

Jin-Soo Kim  
([jinsoo.kim@snu.ac.kr](mailto:jinsoo.kim@snu.ac.kr))

Systems Software &  
Architecture Lab.  
Seoul National University

Fall 2022

# CPU Scheduling



# CPU Scheduling

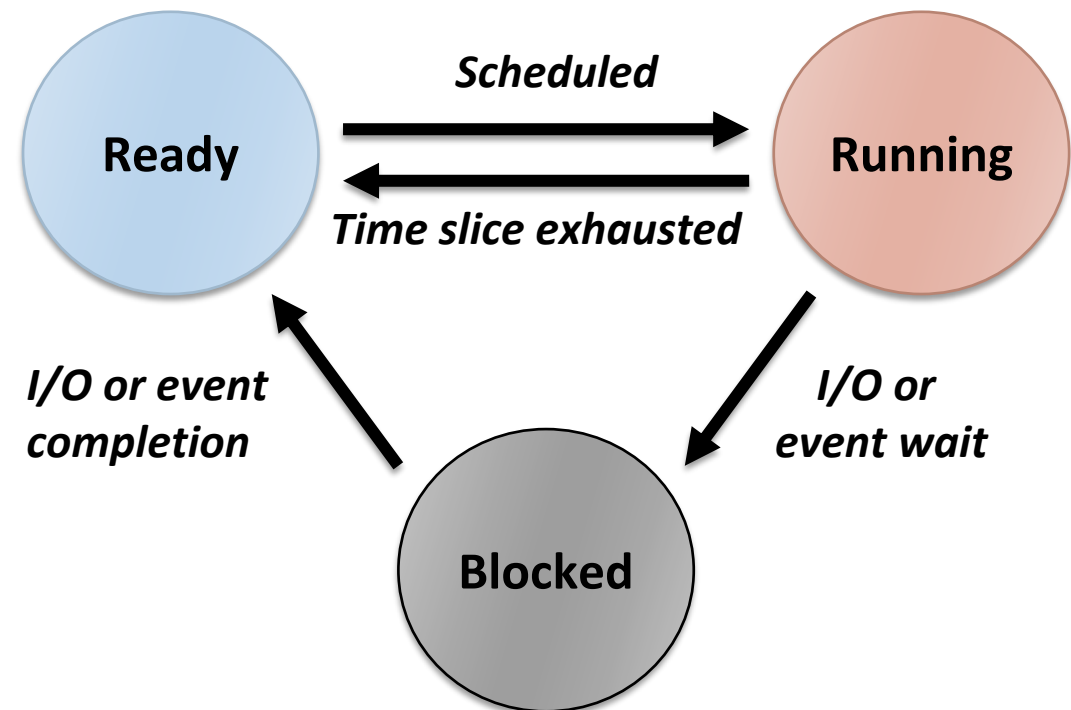
- A policy deciding which process to run next, given a set of runnable tasks (processes or threads)
  - Happens frequently, hence should be fast

- **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 process in a system call is preempted?

# Work-Conserving (or not)

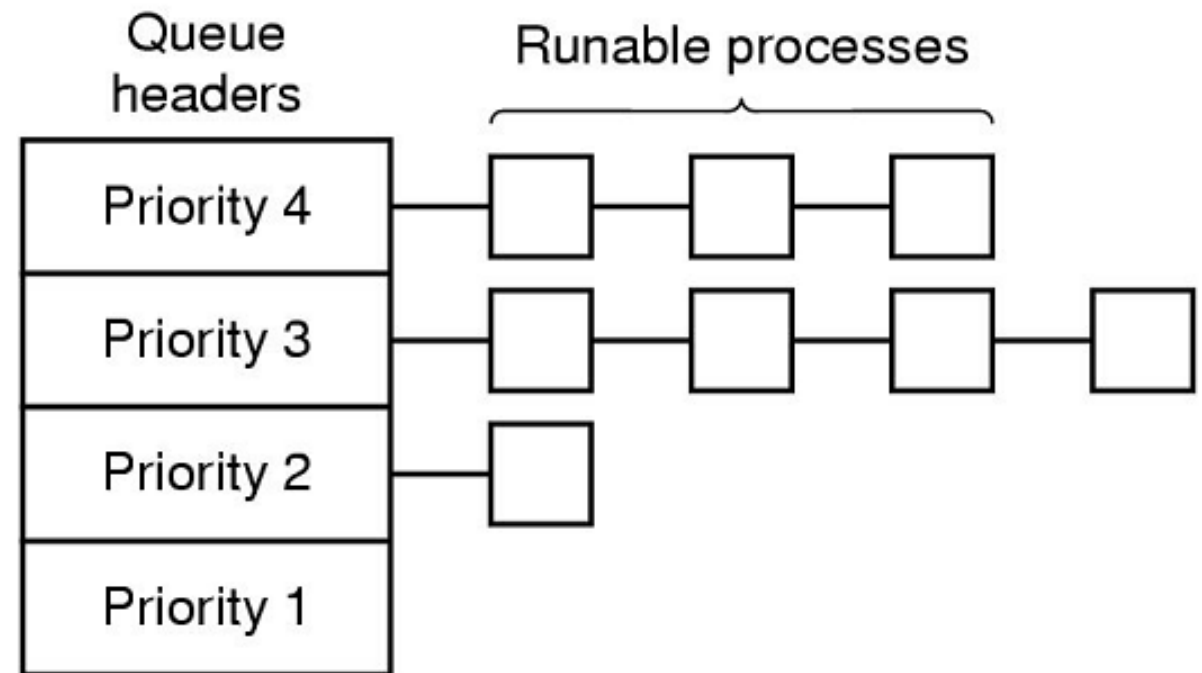
- **Work-conserving scheduler**
  - Never leave a resource idle when someone wants it
  - e.g., Linux CPU scheduler (ideally)
- **Non-work-conserving scheduler**
  - May leave the resource idle despite the presence of jobs
  - e.g., Server waits for short job before starting on a big job
  - e.g., Anticipatory 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	<ul style="list-style-type: none"><li>• Epoch-based priority scheduling</li><li>• <math>O(n)</math> scheduler</li></ul>
Linux 2.6 ~ 2.6.22	<ul style="list-style-type: none"><li>• Active / expired arrays with bitmaps</li><li>• Per-core run queue</li><li>• <math>O(1)</math> scheduler</li></ul>
Linux 2.6.23 ~	<ul style="list-style-type: none"><li>• CFS (Completely Fair Scheduler) by Ingo Molnar</li></ul>
Linux 3.14 ~	<ul style="list-style-type: none"><li>• Sporadic task model deadline scheduling (SCHED_DEADLINE)</li></ul>



