AMP Project 2:

Bounded-Timestamp Register Locks

Here the references:

* Lamport: [1]
* Taubenfeld: [2]
* Jayanti: [3]
* Aravind: [4]
* Szymanski: [5]
* Syncrobench (Gramoli): [6]
* Herlihy : [7]

I use Mendeley as my Reference/Bibliography Manager and the Word Plugin to insert Citations. I added all the papers so you can just copy paste the citation (if you do not use Mendeley yourself). Bibliography is at the end

Contents

[1 Task Description 3](#_Toc44265594)

[2 Program structure 4](#_Toc44265595)

[2.1 Modules 4](#_Toc44265596)

[2.2 Lock class structure 4](#_Toc44265597)

[2.3 Test structure 4](#_Toc44265598)

[3 Lock Implementations 5](#_Toc44265599)

[3.1 Reference Lock 5](#_Toc44265600)

[3.2 Lamport’s Bakery 5](#_Toc44265601)

[3.3 Taubenfeld (Black/White-Bakery) 6](#_Toc44265602)

[3.4 Jayanti 7](#_Toc44265603)

[3.5 Aravind 8](#_Toc44265604)

[4 Results 10](#_Toc44265605)

[5 References 11](#_Toc44265606)

[6 Other Stuff (annex?) 12](#_Toc44265607)

[7 Tests 12](#_Toc44265608)

[8 record\_event\_log 12](#_Toc44265609)

[8.1 Events 13](#_Toc44265610)

[8.1.1 Inaccuracies 13](#_Toc44265611)

[9 Properties 14](#_Toc44265612)

[9.1 First-Come-First-Served (FCFS) 14](#_Toc44265613)

[9.2 Least Recently Used (LRU) 16](#_Toc44265614)

[10 Makefile 16](#_Toc44265615)

# Task Description

Implement Taubenfeld, Lamport, and two out of Szymanski, Jayanti, Aravind and compare to a reference Lock (native C11 locks, simple test-and-set lock, simple test-and-test-and-set lock).

Challenge: Memory behavior. Ensure that memory (register) updates become visible in required order!

# Program structure

## Modules

(list functions?)

## Lock class structure

Two abstract classes were created to to define the common interfaces between the implementations of the various lock types.

class Lock

this pure abstract class is the Lock interface form the Herlihy book.

class DW\_Lock

this pure abstract class is an interface for "doorway locks". These are locks where the lock() function can be written like this

public: void lock(){  
 doorway();  
 wait();  
}

This is used so that fairness properties (e.g. FCFS) can be tested from the outside (i.e. without putting logging functions into the lock class).

## Test structure

# Lock Implementations

## Reference Lock

class Reference\_Lock

this uses these statements

* create an instance of the lock with this statement  
  std::atomic\_flag lock\_stream = ATOMIC\_FLAG\_INIT;
* execute lock() (i.e. doorway and waiting section) with this statement  
  while ( lock\_stream.test\_and\_set() ) {}
* execute unlock() with this statement  
  lock\_stream.clear();

## Lamport’s Bakery

class Lamport\_Lecture

This is the naive straight forward implementation of Lamport Bakery lock according to the lecture notes. It is very bad.

draw\_ticket

wait

class Lamport\_Lecture\_fix

This sub-class of Lamport\_Lecture uses an atomic register latest\_ticket to keep track of the ticket that was last issued. This is done by overriding the draw\_ticket() function like this:

private: virtual int draw\_ticket() override {  
 return ++latest\_ticket;  
}

Like this mutual exclusion holds. It is questionable however, if we can still speak of a wait-free doorway section now. Consider this case: Thread tries to draw a ticket, an atomic read and an atomic write of latest\_ticket are required. If latest\_ticket it is accessed by thread at the time, and thread is stalled, this could mean that there is no maximum number of required can be given in which thread completes the doorway.

class Lamport\_Original

this is the naive straight forward implementation of the Lamport Bakery algorithm from 1974. The variable names and token values are from the paper Jayati 2004.

It is very bad

implementation of wait

## Taubenfeld (Black/White-Bakery)

In 2004, Gadi Taubenfeld presented his modification[2] of Lamport’s bakery algorithm that, first and foremost, promises to bound the size of the needed registers. In total three different versions of his algorithm are presented, where the latter two build upon the first, basic version.

All three versions can be shown to satisfy *mutual exclusion*, *deadlock freedom* and *FIFO (FCFS)*, which also implies *starvation freedom*. The other two versions add the properties *adaptive* and *local-spinning*, although we did not implement them due to time constraints. Though we also want to note, that they are significantly more complex to achieve these added properties, whereby they lose the simple elegance of Lamport’s original algorithm.

The basic idea behind Taubenfeld’s Black/White-Bakery (sometimes also called Color-Bakery) is to add a color to the tickets, either black or white, and then working through a block of one color before considering tickets of the other color. Tickets are colored based on the value of a shared Color bit. Tickets with a color different to the shared Color bit are considered to have higher priority than those that share the shared bits color, regardless of the tickets’ numbers. In each color block, priority is given to the ticket with the lower number, and failing that, the threads’ IDs are considered. A thread that has successfully acquired the lock and executed the CS will afterwards set the shared Color bit to the color different to its ticket and then reset its ticket.

