Schedulers: the plot thickens

[Posted April 17, 2007 by corbet]

The RSDL scheduler (since renamed the staircase deadline scheduler) by Con Kolivas was, for a period of time, assumed to be positioned for merging into the mainline, perhaps as soon as 2.6.22. Difficulties with certain workloads made the future of this scheduler a little less certain. Now Con would appear to have rediscovered one of the most reliable ways of getting a new idea into the kernel: post some code then wait for Ingo Molnar to rework the whole thing in a two-day hacking binge. So, while Con has recently updated the SD scheduler patch, his work now looks like it might be upstaged by Ingo's new completely fair scheduler (CFS), at version 2 as of this writing.

There are a number of interesting aspects to CFS. To begin with, it does away with the arrays of run queues altogether. Instead, the CFS works with a single red-black tree to track all processes which are in a runnable state. The process which pops up at the leftmost node of the tree is the one which is most entitled to run at any given time. So the key to understanding this scheduler is to get a sense for how it calculates the key value used to insert a process into the tree.

That calculation is reasonably simple. When a task goes into the run queue, the current time is noted. As the process waits for the CPU, the scheduler tracks the amount of processor time it would have been entitled to; this entitlement is simply the wait time divided by the number of running processes (with a correction for different priority values). For all practical purposes, the key is the amount of CPU time due to the process, with higher-priority processes getting a bit of a boost. The short-term priority of a process will thus vary depending on whether it is getting its fair share of the processor or not.

It is only a slight oversimplification to say that the above discussion covers the entirety of the CFS scheduler. There is no tracking of sleep time, no attempt to identify interactive processes, etc. In a sense, the CFS scheduler even does away with the concept of time slices; it's all a matter of whether a given process is getting the share of the CPU it is entitled to given the number of processes which are trying to run. The CFS scheduler offers a single tunable: a "granularity" value which describes how quickly the scheduler will switch processes in order to maintain fairness. A low granularity gives more frequent switching; this setting translates to lower latency for interactive responses but can lower throughput slightly. Server systems may run better with a higher granularity value.

Ingo claims that the CFS scheduler provides solid, fair interactive response in almost all situations. There's a whole set of nasty programs in circulation which can be used to destroy interactivity under the current scheduler; none of them, says Ingo, will impact interactivity under CFS.

The CFS posting came with another feature which surprised almost everybody who has been watching this area of kernel development: a modular scheduler framework. Ingo describes it as "an extensible hierarchy of scheduler modules," but, if so, it's a hierarchy with no branches. It's a simple linked list of modules in priority order; the first scheduler module which can come up with a runnable task gets to decide who goes next. Currently two modules are provided: the CFS scheduler described above and a simplified version of the real-time scheduler. The real-time scheduler appears first in the list, so any real-time tasks will run ahead of normal processes.

There is a relatively small set of methods implemented by each scheduler module, starting with the queueing functions:

```
void (*enqueue_task) (struct rq *rq, struct task_struct *p);
void (*dequeue_task) (struct rq *rq, struct task_struct *p);
void (*requeue_task) (struct rq *rq, struct task_struct *p);
```

When a task enters the runnable state, the core scheduler will hand it to the appropriate scheduler module with `enqueue_task()`; a task which is no longer runnable is taken out with `dequeue_task()`. The `requeue_task()` function puts the process behind all others at the same priority; it is used to implement `sched_yield()`.

A few functions exist for helping the scheduler track processes:

```
void (*task_new) (struct rq *rq, struct task_struct *p);
void (*task_init) (struct rq *rq, struct task_struct *p);
void (*task_tick) (struct rq *rq, struct task_struct *p);
```

The core scheduler will call `task_new()` when processes are created. `task_init()` initializes any needed priority calculations and such; it can be called when a process is reniced, for example. The `task_tick()` function is called from the timer tick to update accounting and possibly switch to a different process.

