

Microkernel Construction

I.2 – Threads, System Calls, Thread Switching


Lecture Summer Term 2017

Wednesday 15:45-17:15 R131, 50.34 (INFO)

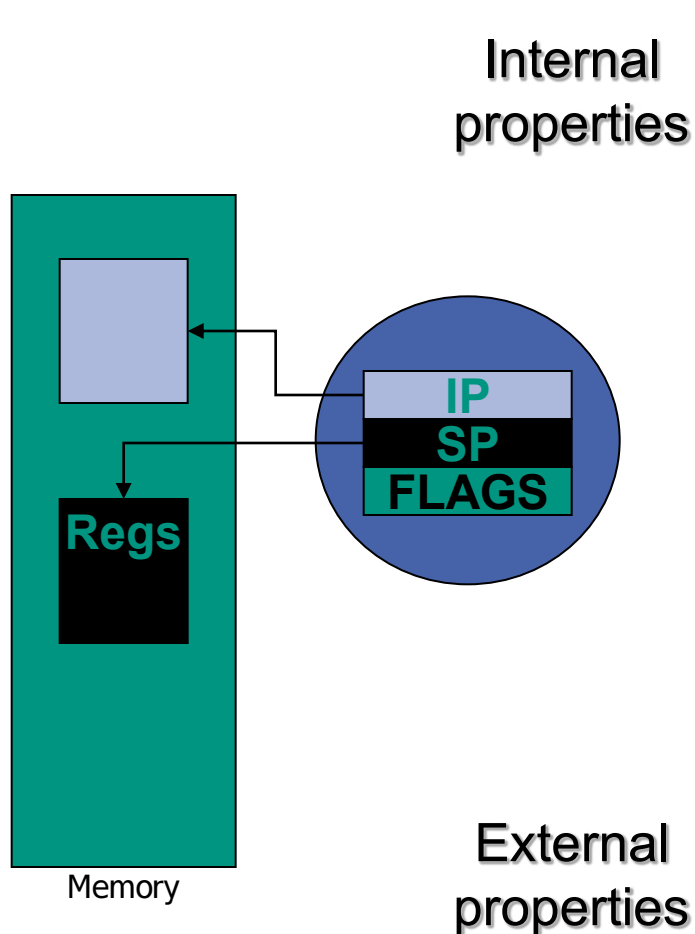
Jens Kehne, Marius Hillenbrand
Operating Systems Group, Department of Computer Science



Fundamental Abstractions

- Thread
 - Address space
- 
- What is a thread?
 - How to implement it?
-
- *What conclusions can we draw from our analysis with respect to μ -kernel construction?*

Thread Properties



- Register set
 - General purpose registers, IP, and SP
- Stack
- Status
 - Flags, privilege, etc.
- OS-specific state
 - Priority, time, etc.

- Address space
- Unique ID
- Communication status

Construction Conclusion

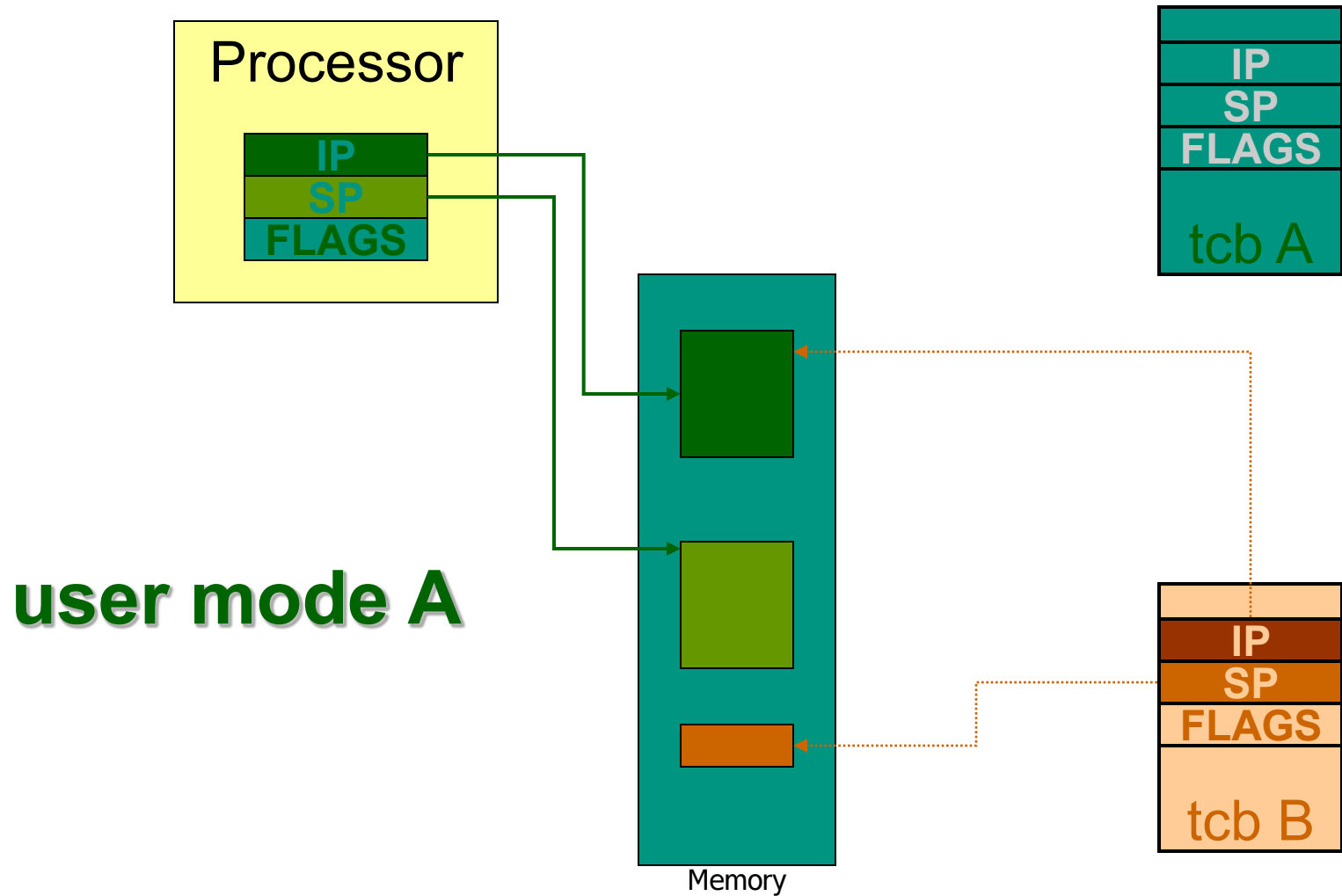
- Thread state must be saved/restored on thread switch
- We need a **Thread Control Block** (TCB) per thread
- TCBs must be kernel objects

- **TCBs implement threads**

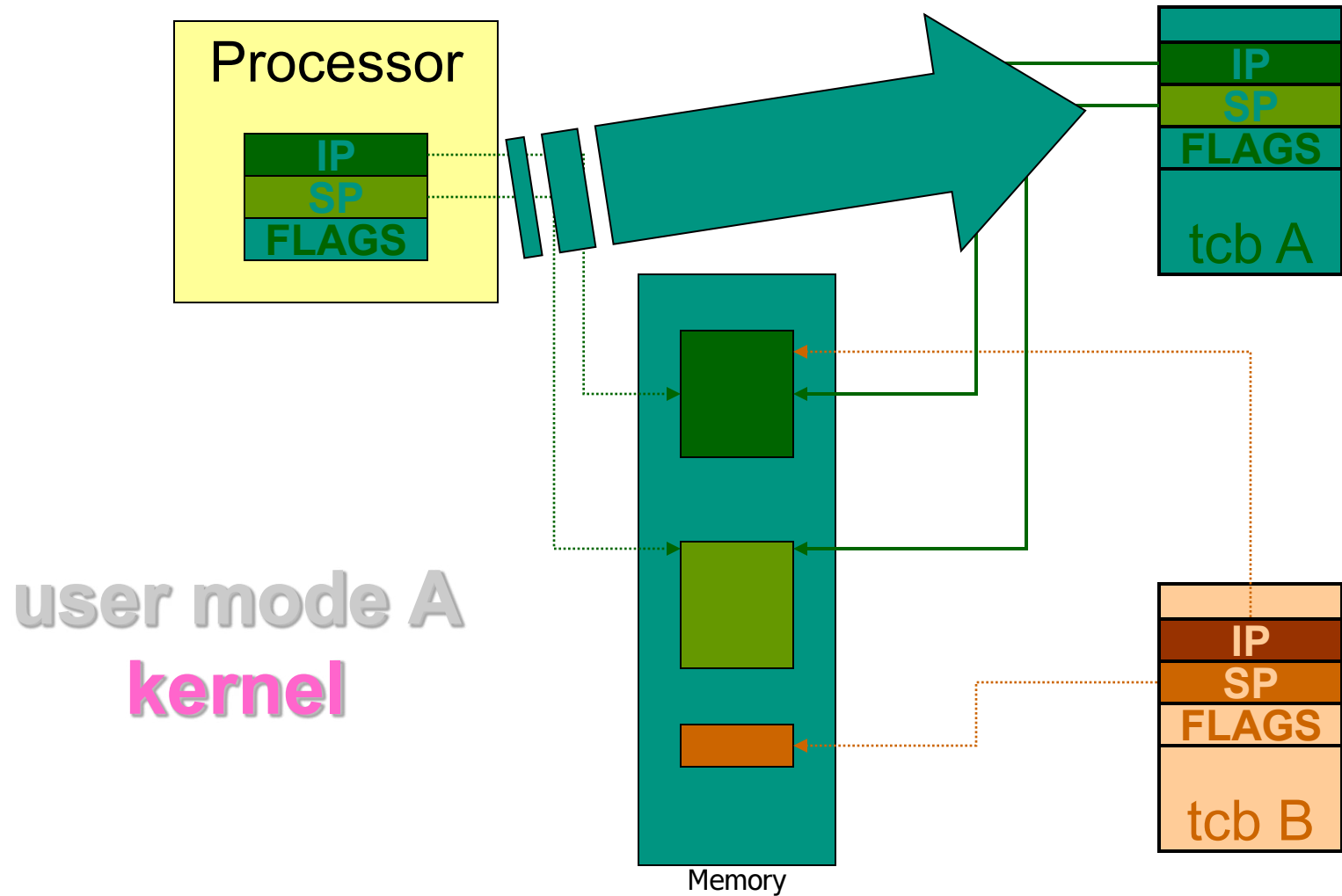
At least partially. We have found some good reasons to implement parts of the TCB in user memory (→ IPC).

- We often need to find
 - The TCB of any thread using its global ID
 - The TCB of the currently executing thread (per processor)

