Scheduling
(continued)

New (Stride Scheduling) Algorithm

- Each thread has a *(possibly crooked)* meter that runs only when the thread is running on the processor
- Each thread's meter is initialized as 1/bribe
- At every clock tick
  - give processor to thread that's had the least processor time as shown on its meter
  - in case of tie, thread with lowest ID wins

Example

- Three threads
  - T_1 has one ticket: meter_rate = 1
  - T_2 has two tickets: meter_rate = 1/2
  - T_3 has three tickets: meter_rate = 1/3
  - six total tickets
- After 6 clock ticks
  - T_1's meter incremented by 1 once
  - T_2's meter incremented by 1/2 twice
  - T_3's meter incremented by 1/3 three times
- Each meter shows increase of 1
  - what "1" means depends on the bribe

More Details

```
typedef struct {
   ...
   float bribe, meter_rate, metered_time;
} thread_t;

void thread_init(thread_t *t, float bribe) {
   if (bribe < 1)
      abort();
   t->bribe = bribe;
   t->meter_rate = t->metered_time = 1/bribe;
   InsertQueue(t);
}
```

More Details (revised)

```
typedef struct {
   ...
   long bribe, meter_rate, metered_time;
} thread_t;

const long BigInt = 2^50;

void thread_init(thread_t *t, long bribe) {
   if (bribe < 1)
      abort();
   t->bribe = bribe;
   t->meter_rate = t->metered_time
         = BigInt/bribe;
}
```

More Details (continued)

```
void OnClockTick() {
  thread_t *NextThread;

  CurrentThread->metered_time +=
      CurrentThread->meter_rate;
  InsertQueue(CurrentThread);
  NextThread =
      PullSmallestThreadFromQueue();
  if (NextThread != CurrentThread)
    SwitchTo(NextThread);
}
```

Handling New Threads

- It's time to get an accountant …
  - keep track of total bribes
    - TotalBribe = total number of tickets in use
  - keep track of total (normalized) processor time: *TotalTime*
    - measured by a "fixed" meter going at the rate of 1/TotalBribe
- New thread
  1) pays bribe, gets meter
  2) `metered_time` initialized to `TotalTime+meter_rate`

Example (continued)

- Three threads
  - T_1 has one ticket: meter_rate = 1
  - T_2 has two tickets: meter_rate = 1/2
  - T_3 has three tickets: meter_rate = 1/3
  - TotalBribe = 6
- Assume one clock interrupt/second
  - at every interrupt: TotalTime += 1/6
- After 6 seconds
  - T_1's meter incremented by 1 once
  - T_2's meter incremented by 1/2 twice
  - T_3's meter incremented by 1/3 three times
  - TotalTime incremented by 1/6 six times

What's Going On ...

- Assume T clock interrupts/second
  - every TotalBribe seconds
    - TotalTime incremented by T
    - each thread's metered_time incremented by T
- TotalTime · TotalBribe
  = actual total processor time
- metered_time · bribe
  = actual time used by thread
- Initialize meter so it appears as if new thread has been getting its share of processor time since beginning of time

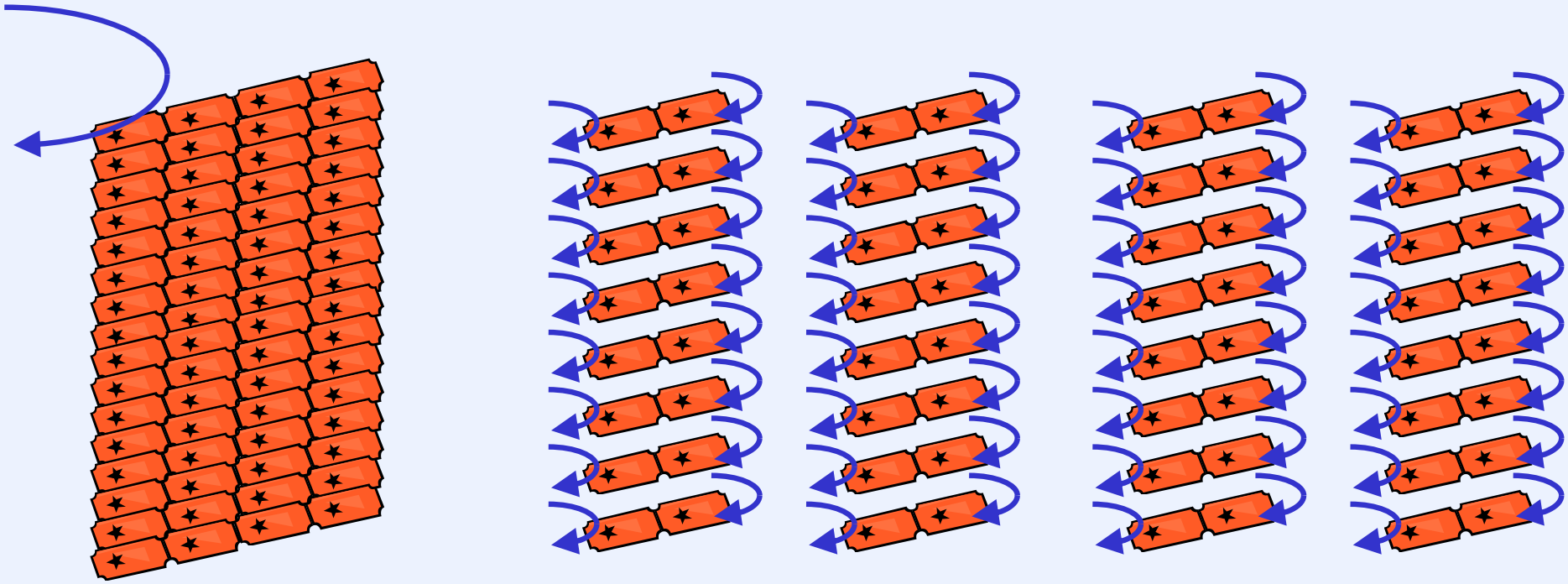
Revised Details

```
void OnClockTick() {
  thread_t *NextThread;

  TotalTime += 1/TotalBribe;
  CurrentThread->metered_time +=
      CurrentThread->meter_rate;
  InsertQueue(CurrentThread);
  NextThread =
      PullSmallestThreadFromQueue();
  if (NextThread != CurrentThread)
    SwitchTo(NextThread);
}
```

