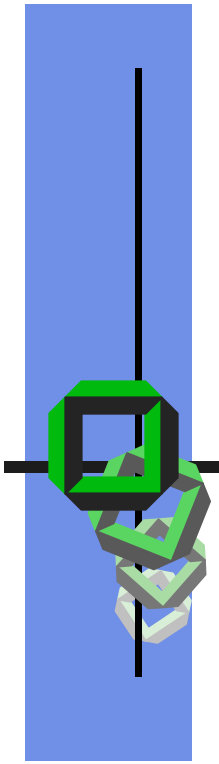
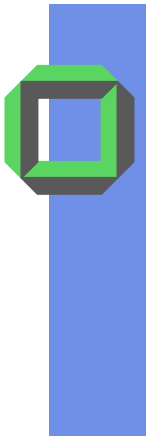


µ-Kernel Construction (6)

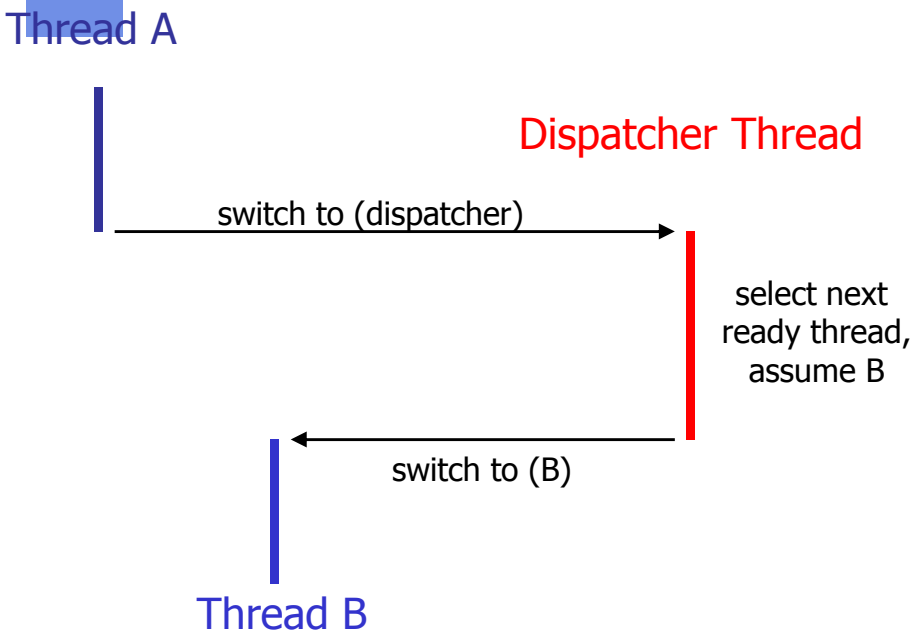
Dispatching



Dispatching Topics

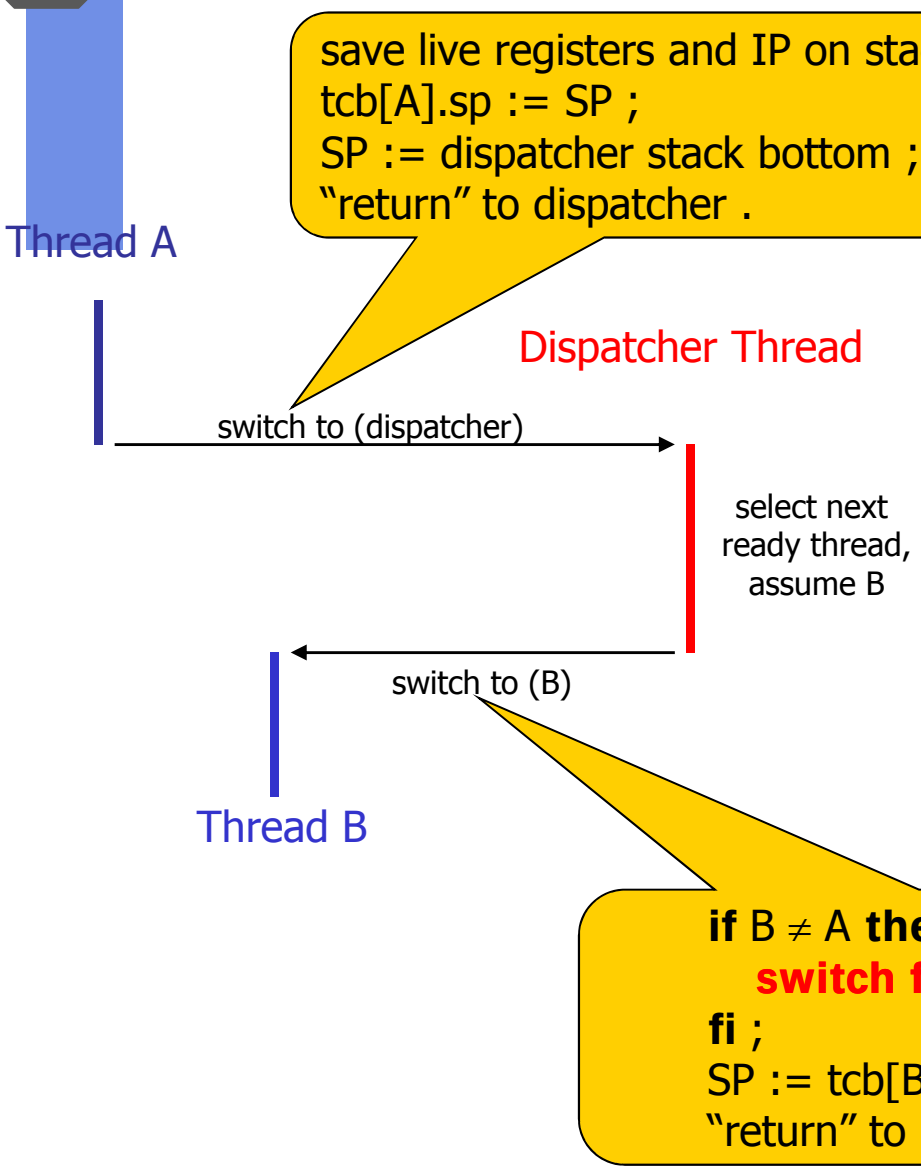
- Thread switch
  - To a specific thread
  - To next thread to be scheduled
    - To nil
    - Implicitly, when IPC blocks
- Priorities
- Preemption
  - Time slices
  - Wakeups, interruptions
- Timeouts and wakeups
  - Time

Switch to ()



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

Switch to ()

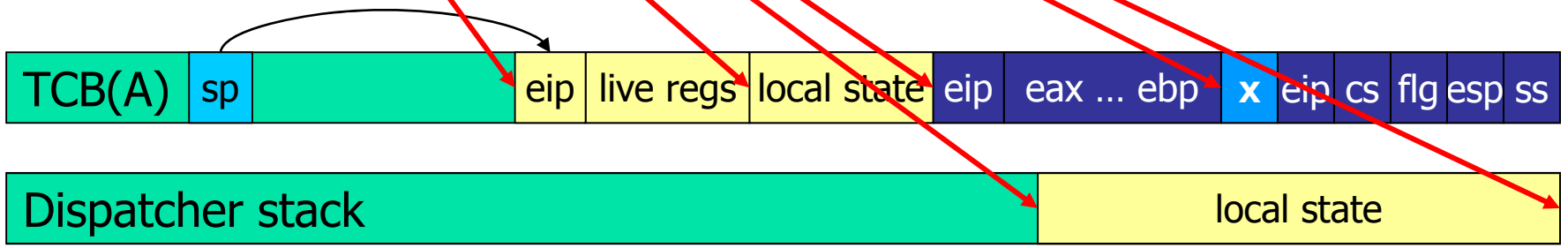


- Dispatcher thread is special
  - No user mode
    - No own AS, hence no AS switch required
    - No ID required
    - Freedom from TCB layout conventions
  - Almost stateless (see *priorities*)
    - No need to preserve internal state between invocations
    - External state must be consistent

- costs (A → B)
- ≈ costs (A → disp → B)
  - costs (select next)
- costs (A → disp → A) are low