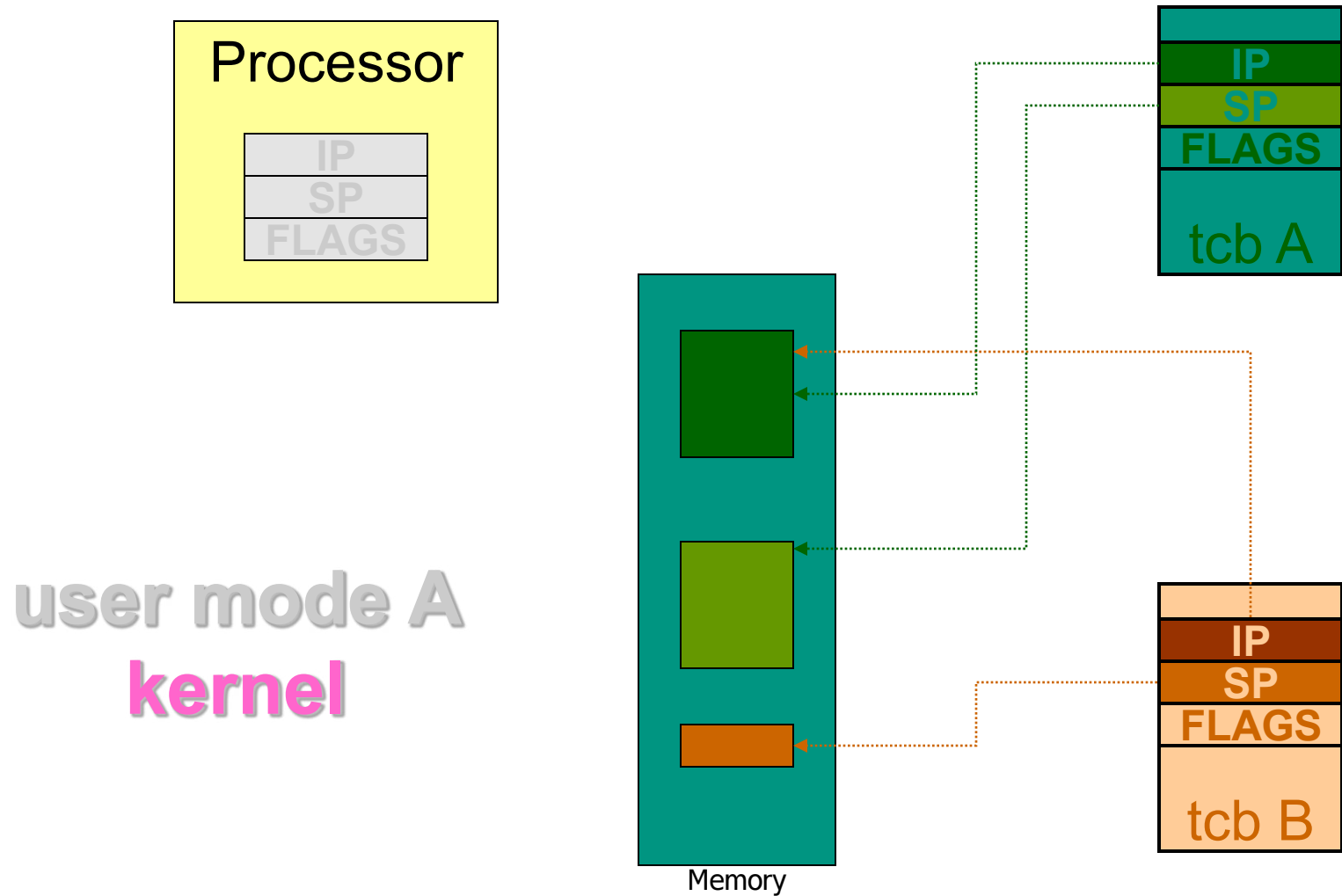
Thread Switch **A** → **B**



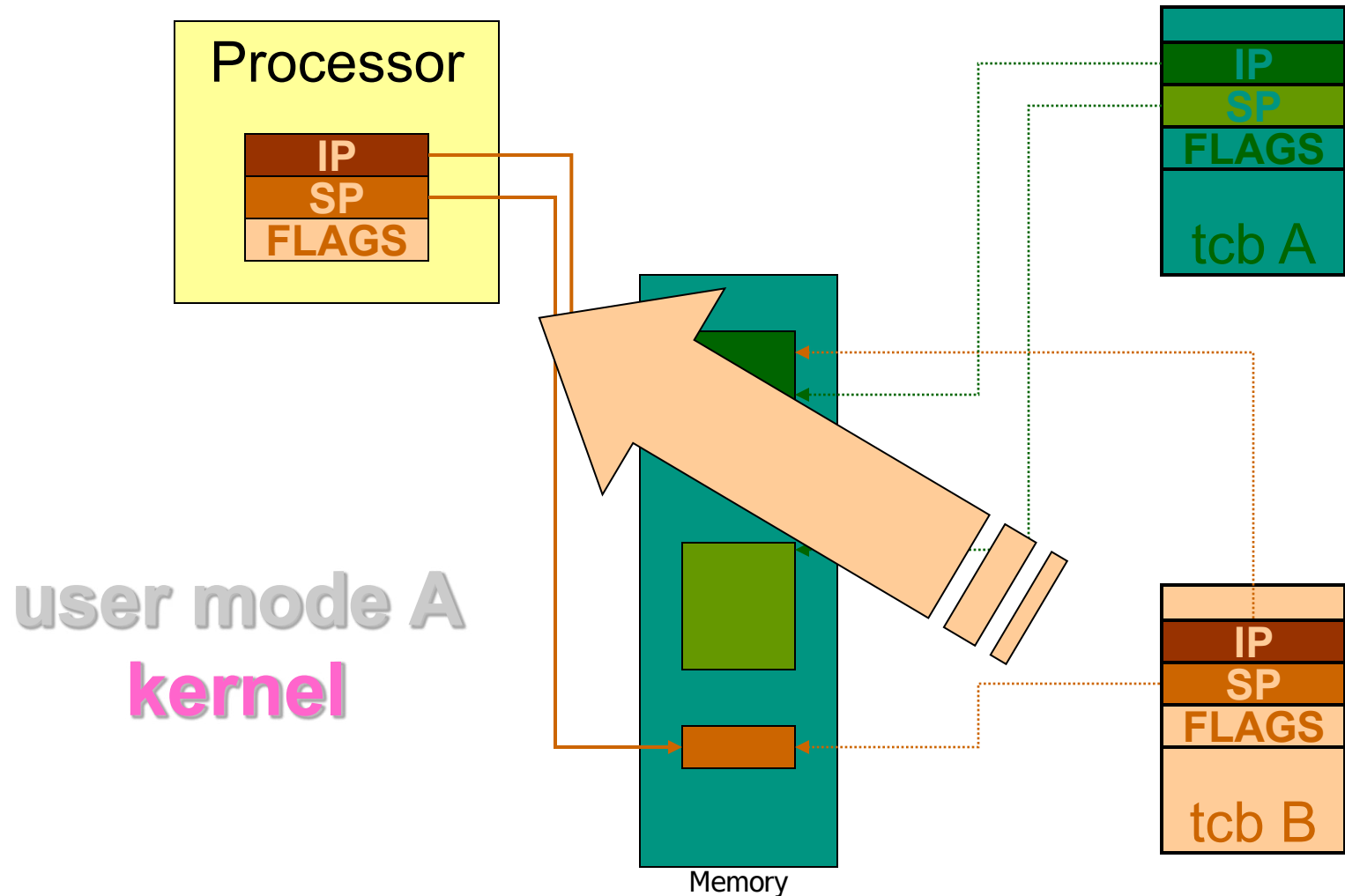
Thread Switch $A \rightarrow B$



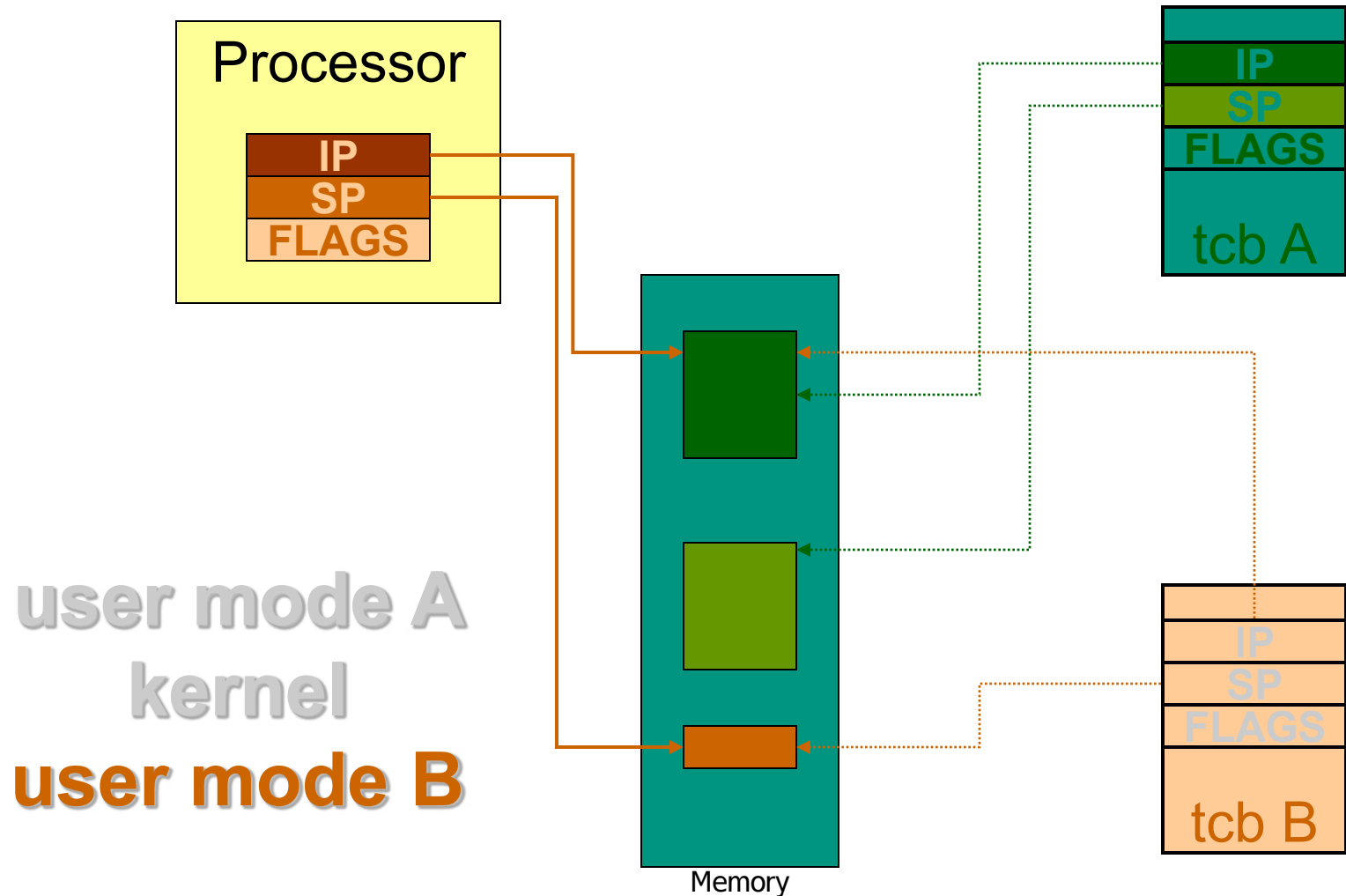
Thread Switch **A** → **B**



Thread Switch $A \rightarrow B$



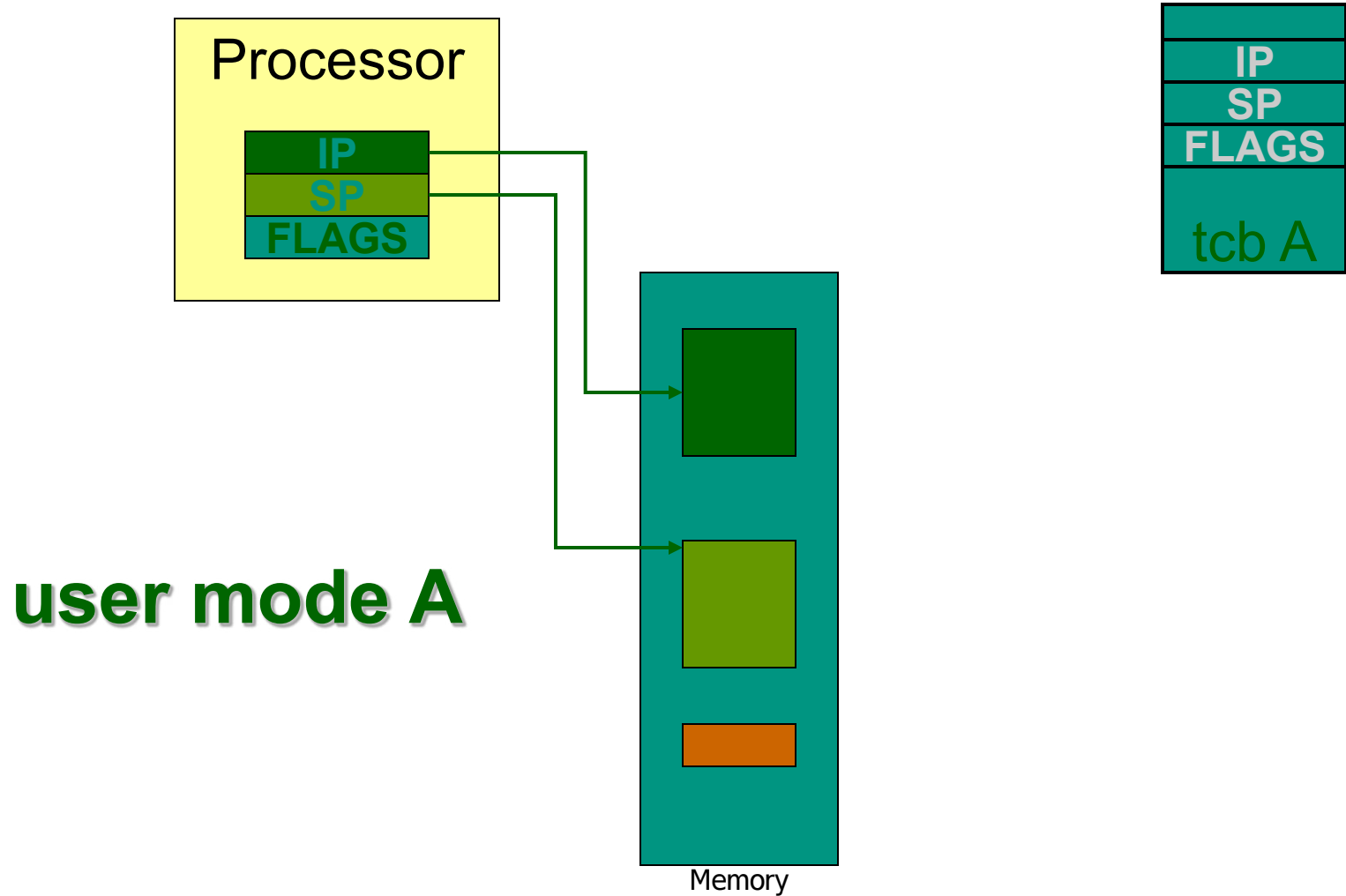
Thread Switch **A** → **B**



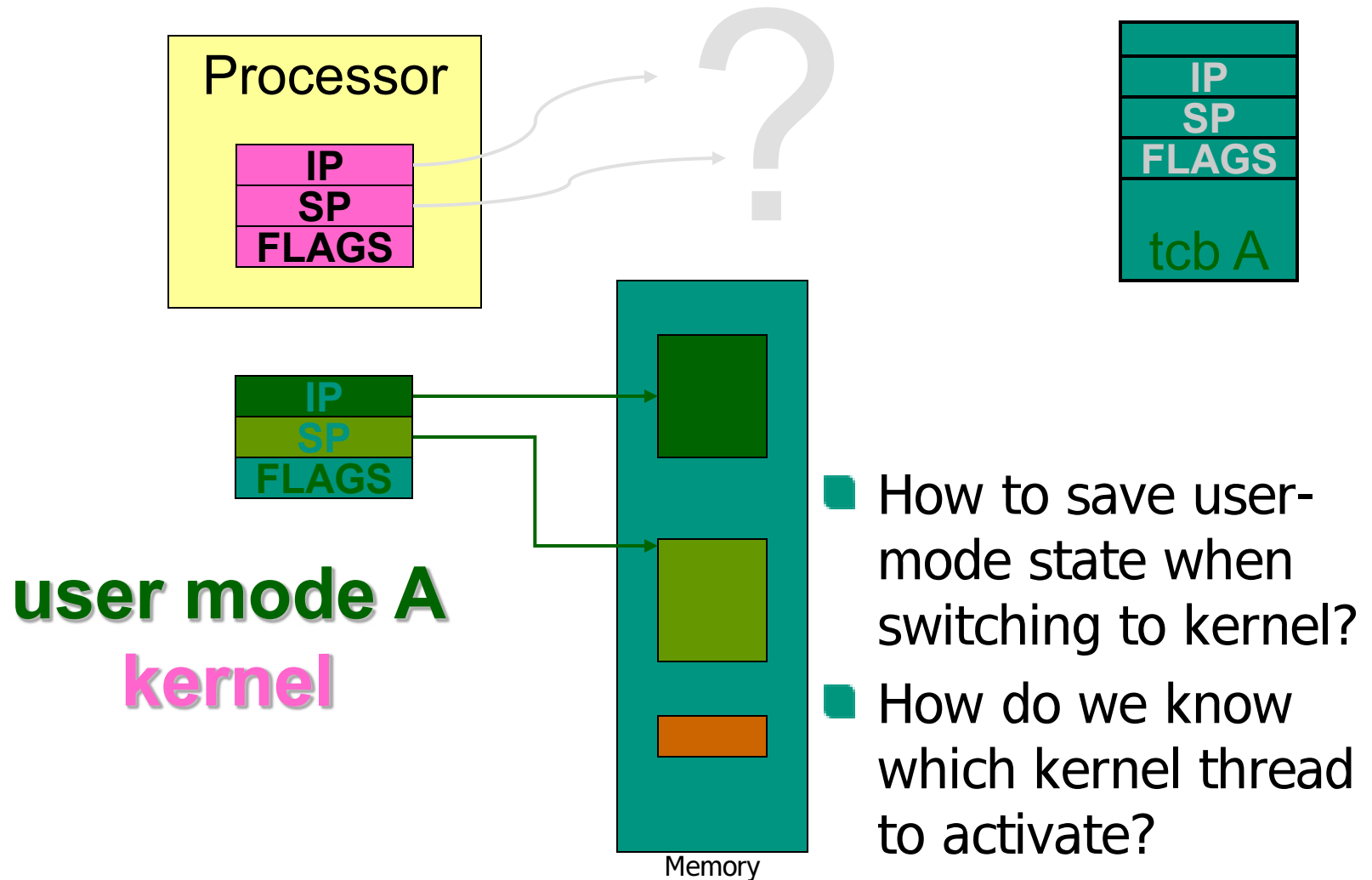
Thread Switch **A** → **B**

- Thread **A** is running in user mode
- Thread **A** experiences its end of time slice or is preempted by a (device) interrupt
- We enter kernel mode
- The microkernel saves the status of thread **A** on **A's** T C B
- The microkernel loads the status of thread **B** from **B's** T C B
- We leave kernel mode
- Thread **B** is running in user mode