# Linux Scheduling Classes

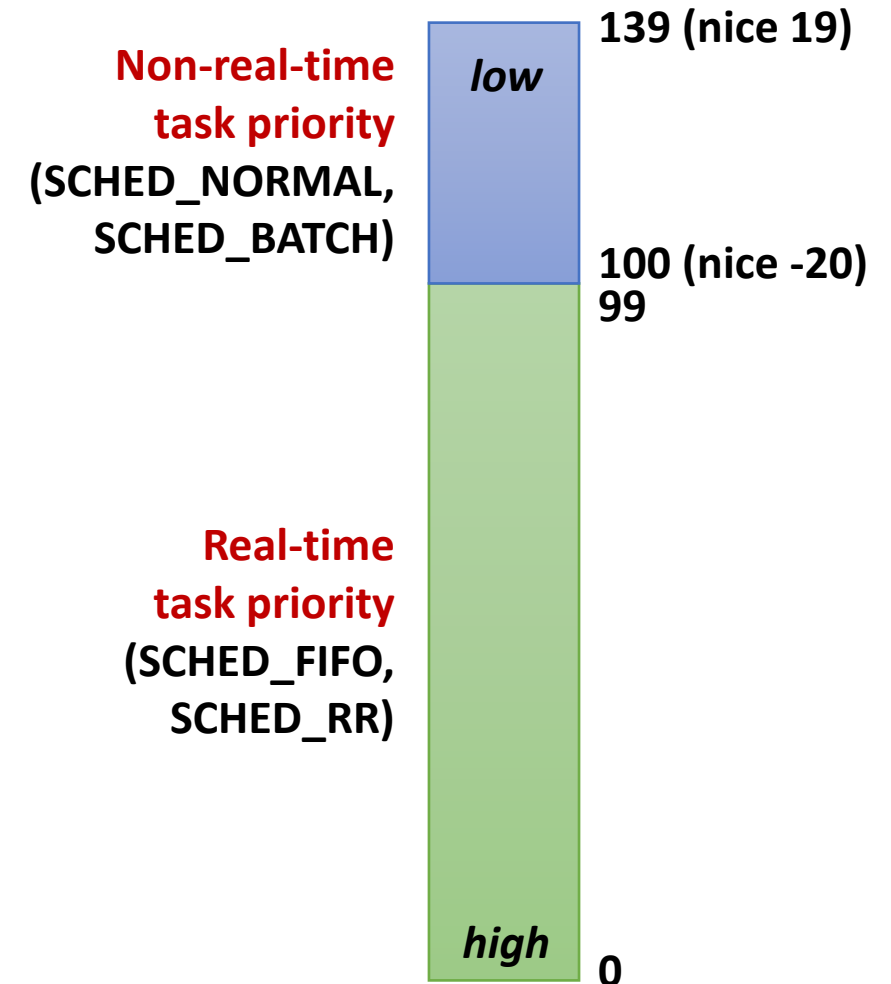
Class	Description	Policy
DL	<ul style="list-style-type: none"><li>• For real-time tasks with deadline</li><li>• Highest priority</li></ul>	SCHED_DEADLINE
RT	<ul style="list-style-type: none"><li>• For real-time tasks</li></ul>	SCHED_FIFO SCHED_RR
Fair	<ul style="list-style-type: none"><li>• For time-sharing tasks</li></ul>	SCHED_NORMAL SCHED_BATCH
Idle	<ul style="list-style-type: none"><li>• For per-CPU idle tasks</li></ul>	SCHED_IDLE

# Linux CFS

(Completely Fair Scheduler)

# Linux Task Priority

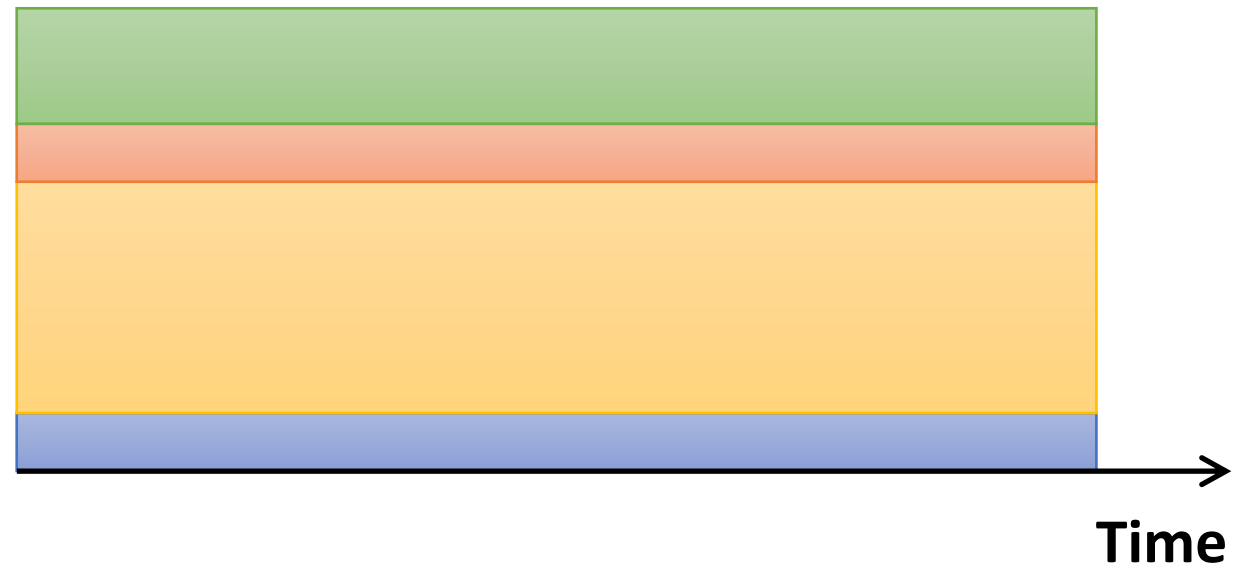
- Total 140 levels (0 ~ 139)
  - A smaller value means higher priority
- Setting priority for non-real-time tasks
  - `nice()`, `setpriority()`
  - $-20 \leq \text{nice value} \leq 19$
  - Default nice value = 0 (priority value 120)
- Setting priority for real-time tasks
  - `sched_setattr()`
  - Static 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



