Scheduling in Linux – Part 2

Acknowledgement

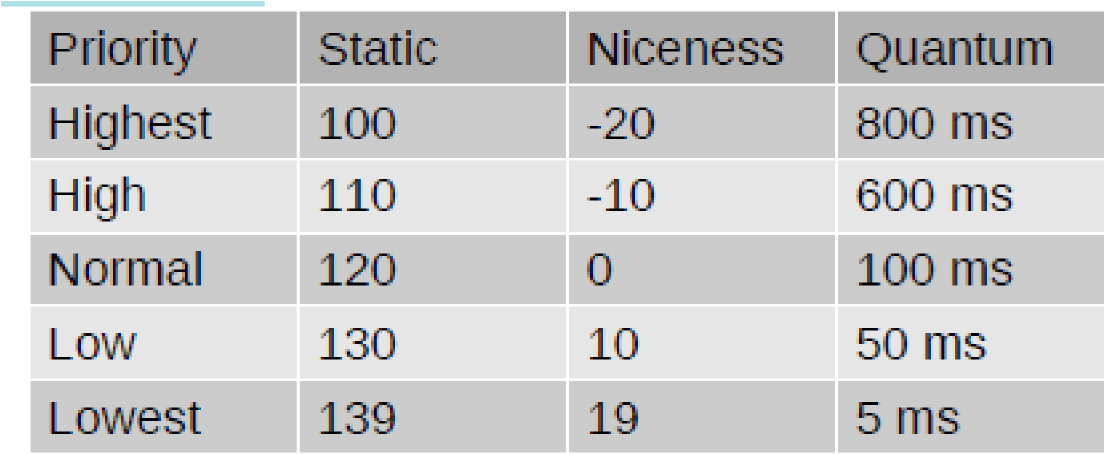
The example of CFS is borrowed from the slides of the same course offered by Prof. Sandip Chakraborty in earlier years (very slight changes done)

The materials for some of the other slides are borrowed from the same source

Scheduling SCHED_NORMAL class

So what was wrong with O(1)?

- Timeslice allocations across priorities were disproportionate, huge difference in allocated timeslices



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

- Why is this a problem?

- Low priority tasks cause frequent context switches, even if there are no other processes
  - Suppose that there are two processes with priority 130, will cause context switches every 50 millisecond unnecessarily
- High priority batch tasks can cause interactive tasks to suffer
  - Suppose that there are two batch processes with priority 110, interactive jobs will not get a chance to run for long
  - Dynamic priority increase will still take time to catch up
- Fixed timeslice based on priority is not good
  - Ignored the current load on the CPU

Completely fair Scheduler (CFS)

- Introduced in Kernel version 2.6.23 (2007)
- Default scheduler for a new task
- Major Idea
  - To select the task to run
    - Choose a task that has used the CPU less so far
  - To decide the timeslice
    - Calculate how long a task should run as a function of the total number of currently runnable processes and their priorities
    - So no fixed timeslice, depends on other tasks in the runqueue
      - Trying to be fair to everyone

Selecting a Task to Run

- Consider two processes, a text editor and a simulation job
  - Ideal proportion of CPU: 50%
  - Text editor will not use its 50% always
    - But will need the CPU immediately when it wants
    - Will use it for a short time and then wait again
  - Simulation job can use more than 50% when the text editor is not using it
    - But must relinquish immediately whenever text editor wants it
- *CFS* Idea
    - Allocate the CPU to a process which has used it less so far
    - So the text editor will get scheduled as soon as it wants the CPU

- But a simple implementation does not take care of priorities
- So weight the runtime with the priority
- Keep track of *virtual runtime* (not exact physical runtime) of each process
  - At every scheduling tick, if a process has run for p milliseconds, set

    vruntime += p*(weight of the process)
  - Weight increases with nice value of a process
- At any point of time, choose the process with the smallest *vruntime*
- Processes with higher nice values have faster increase in *vruntime*, therefore are chosen later (lower priority as it should be) and vice-versa
- When a process sleeps, its *vruntime* remains unchanged.

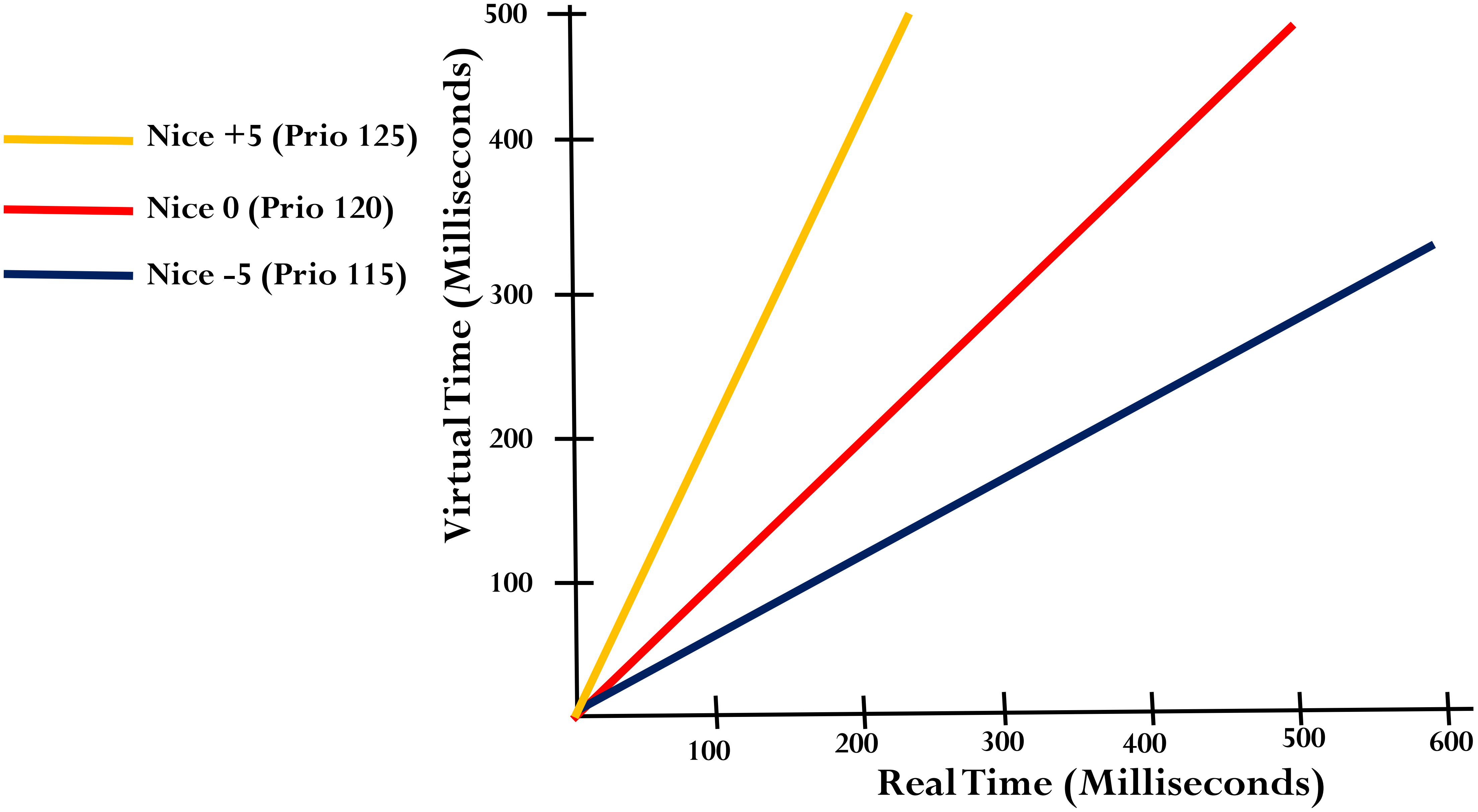
- Weight for each nice value is defined statically

```
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,
};
```

- What is the weight of a process used?
  - (weight for nice value 0)/(weight for nice value of the process)

    = 1024/(weight for nice value of the process)

- Why are the weights like this
  - Ensures that a nice value difference of 1 causes around 10% difference in CPU share
  - Example: Consider two processes A and B at nice 0 and nice 1
    - Share of A = 1024/(1024+820) = 55%
    - Share of B = 820/(1024+820) = 45%
  - Another example: A and B at nice 0 and 2
    - Share of A = 1024/(1024+655) = 61%
    - Share of B = 655/(1024+820) = 39%



Nice +5 (Prio 125)
Nice 0 (Prio 120)
Nice -5 (Prio 115)
Virtual Time (Milliseconds)
500
400
300
200
100
100
200
300
400
500
600
Real Time (Milliseconds)

