Jin-Soo Kim
(*jinsoo.kim@snu.ac.kr*)

Systems Software &
Architecture Lab.

Seoul National University

Spring 2024

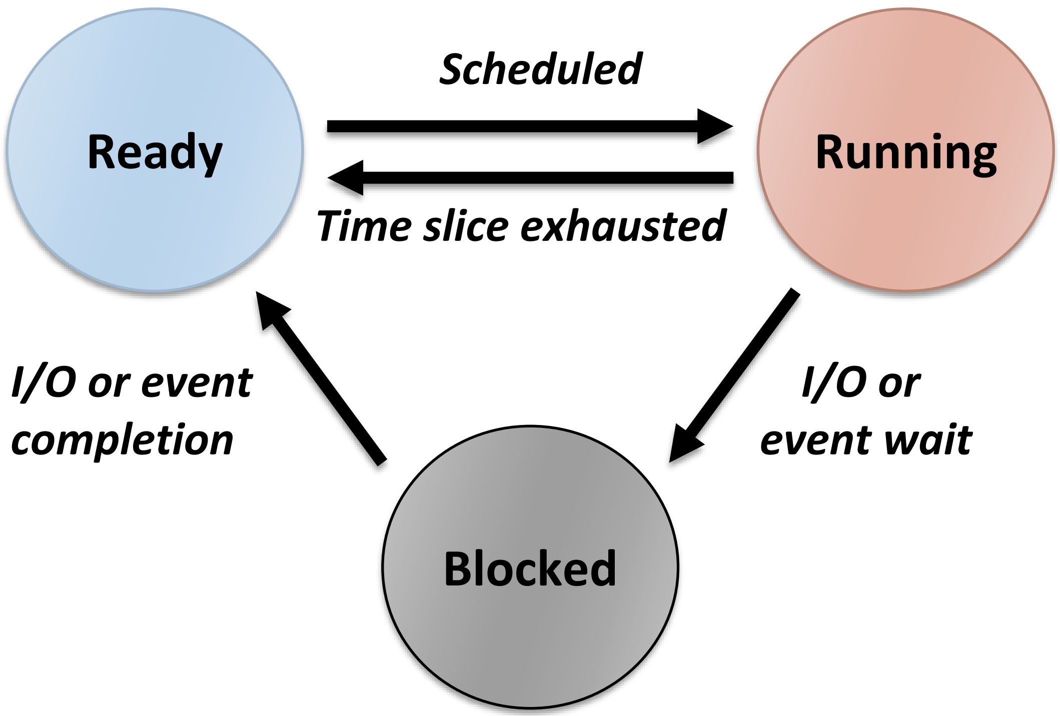
CPU Scheduling

CPU Scheduling

- A policy deciding which process to run next, given a set of runnable tasks (processes or threads)
  - Happens frequently, hence should be fast

- Mechanism
  - ______________________

- Policy
  - ______________________
  - ______________________



Preemptive (or not)

- **Non-preemptive scheduler**
  - The scheduler waits for the running task to voluntarily yield the CPU
    - cf.) `yield()`
  - Tasks should be ___________

- **Preemptive scheduler**
  - The scheduler can interrupt a task and force a context switch
  - Implemented using periodic timer interrupts

  - What if a task is preempted in the midst of updating the shared data?
  - What if a task in a system call is preempted?

Work-Conserving (or not)

- Work-conserving scheduler
  - Never leave a resource idle when someone wants it
  - e.g., Linux CPU scheduler (ideally)

- Non-work-conserving scheduler
  - May leave the resource idle despite the presence of jobs
  - e.g., Server waits for short job before starting on a big job
  - e.g., Anticipatory I/O scheduler: waits for a short time after a read operation in anticipation of another close-by read requests to overcome "deceptive idleness"

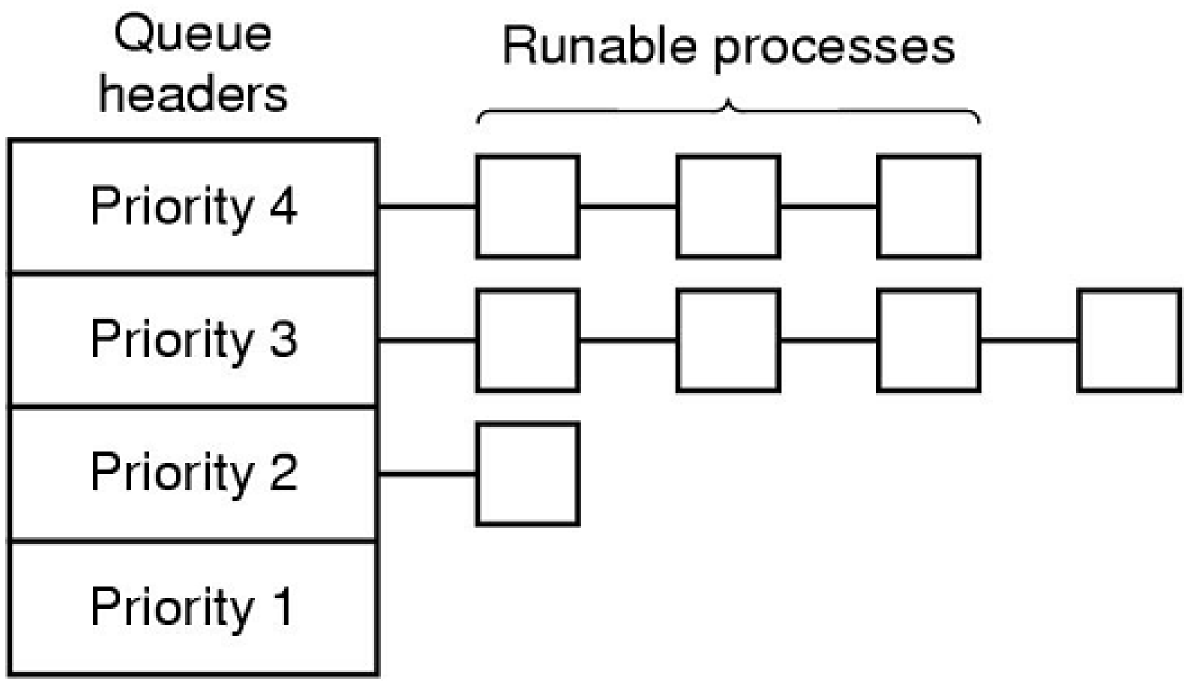
(Static) Priority Scheduling

- Each task has a (static) priority
  - cf.) `nice()`, `renice()`, `setpriority()`, `getpriority()`
- Choose the task with the highest priority to run next
- Round-robin or FIFO within the same priority
- Can be either preemptive or non-preemptive

- Starvation problem
  - If there is an endless supply of high priority tasks, no low priority task will ever run

Priority Scheduling

- Priority is dynamically adjusted at run time
- Modeled as a Multi-level Feedback Queue (MLFQ)
  - A number of distinct queues for each priority level
  - Priority scheduling between queues, round-robin in the same queue



UNIX Scheduler

- MLFQ
  - Preemptive priority scheduling
  - Time-shared based on time slice
  - Tasks dynamically change priority
- Aging for avoiding starvation
  - Increase priority as a function of wait time
  - Decrease priority as a function of CPU time
- Favor interactive tasks over CPU-bound tasks
- Priority vs. time slice?
- Many ugly heuristics have been explored in this area

Linux Scheduler Evolution

Kernel version	CPU Scheduler
Linux 2.4	• Epoch-based priority scheduling • O(*n*) scheduler
Linux 2.6 ~ 2.6.22	• Active / expired arrays with bitmaps • Per-core run queue • O(1) scheduler
Linux 2.6.23 ~	• CFS (Completely Fair Scheduler) by Ingo Molnar
Linux 3.14 ~	• Sporadic task model deadline scheduling (SCHED_DEADLINE)

Linux Scheduling Classes

Class	Description	Policy
DL	• For real-time tasks with deadline • Highest priority	SCHED_DEADLINE
RT	• For real-time tasks	SCHED_FIFO SCHED_RR
Fair	• For time-sharing tasks	SCHED_NORMAL SCHED_BATCH
Idle	• For per-CPU idle tasks	SCHED_IDLE

Linux 2.4 Scheduler

Priorities

- Static priority
  - The base priority represented by the nice value in [-20, 19] (default: 0)
  - Determines the task's timeslice
- Dynamic priority
  - The amount of time remaining in this timeslice
  - Declines with time as long as the task has the CPU
  - When its dynamic priority falls to 0, the task is marked for rescheduling
- Real-time priority (used for SCHED_FIFO and SCHED_RR)
  - Only real-time tasks have the real-time priority
  - Higher real-time priority values always beat lower values

Fields Related to Scheduling

counter: time remaining in the task's current quantum (represents dynamic priority)
nice: nice value, -20 to +19 (represents static priority)

policy: SCHED_OTHER, SCHED_FIFO, SCHED_RR

mm: points to the memory descriptor

processor: CPU ID on which the task will execute

cpus_runnable: CPU currently running on
cpus_allowed: CPUs allowed to run
run_list: the run queue

```
struct task_struct {

 ...
/*
 * offset 32 begins here on 32-bit platforms. We keep
 * all fields in a single cacheline that are needed for
 * the goodness() loop in schedule().
 */
  long counter;

  long nice;

  unsigned long policy;

  struct mm_struct *mm;

  int processor;

  unsigned long cpus_runnable, cpus_allowed;

  struct list_head run_list;
  ...
}
```

Timeslice

- Linux v2.4 gets a timer interrupt or a *tick* once every 10ms on IA-32 (HZ = 100)
- Linux wants the time slice to be around 50ms
  - Decreased from 200ms in Linux v2.2
- Timeslice
  - nice = 20 (lowest): 1 tick
  - nice = 0 (default): 6 ticks
  - nice = -19 (highest): 10 ticks

```
/*
 * Scheduling quanta.
 *
 * NOTE! The unix "nice" value influences how long a process
 * gets. The nice value ranges from -20 to +19, where a -20
 * is a "high-priority" task, and a "+10" is a low-priority
 * task.
 *
 * We want the time-slice to be around 50ms or so, so this
 * calculation depends on the value of HZ.
 */
#if HZ < 200
#define TICK_SCALE(x) ((x) >> 2)
#elif HZ < 400
#define TICK_SCALE(x) ((x) >> 1)
#elif HZ < 800
#define TICK_SCALE(x) (x)
#elif HZ < 1600
#define TICK_SCALE(x) ((x) << 1)
#else
#define TICK_SCALE(x) ((x) << 2)
#endif

#define NICE_TO_TICKS(nice) (TICK_SCALE(20-(nice))+1)
```

Epochs

- The Linux scheduling algorithm works by dividing the CPU time into epochs
  - In a single epoch, every process has a specified time quantum whose duration is computed when the epoch begins
  - The epoch ends when all runnable tasks have exhausted their quantum
  - The scheduler recomputes the time quantum durations of all processes and a new epoch begins

- The base time quantum of a process is computed based on the nice value

Selecting the Next Task to Run

```
repeat_schedule:
  /*
   * Default process to select..
   */
  next = idle_task(this_cpu);
  c = -1000;
  list_for_each(tmp, &runqueue_head) {
    p = list_entry(tmp, struct task_struct, run_list);
    if (can_schedule(p, this_cpu)) {
      int weight = goodness(p, this_cpu, prev->active_mm);
      if (weight > c)
        c = weight, next = p;
    }
  }
```

Calculating Goodness

```
static inline int goodness(struct task_struct * p, int this_cpu,
                           struct mm_struct *this_mm) {
  int weight = -1;

  if (p->policy == SCHED_OTHER) {
    weight = p->counter;
    if (!weight) goto out;
    if (p->mm == this_mm || !p->mm)
      weight += 1;
    weight += 20 - p->nice;
    goto out;
  }
  weight = 1000 + p->rt_priority;
out:
  return weight;
}
```

