Roll no: 2020201026
Topic: Linux Kernel 2.6 Scheduling algorithm

Linux 2.6 Scheduler [i]

Linux releases 2.6.0 to 2.6.22 used the O(1) kernel scheduler which could schedule processes within constant time regardless of how many processes are active in the operating system. This is an improvement over previously used O(n) schedulers and is superseded by Completely Fair Scheduler. This report focuses on the O(1) scheduler as it was used only in the Linux 2.6 version and thus is also known as Linux 2.6 scheduler.

Scheduling Policy [ii][iii]

Linux scheduling is based on the time-sharing technique where CPU time is divided into slices, one for each runnable process. If execution of a process is not complete but its time slice or quantum expires, a timer interrupt occurs and the process is switched with another one from the ready queue. In Linux, processes are assigned priorities dynamically and the scheduler keeps track of and adjusts these priorities periodically according to what the processes are doing such that no process is left behind. Linux has 3 classes of processes:

- Real-time: The deadlines of these processes must be met and thus they should not be blocked by any normal (low priority) process. Once a process is specified as real-time, it always remains real-time. It has priority between 0 and 99.
- Interactive: These processes constantly interact with the users and thus spend a lot of time waiting for input from user. Once the input is received, the process should wake up quickly (within 50 to 150ms).
- Batch: These processes do not need user interaction and often run in the background. They require more CPU time and so they are often penalized by the scheduler.

The Linux 2.6 scheduler implements a sophisticated heuristic algorithm based on the past behavior of the processes to decide whether a given process should be considered as interactive or batch and favors interactive processes. A process may be interactive for some time but then become a batch process. Conventional processes (interactive and batch) are given priorities between 100 and 139 where lowest number denotes higher priority. Every new process is assigned a static priority which it inherits from its parent.

Scheduling of Conventional Processes

Base time quantum [iv]

It is determined by the static priority of the process by the formula:

$$\underset{\text{(in milliseconds)}}{\textit{base time quantum}} = \begin{cases} (140 - \textit{static priority}) \times 20 & \text{if } \textit{static priority} < 120 \\ (140 - \textit{static priority}) \times 5 & \text{if } \textit{static priority} \geq 120 \end{cases}$$

120 is considered as the default static priority. The higher the static priority, the longer is the base time quantum.

Dynamic priority [v]

When selecting the new process to run, the scheduler looks for the dynamic priority of the process. It is related to the static priority by:

dynamic priority = max(100, min(static priority − bonus + 5, 139))

The bonus value ranges from 0 to 10, where value less than 5 indicates a penalty that lowers the dynamic probability, while a value greater than 5 indicates a premium that raises the dynamic priority. The bonus value depends on the history of the process, primarily on its average sleep time.

Average sleep time (sleep_avg) [vi][vii]

It is the average number of nanoseconds that the process spent while sleeping (presumably blocked on I/O). The average sleep time decreases while a process is running and it should not exceed MAX_SLEEP_AVG (usually 1 sec). The correspondence between average sleep times and bonus values is shown below.

Average sleep time	Bonus	Granularity
Greater than or equal to 0 but smaller than 100 ms	0	5120
Greater than or equal to 100 ms but smaller than 200 ms	1	2560
Greater than or equal to 200 ms but smaller than 300 ms	2	1280
Greater than or equal to 300 ms but smaller than 400 ms	3	640
Greater than or equal to 400 ms but smaller than 500 ms	4	320
Greater than or equal to 500 ms but smaller than 600 ms	5	160
Greater than or equal to 600 ms but smaller than 700 ms	6	80
Greater than or equal to 700 ms but smaller than 800 ms	7	40
Greater than or equal to 800 ms but smaller than 900 ms	8	20
Greater than or equal to 900 ms but smaller than 1000 ms	9	10
1 second	10	10

The scheduler also uses sleep_avg to determine if the process is interactive or batch. The process is considered interactive if the following formula is satisfied:

$\textit{dynamic priority} \leq 3 \times \textit{static priority} / 4 + 28$

That is: $\textit{bonus} - 5 \geq \textit{static priority} / 4 - 28$

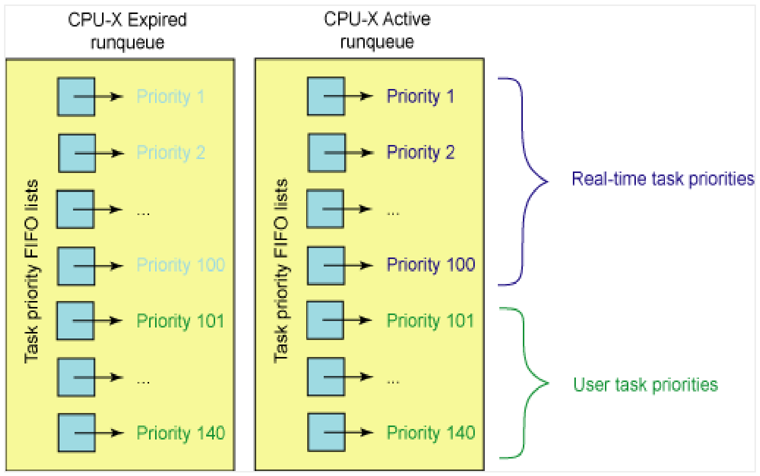
The expression ($\textit{static priority} / 4 - 28$) is known as the interactive delta. The higher a task's sleep_avg is, the higher its dynamic priority will be. It is far easier for high priority than for low priority processes to become interactive. For instance, a process having highest static priority (100) is considered interactive when its bonus value exceeds 2, that is, when its average sleep time exceeds 200ms. Conversely, a process having lowest static priority (139) is never considered as interactive, because the bonus value is always smaller than the value 11 required to reach an interactive delta equal to 6. A process having default static priority (120) becomes interactive as soon as its average sleep time exceeds 700ms.

Active and Expired Process [viii]

Even if conventional processes with higher priorities get larger slices of the CPU time, they should not completely lock out the processes having lower static priority. To avoid process starvation, when a process finishes its time quantum, it can be replaced by a lower priority process whose time quantum has not yet been exhausted. For implementing this mechanism, the scheduler keeps two disjoint sets of runnable processes:

- Active: Those runnable processes who have not yet exhausted their time quantum and are thus allowed to run.
- Expired: Those runnable processes who have exhausted their time quantum and are thus forbidden to run until all active processes expire.

RunQueue [ix][x][xi]



Runqueue lists link the process descriptors of all runnable processes. Each CPU in the system has its own runqueue and all runqueue structures are stored in the runqueues per-CPU variable. Every runnable process in the system belongs to only one runqueue. As long as a runnable process remains in the same runqueue, it can be executed only by the CPU to which that runqueue belongs. The arrays field of the runqueue consists of two prio_array_t structures, each representing a set of runnable processes and 140 doubly linked list heads (one list for each possible process priority), a priority bitmap, and a counter of the processes included in the set. All tasks on a CPU begin in one priority array, the active one, and as they run out of their timeslices they are moved to the expired priority array. During the move, a new timeslice is calculated. When there are no more runnable tasks in the active priority arrays, it is simply swapped with the expired priority array (which requires simply updating two pointers).

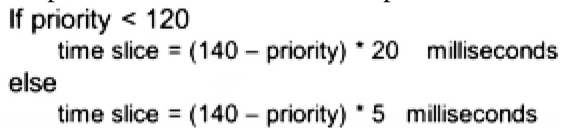
Selecting a process [xii][xiii][xiv]

As the system has 140 different priorities, both the active and expired arrays are of size 140 and they are served in FIFO order. When the 2.6 scheduler has to select the current highest priority process, it does not need to traverse the entire runqueue. Instead, it directly selects the first process in the current highest priority non-empty queue from the active array. The scheduler must first find the lowest numbered (highest priority) non-empty queue and then choose the first task from that queue. Kernel stores a bitmap of runqueues with non-zero entries and uses a special instruction to get the lowest index of set bit in the bitmap for choosing the corresponding queue. Dynamic priority is used to determine which runqueue the task should be placed in.