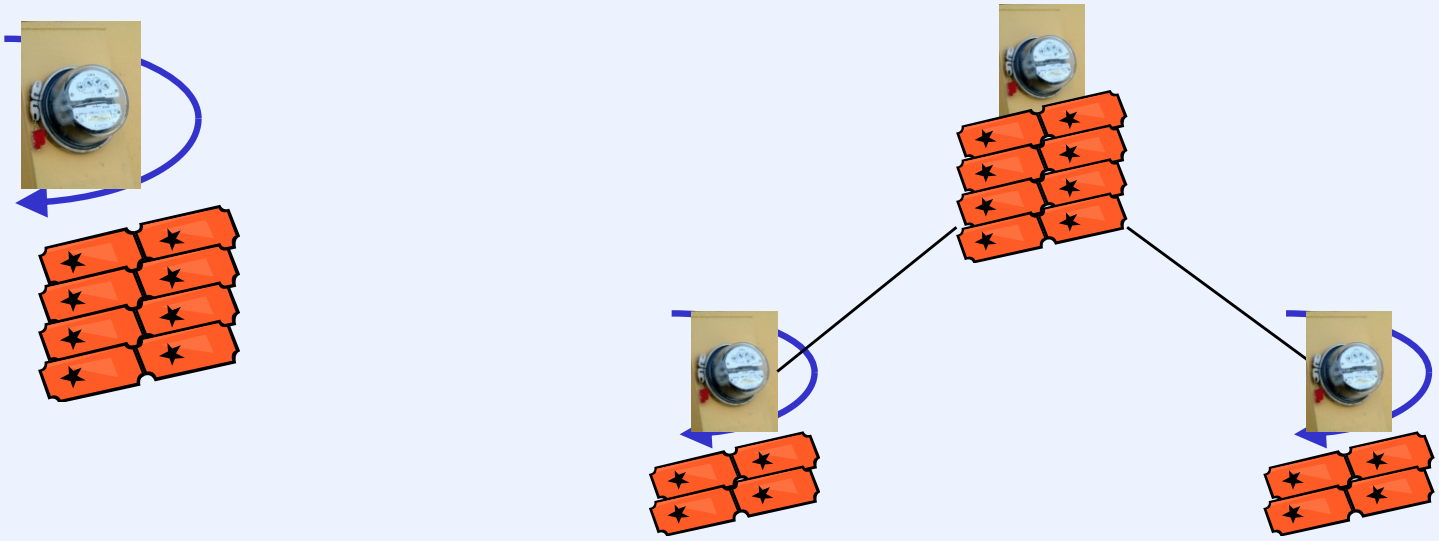
Thread Leaves, then Returns

```
void ThreadDepart(thread_t *t) {
  t->remaining_time =
      t->metered_time - TotalTime;
  // remaining_time is a new component
  TotalBribe -= t->bribe;
}

void ThreadReturn(thread_t *t) {
  t->metered_time =
      TotalTime + t->remaining_time;
  TotalBribe += t->bribe;
}
```

A Mismatch



Hierarchical Stride Scheduling



Quiz 1

Recall our discussion about handling interactive jobs in early Unix. The later BSD scheduler would give priority to threads that were mainly idle, thus favoring interactive jobs.

Does stride scheduling favor such interactive jobs? I.e., if a waiting thread suddenly receives input, will it get to run sooner than other runnable threads of the same priority?

a) yes

b) no

Latency

- Some threads may require low latency
  - should run very soon after becoming runnable
  - could give them higher priorities
    - not good if the thread runs for a long time
  - a possibility: give such threads lower latency in exchange for running for shorter periods of time

EEVDF (1)

- "Earliest Eligible Virtual Deadline First"
- Assume all threads have same priority
- If there are *n* runnable threads, each gets a (1/n)-second execution slot each second

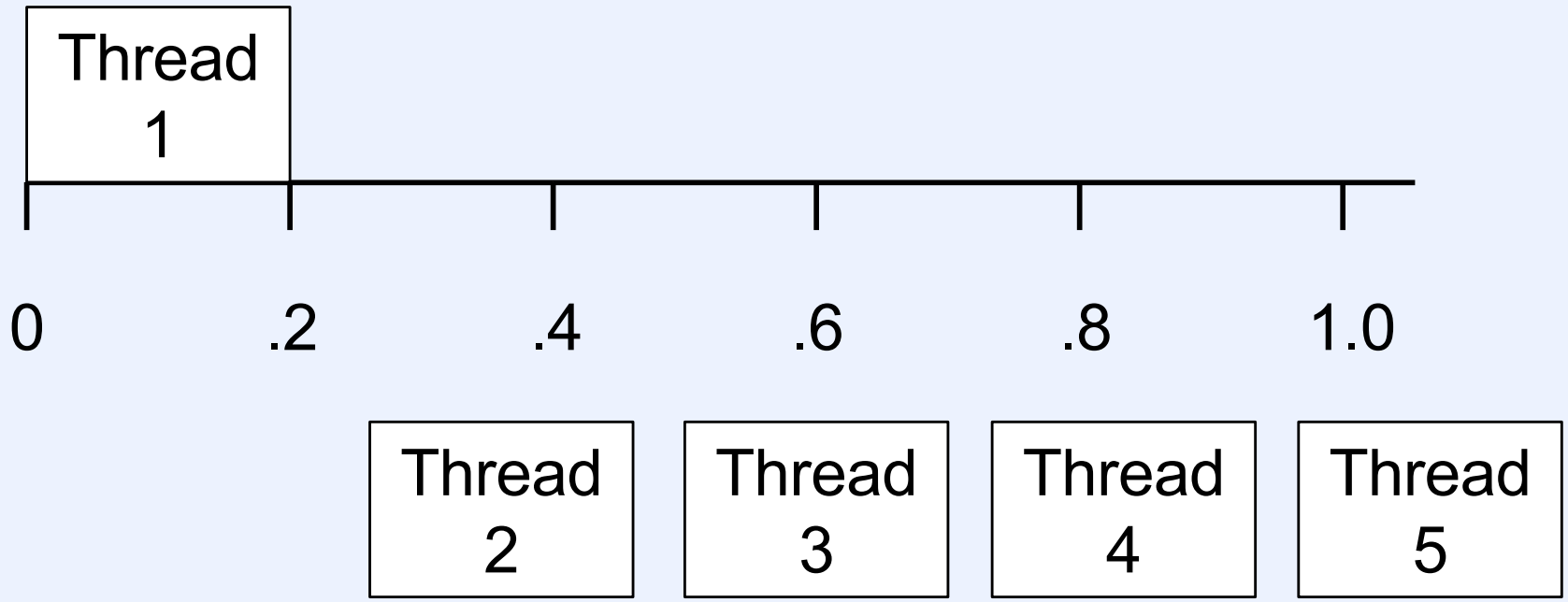
EEVDF (2)

- Implemented by keeping track of how much time a thread has actually run during the period
  - *lag* is how much much more time till it reaches its quota
  - *eligible* for running if lag is positive
  - *virtual deadline* is earliest time by which it can achieve its quota
    - assuming it starts now and runs for 1/n second
  - next thread to run is the one with the *earliest virtual deadline*

