Let's Build Provable Multicore Schedulers!

Redha Gouicem
Sorbonne Universités, Inria, LIP6
redha.gouicem@lip6.fr

Keywords Operating Systems, Multicore, Domain-Specific Language, Scheduling

1. Motivation

The default Linux scheduler, the Completely Fair Scheduler (CFS), tries to be as generic as possible and handle all kinds of workloads. However, handling every workload's specificities is complex. Over time, it has grown at quite a fast pace. For example, as shown in figure 1, one of the main file of the scheduler (`fair.c`) has increased from roughly 600 to 5000 lines of code since CFS introduction in 2007 (nearly $\times 10$ in 10 years). This shows us that it is not likely that one will ever be able to design a perfect scheduling policy. Each workload has different characteristics and resource needs that might be conflicting with another workload's. Yang et al. [13] propose another generic scheduler that differentiates interactive and best-effort applications and handles them differently. On the contrary, one can build the scheduler specifically for an application or a class of applications. Zhuravlev et al. [14] evaluate multiple classification schemes ([3], [7], [12]) and their implementation as a criterion for a user space scheduler. Those policies aim at lowering cache contention. Antonopoulos et al. [2] propose a user space scheduling policy minimizing contention on the memory controller. Teabe et al. [11] propose to change an application's quantum depending on its I/O activity. Usually, those policies are implemented in user space using techniques like thread pinning, since implementing a scheduler in an operating system's kernel requires a high level of expertise due to the complexity of kernel code. Unfortunately, this adds another layer over the kernel space scheduler, which can interfere with the user space policy.

Another problem met when writing a scheduling policy is the ability to have confidence in the policy we are writing. One might want to ensure multiple properties like liveness (no starvation), fairness between processes or work-conservation between cores. Recently, multiple works were conducted with the purpose of proving that parts of an operating system were correct according to their specification. Amani et al. [1] and Chen et al. [4] prove file systems behavior, Gu et al. [5] propose a certified kernel, and Klein et al. [6] propose a formally verified kernel. Yet, proving high level properties (in our case, liveness, fairness, work-conservation) is a highly difficult task that cannot be performed by any software developer willing to implement a scheduler.

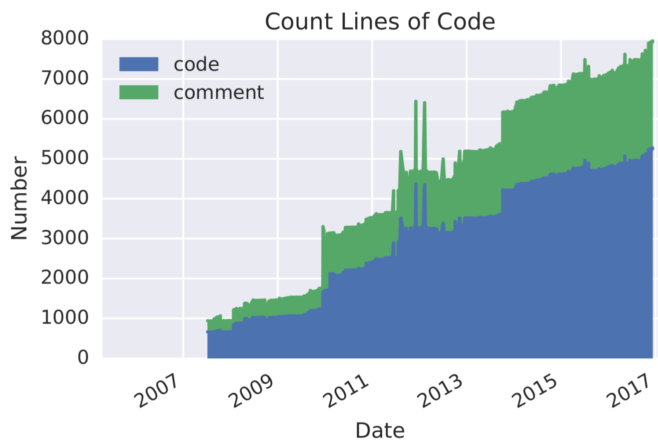

Figure 1. `fair.c` file evolution

2. PhD thesis approach

During this PhD thesis, we will investigate scheduler development and property proving, and try to propose a solution enabling software developers to write their own specific kernel space scheduler with minimal kernel expertise, as well as a way to prove if their scheduler guarantees several properties related to scheduling.

In order to write a kernel space scheduling policy, one must be sure that one's implementation is safe, meaning that the kernel must neither hang nor crash because of the scheduler. Muller et al. [9] propose Bossa, a domain-specific language tailored to write schedulers for single core systems. This allows software developers to implement kernel schedulers without the kernel development expertise needed to

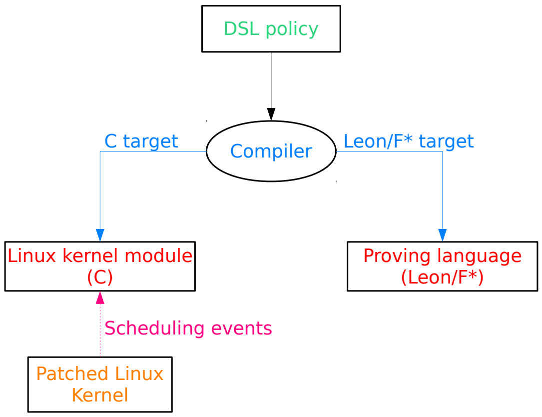

Figure 2. DSL system architecture

write safe code. For this PhD thesis, we propose to follow up this approach and extend it to multicore systems. The first challenge is to define exactly what is related to the scheduling policy and what is not. We break down the scheduler into small parts or events. This way, it is easier for developers to write their scheduling policy, and for us to insert this code into the kernel. This will also allow us to verify and prove small parts of the policy, instead of trying to prove it as a whole.

We propose to use the same design as Bossa, with new abstractions for multicore systems. This means that we have to understand and abstract the load balancing phase of scheduling. As presented in Figure 2, scheduling policies are written in our DSL, then compiled to C-code ready to be compiled into a kernel module. This module will be inserted in a generic kernel patched to support event-based scheduling. The challenging part of this process is that it must be safe: the compiler must not generate code that will hang or crash the kernel.

Our subsequent goal is to provide another target to our compiler in order to generate proofs regarding various scheduling properties (liveness, fairness, work-conservation). This target should be a language that provides automatic proving tools, such as Scala (using Leon [8]) or F* [10]. One major challenge here is to be able to automatically generate proofs while preserving the expressivity of the DSL.

References

[1] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, et al. Cogent: Verifying high-assurance file system implementations. In *ACM SIGPLAN Notices*, volume 51, pages 175–188. ACM, 2016.

[2] Christos D Antonopoulos, Dimitrios S Nikolopoulos, and Theodore S Papatheodorou. Scheduling algorithms with bus bandwidth considerations for smps. In *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, pages 547–554. IEEE, 2003.

[3] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting inter-thread cache contention on a chip multiprocessor architecture. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 340–351. IEEE, 2005.

[4] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 18–37. ACM, 2015.

[5] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Newman Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. Certikos: an extensible architecture for building certified concurrent os kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, 2016.

[6] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

[7] Rob Knauerhase, Paul Brett, Barbara Hohlt, Tong Li, and Scott Hahn. Using os observations to improve performance in multicore systems. *IEEE micro*, 28(3), 2008.

[8] EPFL LARA team. Leon, https://leon.epfl.ch/.

[9] Gilles Muller, Julia L Lawall, and Hervé Duchesne. A framework for simplifying the development of kernel schedulers: Design and performance evaluation. In *High-Assurance Systems Engineering, 2005. HASE 2005. Ninth IEEE International Symposium on*, pages 56–65. IEEE, 2005.

[10] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, et al. Dependent types and multi-monadic effects in f. In *ACM SIGPLAN Notices*, volume 51, pages 256–270. ACM, 2016.

[11] Boris Teabe, Alain Tchana, and Daniel Hagimont. Application-specific quantum for multi-core platform scheduler. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 3. ACM, 2016.

[12] Yuejian Xie and Gabriel Loh. Dynamic classification of program memory behaviors in cmps. In *the 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects*. Citeseer, 2008.

[13] Ting Yang, Tongping Liu, Emery D Berger, Scott F Kaplan, and J Eliot B Moss. Redline: First class support for interactivity in commodity operating systems. In *OSDI*, volume 8, pages 73–86, 2008.

[14] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ACM Sigplan Notices*, volume 45, pages 129–142. ACM, 2010.