2-thread software solutions for the mutual exclusion problem

Andreia Correia
Concurrency Freaks
andreiacraveiroramalhete@gmail.com

Pedro Ramalhete
Cisco Systems
pramalhe@gmail.com

Abstract

The first software solution to the mutual exclusion problem with two competing threads was discovered more than 50 years ago by Dekker. In the meantime, a few other 2-thread algorithms have been proposed, with the most famous being the one by Peterson. Apart from the theoretical interest in this subject, recent work has shown that tournament-based software solutions are amongst the best in solving the N-thread mutual exclusion problem. Tournament solutions themselves use one of the 2-thread algorithms, meaning that having efficient and starvation-free 2-thread algorithms can improve the N-thread solutions as well.

In this paper we present ten new starvation-free solutions to the 2-thread mutual exclusion problem, all of which can achieve a minimum of writes to shared memory to enter and leave the critical section. We compare the throughput of these ten algorithms against the previously known solutions using a microbenchmark under two different architectures: AMD x86-Opteron, and ARMv8.

Categories and Subject Descriptors D.4.1 [*Operating Systems*]: Mutual Exclusion

General Terms Algorithms, Design, Performance

Keywords mutual exclusion, software solution, locks

1. Introduction

Mutual exclusion is one of the first concurrency problems to be tackled [4], and today, it is still a subject of research [2]. In multiprocessor systems with shared memory, the techniques using the special test-and-set instruction or compare-and-exchange instruction — like CLH [3, 11], MCS [13], Ticket Lock [12] or Tidex Lock [18] — can be effective, but there are to this day CPUs that do not provide such instructions, and for those, we can only rely on algorithms that use simple store and load instructions.

The mutual exclusion problem is described as N threads/processes each executing a critical and a noncritical section, the restriction being, that from the moment a process enters its critical section and until it leaves it, no other process is allowed to execute their own critical section. A critical section of a process is a section of its program, for which we wish to ensure is never executed at the same time that any other process is executing its own critical section [16]. A solution to the critical section problem needs to satisfy two important properties:

1. *Mutual exclusion*: There will be at most one process executing the critical section at a time, or as formally stated by Lamport [10], for any pair of distinct processes i and j, no pair of operation executions $CS_i^{[k]}$ and $CS_j^{[k]}$ are concurrent.
2. *Deadlock Freedom*: The critical section will not become inaccessible to all processes. This means that if a number of processes attempts to execute their critical sections, then after a finite amount of time some process will be allowed to do so.

These two properties combined, guarantee that in a finite number of steps, *one and only one process* will have access to a critical section at a time. Later, the property of *starvation-freedom* was introduced that guarantees *all processes* will access the critical section in a finite number of steps. Other properties are stated by Dijkstra [5]:

- The solution must be symmetrical between the N computers; as a result, we are not allowed to introduce a static priority.
- Nothing may be assumed about the relative speeds of the N computers; we may not even assume their speeds to be constant in time.
- If any of the computers is stopped well outside its critical section, this is not allowed to lead to potential blocking of the others.

Dijkstra [4] was the first to formulate the mutual exclusion problem between only two processes, and it is attributed to Dekker [9] the first known solution to this problem. Later, Peterson [15] provided a simpler and more efficient solution for two processes, using 3 variables where there is a write race on the variable `turn`. His algorithm had a special approach because each thread would "give" the `turn` to the opposite thread. This would make the last thread to modify `turn` to be the last thread entering the critical section. Kessels's [8] followed on Peterson solution but the variable `turn` was replaced by two thread local variables, and the logical turn was computed based on both variables. Kessel's solution eliminates the write race on `turn`.

Tournament based algorithms, where each tournament opposes D processes (with D greater than one) and in particular 2 processes, are a N thread solution. The tournament is established following a logical tree where each node is a round that opposes D processes. The algorithm chosen for each round has to guarantee mutual exclusion between those D processes, and typically when D is two, the algorithm used can be Dekker's, Peterson's, Kessels's and several others. The tournament based approach has an advantage in cases where a process is delayed for some time, all other processes that are competing in a different branch of the tree will not get delayed by this process and, subsequently, these processes can overtake the delayed process an unbounded number of times. This is beneficial in scenarios with over-subscription, where the

number of processes is greater than the number of cores.
Peterson and Fischer [15] devised a tournament solution where the logical tree is determined using the bits of each process identification number. On the other hand, Taubenfeld [1] performs a direct tree walk where on each node a Peterson 2-Process algorithm is used to determine the process that will move on to the next level on the tournament. Both Peterson and Taubenfeld's solutions use a maximal binary tree [2]. Kessels also performs a direct tree walk but instead uses a minimal binary tree with Kessels 2-Process algorithm. Recently, Buhr [2] has used Kessels direct minimal tree walk with Peterson 2-Process algorithm for contention on each node, with the additional optimization using a precomputed tree that is accessed as a table lookup.

In this paper we will present ten new two-thread mutual exclusion solutions, all being starvation-free. Table 1 shows a comparison between the new and the previously known algorithms.

	Min Number of stores	Min Number of loads	Number of variables	Starvation Free
Dekker(A,B,C,Orig)	$1+2$	1	3	yes
Dekker(RW)	$1+2$	2	3	yes
Peterson 2-thread	$2+1$	1	3	yes
Peterson-Fischer	$2+1$	3	2	yes
Kessels	$2+1$	2	4	yes
Doran	$1+2$	1	3	no
Tsay	$2+2$	1	3	yes
X2Tv1	$1+1$	2	2	yes
X2Tv2	$1+1$	2	2	yes
X2Tv3	$1+1$	2	2	yes
X2Tv4	$1+1$	1	3	yes
X2Tv5	$1+1$	1	2	yes
X2Tv6	$1+1$	2	3	yes
X2Tv7	$1+1$	1	2	yes
X2Tv8	$1+1$	2	3	yes
X2Tv9	$1+1$	1	2	yes
X2Tv10	$1+1$	2	2	yes

Table 1. Comparison table between the different algorithms for 2-thread mutual exclusion. The first column shows the minimum number of sequentially consistent stores (or stores with release) needed to enter the critical section, plus the number needed to leave the critical section. The second column represents the minimum number of sequentially consistent loads (or loads with acquire) used on the algorithm. The third column shows the memory usage in number of words. As shown on the fourth column, all algorithms are starvation-free with the exception of Doran's solution.

