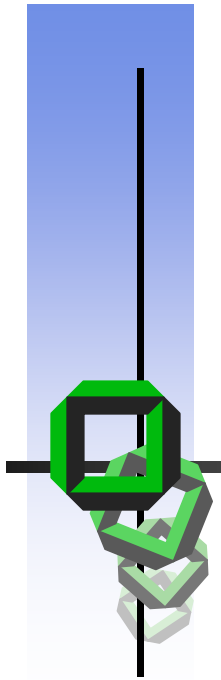
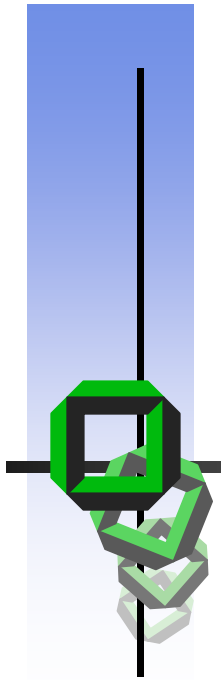


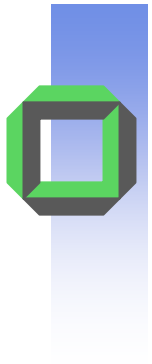
µ-Kernel Construction (2)

Threads, System Calls, and Thread Switching

(updated on 2009-05-08)



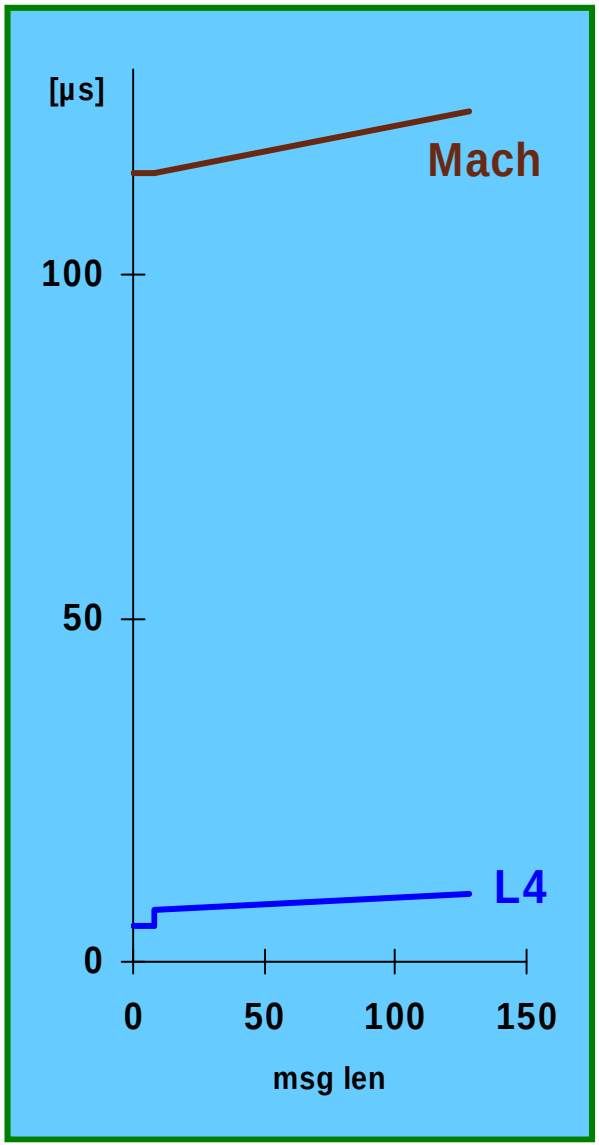
Review from Last Lecture

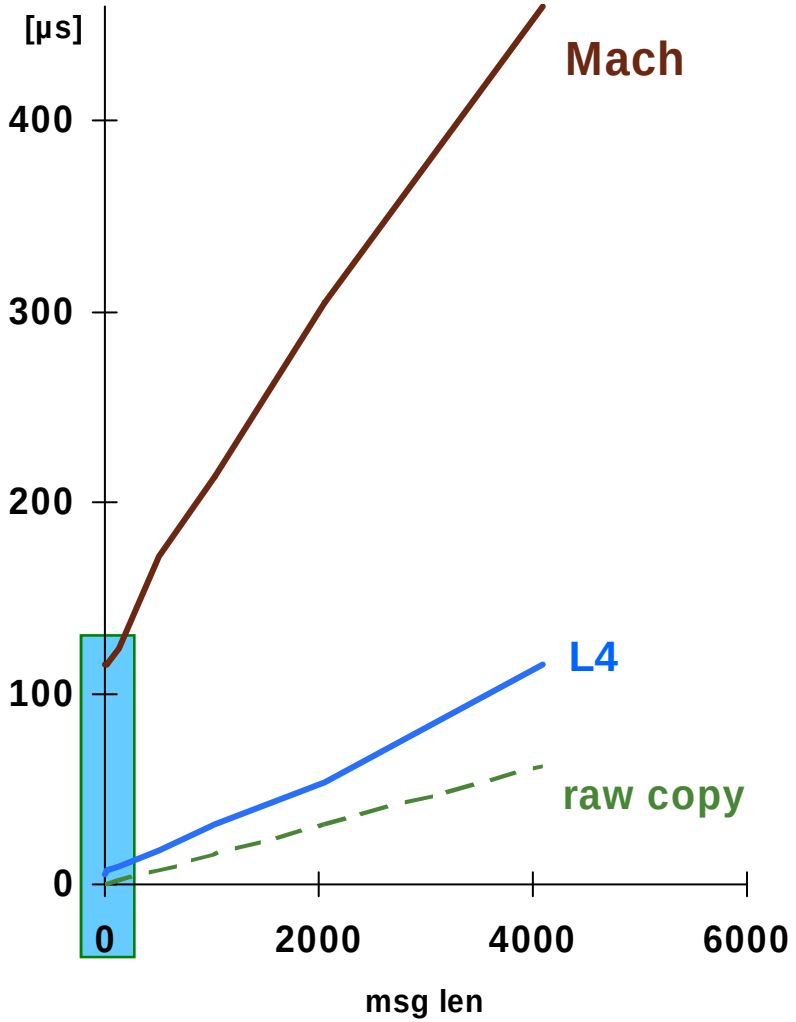


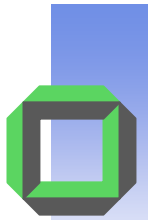
The 100-µs Disaster

25 MHz 386 → **50 MHz 486** → **90 MHz Pentium** → **133 MHz Alpha**

IPC Costs (486, 50 MHz)



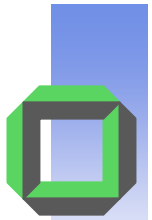




Analysis

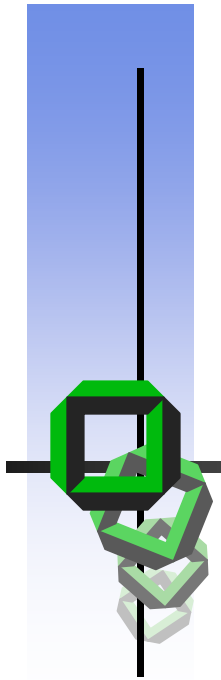
- Mach uses asynchronous IPC
  - **Data buffered in kernel**
  - Cache destruction, infinite resources
  - Memory bus traffic
- Workplace OS switched to synchronous IPC
  - Performance still horrible
  - Why?

Microkernel construction is difficult!

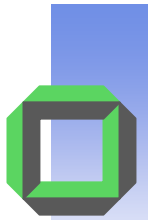


Size Comparison

- Mach 4 x86: 90,000
- Xen 2.0 hypervisor: 45,000
- L4Ka::Pistachio x86: 45,000



Threads and How to Switch Them

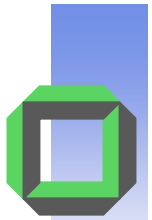


Fundamental Abstractions

- 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?*

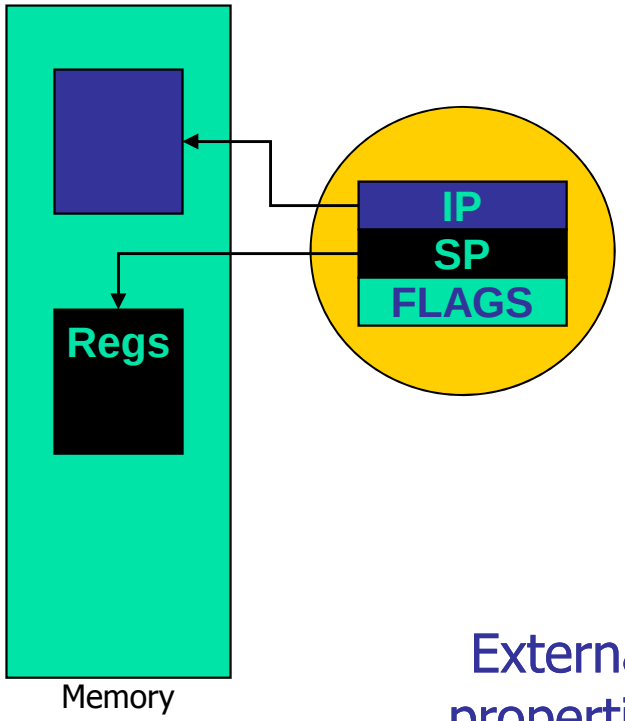


Fundamental Abstractions

A thread is an independent flow of control inside an address space. Threads are identified by unique identifiers and communicate via IPC. Threads are characterized by a set of registers, including at least an instruction pointer, a stack pointer and state information. A thread's state also includes the address space in which the thread currently executes.

Thread Properties

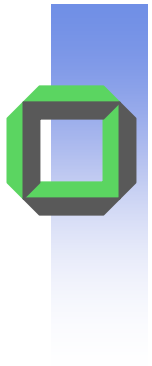
Internal properties



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

External properties

- Address space
- Unique ID
- Communication status

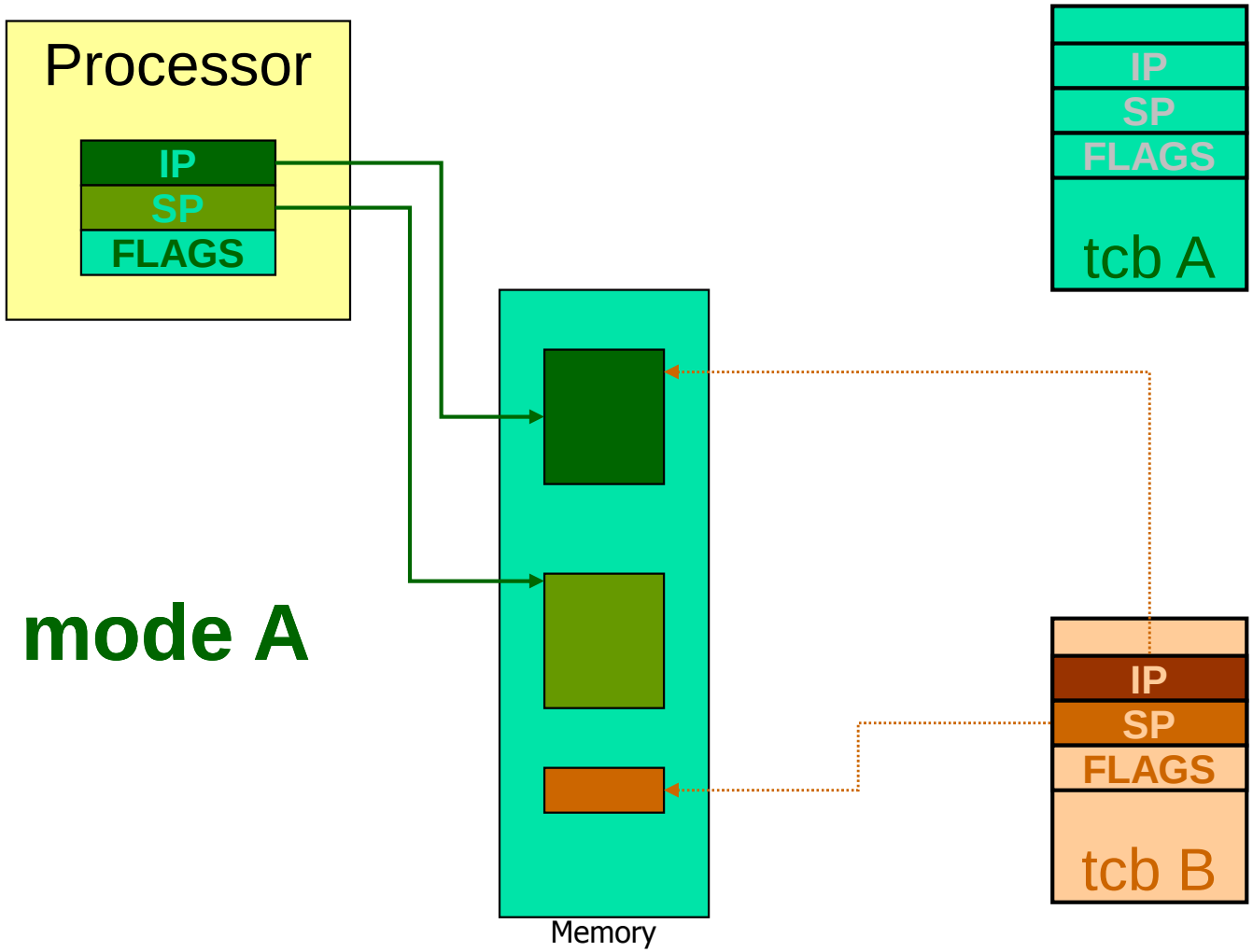


Construction Conclusion

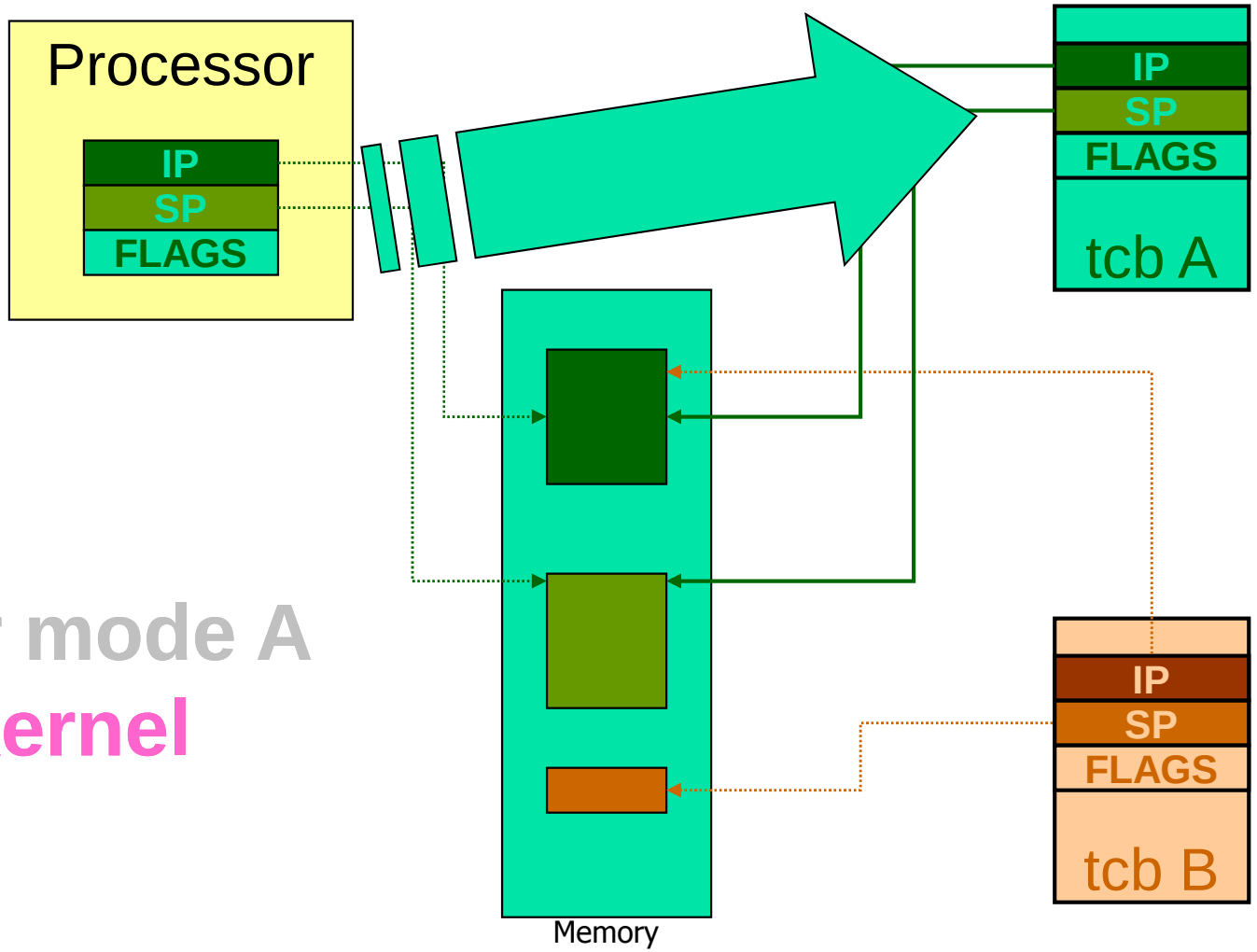
- Thread state must be saved/restored on thread switch
- We need a Thread Control Block (TCB) per thread
- TCBs must be kernel objects
  - TCBs implement threads
- We often need to find
  - Any thread's TCB using its global ID
  - The currently executing thread's TCB (per processor)

