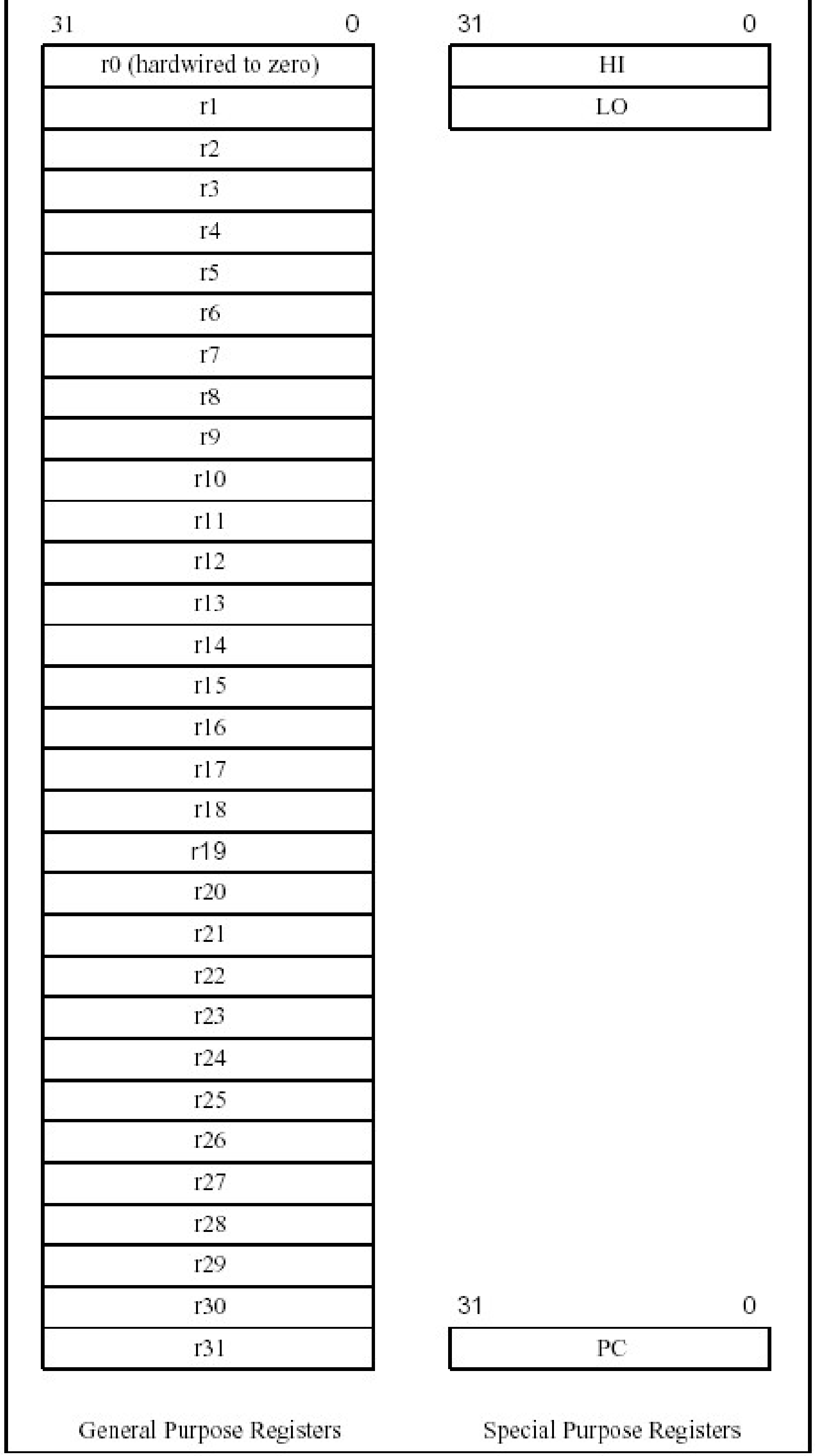
MIPS Registers

- User-mode accessible registers
  - 32 general purpose registers
    - r0 hardwired to zero
    - r31 the *link* register for jump-and-link (JAL) instruction

where we can all the jal from is stored in r31

  - HI/LO
    - 2 * 32-bits for multiply and divide
  - PC
    - Not directly visible
    - Modified implicitly by jump and branch instructions

Figure 2-6 CPU Registers



Branching and Jumping

- Branching and jumping have a *branch delay slot*
  - The instruction following a branch or jump is always executed prior to destination of jump

```
    li   r2, 1
    sw r0,(r3)
    j    1f
    li   r2, 2
    li   r2, 3
1:  sw r2, (r3)
```

when this line is finishing, the next line is starting

we execute li r2,2 because it is in the pipeline of jump instruction, so we essentially set r2 to 2 and store word from r2 to r3

r3 = 2

MIPS R3000

- RISC architecture – 5 stage pipeline
  - Instruction partially through pipeline prior to jmp having an effect

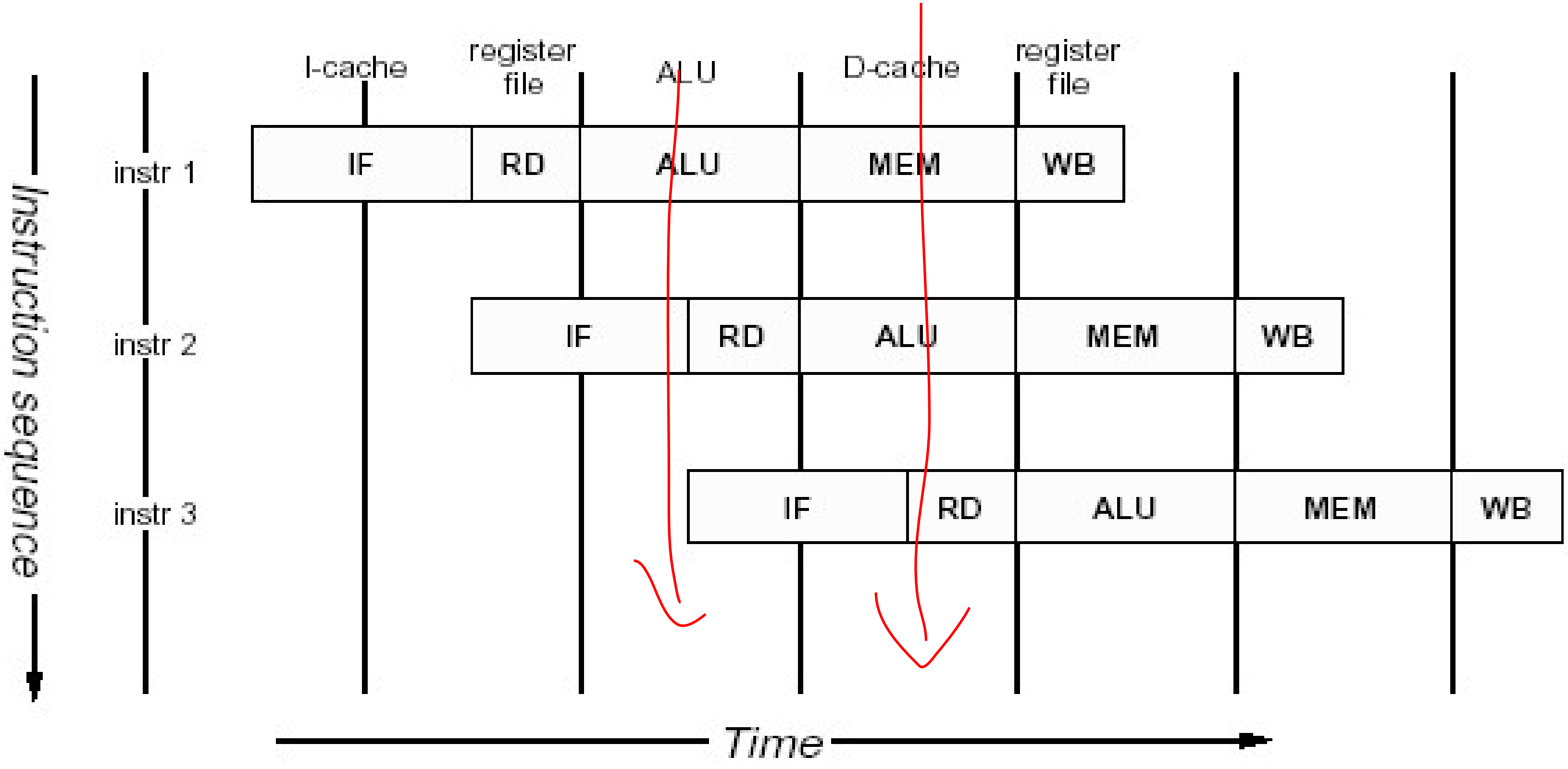


Figure 1.1. MIPS 5-stage pipeline

Jump and Link Instruction

- JAL is used to implement function calls
  - r31 = PC+8
- Return Address register (RA) is used to return from function call

nop is there to avoid confusion

```
⋮
0x10        jal   1f
0x14        nop
0x18        lw r4,(r6)
⋮
1:
0x2a        sw r2, (r3)
⋮
0x38        jr r31
0x3a        nop
```

Compiler Register Conventions

- Given 32 registers, which registers are used for
  - Local variables?
  - Argument passing?
  - Function call results?
  - Stack Pointer?

Compiler Register Conventions

this is defined by the compiler

if we return an integer,
v0 is for 32 bits
v0-v1 is for 64 bits

arguments 1 - 4,
if more, put on stack

value preserved
in the function calls

Reg No	Name	Used for
0	zero	Always returns 0
1	at	(assembler temporary) Reserved for use by assembler
2-3	v0-v1	Value (except FP) returned by subroutine
4-7	a0-a3	(arguments) First four parameters for a subroutine
8-15	t0-t7	(temporaries) subroutines may use without saving
24-25	t8-t9	
16-23	s0-s7	Subroutine "register variables"; a subroutine which will write one of these must save the old value and restore it before it exits, so the *calling* routine sees their values preserved.
26-27	k0-k1	Reserved for use by interrupt/trap handler - may change under your feet
28	gp	global pointer - some runtime systems maintain this to give easy access to (some) "static" or "extern" variables.
29	sp	stack pointer
30	s8/fp	9th register variable. Subroutines which need one can use this as a "frame pointer".
31	ra	Return address for subroutine

Simple factorial

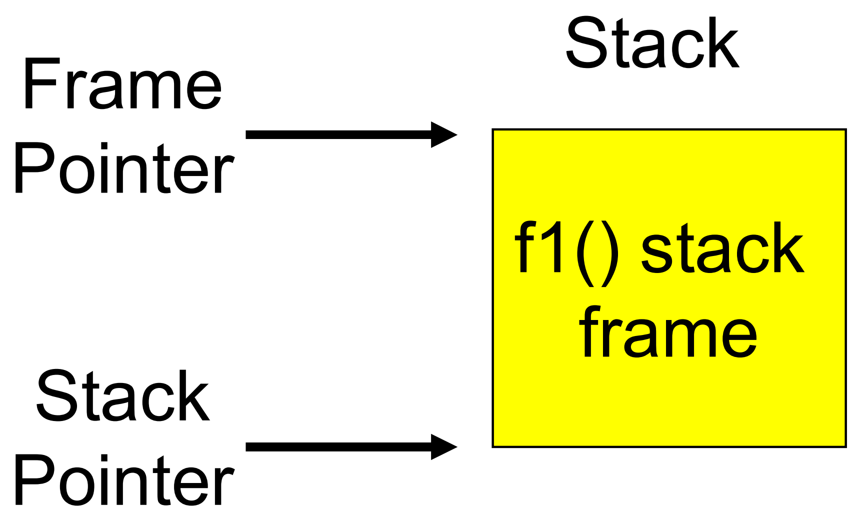
```c
int fact(int n)
{
  int r = 1;
  int i;

  for (i = 1; i < n+1; i++) {
    r = r * i;
  }
  return r;
}
```

n=3 branch

```
 0:   1880000b    blez    a0,30 <fact+0x30>
 4:   24840001    addiu   a0,a0,1       n+1=4
 8:   24030001    li      v1,1          i=1
 c:   24020001    li      v0,1          r=1
10:   00430018    mult    v0,v1         i×r=1
14:   24630001    addiu i v1,v1,1
18:   00001012    mflo    v0            r   move the return to v0
1c:   00000000    nop
20:   1464fffc    bne     v1,a0,14 <fact+0x14>
24:   00430018    mult    v0,v1         we do multiply in branch delay slot
28:   03e00008    jr      ra
2c:   00000000    nop
30:   03e00008    jr      ra
34:   24020001    li      v0,1          return 1
```

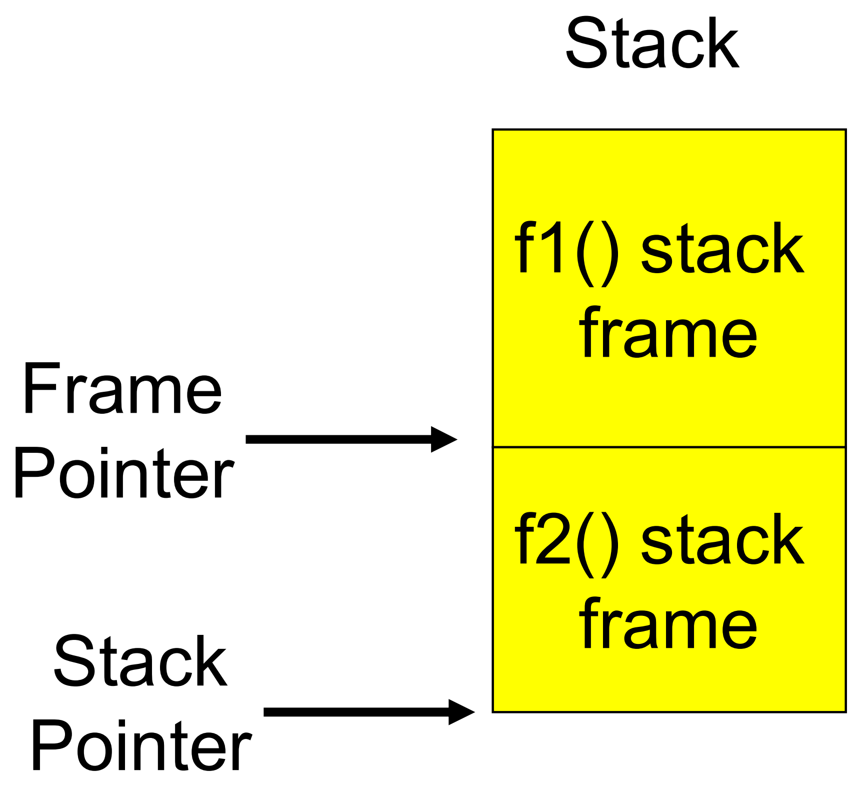
Function Stack Frames

- Each function call allocates a new stack frame for local variables, the return address, previous frame pointer etc.
  - Frame pointer: start of current stack frame
  - Stack pointer: end of current stack frame
- Example: assume f1() calls f2(), which calls f3().