weight = 0
p has exhausted its quantum

0 < weight < 1000
p is a conventional task

weight >= 1000
p is a real-time task

New Epoch

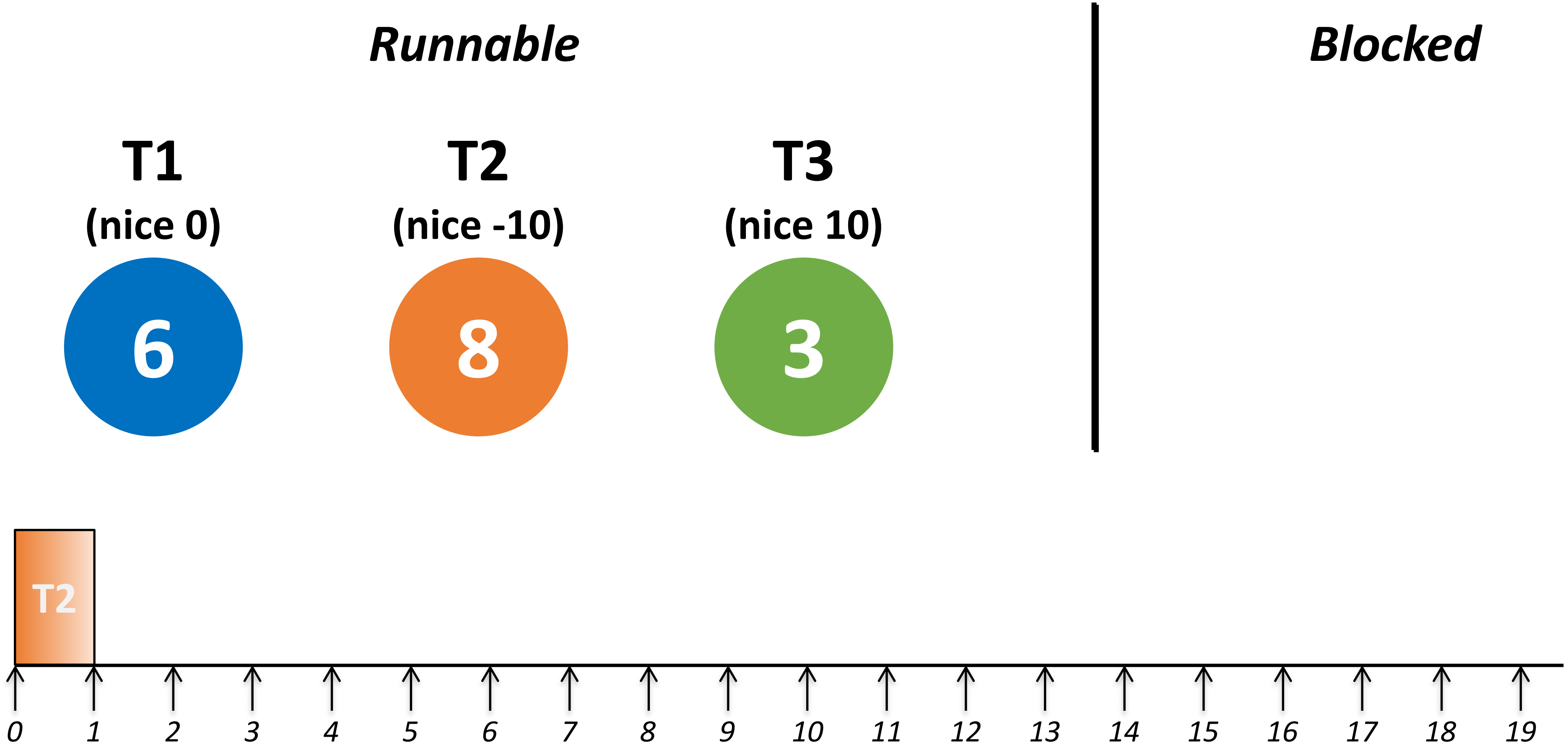
```
/* Do we need to re-calculate counters? */
if (unlikely(!c)) {
  struct task_struct *p;

  spin_unlock_irq(&runqueue_lock);
  read_lock(&tasklist_lock);
  for_each_task(p)
    p->counter = (p->counter >> 1) + NICE_TO_TICKS(p->nice);
  read_unlock(&tasklist_lock);
  spin_lock_irq(&runqueue_lock);
  goto repeat_schedule;
}
```

Preemption Condition

```
/*
 * the 'goodness value' of replacing a process on a given CPU.
 * positive value means 'replace', zero or negative means 'dont'.
 */
static inline int preemption_goodness(struct task_struct * prev,
                                      struct task_struct * p, int cpu)
{
  return goodness(p, cpu, prev->active_mm) - goodness(prev, cpu, prev->active_mm);
}
```

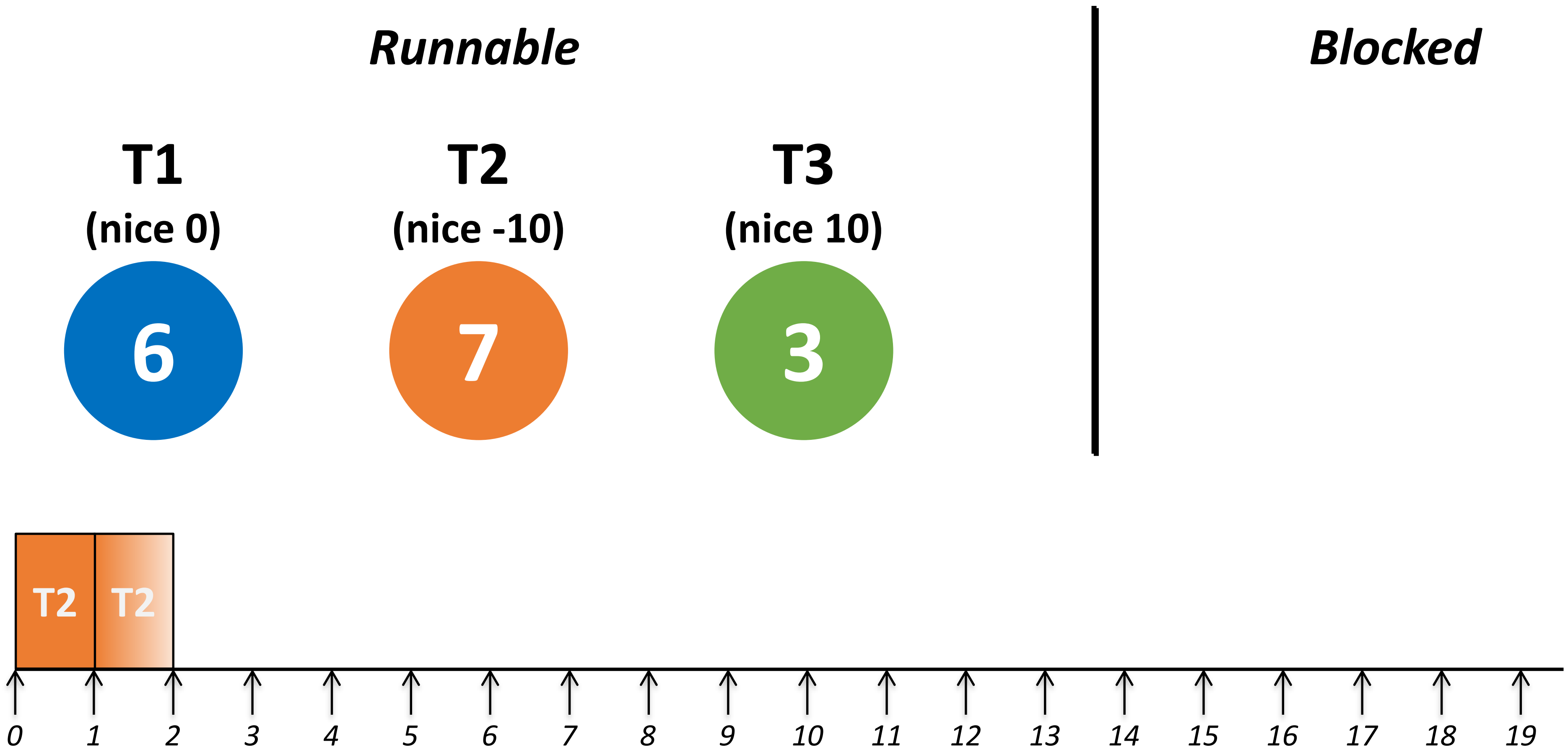
Example:

- Initially choose T2 among the three tasks in the run queue



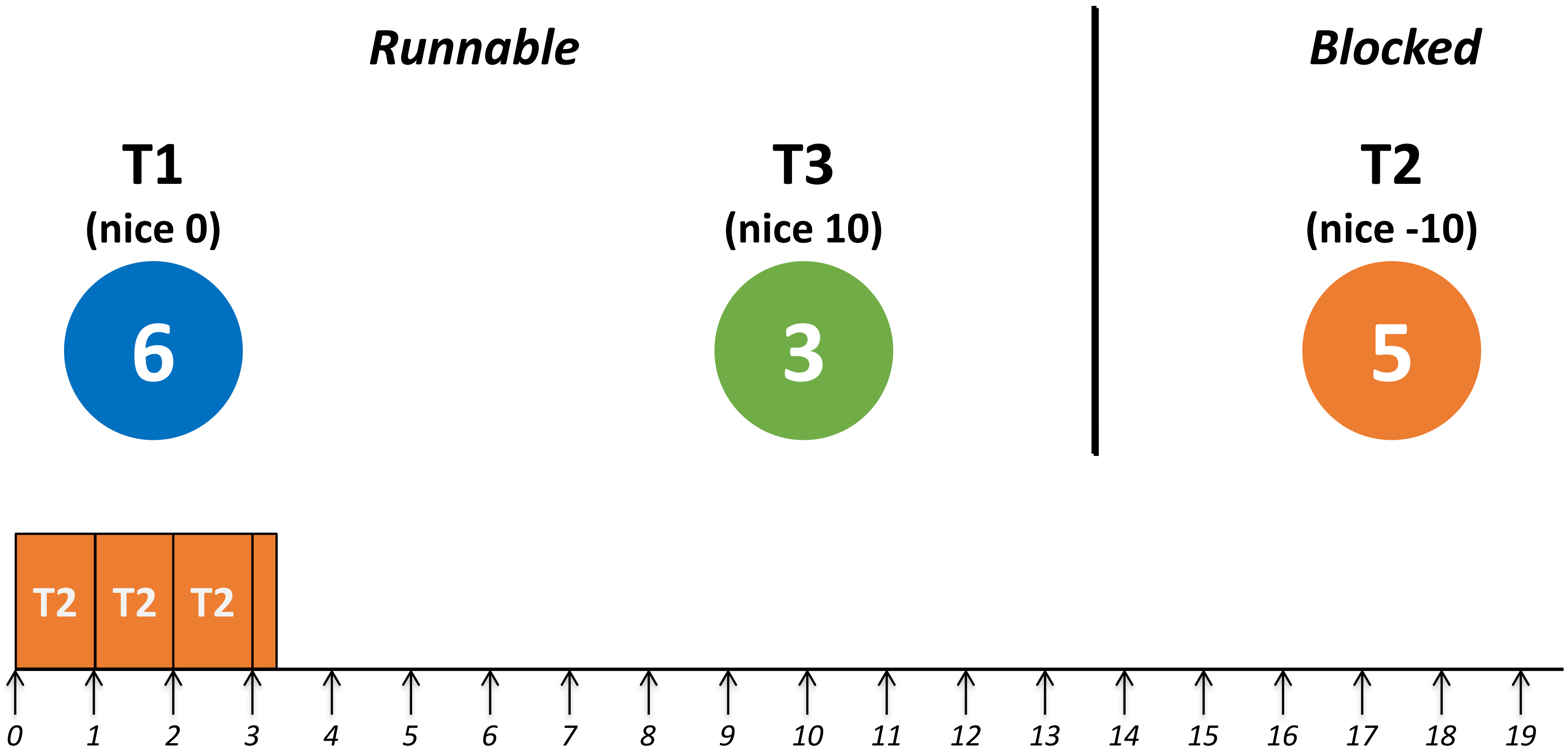
Example:

- At tick 1, decrement the counter and continue



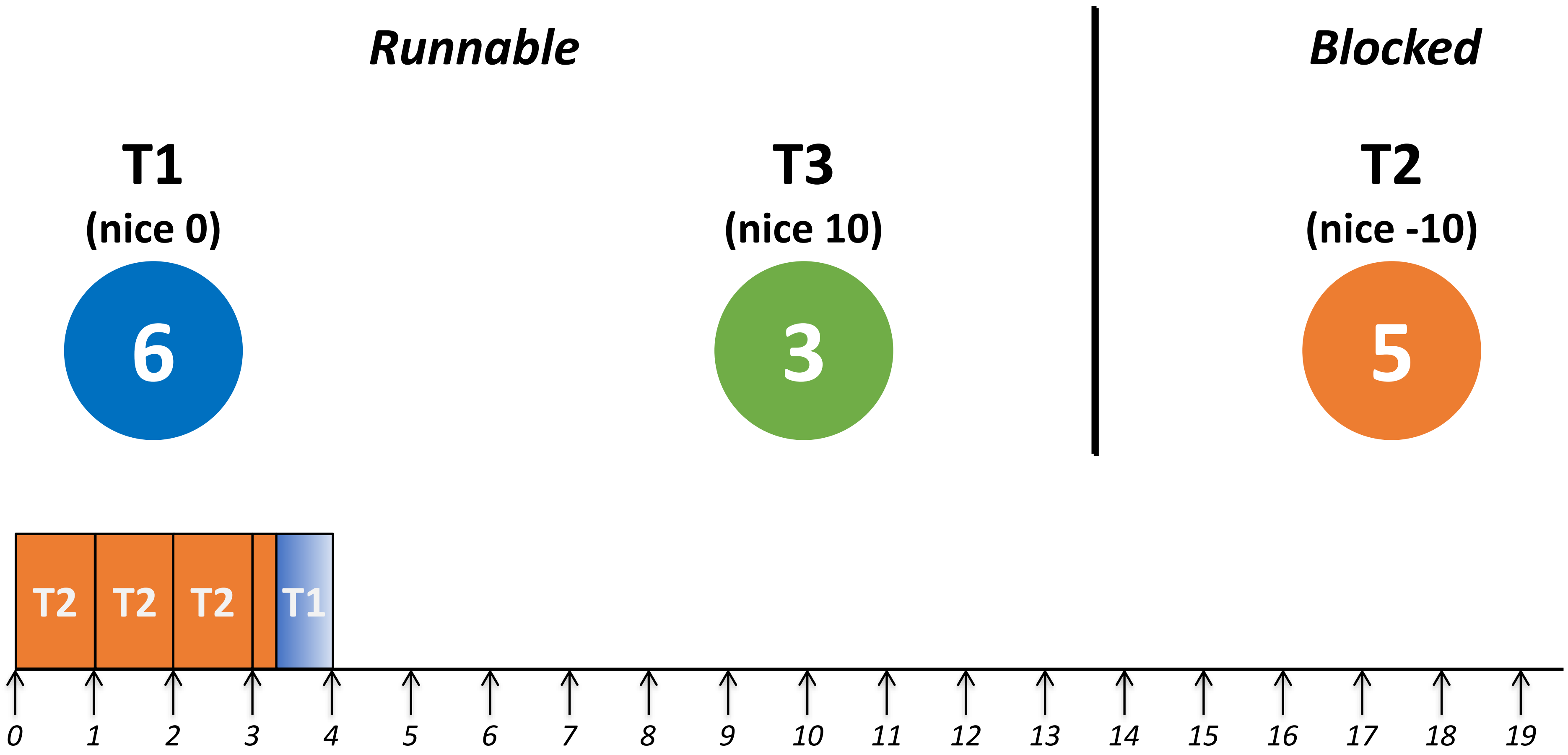
Example:

- Continue until T2 is blocked after tick 3



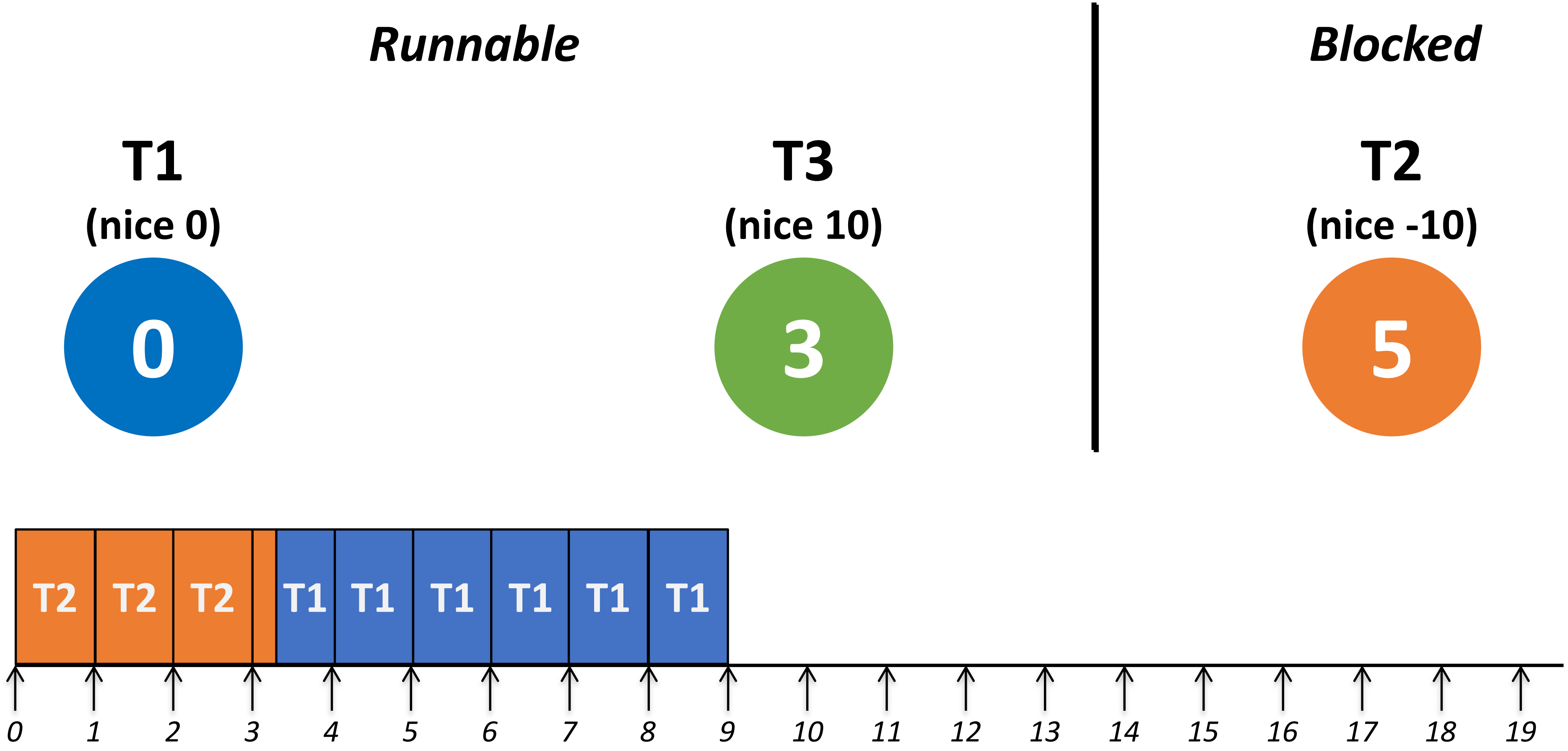
Example:

- Now choose T1



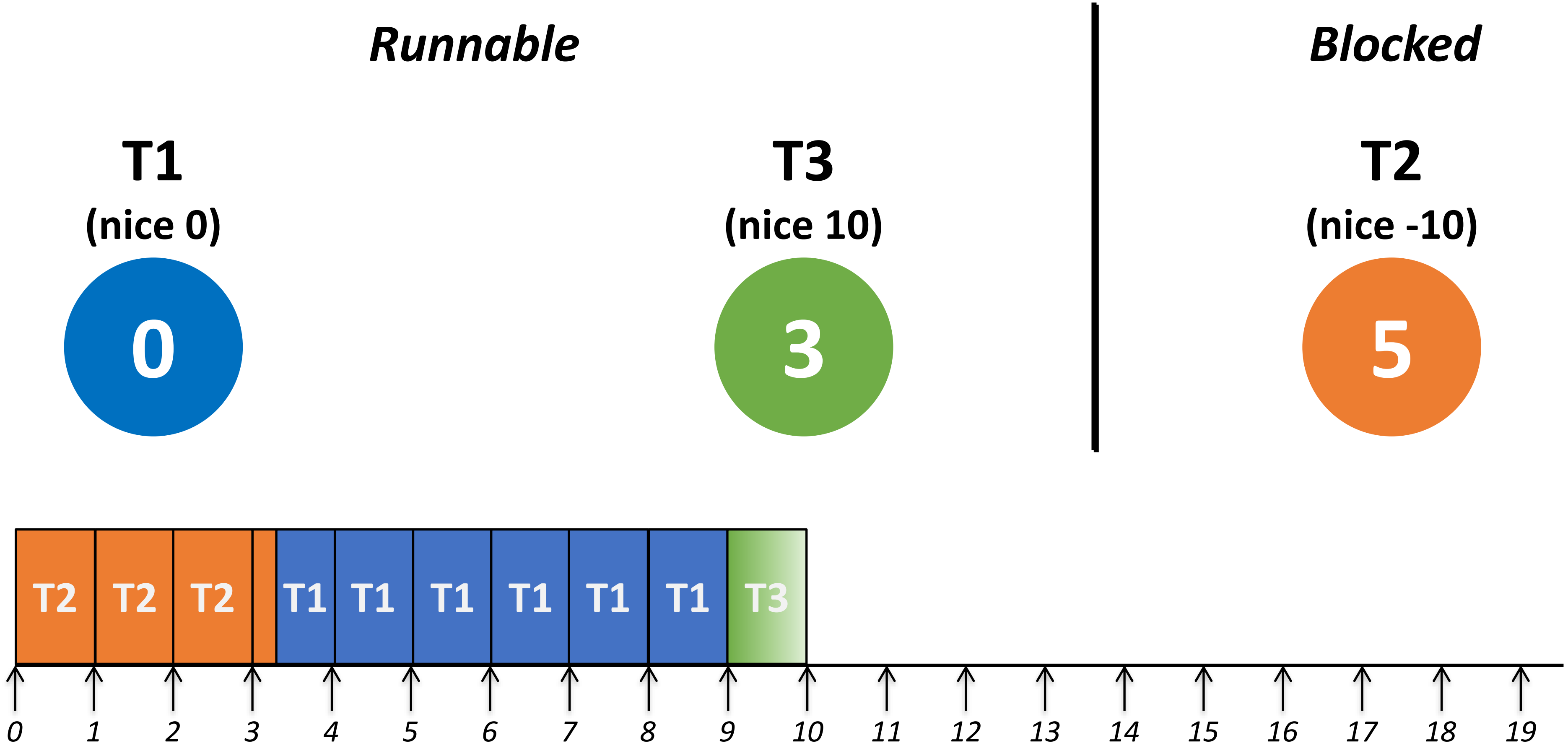
Example:

- T1 runs until it exhausts all the timeslice



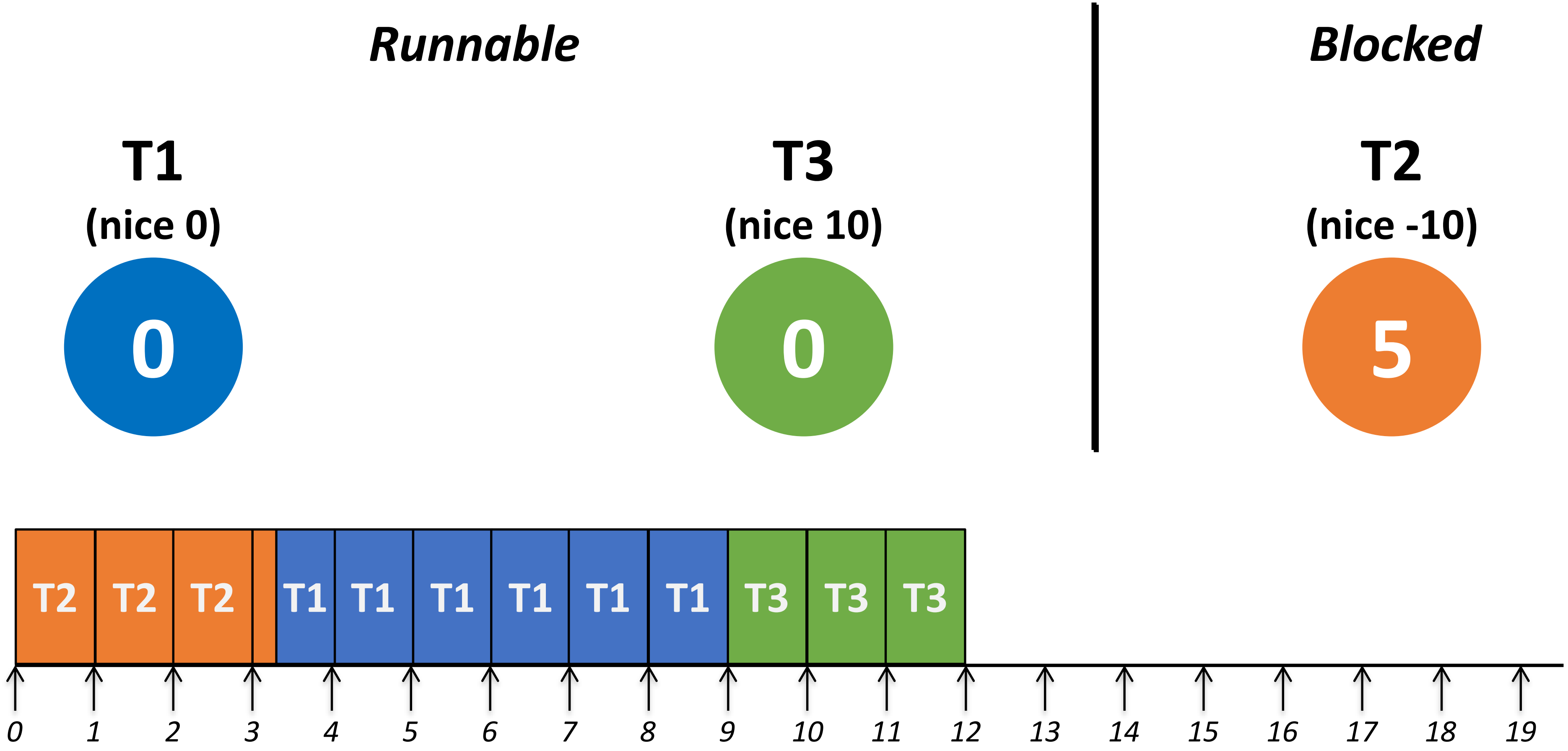
Example:

- Now schedule T3



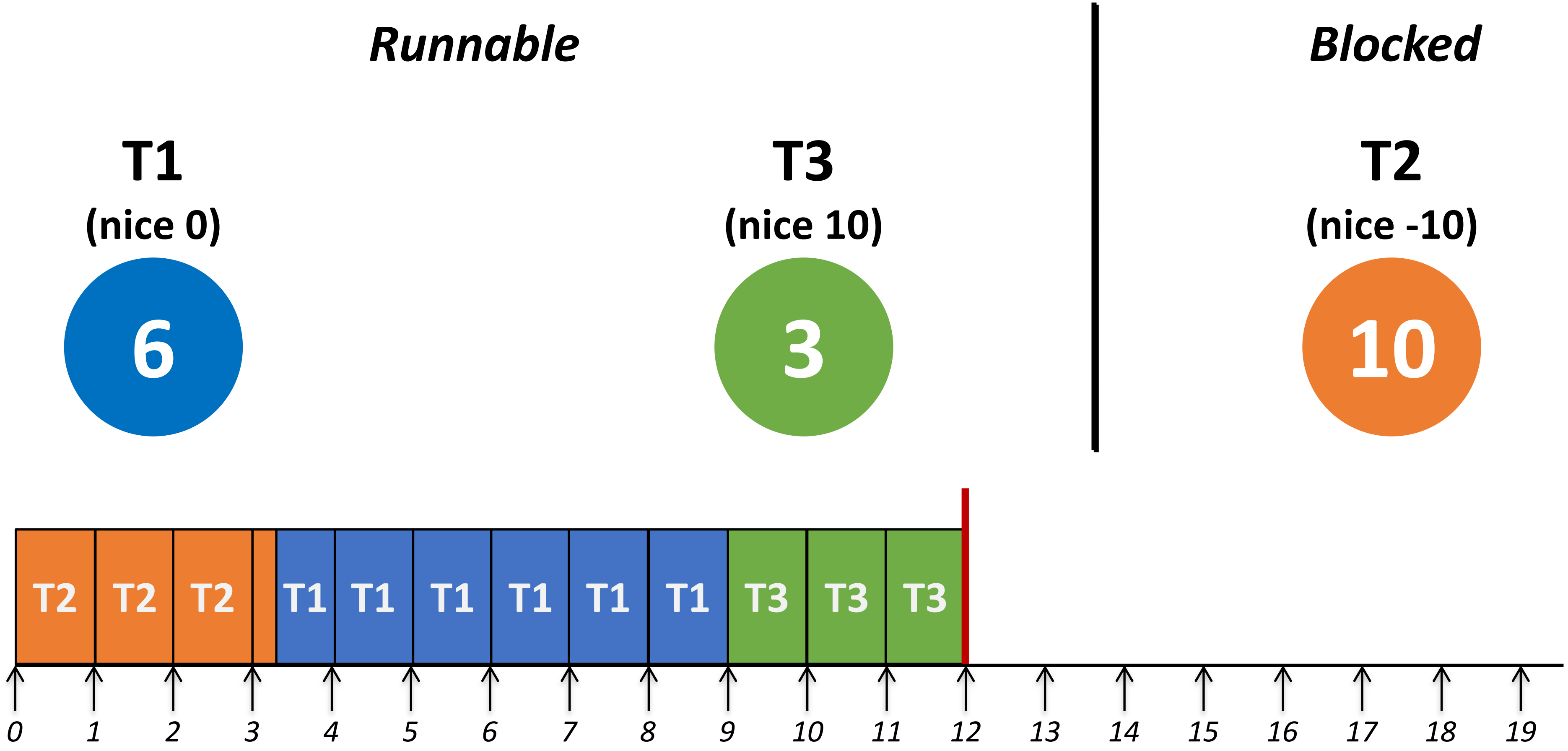
Example:

- T3 has also exhausted all the time slice



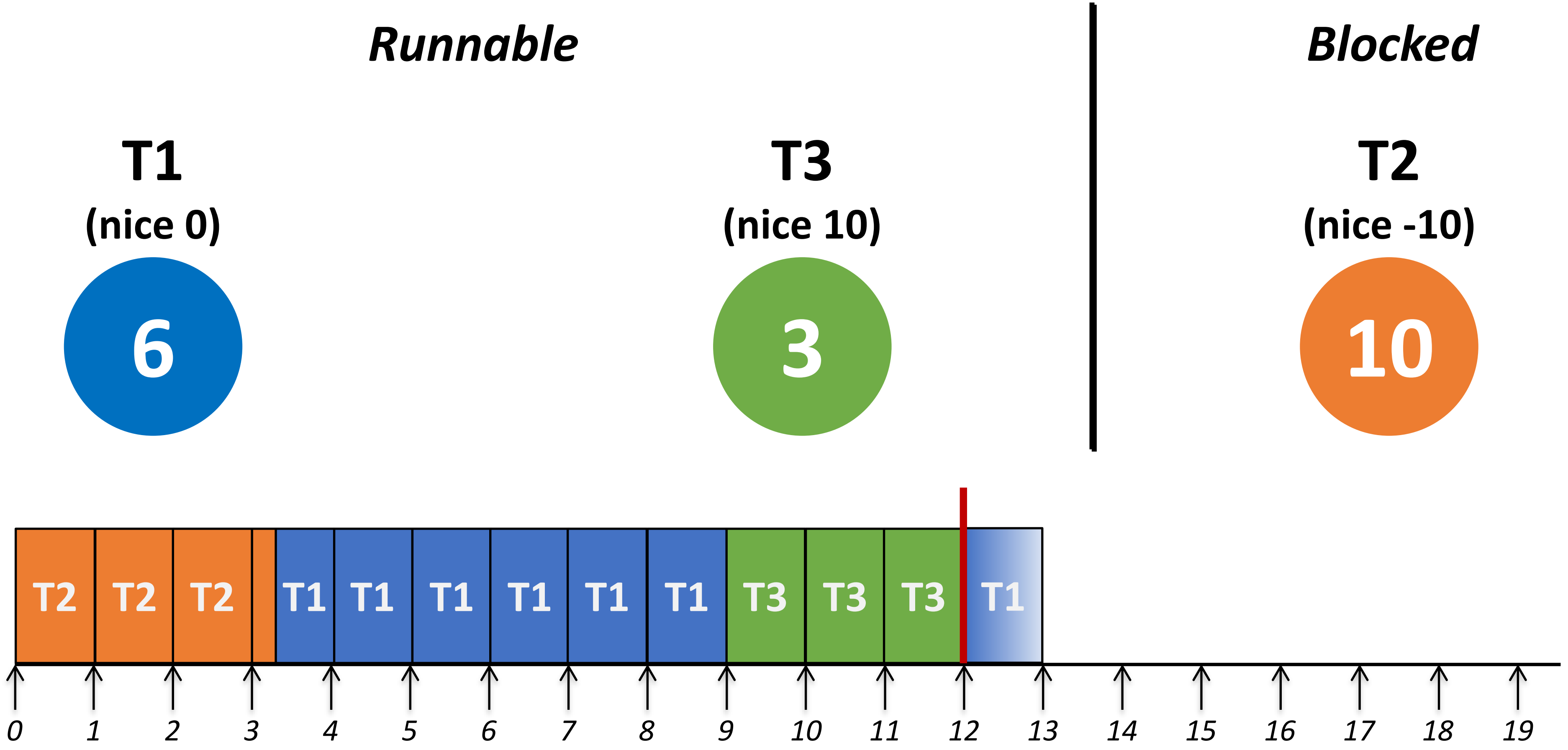
Example:

- Now start a new epoch with recalculating counters



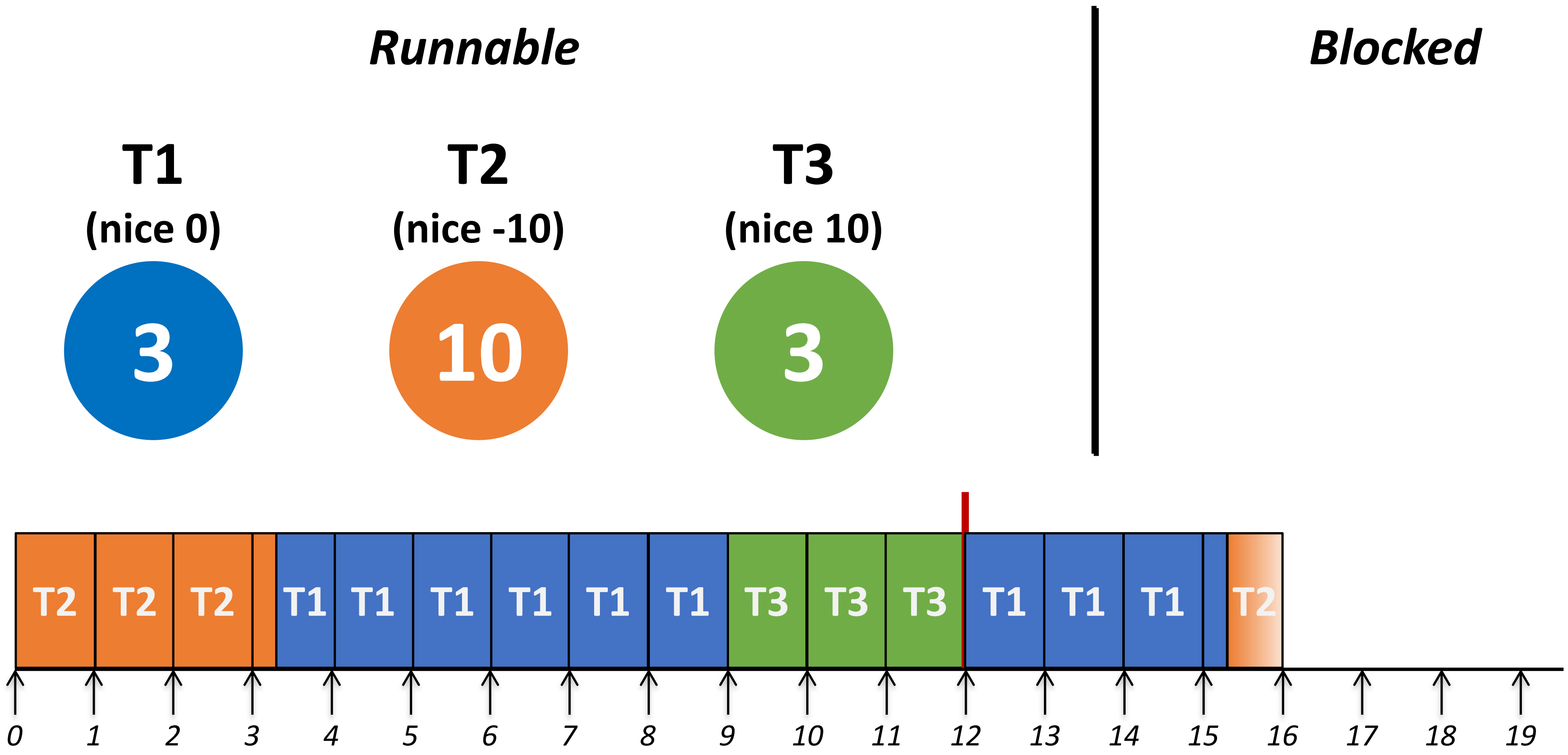
Example:

- Schedule T1



Example:

- T2 is woken up after tick 15, and it preempts T1

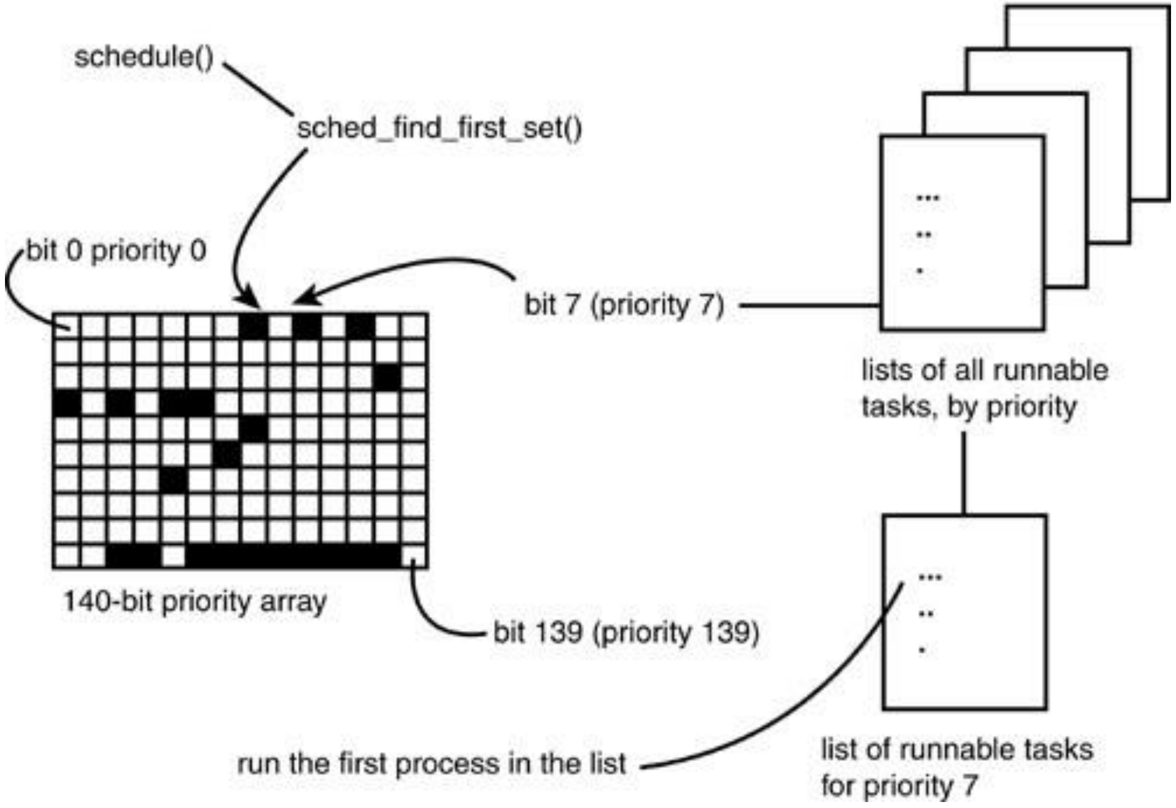


Problems

- O(n) operations
  - When to choose the next task to run
  - When to recalculate counters for each epoch
  - Example: During the execution of VolanoMark, 37~55% of the total time spent in the kernel is spent in the scheduler (for handling 400 ~ 2000 threads)
- Lock contention in the multi-core systems
  - A single runqueue is shared by all the cores
- I/O-bound task is seldom boosted under high system load

Linux 2.6 Scheduler

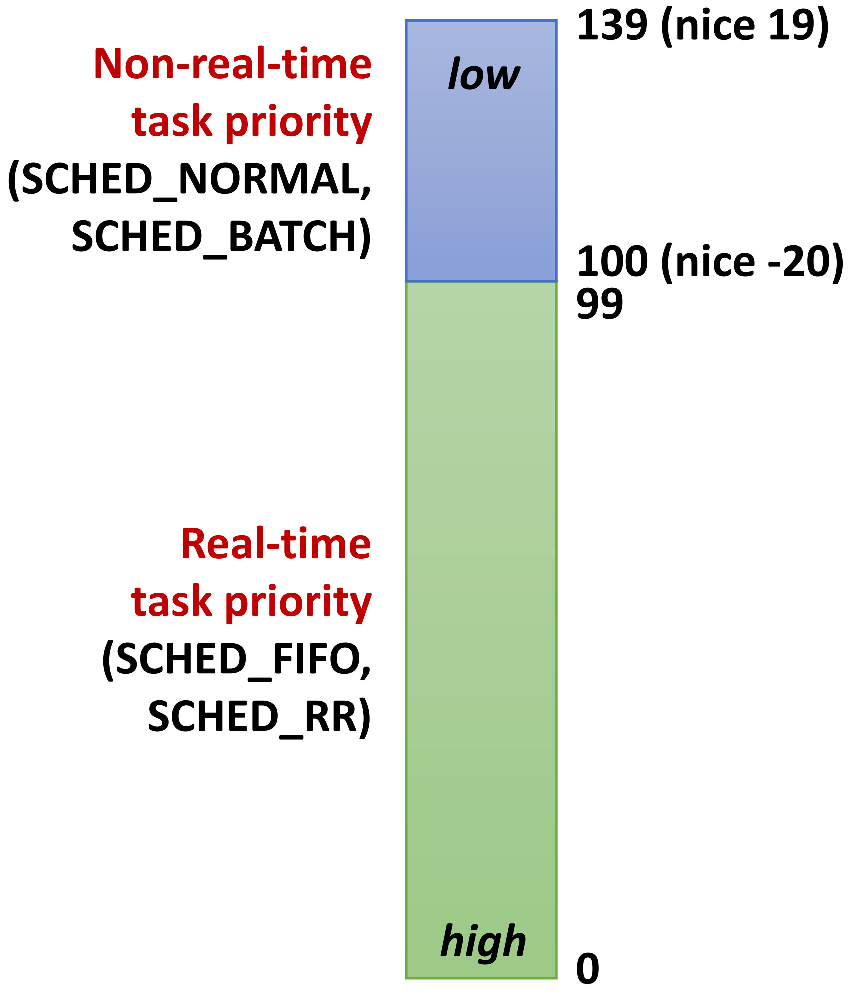
- New priority scheme: 140 levels (0 ~ 139)
  - Normal tasks: 120 + nice ([-20, 19])
  - Real-time tasks: 0 ~ 99
  - Dynamic priority control based on interactivity (e.g., average sleep time)
- O(1) scheduling
  - Active and expired array
  - Each priority array contains a queue of runnable tasks per each priority level
  - Each array also has a bitmap
- Each processor has its own run queue