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

Thread Switch A → B



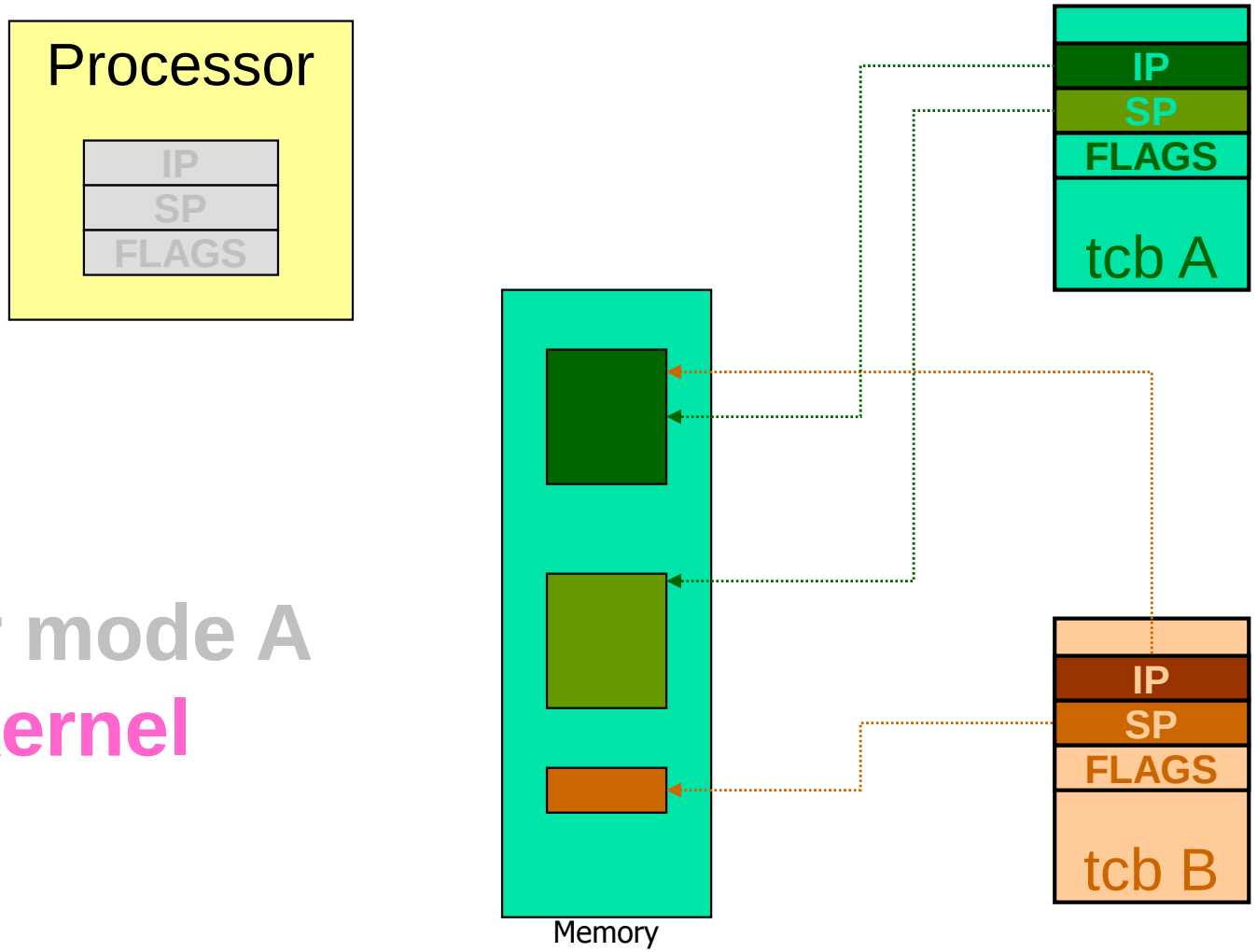
Thread Switch A → B



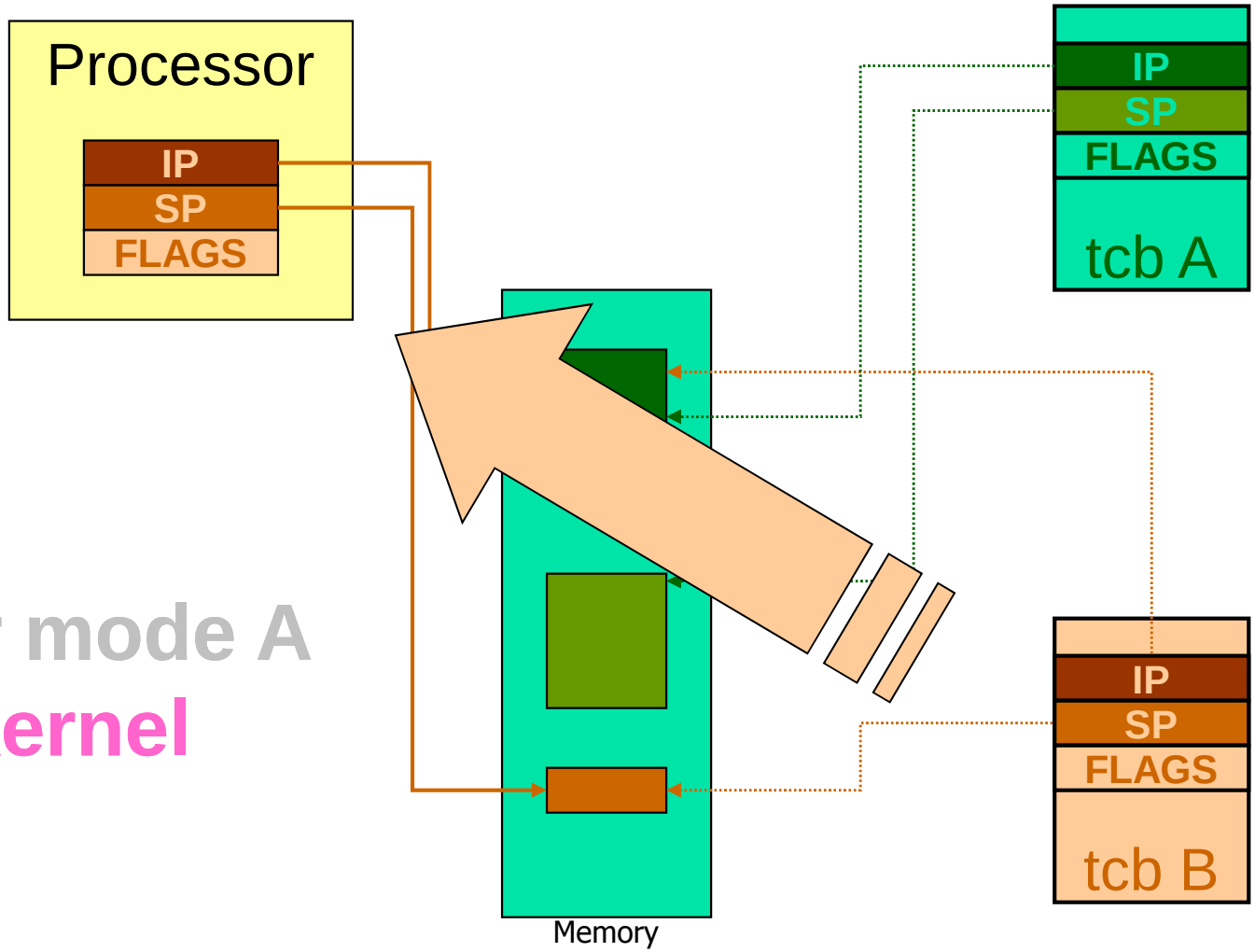
user mode A

kernel

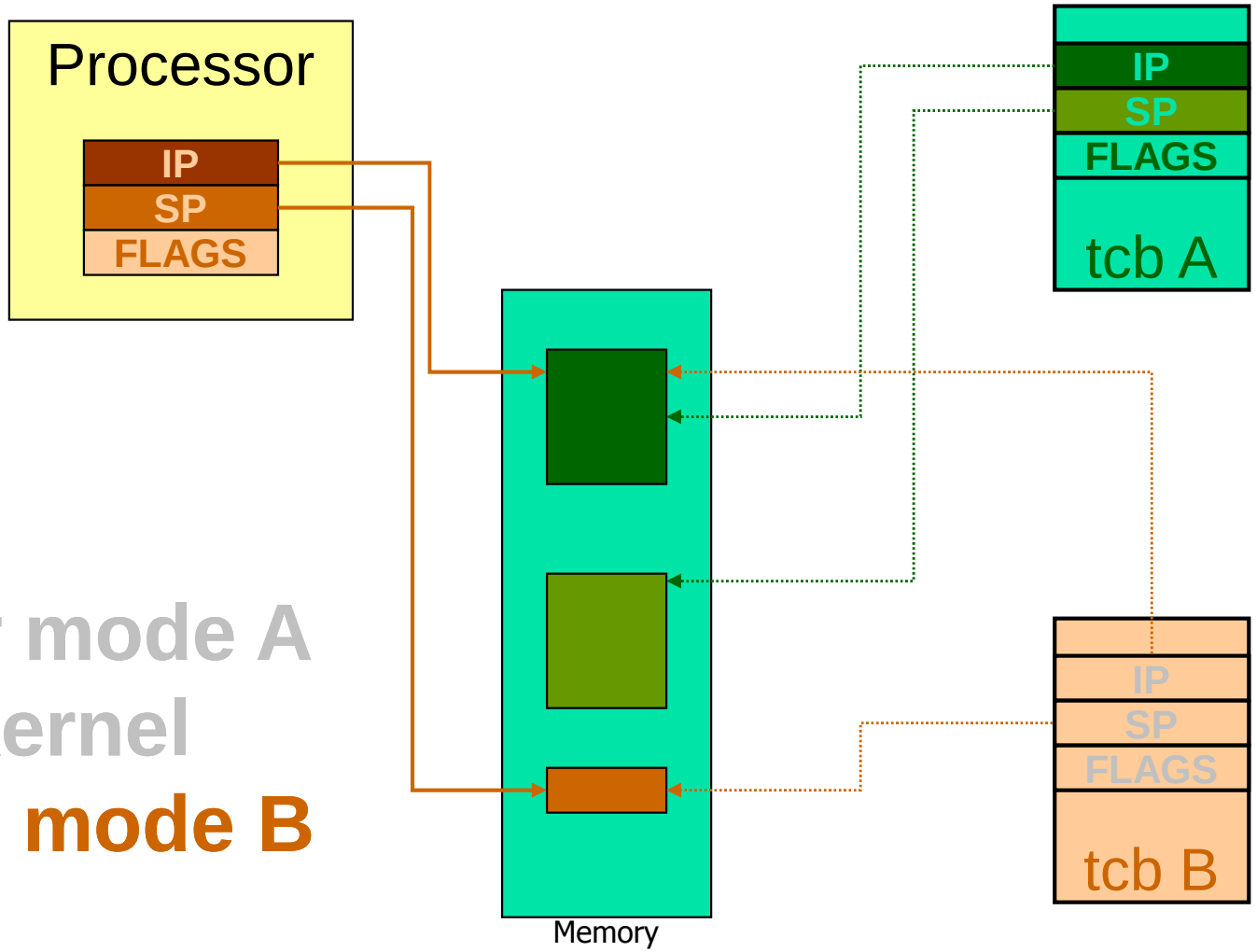
Thread Switch A → B

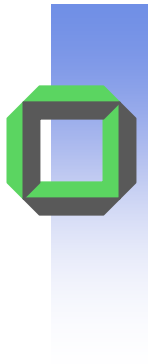


Thread Switch A → B



Thread Switch A → B

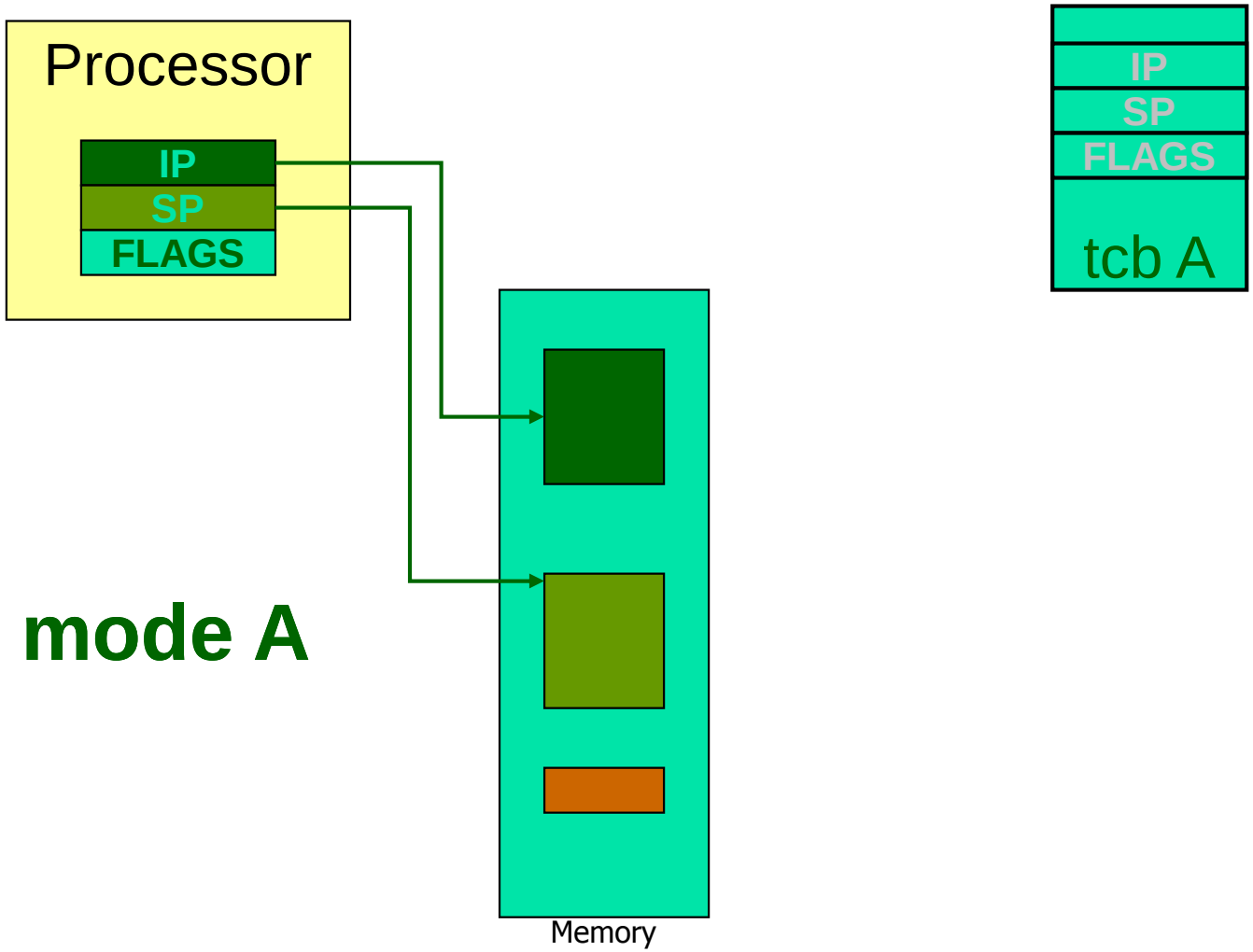




Thread Switch A → B

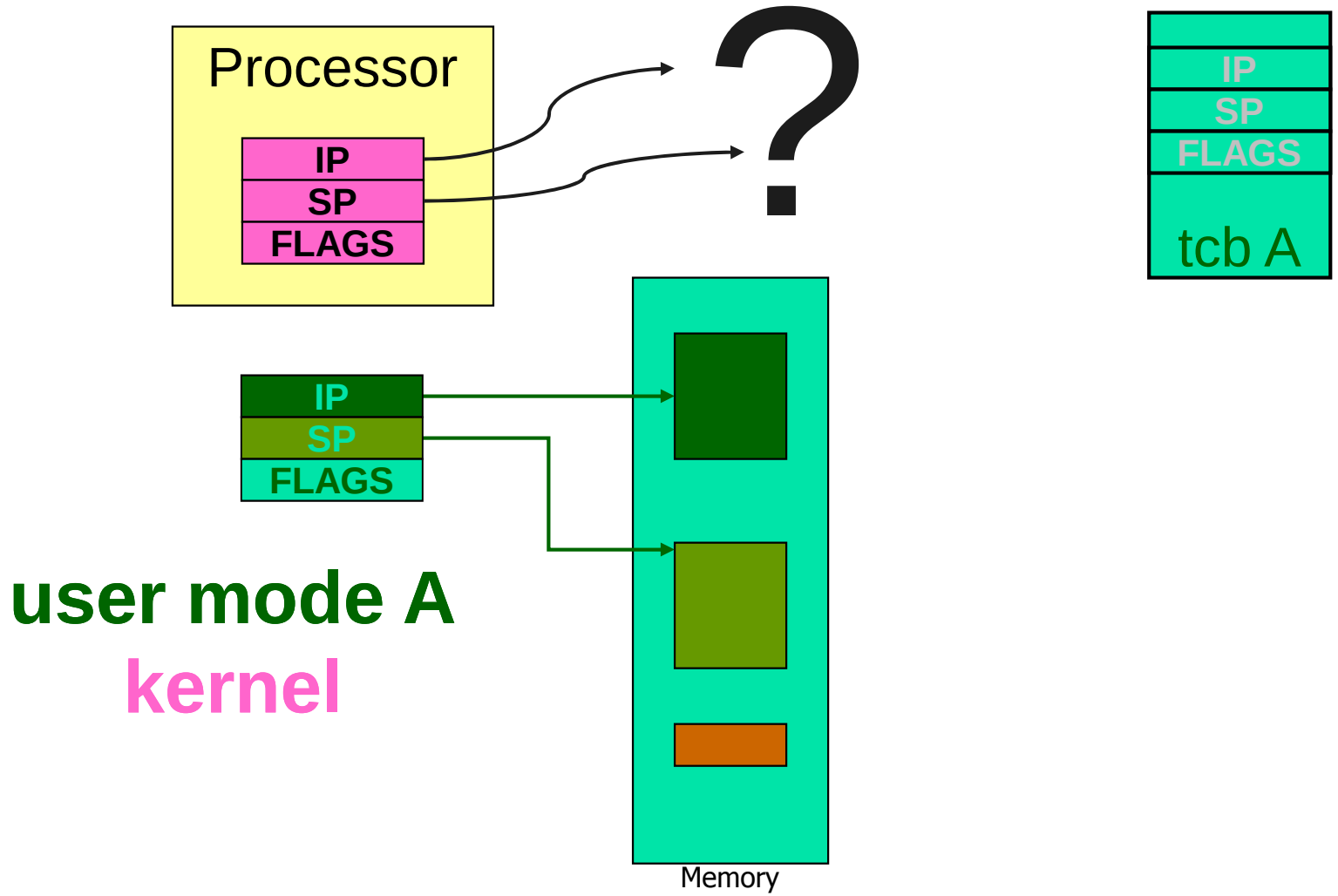
- Thread A is running in user mode
- Thread A experiences its end of time slice or is preempted by a (device) interrupt
- We enter kernel mode
- The microkernel saves the status of thread A on A's TCB
- The microkernel loads the status of thread B from B's TCB
- We leave kernel mode
- Thread B is running in user mode