Function Stack Frames

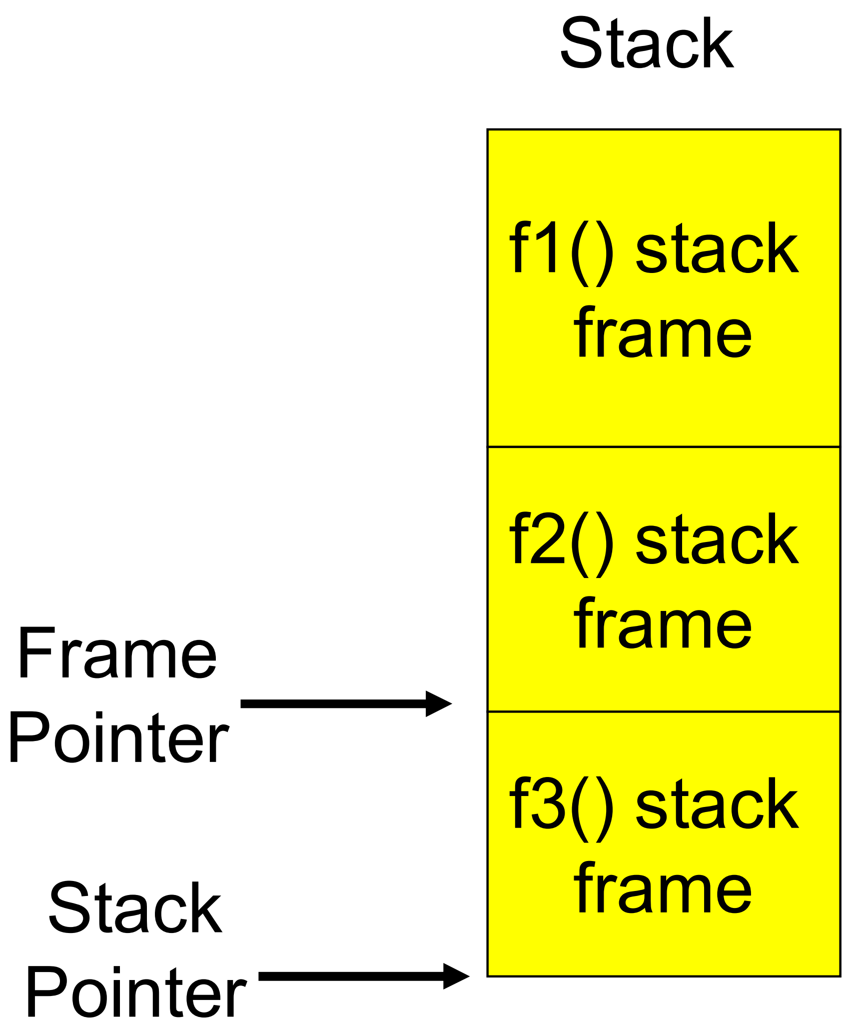
- Each function call allocates a new stack frame for local variables, the return address, previous frame pointer etc.
  - Frame pointer: start of current stack frame
  - Stack pointer: end of current stack frame
- Example: assume f1() calls f2(), which calls f3().



Function Stack Frames

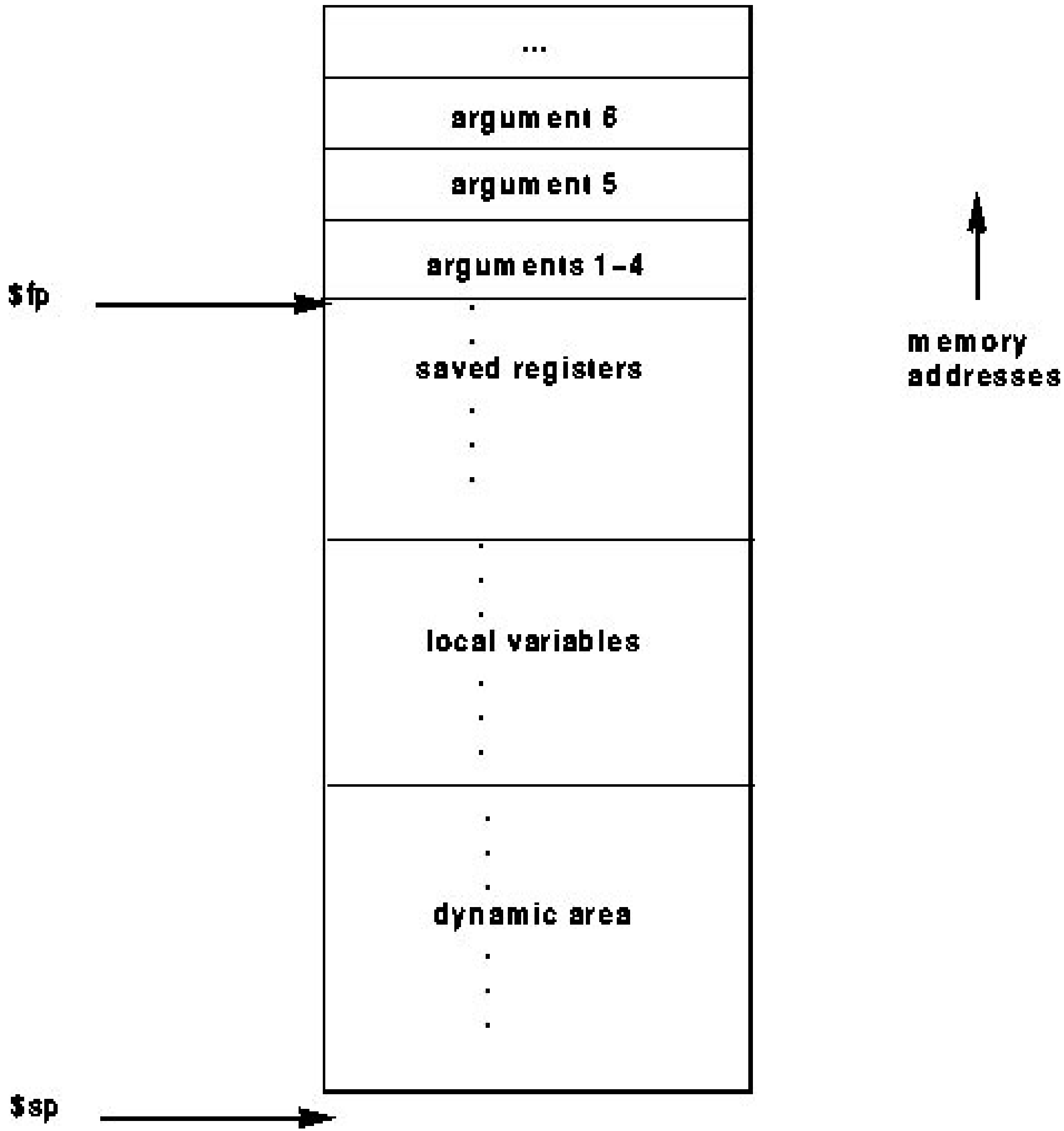
- Each function call allocates a new stack frame for local variables, the return address, previous frame pointer etc.
  - Frame pointer: start of current stack frame
  - Stack pointer: end of current stack frame
- Example: assume f1() calls f2(), which calls f3().

be careful with local variables, don't use recursion in operating system code



Stack Frame

- MIPS calling convention for gcc
  - Args 1-4 have space reserved for them



Example Code

```
main ()
{
  int i;

  i =
 sixargs(1,2,3,4,5,6);
}
```

```
int sixargs(int a, int
 b, int c, int d, int e,
 int f)
{
  return a + b + c + d
      + e + f;
}
```

```
0040011c <main>:
  40011c:     27bdffd8     addiu  sp,sp,-40
  400120:     afbf0024     sw     ra,36(sp)
  400124:     afbe0020     sw     s8,32(sp)
  400128:     03a0f021     move   s8,sp
  40012c:     24020005     li     v0,5
  400130:     afa20010     sw     v0,16(sp)
  400134:     24020006     li     v0,6
  400138:     afa20014     sw     v0,20(sp)
  40013c:     24040001     li     a0,1
  400140:     24050002     li     a1,2
  400144:     24060003     li     a2,3
  400148:     0c10002c     jal    4000b0 <sixargs>
  40014c:     24070004     li     a3,4
  400150:     afc20018     sw     v0,24(s8)
  400154:     03c0e821     move   sp,s8
  400158:     8fbf0024     lw     ra,36(sp)
  40015c:     8fbe0020     lw     s8,32(sp)
  400160:     03e00008     jr     ra
  400164:     27bd0028     addiu  sp,sp,40
  ...
```

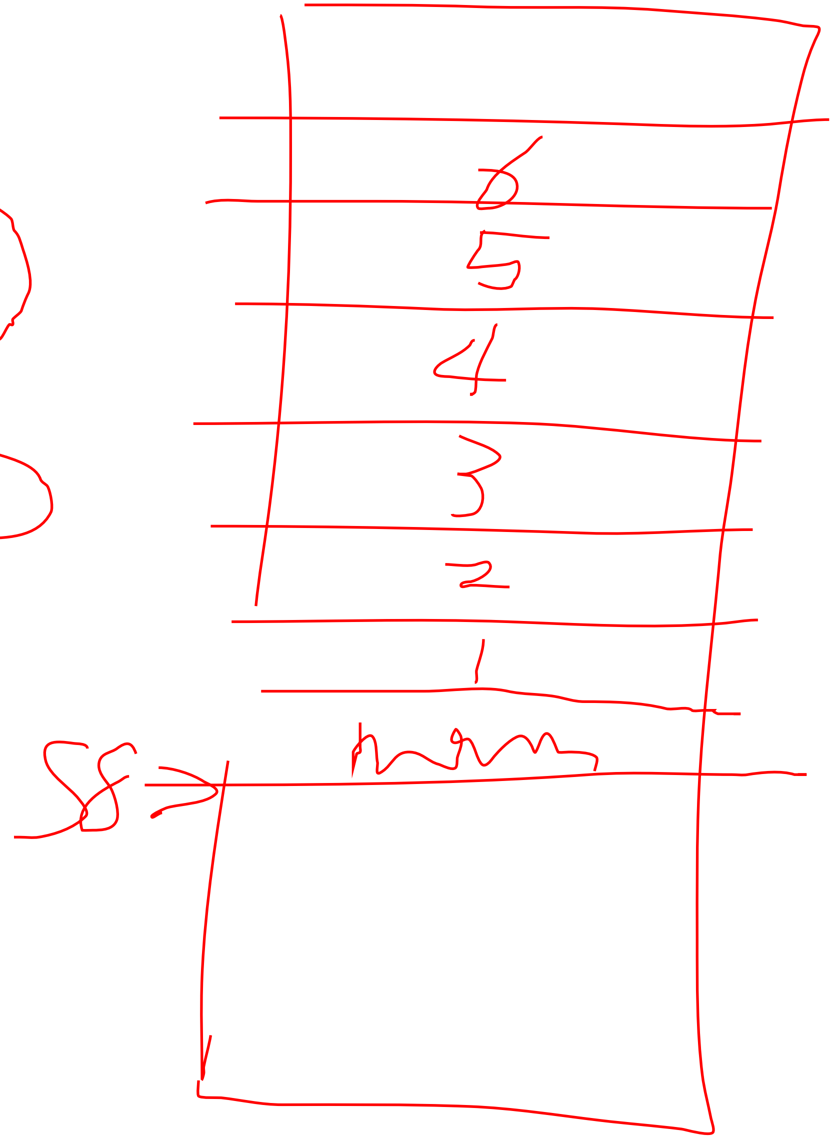
argument 4 and 5 on the stack

jump to function sixargs

jump delay slot

return

```
004000b0 <sixargs>:
  4000b0:   27bdfff8    addiu sp,sp,-8
  4000b4:   afbe0000    sw    s8,0(sp)
  4000b8:   03a0f021    move  s8,sp
  4000bc:   afc40008    sw    a0,8(s8)
  4000c0:   afc5000c    sw    a1,12(s8)
  4000c4:   afc60010    sw    a2,16(s8)
  4000c8:   afc70014    sw    a3,20(s8)
  4000cc:   8fc30008    lw    v1,8(s8)
  4000d0:   8fc2000c    lw    v0,12(s8)
  4000d4:   00000000    nop
  4000d8:   00621021    addu  v0,v1,v0
  4000dc:   8fc30010    lw    v1,16(s8)
  4000e0:   00000000    nop
  4000e4:   00431021    addu  v0,v0,v1
  4000e8:   8fc30014    lw    v1,20(s8)
  4000ec:   00000000    nop
  4000f0:   00431021    addu  v0,v0,v1
  4000f4:   8fc30018    lw    v1,24(s8)
  4000f8:   00000000    nop
```



