Linux Scheduling

Linux Kernel Development by Robert Love Chapter 4

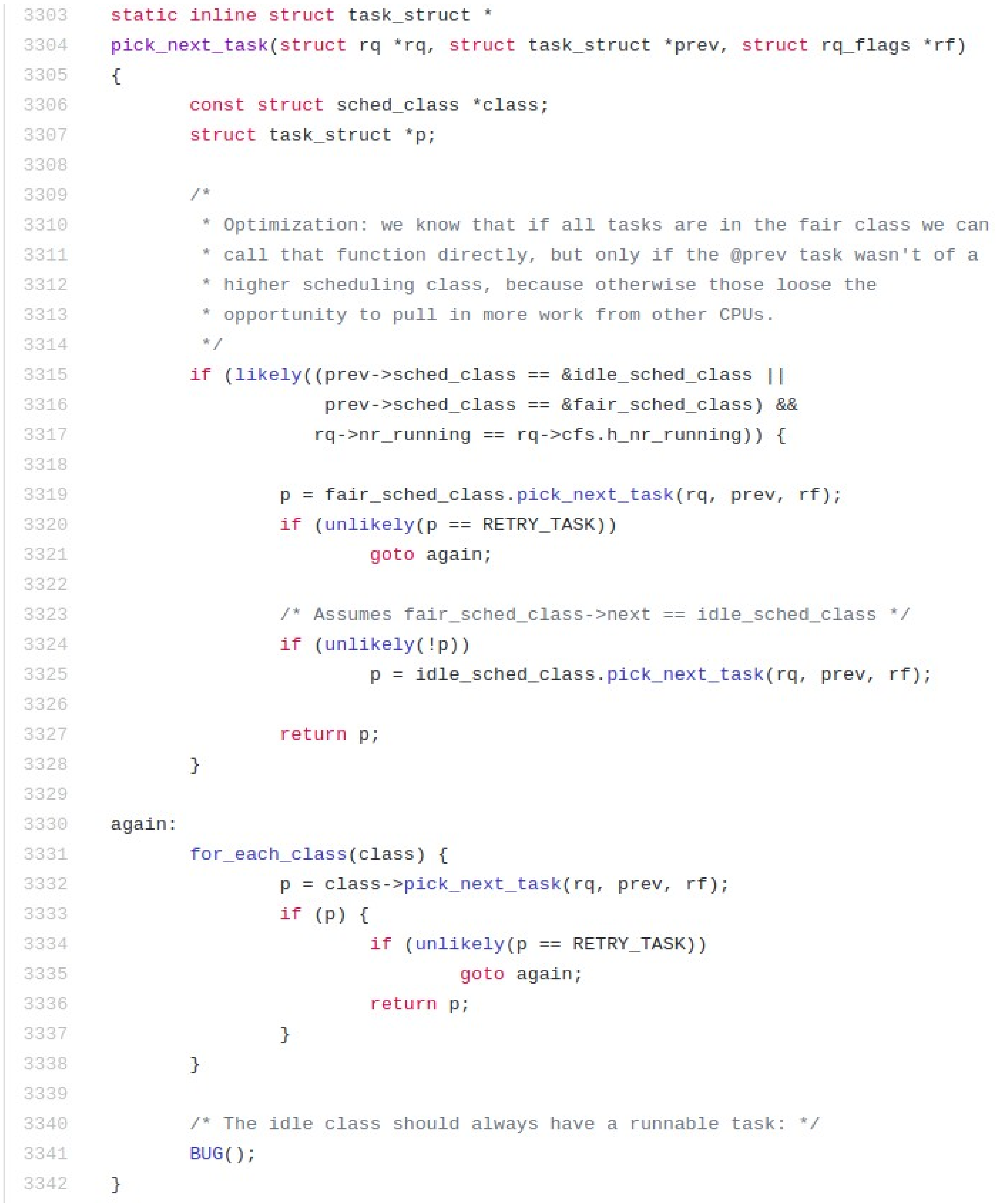
Process Types or Scheduler Classes

- Real time
  https://github.com/torvalds/linux/blob/master/kernel/sched/rt.c
  Should never be blocked by a lower priority task

- Normal processes (SCHED_NORMAL)
  https://github.com/torvalds/linux/blob/master/kernel/sched/fair.c
  I/O bound (interactive), Processor bound (batch)

Pick next task overall

https://github.com/torvalds/linux/blob/master/kernel/sched/core.c



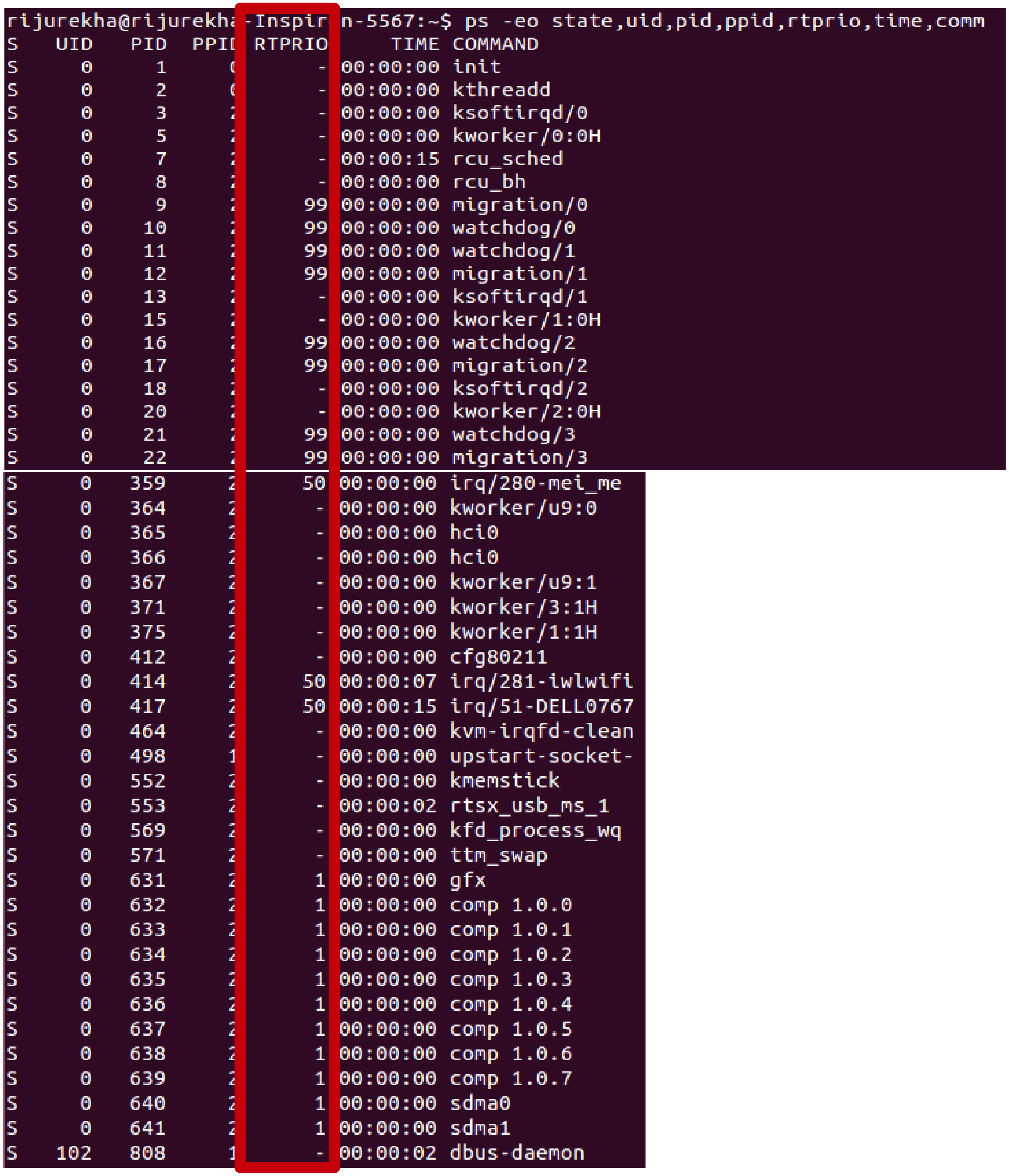
```c
static inline struct task_struct *
pick_next_task(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
{
        const struct sched_class *class;
        struct task_struct *p;

        /*
         * Optimization: we know that if all tasks are in the fair class we can
         * call that function directly, but only if the @prev task wasn't of a
         * higher scheduling class, because otherwise those loose the
         * opportunity to pull in more work from other CPUs.
         */
        if (likely((prev->sched_class == &idle_sched_class ||
                    prev->sched_class == &fair_sched_class) &&
                   rq->nr_running == rq->cfs.h_nr_running)) {

                p = fair_sched_class.pick_next_task(rq, prev, rf);
                if (unlikely(p == RETRY_TASK))
                        goto again;

                /* Assumes fair_sched_class->next == idle_sched_class */
                if (unlikely(!p))
                        p = idle_sched_class.pick_next_task(rq, prev, rf);

                return p;
        }

again:
        for_each_class(class) {
                p = class->pick_next_task(rq, prev, rf);
                if (p) {
                        if (unlikely(p == RETRY_TASK))
                                goto again;
                        return p;
                }
        }

        /* The idle class should always have a runnable task: */
        BUG();
}
```