In this way the finite number of registers needed can be each bound to the sizes *log(2n +2) bit* and *1 bit* respectively. In total three registers are needed, one shared register of size 1 bit for the color

class Taubenfeld

The first, basic variant presented by Taubenfeld in his paper as shown in Figure 1.

It is not mentioned in the paper what type of registers are required for the algorithm to work, so we implemented this version using just *volatile int* C-arrays. We then noticed during our tests that this version of the Taubenfeld lock very rarely led to mutual exclusion failures. We then tried to fix this issue by using atomic registers instead, which led to the next version.

class Taubenfeld\_atomic

This implementation of the Black-White-Bakery uses *atomic* registers for the *color*, *choosing* and *ticket* arrays. We never encountered mutual exclusion failures with this version.

class Taubenfeld\_adaptive

This was our attempt at an implementation of the second variant presented in the paper, the adaptive Black-White-Bakery, which uses a so-called active Set [2] (page 64). Implementing this active set in C++ proved too difficult and time consuming for us. One obstacle was the fact that the STL-*Set* in C++11 is not thread-safe. Although it “only” rarely produces mutual exclusion failures, that also means it does NOT work.

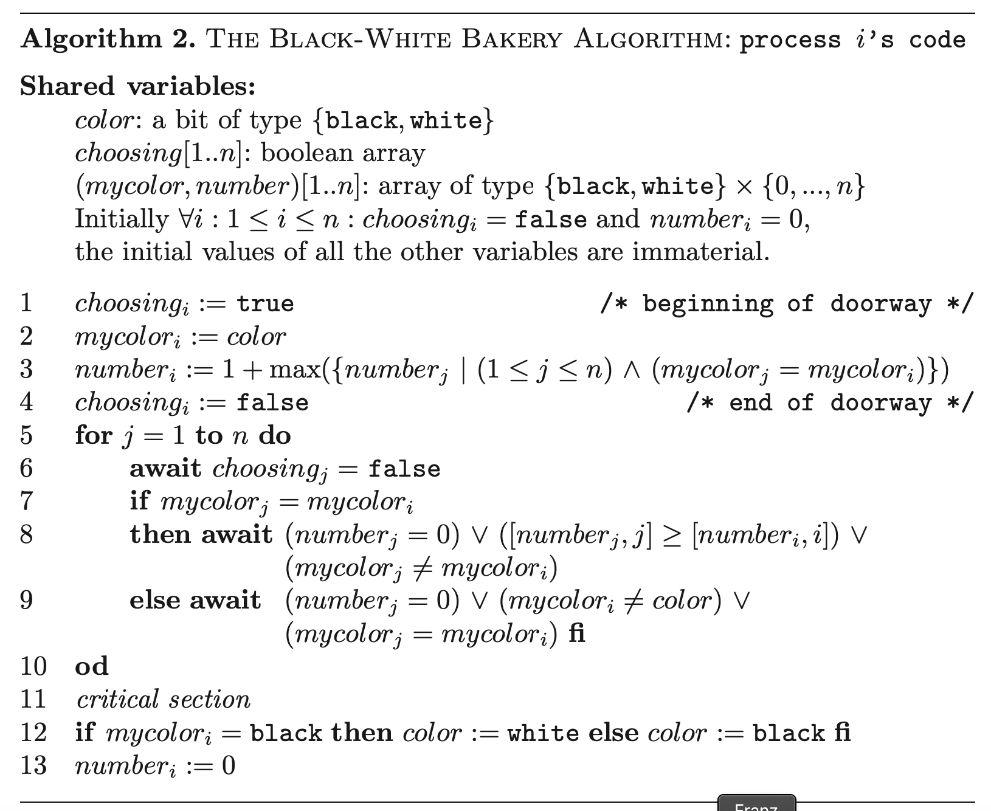
We therefore decided to focus on the basic version of Taubenfeld’s lock algorithm. This class is mentioned only for completeness sake.

Abbildung 1 Black-White Bakery Algorithm[2]

## Jayanti

In the paper by Jayanti et al. [3] another variation to to Lamport's Bakery Lock [1] is described. The advantage of the described lock over Lamport's algorithm is, that is able to keep the token (sometimes called ticket) bounded: for a lock of size , only token values are required (i.e. . The paper describes the algorithm in two steps. First an algorithm with clustered but still unbounded tokens is presented. Then this algorithm is enhanced to make the tokens bounded in value.

Algorithm with Clustered Tokens

It is shown that with Lamport's algorithm in special cases, the differences between tokens can grow without bounds. The first algorithm in the paper fixes that, by introducing an additional atomic register (variable in Figure 2), that holds the token value of the last thread, that acquired the lock. The value stored in is to be considered, when issuing a new token, which prevents token values to reset to their initial value. This prevents the mentioned special case, where the differences between tokens in Lamport's algorithm grow without bounds. Figure 2 shows the algorithm of Jayanti et al.

