

# Solaris and Linux scheduling

## Module 3

Self study material

---

**Operating systems 2019**

**1DT044, 1DT096 and 1DT003**



**solaris™**

# **Scheduling**

Solaris is a Unix operating system originally developed by Sun Microsystems. It superseded their earlier SunOS in 1993. Oracle Solaris, as it is now known, has been owned by Oracle Corporation since Oracle's acquisition of Sun in January 2010

# Multi-level feedback queue

- ★ Each process has a **dynamic priority**, an integer value.
- ★ Uses a **multi-level feedback** queue scheduler.
  - The queues in the system are ranked according to priority. Processes waiting in higher priority queues are always scheduled over those in lower priority queues.
  - Processes at the same priority are usually scheduled in a round-robin fashion.
  - Whenever the CPU is free, the scheduler dispatches the processes at the head of the highest-priority nonempty ready queue.
- ★ A **dispatch table** is used to **dynamically** compute and **update** the **priority** of a process.

# Multi-level feedback queue

- ★ A process begins at priority 29.
- ★ CPU bound processes then filter down to the lower priorities, where they are scheduled less frequently (but for longer time-slices).
- ★ Interactive processes propagate to the higher priorities (where they are scheduled whenever they have work to perform, on the assumption that they will soon relinquish the processor again).
- ★ The priority of a process is lowered after it consumes its allocated time-slice. Its priority is raised if it has not consumed its time-slice before requesting I/O.

# Solaris dispatch table

By default, there is an inverse relationship between priorities and time slices (time quantum).

The higher the priority, the smaller the time slice.

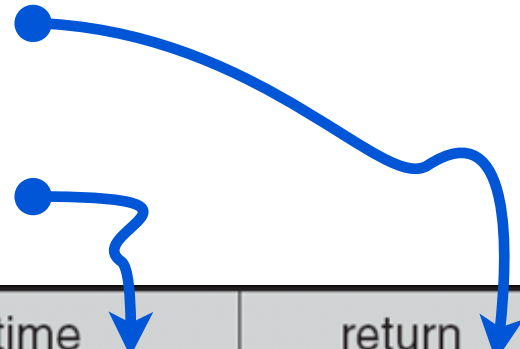
Interactive processes typically have a higher priority: CPU-bound processes, a lower priority.

New priority when returning from sleep, such as waiting for I/O.

New priority when time quantum expires.

low  
priority

higher  
priority



priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

This scheduling policy give good response times for interactive processes and good throughput for CPU-bound processes.



# Schedulers

# Linux 2.6 scheduling

In versions of the Linux kernel 2.6 prior to 2.6.23, the scheduler used is an  **$O(1)$  scheduler** by Ingo Molnár.

The scheduler used thereafter is the **Completely Fair Scheduler**, also by Ingo Molnár.

- Runs in  **$O(\log N)$**  time where  $N$  is the number of tasks in the runqueue.
- Choosing a task can be done in constant time, but reinserting a task after it has run requires  $O(\log N)$  operations, because the runqueue is implemented as a **red-black tree**.
- A red-black tree is a type of **self-balancing binary search tree** data structure.

