Summary of Data Races Solution Algorithms for Multithreaded Programs

Chun Fang(✉)

School of Computer Science, Hubei University of Technology, Wuhan 430068, China
535974648@qq.com

Abstract. As we know that the multithreaded programs are easily to produce data races during the running, and it is really hard for us to locate the position where the mistakes are. After reading the Eraser: A Dynamic Data Race Detector for Multithreaded Programs which introduces "Lockset" algorithm by explaining the shortcomings of the classic "Happens-Before" algorithm [1], and then optimizes the tool Eraser on the basic "Lockset" algorithm, I have summarized the paper and put forward my own views.

Keywords: Happens-before · Lockset · Eraser

1 "Happens-Before" Algorithm

1.1 Features of Algorithm

If a synchronization object is accessed by two threads at the same time, and the synchronization semantics forbids the use of a time series in the thread, the thread accessed first happens-before the thread accessed after. Although the lock is determined by the thread in which it is held, unlike a semaphore, I think this is very similar to what the semaphore mechanism in an operating system does.

Happens-before algorithm has the following characteristics:

1. It is difficult to implement effectively, because this algorithm needs to access each Shared memory.
2. It relies too much on the scheduler. The shortcoming is fatal. When the instruction sequence is written, the bottom layer may optimize and reorder them, so the former order is out of order, and the correctness is difficult to guarantee.

2 The Lockset Algorithm

2.1 General Idea of the Algorithm

To put it simply, the lockset algorithm expects a Shared resource to always be protected by one or more locks. Two versions of lockset algorithm were introduced, the simplest version and the improved version respectively.

© The Editor(s) (if applicable) and The Author(s), under exclusive license
to Springer Nature Singapore Pte Ltd. 2021
M. Atiquzzaman et al. (Eds.): BDCPS 2020, AISC 1303, pp. 597–601, 2021.
https://doi.org/10.1007/978-981-33-4572-0_86

2.2 The Simplest Version

In the simplest version, a Shared variable is initialized, and then the Eraser maintains a collection of all possible locks held by it. Every time the Shared variable is accessed, the lockset updates the possible lock held by the Shared variable to the intersection of the possible lock held by the current thread [2]. If a lock always protects the Shared variable, the lock will always exist in the lockset every time an intersection operation is performed, and if the lockset is empty, the Shared variable is not protected by any lock. However, it is obvious that the simplest Lockset algorithm is too strict. Under the rules of this algorithm, there will be three kinds of false reported data races caused by non-compliance with the requirements of the algorithm:

1. When Shared variables are initialized without locks, then lockset algorithm is used at the beginning, it intersects all possible locks with the lockset of the thread to obtain an empty set [3], and then an error will be reported;
2. The Shared data is only written at the beginning, while the following are all read operations. However, according to the above, read operations can be accessed without locks, so the intersection operation will also be an empty set, resulting in an error reporting;
3. Read-write lock allows multiple reads but only one write operation.

2.3 Improved Version

Because the simplest version of lockset algorithm is too strict, an improved algorithm is given in the literature [4]. That is, if only one thread accesses the Shared data or if multiple threads perform read-only operations on the Shared data, then there will be no data races in these cases, so a data race should only be reported after an initialized variable has been written by multiple threads. In this way, the improved algorithm simplifies the original algorithm and can avoid some false reports caused by too strict requirements.

2.3.1 Problems with the Improved Version

If one thread allocates and initializes a Shared variable without any lock, and lets a second thread access the Shared variable immediately after initialization, it causes an error. However, the Eraser will not be able to detect this error unless the second thread accesses it before the Shared variable is initialized, so the algorithm has some flaws. This is also why the Lockset algorithm cannot detect all race conditions.

2.3.2 Further Improvements to the Improved Version

To suit the style of many programs that use single-write locks, multiple-read locks, and simple locks, one last refinement to the lockset was given. Each time a Shared variable is written, Lock must protect it with a write mode [2]. Each time a Shared variable is read, lock protects it by reading or writing. Because a lock held by a write operation does not prevent data race between the write operation and some other read operations, when a write operation occurs, a lock held in pure read operation mode is removed from the candidate lockset. See Fig. 1 below.