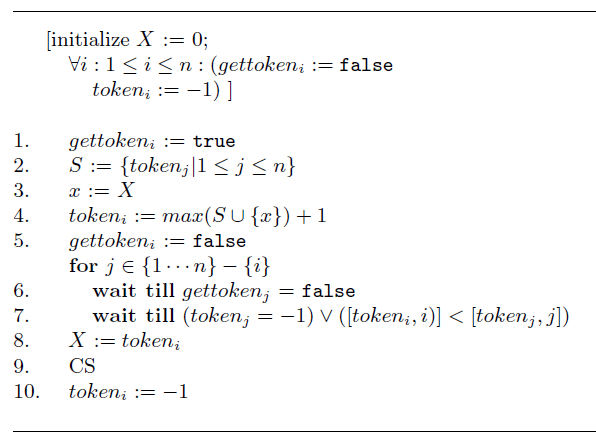


Figure 2 The algorithm by Jayanti et al. [3]

Algorithm with Bounded Tokens

In a second step this algorithm is enhanced such that the now clustered tokens will also be bounded in value. This is being accomplished by changing the way, a new token is issued as well as changing the order relation by which the priority between two threads is decided.

first two new binary arithmetic operations are defined:

let denote

let denote

With these operations a new way to calculate the maximum of a set of tokens is defined (here is the value of read in line 3 of Figure 2):

let

then define

Furthermore, the order relation from line 7 is replaced by the relation , that is defined as follows:

With these replacements, the tokens are bounded.

Properties

Implementation

## Aravind

The paper by Aravind [4] presents yet another variation to Lamport's Bakery Lock. This locks outstanding feature is, that instead of fulfilling the FCFS property the least-recently-used property is fulfilled. This fairness property defines different rules considering, which thread should be allowed to acquire the lock, in case an acquisition is contended: LRU demands that amongst the contenders, the thread, which has not had the lock for the longest time, should be allowed to acquire it. This is accomplished by handing out timestamps (tokens, tickets, ...) to threads when they leave the critical section and are about to unlock, rather than handing them out in the doorway section.

LRU Algorithm

As it was mentioned, the timestamps are handed out, when threads leave the critical section. This has the advantage, that no two threads can ever end up with the same timestamp as long as mutual exclusion holds. Therefore the comparison of threads in terms of which one has the higher priority to acquire the lock is simplified, since it is reduced to just comparing timestamps without the need to consider the possibility of equality. By comparing the which thread was first to finish the critical section rather than which thread was first to go through the doorway, the lock fulfills the LRU property instead of the FCFS property. Figure 3 shows the Algorithm.

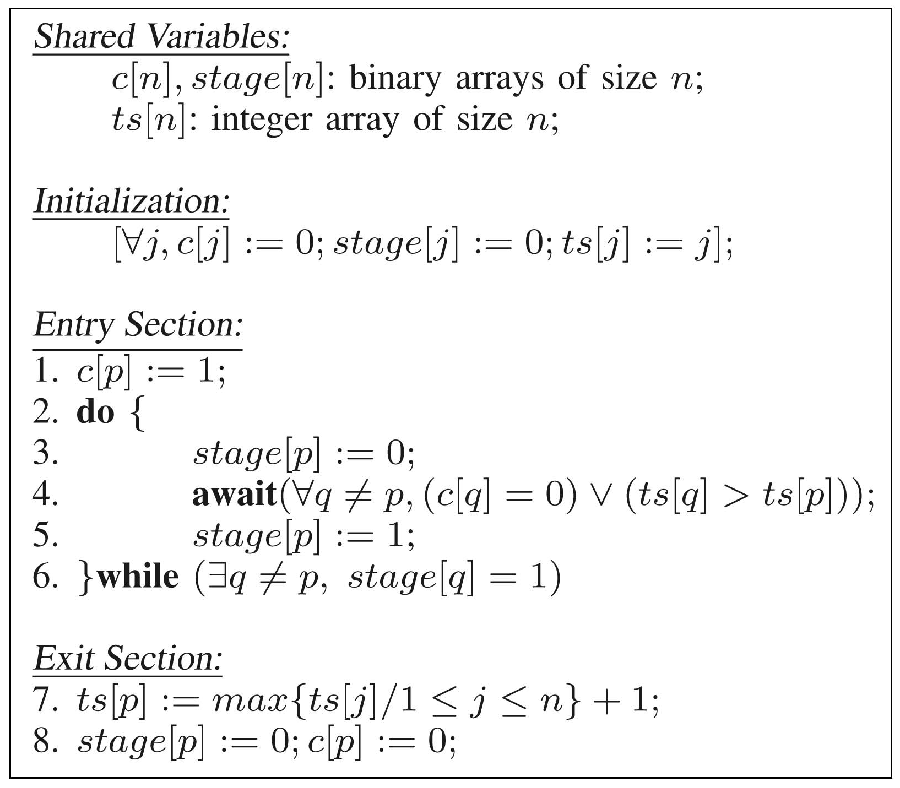


Figure 3 The algorithm by Aravind [4]