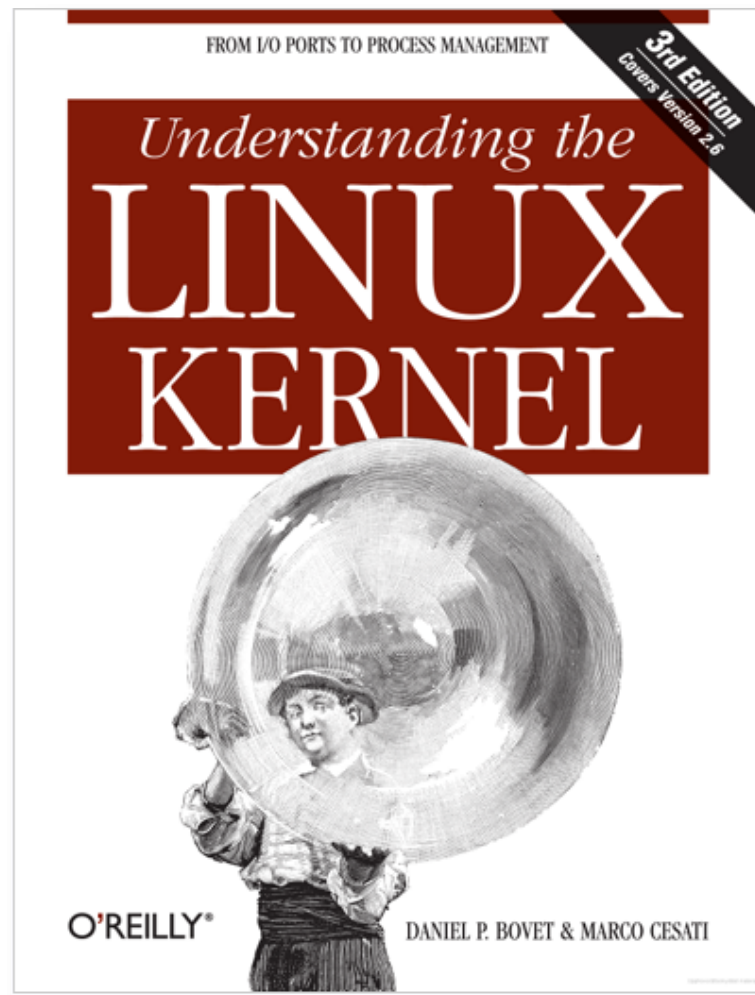


# **O(1) Scheduler**

kernel 2.6 (2003-12-17) — kernel 2.6.22 (2007-07-08)



# Linux O(1) scheduler (prior to 2.6.23)



The scheduling algorithm of Linux 2.6 is much more sophisticated. By design, it scales well with the number of runnable processes, because it selects the process to run in constant time, independently of the number of runnable processes. It also scales well with the number of processors because each CPU has its own queue of runnable processes. Furthermore, the new algorithm does a better job of distinguishing interactive processes and batch processes. As a consequence, users of heavily loaded systems feel that interactive applications are much more responsive in Linux 2.6 than in earlier versions.

# Goals of the $O(1)$ scheduler

Maximise overall CPU utilisation while also maximising interactive performance.

- ★ Implement fully  **$O(1)$  scheduling**. Every algorithm in the new scheduler completes in constant-time, regardless of the number of running processes.
- ★ Implement perfect **SMP scalability**. Each processor has its own locking and individual runqueue.
- ★ Implement improved **SMP affinity**. Attempt to group tasks to a specific CPU and continue to run them there. Only migrate tasks from one CPU to another to resolve imbalances in runqueue sizes.
- ★ Provide **good interactive performance**. Even during considerable system load, the system should react and schedule interactive tasks immediately.
- ★ Provide **fairness**. No process should find itself starved of timeslice for any reasonable amount of time. Likewise, no process should receive an unfairly high amount of timeslice.
- ★ Optimize for the common case of only one or two runnable processes, yet scale well to multiple processors, each with many processes.

# Priority levels and values

High  
priority  
level



Low  
priority  
value

Low  
priority  
level



High  
priority  
value

# **Interactivity and dynamic priority**

The Linux scheduler favours interactive processes.

## **Interactivity heuristic**

A task's interactivity is determined by how long it has been suspended waiting for I/O.

## **Dynamic priority**

A task's priority is recalculated when the task has exhausted its time quantum.

# Rewarding I/O bound processes



To prevent tasks from hogging the CPU and thus starving other tasks that need CPU access, the Linux 2.6 scheduler can **dynamically alter a task's priority**.



Because **I/O-bound** tasks are viewed as **altruistic** for CPU access they are **rewarded** by having their **priority level raised** by up to five steps.

I/O-bound tasks commonly use the CPU to set up I/O and then sleep awaiting the completion of the I/O. This type of behaviour gives other tasks access to the CPU.

