Kernel Data Structures

Completely Fair Scheduling

Department of Computer Science and Engineering

INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

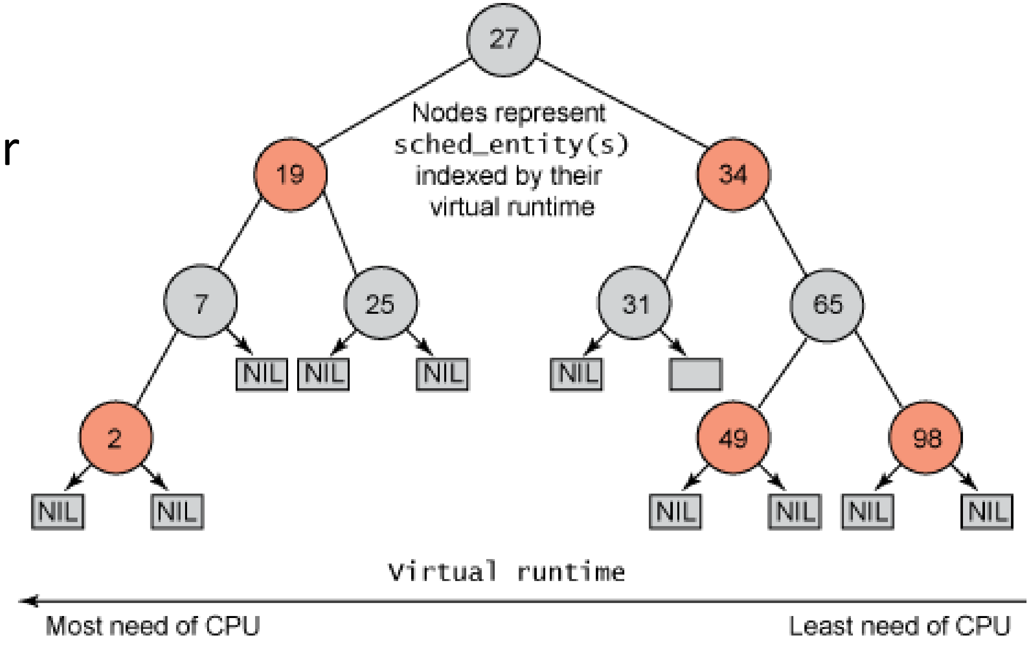
Sandip Chakraborty
sandipc@cse.iitkgp.ac.in

Completely Fair Scheduling (CFS)

References:

1. "Professional Linux Kernel Architecture" by Wolfgang Mauerer (*Chapter 2: Process Management and Scheduling*)

2. "Linux Kernel Development" by Robert Love (*Chapter 4: Process Scheduling*)



Interactive vs Batch Processes

- **Interactive Processes (I/O Bound)**: Needs frequent scheduling but the timeslice duration can be less
  - Example: A text editor – waits for the input from the user, but expects the input to be processed immediately when available

Interactive vs Batch Processes

- **Interactive Processes (I/O Bound)**: Needs frequent scheduling but the timeslice duration can be less
  - Example: A text editor – waits for the input from the user, but expects the input to be processed immediately when available

- **Batch Processes (CPU Bound)**: Needs longer timeslice to complete the task, but may wait for getting scheduled
  - Example: Video encoding – needs a lot of CPU, but runs in the background – does not have a strong deadline, user does not feel much bad if delayed for 0.5 sec.

Issues with the Fixed Timeslice

- Interactive processes are given higher priorities
  - More timeslice to run without getting preempted

Issues with the Fixed Timeslice

- Interactive processes are given higher priorities
  - More timeslice to run without getting preempted

```
Prio < 120
T = (140-Prio)*20

Prio ≥ 120
T = (140-Prio)*5
```

Issues with the Fixed Timeslice

- Interactive processes are given higher priorities
  - More timeslice to run without getting preempted

```
Prio < 120
T = (140-Prio)*20

Prio ≥ 120
T = (140-Prio)*5
```

Priority	Niceness	static_prio	Timeslice
Highest	-20	100	800ms
High	-10	110	600ms
Normal	0	120	100ms
Low	+10	130	50ms
Lowest	+19	139	5ms

Issues with the Fixed Timeslice

- Interactive processes are given higher priorities
  - More timeslice to run without getting preempted

```
Prio < 120
T = (140-Prio)*20

Prio ≥ 120
T = (140-Prio)*5
```

Priority	Niceness	static_prio	Timeslice
Highest	-20	100	800ms
High	-10	110	600ms
Normal	0	120	100ms
Low	+10	130	50ms
Lowest	+19	139	5ms

Issues with the Fixed Timeslice

- Interactive processes are given higher priorities
  - More timeslice to run without getting preempted

```
Prio < 120
T = (140-Prio)*20

Prio ≥ 120
T = (140-Prio)*5
```

Priority	Niceness	static_prio	Timeslice
Highest	-20	100	800ms
High	-10	110	600ms
Normal	0	120	100ms
Low	+10	130	50ms
Lowest	+19	139	5ms

Disproportionate allocation of timeslices, significantly affect the performance of the batch processes

Issues with the Fixed Timeslice

- Interactive processes are given higher priorities
  - More timeslice to run without getting preempted

Frequent context switching when there are two low priority processes

```
Prio < 120
T = (140-Prio)*20

Prio ≥ 120
T = (140-Prio)*5
```

Priority			lice
Highest			ns
High	-10	110	600ms
Normal	0	120	100ms
Low	+10	130	50ms
Lowest	+19	139	5ms

Disproportionate allocation of timeslices, significantly affect the performance of the batch processes

Setting an Ideal Timeslice: A Non-trivial Problem

- Large timeslices affect interactivity – two high priority processes in the system -> interactivity will get affected

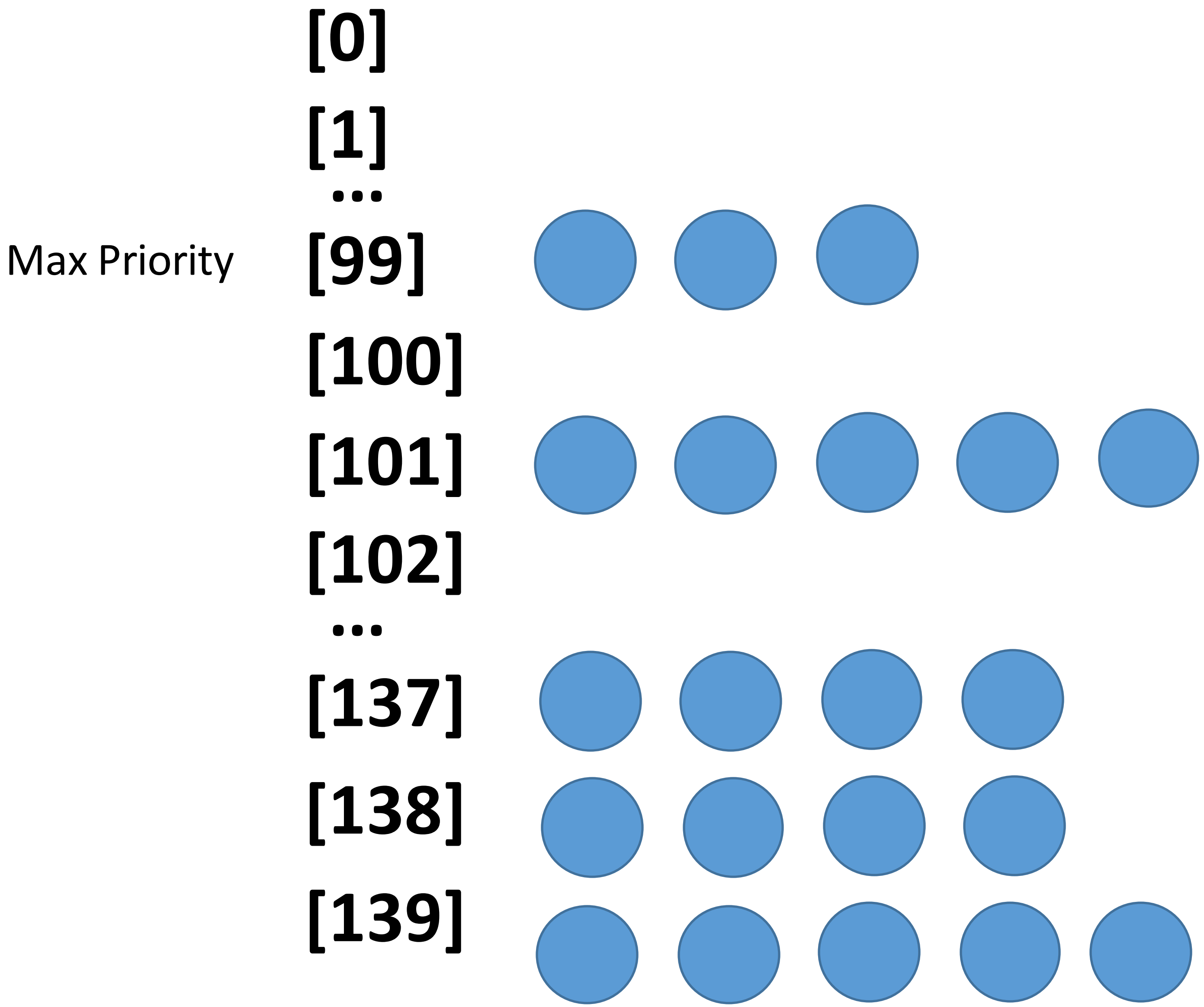
- Small timeslices cause frequent context switching – not desirable

Why a New Scheduler?

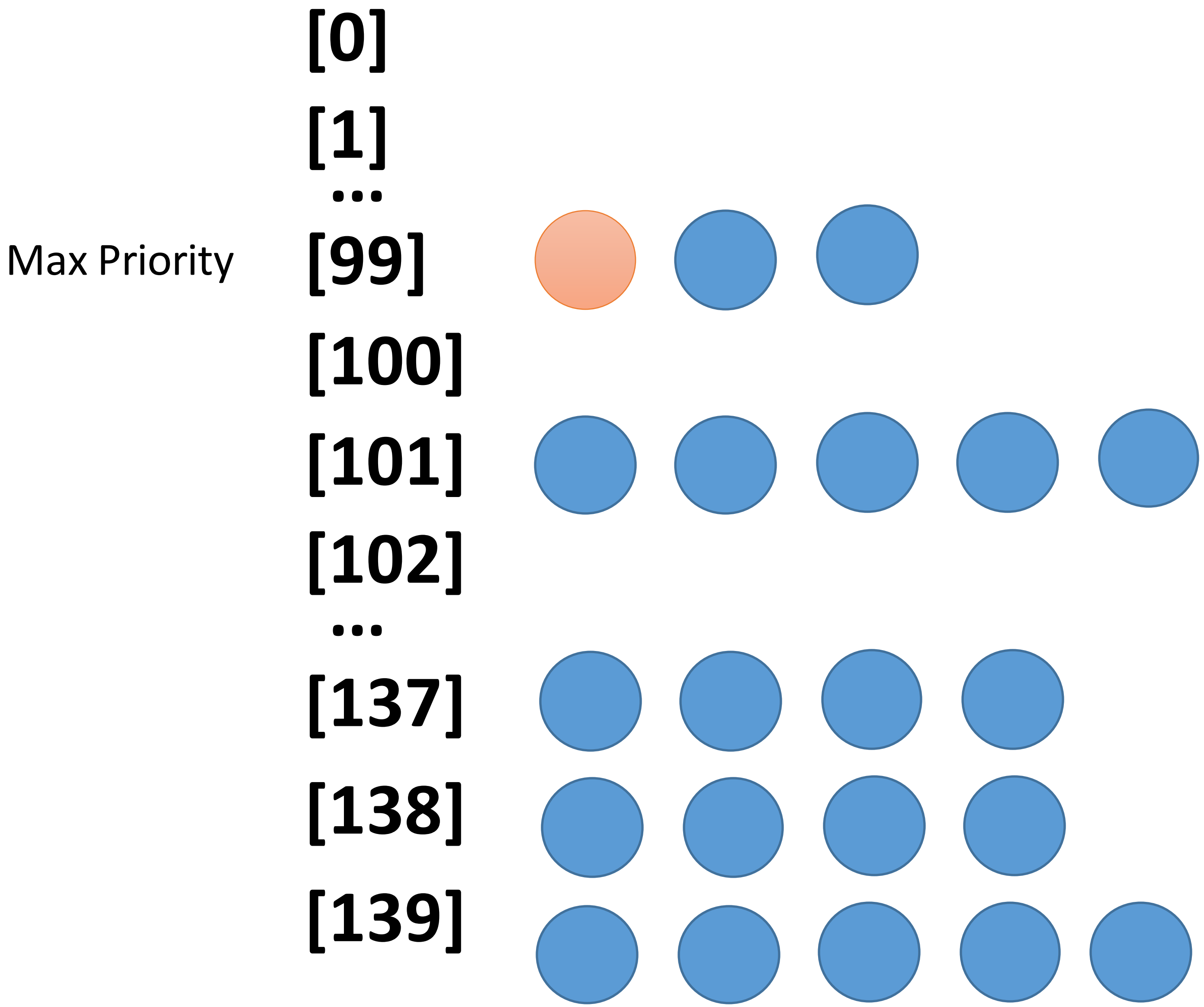
- O(1) scheduler failed to demonstrate its promises in practice
  - Implementation is very complicated – difficult to debug
  - The heuristic for interactivity measures did not work – resulting in poor performance in practice
  - Failure of an O(1) algorithm in real environment !

- Various other patches have been applied on the Kernel scheduler
  - **Staircase Scheduler (2004, Kolivas):** Heuristic for interactivity replaced by a rank-based scheme – runqueue as a ranked array
    - Remove the concept of "expired array"
    - The expired process will fall one priority staired down, and will be added back to the same runqueue
    - Once reached at the bottom for the first time, stair up one priority level below the maximum
    - Once reached at the bottom for the second time, stair up two priority levels below the maximum

Staircase Scheduling



Staircase Scheduling



Staircase Scheduling

[0]

[1]

...

Max Priority [99]

[100]

[101]

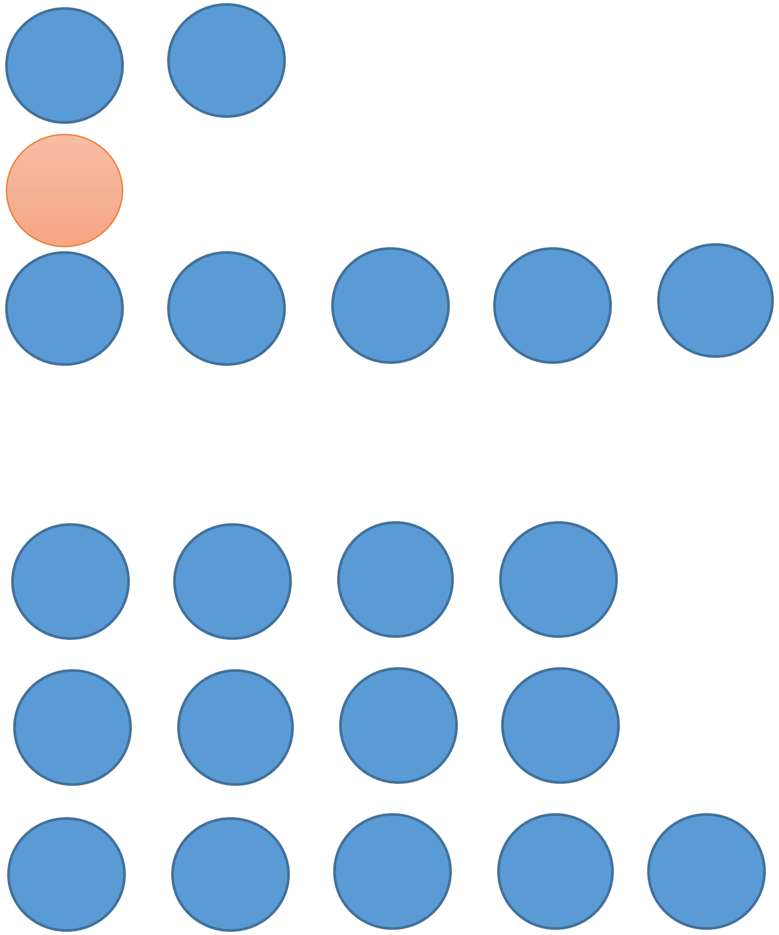
[102]

...

[137]

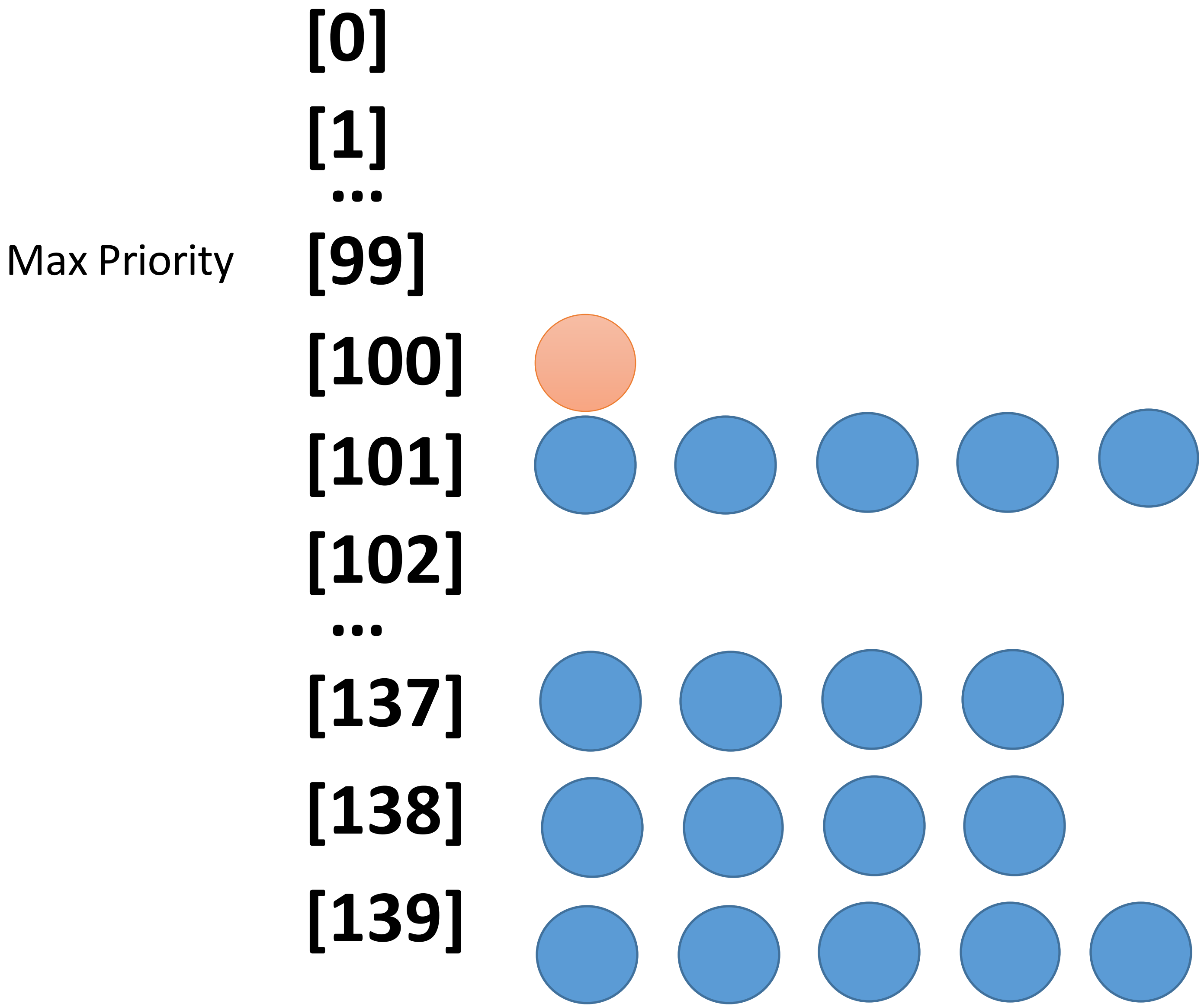
[138]

[139]



Stair down for one level

Staircase Scheduling



Stair down for one level

Staircase Scheduling

[0]

[1]

...

