Liveness in Transactional Memory

Jan Frederik Haltermann

Paderborn University Institute for Computer Science,
33098 Paderborn, Germany
jfh@mail.upb.de

Abstract. Today, a software system does not simply execute different programs one after another, they run in parallel, realized using threads or multi core systems. A software transactional memory (STM) simplifies parallel execution of programs by controlling and synchronizing shared memory. Besides correctness of an STM, it is important to ensure liveness in such an algorithm to avoid phenomena like deadlocks or livelocks. In this paper a formal definition of three different liveness properties is presented and a comparison to existing liveness criteria used for concurrent programming is given. Moreover, it is proven that it is impossible for an STM algorithm in a fault prone system to ensure opacity, a widely used correctness criteria, and local progress, the strongest liveness properties. Finally, it is shown that the previous result can be generalized and even holds for strict serializability..

Keywords: Software transactional memories, liveness, correctness, opacity

1 Introduction

Parallelism is a key factor to speed up the execution of different programs. Instead of executing them one after another, threads or multi-core processors are used. One concept to synchronize several threads is to use locks. A programmer can define a critical section, in which a threat should be executed atomically. These locking mechanisms has some drawbacks, like potential deadlocks or the need for a manual handling of locks. A software transactional memory (STM) can circumvent these problems. Programmers can define transactions for critical sections, whose execution is managed by the STM, controlling memory shared between the different threads.
Correctness criteria for STM implementations like opacity guarantee that 'nothing bad happens'[1], like a process reading an invalid memory location. An algorithm that aborts every transaction directly will guarantee every existing correctness criteria, e.g. opacity. But this algorithm does not behave like it is expected. For that reason, a formal definition of liveness is needed, meaning that 'something good happens'[1], in this case the successful execution of transactions. In the context of concurrent programming, several liveness properties are defined:
In 1991, Herlihy defines the liveness properties *wait-freedom* and *non-blocking*[8]. A concurrent system is wait-free, if it guarantees that no operations are pending,

meaning every operation receives eventually a result. Moreover, every response to a currently pending request can be answered without waiting for other different operations. This definition focuses on termination of operations, meaning that every operation will eventually receive a result, not necessarily a positive result. A weaker property is *non-blocking*, ensuring that some processes will receive a result for their requested operations. *Obstruction-freedom*, defined by Herlihy et al. in 2003 [7] is weaker than non-blocking. If a system is obstruction-free, a response to a operation is guarantees, iff the process is running isolated. All definitions focus on the termination of a concurrent program. They do not fit into the context of STMs, since they only enforce responses to called operations on shared objects. In contrast to concurrent programming, an STM may abort transactions and restart them. An STM implementation that aborts every transaction directly, will ensure wait-freedom, since no transactions needs to wait or is blocked. However, this is not the expected behavior of a STM, thus stronger definitions for liveness properties are needed. These definitions should take the progress of operations into account, meaning that write or read operations on shared memory are executed successfully. They will not guarantee a correct execution of transactions, therefore checking which liveness properties and correctness criteria can be fulfilled in one STM implementation, is of great importance.
In Section 2 a formal definition of the system is given and formal definitions for STM liveness properties are given in Section 3. The Section 4 answers the question, which liveness properties and correctness criteria cannot be fulfilled by a single STM. Finally, the results are summarized and a outlook is given in Section 5.

2 Fundamentals

A system consists of at least one shared object and n asynchronous processes, denoted by $p_1, \ldots, p_n$. A process represents a thread and communicates with other processes via the shared object. Processes can access a shared object via operations. These operations are realized using several base operations like compare-and-swap or load. Each operation starts with an invocation event and ends with a response event.