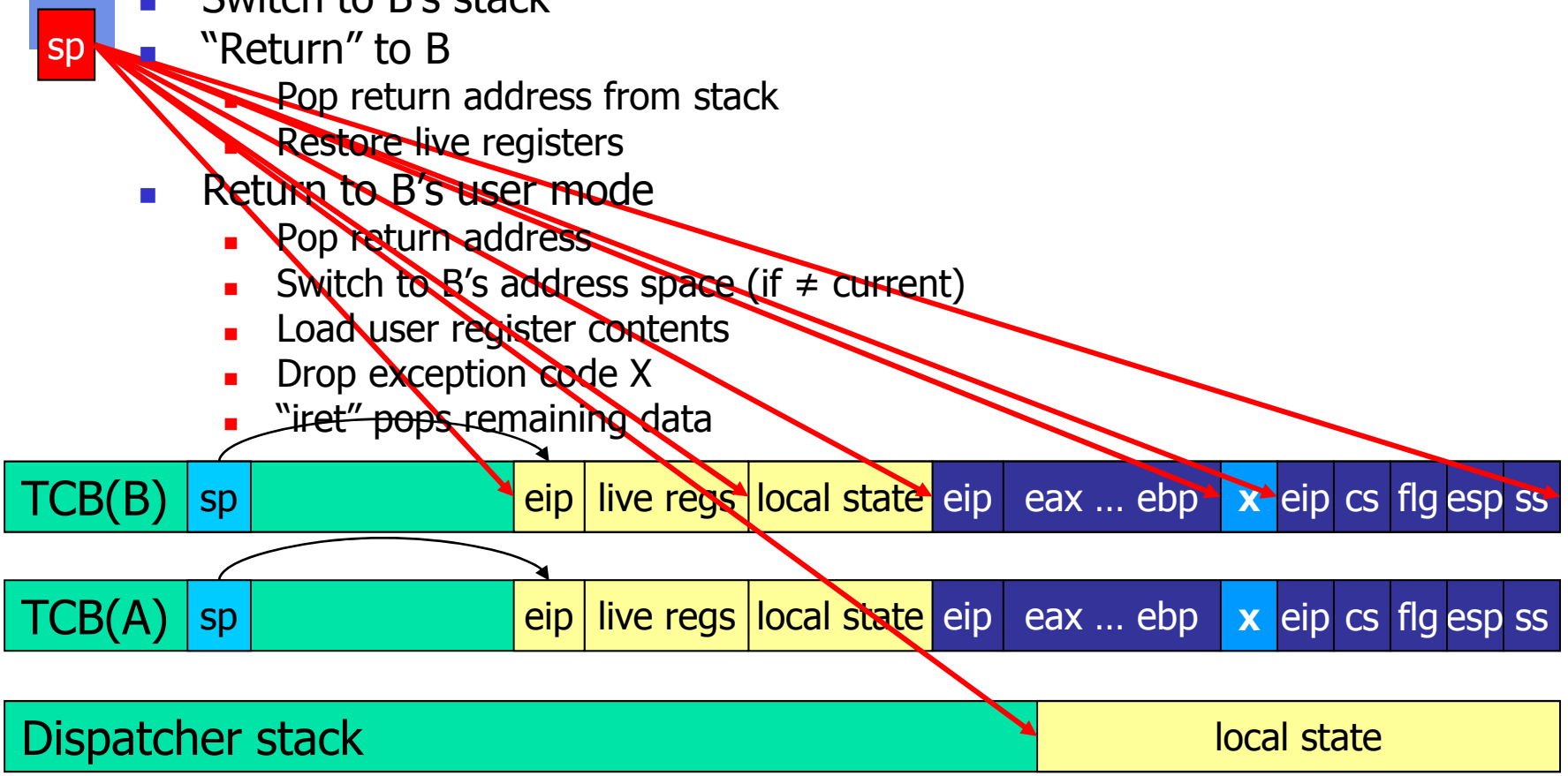
Example: Simple Dispatch

- 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



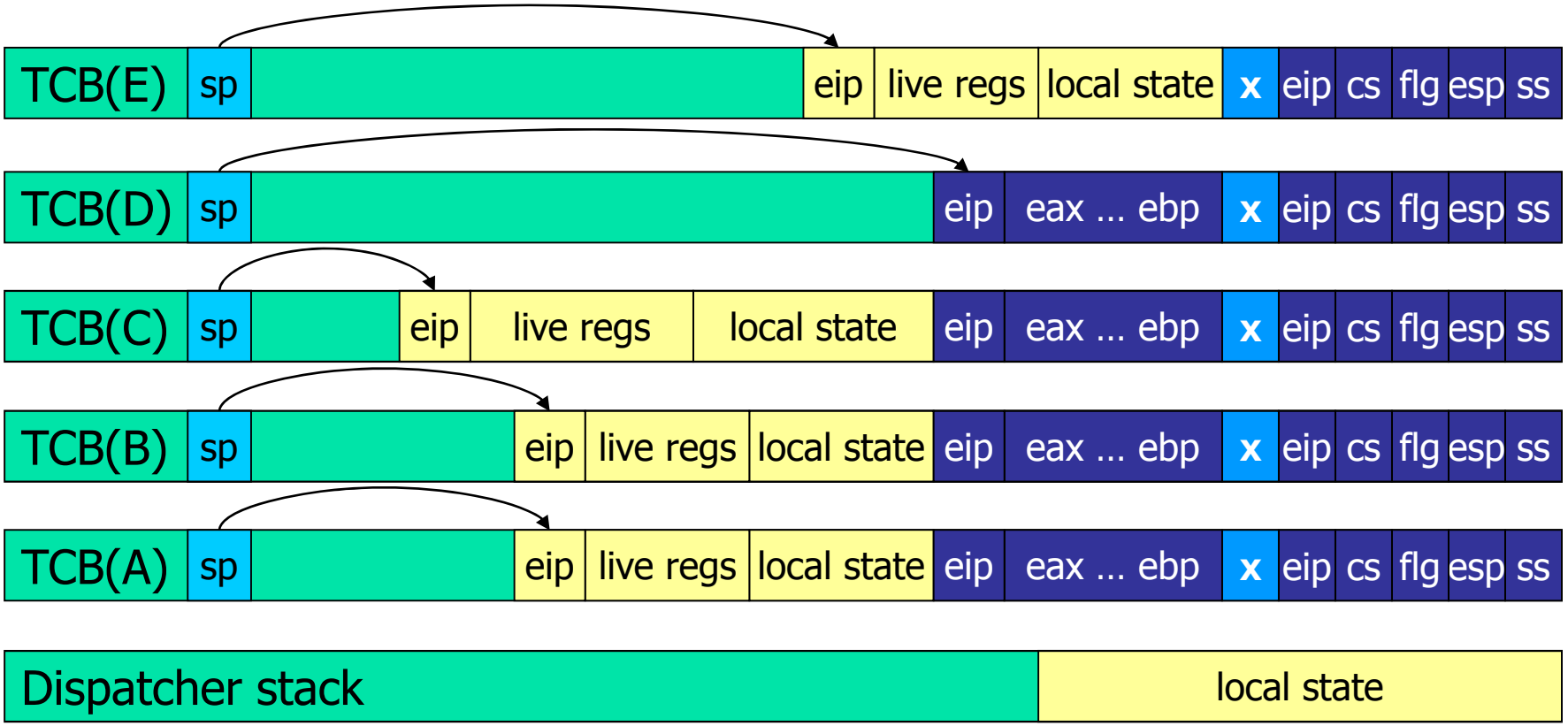
Example: Simple Dispatch (cont'd)

- Dispatcher selected next thread to run (B)
- 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



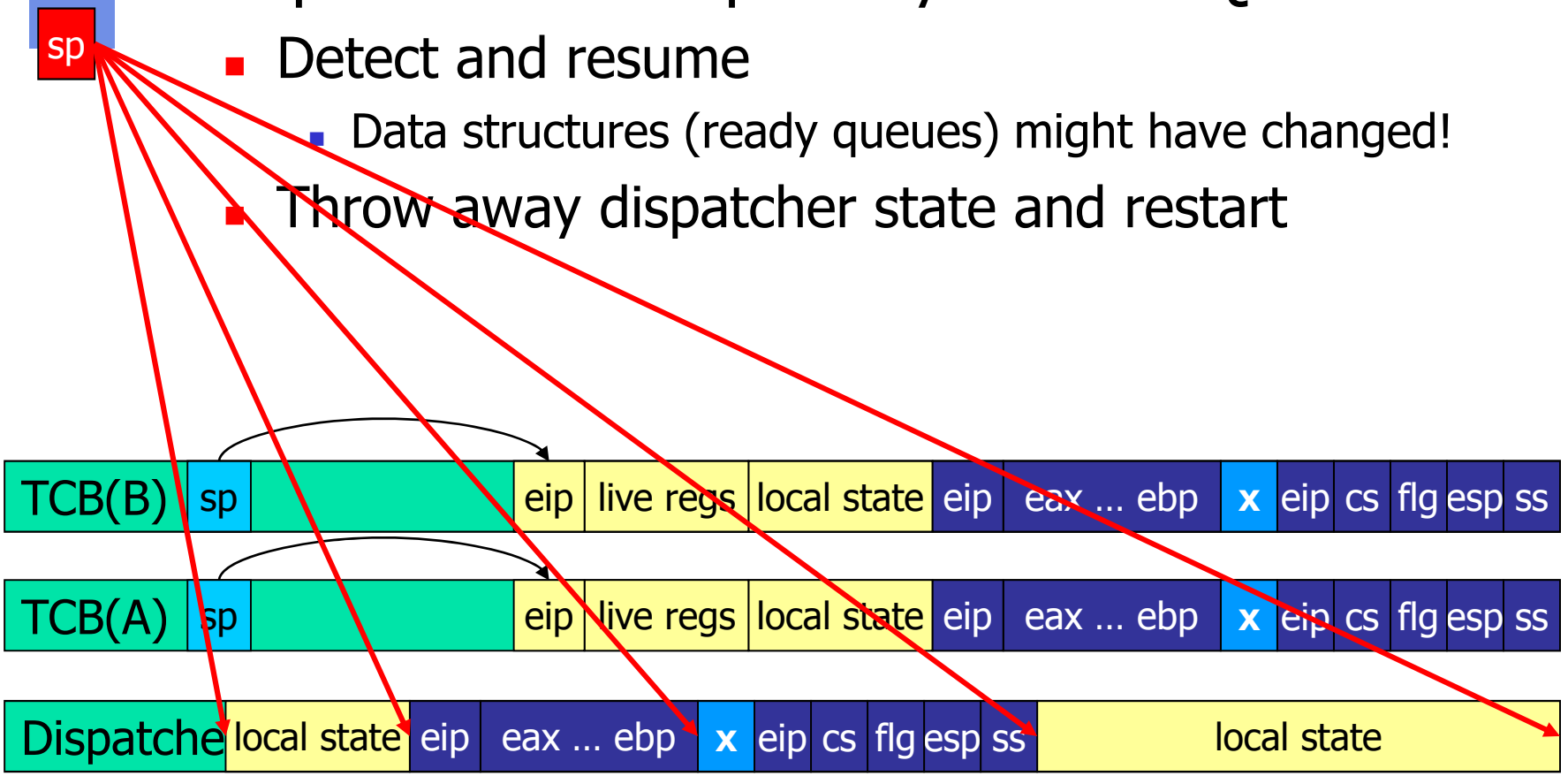
Example: Simple Dispatch (cont'd)

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



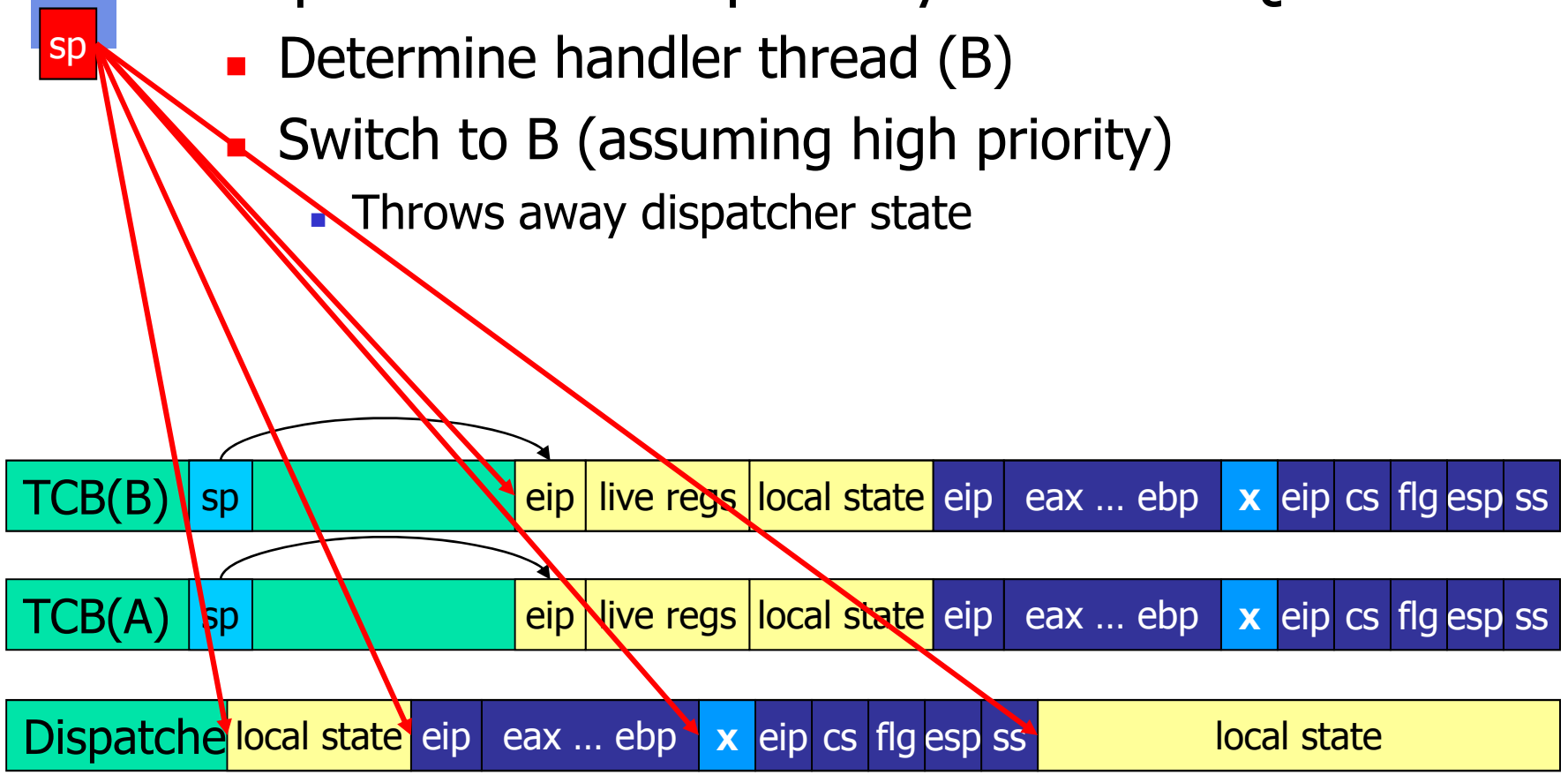
Example: Dispatch with `Tick'

