Completely Fair Scheduler and its tuning[1]

Jacek Kobus and Rafał Szklarski

1 Introduction

The introduction of a new, the so called completely fair scheduler (CFS) to the Linux kernel 2.6.23 (October 2007) does not mean that concepts needed to describe its predecessor, i.e. the O(1) scheduler, cease to be relevant. In order to be able to present the principles of the CFS scheduler and details of its algorithm the basic terminology must be introduced. This should allow to describe the salient features of the new scheduler within the context of the O(1) one. This approach should help the reader already acquainted with the O(1) scheduler to grasp the essential changes introduced to the scheduler subsystem.

All modern operating systems divide CPU cycles in the form of time quantas among various processes and threads (or Linux tasks) in accordance with their policies and priorities. This is possible because the system timer periodically generates interrupts and thus allows the scheduler to do its job of choosing the next task for execution. The frequency of the timer is hard-coded and can be changed at the configuration phase. For the 2.4 kernels this freqency was set to 100 Hz. In case of the 2.6 kernels the user can choose between the following values: 100, 250, 300 or 1000 Hz. The higher freqences mean the faster serving of interrupts and the better interactivity of the system at the expence of the higher system overhead. When there is no much work to be done in the system, i.e. when there are no tasks in the running queue, servicing hundreds of timer interrupts per second seems a waste of CPU cycles and energy. In the kernel 2.6.21 this problem was solved by implementing so called dynamic tics. If the system is idle some clock interrupts are ignored (skipped) and one can have as few as only six interrupts a second. In such a state the system responds only to other hardware interrupts.

2 Scheduling

Time measurement is one of the most important parts of the scheduler subsystem and in the CFS implementation has been improved by addition of the high-resolution timer. This kind of timer appeared in the kernel 2.6.16 and offers a nanosecond (instead of a milisecond) resolution which enables the scheduler to perform its actions with much finer granularity. In the O(1) scheduler every task is given its time share of the CPU of variable length which depends on the task's priority and its interactivity. This quanta can be used in its entirety unless there appears in the system a new task with a higher priority or the task yields (relinquishes) voluntarily the CPU waiting for initialized I/O operations to get finished. When the timeslice is completely used up the scheduler calculates the new time

[1]This article is based on R.Szklarski's M.S. thesis, Toruń, 2009.

quanta for the task and moves the task from the active queue to the expired one. The O(1) scheduler employs the concept of the epoch: when the active queue becomes eventually empty the scheduler swaps the active and expired queues and the new epoch begins. This mechanizm may lead to unacceptable long delays in serving the tasks because expired tasks have to wait (for unknown amount of time) for active tasks to consume their timeslices. When the number of task in the system increases the danger of starvation increases as well. In order to guarantee good interactivity of the system the O(1) scheduler marks some tasks as interactive and reinserts them into the active queue (but in order to prevent starvation this is only done if the tasks in the expired array have run recently).

The CFS scheduler calculates the timeslices for a given task just before it is scheduled for execution. Its length is variable (dynamic) and depends directly on its static priority and the current load of the running queue, i.e. the number and priority of the tasks in the queue.

In case of a general-purpose operating system its scheduler should take into account the fact that tasks may have very different character and exhibit different behaviour and therefore require adequate treatment by the scheduler. The Linux scheduler distinguishes between the real-time, batch and interactive processes.

The O(1) scheduler employs two group to tackle real-time tasks, namely SCHED_FIFO and SCHED_RR. SCHED_IDLE, SCHED_BATCH and SCHED_OTHER groups are used for clasification of the remaining tasks. In principle the CFS scheduler uses scheduling policies similar to the one used by the O(1). However, it introduces the notion of the class of tasks which is characterized by a special scheduling policy. The CFS scheduler defins the following classes: real-time (SCHED_FIFO and SCHED_RR), fair (SCHED_NORMAL and SCHED_BATCH) and idle (SCHED_IDLE). The idle class is very special since it is concerned with task being started when there are no more ready tasks in the system (every CPU has its own task of this sort). In this approach it is user's responsibility to classify properly his/hers tasks (using the *chrt* command) because the scheduler is not able to classify a task by itself.

In the case of the fair class tasks the CFS uses the following two polices:

`SCHED_NORMAL` – this policy is applied to normal tasks and it is also known as the SCHED_OTHER policy in the O(1) scheduler; SCHED_NORMAL replaced SCHED_OTHER in the kernel 2.5.28

`SCHED_BATCH` – the CFS scheduler does not allow these tasks to preempt `SCHED_NORMAL` tasks which results in a better utilization of the CPU cache. As a side effect it makes these tasks less interactive so that this policy should be used (as its very name suggests) for batch tasks.