2.1 Histories

A *configuration* of the system is a snapshot of the states of all transactions and all variables at a certain point in time. Initially, the system is in a ideal state denoted by C_0. Every time an operation is executed, the configuration of the system changes, denoted by a *step* s. Each step is either a invocation event of an operation, a response event on an operation or the execution of a base operation. Base operations used are read, write and try to commit. A read on variable x by process i is denoted $x.read^i$ and will return either the value v ($x.read^i \to v$) or the transaction is aborted, denoted by A^i. A *write* of a value w to a variable x by

process i is denoted by $x.write(w)^i$ and return either *ok* if the write wuccessful or an abort of the transaction A^i. A commit-request of process i, is denoted by $tryC^i$ and either the transaction is committed, denoted by C^i, or aborted, denoted by A^i. In the following, return values of a positive write and all *tryC* events are not displayed to increase the readability of the figures.
An *execution* α of a system can therefore be formalized as the changes of system states through steps, $\alpha = C_0 \xrightarrow{s_0} C_1 \xrightarrow{s_1} C_2 \ldots$. A *history* H is the longest subsequence of operations on a single shared object, containing only invocation and response events. Intermediate steps, like a compare-and-swap base operation on a shared object, are omitted to increase readability.

2.2 Transactions

A *transaction* t is defined of a sequence of operations $e_1, \ldots, e_k$, executed by a process p_i. The operations $e_1, \ldots, e_{k-1}$ are read or write operations, where e_1 is either the first operation of process p_i or is succeeds a commit or an abort event. The last event e_k of a transaction t is either a commit event C^i or an abort event A^i.

A transaction t_1 precedes another transaction t_2,$t_1 \leq t_2$, iff the commit or abort event of t_1 happens before t_2 invokes its first operation. If neither t_1 precedes t_2 nor t_2 precedes t_1, they are concurrent. Transactions can either be *crashed*, *parasitic* or *correct*. A transaction t is crashed, iff t performs a finite number of operations in an infinite execution α, meaning that t does not perform any operation from a point in time on. In contrast, a parasitic transaction takes infinitely many steps in the execution but after a point in time it does not try to commit. If a transaction is prematurely aborted, it is unclear whether the it tries to commit or not, thus the transaction is not parasitic. Formally, a transaction t is parasitic, iff in an infinite execution α a suffix α' can be found such that t does perform infinitely many operations that neither are *tryC* events nor a negative response A. Transactions are correct, iff they are neither crashed nor parasitic.
A system is crash-prone, if transaction can crash and it is parasitic-prone if a transaction can be parasitic. In Figure 1 each process is executing one transaction The first transaction t_1 repeatedly writes the value 1 to variable x and does not try to commit. In contrast, the transaction t_2 performs only a single write on x and does not perform any further steps. The transaction t_1 is parasitic and t_2 is crashed.

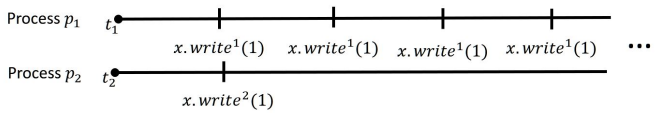

Fig. 1. Example for parasitic and crashed transactions

2.3 Histories

As stated before, a *history* H is the longest subsequence of operations on a single shared object, containing only invocation and response events. A history is called complete, iff the last event of a transaction t (that is executed by p_i) in H is either an A^i event or an C^i event. A history is called sequential, iff there are no concurrent transactions in H. In a history, it is not possible to distinguish crashed transactions from transactions, that are waiting (infinitely) long for a response event. Therefore, the concept of fair histories in crash prone systems is introduced, meaning that in a fair histories F_α crashed transactions are marked. To obtain a fair history, a event $crash^k$ is inserted directly after e, where e is the last event, executed by a crashed transaction t_k. Two histories H, H' are equivalent, if every process behaves equivalent, or formally, iff $H_{|p_k} = H'_{|p_k}$ for every process p_k. The notation $H_{|p_k}$ simply means that all other processes except p_k are removed from H.
As stated in the first section, it is not enough for an STM liveness criteria to require that an operation receives a result. Especially, the non-aborting return value of an invoiced operation is relevant. This idea of liveness is captured in the definition of *progress* for a process. Formally, a process p_k makes progress, if an infinite history contains infinitely many C^k events. This implies that all operations in all transactions executed by process p_k will eventually return a non-aborting result. A process p_k *runs alone* in a history H, if there exists a suffix H' such that p_k is the only correct process in H'.

