High-performance $N$-thread software solutions for mutual exclusion

Peter A. Buhr$^{1,*}$, David Dice$^2$ and Wim H. Hesselink$^3$

$^1$Cheriton School of Computer Science, University of Waterloo, Waterloo, ON, Canada
$^2$Oracle Labs, Burlington, MA, USA
$^3$Department of Computing Science, University of Groningen, 9700 AK Groningen, The Netherlands

SUMMARY

Software solutions for mutual exclusion developed over a 30-year period, starting with complex ad hoc algorithms and progressing to simpler formal ones. While it is easy to dismiss software solutions for mutual exclusion, as this family of algorithms is antiquated and most platforms support atomic hardware instructions, there is still a need for these algorithms in threaded, embedded systems running on low-cost processors lacking atomic instructions. While $N$-thread solutions are usually short (10–25 lines of code), each is ingenious with exceptionally subtle aspects, often making it difficult to prove correctness or construct an implementation. This work examines correctness and performance of the implementations. An extensive survey of existing algorithms is presented, with explanations of the intuition behind the algorithms and how they work. Several errors were found and corrections made, as well as a few small improvements, in the existing algorithms; two new high-performance algorithms were developed. Finally, a worst-case high-contention performance experiment is performed to compare the algorithms and contrast them with three common locks based on hardware atomic instructions. The results show our two new algorithms are highly competitive with an equivalent hardware lock (Mellor-Crummey and Scott) over a range of 1–32 processors. Hence, threading is a viable alternative to event-driven programming for complex embedded systems without atomic instructions. Copyright © 2014 John Wiley & Sons, Ltd.

Received 12 May 2013; Revised 29 January 2014; Accepted 2 February 2014

KEY WORDS: software solution; mutual exclusion; performance experiment

1. INTRODUCTION

Synchronization and mutual exclusion [1, § 4] (SME) are the basic mechanisms necessary to control concurrent execution, allowing timing relationships among threads and controlling access to shared resources. Of these two fundamental mechanisms, synchronization is simpler than mutual exclusion: synchronization only requires a delay if an event has not occurred, while mutual exclusion has a complex set of requirements for a critical section (CS) (Section 3). This paper focuses on the more complex mutual exclusion.

There are two mechanisms for providing SME: using software (assignment and control structures) or using atomic read-modify-write instructions. Within each category, there are a number of alternative solutions. This paper focuses on software solutions.

So why care about software solutions? Simple microcontrollers (e.g., PIC, ARC, Altera, Marvell, and Freescale) are used in low-cost devices like basic cell phones, cameras, printers, music players, toys, and even singing greeting cards. Billions of these devices are created annually and will continue to exist for years to come. In fact, it is highly likely many people are carrying around a
software solution for mutual exclusion in one of their portable devices. Most of these simple devices provide complex functionality, and it is useful to structure the software as preemptive threads, often just so the software is compatible with its advanced counterparts running on a multicore processor with a full set of atomic instructions. However, in many cases, the low-cost processors within these devices may have no atomic instructions. In these embedded systems, it is possible to establish a maximum bound on the number of threads that interact and compete for shared resources and hence to use a software solution for SME rather than restructuring the software sequentially, such as event driven.

Initially, software solutions for mutual exclusion only handled two threads. The first solution was Dekker’s, and it does not rely on hardware atomicity. Peterson–Fischer [4, p. 92], Kessels [5, p. 137], and Peterson [6, p. 115] require atomic read/write for assignment without atomic read-modify-write instructions.

Moving from 2-thread algorithms to $N$-threads turns out to be complex:

Another possible myth is that Dekker’s solution can be trivially modified to solve the $n$ process case. The algorithms known to the author (G. L. Peterson) actually require major changes in form that result in entirely new algorithms, even when $n$ is two. [6, p. 116]

While $N$-thread solutions are usually short (10–25 lines of code), each is ingenious with exceptionally subtle aspects, often making it difficult to prove correctness or construct an implementation. Correctness of these algorithms is often established in the original papers where they were presented. This work examines correctness and performance of the $N$-thread implementations. Implementation correctness is an issue because all software solutions can be corrupted because of compiler and hardware optimization assumptions for sequential execution (Section 4). Performance is an issue because it differentiates the software algorithms and provides a contrast with atomic hardware algorithms.

In contrast to software solutions for creating mutual exclusion, special atomic hardware instructions are available for creating mutual exclusion. These instructions guarantee a read, optional action, and write occur atomically. For example, a test-and-set instruction reads a value, writes a marker value, and provides the original value read; no interruption can occur during the read/write sequence. A fetch-and-increment instruction reads a value, increments the value, writes the incremented value, and provides the original value read; no interruption can occur during the three actions. For atomic instructions, the hardware controls execution ordering, that is, precluding interruption between the read and write, which is impossible with software solutions. In virtually all situations, it is easier and faster to create mutual exclusion using an atomic instruction versus a software solution because there is more fundamental atomicity to build on. While it is interesting that mutual exclusion can be created without any atomic assistance from the hardware, Section 4 shows memory instructions are necessary to handle races on shared variables.

There are non-locking alternatives for implementing preemptive threading on processors without atomic instructions. One approach is to disable interrupts to provide mutual exclusion. This approach works for simple embedded systems, where complete knowledge of the system and its small size make it possible to ensure correct behavior. However, if embedded systems support dynamic inclusion of applications (apps), using interrupt disabling for mutual exclusion by untrusted applications becomes risky. Another approach for uniprocessors is restartable CSs [7, Figure 3], which does not require disabling interrupts for arbitrary user-mode code; for example, various Linux implementations on uniprocessor ARM systems emulate compare-and-swap (CAS) using this approach. However, the cost can be high as it requires an interaction between application and kernel, and some of these systems run without an operating system.

Furthermore, Moore’s law shows it is relatively easy to add more processors, but it remains complex to add atomic instructions among processors. As a platform evolves, application developers push CPU designers to eventually include atomic capabilities (with a memory model and appropriate

---

§Dekker never published his solution directly. The algorithm and credit for its solution are first discussed by Dijkstra in [3, pp. 58–59].
fence instructions). The NVidia GPUs reflect a recent example of this transitional state where atomic instructions, like compare-and-swap, were added only 2–3 years ago. While it is easy to dismiss software solutions for SME, as this family of algorithms is antiquated and most platforms suppose atomics, there always seems to be some CPU in its evolutionary trajectory that can make use of these algorithms. Hence, software solutions still provide a transitional bridge in these contexts.

The contributions of this work are

1. a fairly complete set of \( N \)-thread software algorithms, developed over 30 years, for practitioners to choose from,
2. explanations of how the algorithms work including simplifications and corrections,
3. implementations for these algorithms as many of the published algorithms are difficult to understand and some have implementation errors,
4. show how to make the implementations work given that the compiler and relaxed memory models conspire to break the algorithms because of sequential assumptions,
5. two new software solution algorithms that perform close to locks built from atomic hardware, and
6. compare the performance of the algorithms to see if one is better than the others with respect to a minimal and maximal high-contention scenario, and contrast the software solutions with three atomic hardware locks.

The implementation of the algorithms are as close to the original papers as possible (some small cleanup); as a result, the occasional \texttt{goto} is retained. Removing \texttt{goto}s would require significant refactoring making it difficult to verify an algorithm is the same as the original. All algorithms work from 1 to \( N \) threads. Finally, some of the algorithms contain loops with constant factors, which the reader may believe can be factored out to produce better performance. We have verified, at optimization level three (\texttt{-O3}), the compiler produces extremely efficient code requiring no hand optimization.

2. CRITICAL SECTION

A critical section is a pairing: shared data and code manipulating the data executed by multiple threads. Collectively, this pairing denotes an instance of a CS. It is a common mistake to consider only the code as the CS. Many CS instances may exist for a particular block of code with different objects, but an object can only be associated with one CS at a time; hence, there is a many-to-one, that is, objects to code, relationship. For example, multiple threads may be simultaneously in the write routine writing to different files. In this case, the fact that the threads execute the same code, possibly at the same time, does not imply a CS. Only when threads are in the write routine for the same file is a CS generated.