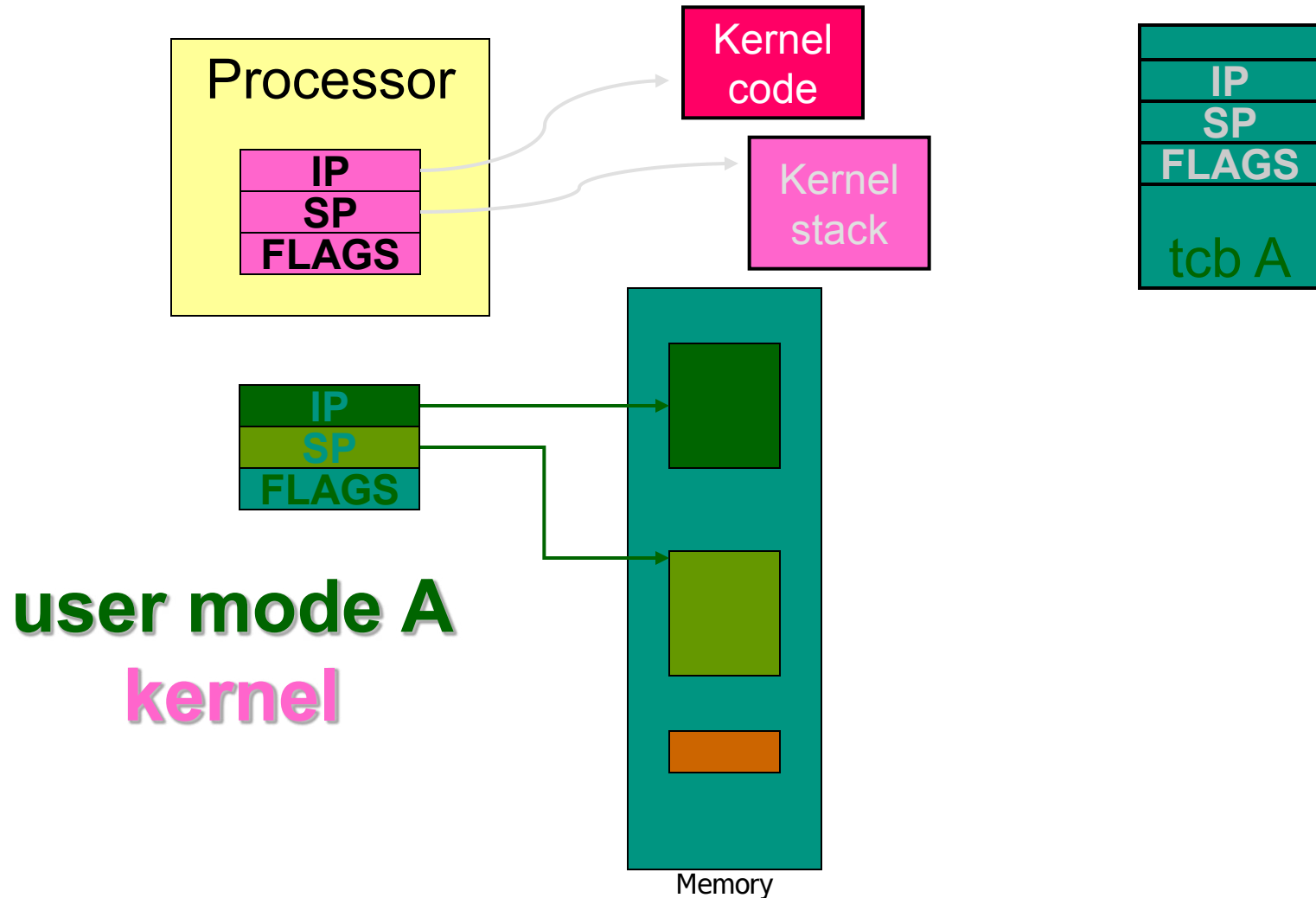
Thread Switch **A** → **kernel** → **B**

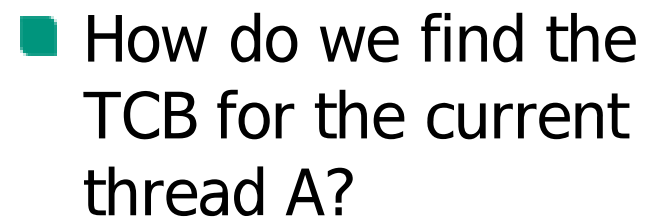


Thread Switch **A** → **kernel** → **B**

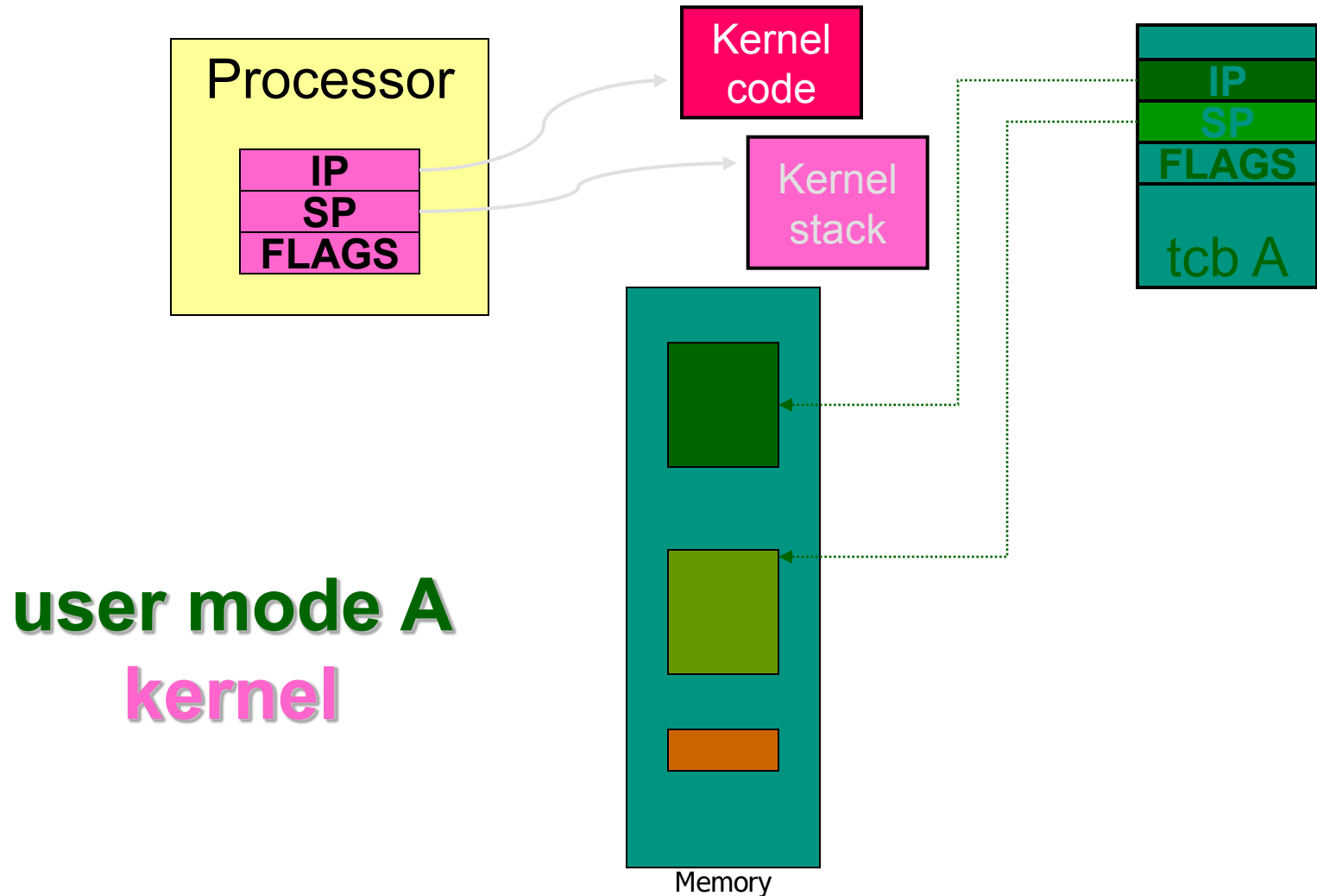


Thread Switch **A** → **kernel** → **B**

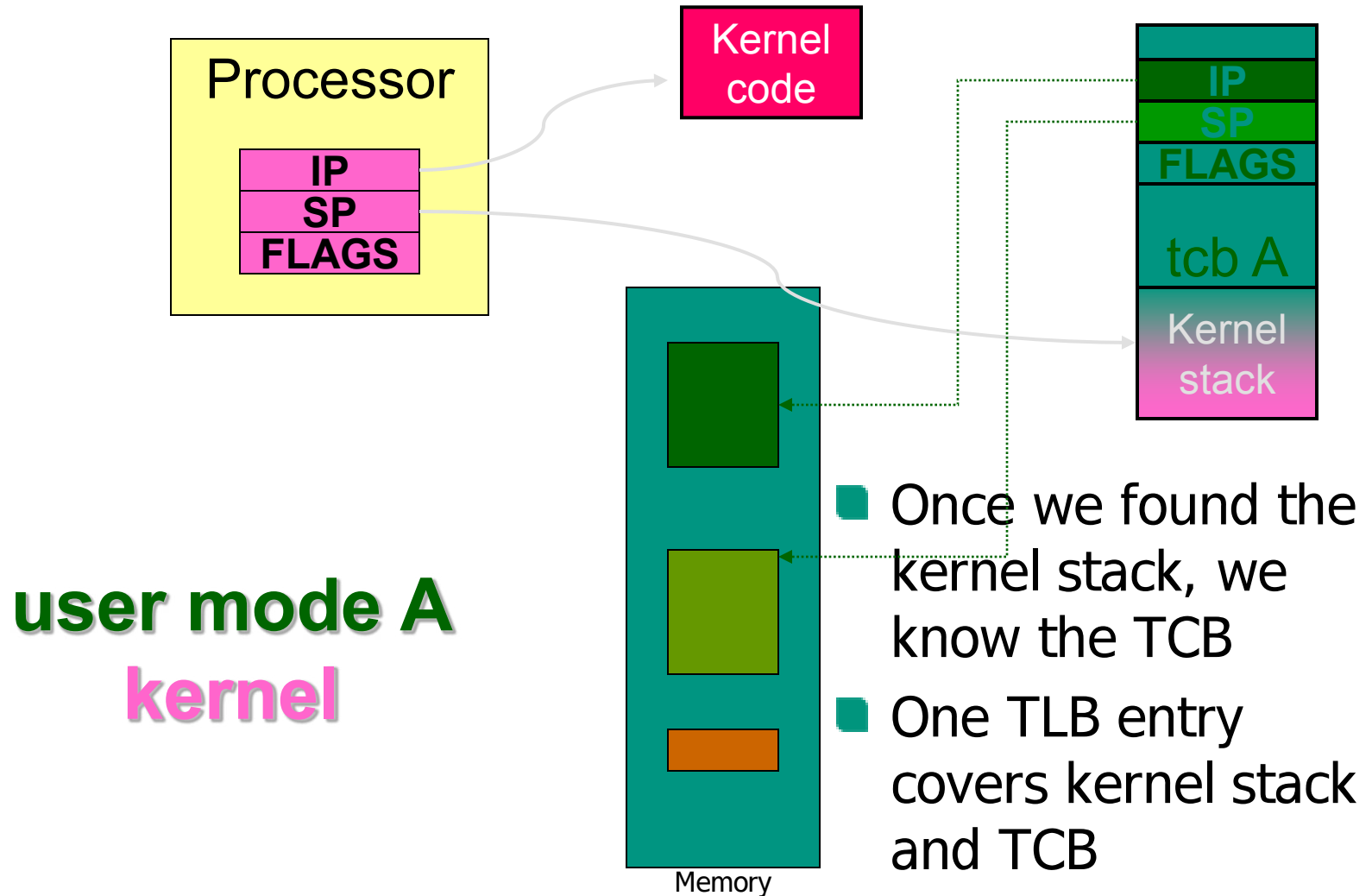




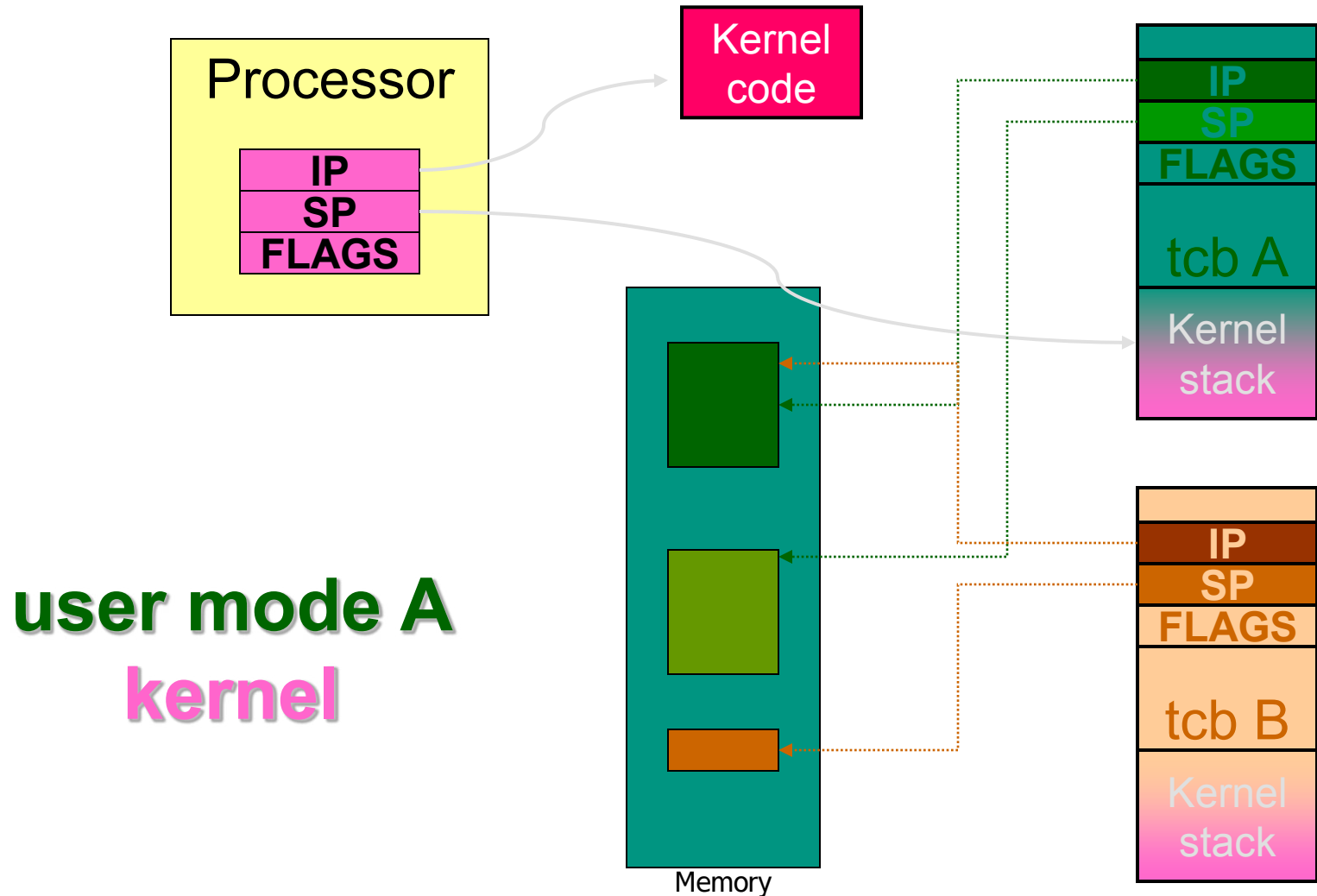
Thread Switch **A** → **kernel** → **B**