Max Priority [99]

[100]

[101]

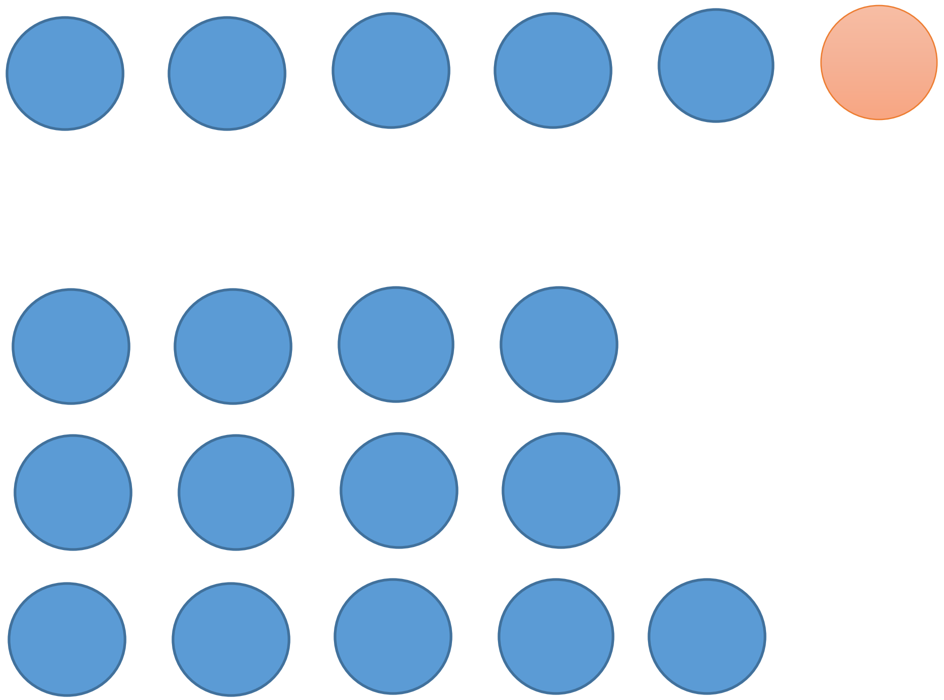
[102]

...

[137]

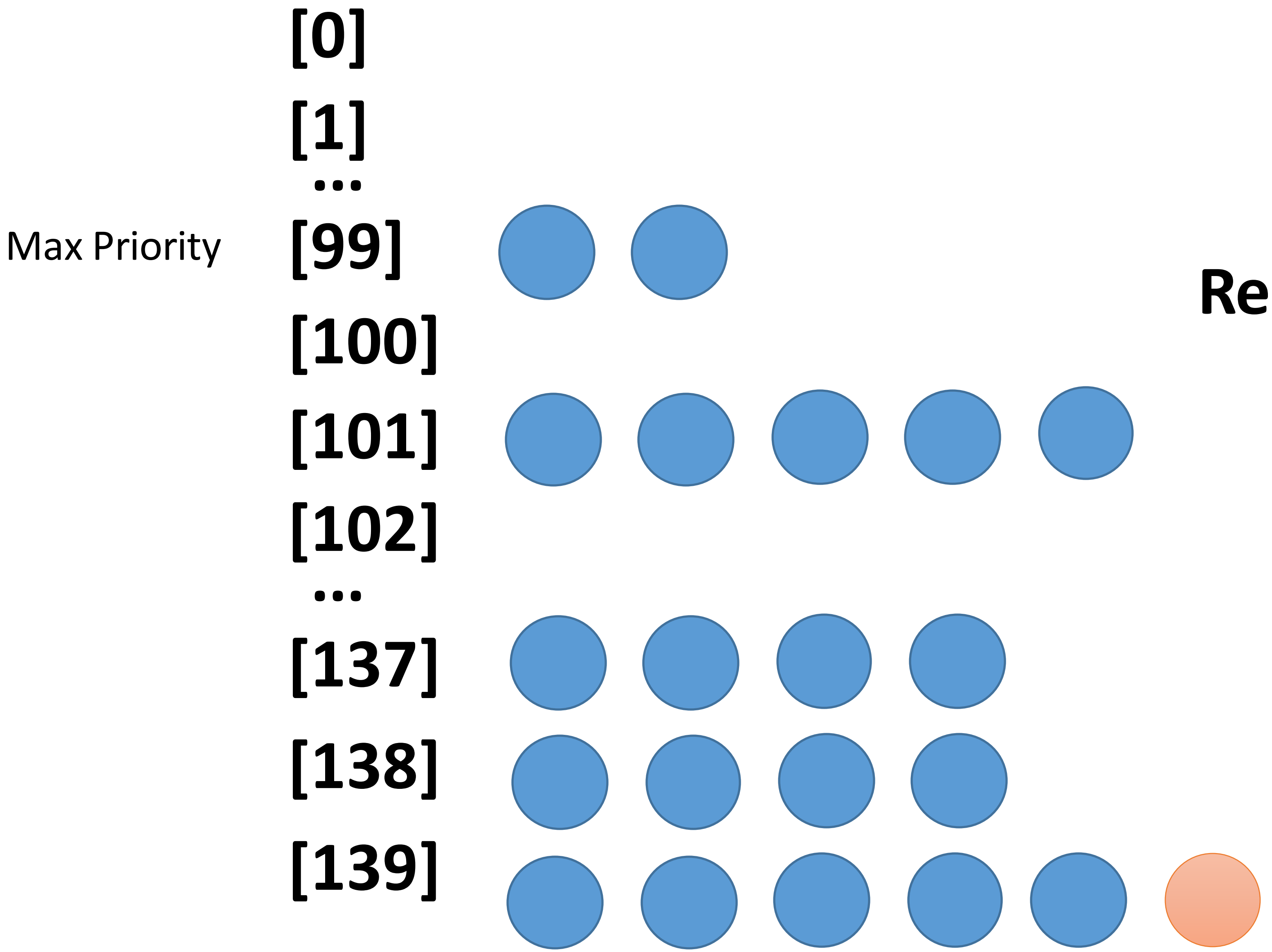
[138]

[139]