Choosing the Timeslice

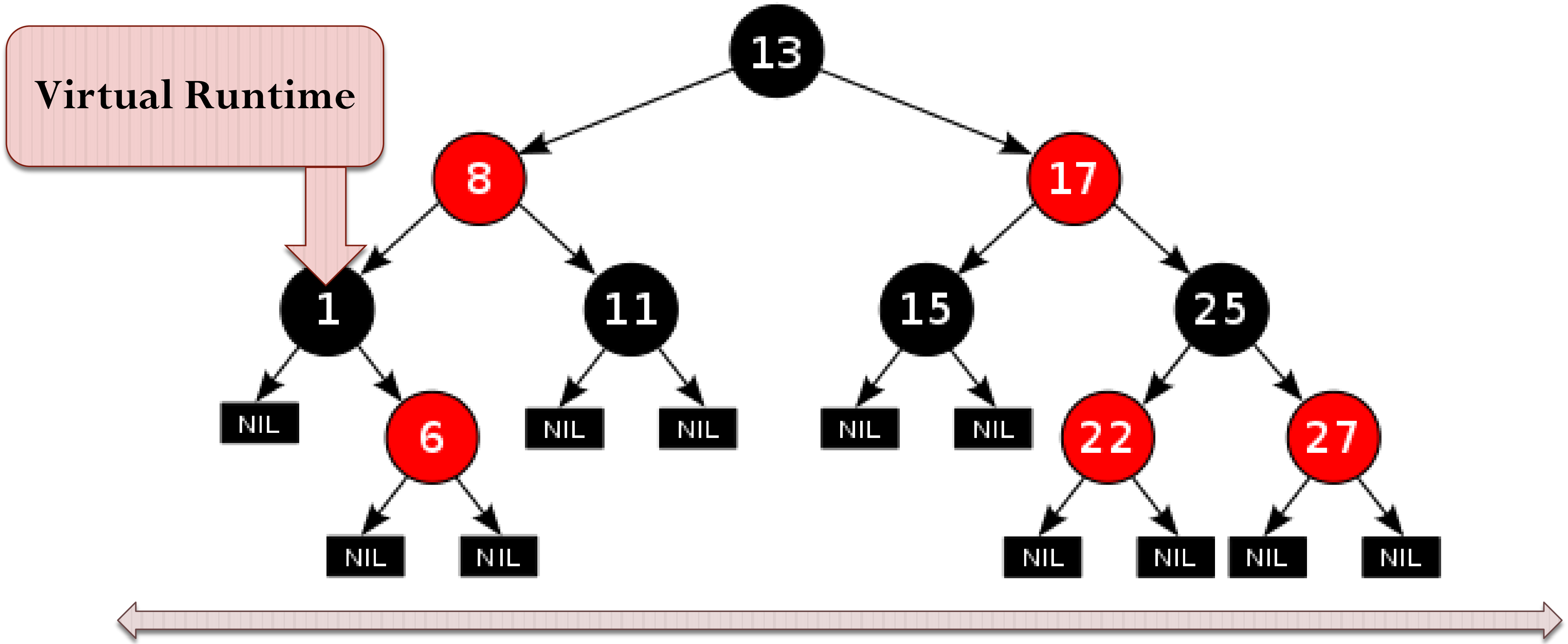
- Calculate how long a task should run as a function of the total number of currently runnable process
  - Run the process for a time slice proportional to its weight divided by the weight of all other runnable processes
  - Use the priority value in the weight to ensure that a higher priority job gets more CPU time proportional to the priority of the other processes in the runqueue
- *Target Latency*
  - A time set by CFS within which it will schedule all runnable processes
    - This is the period whose proportion the processes are getting
  - Default is 20 milliseconds

- Processes at same priority
  - If target latency is T milliseconds and there are N processes, each gets T/N milliseconds
  - Example: the targeted latency is 20 milliseconds and
    - 2 runnable tasks, each will run for 10 milliseconds
    - 4 runnable tasks, each will run for 5 milliseconds
    - 20 runnable tasks, each will run for 1 millisecond.
- What if N becomes very large?
  - Timeslice is too small, context switch will overwhelm the actual running time
  - CFS sets lower limit, called minimum granularity (default 1 millisecond)
    - If timeslice goes below this, target latency is increased dynamically

- Processes at different priority;
  - 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 (table shown already)
      - 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

Implementation Issues

- The runqueue is maintained as a single Red-Black tree organized with the virtual runtimes
  - Leftmost node gives the next process to run (O(log n))
- So processes move from left to right of the tree as they execute
  - Higher priority processes move slower than lower priority process, increasing their chance to be rescheduled sooner
- When are new processes inserted into the tree?
  - When a new process is created
  - When a process becomes runnable
- With what initial *vruntime*?
  - The maximum of the minimum *vruntimes* seen so far (will see later what this means)