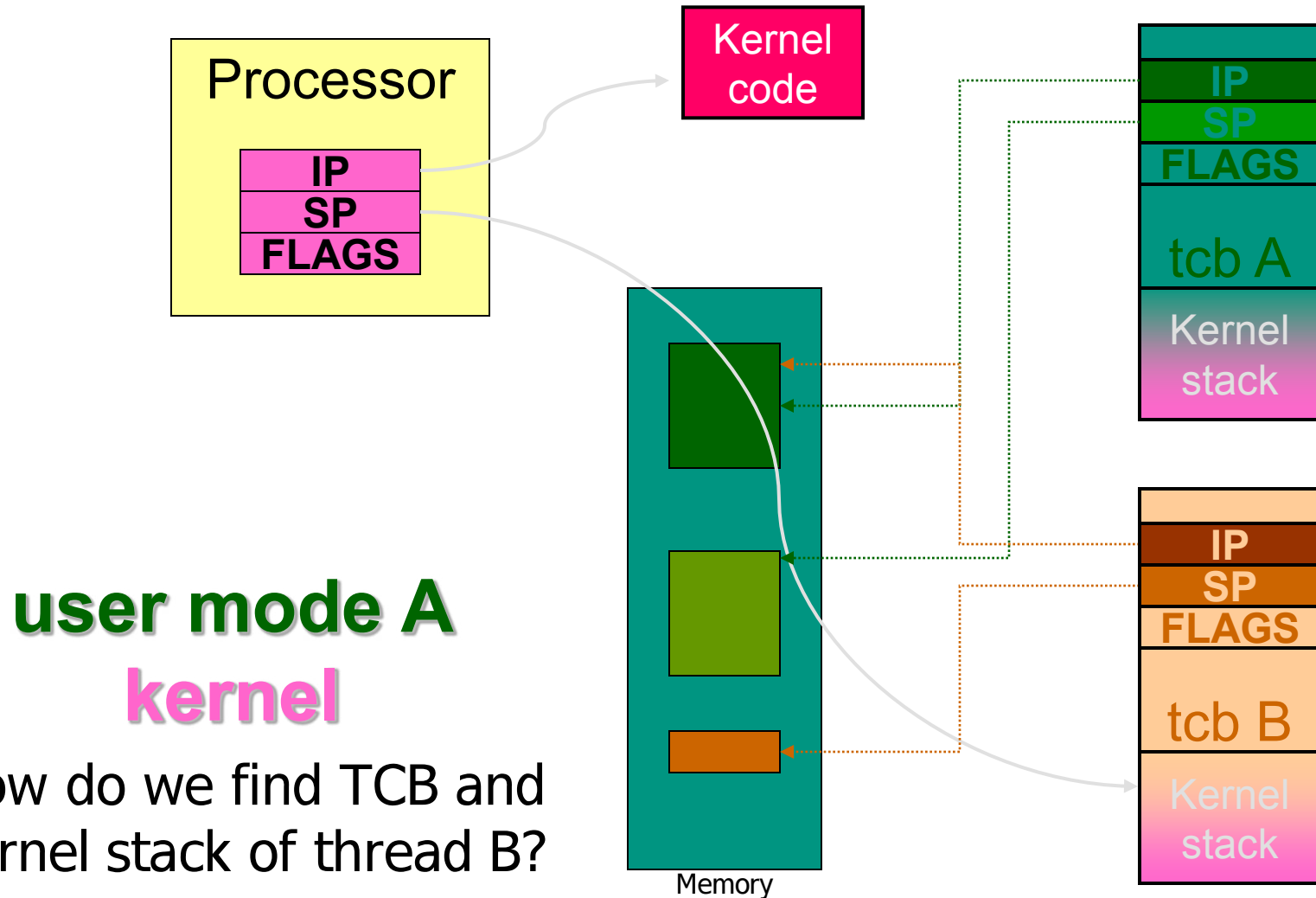
Thread Switch **A** → **kernel** → **B**



Thread Switch **A** → **kernel** → **B**



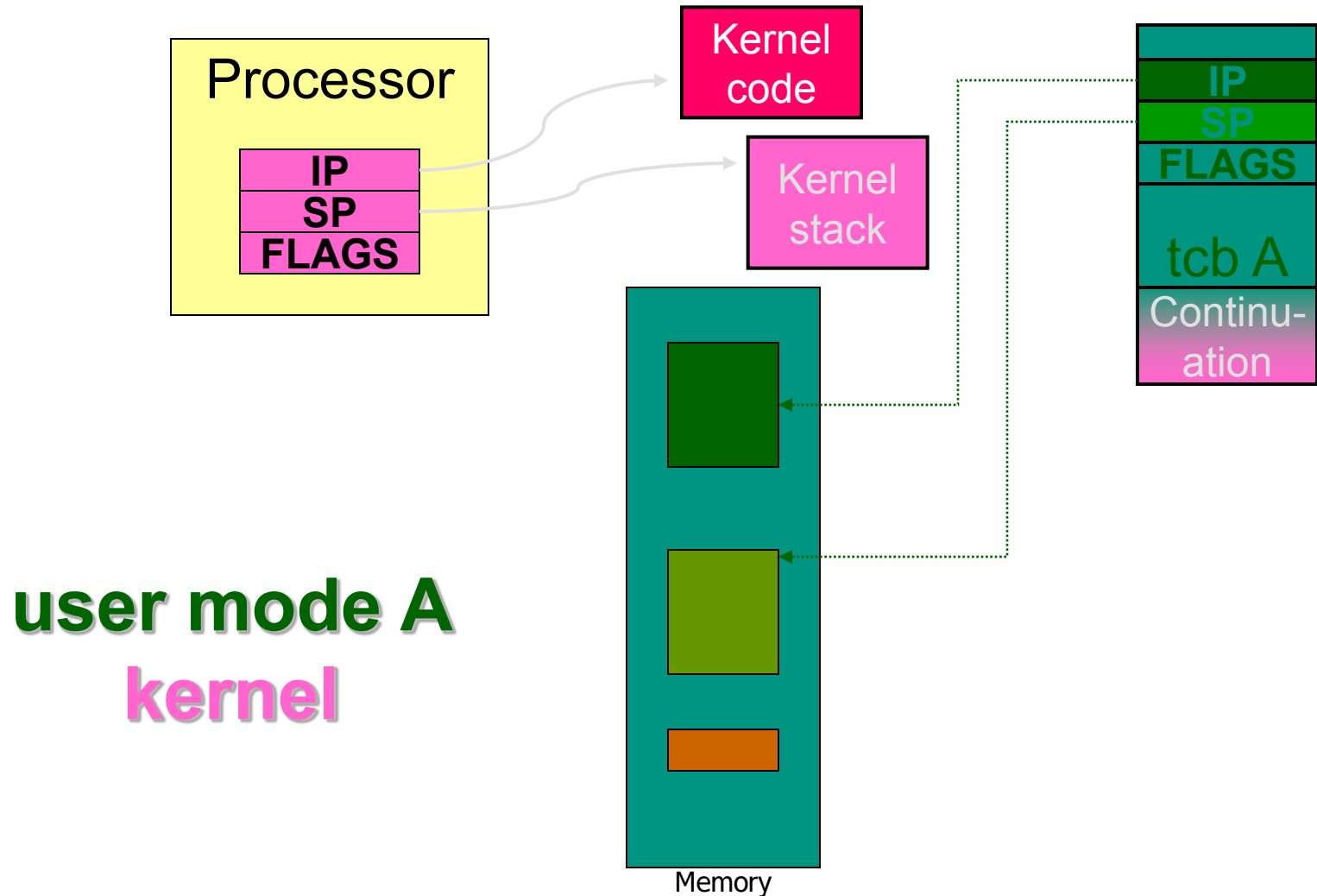
Thread Switch **A** → **kernel** → **B**



user mode A
kernel

- How do we find TCB and kernel stack of thread B?

Thread Switch with single kernel stack



Construction Conclusion

From the view of the designer there are two alternatives:

Single Kernel Stack

- Only one stack is used in kernel mode all the time
- seL4, OKL4

Per-Thread Kernel Stack

- Each thread has its own stack in kernel mode
- Pistachio, Fiasco.OC

Single Kernel Stack

Per processor, event model

■ Either **continuations**

- Complex to program

■ Or **stateless kernel**

- No kernel threads, kernel not interruptible, difficult to program
- Structurally inefficient system calls
- + Kernel can be exchanged on-the-fly
 - E.g. the fluke kernel from Utah

■ Low cache and TLB footprint

- The same stack is always used!

■ Stack can be larger

- Easier to use recursion in the kernel!

■ Easier to prove

Multiple Kernel Stacks

Per thread, activity model

- Kernel can always use threads, no special methods required for keeping state while interrupted/blocked
- No conceptual difference between kernel mode and user mode

Conclusion:

We have to look for a solution that minimizes the kernel stack size

- Larger cache and TLB footprint
- Limited kernel stack size

Conclusion:

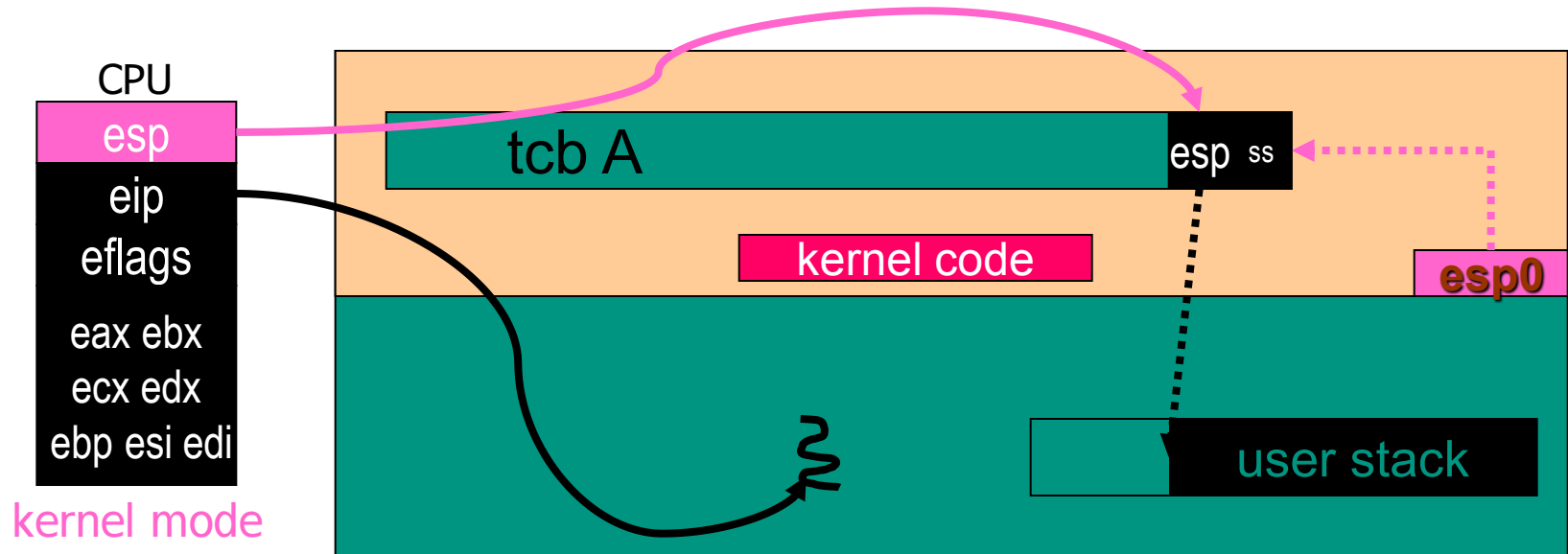
We have to avoid recursion in the kernel
(→ Mapping)

KERNEL ENTRY AND EXIT ON IA-32



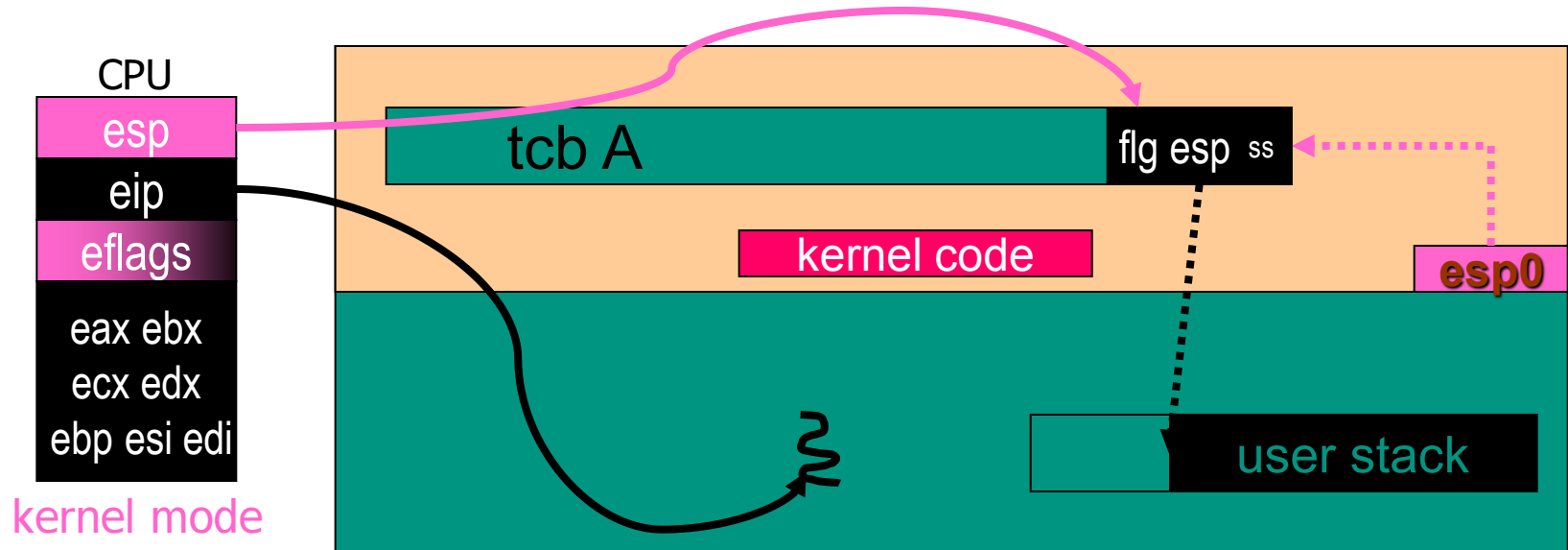
24 03.05.2017

Kernel Entry (IA-32)



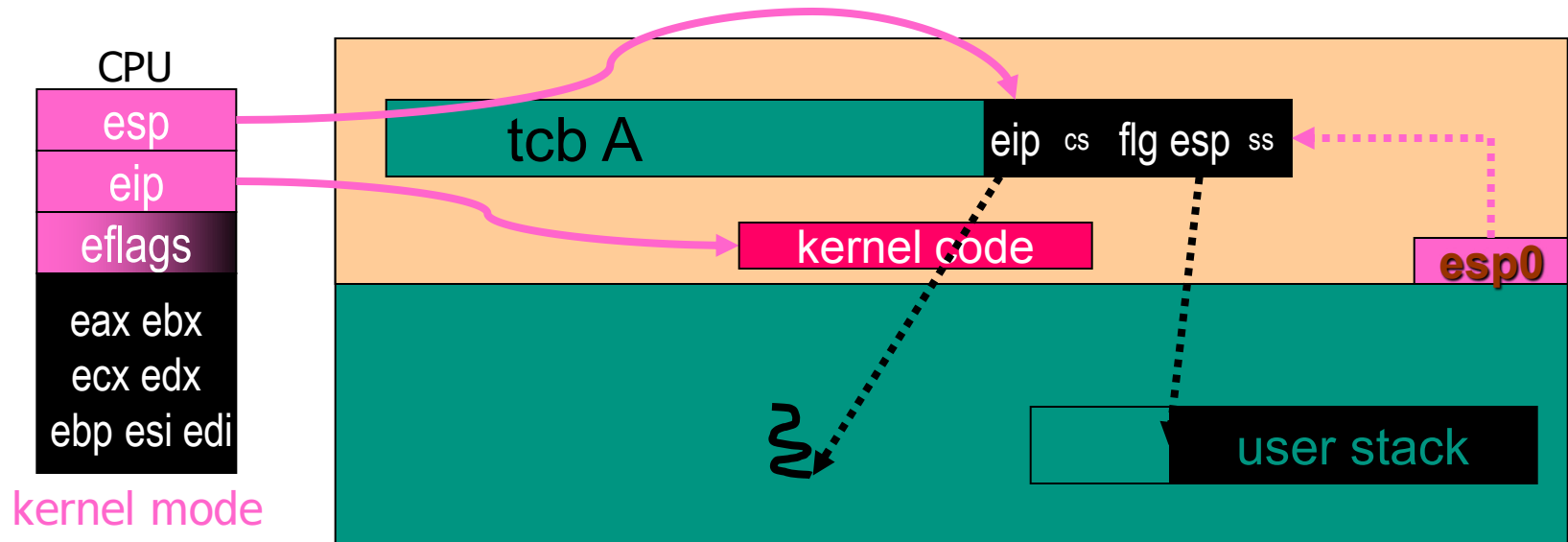
- Trap/fault occurs (int *n* / exception / interrupt)
 - Push user SS:ESP onto kernel stack, load kernel SS:ESP