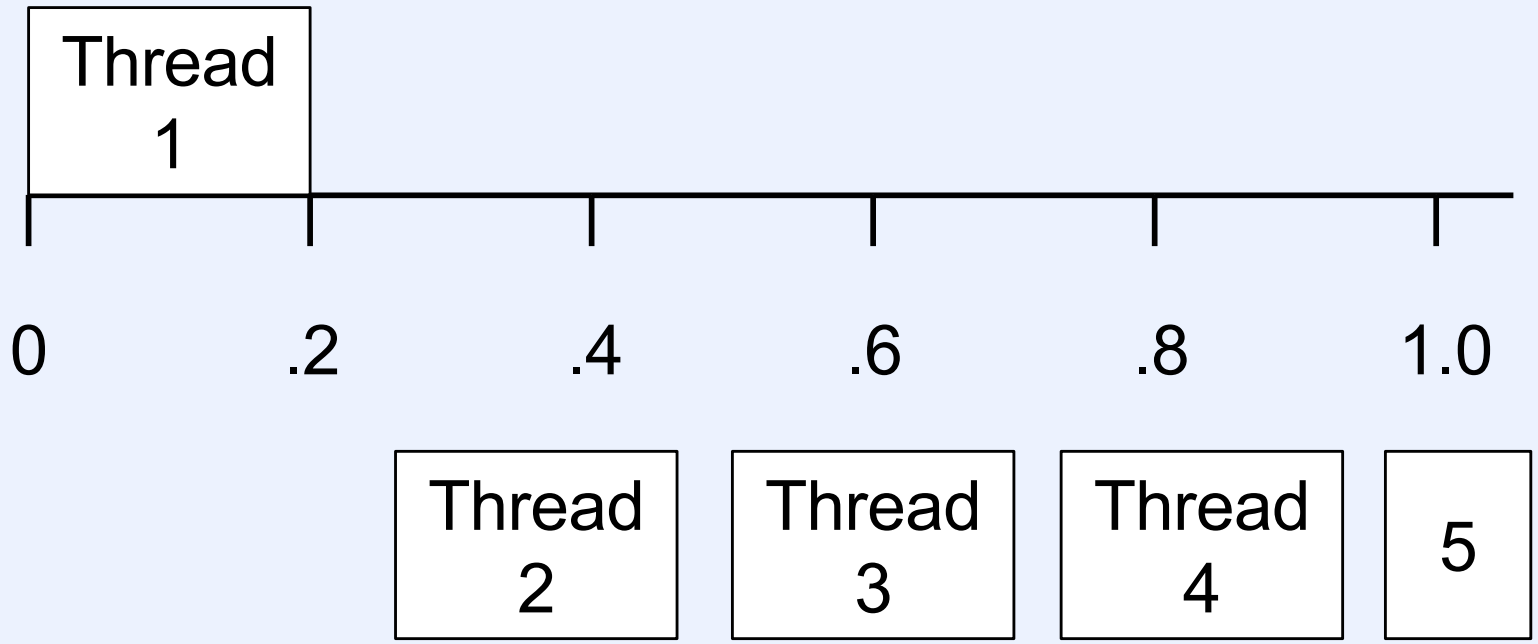
EEVDF (3)

- If a thread requires low latency, it will be given a shorter time slice (say 1/(2n))
  - its virtual deadline thus comes sooner than those for other threads
  - it runs for correspondingly shorter periods of time
  - for any scheduling interval it gets the same total amount of time as other threads
    - but in smaller chunks

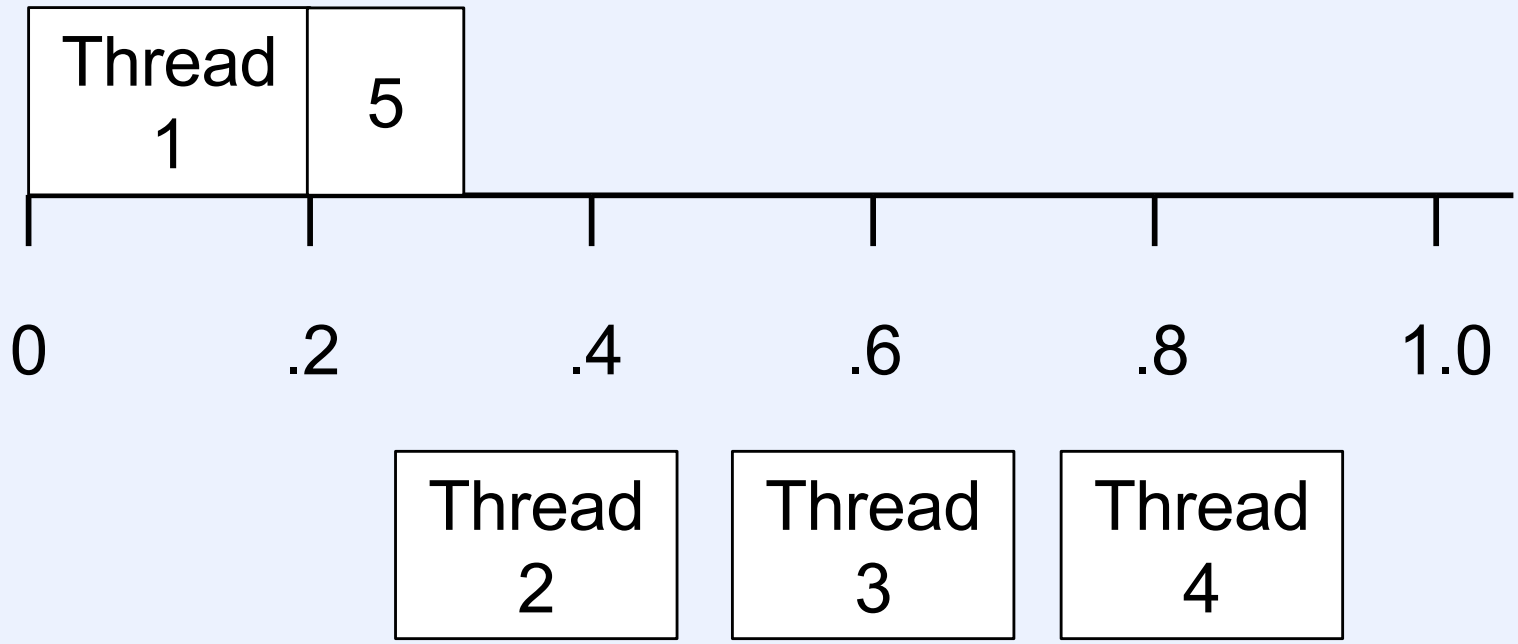
Example (1)