$$\text{Task A's share} = \frac{\text{weight}_A}{\sum \text{weight}_i} = \frac{2}{8} = 25.0\%$$

# 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
- $\text{weight}(i)/\text{weight}(i+1) = 0.55/0.45 = 1.22$  (or  $\simeq 25\%$  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

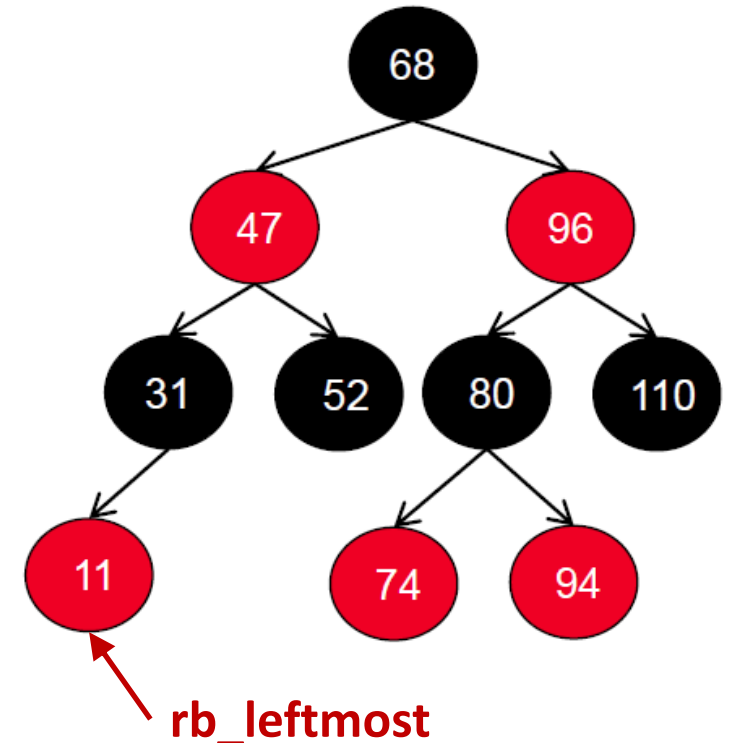
$$VR(T) = \frac{Weight_0}{Weight(T)} \times PR(T) = \left( Weight_0 \times \frac{2^{32}}{Weight(T)} \times PR(T) \right) \gg 32$$

*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

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

**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_RESCHEDED` 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, `schedu1e()` is called when `NEED_RESCHEDED` flag is set
  - Clear the `NEED_RESCHEDED` 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:

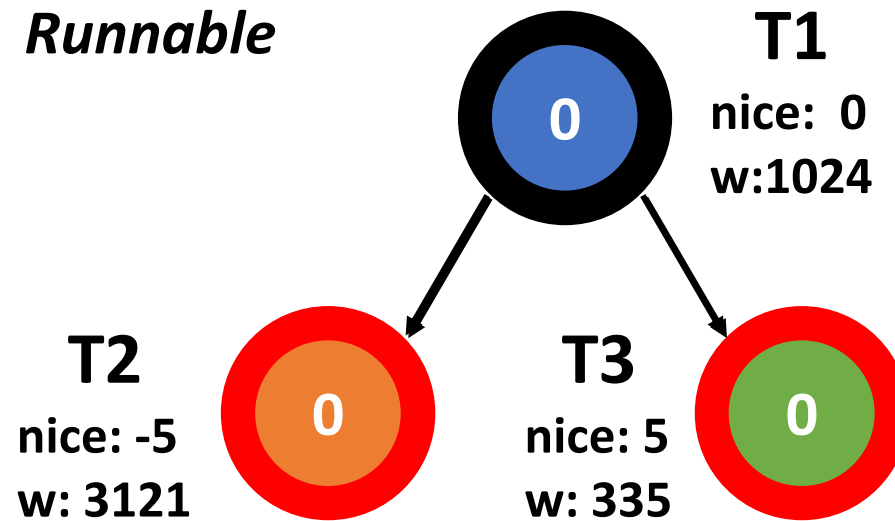
- Initially choose the leftmost task, T2, in this case
- But how long?

$TS(T2)$

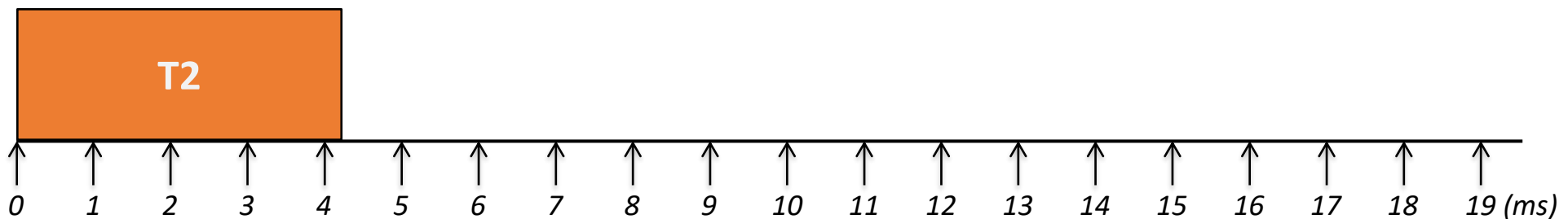
$$= \frac{3121}{1024 + 3121 + 335} \times P$$

