Karlsruhe Institute of Technology

Microkernel Construction
I.6 – Dispatching

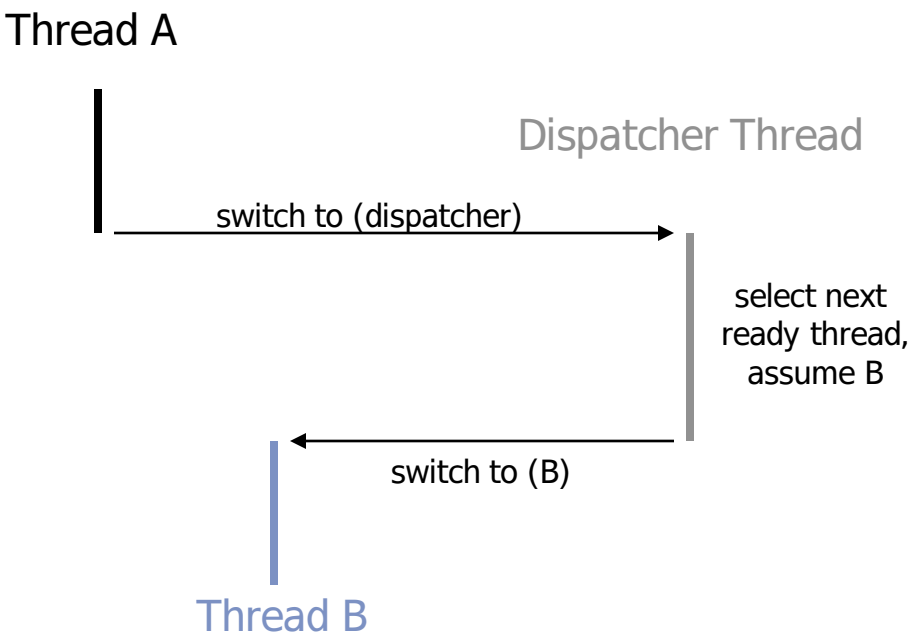
Lecture Summer Term 2017
Wednesday 15:45-17:15 R 131, 50.34 (INFO)

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

Dispatching Topics

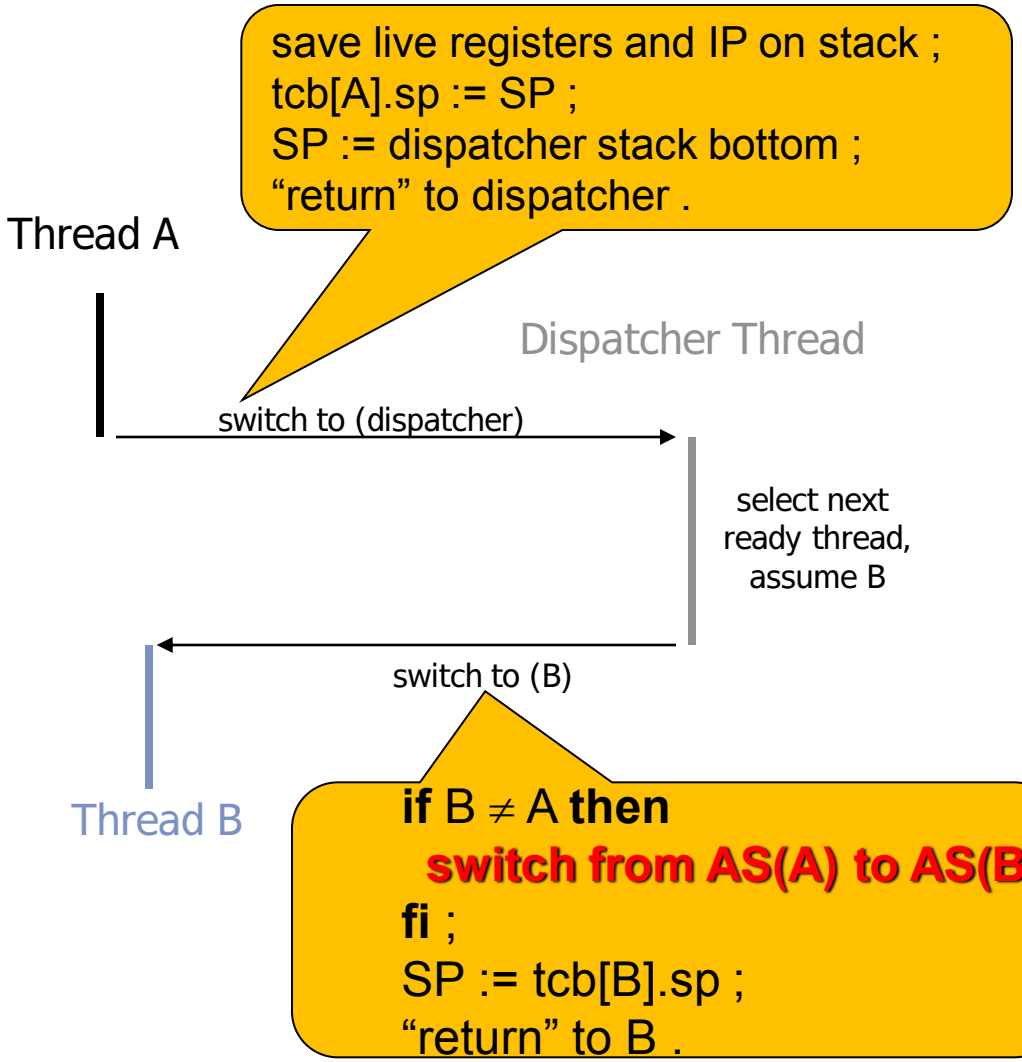
- Thread switch
  - To a specific thread
  - To next thread to be scheduled
    - To nil
    - Implicitly, when IPC blocks
- Priorities
- Preemption
  - Time slices
  - Wakeups, interruptions
- Timeouts and wakeups
  - Time

Switch to ()



- Dispatcher is preemptible
  - "Clean" model
  - Improved interrupt latency if dispatching is time consuming
- Smaller stack per thread

Switch to ()



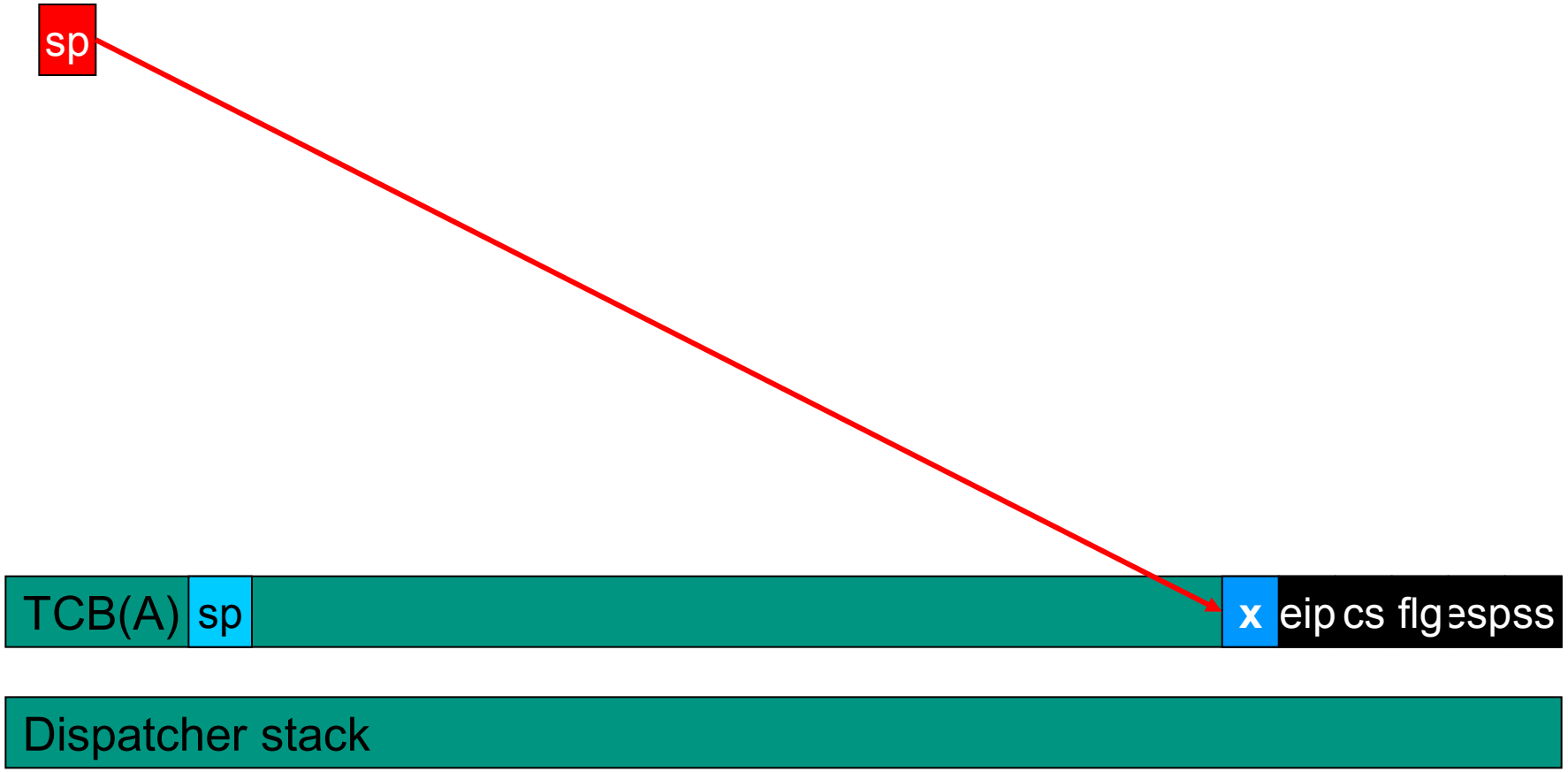
- Dispatcher thread is special
  - No user mode
    - No own AS, hence no AS switch required
    - No ID required
    - Freedom from TCB layout conventions
  - Almost stateless (see *priorities)*
    - No need to preserve internal state between invocations
    - External state must be consistent
  - costs (A → B)
  - ≈ costs (A → disp → B)
    - costs (select next)
  - costs (A → disp → A) are low

Why ??

Example: Simple Dispatch

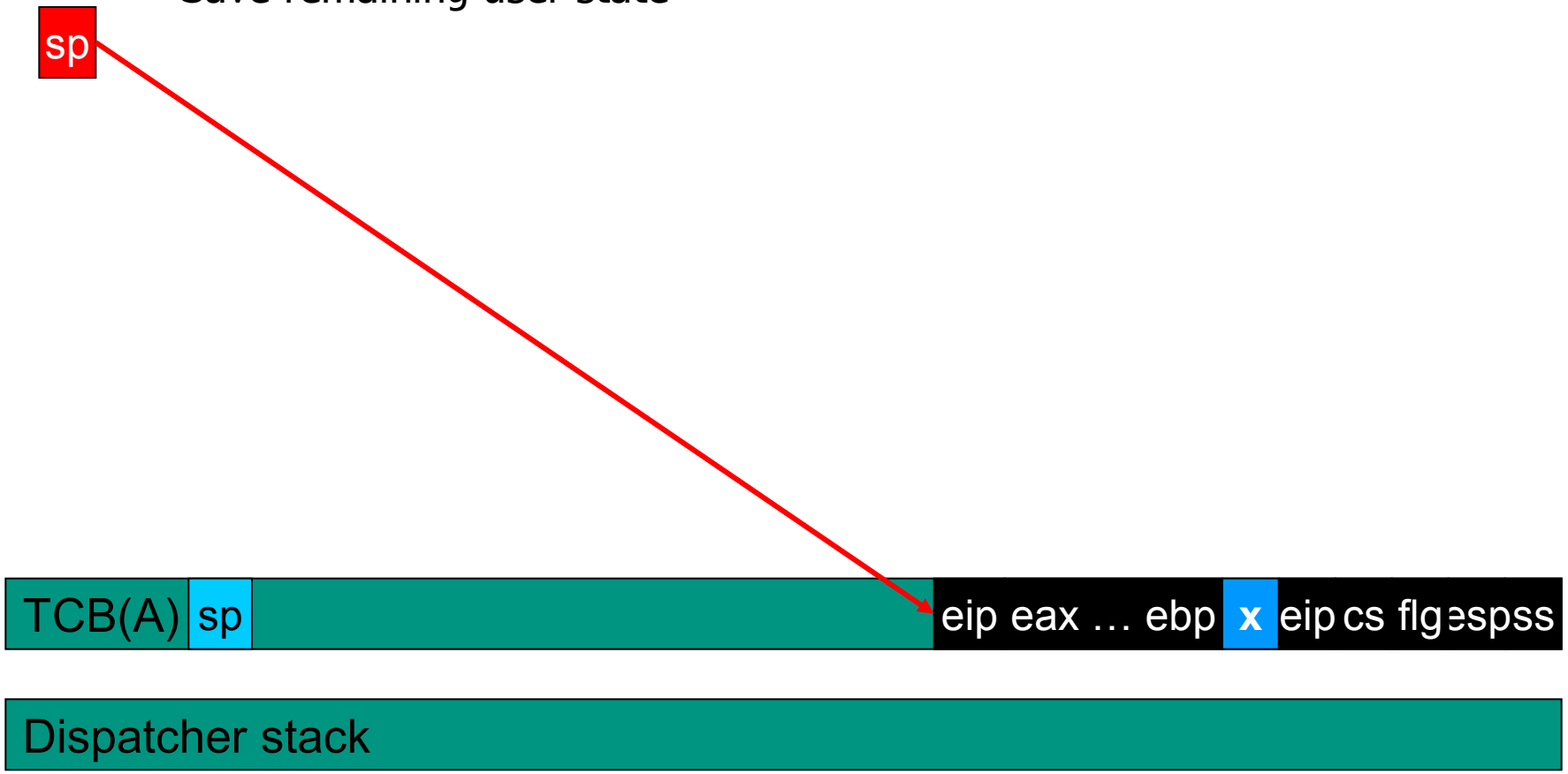
Example: Simple Dispatch

- Enter the kernel, save some user state (in HW)



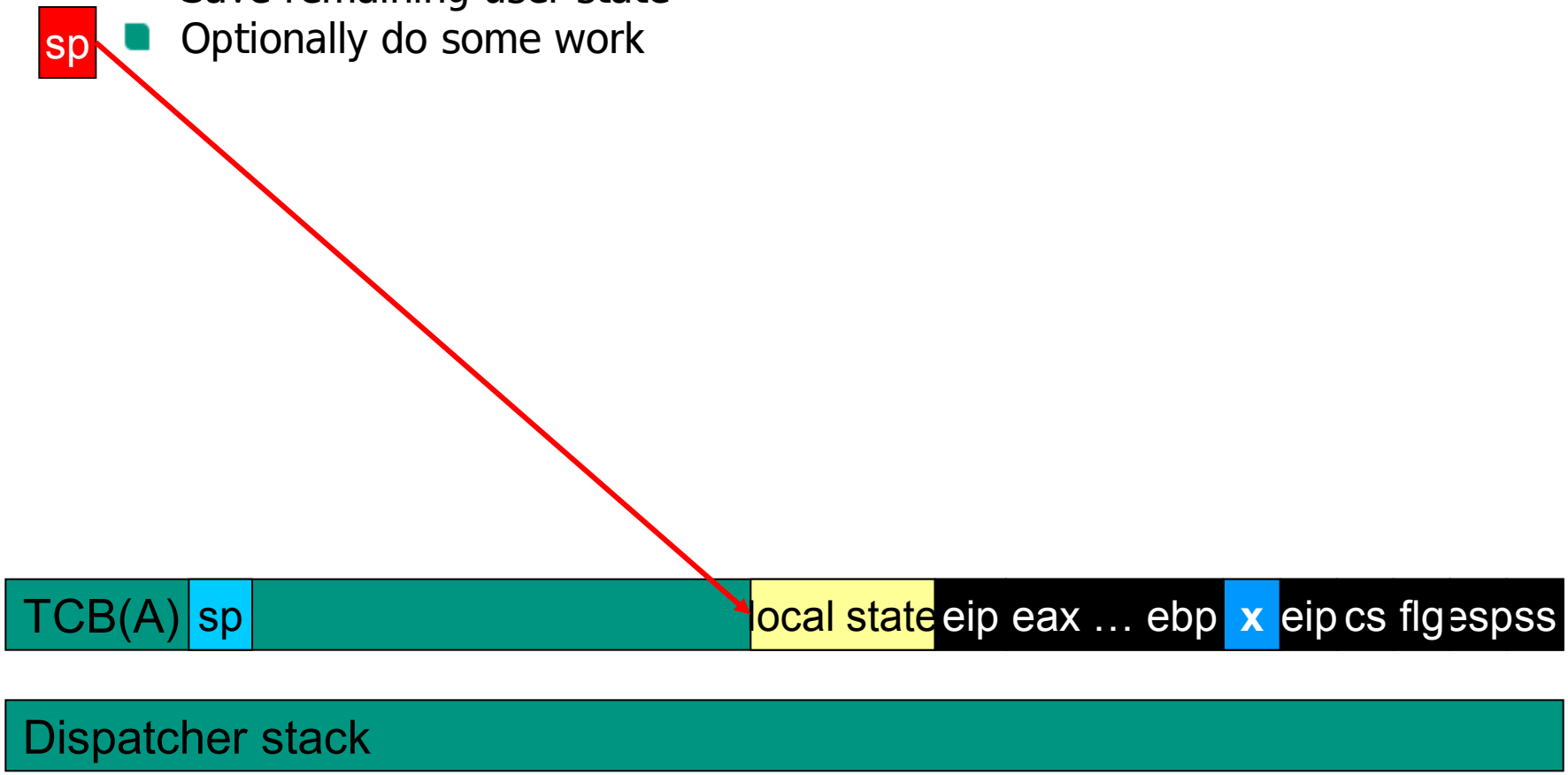
Example: Simple Dispatch

- Enter the kernel, save some user state (in HW)
- Save remaining user state



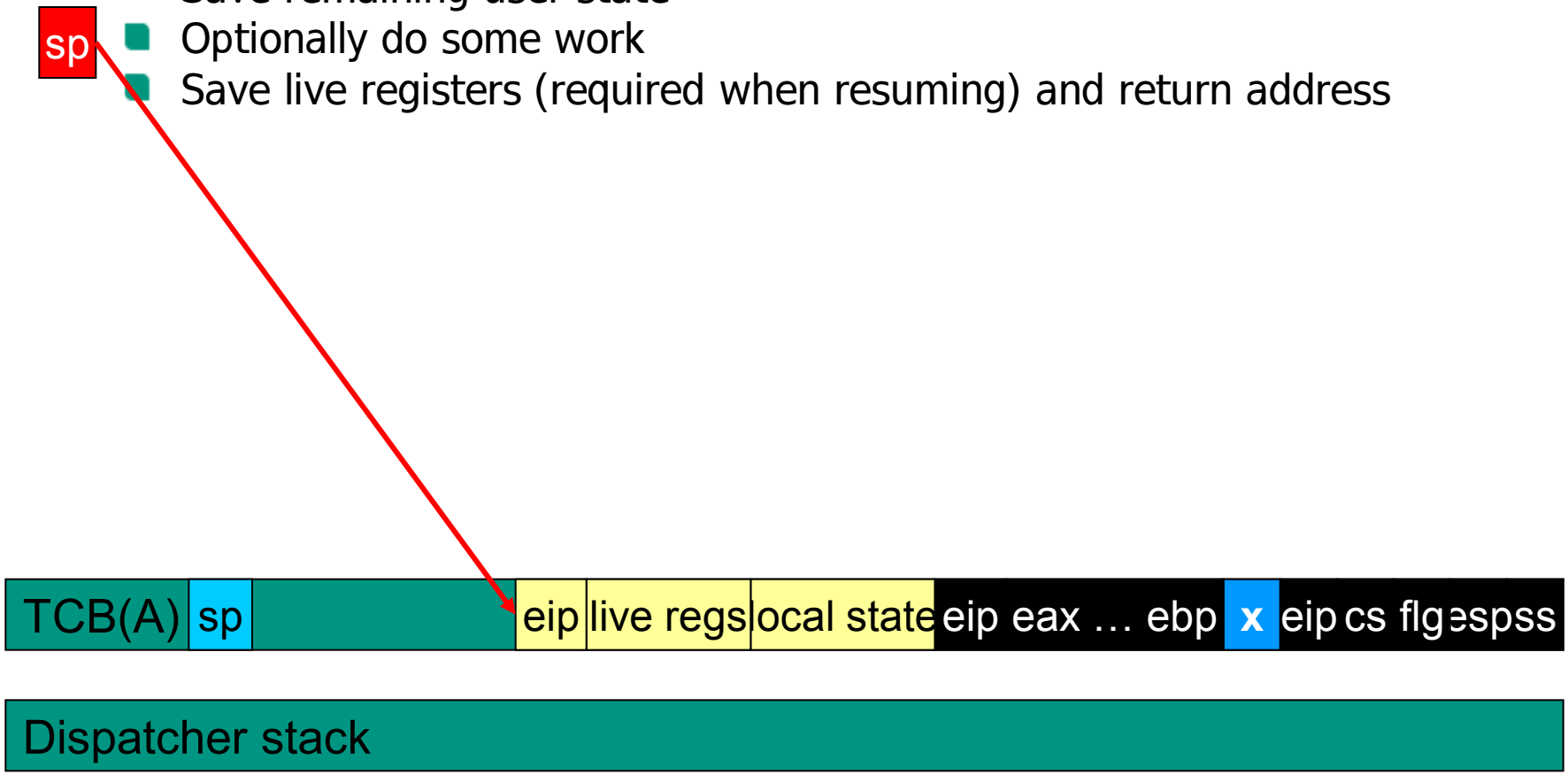
Example: Simple Dispatch

- Enter the kernel, save some user state (in HW)
- Save remaining user state
- Optionally do some work



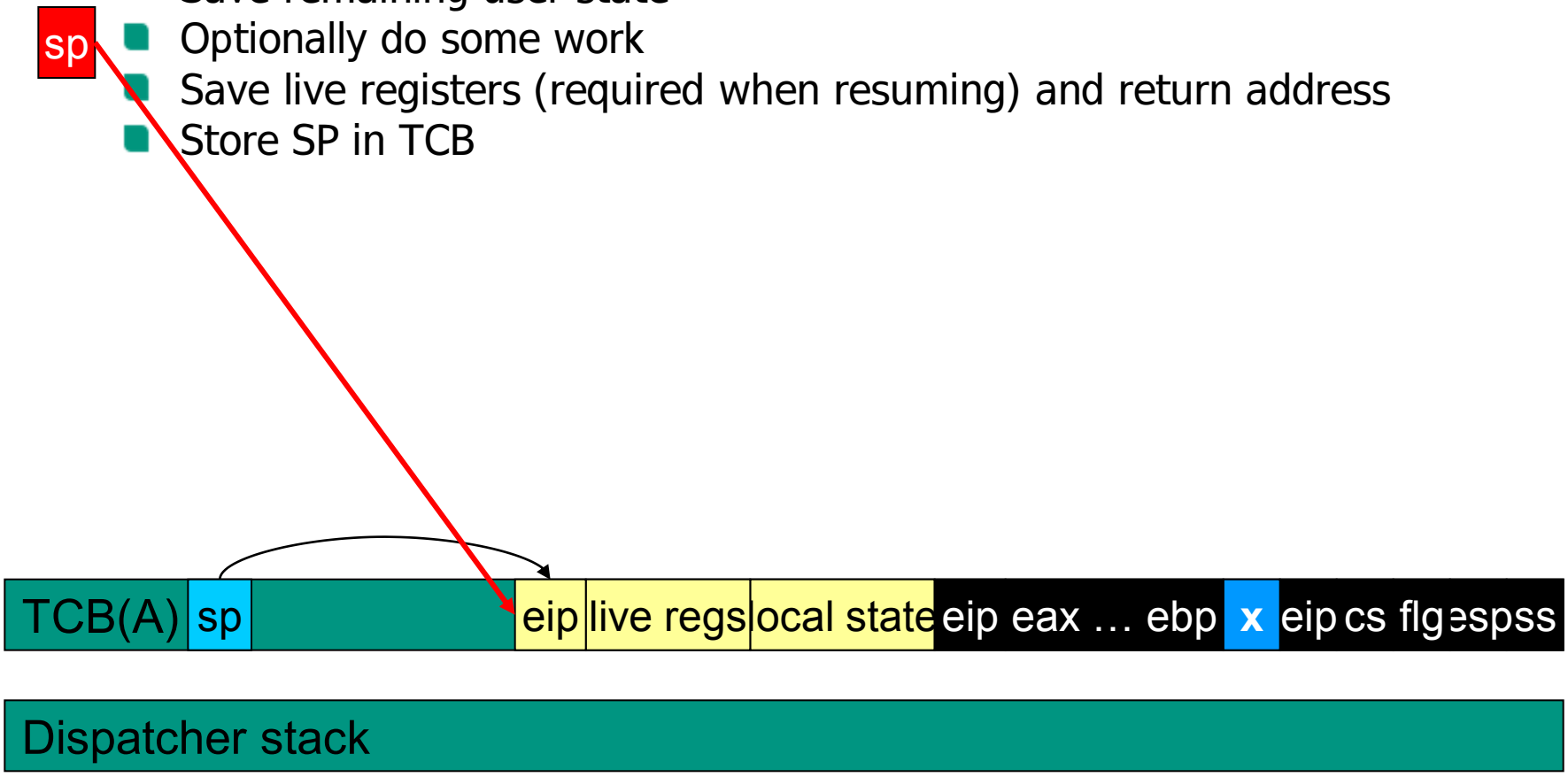
Example: Simple Dispatch

- Enter the kernel, save some user state (in HW)
- Save remaining user state
- Optionally do some work
- Save live registers (required when resuming) and return address