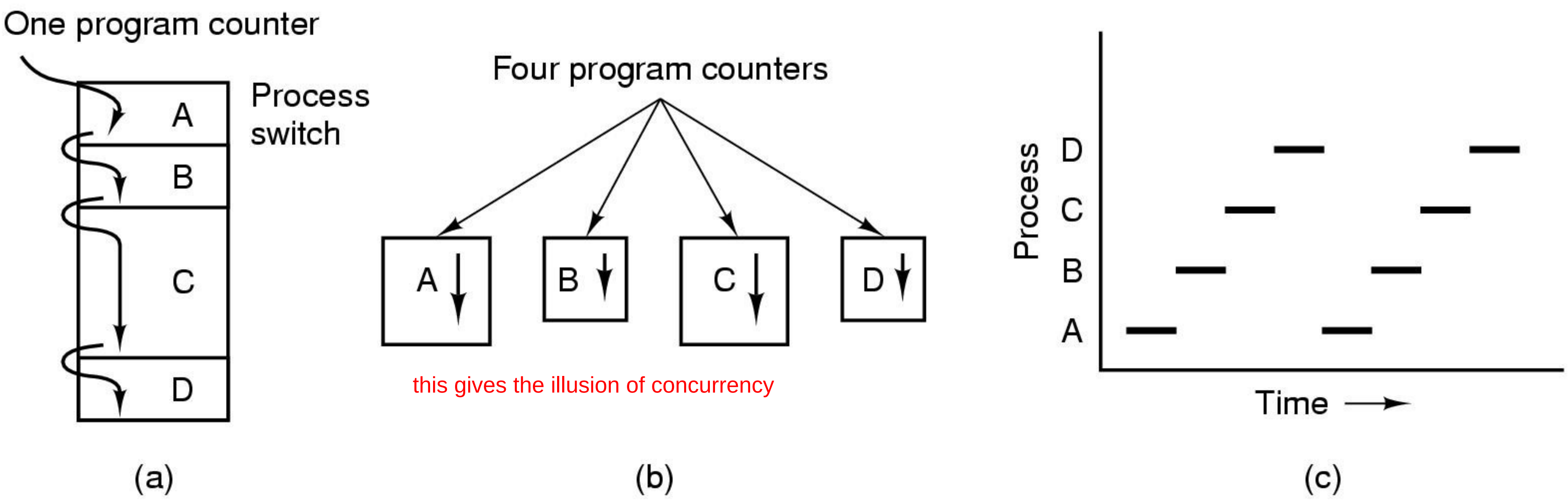
```
4000fc: 00431021  addu v0,v0,v1
400100: 8fc3001c  lw   v1,28(s8)
400104: 00000000  nop
400108: 00431021  addu v0,v0,v1
40010c: 03c0e821  move sp,s8
400110: 8fbe0000  lw   s8,0(sp)
400114: 03e00008  jr   ra
400118: 27bd0008  addiusp,sp,8
```

return

deallocate

The Process Model

- Multiprogramming of four programs

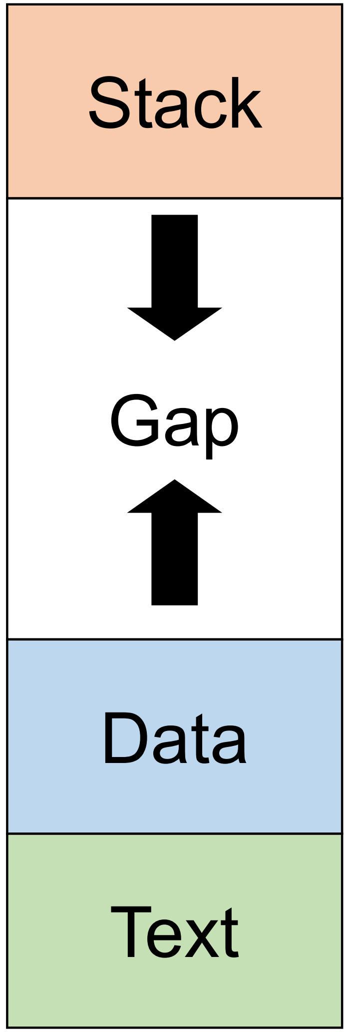


Process

what is a process???

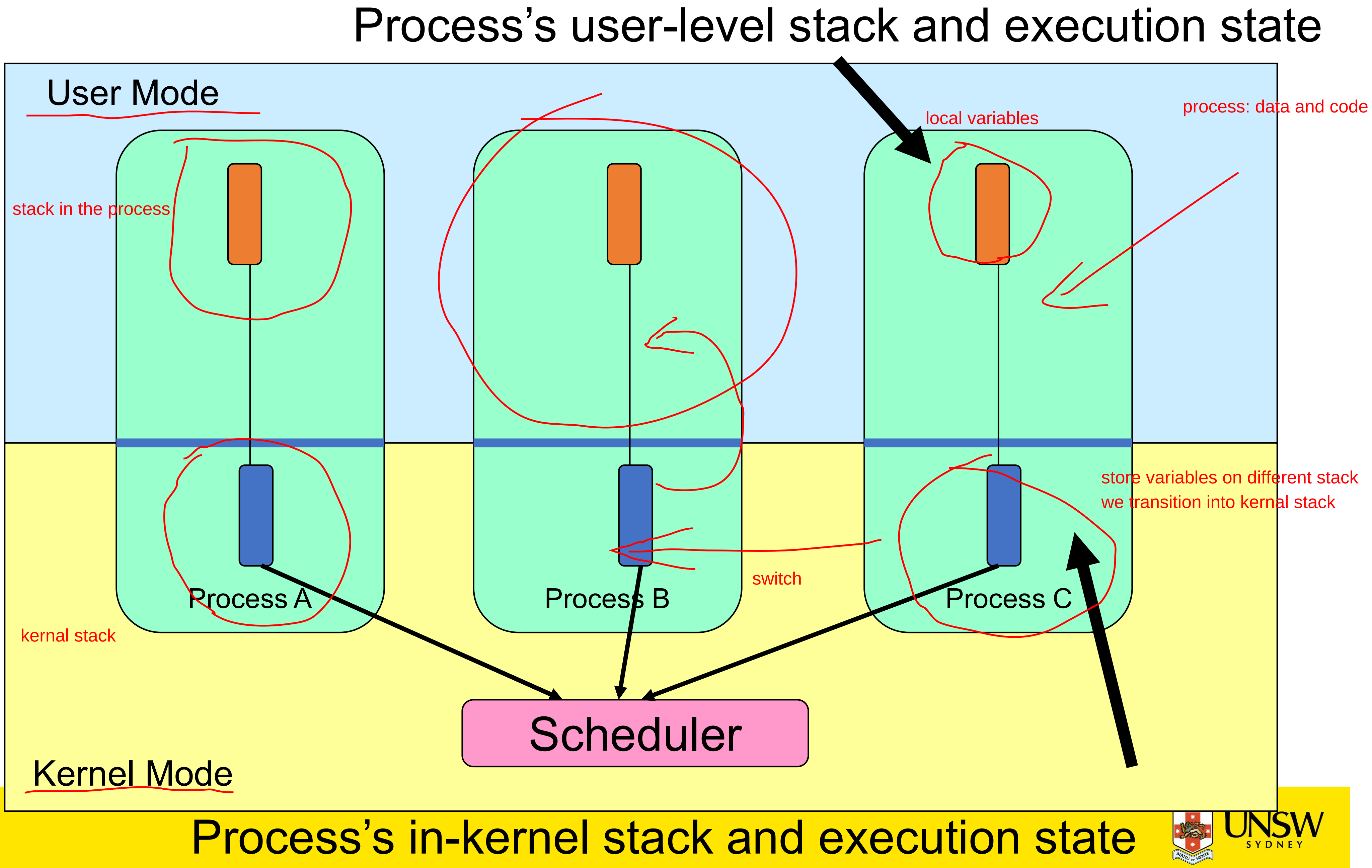
- Minimally consist of three segments
  - Text
    - contains the code (instructions)
  - Data
    - Global variables
  - Stack
    - Activation records of procedure/function/method
    - Local variables
- Note:
  - data can dynamically grow up
    - E.g., malloc()-ing
  - The stack can dynamically grow down
    - E.g., increasing function call depth or recursion