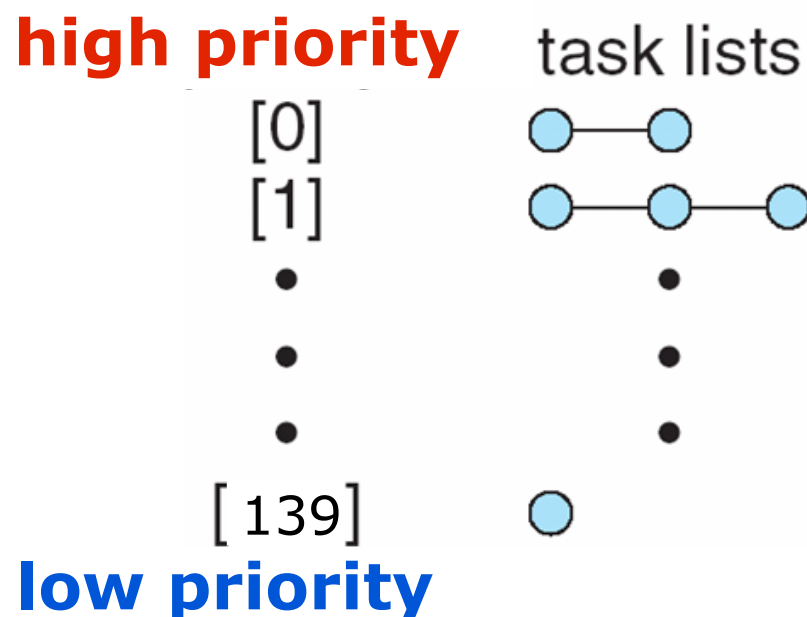


**CPU-bound** tasks are **punished** by having their **priority level lowered** by up to five steps.

# O(1) runqueue data structure

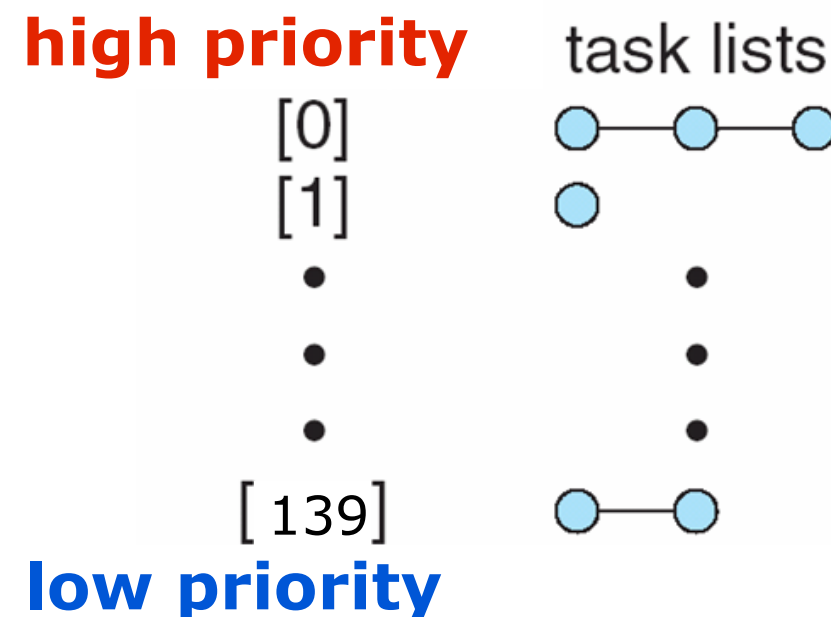
In order to support SMP, each processor maintains its runqueue and schedules itself independently. Each runqueue contains two priority arrays: active and expired.

## active array



These runnable processes have not yet exhausted their time quantum and are thus allowed to run.

