Multi-Threaded Computer Systems

Christopher Rodriguez

Colorado State University Global

22SC-CSC300-1: Operating Systems and Architecture

Dr. Jonathan Vanover

2022-05-29

Multi-Threaded Computer Systems

On current computing systems, it is common to have multiple processes executing at the same time—that is, truly concurrent execution, not simply switching very quickly. This is the result of two separate ideas maturing and coming to the forefront of programs today: *Multithreaded Computing*, where lightweight processes share the machines resources (Serrano et al., 2004, p. 203), and *Multicore Architectures*, where CPUs can have more than one "core processing unit", and therefore do two things at once. This allows for something seemingly magical to become possible: *Parallel Programming*.

Parallel Programming

"Parallel Programming" might be best defined in a way similar to an algorithm: Take a complex bit of work, divide it up into small, independent pieces, and have each one execute by itself to accomplish the original task (Erwig, 2017).

The thing that sticks out about *parallel programming* when opposed to *multiprogramming*—where multiple programs are loaded into memory at the same time—is the concept of *concurrency*: Take the resources of a system and divide them up across as many different subprocess as is possible for a task, *at the same time*. (As an aside: Take a moment and appreciate that this was once the exclusive domain of supercomputers and clusters, and is now in the phone that is in Your pocket.) Then it's a matter of letting these concurrent processes run to completion, and using a scheduler sure they don't bump into each other too much while sharing data, CPU time, and memory (Sattar et al., 2010, p. 24).

When it comes to sharing memory between concurrent processes, Serrano et al. (2004) have asserted that the two most common implementations are *Event-Driven Programming*—that is, an event loop (where a single loop handles "events" like key presses and mouse clicks, and delegates such events to other functions to be handled as they happen)—and *Multithreaded Architectures*—Lightweight processes that share memory and execute according to one of two scheduling

strategies (p. 203). This paper deals with multithreaded architectures.

Threads

Within "Multithreaded Architectures", there primarily exist two classes of scheduling strategy: *competitive* (or, as used in this essay, *preemptive*), and *cooperative*. The main difference between these strategies is how use of the processor is yielded back to the scheduler: In *preemptive* multithreading, any thread can be suspended or resumed at any moment by the scheduler, regardless of the state of the work is was meant to get done (Silberschatz et al., 2018, p. 257) In *cooperative* multithreading, this is not possible: The scheduler instead waits for the thread itself to yield control of the processor before it can be allocated elsewhere (Serrano et al., 2004, pp. 203–204).

It is not without base that *preemptive* threads are more complex than their *cooperative* counterparts: Older OSes, such as Windows 3.1 and MacOS before OS/X, that once used cooperative multithreading have since made a switch towards preemptive multithreading in their more recent versions (Serrano et al., 2004, p. 204). And Sun et al. (2020) made it a point for their educational OS project "Quatros" to begin with a number of cooperative threads, and later transition to preemptive threads with some synchronization primitives (p. 51). However, there is a huge blocker to using cooperative threads on current computing machines (at least in performance-critical applications):

Cooperative Threads cannot (easily) benefit from multiprocessor—or, in today's world, multicore—machines (Serrano et al., 2004, p. 204).

Multicore Systems

If the purpose of using threads is to allow us to increase throughput and responsiveness (Muller et al., 2017, p. 677), then being able to *actually* do more than one thing at a time—rather than pretend to do so very convincingly, in the way that one CPU can "multitask" with multiprogramming and multitasking(Denning, 2017)—is an extremely effective game changer.

Multicore CPUs—that is, CPUs with multiple processing "cores" that act independently

of one another—are the bread-and-butter of personal computers today (Muller et al., 2017, p. 677). These are a boon to the *preemptive* style of multithreading, because (in a basic sense) the scheduler can let more than one process execute at the same time. However, this *does* increase the complexity of the underlying system, and significantly: The data they are all operating on must be protected from mutation by more than one process at a time, or corruption will occur.

A straightforward way to protect against this is a lockfile , though even that adds another layer of complexity to the system. And with more abstraction comes more implementation, and variety thereof: Preemptive threads are difficult to move from platform to platform, as there is no strict standard that they must adhere to (Serrano et al., 2004, p. 204).

Of course, *cooperative* threads, which are normally structured to execute sequentially and not prone to interruption, don't need such things: It's assumed that one—and only one—process is going to act until that process is finished, so there cannot be race conditions. Lockfiles are unnecessary. And thus, the implementation is solid and standardized (at least compared to the *preemptive* threads) across platforms.

However, as mentioned above, cooperative threads cannot reap the benefits of multi-core systems without modification . Instead of lockfiles, there are "cooperation points", which (combined with some other considerations for things like IO) allow cooperative threads to work better on a multi-core machine (Serrano et al., 2004, p. 204).

In the end, many believe that the solution lies somewhere in between one or the other, whether that's simply a mix of the two, or a specific and defined protocol: Serrano et al. (2004) have their "Fair Threads" scheme implementation, and Muller et al. (2017) has their special extension to StandardML.

Conclusion

It is perhaps worthy to note that there was no single standard that emerged over the 13 years between the above-mentioned papers: Perhaps this indicates another avenue of research is needed before we decide on a best-of-both-worlds standard. However, for now, threads still allow

the machines we use to be orders of magnitude more effective than they otherwise would be.

And regardless of how we make use of that hardware, we will always be asking to do just a little more.

References

Denning, P. J. (2017). Multitasking without thrashing. *Communications of the ACM*, *60*(9), 32–34. https://doi.org/10.1145/3126494

Erwig, M. (2017, August 11). Computation and algorithms: Hansel and gretel. *Once upon an algorithm*. MIT Press. https://doi.org/10.7551/mitpress/10786.001.0001

Muller, S. K., Acar, U. A., & Harper, R. (2017, June 14). Responsive parallel computation: Bridging competitive and cooperative threading. In A. Cohen & M. Vechev (Eds.), *Pldi 2017: Proceedings of the 38th acm sigplan conference on programming language design and implementation* (pp. 677–692). Association for Computing Machinery. https://doi.org/10.1145/3062341.3062370

Sattar, A., Mondshein, L., & Lorenzen, T. (2010). An operating systems course with projects in java (J. Impagliazzo, Ed.). *ACM Inroads*, *1*(2), 24–26. https://doi.org/10.1145/1805724.1805734

Serrano, M., Boussinot, F., & Serpette, B. (2004, August 24). Scheme fair threads. In D. S. Warren & E. Moggi (Eds.), *Ppdp '04: Proceedings of the 6th acm sigplan international conference on principles and practice of declarative programming* (pp. 203–214). Association for Computing Machinery. https://doi.org/10.1145/1013963.1013986

Silberschatz, A., Galvin, P. B., & Gagne, G. (2018, May 4). Synchronization tools. In D. Fowley, R. Dannelly, C. Nelson, K. Santor, & A. Pham (Eds.), *Operating system concepts* (10th ed., pp. 257–288). Wiley. https://isbnsearch.org/isbn/9781119320913

Sun, W.-F., Chan, S.-H., Liu, Z.-T., Yeh, Y.-H., & Chou, P. H. (2020). Quatros: A preemptive multithreaded embedded os for education (H. Yun & S. Quinton, Eds.). *SIGBED Review*, *17*(1), 49–55. https://doi.org/10.1145/3412821.3412829