Virtual Runtime

Less CPU-time

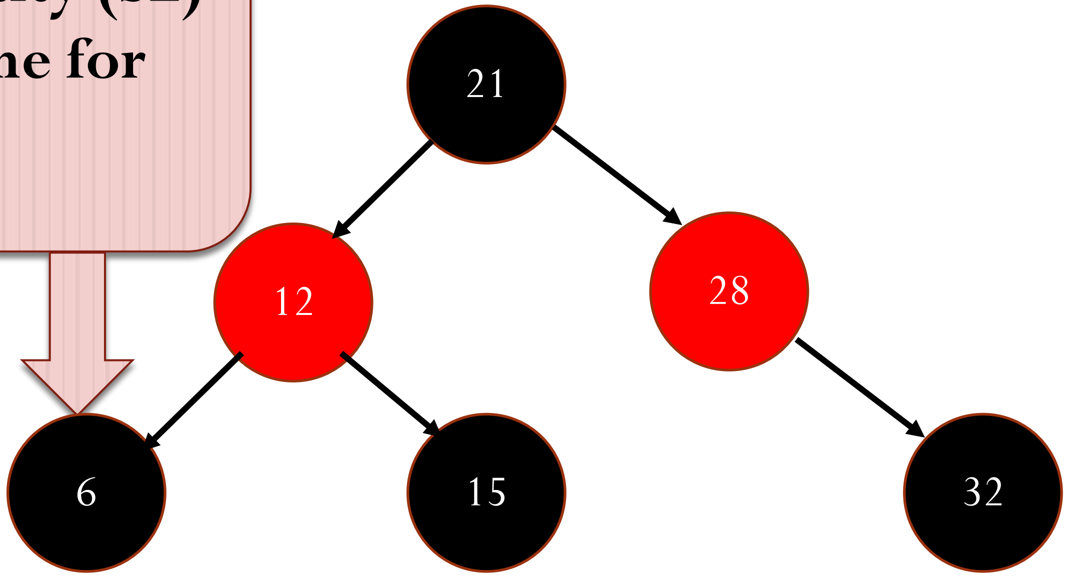
More need of the CPU

More CPU-time

Less need of the CPU

CFS in Action

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



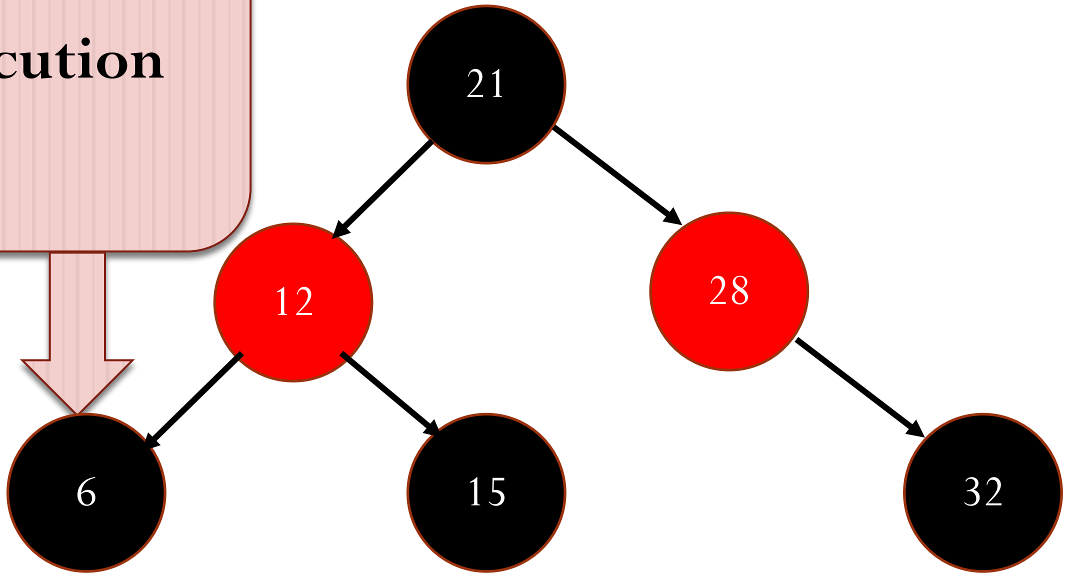
Less CPU-time

More need of the CPU

More CPU-time

Less need of the CPU

Dequeue the SE for execution



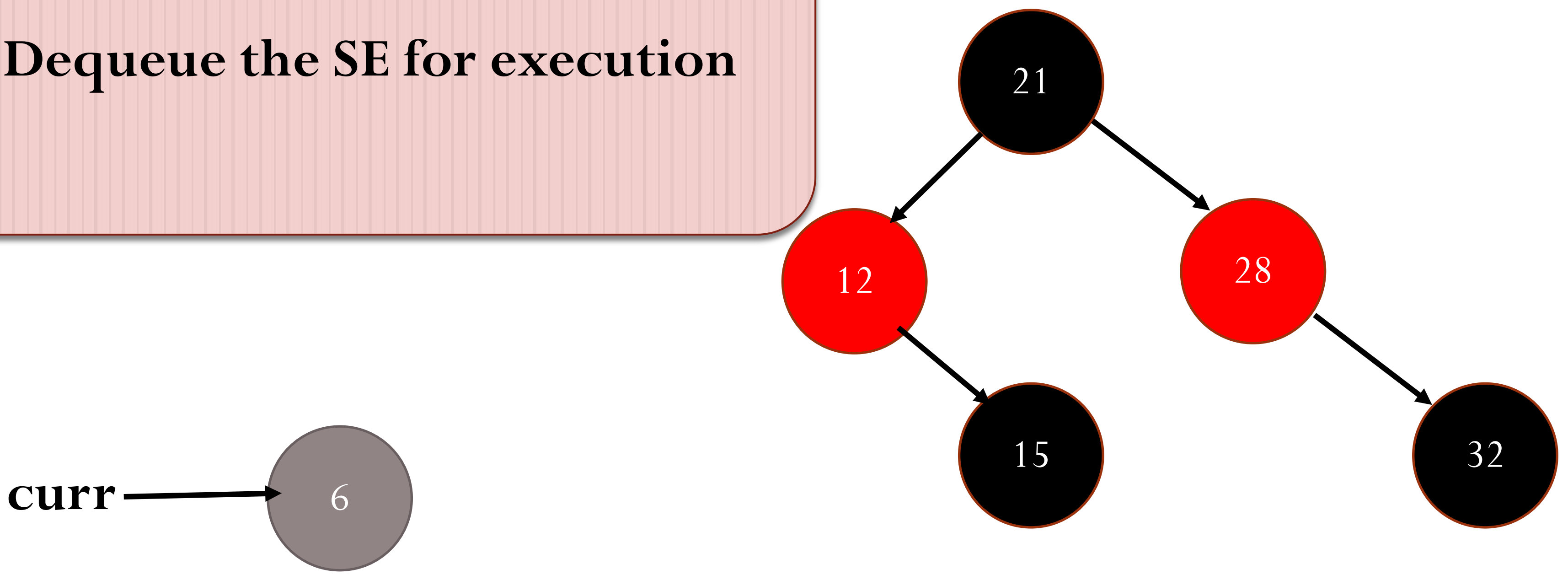
Less CPU-time

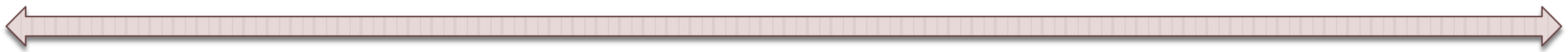
More need of the CPU

More CPU-time

Less need of the CPU

Dequeue the SE for execution





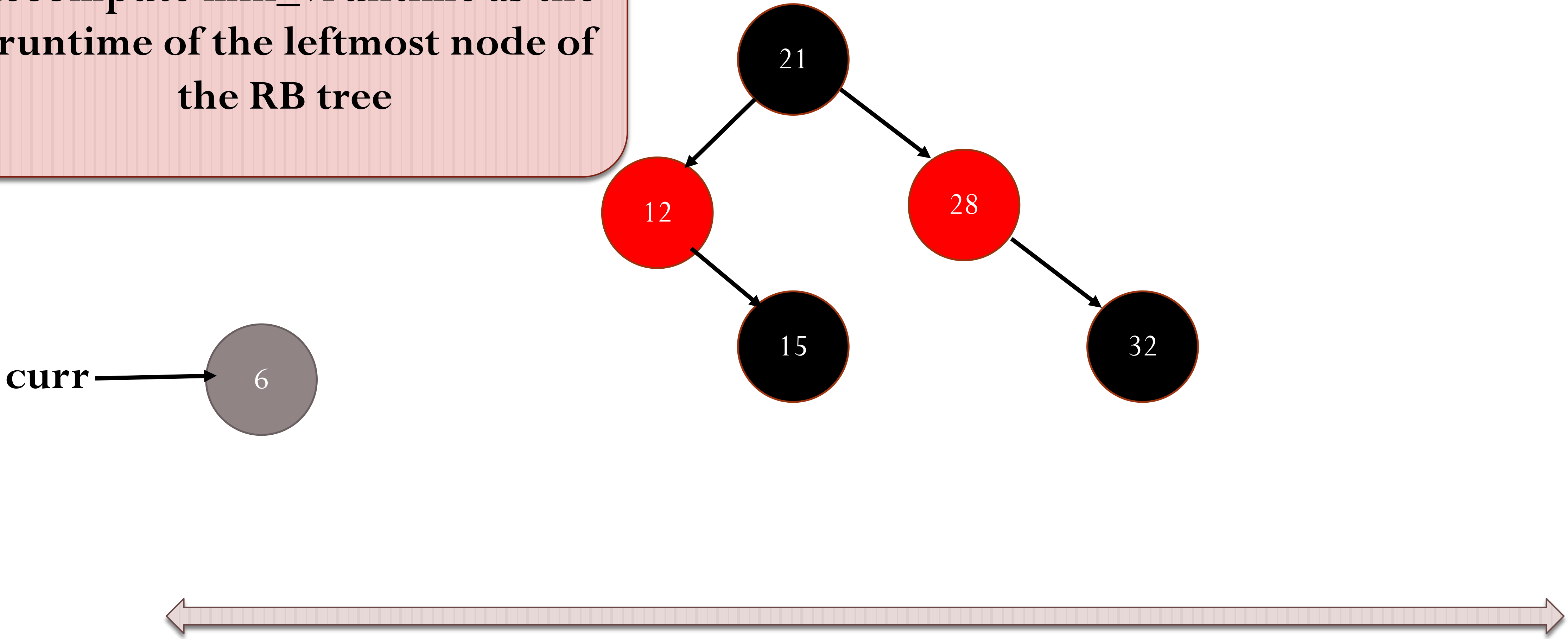
Less CPU-time

More need of the CPU

More CPU-time

Less need of the CPU

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



Less CPU-time

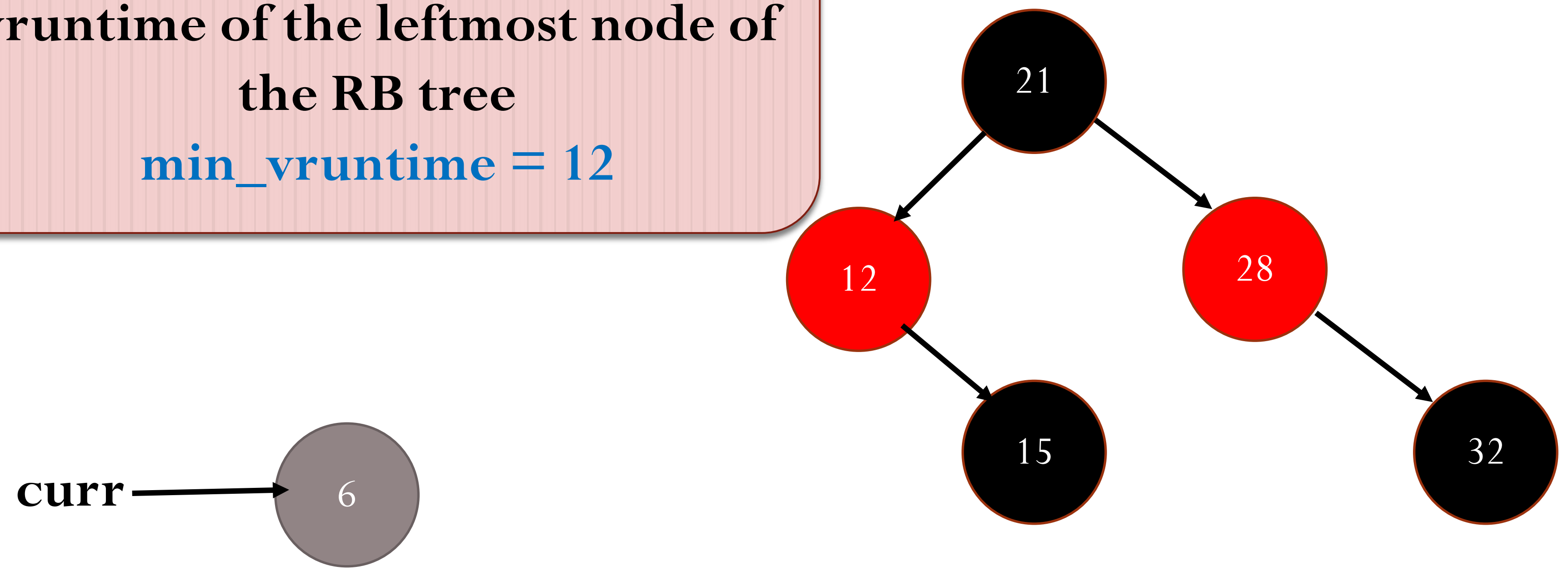
More need of the CPU