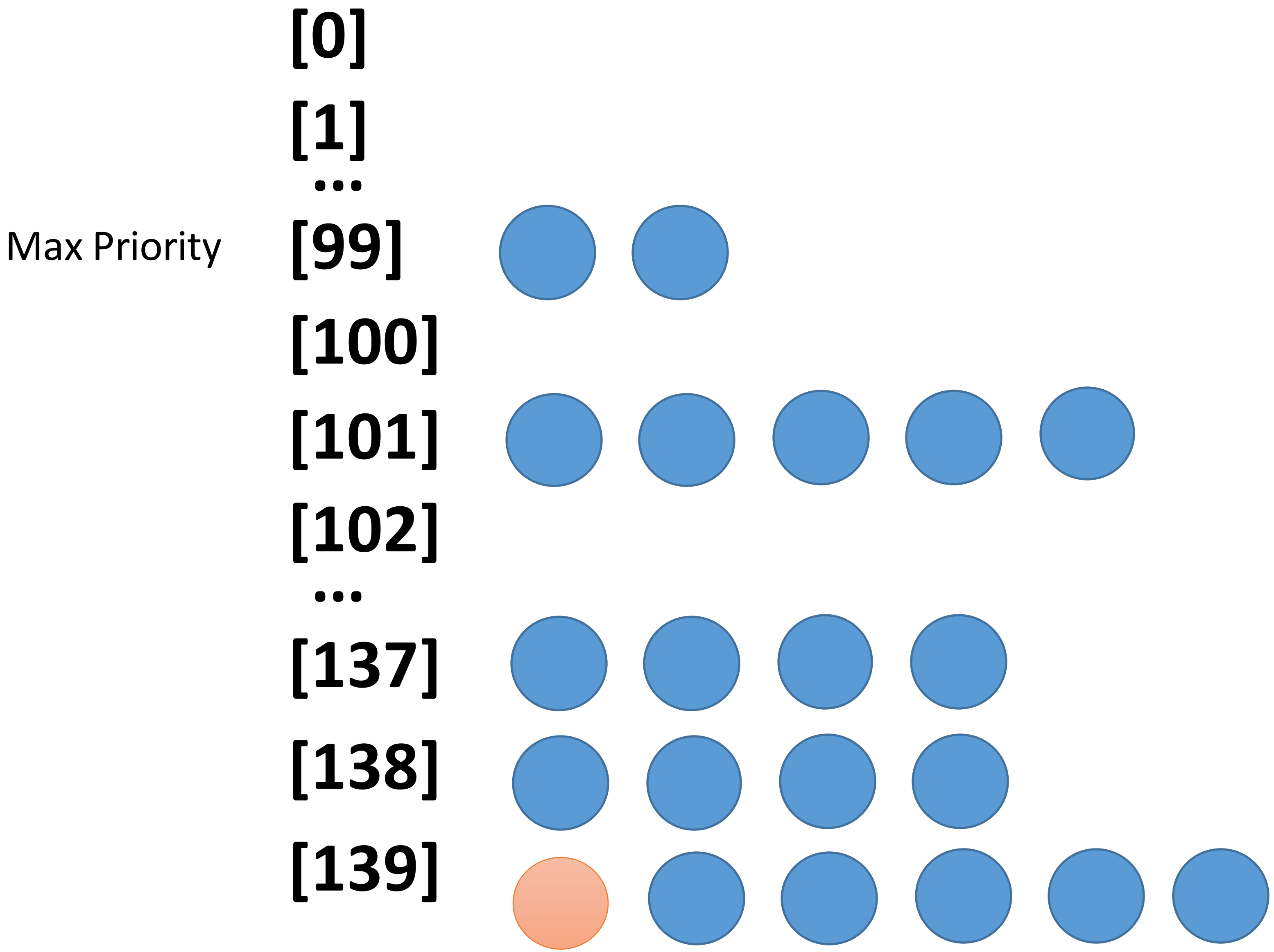
Stair down for one level further

Staircase Scheduling

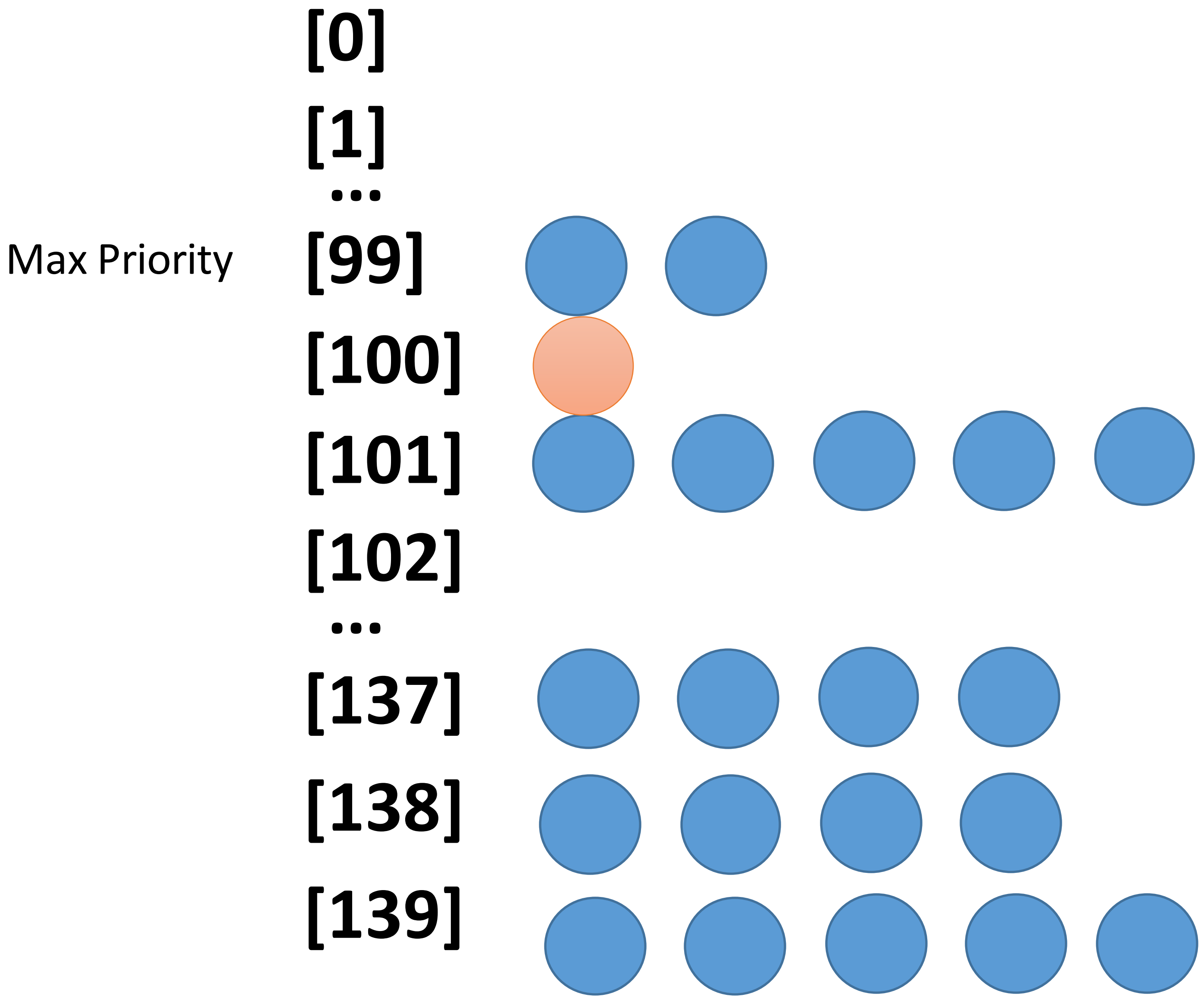


Reach to the bottom

Staircase Scheduling

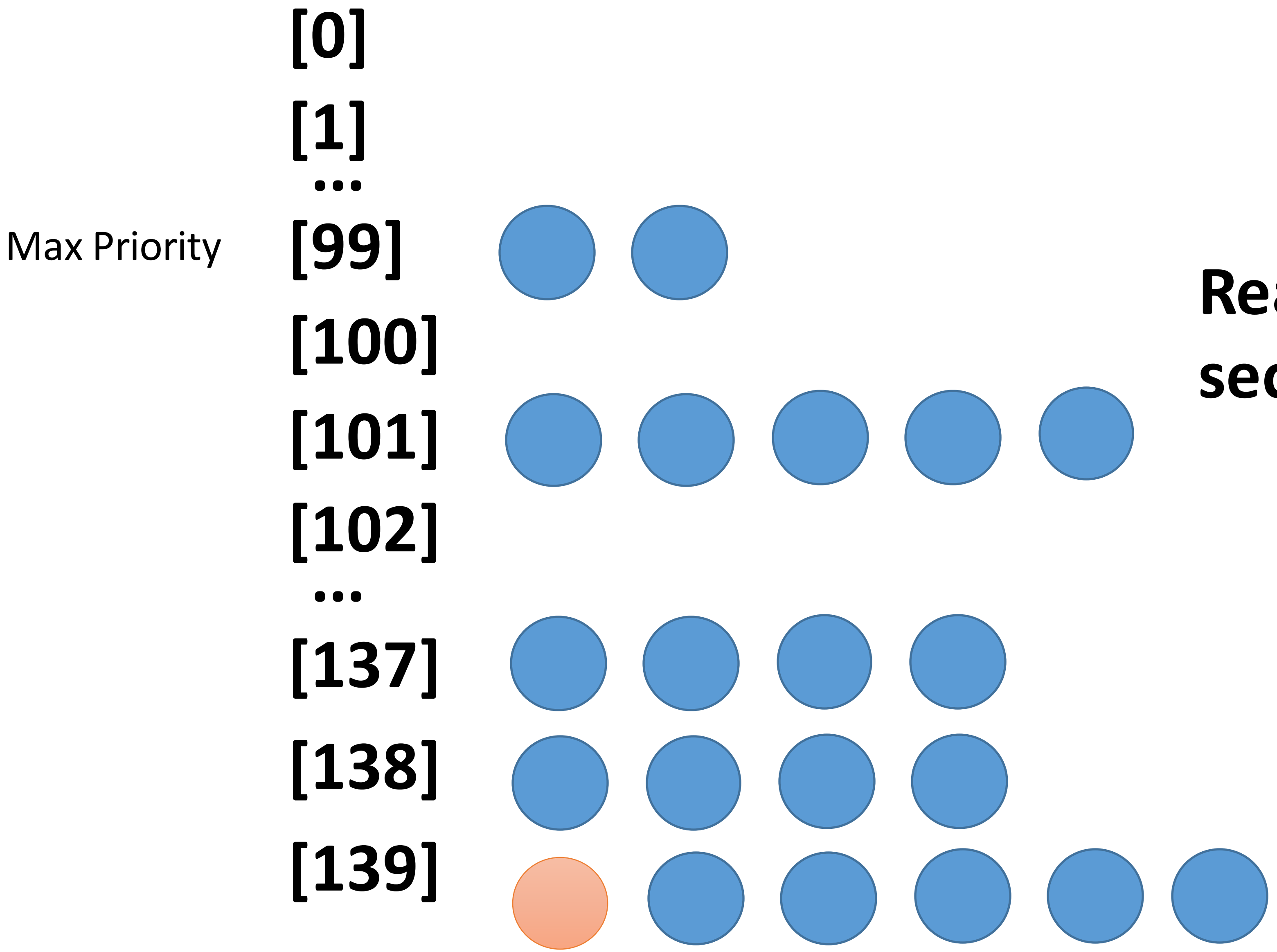


Staircase Scheduling



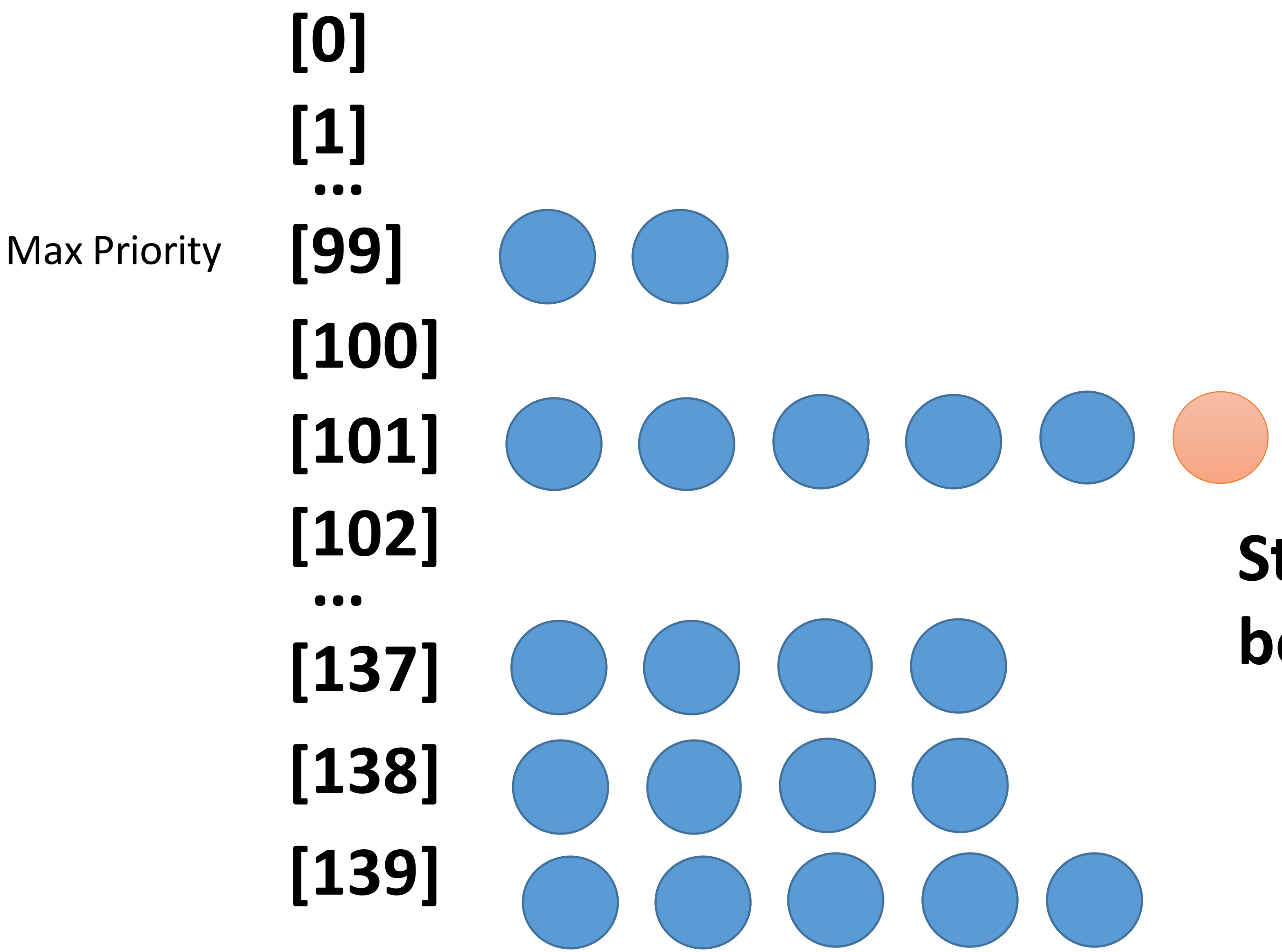
Stair up one level below the maximum

Staircase Scheduling



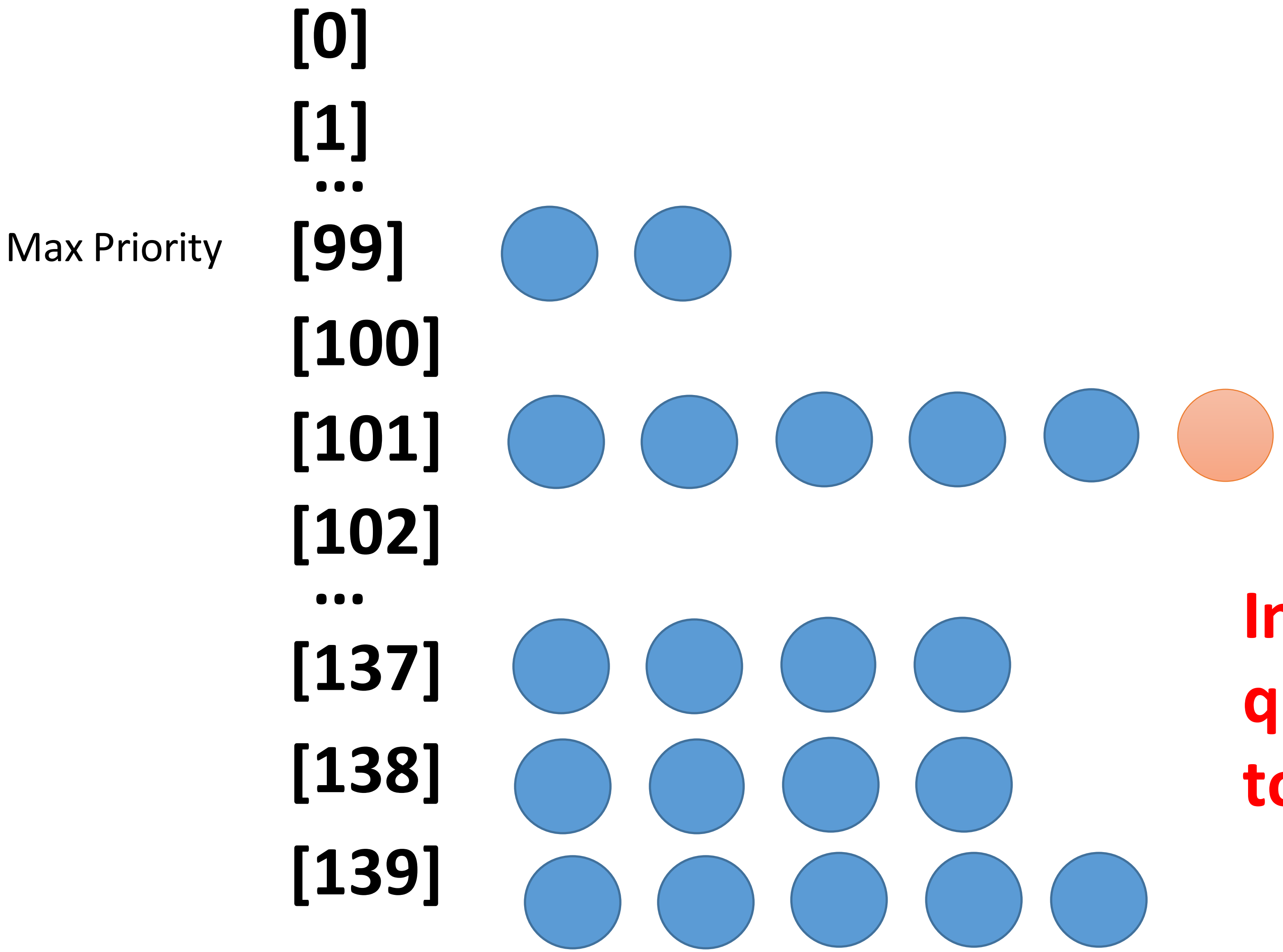
Reach bottom for the second time

Staircase Scheduling



Stair up two levels below the maximum

Staircase Scheduling



Interactive processes quickly move to the top of the queue

Updating the Scheduler Further

- **Rotating Staircase Deadline Scheduler** (RSDS) -- Kolivas, 2007
  - I/O bound processes can get starved in Staircase Scheduler – Interactive processes quickly jump up and eat all the time slices
  - Bring back the "Expired Array" !!
  - Rotate Staircase scheduling between the active and the expired arrays

- Neither Staircase Scheduler nor RSDS were accepted into the Kernel mainline
  - Too much desktop-oriented !! **Remember the use of Linux in early days**

Updating the Scheduler Further

- **Rotating Staircase Deadline Scheduler** (RSDS) -- Kolivas, 2007
  - I/O bound processes can get starved in Staircase Scheduler – Interactive processes quickly jump up and eat all the time slices
  - Bring back the "Expired Array" !!
  - Rotate Staircase scheduling between the active and the expired arrays

- Neither Staircase Scheduler nor RSDS were accepted into the Kernel mainline
  - Too much desktop-oriented !! **Remember the use of Linux in early days**

- **Completely Fair Scheduler (CFS)**
  - Ingo Molner (developer of O(1) Scheduler), 2007
  - Utilized some of the ideas of fairness from staircase scheduler
  - Merged into Kernel 2.6.23 -- default Linux scheduler since then

Completely Fair Scheduler

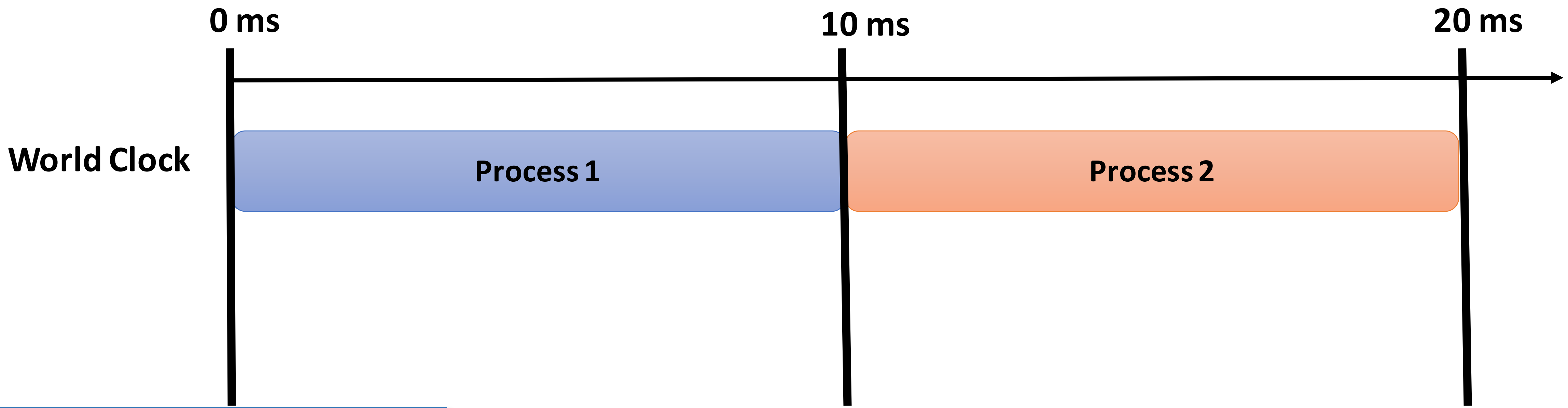
- Models an *"ideal, precise multitasking CPU"* on real hardware

Completely Fair Scheduler

- Models an *"ideal, precise multitasking CPU"* on real hardware
  - With *n* running processes, each process would be having *1/n* amount of CPU time, **when running constantly**

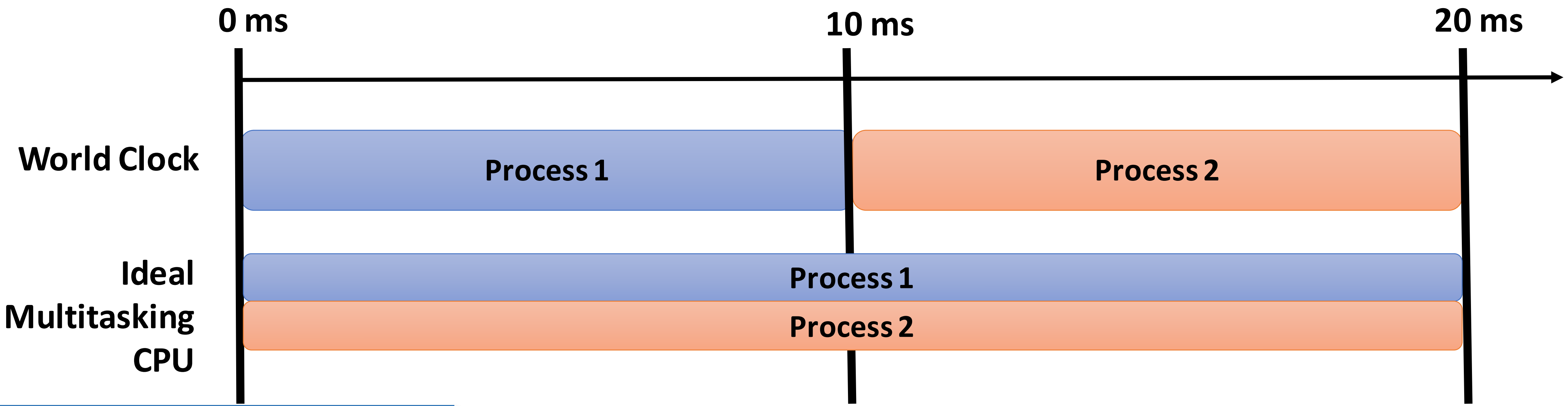
Completely Fair Scheduler

- Models an *"ideal, precise multitasking CPU"* on real hardware
  - With *n* running processes, each process would be having *1/n* amount of CPU time, **when running constantly**
  - Consider two processes – each got executed for 10 ms with 100% CPU utilization



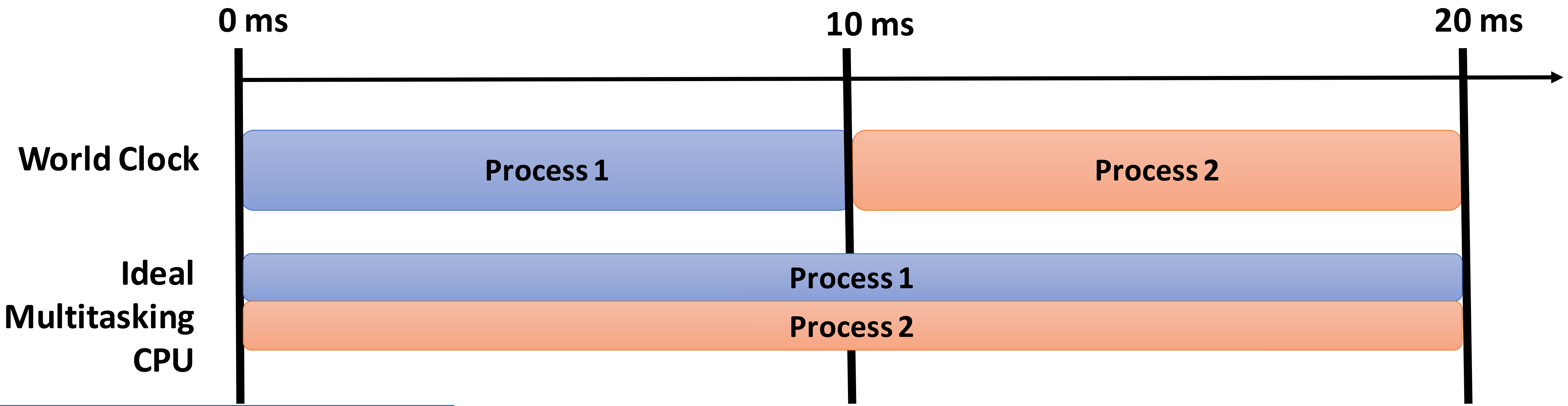
Completely Fair Scheduler

- Models an *"ideal, precise multitasking CPU"* on real hardware
  - With *n* running processes, each process would be having *1/n* amount of CPU time, **when running constantly**
  - Consider two processes – each got executed for 10 ms with 100% CPU utilization
  - **Ideal Multitasking CPU –** each will get executed for 20 ms with 50% CPU utilization



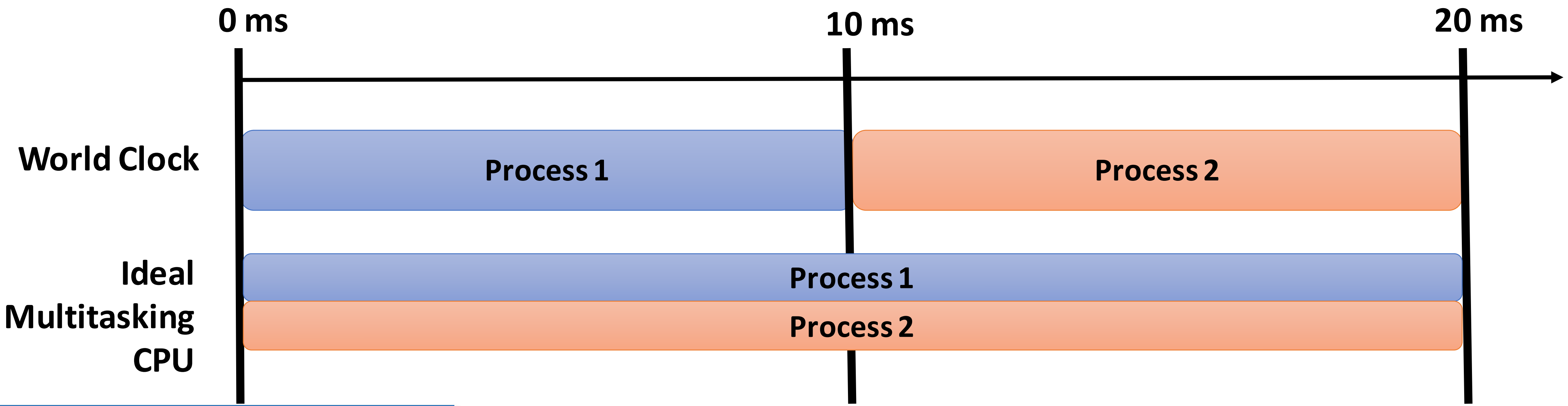
Completely Fair Scheduler

- But, the idea is not practically possible
  - You cannot run two processes on a single CPU !



Completely Fair Scheduler

- But, the idea is not practically possible
  - You cannot run two processes on a single CPU !
- CFS tries to mimic perfectly fair scheduling



Core Idea

- Calculate how long a task should run as a function of the total number of currently runnable process

Core Idea

- Calculate how long a task should run as a function of the total number of currently runnable process
  - Use the priority value to weight a proportion of the CPU the process is to receive – higher priority job should get more CPU time **proportional to the priority of the other processes in the runqueue**

Remember the way we used to set up the time slices earlier:

- Prio < 120; T = (140-Prio) x 20
- Prio ≥ 120; T = (140-Prio) x 5

CFS – Scheduling Algorithm

Case 1: Tasks have the same priority values

CFS – Scheduling Algorithm

<u>Case 1: Tasks have the same priority values</u>

- Set a bounded target (amount of time) *T* within which the scheduler tries to execute all the runnable tasks – if there are *N* runnable tasks, each task will get *T/N* amount of time
  - Let, T = 10ms
    - N = 2, each task will get 5ms
    - N = 5, each task will get 2ms
    - N = 10, each task will get 1ms

CFS – Scheduling Algorithm

Case 1: Tasks have the same priority values

- Set a bounded target (amount of time) *T* within which the scheduler tries to execute all the runnable tasks– if there are *N* runnable tasks, each task will get *T/N* amount of time
  - Let, T = 10ms
    - N = 2, each task will get 5ms
    - N = 5, each task will get 2ms
    - N = 10, each task will get 1ms
  - $N \rightarrow \infty$, each task is likely to get zero time ! CFS dynamically adjusts the target time to ensure each task gets at least 1ms
    - Context switching time should not take over the time for task execution

CFS – Scheduling Algorithm

<u>Case 1: Tasks have the same priority values</u>

- Set a bounded target (amount of time) *T* within which the scheduler tries to execute all the runnable tasks – if there are *N* runnable tasks, each task will get *T/N* amount of time
  - Let, T = 10ms
    - N = 2, each task will get 5ms
    - N = 5, each task will get 2ms
    - N = 10, each task will get 1ms
  - $N \rightarrow \infty$, each task is likely to get zero time ! CFS dynamically adjusts the target time to ensure each task gets at least 1ms
    - Context switching time should not take over the time for task execution
  - Default target time = 20ms, 4ms chunks are added when each task is likely to get less than 1ms

CFS – Scheduling Algorithm

Case 2: Moving from the Task Groups of One Priority Level to the Next

CFS – Scheduling Algorithm

Case 2: Moving from the Task Groups of One Priority Level to the Next

- Assign timeslices in proportion to their priority levels

CFS – Scheduling Algorithm

Case 2: Moving from the Task Groups of One Priority Level to the Next

- Assign timeslices in proportion to their priority levels
  - Assume two processes having priority values (niceness) 5 and 10, respectively

CFS – Scheduling Algorithm

Case 2: Moving from the Task Groups of One Priority Level to the Next

- Assign timeslices in proportion to their priority levels
  - Assume two processes having priority values (niceness) 5 and 10, respectively
  - Default target time (period) = 20ms

CFS – Scheduling Algorithm

Case 2: Moving from the Task Groups of One Priority Level to the Next

- Assign timeslices in proportion to their priority levels
  - Assume two processes having priority values (niceness) 5 and 10, respectively
  - Default target time (period) = 20ms
  - Makes a mapping from niceness to weights:
    - 5 translates to 335
    - 10 translated to 110

CFS – Scheduling Algorithm

Case 2: Moving from the Task Groups of One Priority Level to the Next

- Assign timeslices in proportion to their priority levels
  - Assume two processes having priority values (niceness) 5 and 10, respectively
  - Default target time (period) = 20ms
  - Makes a mapping from niceness to weights:
    - 5 translates to 335
    - 10 translated to 110
  - **Time allocated to the process with niceness 5 = 335/(335+110) x 20ms = 15.056 ms**
  - **Time allocated to the process with niceness 10 = 110/(335+110) x 20ms = 4.944 ms**

CFS – Scheduling Algorithm

Case 2: Moving from the Task Groups of One Priority Level to the Next

- Assign timeslices in proportion to their priority levels
  - Assume two processes having priority values (niceness) 5 and 10, respectively
  - Default target time (period) = 20ms
  - Makes a mapping from niceness to weights:
    - 5 translates to 335
    - 10 translated to 110
  - **Time allocated to the process with niceness 5 = 335/(335+110) x 20ms = 15.056 ms**
  - **Time allocated to the process with niceness 10 = 110/(335+110) x 20ms = 4.944 ms**

CFS calls this as the Wall-Clock Slice

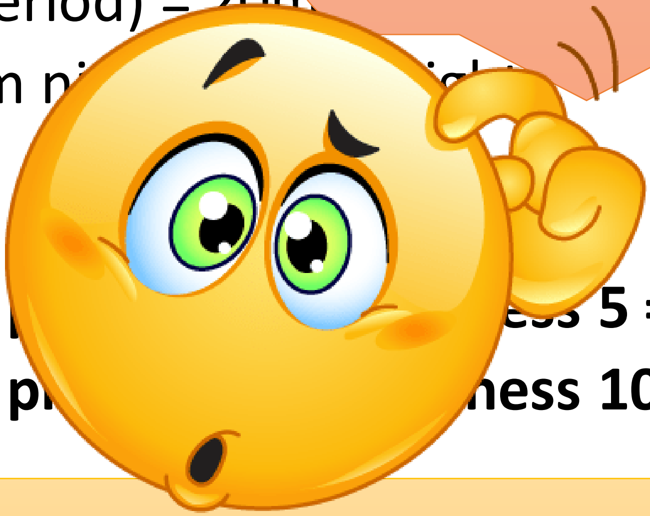
CFS – Scheduling Algorithm

Case 2: Moving from the Task Gr **Next**

- Assign timeslices in proportion t
  - Assume two processes having pri tively
  - Default target time (period) = 20m
  - Makes a mapping from ni ight
    - 5 translates to 335
    - 10 translated to 110
  - **Time allocated to the p ess 5 = 335/(335+110) x 20ms = 15.056 ms**
  - **Time allocated to the p ness 10 = 110/(335+110) x 20ms = 4.944 ms**

How do we calculate the wall-clock slice?