## expired array

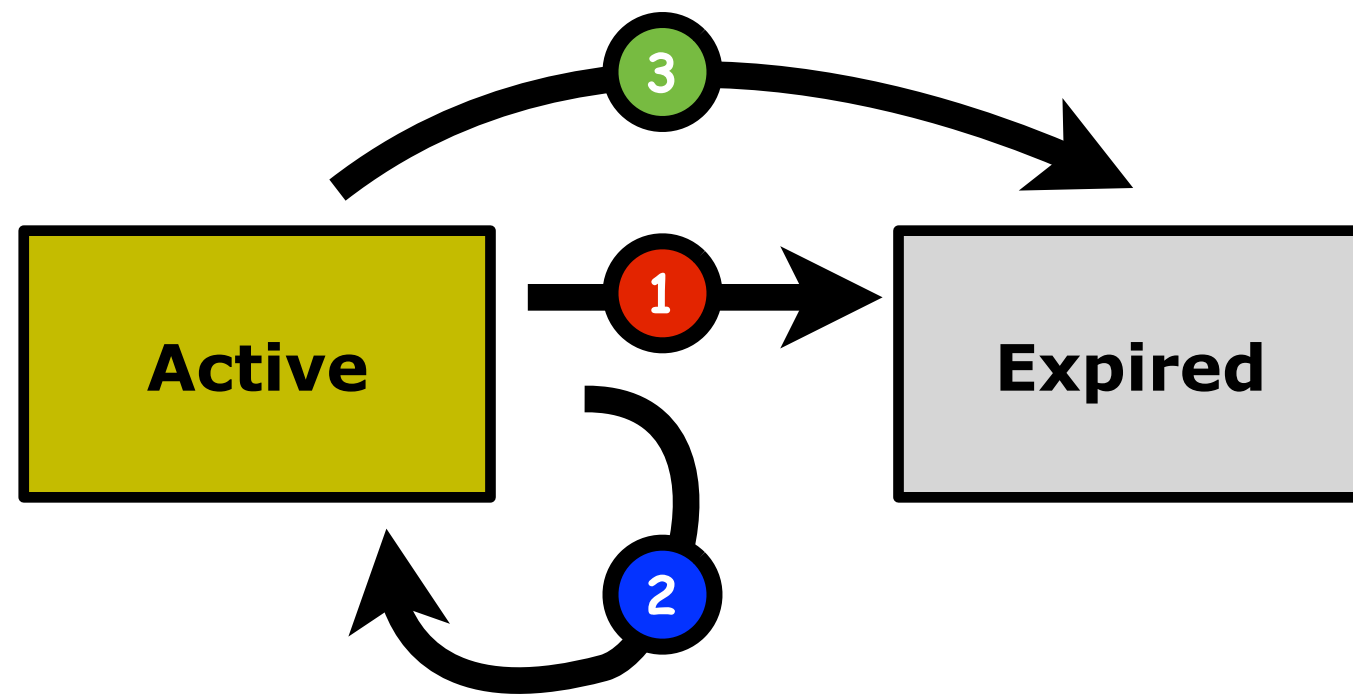


These runnable processes have exhausted their time quantum and are thus forbidden to run until all active processes expire.

Both arrays are indexed according to priority value (0 - 139).

When the set of active processes becomes empty, the expired set becomes the new active set and vice versa.





## Heuristic

- 1 The scheduler tries to boost the performance of interactive processes.
- 2 An active interactive process that finishes its time quantum usually remains active.
- 3 If the eldest expired process has already waited for a long time, or if an expired process has higher static priority (lower nice value) than the interactive process.