Thread Switch A → kernel → B

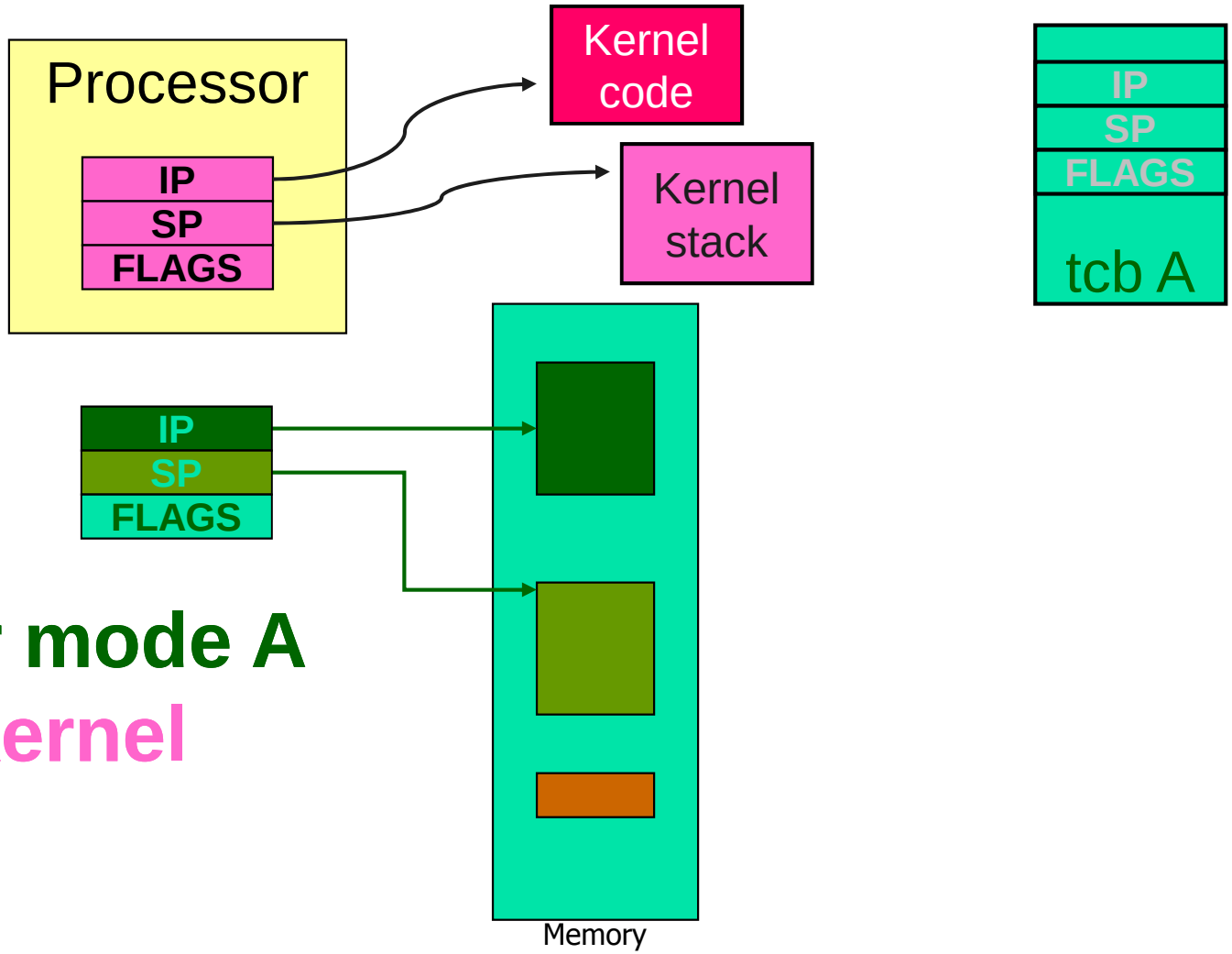


user mode A

Thread Switch A → kernel → B



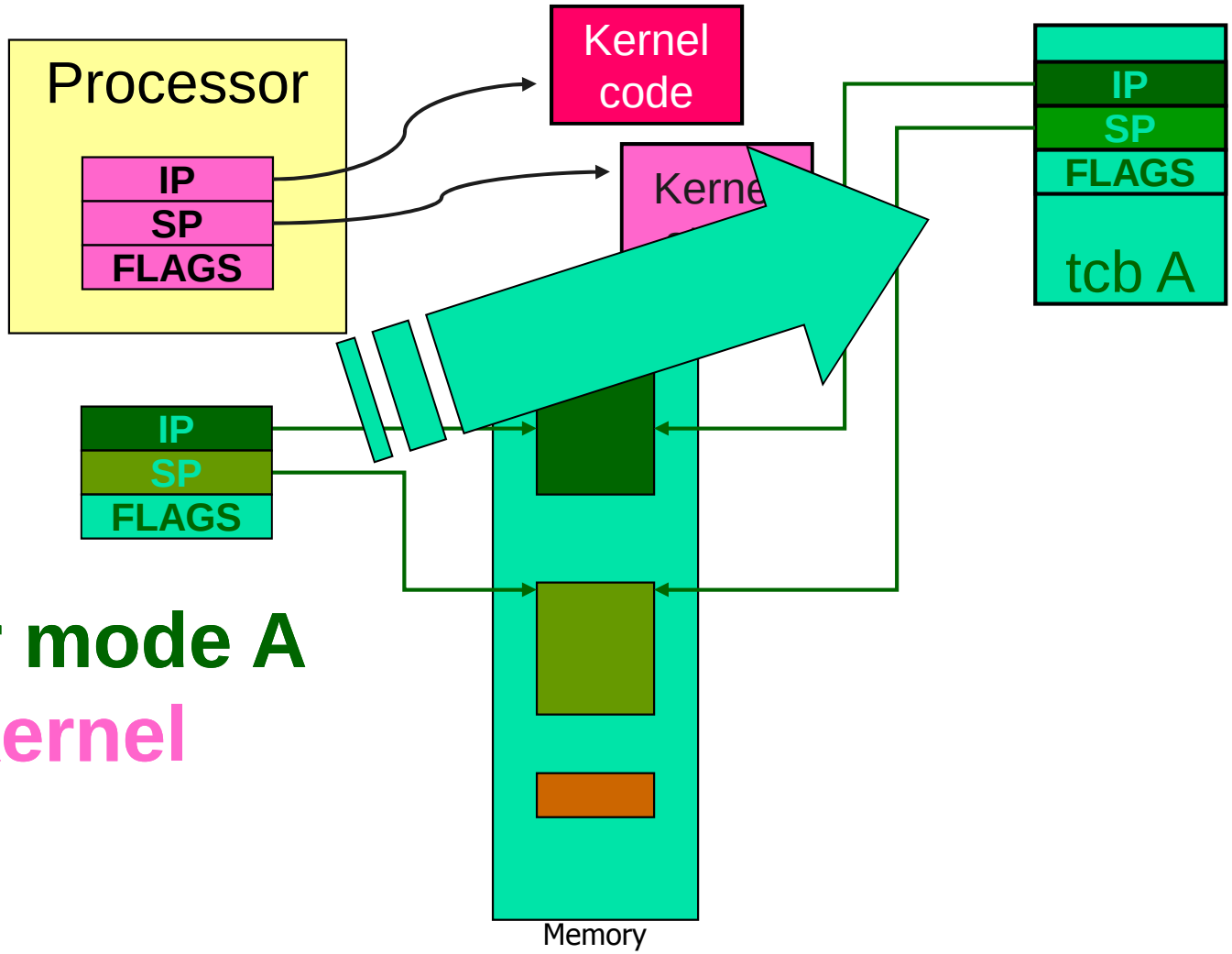
Thread Switch A → kernel → B



user mode A

kernel

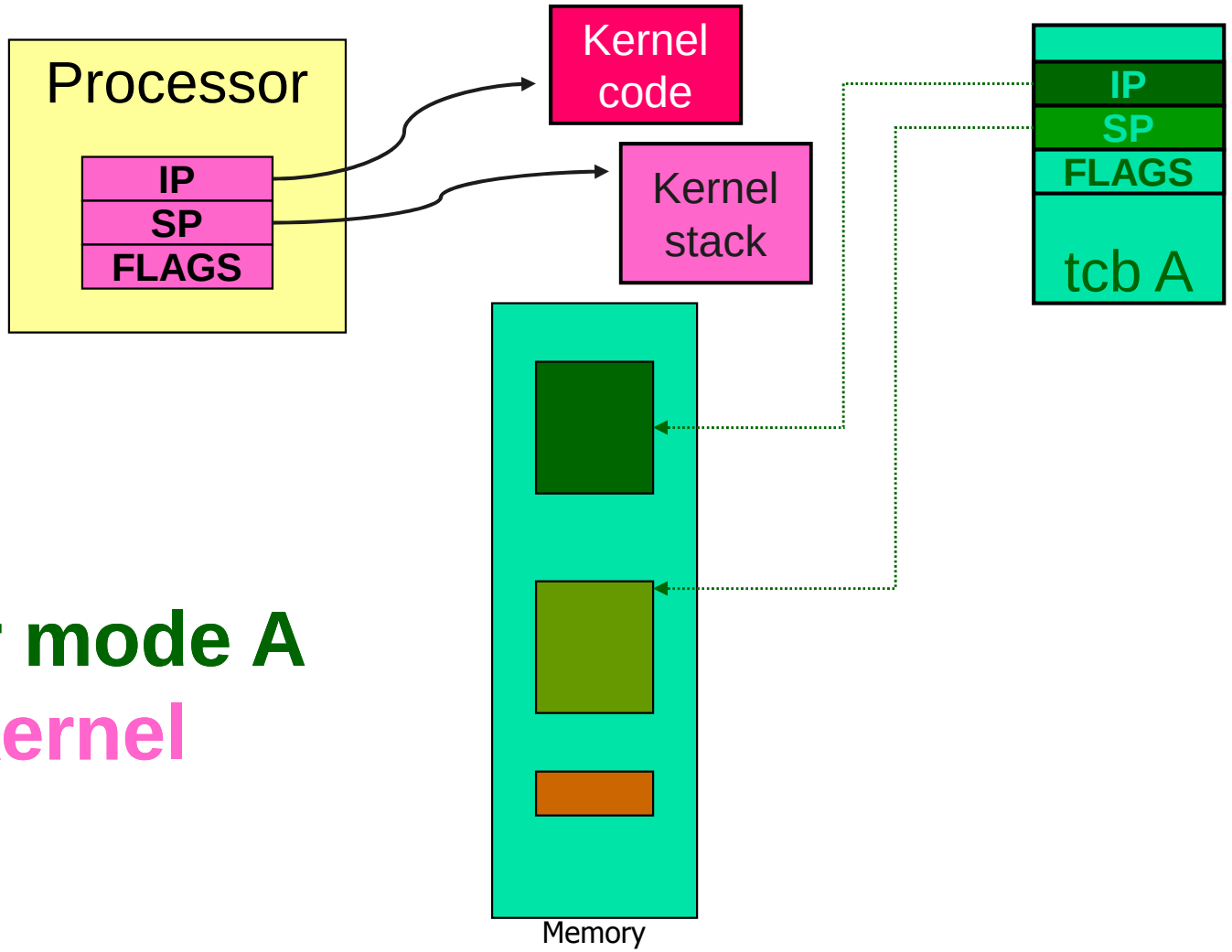
Thread Switch A → kernel → B



user mode A

kernel

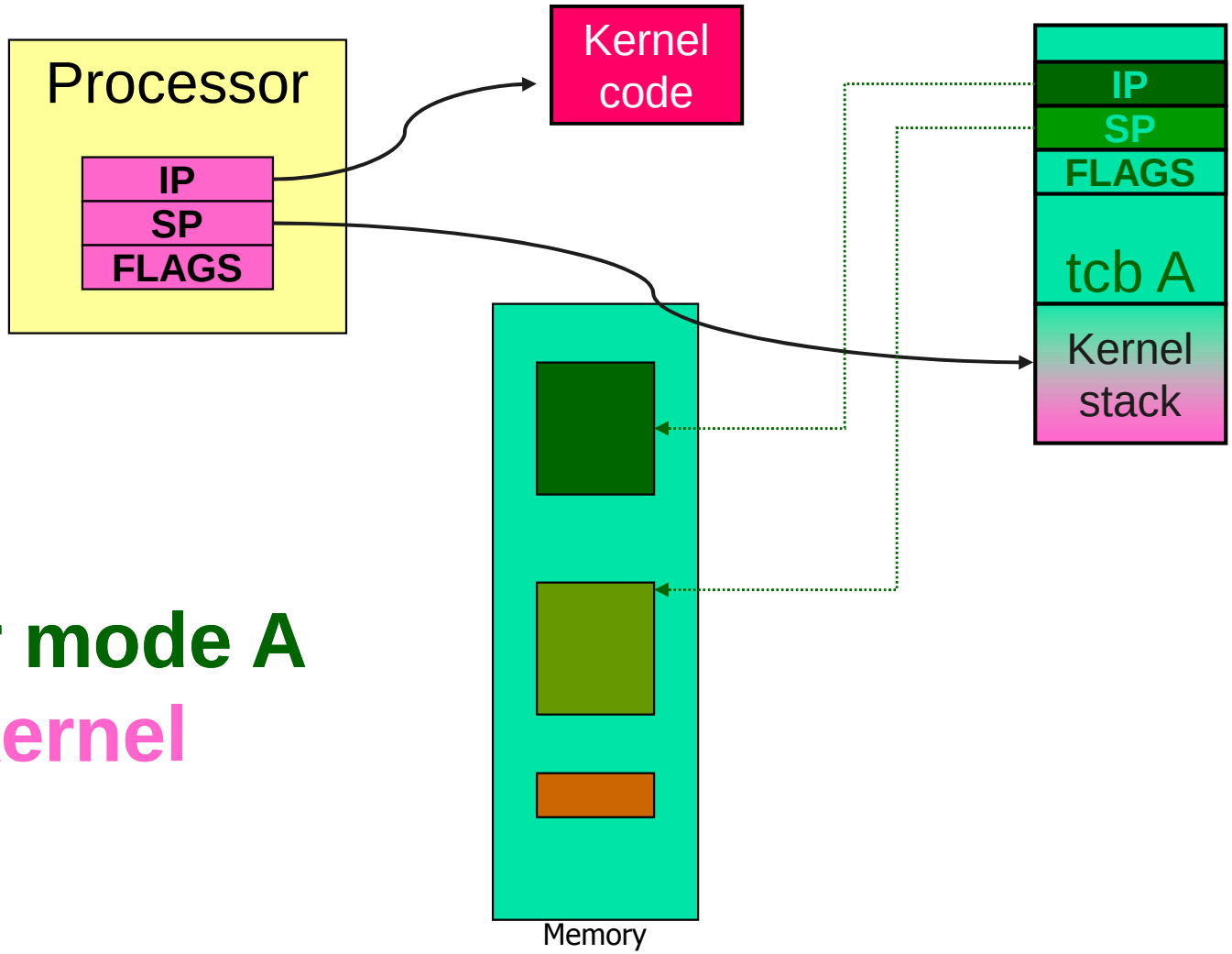
Thread Switch A → kernel → B



user mode A

kernel

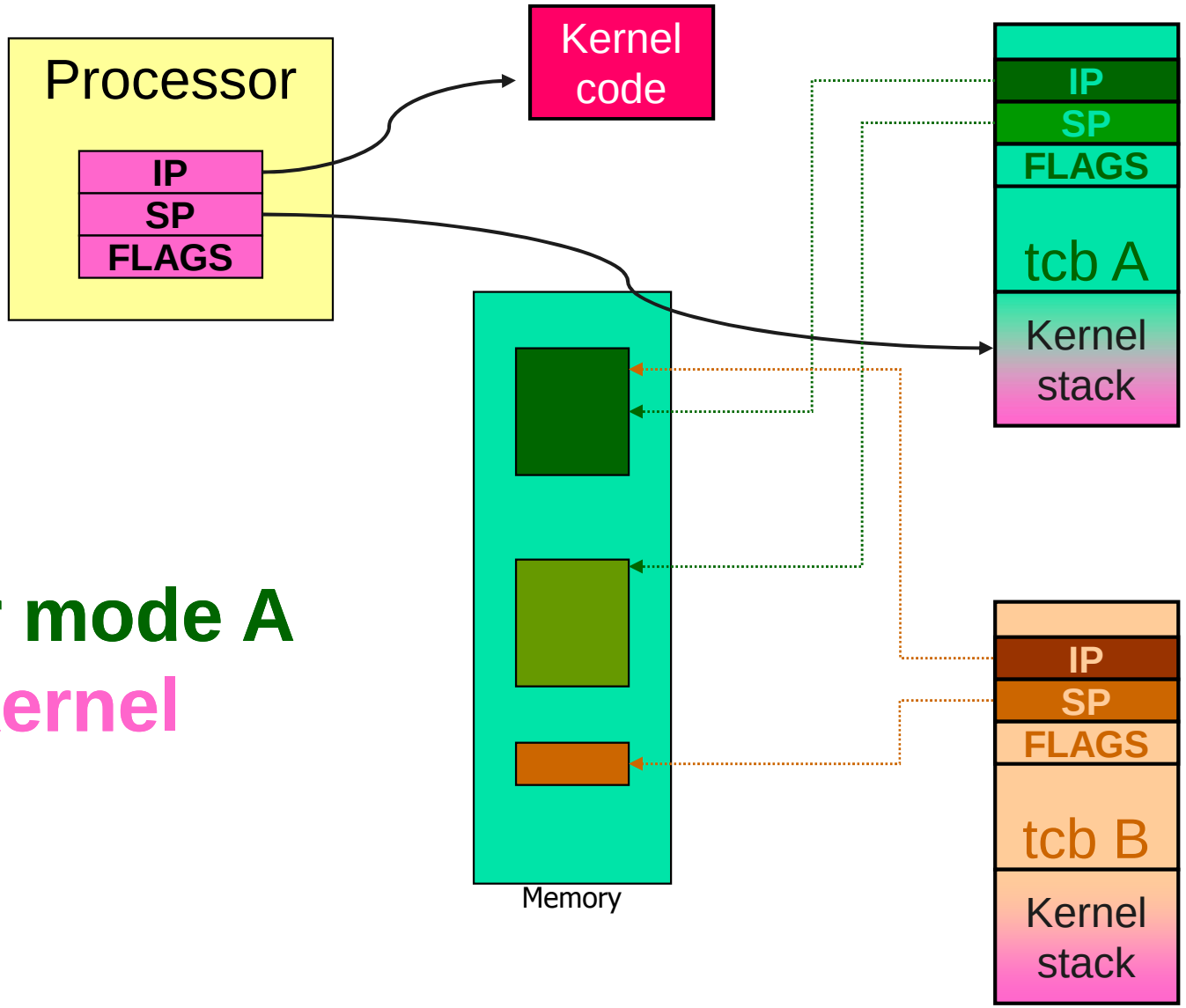
Thread Switch A → kernel → B



user mode A

kernel

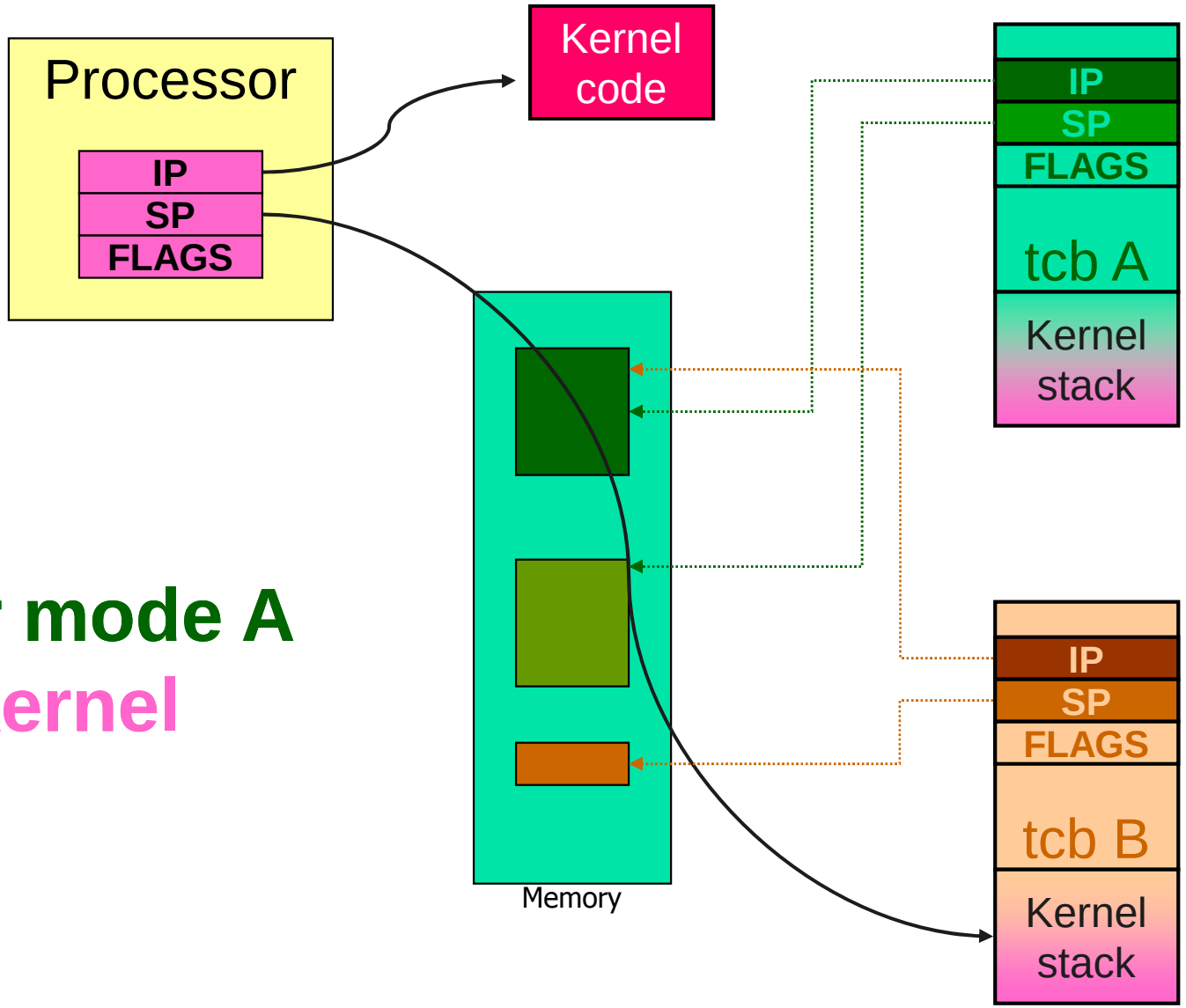
Thread Switch A → kernel → B



user mode A

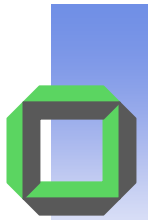
kernel

Thread Switch A → kernel → B



user mode A

kernel



Construction Conclusion

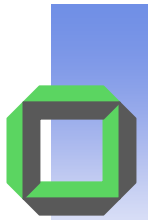
From the view of the designer there are two alternatives:

Single Kernel Stack

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

Per-Thread Kernel Stack

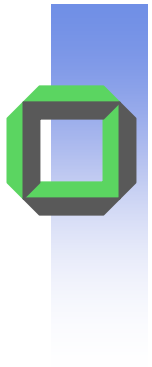
