# Concurrency Control

Physical constraints have resulted in frequency scaling being replaced by multi-processor architectures as the preferred future of hardware performance gains. If processor numbers increase then more parallel computations are required to maintain performance gains. Some problems may be trivially reflected in parallel solutions, yet many problems are not so easily solved in a parallel manner. For this reason, exploiting such architectures for increased performance and ensuring such solutions are free from error presents a substantial research challenge.

Concurrency control regulates access to shared state and therefore plays an important role in parallel computation. However, such regulation is achievable at a cost:

(1) *Limiting overall progress* - paths of execution are either blocked (hopefully temporarily) and/or their execution steps increased to afford coordination;

(2) *Lack of scalability* - Increasing contention of shared state reduces overall progress as described in (1).

(3) *Increased likelihood of error* - Programs may exhibit different interleaving of concurrent execution steps from one run to another, some of which may lead to unforeseen errors.

The items listed above are taken as an unavoidable outcome of concurrency control. Research efforts are made to improve (1) and (2) by considering different coordination techniques. Easing the programing interface coupled with testing and formalism are used to tackle (3). Interestingly, increasing parallelism promotes the difficulties found in the three items listed as a lack of scalability hinders overall progress and gives rise to more unforeseen execution interleaving that must be verified/tested.

## 1. The Dining Philosophers Example

In the dining philosophers example there are forks interspersed equally amongst a number of seated philosophers. The number of philosophers is equal to the number of forks. A philosopher is required to hold two forks in order to eat (could be fork and knife, chopsticks). Sometimes a philosopher may think for a while before attempting to eat. As there are not enough forks for all philosophers to eat simultaneously, a philosopher must occasionally wait for a fork to become available. This may lead to deadlock if all philosophers hold the fork to their left (or right) hand at the same time. In addition, the scenario may lead to "starvation" if some philosophers continuously utilize forks while other philosophers persistently miss out on acquiring forks. Finally there is a chance of livelock if all the philosophers continuously repeated the action of picking up the left, or right, fork at equal time intervals.

A concurrency control technique is required to ensure that events observable to philosophers (picking up and putting down forks) achieve a partial ordering that allows all philosophers to eat and none to starve. In essence, reads and writes that represent access to shared state must be ordered appropriately. In the dining philosophers example a read may assess the availability of a fork and a write changes a fork's availability. If *P* is used to denote a dining philosopher then the ordering of their reads and writes to reflect an ability to eat would be: {$P_{read}$*fork1* (available), $P_{write}$*fork1* (pick up), $P_{read}$*fork2* (available), $P_{write}$*fork2* (pick up), $P_{write}$*fork1* (put down), $P_{write}$*fork2* (put down)}. The priority of concurrency control is to interleave such orderings so deadlock is avoided, livelock is avoided and nobody starves.

## 2. Synchronization

There are operational primitives in most modern hardware that trivialize the support of thread synchronization. Such primitives provide uninterruptable memory access/update. *Compare-and-swap* (CAS) and *load-linked/store-conditional* (LL/SC) are two such mechanisms and may be used to construct controlled access to concurrent objects. Concurrent objects represent stateful programming constructs that multiple threads may attempt to access simultaneously.

The correctness criteria associated to the concurrent history of an object may be reasoned about. *Linearizability* is a correctness criterion that states, "*if one method call precedes another, then the earlier call must have taken effect before the later call*". This property provides the determinism needed to afford reasoned judgment on a program's overall correctness. For example, if thread $T_1$ places item $x$ on a LIFO stack and then requests an item then one of two outcomes is possible: $T_1$ receive $x$, or $T_1$ receives something else (including *null*). "Something else" may occur if other threads had interacted with the stack in-between $T_1$'s calls. Linearizability ensures that all threads involved in the creation of a concurrent history have mutually consistent behavior: the individual sequential histories of each thread are achievable given the overall linearizability of a concurrent object. Furthermore, linearizability is compositional in that if concurrent objects are linearizable then linearizability holds cumulatively across all such objects.

Using linearizability and operational primitives the creation of concurrent objects regulating access via critical sections is possible. This can provide a lock-based approach to the creation of the read/write schedule for dining philosophers.