3 Formal Definition of Liveness Properties

As stated in Section 1, the formal definitions of liveness given by Herlihy [6, 8] are not sufficient for STMs. Thus, the formal definition of the liveness properties *local progress*, *global progress* and *solo progress* is given by Bushkov and Guerraoui[2]. A liveness property L is defined as a set of histories, where each history fulfills the liveness property. An STM algorithm A ensures a liveness property L, iff the following holds: All fair histories F_α of the configurations α produced by A are in L ($F_\alpha \in L$). A liveness property L_1 is stronger than another liveness property L_2, iff $L_1 \subseteq L_2$. L_1 is weaker than L_2 iff $L_1 \supseteq L_2$, otherwise they are incomparable.

3.1 Local Progress

An infinite and fair history ensures local progress, iff every correct process p_k makes progress in H or H does not contain any correct processes. Denote by L_{local} the set containing all histories ensuring local progress. An example for a history that fulfills local progress is given in Figure 2. Therein, the transactions $t1$ and $t3$ are committed and since the four transactions are repeated infinitely often, both processes contain infinitely many commit events and therefore make progress. Comparing local progress to wait-freedom, it holds that $L_{local} \subseteq L_{wait}$,

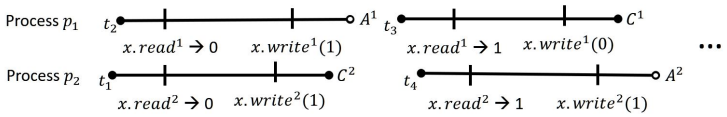

Fig. 2. Example for a history fulfilling local progress, adapted from [2]

since making progress is stronger than receiving a result. L_{wait} denotes the set of histories that are wait-free. On the other hand it holds that $L_{wait} \not\subseteq L_{local}$, since a history H where every transaction is aborted is wait-free but does not ensure local progress.

3.2 Global Progress

An infinite and fair history ensures global progress, iff at least one correct process p_k makes progress in H or H does not contain any correct processes. Denote by L_{global} the set containing all histories ensuring global progress. An example for a history that fulfills global progress is given in Figure 3. Therein, all transactions executed by p_2 are committed and all transactions executed by p_1 are aborted. Since the history is infinite, p_2 contains infinitely many commit events and therefore makes progress. Comparing global progress to non-blocking, it

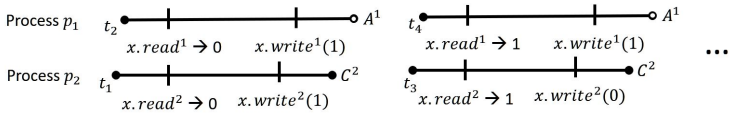

Fig. 3. Example for a history fulfilling global progress, adapted from [2]

holds that $L_{global} \subseteq L_{lock}$, where L_{lock} denotes the set of non-blocking histories. On the other hand it holds that $L_{lock} \not\subseteq L_{global}$, since in a history H where every transaction is aborted, H is non-blocking but does not ensure global progress. L_{global} and L_{wait} are incomparable.

3.3 Biprogressing

An infinite and fair history F is biprogressing, iff F contains at least two correct process only if at least two processes make progress in F. Denote by L_{bi} the set containing all histories that are biprogressing. Biprogressing is weaker compared to local progress $L_{bi} \subseteq L_{local}$, but stronger than global progress $L_{global} \subseteq L_{bi}$ and incomparable to wait-freedom.

3.4 Solo Progress

An infinite and fair history ensures solo progress, iff every correct process p_k makes progress in H, if it runs alone or H does not contain any correct processes. Denote by L_{solo} the set containing all histories ensuring global progress. An

example for a history that fulfills solo progress is given in Figure 4. Therein the process p_1 makes progress from the point one where it is running alone, namely when t_3 is crashed. Comparing solo progress to obstruction-freedom, it holds that

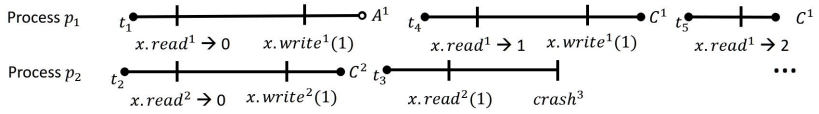

Fig. 4. Example for a history fulfilling solo progress, adapted from [2]

$L_{solo} \subseteq L_{obst}$, where L_{obst} denotes the set of histories that are obstruction-free. On the other hand it holds that $L_{obst} \nsubseteq L_{solo}$, since a history H where every transaction is aborted if it runs alone is obstruction-free but does not ensure solo-progress. L_{solo} and L_{lock} are incomparable.