- Each thread has its own stack in kernel mode



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 footprint
  - The same stack is always used!



Per-Thread Kernel Stack

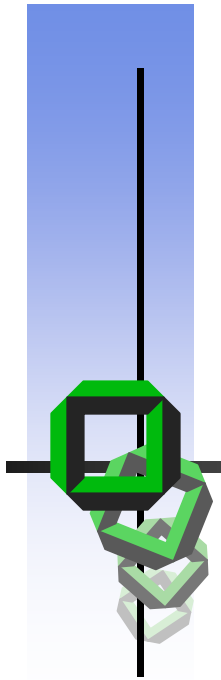
Thread model

- Simple, elegant, flexible
  - 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

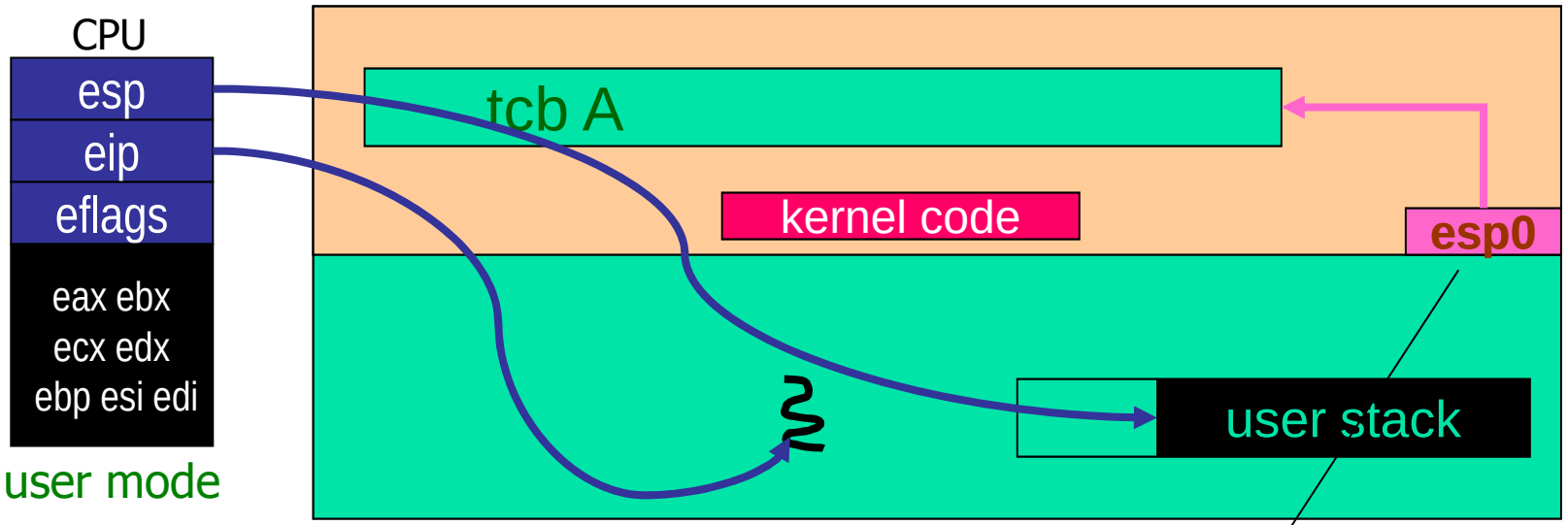
- Larger cache footprint
- Difficult to exchange kernel on-the-fly

Conclusion:
Either no persistent TCBs or TCBs must hold *symbolic* addresses



Kernel Entry and Exit on IA-32

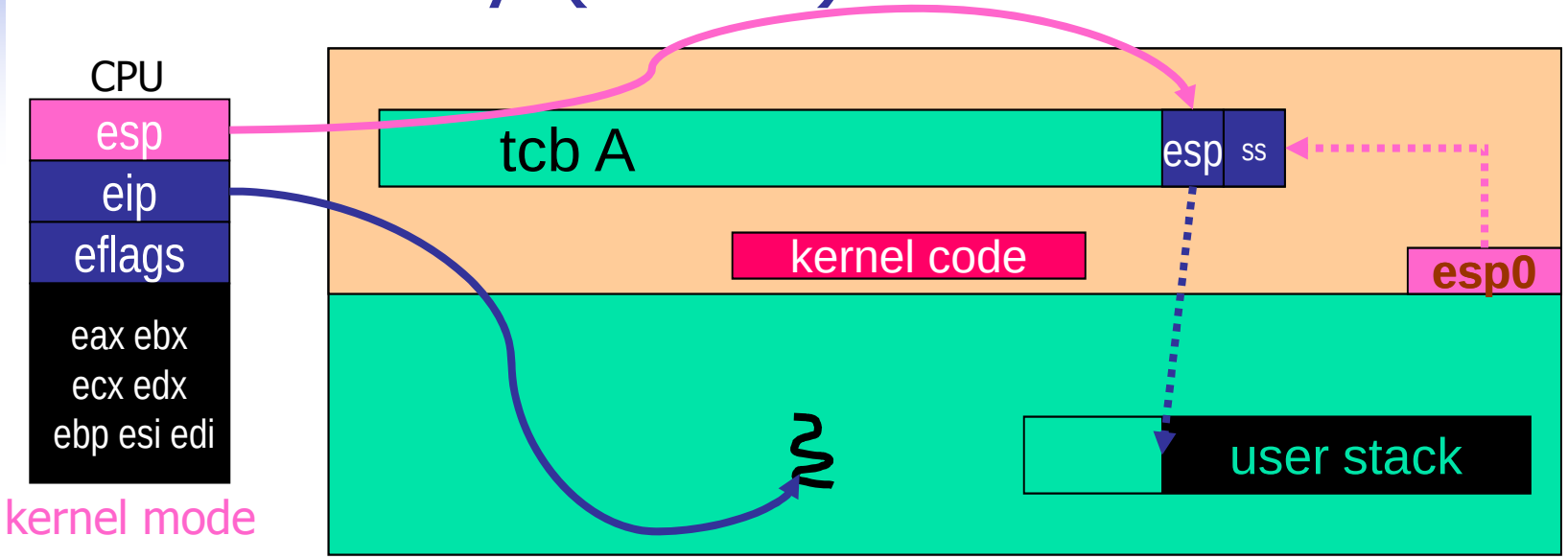
Kernel Entry (IA-32)



- Trap/fault occurs (int *n* / exception / interrupt)

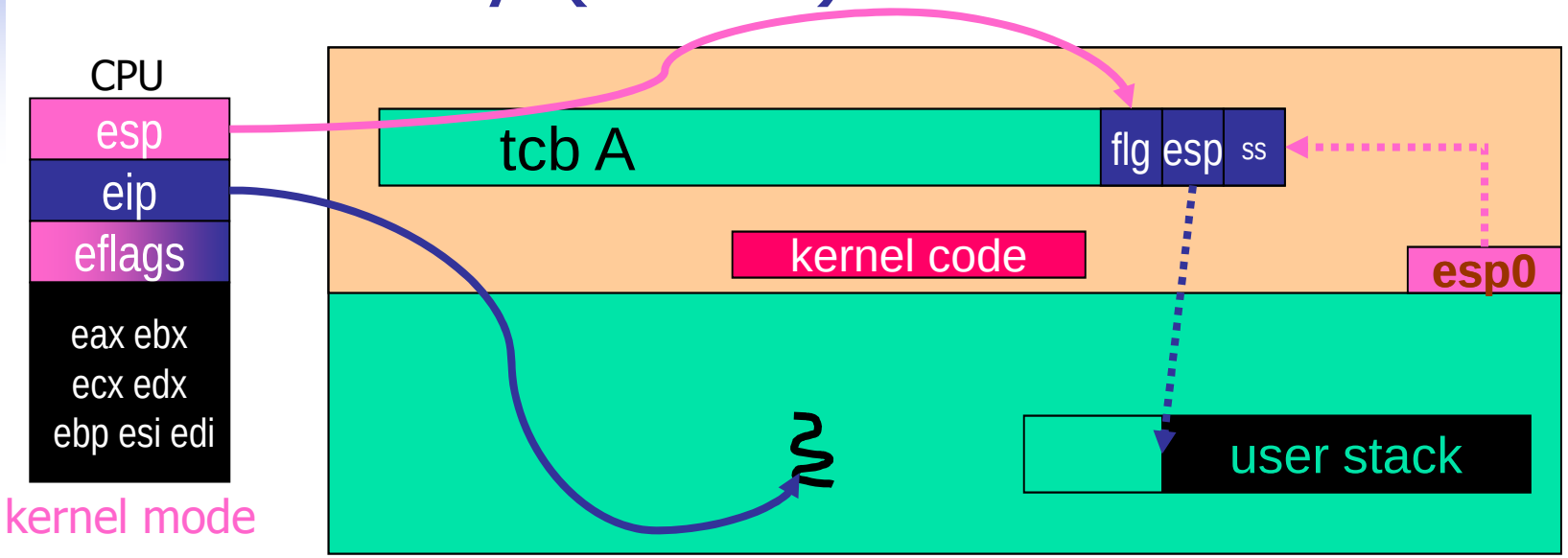
points to the currently running thread's kernel stack

Kernel Entry (IA-32)



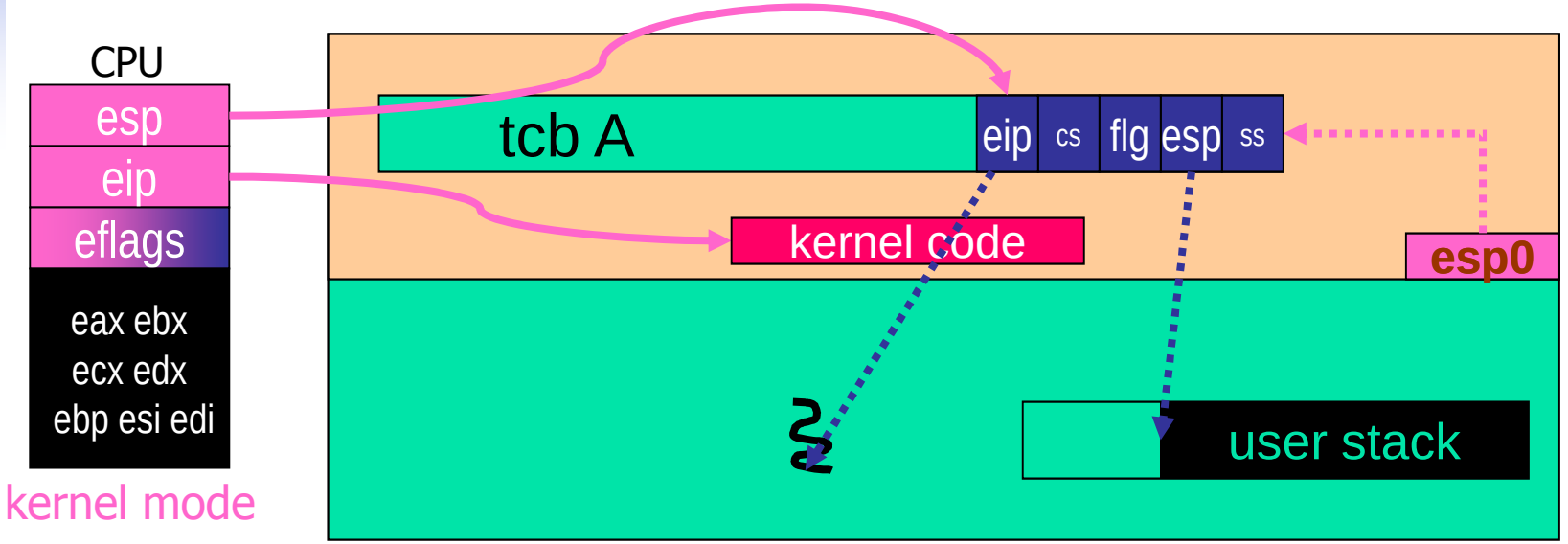
- Trap/fault occurs (int *n* / exception / interrupt)
  - Push user SS:ESP onto kernel stack, load kernel SS:ESP