The algorithm uses three arrays:

* c[j]: Boolean – indicates whether process is trying to acquire the lock
* stage[j]: Boolean – indicates the stage that process is in (see below)
* ts[j]: integer – indicates when process has last acquired the lock

when process tries to acquire the lock it will:

* at stage 0
  + set c[j] = 1
  + check all c[k] and ts[k] to check that no one else should rather get the lock.   
    If ok, set stage[j] = 1; if not, wait and check again,
* at stage 1
  + check that no one else is at stage 1: if ok, proceed to CS, if not, set stage[j] = 0 and start over with stage 0

When a process unlocks, it will:

* Draw a new timestep and write it to ts[j]
* Set stage[j] = 0, c[j] = 0

Proceeding like this the LRU property is not strictly fulfilled as Aravind himself admits in his paper. It is possible, that a thread with higher priority starts to try and acquire the lock, while a lower priority thread is at line 5 of Figure 3. The fact that thread is now a contender will not be considered by thread who will (in many cases) be allowed to go ahead and acquire the lock.

This defect can be reduced by storing a list of contenders when executing line 4 in a thread local boolean array and checking, if the list of contenders has changed before finally acquiring the lock. This has been done in the class Aravind\_fix. This will however still not fully remove the defect, because the checking of the list is not done atomically, which means a higher order thread can become a contender, while the list is checked.

LRU Algorithm with bounded tokens

The paper gives an example of how the presented algorithm can ensure that the timestamps stay bounded. Once a timestamp would exceed the maximum valid value, all timestamps are reset to their initial values. This of course violates the LRU property, but it is stated in the paper, that while it is easily possible to reset the timestamps, such that the LRU property remains fully intact, but it was deliberately chosen not to do so, because priority was given to the simplicity of the algorithm.

This way of bounding the timestamps has not been implemented in this project, because it does not seem to have been the focus of Aravind's work.

# Results

Our main performance criterium is throughput, which we measure by the amount of successful lock acquisitions per second. A higher amount of workload in the CS will then lead a lower amount of acquisitions per second, naturally. A lock algorithm that is more efficient will produce a higher amount of acquisitions per second given the same workload (in and outside the CS).

We tested three different scenarios:

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Workload | Workload\_cs | randomness |
| Scenario 1 | 0 | 0 | 0 |
| Scenario 2 | 100 | 1000 | 0.4 |
| Scenario 3 | 1000 | 100 | 0.4 |

Workload… amount of work to be done by each thread outside of the CS  
Workload\_cs… amount of work to be done inside the CS  
Randomness… randomization factor that scales the actual work to be done in and outside the CS

Both workload and workload\_cs specify the upper limit for a for-loop. A workload of 100 with randomness of 0.4 then means, that the for-loop is executed at least 60 times and at most 100 times. Randomness gives the percentage (of the number of iterations of the respective for-loop), that is randomized each time either the outside work section or CS is called. The purpose of this setting is to make the sequence of lock attempts of the threads more random.

We tested the above scenarios for the following numbers of threads [2, 3, 4, 8, 16, 32, 64]. However, even though the Reference Lock otherwise performed well, when we used 32 and 64 threads in scenarios 1 and 2, the benchmarks were interrupted due to the time constraint of 5mins. There is probably an issue in our implementation that we could not fix. Our implementation of the reference lock worked just fine for scenario 3. Where we could not gather data due to the issue described above, they are simply omitted.

## Scenario 1 (workload=0)

Scenario 0 represents the edge case scenario where threads constantly attempt to acquire the lock and are only really waiting for their turn to do work on the CS. This scenario is of course not realistic but serves as an edge case or limit for poorly parallelized code.

Abbildung 2 and Abbildung 3 show the Throughput (as defined above) with and without the logging functionality of our test function. We notice that the logging has a significant impact on performance, as was expected, but it does not impact the different locks differently and does not make the results incomparable between the different implementations. Error bars are plotted in every plot, but are often so small, that they are not visible. Lamport is the fastest of the ticket locks, whereas the Reference Lock (test\_and\_set) outperforms it for lower amounts of threads.