As both forks are needed to eat in the dining philosophers problem a programmer may use mutual exclusion to ensure that a philosopher locks both forks. This can be trivially achieved by allowing a single lock to represent all forks present. The table itself is the concurrent object. Irrelevant of the number of philosophers there will be no deadlock or livelock as all the forks are either available or not available. In addition, we can enforce ordering to ensure philosophers take it in turn to eat and avoid starvation. However, assuming we want a concurrent solution we must allow philosophers to dine simultaneously when the opportunity arises.

The greatest potential for concurrency is to have locks on a per-fork basis. This allows each fork to become a concurrent object to be used by the programmer in their solution. However, the programming skills required in deriving solutions that afford resource allocation to avoid deadlock, livelock and starvation are not trivial. Although linearizability may hold across the table of forks, linearizability alone does not prevent deadlock, livelock or starvation. Linearizability simply provisions deterministic behavior of the concurrent objects themselves.

An alternative to locking and mutual exclusion would be to attempt a solution in a *wait-free* manner. A concurrent object is considered wait-free if all its calling methods finish in a finite number of steps. The more important quality is that of *bounded wait-free*, where the finite number of steps is bounded (and hopefully known). Wait-free solutions have the benefit of being free from deadlock.

To enable the implementation of a wait-free object the problem of consensus must first be solved. Consensus, generally speaking, is agreement amongst participants on an item suggested by one or more of the participants. If consensus can be implemented in a wait-free manner for $n$ participants ($n$ is the consensus number) then it follows that wait-free concurrent objects may be constructed for $n$ participants. This is an intuitive statement: if an algorithm can advance its state based on periodic consensus of deterministic threads in a determined number of steps then the overall algorithm exhibits determinism. Herlihy's "*wait-free hierarchy*" indicates that operational primitives (CAS being one of them) may solve consensus for an infinite number of threads. A programmer can construct wait-free solutions for any concurrent object on most modern hardware infrastructures.

Wait-free solutions are not the standard approach in many instances because their lock-based counterparts usually exhibit greater efficiency in terms of execution steps and memory usage (which rise as thread numbers rise). The conclusions of succinctly put this argument across: even though CAS and LL/SC may be appropriate for solving wait-free consensus, they are weak in terms of efficiency for most multi-valued objects. As such, non-blocking solutions are rarely attempted in practice.

Irrelevant of the approach taken (e.g., locking, wait-free) and the skill of the programmer, there is one insurmountable difficulty when basing a solution completely on critical sections:

- Concurrent objects are not composable in the context of programming logic.

Although linearizability is compositional, this does not result in achieving the programmer's intent. For example, if two linearizable bank account objects afforded the methods withdraw and deposit, one may want to ensure that a thread must withdraw money from one account and deposit the money in the other account without interference. If other threads could interferer then they could see both accounts with money missing and make incorrect assumptions of the cumulative value of the accounts.

## 3. Optimism

To allow context aware composability, concurrency control may be implemented in a manner similar to that used in databases. In a database, transactions ensure ACID properties are maintained in the presence of conflicted read/writes. In software this approach is embodied within transactional memory. A programmer identifies a number of execution steps that must be achieved atomically. As these steps may span a number of concurrent objects we can implement the earlier bank account example with no difficulty.

Serializability is the correctness criteria most often cited for databases. However, the stronger correctness criteria linearizability is commonly used within transactional memory (there are others, e.g.,). In principle, a read/write schedule on shared state is constructed not explicitly by the programmer via synchronization primitives, but by the transactional sub-system.

The optimistic approach is popular in transactional memory as it allows shared state to be enacted upon simultaneously. The optimistic approach advocates that threads may proceed with local copies of shared state. However, they may not be capable of committing changes to the actual shared state due to the actions of other transactions. Transactions unable to commit will be aborted. Those transactions that are committed maintain a linearizable read/write schedule. Aborted transactions may retry at a later date (but may be aborted again).