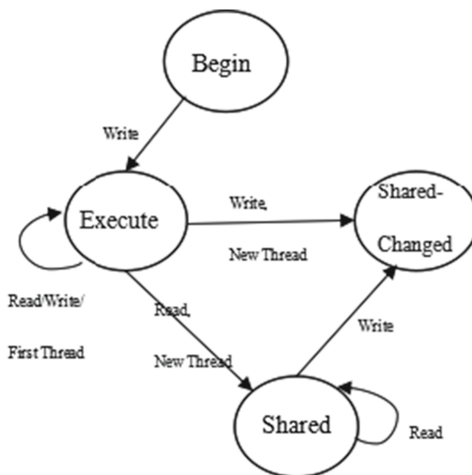

Fig. 1. Further improvement of lockset algorithm

3 Concrete Implementation of "Eraser"

The algorithm mainly needs to look for candidate locksets, because it turns out that only a few different locksets appear in this algorithm's execution, so the algorithm only opens up a small memory space to hold the lockset. [5, 6] To ensure each lockset is unique, a hash table of lock vectors was maintained, searching the hash table each time before creating a new lock set. In this way, even if the cache fails, the comparison can be made with two simple sort vectors. Then, how to find the lock direction scale? It is mentioned in the literature that each 32-bit word in the data segment and heap has a corresponding Shadow word. The 30 bits in the Shadow word are used to represent the index of the lock set, and the remaining 2 bits are used to represent the state. In other word, we just need to find the shadow word to find the locking scale. And how to find this shadow word? All standard memory allocation cases mentioned in literature are for initialization program distribution of each word and a shadow, and when a thread access a memory location, it through this location address and add a fixed displacement will find the shadow word, the operation process and computer composition principle of indexed addressing some similar algorithm.

What happens if there are false reports? There are three main types of false positives:

1. Memory reuse. When a thread shares a resource with another thread and no longer uses it, the other thread modifies it and reports an error. Just like the private allocator in many programs mentioned in this paper, threads have their own area and interact with main memory independently in JAVA;
2. Private locks. Because it does not report lock information to the algorithm at run time, the algorithm does not know the specific condition of the lock.
3. Healthy competition. Some data races are deliberate, but this race does not affect the correctness of the program.

This article introduces program annotations for these three common types of false reports. With these annotations, we can reduce the processing time for these false reports and discover the real data races earlier.

4 Experience with Eraser

According to the paper, the experience of running the Eraser in a large project such as a search engine or a student's assignment shows that the Eraser can find real data race to a great extent and improve the problems caused by previous algorithm. In additional experience, multiple lock protection and deadlocks are mentioned in the literature. Multi-lock protection means that multiple locks are required to protect write Shared variables, rather than one lock protects a Shared variable. Under this requirement, each thread that writes a variable must hold all the protection locks, while the process that reads a variable must hold one. The purpose of using multi-lock protection is not to increase concurrency, but to avoid deadlocks for programs that contain upregulation. Avoiding deadlock is another common topic, and the way we learn to avoid deadlock in operating systems is "banker algorithms", which look for possible sequences of security permits. One way to avoid this mentioned in the literature is to select a partial order among all locks and, when it holds more than one lock, obtain them in ascending order, which I think is similar to the banker's algorithm.

5 Summarize and My View

Data race will lead to our program data chaos, thread uneasy congruent problems, while the Eraser goal is to detect the real multithreaded dynamic data error of competition, it uses the lockset algorithm to achieve the purpose of check the data competition, the lockset is different from the previous "happens-before" algorithm, it can be too affected by scheduling machine, allowing the underlying rearrangement and does not affect the final run results. [2] Without such tool, it would be difficult for us to do such task just using the previous algorithm, because it would take a lot time to access every memory space, and we might detect some false alarms. If my program has a high probability of multithreading, or if sharing many variables is important, I would consider using The Eraser to detect data competition issues. I thinks that the dynamic data race detection should be multi-threaded specification test standard procedure in the work, with the scope of application of expansion of multithreading, data unreliability is increased in competition, unless there are better ways to eliminate them, then the lockset algorithm can be well performed in the data races problems, enough for the users to solve the problem of data races as soon as possible.

References

1. Savage, S., et al.: Eraser: a dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst. (TOCS) **15**(4), 391–411 (1997)

2. Yu, M., Lee, J.-S., Bae, D.-H.: AdaptiveLock: efficient hybrid data race detection based on real-world locking patterns. Int. J. Parallel Prog. **47**(5–6), 805–837 (2019)
3. Kusiak, A.: Fundamentals of smart manufacturing: a multi-thread perspective. Annu. Rev. Control **47**, 214–220 (2019)
4. Bonizzoni, P., Della Vedova, G., Pirola, Y., Previtali, M., Rizzi, R.: Multithread multistring Burrows-Wheeler transform and longest common prefix array. J. Comput. Biol. **26**(9), 948–961 (2019)
5. Zhang, T., Jung, C., Lee, D.: ProRace: practical data race detection for production use. ACM SIGPLAN Not. **52**(4), 149–162 (2017)
6. Yu, M., Park, S.M., Chun, I., Bae, D.H.: Experimental performance comparison of dynamic data race detection techniques. ETRI J. **39**(1), 124–134 (2017)