Process Priority

- A common type of scheduling algorithm is priority-based scheduling.
- The goal is to rank processes based on their worth and need for processor time.
- The Linux kernel implements two separate priority ranges.
  - Real time priority
  - Nice value

Real Time Priority

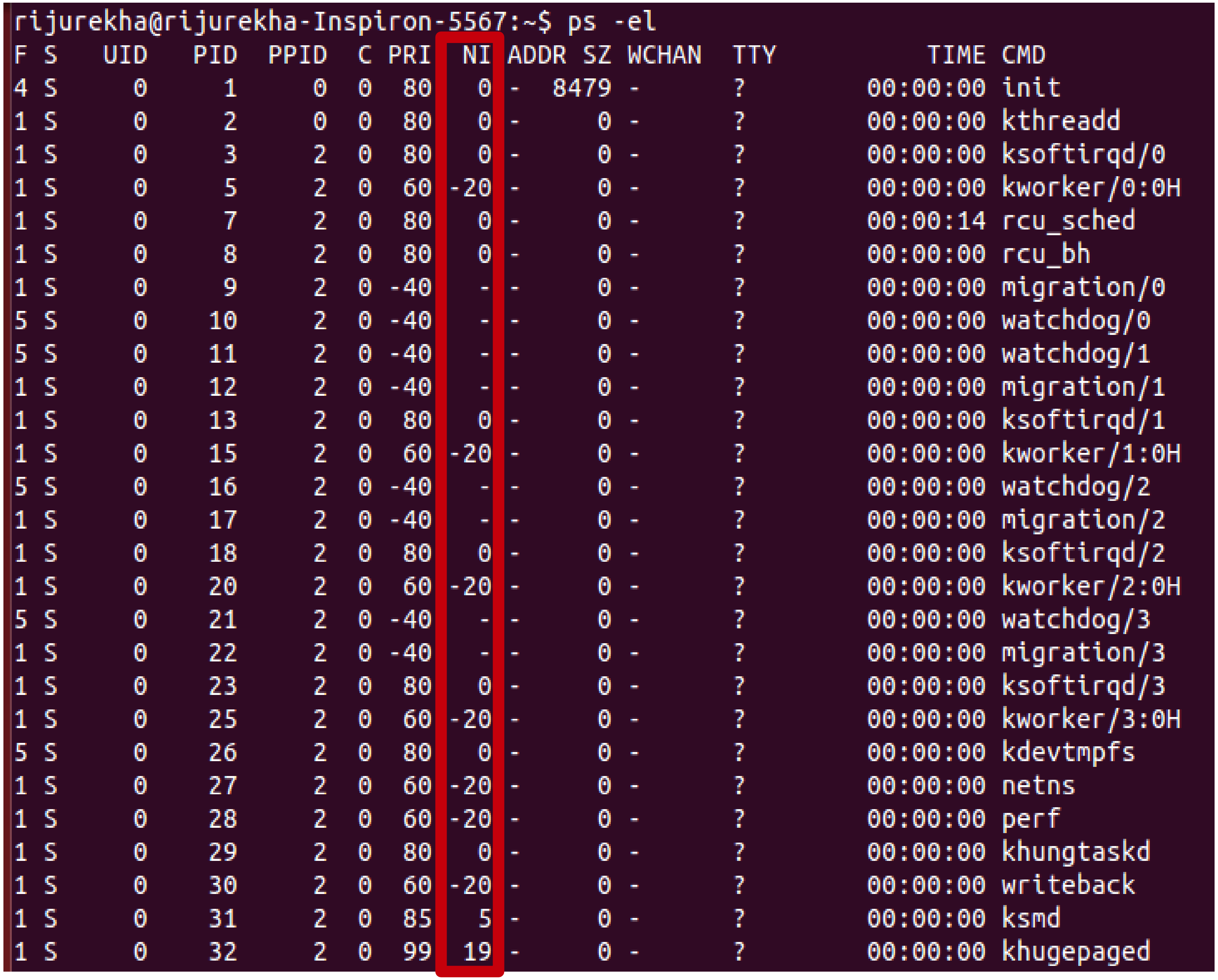
- Range from 0 to 99, inclusive.
- Opposite from nice values, higher real-time priority values correspond to a greater priority.
- All real-time processes are at a higher priority than normal processes; that is, the real-time priority and nice value are in disjoint value spaces.



```
rijurekha@rijurekha-Inspiron-5567:~$ ps -eo state,uid,pid,ppid,rtprio,time,comm
S   UID   PID  PPID RTPRIO     TIME COMMAND
S     0     1     0      - 00:00:00 init
S     0     2     0      - 00:00:00 kthreadd
S     0     3     2      - 00:00:00 ksoftirqd/0
S     0     5     2      - 00:00:00 kworker/0:0H
S     0     7     2      - 00:00:15 rcu_sched
S     0     8     2      - 00:00:00 rcu_bh
S     0     9     2     99 00:00:00 migration/0
S     0    10     2     99 00:00:00 watchdog/0
S     0    11     2     99 00:00:00 watchdog/1
S     0    12     2     99 00:00:00 migration/1
S     0    13     2      - 00:00:00 ksoftirqd/1
S     0    15     2      - 00:00:00 kworker/1:0H
S     0    16     2     99 00:00:00 watchdog/2
S     0    17     2     99 00:00:00 migration/2
S     0    18     2      - 00:00:00 ksoftirqd/2
S     0    20     2      - 00:00:00 kworker/2:0H
S     0    21     2     99 00:00:00 watchdog/3
S     0    22     2     99 00:00:00 migration/3
S     0   359     2     50 00:00:00 irq/280-mei_me
S     0   364     2      - 00:00:00 kworker/u9:0
S     0   365     2      - 00:00:00 hci0
S     0   366     2      - 00:00:00 hci0
S     0   367     2      - 00:00:00 kworker/u9:1
S     0   371     2      - 00:00:00 kworker/3:1H
S     0   375     2      - 00:00:00 kworker/1:1H
S     0   412     2      - 00:00:00 cfg80211
S     0   414     2     50 00:00:07 irq/281-iwlwifi
S     0   417     2     50 00:00:15 irq/51-DELL0767
S     0   464     2      - 00:00:00 kvm-irqfd-clean
S     0   498     1      - 00:00:00 upstart-socket-
S     0   552     2      - 00:00:00 kmemstick
S     0   553     2      - 00:00:02 rtsx_usb_ms_1
S     0   569     2      - 00:00:00 kfd_process_wq
S     0   571     2      - 00:00:00 ttm_swap
S     0   631     2      1 00:00:00 gfx
S     0   632     2      1 00:00:00 comp 1.0.0
S     0   633     2      1 00:00:00 comp 1.0.1
S     0   634     2      1 00:00:00 comp 1.0.2
S     0   635     2      1 00:00:00 comp 1.0.3
S     0   636     2      1 00:00:00 comp 1.0.4
S     0   637     2      1 00:00:00 comp 1.0.5
S     0   638     2      1 00:00:00 comp 1.0.6
S     0   639     2      1 00:00:00 comp 1.0.7
S     0   640     2      1 00:00:00 sdma0
S     0   641     2      1 00:00:00 sdma1
S   102   808     1      - 00:00:02 dbus-daemon
```