It must be noted that the essential difference between the O(1) and CFS scheduler boils down to a completely new treatment of the `SCHED_NORMAL` policy. In one of the earliest commentaries on the new scheduler its author, Ingo Molnar, wrote (see Documentation/-scheduler/sched-design-CFS.txt):

80% of CFS's design can be summed up in a single sentence: CFS basically models an "ideal, precise multi-tasking CPU" on real hardware.

The inventor of the CFS set himself a goal of devising a scheduler capable of the fair devision of available CPU power among all tasks. If one had an ideal multitasking computer capable of concurrent execution on N processes then every process would get exactly 1/N-th of its available CPU power. In reality, on a single CPU machine, only one process can be executed at a time. That is why the CFS scheduler employs the notion of the virtual runtime which should be equal for all tasks. In reality the virtual runtime is a normalized value of the real runtime of a given task with its static priority taken into account. The scheduler selects the task with the smallest virtual runtime value to reduce in the long run the differences between the runtimes for all the tasks.

Real-time tasks are handeled by the CFS scheduler before tasks from other classes. Two different policies are used to this end, namely `SCHED_FIFO` and `SCHED_RR`. `SCHED_FIFO` tasks get no finite time quanta but execute until completion or yielding the CPU. On the contrary tasks from the `SCHED_RR` class get definite time quantas and are scheduled according to the round-robin algorithm (once all `SCHED_RR` tasks of a given prio level exhaust their timeslices, the timeslices are refilled and the tasks continue running).

Policies in Linux rely on priorities associated with every task. Priorities are used by the scheduler to rate the importance (or otherwise) of a given task and to calculate its CPU share. The O(1) scheduler employs the so called dynamical priority which, for a given task, is influenced by its static priority (set by the user) and its interactivity level. To account for the latter the scheduler records for every task its CPU time and the time it sleeps waiting for its I/O operations to be finished. The ratio of these two values is used as an indicator of the task interactivity. For I/O-bound proceses this indicator is small and increases as CPU consumption increases.

When devising the new scheduler Molnar decided to get rid of these statistics and subsequent modifications of the dynamic priorities of the running tasks. The new scheduler tries to divide CPU resources fairly among all processes by taking into account their static priorities. The handling of tasks's queues have also been changed. The O(1) scheduler uses (for every CPU in the system) two separate tables for handling active and passive tasks. When all the tasks are found in the passive table the tables are swaped and a new epoch begins (in fact this is achieved not by swapping the tables themselves but by swapping the corresponding pointers). The CFS scheduler used instead a single red-black binary tree which nodes hold task ready to be executed. The Linux scheduler (either old or new) uses priorities within the range [0..99] for handling real-time tasks and the [100..139] range for all other tasks. A new non real-time task gets by default the priority 120. This value can be modified (adjusted) by means of the static priority (called also the nice value) within the range $[-20.. + 19]$ by using the command *nice* (or *snice*). This means that the dynamic priority can be changed within the [100..139] range as needed. In this way the users can set the relative importance of tasks it owns and the superuser can change priorities of all the tasks in the system. The actual value of the dynamic priority of a given taks is modified by the O(1) scheduler itself witin ± 5 range depending

on the task's interactivity. The O(1) scheduler uses this mechanism to guarantee good responsiveness of the system by decreasing the dynamic priority value (i.e. weighting higher) of the iteractive tasks. This means that the behaviour of a task in a current epoch influences directly the interactivity level calculated by the scheduler and together with its static priority is used to classify the task as interactive or otherwise. This clasification has a heuristic character and is done according to a table of the form (see sched.c)

```
TASK_INTERACTIVE(-20): [1,1,1,1,1,1,1,1,1,0,0]
TASK_INTERACTIVE(-10): [1,1,1,1,1,1,1,0,0,0,0]
TASK_INTERACTIVE(  0): [1,1,1,1,0,0,0,0,0,0,0]
TASK_INTERACTIVE( 10): [1,1,0,0,0,0,0,0,0,0,0]
TASK_INTERACTIVE( 19): [0,0,0,0,0,0,0,0,0,0,0]
```

The responsibility of the CFS scheduler is to guarentee a fair division of the available CPU among all tasks from the fair class. This cannot result in dividing the epoch time equally among the tasks but the schedular must necessarily take into account weights of individual tasks which are related to their static priorities. To this end the CFS schedular uses the following equation:

$$\texttt{time\_slice} = \texttt{period}\frac{\texttt{task\_load}}{\texttt{cfs\_rq\_load}}, \tag{1}$$

where `time_slice` is a CPU timeslice that the task deserves (is due), `period` is the epoch length (the time span of a period depends on the number of tasks in the queue and some parameters that can be tuned by the administrator; see below), `task_load` – the last load and `cfs_rq_load` – the weight of the fair queue.