$$= 4.18 \text{ ms}$$

*Runnable*



*Blocked*



# Example:

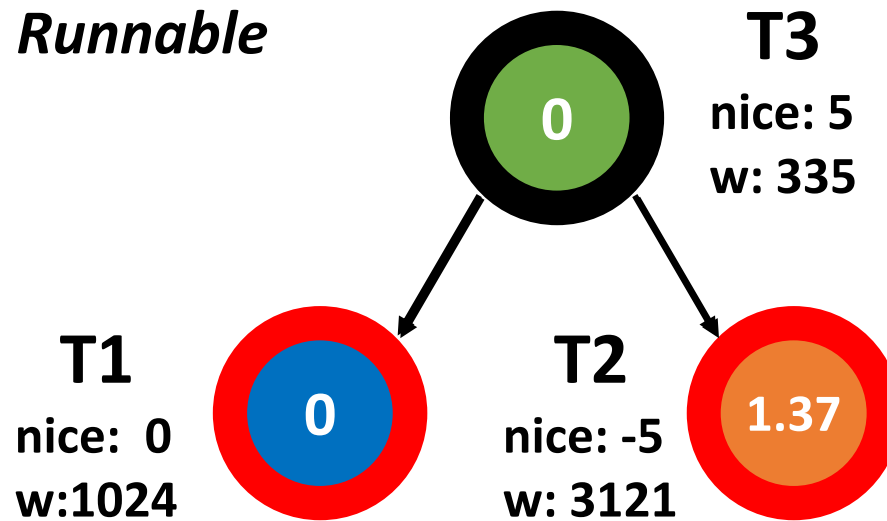
- Update T2's vruntime

$VR(T2)$

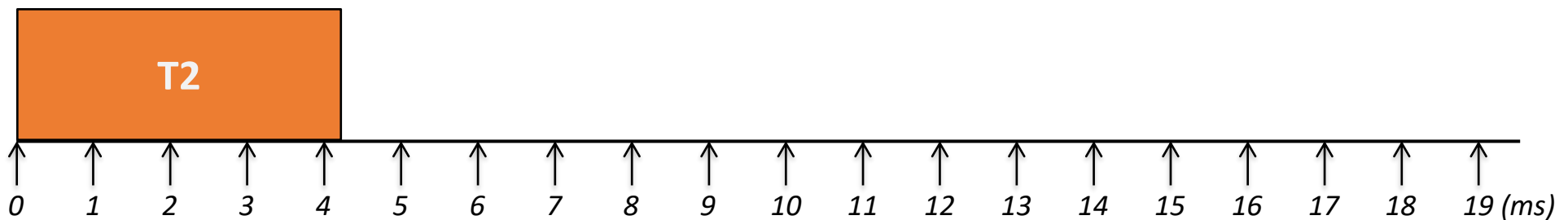
$$= \frac{1024}{3121} \times 4.18$$

$$= 1.37$$

*Runnable*



*Blocked*



# Example:

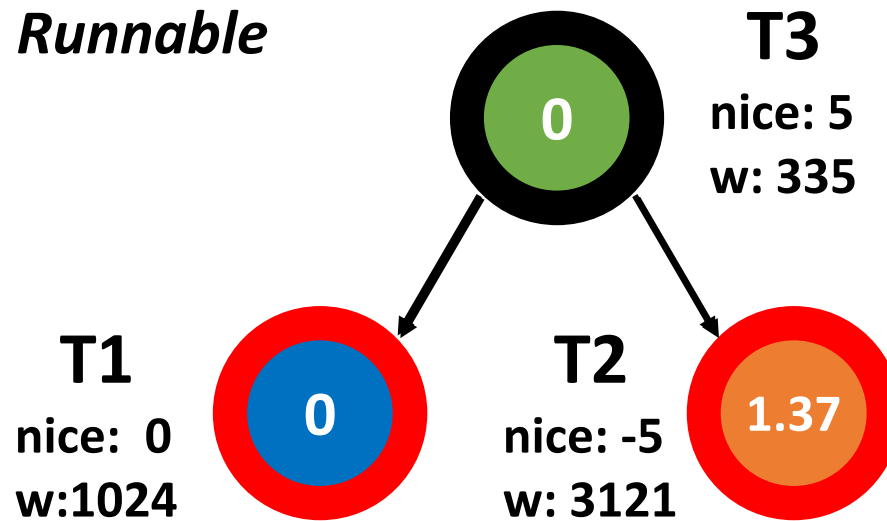
- Now choose T1

$TS(T1)$

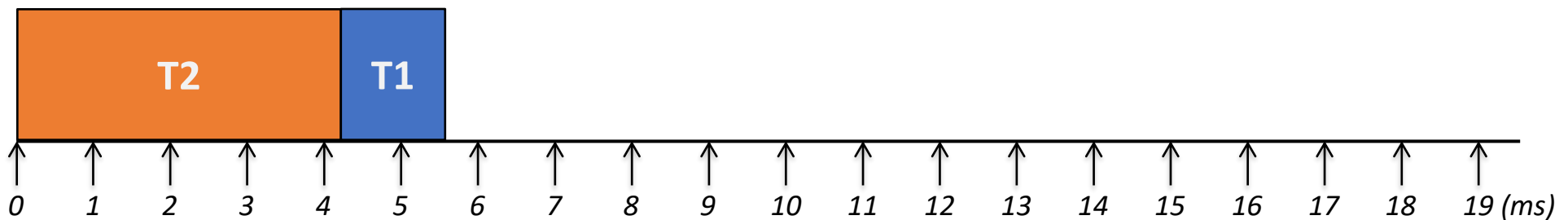
$$= \frac{1024}{1024 + 3121 + 335} \times P$$