Process Memory Layout



Processes

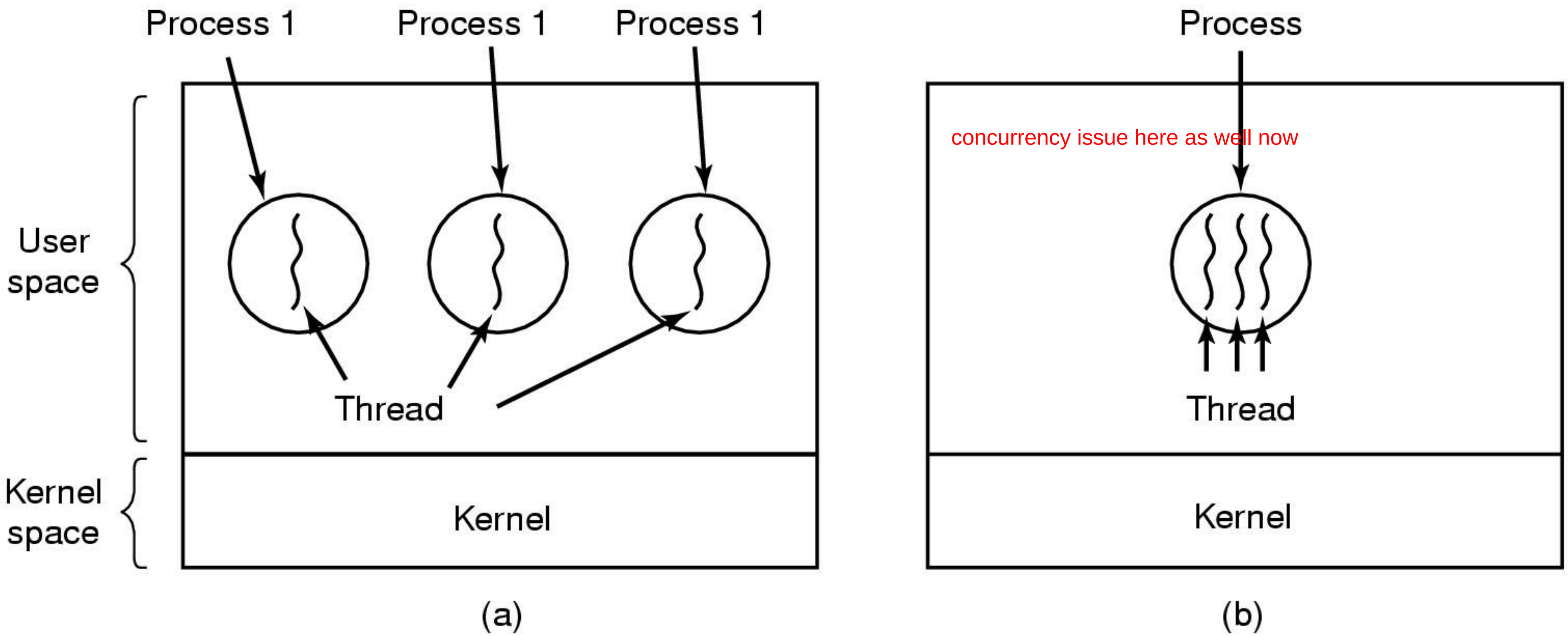
boxes are stacks storing data



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

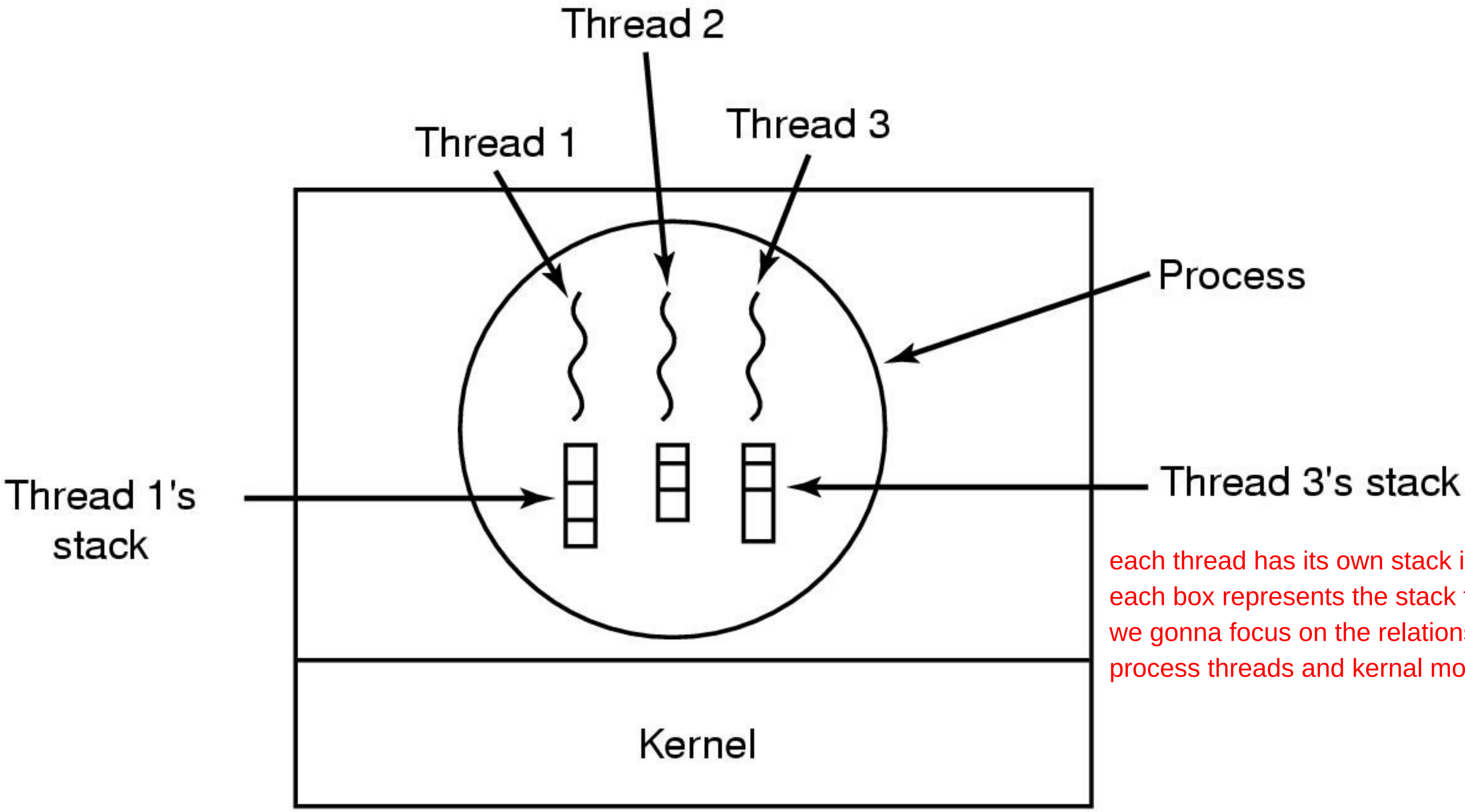
The Thread Model

we need these to bookkeep the threads in a process

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 that exist per thread

The Thread Model



each thread has its own stack in the application
each box represents the stack frame
we gonna focus on the relationship between user mode process threads and kernal mode threads

Each thread has its own stack

A Subset of POSIX threads API

```
int    pthread_create(pthread_t *, const pthread_attr_t *,
           void *(*)(void *), void *);
void   pthread_exit(void *);
```

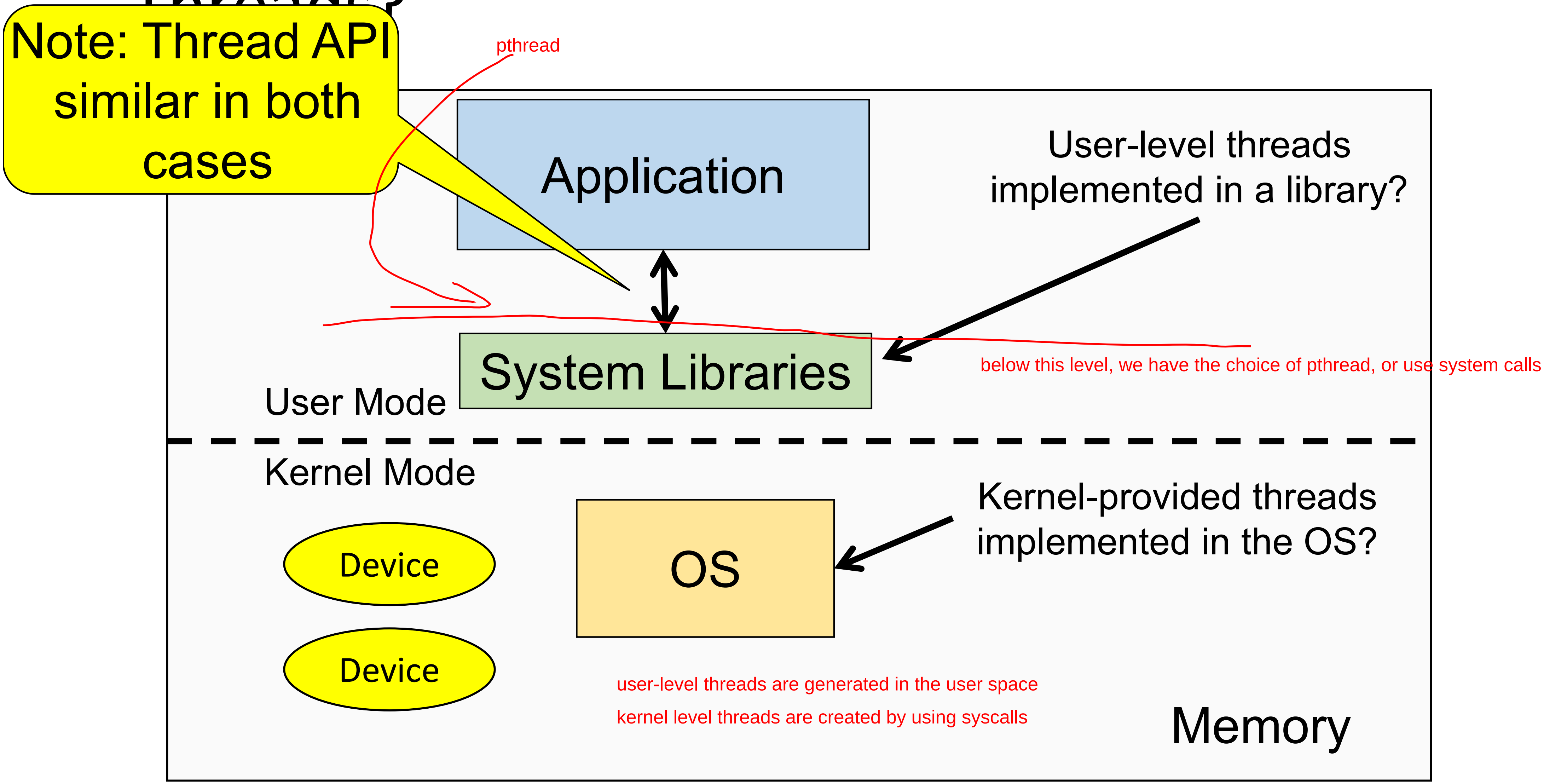
create thread

```
int    pthread_mutex_init(pthread_mutex_t *, const pthread_mutexattr_t *);
int    pthread_mutex_destroy(pthread_mutex_t *);
int    pthread_mutex_lock(pthread_mutex_t *);
int    pthread_mutex_unlock(pthread_mutex_t *);
```

synchronise

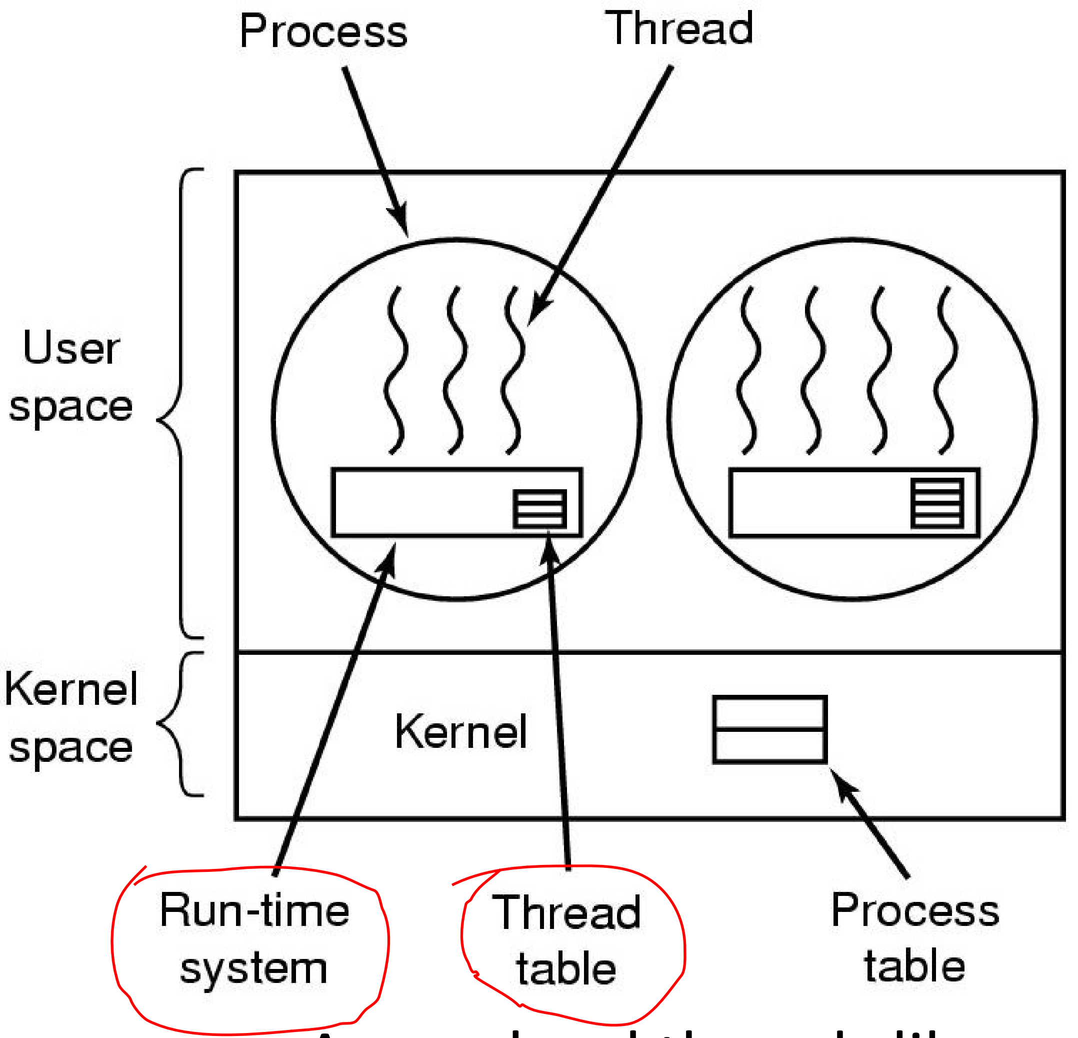
```
int    pthread_rwlock_init(pthread_rwlock_t *,
           const pthread_rwlockattr_t *);
int    pthread_rwlock_destroy(pthread_rwlock_t *);
int    pthread_rwlock_rdlock(pthread_rwlock_t *);
int    pthread_rwlock_wrlock(pthread_rwlock_t *);
int    pthread_rwlock_unlock(pthread_rwlock_t *);
```

Where to Implement Application Threads?