3. MUTUAL EXCLUSION

Simple mutual exclusion ensures only one thread is in a CS at a time. The algorithms in this paper protect only simple CSs. Complex mutual exclusion may allow multiple threads in a CS but with some complex restriction among threads, such as the readers/writer problem [8] where multiple readers can share the resource simultaneously but writers must be serialized. In all cases, a CS is protected by wrapping it with mutual exclusion code placed before (entry protocol) and after (exit protocol). In general, any number of threads may arrive simultaneously at the entry protocol. For software solutions, a fixed bound on the number of participating threads is required because each thread needs a minimum of one bit to communicate a desire to enter.

To preclude trivial solutions for mutual exclusion, such as executing threads serially, all the following rules are required:

1. Only one thread can be in a CS at a time with respect to a particular object [9].
2. Threads may run at arbitrary speed and in arbitrary order [9].
3. If a thread is not in the entry or exit code that controls access to the CS, it may not prevent other threads from entering the CS [10, p. 318].

4. In selecting a thread for entry to a CS, a selection cannot be postponed indefinitely [9]. Not satisfying this rule is called *indefinite postponement* or *livelock*.

5. After a thread starts entry to the CS, it must eventually enter [11, p. 321]. Not satisfying this rule is called *starvation*. Threads waiting to enter can be serviced in any order, as long as each thread eventually enters. If threads are not serviced in first-come first-serve (FCFS) order of arrival, there is a notion of *unfairness*, when waiting threads are overtaken by a thread that arrives later (*bargers*). Barging occurs either by placing newly arriving threads ahead of waiting threads (*cutting in line*) or *bypassing* all waiting threads to receive immediate service.

There is no known \(N\)-thread software solution using a single bit per thread that satisfies all these rules. As will be seen, providing fairness requires additional bits to establish an ordering among delayed threads. In fact, rule 5 is ignored by some practical implementations of mutual exclusion because the chance of long-term starvation is probabilistically low and short bursts of unfairness do not cause problems in these environments. Furthermore, being unfair can result in improved performance when a resource does not have to be directed to a specific thread. For example, a bounded-buffer resource can be unfair by letting an arriving thread barge ahead of waiting threads; the benefit is that the arriving thread has a fast path through the CS to access the resource without blocking. The requirement for the resource is that any thread can process any item from the buffer, that is, there are no directed items to specific threads.

4. MEMORY MODEL

In general, most programs are sequential. Therefore, programming languages and the underlying hardware aggressively optimize the statically generated code and its dynamic execution, respectively, assuming sequential behavior. As a result, concurrent programs that rely on races with shared variables are vulnerable to corruption by compiler or hardware sequential optimizations because many valid sequential optimizations invalidate concurrent programs [12]. Because all software solutions for mutual exclusion have races on shared variable, establishing correct behavior for each solution requires an understanding of the memory model in which the program runs.

Specifically, we use *volatile* to avoid compile-time reordering and elision, hardware fences to control architectural reordering, *pause* to mitigate spinning, and data size and alignment to make efficient use of the cache. Each of these items is discussed in detail.

All software solutions have one or more busy waits in the entry protocols when competing for the CS or waiting for a thread to exit the CS. These busy waits appear as apparent infinite loops reading one or more shared variables, for example,

```c
while ( other == WantIn ); // busy wait
```

spins until the other thread does not want into the CS. Because there is no code in the loop body, it appears to be infinite because variable `other` does not change; however, concurrency allows the value of `other` to be changed by another thread. It is common for a compiler to copy the shared variable into a register making it truly an infinite loop as this thread never sees the actual variable change by another thread. To make this busy loop work, the value of the shared variable must be loaded on each iteration. The mechanism to force loading (and storing) in C is to qualify a variable’s type with *volatile*:

```c
volatile int other;
```

The compiler copies a *volatile* variable for the minimum duration to perform an operation, for example, load the value into a register to increment and immediately store the result. Because *volatile* affects all usages of a variable, it may over-constrain legitimate optimizations in
sequential regions, such as a CS. All shared variables used in the entry/exit protocols for the software implementations are declared with the `volatile` qualifier to prevent problems.

It is also necessary to insert a special instruction, called a `fence`, to prevent the hardware from reordering loads before stores across certain boundaries among different variables (versus the same variable). For example, in Figure 1(b) are the lines

```c
intents[id] = WantIn; // write
Fence();
if ( intents[1 - id] == DontWantIn ) break; // read
```

Without the fence between the write and read of array `intents`, the hardware can execute the load `intents[1 - id]` before the store `intents[id]` when it determines the read and write are to different memory locations (i.e., different subscripts). However, this implicit change allows both threads to simultaneously enter the CS because both first load that the other thread does not want into the CS, then both set they want in, and both proceed out of the loop. In a sequential program, this optimization is benign because of the single thread; in a concurrent program, this optimization must be precluded for correctness. For all the algorithms in this paper, the minimal number of store/load fence instructions are inserted for x86 and SPARC architectures to ensure correctness but allow maximum performance. Platforms with weaker memory models such as ARM or power-PC may need additional fences.

While our fence instructions prevent reordering of access to lock data, additional fence instructions may be required to prevent reordering of application data into or outside of the protecting mutual exclusion code. On hardware platforms with a total store ordering memory model (x86 and SPARC), the fences within the mutual exclusion code also serve to prevent data accesses within the CS from reordering into or before the entry protocol (lock acquire). Furthermore, reordering of the

```c
enum Intent { DontWantIn, WantIn };  
volatile Intent intents[2] = { DontWantIn, DontWantIn };  // SHARED

intents[id] = WantIn; // entry protocol
Fence();
while ( intents[1 - id] == WantIn ) Pause();

CriticalSection();
intents[id] = DontWantIn; // exit protocol
(a) Declare Intent

for ( ;; ) { // entry protocol
  intents[id] = WantIn;
  Fence();
  if ( intents[1 - id] == DontWantIn ) break;
  intents[id] = DontWantIn; // retract intent
  Fence();
  while ( intents[1 - id] == WantIn ) Pause();
}
CriticalSection();
intents[id] = DontWantIn; // exit protocol
(b) Retract Intent

if ( id == 1 ) {
  intents[id] = WantIn;
  Fence();
} else {
  for ( ;; ) {
    intents[id] = WantIn;
    Fence();
    if ( intents[1 - id] == DontWantIn ) break;
    intents[id] = DontWantIn; // retract intent
    Fence();
    while ( intents[1 - id] == WantIn ) Pause();
  }

 CriticalSection();
  intents[id] = DontWantIn; // exit protocol
  (c) Prioritized Retract Intent
```

Figure 1. Three 2-thread algorithms using declare/retract intent: (a, b) have livelock and (c) has starvation.
CS into or after the exit protocol (lock release) is avoided by relying on total store ordering and the fact that every lock release has at least one store instruction.

Note, the volatile qualifier in C (also C++) is not as strong with respect to the language memory model as in Java. In fact, the volatile qualifier in C was created for use with setjmp/longjmp so variables in the scope of a setjmp are the actual values rather than copied values in registers. This weaker semantics for volatile in C is the reason why it is necessary to manually insert memory fences to prevent the hardware from moving loads before stores in the pipeline or in the cache. In Java, this step is handled automatically for all variables declared volatile. However, Java may over-constrain accesses, inserting more memory fences than necessary, which can preclude some valid hardware optimizations and hence reduce performance. Alternatively, C11 (IOS/IEC 9899:2011 and stdatomic.h) and features in C++11 now provide Java like volatile semantics via new atomic primitives; however, C11 and C++11 are not yet available on all systems and platforms and, like Java, can over-constrain accesses.

The busy-waiting in software solutions for mutual exclusion can cause performance problems, for example, this busy wait

```c
while ( other == WantIn ) Pause(); // busy wait
```

loops around a load (because of the volatile qualifier) and compare instruction. As this loop executes, the processor pipeline fills with load/compare instructions, which takes resources (CPU), space (instruction cache), and power (heat). To mitigate these events, many processors have a pause instruction specifically for this situation. The pause instruction is a hint to the processor of a busy loop in progress. The processor has many options: a simple temporal delay, yielding resources to other sibling pipelines, create a speculation barrier to prevent pipeline flooding, reschedule hyper threads on this core, and stop the branch predictor so when the write occurs there are few or no instructions in the pipeline to flush. For all the algorithms in this paper, a pause instruction is inserted in tight busy-waiting loops to increase performance. We have been conservative in inserting pause instructions, so large busy loops may not have them.