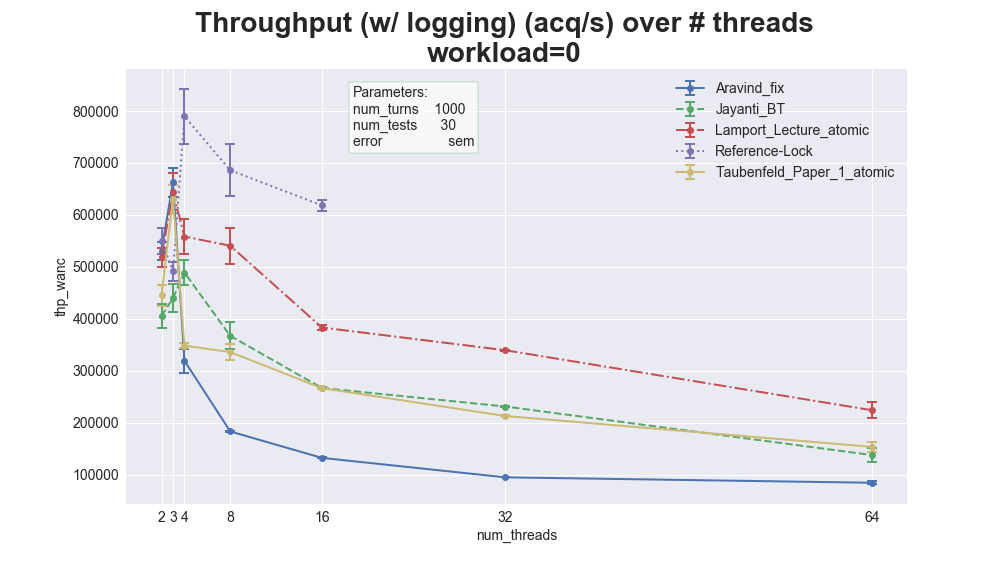


Abbildung 2 Scenario 1 : throughput with logging

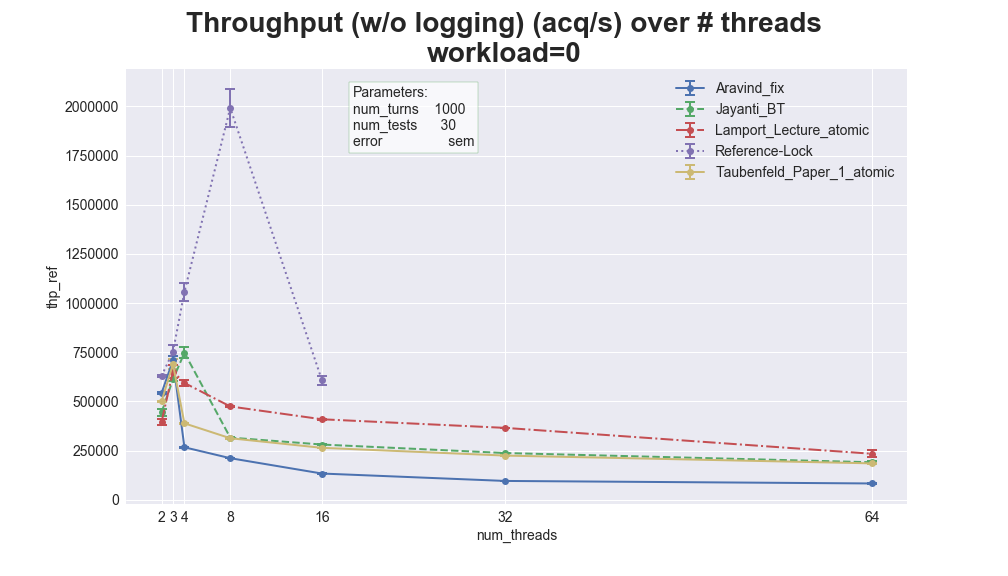


Abbildung 3 Scenario 1 : throughput without logging

## Scenario 2 (workload=100)

This test scenario is similar to the first. It represents a scenario where performance is mainly restricted by a CS and the workload outside is comparatively small (workload outside : workload CS = 0.1), but is a more realistic scenario than 1 as we take vary (randomize) the workload in and outside the CS.