Linux CFS
(Completely Fair Scheduler)

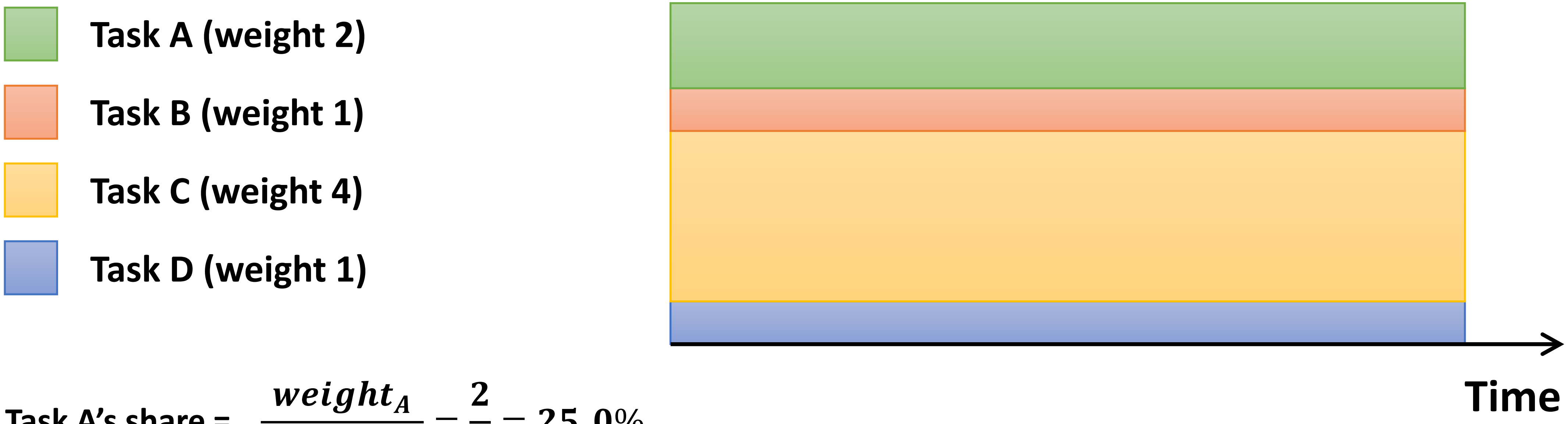
Linux Task Priority

- Total 140 levels (0 ~ 139)
  - A smaller value means higher priority
- Setting priority for non-real-time tasks
  - `nice()`, `setpriority()`
  - -20 ≤ nice value ≤ 19
  - Default nice value = 0 (priority value 120)
- Setting priority for real-time tasks
  - `sched_setattr()`
  - Static priority for SCHED_FIFO & SCHED_RR
  - Runtime, deadline, period for SCHED_DEADLINE



Proportional Share Scheduling

- Basic concept
  - A weight value is associated with each task
  - The CPU is allocated to task in proportion to its weight



Task A's share = $\frac{weight_A}{\sum weight_i} = \frac{2}{8} = 25.0\%$

Nice to Weight

- How to map nice values to weights?
  - Wants a task to get ~10% less CPU time when it goes from nice i to nice $i+1$
  - This will make another task remained on nice i have ~10% more CPU time
  - weight(i)/weight($i+1$) = 0.55/0.45 = 1.22 (or ≃ 25% increase)
- Examples
  - T_1 (nice 0), T_2 (nice 1)
    - T_1: 1024/(1024+820) = 55.5%
    - T_2: 820/(1024+820) = 44.5%
  - + T_3 (nice 1)
    - T_1: 1024/(1024+820*2) = 38.4%
    - T_2: 820/(1024+820*2) = 30.8%
    - T_3: 820/(1024+820*2) = 30.8%

```
const int sched_prio_to_weight[40] = {
 /* -20 */     88761,     71755,     56483,     46273,     36291,
 /* -15 */     29154,     23254,     18705,     14949,     11916,
 /* -10 */      9548,      7620,      6100,      4904,      3906,
 /*  -5 */      3121,      2501,      1991,      1586,      1277,
 /*   0 */      1024,       820,       655,       526,       423,
 /*   5 */       335,       272,       215,       172,       137,
 /*  10 */       110,        87,        70,        56,        45,
 /*  15 */        36,        29,        23,        18,        15,
};
```

Virtual Runtime

- Approximate the "ideal multitasking" that CFS is modeling
- Normalize the actual runtime to the case with nice value 0

$$VR(T) = \frac{Weight_0}{Weight(T)} \times PR(T) = \left(Weight_0 \times \boxed{\frac{2^{32}}{Weight(T)}} \times PR(T) \right) \gg 32$$

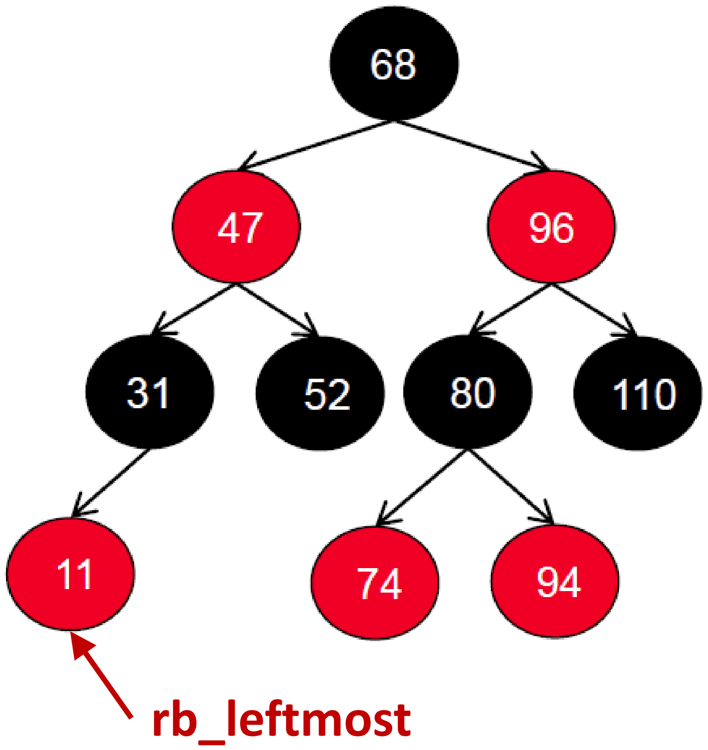
precomputed: `sched_prio_to_wmult[]`

  - $Weight_0$: the weight of nice value 0
  - *Weight(T)*: the weight of the task T
  - *PR(T)*: the actual runtime of the task T
  - *VR(T)*: the virtual runtime (*vruntime*) of the task T

- For a high-priority task, its *vruntime* increases slowly

Runqueue

- CFS maintains a red-black tree where all runnable tasks are sorted by *vruntime*
  - Self-balancing binary search tree
  - The path from the root to the farthest leaf is no more than twice as long as the path to the nearest leaf
  - Tree operations in O(log N) time
  - The leftmost node indicates the smallest *vruntime*



- Choose the task with the smallest virtual runtime (*vruntime*)
  - Small virtual runtime means that the task has received less CPU time than what it should have received

Timeslice

- The time a task runs before it is preempted
  - It gives each runnable task a slice of the CPU's time
  - The length of timeslice of a task is proportional to its weight

$$TS(T) = \frac{Weight(T)}{\sum_{T_i \text{ in } RQ} Weight(T_i)} \times P$$

  - *TS(T)*: Ideal runtime for the task *T*
  - *P*: Scheduling period

$$P = \begin{cases} \text{sysctl\_sched\_latency}, & \textit{if } n < \textit{sched\_nr\_latency} \\ \text{sysctl\_sched\_min\_granularity} * n, & \textit{otherwise} \end{cases}$$

sysctl_sched_latency:
Targeted preemption latency for CPU-bound tasks (6ms*(1+log #cores) by default)
sysctl_sched_min_granularity:
Minimal preemption granularity for CPU-bound tasks (0.75ms*(1+log #cores) by default)
sched_nr_latency = sysctl_sched_latency / sysctl_sched_min_granularity (8 by default)

Scheduling Flow

- Timer interrupt handler calls the CFS scheduler
- Updates the *vruntime* of the current task
- If preemption is needed, mark the NEED_RESCHED flag
  - When the current task has run beyond its timeslice
  - If the current task's *vruntime* exceeds the *vruntime* of the leftmost task in RB tree
- On exit, `schedule()` is called when NEED_RESCHED flag is set
  - Clear the NEED_RESCHED flag and enqueue the previous task
  - Pick the next task to run
  - Context switch to the next task
- The current task can be also preempted when a higher-priority task is inserted into the runqueue

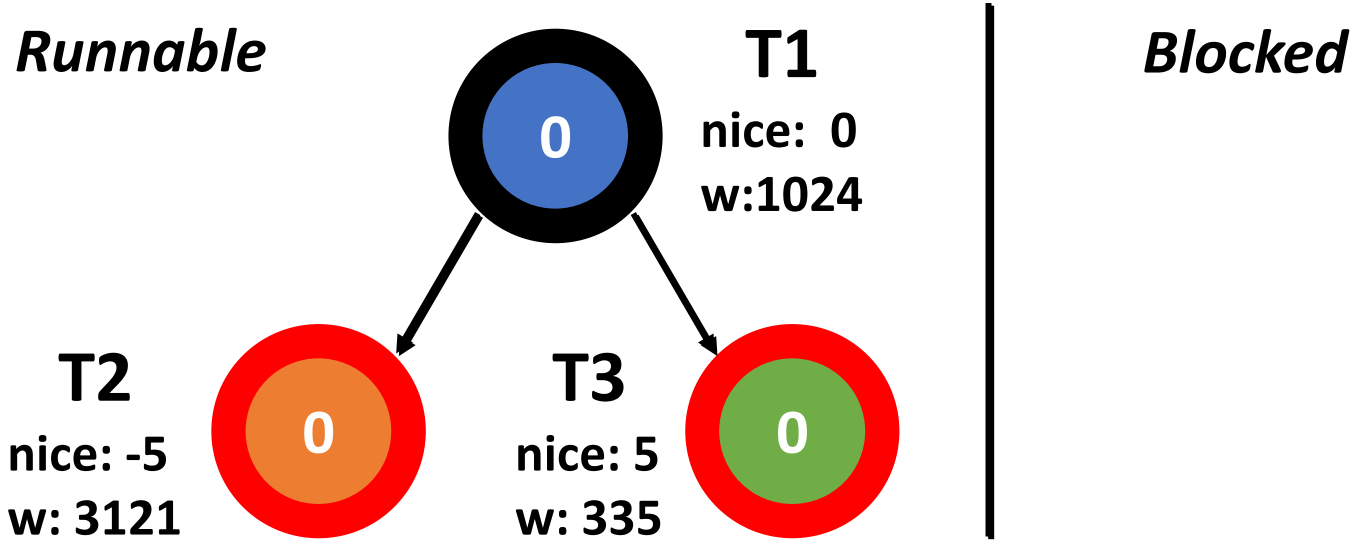
Example:

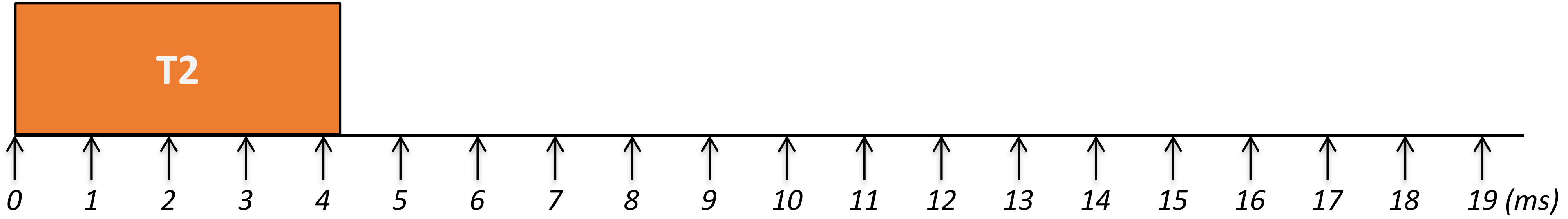
- Initially choose the leftmost task, T2, in this case
- But how long?