- ✓ **Number of processes in the runqueue is dynamic**
- ✓ **Needs an iteration over the entire runqueue – O(N)?**



CFS calls this as the Wall-Clock Slice

Red Black (RB) Tree -- A Self-balancing Binary Search Tree

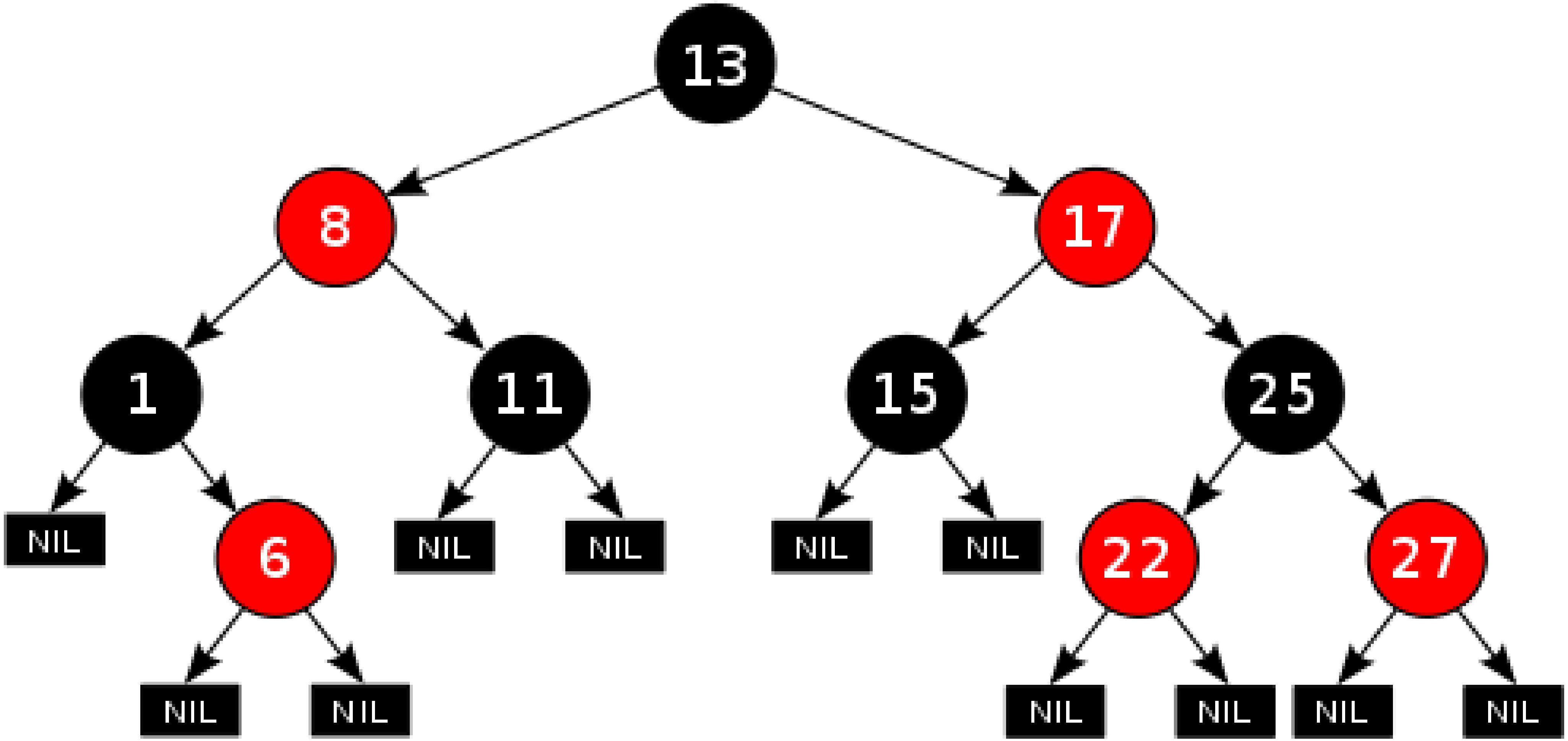


Image source: Wikipedia

CFS Runqueue as a RB Tree

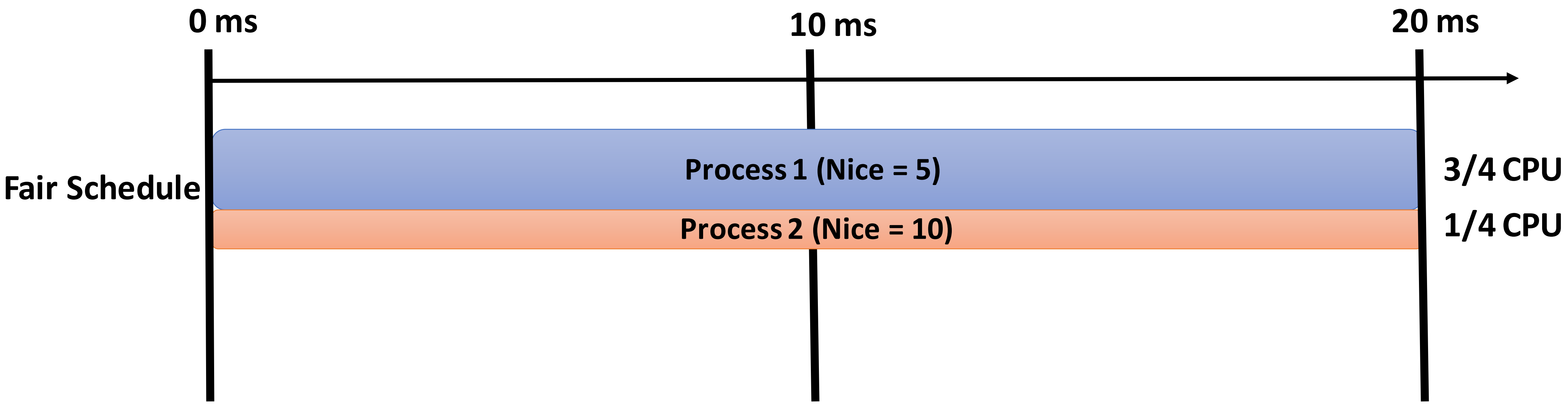
- CFS uses a time-ordered RB tree

- Every processor maintains its own runqueue
  - Each runqueue is implemented as an RB tree

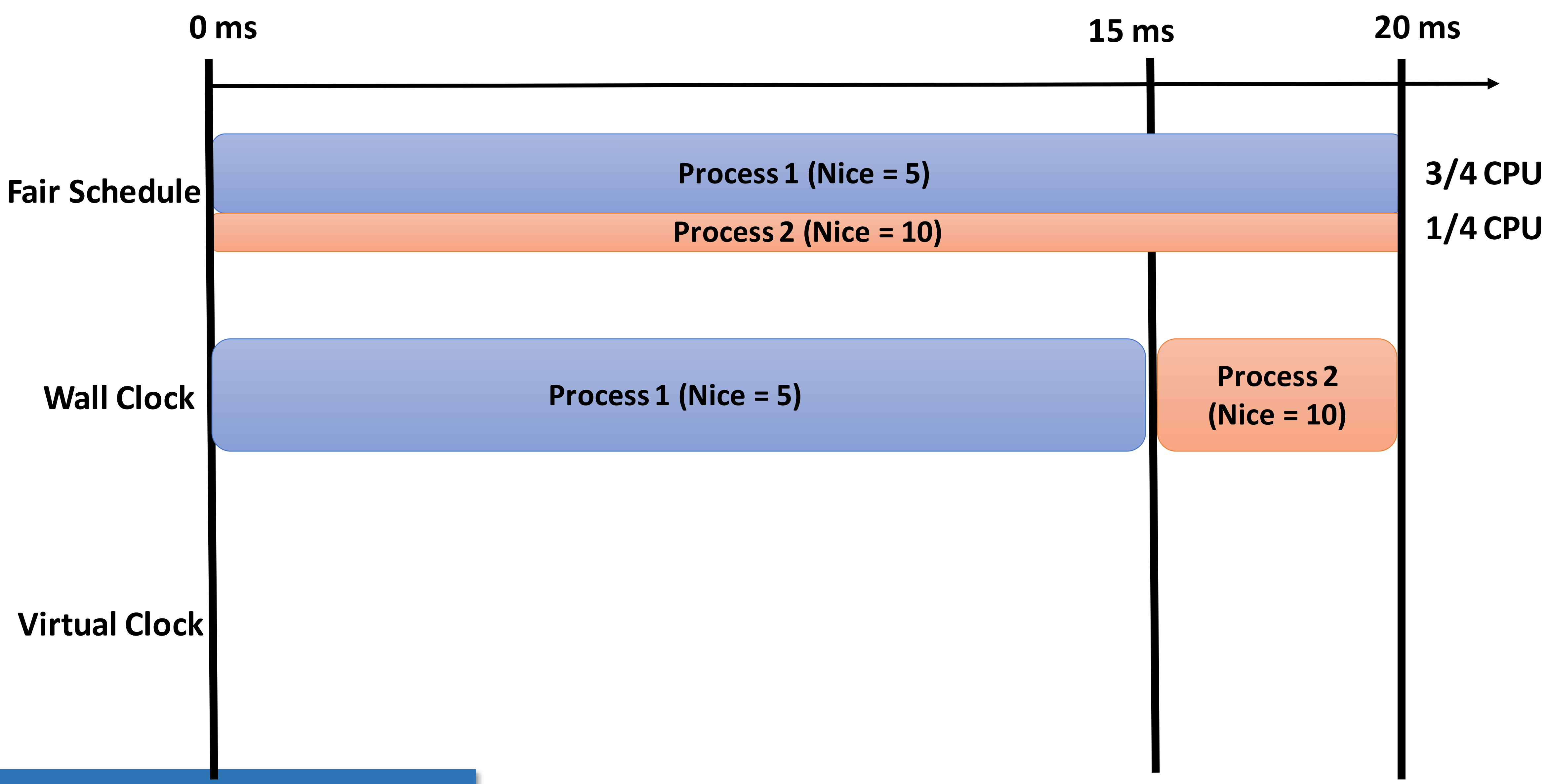
- The nodes of a runqueue is either a task or a task-group (`sched_entity`)
  - **Task group:** A group of tasks (processes) having the same/similar functionalities; ex. An HTTP server having multiple threads / processes for parallelization

- The nodes are indexed by their virtual runtimes (vruntime) -- **amount of time a waiting process would have been allowed to spend on the CPU on a completely fair system**

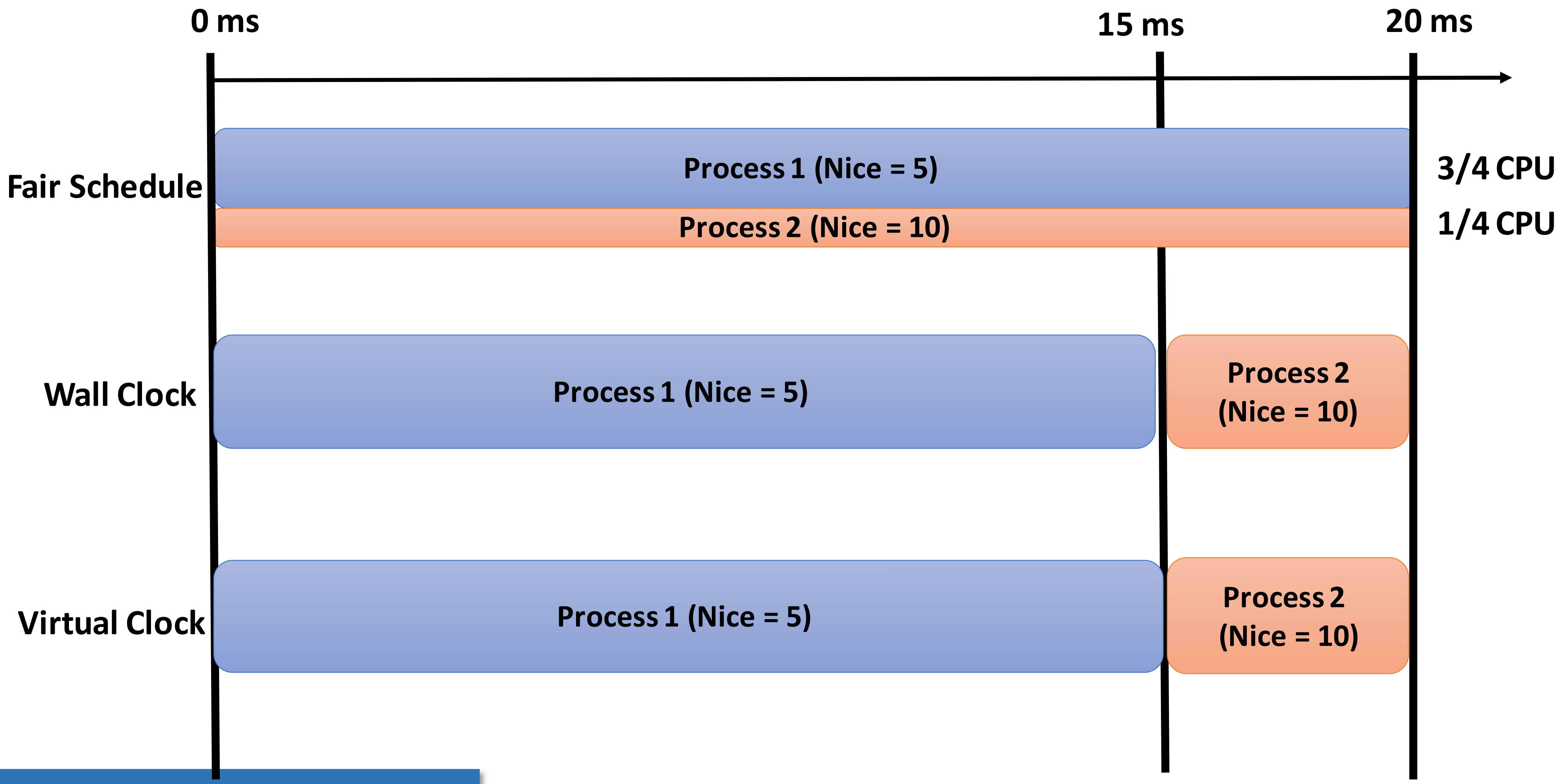
Conceptualizing the Virtual Runtime using a Virtual Clock



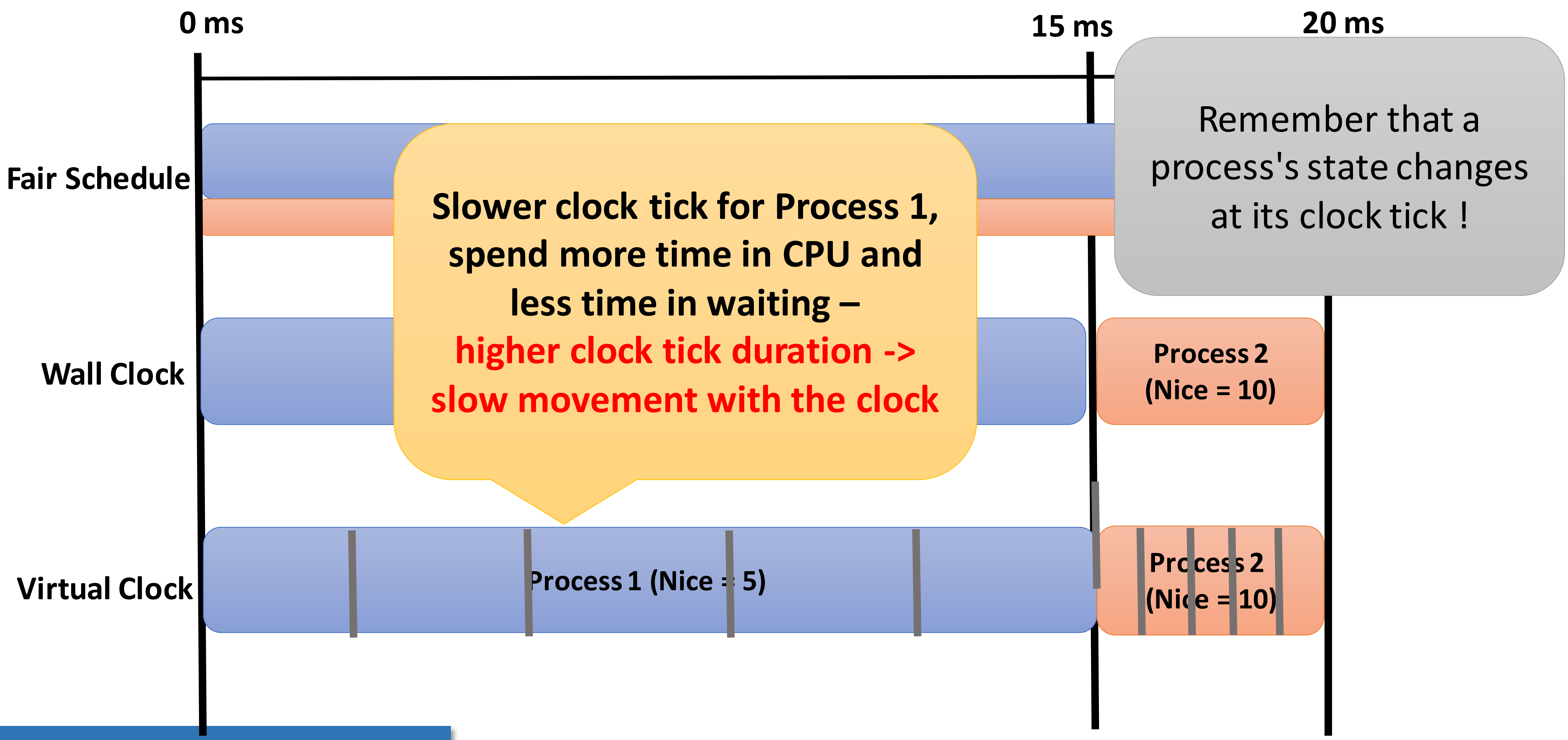
Conceptualizing the Virtual Runtime using a Virtual Clock



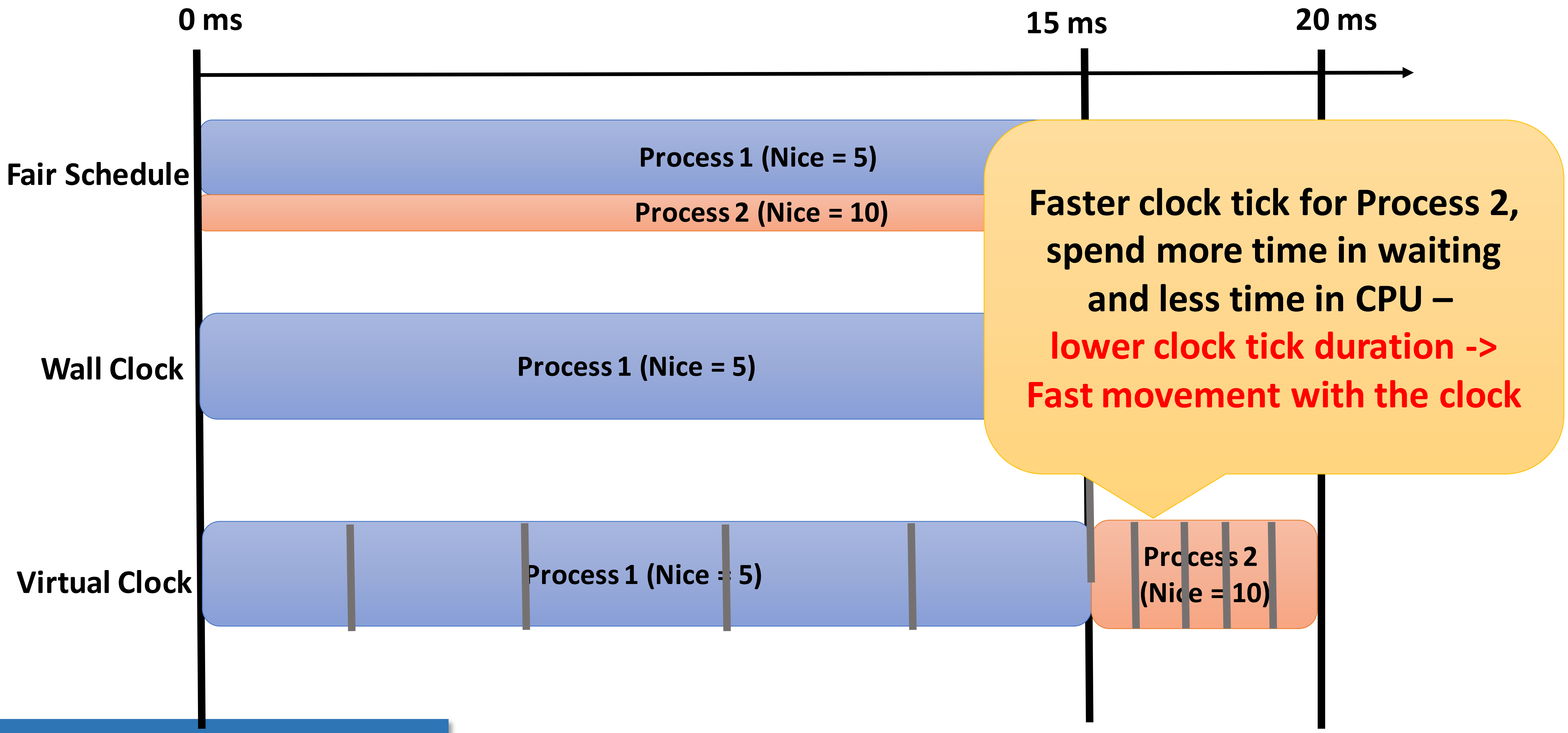
Conceptualizing the Virtual Runtime using a Virtual Clock