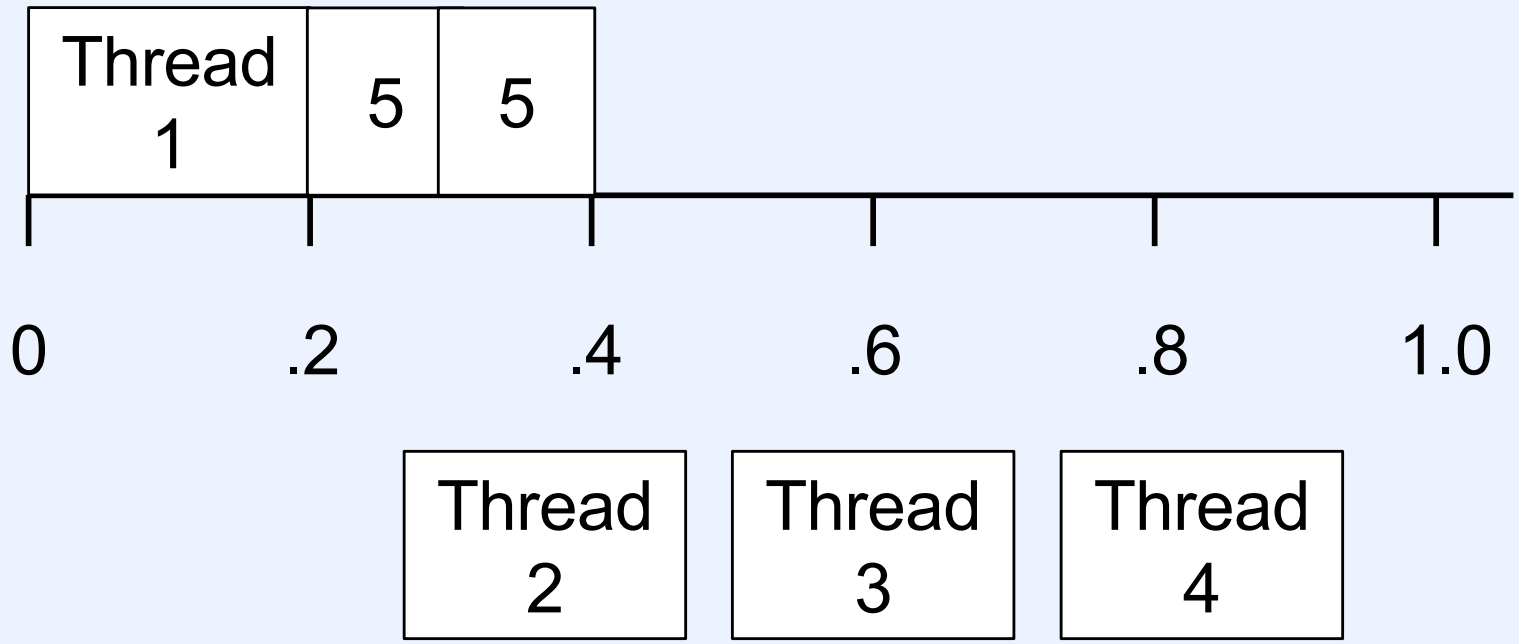
Example (2)



Example (3)



Example (4)



Determining the Time Slice

- Linux defines a new scheduling parameter
  - latency-nice
    - settable by a system call
  - the lower the value, the less latency a thread has, and thus the shorter its time slices

Real-Time Scheduling

- Jobs j_i arrive at times t_i with deadlines d_i
  - find schedule satisfying constraints
  - does schedule exist?
  - if so, what is it?
  - in general, j_i, t_i, and d_i are not known ahead of time

Uniprocessor

- Earliest-deadline first
  - optimal
- Rate-monotonic scheduling of cyclic chores
  - easy

Multiprocessor

- Earliest-deadline first and rate-monotonic still work (non-optimally)
- Finding optimal schedule is NP-complete
  - equivalent to bin-packing problem

Assumptions

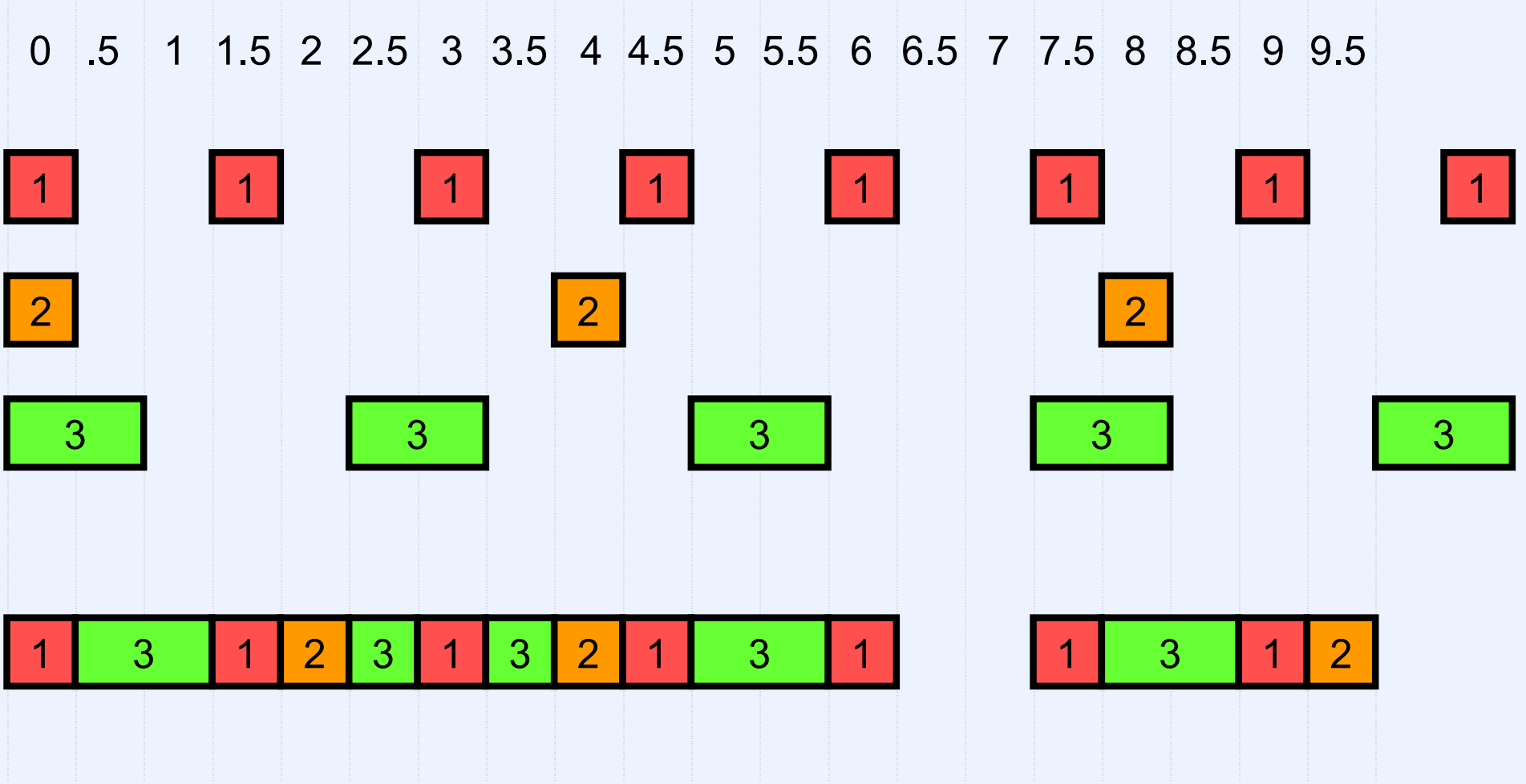
- Interrupts don't interfere (too much) with schedule
  - bounded interrupt delays
- Execution time really is predictable
  - what about effects of caching and paging?

