Operating Systems 2

Process Scheduling, part 2

Arkadiusz Chrobot

Department of Computer Science

March 16, 2020

Outline

O(1) Scheduler Drawbacks

The O(1) Scheduler has some drawbacks inherited from multi-level queuing or more precisely multilevel feedback queue scheduling:

- time slices associated with priorities are invariable, which means that if there are only two low-priority processes in the system then they will be able to run uninterrupted only for a very short period of time and they will be preempted very often,
- the granulation of the time slices can be insufficient, i.e the length of time slices allocated for two high-priority processes (for example -20 and -19) is similar, but time slices of two low-priority processes (for example 18 and 19) differ much,
- the measurement of time slice consumption is not precise,
- heuristics used for measuring the interactivity level of a process is not tampering-proof, which allows the processes to gain more CPU time than they really require.

O(1) Scheduler Drawback

Some of those disadvantages were alleviated in the O(1) Scheduler, but eradicating them showed up to be impossible. That is why the Linux kernel programmers decided to rework the scheduler in the version 2.6.23 of the kernel.

Scheduling Classes

One of the most important additions to the new scheduler are the structures of the type `struct sched_class` called *scheduling classes* which represent a scheduling policy applied to a specified group of processes. Each such a structure contains a set of function pointers that point to functions performing the following activities according to a specific policy:

`enqueue_task()` adds a process to the run queue,

`dequeue_task()` removes a process from the run queue,

`yield_task()` allows a process to relinquish the CPU,

`check_preempt_curr()` checks if the current process has to be preempted by the process that just woken up,

`pick_next_task()` chooses the next process to run,

`put_prev_task()` takes a part in the context switching,

`set_curr_task()` invoked when the scheduling policy of the current process is changed,

`new_task()` responsible for allocating the CPU for new processes.

Scheduling Classes

Scheduling classes handles the following policies:

SCHED_FIFO	real-time processes scheduled with the use of the FCFS algorithm,
SCHED_RR	real-time processes scheduled with the use of the round-robin algorithm,
SCHED_DEADLINE	real-time processes scheduled with the use of the EDF (Earliest Deadline First) algorithm; this policy has been introduced in the 3.14 version of the kernel,
SCHED_NORMAL	regular processes scheduled by the CFS algorithm; this policy corresponds to the SCHED_OTHER policy from the POSIX standard,
SCHED_BATCH	scheduling policy for a low-priority, CPU-bound processes; it is handled by the CFS scheduler,
SCHED_IDLE	scheduling policy for low-priority processes which are run when no other process is ready to run; also handled by the CFS algorithm.

Scheduling Classes

Scheduling classes are linked together in a list, starting with classes for the highest priority processes (the real-time ones) to the lowest priority (the batch and idle processes). The `schedule()` function traverses the list calling the `pick_next_task()` function (method) for each of the class. The one that returns a non-`NULL` value has chosen the next process to run. It is worth to notice that the scheduling classes are one of the several examples of applying the concept of Object-Oriented Programming in the Linux kernel, although the kernel itself is written in plain C, not in C++.

Scheduling Entities

In the 2.6.23 kernel version another important structure of the type `struct sched_entity` was introduced. This structure allows the kernel to schedule not only a single process but a group of such processes. More generally — it allows scheduling so-called *scheduling entities*. Such structures are new members of each process descriptor. An example of group of processed scheduled together is the `rt_bandwidth` group for real-time processes. It assumes that 95% of each second of the processor time is alloted to the real-time processes and the 5% for the regular processes. That ratio can be changed by the system administrator. The group has been introduced in the 2.6.23 version of the kernel, to prevent monopolizing the CPU by the SCHED_FIFO processes.

Starting from the version 2.6.23 of the kernel, priorities of **all** processes are static, with one exception. The priority of a regular process can be temporally boosted to the real-time priority, when the process invokes a system call that uses the so-called RT-mutex. This is to prevent the *priority inversion* problem.

Fair Scheduling — Introduction

The Fair Scheduling is about providing for each of the processes a *fair share* of the CPU computing power. To better understand how it works let's consider a *perfectly multitasking processor*. When such a CPU has to run one and only one process it allocates 100% of its power to the process. In case when it has to run n identical processes it allocates to each of them $\frac{1}{n}$ of its power. As a consequence all processes runs $n\times$ slower than a single process, but still they are performed simultaneously, and without unnecessary breaks. Unfortunately, this scenario cannot be implemented with the use of real-life processors. However, the CPU can be allocated to the process basing on the information of how long it *haven't been allowed to use* the CPU. If there is a single process in the system it can get the CPU for as long as it needs, but when another process becomes ready it immediately preempts the first one, because it used much less of the CPU computing power.