3.5 Comparison of Liveness Properties

In Figure 5 the liveness criteria presented in section 1 and 3 are compared regarding their expressiveness. The figure contains all existing histories (number 8). The strongest liveness criteria is local progress(1), as stated by Bushkov and Guerraoui [2]. Weaker properties are wait-freedom(2), biprogessing(3), global progress(4), non-blocking(5) and solo progress(6). The weakest is of the presented liveness properties is obstruction-freedom(7).

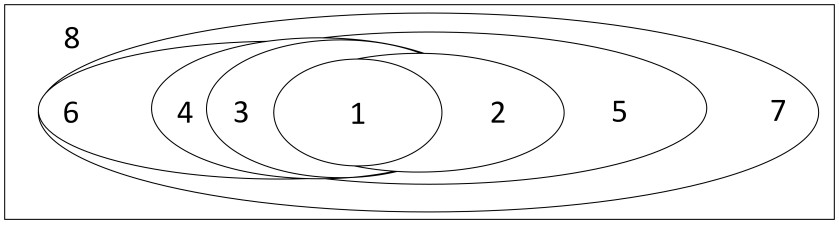

Fig. 5. Venn diagram comparing different liveness properties

4 Nonexistence of Opacity and Local Progress

After having defined liveness properties, that an STM algorithm can fulfill, the question arises which liveness properties and which correctness criteria can be guaranteed in a single STM. In this paper two widely used correctness criteria, *opacity* [4] and *strict serializability*[10], will be used for the comparison. Formally, a STM implementation is opaque iff there exists an equivalent sequential and complete history H', such that H' preserve the real time ordering of H and that every transaction t_i in H' is legal [4]. Important to notice that opacity requires

that committed and aborted transaction are legal. Moreover, the definition by Guerraoui and Kapalka used is not prefix-closed. A prefix-closed definition, presented by Guerraoui and Kapalka in 2010, would additionally require that every prefix of H needs to be opaque[5]. Papadimitriou defined strict serializability as correctness criterion for databases, but it became also important in the context of STM [10]. A history H ensures strict serializability, iff there exists an equivalent sequential and complete history H' containing only committed transactions, such that H' preserves the real time ordering of H and that every transaction t_i in H' is legal.
One main result, proven by Bushkov and Guerraoui [2], is stated in the following theorem:

Theorem 1. *For any fault prone system there does not exists a STM implementation ensuring local progress and opacity, if the process behavior is not known in advance.*

The proof uses the non-prefix-closed definition of opacity, but Bushkov and Guerraoui argue that the following proof also works for prefix-closed opacity[2]. Since it is assumed that the system is error-prone, the proof is given for two cases: First, if the system is assumed to be crash-prone, shown in Section 4.1 and secondly in Section 4.2, if the system is assumed to be parasitic-prone. The theorem is proven by counter-position, assuming that there exist an STM ensuring local progress and opacity. Firstly, an adversary is constructed aiming to execute two processes in such a way, that either local progress or opacity is violated. Therefore, it plays a predefined strategy presented in Figure 6 and in Figure 8. In both strategies a transaction will take steps until receiving a result for its operation and the other transaction does not take any steps in between. If an operation will not receive a result for an operation, it will take infinitely many steps. The process is correct but does not make progress, leading to a contradiction of local progress. Thus, all transactions are obstruction-free.