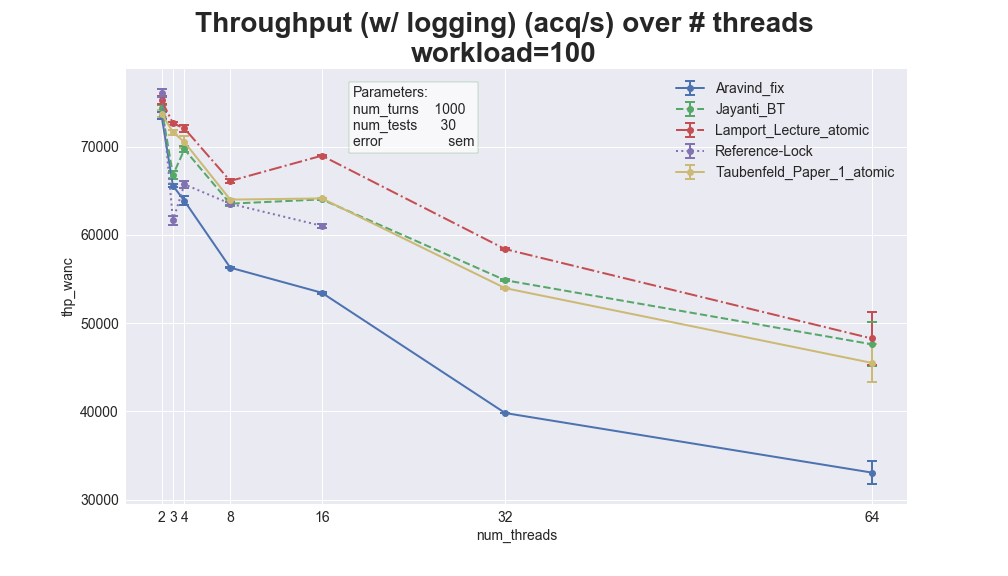


Abbildung 5 Scenario 2: throughput with logging

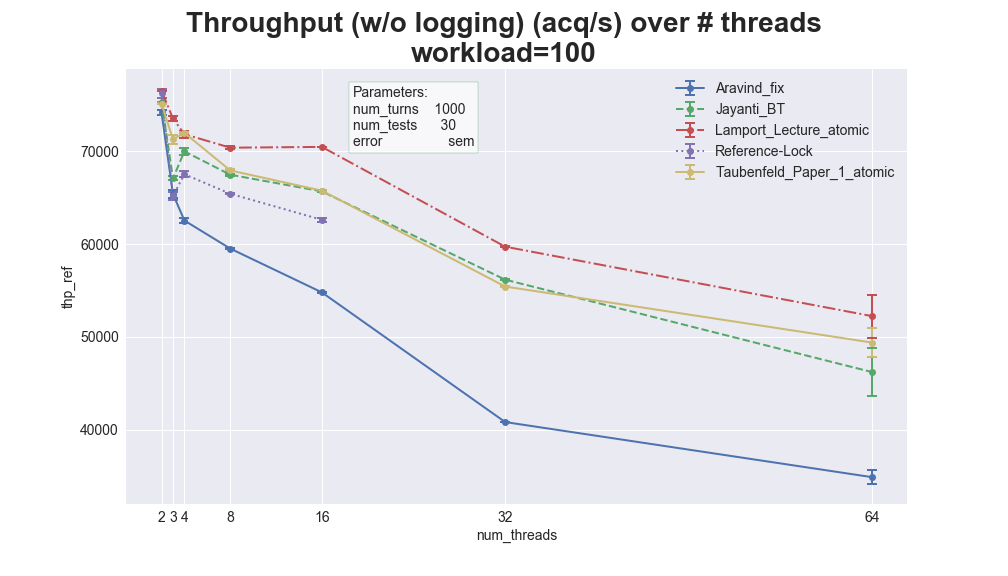


Abbildung 6 Scenario 2: throughput without logging

Abbildung 5 and Abbildung 6 shows the results for the throughput tests for this scenario. This time Lamport’s Bakery performs the best across the board and the Reference Lock is even slower than both Jayanti\_BT and the Taubenfeld Black-White-Bakery lock. Aravind again performs the worst out of all.

## Scenario 3 (workload=1000)

Here, we test the locks for scenarios where the ratio of workload/workload\_CS=10, so where a program is parallelized to a higher degree than before.