Kernel Entry (IA-32)



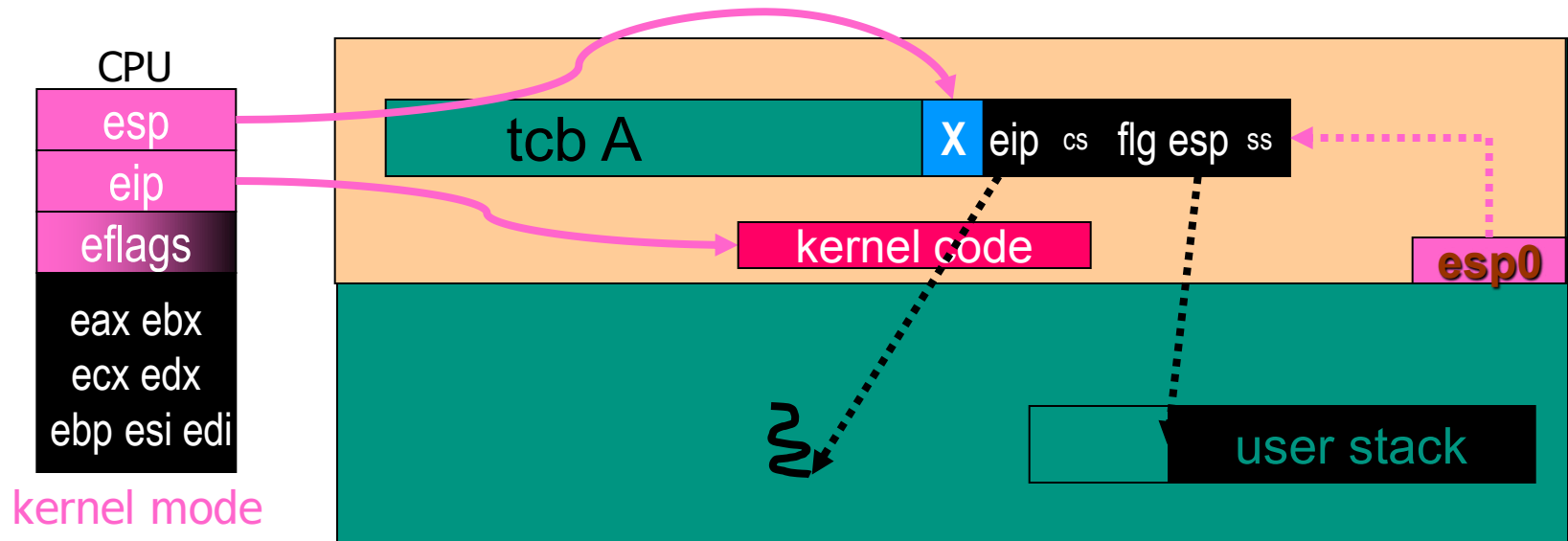
- Trap/fault occurs (int *n* / exception / interrupt)
 - Push user SS:ESP onto kernel stack, load kernel SS:ESP
 - Push user EFLAGS, reset flags (*I* := 0, *CPL* := 0)

Kernel Entry (IA-32)



- Trap/fault occurs (int *n* / exception / interrupt)
 - Push user SS:ESP onto kernel stack, load kernel SS:ESP
 - Push user EFLAGS, reset flags (`I := 0`, `CPL := 0`)
 - Push user CS:EIP, load kernel entry CS:EIP

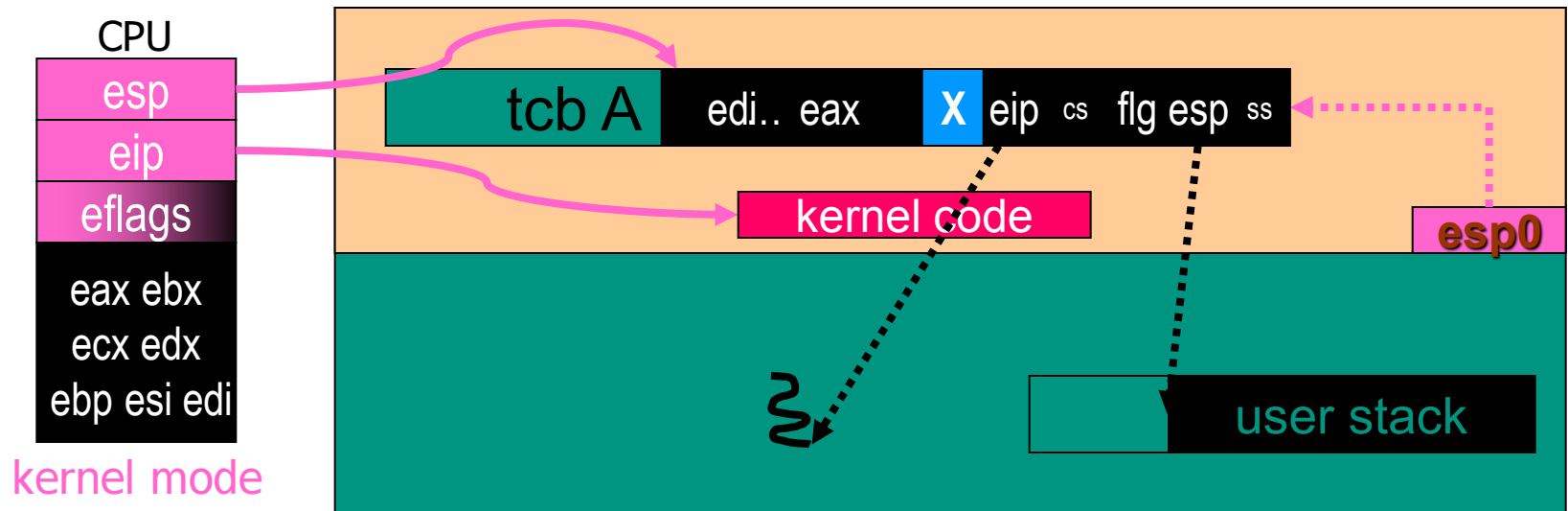
Kernel Entry (IA-32)



- Trap/fault occurs (int *n* / exception / interrupt)
 - Push user SS:ESP onto kernel stack, load kernel SS:ESP
 - Push user EFLAGS, reset flags (`I := 0`, `CPL := 0`)
 - Push user CS:EIP, load kernel entry CS:EIP
- Push X: error code (hw, at exception) or kernel-call type

*hardware
programmed,
single
"instruction"*

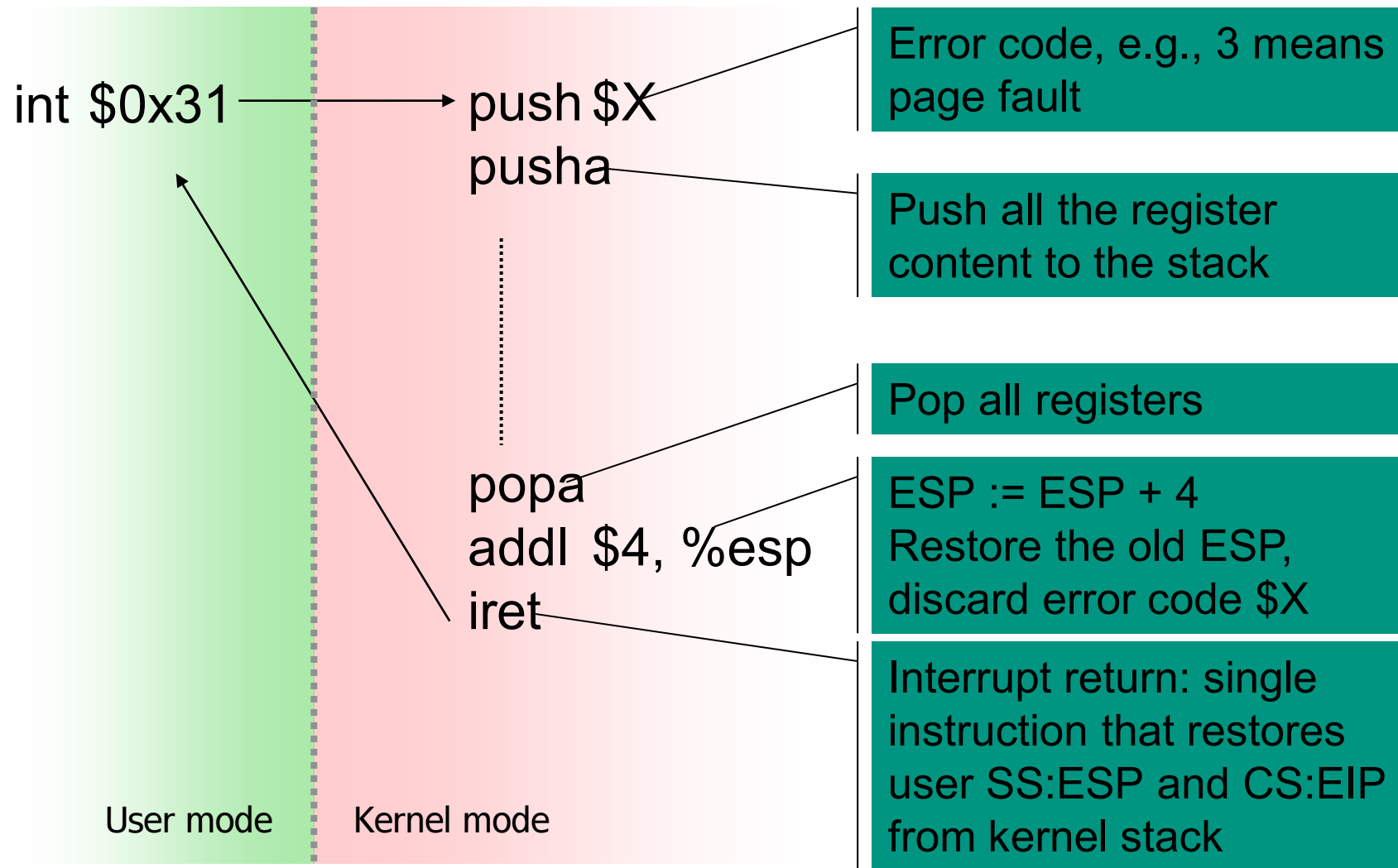
Kernel Entry (IA-32)



- Trap/fault occurs (int n / exception / interrupt)
 - Push user SS:ESP onto kernel stack, load kernel SS:ESP
 - Push user EFLAGS, reset flags ($I := 0$, $CPL := 0$)
 - Push user CS:EIP, load kernel entry CS:EIP
- Push X: error code (hw, at exception) or kernel-call type
- Push registers (optional)

*hardware
programmed,
single
“instruction”*

System Call (IA-32)



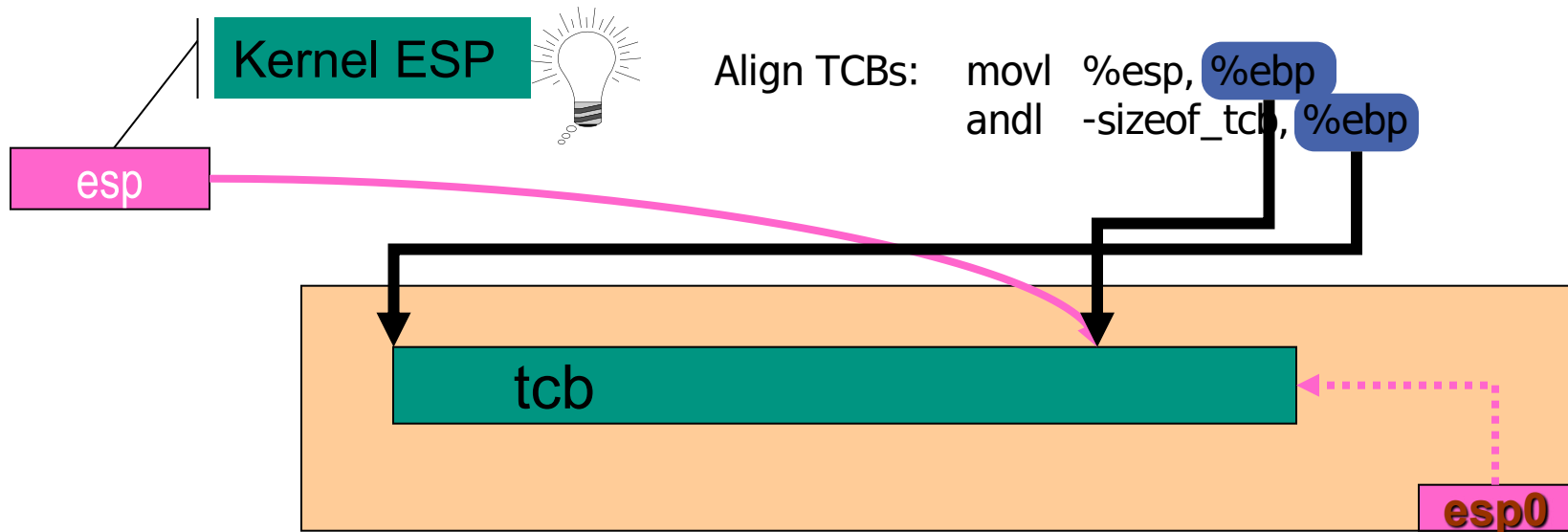
THREAD SWITCH ON IA-32

Locating the TCB

Remember: We need to find

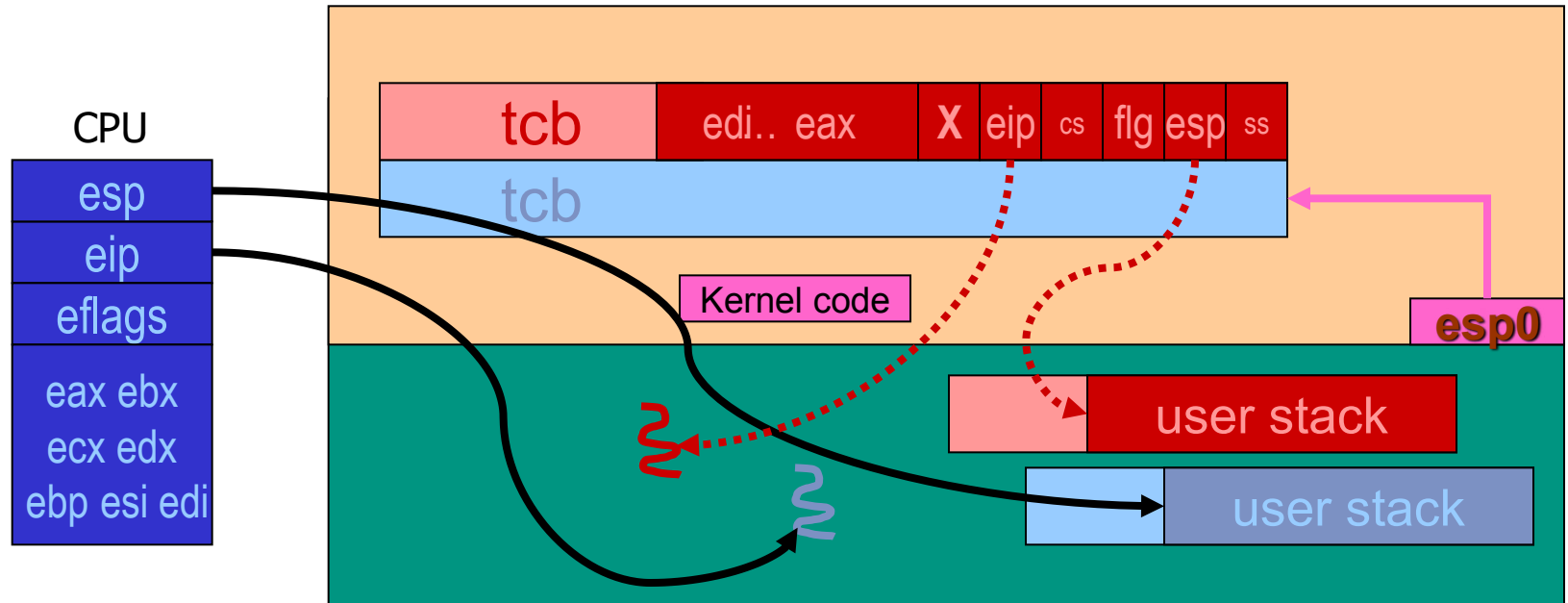
- Anythread's TCB using its global ID
- The currently executing thread's TCB