Kernel Entry (IA-32)



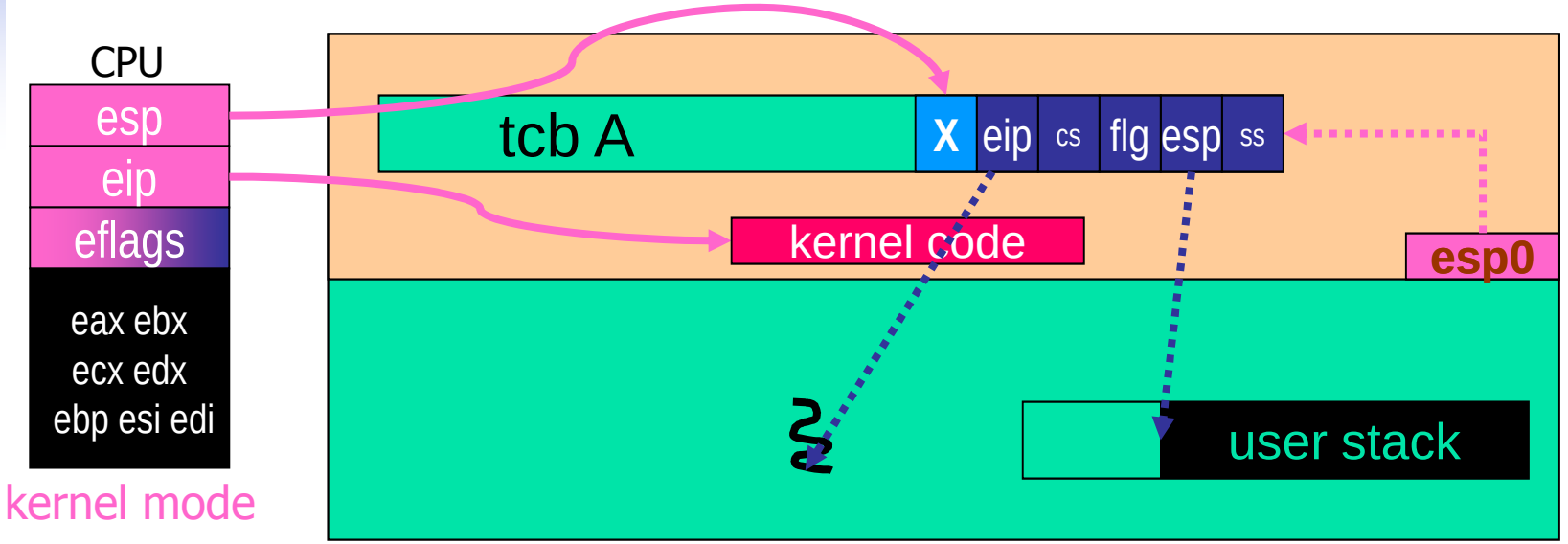
- Trap/fault occurs (int *n* / exception / interrupt)
  - Push user SS:ESP onto kernel stack, load kernel SS:ESP
  - Push user EFLAGS, reset flags (I := 0, CPL := 0)

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

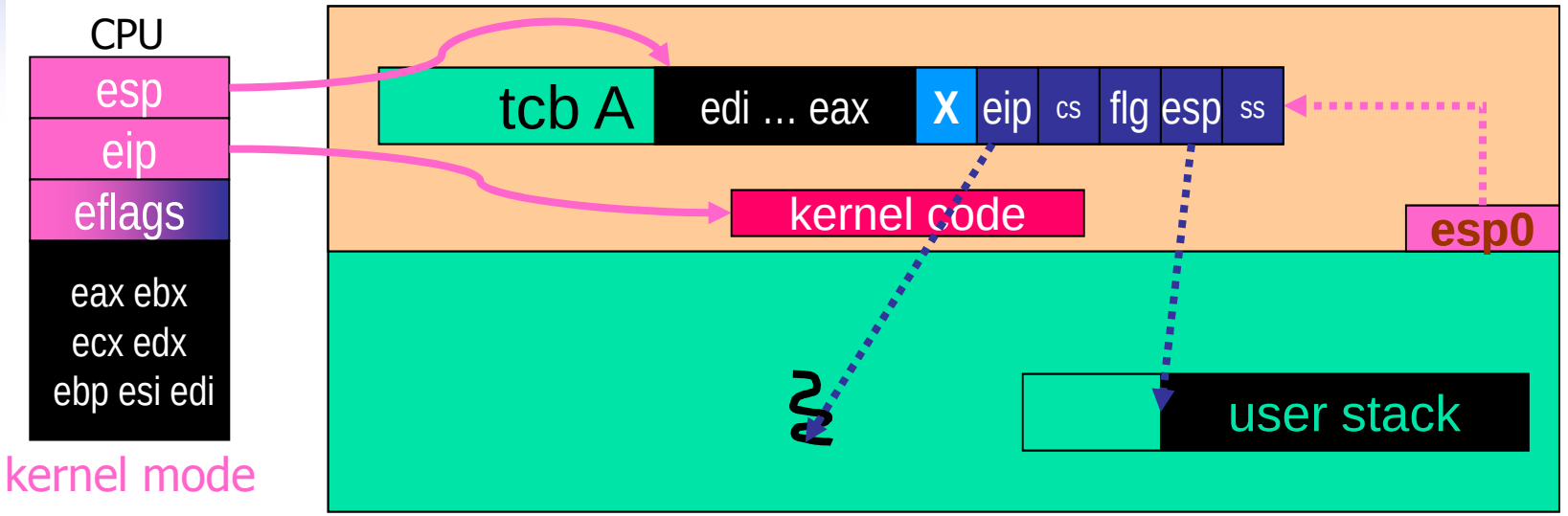
Kernel Entry (IA-32)



- 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"

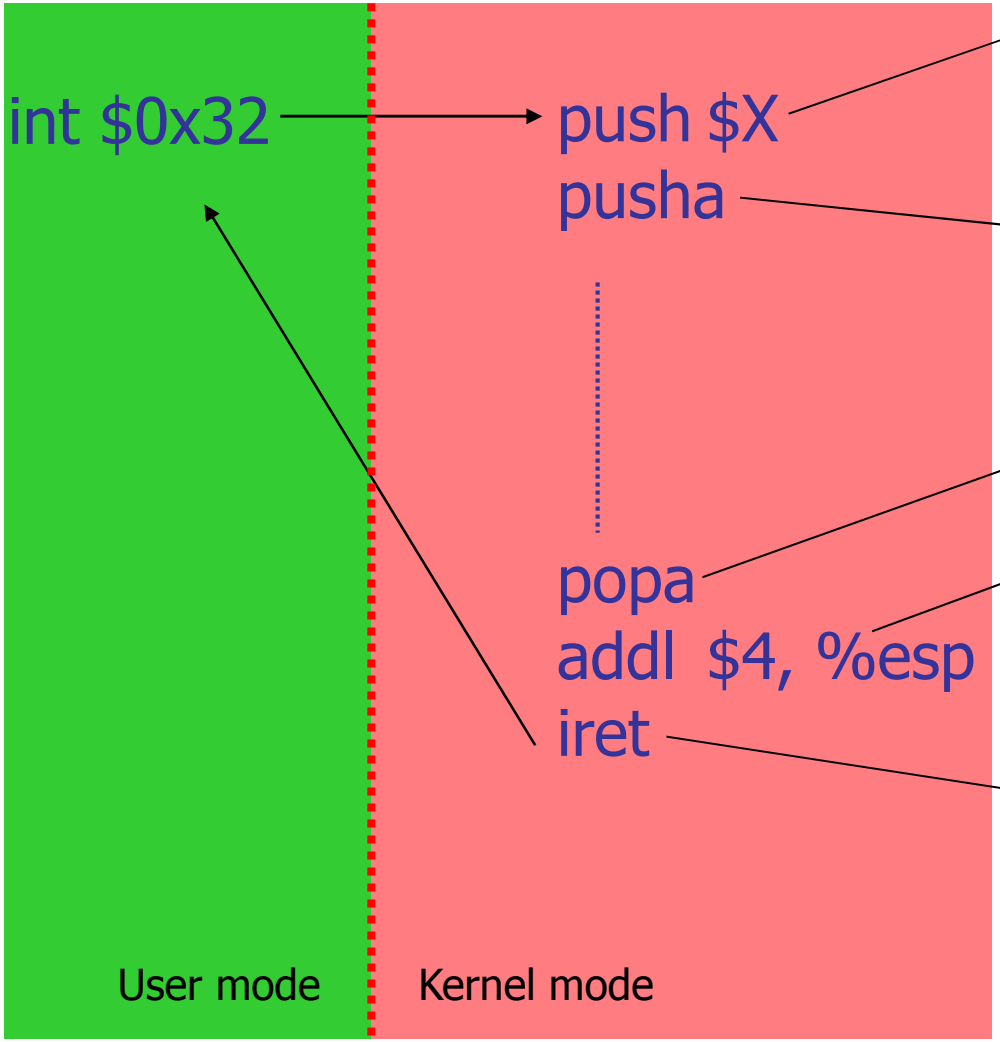
Kernel Entry (IA-32)



- 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)



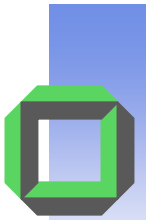
Error code, e.g., 3 means page fault

Push all the register content to the stack

Pop all registers

ESP := ESP + 4
Restore the old ESP, discard error code $X

Interrupt return: single instruction that restores user SS:ESP and CS:EIP from kernel stack

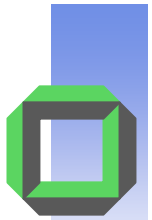


Kernel Stack State

Uniprocessor:

- Any kernel stack ≠ **`myself`** is up-to-date!
  - **`myself`**'s kernel stack is up-to-date when in kernel mode except for brief moment.

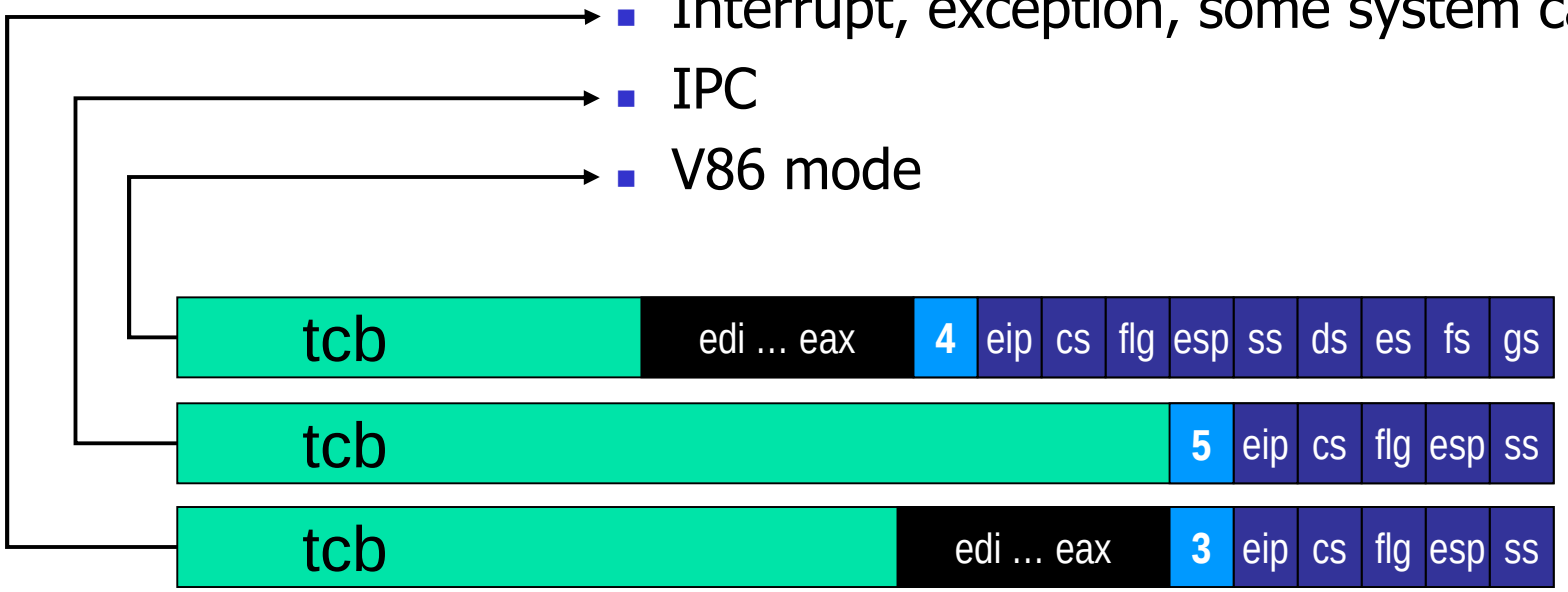
- One thread running
- All the others in their kernel-state
  - We can analyze their kernel stacks (e.g., L4 ExchangeRegisters(), kdb)
- All processes except the one running are in kernel mode

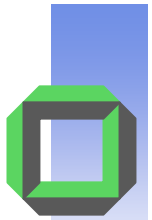


Kernel Stack State

Uniprocessor:

- Any kernel stack ≠ **`myself`** is up-to-date!
  - **`myself`**'s kernel stack is up-to-date when in kernel mode except for brief moment.
- X permits to differentiate between stack layouts
  - Interrupt, exception, some system calls
  - IPC
  - V86 mode

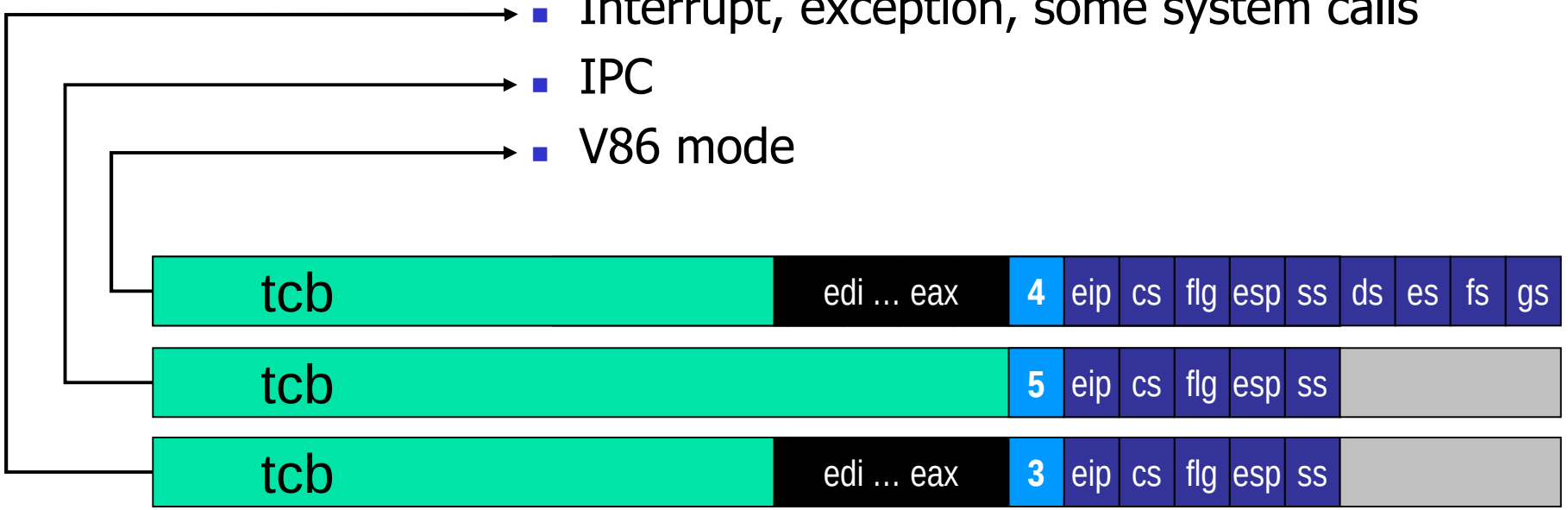


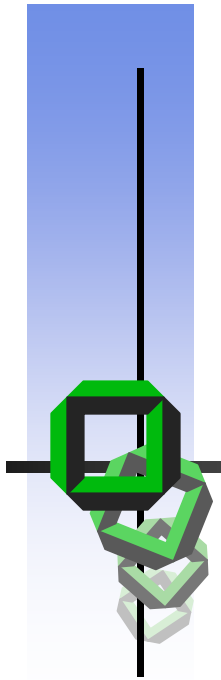


Kernel Stack State

Uniprocessor:

- Any kernel stack ≠ **`myself`** is up-to-date!
  - **`myself`**'s kernel stack is up-to-date when in kernel mode except for brief moment.
- X permits to differentiate between stack layouts
  - Interrupt, exception, some system calls
  - IPC
  - V86 mode