Although the algorithm for calculating timeslices is of utmost importance but the moment they are calculated is crucial as well. The O(1) scheduler evaluates a new timeslice for a task when the old time quota has been used up and subsequently then moves the tasks from the active queue to the expired one. The expired queue is where tasks wait for the active ones to consume their timeslices. Interactive tasks are reinserted into the active queue.

The CFS scheduler uses a notion of the so called dynamic time quanta which is calculated whenever a task is selected for execution. The actual length of this timeslice depends on the current queue load. Additionally, thanks to the high resolution timer, the newly established time quanta is determined within nanosecond accuracy. By means of the timer's handler the scheduler gets a chance to select a task that deserves CPU and – if needed – performs the context switch. The CFS scheduler tries to fairly distribute the CPU time of a given epoch among all tasks (in the fair class) in such a way that each of them gets the same amount of the virtual runtime. For heavy tasks their virtual runtime increases slowly; for light tasks the behaviour is just the opposite. The CFS scheduler selects the task which has the smallest value of the virtual runtime. Let us assume that we have only two tasks with extreme priorities, i.e. one with the priority 100 (the heavy task) and the other with priority 139 (the light task). According to eq. 1 the former enjoys the longer timeslices than the latter. If the scheduler based its selection decision on their

real runtimes then the light task would get small quanta one after the other (in a file) until the real runtime of the heavy task would get smaller than the time used by the lighter task. Within this approach the more important task would get CPU less frequently but in larger chuncks. Thanks to the selection process based on the normalized virtual runtime the CFS scheduler divides the CPU fairly which results in execution of every task once per epoch. If a new task is added to the run queue during a current epoch then the scheduler start imediately the new one.

A normal task experiences the following when the CFS scheduler is at work.

1. The timer generates an interrupt and the scheduler (according to its configuration) selects a task which

   - has the lowest virtual runtime
   - has just changed its status from waiting to running

   The dynamic timeslice is calculated according to eq. 1 for the selected task. If the high resolution timer is used then it is set accordingly. Next, the scheduler switches the context to allow the newly selected task to execute.

2. The task starts using CPU (executing).

3. During the next timer interrupt (sometimes also during a system call) the scheduler examines the current state of affairs and can preempt the task if

   - it used all its timeslice
   - there is a task with smaller virtual runtime in the tree
   - there is a newly created task
   - there is a task that has completed its sleeping stage

   If this is the case

   (a) the scheduler updates the virtual runtime according to the real runtime and the load

   (b) the scheduler adds the task into red-black tree. Then the tasks are ordered according to the key equal to `current_vruntime` − `min_vruntime`, where `current_runtime` is the virtual runtime of the current task and `min_vruntime` – the smallest virtual runtime within the nodes on the tree

3 CFS scheduler tuning

Te CFS scheduler offers a number of parameters which allow to tune its behaviour to actual needs. These parameters can be accessed via the proc file system, i.e. a special file system that exists only in memory and forms an interface between the kernel and the user

space. The content of any of its files is generated when a read operation is being performed. The kernel uses files in /proc both to export its internal data to user space applications and to modify kernel parameters, the CFS scheduler parameters including. This can be done by using the *sysctl* command or by treating the files as the ordinary text files which can be read or written. Thus to get the content of the /proc/sys/kernel/sched_latency_ns file it is necessary to use one of the following commands:

```
# cat /proc/sys/kernel/sched_latency_ns
# sysctl sched_latency_ns
```

To modify its content one can use:

```
# echo VALUE > /proc/sys/kernel/sched_latency_ns
# sysctl -w  sched_latency_ns=VALUE
```

Now, let us describe briefly some of the tunable kernel parameters of the CFS scheduler. All of them can be found in the /proc/sys/kernel directory in files with names exactly matching the names of these parameters.

sched_latency_ns – epoch duration (length) in nanoseconds (20 ms by default)

sched_min_granularity_ns – granularity of the epoch in nanoseconds (4 ms by default); the epoch length is always equal to the multiplicity of the value of this parameter.