2. Algorithms

We will now present ten different algorithms to solve the 2-thread mutual exclusion problem, which we named X2Tv1 to X2Tv10. The first five of these algorithms have a symmetrical code path, while the others have an asymmetric code, i.e. the code that the first thread executes is not the same as the code that the second thread executes.

Although having ten algorithms may seem like too much, they have several different properties and design goals. The first three algorithms, X2Tv1, X2Tv2, and X2Tv3, use only two shared variables (one per thread), using a *composed turn* technique similar to Kessels, and a *backoff strategy* encoded a `WAITING` state. X2Tv4 is based on Peterson's 2-thread algorithm with a modification that allows to enter the critical section doing a single store to a shared variable when running uncontended. On X2Tv5 we improve the idea of X2Tv4 by reducing the number of shared variables from 3 to 2 using a *composed turn* similar to Kessels. X2Tv6 uses a *backoff strategy* similar to X2Tv1-3, with asymmetric code path, and X2Tv7 improves on it by reducing the number of shared variables from 3 to 2 using a *composed turn* technique. X2Tv8 is another asymmetric algorithm that uses three shared variables and the *backoff strategy* in one of its threads. X2Tv9 is an algorithm that focus on giving *priority* to one of the threads, where the first thread will enter the critical section doing a single store (and one or more loads, depending on contention), and leave the critical section doing a single store. XTv10 is yet another asymmetric algorithm with two shared variables.

Table 1 shows a comparison of several properties of the previously known 2-thread algorithms and our 10 new algorithms. Unlike any of the previously known 2-thread starvation-free algorithms, all of our algorithms allow, for an uncontended scenario, at least one of the threads to enter the critical section doing a single store to a shared variable, and another store to leave it.

We will now describe in more detail each of the 10 algorithms, and their C++ implementation using a short-hand notation where, for brevity, we removed the prefix `std::memory_order_` on the atomic load and store operations, and therefore, to compile this code, it is necessary to prepend `std::memory_order_` to the memory order designations `seq_cst`, `acquire`, `release`, and `relaxed`.

2.1 X2Tv1 Algorithm

This algorithm was designed having in mind that each thread should only modify its own shared variable. Each shared variable contains the state of the thread and its turn.

The source code can be seen in Algorithm 1. The algorithm has two variables, `q[0]` and `q[1]`, where each variable consists of one bit encoding the `turn` and two bits encoding one of the three states: `UNLOCKED`(U), `LOCKED`(L), or `WAITING`(W). The macro `COMP_TURN(x,y)` is the XOR of the least significant bit of `x` and `y`, namely: `(x&0x1)^(y&0x1)`. In our implementation, we use the least significant bit to encode the `turn` (macro `TURNBIT(x)`) and the two following bits to encode the state (macro `STATE(x)`).

Algorithm 1 X2Tv1 Algorithm

```
1  int tbit = TURNBIT(q[id].load(relaxed));
2  q[id].store((L | tbit), seq_cst);
3  while (STATE(lqot = q[other].load(seq_cst)) == L) {
4      if (COMP_TURN(tbit, lqot) == other) {
5          q[id].store((W | tbit), seq_cst);
6          while (STATE(lqot = q[other].load(seq_cst)) != U &&
7                  COMP_TURN(tbit, lqot) == other) Pause();
8          q[id].store((L | tbit), seq_cst);
9      }
10 }
11 CriticalSection( id );
12 q[id].store((U | (IS_LW(lqot) ^ TURNBIT(lqot) ^ id)), release);
```

X2Tv1 resembles Kessel's algorithm where the logical turn is computed based on each thread's variable, and there are no write races. On the other hand, the algorithm requires three states instead of two, and the turn is only modified after leaving the critical section. Both Peterson's 2-thread algorithm [15] and Kessel always pass the turn to the other thread regardless of the other thread's state. The X2Tv1 will only pass the turn if the other thread's state is not UNLOCKED.

As previously stated, this algorithm has three states with the `WAITING` state functioning as a kind of *backoff* to let the other thread know that it has seen that it is not its turn. When one thread is in state `LOCKED` and it is the other thread's (logical) turn, then this thread will *backoff* by transitioning to `WAITING` state, as shown in lines 4 and 5 of Algorithm 1. The only time a thread can toggle its turn bit (and therefore, the logical turn) is after leaving its critical section, when transitioning from state `LOCKED` to state `UNLOCKED`, and it does so only when the other thread is in `LOCKED` or `WAITING` state, and only if toggling the bit will cause the turn to be passed

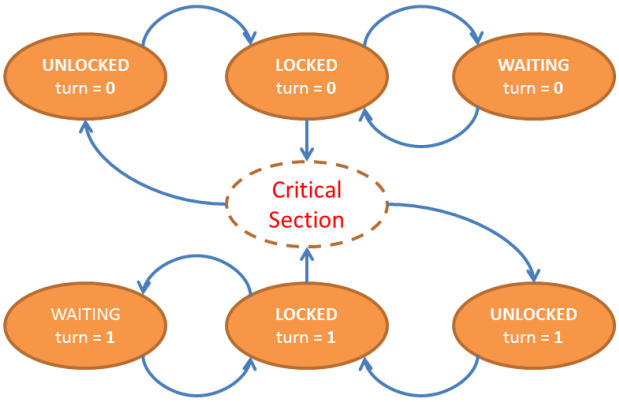

Figure 1. State machine for the X2Tv1 algorithm. The initial state is at `UNLOCKED` with `turn=0`.

