Sayed Ismail Safwat (saysa289)

The Linux Scheduler

The scheduler is supposed to keep a simple and seamless form, but it appears to be not an easy task. The standard methods and tools not being capable of producing the desired effect is a reason for creating new tools which are going to be discussed later.

With multicore entering in the OS world, the results showed that the scheduler were not able to perform the its tasks as before. Four performance bugs were identified and studied in the scheduler in this paper. These bugs affect extremely the kernel performance resulting energy wastes which will be explained thoroughly.

The study shows that the bugs results in threads leaving the cores idle, unlike what it is supposed to do which is not expected. Although these bugs do not cause crash, but affects mostly performance of the system.

Two new tools are introduced for identifying and fixing the bugs, *sanity checker* which constantly checks for the bug through a live system and save the required information for later studies. the latter is a visualization tool that as a result, makes the debugging quicker.

CFS (Completely Fair Scheduling algorithm) is implemented based on WFQ (weight fair queueing) algorithm which typically determines and allocates a processing time (called as time-slice too) to the system where it must run.
Basically, the main point of the scheduling algorithms is to maintain the cores as busy as possible. Back in 2001, where the CPUs had to deal only with single-cores, the algorithms such as load-balancing were functioning as normal but in time it became more complex with multicore systems.

In order to understand load balancing algorithm we need to go through load tracking metric, a metric which CFS uses. In a typical load-balancing algorithm threads are divided equally to each run-queue which appears to be unfair. This can be proved in different scenarios e.g. in a case where we have the same number of threads in both run-queues but with different weights (referred to priority) or in a case where we have

different number of threads with different weights. In each case scenario we have to deal with a sort of trade-off, therefore we rely on a metric called *load* which is a mix of thread's priority value and average CPU usage. For example, in case the CPU is not used by thread, the load will reduce according to the CPU usage. The issue with load-balancing algorithm is that it moves the threads in the system without verifying the NUMA. As the cores are arranged hierarchically depending on the system's size, the load-balancing algorithms are executed based on hierarchies where each hierarchy is called a scheduling domain.
In order to optimize the system, load-balancing algorithm will make sure to check if it the given core has the lowest number of threads or it has the lowest weight.
As we went through the different scenarios for how long and in what times we know that it's time to keep the core idle makes it more complicated to determine. As Linus himself declares "Nobody actually creates perfect code the first time around, except me. But there's only one of me." (Torvalds, 2007) it is quite reasonable to have bugs in the system.

The bugs

Group Imbalance bug: In an attempt on a 64-core system, it appeared that not all available cores were used, on contrary they were divided only to a few cores. Later in the tests, it showed that there were two reasons behind the bug, the complexity of the load tracking metric and the hierarchical design.

The fix: As identified in the bug, when a node tries to steal work from another scheduling group, it checks the group's average load. In case the average load of that scheduling group was greater than itself, it will steal from that group, otherwise not. Now the fix will be checking the group's minimum load instead of average load. As a result, it indicates that if minimum load of a scheduling group is less than the other group, then the first group has a node that has the minimum amount of load among all nodes in the other group so it permits the first group to steal from the second group. This fix has also shown that it has produced a high performance compared to the before.

The scheduling group construction bug: In an attempt it shows that the structure of scheduling group are not compatible with the NUMA

systems we use today. The bug lies in the load balancing algorithm where it hinders the threads to move between two specific nodes.
The fix: The reason behind the bug was that the cores were not able to steal work from each other because they were not able to check the difference. Thus, modifications were needed in the structure of scheduling groups so each node is arranged not from the perspective of any given node but from the perspective of itself.

The Overload-on-Wakeup bug: The bug was identified on an attempt to optimize the wakeup code (select_task_rq_fair) function. In short this bug occurs when the same core is reused again while other cores are idle.
The fix will be applying the opposite of what bug causes, it means changing the code that is executed when a thread wakes up. In other words, the thread that has been idle for the longest amount of time will be used.

The missing scheduling domains bug: An internal malfunction update that represents the wrong number of scheduling domains in the system is the reason behind this bug. It appears that when a node is disabled, and again re-enabled, load balancing cannot be executed between any NUMA nodes.
The fix: It shows that Linux regenerates scheduling domains every time a core is disabled. This regeneration is performed in two steps, one is that the kernel regenerates domains inside NUMA nodes, and the latter is across NUMA nodes. This second step was forgotten by Linux developers during code refactoring, so by adding back the the second step, the fix was complete.

Over time, different designs have been proved/changed and it shows that these new techniques are not an exception meaning that they are not long-term solutions, as it is required to build them on scratch.

Tools

Sanity checker is the term pertaining to the structure for checking the invariants to be checked in fixed intervals. This tool checks that no core is idle while other cores has waiting threads.

Scheduler visualization tool was a helpful tool to outline the size of run queues, the total load of run-queues and the cores that were used in load-balancing and thread wake-ups.

Conclusion

The rapid evolution of hardware and software is something we cannot neglect and therefore it requires us to be regularly updated. This is highly relevant to this article as the authors of article stated e.g. when it comes to scheduler optimizations. Another important thing to discuss is the integration of the scheduler and the methods/techniques to combine them.
I believe that the visualization tool that the authors of the article created cleared the path for them to understand the problem better and quicker. In addition it made them profile and finally fix the given bugs.
As stated before, with the speed the evolution of hardware is keeping and the new algorithms introduced continuously, what if another modern system like multi-core system is introduced? How are they going to be integrated then? We know that visualization is not a new method, but how to know what to visualize?

References

Lozi, J.P. B. Lepers, J. Funston, F. Gaud, V. Quéma, A. Fedorova. (2016) ‘The Linux scheduler: a decade of wasted cores’, *EuroSys ‘16,* 1-16.