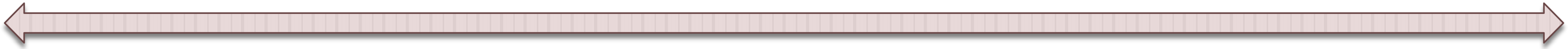
More CPU-time

Less need of the CPU

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

min_vruntime = 12





Less CPU-time

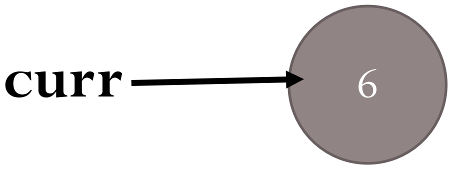
More need of the CPU

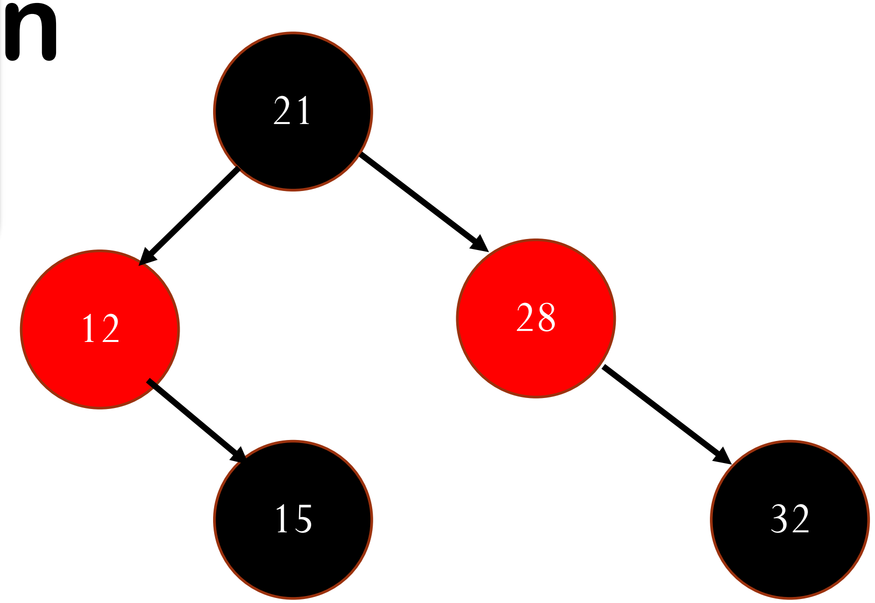
More CPU-time

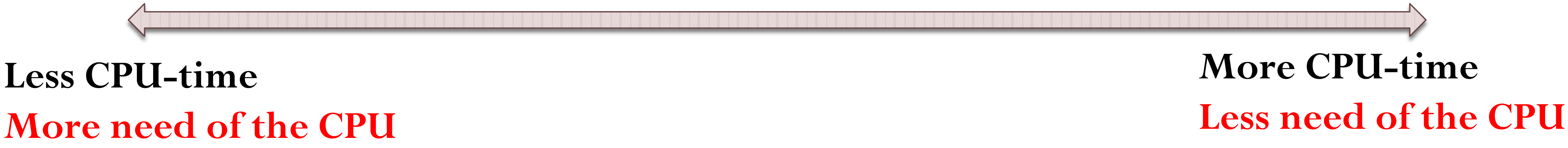
Less need of the CPU

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

n

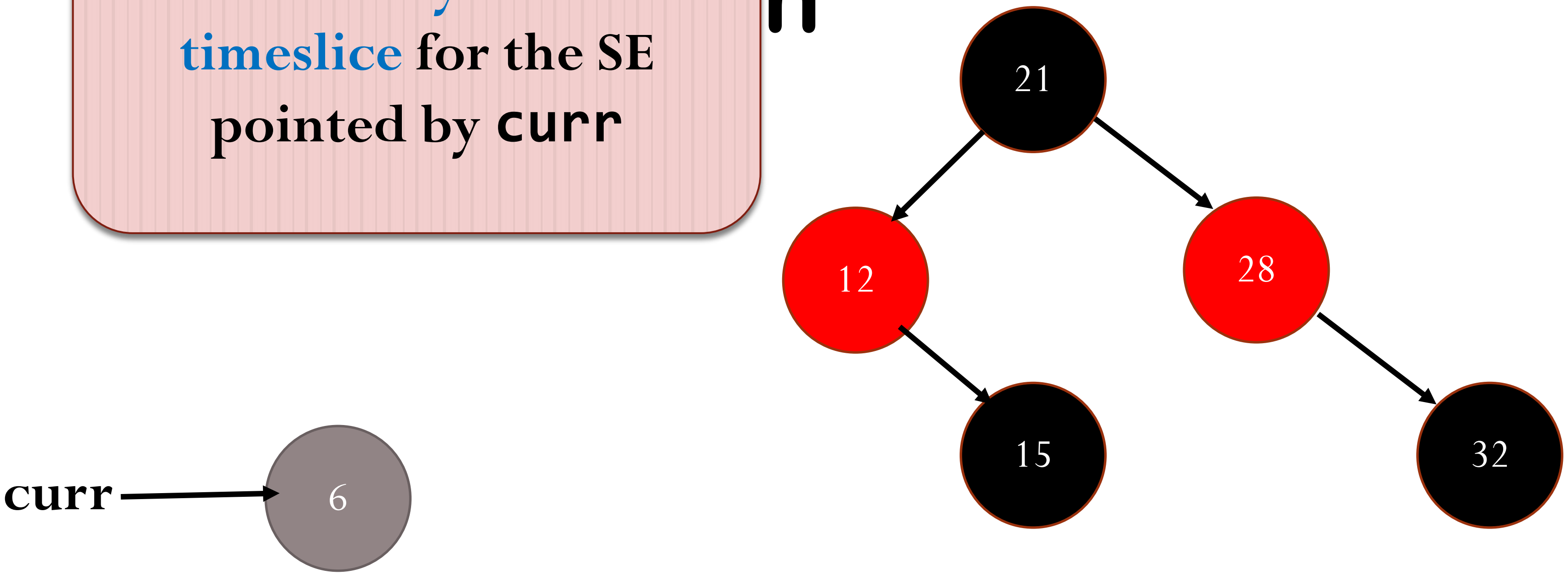




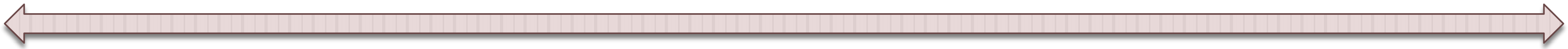


Set the dynamic timeslice for the SE pointed by `curr`

n



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



Less CPU-time

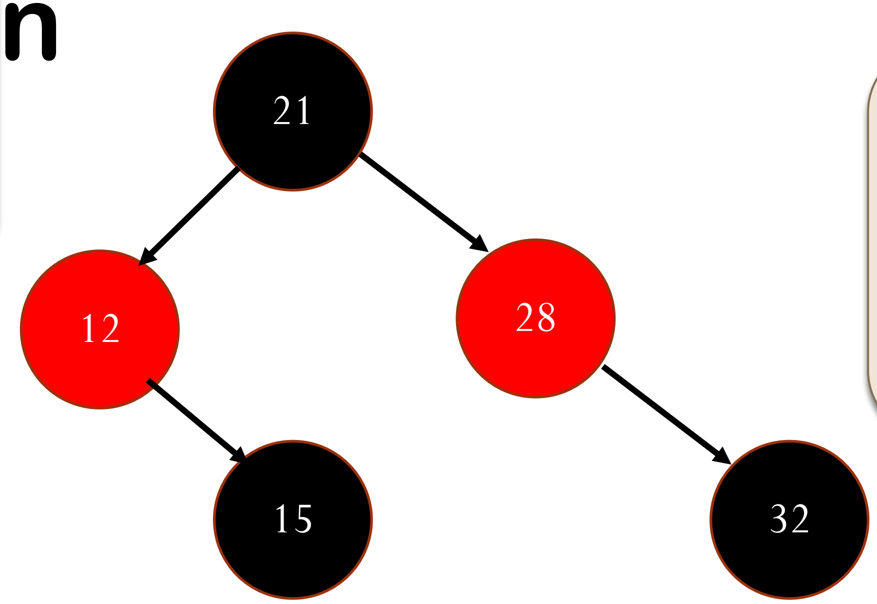
More need of the CPU

More CPU-time

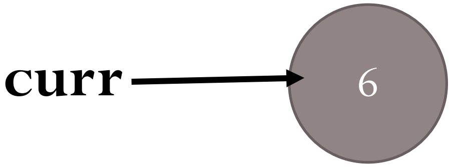
Less need of the CPU

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