4.1 Proof for a crash-prone system

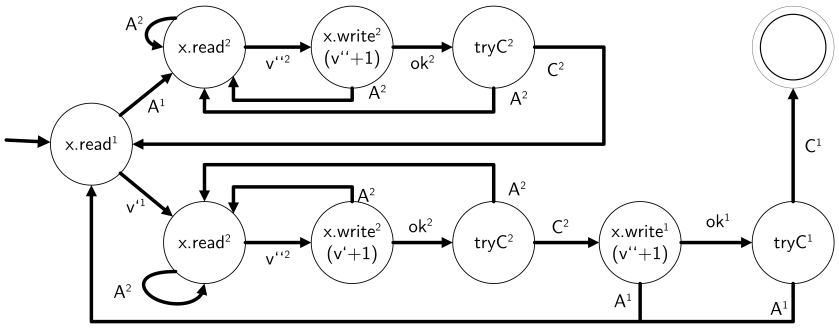

Fig. 6. Strategy 1 of the adversary against a STM in a crash-prone system

As stated above, every finite history needs to be opaque. The only way the given strategy terminates is that the $tryC^1$ event receives a C^1 as response, given that all operations are obstruction-free. The last two transactions of the resulting concurrent history are given in Figure 7. Since the STM algorithm is assumed to be opaque, there must exists an equivalent sequential history. Both possible real-time orderings are given in Figure 7. Non of them is legal, since the second transaction reads an invalid value. Hence, the STM algorithm will never

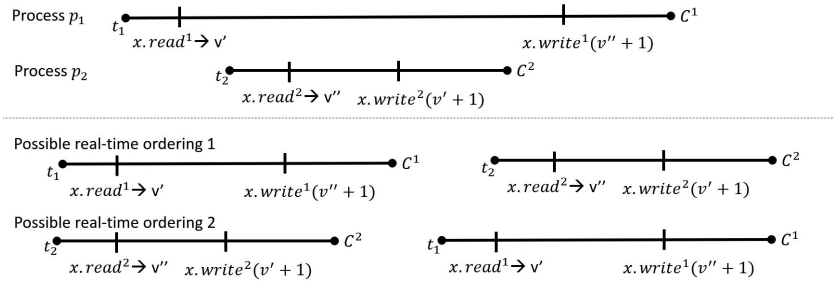

Fig. 7. Possible real-time orderings for the history obtained by strategy 1

return a C^1, because the resulting history will not be opaque, thus the strategy will never terminate. Thereby, process p_1 does not make progress in any infinite history obtained by strategy 1, because the STM will never return a C^1 as response to a $tryC^1$ event. Transactions executed by p_2 cannot be crashed, since they will always be executed after the $x.read^1$, independently of the returned result. Afterwards, p_2 will take steps until all transaction is committed. Since a transaction executed by p_1 will never commit, the strategy will never terminate and the starting state of the automaton will be reached infinitely often. Showing that p_1 is not crashed leads to a contradiction to the assumption. Process p_1 can only crash after the execution of $x.read^1$, if every $tryC^2$ event is aborted. Otherwise, either $x.write^1(v''+1)$ or $x.read^1$ is executed and p_1 is not crashed. Process p_2 is correct and makes process, therefore $tryC^2$ will eventually receive a C^2. Thus, p_1 is correct and does not make progress in H, a contradiction to the assumptions.

4.2 Proof for a parasitic-prone system

Strategy 2 will never terminate, since any finite history produced by Strategy 2 has the postfix shown in Figure 7 and violates opacity. Hence, p_1 will never receive a C^1 and therefore does not make any progress. Process p_1 is parasitic, iff p_2 never receives a C^2 or the transactions of p_1 are not permanently aborted (otherwise p_1 is correct by definition). Every transaction executed by p_2 is either aborted permanently or contains a $tryC^2$ event, thus, p_2 is correct. Therefore, p_2 will eventually make progress, meaning that p_1 is not parasitic but does not make any progress. This is a contradiction to the assumptions. □

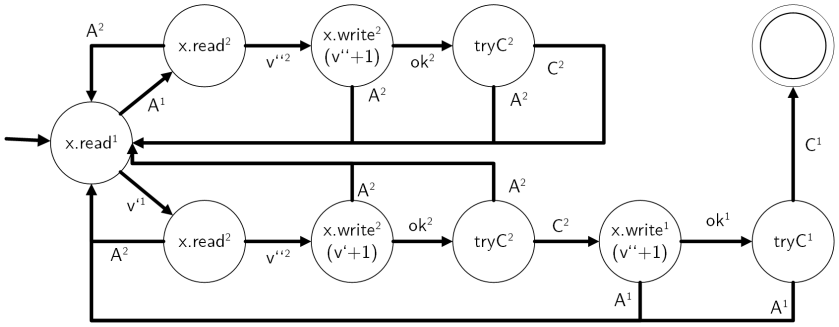

Fig. 8. Strategy 2 of the adversary against an STM in an parasitic-prone system

4.3 Generalized Result

Theorem 2. *For any fault prone system and any liveness property L that is biprogressing and stronger than solo progress, there is no STM implementation that ensures L and strict serializability.*