Nice value

- A number from –20 to +19 with a default of 0. Corresponding priority values are 100 (highest priority) to 139 (lowest priority), as the default base priority is 120.
- Larger nice values correspond to a lower priority—you are being “nice” to the other processes on the system.
- Nice values are the standard priority range used in all Unix systems, although different Unix systems apply them in different ways, reflective of their individual scheduling algorithms.
  - In Mac OS X, the nice value is a control over the absolute timeslice allotted to a process (true earlier for Linux O(1) scheduler)
  - In Linux, it is a control over the proportion of timeslice in CFS.



```
rijurekha@rijurekha-Inspiron-5567:~$ ps -el
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0     1     0  0  80   0 -  8479 -      ?        00:00:00 init
1 S     0     2     0  0  80   0 -     0 -      ?        00:00:00 kthreadd
1 S     0     3     2  0  80   0 -     0 -      ?        00:00:00 ksoftirqd/0
1 S     0     5     2  0  60 -20 -     0 -      ?        00:00:00 kworker/0:0H
1 S     0     7     2  0  80   0 -     0 -      ?        00:00:14 rcu_sched
1 S     0     8     2  0  80   0 -     0 -      ?        00:00:00 rcu_bh
1 S     0     9     2  0 -40   - -     0 -      ?        00:00:00 migration/0
5 S     0    10     2  0 -40   - -     0 -      ?        00:00:00 watchdog/0
5 S     0    11     2  0 -40   - -     0 -      ?        00:00:00 watchdog/1
1 S     0    12     2  0 -40   - -     0 -      ?        00:00:00 migration/1
1 S     0    13     2  0  80   0 -     0 -      ?        00:00:00 ksoftirqd/1
1 S     0    15     2  0  60 -20 -     0 -      ?        00:00:00 kworker/1:0H
5 S     0    16     2  0 -40   - -     0 -      ?        00:00:00 watchdog/2
1 S     0    17     2  0 -40   - -     0 -      ?        00:00:00 migration/2
1 S     0    18     2  0  80   0 -     0 -      ?        00:00:00 ksoftirqd/2
1 S     0    20     2  0  60 -20 -     0 -      ?        00:00:00 kworker/2:0H
5 S     0    21     2  0 -40   - -     0 -      ?        00:00:00 watchdog/3
1 S     0    22     2  0 -40   - -     0 -      ?        00:00:00 migration/3
1 S     0    23     2  0  80   0 -     0 -      ?        00:00:00 ksoftirqd/3
1 S     0    25     2  0  60 -20 -     0 -      ?        00:00:00 kworker/3:0H
5 S     0    26     2  0  80   0 -     0 -      ?        00:00:00 kdevtmpfs
1 S     0    27     2  0  60 -20 -     0 -      ?        00:00:00 netns
1 S     0    28     2  0  60 -20 -     0 -      ?        00:00:00 perf
1 S     0    29     2  0  80   0 -     0 -      ?        00:00:00 khungtaskd
1 S     0    30     2  0  60 -20 -     0 -      ?        00:00:00 writeback
1 S     0    31     2  0  85   5 -     0 -      ?        00:00:00 ksmd
1 S     0    32     2  0  99  19 -     0 -      ?        00:00:00 khugepaged
```

I/O bound or interactive processes

- Characterized as a process that spends much of its time submitting and waiting on I/O requests.
- By I/O, we mean any type of blockable resource, such as keyboard input or network I/O, and not just disk I/O.)
- Most graphical user interface (GUI) applications, for example, are I/O-bound, even if they never read from or write to the disk, because they spend most of their time waiting on user interaction via the keyboard and mouse.
- Runnable for only short durations, because it eventually blocks waiting on more I/O.
- If scheduled at minimum latency once runnable, then I/O will be utilized better.

Processor bound or batch processes

- Spend much of their time executing code.
- Tend to run until they are preempted because they do not block on I/O requests very often.
- The ultimate example of a processor-bound process is one executing an infinite loop.
- More palatable examples include programs that perform a lot of mathematical calculations, such as ssh-keygen or MATLAB.
- A scheduler policy for processor-bound processes, therefore, tends to run such processes less frequently but for longer durations.

Not mutually exclusive

- Processes can exhibit both behaviors simultaneously
- Processes can be I/O-bound but dive into periods of intense processor action.
- E.g. word processor, which normally sits waiting for key presses but at any moment might peg the processor in a rabid fit of spell checking or macro calculation.

Unix favors I/O bound processes

- The scheduler policy in Unix systems tends to explicitly favor I/O-bound processes, thus providing good response time.
- Linux, aiming to provide good interactive response and desktop performance, optimizes for process response (low latency), thus favoring I/O-bound processes over processor-bound processes.
- As we will see, this is done in a creative manner that does not neglect processor-bound processes.

Policy is decided. Needs to create mechanisms to support this policy.

Priority: decides which task is picked next. Timeslice: decides how long a picked task is run.