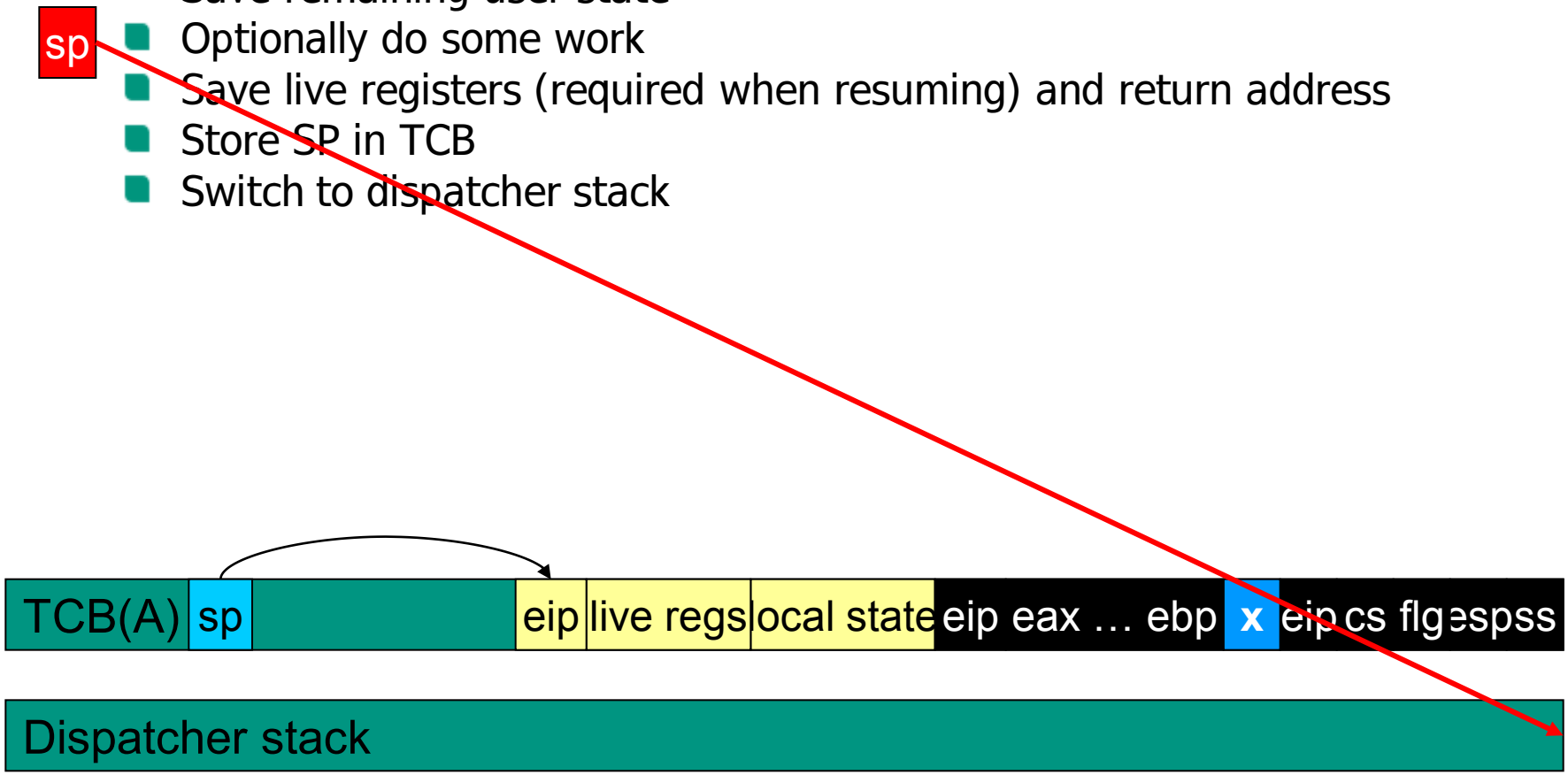
Example: Simple Dispatch

- Enter the kernel, save some user state (in HW)
- Save remaining user state
- Optionally do some work
- Save live registers (required when resuming) and return address
- Store SP in TCB



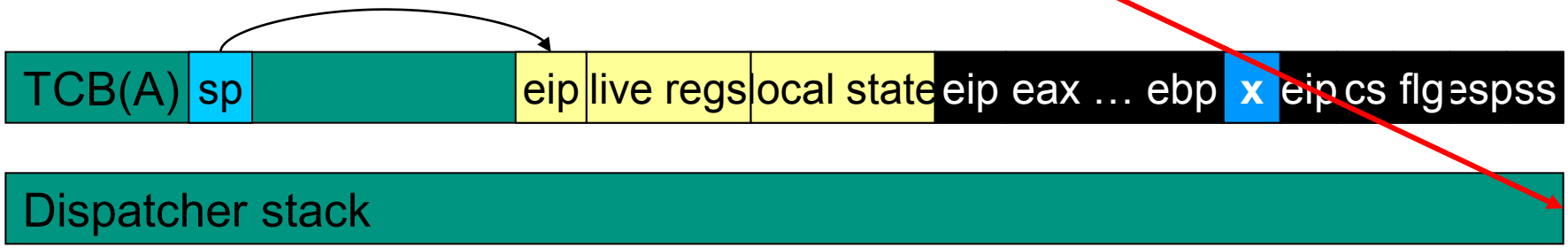
Example: Simple Dispatch

- Enter the kernel, save some user state (in HW)
- Save remaining user state
- Optionally do some work
- Save live registers (required when resuming) and return address
- Store SP in TCB
- Switch to dispatcher stack



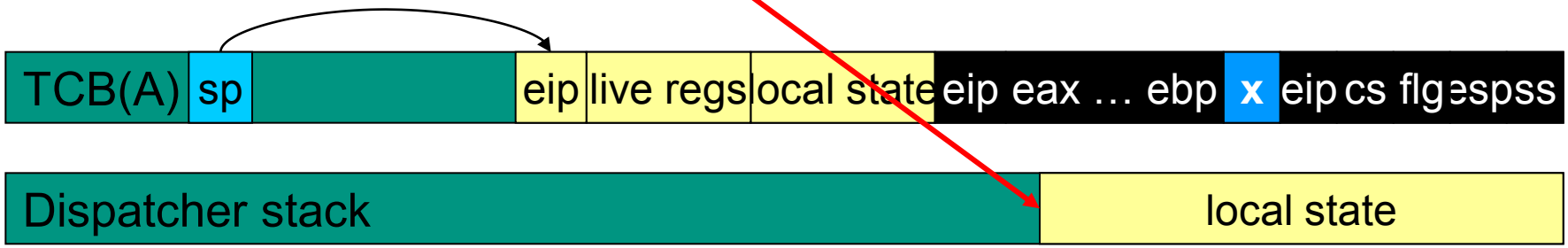
Example: Simple Dispatch

- Enter the kernel, save some user state (in HW)
- Save remaining user state
- Optionally do some work
- Save live registers (required when resuming) and return address
- Store SP in TCB
- Switch to dispatcher stack
- Jump to dispatcher code



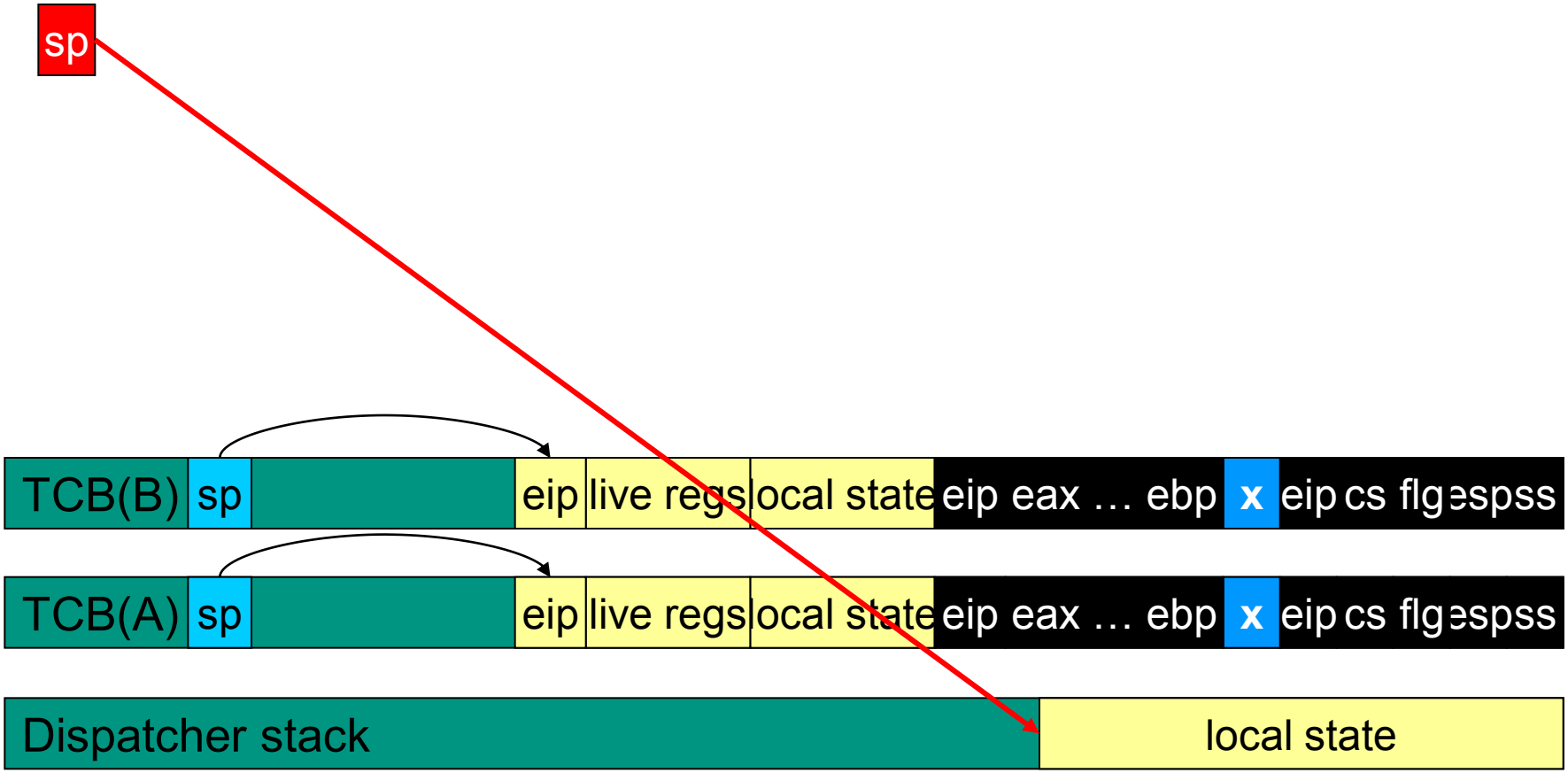
Example: Simple Dispatch

- Enter the kernel, save some user state (in HW)
- Save remaining user state
- Optionally do some work
- Save live registers (required when resuming) and return address
- Store SP in TCB
- Switch to dispatcher stack
- Jump to dispatcher code
  - Select next thread to run



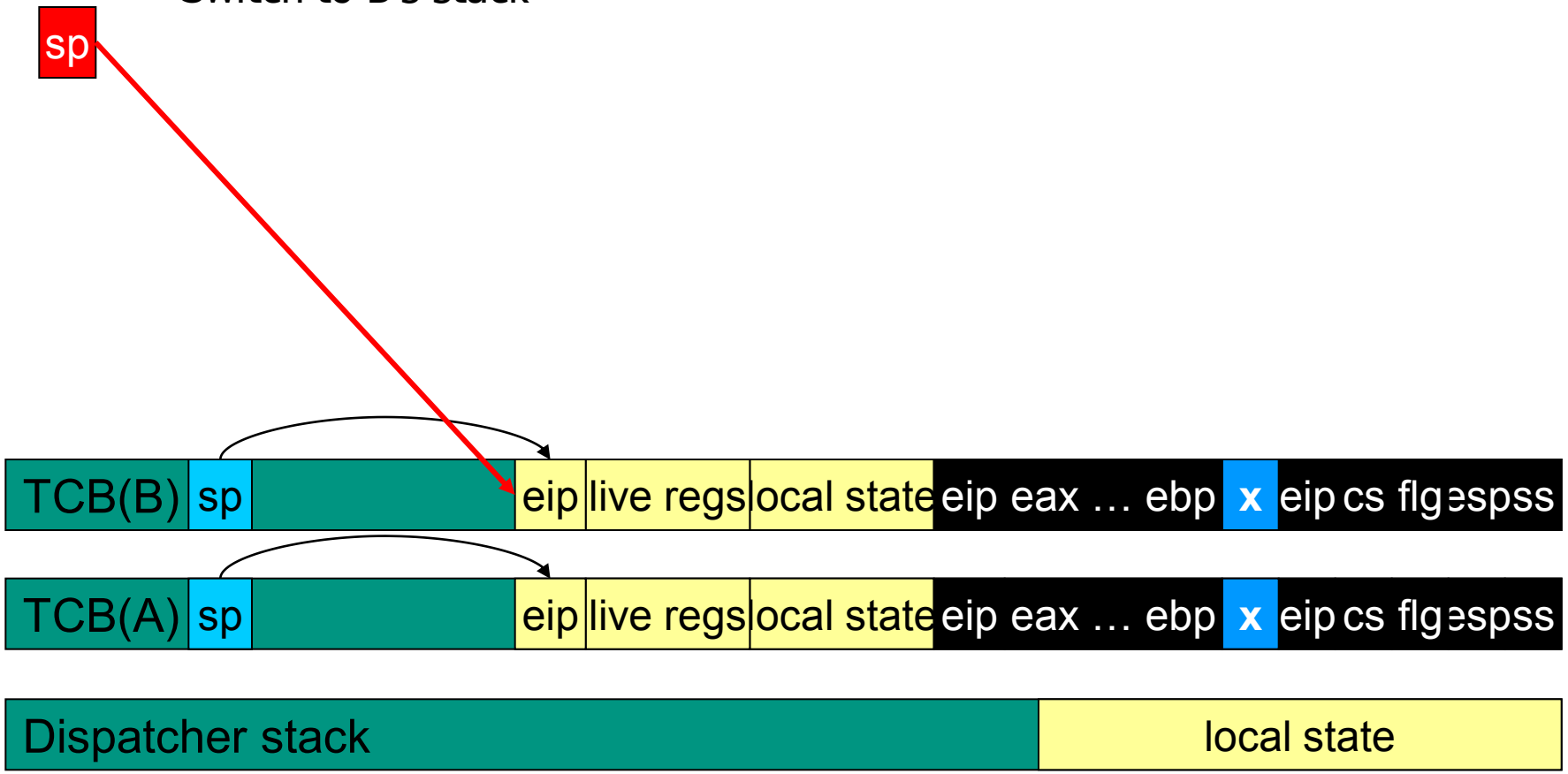
Example: Simple Dispatch (cont'd)

- Dispatcher selected next thread to run (B)



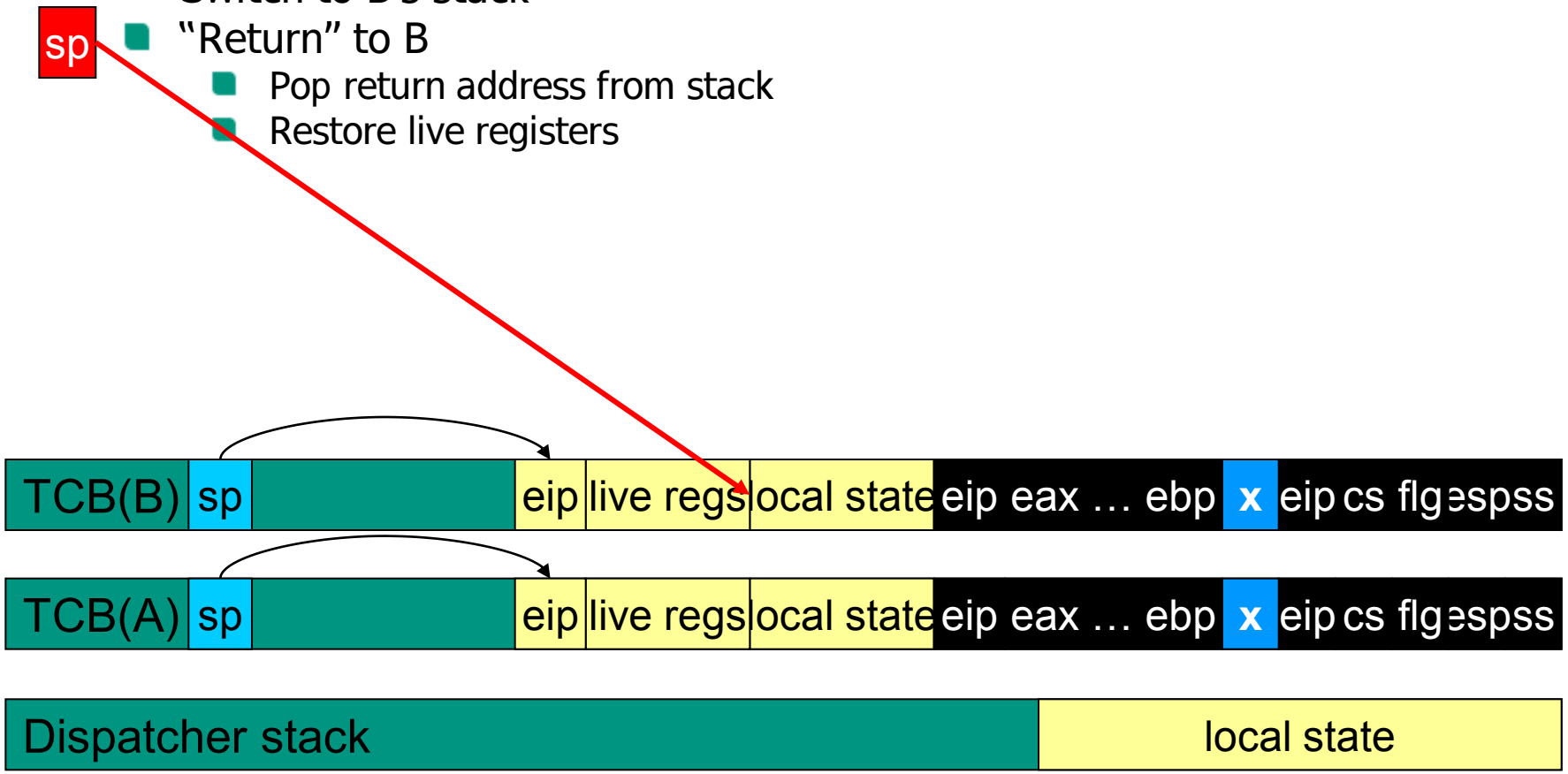
Example: Simple Dispatch (cont'd)

- Dispatcher selected next thread to run (B)
- Switch to B's stack



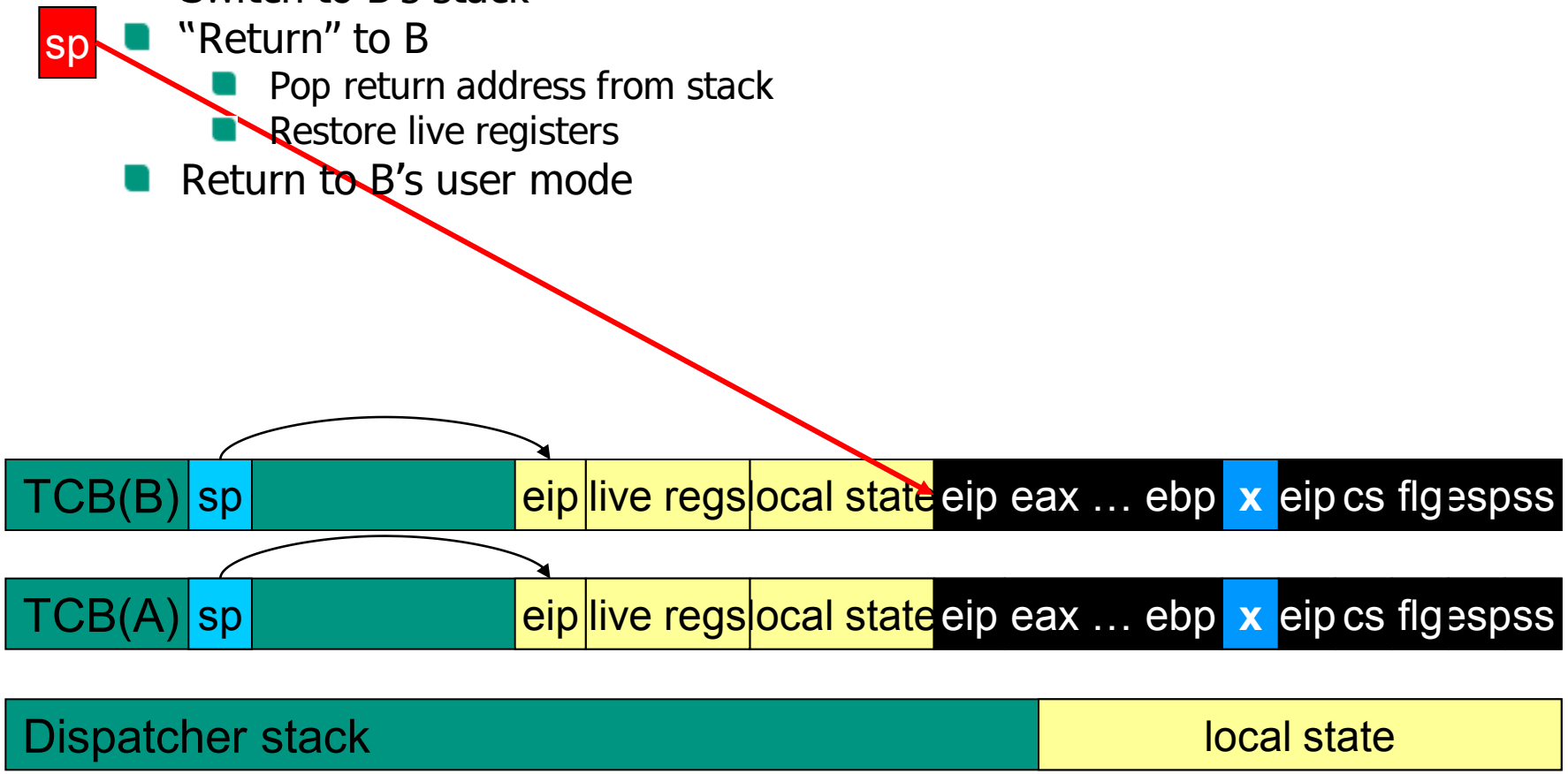
Example: Simple Dispatch (cont'd)

- Dispatcher selected next thread to run (B)
- Switch to B's stack
- "Return" to B
  - Pop return address from stack
  - Restore live registers



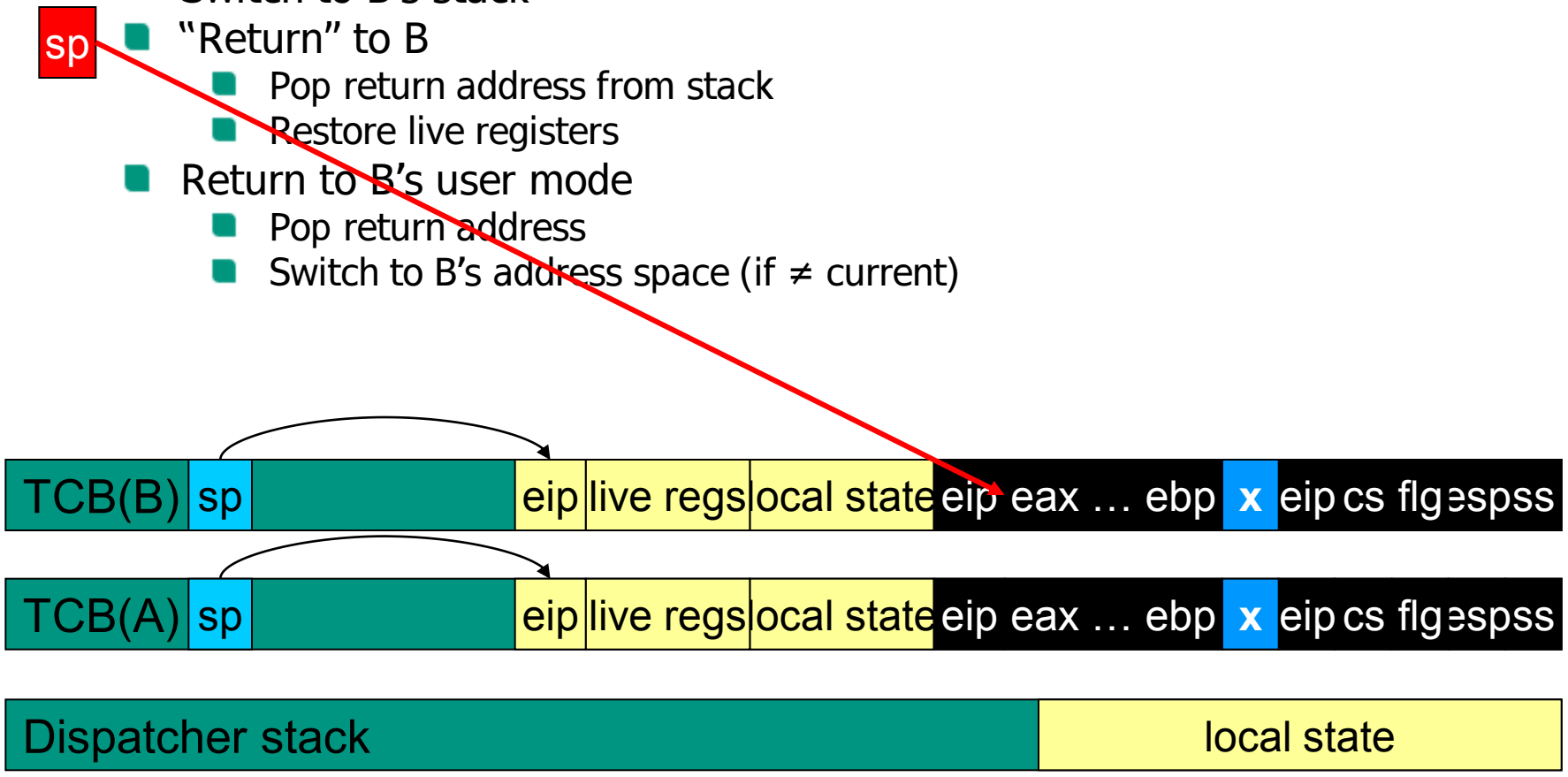
Example: Simple Dispatch (cont'd)

- Dispatcher selected next thread to run (B)
- Switch to B's stack
- "Return" to B
  - Pop return address from stack
  - Restore live registers
- Return to B's user mode



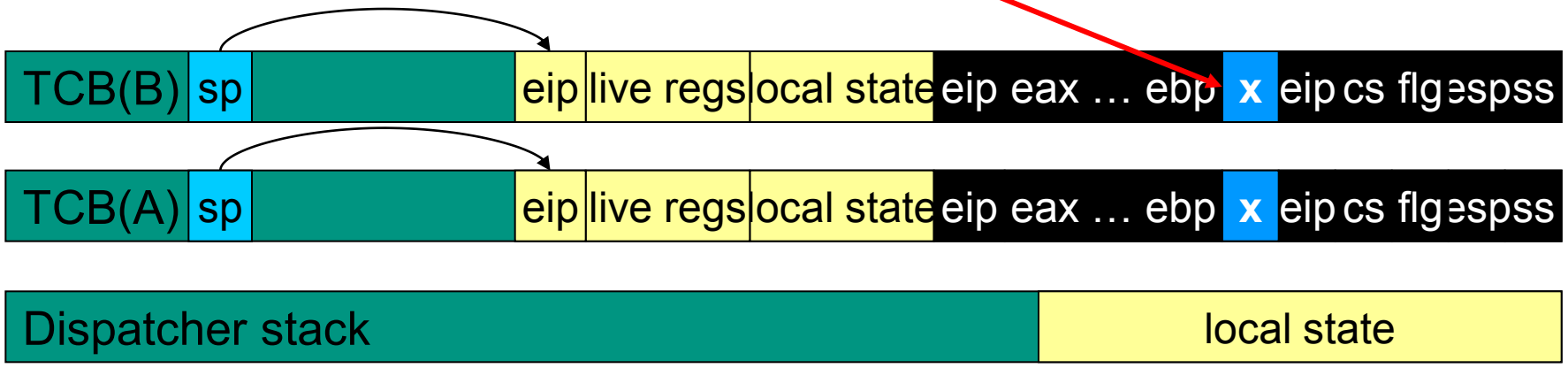
Example: Simple Dispatch (cont'd)