Often, an algorithm may only require a single bit to store information, such as intent to enter. However, from a performance perspective, it is better to store information in the atomically addressable word size for the underlying computer: intptr_t. (All programs in the paper use int as a generic type for the underlying addressable word size.) Because \( N \) is normally \( \leq 64 \), the additional storage cost for using words rather smaller addressable units is minimal. As well, the choice of width can influence the vulnerability to false sharing. There is also a small benefit to align storage on a cache-size boundary; a 128-byte boundary was chosen as a worst-case value. Specific variables, structures, and arrays used in algorithms are aligned on a cache-size boundary using either memalign for dynamically allocated storage or the following macro for static allocation:

```c
#define CALIGN__attribute__(( aligned (128) ))
```

No algorithm has more than two small, shared arrays or matrices, so wasted storage due to over alignment is minimal.

Some attempt is made to avoid the use of division and modulus as these operations are expensive on certain architectures. Therefore, operation \( V \% 2 \) is converted to \( V \& 1 \) and \( V / 2 \) is converted to \( V >> 1 \), where \( V \) is an integer containing small values. As well, instead of cycling around an array using modulus, as in

```c
for ( int i = m; i != k; i = (i + 1) % N ) ...
```

the following is used instead:

```c
inline int cycleUp( int v, int n ) { return ( ((v) >= (n-1)) ? 0 : (v + 1) ); }
inline int cycleDown( int v, int n ) { return ( ((v) <= 0) ? (n - 1) : (v-1) ); }
for ( j = HIGH; j != id; j = cycleUp( j, N ) ) ... 
for ( j = turn; j != id; j = cycleDown( j, N ) ) ...
```

For many compilers, these hand optimizations are unnecessary, but we coded conservatively here.

For clarity when reading the algorithms, all code needed solely to avoid compile-time reordering/elision and architectural reordering is marked in red.
Finally, low-cost low-power devices exist, which have multiple computational-units, for example, DSPs, CPUs, and antennas, that can simultaneously read and write to memory. However, to reduce cost/power, atomicity (conflict resolution) may not be provided so simultaneous read/write causes reads to flicker and simultaneous writes cause scrambled bits.

Multiport memory is widely used in multiprocessor systems because it can provide parallel access to the shared memory. ... Most of these produces do not consider the conflicts caused by accessing to the same data, while a few products only provide simple conflict mechanisms, such as busy control scheme. [13, p. 251]

In these devices, conflict errors occur infrequently or are not an issue. Interestingly, some software mutual exclusion algorithms are so robust they do not require the hardware to provide an underlying notion of atomicity and can safely work in these devices.

Any (software mutual exclusion) algorithm based upon atomic operations cannot be considered a fundamental solution to the mutual exclusion problem. [10, p. 314]

Hence, mutual exclusion algorithms are divided into RW-safe and RW-unsafe [10, 14]. In general, RW-unsafe algorithms are simpler as they are not providing pure mutual exclusion but are building on mutual exclusion in the underlying hardware. All modern computers provide atomic reads and writes, so at that level, the distinction is only useful when explaining differences in algorithms that otherwise do the same thing. In some cases, the RW-safety can have an affect on cache behavior because of their different kinds of memory accesses.

5. SELF-CHECKING CRITICAL SECTION

It is difficult to verify programmatically if an algorithm satisfies all five rules for mutual exclusion. In most cases, failure to meet a rule is demonstrated by a counterexample, which is often difficult to construct. However, it is possible to check for violation of rule 1, that is, two threads in the CS, by creating a self-testing CS.

```c
void CriticalSection(const unsigned int id) {
    static volatile unsigned int CurrTid; // shared, current thread id in critical section
    CurrTid = id; // RACE
    Fence(); // optional
    for(int i = 1; i <= 100; i += 1) // delay
        if(CurrTid != id) abort(); // mutual exclusion violation ?
}
```

The shared variable CurrTid holds the id of the thread currently in the CS, and it is initialized at the beginning of CriticalSection to the calling thread’s id. A thread then loops 100 times pretending to perform the CS, but at the end of each iteration, the thread compares its id with the one in CurrTid for any change. If there is a change, rule 1 has been violated, and the program is stopped. All algorithms in the performance experiments (Section 21) pass this test for billions of entries into the CS during testing.

However, because of the race on the shared variable CurrTid, it is possible to get false negatives, that is, miss mutual exclusion violations. The assignment to CurrTid can delay in a store buffer, that is, committed but not pushed into coherent space, so subsequent loads in the loop fetch the value from the store buffer via look aside instead of the coherent version. If this scenario occurs for multiple threads in the CS, these threads do not detect the violation because their copy of CurrTid in the store buffer is unchanged. The fence after the assignment of CurrTid guarantees seeing a coherent value because the store to coherent space has occurred before any subsequent loads. Hence, the loads in the loop either see the value just stored or the value stored by another thread, which is sufficient to detect violation of mutual exclusion. The 100 loop iterations in the CS increase the potential overlap among threads for a broken algorithm, so the detection time is longer, and hence, the greater the odds of detecting a mutual exclusion violation. As a result, these techniques reduce vulnerability to the race and provide a strong check of mutual exclusion and went a long way in debugging the algorithms by detecting many mutual exclusion errors.
6. PRIORITIZED RETRACT INTENT

As mentioned, it is difficult to generate a 2-thread algorithm for mutual exclusion; it is even more complex to extend it for \( N \)-threads. Two algorithms are presented showing a simple 2-thread algorithm and its \( N \)-thread counterpart. These algorithms are simple because each uses a single, shared bit per thread to indicate intent to enter the CS. The point is to illustrate techniques used by all subsequent algorithms with respect to scanning and retries.

6.1. 2-Thread prioritized retract intent

The basic approach in any software solution is for a thread to arrive at a CS, indicate (declare) its intent to enter, and wait, if necessary, for a thread in the CS to leave [3, § 2.1]. Hence, there is a shared bit for each thread indicating intent, which is set for entry and reset after exit from the CS. Waiting involves scanning the intent bits until none are on implying the CS is empty, and a thread may enter. Figure 1(a) shows the basic structure of this algorithm for two threads with thread ids 0 and 1, respectively. This approach satisfies rules 1–3. Mutual exclusion is guaranteed indirectly because the algorithm fails rule 4 (discussed shortly). There is no code that depends on order and speed of execution (rule 2), and there is no code outside the CS that precludes entry by other threads (rule 3).

The declare-intent algorithm fails rule 4 because in selecting a thread for entry to a CS, the selection can be postponed indefinitely. The failure scenario occurs when threads arrive simultaneously. On a multiprocessor, both threads indicate intent to enter at the same time, both notice the other thread wants in, and both wait for the other thread to retract intent on exit from the CS. However, neither thread can now retract intent because no progress is occurring toward the CS; both threads are spinning in the empty busy loop, mistakenly thinking the CS is occupied. On a uniprocessor, the failure scenario is slightly more complex. The CPU must switch between threads immediately after one thread indicates intent but before the other thread’s intent is checked, that is, between the two entry protocol statements. The other thread now attempts to enter the CS, sets its intent, and detects the other thread’s intent is set. Both threads now spin in the busy loop, taking turns using the CPU to check the other thread’s intent, which never changes. Both the multiprocessor and uniprocessor scenarios result in livelock, denoted by high CPU consumption because of spinning. When threads do not arrive simultaneously, each waits as necessary, and if they arrive simultaneously, livelock occurs; hence, mutual exclusion is never violated (rule 1).