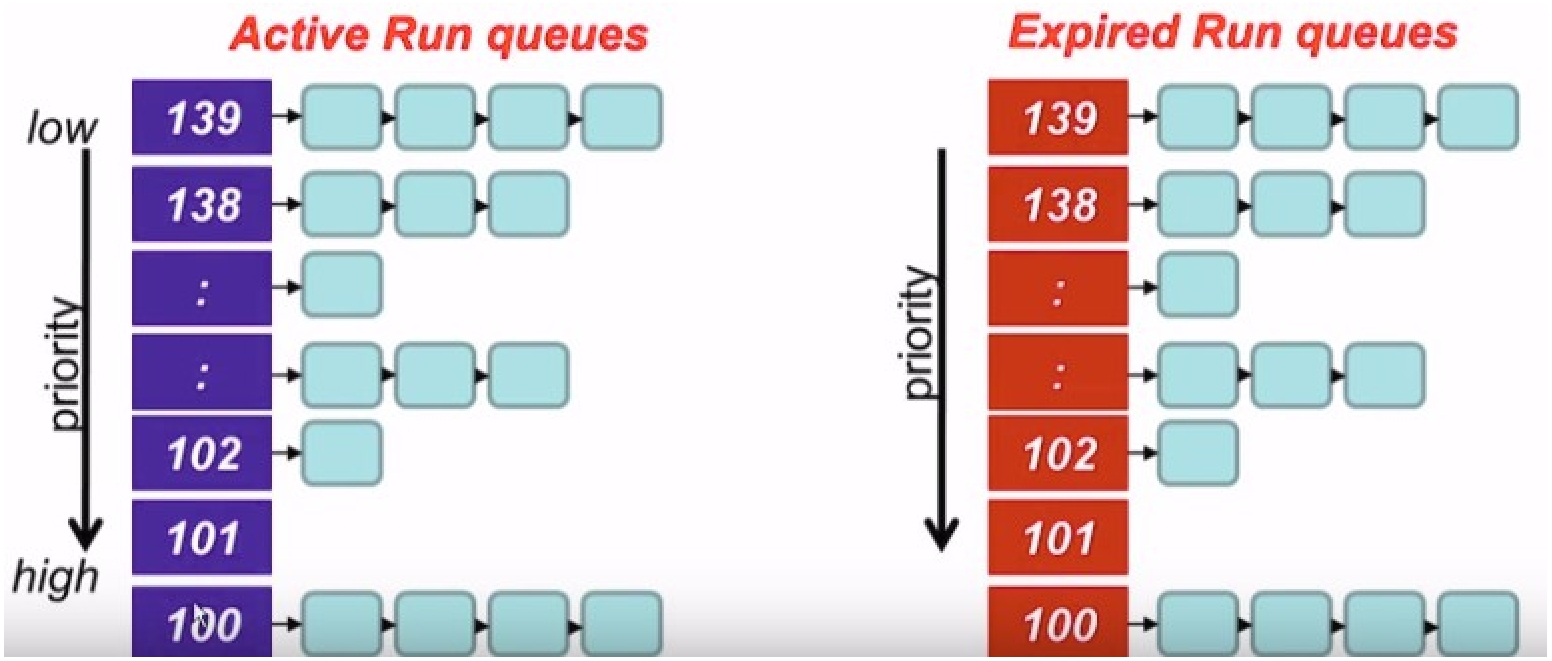
Using static priority O(N) scheduler

- At every context switch
  - Scan the list of runnable processes
  - Compute priorities
  - Select the best process to run
- O(n) where n is the number of processes to run
- Scalability issues observed when Java was introduced (JVM spawns many tasks)

Queue of Ready Processes

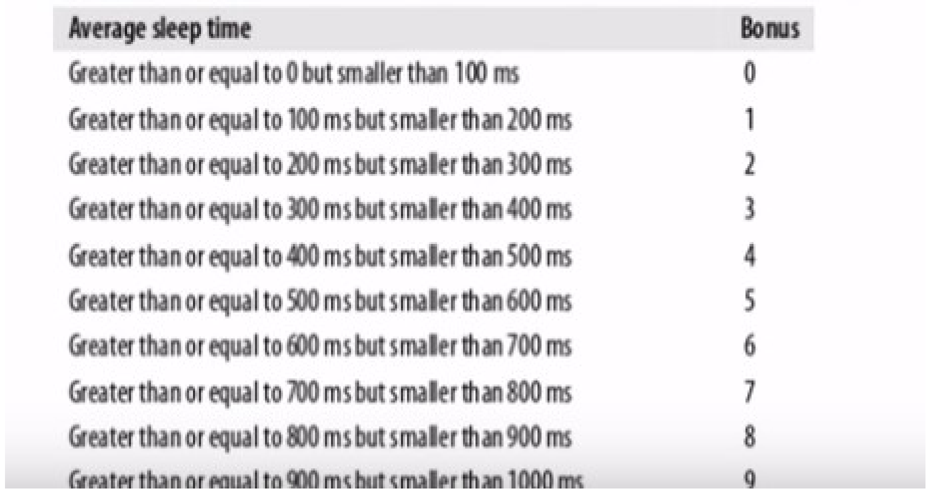
Using dynamic priority O(1) scheduler

- Two ready queues in each CPU
  - Two queues needed to prevent starvation
  - Each queue has 40 priority classes
  - 100 has highest priority, 139 has lowest priority
  - Bitmap to get lowest numbered queue with at least one task in O(1).
  - Dequeue is O(1)

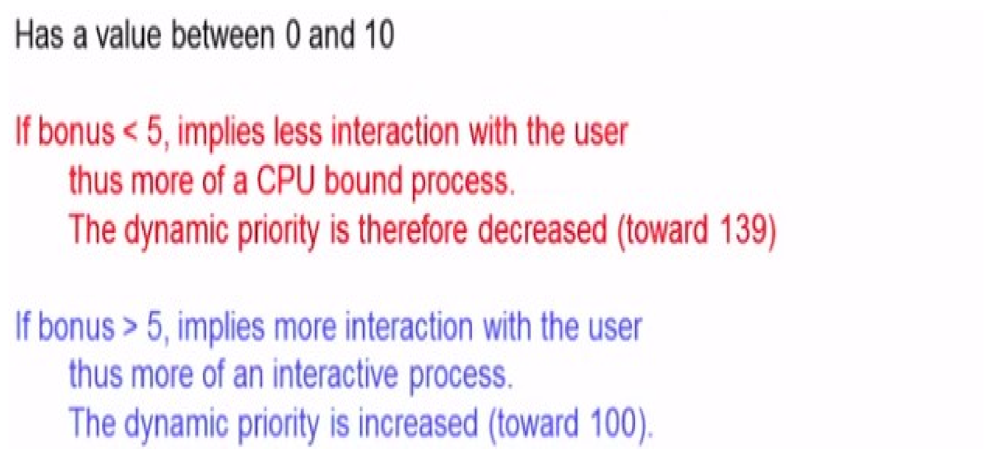


Dynamic Priority

- Max(100, min(static priority – **bonus** +5), 139)
- To distinguish between processor bound (batch) and I/O bound (interactive) processes
- Based on average sleep time
  - An I/O bound process will sleep more, and therefore should get a higher priority
  - A CPU bound process will sleep less, and therefore will get a lower priority.



Average sleep time	Bonus
Greater than or equal to 0 but smaller than 100 ms	0
Greater than or equal to 100 ms but smaller than 200 ms	1
Greater than or equal to 200 ms but smaller than 300 ms	2
Greater than or equal to 300 ms but smaller than 400 ms	3
Greater than or equal to 400 ms but smaller than 500 ms	4
Greater than or equal to 500 ms but smaller than 600 ms	5
Greater than or equal to 600 ms but smaller than 700 ms	6
Greater than or equal to 700 ms but smaller than 800 ms	7
Greater than or equal to 800 ms but smaller than 900 ms	8
Greater than or equal to 900 ms but smaller than 1000 ms	9



Has a value between 0 and 10

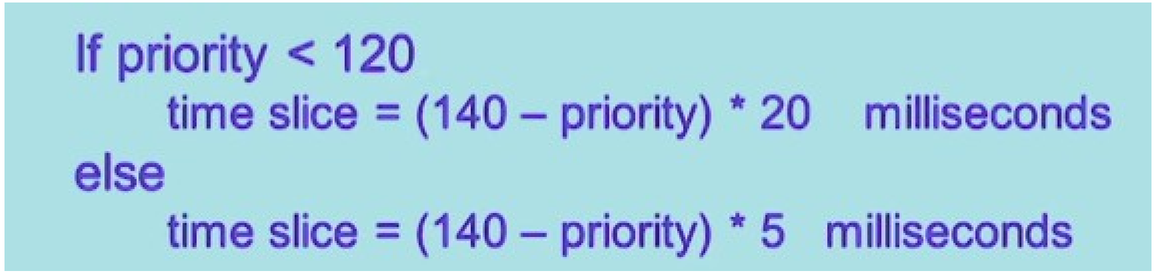
If bonus < 5, implies less interaction with the user
thus more of a CPU bound process.
The dynamic priority is therefore decreased (toward 139)

If bonus > 5, implies more interaction with the user
thus more of an interactive process.
The dynamic priority is increased (toward 100).