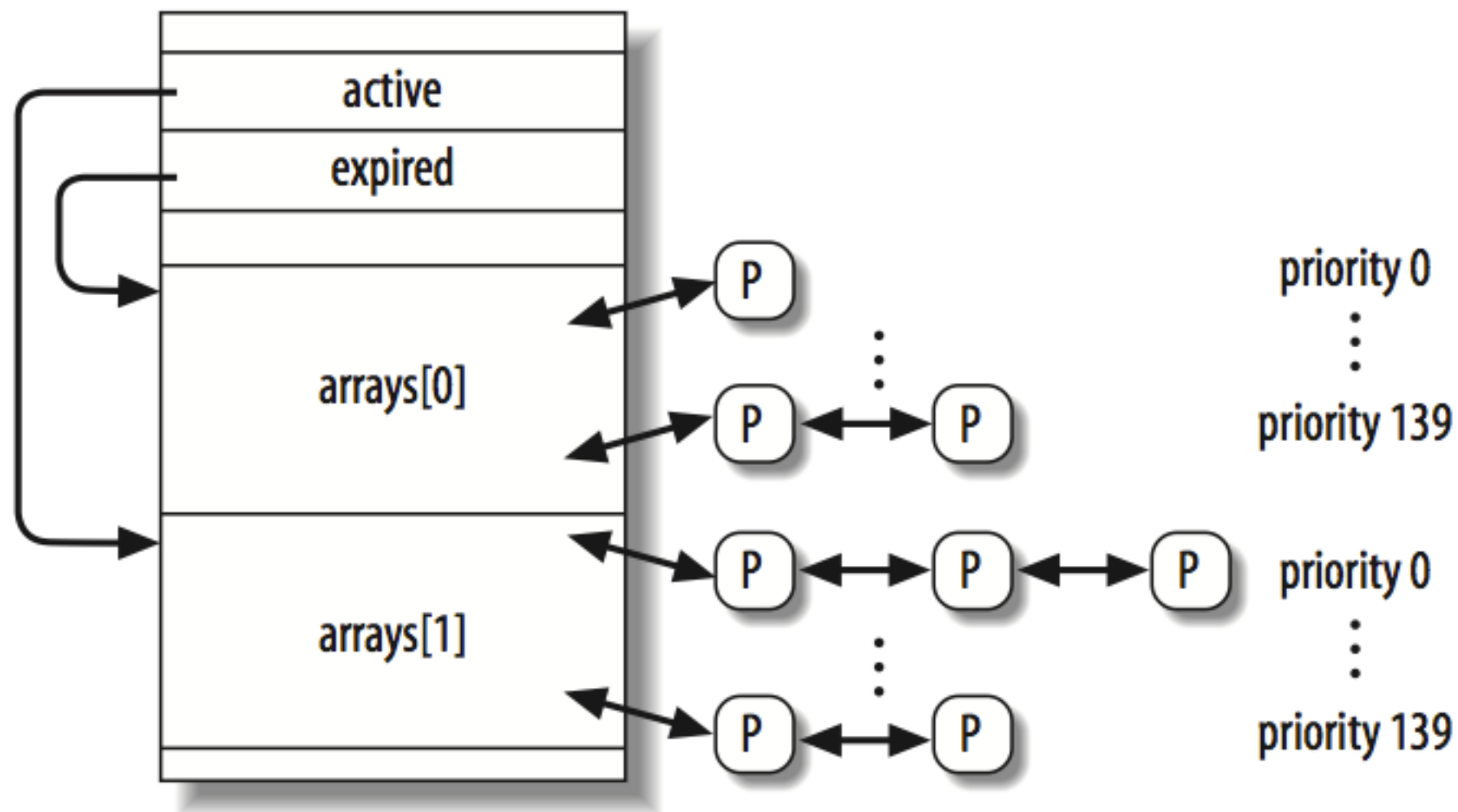
## Scheduler action

- An active batch process that finishes its time quantum always becomes expired.
- The scheduler refills its time quantum and leaves it in the set of active processes.
- The scheduler moves an interactive process that finished its time quantum into the set of expired processes

As a consequence, the set of active processes will eventually become empty and the expired processes will have a chance to run.

# Swapping active and expired

Periodically, the roles of the set of active and the set of expired processes are swapped: the active processes suddenly become the expired processes, and the expired processes become the active ones.



Swapping the active and expired process sets is simple, only swap the pointers active and expired.



# Choose a task to run as fast as possible

The job of the scheduler is simple, choose the task on the highest priority list to execute.

To make it fast to find the next process to execute, a **bitmap** is used to keep track of the existence or not of tasks for all priority lists.

	Task bitmap																												
	32 bit word						32 bit word						32 bit word						32 bit word						32 bit word				
Priority	Highest priority																								Lowest priority				
	0	1	2	3	...	31	32	33	34	...	63	64	65	66	...	95	96	97	98	...	127	128	129	130	...	139			
Bit value	1	1	0	0	...	0	0	1	1	...	1	0	0	0	...	1	0	1	0	...	0	0	1	0	...	1			

If there are tasks in a task list with priority X, bit number X is set to 1 in the task bit map, otherwise the bit is set to 0.

On most architectures, a **find-first-bit-set instruction** is used to find the highest priority bit set in one of five 32-bit words (for the 140 priorities).

**O(1)**

The time it takes to find a task to execute depends not on the number of active tasks but instead on the number of priorities.

This makes the 2.6 scheduler an **O(1)** process because **the time to schedule is both fixed and deterministic regardless of the number of active tasks.**



# **Completely Fair** **$O(\log n)$ Scheduler**

Default scheduler since kernel 2.6.23 (2007-10-09)

# Completely Fair Scheduler (CFS)

The Completely Fair Scheduler (CFS) is the name of a process scheduler which was merged into the 2.6.23 release of the Linux kernel and is the default scheduler.

Instead of the run queues used in the  $O(1)$  scheduler, a single **red-black tree** is used to track all processes which are in a runnable state.

The process which pops up at the leftmost node of the tree is the one which is most entitled to run at any given time.