- Dispatcher selected next thread to run (B)
- Switch to B's stack
- "Return" to B
  - Pop return address from stack
  - Restore live registers
- Return to B's user mode
  - Pop return address
  - Switch to B's address space (if ≠ current)



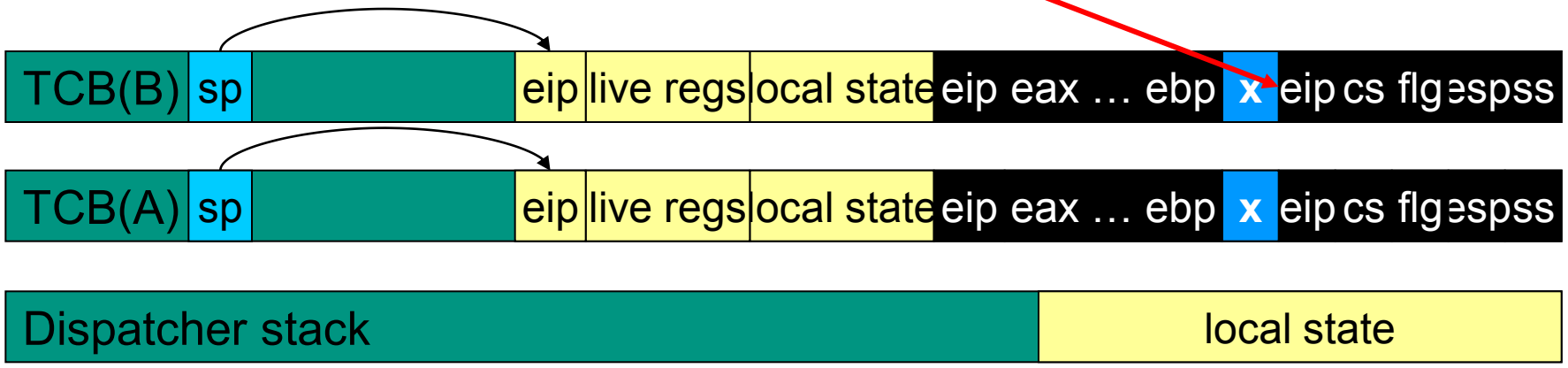
Example: Simple Dispatch (cont'd)

- Dispatcher selected next thread to run (B)
- Switch to B's stack
- "Return" to B
  - Pop return address from stack
  - Restore live registers
- Return to B's user mode
  - Pop return address
  - Switch to B's address space (if ≠ current)
  - Load user register contents



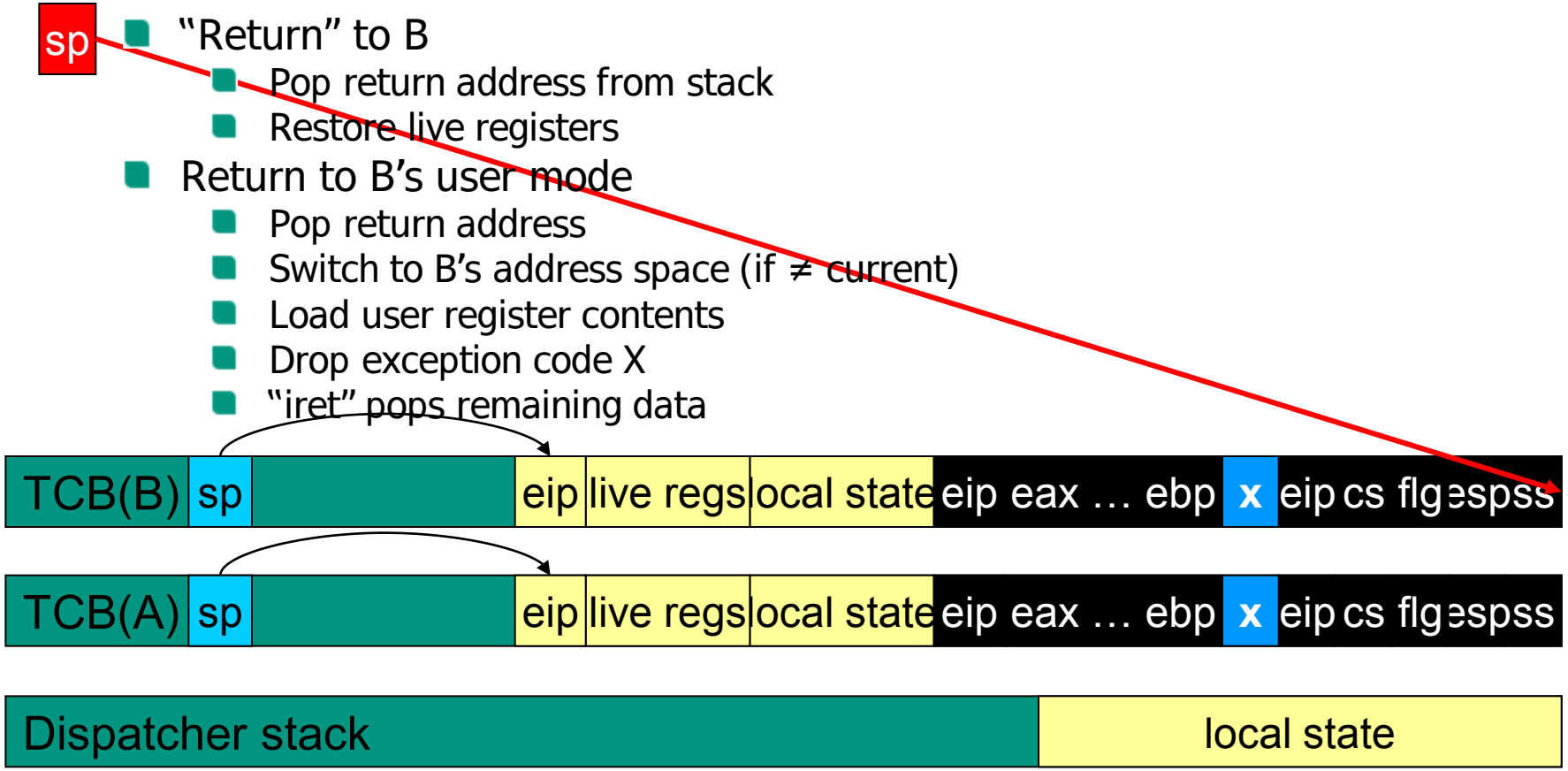
Example: Simple Dispatch (cont'd)

- Dispatcher selected next thread to run (B)
- Switch to B's stack
- "Return" to B
  - Pop return address from stack
  - Restore live registers
- Return to B's user mode
  - Pop return address
  - Switch to B's address space (if ≠ current)
  - Load user register contents
  - Drop exception code X



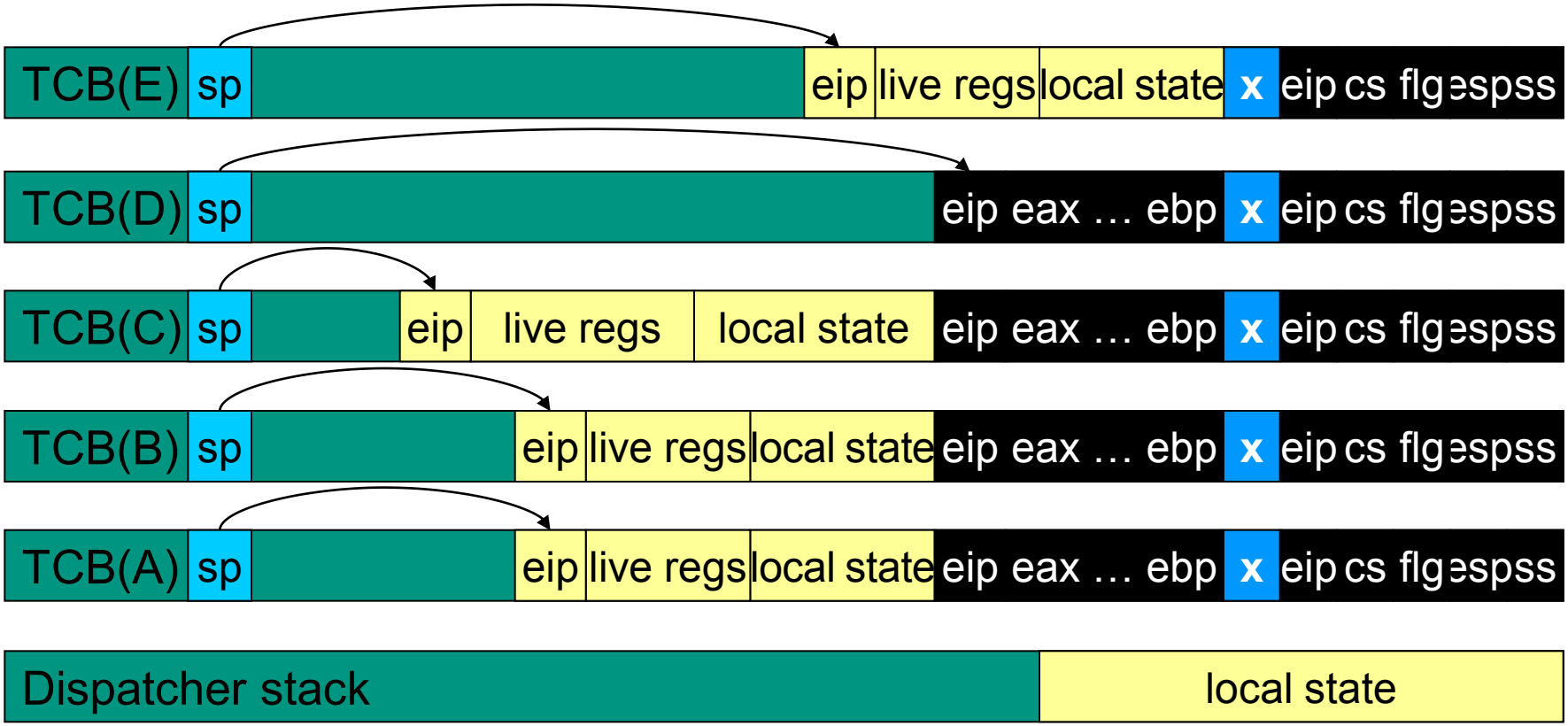
Example: Simple Dispatch (cont'd)

- Dispatcher selected next thread to run (B)
- Switch to B's stack
- "Return" to B
  - Pop return address from stack
  - Restore live registers
- Return to B's user mode
  - Pop return address
  - Switch to B's address space (if ≠ current)
  - Load user register contents
  - Drop exception code X
  - "iret" pops remaining data



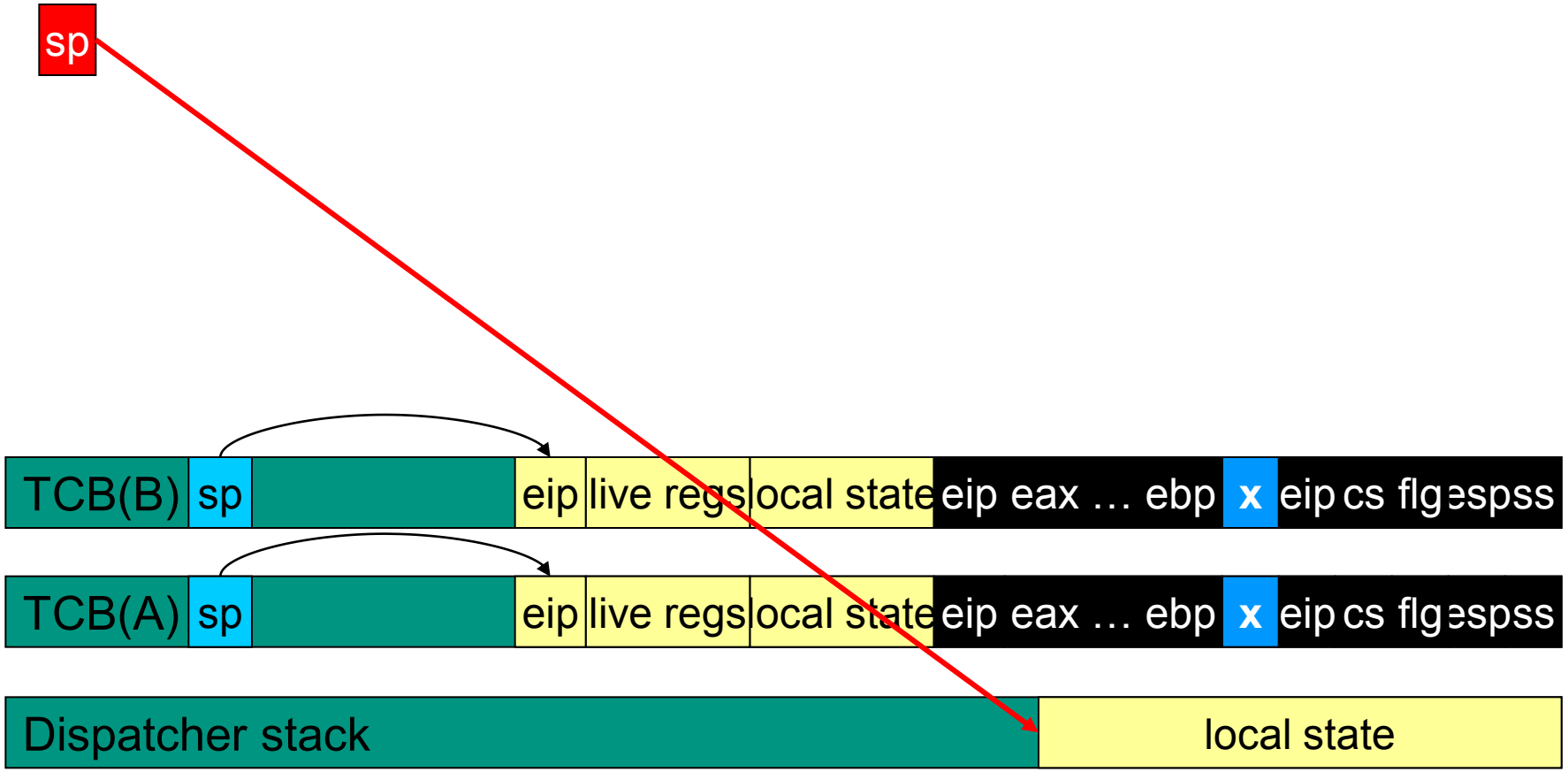
Example: Simple Dispatch (cont'd)

- Stack layout depends on cause of prior thread switch
  - Timer vs. device IRQ, IPC send vs. recv, yield(), …



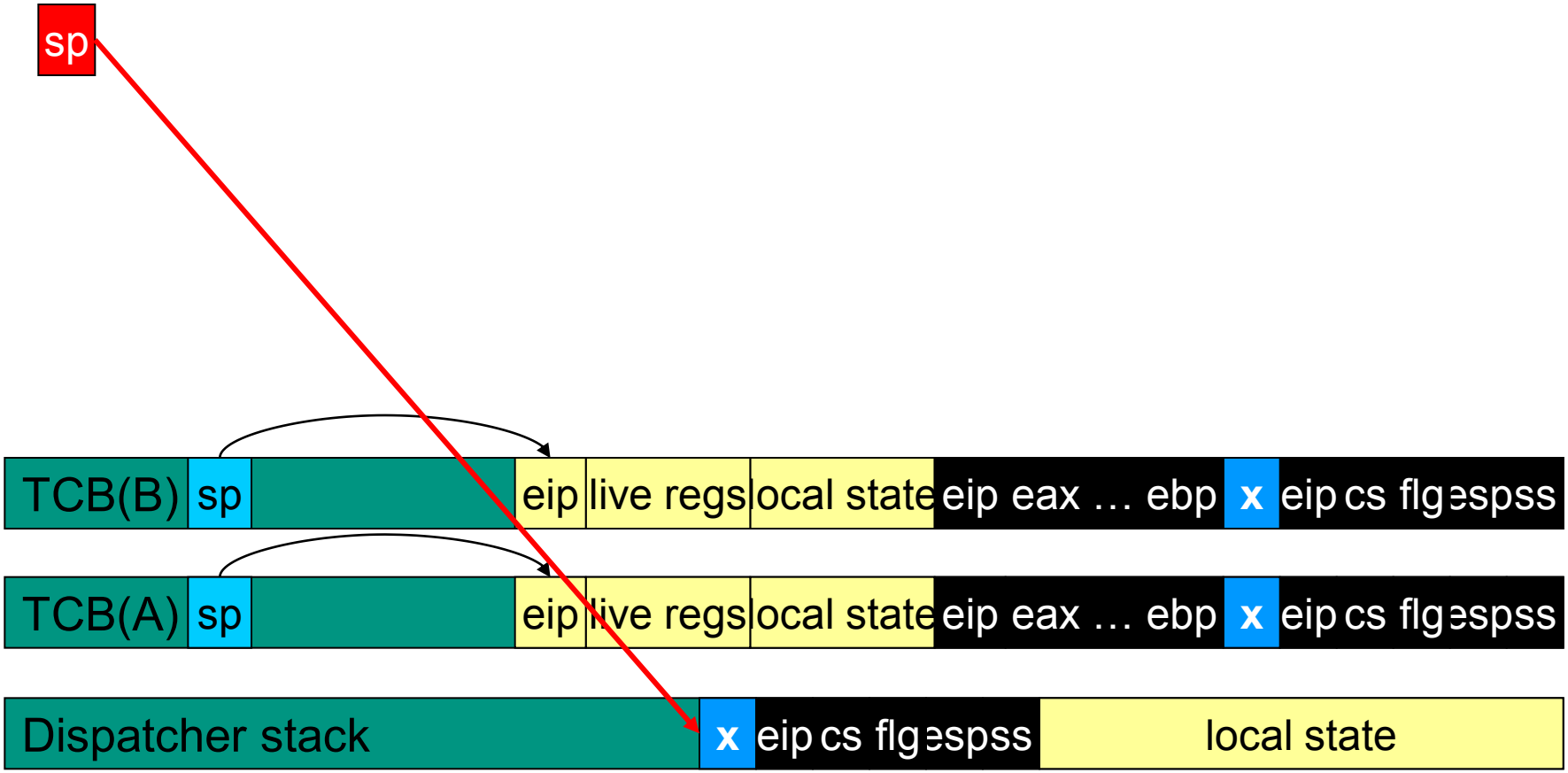
Example: Dispatch with `Tick'

- Dispatcher interrupted by timer IRQ



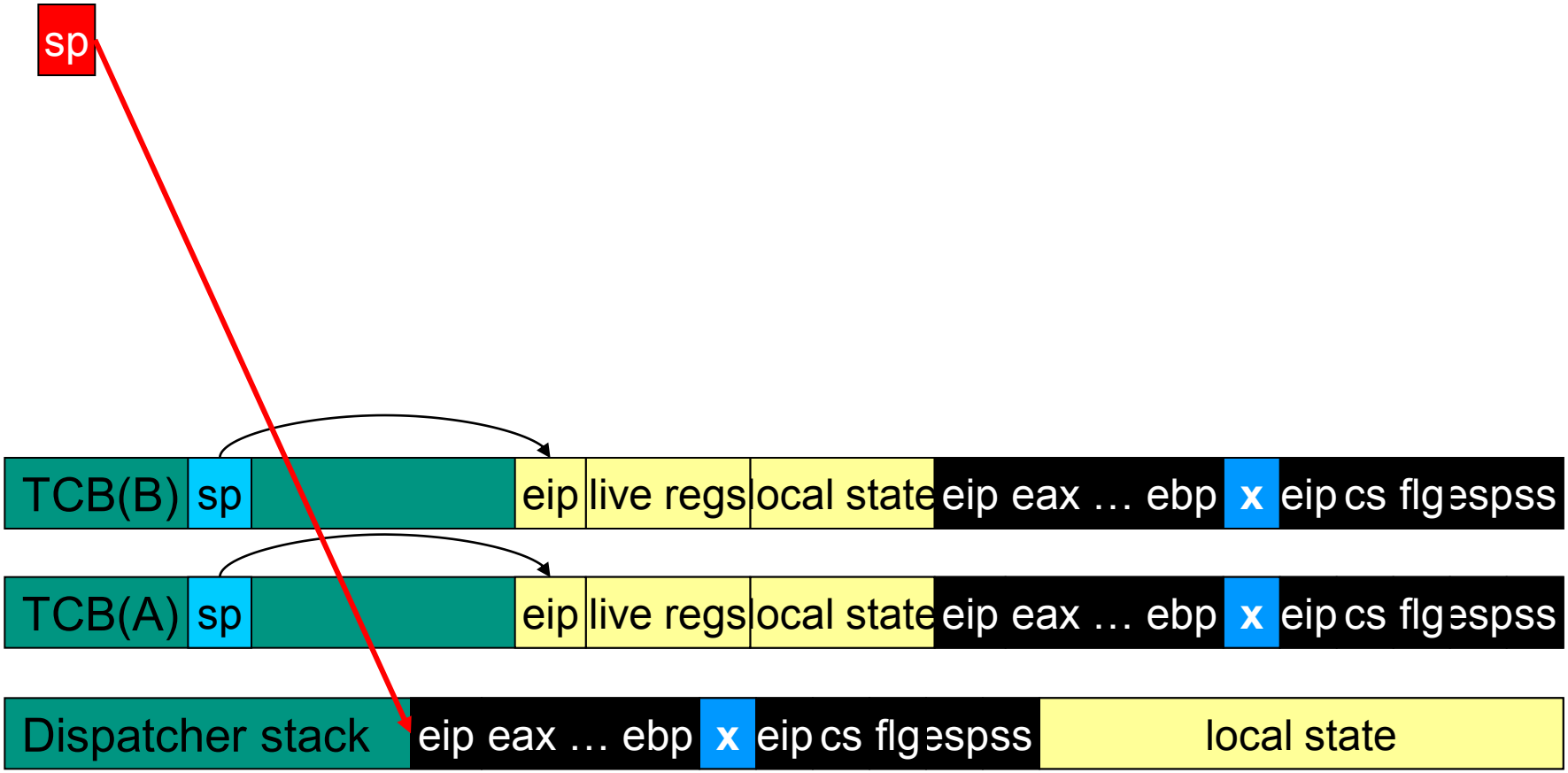
Example: Dispatch with `Tick'

- Dispatcher interrupted by timer IRQ



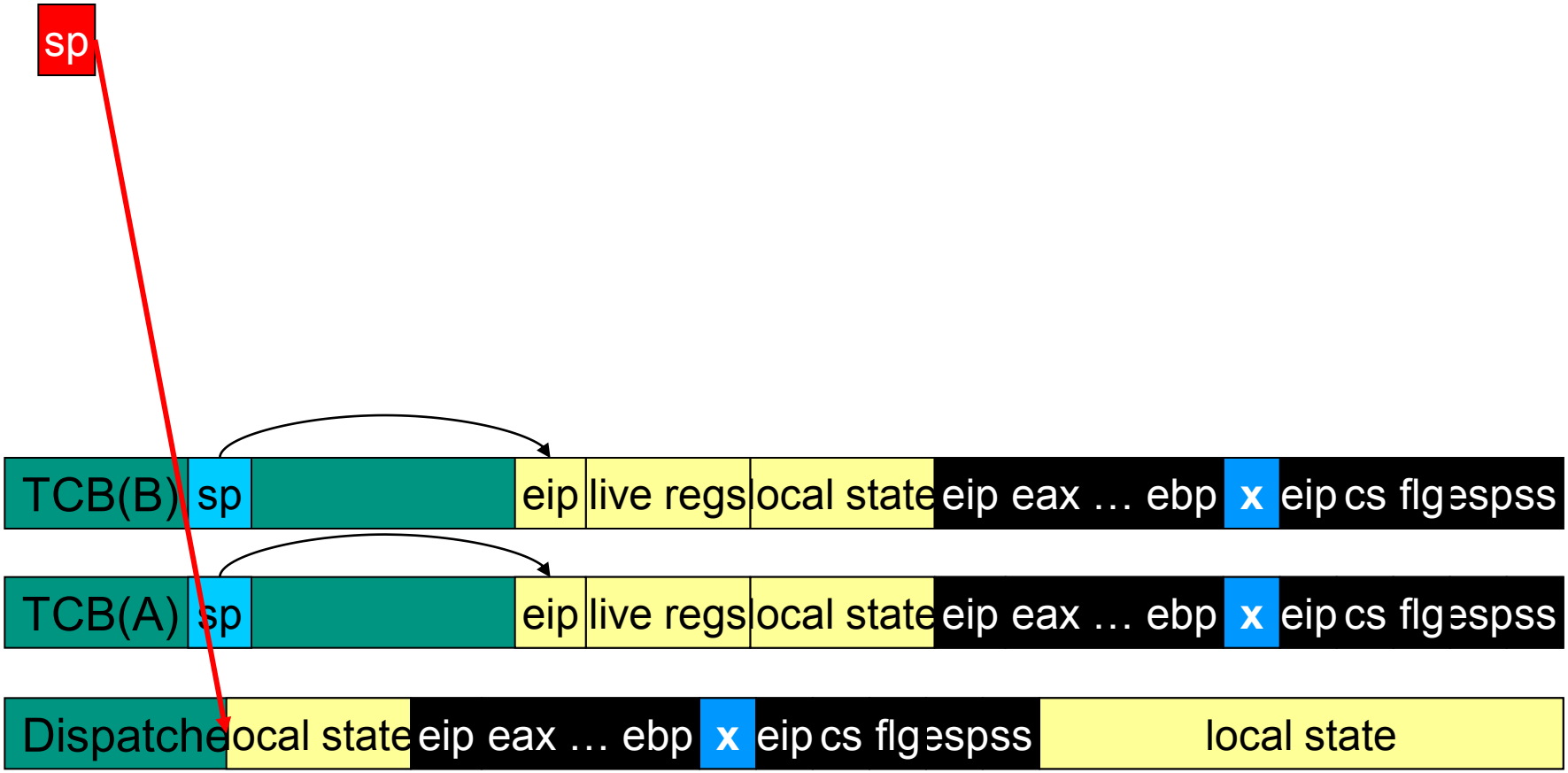
Example: Dispatch with `Tick'