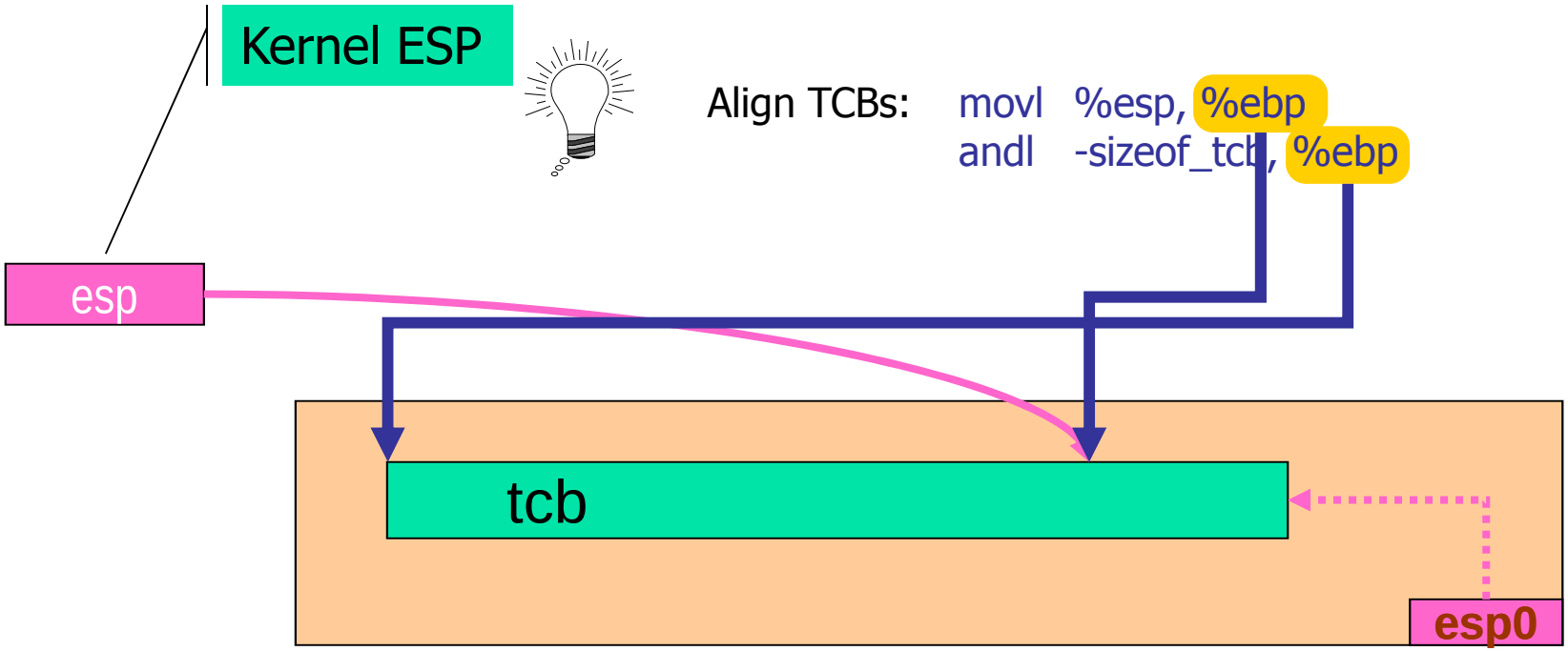
Thread Switch on IA-32

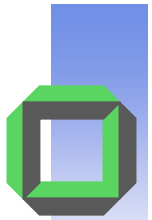
Locating the TCB

Remember: We need to find

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

Next lecture





Thread Switch (IA-32)

Thread A

int $0x32

```
pushl  $X
pusha
movl   %esp, %ebp
andl   -sizeof_tcb, %ebp
/* edi == address of B's TCB */

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

addl   sizeof_tcb, %edi
movl   %edi, esp0

popa
addl   $4, %esp
iret
```

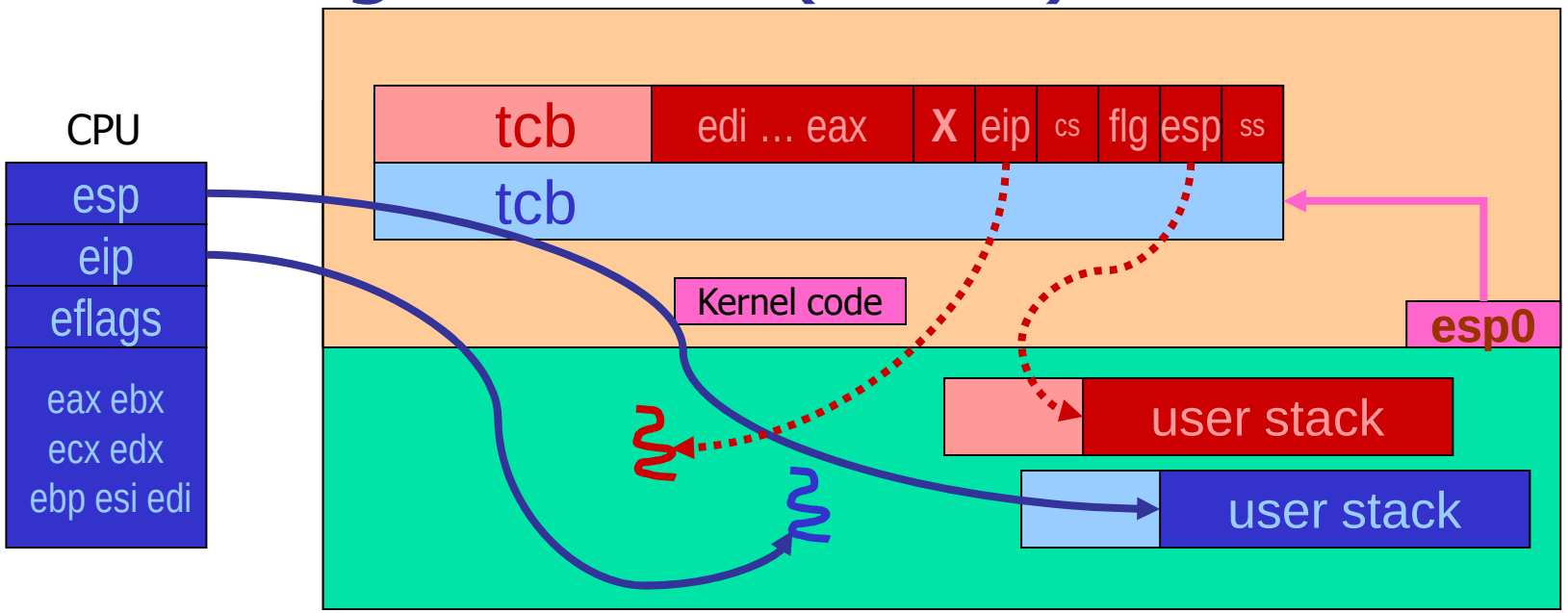
Switch current kernel stack pointer

Thread B

int $0x32

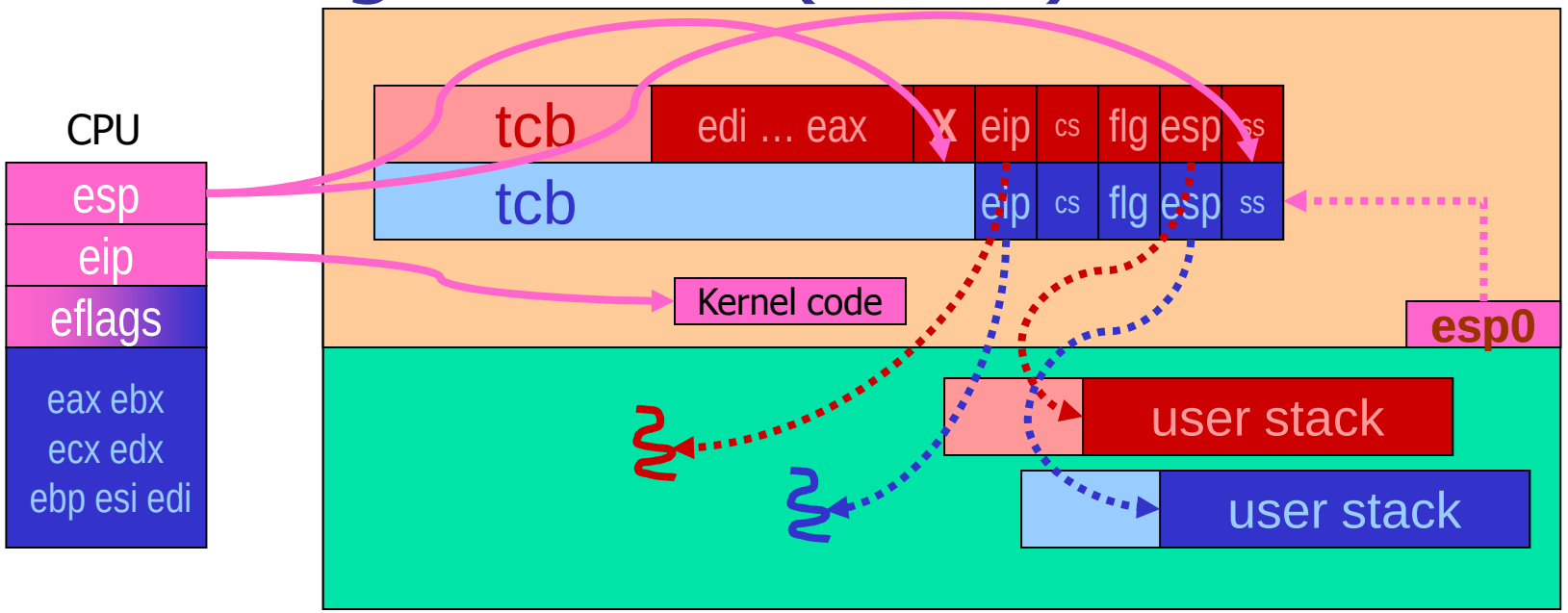
Switch ESP0 so that next kernel *entry* uses new kernel stack

Switching Threads (IA-32)



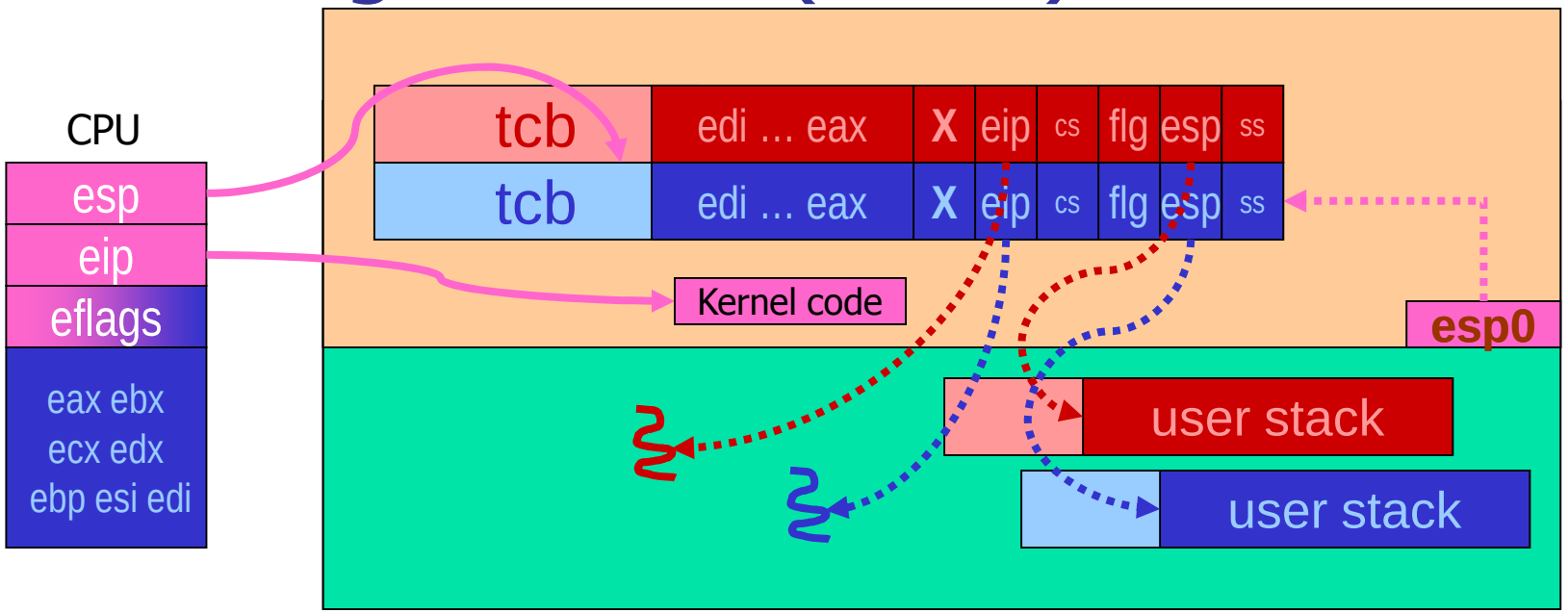
- int $0x32

Switching Threads (IA-32)



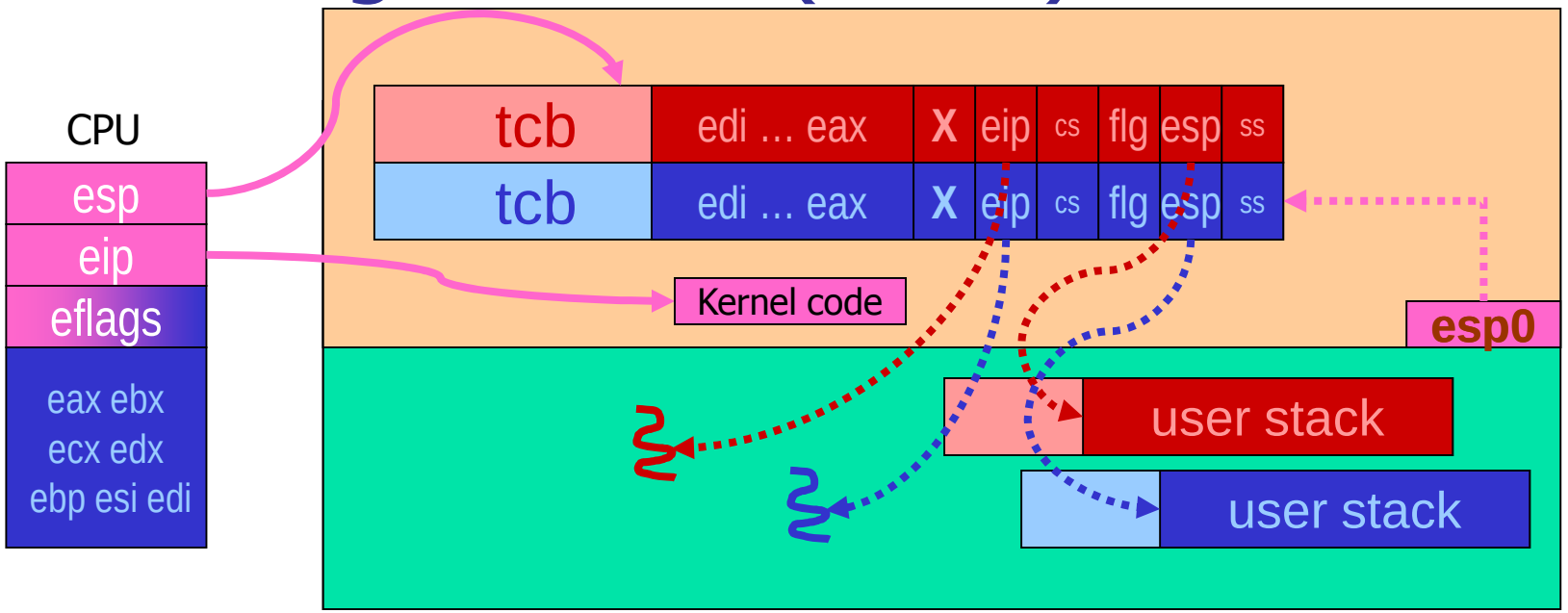
- int $0x32

Switching Threads (IA-32)



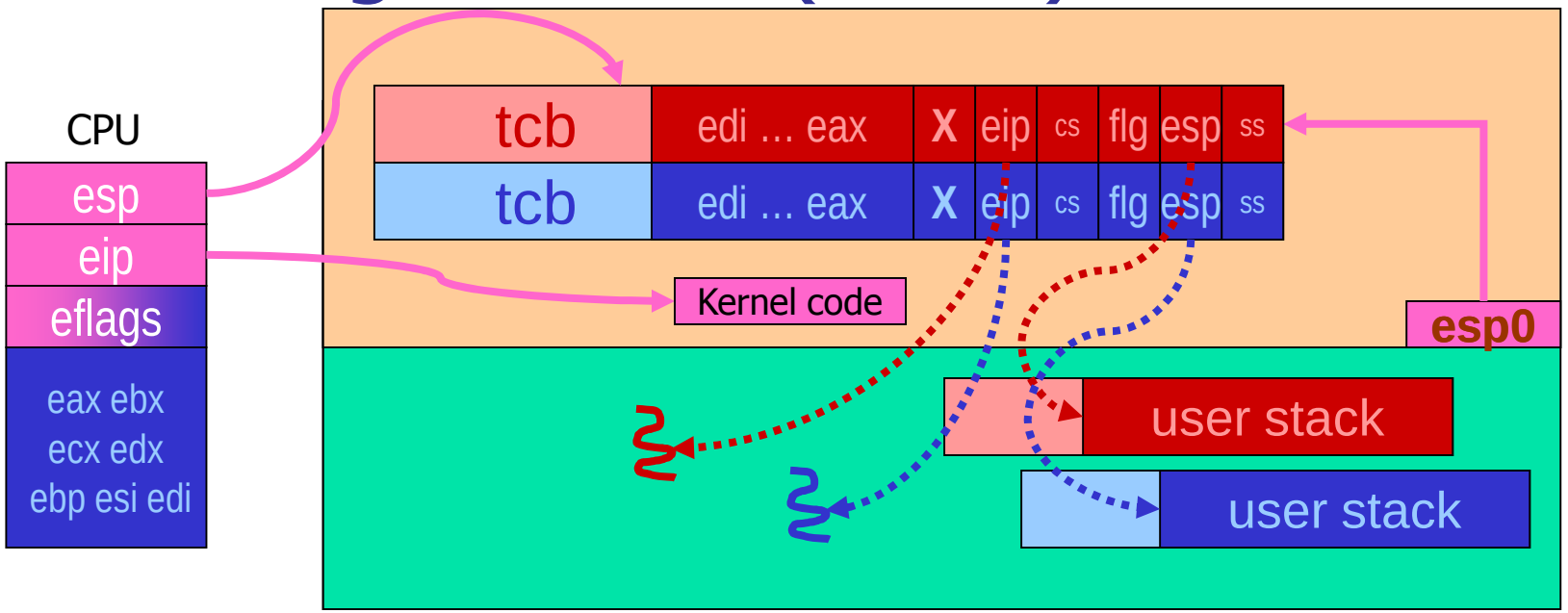
- int $0x32, push registers of blue thread

Switching Threads (IA-32)