Rate-Monotonic Scheduling

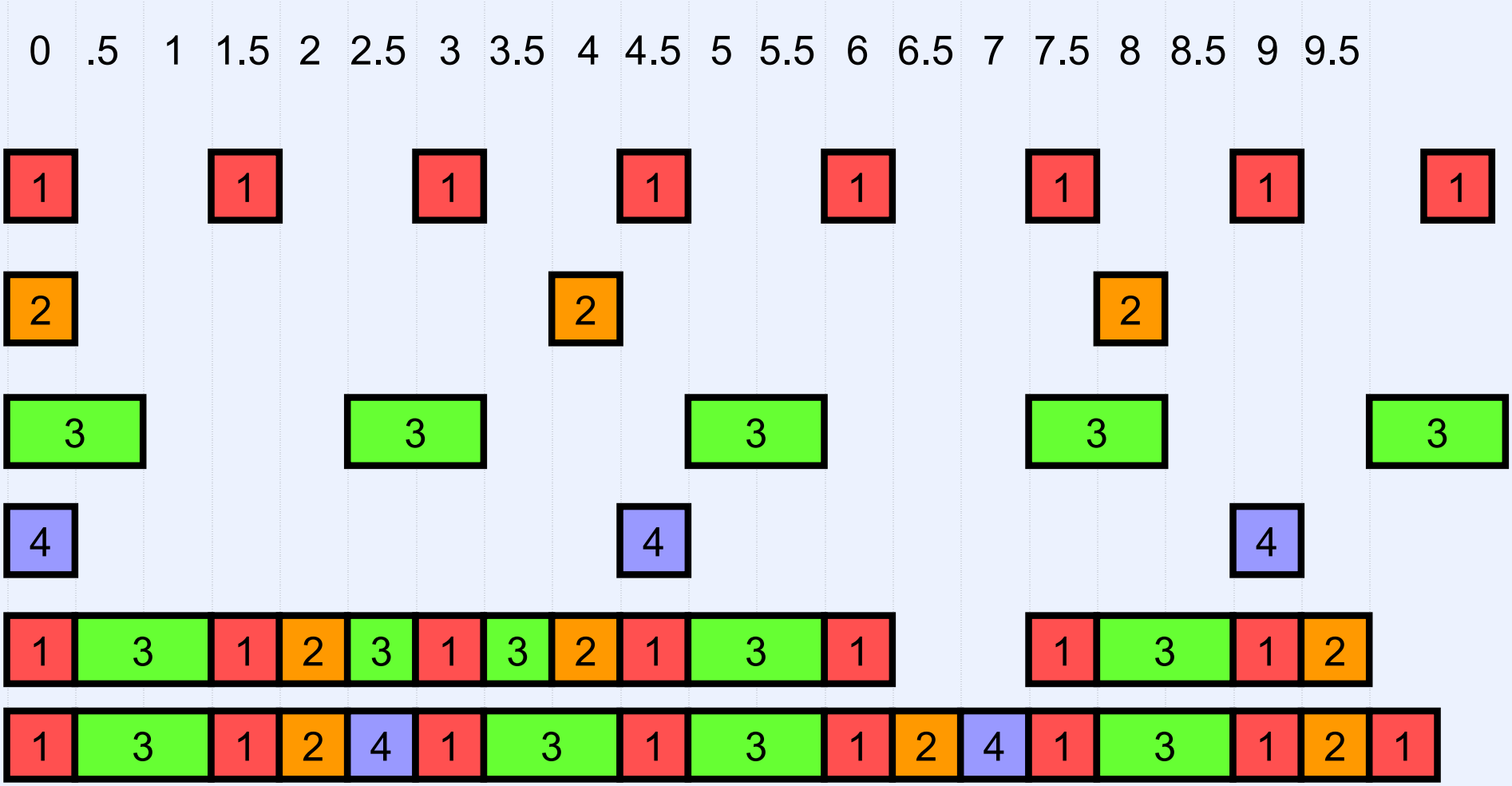
- Periodic chores
  - period P_i
  - per-cycle processing time T_i (<= P_i)
  - feasible if $sum(T_i/P_i) <= 1$
- Rate-monotonic scheduling
  - each chore *i* is handled by a thread with priority $1/P_i$
  - preemptive, priority scheduling
  - works when $sum(T_i/P_i) <= n(2^{1/n}-1)$