Timeslice: how long task is run

Interactive processes have high priorities

- But likely to not complete their timeslice
- Give it the largest timeslice to ensure it completes its burst without being preempted

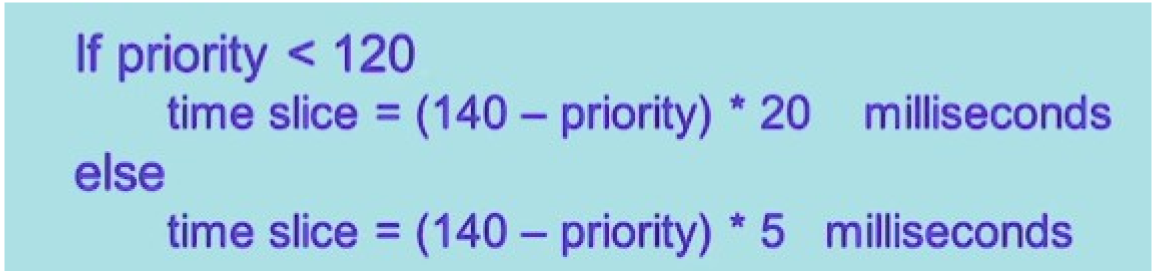


```
If priority < 120
    time slice = (140 – priority) * 20    milliseconds
else
    time slice = (140 – priority) * 5   milliseconds
```

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

Timeslice Issue

- Represents how long a task can run until it is pre-empted.
- Interactive processes have high priorities
  - But likely to not complete their timeslice
  - Give it the largest timeslice to ensure it completes its burst without being preempted



```
If priority < 120
    time slice = (140 – priority) * 20   milliseconds
else
    time slice = (140 – priority) * 5   milliseconds
```

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

Disproportionate timeslice increase

More issues with fixed timeslice

- Can lead to suboptimal switching behavior. E.g.
  - processes of the default nice value (zero) a timeslice of 100 ms
  - processes at the highest nice value (+20, the lowest priority) a timeslice of 5 ms
  - default priority process thus receives 20⁄21 (100 out of 105 milliseconds) of the processor
  - low priority process receives 1/21 (5 out of 105 milliseconds) of the processor
- Exactly two low priority processes?
  - each receives 50% of the processor (matches expectation)
  - But they each enjoy the processor for only 5 ms at a time (5 out of 10 milliseconds each)!
  - That is, instead of context switching twice every 105 milliseconds, we now context switch twice every 5 ms.
- Exactly two normal priority processes?
  - the correct 50% of the processor (matches expectation)
  - but in 100 millisecond increments
- Neither of these timeslice allotments are necessarily ideal; each is simply a byproduct of a given nice value to timeslice mapping coupled with a specific runnable process priority mix.

Non-trivial to set the timeslice?

- Too long a timeslice causes the system to have poor interactive performance
- Too short a timeslice causes significant context switching overhead
- I/O-bound processes like to run often, but do not need longer timeslices
- Processor-bound processes crave long timeslices to keep their caches hot
- As long timeslice would result in poor interactive performance, many OS have low default timeslice e.g. 10 ms.

Linux’s CFS scheduler, does not directly assign timeslices to processes.
CFS assigns processes a proportion of the processor.

Two sample tasks

- I/O-bound because it spends nearly all its time waiting for user key presses.
- No matter how fast the user types, it is not that fast.
- Despite this, when the text editor does receive a key press, the user expects the editor to respond immediately.
- First, we want it to have a large amount of processor time available to it; not because it needs a lot of processor (it does not) but because we want it to always have processor time available the moment it needs it.
- Second, we want the text editor to preempt the video encoder the moment it wakes up (say, when the user presses a key).

Text Editor

- Processor-bound.
- Aside from reading the raw data stream from the disk and later writing the resulting video, the encoder spends all its time applying the video codec to the raw data, easily consuming 100% of the processor.
- Does not have any strong time constraints on when it runs—if it started running now or in half a second, the user could not tell and would not care.
- Of course, the sooner it finishes the better, but latency is not a primary concern.

Video Encoder

CFS intuition

- If these are the only running processes and both are at the same nice level, this proportion would be 50%.
- Because the text editor spends most of its time blocked, waiting for user key presses, it does not use anywhere near 50% of the processor.
- Conversely, the video encoder is free to use more than its allotted 50%, enabling it to finish the encoding quickly.
- When the editor wakes up, CFS notes that it is allotted 50% of the processor but has used considerably less. Specifically, CFS determines that the text editor has run for less time than the video encoder.
- Attempting to give all processes a fair share of the processor, it then preempts the video encoder and enables the text editor to run.The text editor runs, quickly processes the user’s key press, and again sleeps, waiting for more input.
- As the text editor has not consumed its allotted 50%, we continue in this manner, with CFS always enabling the text editor to run when it wants and the video encoder to run the rest of the time.

Virtual Runtimes

- With each runnable process is included a virtual runtime vruntime
- At every scheduling point, if process has run for t ms, then vruntime += t
- vruntime for a process therefore monotonically increases

CFS uses vruntime to pick task

- When the timer interrupt occurs
  - Choose the task with lowest vruntime min_vruntime
  - Compute its dynamic timeslice
  - Program the high resolution timer with this timeslice
- The process begins to execute in the CPU
- When interrupt occurs again, context switch if there is another task with a smaller vruntime

Compute the dynamic timeslice

- Each process runs for a “timeslice” proportional to its weight divided by the total weight of all runnable threads.
- To calculate the actual timeslice, CFS sets a target for its approximation of the “infinitely small” scheduling duration in perfect multitasking. This target is called the targeted latency.
- Smaller targets yield better interactivity and a closer approximation to perfect multitasking, at the expense of higher switching costs and thus worse overall throughput.
- Let’s assume the targeted latency is 20 milliseconds and
  - 2 runnable tasks at the same priority, each will run for 10 ms.
  - 4 runnable tasks at the same priority, each will run for 5 ms.
  - 20 runnable tasks at the same priority, each will run for 1 millisecond.
  - As the number of runnable tasks approaches infinity, the proportion of allotted processor and the assigned timeslice approaches zero. This will eventually result in unacceptable switching costs.
- CFS imposes a floor on the timeslice assigned to each process. This floor is called the minimum granularity. By default it is 1 millisecond.
- Thus even as the number of runnable processes approaches infinity, each will run for at least 1 ms, to ensure there is a ceiling on the incurred switching costs.