- Dispatcher interrupted by timer IRQ



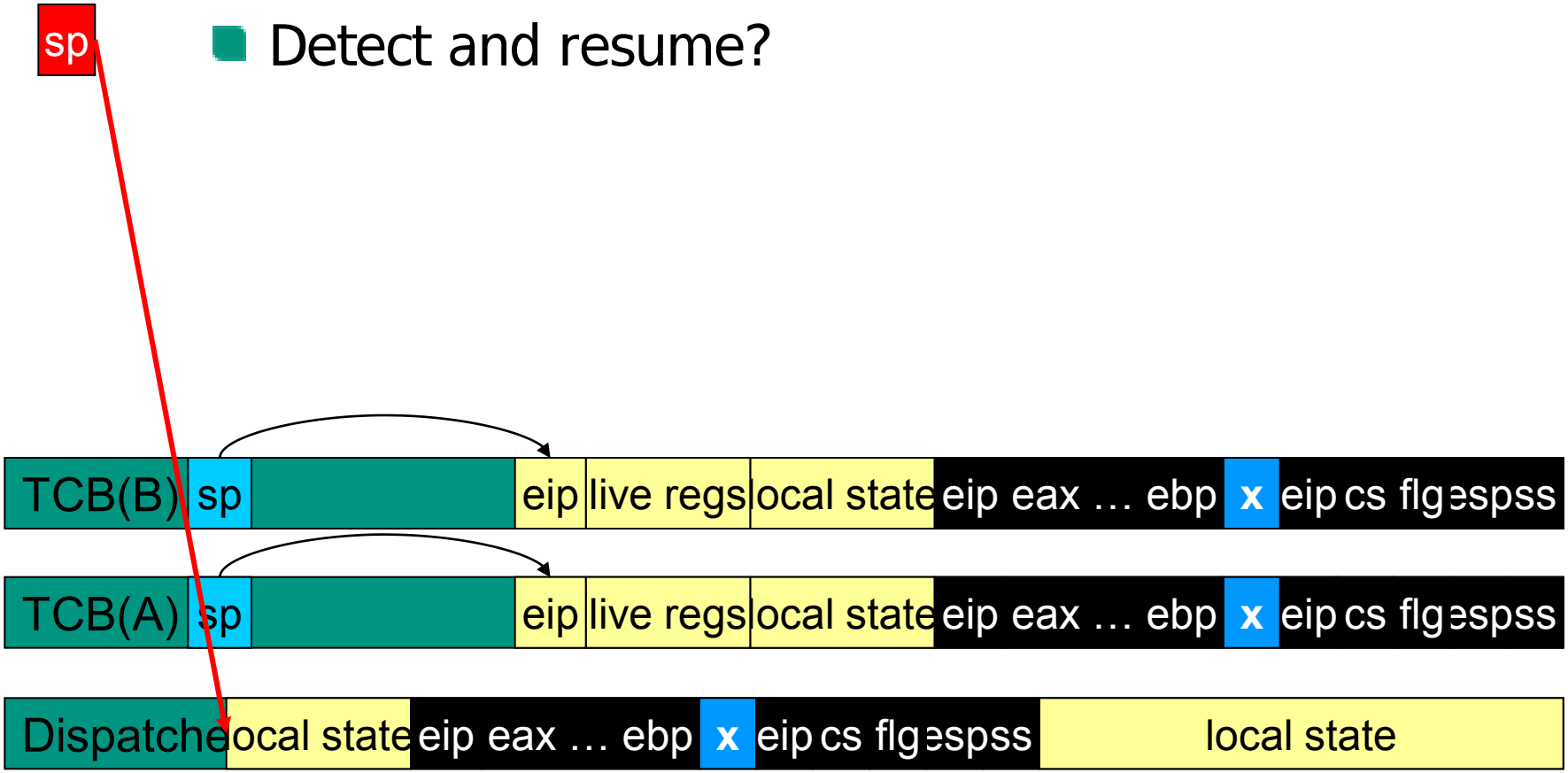
Example: Dispatch with `Tick'

- Dispatcher interrupted by timer IRQ



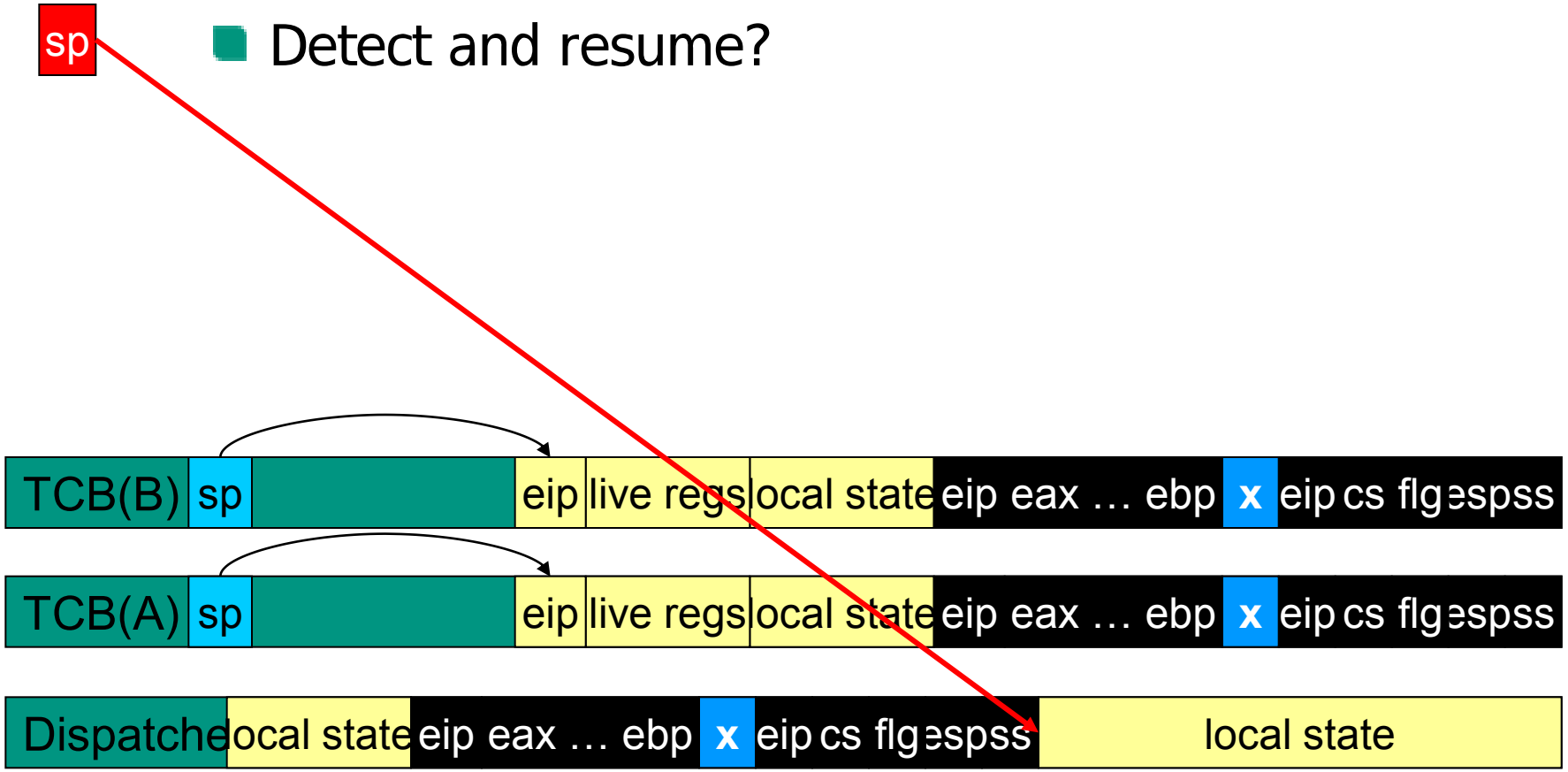
Example: Dispatch with `Tick'

- Dispatcher interrupted by timer IRQ
  - Detect and resume?



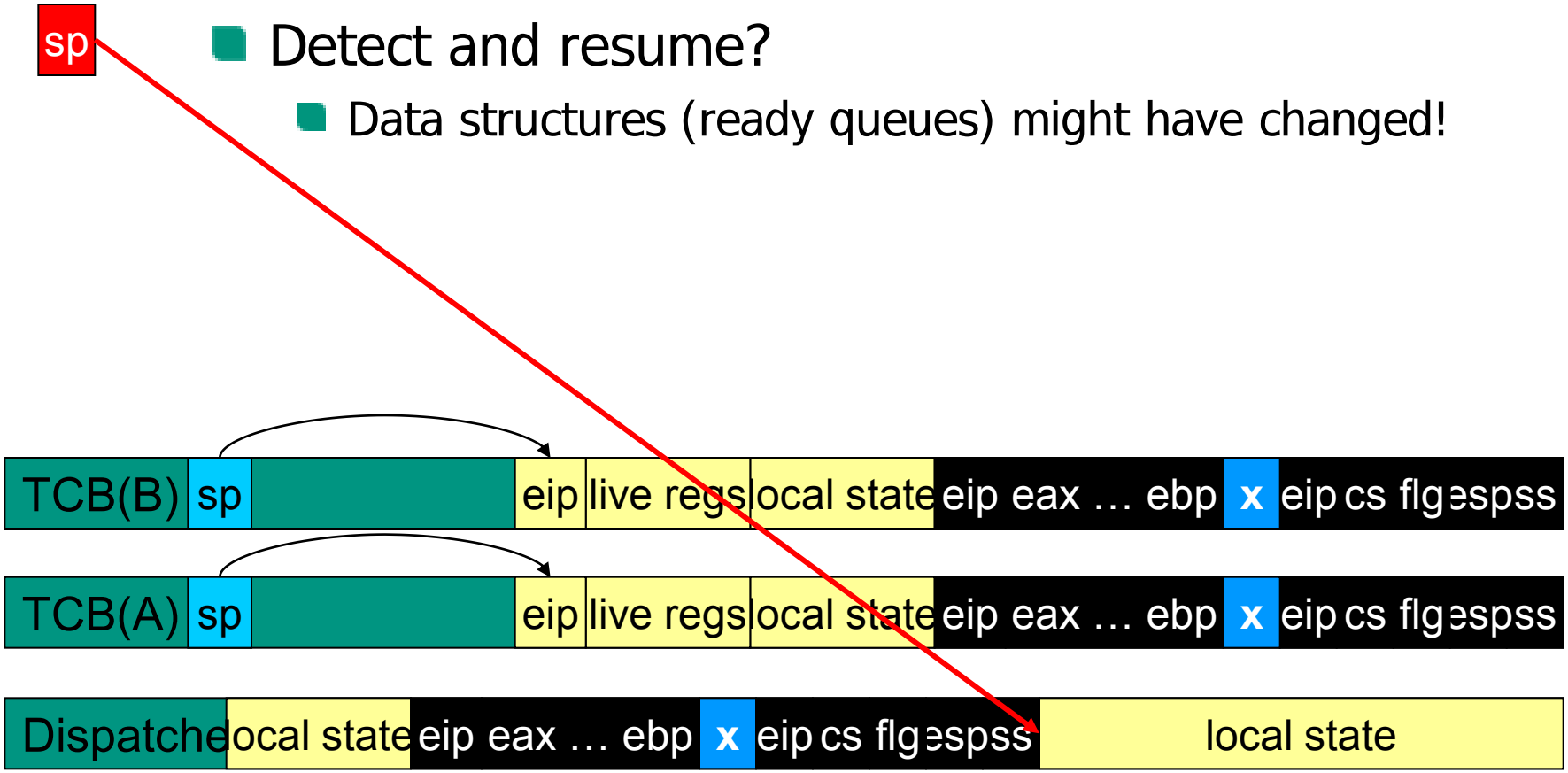
Example: Dispatch with `Tick'

- Dispatcher interrupted by timer IRQ
  - Detect and resume?



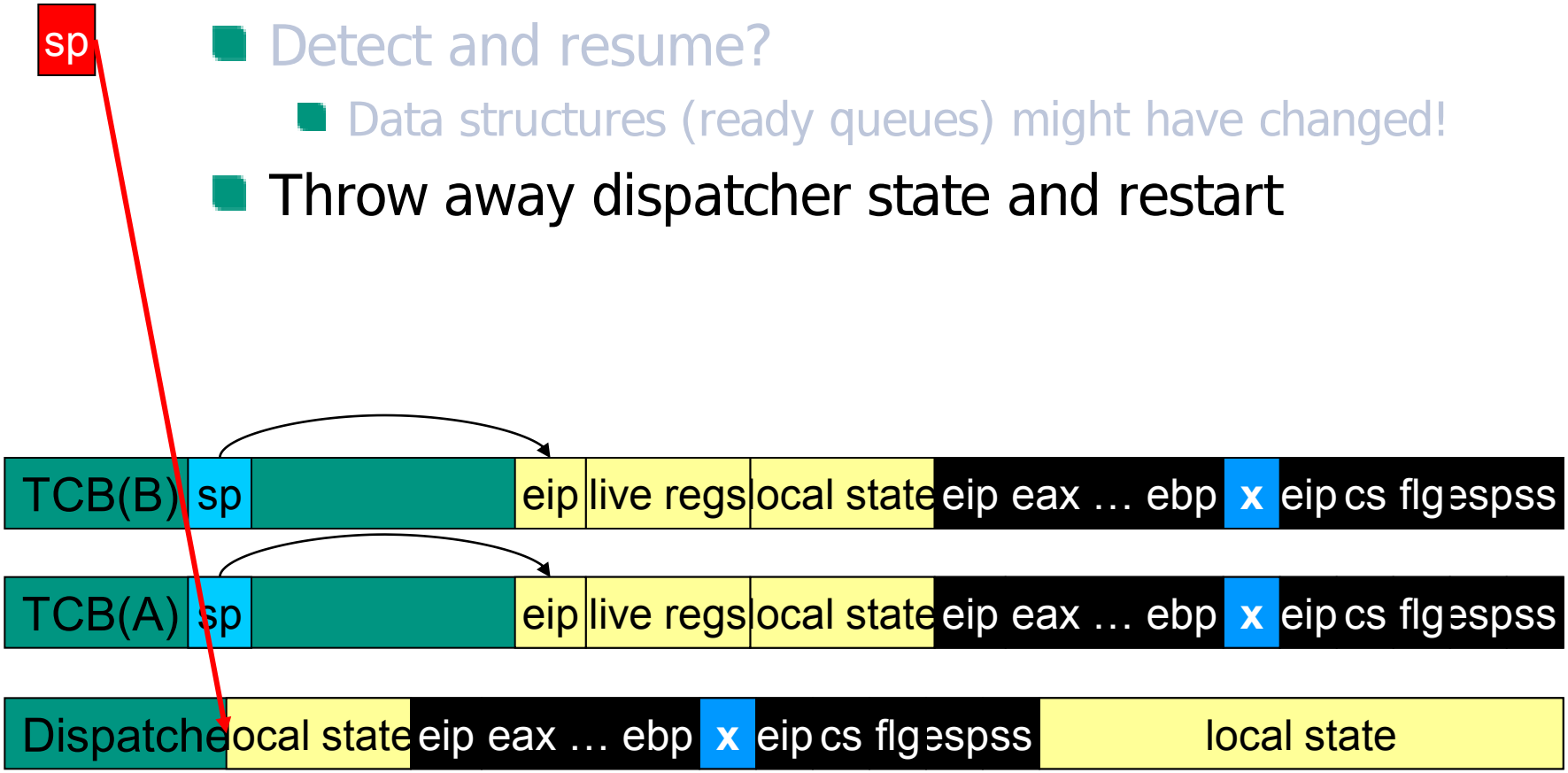
Example: Dispatch with `Tick'

- Dispatcher interrupted by timer IRQ
  - Detect and resume?
    - Data structures (ready queues) might have changed!



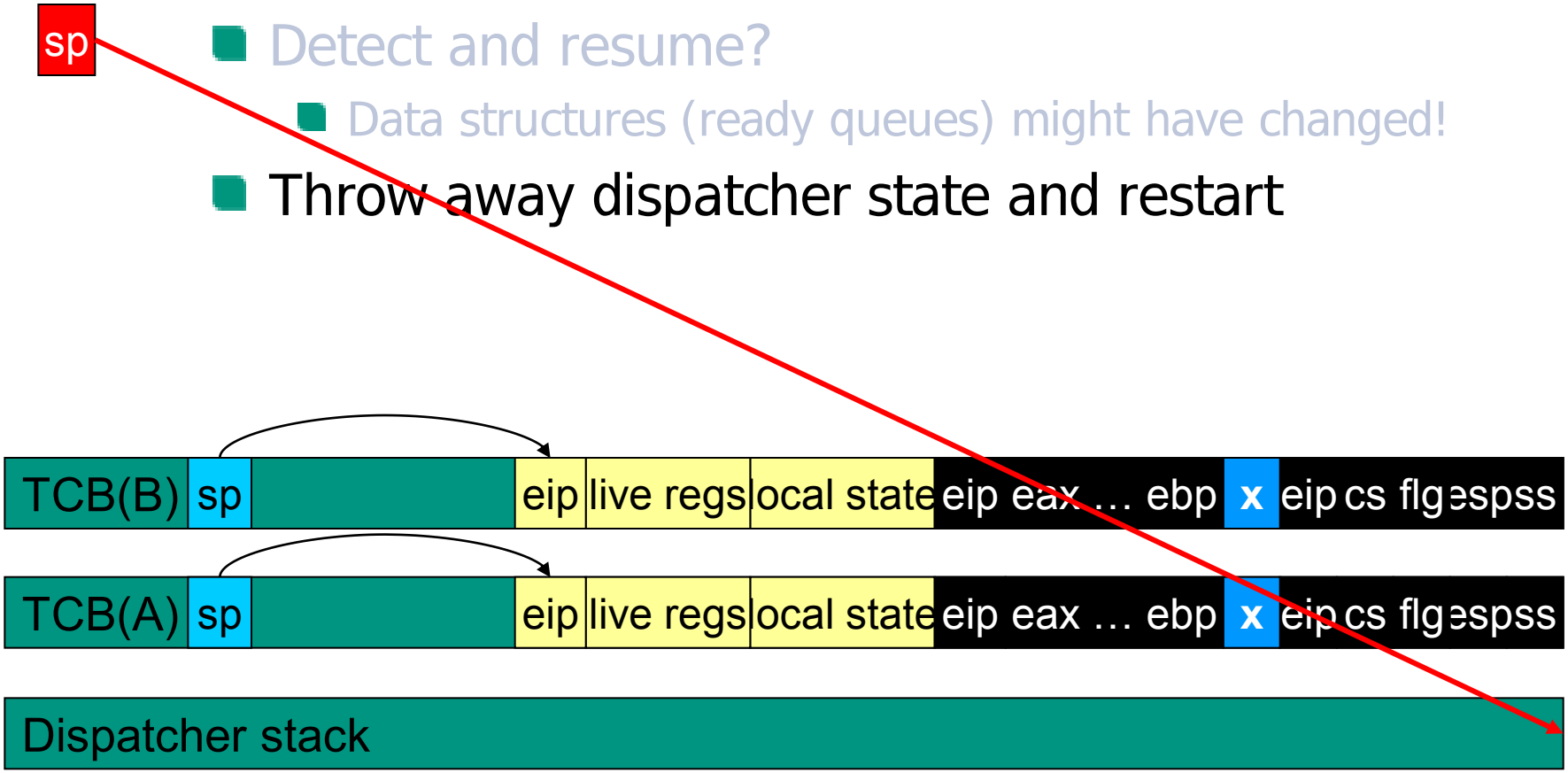
Example: Dispatch with `Tick'

- Dispatcher interrupted by timer IRQ
  - Detect and resume?
    - Data structures (ready queues) might have changed!
  - Throw away dispatcher state and restart



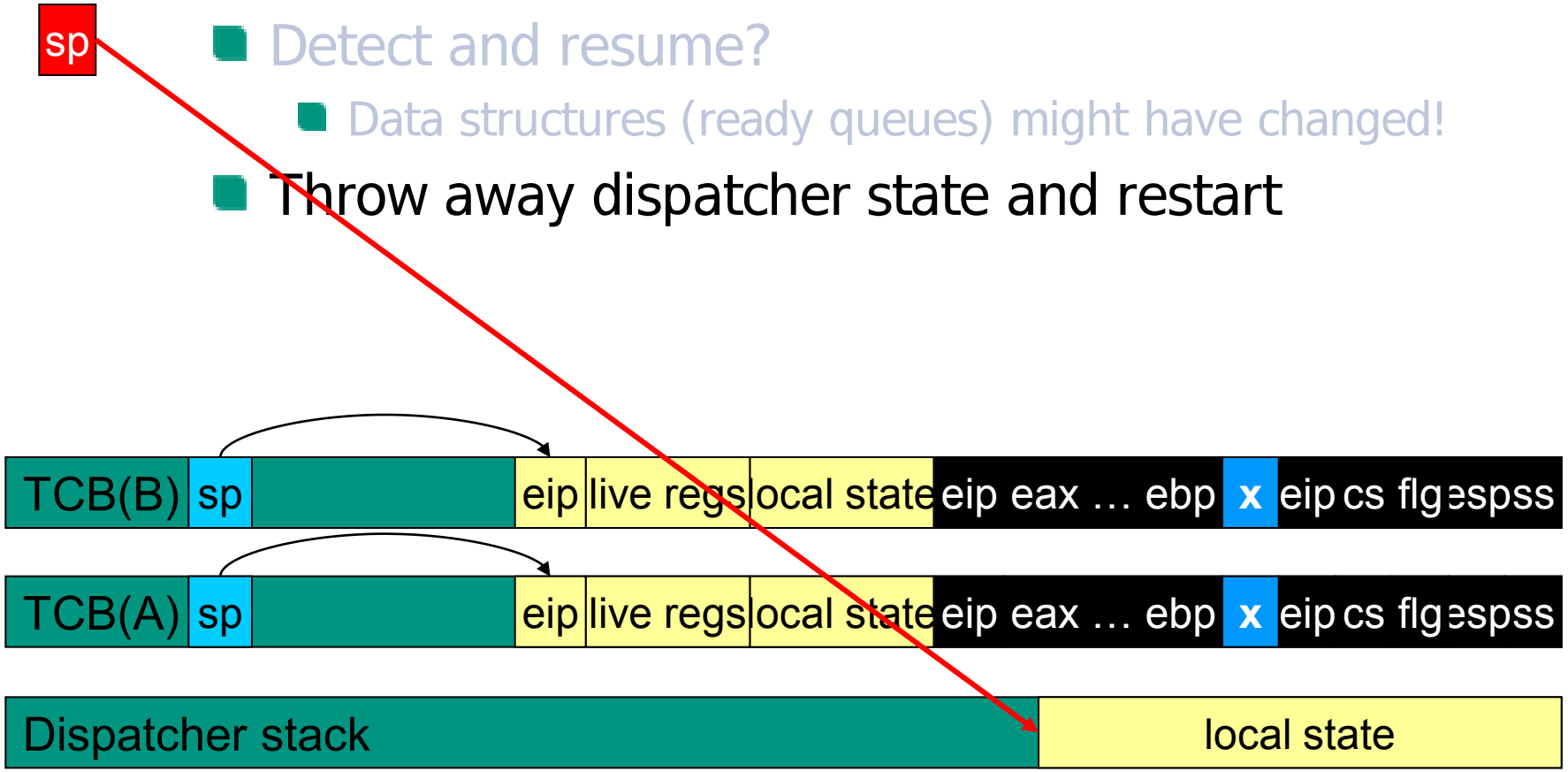
Example: Dispatch with `Tick'

- Dispatcher interrupted by timer IRQ
  - Detect and resume?
    - Data structures (ready queues) might have changed!
  - Throw away dispatcher state and restart



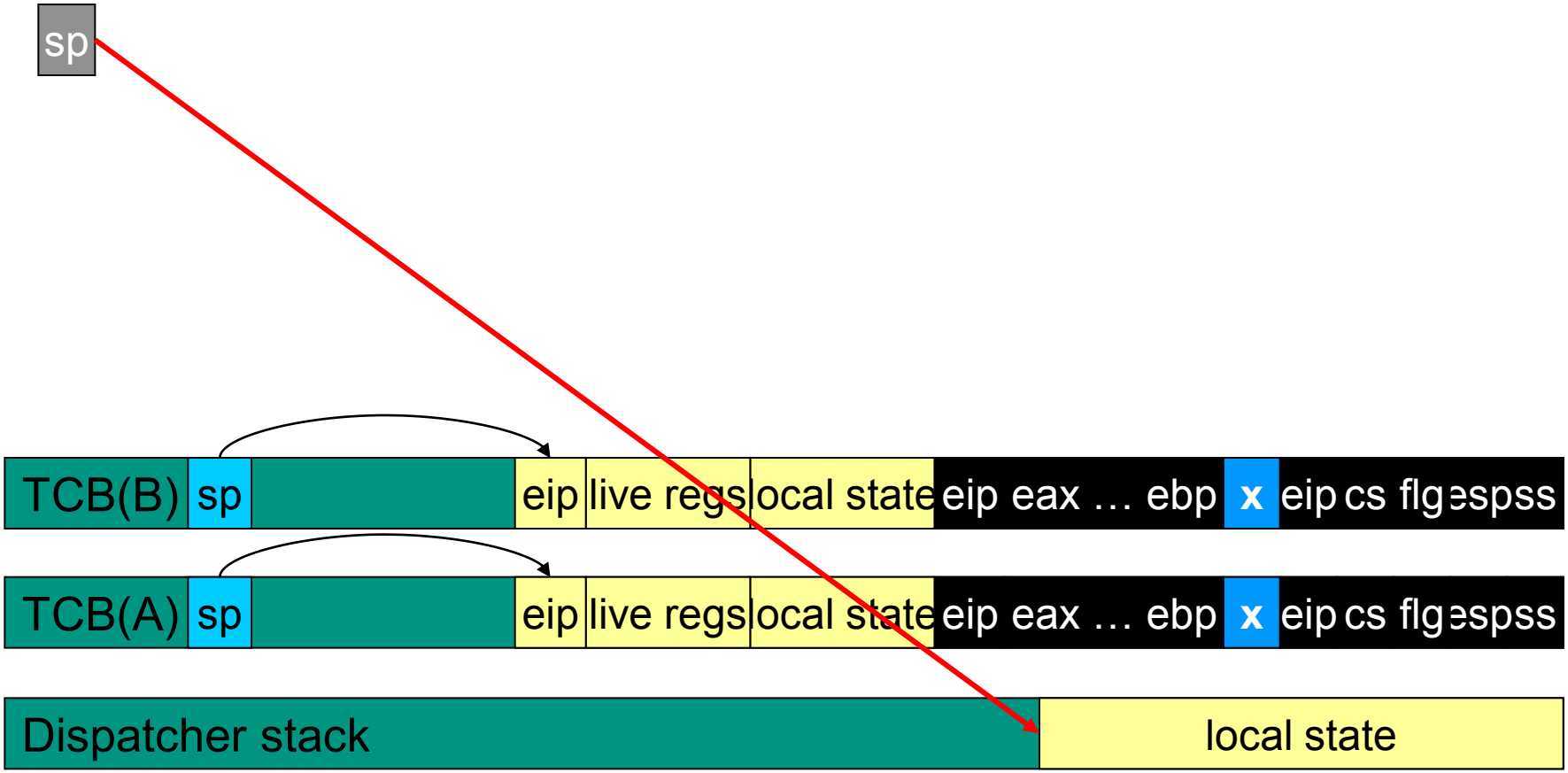
Example: Dispatch with `Tick'