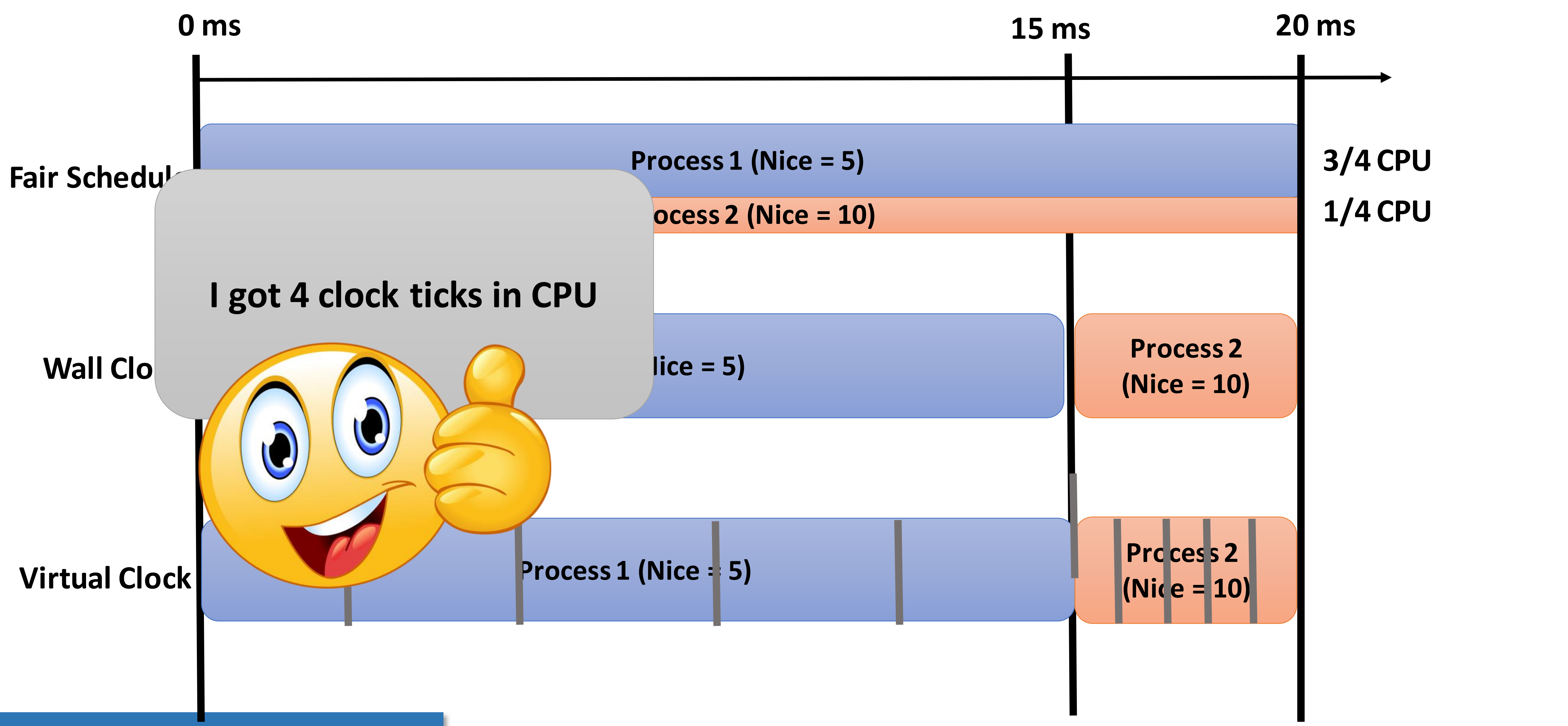
Conceptualizing the Virtual Runtime using a Virtual Clock



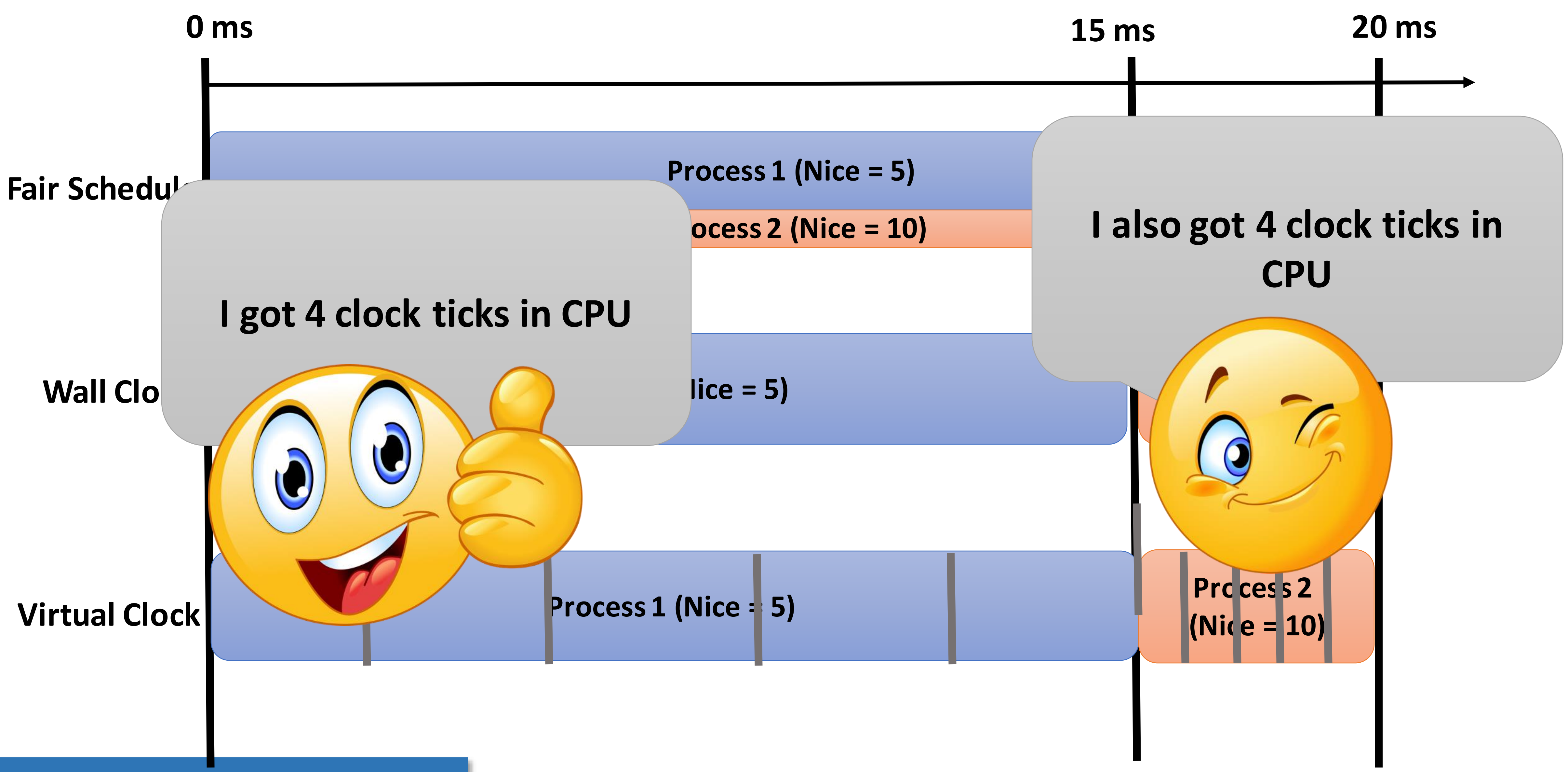
Conceptualizing the Virtual Runtime using a Virtual Clock



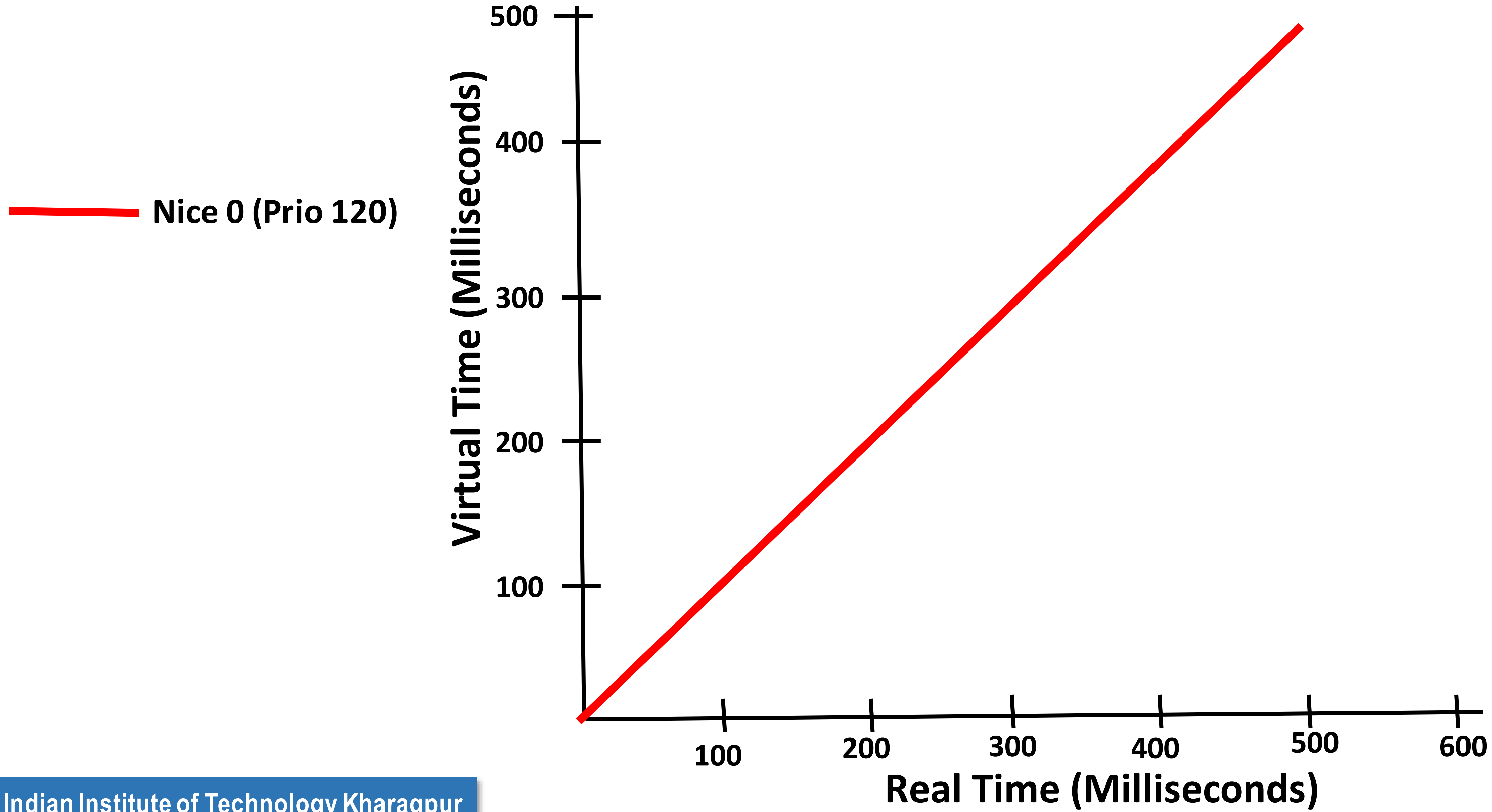
Conceptualizing the Virtual Runtime using a Virtual Clock



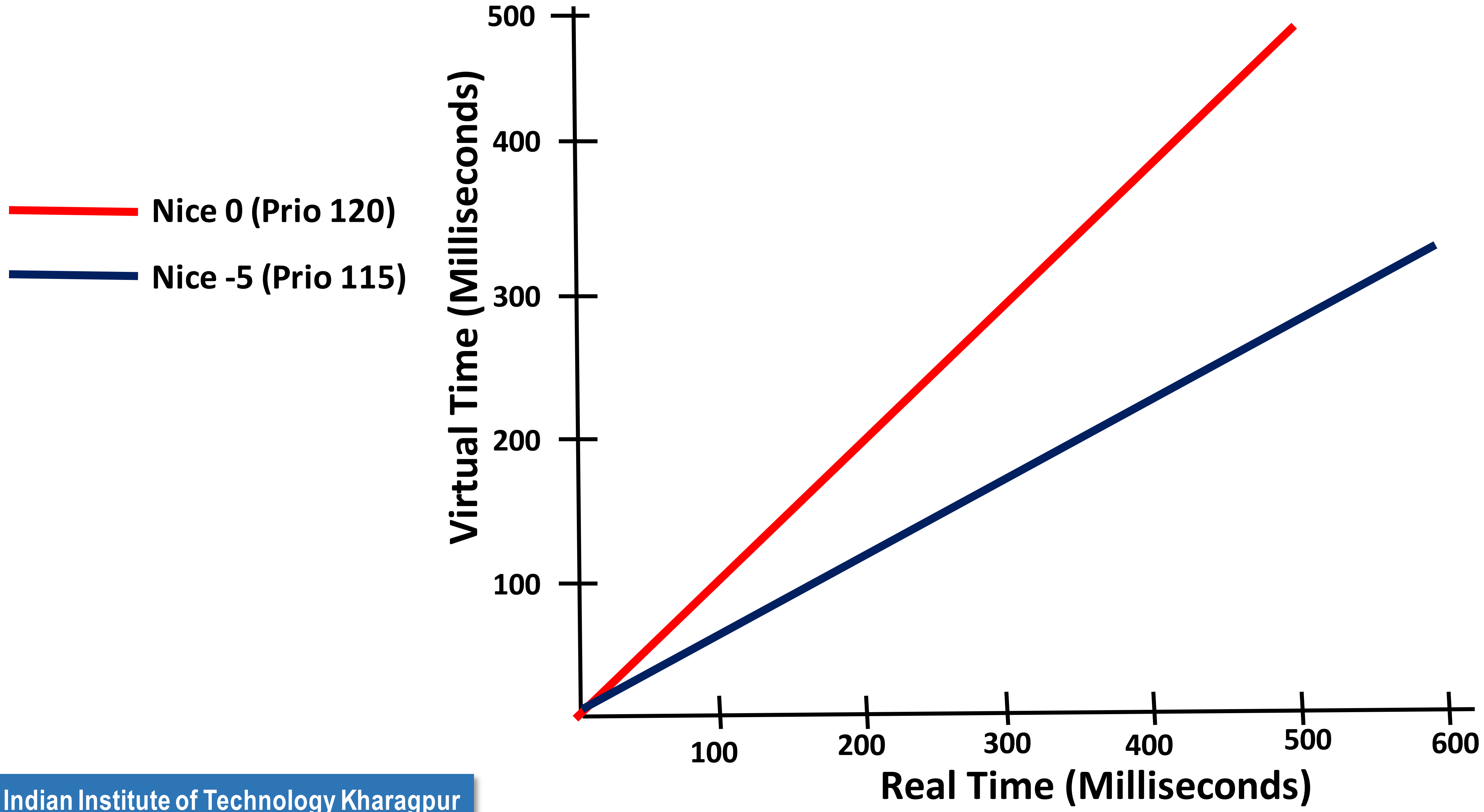
Conceptualizing the Virtual Runtime using a Virtual Clock



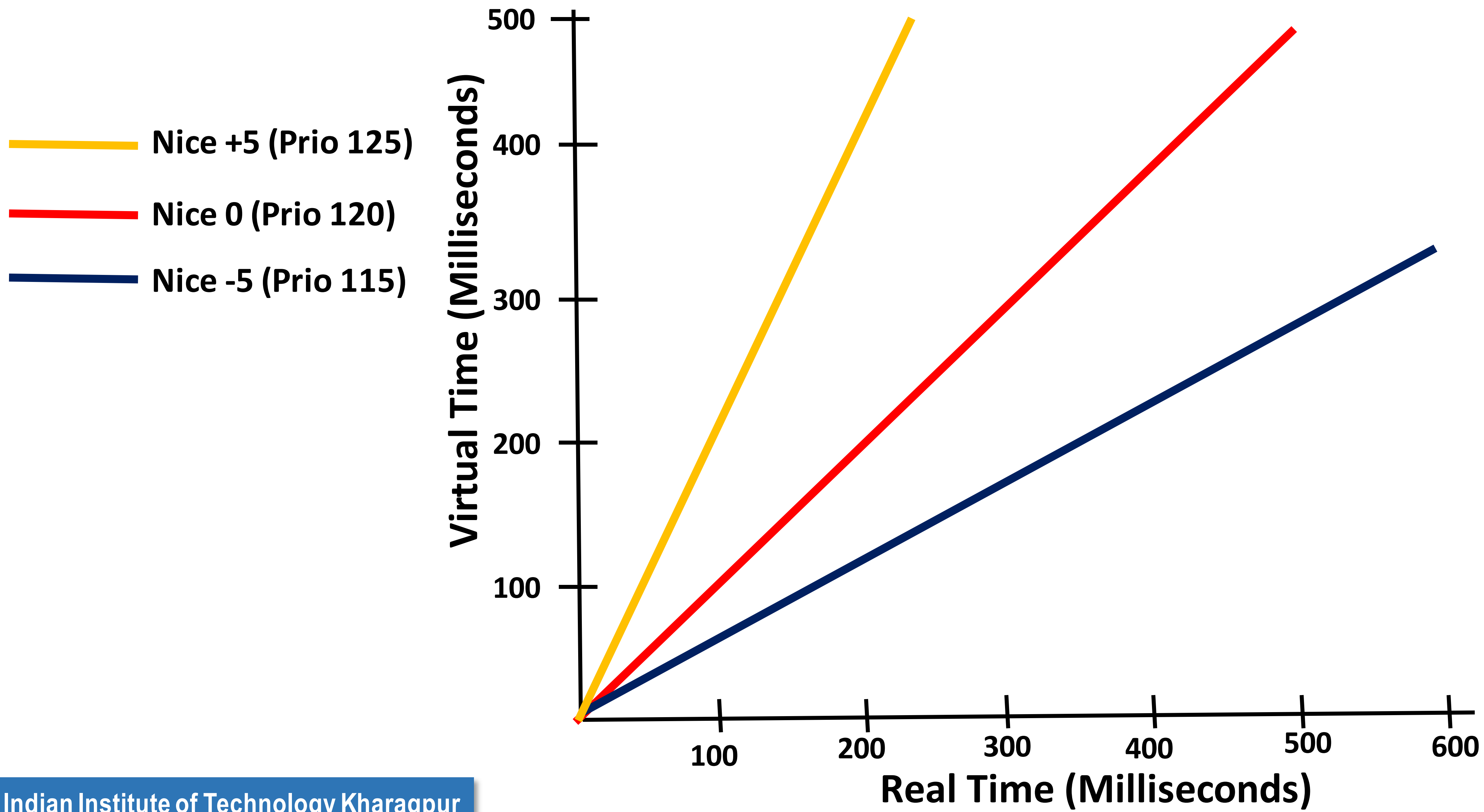
Real Time (Wall Clock) vs Virtual Time



Real Time (Wall Clock) vs Virtual Time



Real Time (Wall Clock) vs Virtual Time



Implementing the Virtual Runtime

- CFS uses a virtual clock to implement vruntime.
- For a runnable task (or task group), the vruntime is updated as the task executes in the CPU, as follows:

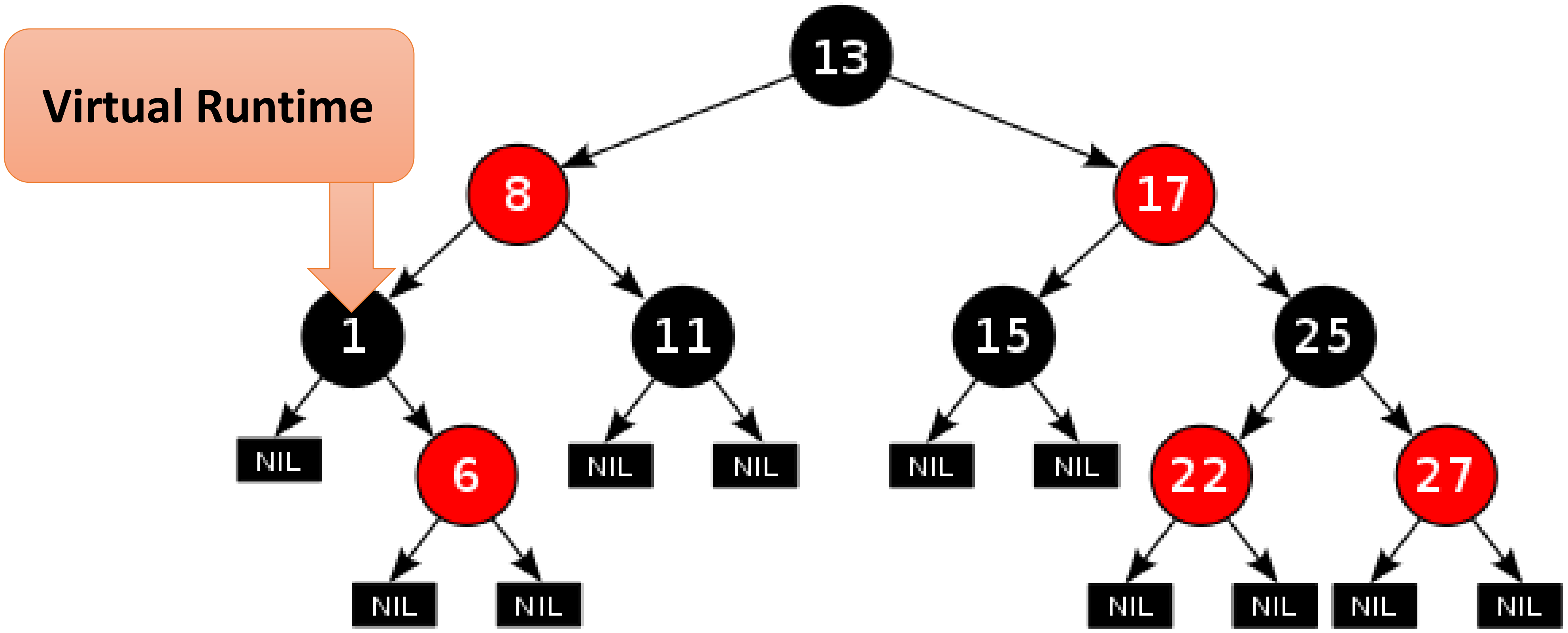
```
vruntime += delta_exec x (NICE_0_LOAD / se->load.weight)
```

Here, `delta_exec` is the real-time elapsed for the currently running process, `NICE_0_LOAD` is 1024 and the `load.weight` for a process is computed from its nice value

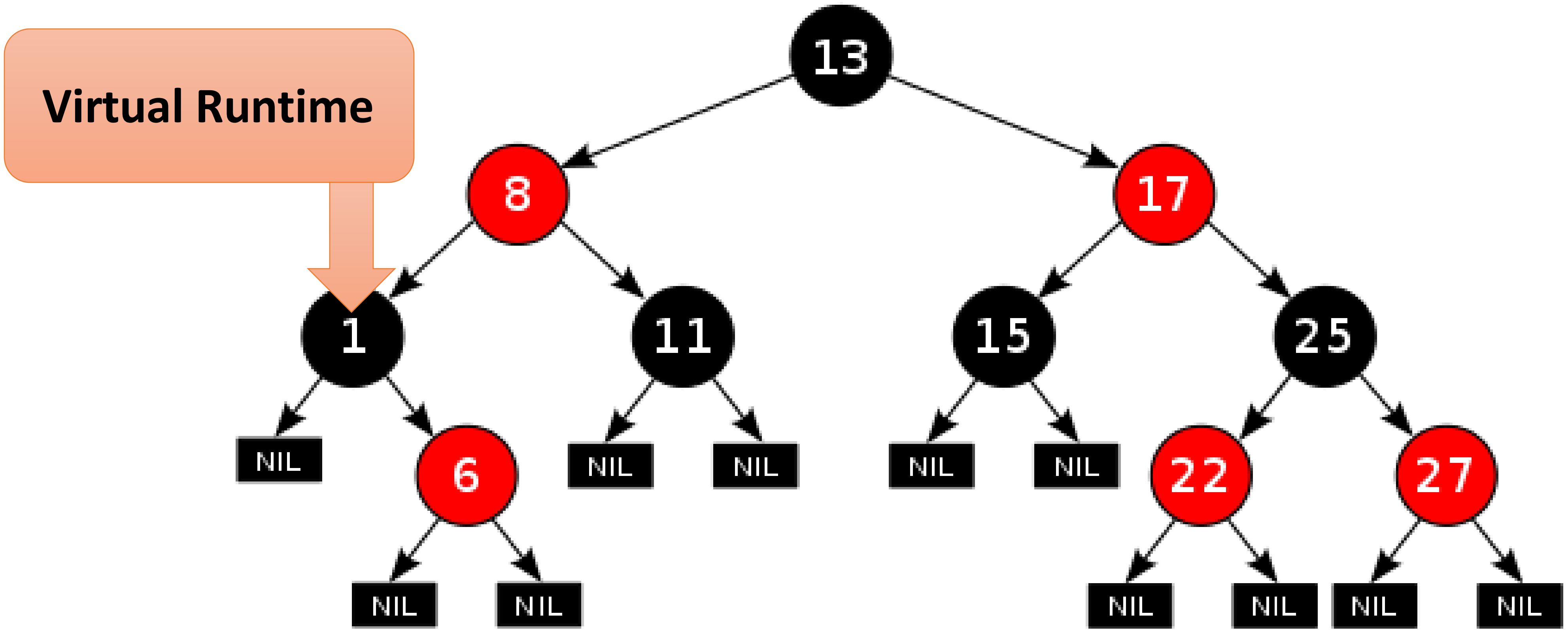
From Nice Value to Weight

```
static const int 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,
};
```