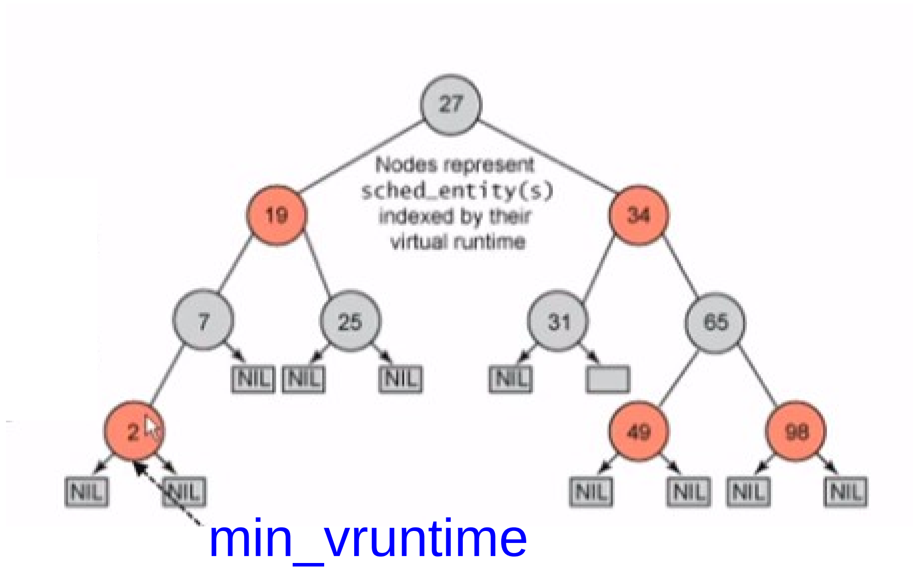
Nice values in CFS

- Affect both which task is picked next and dynamic timeslice allotted.
- If a process has run for t ms, then

  vruntime += t * (weight based on nice value)
- Higher nice value i.e. lower priority implies time moves at a faster rate compared to a higher priority task and moves quicker to the right of the rb-tree.
- Assigned proportion is affected by each process's nice value. The nice value acts as a weight, changing processor proportion.
  - Two runnable processes, one with the default nice value 0 and one with a nice value of 5.
  - These nice values have dissimilar weights and thus our two processes receive different proportions of the processor's time.
  - The weights work out to about a 1⁄3 penalty for the nice-5 process.
  - If our target latency is again 20 milliseconds, our two processes will receive 15 milliseconds and 5 milliseconds each of processor time, respectively.

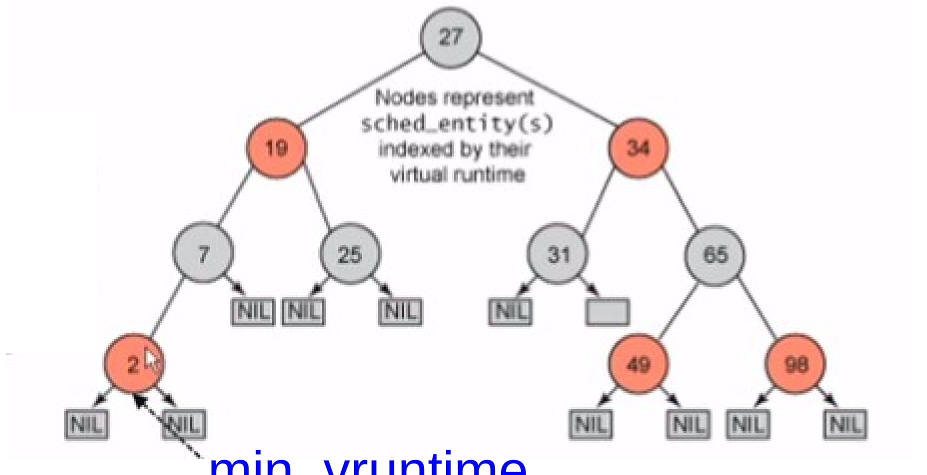
Picking the next task to run

- CFS uses a red black tree
  - Each node represents a runnable task
  - Nodes ordered according to their vruntime
  - Nodes on the left have lower vruntime than nodes on the right (since rb tree is a binary search tree)
  - The leftmost node is the task with the least vruntime, cached in min_vruntime


27
Nodes represent sched_entity(s) indexed by their virtual runtime
19
34
7
25
31
65
NIL
2
49
98
min_vruntime

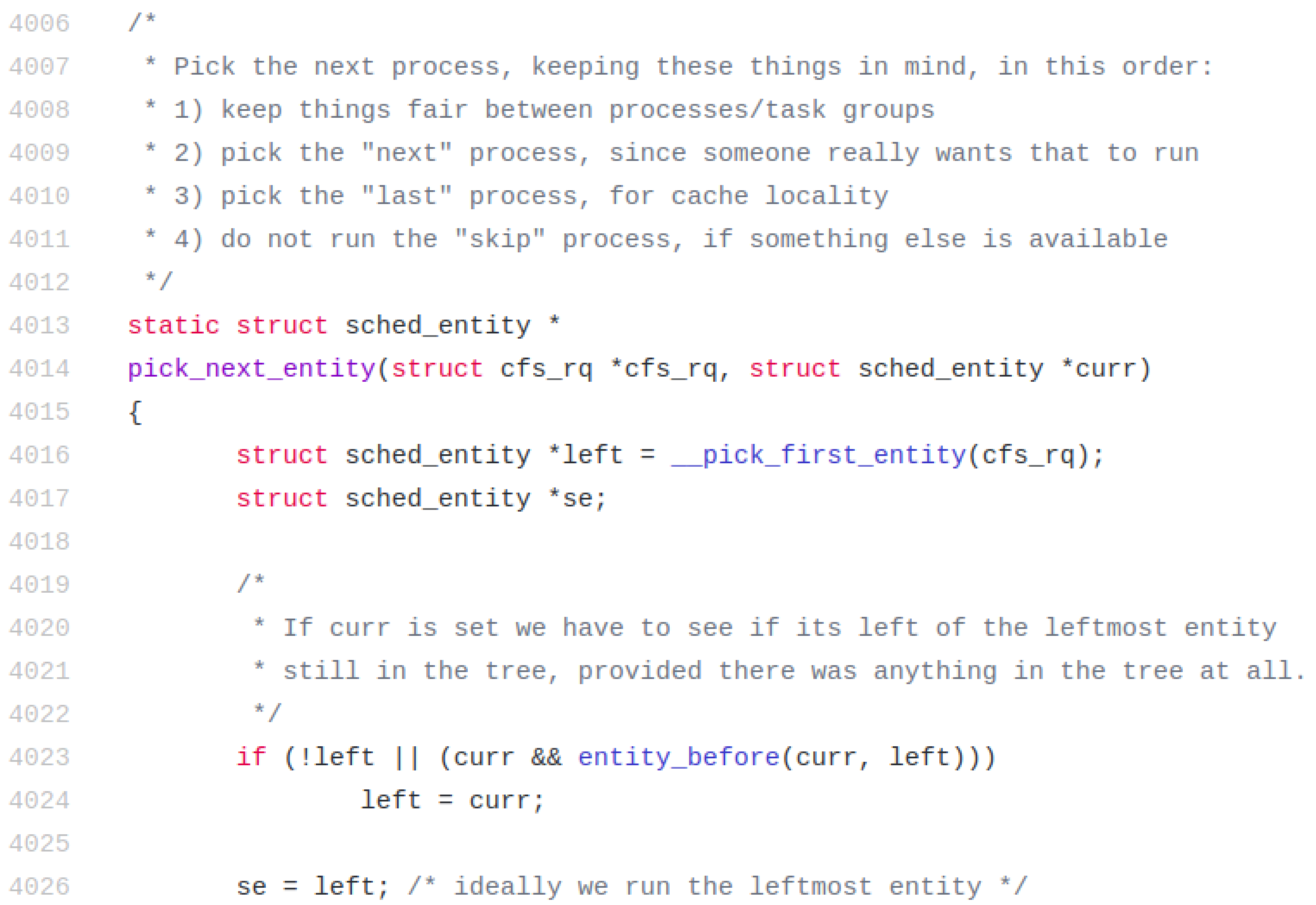
At a context switch

- Pick the leftmost node of the tree
  - This has the lowest vruntime
  - It is cached in min_vruntime, so accessed in O(1)
- If the preempted process is runnable, it is inserted into the tree depending on its new vruntime
  - This is O(log(n)) (by property of rb tree)
- Tasks move from left to right of the tree after its execution completes, prevents starvation

