Load tracking in the scheduler

April 15, 2015

This article was contributed by Preeti U Murthy.

The scheduler is an essential part of an operating system, tasked with, among other things, ensuring that processes get their fair share of CPU time. This is not as easy as it may seem initially. While some processes perform critical operations and have to be completed at high priority, others are not time-constrained. The former category of processes expect a bigger share of CPU time than the latter so as to finish as quickly as possible. But how big a share should the scheduler allocate to them?

Another factor that adds to the complexity in scheduling is the CPU topology. Scheduling on uniprocessor systems is simpler than scheduling on the multiprocessor systems that are more commonly found today. The topology of CPUs is only getting more complex with hyperthreads and heterogeneous processors like the big.LITTLE taking the place of symmetric processors. Scheduling a process on the wrong processor can adversely affect its performance. Thus, designing a scheduling algorithm that can keep all processes happy with the computing time allocated to them can be a formidable challenge.

The Linux kernel scheduler has addressed many of these challenges and matured over the years. Today there are different scheduling algorithms (or "scheduling classes") in the kernel to suit processes having different requirements. The Completely Fair Scheduling (CFS) class is designed to suit a majority of today's workloads. The realtime and deadline scheduling classes are designed for latency-sensitive and deadline-driven processes respectively. So we see that the scheduler developers have answered a range of requirements.

The Completely Fair Scheduling class

The CFS class is the class to which most tasks belong. In spite of the robustness of this algorithm, an area that has always had scope for improvement is process-load estimation.

If a CPU is associated with a number *C* that represents its ability to process tasks (let's call it "capacity"), then the load of a process is a metric that is expressed in units of *C*, indicating the number of such CPUs required to make satisfactory progress on its job. This number could also be a fraction of *C*, in which case it indicates that a single such CPU is good enough. The load of a process is important in scheduling because, besides influencing the time that a task spends running on the CPU, it helps to estimate overall CPU load, which is required during load balancing.

The question is how to estimate the load of a process. Should it be set statically or should it be set dynamically at run time based on the behavior of the process? Either way, how should it be calculated? There have been significant efforts at answering these questions in the recent past. As a consequence, the number of load-tracking metrics has grown significantly and load estimation itself has gotten quite complex. This landscape appears quite formidable to reviewers and developers of CFS. The aim of this article is to bring about clarification on this front.

Before proceeding, it is helpful to point out that the granularity of scheduling in Linux is at a thread level and not at a process level. However, the scheduling jargon for thread is "task." Hence throughout this article the term "task" has been used to mean a thread.

Scheduling entities and task groups

The CFS algorithm defines a time duration called the "scheduling period," during which every runnable task on the CPU should run at least once. This way no task gets starved for longer than a scheduling period. The scheduling period is divided among the tasks into time slices, which are the maximum amount of time that a task runs within a scheduling period before it gets preempted. This approach may seem to avoid task starvation at first. However it can lead to an undesirable consequence.

Linux is a multi-user operating system. Consider a scenario where user A spawns ten tasks and user B spawns five. Using the above approach, every task would get ~7% of the available CPU time within a scheduling period. So user A gets 67% and user B gets 33% of the CPU time during their runs. Clearly, if user A continues to spawn more tasks, he can starve user B of even more CPU time. To address this problem, the concept of "group scheduling" was introduced in the scheduler, where, instead of dividing the CPU time among tasks, it is divided among groups of tasks.

In the above example, the tasks spawned by user A belong to one group and those spawned by user B belong to another. The granularity of scheduling is at a group level; when a group is picked to run, its time slice is further divided between its tasks. In the above example, each group gets 50% of the CPU's time and tasks within each group divide this share further among themselves. As a consequence, each task in group A gets 5% of the CPU and each task in group B gets 10% of the CPU. So the group that has more tasks to run gets penalized with less CPU time per task and, more importantly, it is not allowed to starve sibling groups.

Group scheduling is enabled only if `CONFIG_FAIR_GROUP_SCHED` is set in the kernel configuration. A group of tasks is called a "scheduling entity" in the kernel and is represented by the `sched_entity` data structure:

```
struct sched_entity {
    struct load_weight load;
    struct sched_entity *parent;
    struct cfs_rq *cfs_rq;
    struct cfs_rq *my_rq;
    struct sched_avg avg;
    /* ... */
};
```

Before getting into the details of how this structure is used, it is worth considering how and when groups of tasks are created. This happens under two scenarios:

1. Users may use the control group ("cgroup") infrastructure to partition system resources between tasks. Tasks belonging to a cgroup are associated with a group in the scheduler (if the scheduler controller is attached to the group).

2. When a new session is created through the `set_sid()` system call. All tasks belonging to a specific session also belong to the same scheduling group. This feature is enabled when `CONFIG_SCHED_AUTOGROUP` is set in the kernel configuration.