CFS Runqueue



CFS Runqueue

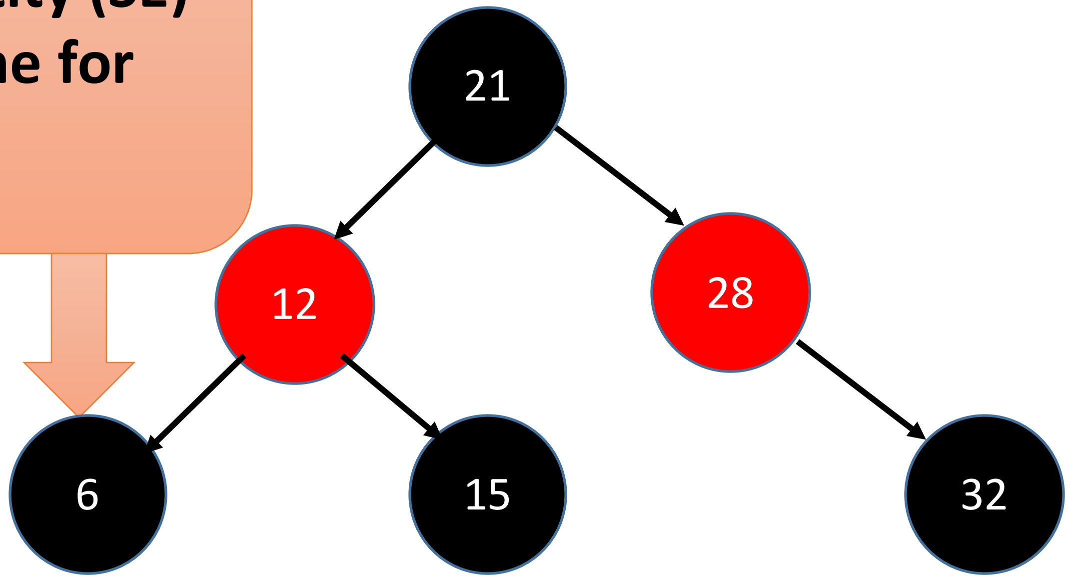


Less CPU-time
More need of the CPU

More CPU-time
Less need of the CPU

CFS in Action

Select the Scheduling Entity (SE) with minimum vruntime for execution

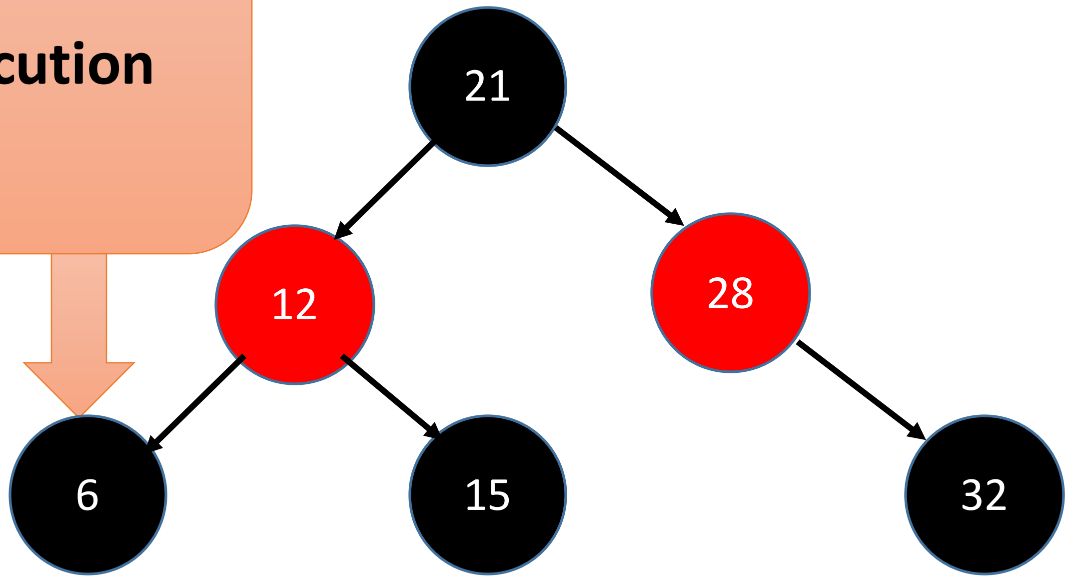


Less CPU-time
More need of the CPU

More CPU-time
Less need of the CPU

CFS in Action

Dequeue the SE for execution

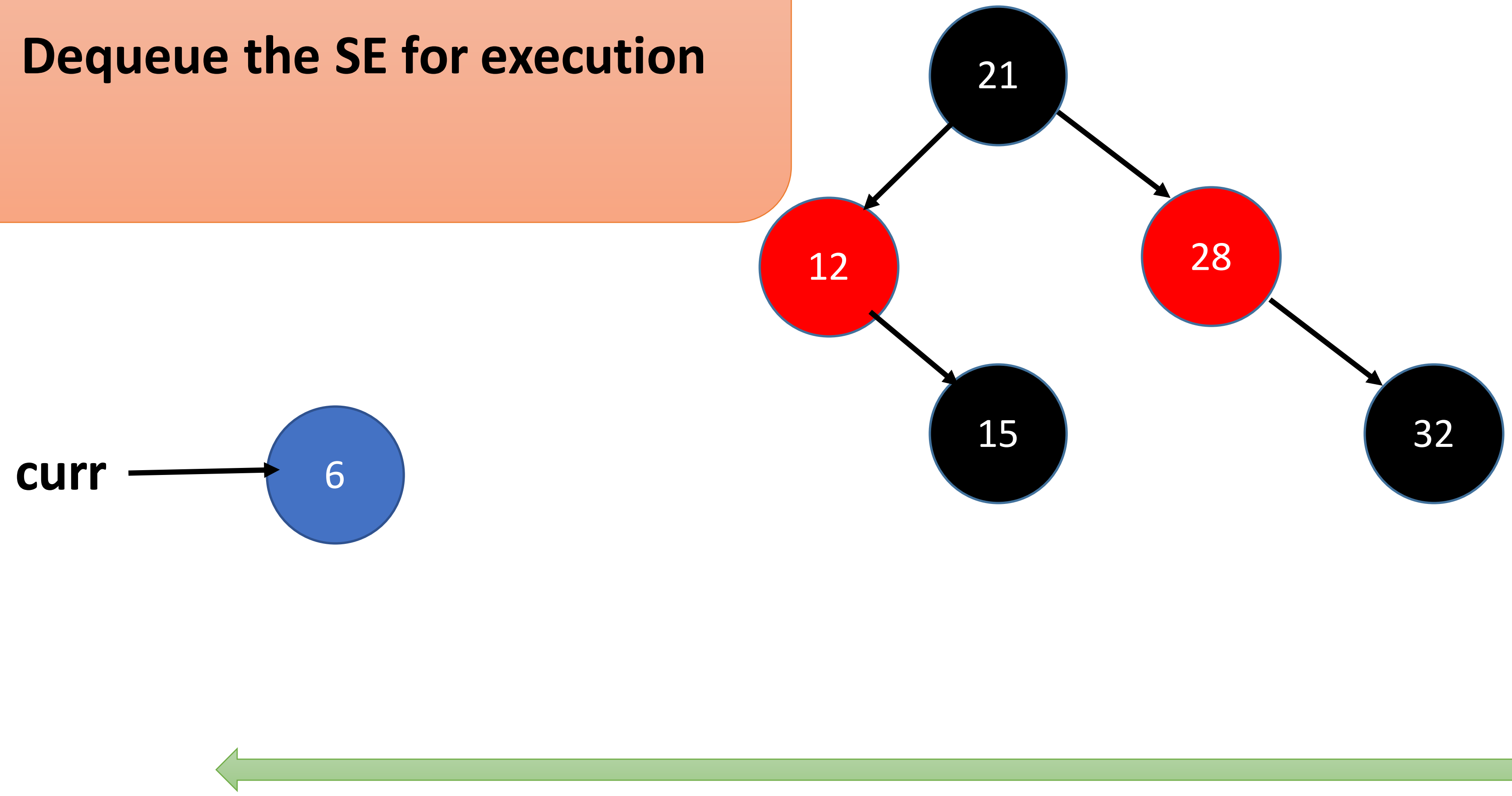


Less CPU-time
More need of the CPU

More CPU-time
Less need of the CPU

CFS in Action

Dequeue the SE for execution

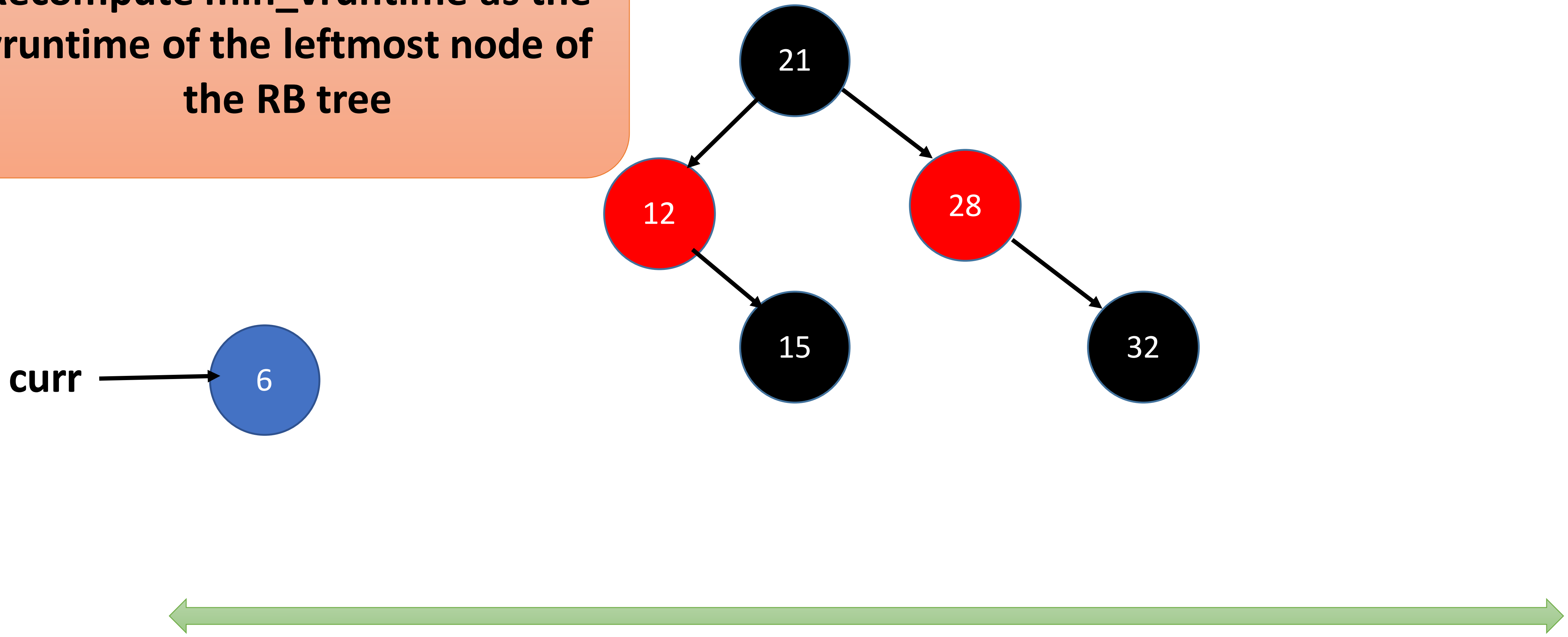


Less CPU-time
More need of the CPU

More CPU-time
Less need of the CPU

CFS in Action

Recompute min_vruntime as the vruntime of the leftmost node of the RB tree



Less CPU-time

More need of the CPU

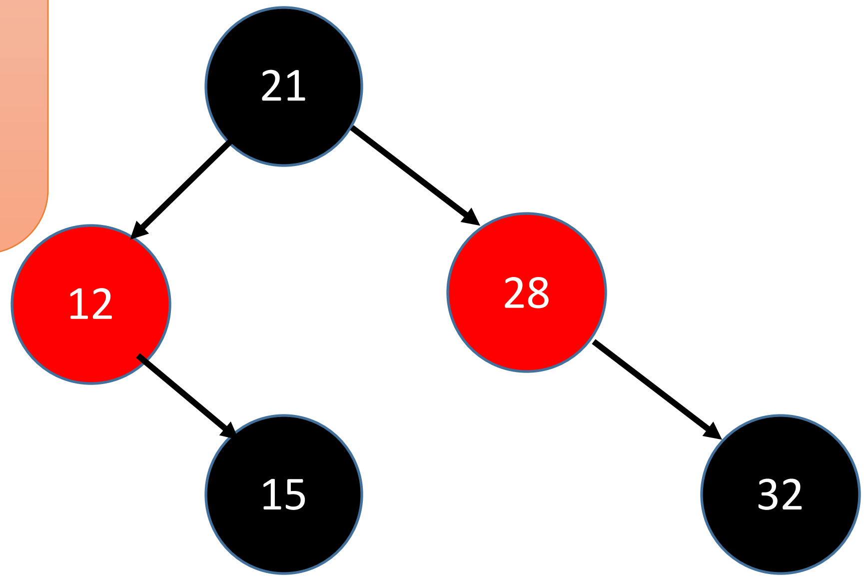
More CPU-time

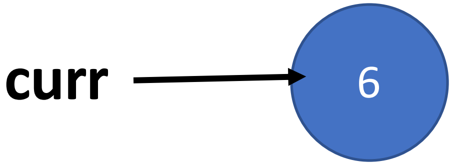
Less need of the CPU

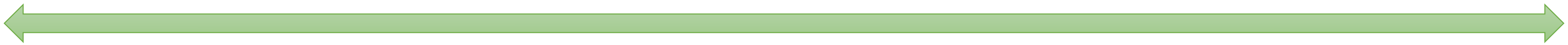
CFS in Action

Recompute min_vruntime as the vruntime of the leftmost node of the RB tree

min_vruntime = 12





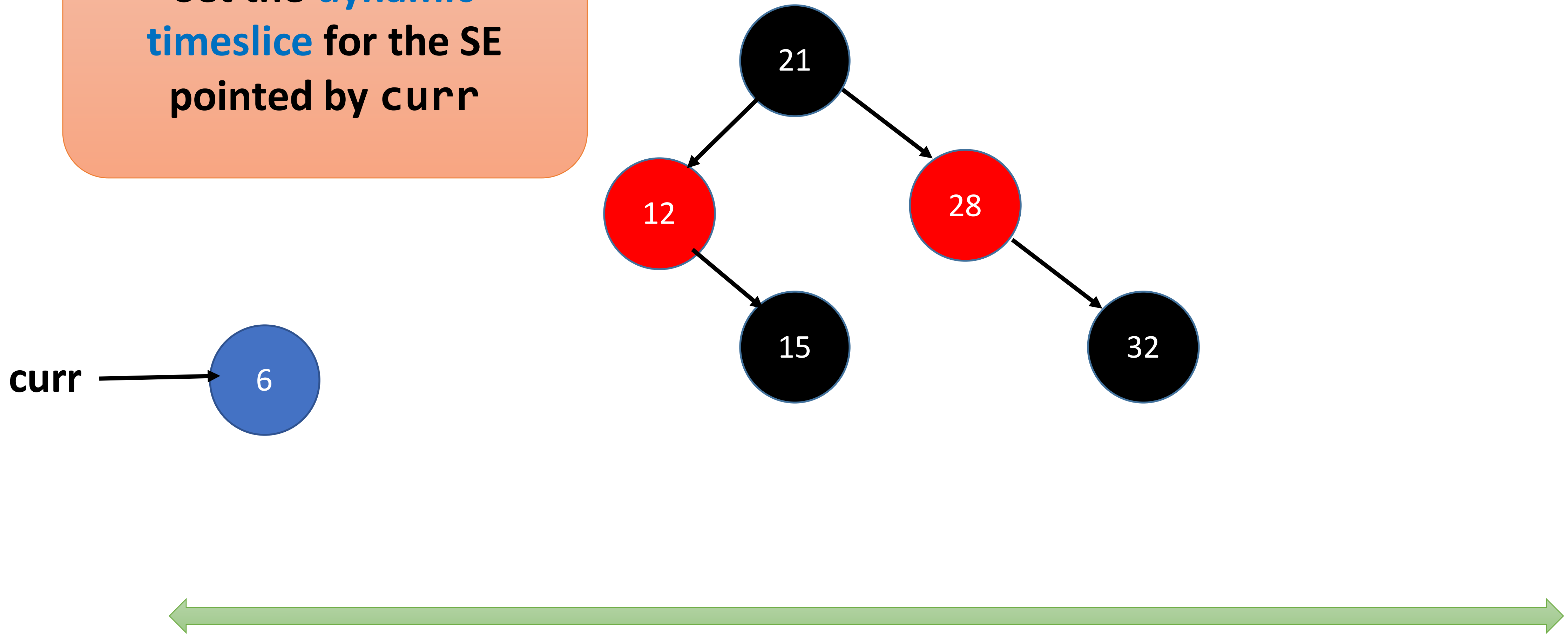


Less CPU-time
More need of the CPU

More CPU-time
Less need of the CPU

CFS in Action

Set the **dynamic timeslice** for the SE pointed by `curr`

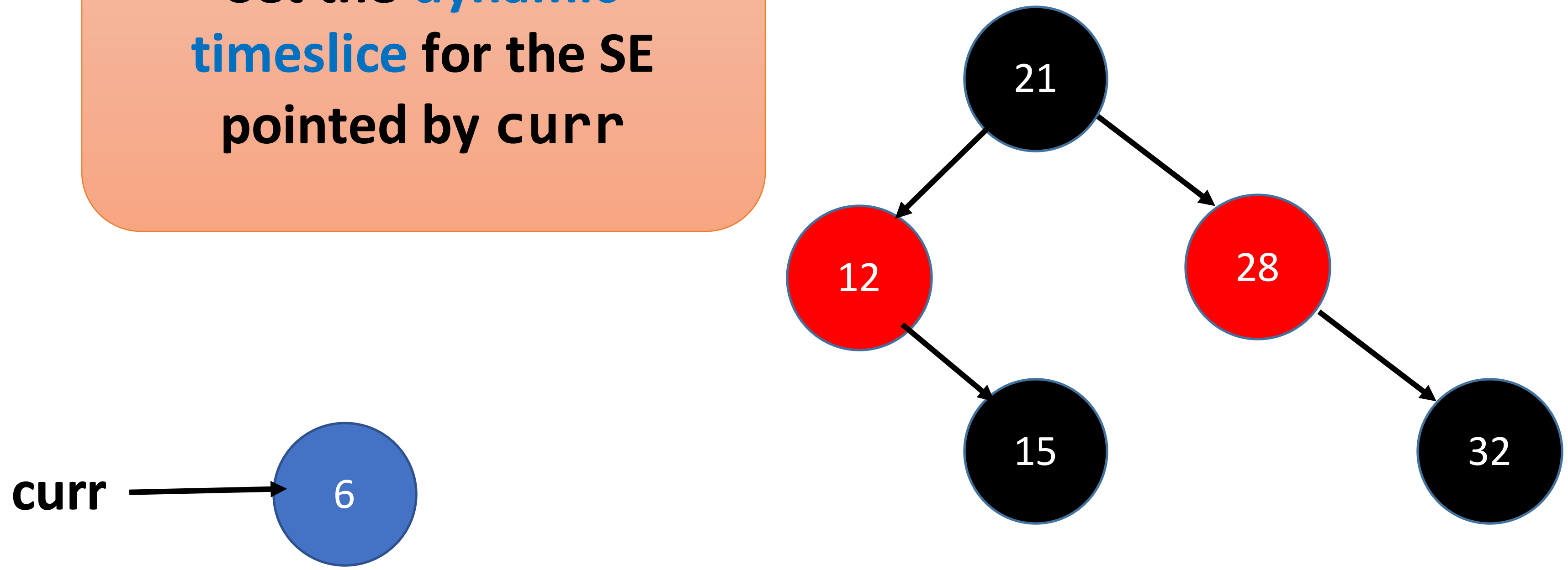


Less CPU-time
More need of the CPU

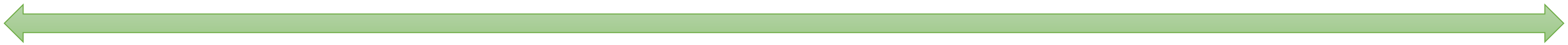
More CPU-time
Less need of the CPU

CFS in Action

Set the dynamic timeslice for the SE pointed by `curr`




```
slice = sched_period x (se->load.weight / cfs_rq->load)
```

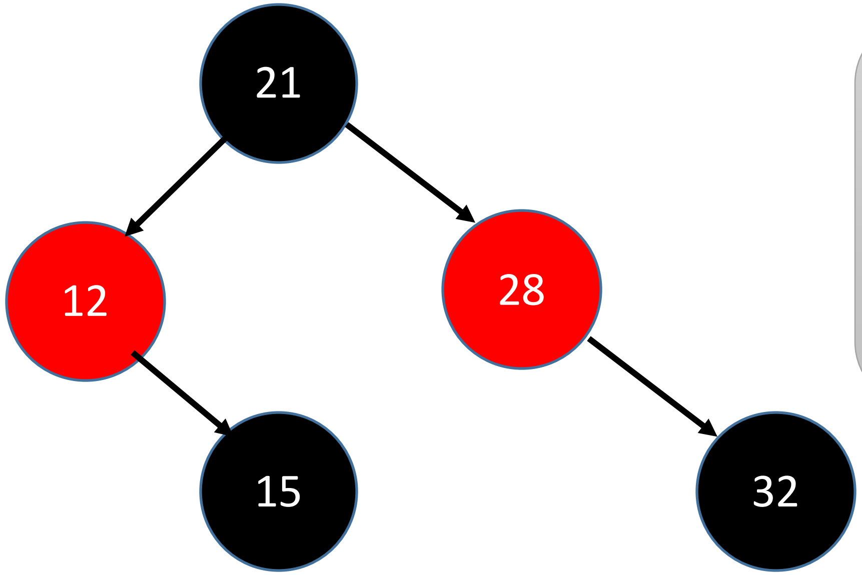


Less CPU-time
More need of the CPU

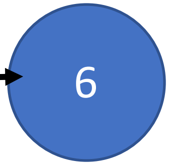
More CPU-time
Less need of the CPU

CFS in Action

Set the **dynamic timeslice** for the SE pointed by `curr`



curr



Remember, that the `sched_period` is dynamic

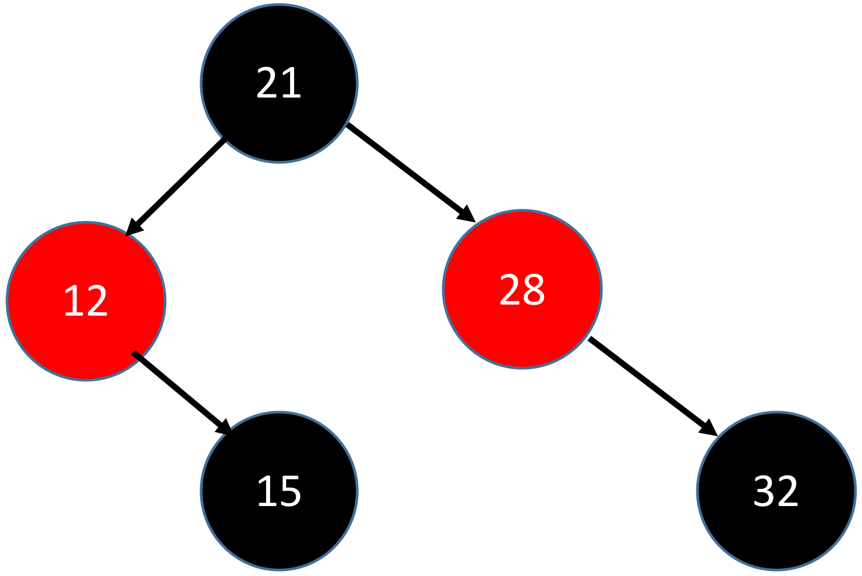
```
slice = sched_period x (se->load.weight / cfs_rq->load)
```

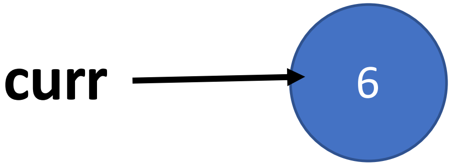
Less CPU-time
More need of the CPU

More CPU-time
Less need of the CPU

CFS in Action

Set the dynamic timeslice for the SE pointed by `curr`





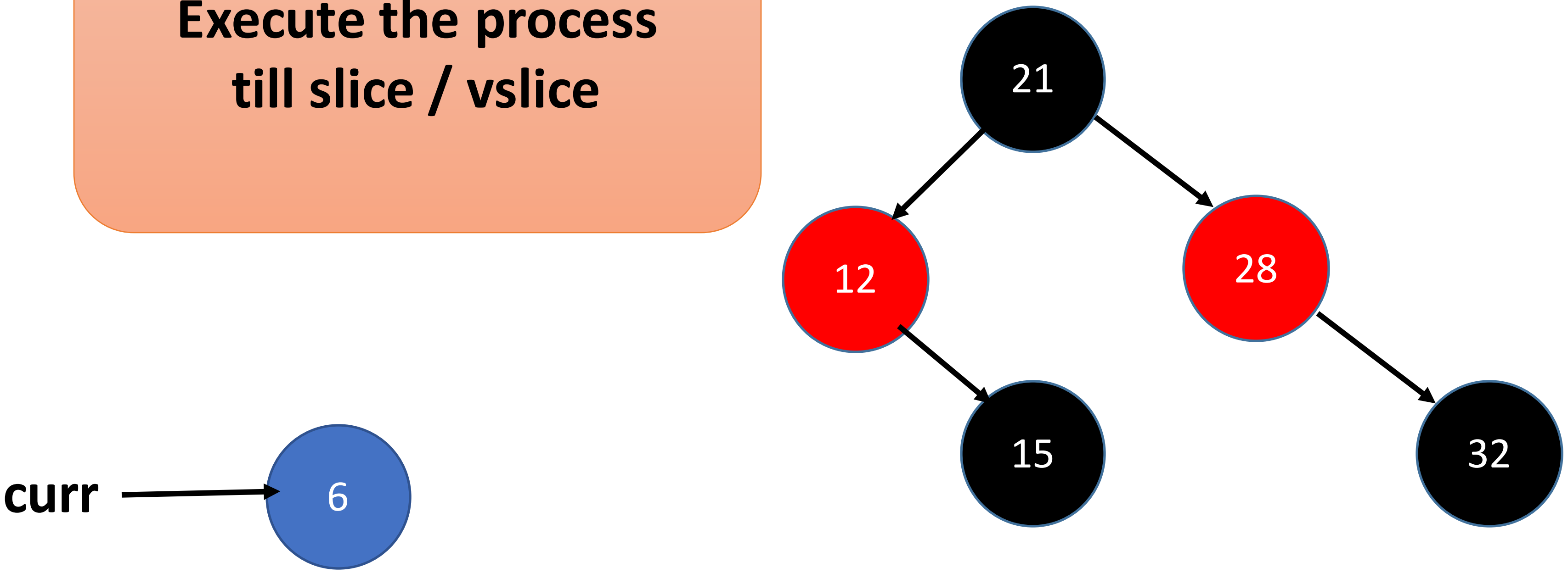

```
vslice = slice x (NICE_0_LOAD / se->load.weight)
```

Less CPU-time
More need of the CPU

More CPU-time
Less need of the CPU

CFS in Action

Execute the process till slice / vslice

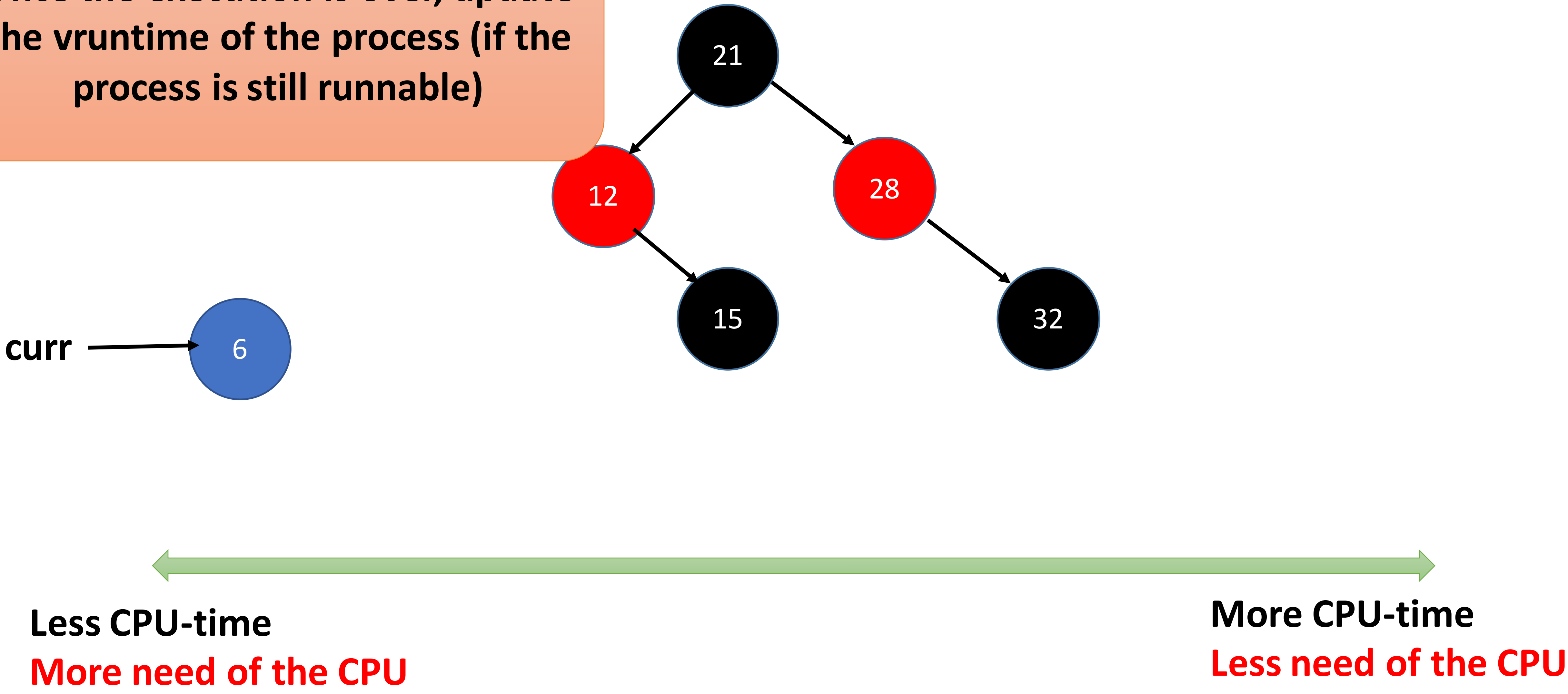


Less CPU-time
More need of the CPU

More CPU-time
Less need of the CPU

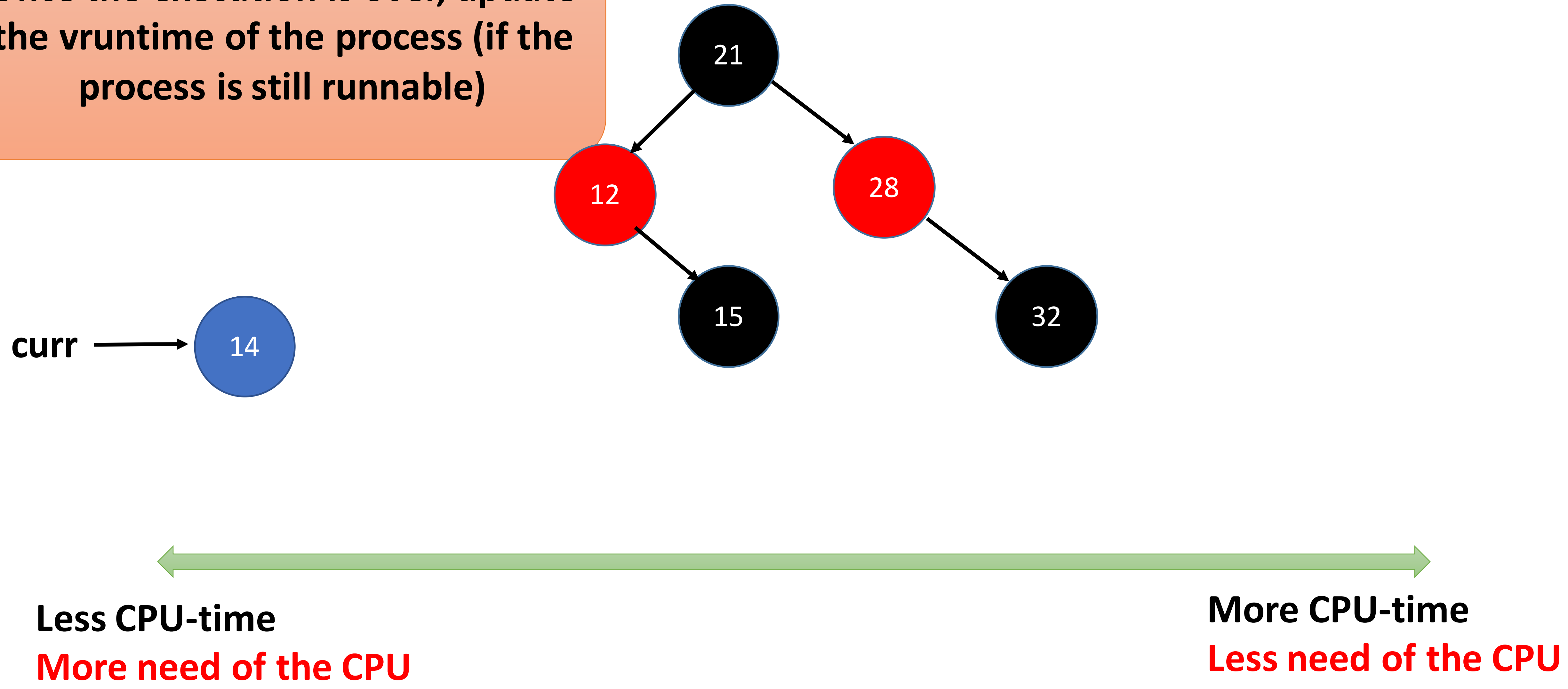
CFS in Action

Once the execution is over, update the vruntime of the process (if the process is still runnable)