Next
lecture



esp value		0000	0000	1011	1011	1011	1011	1011	1011		0x00BB_BB		
sizeof_tcb		0000	0000	0000	0000	0001	0000	0000	0000		0x0000_1000		4096
~sizeof_tcb		1111	1111	1111	1111	1110	1111	1111	1111		0xFFFF_EFFF		-4097
-sizeof_tcb		1111	1111	1111	1111	1111	0000	0000	0000		0xFFFF_F000		-4096
esp & -sizeof_tcb		0000	0000	1011	1011	1011	0000	0000	0000		0x00BB_B000		

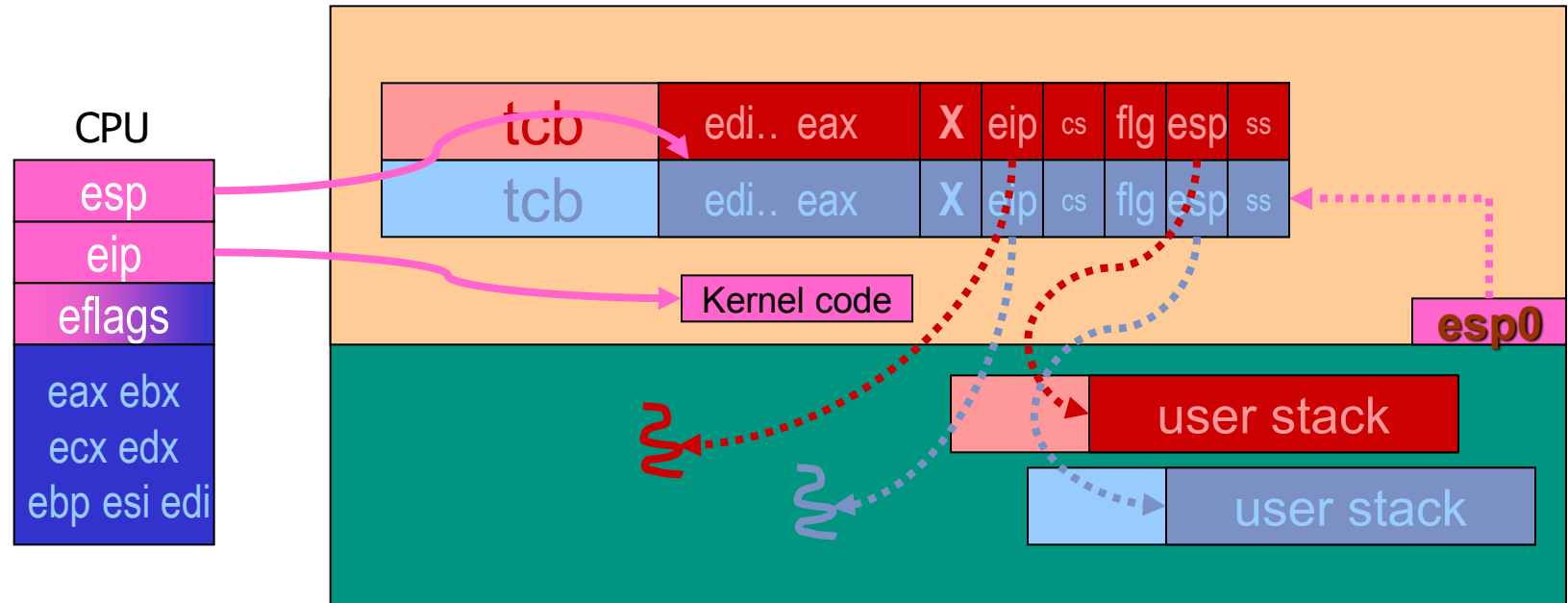
Switching Threads (IA-32, per-thread stack)



■ `int $0x0`

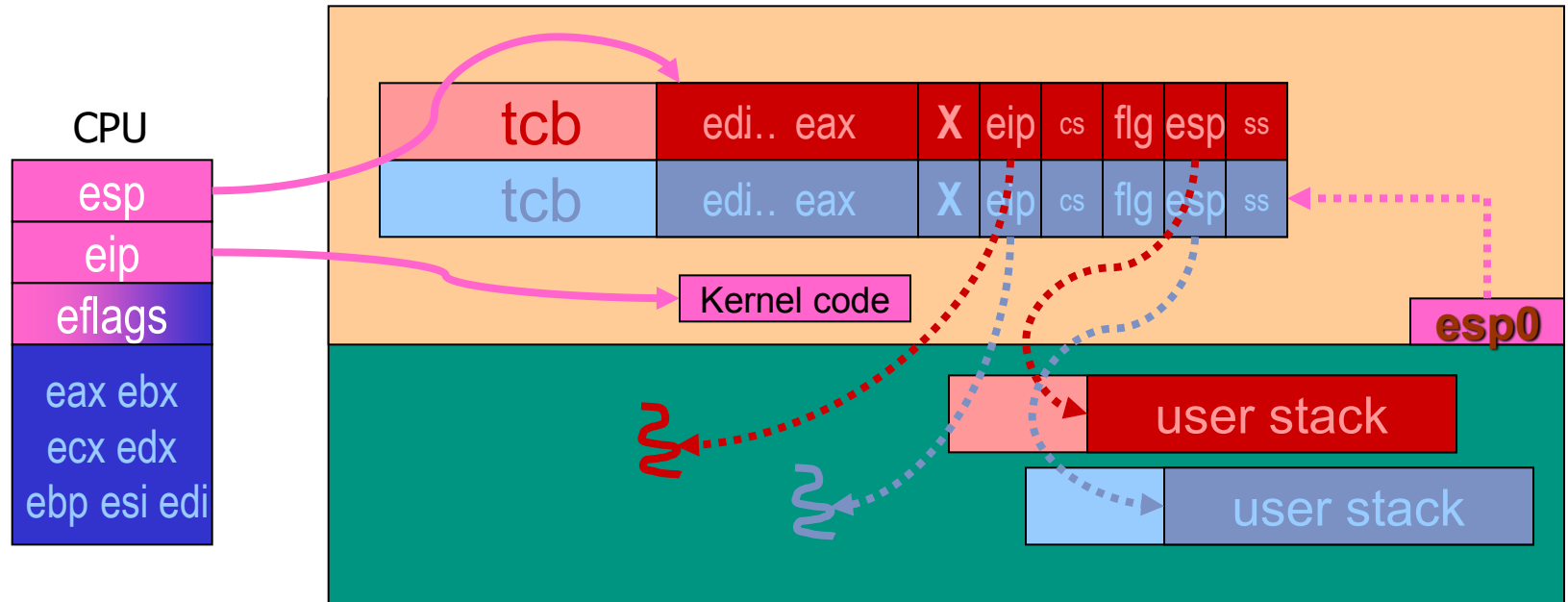


Switching Threads (IA-32, per-thread stack)



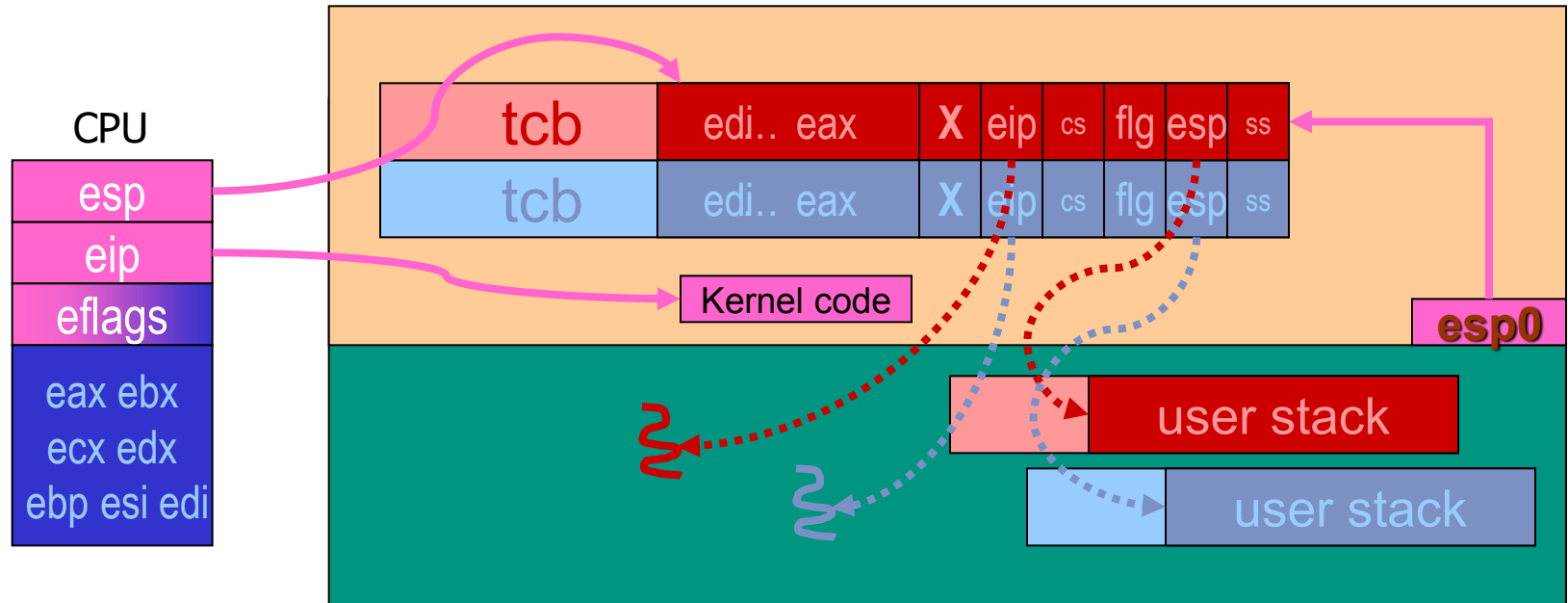
■ `int $0x0, push registers of blue thread`

Switching Threads (IA-32, per-thread stack)

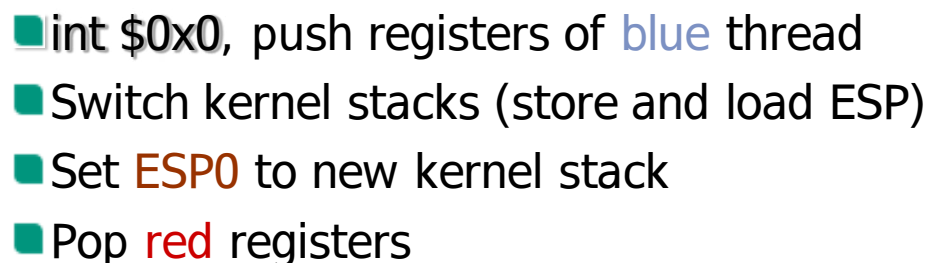


- `int $0x0`, push registers of blue thread
- Switch kernel stacks (store and load ESP)

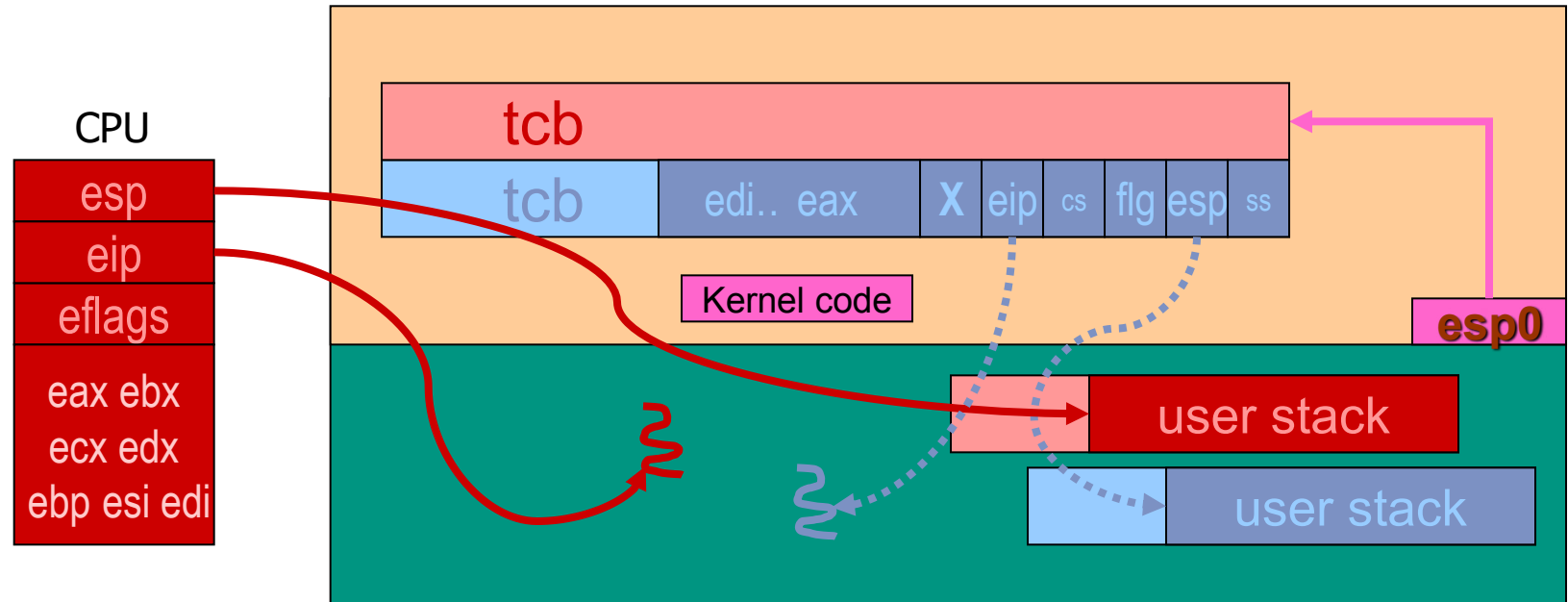
Switching Threads (IA-32, per-thread stack)