Notice, livelock only occurs on simultaneous arrival, when threads say ‘You go first’, ‘No, you go first’. People solve this problem when one backs down so the other can make progress. Figure 1(b) shows how this notion of retracting intent can be incorporated into the previous approach by backing down in the entry protocol if the other thread has its intent set. The thread that backs down then waits until the other thread exits the CS and retracts its intent. At that point, the thread that backed down starts the entire entry protocol again. Notice, retracting is a pessimistic action: when one thread sees the other thread’s intent is set, it probably means the other thread is in the CS not the entry protocol, so in many cases, it is unnecessary to back down. But it is impossible to tell if a thread is in the entry protocol or CS when its intent is set, so simultaneous arrival must be assumed.

Unfortunately, rule 4 is still broken but with a low probability. Again, the problem occurs if threads arrive simultaneously. On a multiprocessor, the two threads both indicate intent to enter at the same time. They both notice the other thread wants in, and so both retract intent. They both exit the busy loop immediately because neither thread has indicated intent and so both start the protocol over again. If the two threads continue to execute in perfect synchronization, they defer entry forever. However, the chance of staying in perfect synchronization is very low. The moment one thread gets ahead of the other, it enters the CS because the other thread still has its intent retracted. On a uniprocessor, it is even more complex to produce a failure of rule 4. The CPU must switch back and forth after almost every statement in the entry protocol. Furthermore, this perfect synchronization has to be maintained for the livelock situation. The chances of this occurring in practice is extremely low; nevertheless, it fails in theory.
While the retract intent solution failed to provide a perfect solution to rule 4, it comes probabilistically close. If there is some way of breaking the tie on simultaneous arrival, the problem is solved. Figure 1(c) shows one possible approach by assigning different priorities to the threads as a tie-breaking mechanism. When simultaneous arrival occurs, the high-priority thread always goes ahead of the low-priority thread. The entry protocol is now divided into two parts by the if statement: code for the high-priority thread and code for the low-priority thread. Alternatively, two different code fragments can be created with the high-priority code executed by one thread and the low-priority code executed by the other. The code for the high-priority thread is the declare-intent algorithm, and the code for the low-priority thread is the retract intent algorithm. Basically, the high-priority thread never retracts intent; if it detects the other thread has indicated its intent, it just spins with its intent set until the other thread retracts intent, either in the entry protocol or by exiting the CS. The low-priority thread always retracts intent, if it detects the high-priority thread wants into or is already in the CS. The priorities ensure there is never a livelock.

Unfortunately, this approach violates rule 5: after a thread starts entry to the CS, it must eventually enter. In theory, the high-priority thread can always barge ahead of the low-priority thread by racing out of the CS and back into it, so the low-priority thread starves, that is, never enters the CS; in practice, contention for the CS is never high enough for high-priority threads to cause long-term starvation. Section 7 discusses techniques for alternating entry on repeated simultaneous arrival. However, fairness requires more than a single bit per thread to indicate intent to enter.

6.2. $N$-Thread prioritized retract intent

The next step is moving from a 2-thread solution for mutual exclusion to $N$-threads by extending the 2-thread solution. Figure 2 shows the implementation of two $N$-thread prioritize retract intent algorithms. The left version is by Burns and Lynch [15, p. 836], and the right version is by Lamport [14, p. 337] (collectively called the B–L algorithms). Like the 2-thread version, both algorithms are based on fixed priorities and hence have starvation (break rule 5), as a high-priority thread can theoretically preclude a low-priority thread from making progress. Nevertheless, this simple approach illustrates solving at least rules 1–4 for $N$-threads. The algorithm has an array of bits, one per thread, to represent the $N$ intents needed by the threads. A thread’s unique id is used as both its position in the intent array and its priority level, where id = 0 is the highest priority. For example, thread 5 sets and resets element 5 of the intent array and has lower priority than threads 0–4 and higher priority than threads 6–9.

The entry protocol has two steps. In the first step, a thread begins by setting its intent and then linearly searching in the high-priority direction to see if a thread with higher priority has declared intent. (The direction of the search is immaterial.) If such a thread is found, the searching thread retracts its intent and busy waits until the higher priority thread retracts its

---

```c
enum Intent { DontWantIn, WantIn }; volatile Intent intents[N] = { DontWantIn, ... }; // SHARED
L0: intents[id] = DontWantIn; // entry protocol
    Fence();
    for ( int j = 0; j < id; j += 1 )
        if ( intents[j] == WantIn ) { Pause(); goto L0; }
    intents[id] = WantIn;
    Fence();
    for ( int j = 0; j < id; j += 1 )
        if ( intents[j] == WantIn ) goto L0;
L1: for ( int j = id + 1; j < N; j += 1 )
    if ( intents[j] == WantIn ) { Pause(); goto L1; }
    CriticalSection();
    intents[id] = DontWantIn; // exit protocol
    (a) Burns-Lynch
```

---

```c
L: intents[id] = WantIn; // entry protocol
    Fence();
    for ( j = 0; j < id; j += 1 )
        if ( intents[j] == WantIn ) { intents[id] = DontWantIn;
            Fence();
            while ( intents[j] == WantIn ) Pause();
            goto L;
        }
    for ( j = id + 1; j < N; j += 1 )
        while ( intents[j] == WantIn ) Pause();
    CriticalSection();
    intents[id] = DontWantIn; // exit protocol
    (b) Lamport
```

Figure 2. Two similar $N$-thread algorithms based on prioritized retract intent, both with starvation.
intent, and then, the searching thread begins the search over again from its starting position, as in

When a thread has searched the entire high-priority direction and found no thread with higher priority intent, it moves to the second step. In essence, a thread sees a moment in time when there are no higher priority threads ahead of it.

In the second step of the entry protocol, a thread searches in the low-priority direction. However, because all threads in this direction have lower priority, the searching thread never retracts its intent; it just busy waits until the low-priority thread retracts its intent either by leaving the CS or retracting its intent in the entry protocol and then continues along the list, as in

When a thread has searched the entire low-priority direction and found no thread with lower priority intent, it can safely enter the CS.

Two of many complex cases are examined to help in understanding. First, what happens if a thread is searching in the high-priority direction and a higher priority thread starts searching after its intent has been passed by the lower priority thread, as in

This situation is not a problem because when the higher priority thread starts step 2 of the entry protocol, it waits for the lower priority thread to exit the CS. In essence, the lower priority thread started searching before the higher priority thread, and now, the higher priority thread waits for it to complete. In this situation, the time of arrival affects the priority of a thread. The second case is where a low-priority thread declares its intent after the higher priority thread has started its search, as in
This situation is not a problem because when the lower priority thread finds a high-priority thread’s intent, it retracts its intent, and the high-priority thread does not retract its intent when performing the low-priority search. Other cases need to be examined before this algorithm can be shown to satisfy rules 1–4 (see [15] for details). The purpose of this discussion is to illustrate the techniques of scanning, retracting, and retrying, and the reasons for them in software solutions.

While this solution uses only one bit per thread to indicate the thread’s intent to enter the CS, no complete N-thread solution has achieved this lower bound [15]. In effect, dealing with rule 5 (starvation) requires additional information, which correspondingly requires additional bits to represent the information. Furthermore, bits on commodity computers are not atomically accessible, and it is more efficient to access words (versus bytes) on most architectures. Therefore, minimum storage at the bit/byte level is only of theoretical interest. Finally, B–L is RW-safe requiring no underlying atomicity [16].

7. TIE BREAKING/FAIRNESS

There are two basic approaches for dealing with tie-breaking on simultaneous arrival and ensuring fairness on entry to the CS: alternation and racing.

7.1. Alternation

Alternation involves keeping track of the last thread in the CS and preventing that thread from immediate reentry if other threads are waiting. Of course, strict alternation does not work

```c
while (last == me); //entry protocol
CriticalSection();
last = me; //exit protocol
```

because it violates rule 3: if a thread is not in the CS or the entry or exit protocol, it may not prevent other threads from entering the CS. If one thread does not want into the CS, it indirectly precludes the other thread from entering once that thread sets last. For example, a thread cannot make consecutive entries into the CS when the other thread does not want in, because after the first entry/exit, last is set to the thread’s id precluding another entry. Therefore, declaring intent is still the primary mechanism to determine entry, but alternation can be used as a secondary mechanism to break ties and prevent starvation.