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



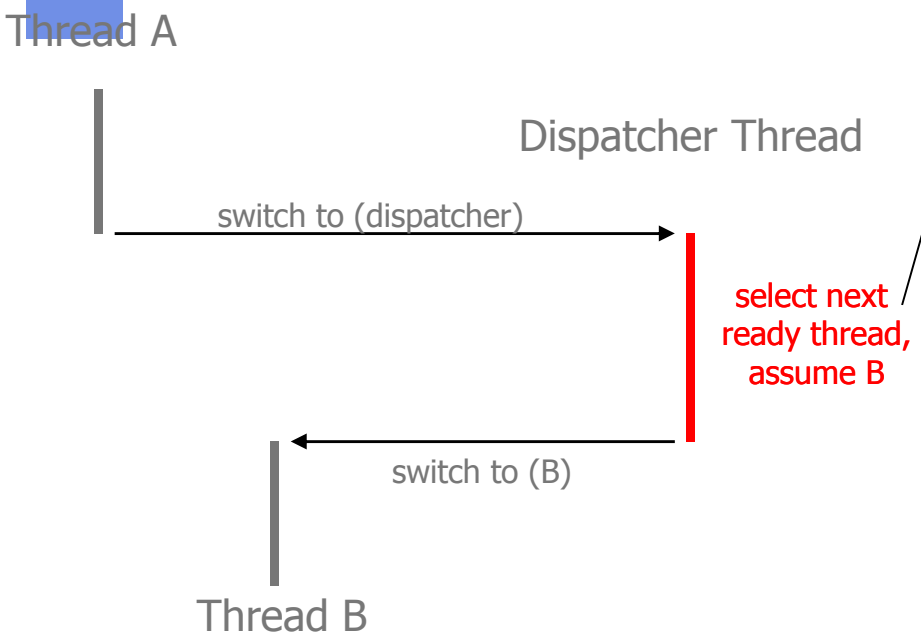
Example: Dispatch with Device IRQ

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

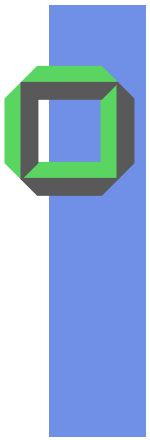


Switch to ()