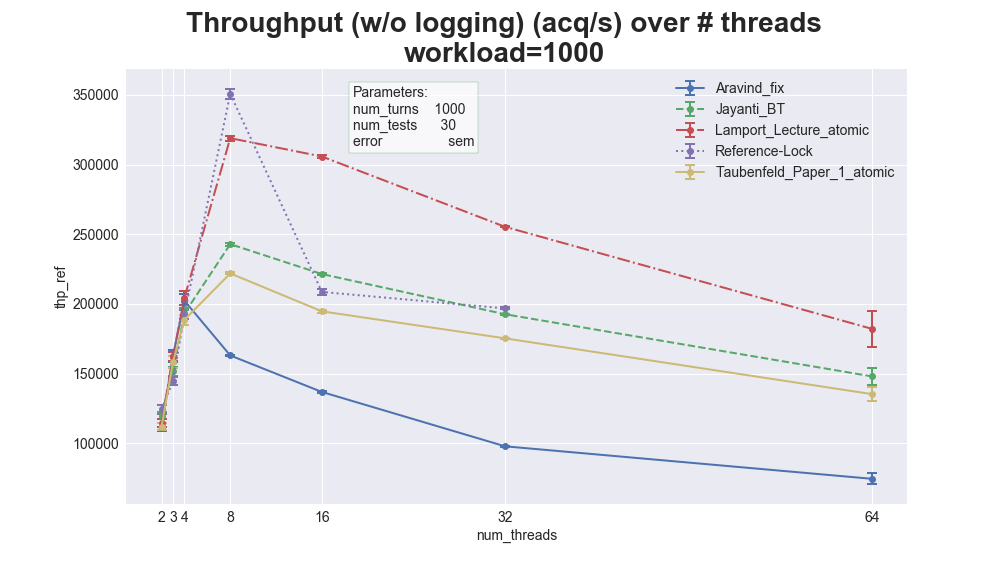


Abbildung 8 Scenario 3 : throughput with logging

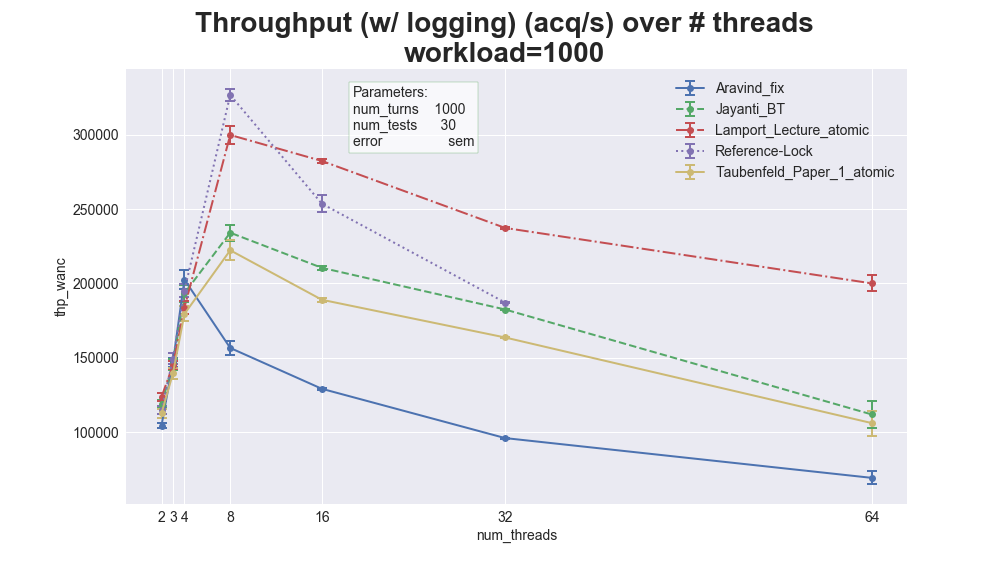


Abbildung 9 Scenario 3 : throughput without logging

In Abbildung 8 and Abbildung 9, we can see the throughput for this scenario. Again, Lamport’s Bakery performs the best across the board, while the Reference lock (test\_and\_set) performs the best for low thread numbers but performs similar to the Jayanti\_BT and Taubenfeld locks. Aravind is again the worst out of all.

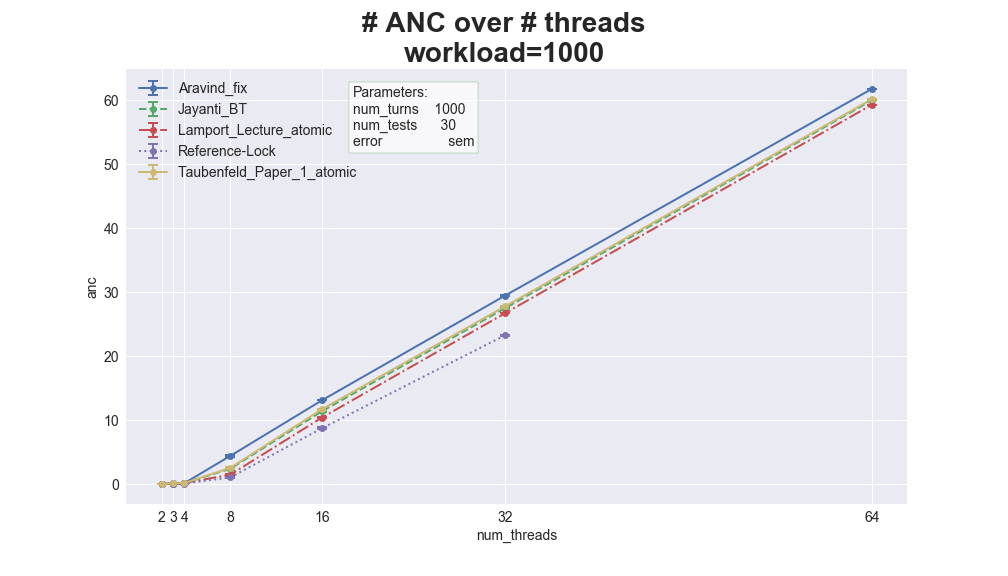


Abbildung 10 : ANC (Average Number of Contenders)

Abbildung 10 shows the average number of contenders for acquisition of the lock. For the ticket locks, the appear in the same ordering as in the throughput graphs. A lower ANC would indicate a more efficient lock algorithm as that would mean that less threads are . This holds true at least for the ticket locks. However, the Reference lock (test\_and\_set) does not adhere to this.

# Other Stuff (annex?)

# Tests

do\_some\_work()

WIP

test\_mutex()

write about random workload

this will test mutual exclusion of a passed DW\_Lock object with given parameters (more detail)

The idea is to log the events of threads entering the critical section and threads leaving the CS. Mutual exclusion holds true, if every entering event is followed by a leaving event. I.e. no entering event is immediately followed by another entering event.

test\_fcfs()

write about random workload

this will test first-come-first-served of a passed DW\_Lock object with given parameters. (more detail)

At the moment this is done by making the doorway section mutually exclusive. But this will have to change.

The idea is to write an array, that keeps track of the sequence in which threads complete the doorway and another array that keeps track of threads acquiring the lock. FCFS holds true if these arrays are identical.

main()

write about initialization

An instance of one of the lock classes is created.

The lock object is then passed to one or more testing functions.

# record\_event\_log

this runs the test and records the relevant events in an array

one global atomic counter is used

Threads are sent through the lock and try to acquire it for a fixed number of times. Thread local arrays are recorded

after the test finished, threads assemble the global event log from their thread local event logs.