Implementing Threads in User Space

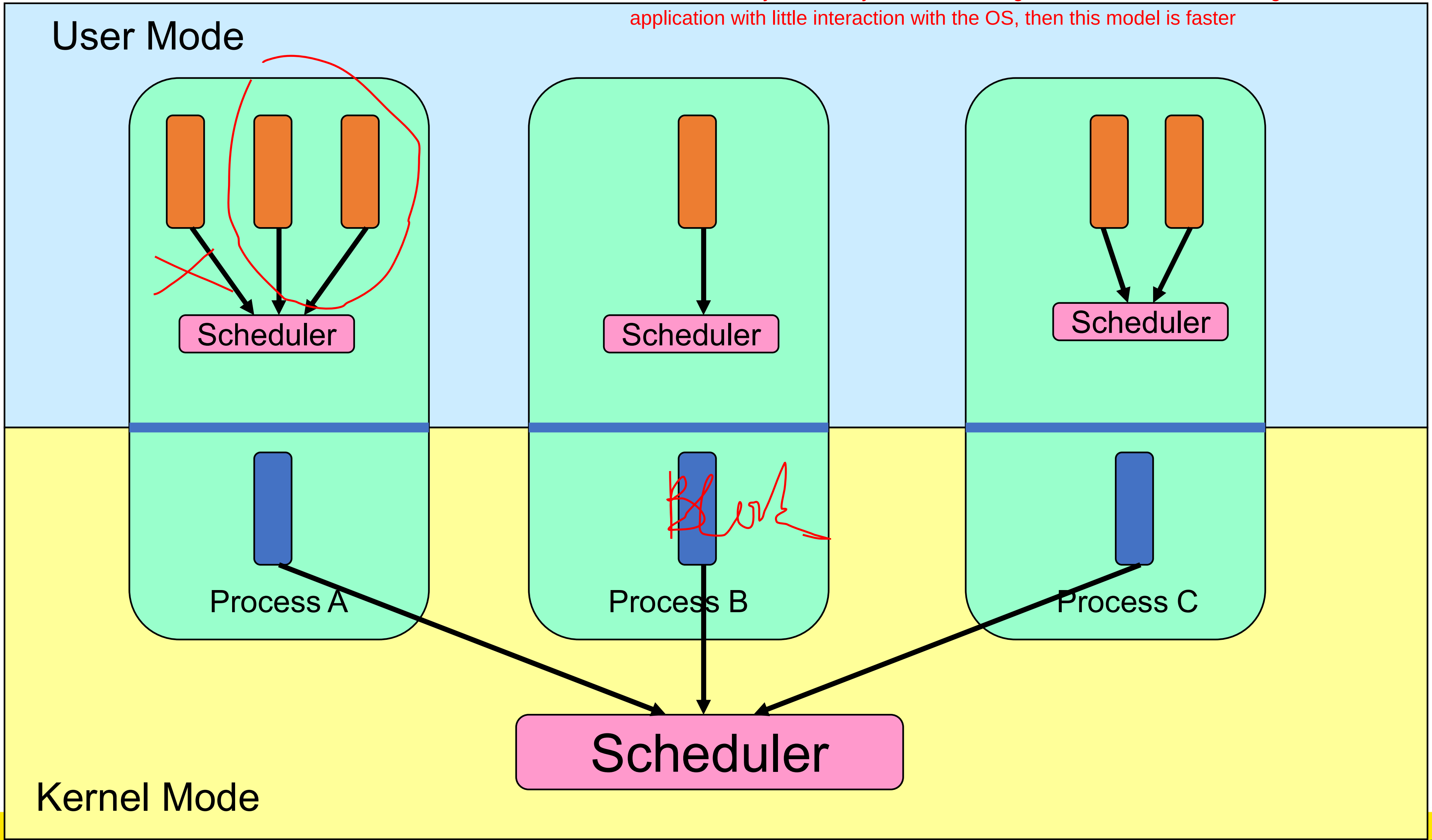
choice 1



A user-level threads library

User-level Threads

downside of this mode, if a thread performs a syscall, if the system call is blocked, then the whole process is blocked, OS has no idea of the concurrency exist, so the option for the OS is to switch to another process
for example, if process A and process B are blocked, and we have 2 cpu machines, only process C can be ran. There is no ability in this mode to take advantage of multiple CPUs
The advantage of this approach is if I want to create a thread, I just call into the local library, I can synch the threads very fast locally without involving the OS. If we want to manage a lot of concurrency in the application with little interaction with the OS, then this model is faster



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 inside the same process, 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

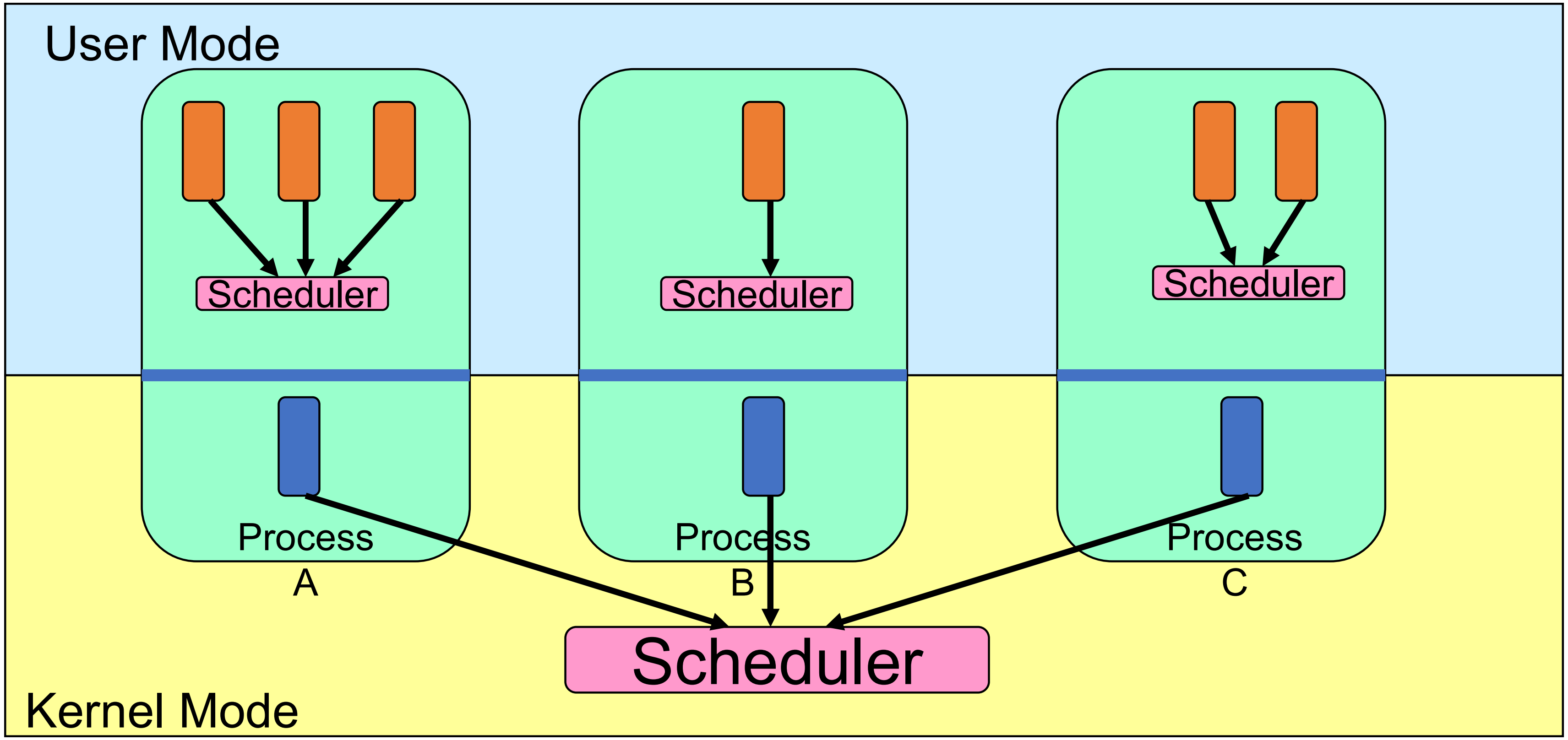
- Pros
  - Thread management and switching at user level is much faster than doing it in kernel level
    - No need to trap (take syscall exception) into kernel and back to switch
  - Dispatcher algorithm can be tuned to the application
    - E.g. use priorities
  - Can be implemented on any OS (thread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

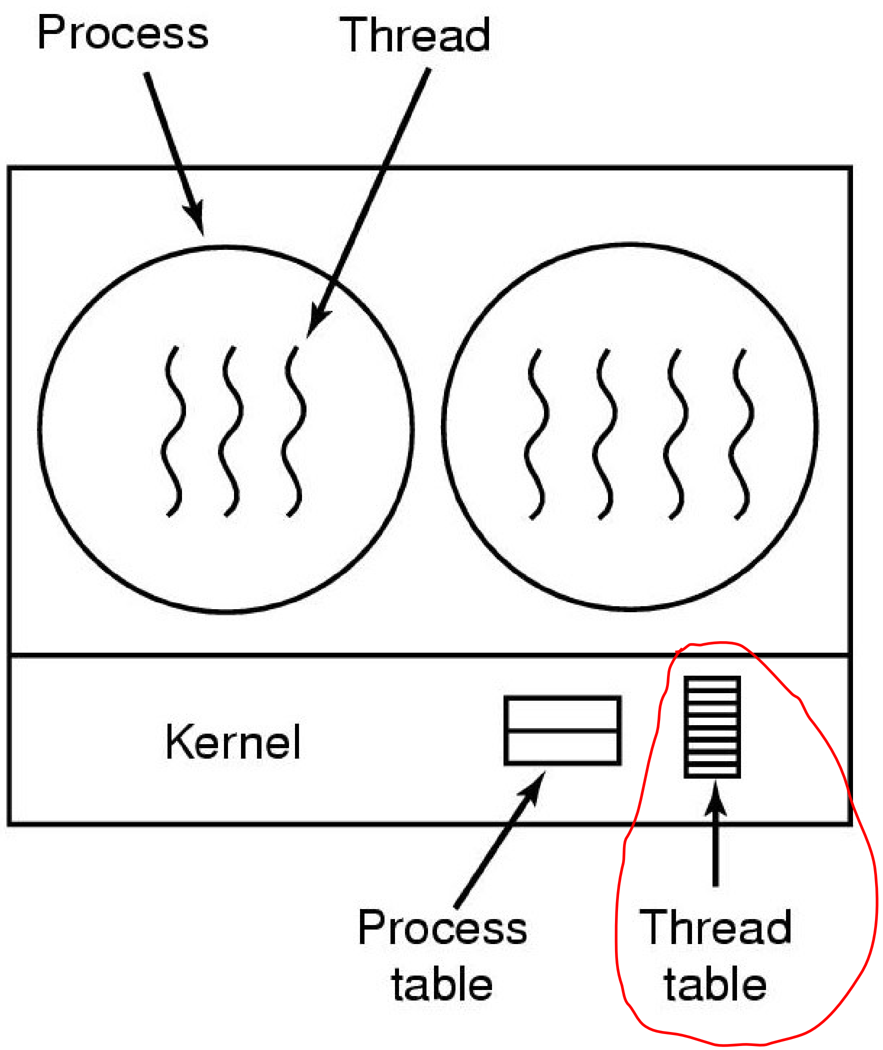
User-Level Threads

- 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



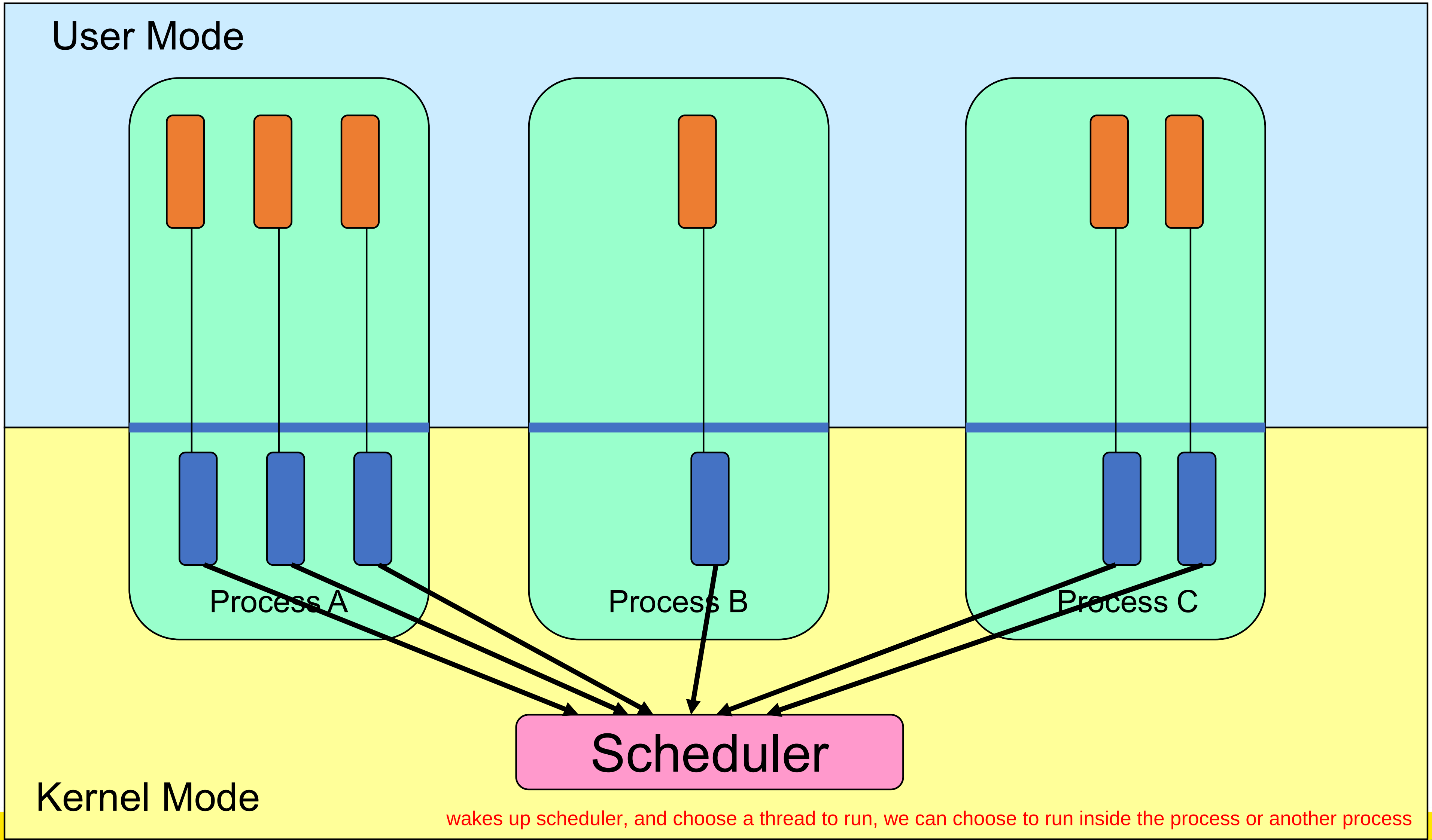
Implementing Threads in the Kernel

alternative



A threads package managed by the kernel

Kernel-provided Threads



wakes up scheduler, and choose a thread to run, we can choose to run inside the process or another process