    $= ln\ 2$ in the limit

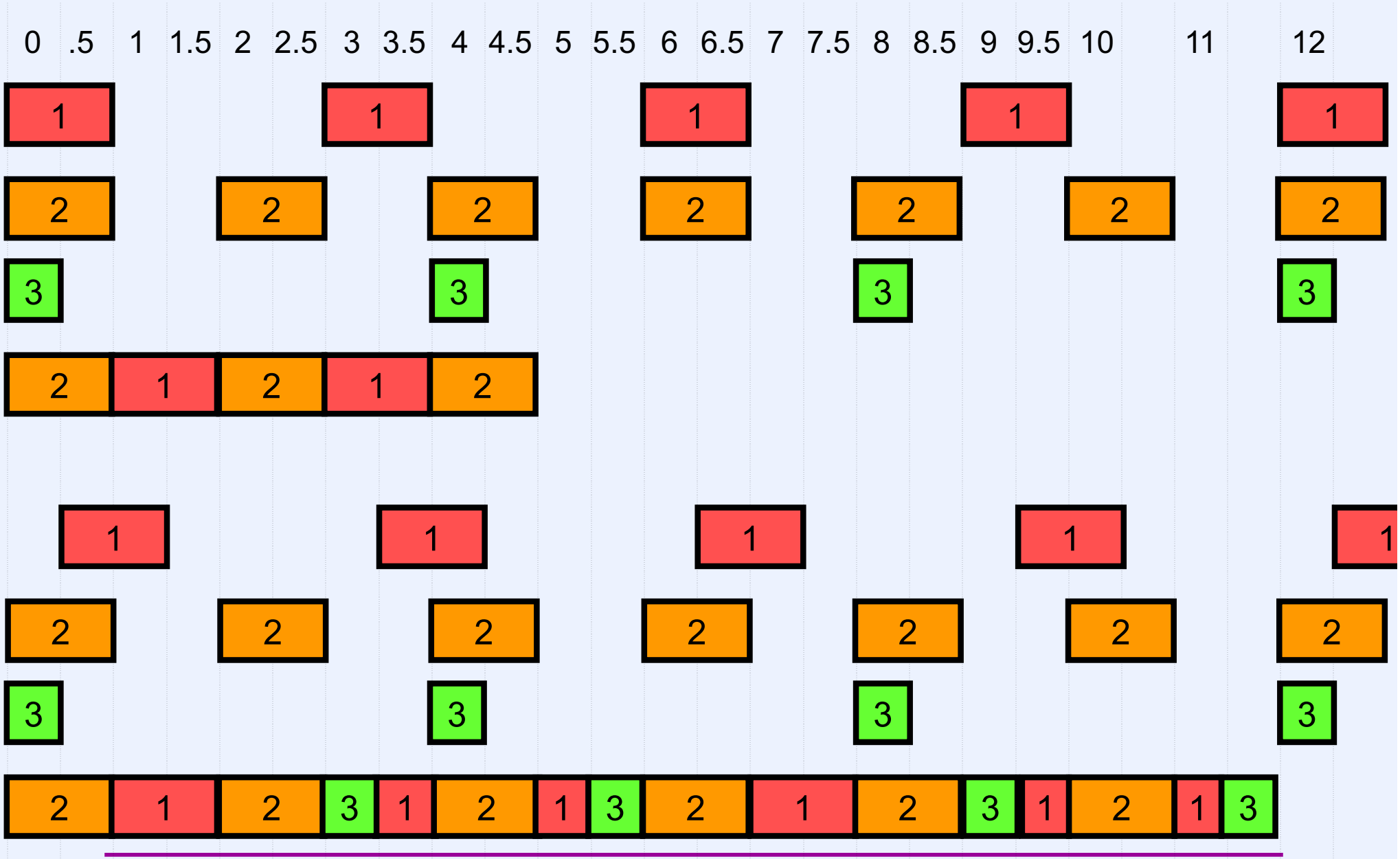
Scenario 1



Scenario 2



Phase Problems

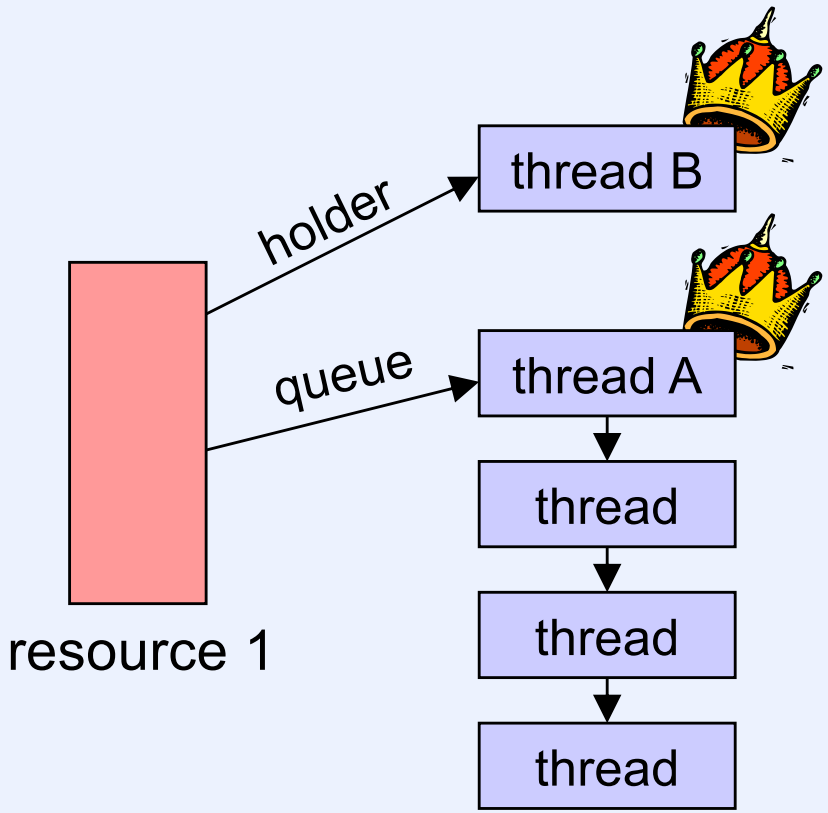


Priority Problem

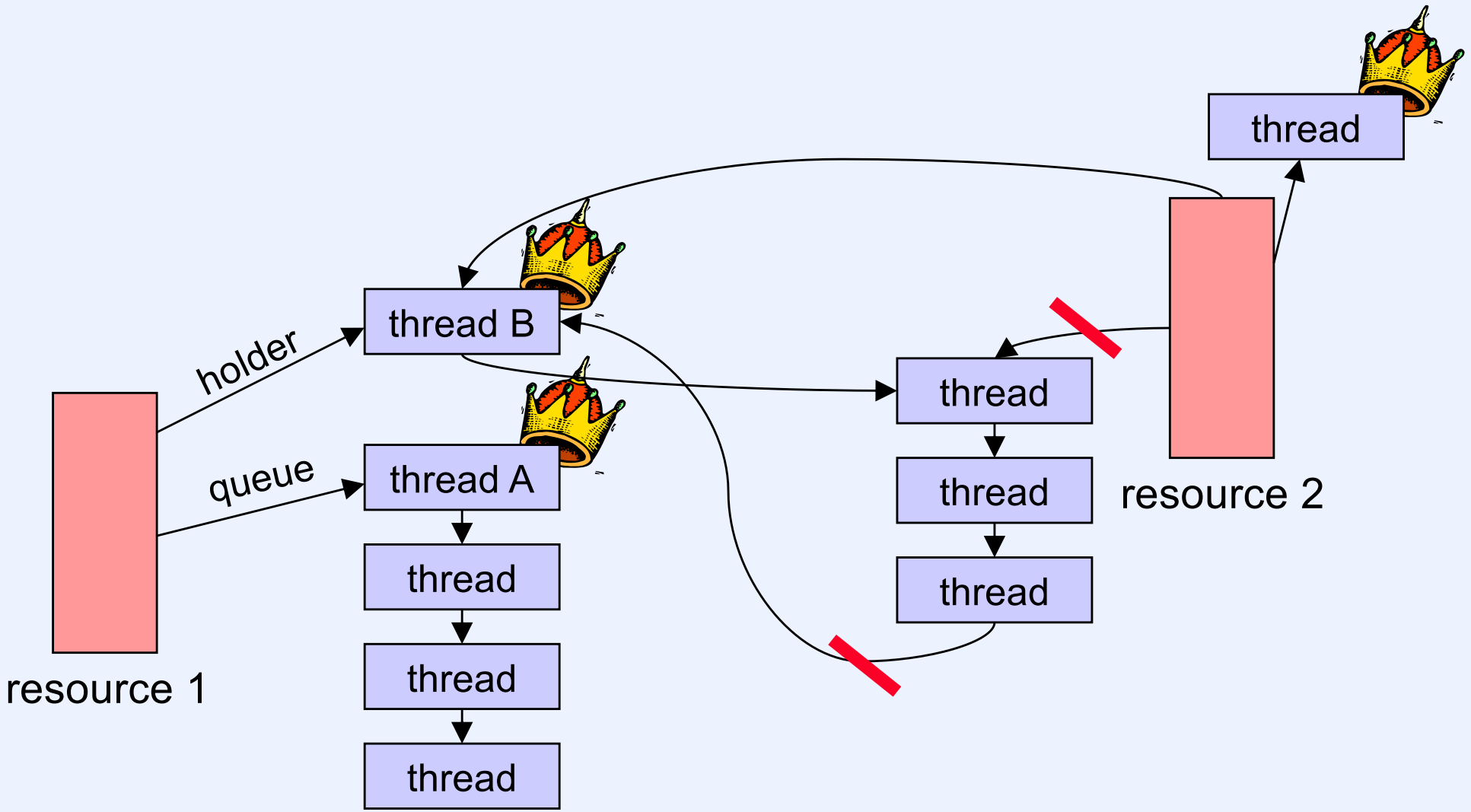
- High-priority thread A blocks on mutex 1
- Low-priority thread B holds mutex 1
- Thread B can't run because medium-priority thread C is running
- A is effectively waiting at B's priority
  - *priority inversion*

Priority Inheritance

- **While A is waiting for resource held by B, it gives B its priority**



Cacading Inheritance



Linux Scheduling

- Policies
  - SCHED_FIFO
    - "real time"
    - infinite time quantum
  - SCHED_RR
    - "real time"
    - adjustable time quantum
  - SCHED_DEADLINE
    - earliest-deadline first
  - SCHED_OTHER
    - "normal" scheduler
    - parameterized allocation of processor time