- Dispatcher interrupted by timer IRQ
  - Detect and resume?
    - Data structures (ready queues) might have changed!
  - Throw away dispatcher state and restart



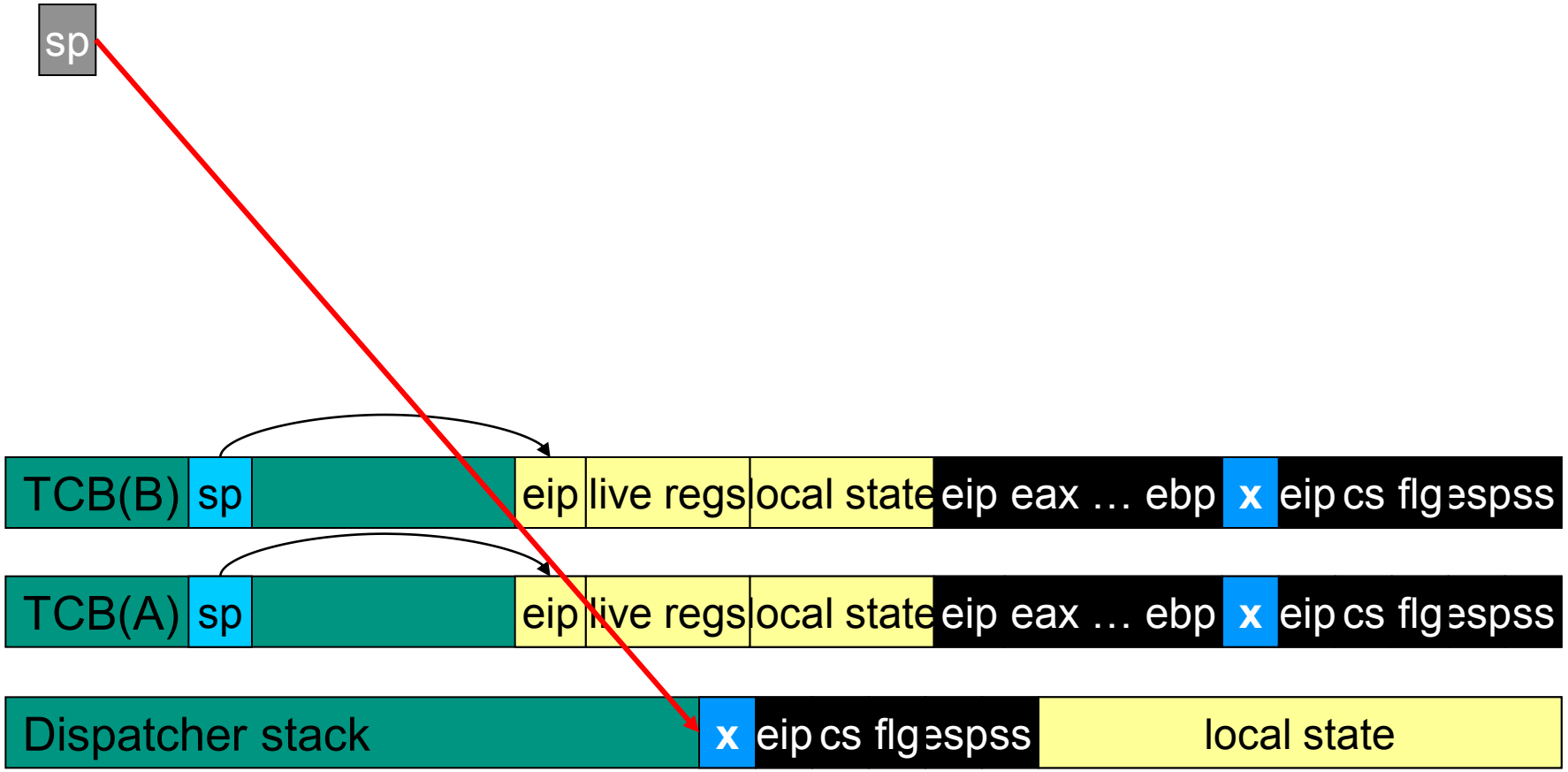
Example: Dispatch with Device IRQ

- Dispatcher interrupted by device IRQ



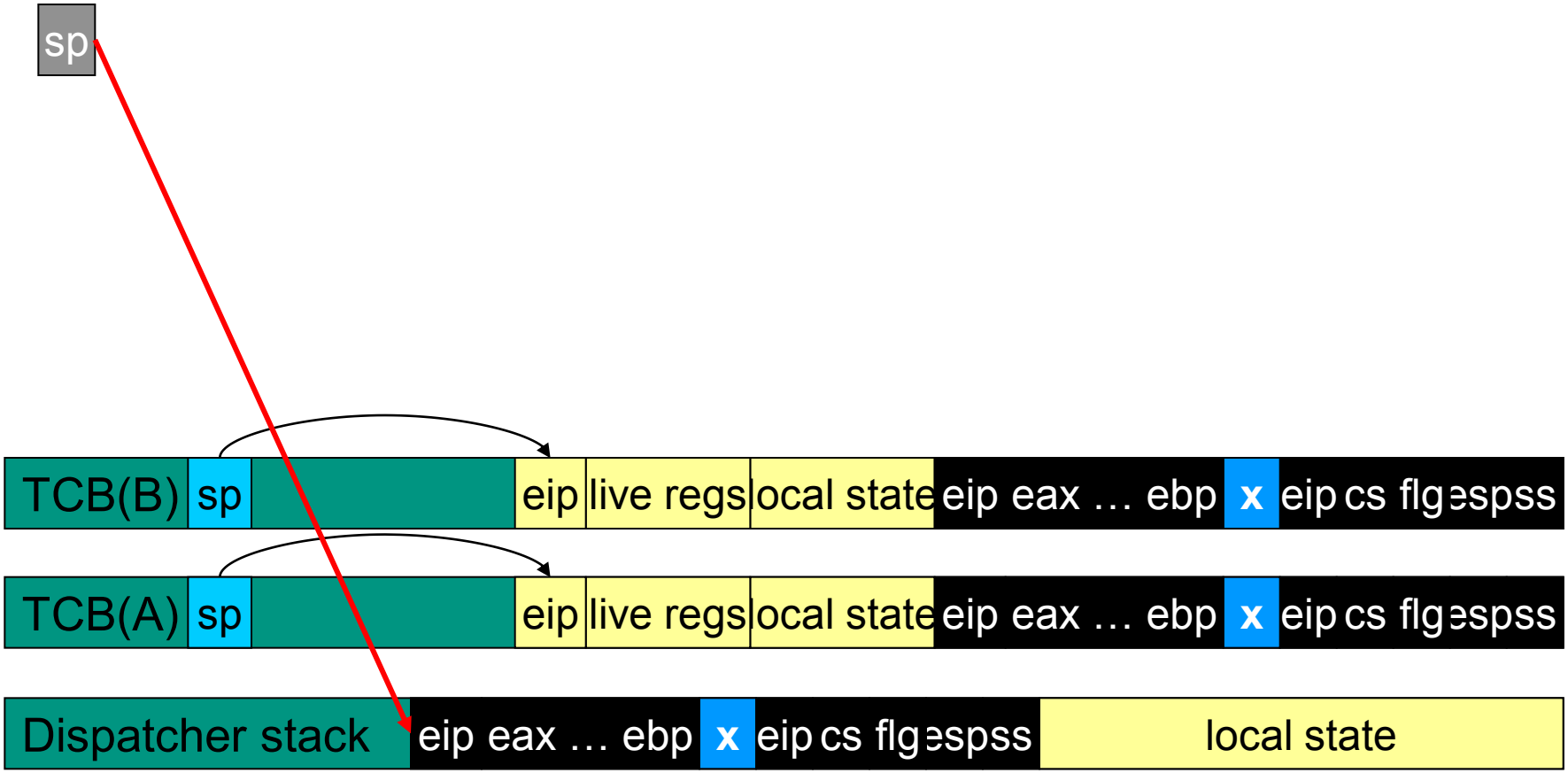
Example: Dispatch with Device IRQ

- Dispatcher interrupted by device IRQ



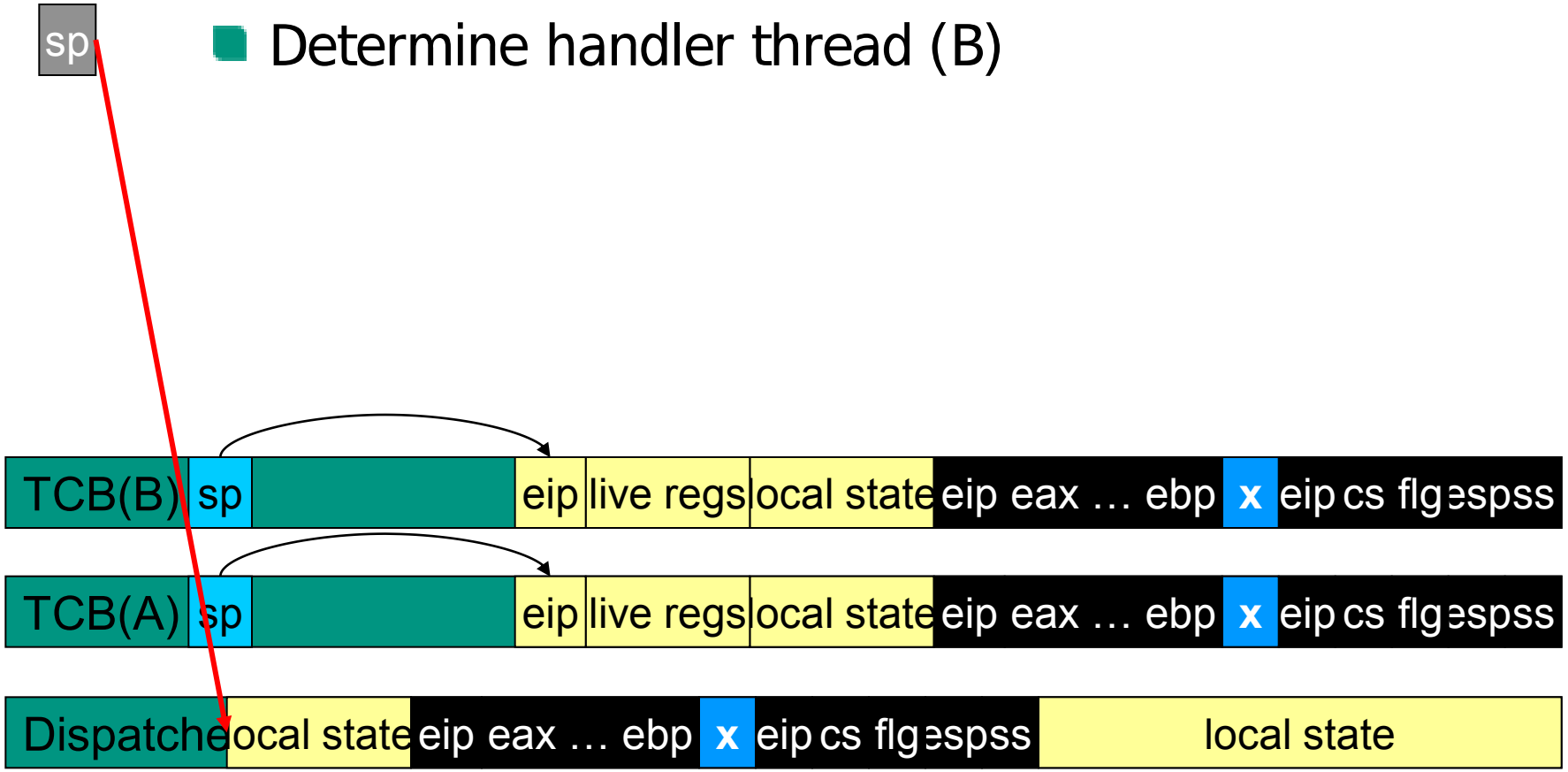
Example: Dispatch with Device IRQ

- Dispatcher interrupted by device IRQ



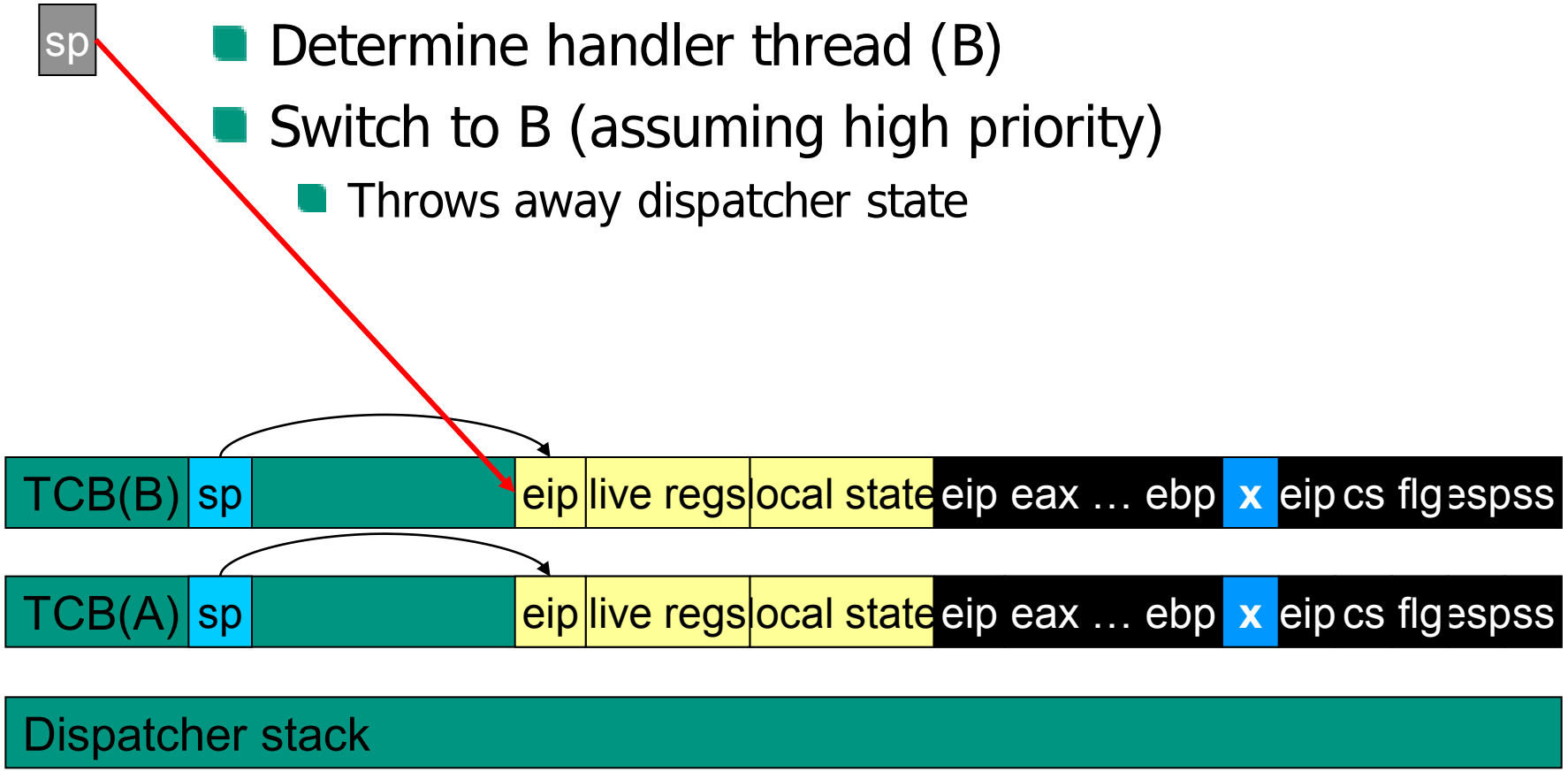
Example: Dispatch with Device IRQ

- Dispatcher interrupted by device IRQ
  - Determine handler thread (B)



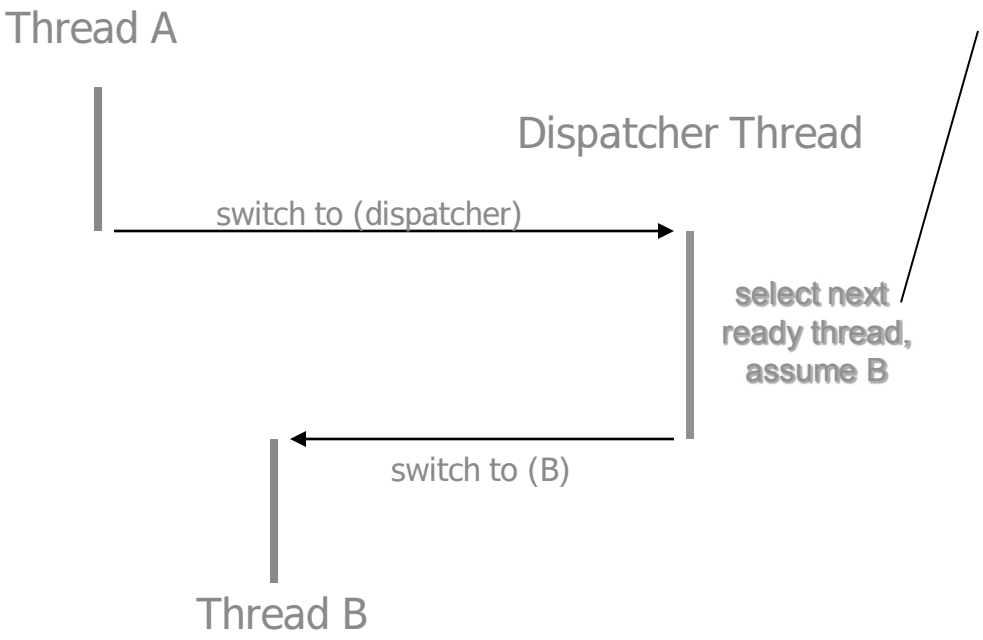
Example: Dispatch with Device IRQ

- Dispatcher interrupted by device IRQ
  - Determine handler thread (B)
  - Switch to B (assuming high priority)
    - Throws away dispatcher state



Switch to ()

- Dispatcher thread is also idle thread



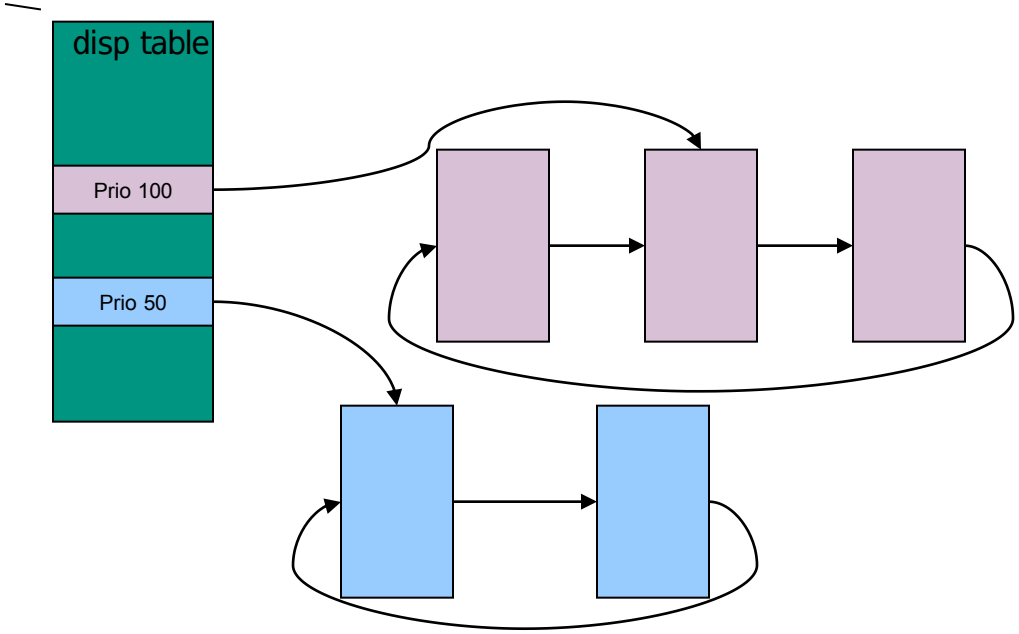

```
B := A ;
do
   B := next ready (B) ;
   if B ≠ nil then
      return
   fi ;
   idle
od .
```

Priorities

- 0 (lowest) … 255
- Hard priorities
- Dynamically changeable
- Round robin per priority

- Ready TCB list per priority
- ‘Current TCB’ per list

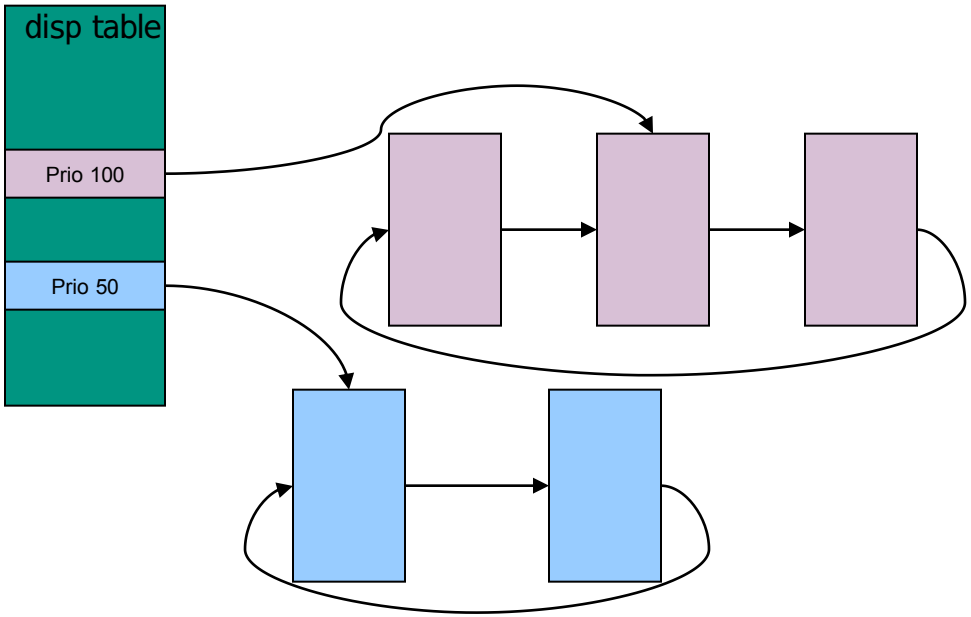
```
do
   p := 255 ;
   do
      if current[p] ≠ nil then
         B := current[p] ;
         return
      fi ;
      p -= 1
   until p < 0 od ;
   idle
od .
```



Priorities

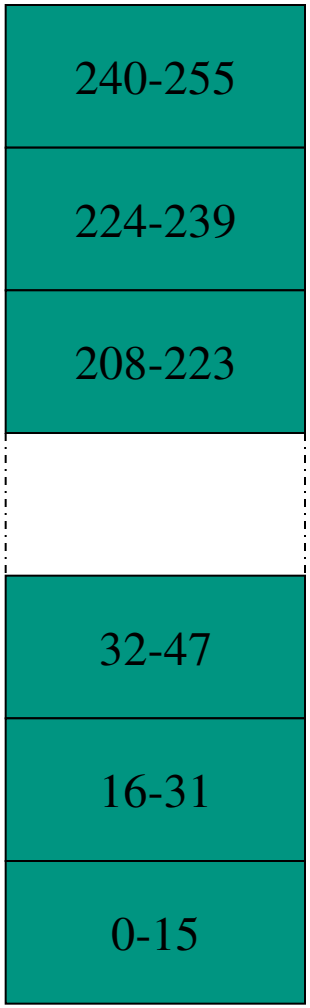
- Optimizations
  - Remember highest active priority
  - Bitmap of active priority levels

```
do
   if current[highest active p] ≠ nil then
      B := current[highest active p] ;
      return
   elif highest active p > 0 then
      highest active p -= 1
   else
      idle
   fi
od .
```



Optimization: Priorities

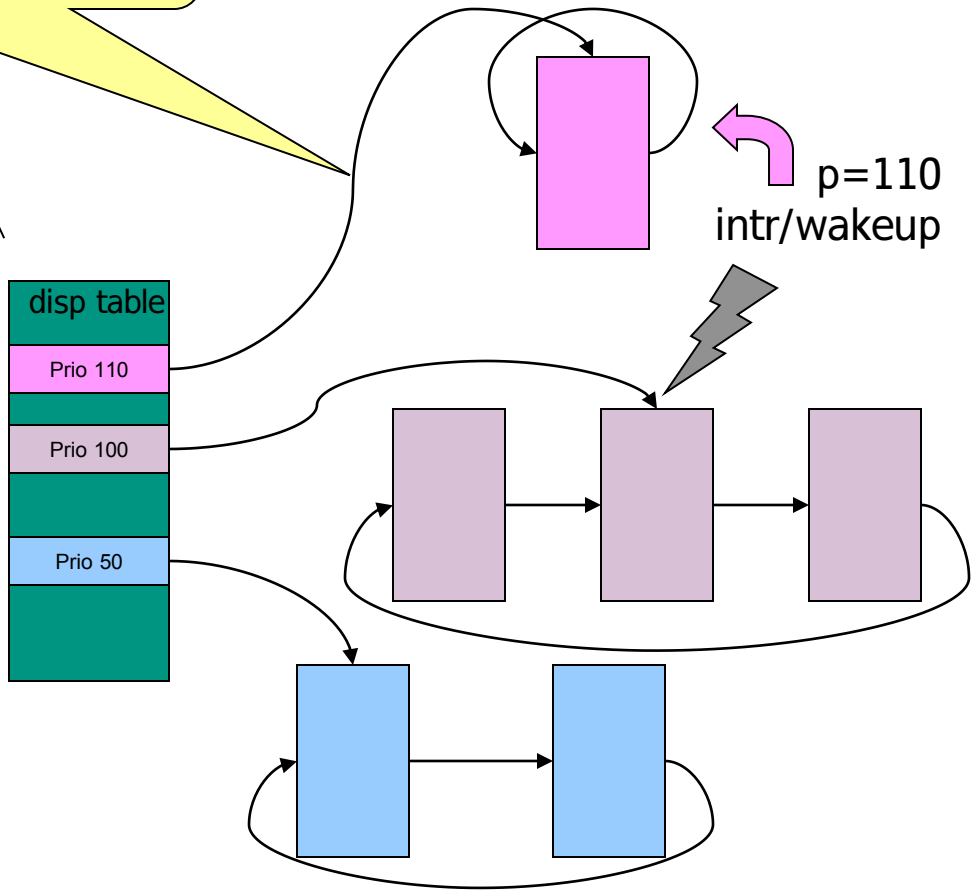
- Bitmap
  - Set bit on insertion
  - Clear when group empty
- IA-32: **`BSR`**
  - **`BIT SCAN REVERSE`**
  - 2 cycles (?)