Kernel-provided Threads

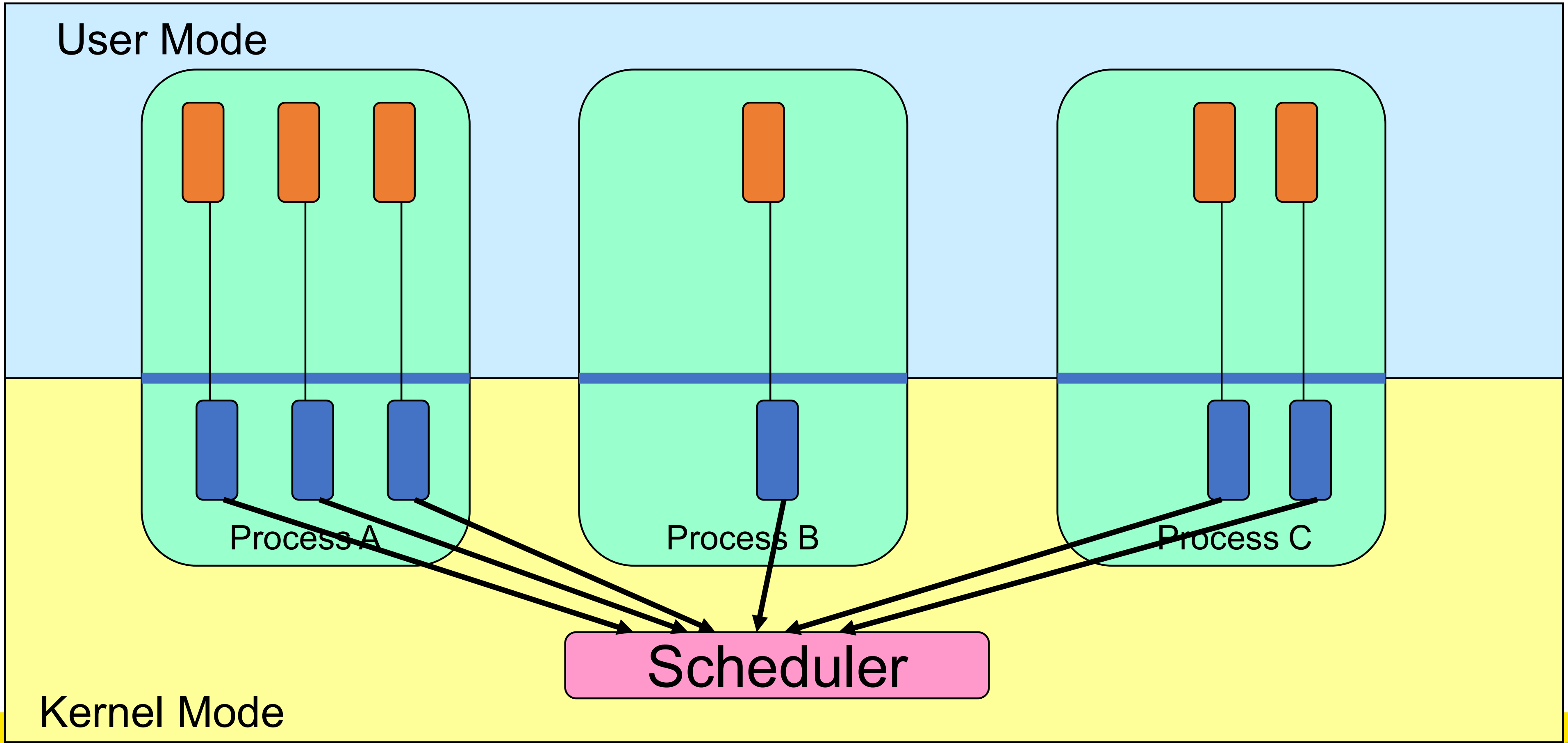
- Also called kernel-level threads
  - Even though they provide threads to applications
- Threads are implemented by the kernel
  - TCBs are stored in the kernel
    - A subset of information in a traditional PCB
      - The subset related to execution context
    - TCBs have a PCB associated with them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

Kernel-provided Threads

- Cons
  - Thread creation and destruction, and blocking and unblocking threads requires kernel entry and exit.
    - More expensive than user-level equivalent

Kernel-provided Threads

- 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 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*

in C code, a line of code can contain multiple lines of instructions, and context switch is for the switch of instruction

A thread switch can happen between any two instructions

Note instructions do not equal program statements

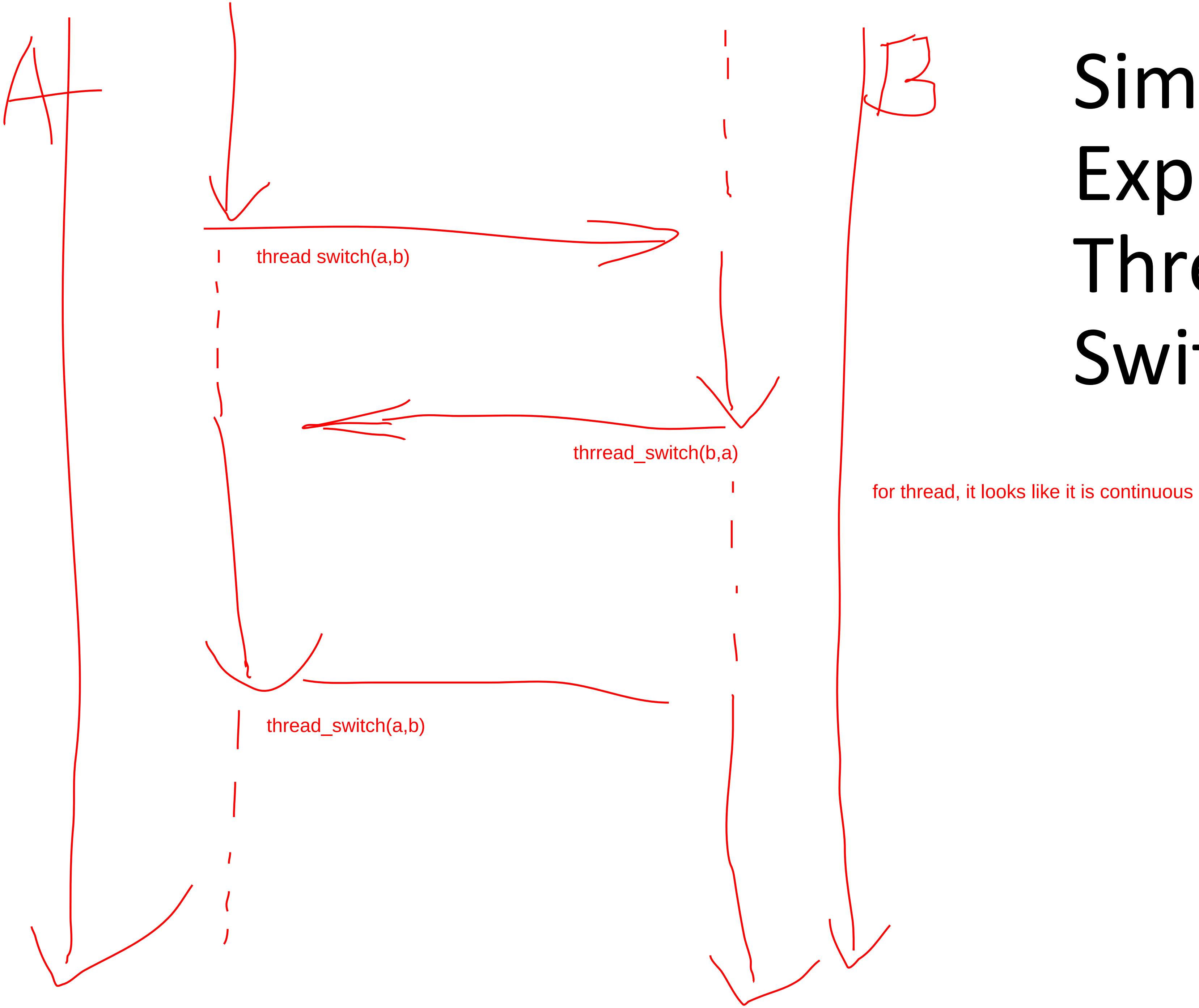
Context Switch

- 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)

$\Rightarrow$OS must save all state that affects the thread

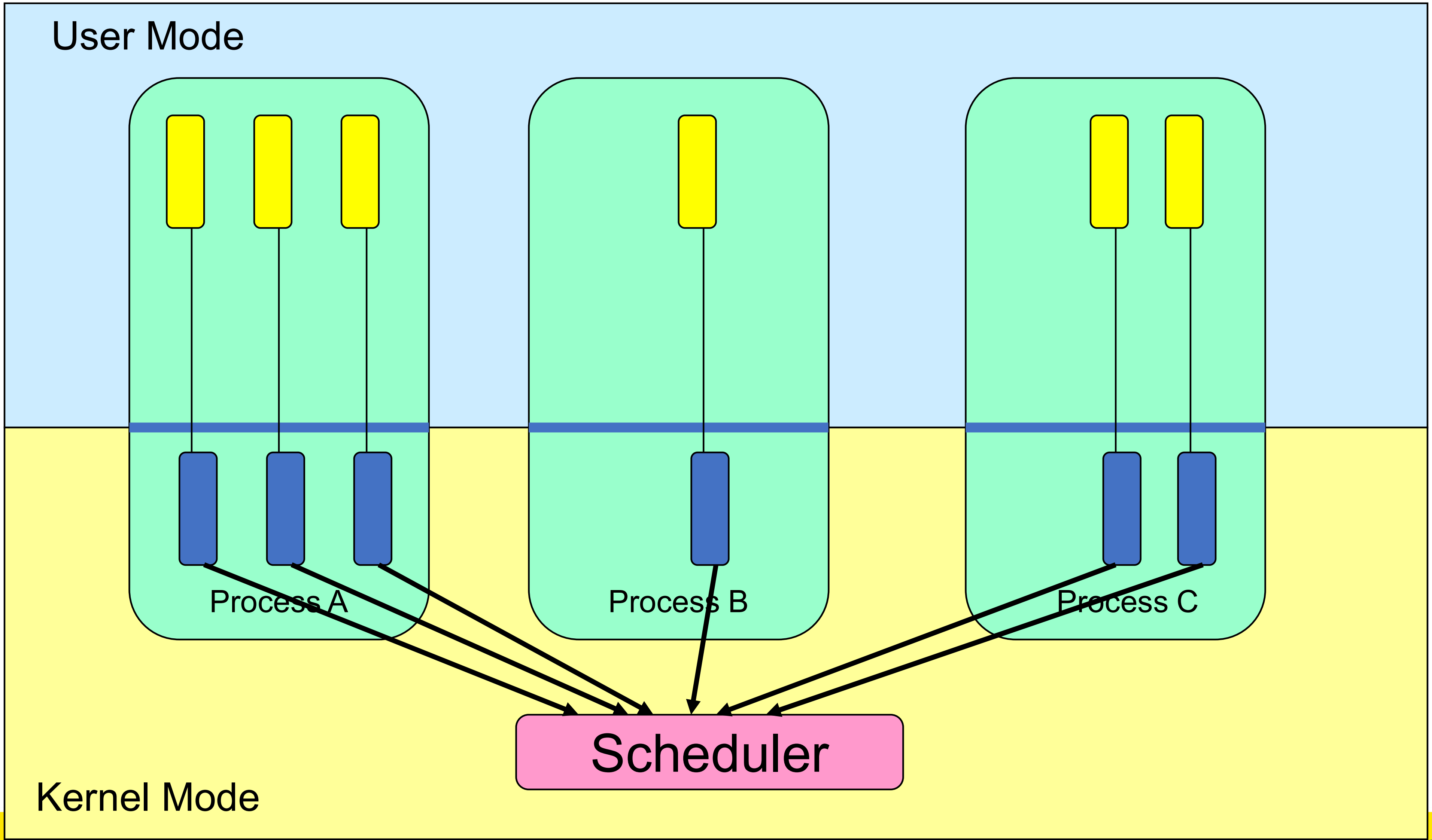
- This state is called the *process/thread context*
- Switching between process/threads consequently results in a *context switch*.