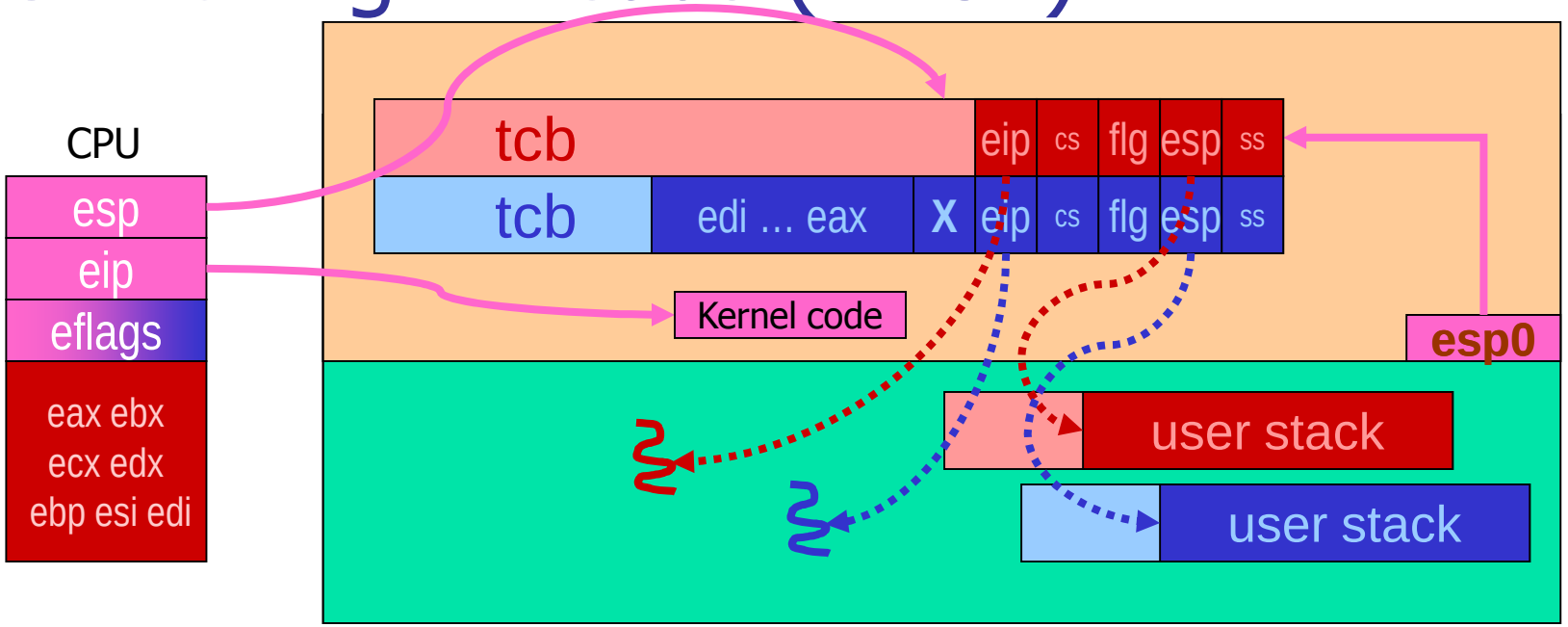
- int $0x32, push registers of blue thread
- Switch kernel stacks (store and load ESP)

Switching Threads (IA-32)



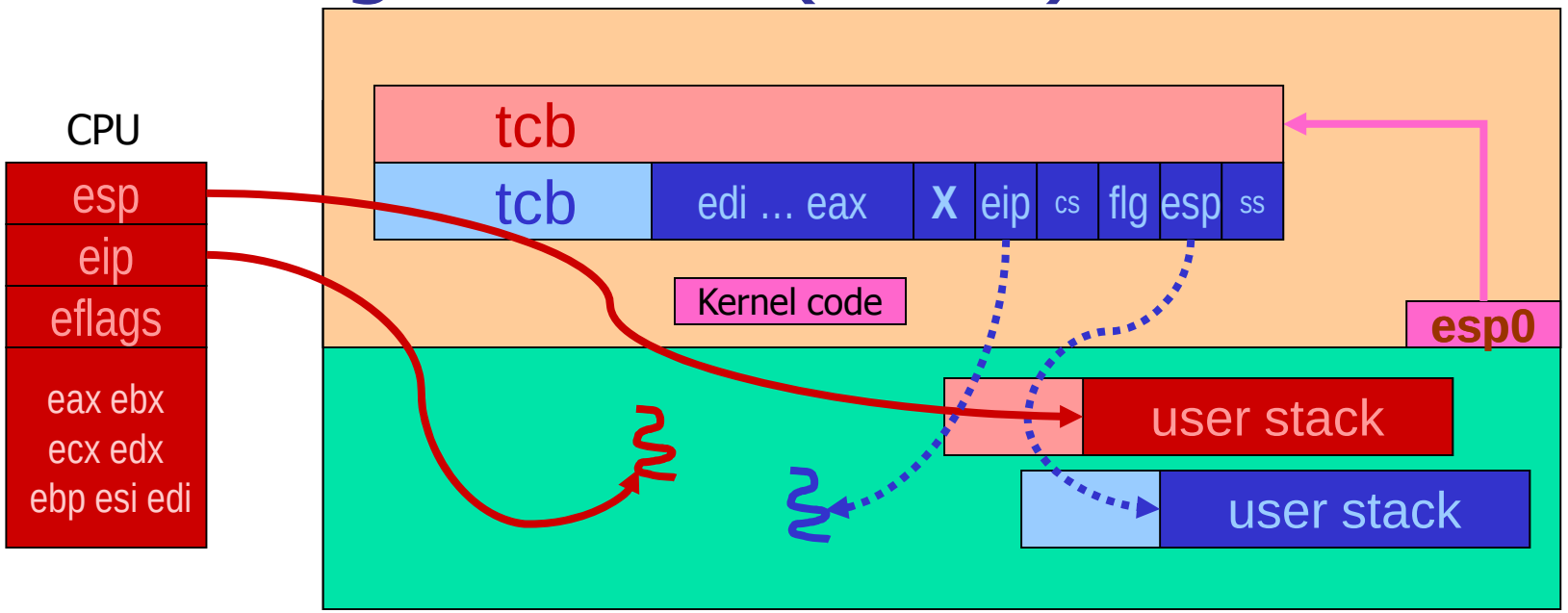
- int $0x32, push registers of blue thread
- Switch kernel stacks (store and load ESP)
- Set ESP0 to new kernel stack

Switching Threads (IA-32)

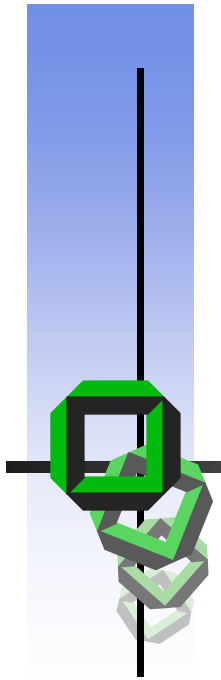


- int $0x32, push registers of blue thread
- Switch kernel stacks (store and load ESP)
- Set ESP0 to new kernel stack
- Pop red registers

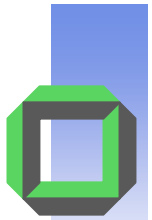
Switching Threads (IA-32)



- int $0x32, 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)

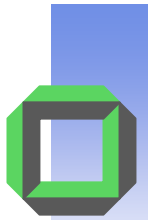


Fast System Calls on IA-32



Syscalls via int *n* are too slow

- int *n* touches too much memory (cache, TLB)
  - Save user mode CS:EIP, SS:ESP, EFLAGS
  - Load kernel mode SS0:ESP0 (from TSS)
  - Load kernel mode CS:EIP (from IDT)
- Does too much (microcode)
  - Segmentation not used today
  - Many checks could be avoided



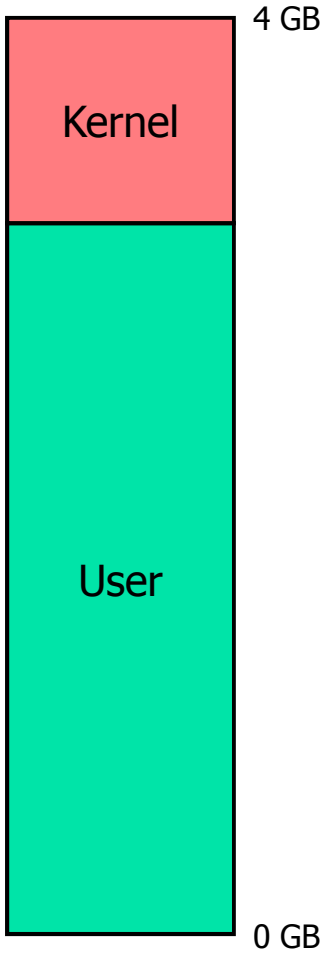
Sysenter/sysexit (IA-32)

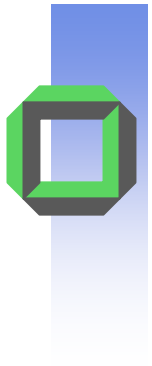
- Model Specific Register (MSR)
  - Kernel IP
  - Kernel SP
  - Flat 4 GB segments
- Sysenter
  - EIP := MSR[Kernel IP]
  - ESP := MSR[Kernel SP]
  - EFlags.I := 0, CPL := 0

Interrupts off

Enter kernel mode

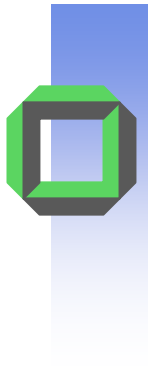
- Sysexit
  - ESP := ECX
  - EIP := EDX
  - CPL := 3





Sysenter/sysexit (IA-32)

- Model Specific Register (MSR)
  - Kernel IP
  - Kernel SP
  - Flat 4 GB segments
- Sysenter
  - EIP := MSR[Kernel IP]
  - ESP := MSR[Kernel SP]
  - EFlags.I := 0, CPL := 0
  - User has to provide current user IP and SP
  - By convention (e.g., ECX, EDX)
  - Flags undefined
- Sysexit
  - ESP := ECX
  - EIP := EDX
  - CPL := 3
  - Kernel has to re-enable interrupts



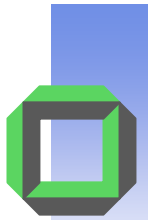
Sysenter/sysexit and int *n*

- Emulate effects of int instruction after sysenter
  (assuming ECX=USP, EDX=UIP)

```
subl   $20, %esp
movl   %ecx, 16(%esp)
movl   %edx, 4(%esp)
movl   $5, (%esp)
```

- Emulate iret instruction

```
movl   16(%esp), %ecx
movl   4(%esp), %edx
sti
sysexit
```



Sysenter/sysexit and int *n*

- Emulate effects of int instruction after sysenter (assuming ECX=USP, EDX=UIP)

```
movl  (%esp), %esp
subl  $20, %esp
movl  %ecx, 16(%esp)
movl  %edx, 4(%esp)
movl  $5, (%esp)
```

- Emulate iret instruction

```
movl  16(%esp), %ecx
movl  4(%esp), %edx
sti
sysexit
```

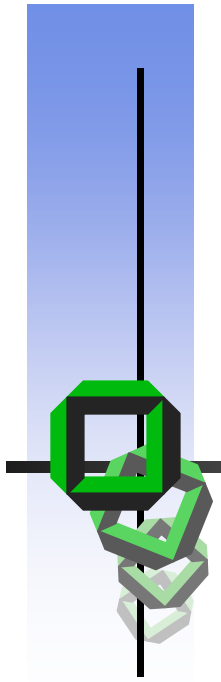
Trick: MSR points to TSS.esp0

\+ Avoids slow updates of the MSR

\+ Update only TSS.esp0 on address space switch

\+ Keeps MSR (used by sysenter) and TSS.esp0 (used otherwise) in sync

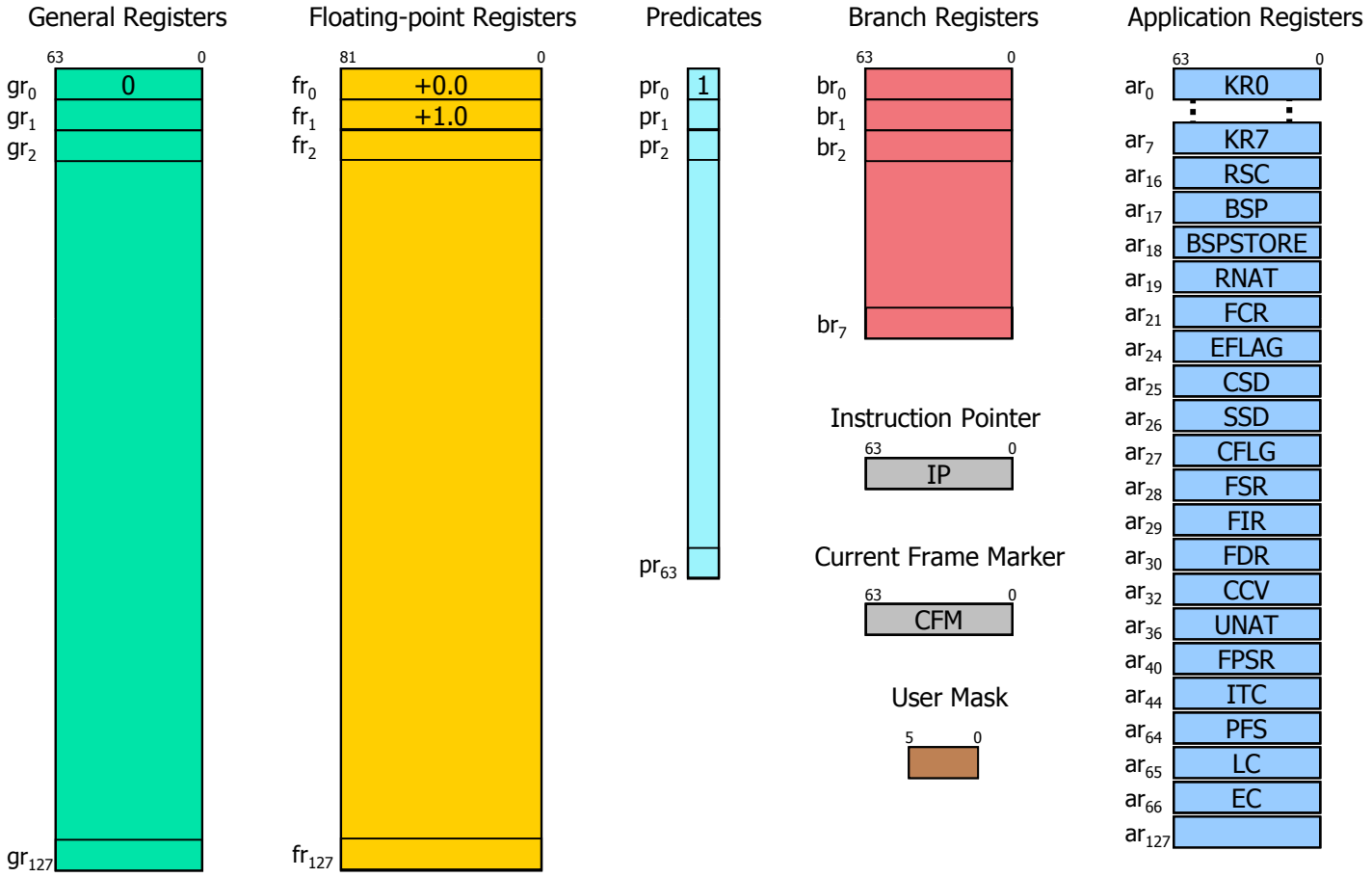
\- Requires memory access



Case study: Itanium (aka IA-64)