Dekker’s algorithm combines retract intent (Figure 1(b)) and alternation. Figure 3 shows the original Dekker algorithm [3, pp. 58–59] and a structured version based on [1, Figure 3], which is analyzed in detail. The shared variable last indicates which thread was last in the CS and is initialized to one of the threads. (Figure 3(a) uses turn in the opposite way to last in Figure 3(b).)

Like priority entry, there is code for the high and low-priority threads. However, the high or low

---

---

Figure 3. Two equivalent Dekker algorithms, original version and structured (no gotos).
thread alternates depending on which thread was last in the CS. The outer loop (line 3) is where
the high-priority thread busy waits, and the inner loop (line 7) is where the low-priority thread
busy waits. The if statement (line 4) controls which thread has low or high priority. The low-
priority loop is nested in the high-priority loop because of the two steps in the exit protocol: setting last and retracting intent. The low-priority thread in the nested loop of the entry protocol might
detect last having changed, exit the busy loop, but still have to wait for the other thread to com-
plete the exit protocol by retracting its intent. In the case of simultaneous arrival, the thread last
in the CS is directed to the low-priority code, where it retracts intent (line 5), so the other thread
can make progress. Hence, livelock is prevented by alternating on simultaneous arrival, and rule 4
is satisfied.

Starvation cannot occur using alternation, because once the low-priority thread sets its intent, it
becomes the high-priority thread as last is no longer equal to this thread; hence, the other thread
cannot enter again, unless it is just entering or is already in the CS. Only if the new high-priority
thread does not get any CPU time could it starve, but it is assumed all threads eventually execute
with some degree of fairness (e.g., because of a preemptive round-robin scheduler).

In Dekker’s algorithm, the race loser retracts intent, so if the thread in the CS attempts reentry,

it does not exclude itself, while the new high-priority thread delays exiting the low-priority
busy loop and resetting its intent. Therefore, unbounded overtaking is allowed by the new low-
priority thread. However, once the new high-priority thread sets its intent, there is a bound of one
as the new low-priority thread can be entering or in the CS. Unbounded overtaking is allowed
by rule 3: not preventing entry to the CS by the delayed thread. Finally, Dekker’s algorithm
is RW-unsafe. Assume thread T0 exits the CS and the assignment to intents[0] (line 14)
flickers from WantIn to DontWantIn, so thread T1 can now enter and exit the CS, setting
last to 1. Assume T1 enters again, intents[0] flickers to WantIn, directing T1 to line 4. T0
finally sets intents[0] to DontWantIn and terminates. Then, T1 is trapped at line 7 because
last == 1.

7.2. Racing

While alternation uses information set in the exit protocol to prevent livelock and starvation in the
entry protocol, racing performs the tie-break in the entry protocol with no additional code in the exit
protocol. There are two kinds of races: read and write.

7.2.1. Read race. Peterson–Fischer’s 2-thread algorithm [4, p. 92] is the first algorithm to use a
read race to prevent livelock and starvation. The following intuition is given for the algorithm:

The general idea is for process P0 to set its state variable the same as process P1’s and wait for P1’s
to be opposite before entering. P1 on the other hand sets its state unequal and waits for equality. [4, p. 92]

However, Peterson-Fischer’s algorithm is complex and difficult to follow.

Figure 4(a) shows Kessels’ version of this algorithm [5, p. 137], which replaces Peterson-
Fischer’s complex if expressions with an arithmetic formula. The mechanism to prevent livelock
and starvation is still subtle. Assume threads are 0 and 1. At line 1, a thread races to read the other

```c
volatile int Q[2] = { 0, 0 }; R[2];
#define inv( c ) ((c + 1) % 2)
#define plus( a, b ) ((a + b) % 2)

1 Q[id] = 1;
2 R[id] = plus( R[inv(id)], id );
3 Fence();
4 while ( Q[inv(id)] == 1 &&
5 R[id] == plus( R[inv(id)], id ) ) Pause();
6 CriticalSection();
7 Q[id] = 0;
```

```
enum Intent { DontWantIn, WantIn };
volatile int intents[2] = { DontWantIn, DontWantIn }, last;

intents[id] = WantIn; // entry protocol
last = id; // RACE
Fence();
while ( intents[1 - id] == WantIn &&
last == id ) Pause();
CriticalSection();
intents[id] = DontWantIn; // exit protocol
```

(a) Kessels Read Race

(b) Peterson Write Race

Figure 4. Race tie-breaking algorithms.
A write race is another common mechanism used to deal with livelock
and starvation and appeared first in Dijkstra’s software solution for
2-thread mutual exclusion (Section 8). However, the most popular occurrence of a write race is
Peterson’s 2-thread solution. Both Peterson–Fischer and Kessels’ algorithm are RW-unsafe. Model checking with Spin [17] gives mutual exclusion violations when assignments to shared variables are made to flicker.

7.2.2. Write race. A write race is another common mechanism used to deal with livelock
and starvation and appeared first in Dijkstra’s software solution for N-thread mutual exclusion
(Section 8). However, the most popular occurrence of a write race is Peterson’s other 2-thread solution [6, Figure 1] for mutual exclusion (see Figure 4(b) for a simplified version). Peterson’s 2-thread write race starts with the declare-intent program (Figure 1(a)) and makes two additions to the entry protocol. If both threads arrive simultaneously, they both indicate their intent to enter, and they both race to put their id into  last . Because the assignment occurs atomically, one of the values is overwritten. The race winner is the first assigner (value overwritten), that is, the value in last is the race loser. Now, each thread enters the busy loop, checking first if the other thread even wants in, and if it does, who won the race. The latter check breaks any ties, and livelock cannot occur.

Racing relies on the underlying hardware for atomicity and fairness on simultaneous arrival. Essentially, when two threads arrive simultaneously, a coin is flipped (independent trial). If the coin toss is unfair across many simultaneous arrivals, one thread receives some amount of unfairness. In contrast, alternation retains a history across simultaneous arrival (the thread last in the CS), so threads alternate winning at each simultaneous arrival (dependent trails). Given that simultaneous arrivals are rare, this point is unlikely to generate any practical effects.

Starvation cannot occur in the read or write race for the following reason. If a thread exits the CS, and tries to barge ahead of a waiting thread, it must first indicate its intent and run a race with itself in the entry protocol (the other thread is still in the busy loop). In this case, the thread is the race loser, so it waits. The other thread, which is waiting in the busy loop, detects it has won the previous race, that is, this thread has now become the high-priority thread. Therefore, it makes progress and enters the CS. Thus, a thread cannot win two races in a row, and hence, the threads alternate if there is contention.

Copyright © 2014 John Wiley & Sons, Ltd.
DOI: 10.1002/cpe
In Peterson’s algorithms, the race loser does not retract intent, so if the thread in the CS attempts reentry, it immediately excludes itself from reentering. Therefore, there is bounded overtaking regardless of how long it takes the other thread to complete the entry protocol and enter the CS. On the surface, this situation seems to violate rule 3, that is, preventing access to the CS if unoccupied, but the prevention is occurring in the entry protocol, which is allowed by rule 3.

7.2.3. Atomicity. All software algorithms for mutual exclusion use one or more of these approaches to guarantee rule 1 of mutual exclusion (i.e., only thread in the CS) and to prevent livelock and starvation. Clearly, algorithms using a write race are RW-unsafe because they require the underlying hardware to ensure simultaneous writes do not scramble bits (see end of Section 4). However, for algorithms using only alternation and/or read race, RW-safety is more difficult to decide as it depends on how the reading of flickering values affects the algorithm.