Simplified Explicit Thread Switch



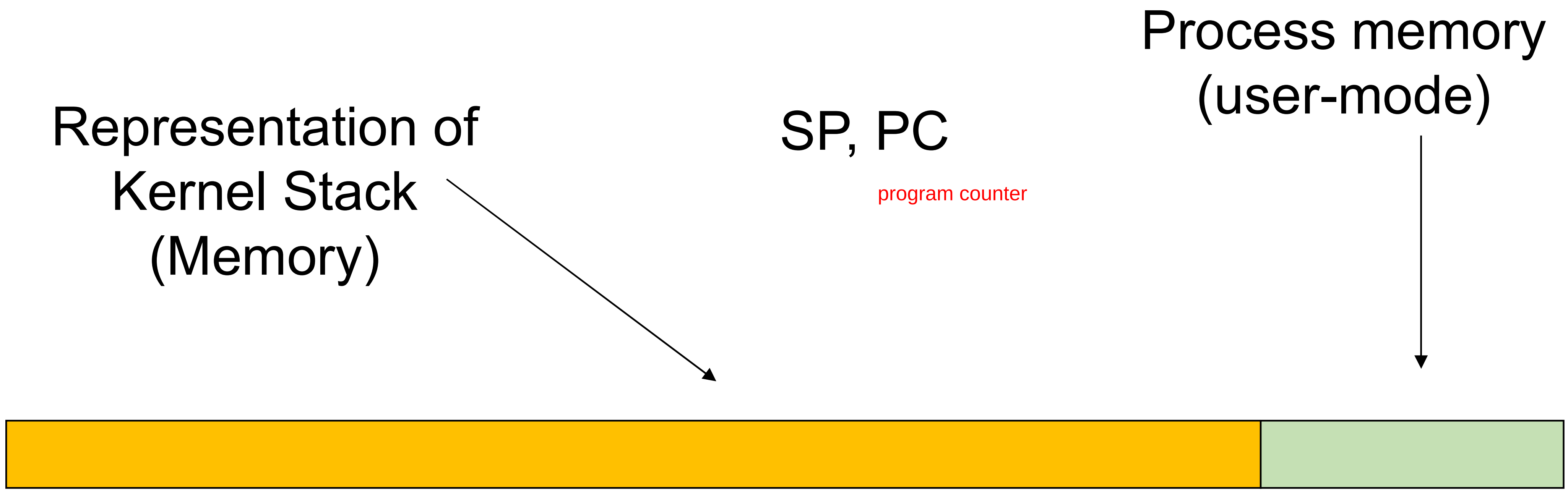
Assume Kernel-Level Threads

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

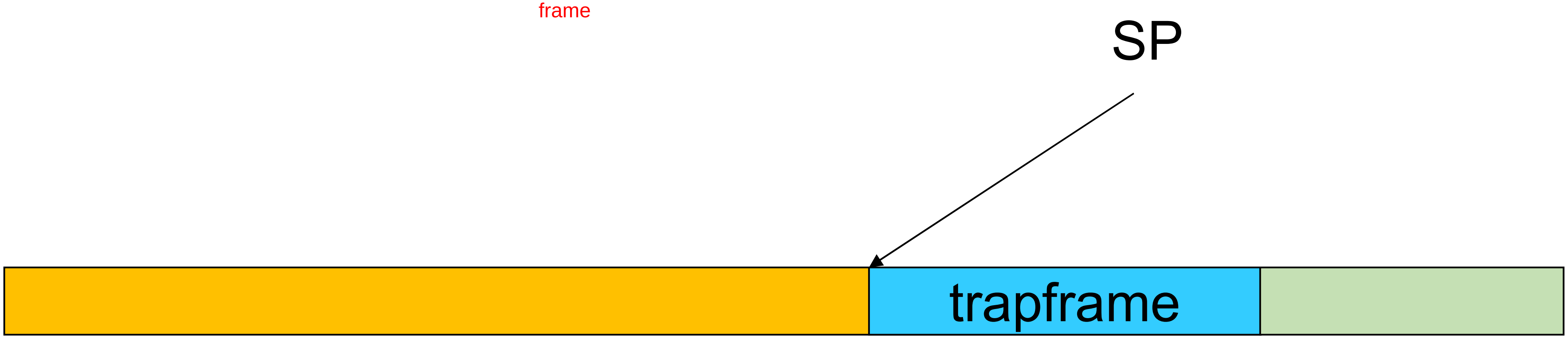
at the beginning, the stack pointer is pointing to somewhere in the application
they both point to the application, when we get an exception, syscall, or interrupt, we switch to kernel stack

Example Context Switch

frame on the stack that stores what application is doing

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

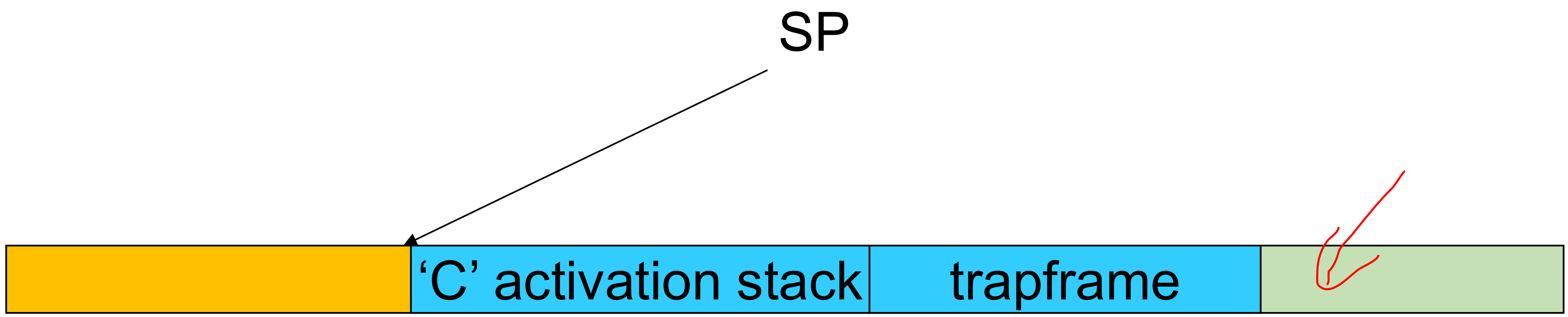
a trapframe is pushed to the kernel stack
we are saving the states of the register existing in the processor at the point when execution occurs
this trapframe represents what's going on in the application, everything in the application stack will be stored in the trap frame



Example Context Switch

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

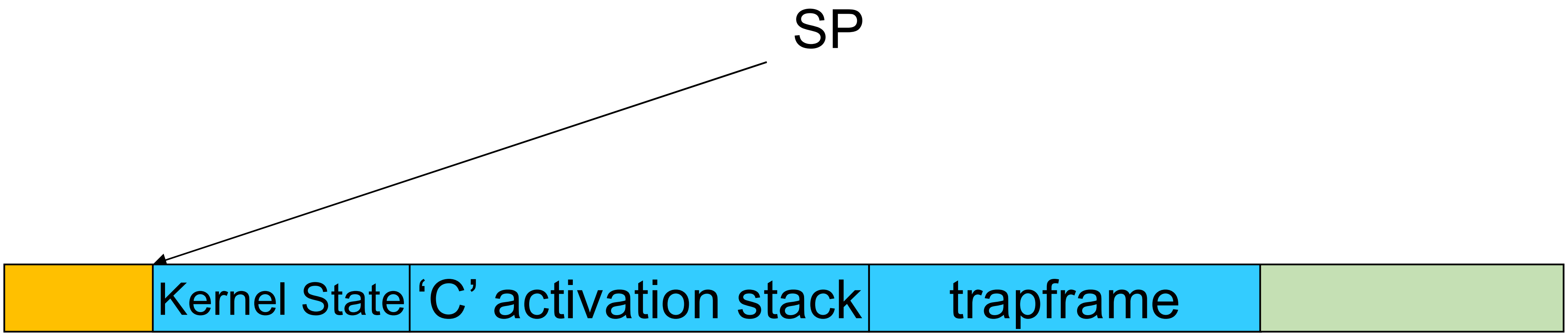
it grows to a point where we decide to switch to another stack



Example Context Switch

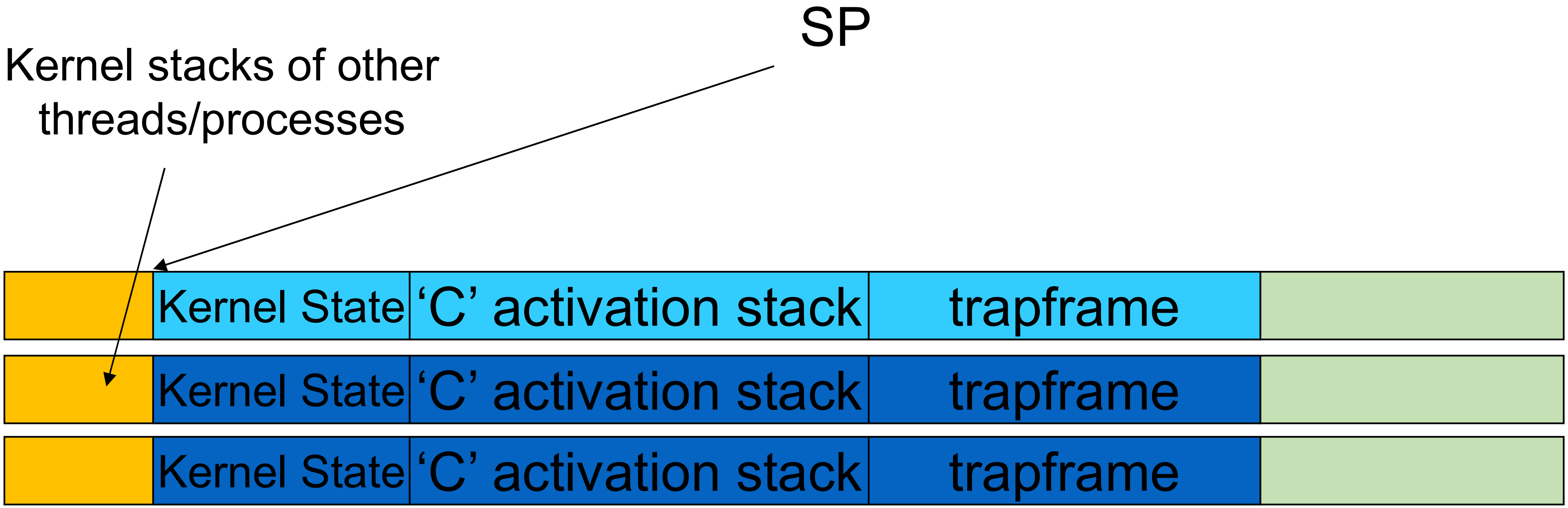
- The kernel decides to perform a context switch
  - It chooses a target thread (or process)
  - It pushes remaining kernel context onto the stack

we save the execution in the kernel into a different process(the other process must exist)