**Source:** <http://lwn.net/Articles/230574/>

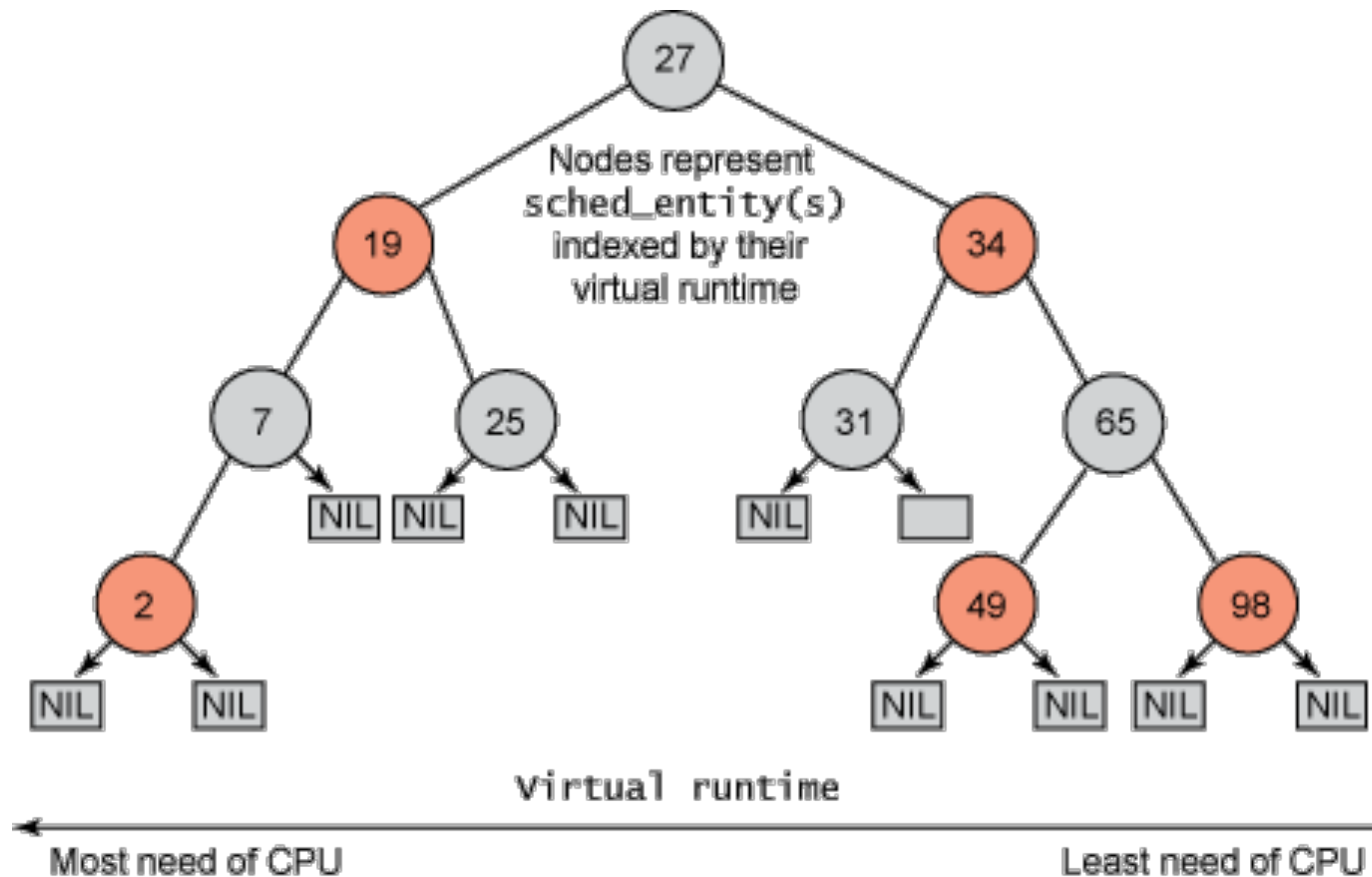
Accessed 2014-02-01

Because of this simple design, CFS no longer uses active and expired arrays and dispensed with sophisticated heuristics to mark tasks as interactive versus non-interactive.

# Self-balancing time-ordered red-black-tree (1)

CFS maintains a time-ordered red-black tree using virtual runtime as keys.

Each runnable task chases the other to maintain a balance of virtual execution time across the set of runnable tasks in the tree.



# Self-balancing red-black-tree

A red-black tree is a tree with a couple of interesting and useful properties.

First, it's **self-balancing**, which means that no path in the tree will ever be more than twice as long as any other.

Second, operations on the tree occur in  **$O(\log n)$**  time (where  $n$  is the number of nodes in the tree).

This means that you can insert or delete a task quickly and efficiently.

# Virtual runtime

To balance the black-red-tree, CFS maintains the amount of time provided to a given task in what's called the virtual runtime.

The smaller a task's virtual runtime,

*meaning the smaller amount of time a task has been permitted access to the processor*

, the higher its need for the processor.