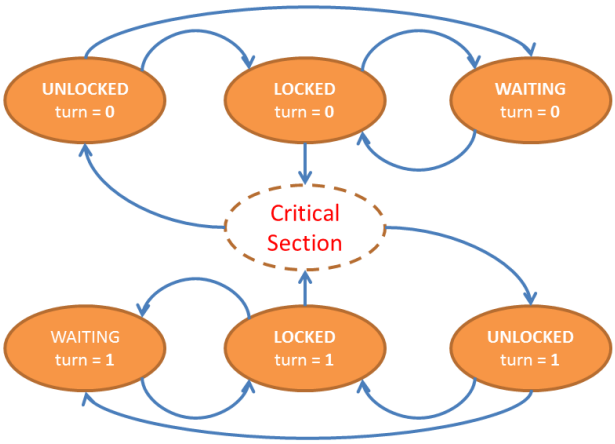

Figure 2. State machine for the X2Tv2 algorithm. The initial state is at `UNLOCKED` with `turn=0`. The transitions from `UNLOCKED` to `WAITING` will occur when the opposite thread is in `LOCKED` or `WAITING` and is the opposite thread's turn.

to the opposite thread, as shown in line 12 of Algorithm 1. The other thread will be waiting for either the current thread to go into `UNLOCKED` state, or for the turn to change to its own, as seen in lines 6 and 7 or Algorithm 1.

A thread will never enter the critical section unless it is in `LOCKED` state, as shown on the state machine in figure 1. This provides the invariant that when the current thread is in `LOCKED` and the other thread is in `UNLOCKED` or `WAITING` and the logical turn belongs to the current thread, the other thread will not be able to toggle its turn bit and therefore, the logical turn will not change.

When moving to state `UNLOCKED`, the current thread holding the lock checks if the other thread is waiting for the lock by calling the macro `IS_LW()` (i.e. is in `WAITING` or `LOCKED` state), and if so, it sets its own turn bit such that it will be the other thread's logical turn. The simplest way to write this logic is with two XOR operations, as shown in line 12 of Algorithm 1.

2.2 X2Tv2 Algorithm

This algorithm is a variant of `X2Tv1` where instead of going directly from `UNLOCKED` state to `LOCKED` state, the current thread first checks if it is its own turn, and if it is, moves to state `LOCKED`. If it is not its turn, it checks the state of the opposite thread and unless it is also in `UNLOCKED`, it will transition to `WAITING`. This subtle difference between the two algorithms allows for an advantage of `X2Tv2` over `X2Tv1` when there is more contention, because it is less likely for a thread whose turn is not its own to prevent the thread that is on its turn from going into its critical section after setting its state to `LOCKED`. This algorithm's state machine can be seen in figure 2, and the only difference from figure 1 is that now it's possible to transition directly from `UNLOCKED` to `WAITING`.

Similarly to `X2Tv1`, it has two variables, `q[0]` and `q[1]`, where each variable has one of six possible states, with one bit encoding the `turn` and two bits encoding one of the three states: `UNLOCKED(0)`, `LOCKED(2)`, or `WAITING(6)`. The source code can be seen in Algorithm 2.

2.3 X2Tv3 Algorithm

This algorithm has two variables, `q[0]` and `q[1]`, where each variable has one of six possible states, with one bit encoding the `turn` and two bits encoding one of the three states: `UNLOCKED` (`U`), `LOCKED` (`L`), or `WAITING` (`W`). `X2Tv3` is a variation on `X2Tv2` where it is possible to transition directly from `WAITING` into the critical section, without having to set the state to `LOCKED`. This makes the algorithm significantly more complex but can save one seq-

Algorithm 2 X2Tv2 Algorithm

```
1  int tbit = TURNBIT(q[id].load(relaxed));
2  if (COMP_TURN(tbit, q[other].load(seq_cst)) == other) {
3      q[id].store((W | tbit), seq_cst);
4      while (STATE(lqot = q[other].load(seq_cst)) != U &&
5             COMP_TURN(lqid, lqot) == other) Pause();
6  }
7  q[id].store((L | tbit), seq_cst);
8  while (STATE(lqot = q[other].load(seq_cst)) == L) {
9      if (COMP_TURN(tbit, lqot) == other) {
10         q[id].store((W | tbit), seq_cst);
11         while (STATE(lqot = q[other].load(seq_cst)) != U &&
12                COMP_TURN(tbit, lqot) == other) Pause();
13         q[id].store((L | tbit), seq_cst);
14     }
15 }
16 CriticalSection( id );
17 q[id].store(U | (IS_LW(lqot) ^ TURNBIT(lqot) ^ id), release);
```

cst store under high contention, which increases throughput. The source code can be seen in Algorithm 3.

Similarly to `X2Tv1` and `X2Tv2`, one of the invariants of this algorithm is that a thread in state `WAITING` that doesn't have the turn, will never enter the critical section.

In this algorithm, spinning occurs only when in the `WAITING` state, unlike in `X2Tv1` and `X2Tv2` where spinning can occur in `LOCKED` or `WAITING`. The state machine can be seen in figure 3.

2.4 X2Tv4 Algorithm

This algorithm has three variables, `q[0]`, `q[1]`, and `turn`, where each of the `q` variables has one of two possible states: `UNLOCKED` or `LOCKED`. `X2Tv4` is a variation on the original Peterson's 2-thread algorithm, with an extra check to see if the other thread is in `LOCKED` state. This means that `X2Tv4` always does an extra sequentially consistent load (line 2 of Algorithm 4), but if the other thread is not in `LOCKED` state then the current thread will save one sequentially consistent store, thus needing a single store to a shared variable to enter the critical section, while Peterson's original algorithm always needs two stores. The difference between the two state machines can be seen in figure 4. However, in Pe-