In the dining philosophers problem the complete act of dining (pick up both forks, eat, put down both forks) can be encapsulated within a single transaction and attempted in succession by a philosopher. Linearizability would ensure that the schedule of reads and writes would be adhered to. Deadlock would be avoided as those diners who have eaten but couldn't commit this act to the schedule would be rolled back and attempt to eat again (they didn't *really* eat). If a more pessimistic transactional approach was employed a philosopher could be rolled back earlier in their transaction (identifying possible conflicts in the schedule earlier in the dining process). Transactions would allow concurrency as diners not sharing forks could proceed in parallel and there is no need for the programmer to be concerned with deadlock or livelock.

Using transactions alone would not prevent starvation because there is no guarantee that philosophers would be able to dine (their transactions may be rolled back more frequently than others). Therefore, many solutions that utilize software transactional memory in their approaches do propose a number of techniques to alleviate this issue of contention management (e.g., passive, polite, karma, greedy).

A transactional approach has many benefits over a solution based only on mutual exclusion. They are composable in that the correctness criteria together with appropriate contention management scheme are sufficiently general to be almost universally applicable. This allows the programmer to construct complex parallel solution without programming problem-centric solutions based on mutual exclusion. However, they do have drawbacks.

The major drawback of transactional approaches is the construction of the read/write schedule in a delayed manner. Transactions that rollback hinder forward progress of their threads (philosopher would eat, but then be told they hadn't eaten really). In the context of a program execution, it may be that some threads could have committed even if linearizability was invalidated by such a transaction (particularly the case for longer transactions). In addition, the schedule creation process itself may not be a scalable act if transaction numbers rise on a contended data item (also a problem with blocking and wait-free approaches). Finally, programmer determined synchronization of threads is available only at the granularity of a transaction, promoting the use of long-lived transactions (which have a higher chance of rollback). For example, if a programmer requires two functions to be applied in sequence and without interference to one million instances of a concurrent object then this must be combined within one transaction.

## 4. Summary

In one sense the summary to this area is that we just can't create suitable concurrency control mechanisms without limiting the parallel execution present in a system. Concurrency control is difficult to understand and is a major difficulty for all programmers. Look through the literature listed in the bibliography used to write this brief overview and you will discover just how tricky this area of research is.

## 5. Bibliography

Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463-492. DOI=10.1145/78969.78972 http://doi.acm.org/10.1145/78969.78972

Judit Bar-Ilan and David Peleg. 1992. Distributed Resource Allocation Algorithms (Extended Abstract). In *Proceedings of the 6th International Workshop on Distributed Algorithms* (WDAG '92), Adrian Segall and Shmuel Zaks (Eds.). Springer-Verlag, London, UK, 277-291.

Eugene Styer and Gary L. Peterson. 1988. Improved algorithms for distributed resource allocation. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing* (PODC '88). ACM, New York, NY, USA, 105-116. DOI=10.1145/62546.62567 http://doi.acm.org/10.1145/62546.62567

Daniel Lehmann and Michael O. Rabin. 1981. On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (POPL '81). ACM, New York, NY, USA, 133-138. DOI=10.1145/567532.567547 http://doi.acm.org/10.1145/567532.567547

David Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In Proceedings of the 20th International Symposium on Distributed Computing (DISC), pages 194--208, September 2006.

Maurice Herlihy , Victor Luchangco , Mark Moir , William N. Scherer, III, Software transactional memory for dynamic-sized data structures, Proceedings of the twenty-second annual symposium on Principles of distributed computing, p.92-101, July 13-16, 2003, Boston, Massachusetts [doi>10.1145/872035.872048]

William N. Scherer, III and Michael L. Scott. 2005. Advanced contention management for dynamic software transactional memory. In Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing (PODC '05). ACM, New York, NY, USA, 240-248. DOI=10.1145/1073814.1073861 http://doi.acm.org/10.1145/1073814.1073861

Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. 2005. Toward a theory of transactional contention managers. In Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing (PODC '05). ACM, New York, NY, USA, 258-264. DOI=10.1145/1073814.1073863 http://doi.acm.org/10.1145/1073814.1073863

B. Awerbuch and M. Saks. 1990. A dining philosophers algorithm with polynomial response time. In Proceedings of the 31st Annual Symposium on Foundations of Computer Science (SFCS '90). IEEE Computer Society, Washington, DC, USA, 65-74 vol.1. DOI=10.1109/FSCS.1990.89525 http://dx.doi.org/10.1109/FSCS.1990.89525