n



Remember, that the `sched_period` is dynamic



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

Less CPU-time

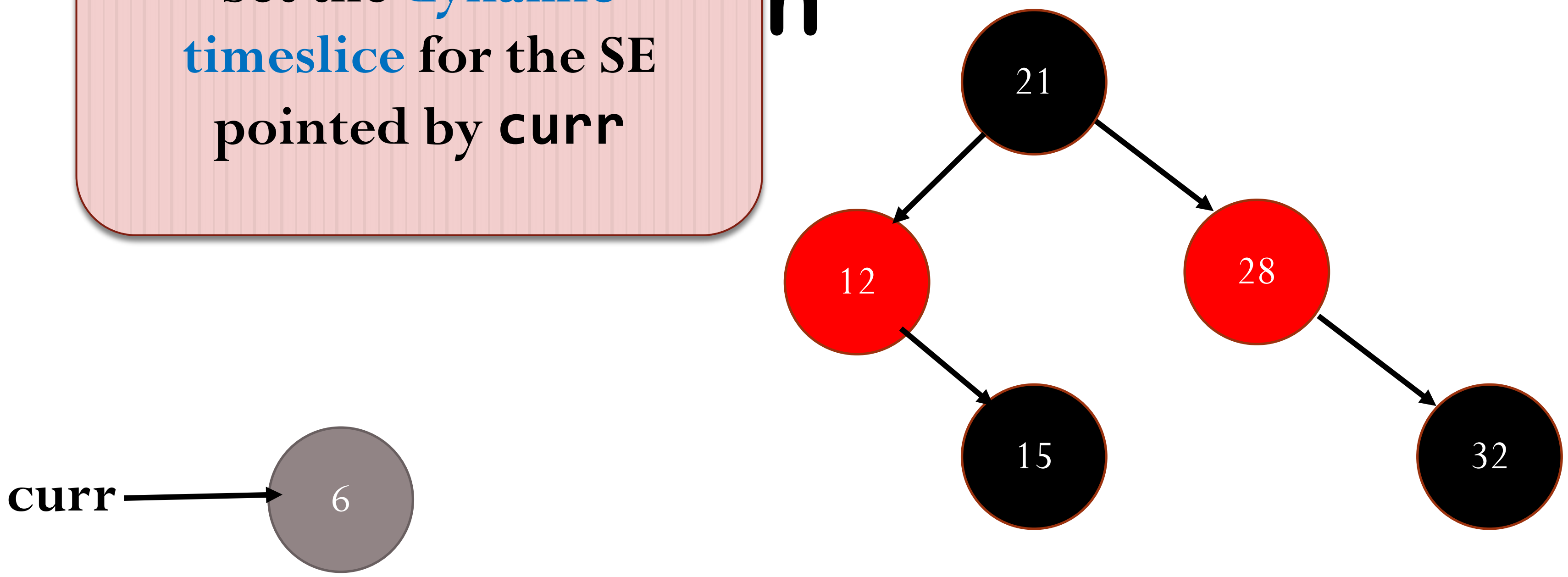
More need of the CPU

More CPU-time

Less need of the CPU

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

n



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

Less CPU-time

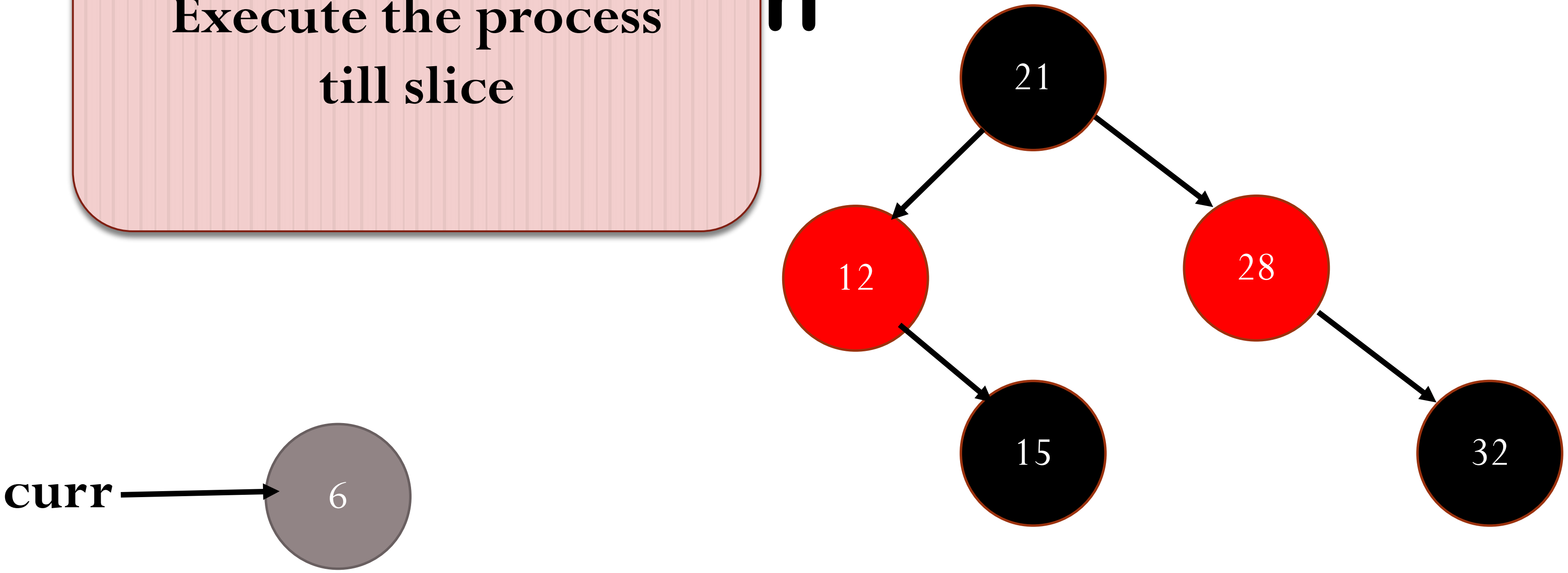
More need of the CPU

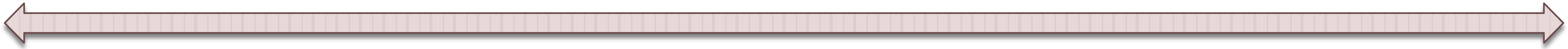
More CPU-time

Less need of the CPU

n

Execute the process till slice





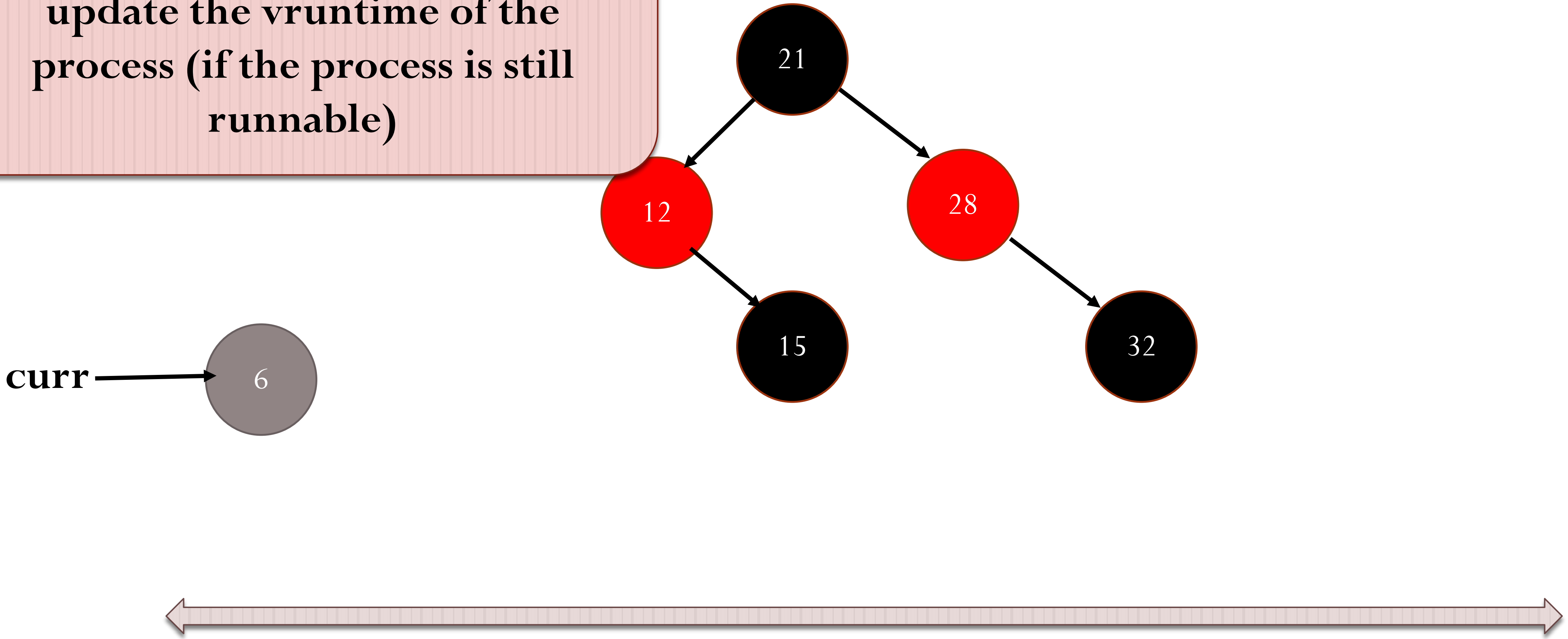
Less CPU-time

More need of the CPU

More CPU-time

Less need of the CPU

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



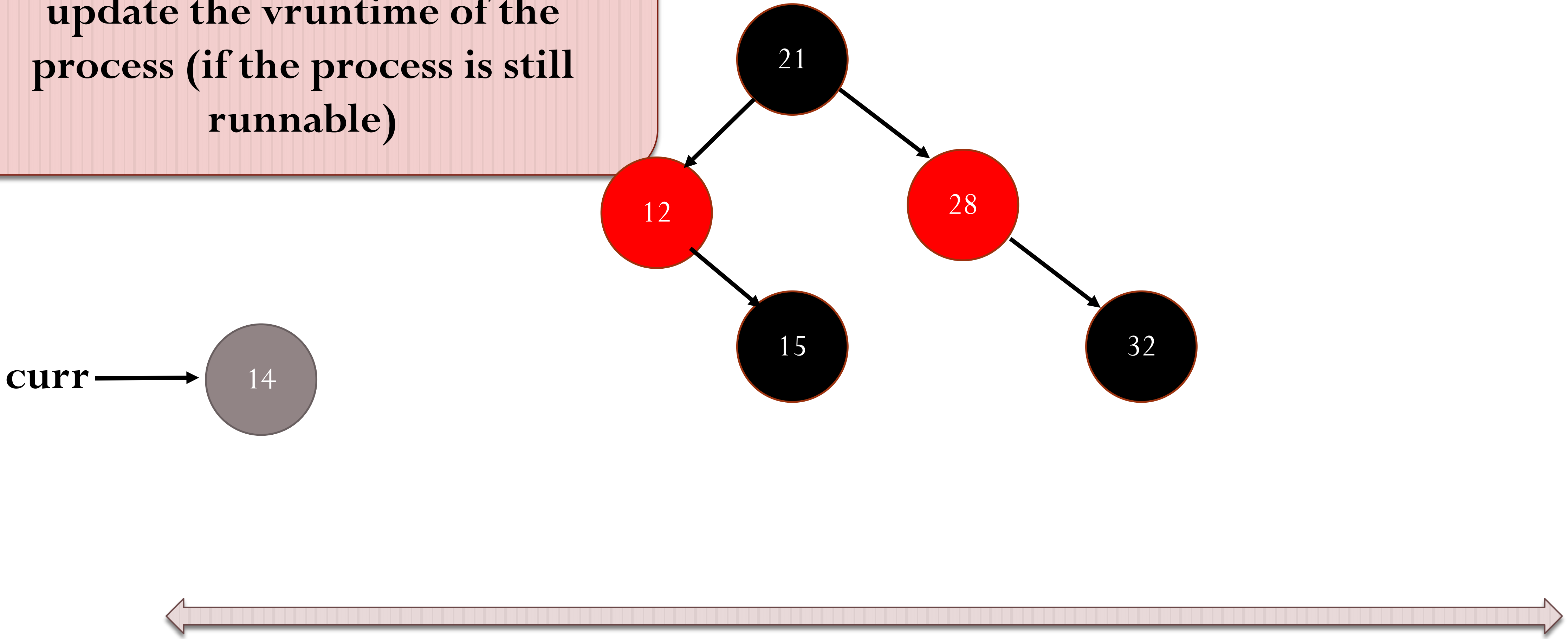
Less CPU-time