Algorithm 3 X2Tv3 Algorithm

```
1  int tbit = TURNBIT(q[id].load(relaxed));
2  if (STATE(q[other].load()) == U) {
3       q[id].store((L | tbit), seq_cst);
4       if (STATE(lqot = q[other].load(seq_cst)) == U) goto CS;
5       if (STATE(lqot) == W && COMP_TURN(tbit, lqot) == id) goto CS;
6  }
7  q[id].store((W | tbit), seq_cst);
8  while (true) {
9       Pause();
10      if (COMP_TURN(tbit, lqot = q[other].load(seq_cst)) == id) {
11           if (STATE(lqot) != L) goto CS;
12      } else {
13           if (STATE(lqot) != U) continue;
14           q[id].store((L | tbit), seq_cst);
15           if (STATE(lqot = q[other].load(seq_cst)) == U) goto CS;
16           if (STATE(lqot) == W && COMP_TURN(tbit, lqot) == id) goto CS;
17           q[id].store((W | tbit), seq_cst);
18      }
19 }
20 CS: CriticalSection( id );
21 q[id].store(U | (IS_LW(lqot) ^ TURNBIT(lqot) ^ id), release);
```

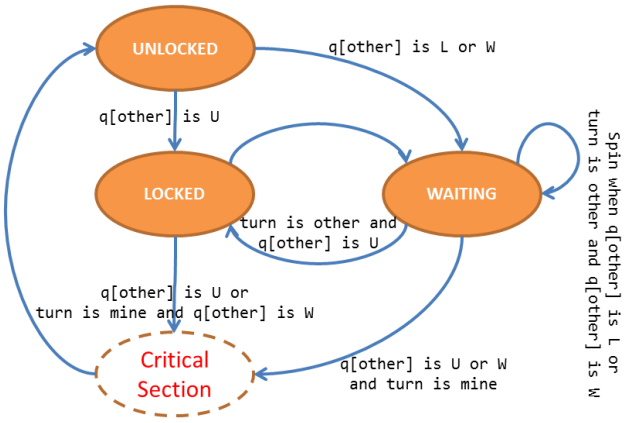

Figure 3. State machine for the X2Tv3 algorithm. The thread will pause/spin/yield only after entering `WAITING` state, when the other thread is in `LOCKED` state, or when it does not have the turn and the other thread is in `WAITING` (or `LOCKED`).

terson's algorithm, the first store can use `memory_order_release` or even `memory_order_relaxed` while on X2TV4 we have to use `memory_order_seq_cst`. This is needed so that the first store doesn't get re-ordered with the load in line 2, and for the second store (in line 3) so it doesn't get re-ordered with the load in line 4.

Algorithm 4 X2Tv4 Algorithm

```
1  q[id].store(L, seq_cst);
2  if (q[other].load(seq_st) == L) {
3       turn.store(other, seq_cst);
4       while (turn.load(seq_cst) == other &&
5                 q[other].load(acquire) == L) Pause();
6  }
7  CriticalSection( id );
8  q[id].store(U, release);
```

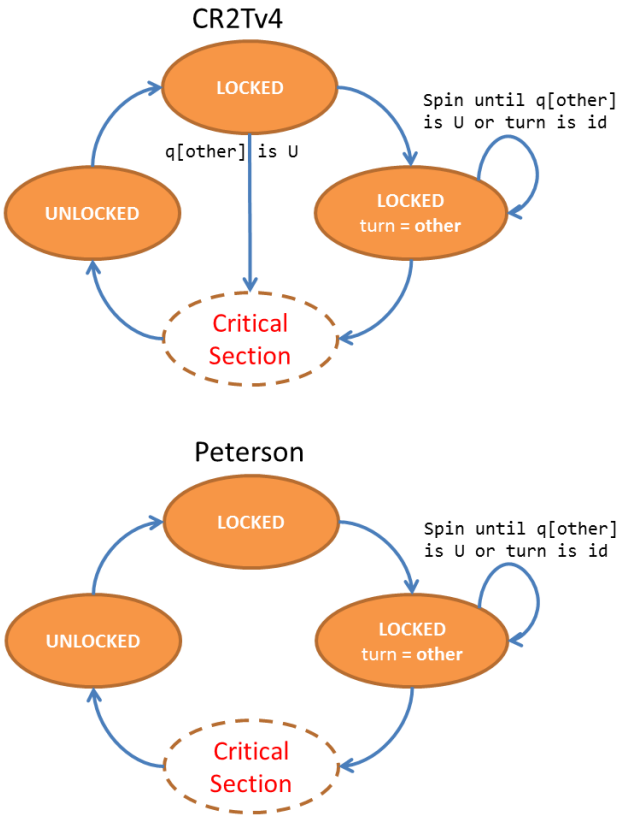

Figure 4. The state machine on the top is for the X2Tv4 algorithm, while the schematic on the bottom is for the original 2-thread algorithm by Peterson. The only difference is that X2Tv4 can transition directly to the critical section without having to set the `turn`.

2.5 X2Tv5 Algorithm

This algorithm is similar to X2Tv4 where instead of doing a write-race on the `turn` variable, it uses the *logical turn* technique similar to Kessels and X2Tv1, X2Tv2, and X2Tv3, thus using only two variables, `q[0]` and `q[1]`.

Algorithm 5 X2Tv5 Algorithm

```
1  int tbit = TURNBIT(q[idx].load(relaxed));
2  q[idx].store(L|tbit, seq_cst);
3  if (STATE(qot = q[otx].load(seq_cst)) == L) {
4       if (tbit != (ot ^ TURNBIT(qot))) {
5            tbit = ot ^ TURNBIT(qot);
6            q[idx].store(L|tbit, seq_cst);
7       }
8       while (COMP_TURN(tbit,qot = q[otx].load(seq_cst)) == ot
9                 && STATE(qot) == L) Pause();
10 }
11 CriticalSection( id );
12 q[idx].store(U|tbit, memory_order_release);
```

2.6 X2Tv6 Algorithm

This algorithm is asymmetric and has three variables, `q0`, `q1`, and `turn`. The state variable `q0` can have one of two possible states: `UNLOCKED` (`U`), or `LOCKED` (`L`). The `q1` state variable can have one of three possible states: `UNLOCKED` (`U`), `LOCKED` (`L`), or `WAITING` (`W`).