Linux Scheduler Evolution

- Old scheduler
  - very simple
  - poor scaling
- O(1) scheduler
  - less simple
  - better scaling
- Completely fair scheduler (CFS)
  - even better
  - simpler in concept
  - much less so in implementation
  - based on stride scheduling

Old Scheduler

- Four per-process scheduling variables
  - *policy*: which one
  - *rt_priority*: real-time priority
    - 0 for SCHED_OTHER
    - 1 – 99 for others
  - *priority*: time-slice parameter
  - *counter*: records processor consumption

Old Scheduler: Time Slicing

- Clock “ticks” HZ times per second
  - interrupt/tick
- Per-process *counter*
  - current process’s is decremented by one each tick
  - time slice over when counter reaches 0

Old Scheduler: Throughput

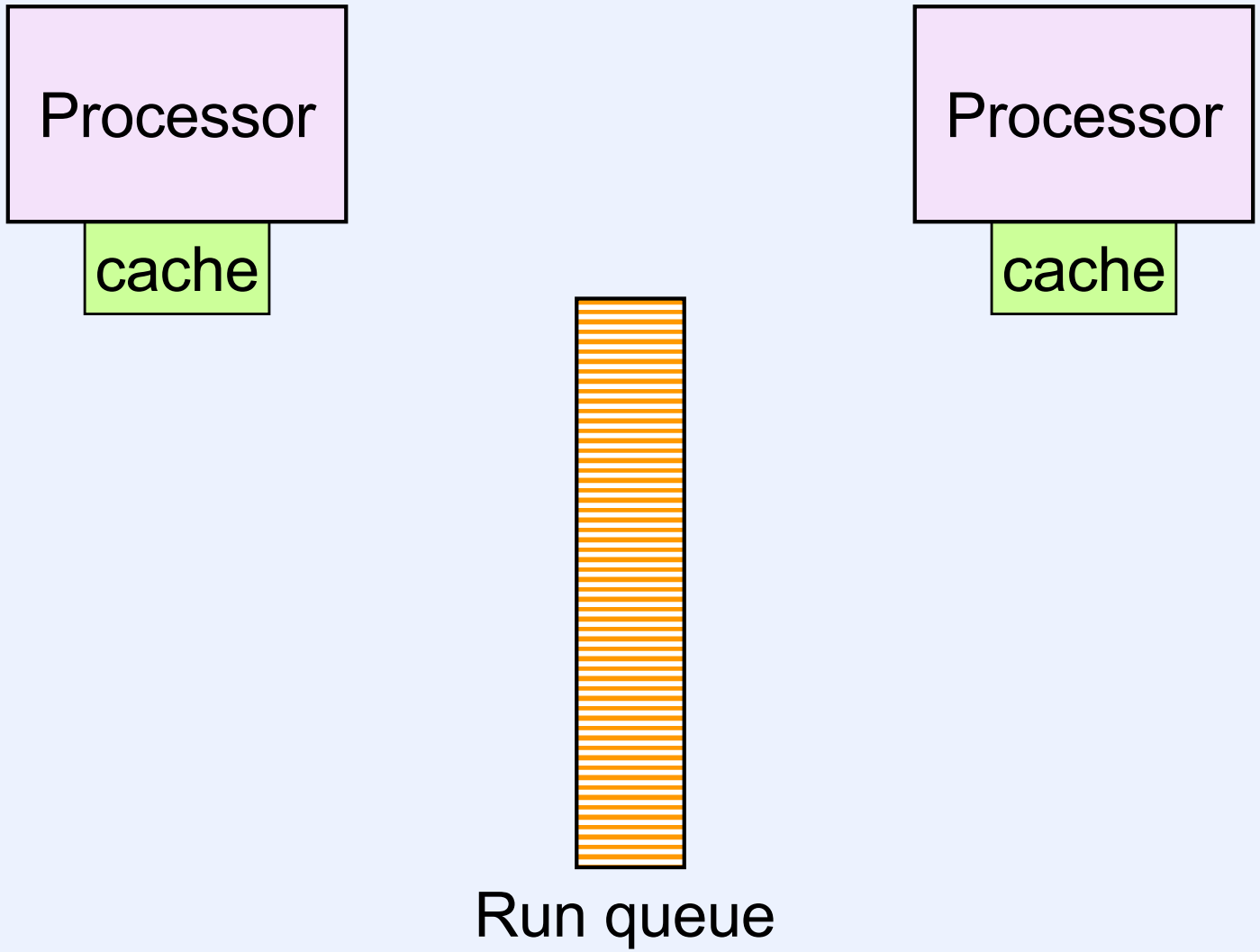
- Scheduling cycle
  - length, in "ticks," is sum of priorities
  - each process gets *priority* ticks/cycle
    - *counter* set to *priority*
    - cycle over when *counter*s for runnable processes are all 0
  - sleeping processes get "boost" at wakeup
    - at beginning of each cycle, for each process (runnable or not):

```
counter = counter/2 + priority
```

Old Scheduler: Who's Next?

- Run queue searched beginning to end
  - new arrivals go to beginning
  - SCHED_RR processes go to end at completion of time slices
- Next running process is first process with highest "goodness"
  - 1000 + *rt_priority* for SCHED_FIFO and SCHED_RR processes
  - *counter* for SCHED_OTHER processes

