Scheduling domains

[Posted April 19, 2004 by corbet]

Back in the 2.6.0-test days, there was a lot of concern that the 2.6 CPU scheduler wasn't up to the task. In particular, performance on higher-end systems - those with hyperthreaded processors, NUMA architectures, etc. - wasn't as good as the developers would have liked. The scheduler front has been quiet for some time, but it has not been forgotten; a set of hackers (including Nick Piggin, Ingo Molnar, Con Kolivas, and Rusty Russell) has been steadily working behind the scenes to improve scheduling in 2.6. The result, broadly known as "scheduling domains," has been evolving in the -mm tree for some time, but this work looks like it is getting close to ready to break into the mainline. So, it would seem that a look at scheduling domains is in order.

The new scheduler work is a response to the needs of modern hardware and, in particular, the fact that the processors in multi-CPU systems have unequal relationships with each other. Virtual CPUs in a hyperthreaded set share equal access to memory, cache, and even the processor itself. Processors on a symmetric multiprocessing system have equal access to memory, but they maintain their own caches. NUMA architectures create situations where different nodes have different access speeds to different areas of main memory. A modern large system can feature all of these situations: each NUMA node looks like an SMP system which may be made up of multiple hyperthreaded processors.

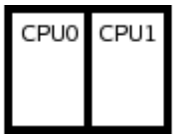
One of the key problems a scheduler must solve on a multi-processor system is balancing the load across the CPUs. It doesn't do to have some processors being heavily loaded while others sit idle. But moving processes between processors is not free, and some sorts of moves (across NUMA nodes, for example, where a process could be separated from its fast, local memory) are more expensive than others. Teaching the scheduler to migrate tasks intelligently under many different types of loads has been one of the big challenges of the 2.5 development cycle.

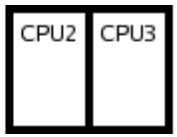
The domain-based scheduler aims to solve this problem by way of a new data structure which describes the system's structure and scheduling policy in sufficient detail that good decisions can be made. To that end, it adds a couple of new structures:

- A **scheduling domain** (`struct sched_domain`) is a set of CPUs which share properties and scheduling policies, and which can be balanced against each other. Scheduling domains are hierarchical; a multi-level system will have multiple levels of domains.

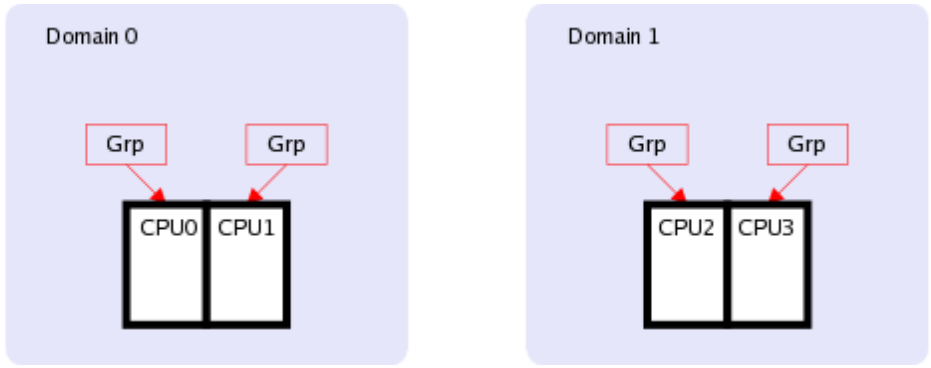
- Each domain contains one or more **CPU groups** (`struct sched_group`) which are treated as a single unit by the domain. When the scheduler tries to balance the load within a domain, it tries to even out the load carried by each CPU group without worrying directly about what is happening within the group.