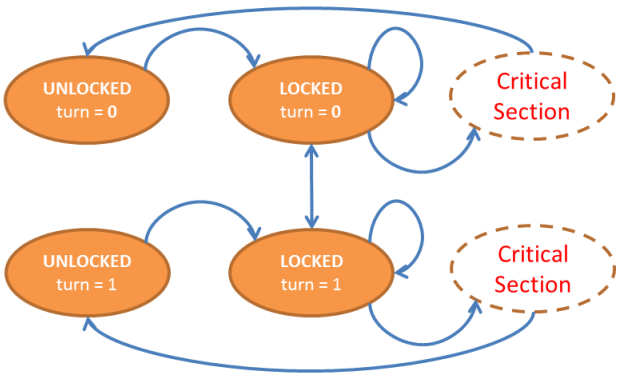

Figure 5. When in `LOCKED` state, a thread will change its own turn bit if and only if the other thread is in `LOCKED` state and the composed turn is hers.

The source code can be seen in Algorithm 6 and the first noticeable thing is that the code paths are very asymmetrical, in the sense that the code the thread with id 0 will execute is considerably smaller than the code that will be executed by the thread with id 1. This asymmetry provides increased throughput when the only thread executing is the thread with id 0, thus being beneficial in low contention, or when applied to tournament algorithms when the total number of competing threads is not a power of 2.

Algorithm 6 X2Tv6 Algorithm

```
1  if ( id == 0 ) {
2      q0.store(L, seq_cst);
3      while (turn.load(seq_cst) == 1) Pause();
4      while (q1.load(acquire) == L) Pause();
5      CriticalSection( id );
6      if (q1.load(relaxed) != U) turn.store(1, release);
7      q0.store(U, release);
8  } else {
9      q1.store(W, seq_cst);
10     while (q0.load(seq_cst) != U &&
11             turn.load(acquire) == 0) Pause();
12     if (turn.load(acquire) == 1) goto CS;
13     if (q0.load(acquire) != U) {
14         while (turn.load(acquire) == 0) Pause();
15     } else {
16         q1.store(L, seq_cst);
17         if (q0.load(seq_cst) == U) goto CS;
18         q1.store(W, seq_cst);
19         while (turn.load(seq_cst) == 0) Pause();
20     }
21     CS: CriticalSection( id );
22     if (turn.load(relaxed)) turn.store(0, release);
23     q1.store(U, release);
24 }
```

In X2Tv6, the thread with id 0 is the only one that can toggle the `turn` from 0 to 1, and the thread with id 1 can only toggle from 1 to 0. The toggling of the `turn` variable is used as a mechanism to *pass the turn* to the other thread. After leaving its critical section, the thread with id 0 will toggle the `turn` if it sees the other thread in `LOCKED` or `WAITING` state (line 6 of Algorithm 6).

For the thread with id 1 the behavior is slightly different. It will always set the `turn` to the other thread (value 0), regardless of the other thread's state (line 21 of Algorithm 6). This means that the next time it will try to enter the critical section, thread 1 will need to either wait for the `turn` to be set to 1 by thread 0, or wait for thread

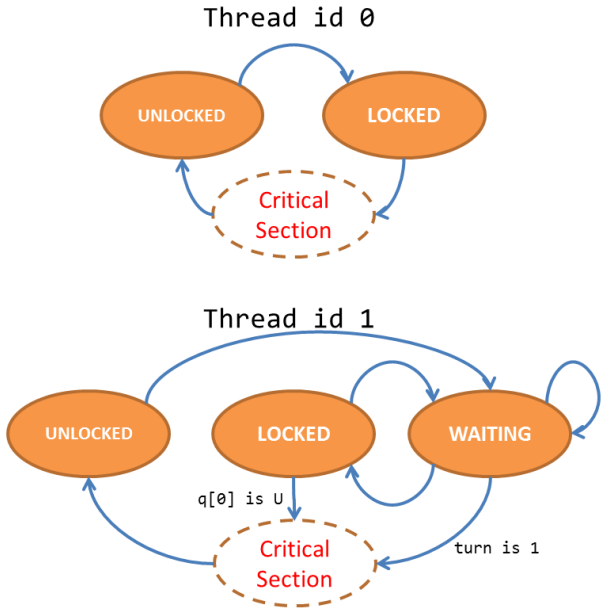

Figure 6. State machine for the X2Tv6 algorithm. The initial state is at `UNLOCKED`. There are two state machines, one for each thread.

0's state to be `UNLOCKED` (lines 10, 13, and 14 of Algorithm 6). Notice that unlike other algorithms (like Peterson's), there is no write-race on `turn`, because it is always toggled by the thread that entered the critical section, either in line 6 or 22 of Algorithm 6.

This algorithm assumes that the thread with id 0 has priority relative to the thread with id 1, since by default, `turn` is set to 0 and if both threads arrive at the same time, thread 0 will be the one that gets to enter the critical section. Despite the preference, this algorithm is starvation-free because when thread 1 arrives (state `WAITING`) it will have to wait for thread 0 to be in `UNLOCK`, or for thread 0 to handover the turn to thread 1 next time it notices that thread 0 intents to enter the critical section.

2.7 X2Tv7 Algorithm

This algorithm is similar to X2Tv6 but uses just two variables and a *logical turn* based on an extra bit of `q0` and `q1`, similarly to what was done in X2Tv1. The variables `q0` and `q1` can each have one of three states `LOCKED`, `WAITING`, or `UNLOCKED`, and they use one extra bit for the turn.

Line 7 is not needed for the correctness of the algorithm because the local variable `lq1` was read before in 5. Same thing for 25 because `lq0` was read in line 18.

2.8 X2Tv8 Algorithm

This algorithm has three variables, `q0`, `q1`, and `turn`, where each of the `q0` variable has three possible states: `UNLOCKED`, `LOCKED`, or `WAITING`; and the `q1` variable has two possible states: `UNLOCKED` or `LOCKED`.