More need of the CPU

More CPU-time

Less need of the CPU

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



Less CPU-time

More need of the CPU

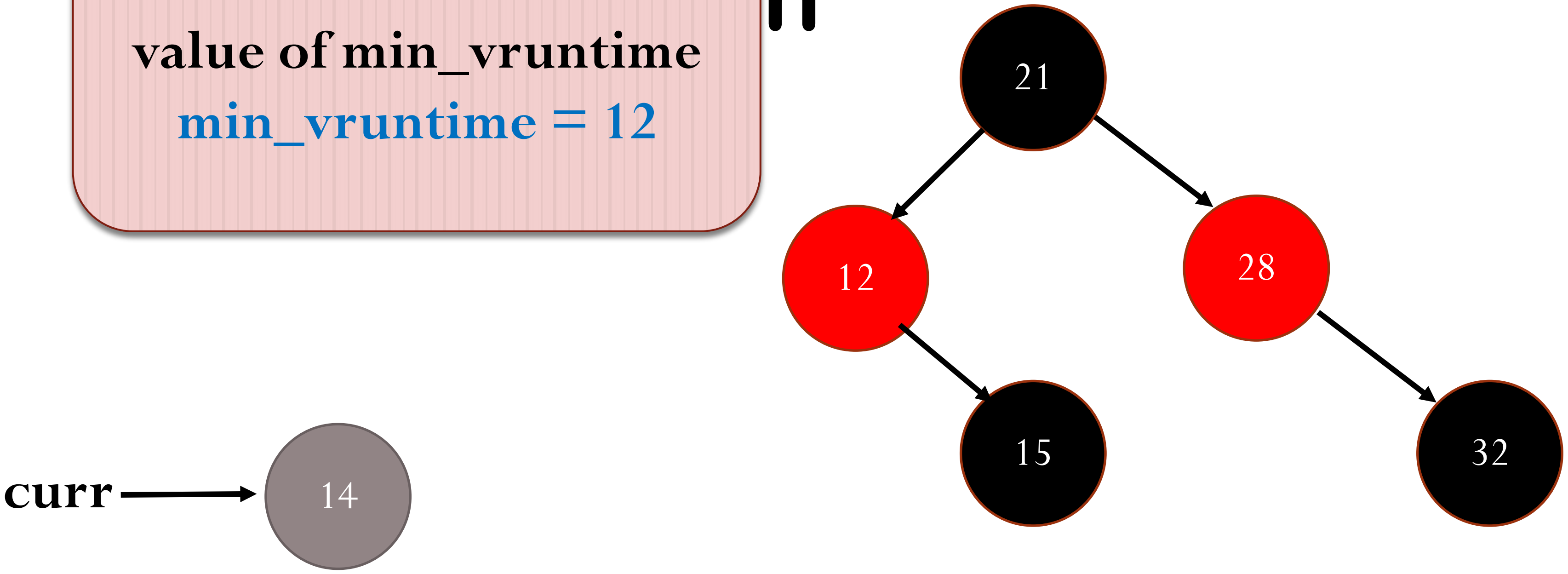
More CPU-time

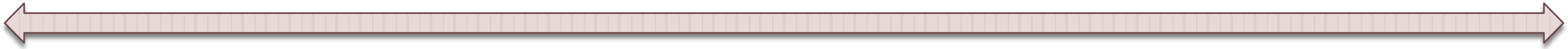
Less need of the CPU

n

Check with the cached value of min_vruntime

min_vruntime = 12





Less CPU-time

More need of the CPU

More CPU-time

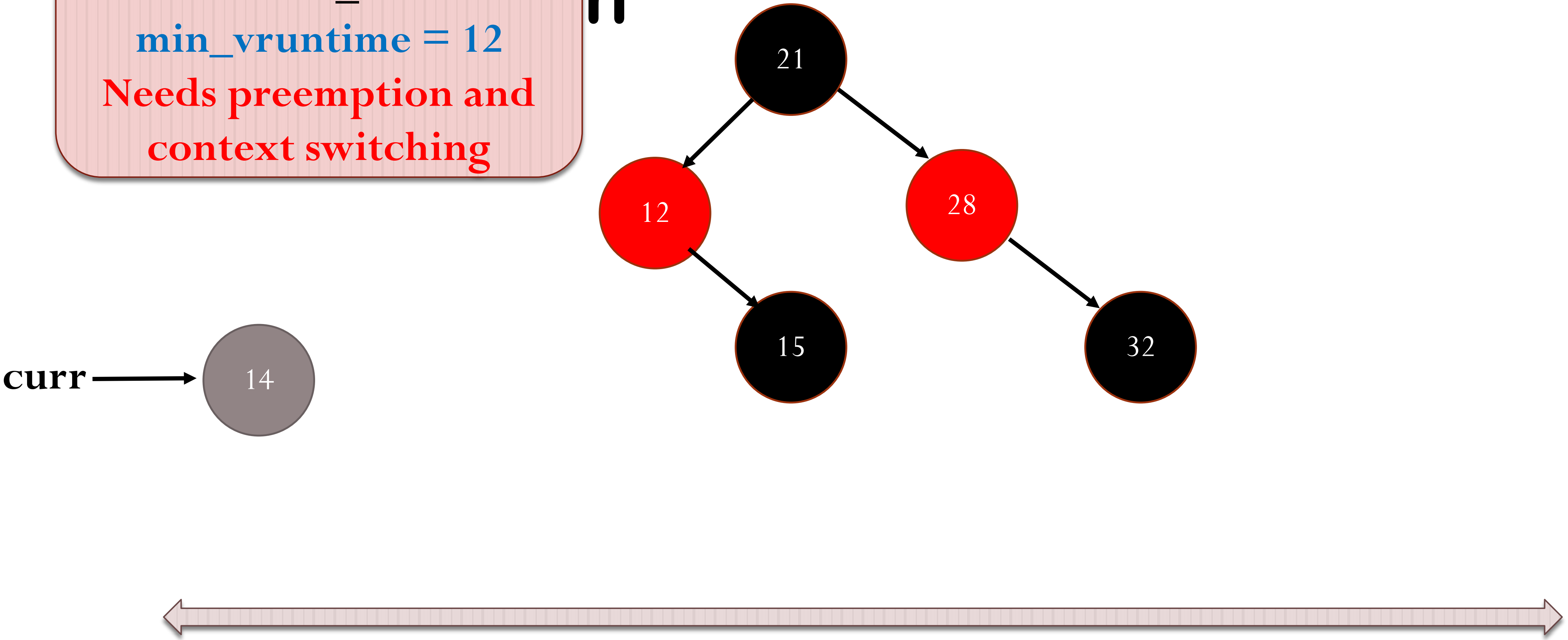
Less need of the CPU

Check with the cached value of min_vruntime

min_vruntime = 12

Needs preemption and context switching

n



Less CPU-time

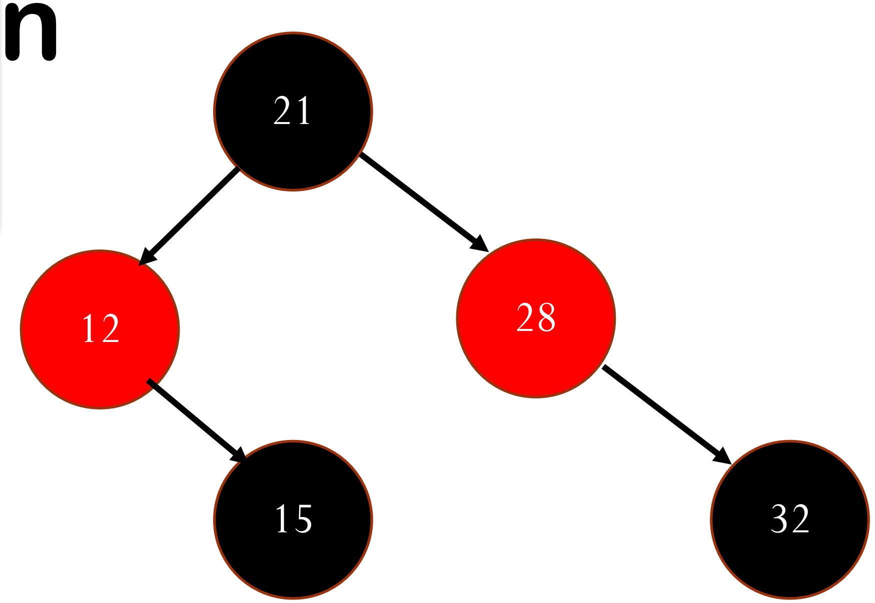
More need of the CPU

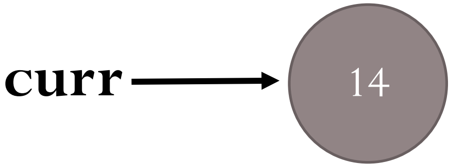
More CPU-time

Less need of the CPU

Insert the SE in the RB tree with the updated vruntime

n





Less CPU-time