The core scheduler can ask a scheduler module whether the currently executing process should be preempted now:

```
void (*check_preempt_curr) (struct rq *rq, struct task_struct *p);
```

In the CFS scheduler, this check tests the given process's priority against that of the currently running process, followed by the fairness test. When the fairness test is done, the scheduling granularity is taken into account, possibly allowing a process to run a little longer than strict fairness would allow.

When it's time for the core scheduler to choose a process to run, it will use these methods:

```
struct task_struct * (*pick_next_task) (struct rq *rq);
void (*put_prev_task) (struct rq *rq, struct task_struct *p);
```

The call to `pick_next_task()` asks a scheduler module to decide which process (among those in the class managed by that module) should be running currently. When a task is switched out of the CPU, the module will be informed with a call to `put_prev_task()`.

Finally, there's a pair of methods intended to help with load balancing across CPUs:

```
struct task_struct * (*load_balance_start) (struct rq *rq);
struct task_struct * (*load_balance_next) (struct rq *rq);
```

These functions implement a simple iterator which the scheduler can used to work through all processes currently managed by the scheduling module.

One assumes that this framework could be used to implement different scheduling regimes in the future. It might need some filling out; there is, for example, no way to prioritize scheduling modules (or choose the default module) other than changing the source. Beyond that, if anybody ever wants to implement modules which schedule tasks at the same general priority level, the strict priority ordering of the current framework will have to change - and that could be an interesting task. But it's a start.

The reason that this development is so surprising is that nobody had really been talking about modular schedulers. And the reason for that silence is that pluggable scheduling frameworks had been soundly rejected in the past - by Ingo Molnar, among others:

> So i consider scheduler plugins as the STREAMS equivalent of scheduling and i am not very positive about it. Just like STREAMS, i consider 'scheduler plugins' as the easy but deceptive and wrong way out of current problems, which will create much worse problems than the ones it tries to solve.

So the obvious question was: what has changed? Ingo has posted an explanation which goes on at some length. In essence, the previous pluggable scheduler patches were focused on replacing the entire scheduler rather than smaller pieces of it; they did not help to make the scheduler simpler.

So now there are three scheduler replacement proposals on the table: SD by Con Kolivas, CFS by Ingo Molnar, and "nicksched" by Nick Piggin (a longstanding project which clearly deserves treatment on this page as well). For the moment, Con appears to have decided to take his marbles and go home, removing SD from consideration. Still, there are a few options out there, and one big chance (for now) to replace the core CPU scheduler. While Ingo's work has been generally well received, not even Ingo is likely to get a free pass on a decision like this; expect there to be some serious discussion before an actual replacement of the scheduler is made. Among other things, that suggests that a new scheduler for 2.6.22 is probably not in the cards.

Index entries for this article

Kernel	Interactivity
Kernel	Scheduler/Completely fair scheduler

Schedulers: the plot thickens

Posted Apr 19, 2007 2:33 UTC (Thu) by **dlang** (guest, #313) [Link]

if Ingo or Linus get fired up on a subject the process for testers is

1. find the latest version of the patch
2. download it
3. if on a slow link, go back to #1
4. compile it
5. if on a slow cpu, go back to #1
6. start testing
7. find bug
8. go back to #1
9. report bug.

with normal developers you can count on the code being stable for a day or two after release, you don't have to keep checking for new releases

yes, this is slightly exaggerating things, but not my much (sometimes it seems like the time it takes for you to read the e-mail announceing a release is enough time for an update)

Ouch.

Posted Apr 19, 2007 6:09 UTC (Thu) by **smurf** (subscriber, #17840) [Link]

I can certainly understand Con.

But, on the other hand, runqueue handling is an issue that, so far, every scheduler has shown problem with, one time or another. The radical idea of doing away with them altogether certainly deserves a closer look.

I'm interested how the single-RB-tree idea will handle on a machine with a lot of CPUs. Off to check gmane.lists.linux.kernel ...