In this algorithm, the thread with id 0 has a kind of preference for entering the critical section, needing only one store and one load to enter the critical section in the uncontended case (lines 2 and 3 of Algorithm 10). The thread with id 1 will always *pass the turn* to the thread with id 0 by setting `turn` to 0 (line 20 of Algorithm 8), and when it wants to enter the critical section, it must first make sure that thread id 0 is not attempting to also enter, and if so, wait for thread id 0 to *see* that thread id 1 is attempting to enter the critical section (in line 4), and move to `WAITING` or `UNLOCKED` state.

Algorithm 7 X2Tv7 Algorithm

```
1  if ( id == 0 ) {
2      const int tbit = TURNBIT(q0.load(relaxed));
3      q0.store(L|tbit, seq_cst);
4      while (COMP_TURN(q1.load(seq_cst),tbit) == 1) Pause();
5      while (STATE(lq1 = q1.load(acquire)) == L) Pause();
6      CriticalSection( id );
7      q0.store((U | (IS_LW(lq1) ^ TURNBIT(lq1))), release);
8  } else {
9      const int tbit = TURNBIT(q1.load(relaxed));
10     q1.store(W|tbit, seq_cst);
11     while (true) {
12         lq0 = q0.load(seq_cst);
13         if (COMP_TURN(lq0, tbit) == 1) goto CS;
14         if (STATE(lq0) == U) break;
15         Pause();
16     }
17     if (STATE(q0.load(acquire)) == U) {
18         q1.store(L|tbit, seq_cst);
19         if (STATE(q0.load(seq_cst)) == U) goto CS;
20         q1.store(W|tbit, seq_cst);
21     }
22     while (COMP_TURN(lq0 = q0.load(seq_cst),tbit) == 0) Pause();
23     CS: CriticalSection( id );
24     q1.store((U | TURNBIT(lq0)), release);
25 }
```

There is a particularly tricky scenario where the `turn` is set to 1 but the lock is taken by thread with id 0. It occurs when `turn` starts at 0 and the thread with id 1 runs up to line 13, then thread id 0 runs to line 3 (it now has the lock), then thread id 1 executes lines 14 and 15 thus setting `turn` to 1 to indicate that the turn should be its own, but it will do one last check on `q0` to see if the lock has been taken (line 16), which in this case will be in state `LOCKED` and cause thread id 1 to spin until thread id 0 changes `q0` back to `UNLOCKED`.

Algorithm 8 X2Tv8 Algorithm

```
1  if ( id == 0 ) {
2      q0.store(L, seq_cst);
3      if (turn.load(seq_cst) == 1) {
4          if (q1.load(acquire) == L) q0.store(W, seq_cst);
5          while (turn.load(seq_cst) == 1) Pause();
6      }
7      CriticalSection( id );
8      if (q1.load(acquire) == L) turn.store(1, release);
9      q0.store(U, release);
10 } else {
11     q1.store(L, seq_cst);
12     while (turn.load(seq_cst) == 0 &&
13             q0.load(acquire) != U) Pause();
14     if (turn.load(acquire) == 0) {
15         turn.store(1, seq_cst);
16         while (q0.load(seq_cst) == L) Pause();
17     }
18     CriticalSection( id );
19     q1.store(U, relaxed);
20     turn.store(0, release);
21 }
```

2.9 X2Tv9 Algorithm

This algorithm has two variables, `q0`, and `q1`, with each using two bits. In `q0` one of the bits encodes the version, and the other bit encodes the state `LOCKED` or `UNLOCKED`. In `q1` one bit represents

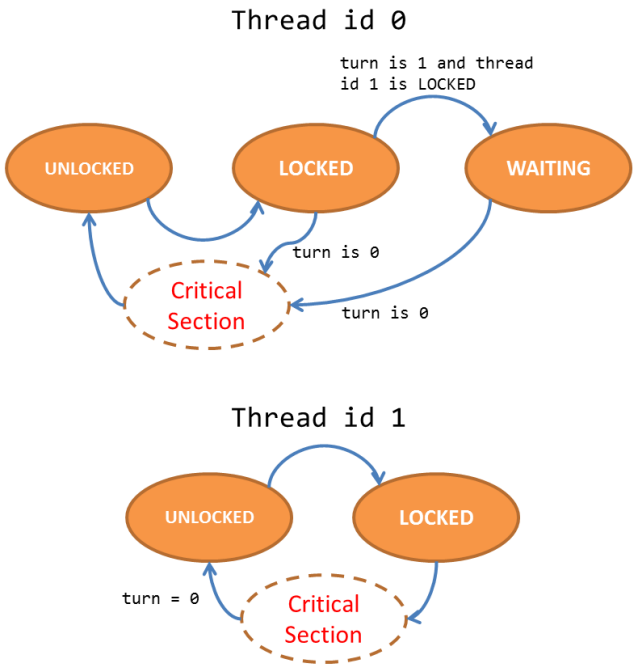

Figure 7. State machine for the X2Tv8 algorithm. There are two state machines, one for each thread. The thread with id 0 will only enter the critical section when the `turn` is 0. The thread with id 1 will always set the `turn` to 0, thus giving *preference* to the other thread.

a logical `LOCKED` state on version `0`, and the other bit represents `LOCKED` on version `1`. This algorithm is based on the algorithm used in Left-Right [17] but adapted for mutual exclusion, encoding the two *ReadIndicators* in a single variable `q1`, and the *versionIndex* as one of the bits of `q0`.

Besides being relatively simple, using just two variables, and providing freedom from starvation, this algorithm allows to enter the critical section doing a single (sequentially consistent) atomic store when running uncontended, and always exits the critical section doing a single (release) atomic store.

In our implementation we used the macro `VERSION()` to extract the bit of the version from `q0` and the macro `IS_LOCKED()` to extract the bit representing the state `LOCKED/UNLOCKED`.

2.10 X2Tv10 Algorithm

This algorithm is highly asymmetric and uses just two variables, `q0` and `q1`.