Fair Scheduling — Introduction

Let’s consider another scenario in which two identical processes has to be scheduled at the same time. The scheduler can calculate the time of running (the time when each process has assigned the CPU) of each of the processes by assuming a *targeted latency* and allocating a share of it to each of them. The targeted latency is a short period of time, typically several microseconds. However, it has to be longer then the time needed to switch processes. It should be noted, that extending the scenario to n processes leads to an issue. When the number of processes approaches to infinity to time they are allowed to use the CPU goes to zero. Therefore some bottom limit for that time has to be defined and it is called the *minimum granularity*. In real-life systems some of the processes are more important than the others, which is expressed by their priorities. In the fair scheduling the priorities are converted to *weights* which are used by the scheduler to compute the portions of the targeted latency for each of the processes.

Completely Fair Scheduler

The Completely Fair Scheduler (the CFS for short) has replaced the O(1) Scheduler in the Linux kernel. It is authored by Ingo Molnar, who was inspired by the ideas of Con Kolivas, an Australian kernel programmer. The change was introduced to address some issues with scheduling interactive processes for desktop computers. As the name suggests the scheduler implements fair scheduling, although it *is not* completely fair if the number of ready-to-run processes is large. Fortunately it is a very rare scenario.

The CFS is implemented in the `kernel/sched_fair.c` file. It utilizes two 40-elements arrays to convert priorities to weights and weights to priorities. The first one is named `prio_to_weight`. The weight for the default priority (the nice level equal 0) is set to 1024. The weights of processes of higher priorities are computed by sequentially multiplying this value by 1.25. The weights for lower priorities are calculated by sequentially dividing default weight by the 1.25. The other array is called `prio_to_wmul` and it stores the inverses of the weights.

Completely Fair Scheduler

The processes are scheduled according to their *virtual runtime* which is an actual runtime weighted by the by the number of ready-to-run processes. The process with the shortest virtual runtime gets the CPU as next. The virtual runtime is measured in nanoseconds and stored in the `vruntime` member of the `se` field of the process descriptor. This field is a structure of the `struct sched_entinty` type. The value of the `vruntime` member is updated periodically and after some events by the `update_curr()` function. The targeted latency is stored in the variable of the name `sched_latency_ns` and is set by default to $20ms$. This value can be changed by system administrator. The maximal number of processes that has to be scheduled in that period of time is stored in the `sched_nr_latency` and its updated by the kernel. The minimal amount of time (the bottom limit) in which each process is allowed to run is set to $1ms$.

Completely Fair Scheduler

The run queue for the CFS is actually a red-black tree. It is a type of binary search tree in which each node has an additional property that is called a colour. The collocation of colours in that tree is governed by the following rules:

1. The root of the tree is always black.
2. Each node is either black or red.
3. Children of the red node are always black.
4. Leafs are always black.
5. Every simple path from a given node to its descendant leaf goes through the same number of black nodes.

If all those rules are fulfilled, it means the tree is balanced. When one of them is not satisfied, which is a consequence of adding or removing a node from the tree, then the balance has to be restored by left and right rotating some of the subtrees or changing colours of several nodes.

Completely Fair Scheduler

Linux kernel has its own generic implementation of a red-black tree (see the third instruction for the laboratory classes; for more details on the red-black trees see the "Introduction to Algorithms" book by T. H. Cormen et al.). The CFS uses this implementation to sort the processes according to their virtual runtime. The leftmost node in the tree specifies the process with the shortest virtual runtime. If its even shorter than the virtual runtime of the current process than process denoted by the leftmost node of the tree preempts the current process. Locating the leftmost node in the red-black tree takes $O(log_2(n))$ time, where n is the number of ready-to-run processes. To speed up finding the node the kernel function responsible for adding a new node to the tree sets a special pointer when it inserts the leftmost node. Detecting such a case is quite easy: if the function always takes the left branch while traversing the tree to insert a new node, then it means that the new node is the leftmost one.

Completely Fair Scheduler

If the CFS scheduler finds the pointer to be `NULL` then it means the SCHED_NORMAL policy class is empty and it should move to another class (SCHED_BATCH).

Just like the O(1) Scheduler, the CFS tries to run the new child process before its parent. To achieve the goal it sometimes swaps virtual runtimes of both processes.

It takes the CFS longer to chose the next process to run, when compared with the O(1) Scheduler. However, the CFS is more fair as it goes to the scheduling the interactive processes. That's why it has replaced the latter in Linux kernel.

Questions

?

The End

Thank You for Your attention!