8. DIJKSTRA

Figure 5 shows the first software solution for the $N$-thread mutual exclusion problem by Edsger W. Dijkstra [9, p. 569]. Threads are numbered from 1 to $N$; a non-existent thread $T_0$ never wants into the CS (arrays are dimensioned $0..N$). Thread $T_i$ declares its intent to enter the CS by setting $b_{id}$ to 1 (wantin), and sets $c_{id}$ to 1 to start a write race with other threads wanting entry, where variable $\text{turn}$ holds the result of a race. If $T_i$ is not equal to $T_{\text{turn}}$, it implies a thread is attempting entry to the CS or is in the CS. Therefore, $T_i$ waits for $T_{\text{turn}}$ to enter or leave the CS by busy-waiting until $T_{\text{turn}}$ retracts its intent $b_{\text{turn}}$. After which, all threads at this point in the entry protocol race to set $\text{turn}$ and reset $c_{id}$ to indicate their race is finished. Each of these threads scans all threads (excluding self), and if any other thread has also finished the race, the thread restarts the entry protocol to check which thread is the race winner. A thread loses the race if its id is overwritten, and a losing thread waits for the winning thread, as no thread in the entry protocol retracts its intent on a restart. Eventually, all racing threads set $\text{turn}$, and the last one to set turn is the winner as $T_{\text{turn}}$ equals $T_i$. The other threads either do not want in or are in the entry protocol waiting for the current race winner to exit the CS by retracting its intent. If the race executes at the fastest possible speed, all threads simultaneously assign to turn and the hardware (atomic assignment) ensures one value is present (winner). If the race executes at the slowest possible speed, each thread in the entry protocol sees the next thread as the winner, one after the other, as $\text{turn}$ changes. None of the losers can

```c
volatile int turn = 0, b[N + 1] = { 1, ... }; c[N + 1] = { 1, ... };
1   b[id] = 0; // entry protocol
2   L: c[id] = 1; // start racing
3   Fence();
4   if ( turn != id ) {
      // in use ?
5       while ( b[turn] != 1 ) Pause(); // busy wait
6       turn = id; // RACE
7       Fence();
8   }
9   c[id] = 0; // finish racing
10  Fence();
11  for ( int j = 1; j <= N; j += 1 ) // check for racers
12     if ( j != id && c[j] == 0 ) goto L; // someone racing ?
13  CriticalSection( id );
14  turn = 0; // (optional) increases fairness
15  c[id] = b[id] = 1; // exit protocol
```

Figure 5. Dijkstra.

The version published earlier in [3, p. 61] has turn = 0 in the exit protocol. If turn is not reset, the last thread in the CS has an advantage on reentering. As a result, the algorithm is slower because it forces more competition because of fairness. Nevertheless, we decided to test the slower version because of its fairness.
make progress because the other racers still have \( c_{id} \) set, which is not retracted until the race is over and each thread has rechecked the winner by looping back to label \( L \). This second loop in the algorithm guarantees mutual exclusion because it maintains the invariant that, for every pair of different threads \( p \) and \( q \), if they are in lines 9–12, \( c[p] \) has yet to be inspected by \( q \) or \( c[q] \) has yet to be inspected by \( p \). Consequently, at least one of the two is not yet in line 13 (CriticalSection). When \( T_i \) exits the CS, the intent, \( b_{id} \), and racing flag, \( c_{id} \), are reset. The algorithm uses a minimum of \( 2N \) bits for arrays \( b \) and \( c \), and \( \log N \) bits for variable \( \text{turn} \); an implementation uses \( 2N + 3 \) words.

Knuth points out Dijkstra’s algorithm has the potential for starvation (breaks rule 5) [11, p. 321]. If the thread exiting the CS immediately attempts reentry, it joins any threads in the entry protocol and can be the last thread to set \( \text{turn} \) so it wins the race. This situation can happen an unbounded number of times preventing other threads from entering the CS.

9. KNUTH/DE BRUIJN

Figure 6 shows an \( N \)-thread solution by Knuth [11, p. 321], which uses cyclic scanning (alternation), rather than racing. As well, Knuth combined Dijkstra’s \( b \) and \( c \) arrays into a single control array with three values: \( \text{DontWantIn} \) (\( D \)), \( \text{WantIn} \) (\( W \)), and \( \text{EnterCS} \) (\( E \)). Knuth’s algorithm cycles in descending order, which is different from other algorithms, and needs to be noted during the discussion.

The entry protocol has two steps. In the first step, a thread begins by setting its intent and then scanning in a cycle from \( \text{turn} \) (one less than the last thread to use the CS in cyclic order) to its \( \text{id} \) for no threads wanting into the CS (Don\( \text{tWantIn} \)) in that range; otherwise, it restarts the scan. For maximal contention, all intents are set; hence, step 1 (lines 3–4) only allows thread \( T_{\text{turn}} \) to progress as all other threads starting at position \( \text{turn} \) see a control value of \( \text{WantIn} \) for \( T_{\text{turn}} \) and restart their scan. In this case, threads execute in cyclic order, so the bound on service is \( N/\text{NUL} \), that is, the number of threads ahead in the cycle. For less than maximal contention, there is a race (around the cycle) from \( \text{turn} \) to \( \text{id} \), through \( \text{DontWantIn} \) threads. Therefore, up to \( N - 1 \) threads can proceed to the next step. For example, assume \( T = 4 \), \( \text{turn} \) is 3, and all intents are \( \text{DontWantIn} \) (e.g., a starting scenario):

<table>
<thead>
<tr>
<th>intents</th>
<th>exec order</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>T1</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>W</td>
<td>D</td>
</tr>
<tr>
<td>W</td>
<td>W</td>
</tr>
</tbody>
</table>

```
enum intent { DontWantIn, WantIn, EnterCS };  
volatile int turn, control[N] = { DontWantIn, \ldots };  
  L0: control[id] = WantIn;  
  // entry protocol  
  Fence();  
  L1: for ( j = turn; j != id; j = cycleDown(j, N) )  
    if ( control[j] != DontWantIn ) { Pause(); goto L1; } // restart search  
  L2: control[id] = EnterCS;  
  Fence();  
  for ( j = N - 1; j >= 0; j -= 1 )  
    if ( j != id && control[j] == EnterCS ) goto L0;  
  // turn = id;  
  CriticalSection( id );  
  turn = cycleDown(id, N);  
  // exit protocol  
  L4: control[id] = DontWantIn;  
  L5: ;
```

Figure 6. Knuth.
T₀ arrives and sets its intent, cycles from element 3 to 1 in control only seeing Don’tWantIn and enters step 2. T₁ and T₂ follow in that order, cycling from element 3 to id-1 in control only seeing Don’tWantIn and enter step 2. T₃ does not arrive yet.

As in Dijkstra’s algorithm, the second step of the entry protocol provides the mutual exclusion. A thread sets its intent to EnterCS and checks if any thread (excluding self) is also in this step. If so, the entry protocol is restarted, resetting intent from EnterCS to WantIn, which narrows the selection because intents are not retracted (Don’tWantIn). The selection narrows because on every restart, step 1 only lets one thread continue to step 2 among those returning from the second step. The continuing thread is the one having the minimal distance from turn. For example, T₀ and T₁ see T₂’s WantIn after the restart and delay, while T₂ continues to step 2 to compete with any newly arriving threads, such as T₃, that might have gotten through step 1 during this process because turn is still 3. Eventually, a thread is selected and enters the CS. Note, line 9 before entering the CS does not appear to provide any benefits and is commented out for this paper. At best, changing turn before the selected thread enters the CS only affects the execution of the waiting threads in the entry protocol as they cannot make progress. All these threads quickly move to step 1 and spin there because T₄ turn has its intent set to EnterCS. For high contention, turn is almost always equal to id at line 9. Finally, the thread exiting the CS resets turn to a different value, so there is little purpose in setting it before entry.