Priorities, Preemption

highest active p :=
max (new p, highest active p) .

```
do
   if current[highest active p] ≠ nil then
      B := current[highest active p] ;
      return
   elif highest active p > 0 then
      highest active p -= 1
   else
      idle
   fi
od .
```

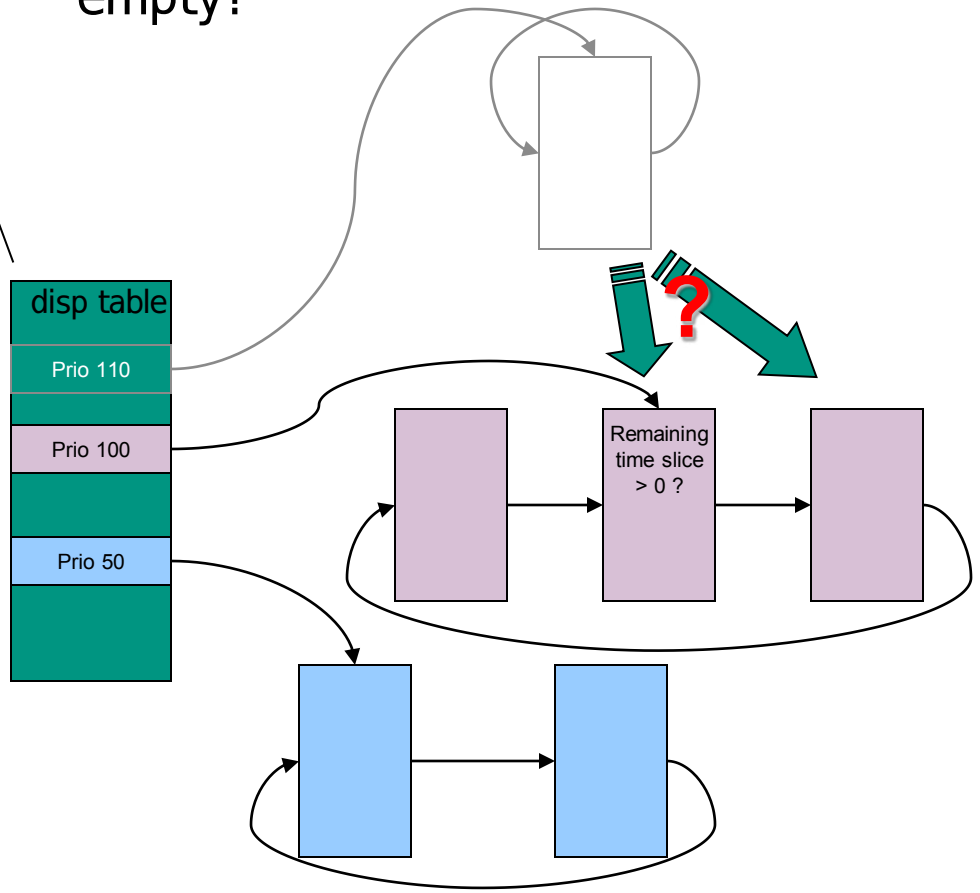


Priorities, Preemption

```
do
   if current[highest active p] ≠ nil then
      round robin if necessary ;
      B := current[highest active p] ;
      return
   elif highest active p > 0 then
      highest active p -= 1
   else
      idle
   fi
od .

round robin if necessary:
   if current[hi act p].rem ts = 0 then
      current[hi act p].next.rem ts := new ts;
      current[hi act p] := current[hi act p].next
   fi .
```

- What happens when a priority falls empty?

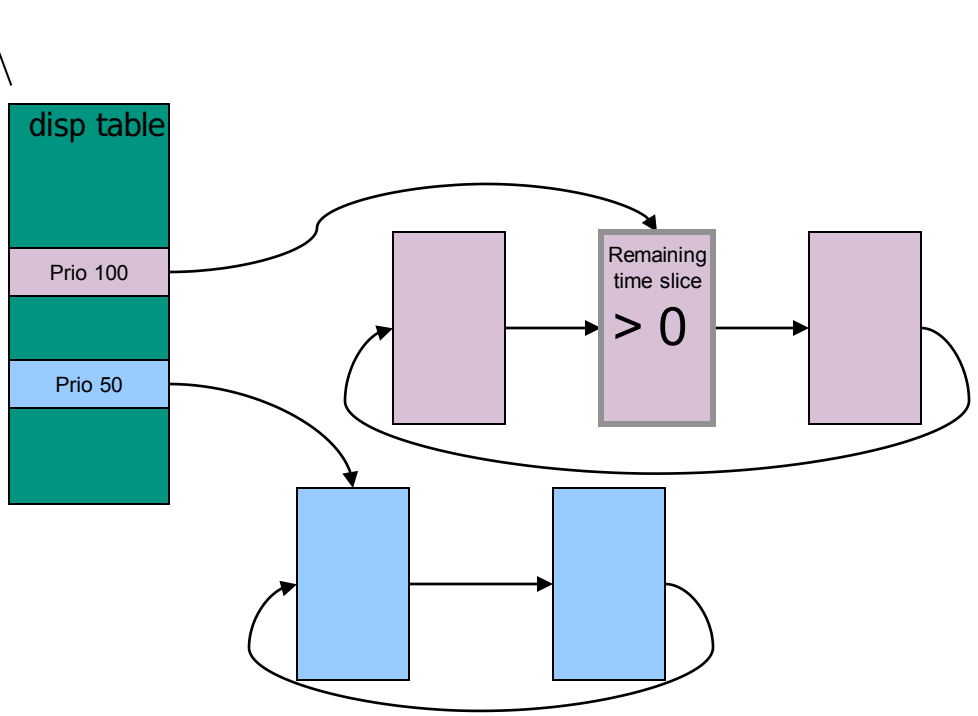


Priorities, Preemption

```
do
   if current[highest active p] ≠ nil then
      round robin if necessary ;
      B := current[highest active p] ;
      return
   elif highest active p > 0 then
      highest active p -= 1
   else
      idle
   fi
od .

round robin if necessary:
   if current[hi act p].rem ts = 0 then
      current[hi act p].next.rem ts := new ts;
      current[hi act p] := current[hi act p].next
   fi .
```

- Preemption, time slice not used up

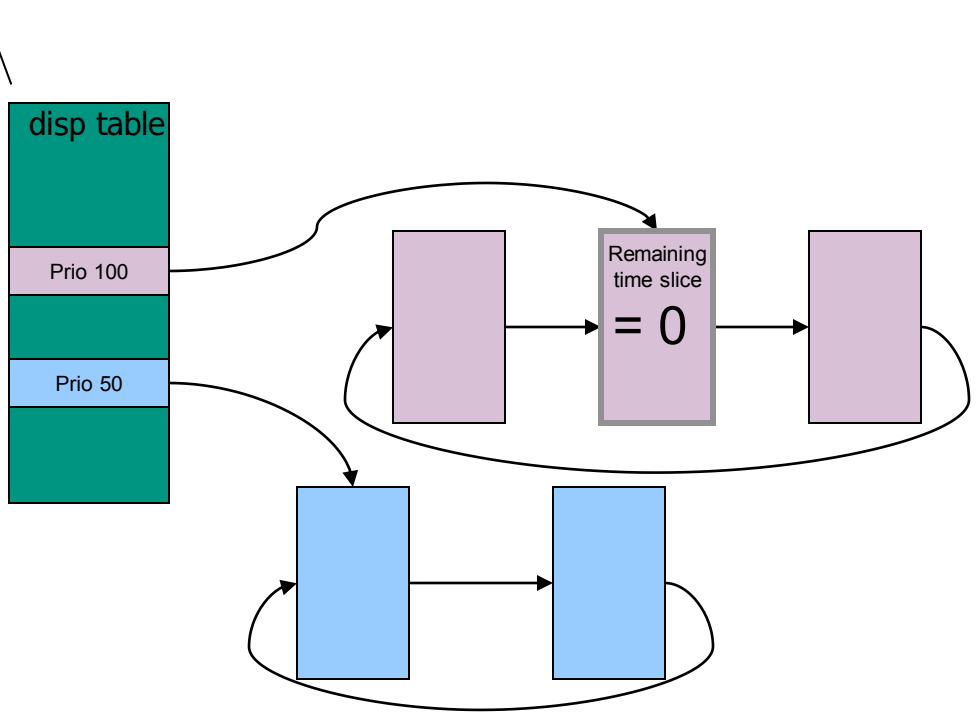


Preemption

```
do
   if current[highest active p] ≠ nil then
      round robin if necessary ;
      B := current[highest active p] ;
      return
   elif highest active p > 0 then
      highest active p -= 1
   else
      idle
   fi
od .

round robin if necessary:
   if current[hi act p].rem ts = 0 then
      current[hi act p].next.rem ts := new ts;
      current[hi act p] := current[hi act p].next
   fi .
```

- Preemption, time slice exhausted

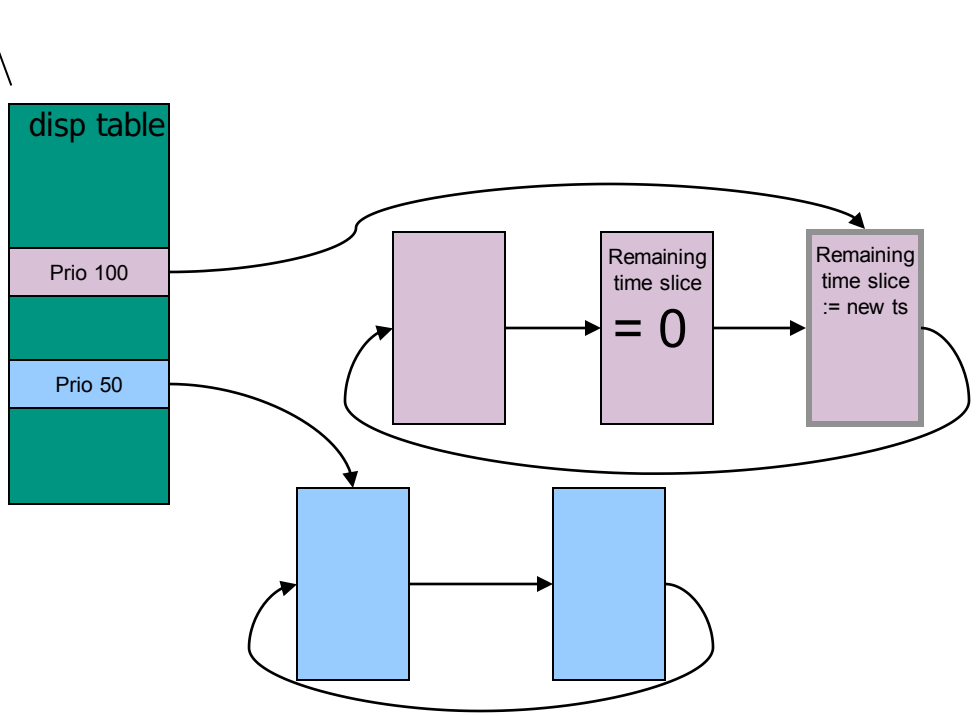


Preemption

```
do
   if current[highest active p] ≠ nil then
      round robin if necessary ;
      B := current[highest active p] ;
      return
   elif highest active p > 0 then
      highest active p -= 1
   else
      idle
   fi
od .

round robin if necessary:
   if current[hi act p].rem ts = 0 then
      current[hi act p].next.rem ts := new ts;
      current[hi act p] := current[hi act p].next
   fi .
```

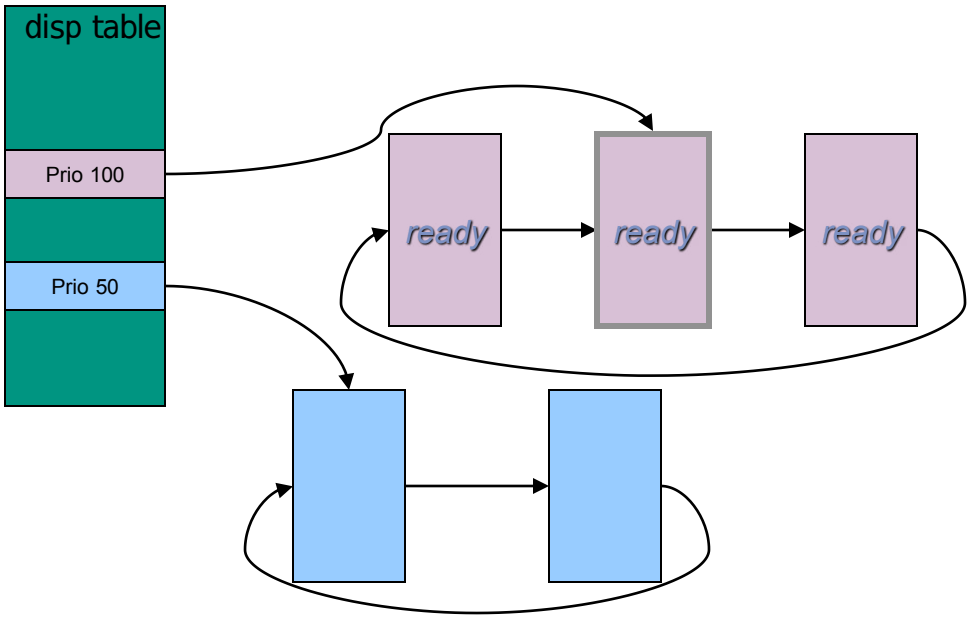
- Preemption, time slice exhausted



Lazy Dispatching

Thread state toggles frequently (per IPC)

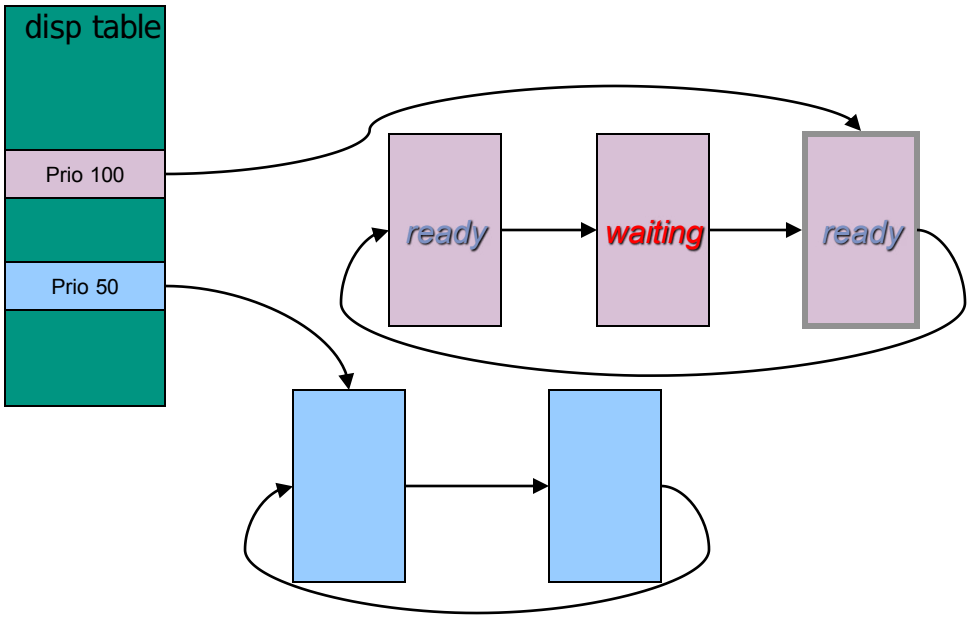
- *ready* ↔ *waiting*
  - Delete/insert ready list is expensive
  - Therefore: delete *lazily* from ready list



Lazy Dispatching

Thread state toggles frequently (per IPC)

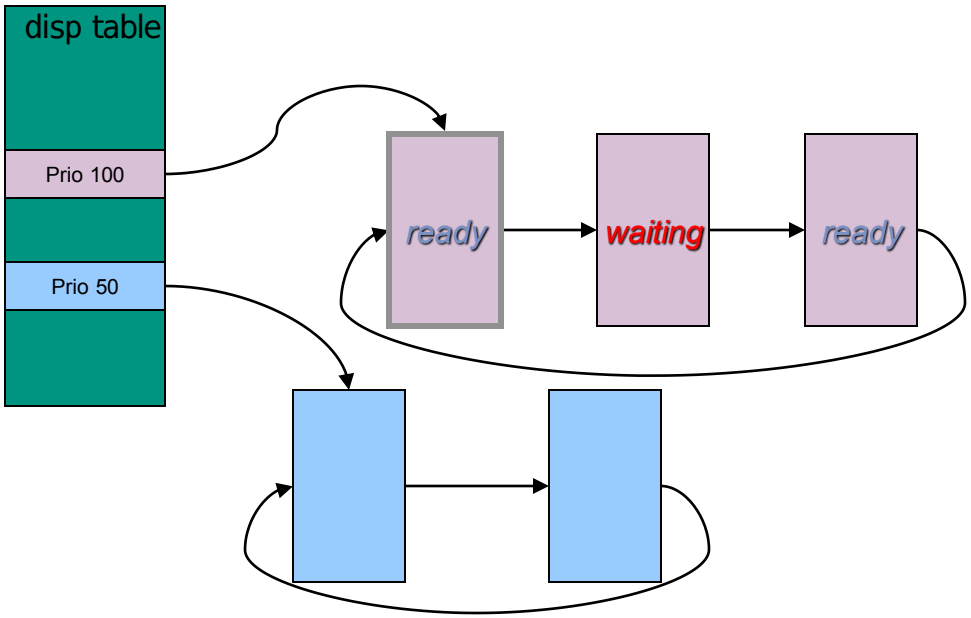
- *ready* ↔ *waiting*
  - Delete/insert ready list is expensive
  - Therefore: delete *lazily* from ready list



Lazy Dispatching

Thread state toggles frequently (per IPC)

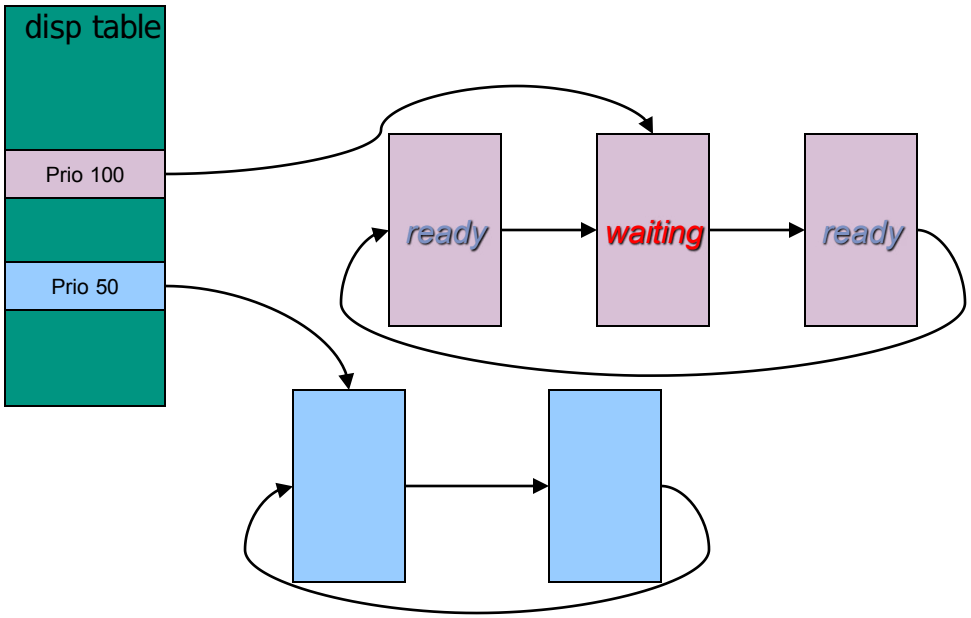
- *ready* ↔ *waiting*
  - Delete/insert ready list is expensive
  - Therefore: delete *lazily* from ready list



Lazy Dispatching

Thread state toggles frequently (per IPC)

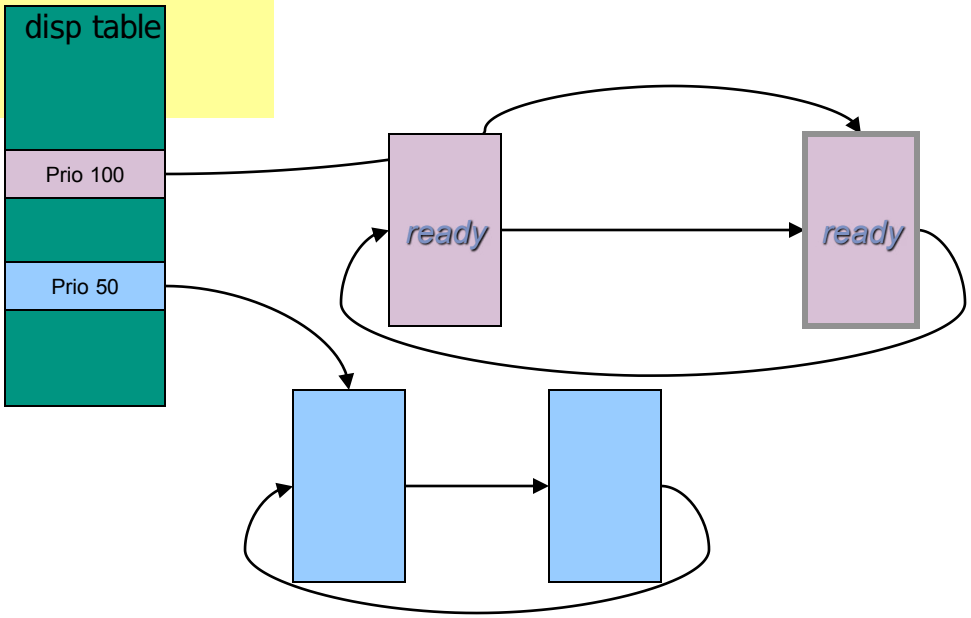
- *ready* ↔ *waiting*
  - Delete/insert ready list is expensive
  - Therefore: delete *lazily* from ready list



Lazy Dispatching

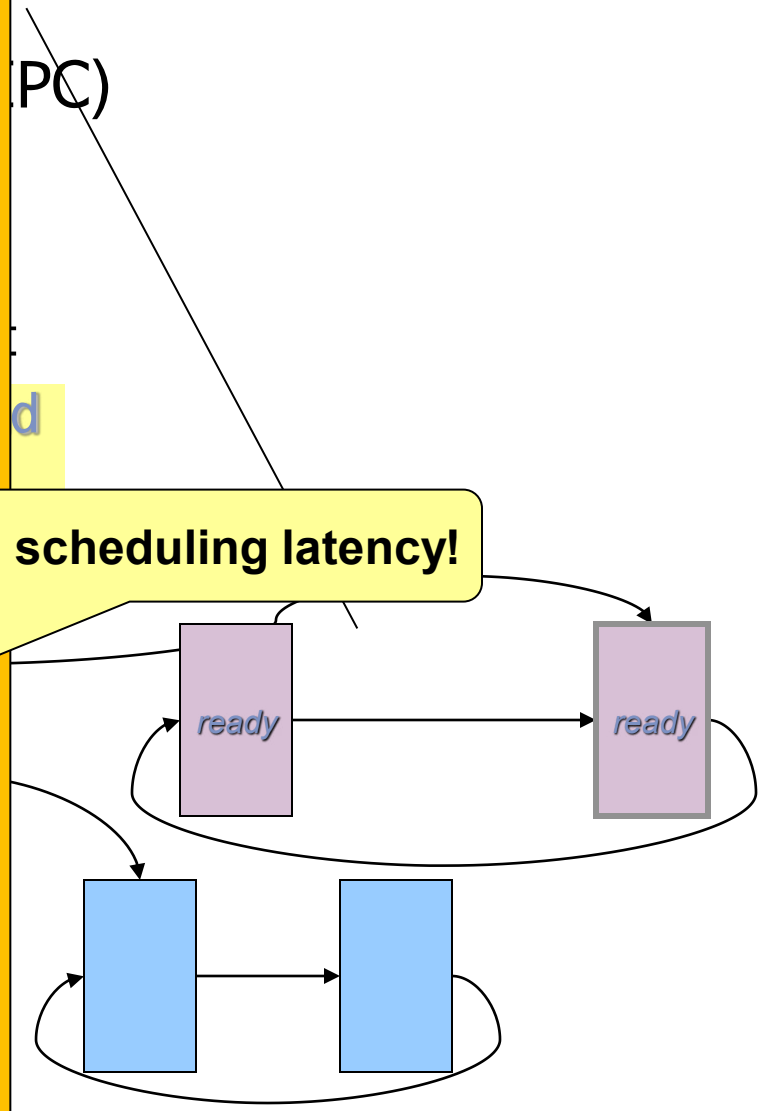
Thread state toggles frequently (per IPC)

- *ready* ↔ *waiting*
  - Delete/insert ready list is expensive
  - Therefore: delete *lazily* from ready list
  - Whenever reaching a non-ready thread
    - Delete it from list
    - Proceed with next



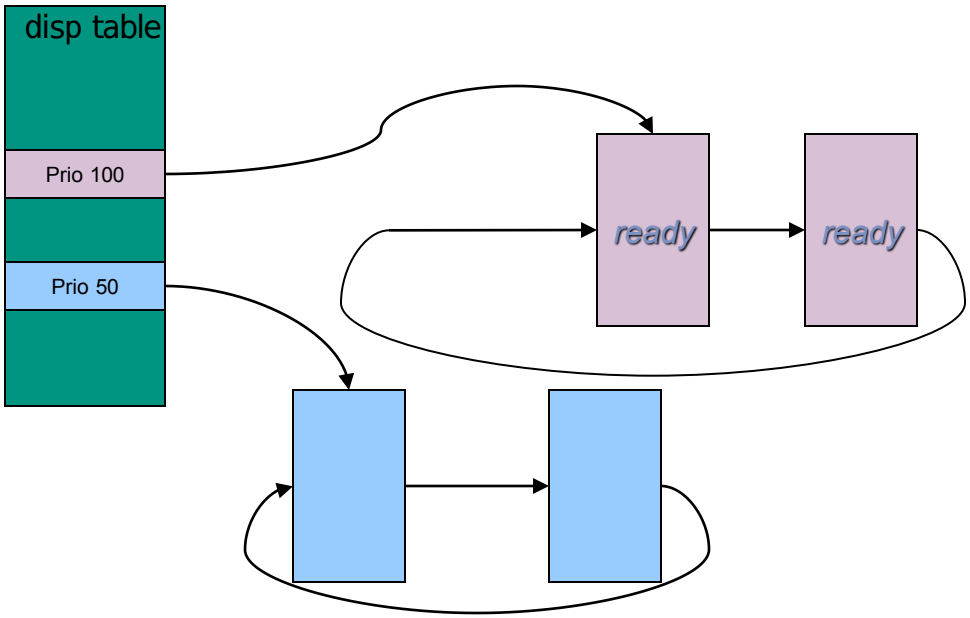
Lazy Dispatching

```
do
   round robin if necessary ;
   if current[highest active p] ≠ nil then
      B := current[highest active p] ; return
   elif highest active p > 0 then
      highest active p -= 1
   else
      idle
   fi
od .
round robin if necessary:
   while current[highest active p] ≠ nil do
      next := current[highest active p].next ;
      if current[highest active p].state ≠ ready then
         delete from list (current[highest active p])
      elif current[highest active p].rem ts = 0 then
         next.rem ts := new ts
      else
         return
      fi ;
      current[highest active p] := next
   od .
```