In the implementation shown in Algorithm 10 we have chosen the following encoding: For thread id `0`, `VERSIONBITS=0x3`, `WAITV0=0x4`, `WAITV1=0x8`; For thread id `1`, `ARRIVEV0=0x1`, `ARRIVEV1=0x2`.

The idea behind X2Tv10 is loosely inspired by X2Tv9, where the thread with id 0 indicates to thread 1 for which version it will wait using the bits `WAITV0` and `WAITV1`. Similarly to X2Tv9, the thread with id 1 will use the bits `ARRIVEV0` and `ARRIVEV1` to indicate the *arrival* and *depart* depending on which *version* it read.

An interesting characteristic of X2Tv10 is that the thread with id 1 will *always* enter the critical section doing only one sequentially consistent store, and leave it with another sequentially consistent store. This provides a good throughput for the uncontended scenario, as will be shown in section 3. The design goal in this algorithm was to make one of the threads do as little stores as possible,

Algorithm 9 X2Tv9 Algorithm

```
1  if ( id == 0 ) {
2      const int newVersion = 1 − VERSION(q0.load(relaxed));
3      q0.store(L|newVersion, seq_cst); // Toggle current version
4      while ((q1.load(seq_cst) & 1<<(1−newVersion)) != 0) Pause();
5      CriticalSection( id );
6      q0.store(U|newVersion, release);
7  } else {
8      const int firstq0 = q0.load(acquire);
9      q1.store(1<<VERSION(firstq0), seq_cst); // Set the bit seen in version
10     int lq0 = q0.load(seq_cst);
11     if (lq0 != firstq0) {
12         q1.store(0x3, seq_cst); // Set both version bits
13         lq0 = q0.load(seq_cst);
14         q1.store(1<<VERSION(lq0), seq_cst); // Set just the latest version bit
15     }
16     if (IS_LOCKED(lq0)) {
17         while (q0.load(seq_cst) == lq0) Pause();
18     }
19     CriticalSection( id );
20     q1.store(0, release);
21 }
```

Algorithm 10 X2Tv10 Algorithm

```
1  if (id == 0) {
2      int lq0 = q0.load(relaxed);
3      const int ver = (q0 & VERSIONBITS);
4      const int otver = 1 − ver;
5      while (true) {
6          while ((q1.load(seq_cst) & ARRIVEV(otver)) != 0) Pause();
7          lq0 |= WAITV(otver);
8          q0.store(lq0, seq_cst);
9          if ((q1.load(seq_cst) & ARRIVEV(otver)) == 0) break;
10         lq0 &= ~WAITV(otver);
11         q0.store(lq0, seq_cst);
12     }
13     lq0 += 1+(otver∗2); // Toggle versionIndex: 0 to 3 or 1 to 2
14     q0.store(lq0, seq_cst);
15     while (true) { // Wait if other thread is on previous version
16         while ((q1.load(seq_cst) & ARRIVEV(ver)) != 0) Pause();
17         lq0 |= WAITV(ver);
18         q0.store(lq0, seq_cst);
19         if ((q1.load(seq_cst) & ARRIVEV(ver)) == 0) break;
20         lq0 &= ~WAITV(ver);
21         q0.store(lq0, seq_cst);
22     }
23     q0 −= 2; // Toggle versionIndex: 3 to 1 or 2 to 0
24     CriticalSection( id );
25     q0.store(lq0 & (~(WAITV0|WAITV1)), release); // Set WAITV0/V1 to 0
26 } else {
27     const int ver = q0.load(acquire) & 1;
28     const int otver = 1 − ver;
29     q1.store(ARRIVEV(ver), seq_cst);
30     while (q0.load(seq_cst) & WAITV(ver)) Pause();
31     if (q0.load(acquire) == otver) {
32         q1.store(ARRIVEV(otver), seq_cst);
33         while (q0.load(seq_cst) & WAITV(otver)) Pause();
34     }
35     CriticalSection( id );
36     q1.store(0, release);
37 }
```

at the cost of increasing the complexity and the minimum number of stores in the other thread.

2.11 Correctness

All the X2T algorithms described in this section have been ported to the Promela language and ran on SPIN [6]. These two-thread algorithms are simple enough that the SPIN model checker is capable of examining all possible interleavings and prove that all these algorithms satisfy mutual exclusion, assuming sequentially consistent access for all shared variables. The Promela files will be made available before publication.

3. Experimental Results

In order to compare the performance of our algorithms with other previously known algorithms, we ran a set of microbenchmarks on the following architectures: x86-AMD and ARMv8. The following machines and tools were used to run the microbenchmark:

- AMD Opteron 6272 with 2 CPUs total of 32 cores, Windows 10, GCC 5.2.0 (Mingw);
- ARMv8 64bit with 8 cores, Ubuntu Linux 15.04, GCC 5.2.1;

We used as basis for our benchmarks the framework developed by Buhr, Dice and Hesselink [2] and available on Github [14]. Based on their implementations, we ported the following starvation-free algorithms from C99 to C++11 and ran them in our microbenchmarks, using the same naming convention as on the survey by Buhr, Dice and Hesselink [2]:

- Dekker(A,B,C,Orig,RW): Dekker's algorithm [5] and variations;
- Peterson: Peterson's 2-thread algorithm [15];
- Peterson-Fischer: An algorithm by Peterson and Fischer [16];
- Kessels: An algorithm by Kessels [8];
- Tsay: An algorithm by Tsay [19];
- X2Tv*x*: Our own algorithms, as described in section 2;

With the advent of memory models on modern programming languages, like the one in C11/C++1x, the algorithms can be written once, and safely ran on any CPU architecture without fear of incorrect results due to re-ordering on specific architectures. This ability of C11 and C++1x to let the compiler insert the needed memory fences depending on the target build, led us to write the code for the benchmarks in C++11.

We ran the microbenchmarks with two scenarios: one thread (uncontended) and two threads (contended). In these implementations, we replaced the `Pause()` function with a `PAUSE` instruction when running on x86, and replaced it with a call to `sched_yield()` for ARM. The results are shown in table 2, where each data point is the median of 9 runs, and each run taking 20 seconds. The three highest performing algorithms of each column are displayed in bold.