The scheduler checks if the following inequality holds:

$$\texttt{nr\_running} > \frac{\texttt{sched\_latency\_ns}}{\texttt{sched\_min\_granularity\_ns}},$$

where `nr_running` is equal to the number of tasks in the queue. If this relation is satisfied then the scheduler realizes that there are too many task in the system and that the epoch duration has to be increased. It is done according to the equation

$$\texttt{period} = \texttt{sched\_min\_granularity\_ns} \cdot \texttt{nr\_running},$$

i.e the length of the epoch is equal to the granularity multiplied by the number of `TASK_RUNNING` tasks from the fair class. This means that for the default values of the schedular parameters the epoch would be increased when the number of tasks in `TASK_RUNNING` state is larger then five. Otherwise the epoch length would be set to 20 ms.

sched_child_runs_first – this parameter influences the order of execution of the parent process and its child. If it is equal to one (the default value), then the child process will get CPU prior to its parent. If the parameter is zero – the reverse will happen.

sched_compat_yield – this parameter is used to block the current process from yielding voluntarily CPU by setting its value to zero (the default). The process yields the CPU by calling the `sched_yield()` function.

sched_wakeup_granularity_ns – this parameter describes the ability of tasks being waken up to preempt the current task (5 ms by default). The larger the value the more difficult it is for the task to force the preemption.

sched_migration_cost – this parameter is used by the scheduler to determine whether the procedure to select the next task should be called when a task is being waken up. If the mean real runtime for the current task and the one being waken up are smaller than this parameter (0.5 ms by default), the scheduler chooses another task. To make this work the `WAKEUP_OVERLAP` feature must be active.

In case of a multiprocesor computer system the scheduler tries also to spread load over available CPUs. In this respect the scheduler uses the following parameters:

sched_migration_cost – the cost of task migration among CPUs (0.5 ms by default) which is used to estimate whether the code of a task is still present in a CPU cache. If the real runtime of the task is smaller than the values of this parameter then the scheduler assumes that it is still in the cache and tries to avoid moving the task to another CPU during the load balancing procedure.

sched_nr_migrate – the maximum number of task the scheduler handles during the load balancing procedure (32 by default).

The CFS scheduler makes it possible to control the time usage of real-time tasks up to microsecond resolution:

sched_rt_runtime_us – the maximum CPU time that can be used by all the real-time tasks (1 second by default). When this amount of time is used up these tasks must wait for **sched_rt_period_us** period before the are allowed to be executed again.

sched_rt_period_us – the CFS scheduler waits this amount of time (0.95 s by default) before scheduling any of the real-time tasks again.

The last parameter, namely **sched_features**, represents certain features of the scheduler that can be switched on and off during the system operation and they are described in a separate subsection (see below).

3.1 Scheduler features

The CFS scheduler gives a system's administrator opportunity to modify its operation by turning on and off particular features which can be found in the /sys/kernel/debug/-sched_features file. When the content of this file is displayed one gets the list of all the features available. Some of the names may be prefixed by 'NO' which means that the given feature has been switched off and is therefore not active. The execution of the following commands

```
echo FEATURE_NAME    > sched_features
echo NO_FEATURE_NAME > sched_features
```

results first in activating the feature FEATURE_NAME and its subsequent deactivation. The description of the most important ones follows:[2]

NEW_FAIR_SLEEPERS – the feature relevant when a task is waken up. Every tasks that is waken up gets the virtual runtime equal to the smallest virtual runtime among the tasks in the queue. If this feature is on then this virtual runtime is additionally decreased by sched_latency_ns. However, the new value cannot be smaller than the virtual runtime of the task prior to its suspension.

NORMALIZED_SLEEPER – this feature is taken into account when the NEW_FAIR_SLEEPERS feature is active. By switching on the NORMALIZED_SLEEPER feature the value sched_latency_ns gets normalized (see above). The normalization is performed similar to the normalization of the virtual runtime.

WAKEUP_PREEMPT – if this feature is active then the task which is waken up immediately preempts the current task. Swiching off this feature casuses the scheduler to block the possibility of preemption of the current task until it uses up its time slice. Moreover, if this feature is active and there is only one running task in the system then the scheduler will refrain from calling the procedure responsible for the selection of the next task untill additional task is added to the queue.

START_DEBIT – this feature is taken into acount during initialization of virtual runtime for a newly created task. If this feature is active then the initial virtual runtime is increased by the amount equal to the normalized timeslice that the task would be allowed to use.

SYNC_WAKEUPS – this feature allows for synchronous wakening up of tasks. If active the scheduler preempts the current task and schedules the one being waken up (WAKEUP_PREEMPT must be active).

HRTICK – this feature is used to switch on and off the high resolution timer.

DOUBLE_TICK – this feature enables to control whether the procedure checking the CPU time consumend by the task is being called when the timer interrupt occurs. If active the interrupts generated by the high resolution timer or the system timer cause the scheduler to check if the task should be preempted. If not active, then only the interrupts generated by the high resolution timer results in the time checking.

ASYM_GRAN – this feature is related to preemption and influences the granularity of wake ups. If active the scheduler sched_wakeup_granularity_ns value is normalized (the normalization is performed similar to the normalization of the virtual runtime).

[2]We bypass the features related to the scheduler operation on a multiprocessor machine and grouping. See Documentation/scheduler/sched-design-CFS.txt for details.