- `int $0x0`, push registers of blue thread
- Switch kernel stacks (store and load `ESP`)
- Set `ESP0` to new kernel stack



Switching Threads (IA-32, per-thread stack)



- `int $0x0`, push registers of blue thread
- Switch kernel stacks (store and load ESP)
- Set **ESP0** to new kernel stack
- Pop red registers, return to red user thread (`iret`)

Thread Switch (IA-32, per-thread stack)

Thread A

⋮

int \$0x0

```
pusha
movl  %esp, %ebp
andl  -sizeof_tcb, %ebp
/* edi == address of B's T C B */
```

```
movl  %esp, OFF_ESP(%ebp)
movl  OFF_ESP(%edi), %esp
```

```
addl  sizeof_tcb, %edi
movl  %edi, %esp0
```

```
popa
addl  $4, %esp
iret
```

Switch current kernel
stack pointer

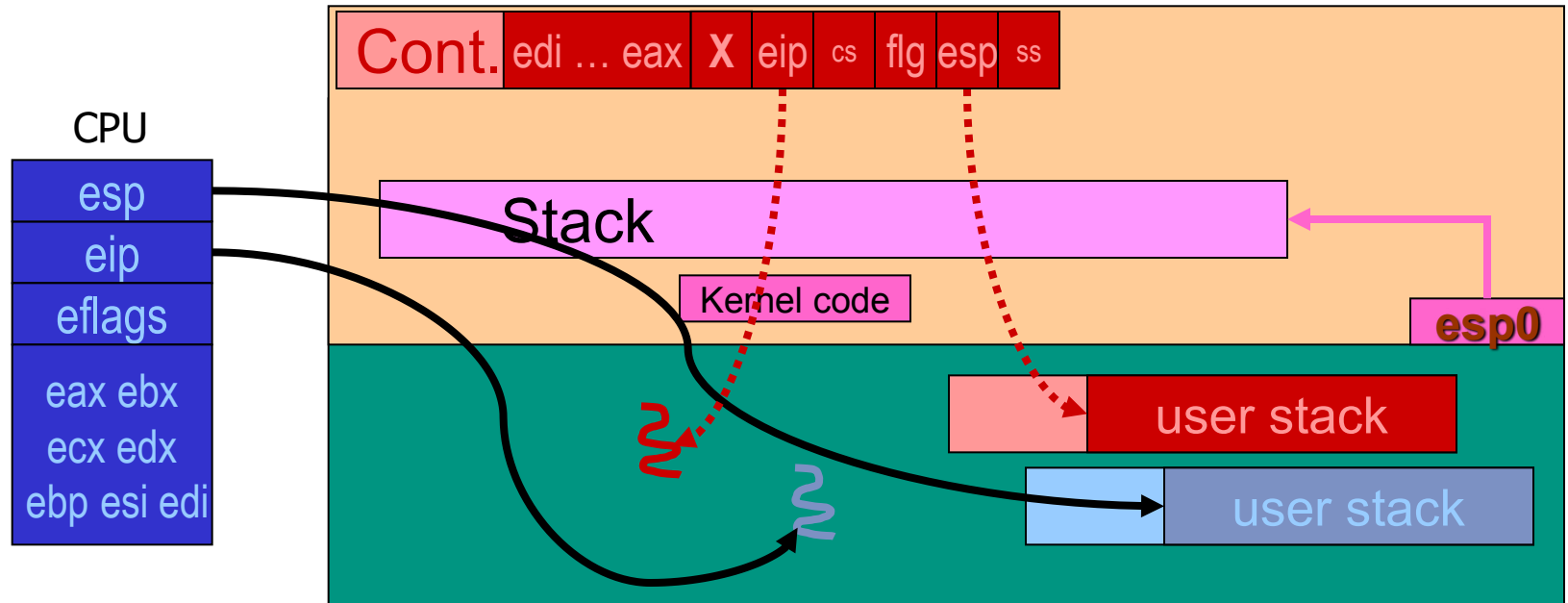
Switch ESP0
so that next
kernel *entry*
uses new
kernel stack

Thread B

⋮

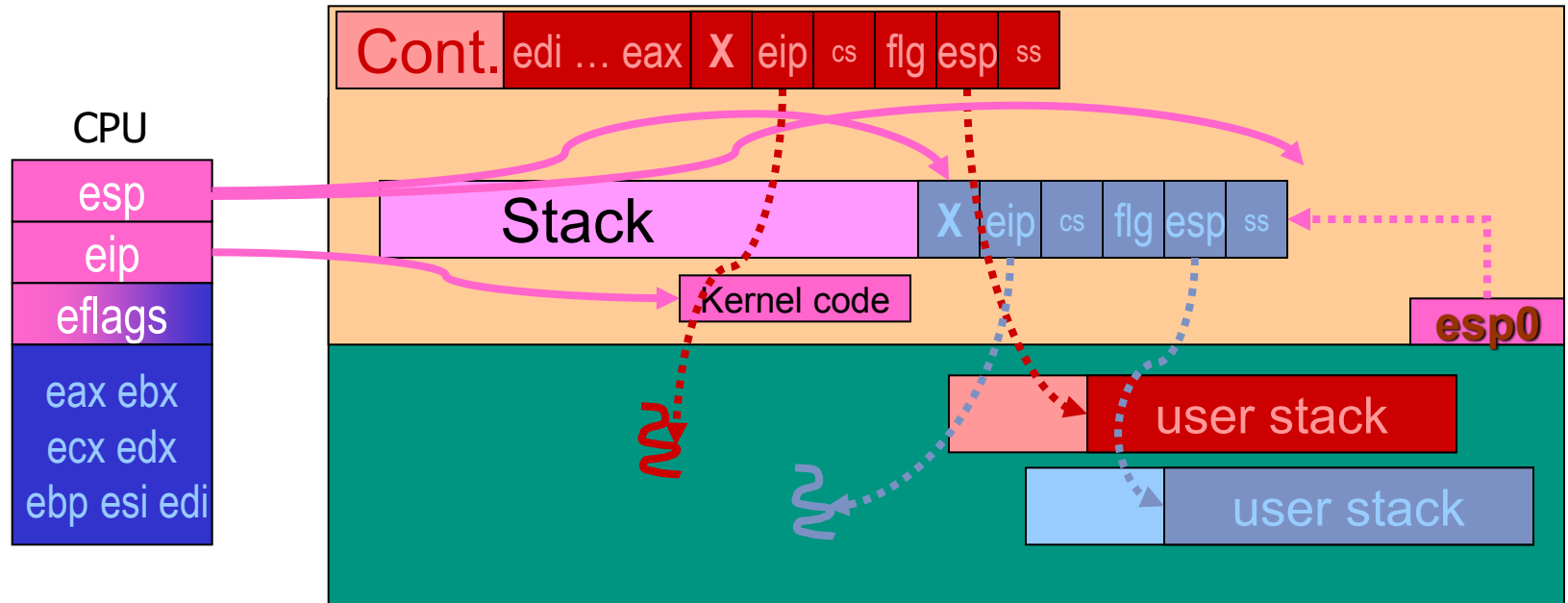
int \$0x0

Switching Threads (IA-32, single stack)



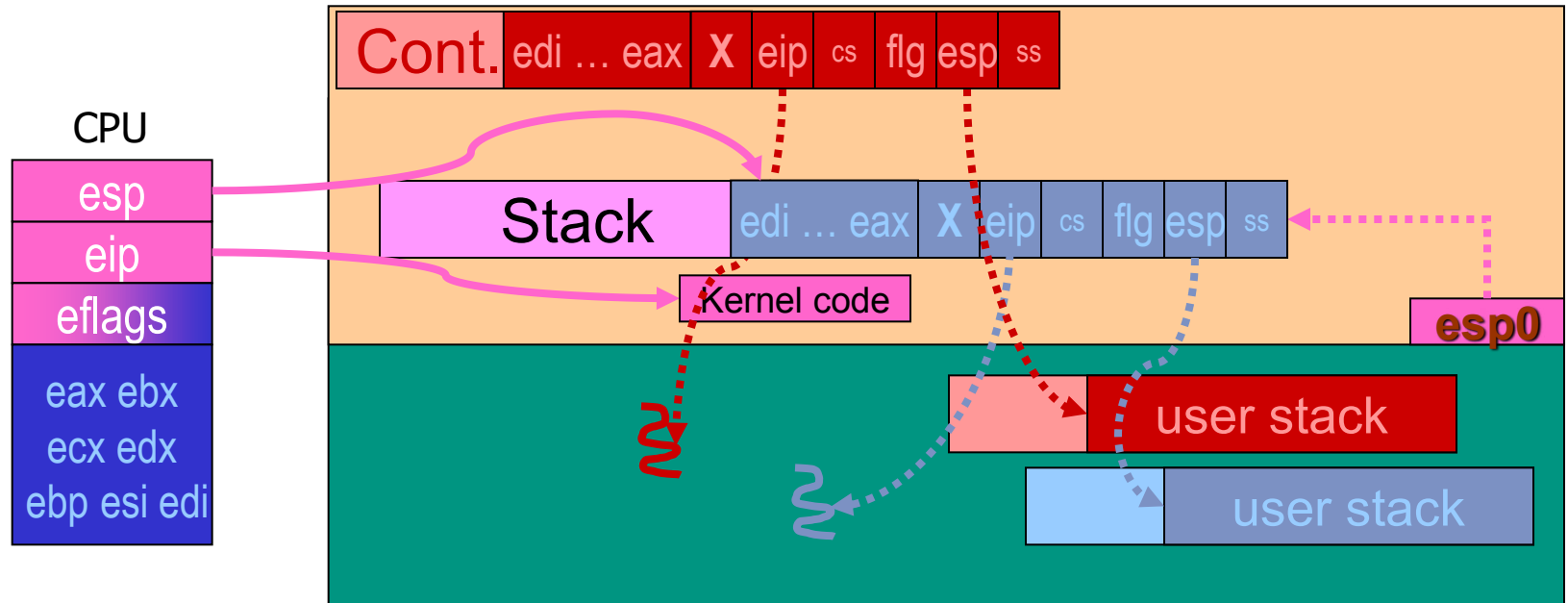
■ `int $0x0`

Switching Threads (IA-32 , single stack)



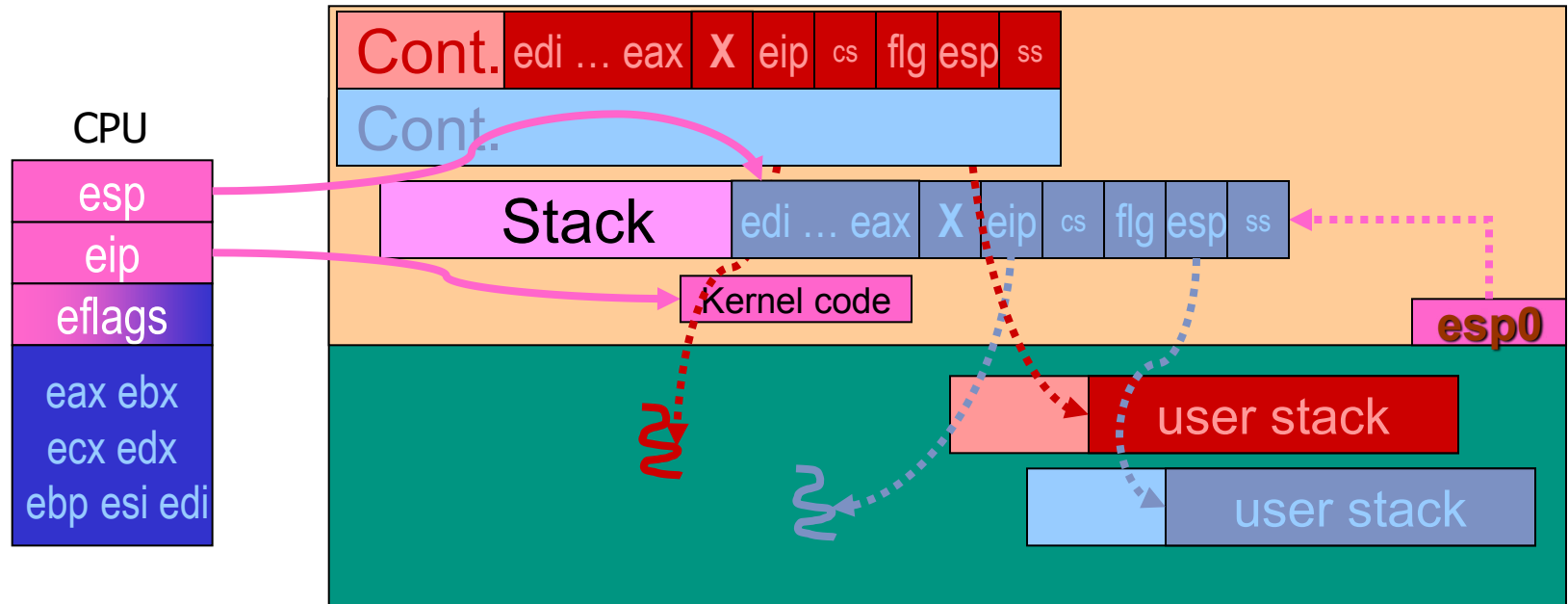
int \$0x0

Switching Threads (IA-32 , single stack)



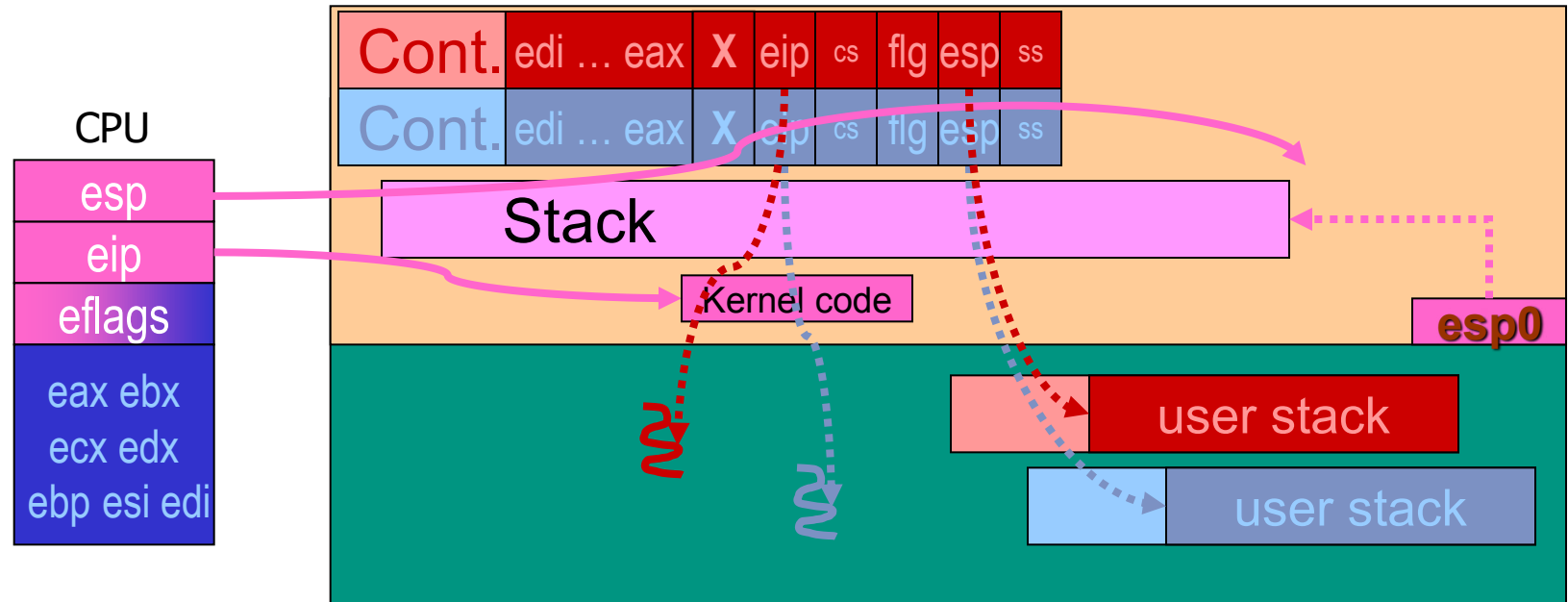
■ `int $0x0, push registers of blue thread`