Knuth states the maximum delay-bound for any thread entering the CS is $2^{N-1} - 1$; however, only an informal outline is given for a scenario that generates the bound, from which the reader must infer a proof (‘k’ is turn):

For example, if $N = 4$ suppose computer 4 is at L1 and computers 1, 2, 3 are at L2. Then,

(i) computer 1 goes (at high speed) from L2 to L5;
(ii) computer 2 goes from L2 to L5, then computer 1 goes from L5 to L2;
(iii) computer 1 goes from L2 to L5;
(iv) computer 3 goes from L2 to L5, then computer 1 goes from L5 to L2, then computer 2 goes from L5 to L2;
(v) (vi), (vii) like (i), (ii), (iii), respectively;

meanwhile computer 4 has been unfortunate enough to miss the momentary values of k which would enable it to get through to L2. [11, p. 322]

Notice, cycling back at step (v) to restart at (i), computer 3 is still at position L5, and it is impossible for computer 3 to get to position L2 because there is no way for turn to equal 3 without executing computer 4, and there is always a WantIn in intents[4]; hence, computer 3 must wait at step 1. Effectively, computer 3 drops out, and the cycle continues with computers 1 and 2, until 2 drops out, and then computer 4 makes progress. Figure 7 shows the complete expansion of Knuth’s delay-sequence for 4 threads; the dash line indicates where Knuth’s outline stopped. Therefore, T₀ enters after threads 1, 2, 1, 3, 1, 2, 1 precede it. In general, the sequence of passing threads is

<table>
<thead>
<tr>
<th>N passing threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0</td>
</tr>
<tr>
<td>2 1 0</td>
</tr>
<tr>
<td>3 1 2 1 0</td>
</tr>
<tr>
<td>4 1 2 1 3 1 2 1 0</td>
</tr>
<tr>
<td>5 1 2 1 3 1 2 1 4 1 2 1 3 1 2 1 0</td>
</tr>
<tr>
<td>6 1 2 1 3 1 2 1 4 1 2 1 3 1 2 1 5 1 2 1 3 1 2 1 4 1 2 1 3 1 2 1 0</td>
</tr>
</tbody>
</table>

Hence, $T_{N-1}$ passes T₀ once, $T_{N-2}$ passes twice, $T_{N-1-k}$ passes $2^k$ times. Therefore, the bound on the numbers of threads that can pass T₀ before it executes can be as great as $\sum_{k=0}^{N-2} 2^k = 2^{N-1} - 1$. Interestingly, the earlier sequence of numbers is the same as the sequence of disk numbers moved in a solution of the classical problem of the towers of Hanoi. Note, substrings of this sequence or the complete sequence are rare, and therefore, even small delays for any thread are unlikely. Because Knuth’s algorithm provides a bound, it is the first correct algorithm for $N$-Thread mutual exclusion.
De Bruijn [18, p. 137] changed Knuth’s algorithm by removing line 9 (as we did, too) and replacing line 12 with two lines:

```c
if (control[turn] == DontWantIn || turn == id)
    turn = cycleDown(turn, N); // absolute cycle
```

The key change is from a relative to an absolute cycle, that is, `id` is changed to `turn`. As a result, `turn` cannot `jump around` based on the id of the thread exiting the CS; instead, `turn` cycles unidirectionally around the intents based on the last id in the CS (like Dekker), which precludes the delay pattern in Knuth’s algorithm. As a result, the maximum delay-bound for any thread entering the CS is $N(N - 1)/2$.

However, the decrement of `turn` must be conditional. If it is unconditionally, waiting threads can be skipped during the cycle resulting in starvation. For example, $N = 3$, `turn = 2`, and using Figure 6 with line 11 changed to `turn = cycleDown(turn,N)`, assume $T_0$ executes L0 and repeat the sequence:

- $T_1$ executes lines L0–L2.
- $T_2$ executes lines L0–L2.
- $T_1$ executes lines L2–L5, changing `turn` to 1.
- $T_1$ executes lines L0–L2.
- $T_1$ executes lines L2–L5, changing `turn` to 0.
- $T_2$ executes lines L2–L5, changing `turn` to 2.

During this loop, $T_0$ jumps back to L1 because `control[from turn down to 0] != DontWantIn` (actually, $T_0$ may enter the loop at line 7, provided it jumps back to L0 and is not observed by the other threads). The problem is the two threads cycle `turn` passed the waiting thread $T_0$.

This scenario is prevented by the `if` statement, which only advances `turn` if the thread associated with the current turn does not want in. (This includes the thread exiting the CS, which no longer wants in, but it cannot retract its intent until after `turn` is advanced or there is a race with an entering thread, so there is a special case in the `if` statement.) Hence, $T_2$ does not advance `turn` because `control[0]` wants in, and the cycle earlier is broken. Therefore, starvation cannot occur because the waiting threads are examined in cyclic order, so each thread eventually gets a change to execute but in cyclic order rather than FCFS.
Like Knuth, De Bruijn left the proof that the upper bound $N(N-1)/2$ can actually occur as an exercise for the reader. Therefore, a proof is presented. Assume $\text{turn} = N-1$ and $T_0$ executes L0 so $\text{control}[0] == \text{WantIn}$:

```
    enum Intent { Don'tWantIn, WantIn, EnterCS };
    volatile int HIGH = 0, control[N] = { Don'tWantIn, ... };

L0: control[id] = WantIn;               // entry protocol
    Fence();
L1: for ( j = HIGH; j != id; j = cycleUp(j, N) )
    if ( control[j] != Don'tWantIn ) { Pause(); goto L1; } // restart search
    control[id] = EnterCS;
    Fence();
    for ( j = 0; j < N; j += 1 )
        if ( j != id & control[j] == EnterCS ) goto L0;     // retry
    if ( control[HIGH] != Don'tWantIn & HIGH != id ) goto L0; // retry
    HIGH = id;
    CriticalSection( id );
    for ( j = cycleUp( HIGH + 1, N ); j = cycleUp( j, N ) ) // exit protocol
        if ( control[j] != Don'tWantIn ) { HIGH = j; break; }
    control[id] = Don'tWantIn;
```

During this loop, $T_0$ jumps back to L1 because $\text{control[from turn downto id]} != \text{Don'tWant}$. After the repeat loop, $\text{turn} == 0$ and $T_0$ executes lines L2–L5, decrements turn back to N–1, and goes back to L0. In each iteration of the repeat loop, the number of threads entering the CS before $T_0$ is one less because turn is decremented. Therefore, the total number of threads entering the CS before $T_0$ is $(N-1) + (N-2) + ... + 1 = N(N-1)/2$. Both algorithms are RW-unsafe. Assume the assignment $\text{intents}[id] = \text{WantIn}$ (line 1) flickers to $\text{EnterCS}$, then a winning thread in the second loop (lines 7, 8) can read $\text{intents}[id] == \text{EnterCS}$ and restart. As a result, progress is no longer guaranteed.\

Note, both the Knuth and De Bruijn algorithms and their analysis appear in Letters-To-The-Editor and hence are not complete refereed papers.

10. EISENBERG–MCGUIRE

Figure 8 shows an $N$-thread solution by Eisenberg and McGuire [19] that also cycles (alternation) through waiting threads to establish a bound. This solution is similar to prioritize retract intent (Section 6.2) with a variation on how the priorities work. Instead of having fixed priorities assigned to the array positions, the priorities rotate around the array. The thread currently in the CS or last in the CS always has the highest priority. The threads waiting to enter, in the clockwise direction from the highest priority, have decreasing priority levels. When the thread in the CS exits, it rotates the priorities clockwise to the next thread attempting entry, making it the highest priority thread. If no threads are attempting entry, the priorities are not rotated. This algorithm uses three states ($\text{Don'tWantIn, WantIn, and EnterCS}$) to indicate intention and $N$ intents plus an integer variable.

The entry protocol has two steps. In the first step, a thread at position $i$ begins by setting its intent and then linearly searching clockwise from position $\text{HIGH}$, that is, the position of the current/last user of the CS, to itself for any thread that wants into the CS. If such a thread is found, for example,
that thread has higher priority, so the searching thread restarts the search but without retracting its intent, as in

Only after a thread has searched from position HIGH to its position and found no thread waiting to enter does it move to the second step. In essence, a thread sees a moment in time when there are no higher priority threads between HIGH and its position.

In the second step of the entry protocol, a thread sets its intent to state EnterCS indicating it has completed the first step of the entry protocol. It then linearly searches all other threads (excluding self) for any that have also reached the second step of the entry protocol and hence have intent set to EnterCS, as in

Multiple EnterCS states can occur when a higher priority thread arrives after its position has been examined by a lower priority thread in step 1. In this case, all threads restart to find the new thread that is closest to HIGH. Threads drop out of this cycle by waiting in step 1 until only the closest thread to HIGH progresses to step 2. Finally, the thread closest to HIGH must restart if thread_{HIGH} is still in the CS (excluding self). When all the conditions become true, the thread rotates the priorities so that it is now the highest priority thread, that is, HIGH = id.