$$TS(T2)$$

$$= \frac{3121}{1024 + 3121 + 335} \times P$$

$$= 4.18\ ms$$

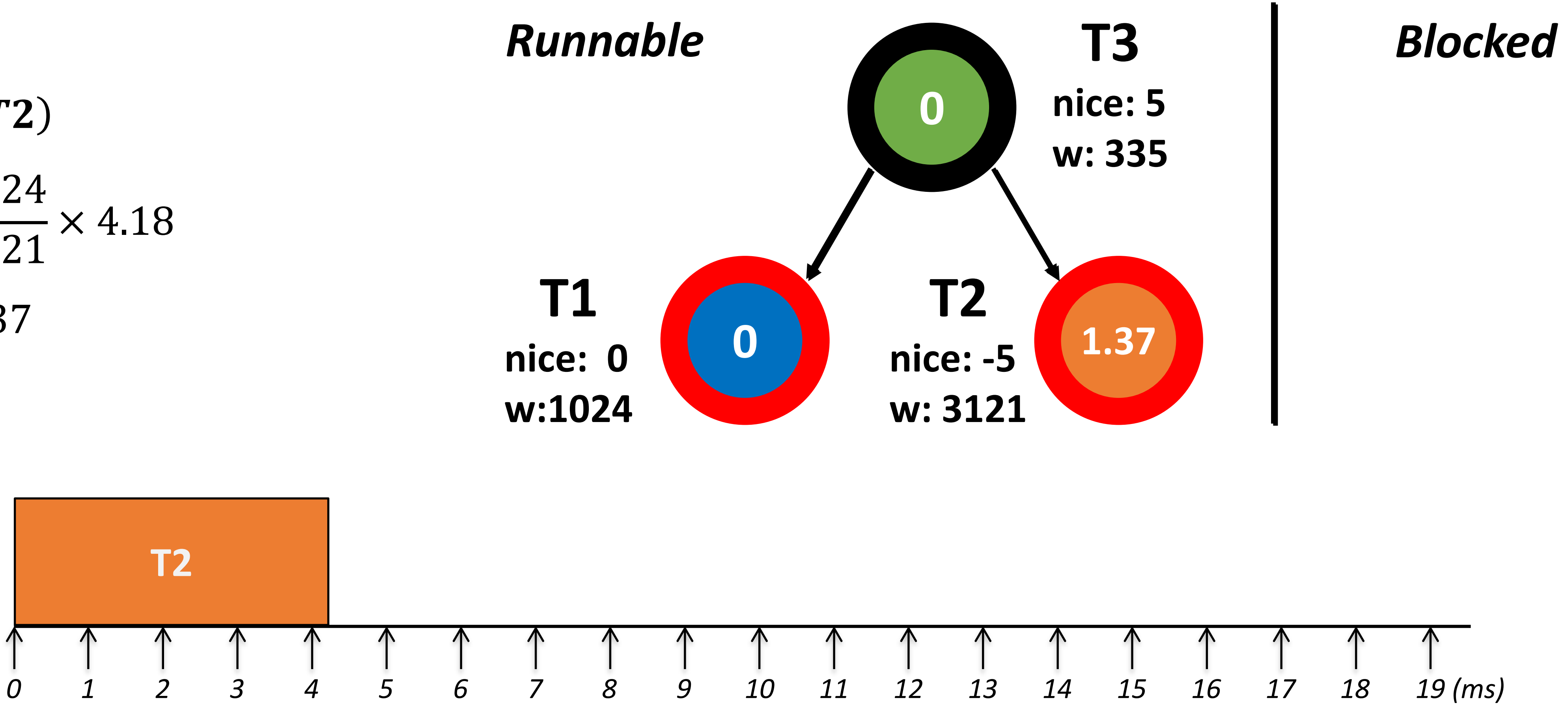




Example:

- Update T2's vruntime

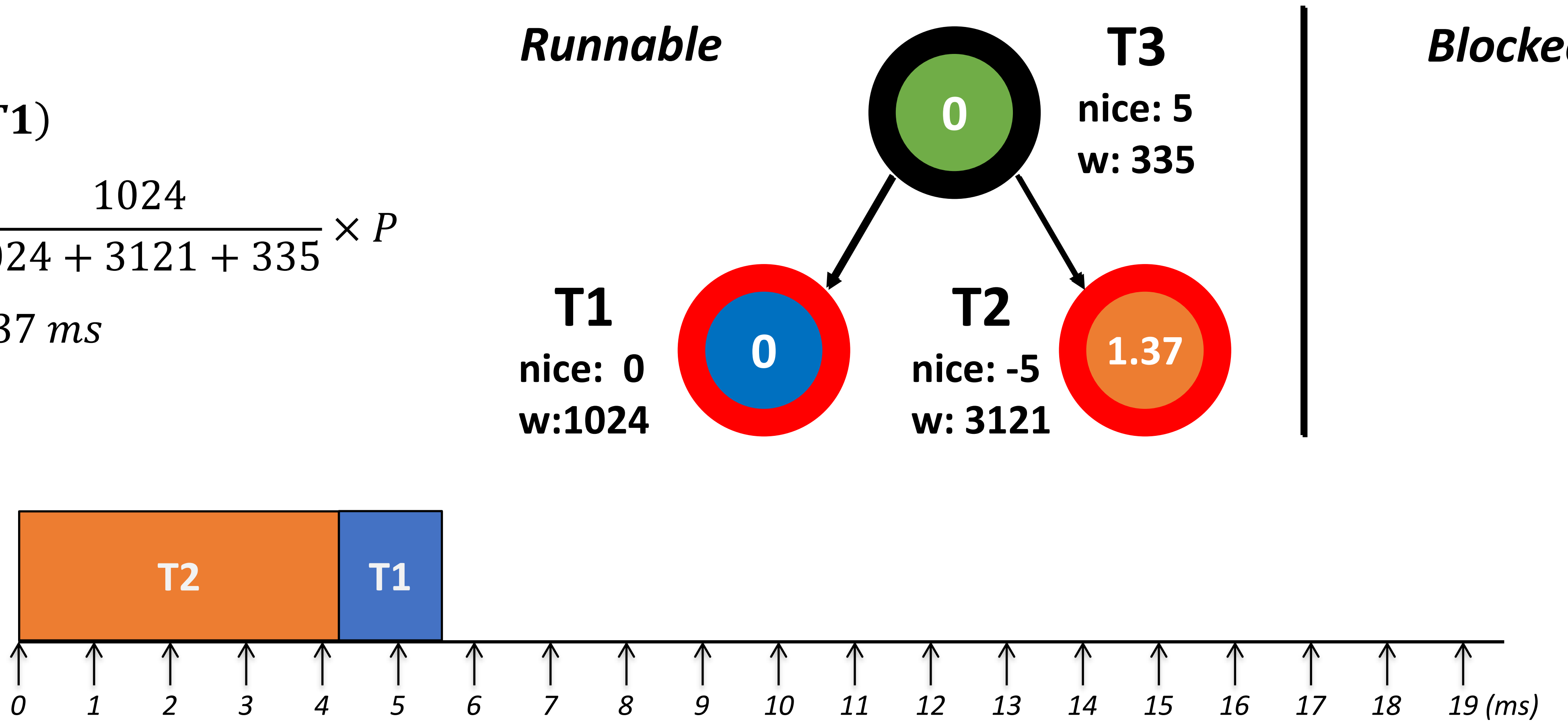
$$VR(T2) = \frac{1024}{3121} \times 4.18 = 1.37$$



Example:

- Now choose T1

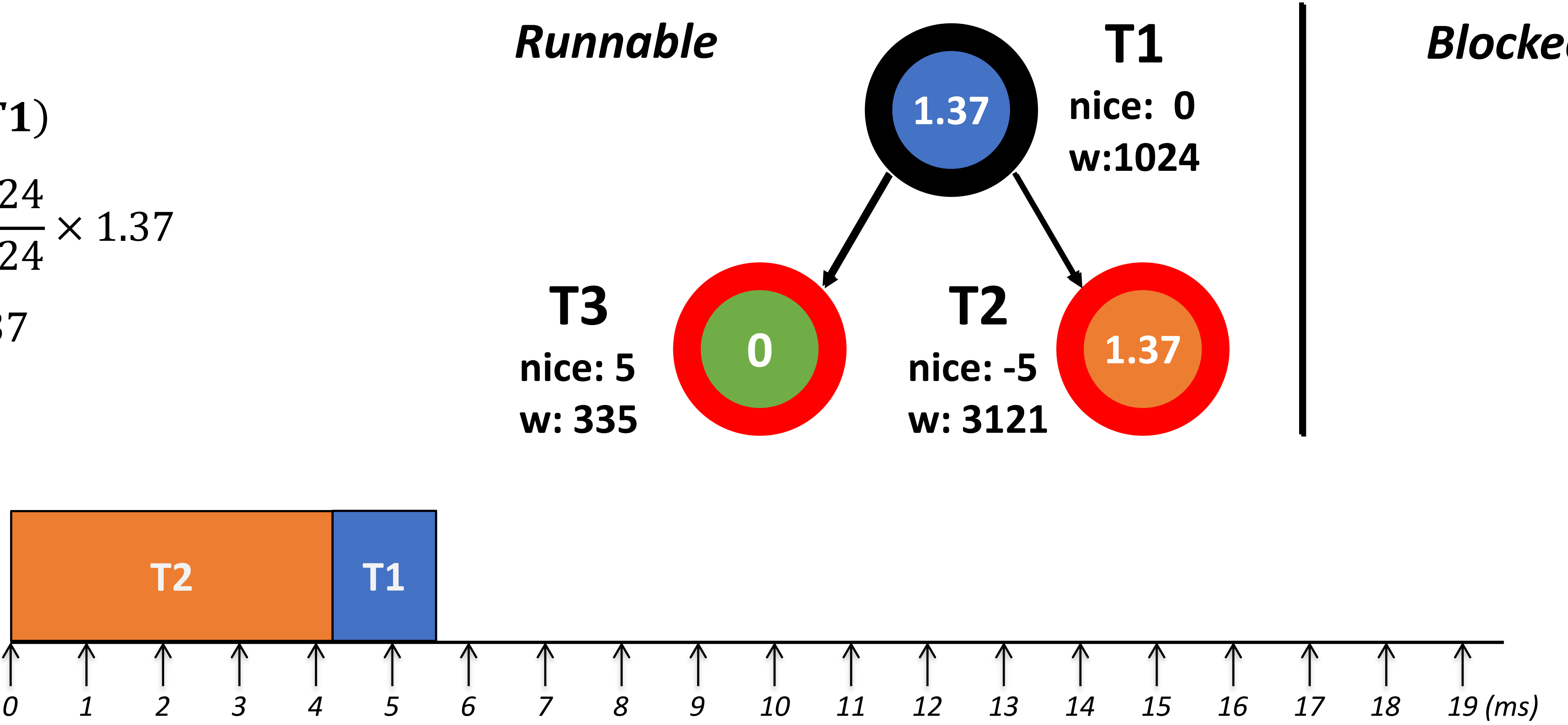
$$TS(T1) = \frac{1024}{1024 + 3121 + 335} \times P = 1.37\ ms$$



Example:

- Update T1's runtime

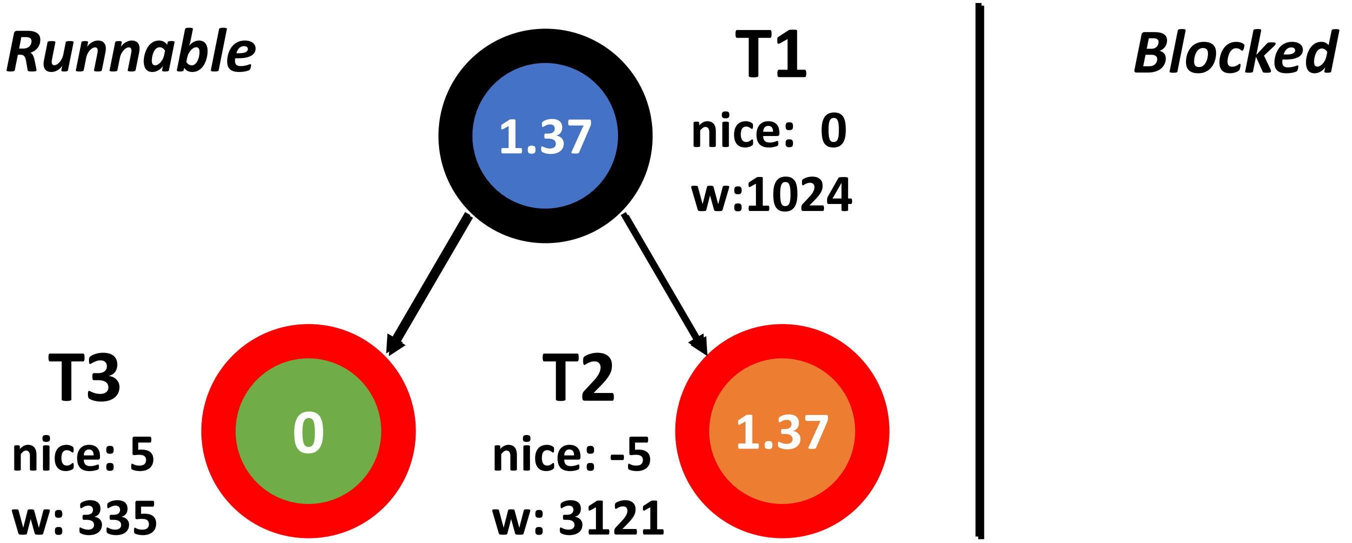
$$VR(T1) = \frac{1024}{1024} \times 1.37 = 1.37$$