Switching Threads (IA-32 , single stack)



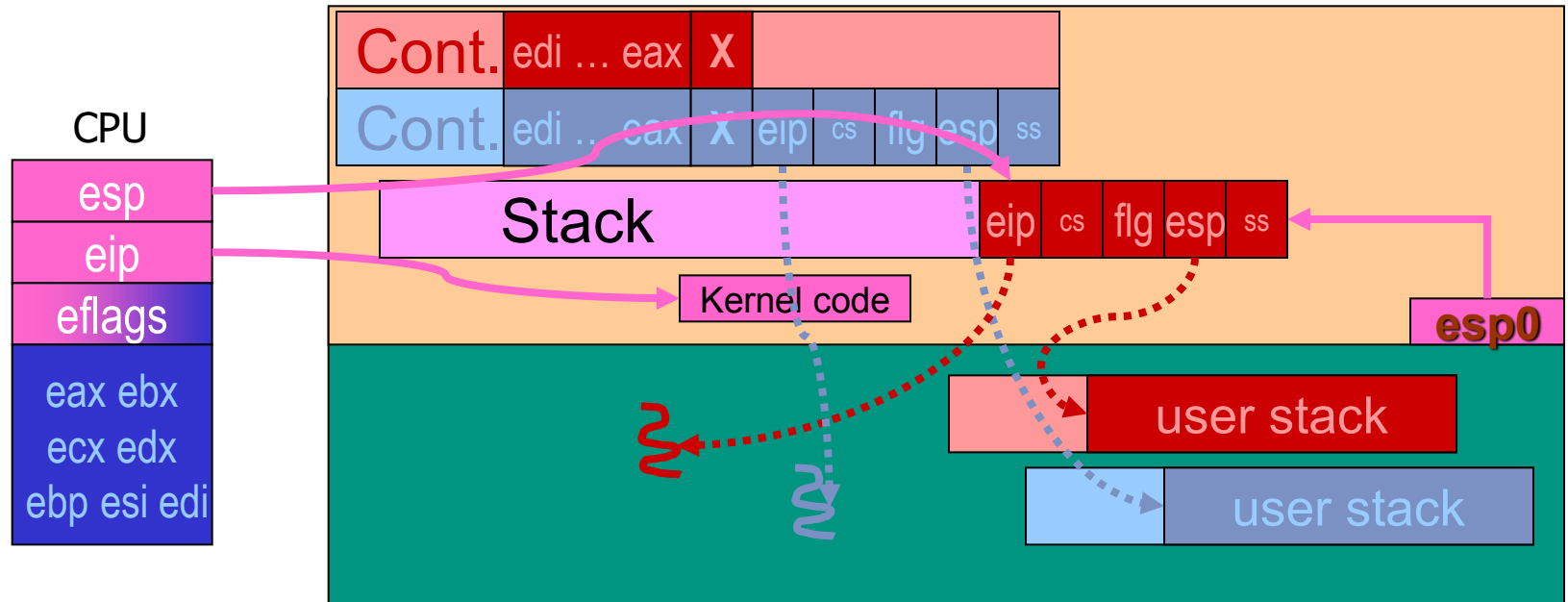
- `int $0x0`, push registers of blue thread
- Find blue continuation

Switching Threads (IA-32 , single stack)



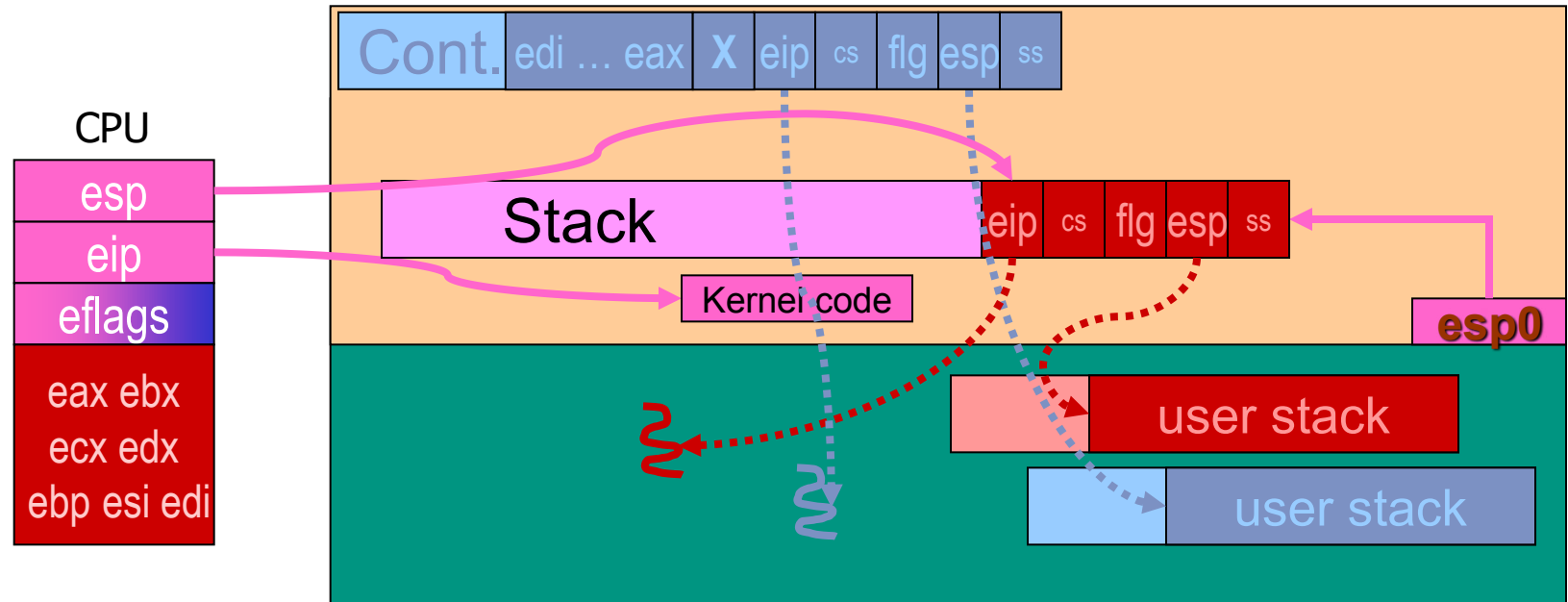
- `int $0x0`, push registers of blue thread
- Find blue continuation
- Move registers of blue thread to continuation

Switching Threads (IA-32 , single stack)



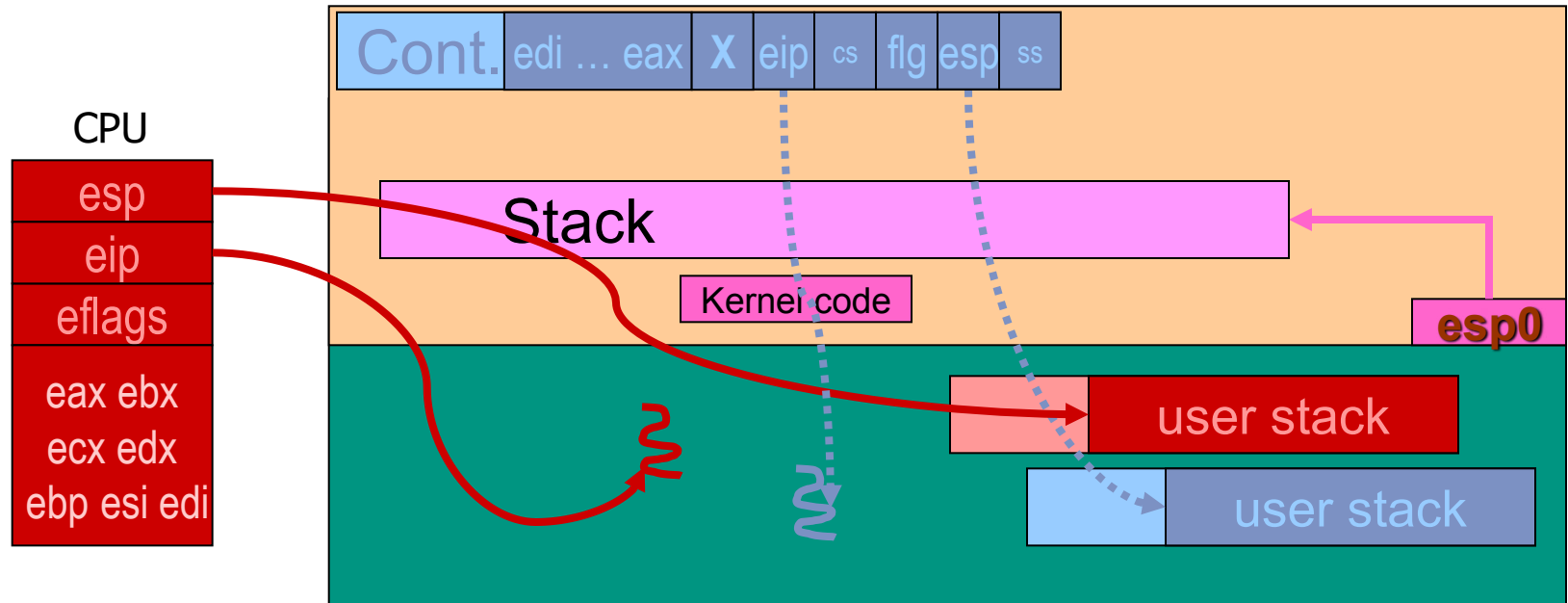
- `int $0x0`, push registers of blue thread
- Move registers of blue thread to continuation
- Restore red IP/SP/Flags from continuation

Switching Threads (IA-32 , single stack)



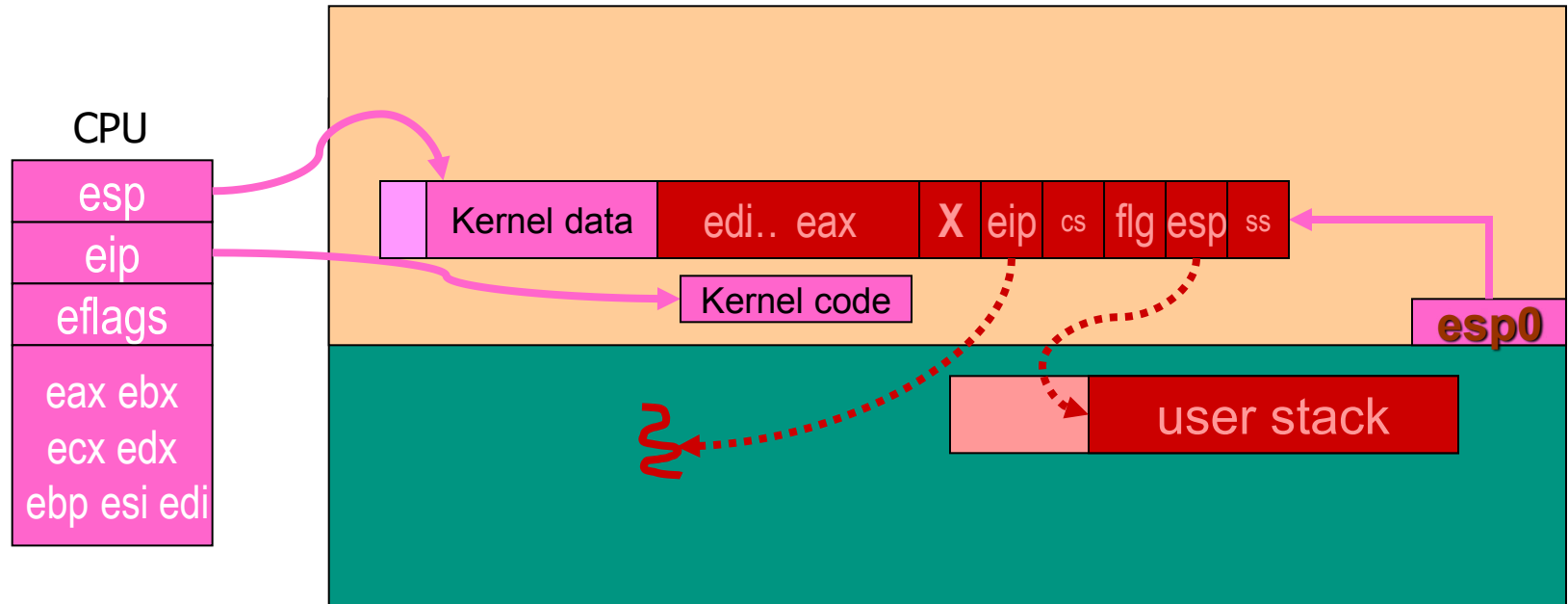
- `int $0x0`, push registers of blue thread
- Move registers of blue thread to continuation
- Restore red IP/SP/Flags from continuation
- Restore red registers

Switching Threads (IA-32 , single stack)



- `int $0x0`, push registers of blue thread
- Move registers of blue thread to continuation
- Restore red IP/SP/Flags from continuation
- Restore red registers, return to red user thread (`iret`)

Kernel preemption with single stack



- Where to save kernel state (stack + regs)?
 - Kernel stack? Stack size unbounded with nested interrupts
 - Continuation? Might as well have per-thread stacks
 - User-mode stack? What could possibly go wrong...?

What about other registers?

- So far, we have only considered general purpose registers
- What about FPU, SSE?
- Extremely expensive
 - IA-32 full SSE2 state is 512 Bytes
 - IA-64 floating point state is ~ 1.5 KiB
- Saving/restoring must be extremely efficient
- Need a place to store FP state
 - WAY too large for TCB/Kernel stack

Hardware to the rescue

- x86 has HW support for saving/restoring FP state
- `xsave addr` → Store FP state to `addr`
- `xrstor addr` → Restore FP state from `addr`
 - `Addr` is called the Xsave area
- EDX:EAX contain contain info what to save (bitmap)

Reserved				P K R	P T	AVX- 512	MPX	A V X	S S E	x 8 7
----------	--	--	--	-------------	--------	-------------	-----	-------------	-------------	-------------

- Bitmap stored in xsave area
- xsave area can be anywhere
 - Pointer kept in TCB

Xsave optimizations

- xsave/xrstor provide highly efficient save/restore
- But: Still a lot of data to copy

- Init optimization:
 - If FU is in initial state (=unused), do not save
- Modified optimization:
 - If register was not modified since last xrstor, do not save

Summary

■ TCBs

- Implement threads
- Must store thread state while preempted

■ Kernel stacks

- Either per thread (large TLB footprint, no recursion)
- Or per core (need continuations, no kernel preemption)

■ Thread switch

- Switch kernel stack
- Or switch state on kernel stack