## Events

Four types of events are being recorded

1. being about to begin the doorway

2. having just finished the doorway

3. having just acquired the lock

4. being about to unlock

### Inaccuracies

There is a certain inaccuracy, when recording these events. In between a thread, acquiring the lock and noting that it has just acquired the lock, other threads can record events. In the this section the consequences of these inaccuracies are being discussed.

The method of this discussion:

there is a condition based on the sequence of events.

There is a noted sequence of events and a real sequence of events.

The inaccuracy causes a difference between these sequences.

Here it is discussed, whether the condition becomes stronger or weaker. If it becomes stronger this can cause false positives when checking for violations of the property. If the condition becomes weaker it is possible that violations of the property are not found.

(1) begin

A thread notes, that it is about to begin its doorway. Other threads can cause events between a thread noting, that it will now begin its doorway, and it actually beginning the doorway. So has actually happened later than what was noted.

In the real sequence can be later than in the recorded sequence. ("recorded begins appear earlier than they are").

Relevant in fcfs: condition

In the real sequence is will be more often satisfied than in the recorded sequence. Therefore the inaccuracy makes the condition weaker because is hard to satisfy if is easy to satisfy.

(2) finish

A thread notes, that has just finished its doorway. Other threads can cause events between a thread finishing its doorway and logging it. So has actually happened earlier than what was logged. ("logged finishes appear later than they are")

Relevant in fcfs: condition

Once again: In the real sequence is will be more often satisfied than in the recorded sequence. Therefore the inaccuracy makes the condition weaker.

(3) acquire

A thread notes, that has just acquired the lock. Other threads can cause events between a thread acquiring the lock and logging the acquisition. Assuming mutual exclusion for the lock, this cannot be acquisition or unlock events. So has actually happened earlier than what was logged. ("logged acquisitions appear later than they are")

* Relevant for mutual exclusion: condition

Acquisitions appearing later than they are, can cause satisfaction of the condition where it was actually violated.

* Relevant for fcfs: condition

Acquisitions appearing later than they are can cause both wrongful satisfaction of the condition as well as false positives when checking the log.

* Relevant for LRU: condition

If mutual exclusion is assumed, the order of the acquisitions cannot be changed by this inaccuracy. Therefore only is affected, which is harder to satisfy due to the inaccuracy. Therefore the LRU condition is easier to satisfy. Therefore there might be violations of the LRU property that are not registered.

(4) unlock

A thread notes, that it is about to unlock. Other threads can cause events between a thread logging the unlock and the actual unlock. Assuming mutual exclusion for the lock, this cannot be acquisition or unlock events. So has actually happened later than what was logged. ("logged unlocks appear earlier than they are")

* Relevant for mutual exclusion: condition

Unlock events appearing earlier than they are, can cause satisfaction of the condition when it was actually violated.

# Properties

## First-Come-First-Served (FCFS)

interpretation high numbers of violations

lets say we got 4 threads each acquiring the lock 3 million times

lets say the workload in the cs is so high, that on average there are 3 contenders at each acquisition

for each such acquisition there is a

* 33% chance, that 2 violations occur
* 33% chance, that 1 violation occurs
* 33% chance, that 0 violations occur

so on average 1 violation will occur per acquisition.

if we have on average 4 contenders...

* 25% chance, that 3 violations occur
* 25% chance, that 2 violation occurs
* 25% chance, that 1 violation occur
* 25% chance, that 0 violations occur

so on average: 1.5 violation. so we should see a convergence to 18 million violations if we increase the workload in the cs

## Least Recently Used (LRU)

# Makefile

Compiler flags

they do weird things

O1 and O3 will break mutex for Lamport\_Original

O1 and O3 will improve mutex for Lamport\_Lecture (lol)

O2 seems to do nothing (test some more?)

# References

[1] L. Lamport, “A New Solution of Dijkstra’s Concurrent Programming Problem,” *Commun. ACM*, vol. 17, no. 8, pp. 453–455, 1974.

[2] G. Taubenfeld, “The black-white bakery algorithm and related bounded-space, adaptive, local-spinning and FIFO algorithms,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 3274, no. 4, pp. 56–70, 2004.

[3] P. Jayanti, K. Tan, G. Friedland, and A. Katz, “Bounding Lamport ’ s Bakery Algorithm,” pp. 261–270, 2001.

[4] A. A. Aravind, “Yet another simple solution for the concurrent programming control problem,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 6, pp. 1056–1063, 2011.

[5] B. K. Szymanski, “A simple solution to Lamport’s concurrent programming problem with linear wait,” *Proc. Int. Conf. Supercomput.*, vol. Part F1301, pp. 621–626, 1988.

[6] V. Gramoli, “More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms,” *Proc. ACM SIGPLAN Symp. Princ. Pract. Parallel Program. PPOPP*, vol. 2015-Janua, pp. 1–10, 2015.

[7] M. Herlihy, *The art of multiprocessor programming*. 2006.