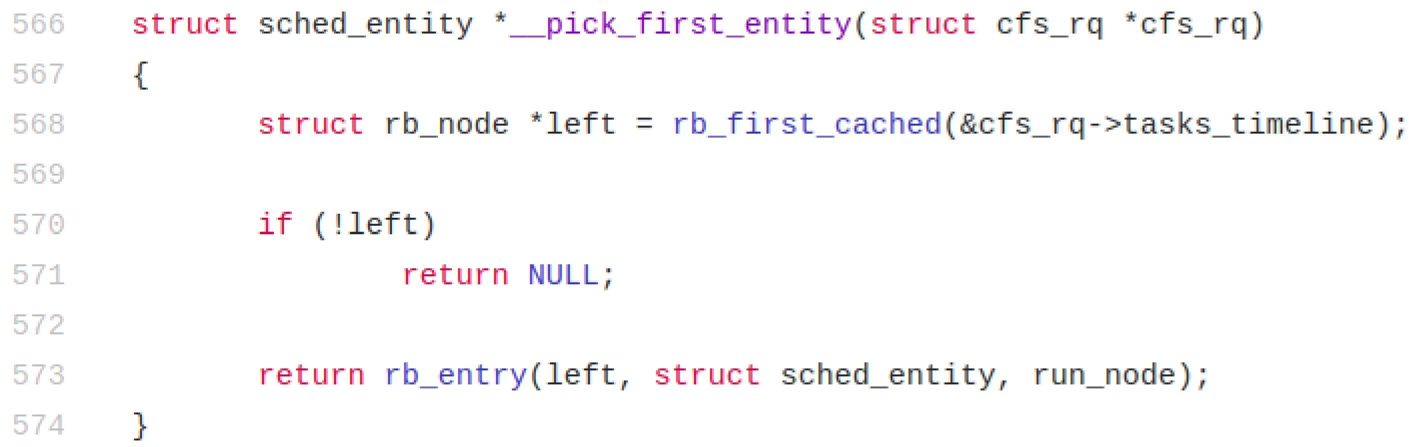

27
Nodes represent sched_entity(s) indexed by their virtual runtime
19
34
7
25
31
65
2
49
98
NIL
min_vruntime

Pick next task from CFS class

https://github.com/torvalds/linux/blob/master/kernel/sched/fair.c



```c
/*
 * Pick the next process, keeping these things in mind, in this order:
 * 1) keep things fair between processes/task groups
 * 2) pick the "next" process, since someone really wants that to run
 * 3) pick the "last" process, for cache locality
 * 4) do not run the "skip" process, if something else is available
 */
static struct sched_entity *
pick_next_entity(struct cfs_rq *cfs_rq, struct sched_entity *curr)
{
        struct sched_entity *left = __pick_first_entity(cfs_rq);
        struct sched_entity *se;

        /*
         * If curr is set we have to see if its left of the leftmost entity
         * still in the tree, provided there was anything in the tree at all.
         */
        if (!left || (curr && entity_before(curr, left)))
                left = curr;

        se = left; /* ideally we run the leftmost entity */
```



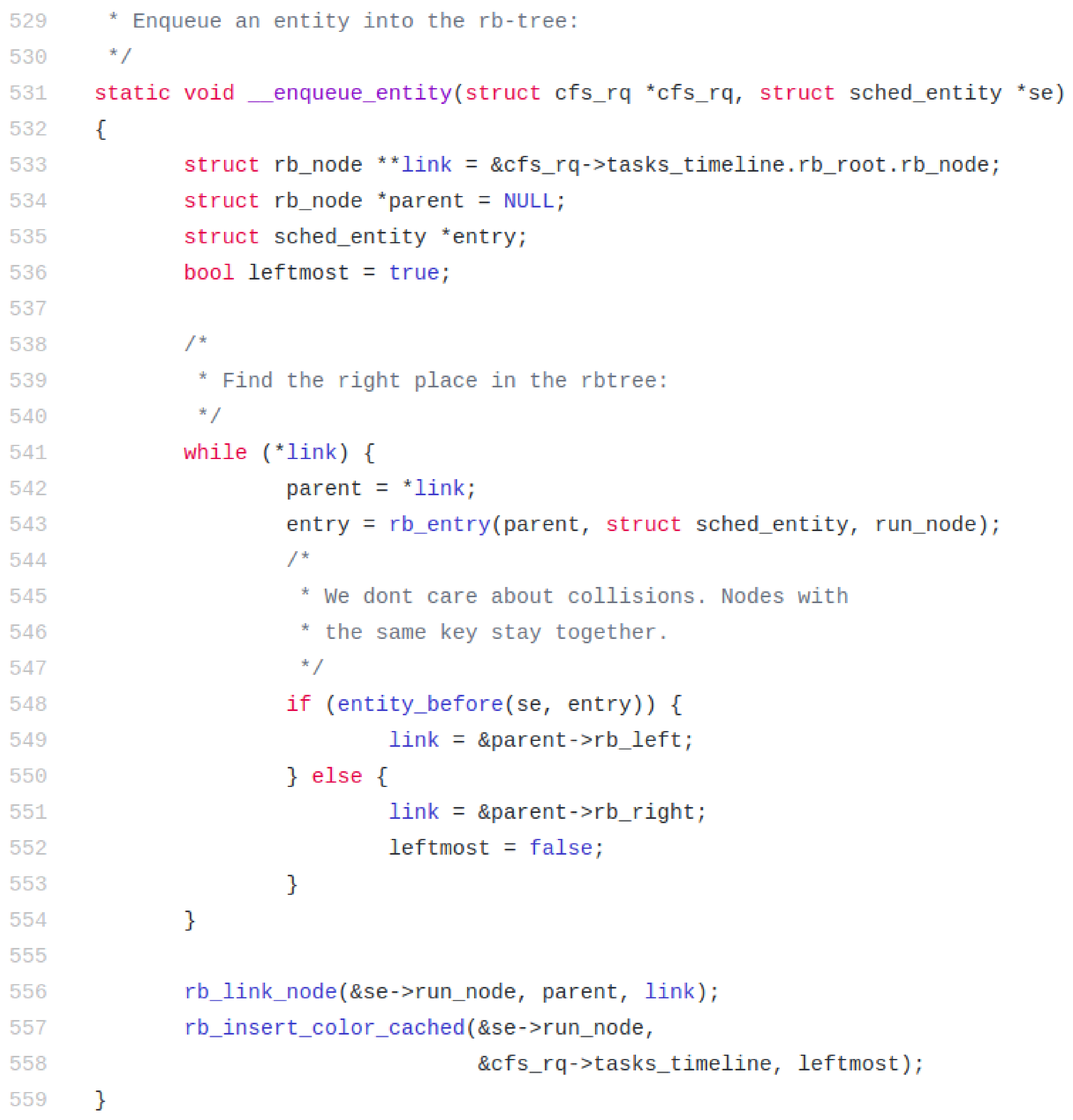
```c
struct sched_entity *__pick_first_entity(struct cfs_rq *cfs_rq)
{
        struct rb_node *left = rb_first_cached(&cfs_rq->tasks_timeline);

        if (!left)
                return NULL;

        return rb_entry(left, struct sched_entity, run_node);
}
```

I/O bound processes, new processes

- I/O bound processes have small CPU bursts, therefore will have low vruntime. They would appear to the left of the tree and get higher priorities.
- I/O bound processes will typically have larger time slices, because they have smaller vruntime.
- New processes get added to the rb-tree.
- Starts with the initial value of min_vruntime, to ensure it gets to execute quickly.