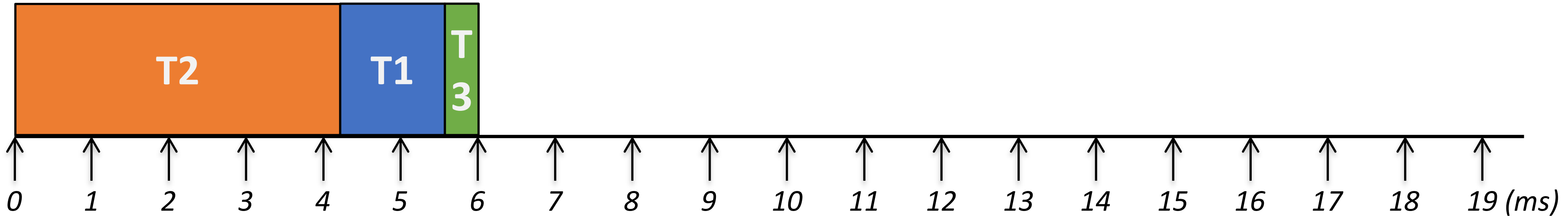


Example:

- Choose T3

$$TS(T3) = \frac{335}{1024 + 3121 + 335} \times P = 0.45\ ms$$





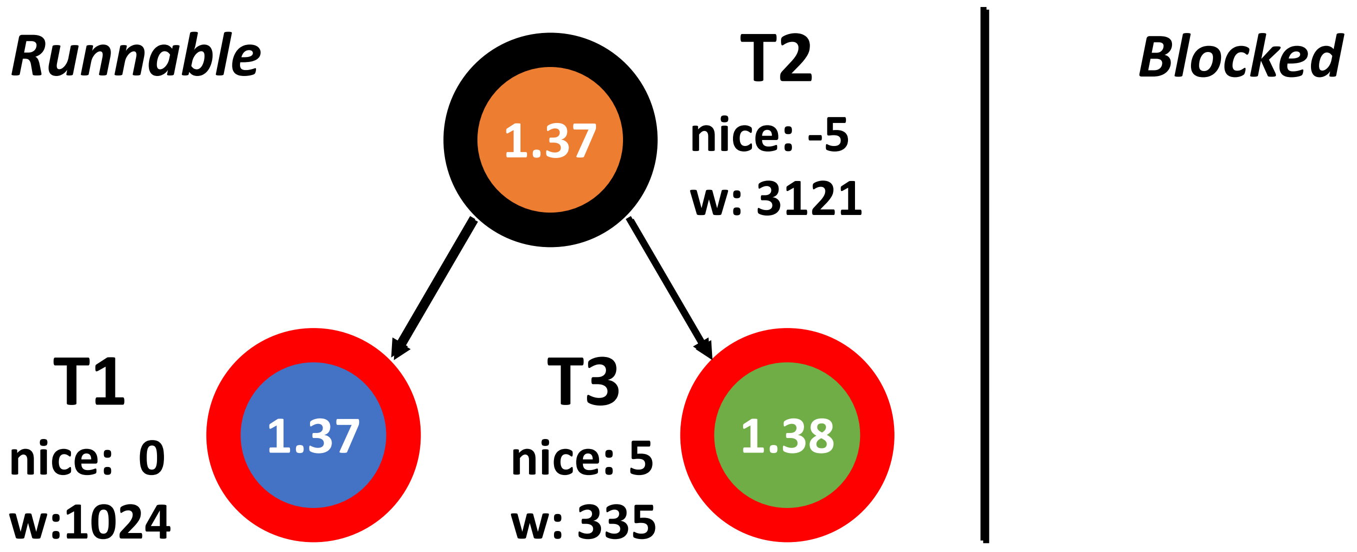
Example:

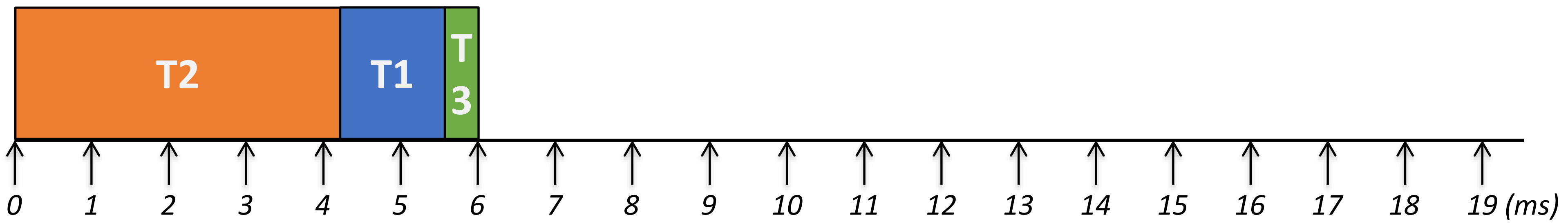
- Update T3's vruntime

$$VR(T3)$$

$$= \frac{1024}{335} \times 0.45$$

$$= 1.38$$

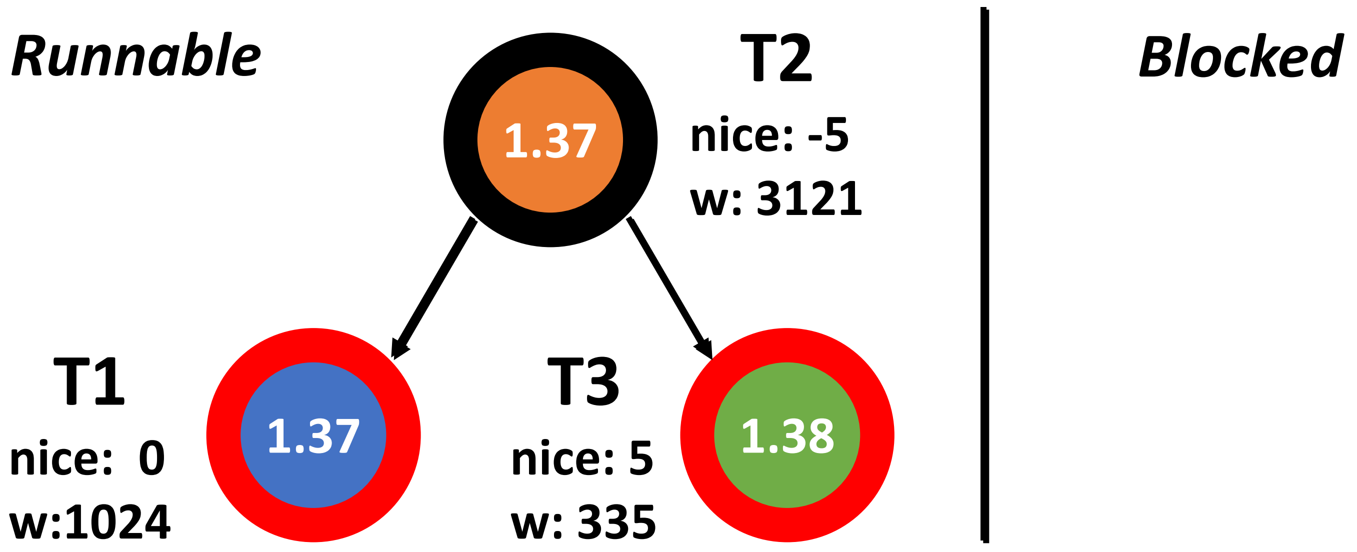


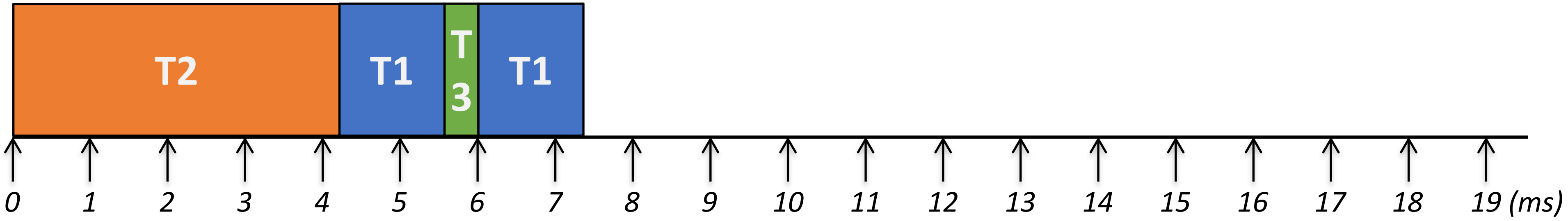


Example:

- Choose T1

$$TS(T1) = \frac{1024}{1024 + 3121 + 335} \times P = 1.37\ ms$$

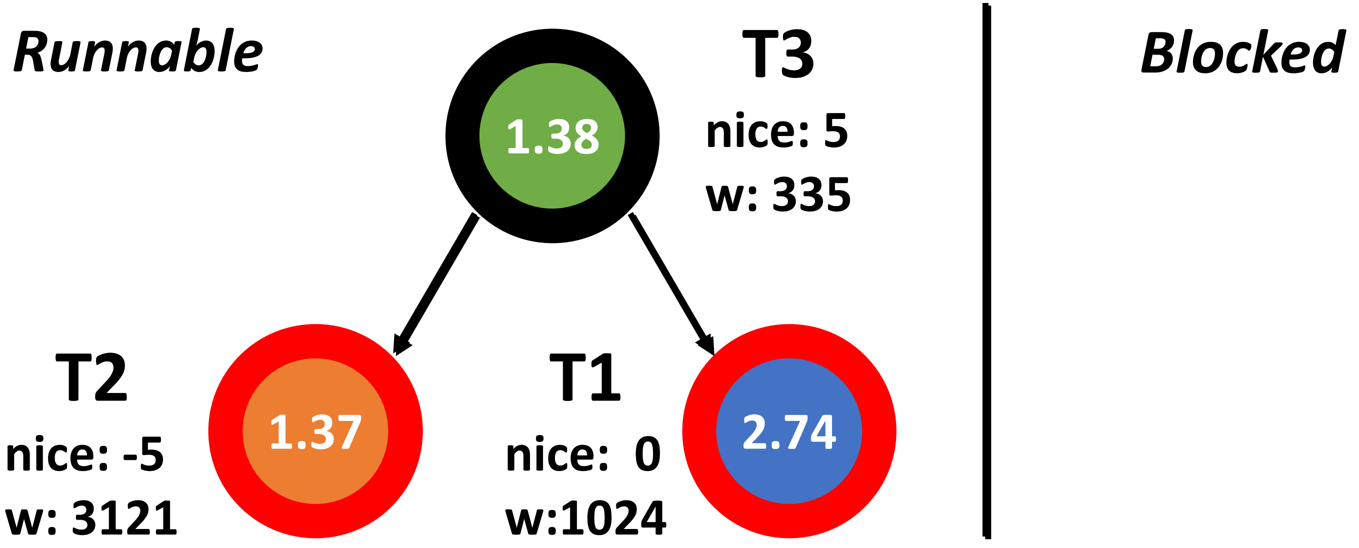


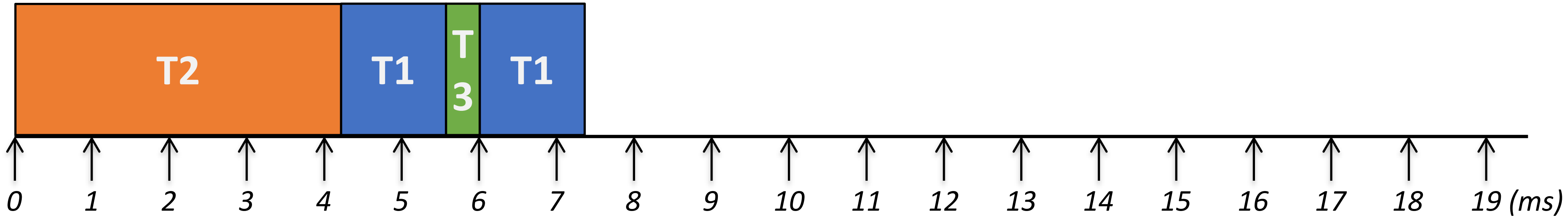


Example:

- Update T1's vruntime

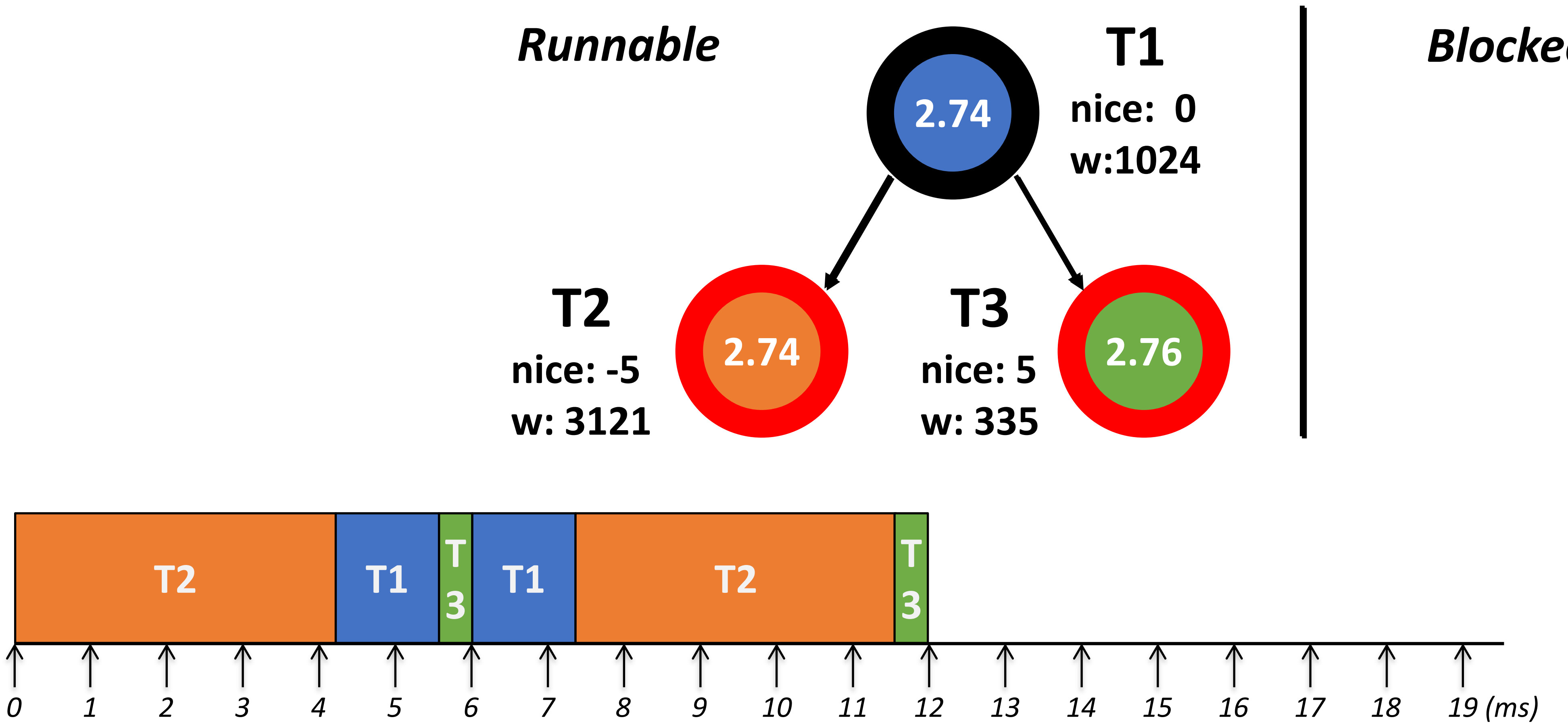
$$VR(T1)$$
$$= 1.37 + \frac{1024}{1024} \times 1.37$$
$$= 2.74$$