As seen on table 2, for the AMD Opteron, the top three winners in the 1 thread scenario are DekkerRW, Tsay, and X2Tv3, while for the 2 thread scenario are Tsay, Peterson-Fischer and X2Tv9. For the ARMv8, the top three winners in the 1 thread scenario are X2Tv4, DekkerRW, and Peterson 2-thread, while for the 2 thread scenario are DekkerA, X2Tv1, and DekkerRW.

4. Conclusion

Finding solutions to the two-thread mutual exclusion problem remains an interesting topic, not just due to its theoretical allure and possible insights into other related problems (like mutual exclusion locks or reader-writer locks), but because these simple algorithms

	x86 AMD-Opteron		ARMv8	
	1 thread	2 threads	1 thread	2 threads
DekkerOrig	131	40	135	70
DekkerA	153	47	134	**121**
DekkerB	131	44	135	73
DekkerC	131	33	114	60
DekkerRW	**163**	37	**141**	**74**
Peterson 2-thread	131	44	**139**	40
Peterson-Fischer	108	**51**	131	40
Kessels	109	37	133	39
Tsay	**162**	**52**	129	38
X2Tv1	127	42	115	**88**
X2Tv2	156	35	134	48
X2Tv3	**157**	46	136	48
X2Tv4	132	39	**146**	31
X2Tv5	130	39	118	33
X2Tv6	129	35	138	38
X2Tv7	144	44	135	43
X2Tv8	149	40	119	39
X2Tv9	138	**48**	138	44
X2Tv10	129	37	116	59

Table 2. Results for x86 and ARMv8. The *1 thread* columns show the results when there is a single thread (the thread id 0) attempting to enter the critical section, and the *2 threads* columns show the results when both threads are continuously attempting to enter the critical section. Values are in millions of operations per 20 seconds.

can be directly used as a building block to more complex concurrency algorithms, for example, using Buhr's tournament [2] to make a mutual exclusion lock, or Hseih and Weihl's algorithm [7] to implement a Reader-Writer lock that scales well for Readers. 2-thread algorithms have the extra advantage that due to their simplicity, they are easy to understand, and a proof of mutual exclusion can be provided automatically with formal verification tools such as SPIN [6].

In this paper, we've shown ten new 2-thread algorithms, all with freedom from starvation, and that when executing without contention, they all require a single atomic store to a shared variable to enter the critical section and another atomic store to leave it, a property that no other previously known 2-thread solution is able to provide. Furthermore, some of these algorithms, like X2Tv6 and X2Tv10, will provide such a guarantee for one of its threads, even when the two threads are simultaneously competing for the critical section. On top of it, seven of these algorithms have a memory usage of only one word per thread (two words in total) when, of the previously known algorithms, only Peterson-Fischer was able to provide such a low memory requirement.

The 2-thread algorithms by Dekker and Peterson have been examined, coded, explained, and taught, many times over since their discovery, and we hope some of the new algorithms described in this paper will follow on their footsteps and open doors to new insights for students and researchers to explore.

Acknowledgments

We would like to thank Dave Dice, Peter Buhr, and Wim Hesselink for making all of the source code used in their paper publicly available, which allowed us to make a faster progress on our own algorithms.

References

[1] Y. Bar-David and G. Taubenfeld. Automatic discovery of mutual exclusion algorithms. In *Distributed Computing*, pages 136–150. Springer, 2003.

[2] P. A. Buhr, D. Dice, and W. H. Hesselink. High-performance n-thread software solutions for mutual exclusion. *Concurrency and Computation: Practice and Experience*, 27(3):651–701, 2015.

[3] T. Craig. Building fifo and priorityqueuing spin locks from atomic swap. Technical report, Technical Report 93-02-02, University of Washington, Seattle, Washington, 1994.

[4] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[5] E. W. Dijkstra. Solution of a problem in concurrent programming control. In *Pioneers and Their Contributions to Software Engineering*, pages 289–294. Springer, 2001.

[6] G. J. Holzmann. The model checker spin. *IEEE Transactions on software engineering*, (5):279–295, 1997.

[7] W. C. Hsieh and W. E. Weihl. Scalable reader-writer locks for parallel systems. Technical Report MIT/LCS/TR-521, MIT Laboratory for Computer Science, Cambridge, MA, 1991. Available: `http://www.lcs.mit.edu/publications/pubs/pdf/MIT-LCS-TR-521.pdf`.

[8] J. L. W. Kessels. Arbitration without common modifiable variables. *Acta informatica*, 17(2):135–141, 1982.

[9] D. Knuth. Additional comments on a problem in concurrent programming control. *Communications of the ACM*, 9(5):321–322, 1966.

[10] L. Lamport. The mutual exclusion problem: partiistatement and solutions. *Journal of the ACM (JACM)*, 33(2):327–348, 1986.

[11] P. Magnusson, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *Parallel Processing Symposium, 1994. Proceedings., Eighth International*, pages 165–171. IEEE, 1994.

[12] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

[13] J. M. Mellor-Crummey and M. L. Scott. Synchronization without contention. In *ACM SIGARCH Computer Architecture News*, volume 19, pages 269–278. ACM, 1991.

[14] D. D. Peter Buhr and W. Hesselink. concurrent-locking. `https://github.com/pabuhr/concurrent-locking`, 2015.

[15] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.

[16] G. L. Peterson and M. J. Fischer. Economical solutions for the critical section problem in a distributed system. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 91–97. ACM, 1977.

[17] P. Ramalhete and A. Correia. Brief announcement: Left-right-a concurrency control technique with wait-free population oblivious reads. *Distributed*, page 663, 2015.

[18] P. Ramalhete and A. Correia. Tidex: a mutual exclusion lock. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, page 52. ACM, 2016.

[19] Y.-K. Tsay. Deriving a scalable algorithm for mutual exclusion. In *Distributed Computing*, pages 393–407. Springer, 1998.