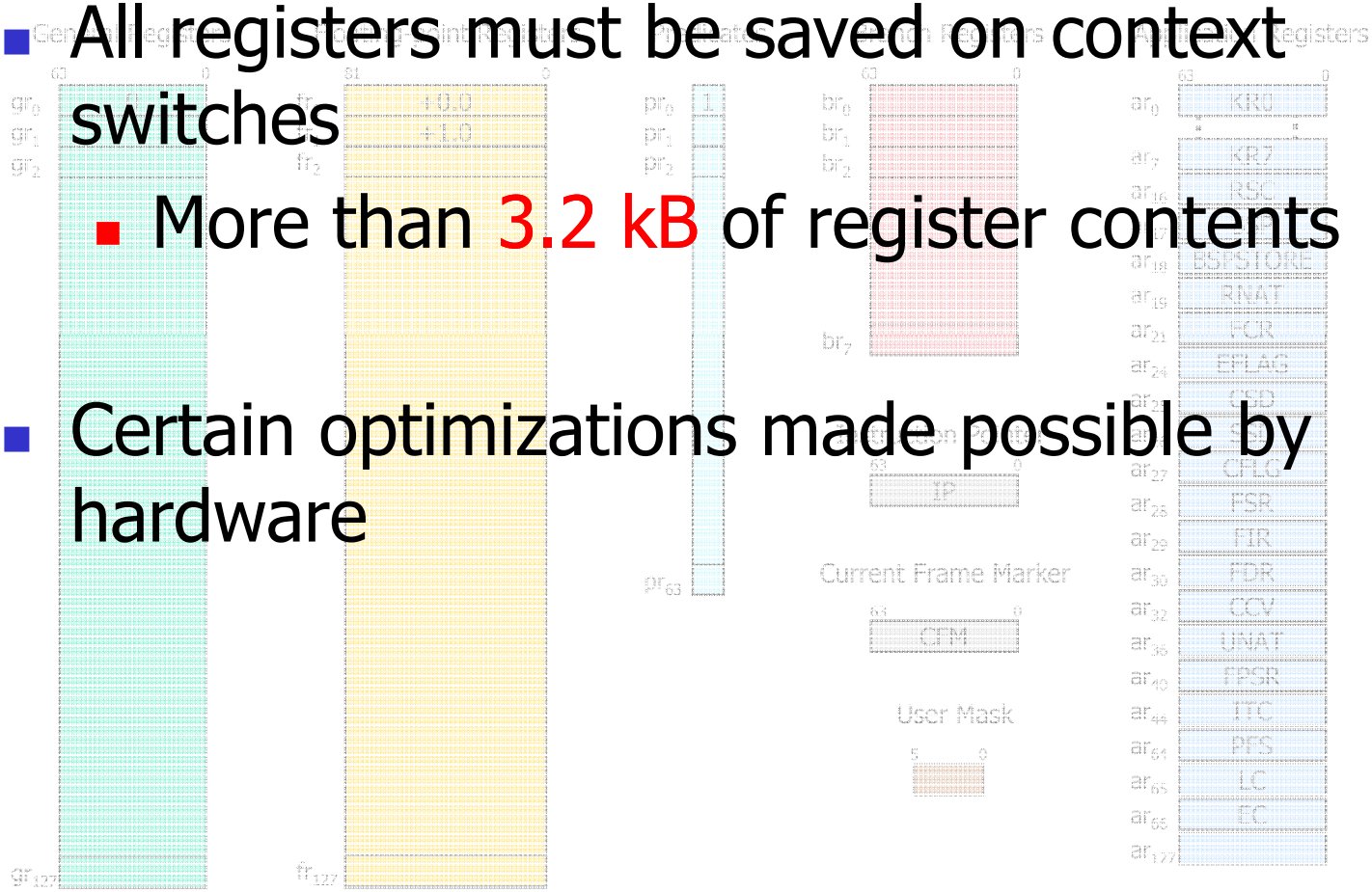
Thread Switching and
Kernel Entry

IA-64 User Accessible Registers

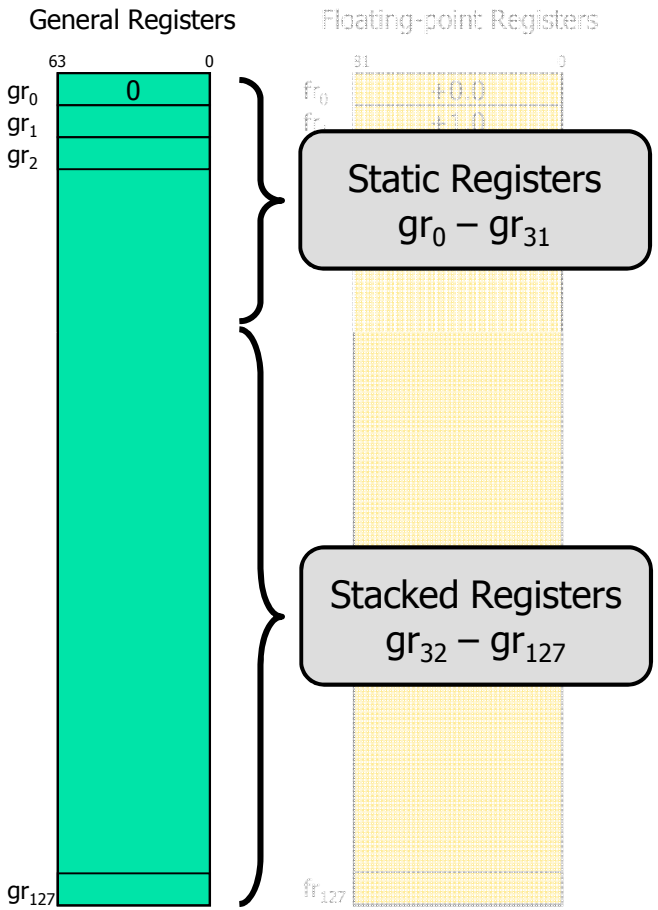


Thread Switching Overhead

- All registers must be saved on context switches
  - More than 3.2 kB of register contents
- Certain optimizations made possible by hardware

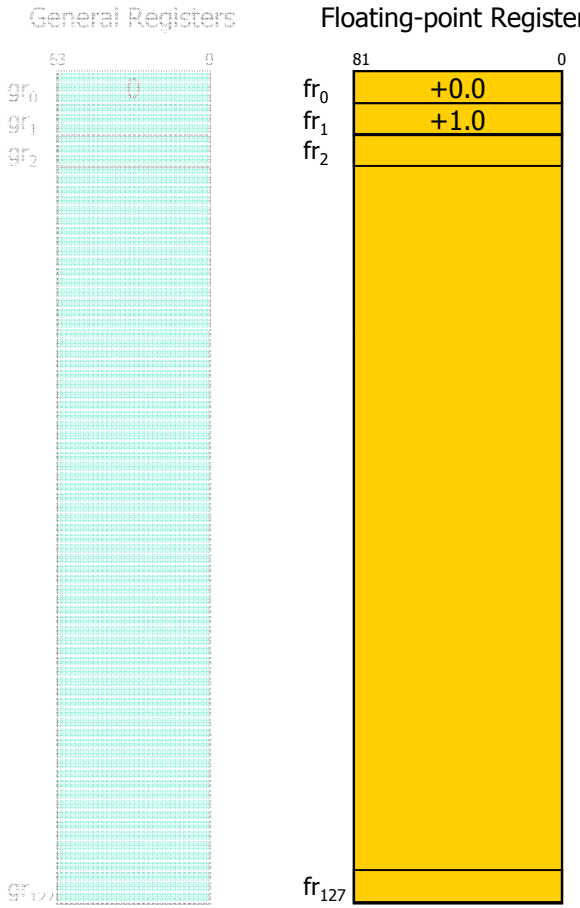


Thread Switching Overhead



- gr_0 fixed to zero
- On thread switch
  - Static registers must be saved explicitly
  - Stacked registers handled by register stack engine (RSE)
- "Only" 2.5 kB of register contents left

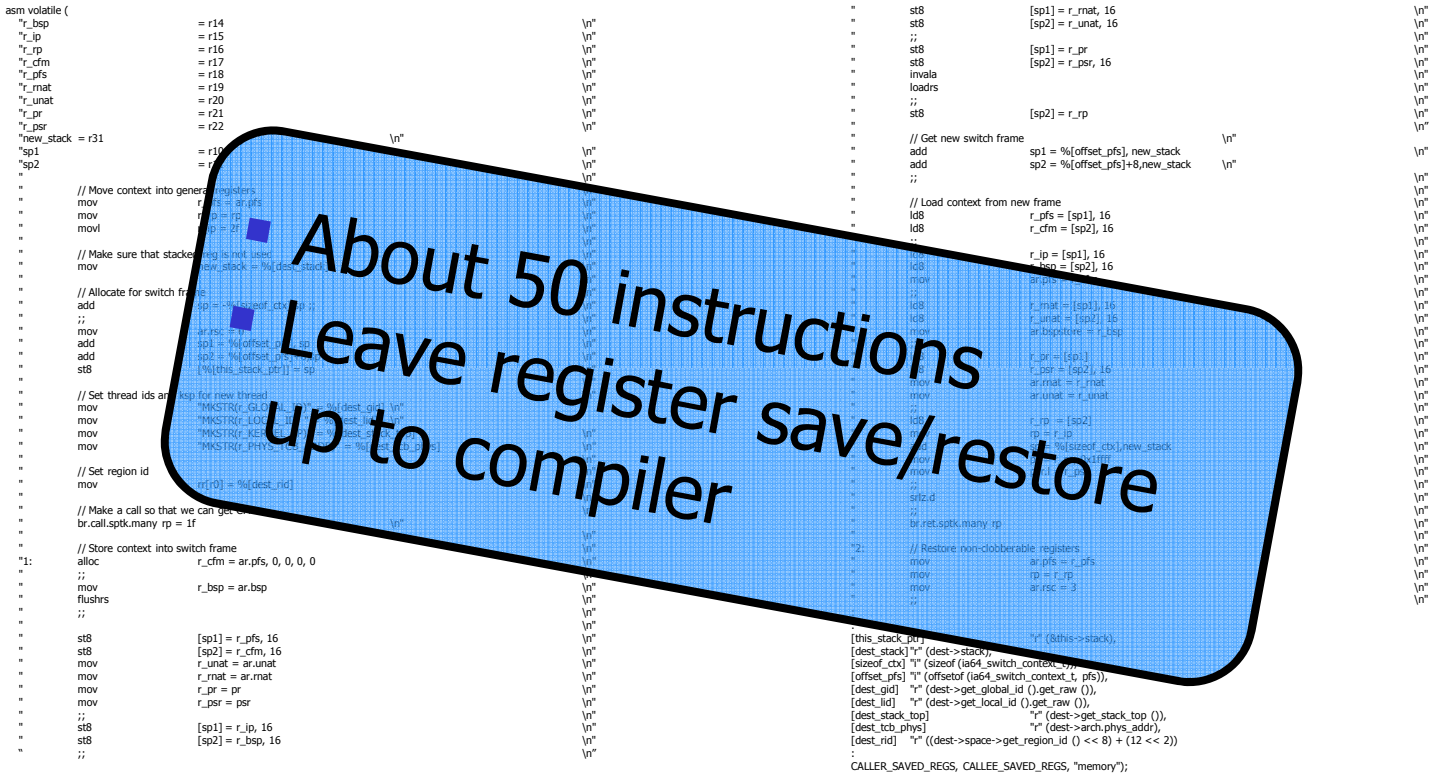
Thread Switching Overhead



- fr_0 and fr_1 fixed
- Remaining floating-point registers can be handled lazily
- Only 28 (of 128) ARs implemented
- "Only" ~0.5 kB of register contents left

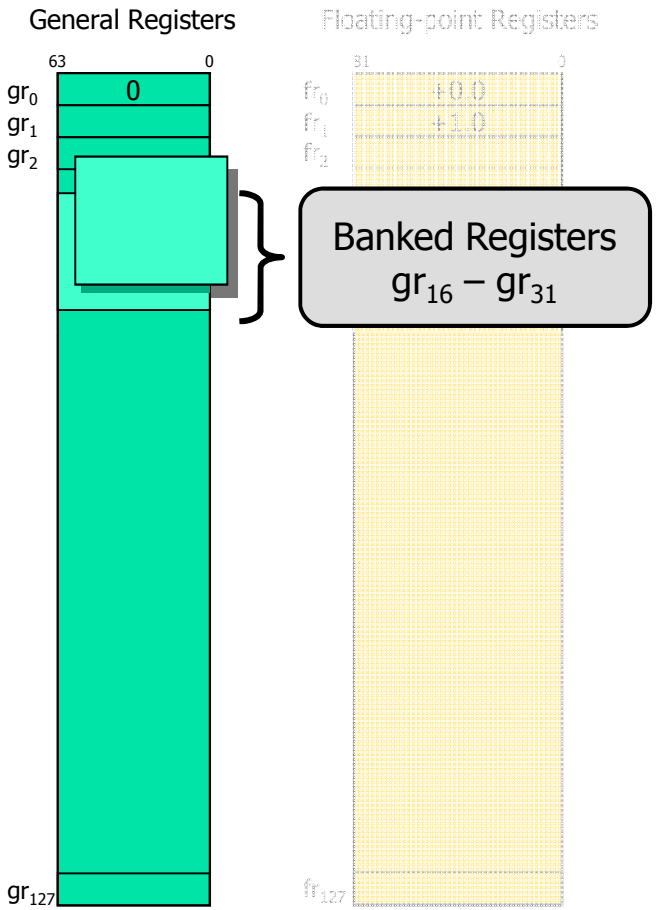
Thread Switch Example

[pistachio/kernel/include/glue/v4-ia64/tcb.h]



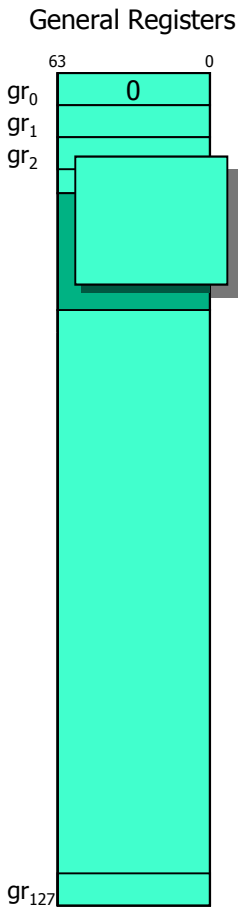
- About 50 instructions
- Leave register save/restore up to compiler

Exception Handling



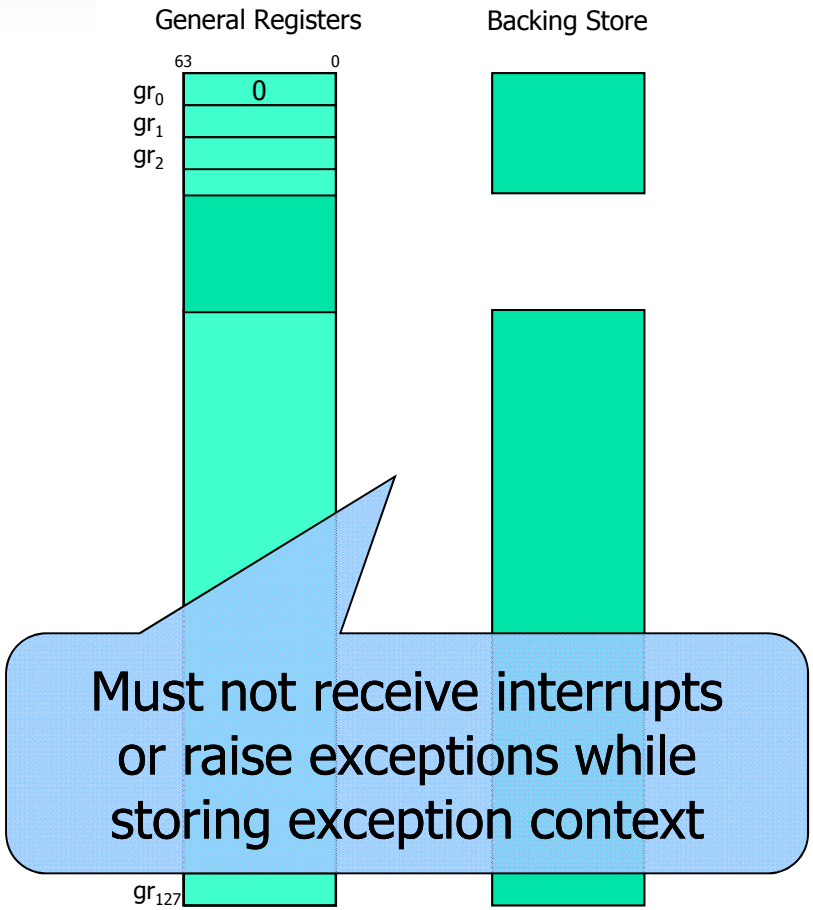
- Bank 1 used normally
- Automatic switch to bank 0 on exceptions
  - Frees up registers for storing context
- Can switch manually

Exception Handling

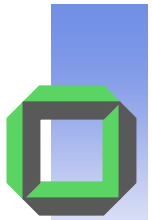


- Run on bank 1
- Exception
  - Switches to bank 0
- Store other registers

Exception Handling

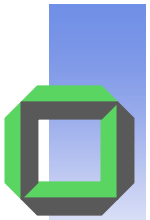


- Run on bank 1
- Exception
  - Switches to bank 0
- Store other registers
- Switch to bank 1
- Store remaining registers

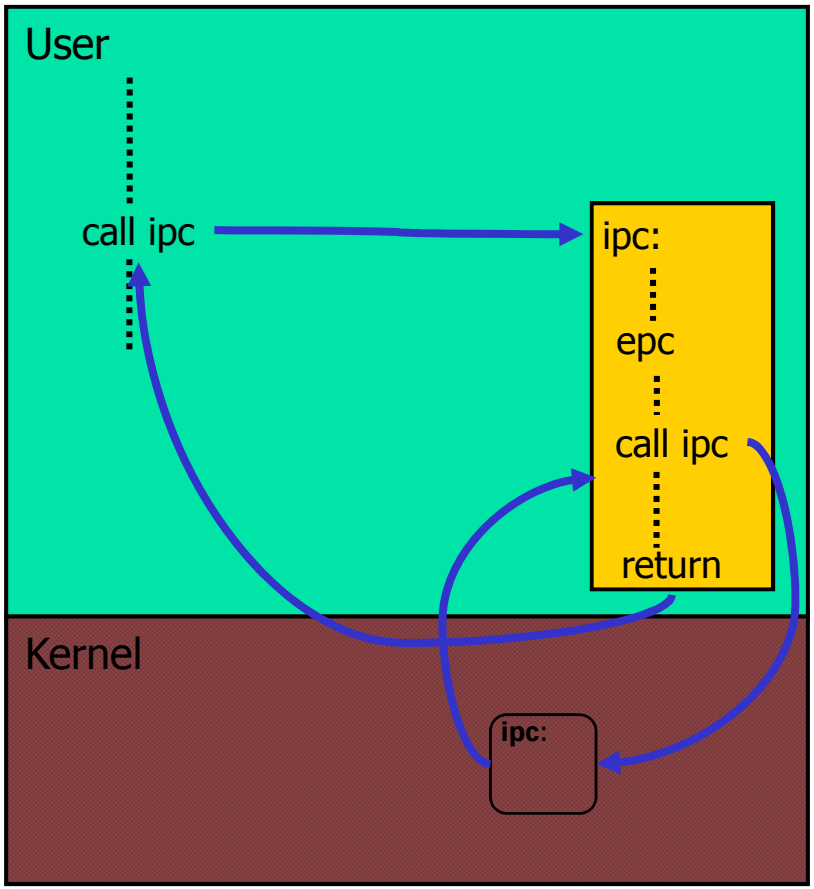


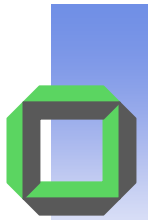
Kernel Entry

- Kernel entry by exception is slow
  - Must flush instruction pipeline
- IA-64 provides an epc instruction
  - Raises privileges to kernel mode
  - Continues execution on next instruction
  - Can only be executed in special regions of virtual memory



System Call Trampoline





System Call Trampoline for IA-32

- Is it useful?
  - Yes! Kernel dictates trampoline code
    - Kernel can offer best kernel entry method
      (e.g., sysenter/sysexit if supported by CPU)
  - Some system calls can execute entirely in user-mode
      (e.g., reading current time)
- Used in L4Ka::Pistachio
- Other systems are starting to use system call trampolines (e.g., Linux 2.6)