$$= 1.37 \text{ ms}$$

*Runnable*



*Blocked*



# Example:

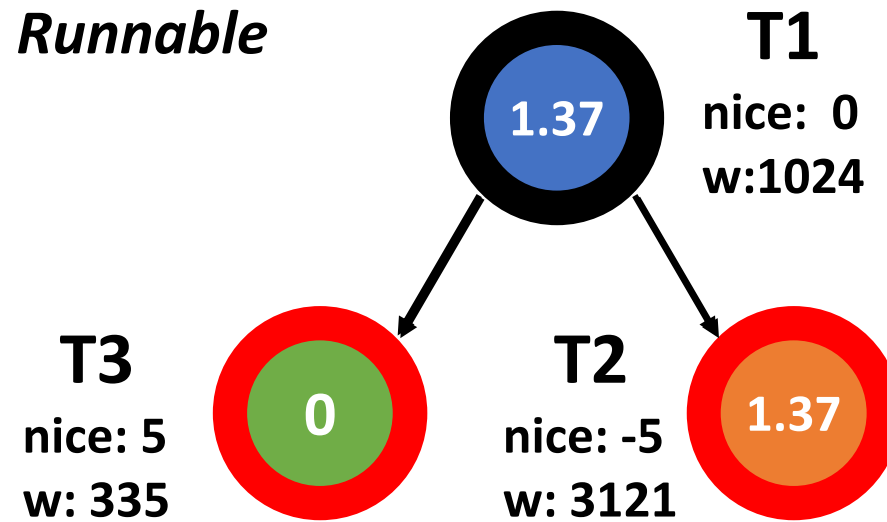
- Update T1's runtime

$VR(T1)$

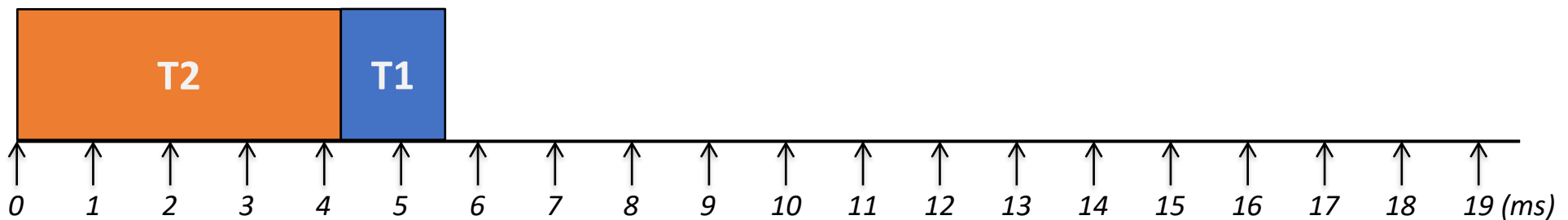
$$= \frac{1024}{1024} \times 1.37$$

$$= 1.37$$

*Runnable*



*Blocked*



# Example:

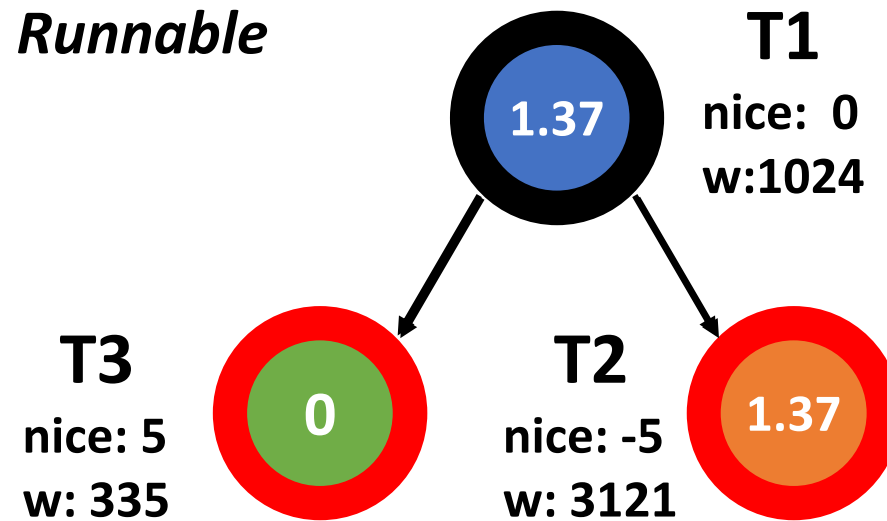
- Choose T3

$TS(T3)$

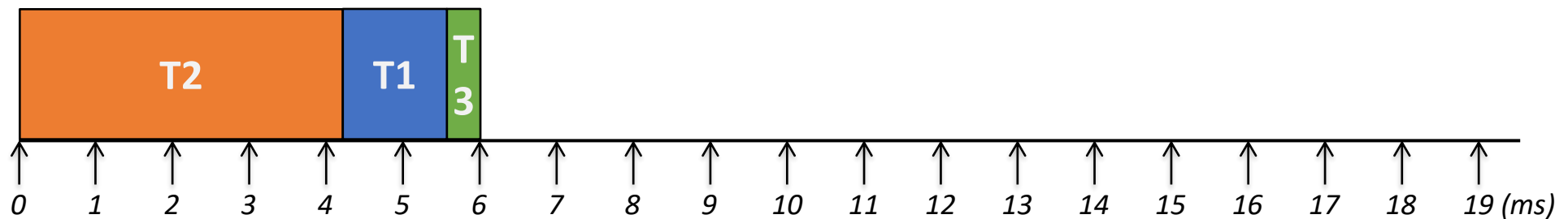
$$= \frac{335}{1024 + 3121 + 335} \times P$$