It's time for your editor to try to explain this structure via a series of cheesy diagrams. Imagine a system with two physical processors, each of which provides two hyperthreaded CPUs. We'll diagram the processors in this way:



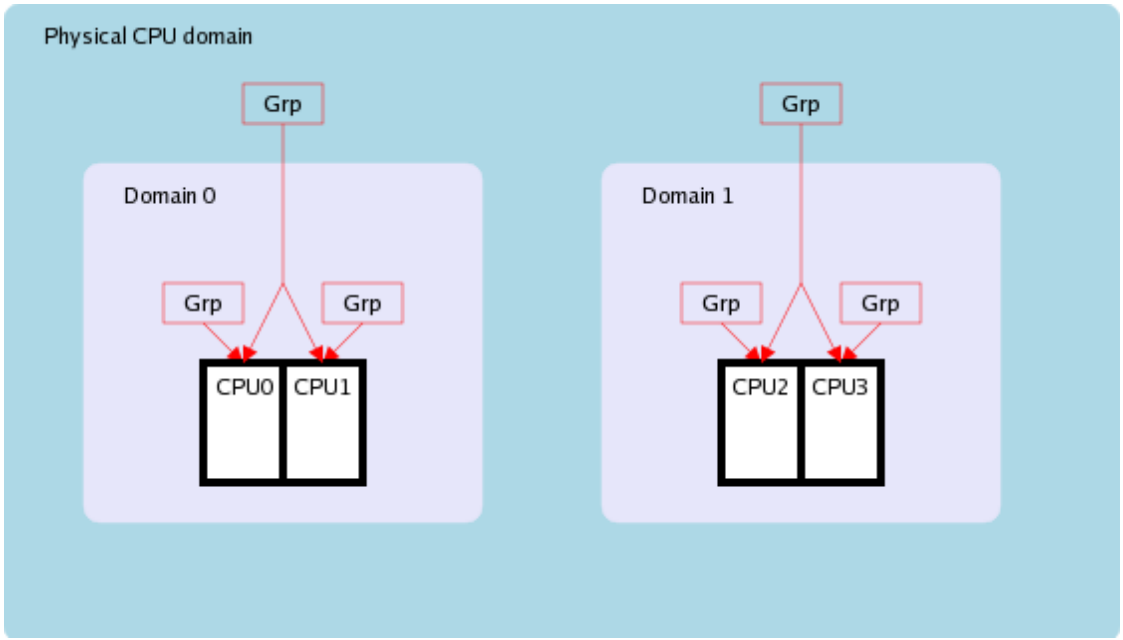


Here, the four hyperthreaded processors are shown bonded together into two physical packages. When this system boots, it will put each pair of processors into a scheduling domain, with a result that might look something like this:



In this setup, our four processors are gathered into two scheduling domains. Each domain contains two CPU groups, and each group contains exactly one CPU. These domains reflect the fact that, while each CPU appears to be a distinct processor, a pair of hyperthreaded processors has a different relationship internally than with the other processors.

This system will have a two-level hierarchy of scheduling domains; when we add the top level the picture becomes:



This top-level domain is the parent of the processor-level domains. It contains two CPU groups, each of which contains the CPUs contained within one hyperthreaded processor package.

If this were a NUMA system, it would have multiple domains which look like the above diagram; each of those domains would represent one NUMA node. The hierarchy would have a third, system-level domain which contains all of the NUMA nodes.

Note that, in the actual code, the hierarchy is represented a little differently than has been portrayed above; each CPU has its own copy of every domain it belongs to. So our little system would actually contain eight `sched_domain` structures: one copy of the CPU-level domain and one copy of the top-level domain for every processor. Things are implemented this way for performance reasons: the scheduler must be very fast, which contraindicates sharing this fundamental data structure between processors. The structure is, in any case, almost entirely read-only after it has been set up, so it can be replicated without trouble.

Each scheduling domain contains policy information which controls how decisions are made at that level of the hierarchy. The policy parameters include how often attempts should be made to balance loads across the domain, how far the loads on the component processors are allowed to get out of sync before a balancing attempt is made, how long a process can sit idle before it is considered to no longer have any significant cache affinity, and various policy flags. These policies tend to be set as follows:

- **At the hyperthreaded processor level:** balancing attempts can happen often (every 1-2ms), even when the imbalance between processors is small. There is no cache affinity at all: since hyperthreaded processors share cache, there is no cost to moving a process from one to another. Domains at this level are also marked as sharing CPU power; we'll see how that information is used shortly.

- **At the physical processor level:** balancing attempts do not have to happen quite so often, and they are curtailed fairly sharply if the system as a whole is busy. Processor loads must be somewhat farther out of balance before processes will be moved within the domain. Processes lose their cache affinity after a few milliseconds.