Michael O. Rabin and Daniel Lehmann. 1994. The advantages of free choice: a symmetric and fully distributed solution for the dining philosophers problem. In A classical mind, A. W. Roscoe (Ed.). Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK 333-352

Jeff Kramer and Jeff Magee. 1990. The Evolving Philosophers Problem: Dynamic Change Management. IEEE Trans. Softw. Eng. 16, 11 (November 1990), 1293-1306. DOI=10.1109/32.60317 http://dx.doi.org/10.1109/32.60317

Torval Riegel, Christof Fetzer, Heiko Sturzrehm, and Pascal Felber. 2007. From causal to z-linearizable transactional memory. In Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing (PODC '07). ACM, New York, NY, USA, 340-341. DOI=10.1145/1281100.1281162 http://doi.acm.org/10.1145/1281100.1281162

Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. 2005. Composable memory transactions. In Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '05). ACM, New York, NY, USA, 48-60. DOI=10.1145/1065944.1065952 http://doi.acm.org/10.1145/1065944.1065952

Kung, H., T. and Robinson, T., John, On optimistic methods for concurrency control. ACM Trans. Database Syst, 6, 2: p. 213-226. 1981.

Brian Randell. System structure for software fault-tolerance. *IEEE Transactions on Software Engineering*, SE- 1(2):220–232, 1975.

Maurice Herlihy. 1991. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.* 13, 1 (January 1991), 124-149. DOI=10.1145/114005.102808 http://doi.acm.org/10.1145/114005.102808

Maurice P. Herlihy. 1988. Impossibility and universality results for wait-free synchronization. In*Proceedings of the seventh annual ACM Symposium on Principles of distributed computing* (PODC '88). ACM, New York, NY, USA, 276-290. DOI=10.1145/62546.62593 http://doi.acm.org/10.1145/62546.62593

Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of distributed consensus with one faulty process. *J. ACM* 32, 2 (April 1985), 374-382. DOI=10.1145/3149.214121 http://doi.acm.org/10.1145/3149.214121

Loui, M. C., and Abu-Amara, H. H. (1987), Memory requirements for agreement among unreliable asynchronous processes, in "Advances in Computing Research," Vol. 4, pp. 163-283, JAI Press, London.

Faith Fich, Danny Hendler, and Nir Shavit. 2004. On the inherent weakness of conditional synchronization primitives. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing* (PODC '04). ACM, New York, NY, USA, 80-87. DOI=10.1145/1011767.1011780 http://doi.acm.org/10.1145/1011767.1011780

Vallejo, E., Sanyal, S., Harris, T., Vallejo, F., Beivide, R., Unsal, O.S., Cristal, A., and Valero, M. Hybrid Transactional Memory with Pessimistic Concurrency Control. In Proceedings of International Journal of Parallel Programming. 2011, 375-396.

D. Padua, D. Kuck, and D. Lawrie. High-Speed Multiprocessors and Compilation Techniques. IEEE Transactions on Computers, C-29(9):763-776, September 1980.

Dongkeun Kim and Donald Yeung. 2002. Design and evaluation of compiler algorithms for pre-execution. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems* (ASPLOS-X). ACM, New York, NY, USA, 159-170. DOI=10.1145/605397.605415 http://doi.acm.org/10.1145/605397.605415

D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. 1995. Managing update conflicts in Bayou, a weakly connected replicated storage system.*SIGOPS Oper. Syst. Rev.* 29, 5 (December 1995), 172-182. DOI=10.1145/224057.224070 http://doi.acm.org/10.1145/224057.224070

A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. Journal of the ACM, 32(3):733–749, 1985

G.H. Hwang, K.C. Tai, and T.L. Huang, “Reachability Testing: An Approach to Testing Concurrent Software,” Int'l J. Software Eng. and Knowledge Eng., vol. 5, no. 4, pp. 493-510, 1995.

Torval Riegel, Christof Fetzer, Heiko Sturzrehm, and Pascal Felber. 2007. From causal to z-linearizable transactional memory. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing* (PODC '07). ACM, New York, NY, USA, 340-341. DOI=10.1145/1281100.1281162 http://doi.acm.org/10.1145/1281100.1281162