$$= 0.45 \text{ ms}$$

*Runnable*



*Blocked*



# Example:

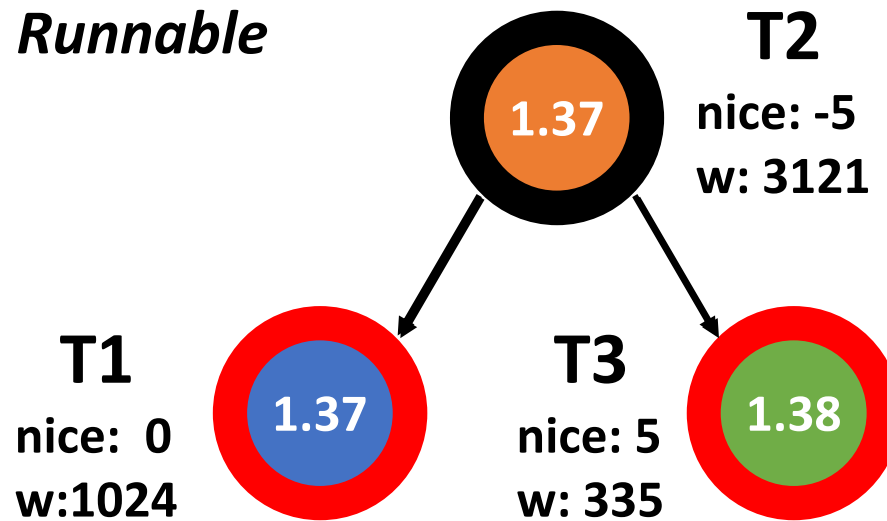
- Update T3's vruntime

$VR(T3)$

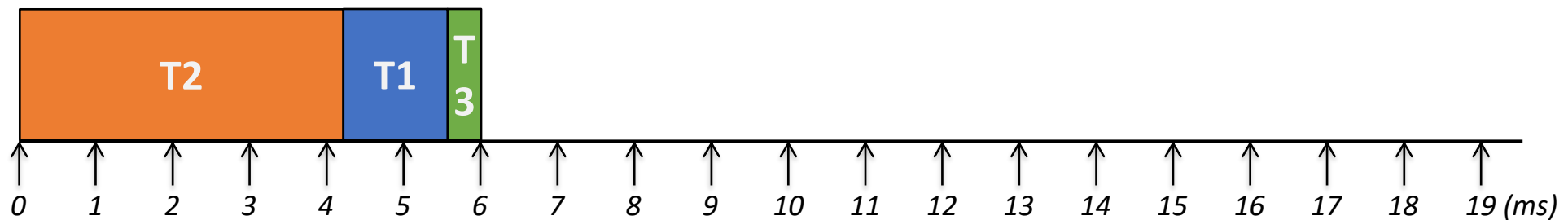
$$= \frac{1024}{335} \times 0.45$$

$$= 1.38$$

*Runnable*



*Blocked*



# Example:

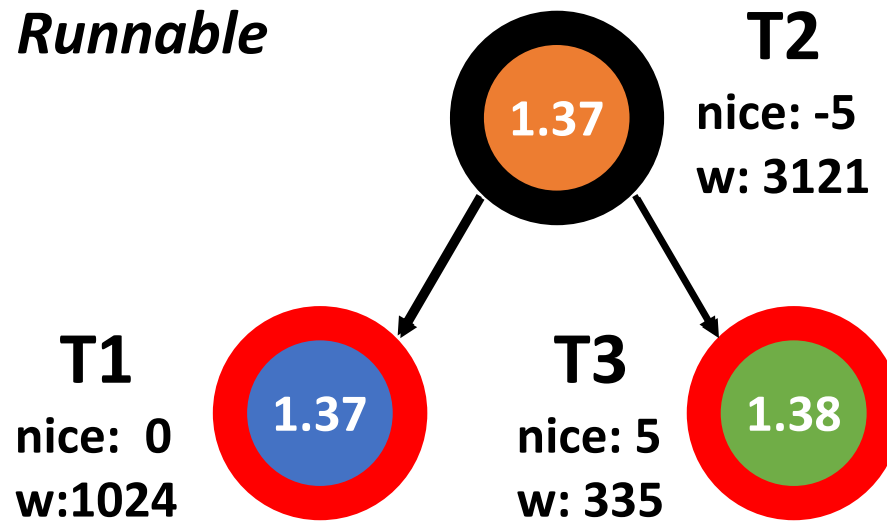
- Choose T1

$TS(T1)$

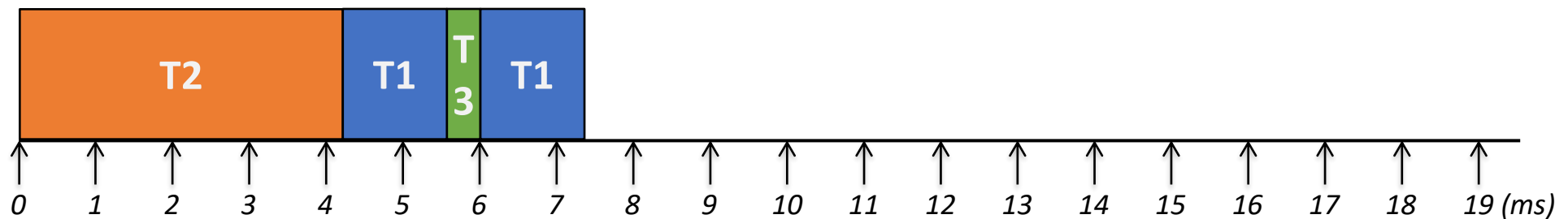
$$= \frac{1024}{1024 + 3121 + 335} \times P$$