- Dispatcher thread is also **idle thread**




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

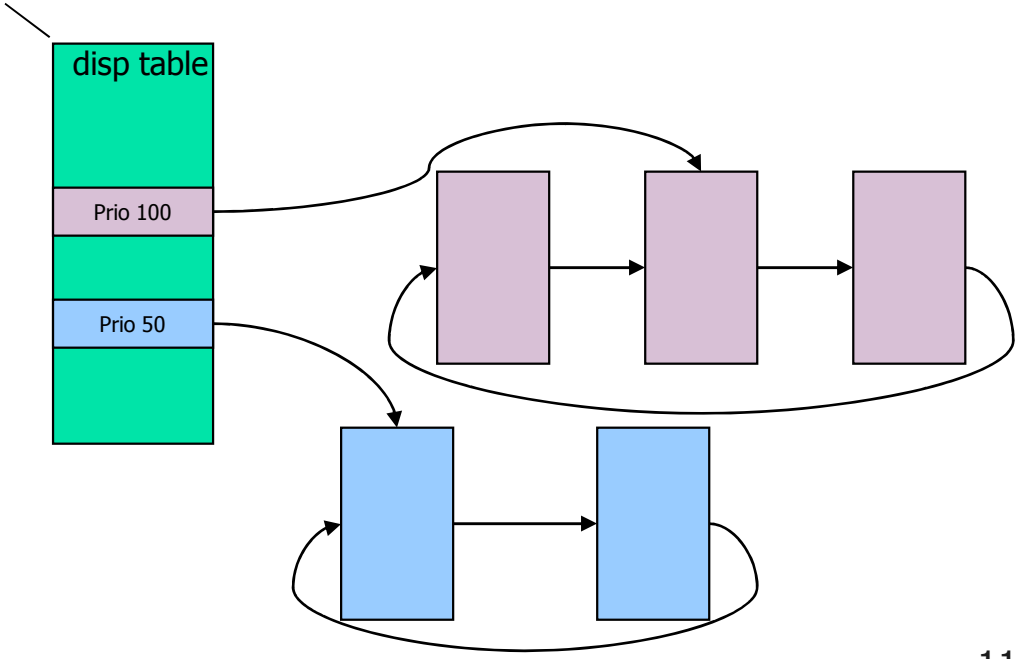


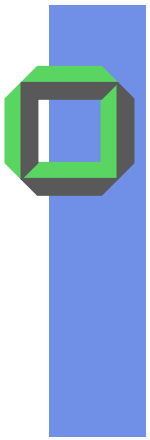
Priorities

- 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 .
```

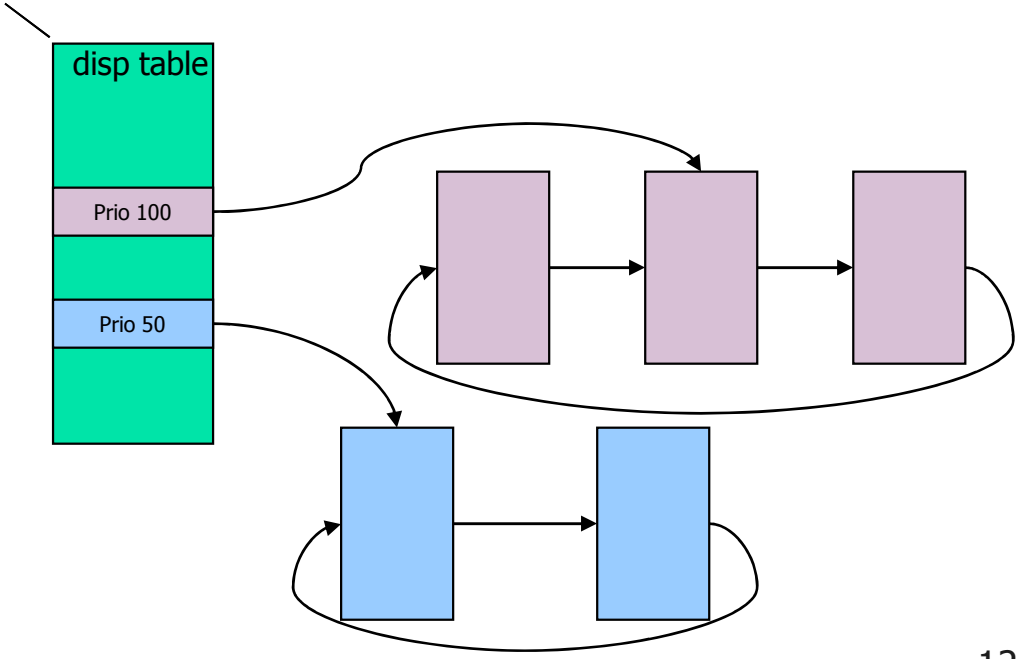


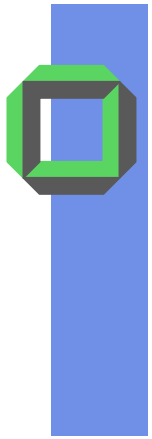


Priorities

- Optimizations
  - Remember highest active priority
  - Bitmask

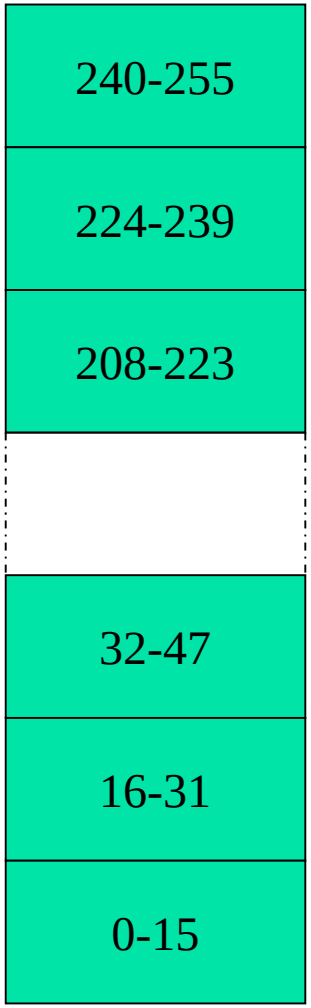
```
do
   if current[highest active p] ≠ nil then
      B := current[highest active p] ;
      return
   elif highest active p > 0 then
      highest active p -= 1
   else
      idle
   fi
od .
```





Optimization: Priorities

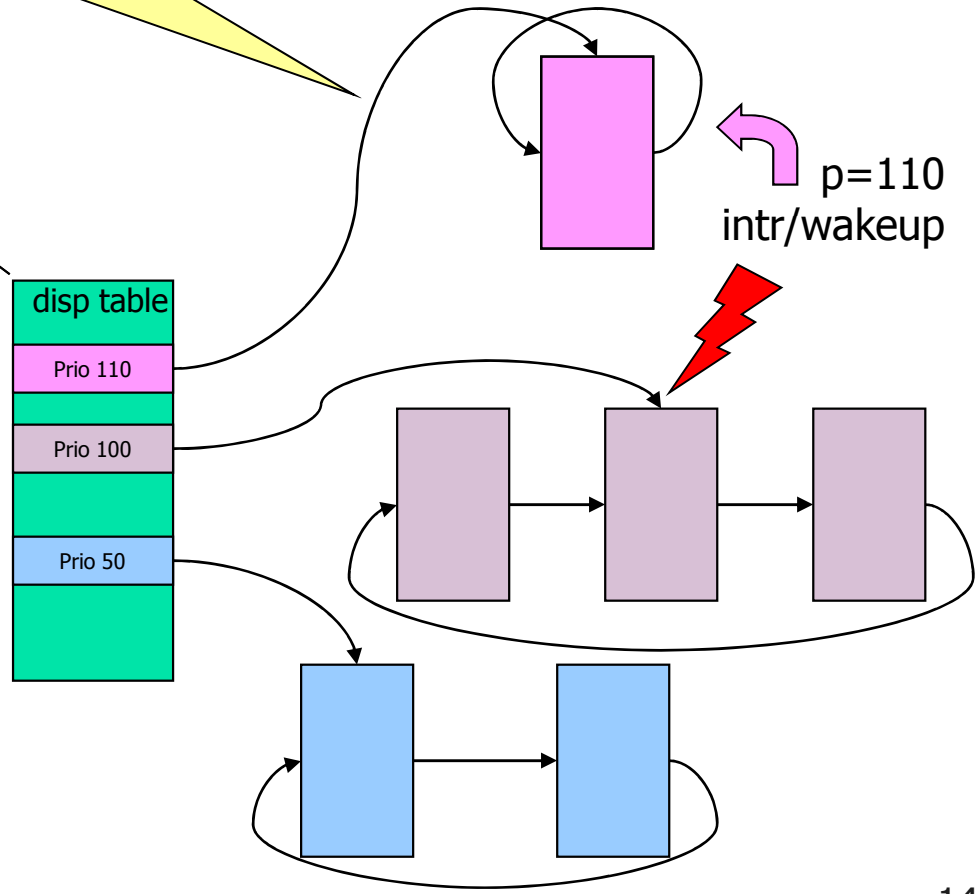
- Bitmap
  - Set bit on insertion
  - Clear when group empty
- IA-32: **BSR**
  - **BIT SCAN REVERSE**
  - 2 cycles



Priorities, Preemption

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

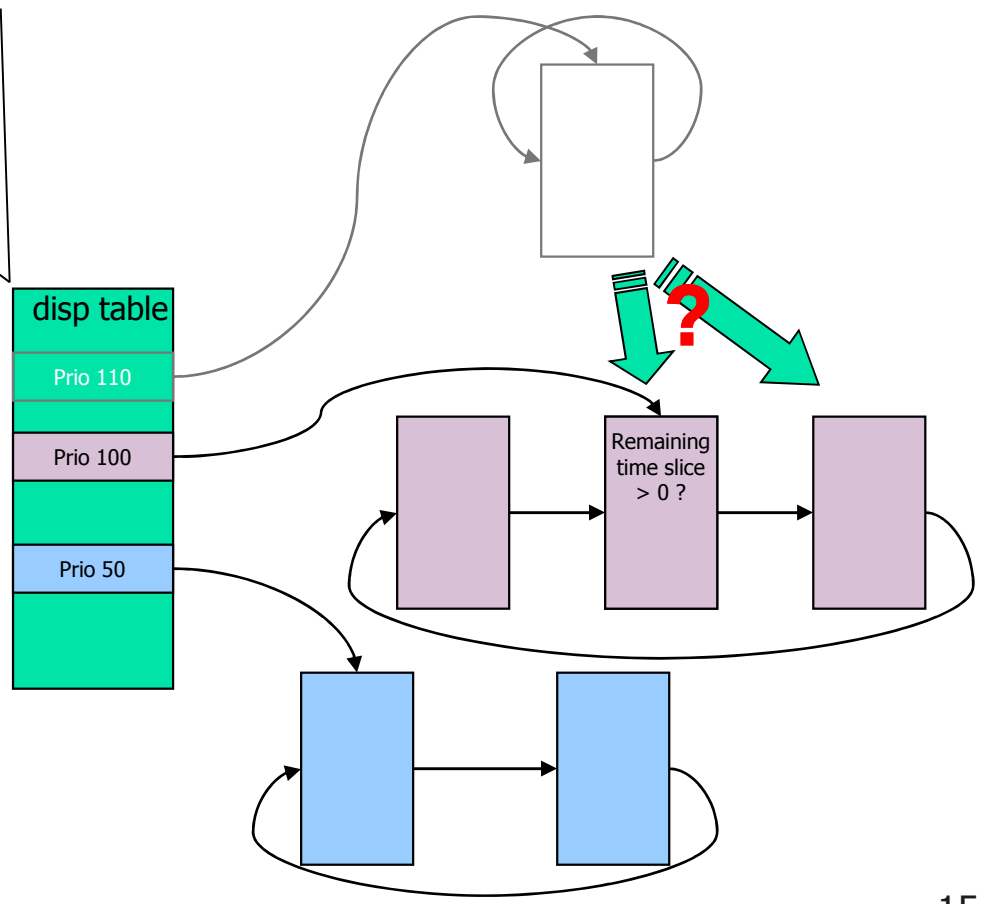
```
do
   if current[highest active p] ≠ nil then
      B := current[highest active p] ;
      return
   elif highest active p > 0 then
      highest active p -= 1
   else
      idle
   fi
od .
```



Priorities, Preemption

- What happens when a priority falls empty?

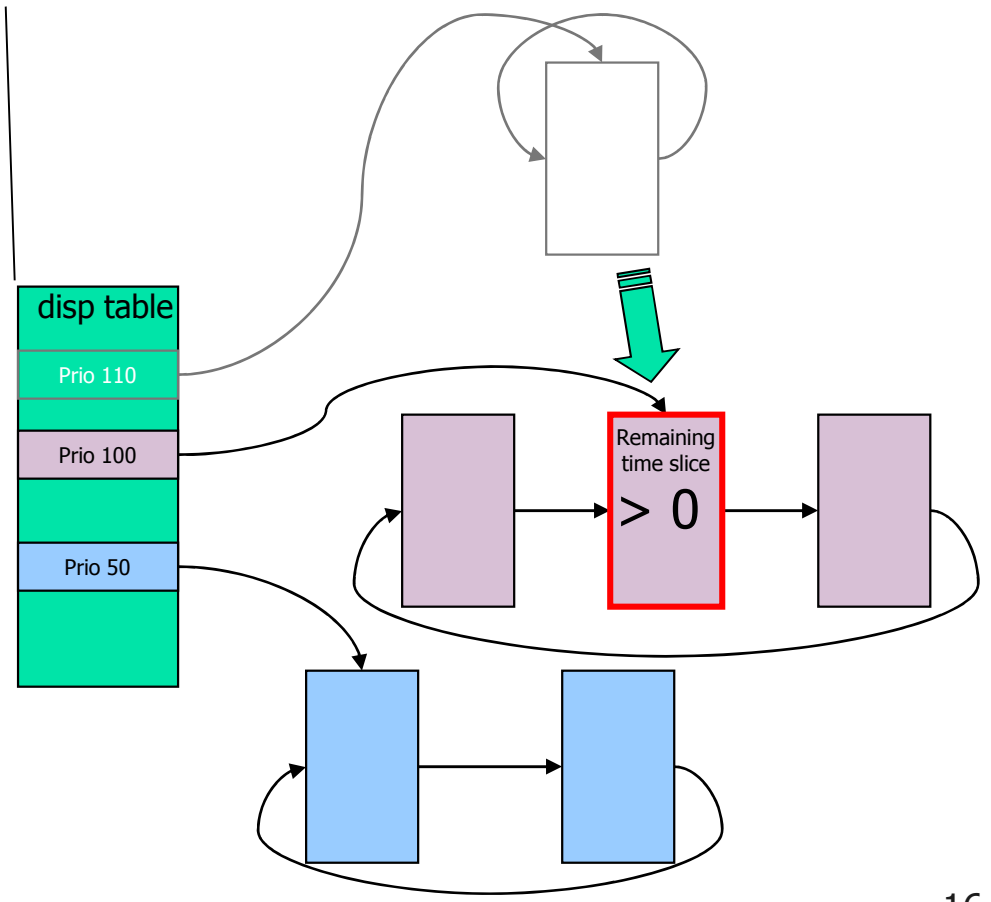
```
do
   if current[highest active p] ≠ nil then
      round robin if necessary ;
      B := current[highest active p] ;
      return
   elif highest active p > 0 then
      highest active p -= 1
   else
      idle
   fi
od .

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



Priorities, Preemption

- What happens when a priority falls empty?

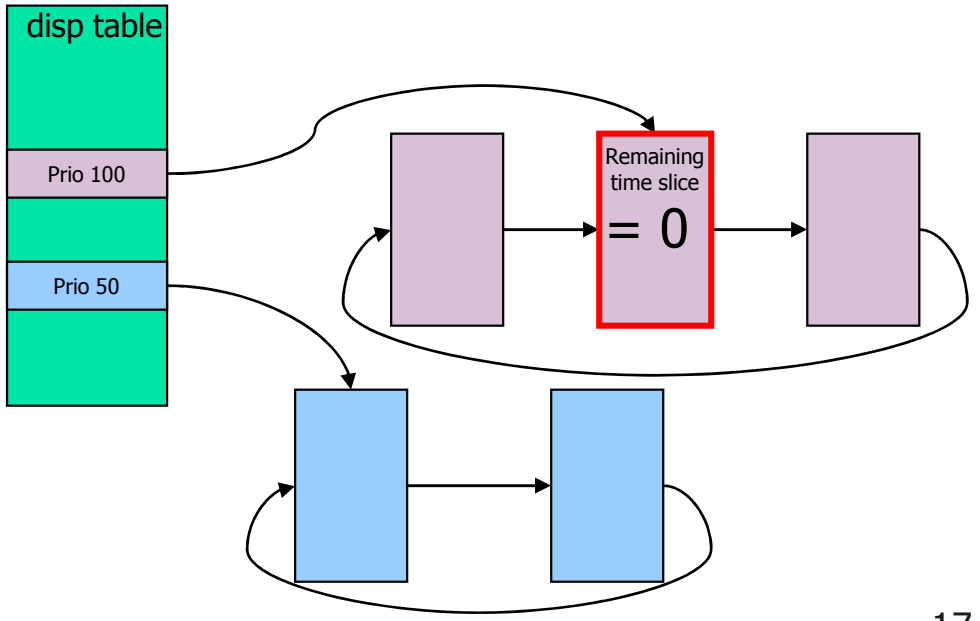
```
do
   if current[highest active p] ≠ nil then
      round robin if necessary ;
      B := current[highest active p] ;
      return
   elif highest active p > 0 then
      highest active p -= 1
   else
      idle
   fi
od .

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



Preemption

- Preemption, time slice exhausted

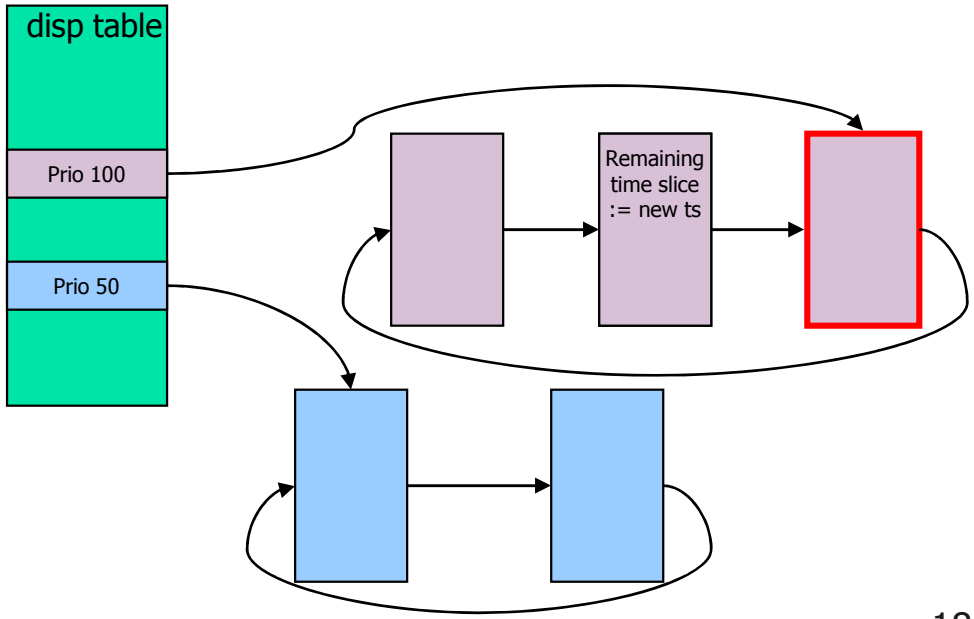
```
do
   if current[highest active p] ≠ nil then
      round robin if necessary ;
      B := current[highest active p] ;
      return
   elif highest active p > 0 then
      highest active p -= 1
   else
      idle
   fi
od .