Outside of these scenarios, a single task becomes a scheduling entity on its own. A task is represented by the `task_struct` data structure:

```
struct task_struct {
    struct sched_entity se;
    /* ... */
};
```

Scheduling is always at the granularity of a `sched_entity`. That is why every `task_struct` is associated with a `sched_entity` data structure. CFS also accommodates nested groups of tasks. Each scheduling entity contains a run queue represented by:

```
struct cfs_rq {
    struct load_weight load;
    unsigned long runnable_load_avg;
    unsigned long blocked_load_avg;
    unsigned long tg_load_contrib;
    /* ... */
};
```

Each scheduling entity may, in turn, be queued on a parent scheduling entity's run queue. At the lowest level of this hierarchy, the scheduling entity is a task; the scheduler traverses this hierarchy until the end when it has to pick a task to run on the CPU.

The parent run queue on which a scheduling entity is queued is represented by `cfs_rq`, while the run queue that it owns is represented by `my_rq` in the `sched_entity` data structure. The scheduling entity gets picked from the `cfs_rq` when its turn arrives, and its time slice gets divided among the tasks on `my_rq`.

Let us now extend the concept of group scheduling to multiprocessor systems. Tasks belonging to a group can be scheduled on any CPU. Therefore it is not sufficient for a group to have a single scheduling entity; instead, every group must have one scheduling entity for each CPU. Tasks belonging to a group must move between the run queues in these per-CPU scheduling entities only, so that the footprint of the task is associated with the group even during task migrations. The data structure that represents scheduling entities of a group across CPUs is:

```
struct task_group {
    struct sched_entity **se;
    struct cfs_rq **cfs_rq;
    unsigned long shares;
    atomic_long_t load_avg;
    /* ... */
};
```

For every CPU `c`, a given task_group `tg` has a `sched_entity` called `se` and a run queue `cfs_rq` associated with it. These are related as follows:

```
tg->se[c] = &se;
tg->cfs_rq[c] = &se->my_rq;
```

So when a task belonging to tg migrates from CPUx to CPUy, it will be dequeued from `tg->cfs_rq[x]` and enqueued on `tg->cfs_rq[y]`.

Time slice and task load

The concept of a time slice was introduced above as the amount of time that a task is allowed to run on a CPU within a scheduling period. Any given task's time slice is dependent on its priority and the number of tasks on the run queue. The priority of a task is a number that represents its importance; it is represented in the

kernel by a number between zero and 139. The lower the value, the higher the priority. A task that has a stricter time requirement needs to have higher priority than others.

But the priority value by itself is not helpful to the scheduler, which also needs to know the load of the task to estimate its time slice. As mentioned above, the load must be the multiple of the capacity of a standard CPU that is required to make satisfactory progress on the task. Hence this priority number must be mapped to such a value; this is done in the array `prio_to_weight[]`.

A priority number of 120, which is the priority of a normal task, is mapped to a load of 1024, which is the value that the kernel uses to represent the capacity of a single standard CPU. The remaining values in the array are arranged such that the multiplier between two successive entries is ~1.25. This number is chosen such that if the priority number of a task is reduced by one level, its gets 10% higher share of CPU time than otherwise. Similarly if the priority number is increased by one level, the task will get a 10% lower share of the available CPU time.

Let us consider an example to illustrate this. If there are two tasks, A and B, running at a priority of 120, the portion of available CPU time given to each task is calculated as:

```
1024/(1024*2) = 0.5
```

However if the priority of task A is increased by one level to 121, its load becomes:

```
(1024/1.25) = ~820
```

(Recall that higher the number, lesser is the load). Then, task A's portion of the CPU becomes:

```
820/(1024+820)) = ~0.45
```

while task B will get:

```
(1024/(1024+820)) = ~0.55
```

This is a 10% decrease in the CPU time share for Task A.

The load value of a process is stored in the `weight` field of the `load_weight` structure (which is, in turn, found in `struct sched_entity`):

```
struct load_weight {
    unsigned long weight;
};
```

A run queue (`struct cfs_rq`) is also characterized by a "weight" value that is the accumulation of weights of all tasks on its run queue.

The time slice can now be calculated as:

```
time_slice = (sched_period() * se.load.weight) / cfs_rq.load.weight;
```

where `sched_period()` returns the scheduling period as a factor of the number of running tasks on the CPU. We see that the higher the load, the higher the fraction of the scheduling period that the task gets to run on the CPU.

Runtime and task load

We have seen how long a task runs on a CPU when picked, but how does the scheduler decide which task to pick? The tasks are arranged in a red-black tree in increasing order of the amount of time that they have spent running on the CPU, which is accumulated in a variable called `vruntime`. The lowest `vruntime` found in the queue is stored in `cfs_rq.min_vruntime`. When a new task is picked to run, the leftmost node of the red-black tree is chosen since that task has had the least running time on the CPU. Each time a new task forks or a task wakes up, its `vruntime` is assigned to a value that is the maximum of its last updated value and `cfs_rq.min_vruntime`. If not for this, its `vruntime` would be very small as an effect of not having run for a long time (or at all) and would take an unacceptably long time to catch up to the `vruntime` of its sibling tasks and hence starve them of CPU time.