CFS in Action

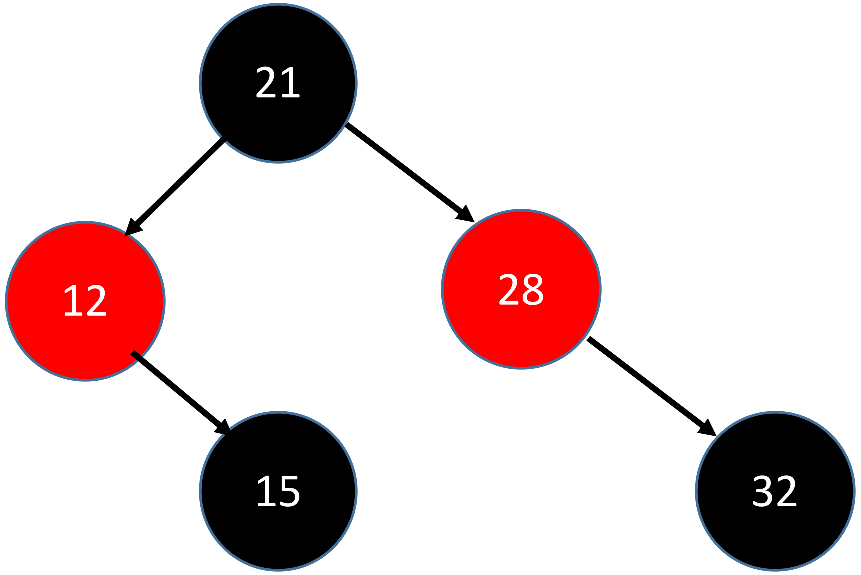
Once the execution is over, update the vruntime of the process (if the process is still runnable)

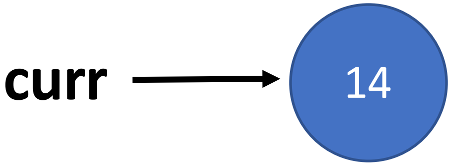


CFS in Action

Check with the cached value of min_vruntime

min_vruntime = 12





Less CPU-time

More need of the CPU

More CPU-time

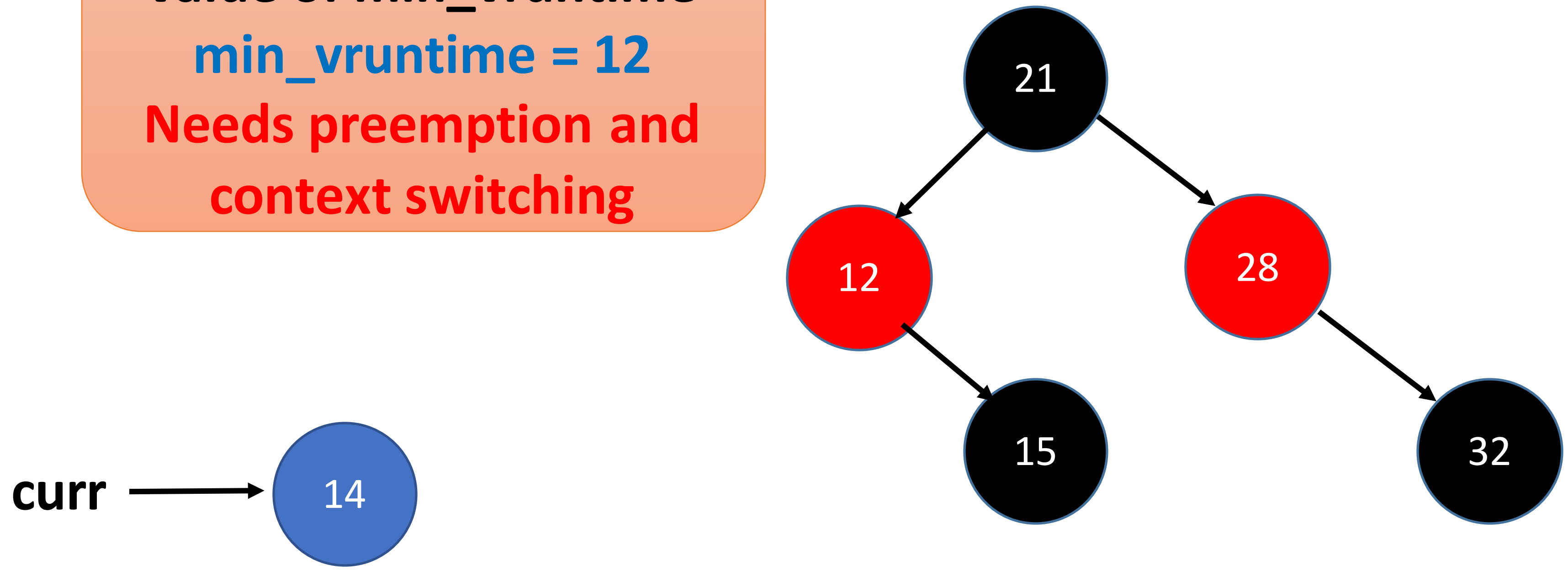
Less need of the CPU

CFS in Action

Check with the cached value of min_vruntime

min_vruntime = 12

Needs preemption and context switching



Less CPU-time

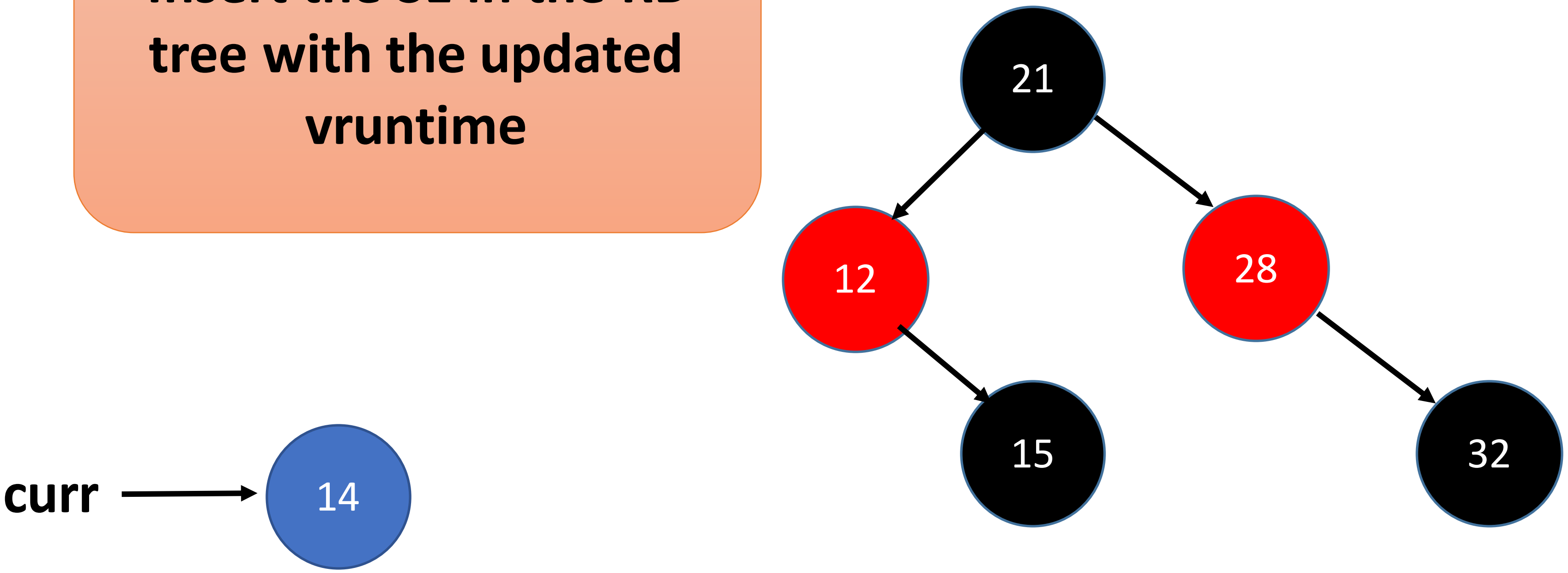
More need of the CPU

More CPU-time

Less need of the CPU

CFS in Action

Insert the SE in the RB tree with the updated vruntime

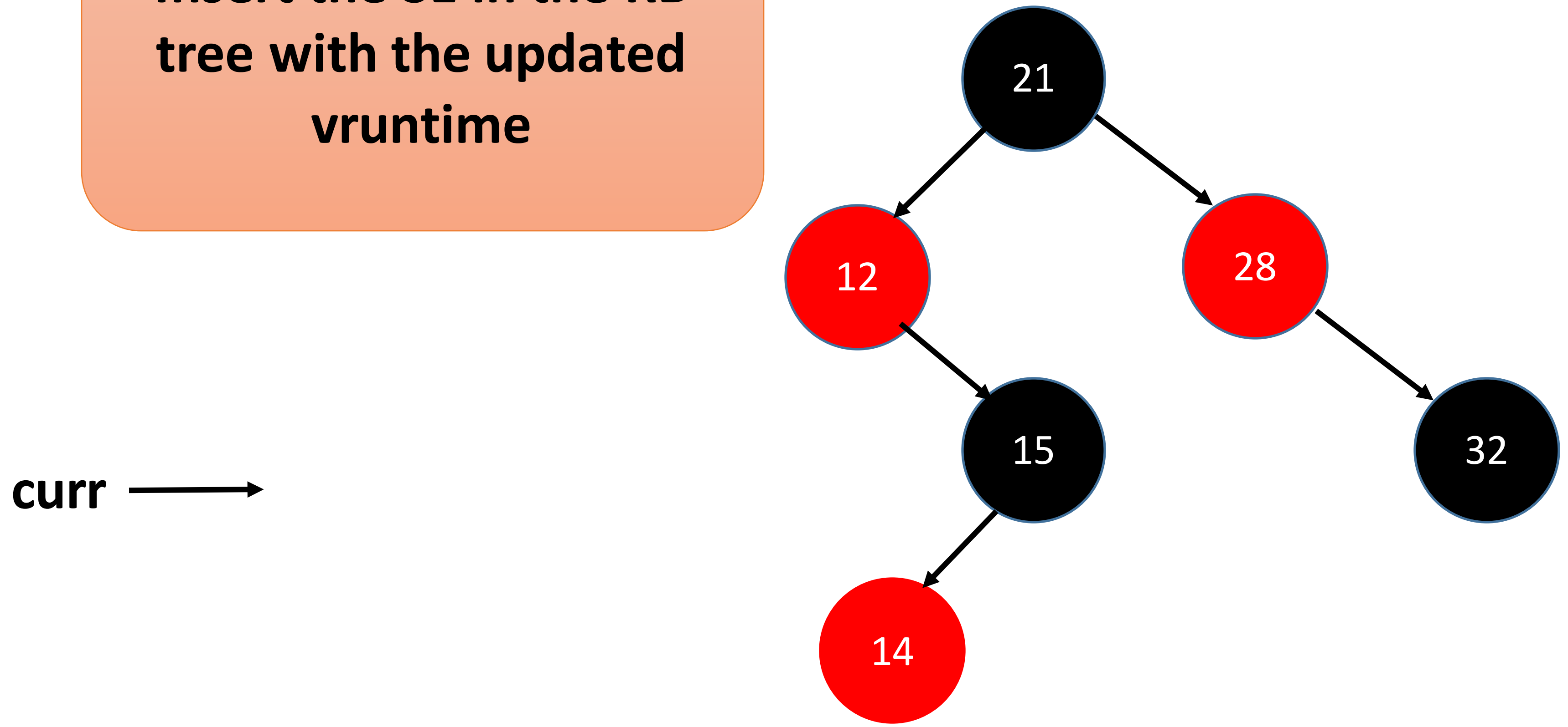


Less CPU-time
More need of the CPU

More CPU-time
Less need of the CPU

CFS in Action

Insert the SE in the RB tree with the updated vruntime



Less CPU-time

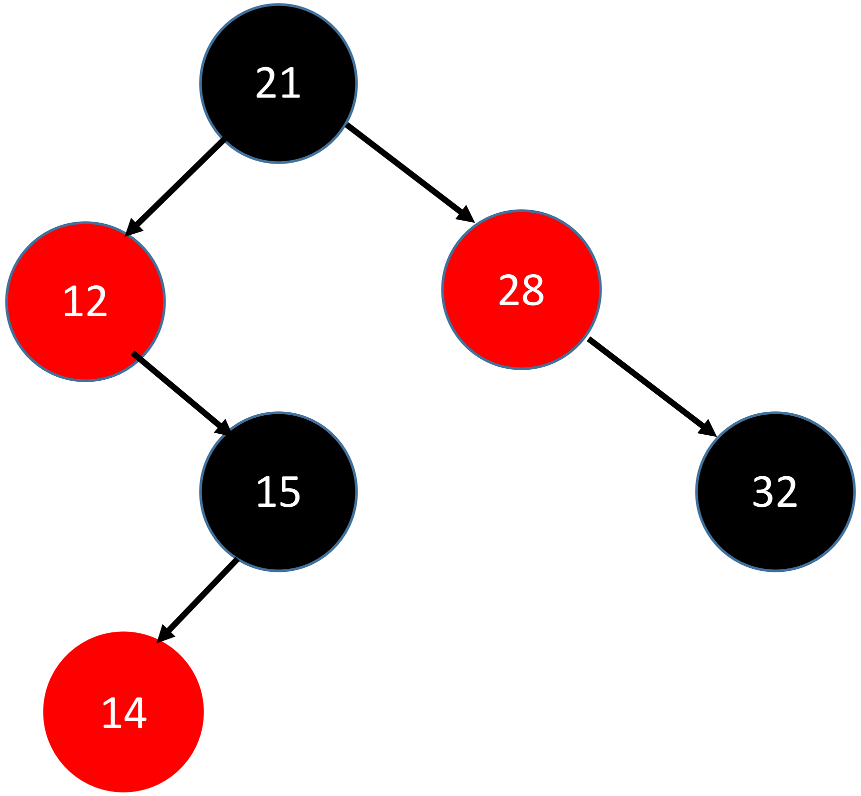
More need of the CPU

More CPU-time

Less need of the CPU

CFS in Action

Insert the SE in the RB tree with the updated vruntime



An AVL tree could result a better hight balancing – Why don't we use that?

curr →

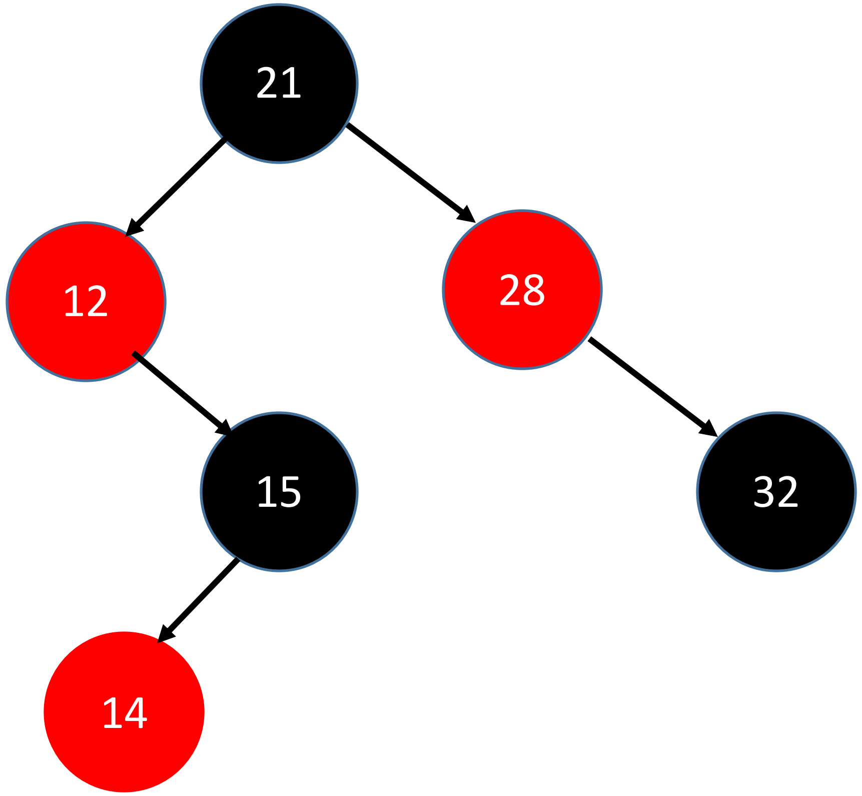
Less CPU-time
More need of the CPU

More CPU-time
Less need of the CPU

CFS in Action

Extract the leftmost node for scheduling, update min_vruntime , rebalance the tree

curr



Less CPU-time

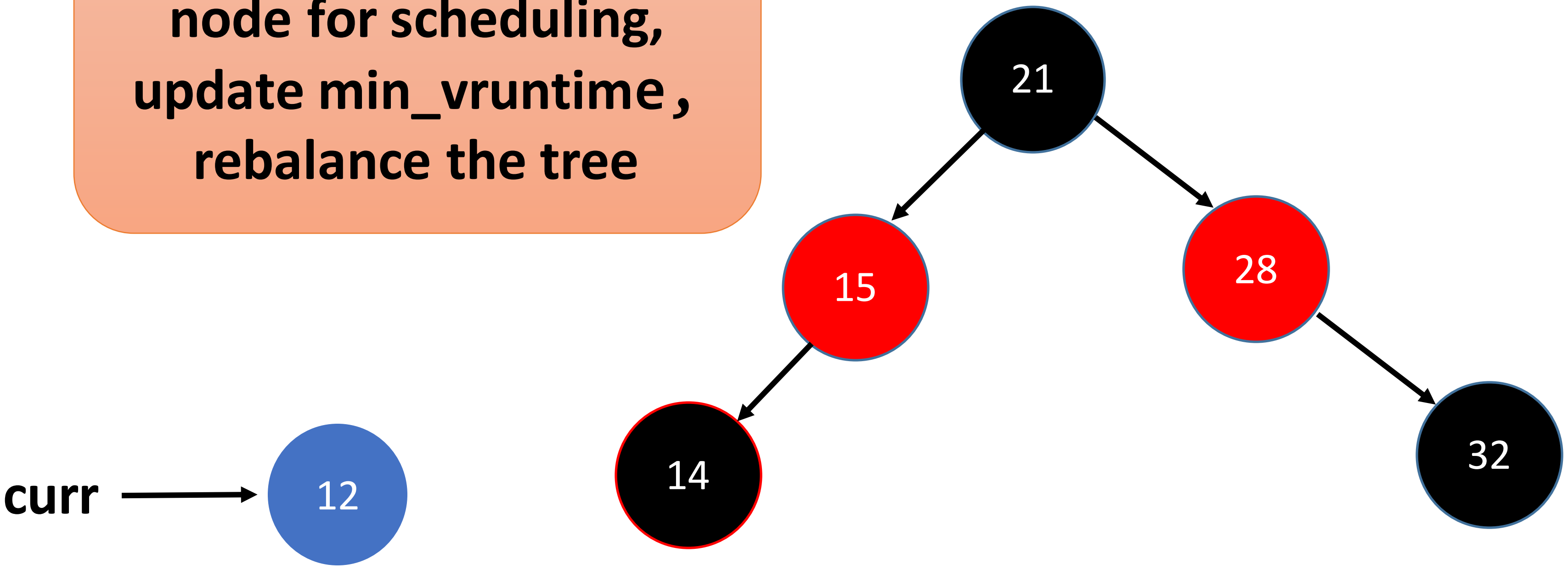
More need of the CPU

More CPU-time

Less need of the CPU

CFS in Action

Extract the leftmost node for scheduling, update min_vruntime , rebalance the tree



Less CPU-time
More need of the CPU

More CPU-time
Less need of the CPU

CFS in Action – Core Premitives

- When a task is executing, its vruntime increases, so it moves to the right in the red-black tree

CFS in Action – Core Premitives

- When a task is executing, its vruntime increases, so it moves to the right in the red-black tree
  - **Interactive processes**: Have small CPU burst -> low vruntime
    - Remains at the left side of the RB tree, gets scheduled quickly
    - Have higher priority, virtual clock ticks slowly – spends more time in the CPU whenever needed

CFS in Action – Core Premitives

- When a task is executing, its vruntime increases, so it moves to the right in the red-black tree

- Virtual clock ticks slowly for higher priority tasks, so they move slower to the right of the RB tree, and their chance to be scheduled again soon is bigger than lower priority tasks

CFS in Action – Core Premitives

- When a task is executing, its vruntime increases, so it moves to the right in the red-black tree
- Virtual clock ticks more slowly for higher priority tasks, so they move slower to the right of the RB tree, and their chance to be scheduled again soon is bigger than lower priority tasks
- When a process sleeps, its vruntime remains unchanged.

CFS in Action – Core Primitives

- When a task is executing, its vruntime increases, so it moves to the right in the red-black tree
- Virtual clock ticks more slowly for higher priority tasks, so they move slower to the right of the RB tree, and their chance to be scheduled again soon is bigger than lower priority tasks
- When a process sleeps, its vruntime remains unchanged
- New processes start with `min_vruntime` of the RB tree, to allow them to get scheduled quickly

Happy Learning