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



Preemption

- Preemption, time slice exhausted

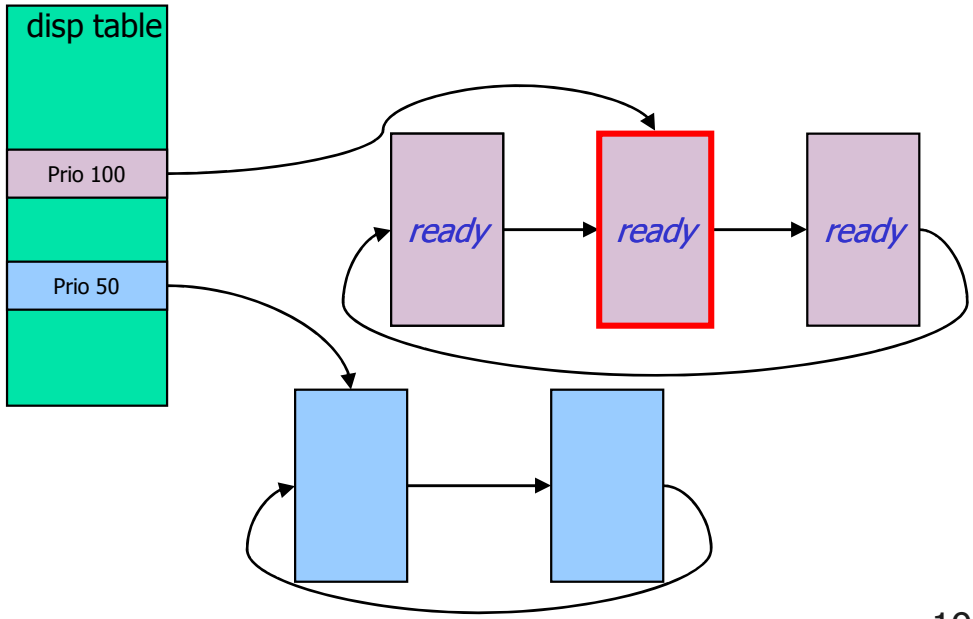
```
do
   if current[highest active p] ≠ nil then
      round robin if necessary ;
      B := current[highest active p] ;
      return
   elif highest active p > 0 then
      highest active p -= 1
   else
      idle
   fi
od .

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



Lazy Dispatching

Thread state toggles frequently (per IPC)

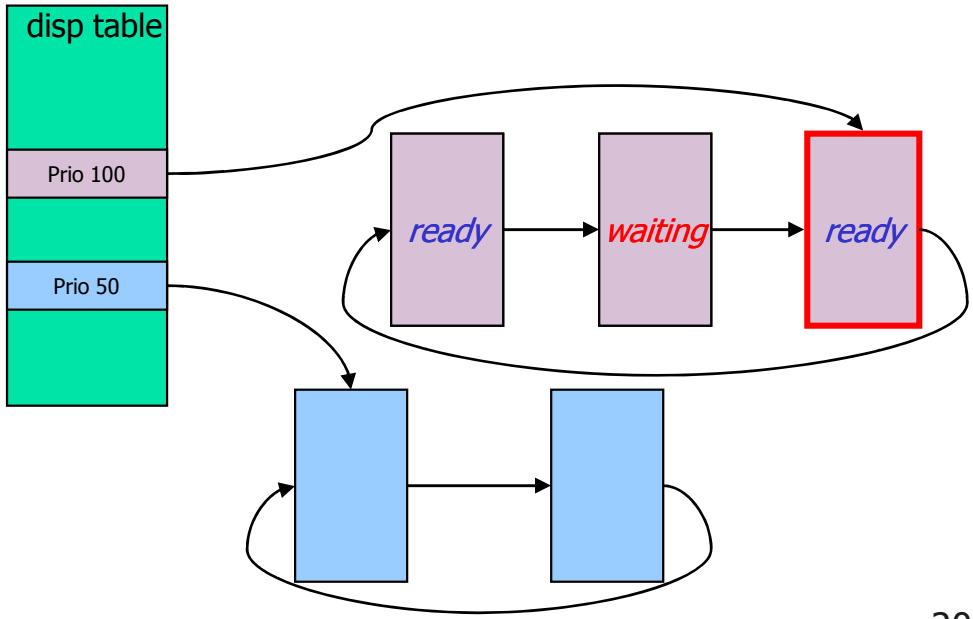
- *ready* ↔ *waiting*
  - Delete/insert ready list is expensive
  - Therefore: delete *lazily* from ready list



Lazy Dispatching

Thread state toggles frequently (per IPC)

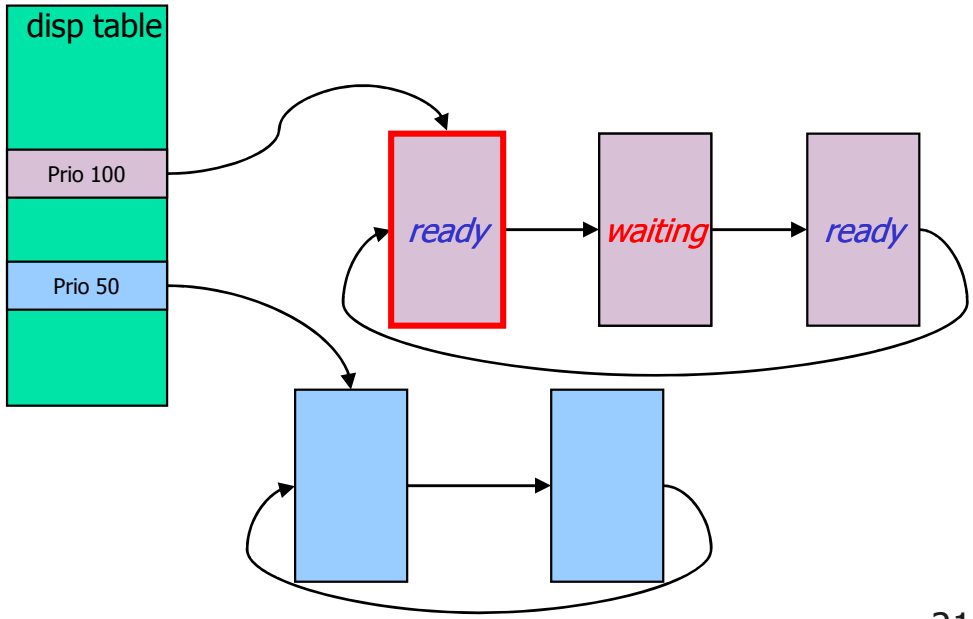
- *ready* ↔ *waiting*
  - Delete/insert ready list is expensive
  - Therefore: delete *lazily* from ready list



Lazy Dispatching

Thread state toggles frequently (per IPC)

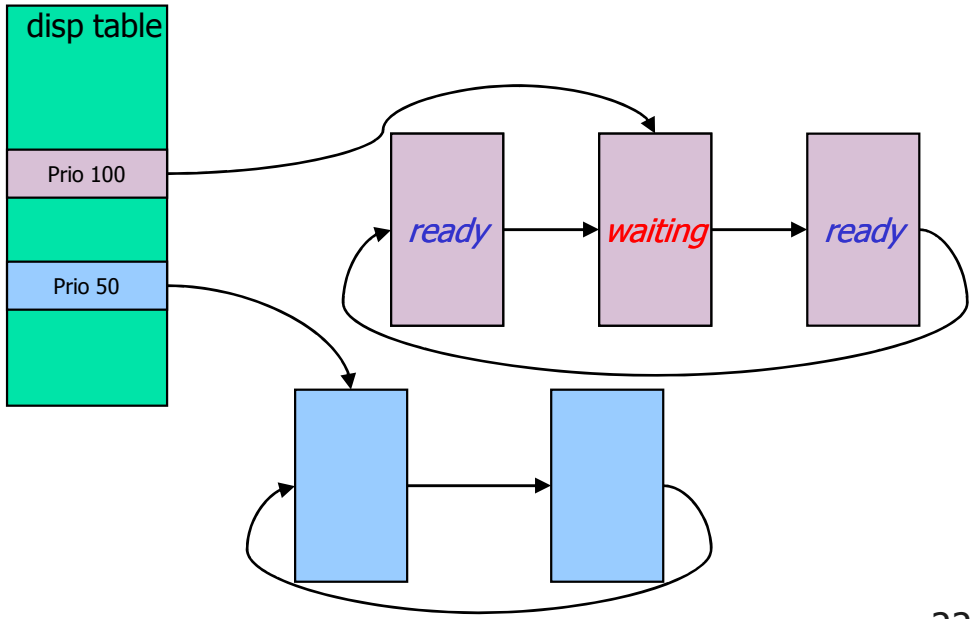
- *ready* ↔ *waiting*
  - Delete/insert ready list is expensive
  - Therefore: delete *lazily* from ready list



Lazy Dispatching

Thread state toggles frequently (per IPC)

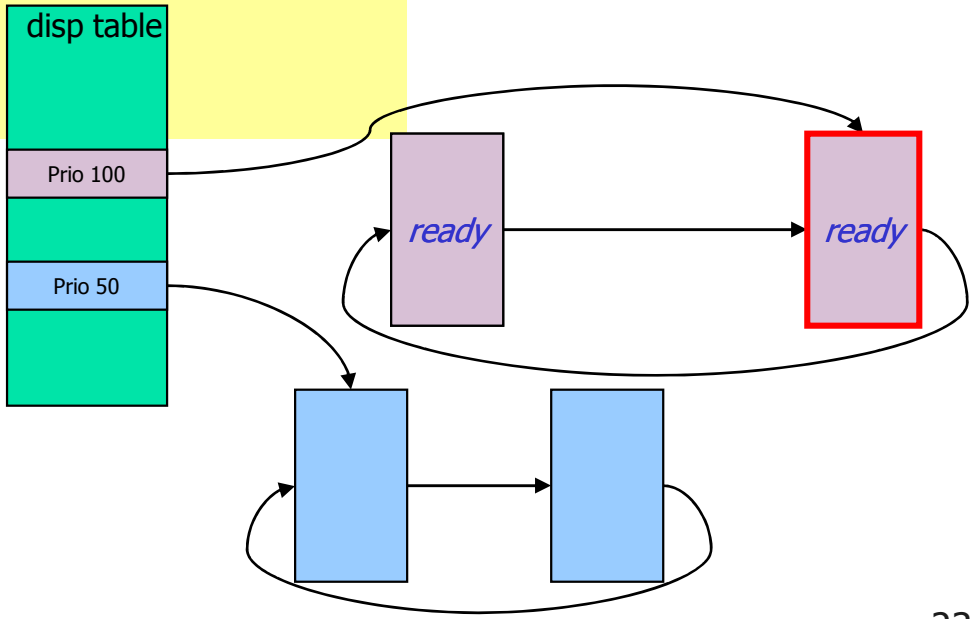
- *ready* ↔ *waiting*
  - Delete/insert ready list is expensive
  - Therefore: delete *lazily* from ready list



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

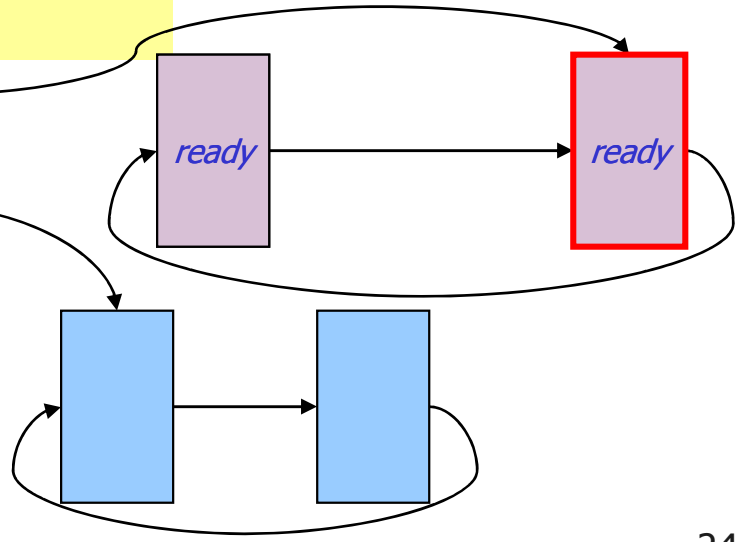
```
do
   round robin if necessary ;
   if current[highest active p] ≠ nil then
      B := current[highest active p] ; return
   elif highest active p > 0 then
      highest active p -= 1
   else
      idle
   fi
od .