The proof for Theorem 2 for a crash-prone system works analogous as in Section 4.1: The adversary uses Strategy 1 from Figure 6 in a crash-prone system. Strategy 1 will also never terminate, because the reasoning from Section 4.1 does not take differences between aborted and committed transactions into account. For that reason the arguments used for opacity can be applied to strict serializability. Using a similar argumentation as in Section 4.1, p_1 is correct. If p_1 would be crashed, p_2 would run alone and makes progress. Thereby, p_1 will eventually perform operations and is not crashed. The infinite history F obtained by an execution of strategy 1 will not contain any C^1 event, thus, p_1 does not make progress. Since p_1 and p_2 are correct, biprogressing is not ensured, a contradiction to the generalized assumption.
In a parasitic-prone system, the adversary is playing Strategy 2 from Figure 8. Using the same arguments as in Section 4.2, it can be shown that the Strategy 2 will never terminate and produced only infinite histories. As argued before, p_2 is correct in an arbitrary execution α. Process p_1 does not make progress in α, thus, biprogresing is ensured, iff p_1 would be parasitic. As stated in Section 4.2, p_1 is parasitic, iff p_2 does not make progress. If p_1 is parasitic, p_2 is the only correct process and therefore runs alone. Since the system ensures solo-progress, p_2 makes progress. Thus, p_1 cannot be parasitic.

5 Conclusion

In this paper, formal definitions of liveness properties are given, namely for local progress, global progress and solo progress. The three properties are compared regarding their expressiveness with properties introduced for concurrent programming. The main differences are outlined and the incomparability between

some properties is exemplary shown. Furthermore, it is proven that it is impossible for an STM to ensure the liveness property local progress and the correctness criteria opacity. This result is generalized, by showing that it is impossible to ensure strict serializability as correctness criteria and biprogressing and soloprogress as liveness criteria, assuming that the given system is error-prone.

There are several options to circumvent this fact and ensure liveness properties and correctness conditions in a STM: Firstly, if the system is assumed to be fault free, opacity and local progress can both be ensured in a system using deferred update, as shown by Lesani and Palsberg[9]. A second option is to weaken the liveness property. One example for an STM algorithm, that ensures global progress and opacity, is OSTM, developed by Fraser in 2010 [3]. Moreover, weakening the assumptions made will circumvent the problem. In case all operations executed are known in advance, a STM is able to determine a schedule that ensures opacity and local progress. Thereby, the option to execute specific operations based on previous results is impossible, thus the strategy used in Section 4 will not work anymore. The approach to compute every possible scheduling in advance is inefficient, but will work for a finite number of processes.

In further research two points are of interest: On the one hand it should be investigated if local progress is the strongest liveness property or if even stronger liveness properties can be defined. On the other hand, a generalization of the impossibility results presented in Section 4.3 is investigated as one field of research.

References

[1] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 1985.
[2] Victor Bushkov and Rachid Guerraoui. Liveness in Transactional Memory. In *Transactional Memory. Foundations, Algorithms, Tools, and Applications*, pages 32–49. Springer, Cham, 2015.
[3] Keir Fraser. *Practical lock-freedom.* PhD thesis, University of Cambridge, 2004.
[4] Rachid Guerraoui and Michał Kapałka. Opacity : A Correctness Condition for Transactional Memory. *Technical Report LPD-REPORT-2007-004*, 2007.
[5] Rachid Guerraoui and Michał Kapałka. Principles of Transactional Memory. *Synthesis Lectures on Distributed Computing Theory*, 1(1):1–193, jan 2010.
[6] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: double-ended queues as an example. In *23rd International Conference on Distributed Computing Systems, 2003. Proceedings.*, pages 522–529. IEEE.
[7] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: double-ended queues as an example. In *23rd International Conference on Distributed Computing Systems, 2003. Proceedings.*, pages 522–529. IEEE, 2003.
[8] Maurice P. Herlihy. Wait-free synchronization. *Toplas*, 13(1):124–149, jan 1991.
[9] Mohsen Lesani and Jens Palsberg. LNCS 8205 - Proving Non-opacity.
[10] Christos H. Papadimitriou and Christos H. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, oct 1979.