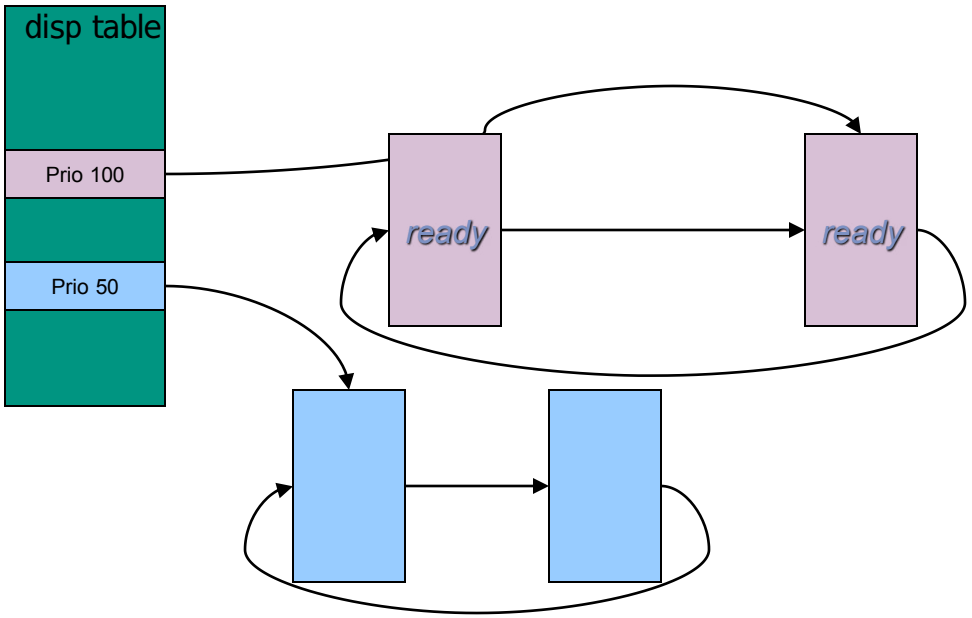
Benno Scheduling (seL4)

Ready list contains all threads
except the currently running thread



Benno Scheduling (seL4)

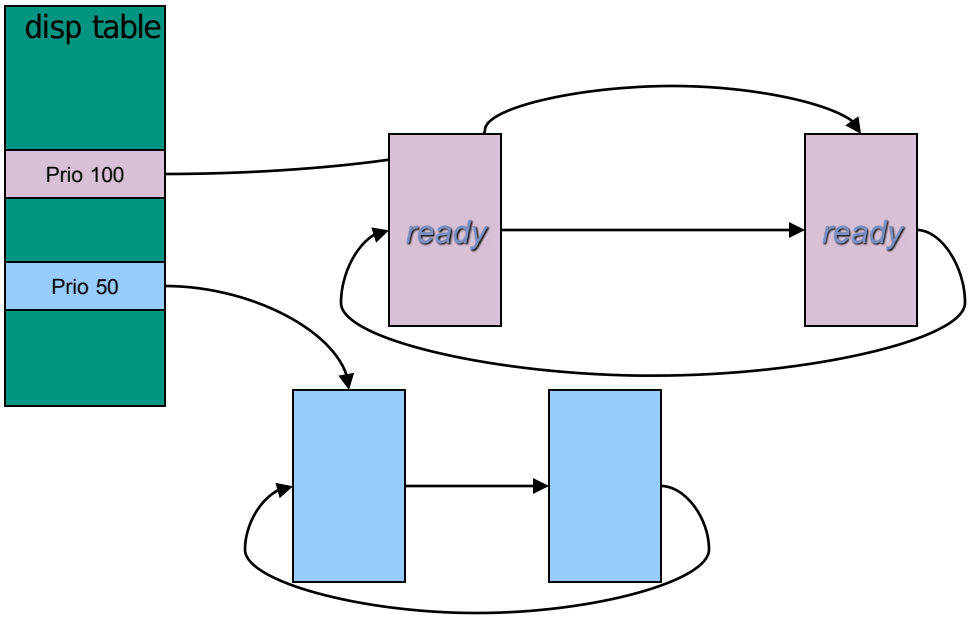
Ready list contains all threads
except the currently running thread



Benno Scheduling (seL4)

Ready list contains all threads
except the currently running thread

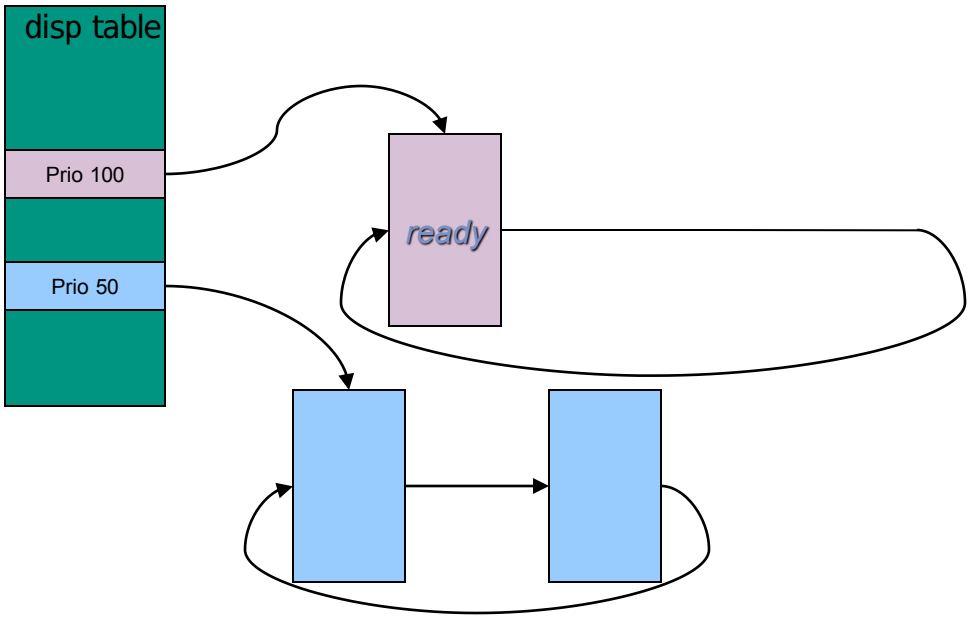
- *ready* ↔ *waiting*
  - Don't re-insert blocked thread



Benno Scheduling (seL4)

Ready list contains all threads
except the currently running thread

- *ready* ↔ *waiting*
  - Don't re-insert blocked thread



Benno Scheduling (seL4)

Predictable scheduling latency
Easy to verify (no loop)

Items touched	Lazy scheduling	Benno scheduling
schedule()	2..n	4
send	2	6
call	2	2

Most common case!
Sender blocks (no insertion)
receiver is blocked (no removal)

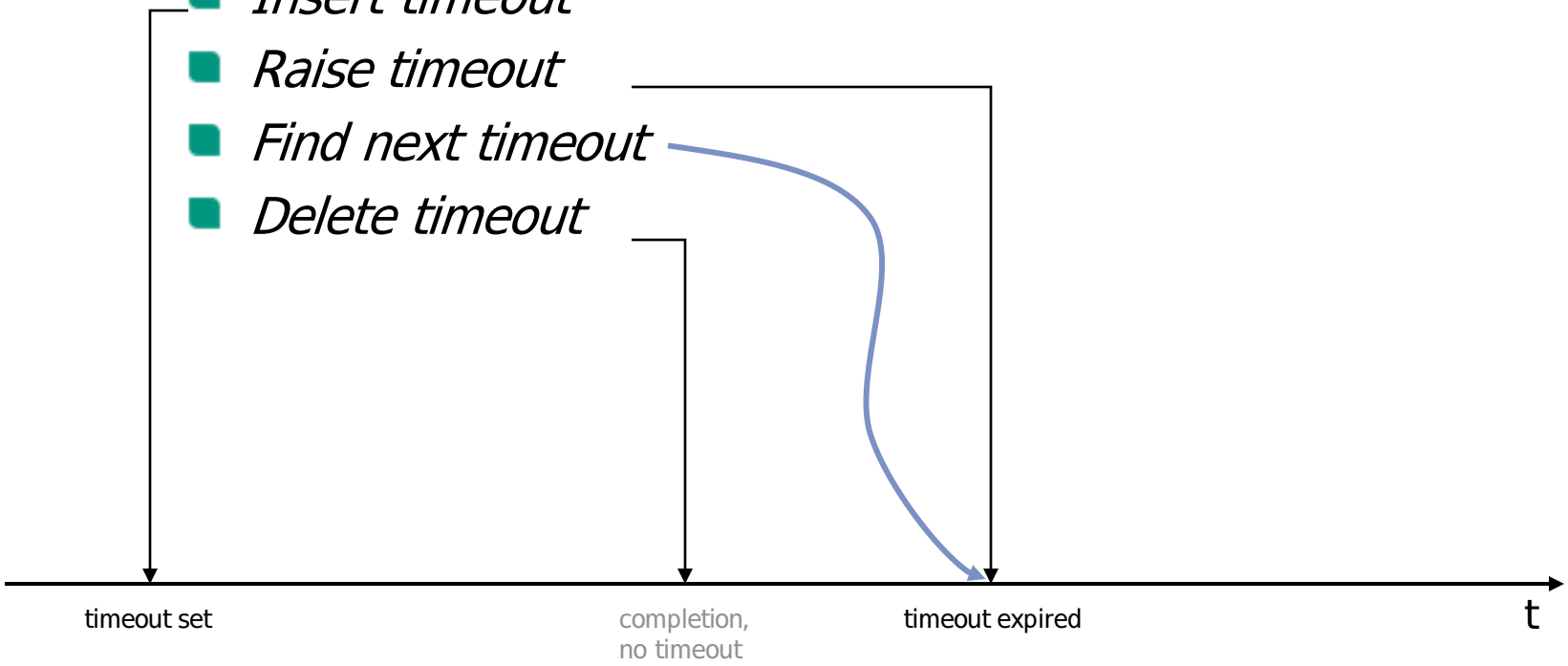
Scheduling inside the kernel?

> A gross heresy against the core microkernel religion of policy-freedom.
>
> (Gernot Heiser)

- User-mode scheduling: Generally unacceptable overheads
  - Some promising indirect approaches though (e.g., Lackorzynski et al., 2012)
- For proper user-scheduling, we need to map CPU time!
  - Need a general, abstract representation of time

Timeouts & Wakeups

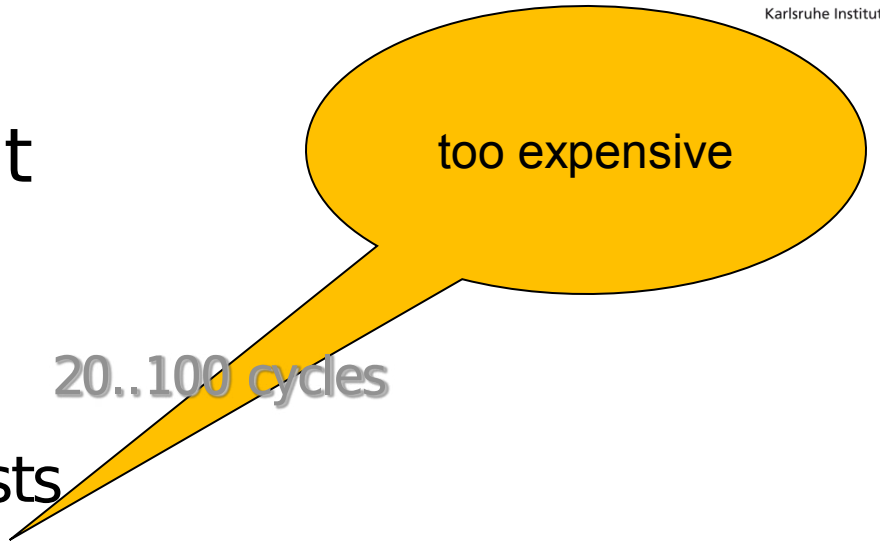
- Operations
  - *Insert timeout*
  - *Raise timeout*
  - *Find next timeout*
  - *Delete timeout*



- Raised-timeout costs are uncritical (occur only after timeout exp time).
- **Most timeouts are never raised!**

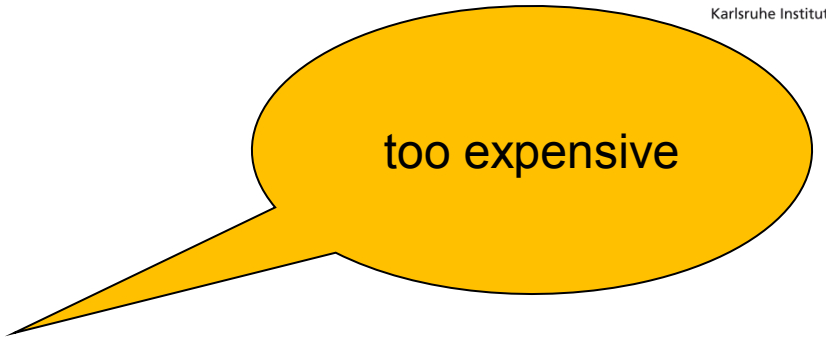
Timeouts & Wakeups

- Idea 1: unsorted list
  - *Insert timeout* costs
    - Prepend entry — 20..100 cycles
  - *Find next timeout* costs
    - Parse entire list — $n \times$ 10..50 cycles
  - *Raise timeout* costs
  - *Delete timeout* costs
    - Delete known entry — 20..100 cycles



Timeouts & Wakeups

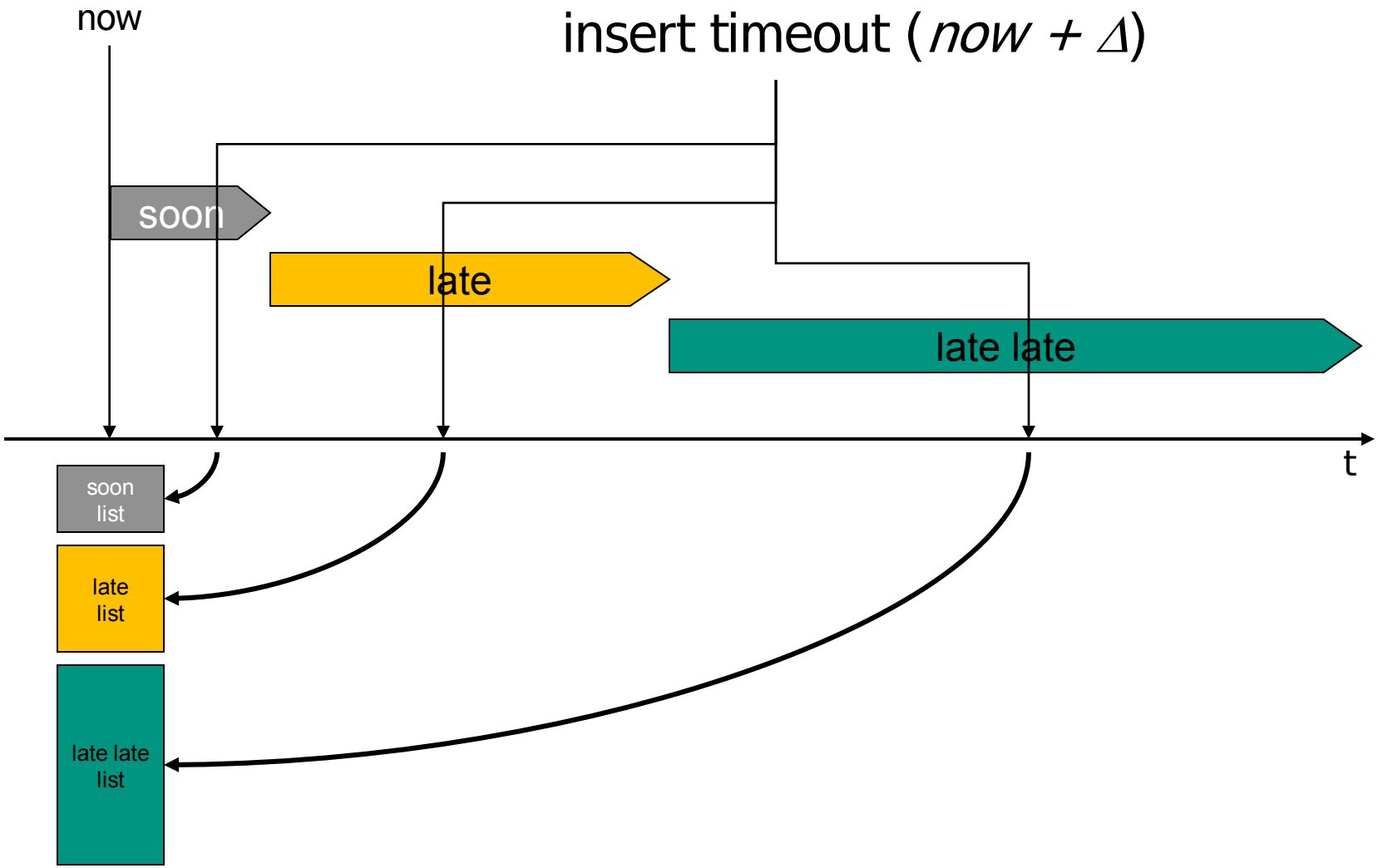
- Idea 2: sorted list
  - *Insert timeout* costs
    - Search + insert — $n/2 \times$ 10..50 + 20..100 cycles
  - *Find next timeout* costs
    - Check head — 10 cycles
  - *Raise timeout* costs
  - *Delete timeout* costs
    - Delete known entry — 20..100 cycles



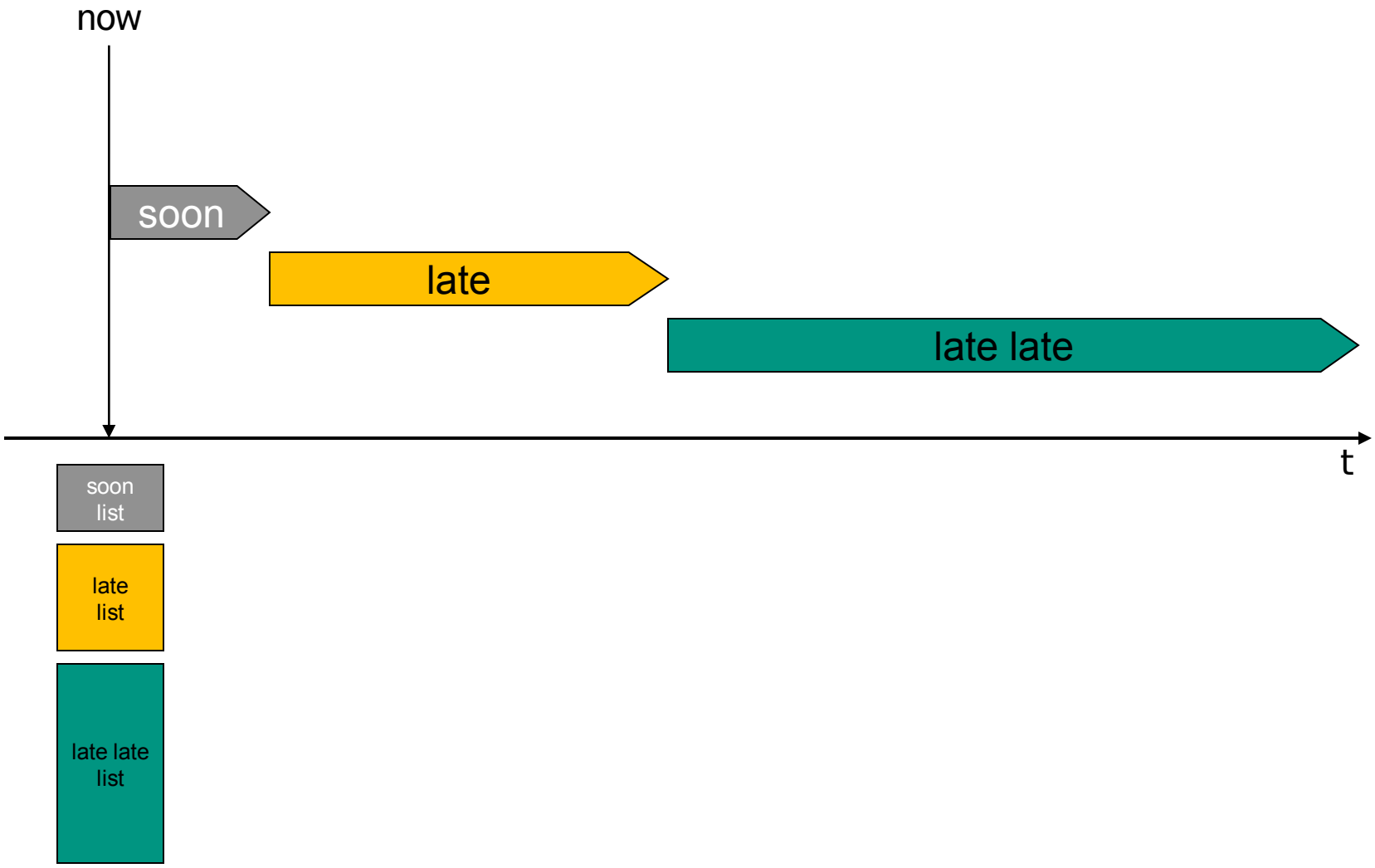
Timeouts & Wakeups

- Idea 3: `sorted tree | heap`
  - *Insert timeout* costs
    - Search + insert — $\log n \times 10..50 + 20..100$ cycles
  - *Find next timeout* costs
    - Find min node | root — $\log n \times 10..50 \mid 10$ cycles
  - *Raise timeout* costs
  - *Delete timeout* costs
    - Delete known node — $\log n \times 10..50 + 20..100 \mid \log n \times 20..100$

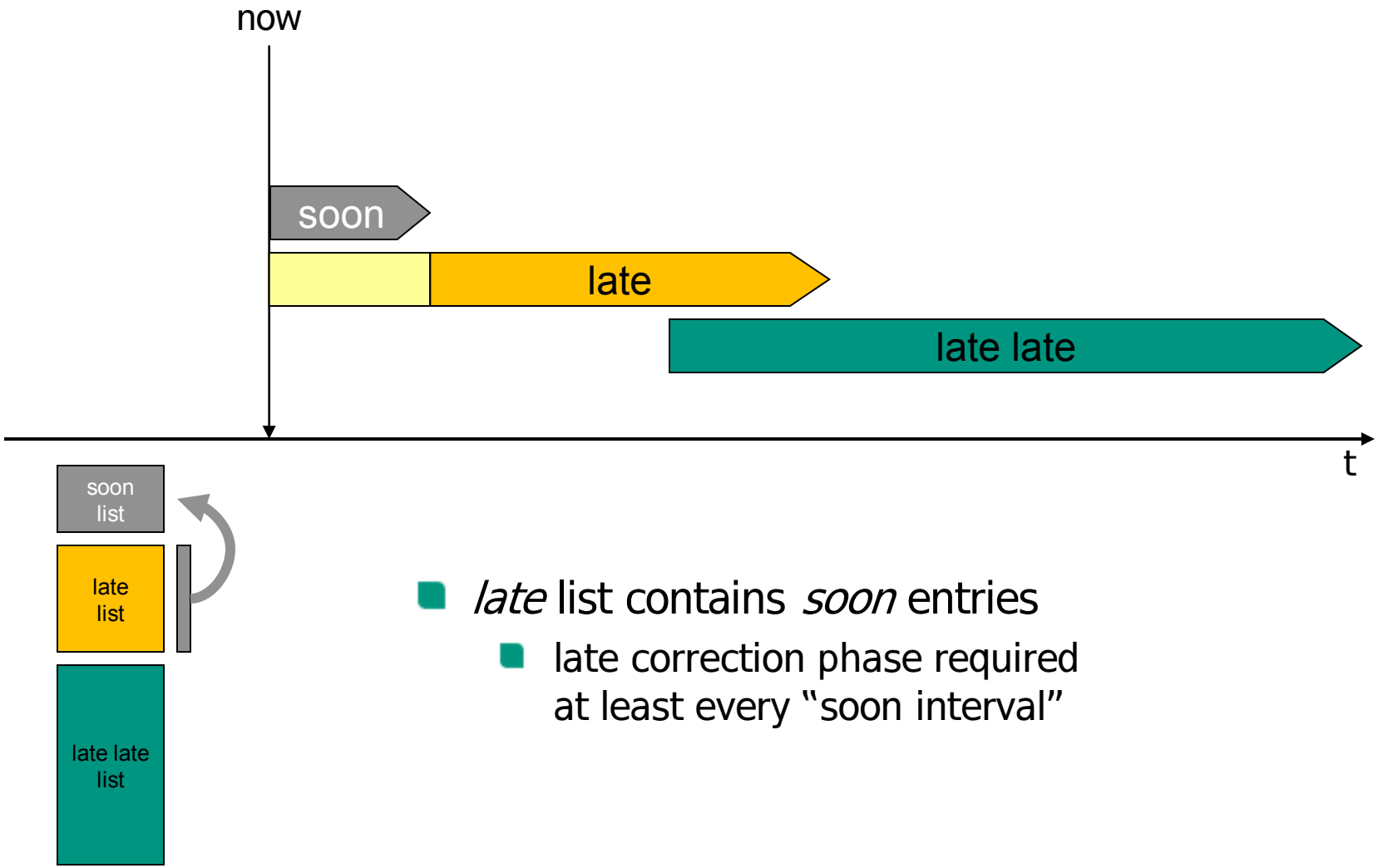
Wakeup Classes



Wakeup Classes

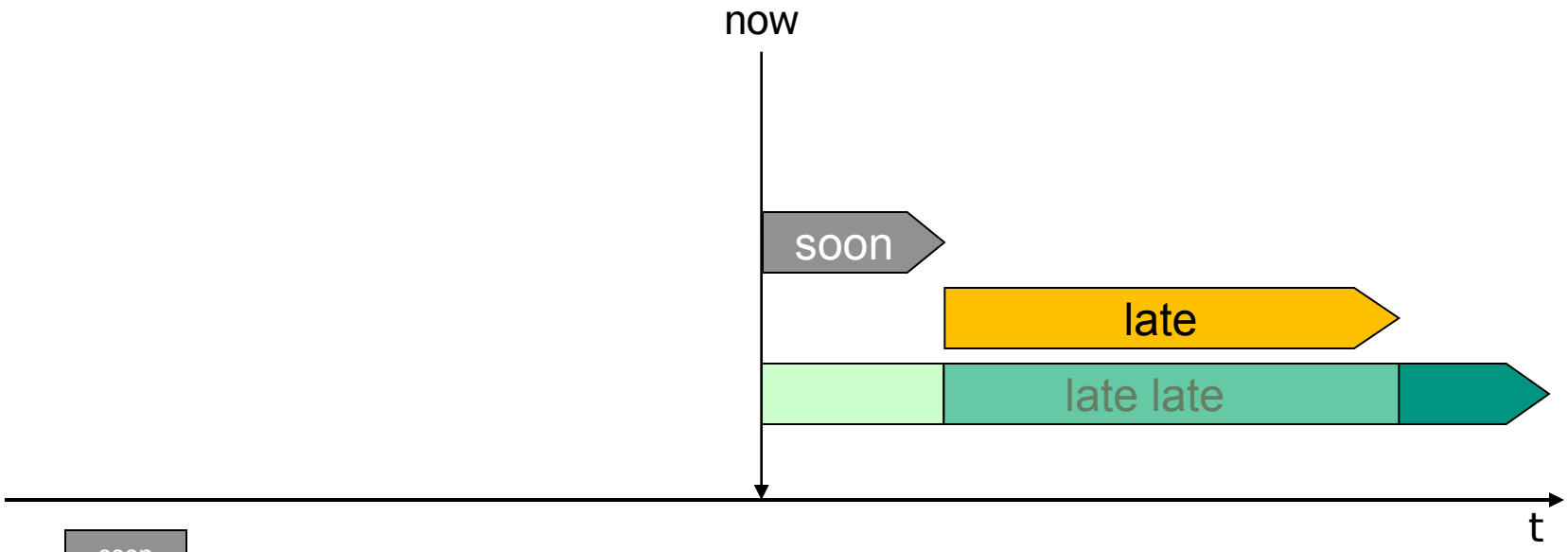


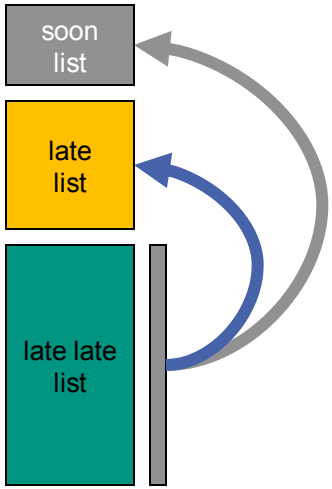
Wakeup Classes



- *late* list contains *soon* entries
  - late correction phase required at least every "soon interval"