Every periodic tick, the `vruntime` of the currently-running task is updated as follows:

```
vruntime += delta_exec * (NICE_0_LOAD/curr->load.weight);
```

where `delta_exec` is the time spent by the task since the last time `vruntime` was updated, `NICE_0_LOAD` is the load of a task with normal priority, and `curr` is the currently-running task. We see that `vruntime` progresses slowly for tasks of higher priority. It has to, because the time slice for these tasks is large and they cannot be preempted until the time slice is exhausted.

Per-entity load-tracking metrics

The load of a CPU could have simply been the sum of the load of all the scheduling entities running on its run queue. In fact, that was once all there was to it. This approach has a disadvantage, though, in that tasks are associated with load values based only on their priorities. This approach does not take into account the nature of a task, such as whether it is a bursty or a steady task, or whether it is a CPU-intensive or an I/O-bound task. While this does not matter for scheduling within a CPU, it does matter when load balancing across CPUs because it helps estimate the CPU load more accurately. Therefore the per-entity load tracking metric was introduced to estimate the nature of a task numerically. This metric calculates task load as the amount of time that the task was runnable during the time that it was alive. This is kept track of in the `sched_avg` data structure (stored in the `sched_entity` structure):

```
struct sched_avg {
    u32 runnable_sum, runnable_avg_period;
    unsigned long load_avg_contrib;
};
```

Given a task `p`, if the `sched_entity` associated with it is `se` and the `sched_avg` of `se` is `sa`, then:

```
sa.load_avg_contrib = (sa.runnable_sum * se.load.weight) / sa.runnable_period;
```

where `runnable_sum` is the amount of time that the task was runnable, `runnable_period` is the period during which the task *could* have been runnable.

Therefore `load_avg_contrib` is the fraction of the time that the task was ready to run. Again, the higher the priority, the higher the load.

So tasks showing peaks of activity after long periods of inactivity and tasks that are blocked on disk access (and thus non-runnable) most of the time have a smaller `load_avg_contrib` than CPU-intensive tasks such as code doing matrix multiplication. In the former case, `runnable_sum` would be a fraction of the `runnable_period`. In the latter, both these numbers would be equal (i.e. the task was runnable throughout the time that it was alive), identifying it as a high-load task.

The load on a CPU is the sum of the `load_avg_contrib` of all the scheduling entities on its run queue; it is accumulated in a field called `runnable_load_avg` in the `cfs_rq` data structure. This is roughly a measure of how heavily contended the CPU is. The kernel also tracks the load associated with blocked tasks. When a task gets blocked, its load is accumulated in the `blocked_load_avg` metric of the `cfs_rq` structure.

Per-entity load tracking in presence of task groups

Now what about the `load_avg_contrib` of a scheduling entity, `se`, when it is a *group* of tasks? The `cfs_rq` that the scheduling entity owns accumulates the load of its children in `runnable_load_avg` as explained above. From there, the parent task group of `cfs_rq` is first retrieved:

```
tg = cfs_rq->tg;
```

The load contributed by this `cfs_rq` is added to the load of the task group `tg`:

```
cfs_rq->tg_load_contrib = cfs_rq->runnable_load_avg + cfs_rq->blocked_load_avg;
tg->load_avg += cfs_rq->tg_load_contrib;
```

The `load_avg_contrib` of the scheduling entity `se` is now calculated as:

```
se->avg.load_avg_contrib =
      (cfs_rq->tg_load_contrib * tg->shares / tg->load_avg);
```

Where `tg->shares` is the maximum allowed load for the task group. This means that the load of a `sched_entity` should be a fraction of the shares of its parent task group, which is in proportion to the load of its children.

`tg->shares` can be set by users to indicate the importance of a task group. As is clear now, both the `runnable_load_avg` and and `blocked_load_avg` are required to estimate the load contributed by the task group.

There are still drawbacks in load tracking. The load metrics that are currently used are not CPU-frequency invariant. So if the CPU frequency increases, the load of the currently running task may appear smaller than otherwise. This may upset load-balancing decisions. The current load-tracking algorithm also falls apart in a few places when run on big.LITTLE processors. It either underestimates or overestimates the capacity of these processors. There are efforts ongoing to fix these problems. So there is good scope for improving the load-tracking heuristics in the scheduler. Hopefully this writeup has laid out the basics to help ease understanding and reviewing of the ongoing improvements on this front.

Index entries for this article

Kernel	Scheduler/Load tracking
GuestArticles	Murthy, Preeti U

Load tracking in the scheduler

Posted Apr 16, 2015 0:32 UTC (Thu) by **WanpengLi** (guest, #89964) [Link]

Good summary.

Load tracking in the scheduler

Posted Apr 21, 2015 15:04 UTC (Tue) by **bristot** (guest, #61569) [Link]

Nice Article! it is really easy to read and understand, despite of the complexity of the subject. As a suggestion, a set of charts explaining the abstractions will certainly make the article easier to understand.

Reply to this comment

Load tracking in the scheduler

Posted Apr 11, 2017 2:41 UTC (Tue) by **nixer** (guest, #103597) [Link]

Very informative and nice to read, thx

Reply to this comment