Adding processes to rb tree

https://github.com/torvalds/linux/blob/master/kernel/sched/fair.c

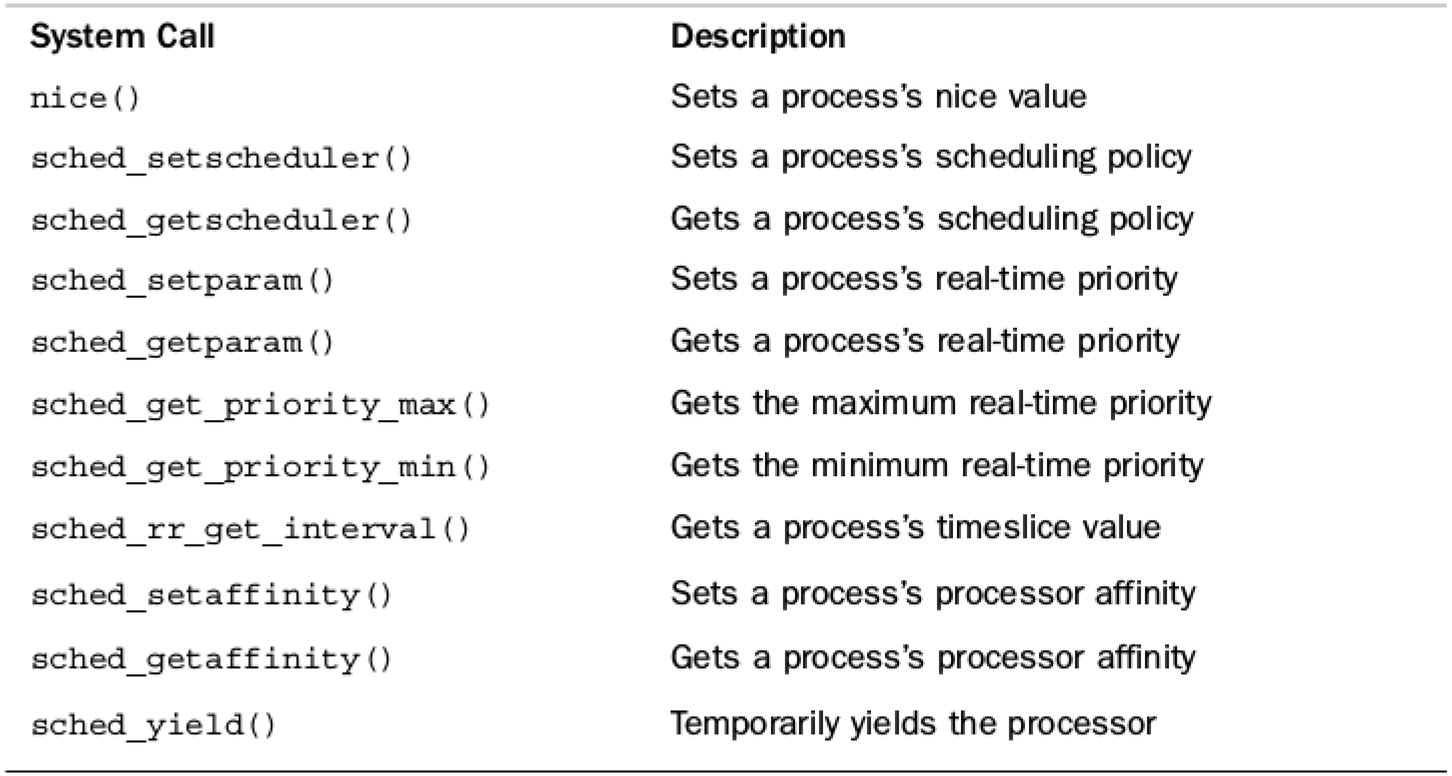


```c
 * Enqueue an entity into the rb-tree:
 */
static void __enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
        struct rb_node **link = &cfs_rq->tasks_timeline.rb_root.rb_node;
        struct rb_node *parent = NULL;
        struct sched_entity *entry;
        bool leftmost = true;

        /*
         * Find the right place in the rbtree:
         */
        while (*link) {
                parent = *link;
                entry = rb_entry(parent, struct sched_entity, run_node);
                /*
                 * We dont care about collisions. Nodes with
                 * the same key stay together.
                 */
                if (entity_before(se, entry)) {
                        link = &parent->rb_left;
                } else {
                        link = &parent->rb_right;
                        leftmost = false;
                }
        }

        rb_link_node(&se->run_node, parent, link);
        rb_insert_color_cached(&se->run_node,
                               &cfs_rq->tasks_timeline, leftmost);
}
```

Processor Affinity

- The Linux scheduler
  - Tries to provide soft or natural affinity by attempting to keep processes on the same processor.
  - Enforces hard processor affinity, enabling a user to say,“This task must remain on this subset of the available processors no matter what.”
- This hard affinity is stored as a bitmask
  - in the task’s task_struct as cpus_allowed .
  - contains one bit per possible processor on the system.
  - By default, all bits are set and, therefore, a process is potentially runnable on any processor.
  - The user, however, via sched_setaffinity(), can provide a different bit-mask of any combination of one or more bits. Likewise, the call sched_getaffinity() returns the current cpus_allowed bitmask.
- The kernel enforces hard affinity in a simple manner.
  - First, when a process is initially created, it inherits its parent’s affinity mask. Because the parent is running on an allowed processor, the child thus runs on an allowed processor.
  - Second, when a processor’s affinity is changed, the kernel uses the **migration threads** to push the task onto a legal processor.
  - Finally, the **load balancer** pulls tasks to only an allowed processor.

System calls to change scheduler parameters



System Call	Description
`nice()`	Sets a process's nice value
`sched_setscheduler()`	Sets a process's scheduling policy
`sched_getscheduler()`	Gets a process's scheduling policy
`sched_setparam()`	Sets a process's real-time priority
`sched_getparam()`	Gets a process's real-time priority
`sched_get_priority_max()`	Gets the maximum real-time priority
`sched_get_priority_min()`	Gets the minimum real-time priority
`sched_rr_get_interval()`	Gets a process's timeslice value
`sched_setaffinity()`	Sets a process's processor affinity
`sched_getaffinity()`	Gets a process's processor affinity
`sched_yield()`	Temporarily yields the processor

Remaining questions on linux scheduing

- When is the scheduler called?
- How often is the scheduler called?
- What is the overhead of running the scheduler?
- What are cgroups and are the relevant in scheduling decisions?
- What alternative is there to CFS? When is it useful?
- ....