Deadlocks

Christopher Rodriguez

Colorado State University Global

22SC-CSC300-1: Operating Systems and Architecture

Dr. Jonathan Vanover

2022-06-05

Deadlocks

In multi-threaded computing, “threads” are lightweight processes that can run concurrently with one another and share resources (Serrano et al., 2004, p. 203). This is a boon for computing efficiency, but it also introduces a few novel problems that can often be difficult to diagnose, track, and fix: As noted by Cai and Cao (2016), manually fixing these problems is error prone, tedious, and (in general) an unsolved problem (p. 1109) that is best prevented and avoided rather than fixed after the fact.

The main issue this paper focuses on is related to synchronization, or “Making sure processes stay in sync as they run concurrently”. Specifically, this paper addresses the *Deadlock*: one of three common synchronization errors (the other two being data races and atomicity violations) that is said to occur when some threads are irrevocably blocked from making further progress (Agarwal & Stoller, 2006, p. 51).

Deadlock Characteristics

Deadlock bugs are different from other types of synchronization bugs because they are not the result of missed synchronizations, but are the result instead of incorrect synchronization instructions in the first place—And this difference is so monumental that it has resulted in many techniques focused on mitigating synchronization bugs classifying such bugs as either “deadlock” or “non-deadlock” bugs (Cai & Cao, 2016, p. 1109).

It's worth mentioning the distinction between “Deadlock”, which this paper focuses on, and the rarer case of “Livelock”, where processes continually reattempt a failing action and therefore cannot succeed (Silberschatz et al., 2018, p. 321). In deadlocks, progress cannot continue due to endless waiting for resources. In livelocks, progress cannot continue *and* resources are being consumed by constantly-failing repetition.

There are generally two categories of deadlocks: “Resource Deadlocks”—when a set of threads are holding some locks and waiting for other locks held by members of their own set—and

"Communication Deadlocks"—when threads are awaiting a message or group of messages that will never arrive (Cai & Cao, 2016, p. 1109). Both arise from a failure in the logic of a concurrent system, and both need to be accounted for when designing concurrent systems.

Both also satisfy what Silberschatz et al. (2018) deem to be the conditions necessary to produce a deadlock: At least one resource is non-shareable ("Mutual Exclusion", a thread is waiting for additional resources while holding a resource ("Hold and Wait"), a resource cannot be released preemptively from a process before its completion ("No Preemption"), and the processes are waiting in a cyclic graph of some form ("Circular Wait") (p. 321). These four traits are useful because they are easy to conceptualize, and therefore easy to notice within a system design as it is being built.

Prevention and Avoidance

One general approach to avoiding synchronization issues is to introduce "gate locks", which are locks that are inserted specifically to allow processes to re-sync after drift might have occurred by serializing otherwise concurrent execution (Cai & Cao, 2016, p. 1109).. Other general approaches for synchronization issue prevention is the concept of "semaphores"—which can be used to create either be single- or dual-alternative decision-structure-like synchronization, and therefore add a large amount of complexity to programs—or "condition variables"—which allow for a more event-like signaling structure of synchronization (Agarwal & Stoller, 2006, p. 51).

However, all of these have the potential to create new deadlocks from their implementation, and therefore are *not* a "one-size-fits-all" solution to avoiding deadlocks—and such a thing may not exist, as each situation may need some combination of solutions to address (Agarwal & Stoller, 2006, p. 51; Cai & Cao, 2016, pp. 1109–1110; Silberschatz et al., 2018, p. 326).

Silberschatz et al. (2018) assert that there are three basic solutions available to deal with the deadlock problem which can be succinctly described as "Ignore the Problem", "Protocol-based prevention or avoidance", and "Detection after Deadlocks occur" (p. 326). Most modern operating systems use the first, applications generally try to implement the second, and the third is (rarely) used in specific systems (such as databases).

As it is the most common, this paper will focus on the second solution, “Protocol-based prevention or avoidance”. In this context, “Prevention” means putting rules in place to make it so that one of the four required conditions for a deadlock simply cannot occur. And “Avoidance” means to provide the Operating System with information about the resources a process may need during its execution, and allowing it to decide how and when to allow the processes to execute or wait (Silberschatz et al., 2018, p. 326).

In terms of prevention, the only generally-applicable method available is to prevent the “Circular Wait” condition from occurring through an ordered allocation of resources across all processes. The other conditions are either impractical outside of specific situations (“No Preemption”) or ineffective and prone to causing other issues (the others) (Silberschatz et al., 2018, pp. 326–328).

In terms of avoidance, the main solution of providing the kernel with information about what resources will be requested by each thread as it is spawned is only somewhat varied over the different solutions presented: In the simplest, the *maximum* for each resource is provided to the kernel. In the most complex, a more accurate amount of each resource is requested, and then various states and graphs are constructed by the OS to make a best-effort solution (Silberschatz et al., 2018, pp. 330–336).

Conclusion

Deadlocks are a problem that arises from faulty logic, not from emergent circumstance (Cai & Cao, 2016, p. 1109) and therefore cannot be trivially “solved for” once they occur, at least in operating systems as they exist today. Avoidance and Prevention are the best tools we, as programmers, have to attempt to deal with the deadlock problem, and though research on new methodologies to apply one or the other (or both) are constantly being researched, no clear solution has been created yet. However, remaining knowledgeable about deadlocks (what they are, how they occur, how to protect against them) will always be a useful tool when it comes to creating better and more resilient systems for people to use.

References

Agarwal, R., & Stoller, S. D. (2006, July 17). Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables. In L. Pollock & M. Pezzé (Eds.), *Padtad '06: Proceedings of the 2006 workshop on parallel and distributed systems: Testing and debugging* (pp. 51–60). Association for Computing Machinery (ACM). https://doi.org/10.1145/1147403.1147413

Cai, Y., & Cao, L. (2016, May 14). Fixing deadlocks via lock pre-acquisitions. In H. Asuncion, C. Zhang, & X. Zhang (Eds.), *Icse '16: Proceedings of the 38th international conference on software engineering* (pp. 1109–1120). Association for Computing Machinery (ACM). https://doi.org/10.1145/2884781.2884819

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. 317–346). Wiley. https://isbnsearch.org/isbn/9781119320913