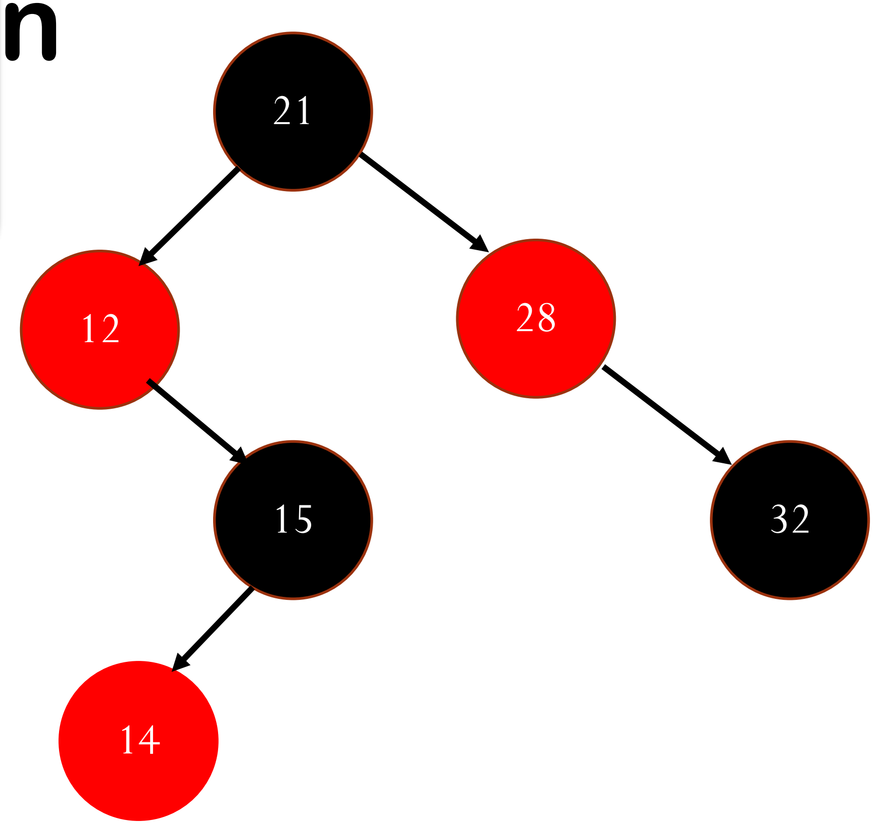
More need of the CPU

More CPU-time

Less need of the CPU

Insert the SE in the RB tree with the updated vruntime

n



curr →

Less CPU-time

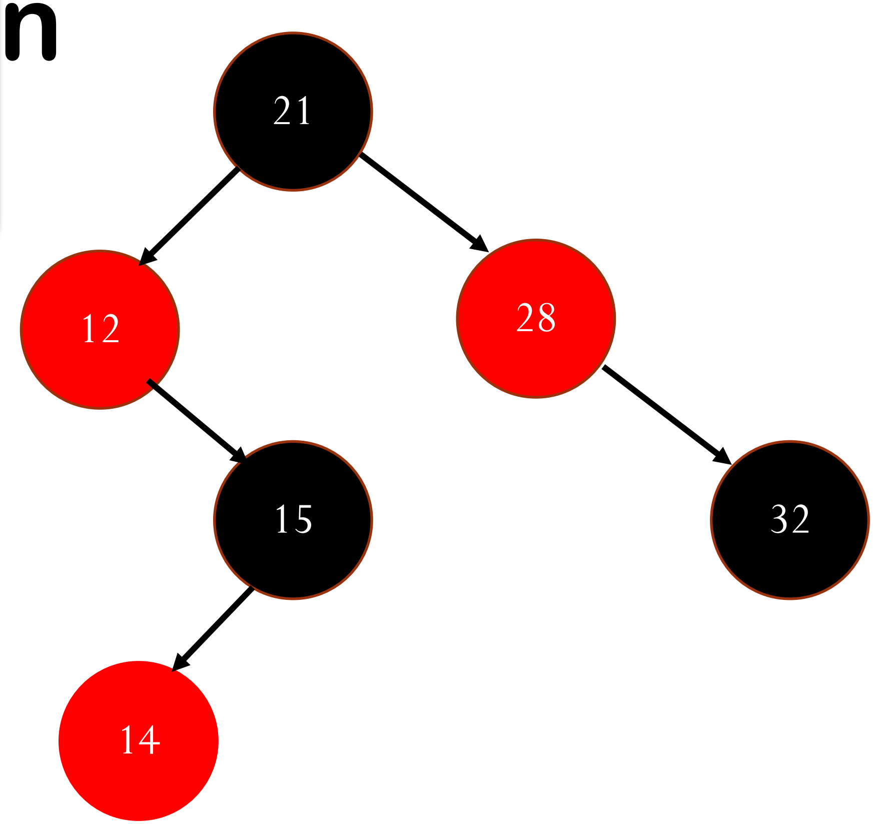
More need of the CPU

More CPU-time

Less need of the CPU

Insert the SE in the RB tree with the updated vruntime

n



curr →

Less CPU-time

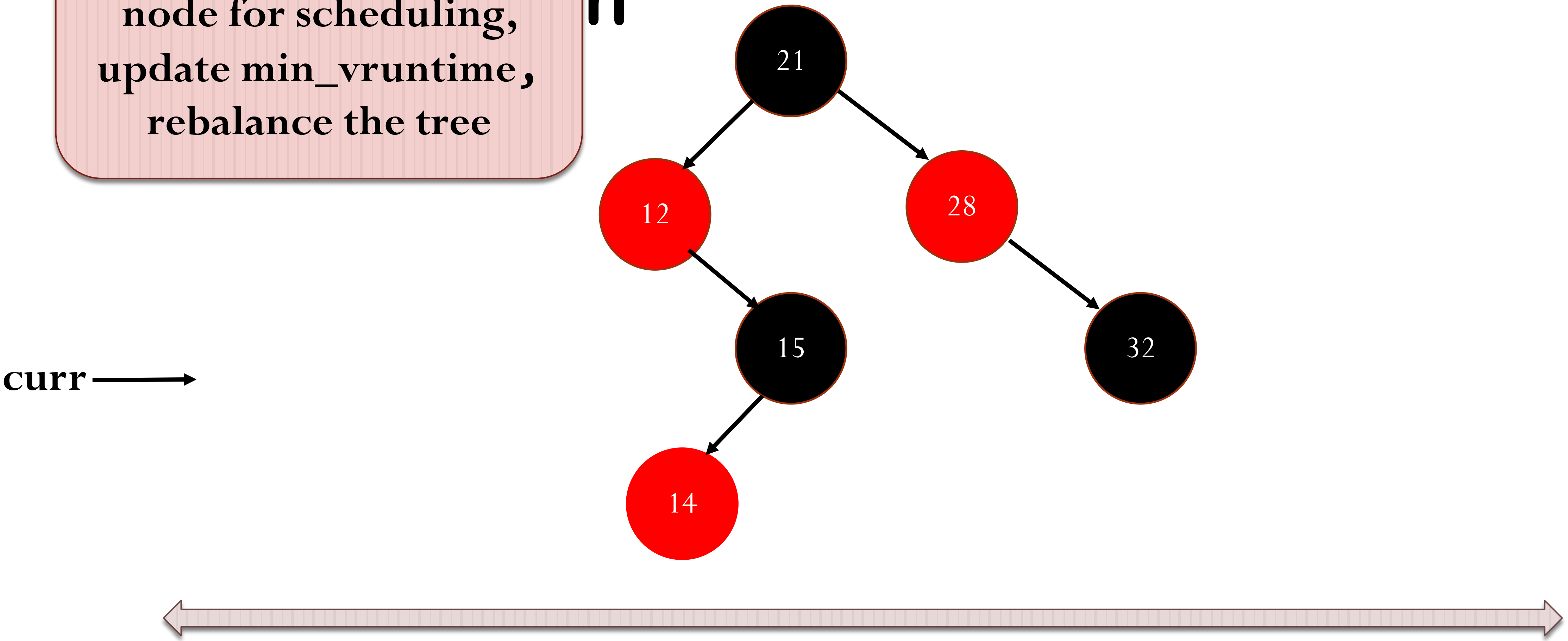
More need of the CPU

More CPU-time

Less need of the CPU

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

n



curr

Less CPU-time

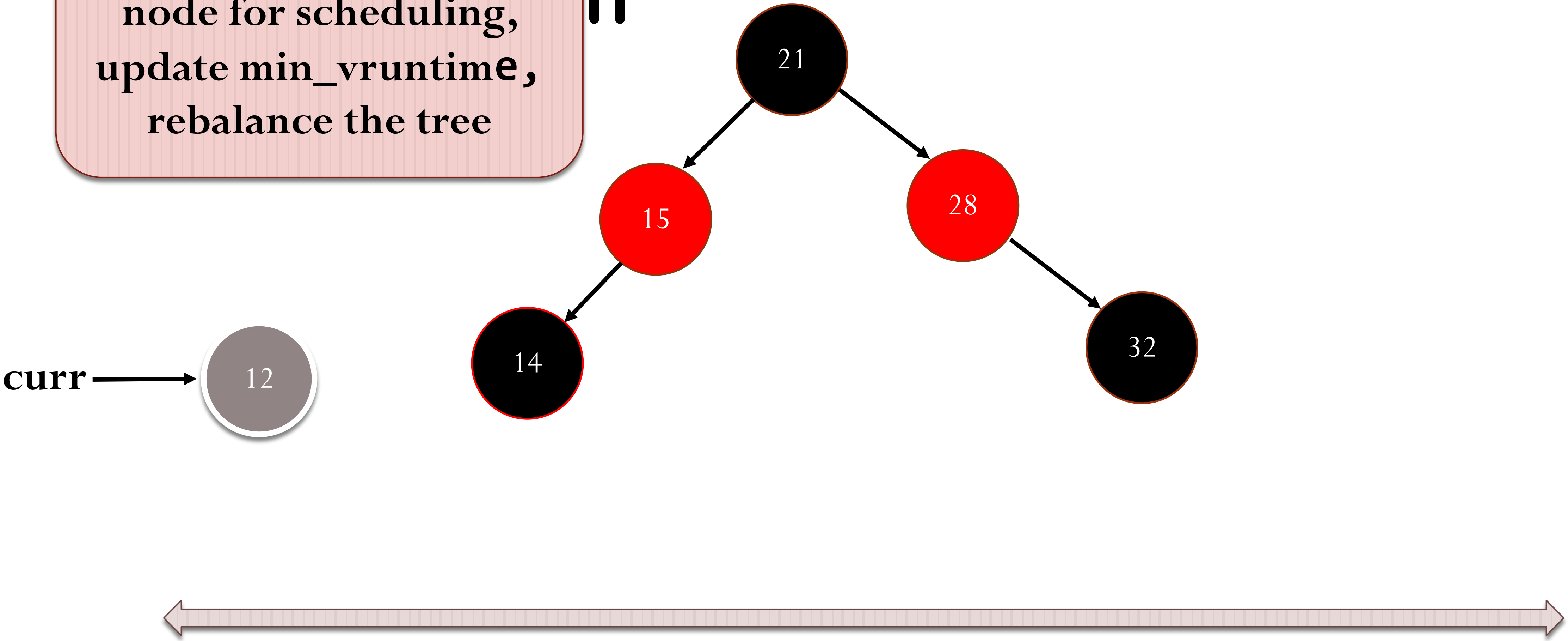
More need of the CPU

More CPU-time

Less need of the CPU

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

n



Less CPU-time

More need of the CPU

More CPU-time

Less need of the CPU

Behavior of Types of Tasks with CFS

- Interactive Tasks
  - Uses less CPU time, so *vruntime* stays low, so stays more on left side of the tree
    - Scheduled again earlier
- Batch Tasks
  - Uses more CPU time, so *vruntime* is high, so moves more to the right side of the tree
    - Scheduled later
- So CFS favors interactive tasks

Group Scheduling

- Consider that you have 2 processes initially
  - So each gets 50% CPU
- Now the first task spawns 100 threads
  - Total 102 tasks, CPU is shared between them
  - So second process gets very little CPU, not fair
- CFS allows Group Scheduling for such cases
  - A set of tasks are scheduled as a group
  - CPU allocation is fair between groups
- We will not look at this in this course

- We will look at Linux implementation of CFS and associated routines next