$$= 1.37 \text{ ms}$$

*Runnable*



*Blocked*





# Example:

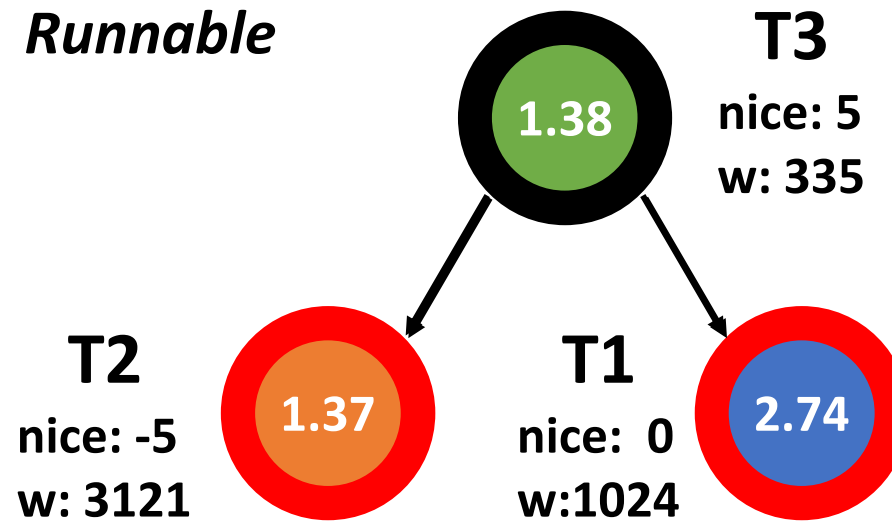
- Update T1's vruntime

$VR(T1)$

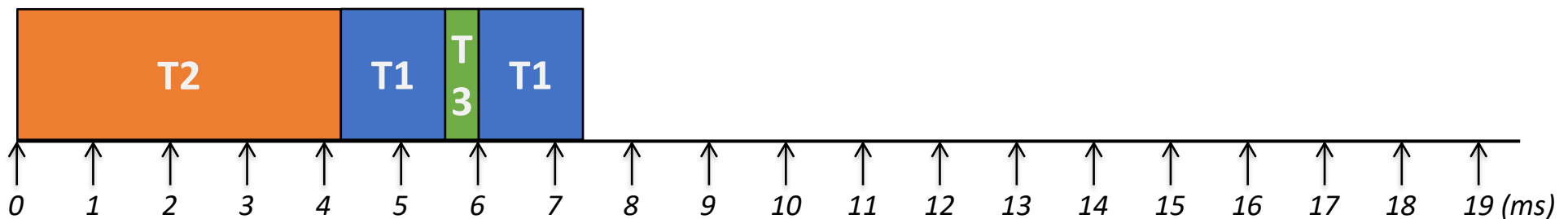
$$= 1.37 + \frac{1024}{1024} \times 1.37$$