round robin if necessary:
   while current[hi act p] ≠ nil do
      next := current[hi act p].next ;
      if current[hi act p].state ≠ ready then
         delete from list (current[hi act p])
      elif current[hi act p].rem ts = 0 then
         current[hi act p].rem ts := new ts
      else
         return
      fi ;
      current[hi act p] := next
   od .
```

(per IPC)

sive

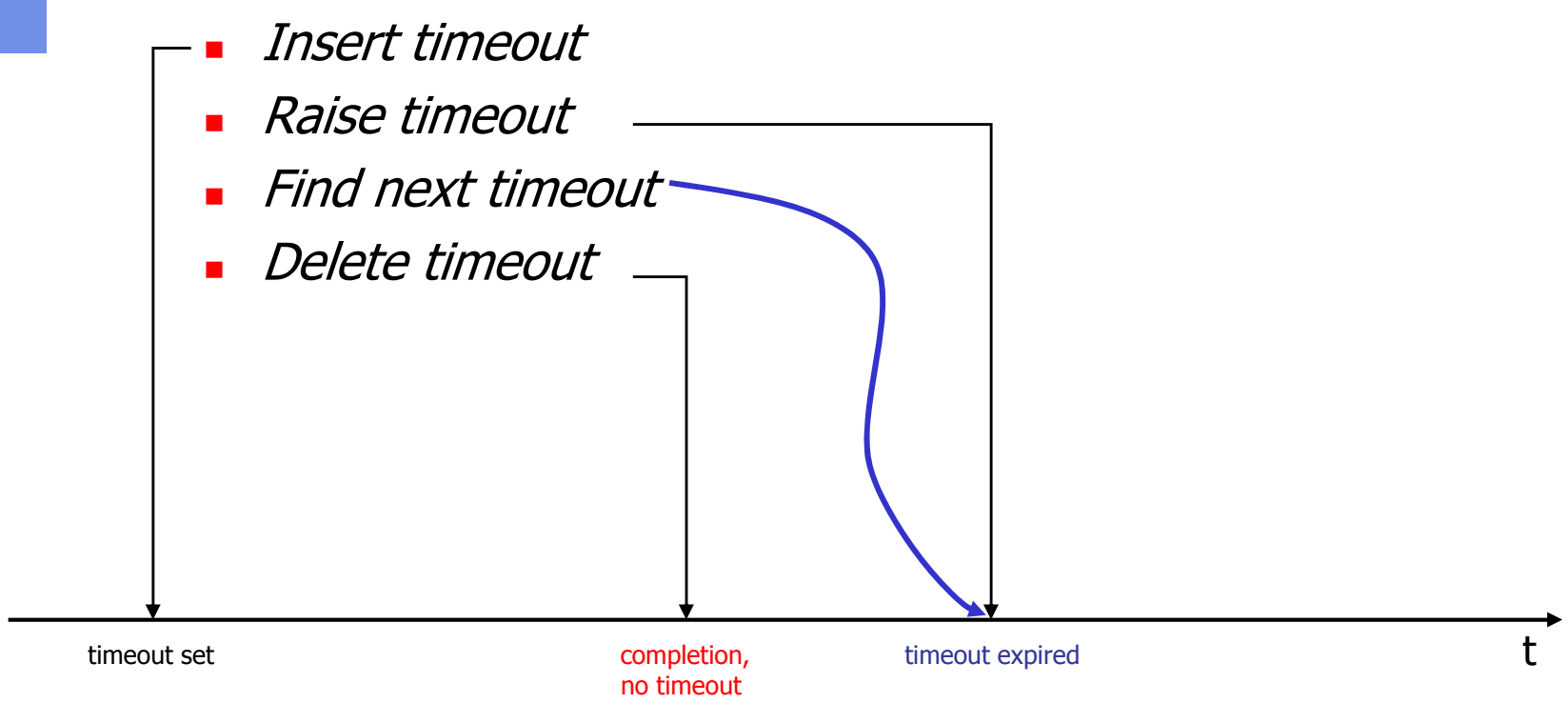
ady list

y thread

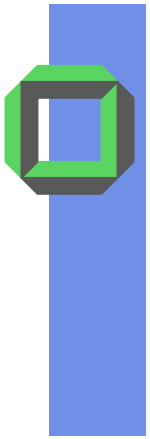


Timeouts & Wakeups

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



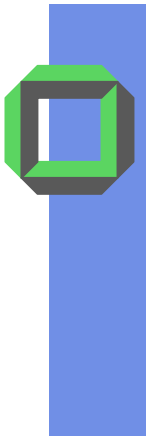
- Raised-timeout costs are uncritical (occur only after timeout exp time).
- Most timeouts are never raised!



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