Diagram



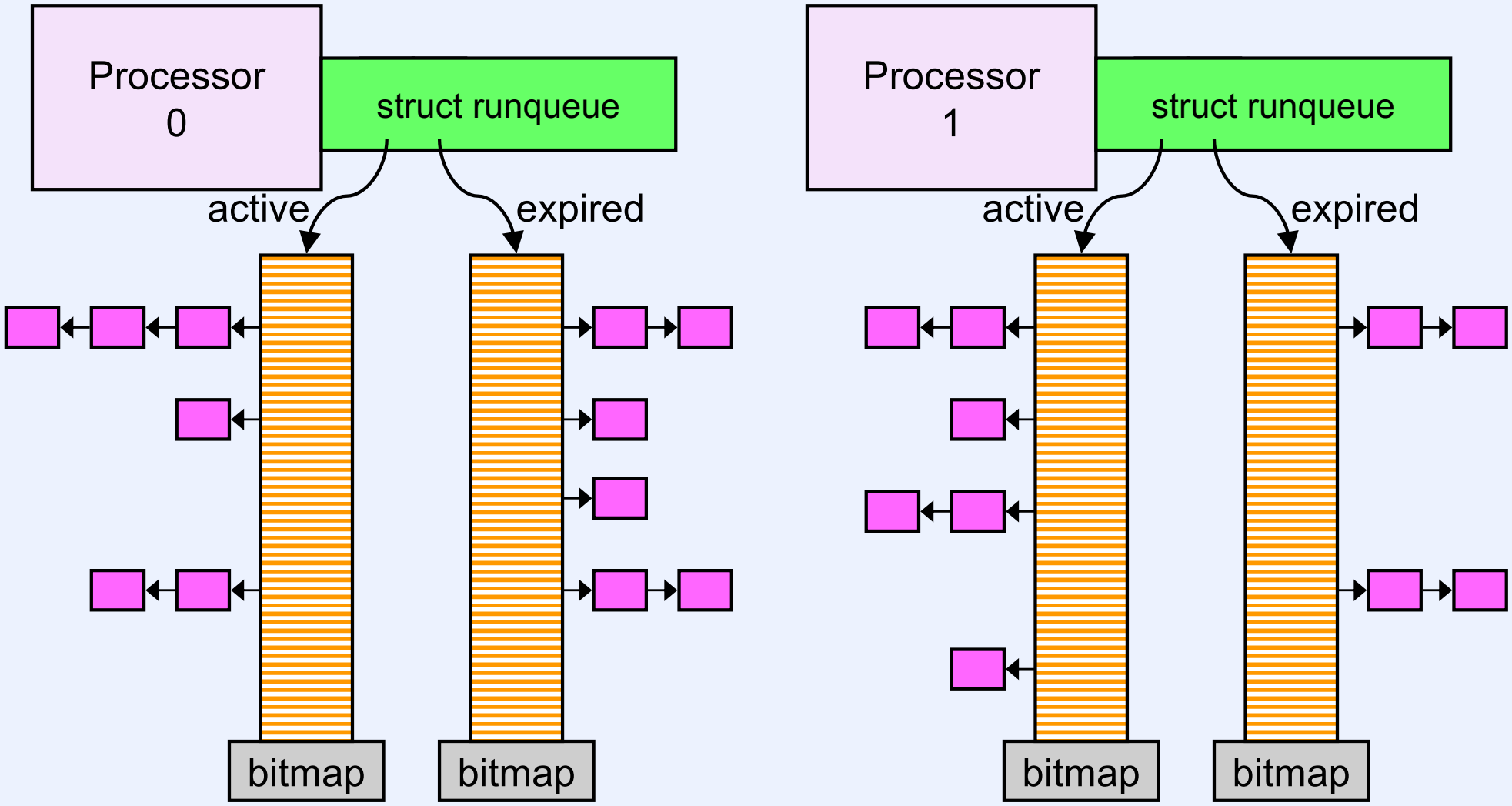
Old Scheduler: Problems

- O(n) execution
- Poor interactive performance with heavy loads
- SMP contention for run-queue lock
- SMP affinity
  - cache “footprint”

O(1) Scheduler

- All concerns of old scheduler plus:
  - efficient, scalable execution
  - identify and favor interactive processes
  - good SMP performance
    - minimal lock overhead
    - processor affinity

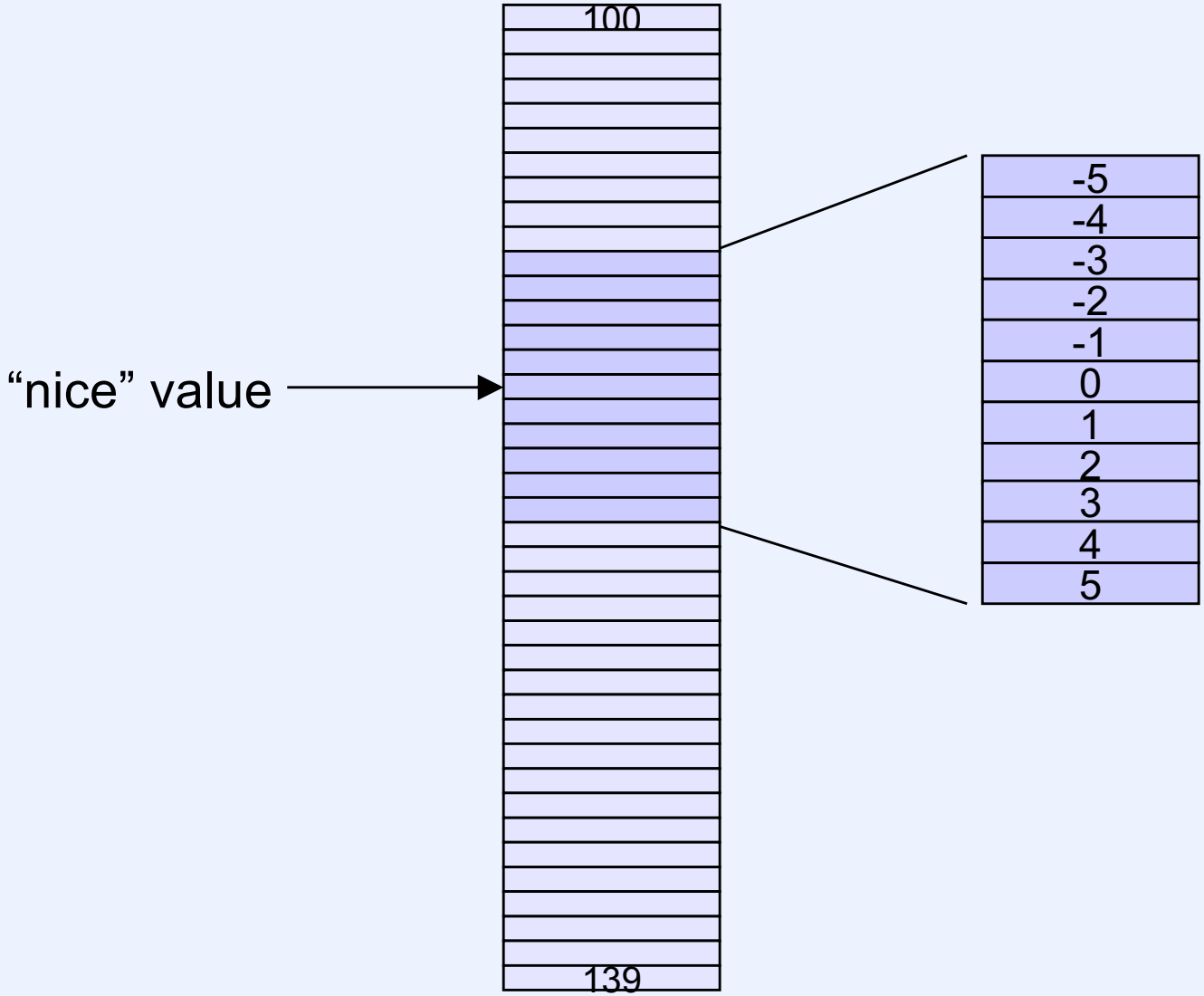
O(1) Scheduler: Data Structures



O(1) Scheduler: Queues

- Two queues per processor
  - active: processes with remaining time slice
  - expired: processes whose time slices expired
  - each queue is an array of lists of processes of the same priority
    - bitmap indicates which priorities have processes
  - processors scheduled from private queues
    - infrequent lock contention
    - good affinity

O(1) Scheduler: Priorities



O(1) Scheduler: Actions

- Process switch
  - pick best priority from active queue
    - if empty, switch active and expired
  - new process's time slice is function of its priority
- Wake up
  - priority is boosted or dropped depending on sleep time
  - interactive processes are defined as those whose priority is above a certain threshold
- Time-slice expiration
  - normal processes join expired queue
  - real-time join active queue

O(1) Scheduler: Load Balancing

- Processors with empty queues steal from busiest processor
  - checked every millisecond
- Processors with relatively small queues also steal from busiest processor
  - checked every 250 milliseconds