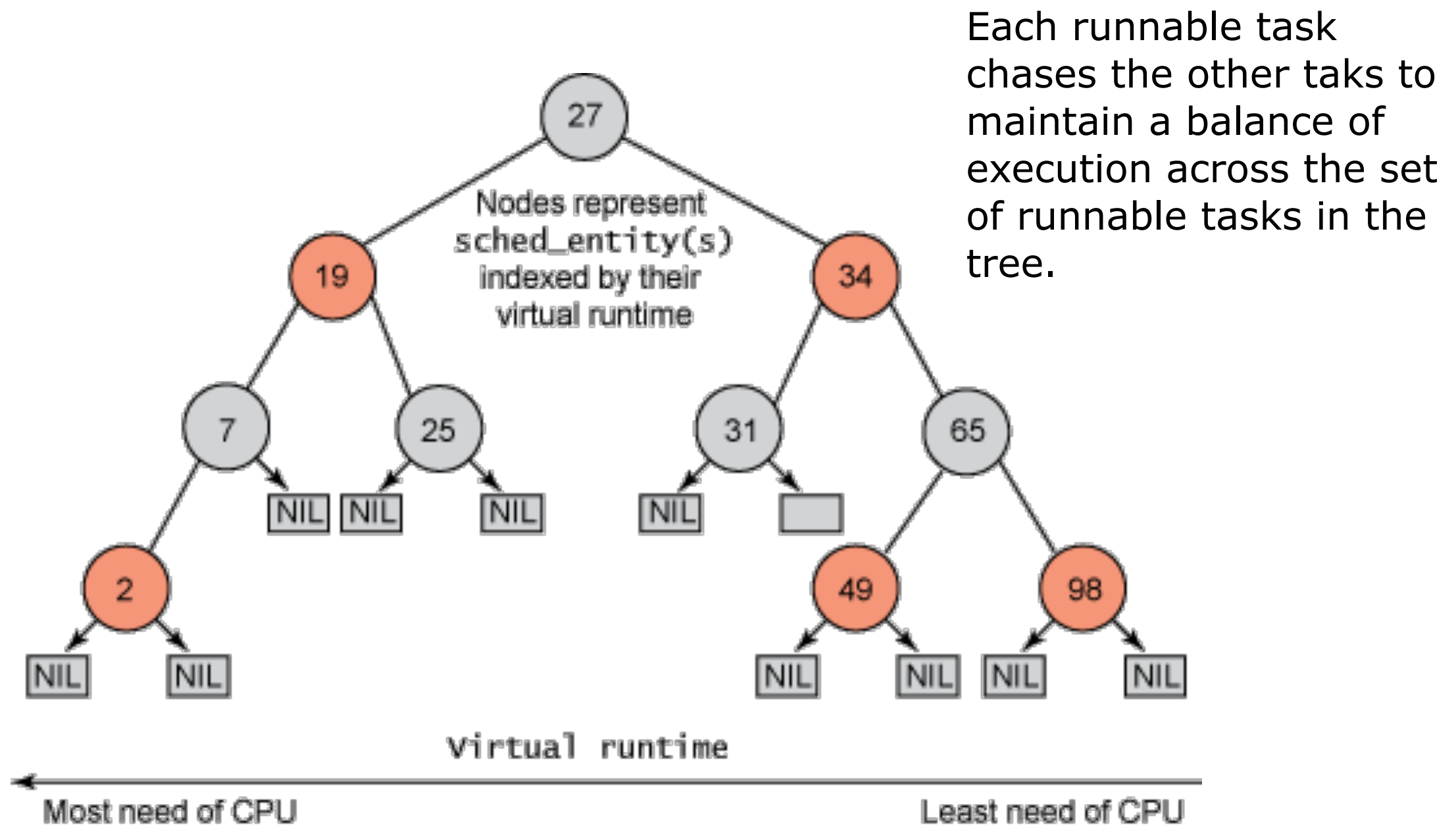
# Sleeper fairness

Like the old  $O(1)$  scheduler, CFS uses a concept called **sleeper fairness**, which considers sleeping or waiting tasks equivalent to those on the runqueue.

- ★ Interactive tasks which spend most of their time waiting for user input or other events get a comparable share of CPU time when they need it.
- ★ If a task spends a lot of its time sleeping, then its virtual runtime value is low and it automatically gets a priority boost when it finally needs it. Hence such tasks do not get less processor time than the tasks that are constantly running.

# Self-balancing time-ordered red-black-tree (2)

Tasks on the left side of the tree are given time to execute, and the **tasks** in the tree **migrate from the right to the left** to maintain **fairness**.