Updating timeslice [xv][xvi]

If the current process is a conventional process, the following operations are performed by the scheduler_tick() function:

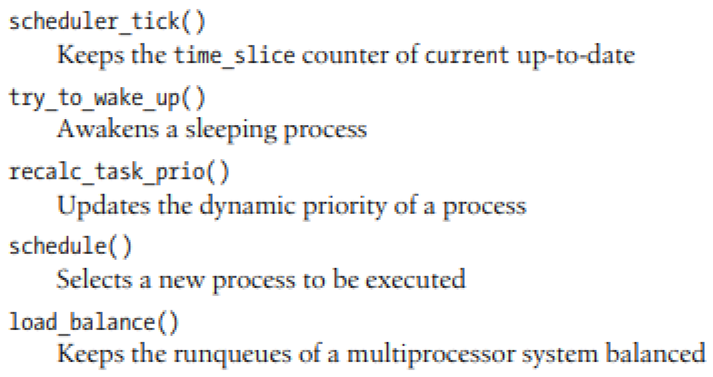
1. Decrements the timeslice counter
2. If its time quantum is exhausted, the process is removed from active queue for rescheduling. Dynamic priority of the process is recalculated and updated. Also time quantum of the process is refilled as:



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

   The process is then inserted into the corresponding expired queue.
3. If the time quantum is not exhausted, it is checked if the remaining time slice of the current process is too long. the time quantum of interactive processes with high priorities is split into several pieces of TIMESLICE_ GRANULARITY, so that they do not monopolize the CPU.

Functions Used by the Scheduler [xvii]



`scheduler_tick()`	Keeps the `time_slice` counter of `current` up-to-date
`try_to_wake_up()`	Awakens a sleeping process
`recalc_task_prio()`	Updates the dynamic priority of a process
`schedule()`	Selects a new process to be executed
`load_balance()`	Keeps the runqueues of a multiprocessor system balanced

Schedule() [xviii][xix]

1. The schedule() function once called, disables preemption.
2. It determines the for what amount of time the task to be scheduled out has been running and reduces its time quantum.
3. It then checks the interactivity credit status of the process and places it in the appropriate queue.
4. Now, to select a new task for running from the runnable processes in the runqueue.
   a. If the active runqueue is empty, then an attempt at load balancing is made. If balancing does not bring any runnable tasks, then a switch to the idle task is made. If there are runnable tasks in the runqueue but not in the active priority array, then the active and expired priority arrays are swapped.
   b. If there is a runnable task in the active priority array, its bitmap is checked to find the highest priority level with a runnable task.
      i. If there is a dependent sleeper, the current CPU switches to idle so the dependent sleeper can wake up and perform its task.
      ii. In either case, if it is not a real-time process and is woken up, it is given a slightly higher sleep_avg and its dynamic priority is recalculated.
5. The actual task switch is now made and pre-emption is again enabled.
6. It is checked if there were any interrupts received since pre-emption was disabled. If yes, then rescheduling is done again to service the interrupt.

[i] https://en.wikipedia.org/wiki/O(1)_scheduler
[ii] Daniel P. Bovet, Marco Cesati - Understanding the Linux Kernel, Third Edition-O'Reilly Media (2005)
[iii] Understanding the Linux 2.6.8.1 CPU Scheduler By Josh Aas
[iv] Daniel P. Bovet, Marco Cesati - Understanding the Linux Kernel, Third Edition-O'Reilly Media (2005)
[v] Scheduling in Linux by Prof. Chester Rebeiro - NPTEL
[vi] Daniel P. Bovet, Marco Cesati - Understanding the Linux Kernel, Third Edition-O'Reilly Media (2005)
[vii] Understanding the Linux 2.6.8.1 CPU Scheduler By Josh Aas
[viii] Daniel P. Bovet, Marco Cesati - Understanding the Linux Kernel, Third Edition-O'Reilly Media (2005)
[ix] Daniel P. Bovet, Marco Cesati - Understanding the Linux Kernel, Third Edition-O'Reilly Media (2005)
[x] https://www.programmersought.com/article/292157712/
[xi] Scheduling in Linux by Prof. Chester Rebeiro - NPTEL
[xii] Daniel P. Bovet, Marco Cesati - Understanding the Linux Kernel, Third Edition-O'Reilly Media (2005)
[xiii] https://www.programmersought.com/article/292157712/
[xiv] Scheduling in Linux by Prof. Chester Rebeiro - NPTEL
[xv] Daniel P. Bovet, Marco Cesati - Understanding the Linux Kernel, Third Edition-O'Reilly Media (2005)
[xvi] Understanding the Linux 2.6.8.1 CPU Scheduler By Josh Aas
[xvii] Daniel P. Bovet, Marco Cesati - Understanding the Linux Kernel, Third Edition-O'Reilly Media (2005)
[xviii] Daniel P. Bovet, Marco Cesati - Understanding the Linux Kernel, Third Edition-O'Reilly Media (2005)
[xix] Understanding the Linux 2.6.8.1 CPU Scheduler By Josh Aas