Reply to this comment

rbtree is per-CPU

Posted Apr 19, 2007 6:18 UTC (Thu) by **axboe** (subscriber, #904) [Link]

Hi,

The rbtree task timeline is per CPU, it's not a global entity. The latter would naturally never fly.

Reply to this comment

rbtree is per-CPU

Posted Apr 19, 2007 6:28 UTC (Thu) by **kwchen** (guest, #13445) [Link]

Has anyone experimented with one scheduling entity per numa-node, or one per physical CPU package etc, instead of current one per CPU?

Reply to this comment

rbtree is per-CPU

Posted Apr 19, 2007 6:47 UTC (Thu) by **smurf** (subscriber, #17840) [Link]

Gah. Forgive my sloppy use of language. By "single RB tree" I meant "a single tree to replace the former run queues structure", not "a single tree on the whole system". Process migration between CPUs is, after all, not going to go away.

On another front, despite Con being rather miffed by Ingo's original patch, the subsequent dialog between them is a model of mutual respect that lots of people can learn from. Myself included.

Reply to this comment

Ouch.

Posted Apr 26, 2007 21:58 UTC (Thu) by **slamb** (guest, #1070) [Link]

> But, on the other hand, runqueue handling is an issue that, so far, every scheduler has shown problem with, one time or another. The radical idea of doing away with them altogether certainly deserves a closer look.

I'm surprised by the combination of no one doing this before and someone doing it now. Before reading the recent article about the scheduler, I'd only seen priority queues implemented as an array of plain queues in cases where there were only a handful of priorities. When there's one per nice level (140?) or many more (priority is a function of nice level and timeslice left), it seems like trees or heaps would be an obvious choice. Having a sorted structure seems much simpler than doing these minor and major rotations to the array, with this secondary "expired" array.

So given that they originally did this a different way, the logical question is why. Was it so the time complexity could be O(p) [*] rather than O(log n)? Well, now Ingo's apparently thinking that's not important. How many processes was the O(1) scheduler designed to support? How long does Ingo's scheduler take to run in that situation?

If O(log n) does turn out to be a problem, I wonder if a mostly-sorted soft heap would be better at amortized O(1). Faster as a whole, and "corrupting" a fixed percentage of priorities might not be a bad way to avoid total starvation of low-priority processes, but amortized time might mean it'd be too jerky to be used.

[*] - p = number of priorities...from skimming the description, it doesn't look like it was ever O(1) like the cool kids said. They just considered p to be a constant 140.

Reply to this comment

Schedulers: the plot thickens

Posted Apr 19, 2007 12:34 UTC (Thu) by **i3839** (guest, #31386) [Link]

Another important property of Ingo's scheduler is that the time is measured in nanoseconds. According to a later email, Ingo said that the earlier version without the high precision and using queues didn't work well at all. This are the two things that seem to have been limiting RSDL and other schedulers, as strange artefacts cropped up because of the queue and low granularity design.

Peter William (of plugsched) also wrote a scheduler, and experience with trying out different things.

William Lee Irwin III (now I understand why people call him wli ;-) is hammering on the importance of a standard test suite for schedulers, so if there are people with free time who want to help him with setting one up...

Reply to this comment

Schedulers: the plot thickens

Posted Apr 24, 2007 12:25 UTC (Tue) by **jospoortvliet** (guest, #33164) [Link]

there isn't really a standard testsuite, but many of the problem cases (and the little code snippets to show 'em) are sticking around, and are used for testing new scheduler improvements/changes. Quite a few float around on the LKML and also on Dr Con's mailinglist.

Reply to this comment

Scheduler architecture and modularity

Posted Apr 19, 2007 13:12 UTC (Thu) by **brugolsky** (subscriber, #28) [Link]

The "modularity" in Ingo's queue interface would seem to lend itself toward implementing something similar to the traffic control packet scheduler framework. IIRC, OpenVZ uses a TBF-based hierarchical fair scheduler; it would be interesting to see it ported to CFS.

Reply to this comment