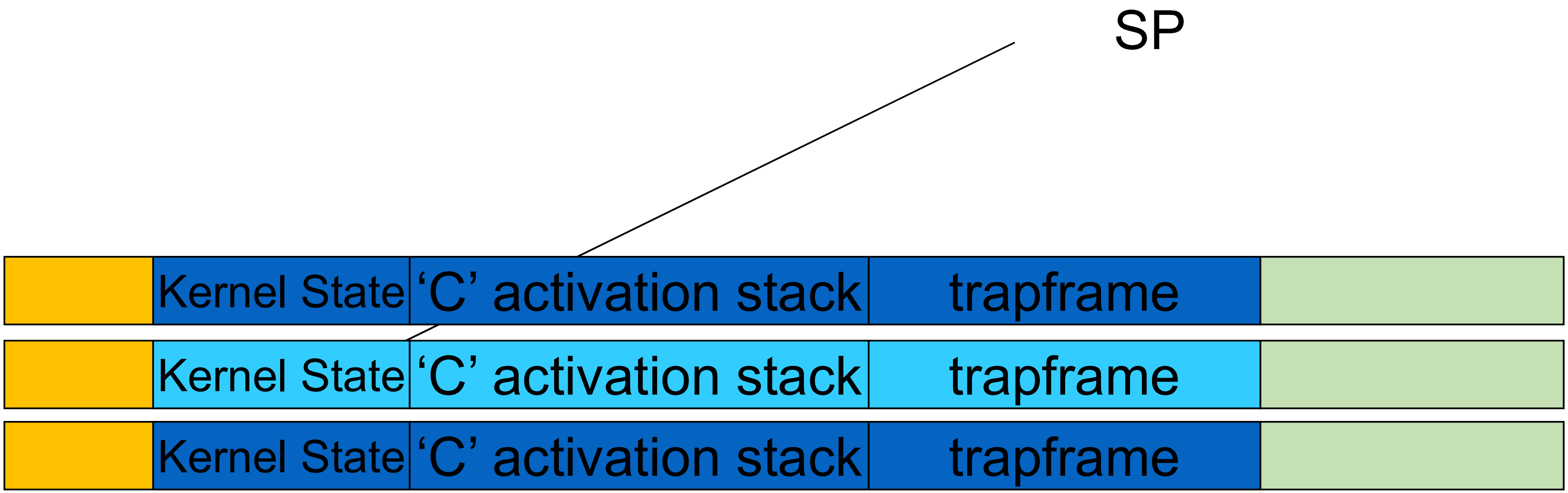
Example Context Switch

- 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



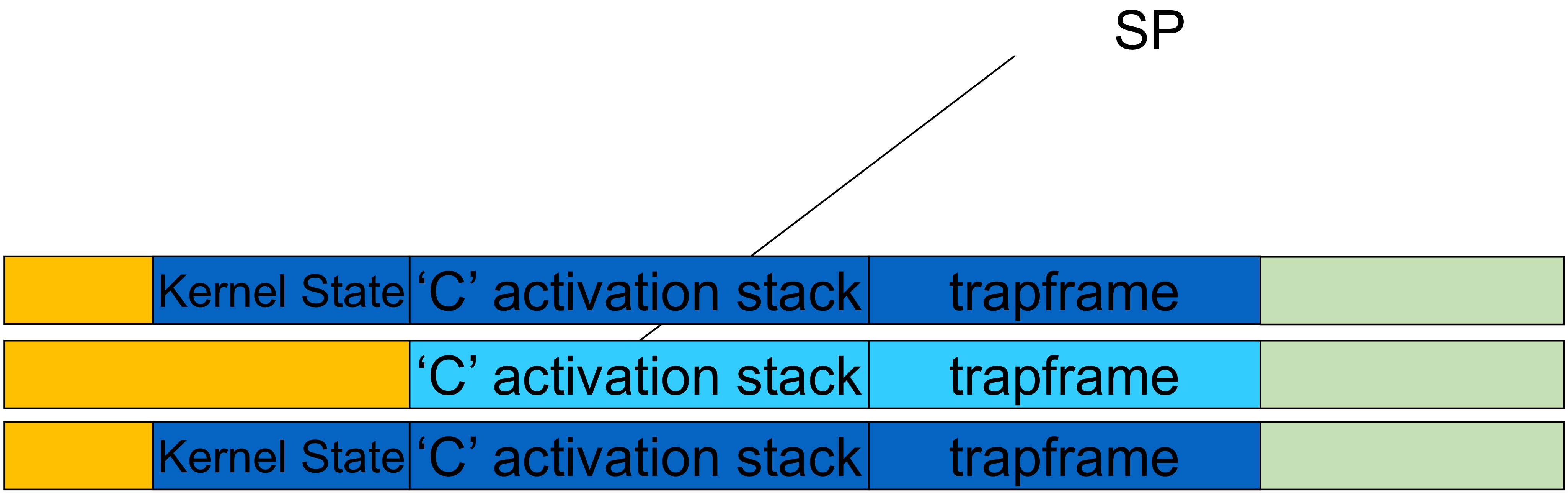
Example Context Switch

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



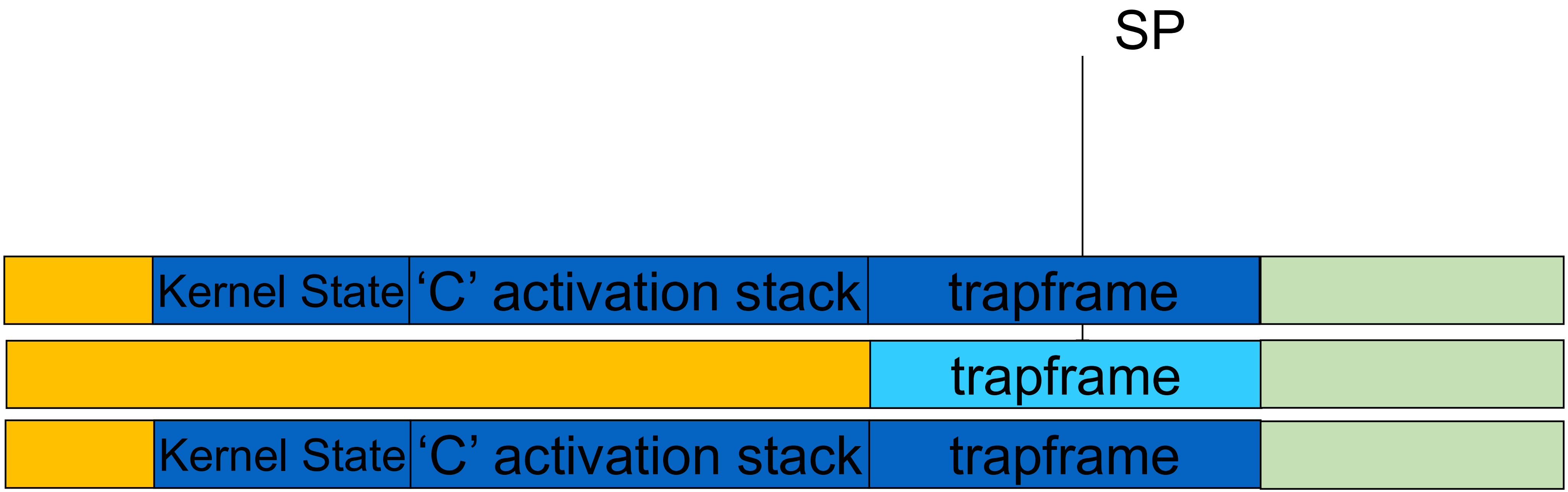
Example Context Switch

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



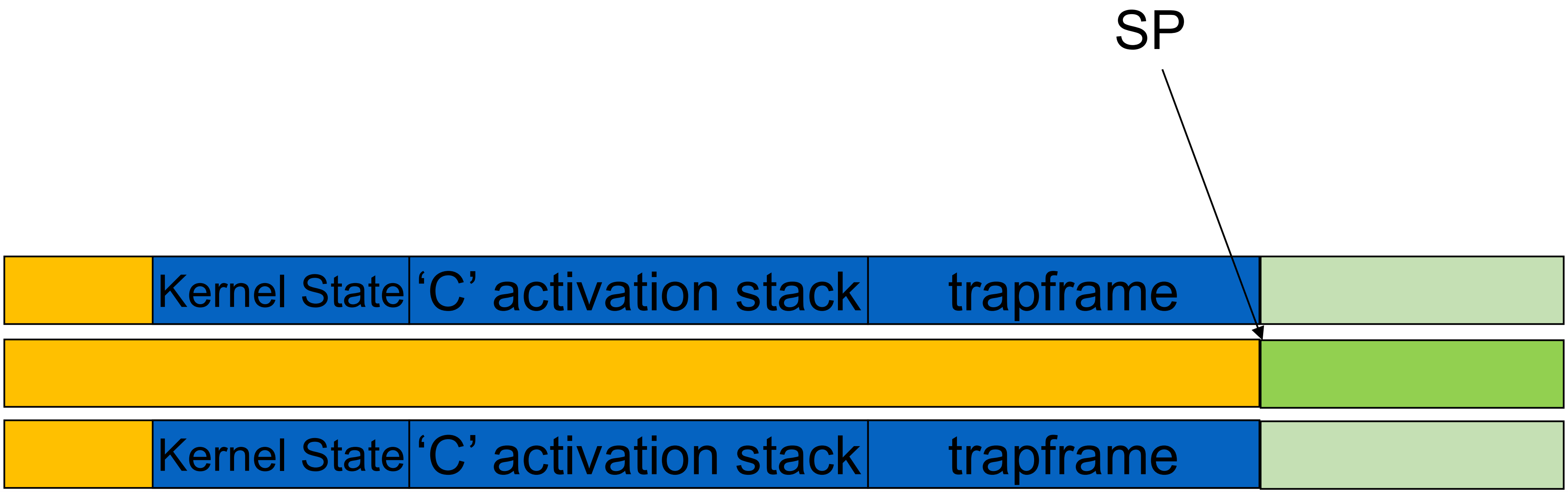
Example Context Switch

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



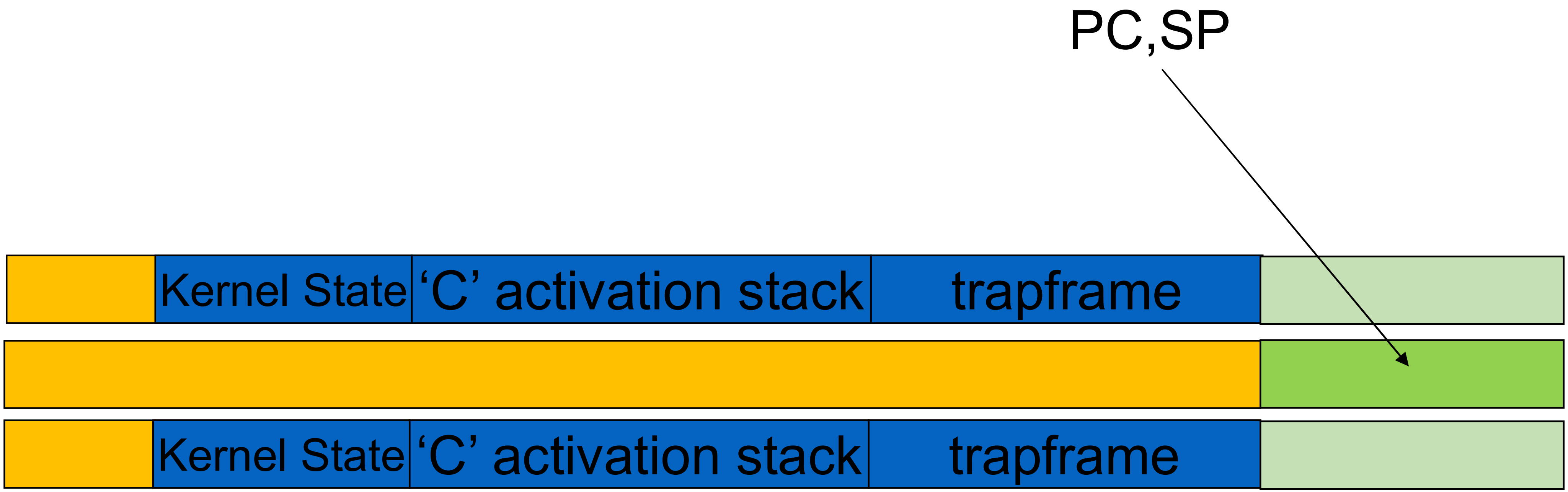
Example Context Switch

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



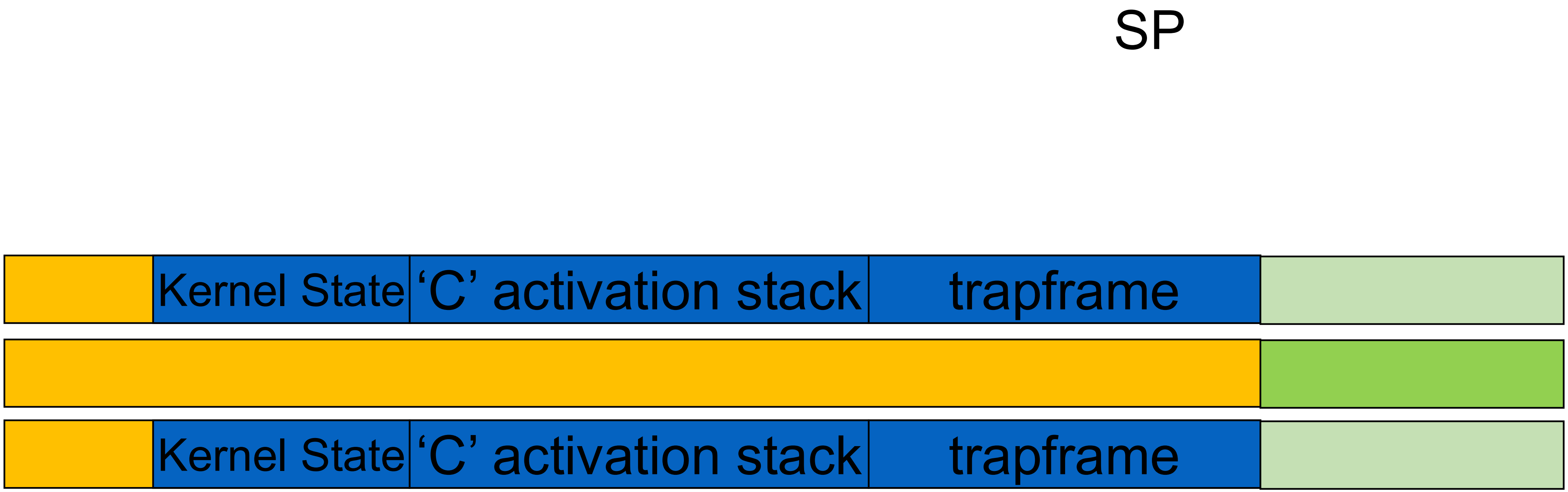
Example Context Switch

- 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_context);

}
```

ready queue

if removal is unesscaful

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)
```

store the current/switch from old stack pointer in the old thread 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 */
```

Revisiting Thread Switch

Thread a

Thread b

```
switchframe_switch(a,b)
{

}
```

```
}
```

```
switchframe_switch(b,a)
{
```

```
switchframe_switch(a,b)
{
```

```
}
```