$$= 2.74$$

*Runnable*

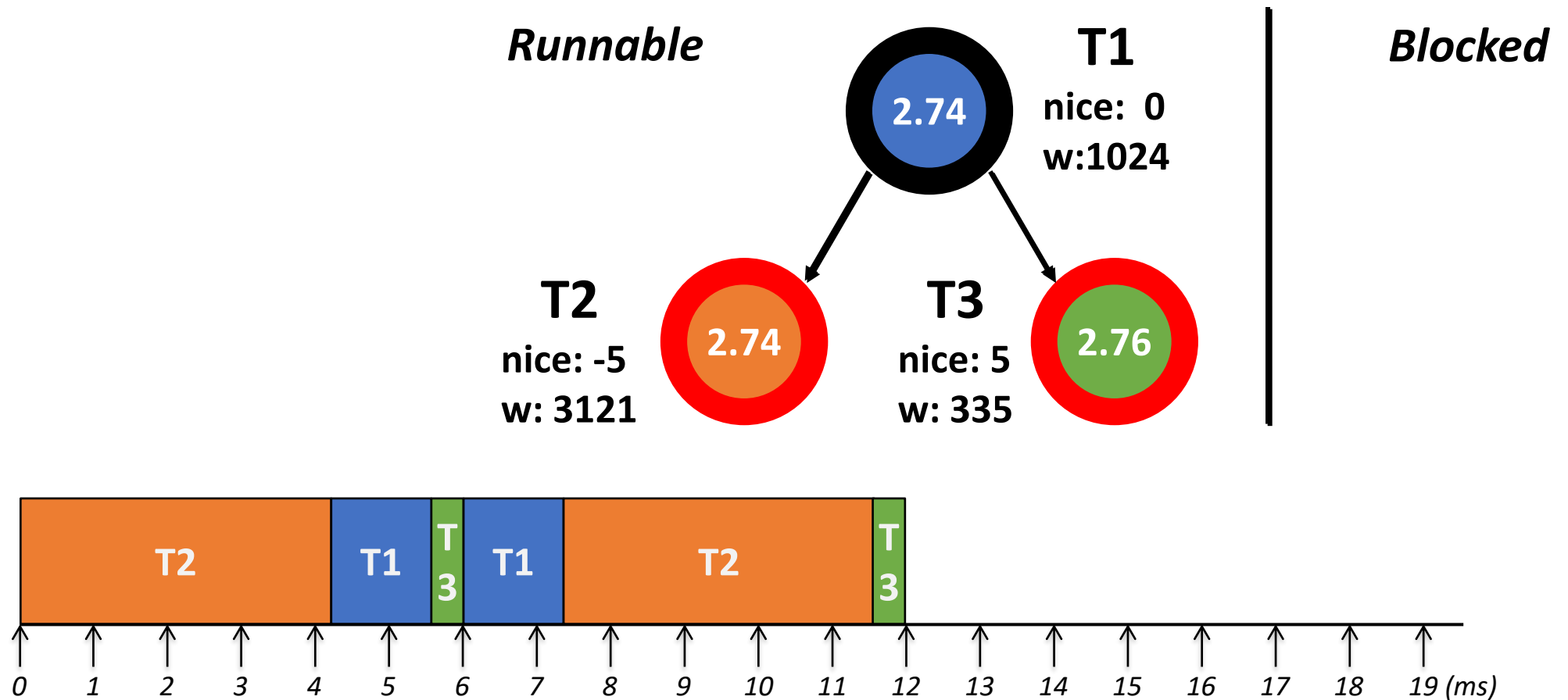


*Blocked*



# Example:

- Update T2 for 4.18ms and T3 for 0.45ms



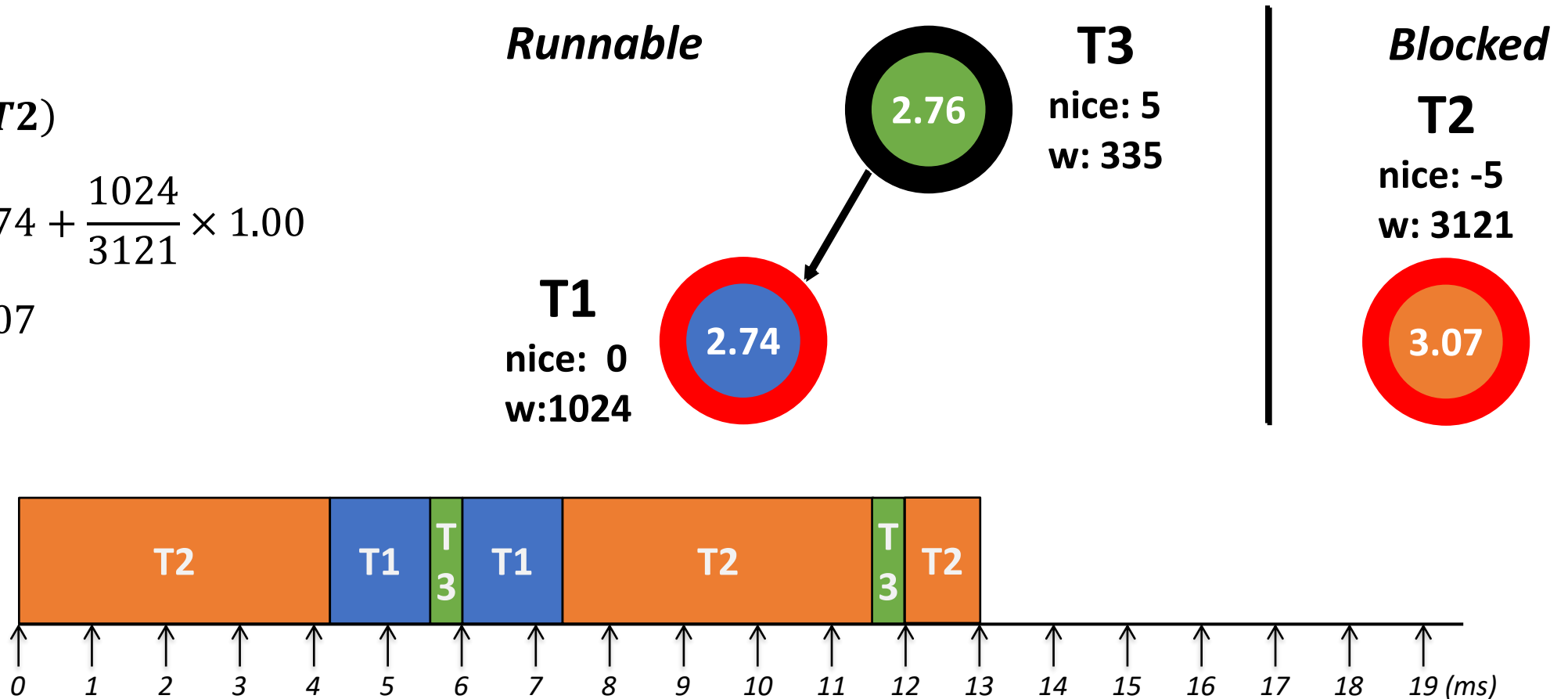
# Example:

- 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

$TS(T1)$

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

$$= 4.52 \text{ ms}$$

$VR(T1)$

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

$$= 7.26$$

**Runnable**

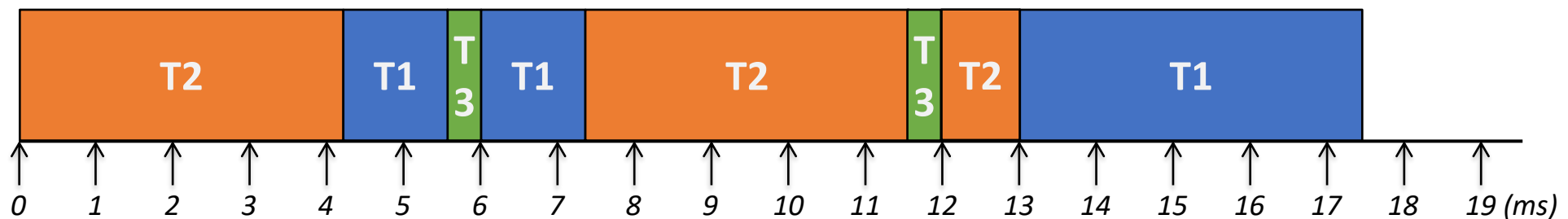
**T3**  
nice: 5  
w: 335



**T1**  
nice: 0  
w: 1024

**Blocked**

**T2**  
nice: -5  
w: 3121



# Example:

- T3 runs

$TS(T3)$

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

$$= 1.48 \text{ ms}$$

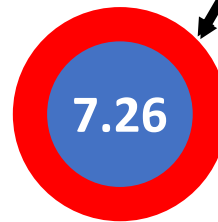
$VR(T3)$

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

$$= 7.28$$

**Runnable**

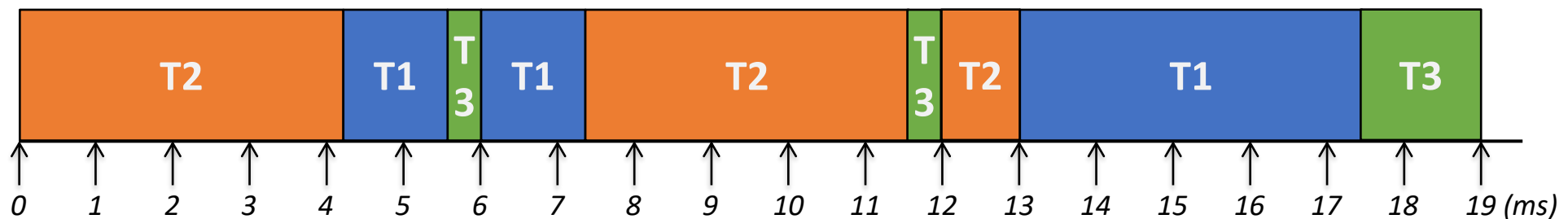
**T1**  
nice: 0  
w:1024



**T3**  
nice: 5  
w: 335

**Blocked**

**T2**  
nice: -5  
w: 3121



# 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