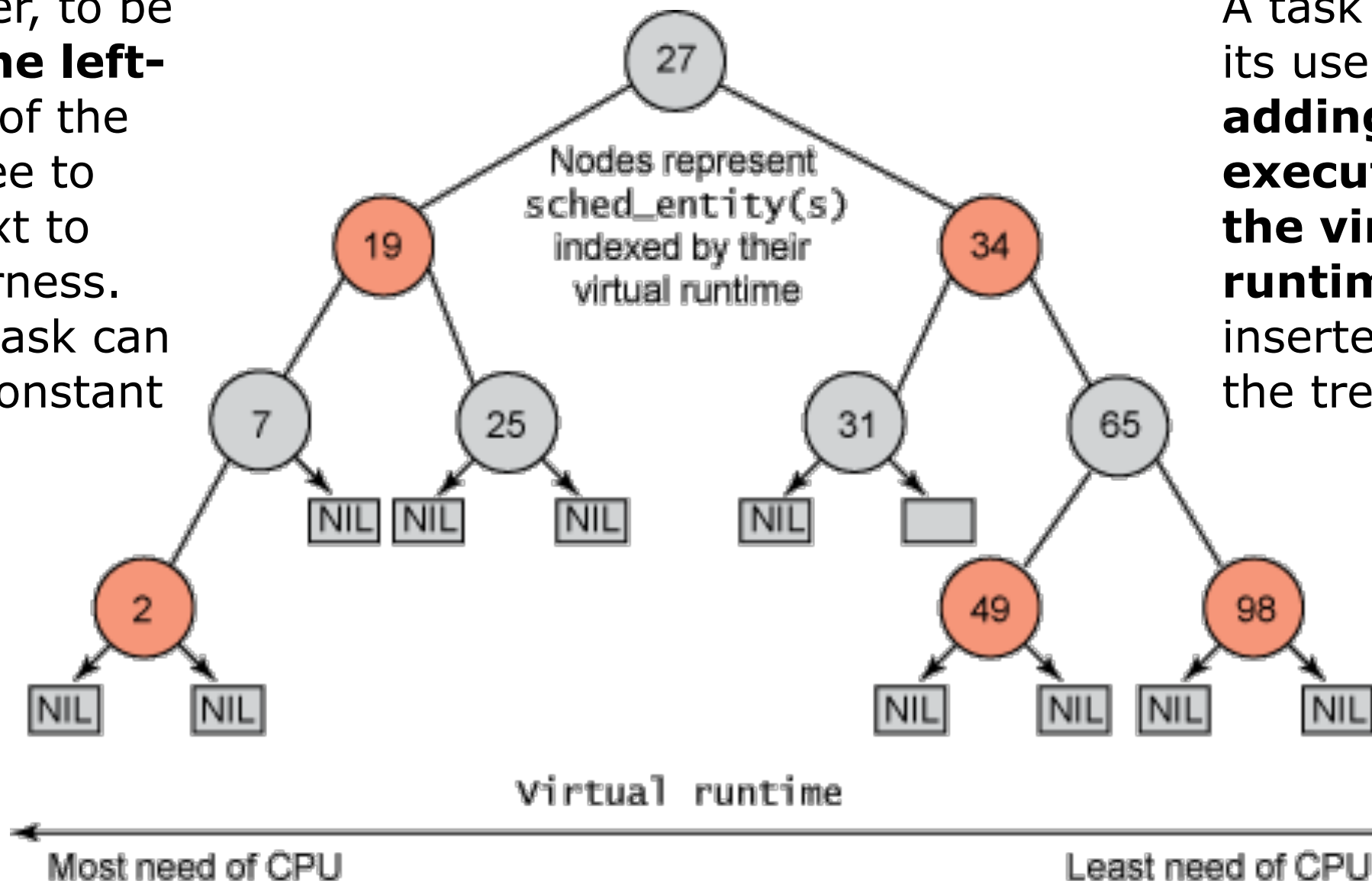


# Self-balancing time-ordered red-black-tree (3)

Choosing a task can be done in constant time  $O(1)$  but inserting a task back into the tree if runnable is  $O(\log n)$ .

## $O(1)$

The scheduler, to be fair, **picks the left-most node** of the red-black tree to schedule next to maintain fairness. Choosing a task can be done in constant time.



## $O(\log n)$

A task accounts for its use of the CPU by **adding its execution time to the virtual runtime** and is then inserted back into the tree if runnable.