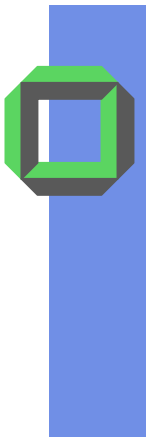
too expensive



Timeouts & Wakeups

too expensive

- 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

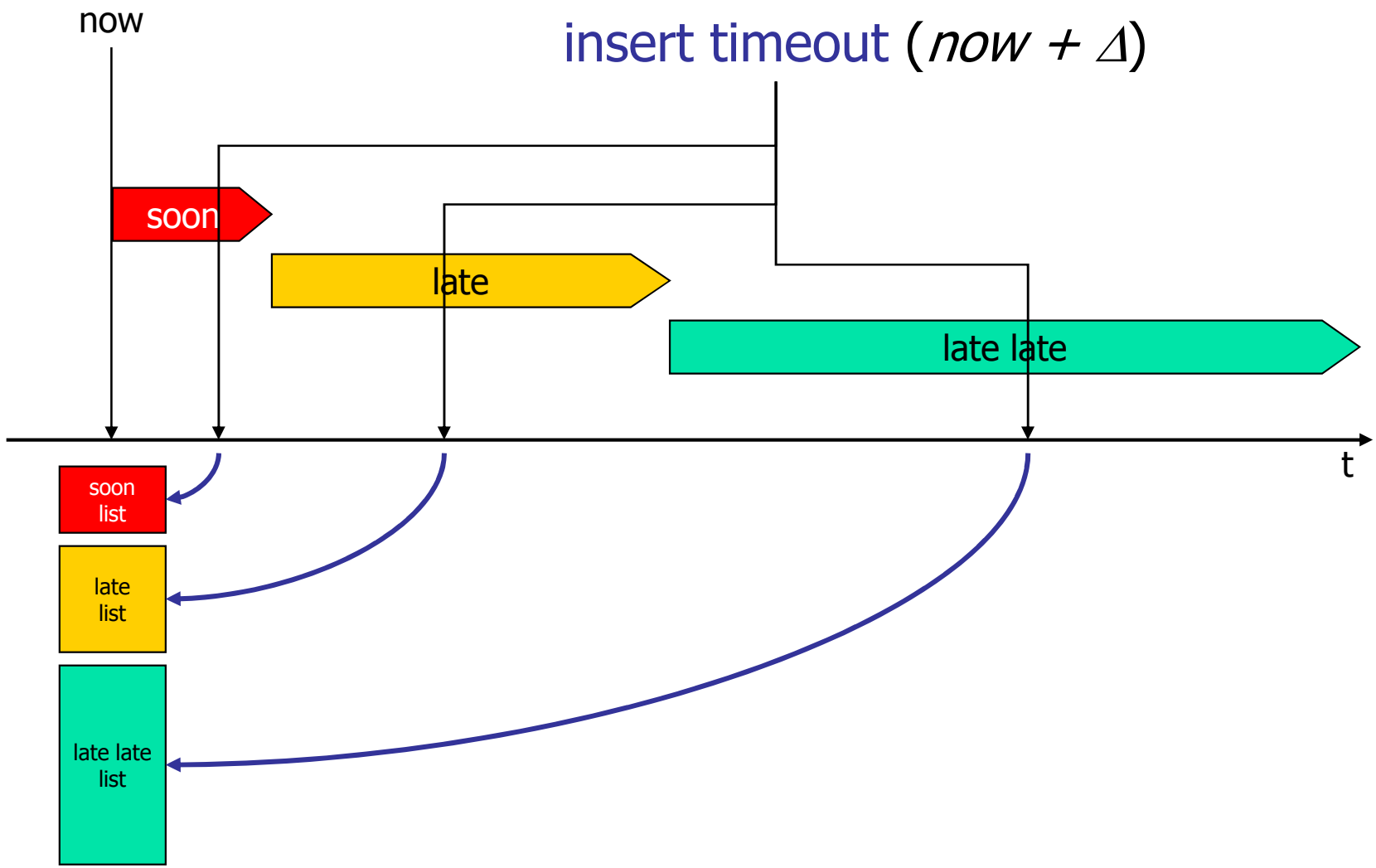


Timeouts & Wakeups

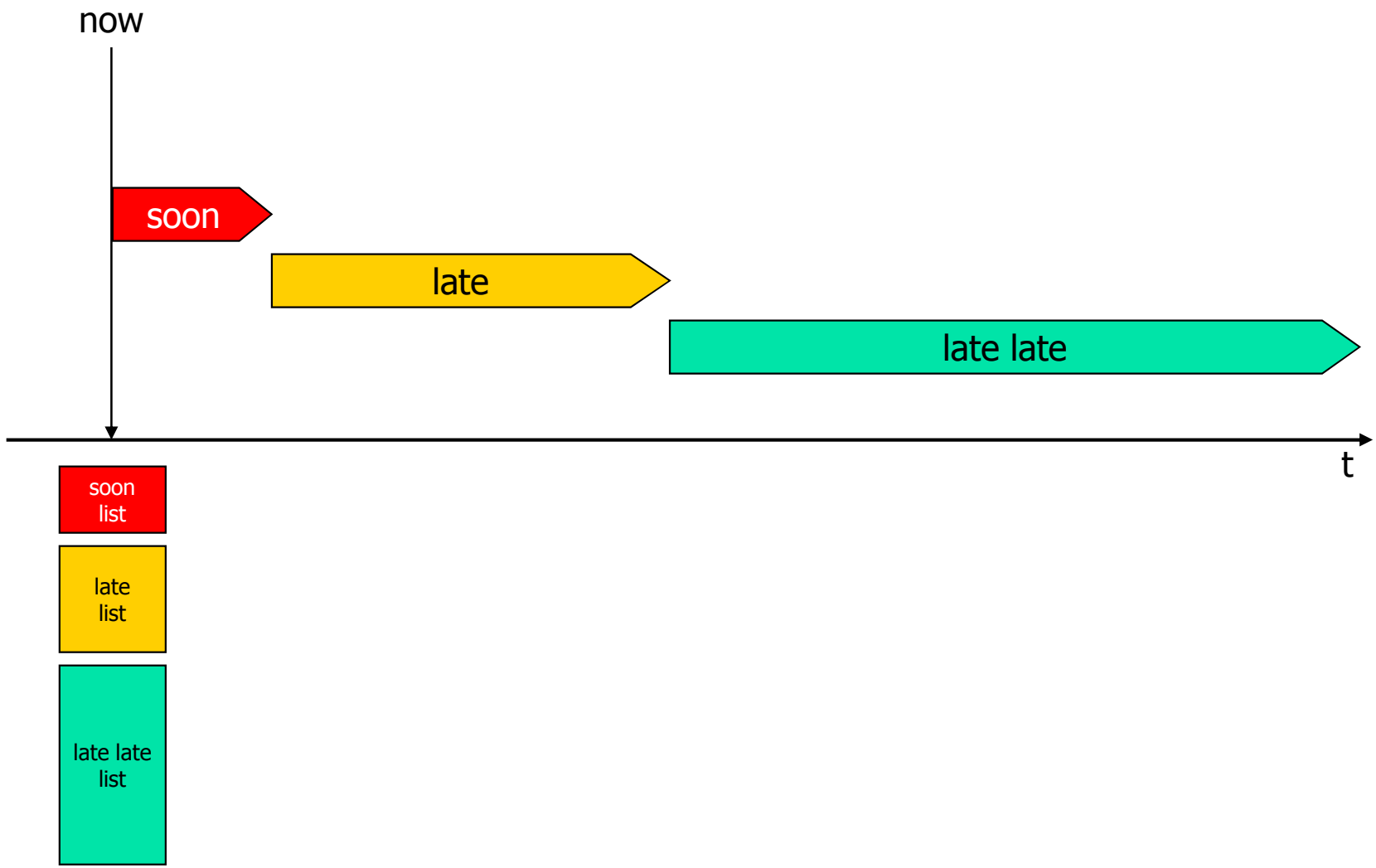
too expensive
too complicated

- 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 x 10..50 / 10 cycles
  - *Raise timeout* costs
  - *Delete timeout* costs
    - Delete known node — log n x 10..50 + 20..100 / log n x 20..100

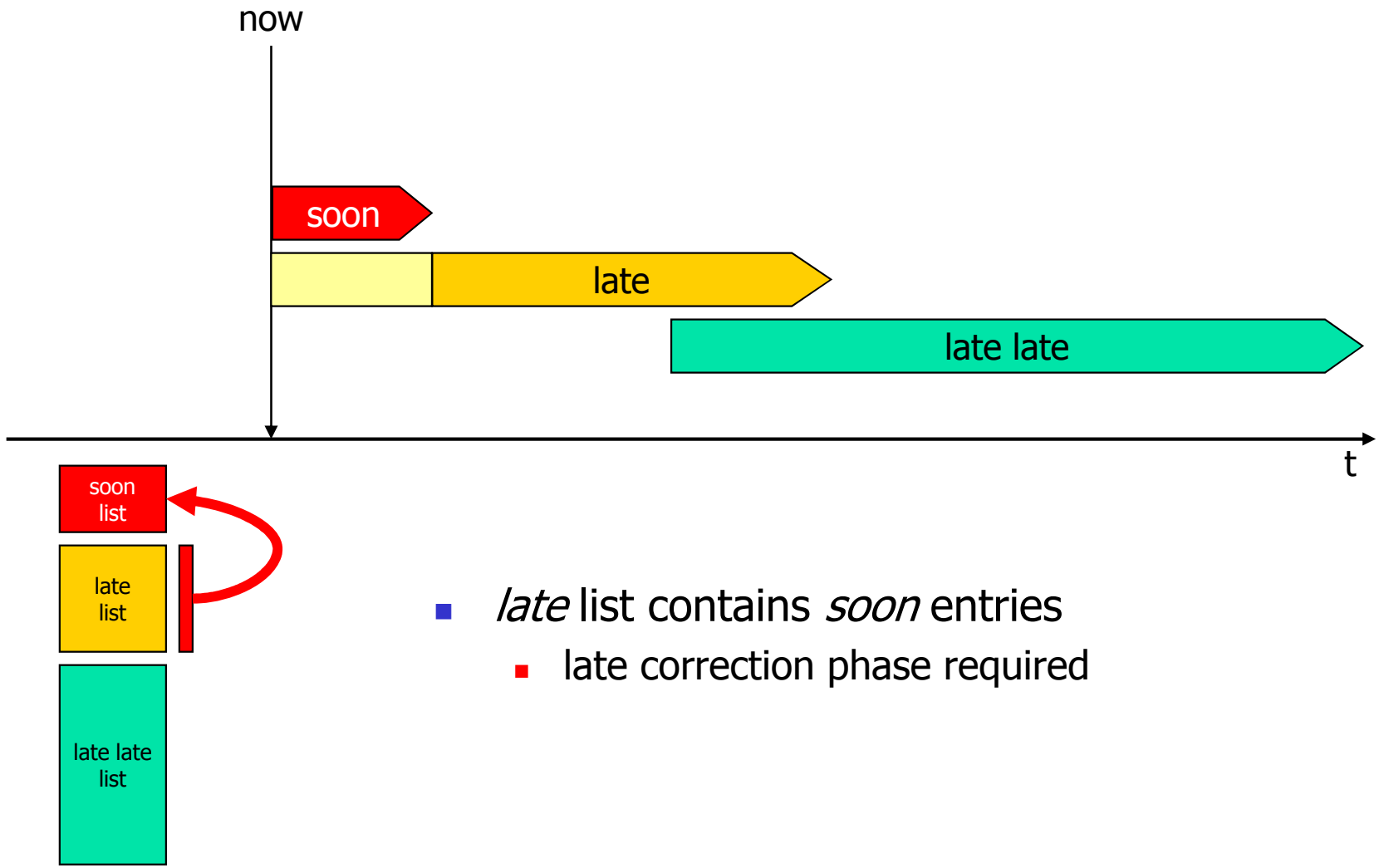
Wakeup Classes



Wakeup Classes

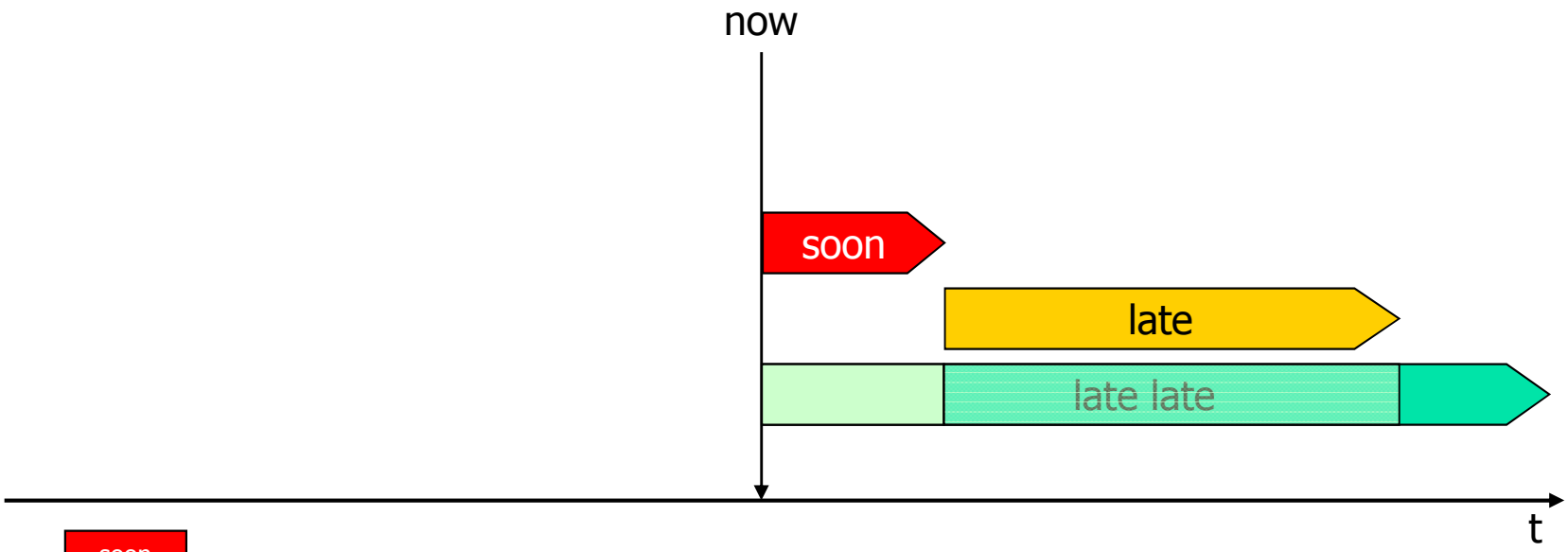


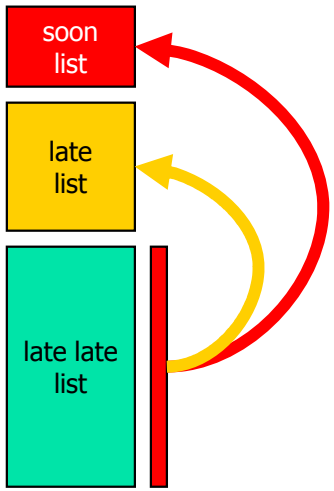
Wakeup Classes



- *late* list contains *soon* entries
  - late correction phase required

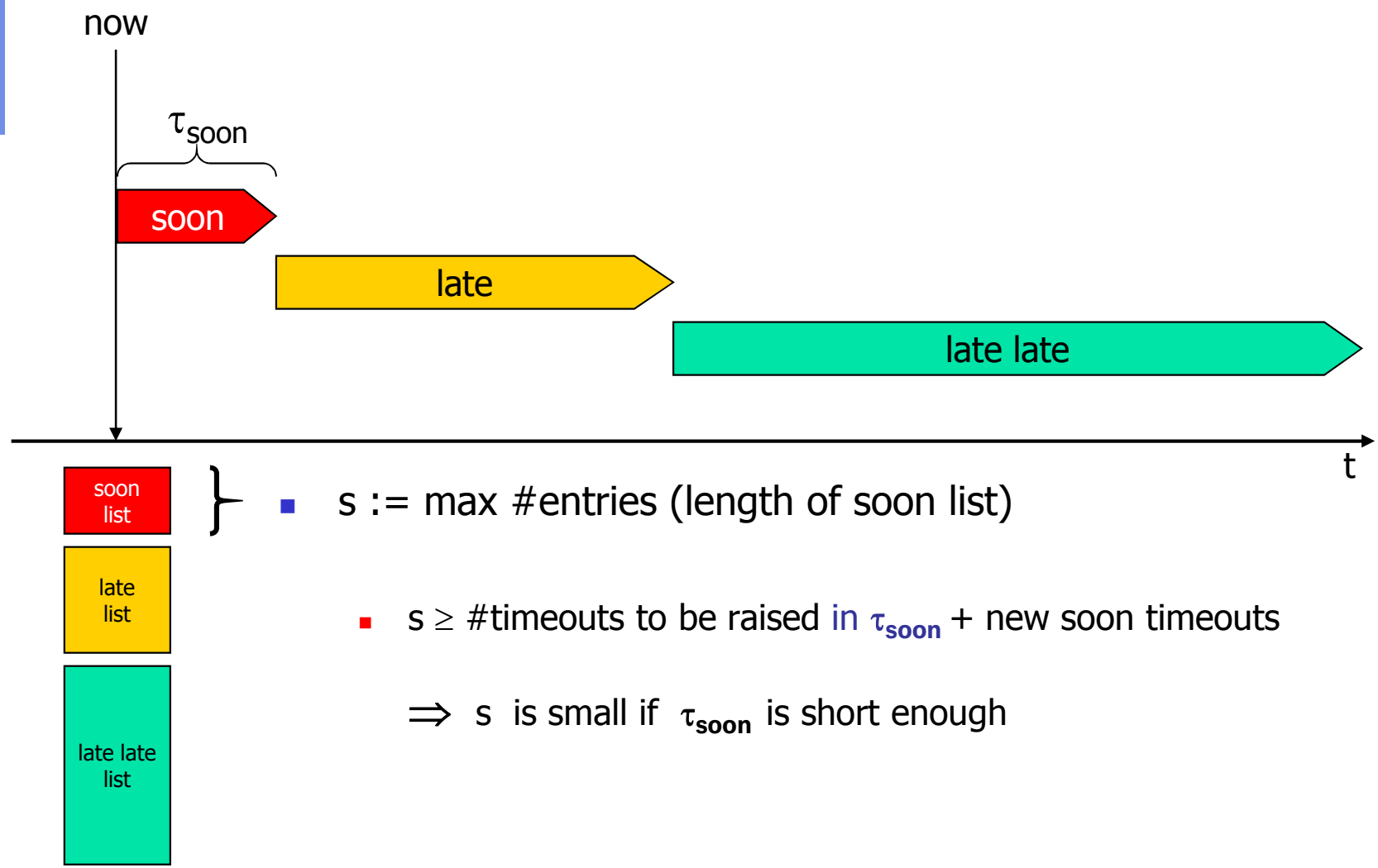