Example:

- Update T2 for 4.18ms and T3 for 0.45ms



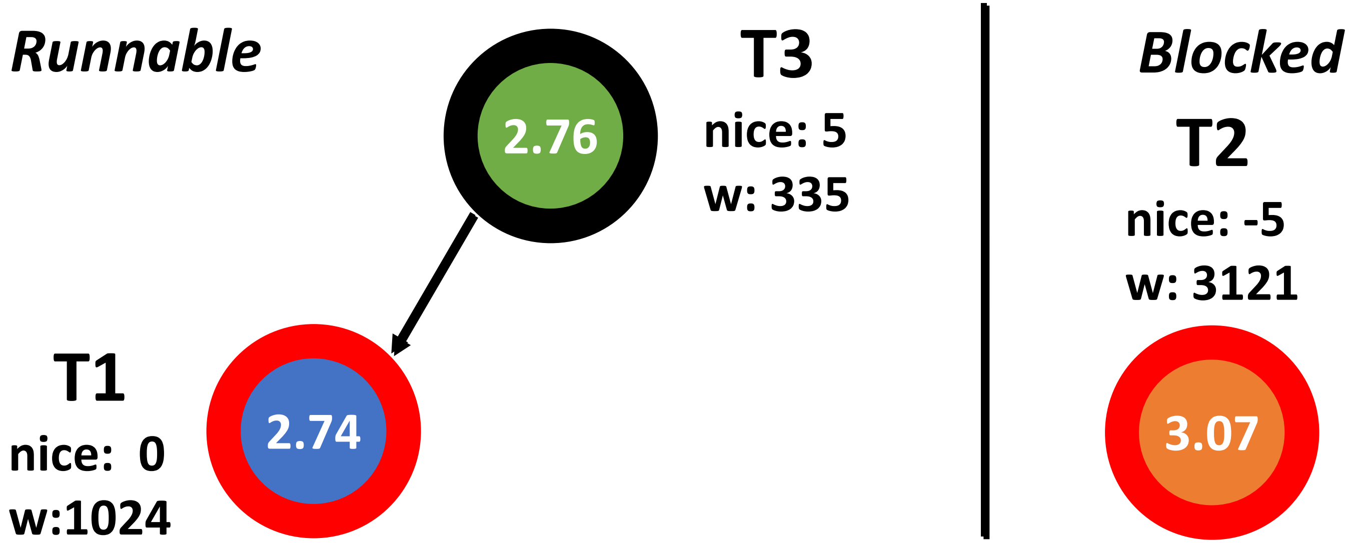
Example:

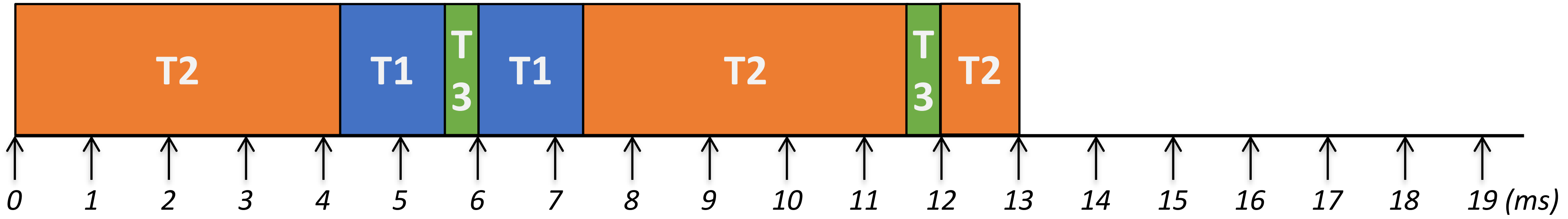
- Now T2 is scheduled, but it is blocked after running 1ms

$$VR(T2)$$

$$= 2.74 + \frac{1024}{3121} \times 1.00$$

$$= 3.07$$





Example:

- Now T1 runs

$TS(T1)$

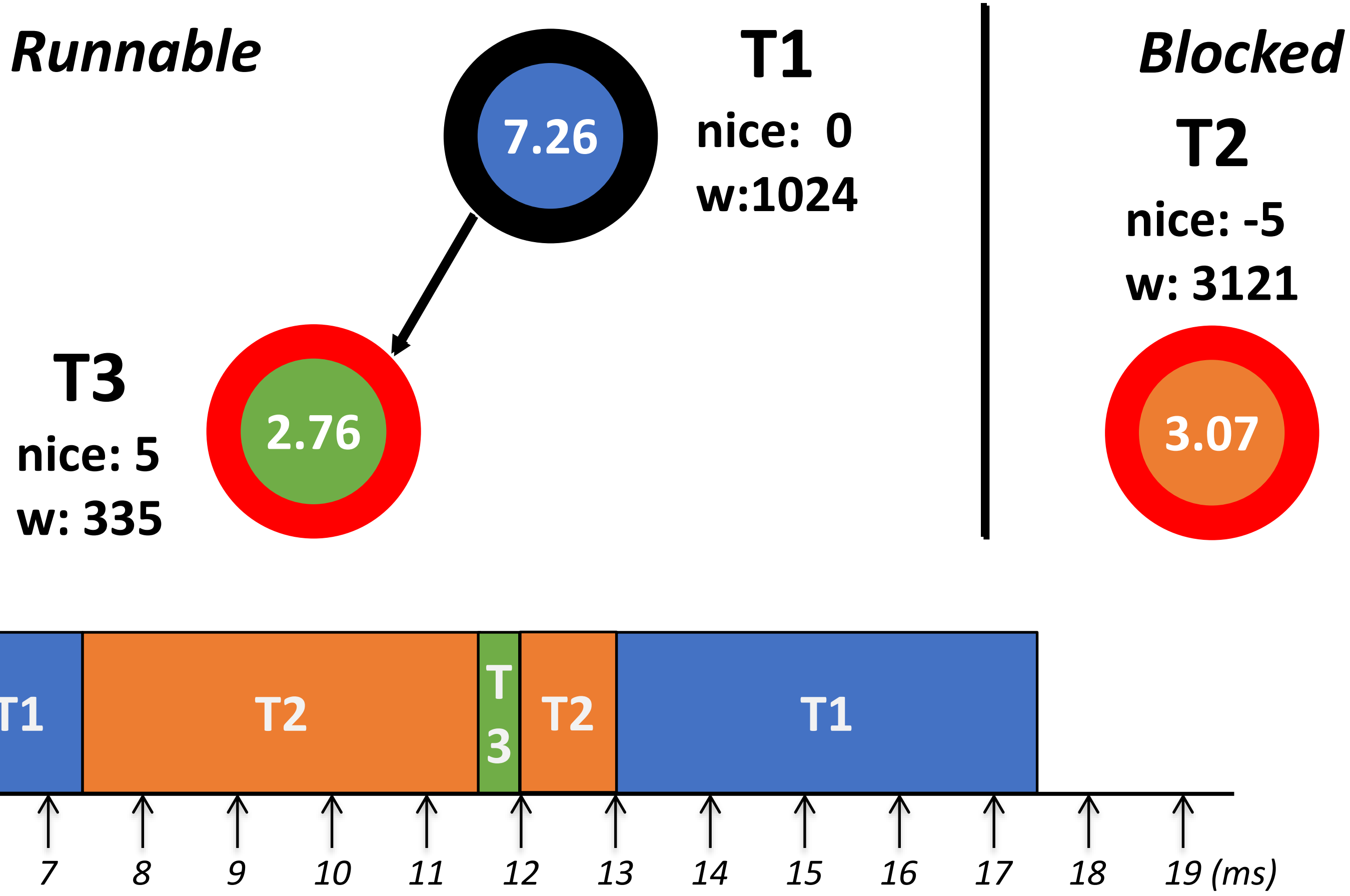
$= \frac{1024}{1024 + 335} \times P$

$= 4.52\ ms$

$VR(T1)$

$= 2.74 + \frac{1024}{1024} \times 4.52$

$= 7.26$



Example:

- T3 runs

$\boldsymbol{TS(T3)}$

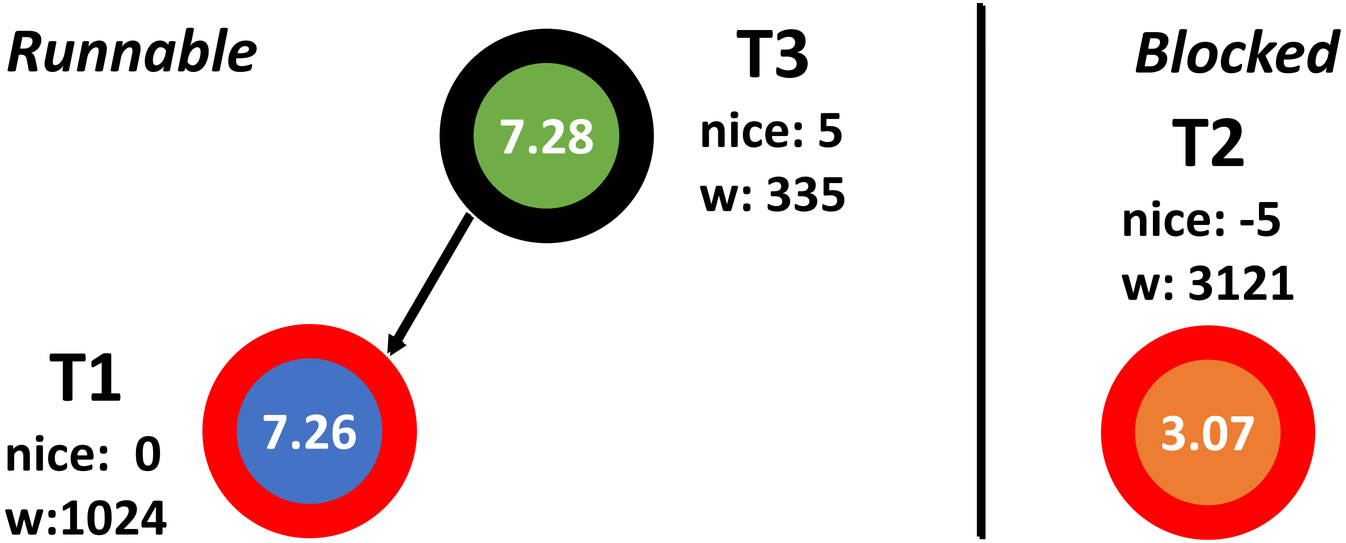
$$= \frac{335}{1024 + 335} \times P$$

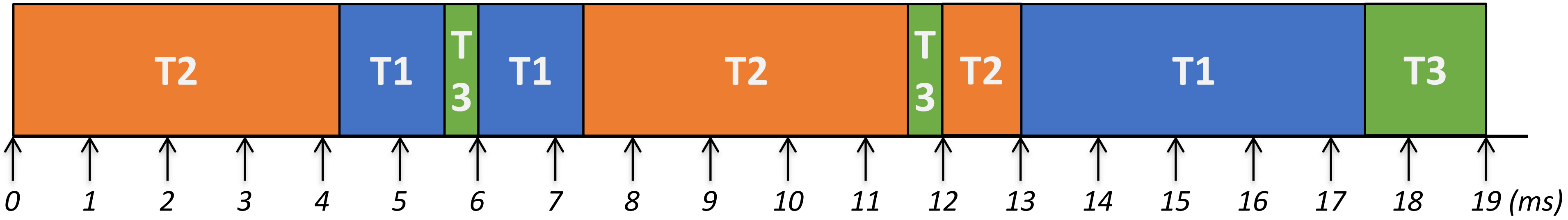
$$= 1.48\ ms$$

$\boldsymbol{VR(T3)}$

$$= 2.76 + \frac{1024}{335} \times 1.48$$

$$= 7.28$$





Tickless (or DynTick) Kernel

- Full tickless operation introduced in Linux 3.10
  - No need for a periodic tick in the system, particularly when the system is idle
  - Idle CPUs save power
- `CONFIG_HZ_PERIODIC`
  - Old-style mode where the timer tick runs at all times
- `CONFIG_NO_HZ_IDLE` (formerly `CONFIG_NO_HZ`) – default
  - Disable the tick at idle, with re-programming it for the next pending timer
- `CONFIG_NO_HZ_FULL`
  - The CPUs without a timer tick must be designated at boot time
  - At least one CPU needs to receive interrupts and do the necessary housekeeping
  - The timer tick is disabled if there is only a single runnable process on that CPU