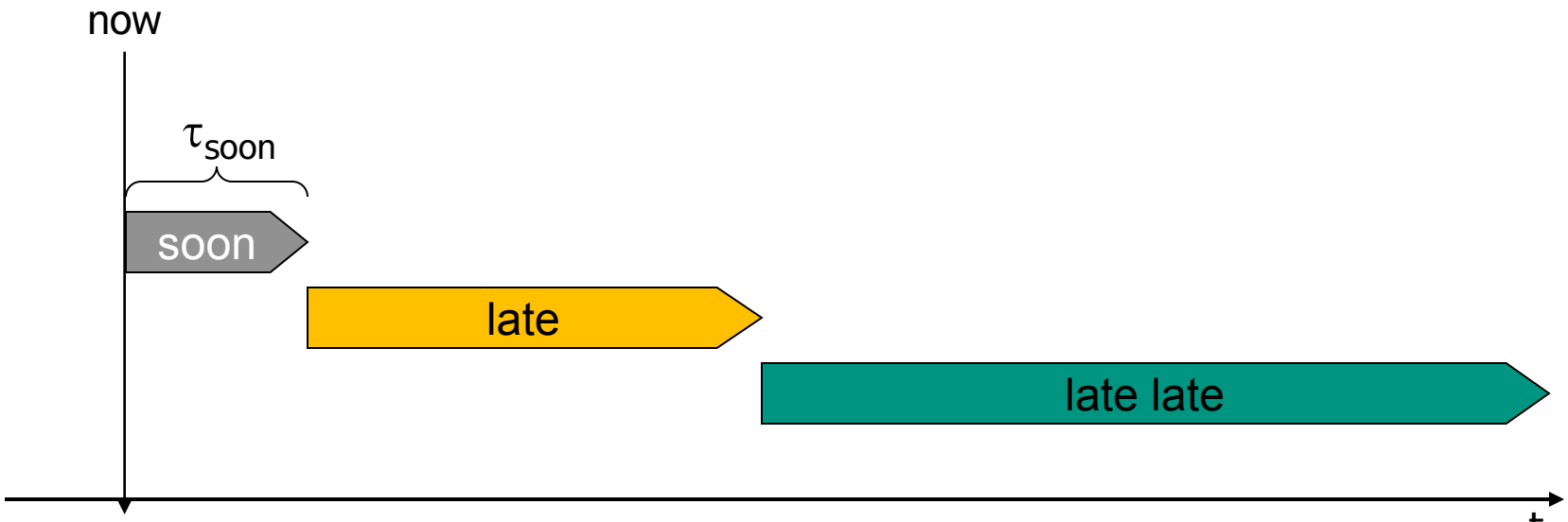
Wakeup Classes

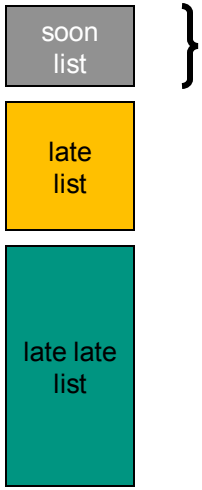




- *late late* list contains *soon* and *late* entries
  - late late correction phase required at least every "soon + late interval"
  - Also move *late late* entries to *late* list
    - Postpone further late late correction phases

Wakeup Classes

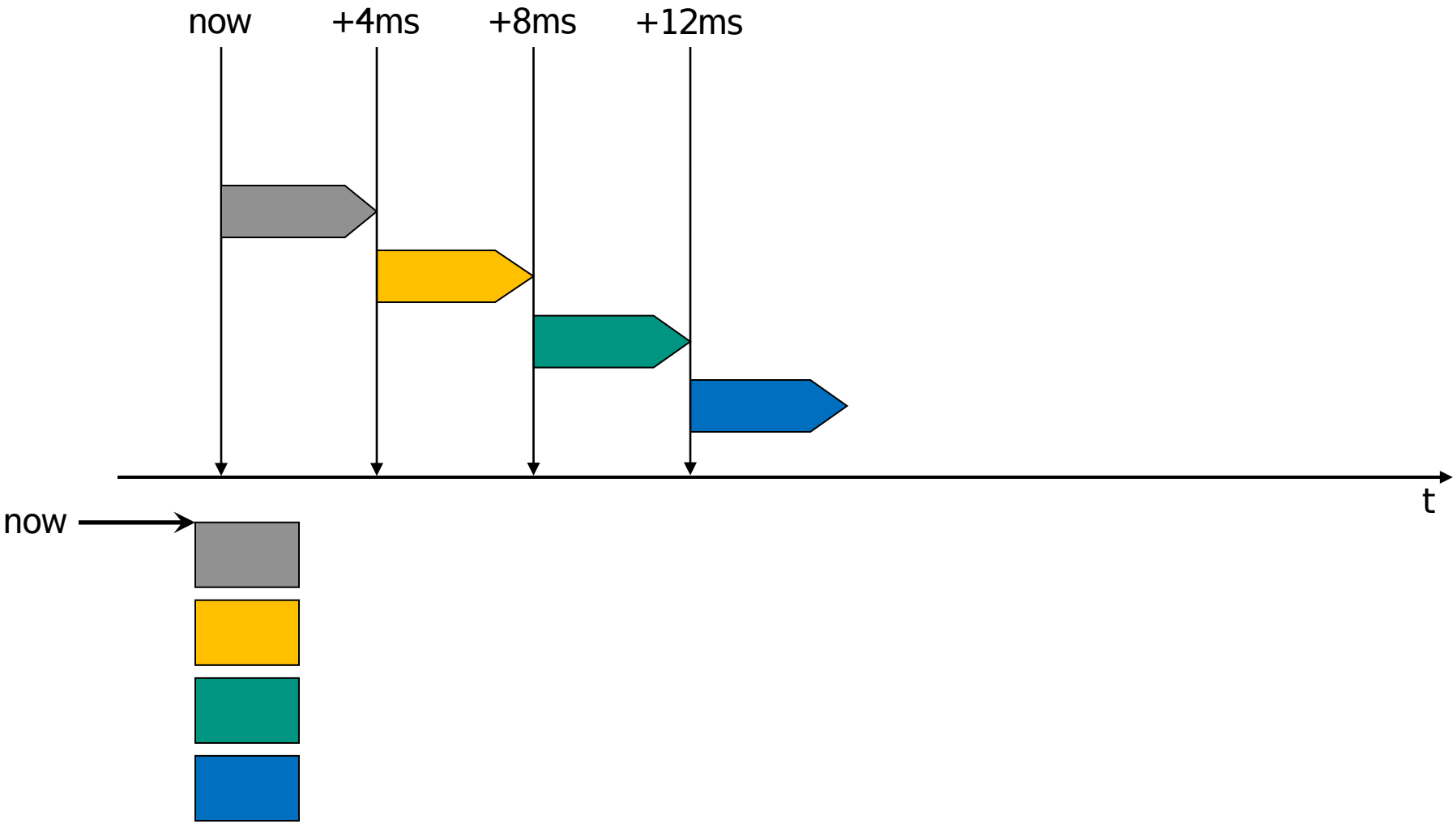




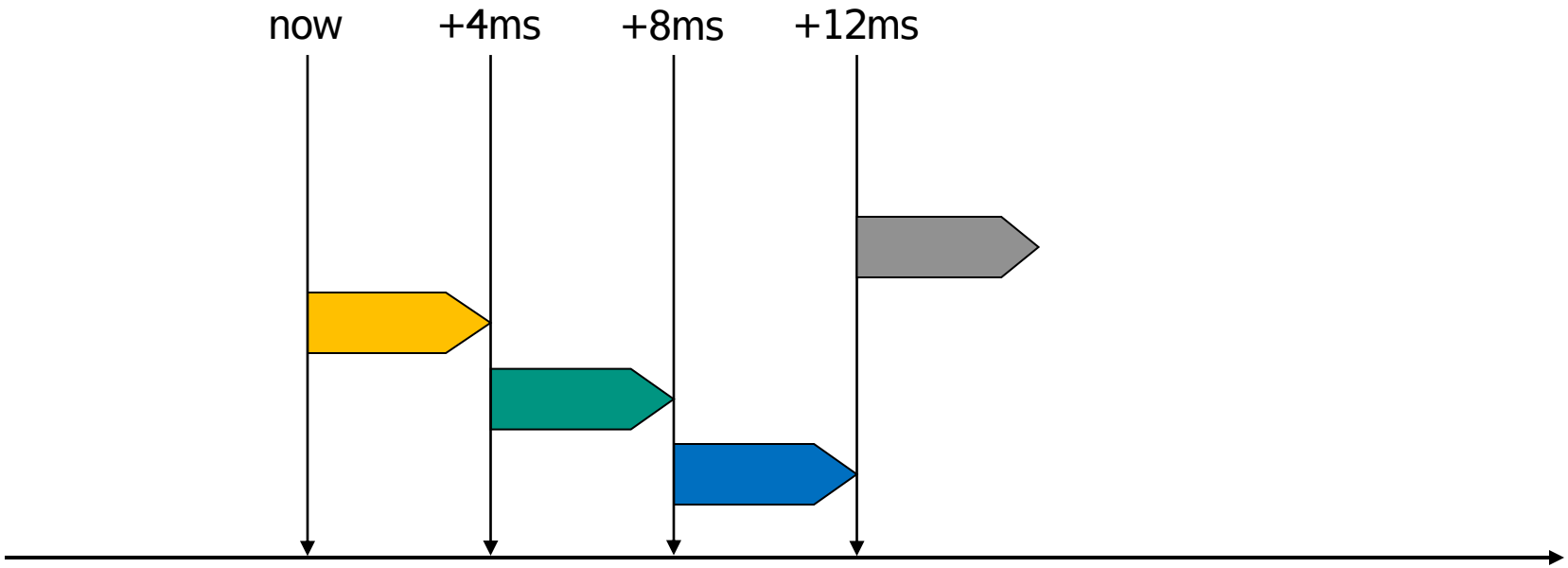
- s := max #entries (length of soon list)
  - s ≥ #timeouts to be raised in τ_{soon} + new soon timeouts

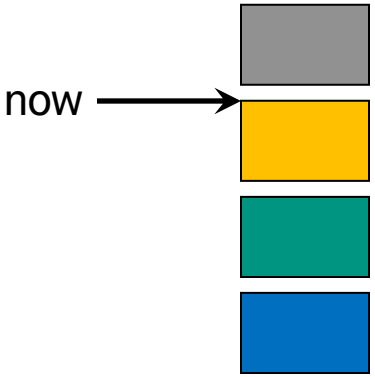
    ⇒ s is small if τ_{soon} is short enough

Absolute timeout classes (Fiasco.OC)



Absolute timeout classes (Fiasco.OC)

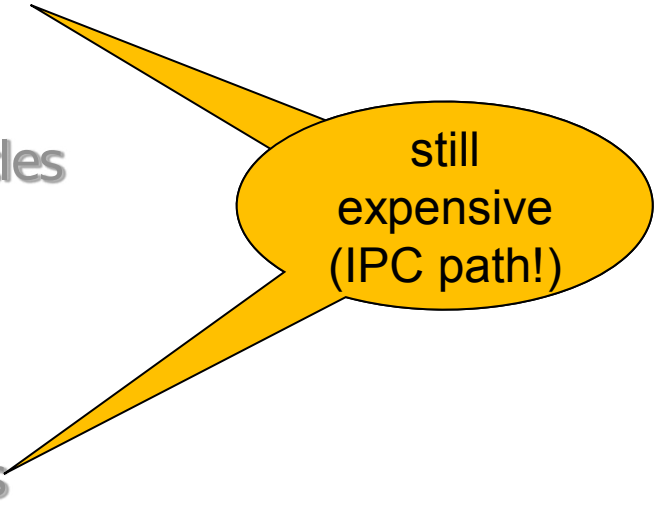




- 8 classes of 4096 µs each by default (configurable)
- Advantages:
  - Typically few, small lists to scan
  - O(1) insert (classes are unsorted linked lists)
  - No copying between soon/late lists, just flip buffers instead
- But: Long timeouts can wrap ➔ Must scan entire lists

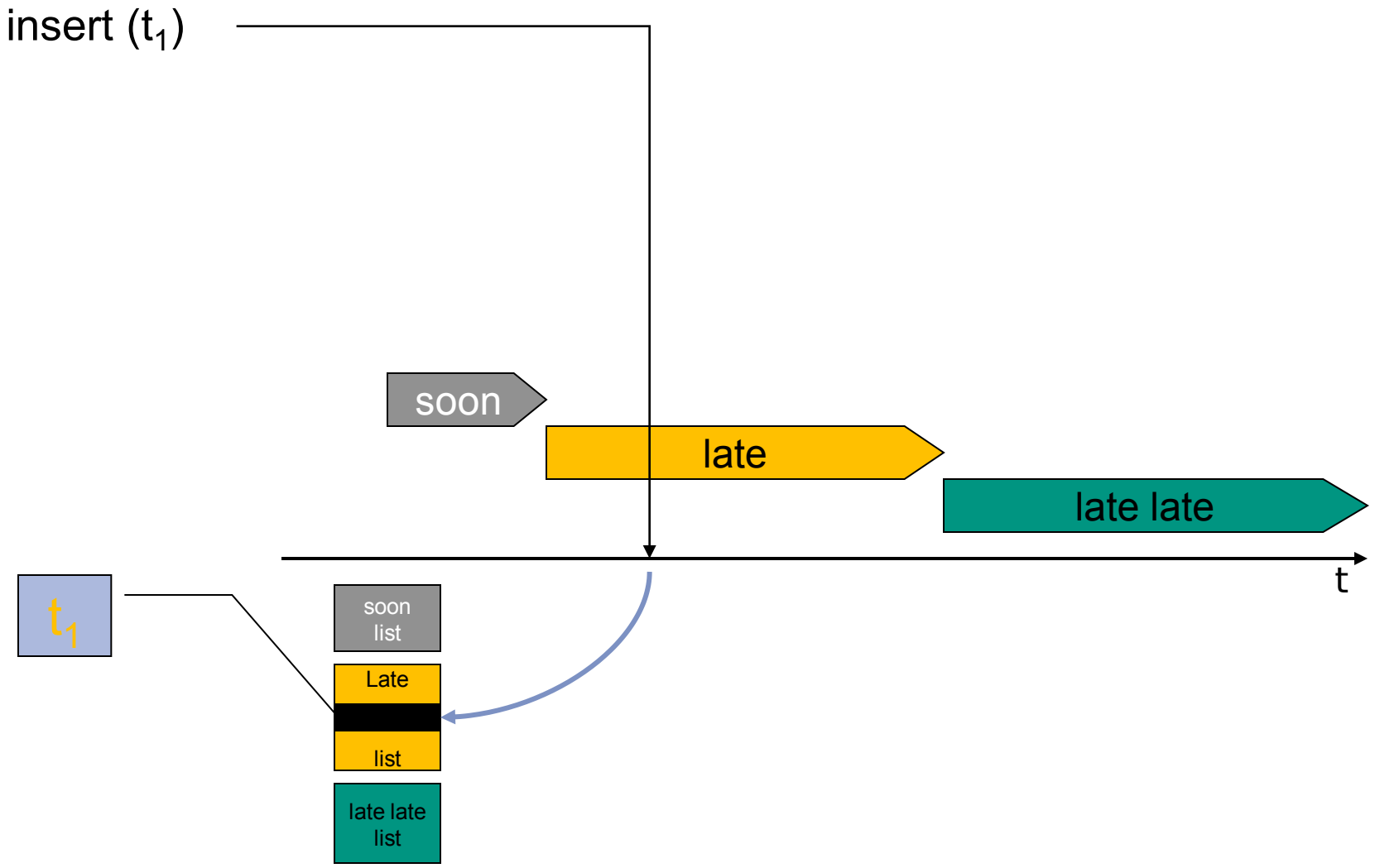
Timeouts & Wakeups

- Idea 4: unsorted wakeup classes
  - *Insert timeout* costs
    - Select class + prepend 10 + 20..100 cycles
  - *Find next timeout* costs
    - Search *soon* class $s \times$ 10..50 cycles
  - *Raise timeout* costs
  - *Delete timeout* costs
    - Delete known entry 20..100 cycles



- Raised-timeout costs are uncritical (occur only after timeout exp time).
- Most timeouts are never raised!

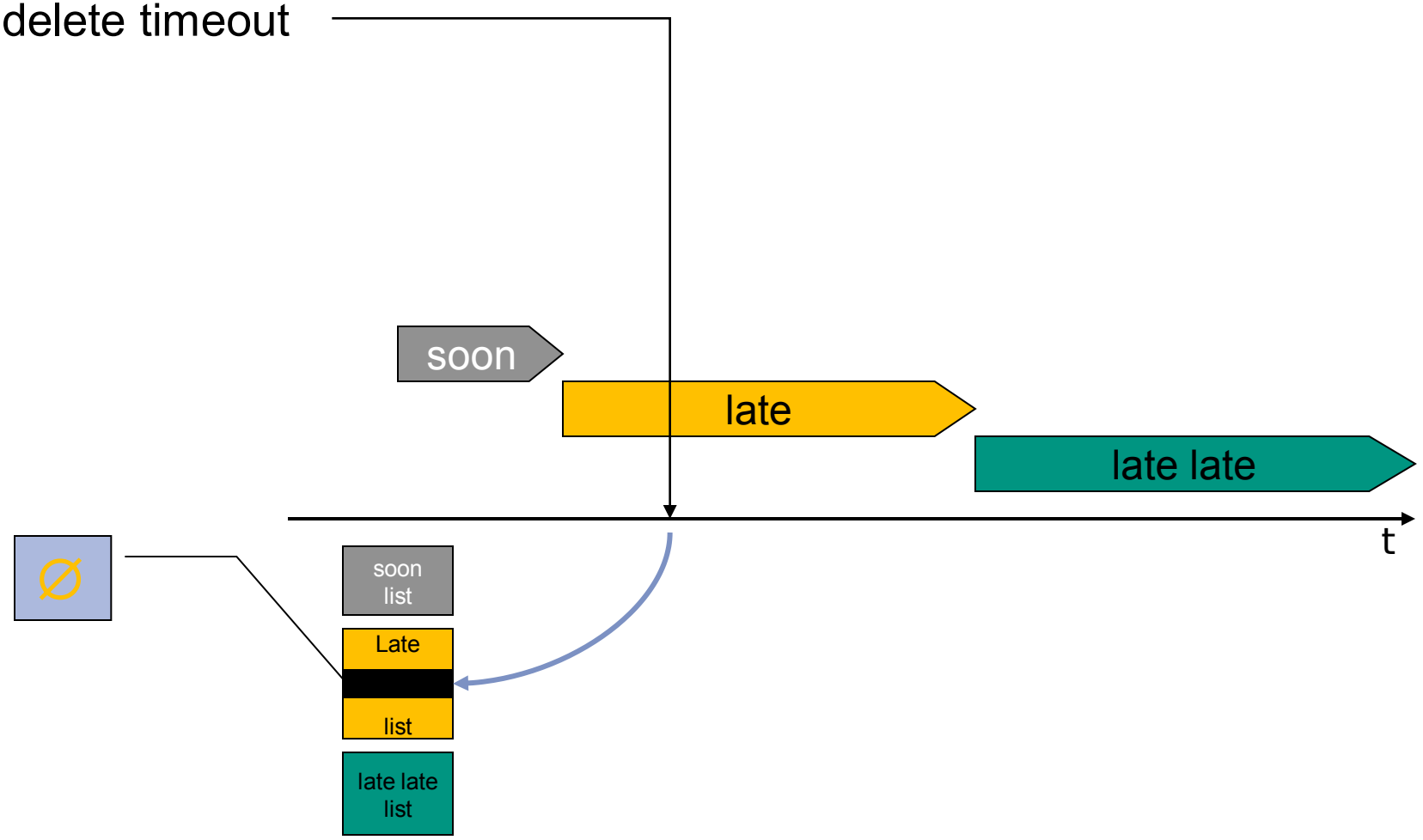
Lazy Timeouts



Lazy Timeouts

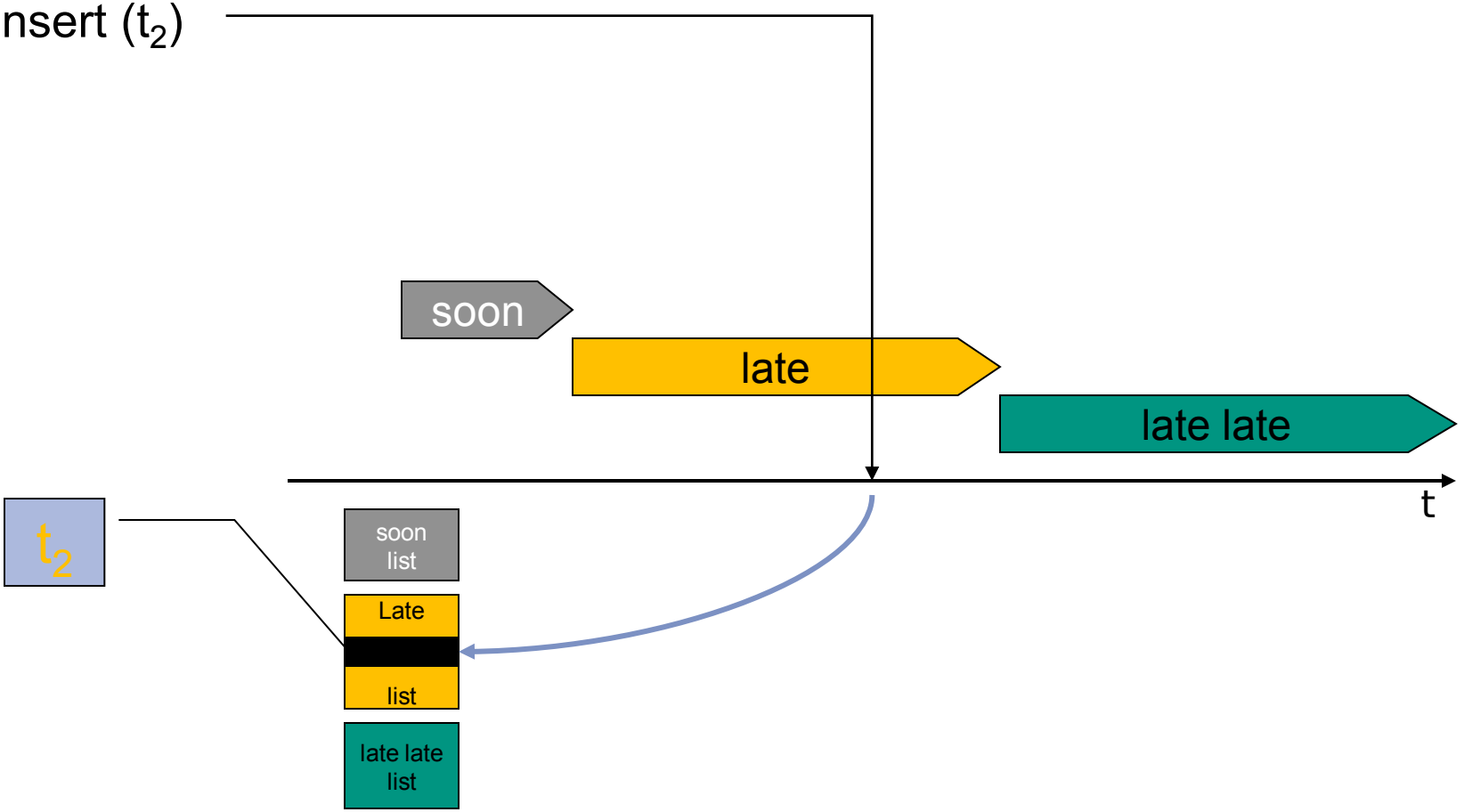
insert (t_1)

delete timeout



Lazy Timeouts

insert (t_1)
delete timeout
insert (t_2)



Summary

- Dispatcher in kernel context
  - Almost stateless
- Static priorities
  - Round-robin per priority
- Ready list updating
  - Lazy ➔ long worst-case scheduling latency
  - Benno scheduling ➔ Slightly increased but predictable latency
- Timeouts
  - Unsorted wakeup classes