Wakeup Classes





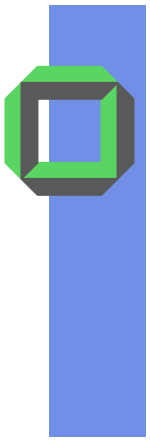
- *late late* list contains *soon* and *late* entries
  - late late correction phase required
  - Also move *late late* entries to *late* list
    - Postpone further late late correction phases

Wakeup Classes



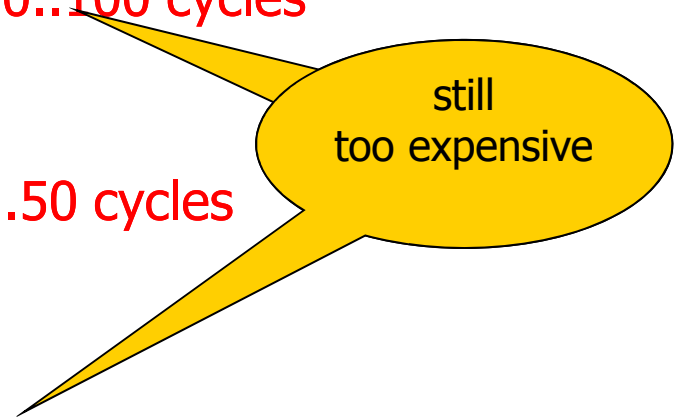
- s := max #entries (length of soon list)
  - s ≥ #timeouts to be raised in τ_{soon} + new soon timeouts

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



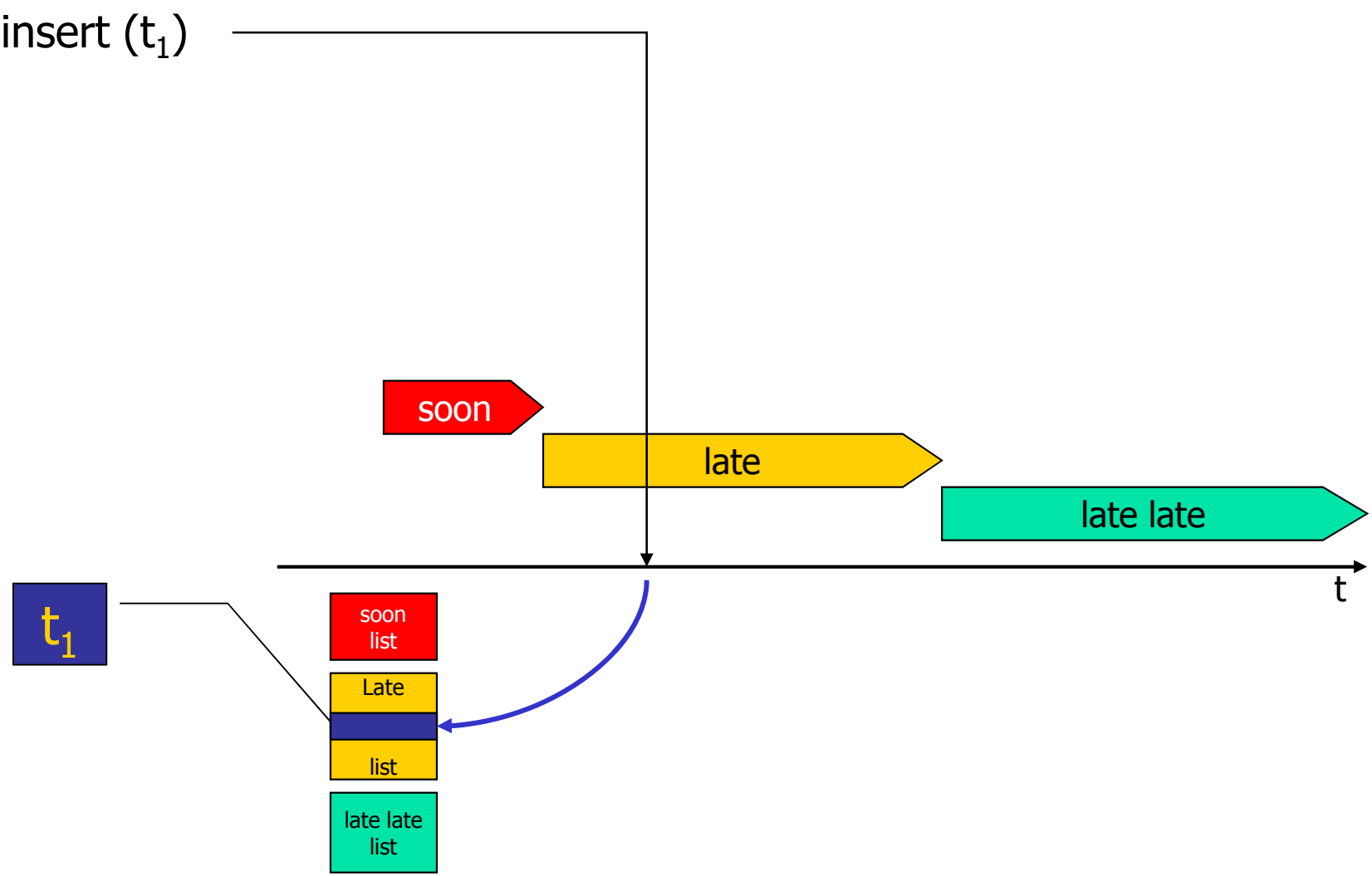
Timeouts & Wakeups

- 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!

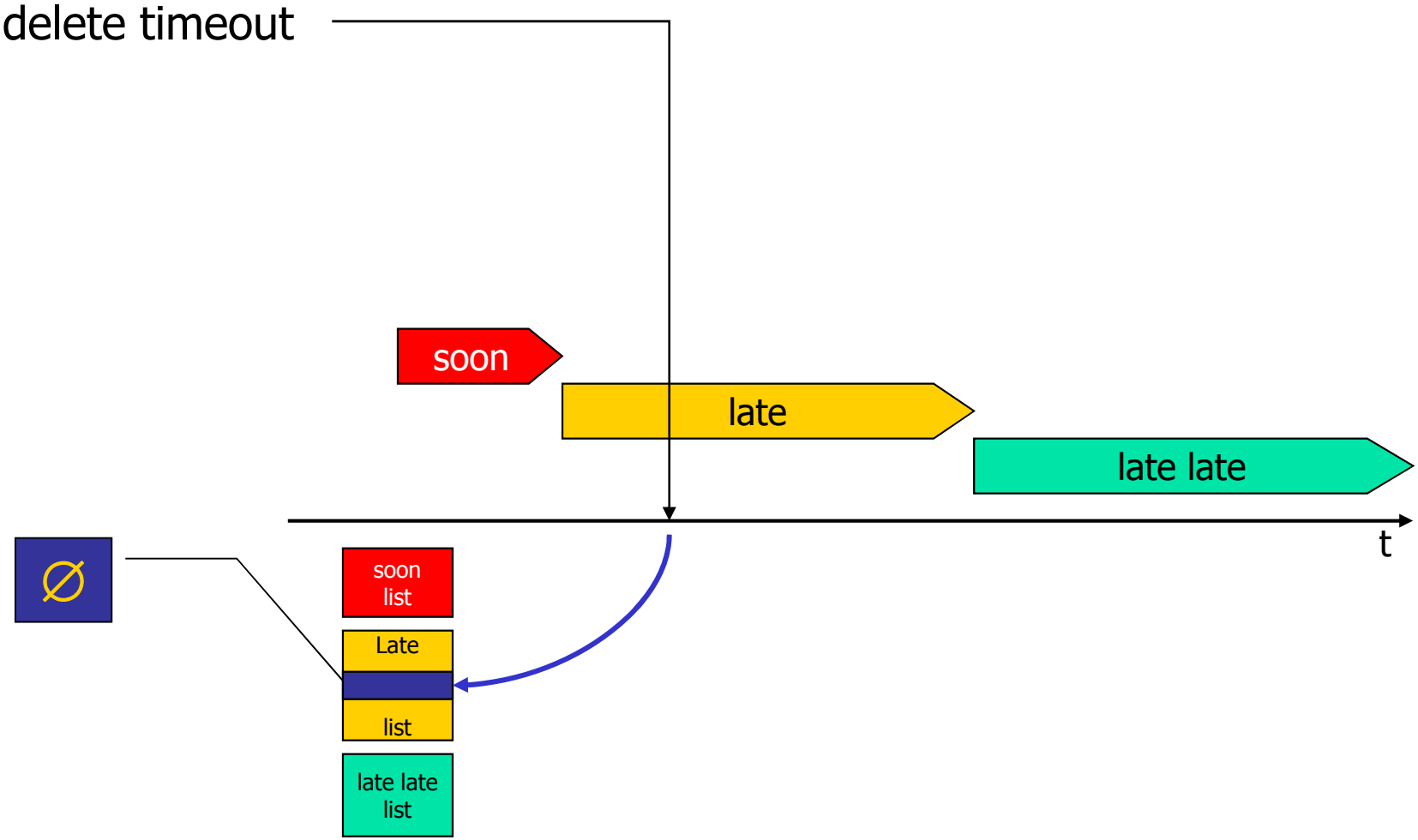
Lazy Timeouts



Lazy Timeouts

insert (t_1)

delete timeout



Lazy Timeouts

insert (t_1)
delete timeout
insert (t_2)