- **At the NUMA node level**: balancing attempts are made relatively rarely, and cache affinity lasts longer. The cost of moving a process between NUMA nodes is relatively high, and the policy reflects that.

The scheduler uses this structure in a number of ways. For example, when a sleeping process is about to be awakened, the normal behavior would be to keep it on the same processor it was using before, on the theory that there might still be some useful cache information there. If that processor's scheduling domain has the `SD_WAKE_IDLE` flag set, however, the scheduler will look for an idle processor within the domain and move the process immediately if one is found. This flag is used at the hyperthreading level; since the cost of moving processes is insignificant, there is no point in leaving a processor idle when a process wants to run.

When a process calls `exec()` to run a new program, its current cache affinity is lost. At that point, it may make sense to move it elsewhere. So the scheduler works its way up the domain hierarchy looking for the highest domain which has the `SD_BALANCE_EXEC` flag set. The process will then be shifted over to the CPU within that domain with the lowest load. Similar decisions are made when a process forks.

If a processor becomes idle, and its domain has the `SD_BALANCE_NEWIDLE` flag set, the scheduler will go looking for processes to move over from a busy processor within the domain. A NUMA system might set this flag within NUMA nodes, but not at the top level.

The new scheduler does an interesting thing with "shared CPU" (hyperthreaded) processors. If one processor in a shared pair is running a high-priority process, and a low-priority process is trying to run on the other processor, the scheduler will actually idle the second processor for a while. In this way, the high-priority process is given better access to the shared package.

The last component of the domain scheduler is the active balancing code, which moves processes within domains when things get too far out of balance. Every scheduling domain has an interval which describes how often balancing efforts should be made; if the system tends to stay in balance, that interval will be allowed to grow. The scheduler "rebalance tick" function runs out of the clock interrupt handler; it works its way up the domain hierarchy and checks each one to see if the time has come to balance things out. If so, it looks at the load within each CPU group in the domain; if the loads differ by too much, the scheduler will try to move processes from the busiest group in the domain to the most idle group. In doing so, it will take into account factors like the cache affinity time for the domain.

Active balancing is especially necessary when CPU-hungry processes are competing for access to a hyperthreaded processor. The scheduler will not normally move running processes, so a process which just cranks away and never sleeps can be hard to dislodge. The balancing code, by way of the migration threads, can push the CPU hog out of the processor for long enough to allow it to be moved and spread the load more widely.

When the system is trying to balance loads across processors, it also looks at a parameter kept within the `sched_group` structure: the total "CPU power" of the group. Hyperthreaded processors look like independent CPUs, but the total computation power of a pair of hyperthreaded processors is far less than that of two separate packages. Two separate processors would have a "CPU power" of two, while a hyperthreaded pair would have something closer to 1.1. When the scheduler considers moving a process to balance out the load, it looks at the total amount of CPU power currently being exercised. By maximizing that number, it will tend to spread processes across physical processors and increase system throughput.

The new scheduling code has been under development for some time, and it has seen a great deal of tweaking. The domain mechanism has done a lot to make it possible to make good scheduling decisions, but much of detail work was still required. It would appear that that work is now reaching a point where the domain mechanism may soon be merged into the mainline. At that point, with luck, people will be able to stop complaining about the 2.6 scheduler.

(Thanks to Nick Piggin for his comments on an early version of this article).

Index entries for this article

Kernel	NUMA
Kernel	Scheduler

Wait just a minute!

Posted Apr 22, 2004 15:51 UTC (Thu) by **heinlein** (guest, #1029) [Link]

Jon,

I've got to complain. I'm not much of a hardware guy, nor am I anything close to a kernel hacker. I can blissfully allow my eyes to glaze over whenever I encounter articles that purport to explain the interactions between the kernel and the hardware.

You, however, write much too clearly. I'm pretty sure I actually understood your explanation of scheduling domains. That means that I had to pay attention -- no daydreaming while nominally doing my "professional development" reading. Shame on you.

Wait just a minute!

Posted Apr 23, 2004 12:11 UTC (Fri) by **pimlott** (guest, #1535) [Link]

You think you're suffering? Think of the poor kernel hackers who are losing their mystique!

Wait just a minute!

Posted Apr 23, 2004 12:46 UTC (Fri) by **alspnost** (guest, #2763) [Link]

Right on - I have learned more about this sort of stuff from LWN than from any other source. I find kernel internals rather fascinating, but trying to learn by reading the raw LKML discussions is completely beyond me, and I have never got very far that way; Jon's kernel articles are superb, and I know of no other place where I can get this type of analysis. It's not watered down or simplified, yet it's expressed in a way that people like us actually have a chance of understanding. A rare talent indeed!

Now, where can I get a nice 8-way NUMA Opteron laptop so that I can play with all this scheduling stuff for real? ;-)

Reply to this comment

What about threads?

Posted Apr 22, 2004 20:36 UTC (Thu) by **stuart2048** (guest, #6241) [Link]

This balancing technique, as it is described here, seems to talk about scheduling processes. But in my notion of the OS world, processes don't actually run -- threads do. So isn't this more about thread scheduling than process scheduling? Granted, I have never poked around any of the scheduler or process management code in Linux, so I could be way off...

I would be interested to learn how this scheduler deals with processes with multiple busy threads on, say, NUMA or SMP hardware.

Thanks for the great article, and keep those cheesy diagrams coming!

--Stuart

Reply to this comment

What about threads?

Posted Apr 23, 2004 0:32 UTC (Fri) by **hmh** (subscriber, #3838) [Link]

A thread and a process are usually about the same thing in Linux.

Reply to this comment

What about threads?

Posted Apr 23, 2004 21:54 UTC (Fri) by **giraffedata** (guest, #1954) [Link]

We're actually in a thread/process terminology crisis in Linux. Various people have various ideas about what we should mean by "thread," "process," "task," and "thread group."

It's bad right now because the thread/process model in Linux only recently changed, making what was once a pretty well agreed upon terminology less useful.

So the article is about Linux processes in the old terminology, the terminology you will still see in most of the comments in the Linux code. In that terminology, a "process" is in fact the most basic unit of scheduling known to the Linux kernel, and is what implements a thread in the POSIX thread model. It is alternatively called a "thread" and a "task."

Reply to this comment

So is this stable?

Posted Apr 29, 2004 10:24 UTC (Thu) by **dash2** (guest, #11869) [Link]

As a non-expert, it sounds to me like this is quite a radical new feature. I thought the idea with 2.6 was to avoid the problems of 2.4 where big changes were made throughout the stable series. But we are at 2.6.6 now, and these kind of changes are still being accepted. So what exactly does the designation "stable" for even-numbered kernels mean?

Reply to this comment

So is this stable?

Posted Apr 29, 2004 11:03 UTC (Thu) by **russell** (guest, #10458) [Link]

Don't look at the numbers. Try it out, if it crashes, it's not stable, if it doesn't, then it's stable for you.

Reply to this comment

Scheduling domains

Posted Apr 29, 2004 19:08 UTC (Thu) by **james_northrup** (guest, #7684) [Link]

wow very very nice

This effort needs to dovetail with openmosix to seriously improve both fronts of specialized understanding.

Openmosix has a very capable process virtualization model which is elegant and effective, and coincides perfectly with (and benefits greatly from..) domain managed concepts of specific relative wieghted performance windows.

as openmosix ties distributed code execution to the core cpu scheduler, this 3-tier domain hierarchy becomes something truly inspiring with a 4th virtual execution backdrop.

Reply to this comment

Different "length" to memory on the same doamin?

Posted Apr 29, 2004 20:44 UTC (Thu) by **perlid** (guest, #6533) [Link]

Just a question:
If you have for example an 8-way Opteron system, then each processor has it's own memory. If a processor wants some data which is in an other processors memory, it has to go through the other processors. But the Opterons are connected in some sort of a ring, so sometimes the data may be just one neighbour away, and sometimes it must go through 2 (or maybe even more?) processors to get it's data.
For me it looks like all these eight processors are on the same scheduling domain, even though some are "more close" to each other than others?
How does the scheduling domains system handel this?

Reply to this comment