The difficult case is when no thread is in the CS, and HIGH is at an arbitrary location when threads arrive, for example, the initial condition when the threads start. This situation is the reason for step 2 in the entry protocol. In this case, the entering threads race to become the next thread to enter, and the previous value of HIGH is used to control the selection. As earlier, the thread minimal distance from the previous value of HIGH exits step 1 first because it finds no intent set for higher priority threads; the other threads wait for it. However, there can be many false starts by the waiting threads as other threads arrive with different priorities. For example, if threads arrive from low to high priority, all the threads can simultaneously get to step 2 of the algorithm, as the low-priority threads miss seeing the later arriving higher priority threads in step 1. This situation results in all of the threads restarting the entry protocol. The arrival of a higher priority thread, including the one at the current HIGH position, can force this cycle to start over again at any time. Hence, the selection process can be expensive, when there is no thread in the CS and there are simultaneous arrivals.

In the exit protocol, the exiting thread searches in priority order for any thread that wants in other than itself. If a thread is found, that thread is made the highest priority, in effect rotating the priorities; otherwise, HIGH remains unchanged. Then, the exiting thread retracts its intent.

The waiting bound is N - 1 because a thread may have to wait for all the other threads to enter and leave the CS before it can proceed. The bound is ensured because rotating the priorities in a fixed direction in the exit protocol eventually services all waiting threads. However, arriving threads are not serviced in FCFS order but in cyclic order. A thread arriving after the current
HIGH position is serviced only when threads before the current HIGH position are serviced, even though thread’s after the current HIGH position may have arrived prior to thread’s before it. Therefore, this algorithm, like Knuth/De Bruijn, is an example of bounded but non-FCFS selection. As well, this algorithm is RW-unsafe for the same reason as Knuth/De Bruijn: if the first assignment \texttt{ints[id] = WantIn} flickers to \texttt{EnterCS}, then a winning thread in the second loop can read \texttt{ints[id] = = EnterCS and restart}, so progress is no longer guaranteed.

11. LAMPORT BAKERY

Figure 9 shows Lamport’s [20, p. 454] RW-safe \(N\)-thread solution, which involves taking a ticket on arrival and waiting until the ticket is chosen for service (complex read-race). The \texttt{ticket} array is initialized to 0, and ticket 0 is not used and means do not want in. Tickets are serviced in increasing order by ticket number, so threads are serviced in FCFS order (in contrast to cyclic order in the previous algorithms). In fact, this algorithm is often referred to as the bakery algorithm, after taking a ticket for service in a bakery. However, taking a ticket is quite different from that used by people in a store (see later in the text), and there is an auxiliary array indicating when a thread is choosing a ticket. For example, in

\[
\begin{array}{ccccccccccc}
\text{HIGH priority} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\text{ticket} & 0 & 0 & 17 & 0 & 0 & 18 & 18 & 0 & 20 & 19 \\
\text{choosing} & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1
\end{array}
\]

threads 0, 1, 3, 4, 7 do not want in, threads 2, 5, 6, 8 have selected tickets, and threads 7, 9 are choosing a ticket.

The entry protocol has two steps. In the first step, thread \(T_i\) computes a ticket and stores it at position \(i\) in the shared array \texttt{ticket}. However, selecting a ticket is performed in an unusual manner. Instead of selecting a ticket from a machine that atomically generates increasing values, a thread computes a ticket from the current ticket values held by the waiting threads. A thread finds the maximum ticket value of the waiting threads and creates a new ticket value one greater, requiring an \(O(N)\) search.

```c
volatile int choosing[N] = { 0, ... }, ticket[N] = { 0, ... };

// step 1, compute a ticket
choosing[id] = 1;  // entry protocol
Fence();
int max = 0;       // O(N) search for largest ticket
for ( int j = 0; j < N; j += 1 ) {
    int v = ticket[j];
    if ( max < v ) max = v;
}
max += 1;          // advance ticket
ticket[id] = max;
choosing[id] = 0;
Fence();

// step 2, wait for ticket to be selected
for ( int j = 0; j < N; j += 1 ) {
    while ( choosing[j] == 1 ) Pause();       // check other tickets
    while ( ticket[j] != 0 &&
        ( ticket[j] < max ||
        ( ticket[j] == max && j < id ) ) ) Pause();
}
CriticalSection( id );
ticket[id] = 0;      // exit protocol
```

Figure 9. Lamport (bakery algorithm I).

Unfortunately, this procedure does not ensure a unique ticket value. Two (or more) threads can simultaneously compute the same maximum ticket value and add one to it, resulting in multiple tickets with the same value. While the chance of this problem occurring is probabilistically low, it still must be dealt with. The solution involves the same technique used for the \(N\)-thread prioritize retract intent solution (Section 6.2), which is to assign a priority to each position in the intent array. When two threads have the same ticket value, their position in the intent array is used to break the tie. As a result, it is possible for \(T_i\) to arrive ahead of \(T_j\) and begin computing its ticket but be serviced after \(T_j\) because both threads generate identical tickets because of different rates of progress during the ticket computation. Such behavior at (essentially) simultaneous arrival occurs for all algorithms striving for FIFO servicing.

The strongest fairness condition that can be satisfied is the following FCFS condition. We assume that the \textit{trying} statement (entry protocol) consists of two substatements – a \textit{doorway} whose execution requires only a bounded number of elementary operation executions (and hence always terminates), followed by a \textit{waiting} statement. We can require that, if process \(i\) finishes executing its \textit{doorway} statement before process \(j\) begins executing its \textit{doorway} statement, then \(i\) must execute its critical section before \(j\) does. [14, pp. 330]

In the second step of the entry protocol, a thread linearly searches through all the waiting threads (including self), ignoring threads that do not want in (0 ticket value), and waits for any thread with higher priority, because it has a lower ticket value or lower subscript for equal tickets. However, there is a problem if two (or more) threads compute the same ticket value but the lower priority thread advances to the second step before the higher priority thread assigns its equal ticket value into the ticket array, for example, it is interrupted just before the assignment. When the lower priority thread checks the ticket array, it finds no thread with an equal ticket value, as the other thread has not performed its assignment; therefore, it proceeds into the CS. When the higher priority thread checks the ticket array, it finds a lower priority thread with the same ticket value but higher subscript so it proceeds into the CS, violating mutual exclusion.

To solve this problem, it is crucial to delay checking a thread’s ticket value if it is computing a ticket. This information is provided by the \textit{choosing} array during ticket selection. A thread sets its position in this array before starting to compute a ticket and resets it only after assigning its new ticket. The search in the second step is now composed of two checks. First, determine if a thread is computing a ticket, and if so, wait for the ticket to be assigned. Second, check if the thread wants in, and if so, determine if it is higher priority. Now, a lower priority thread knows if a higher priority thread is computing a ticket and waits for the new ticket value to appear. When the ticket value appears, it is either less, equal, or greater than the checking thread’s ticket, and it proceeds or waits appropriately.

This algorithm uses \(N\) binary states for the choosing arrays, plus \(B\) bits (usually the size of an integer), for the \(N\) intention states (tickets); the implementation uses \(2N\) words.

The problem with this and any other approach based on tickets is that the ticket values cannot increase indefinitely because of the finite size of the integer type, say \(2^{32}\) values. In the worst case, a ticket value overflows back to zero (or a negative value), which results in multiple threads entering the CS simultaneously, that is, a mutual exclusion violation. However, in these bakery algorithms, whenever the CS becomes unoccupied, that is, no thread is using it or waiting to use it, the maximal ticket value resets because the next thread selects ticket 1. The ticket values only increase when threads arrive and must wait for entry to the CS. For example, this failure occurs when \(2^{32}\) threads arrive at the CS while it is being used, which is highly unlikely. Even two threads can overflow the tickets by perfectly alternating \(2^{32}\) times so one arrives at the CS while the other is using it; again, this scenario is equally unlikely. Therefore, this and the next ticket algorithm are probabilistically correct because the chance of ticket overflow is extremely small.

12. HEHNER–SHYAMASUNDAR

Figure 10 shows Hehner and Shyamasundar’s [21, pp. 196–197] simplification of Lamport’s algorithm by folding the \textit{choosing} array into the ticket array. In essence, two ticket values are set.
volatile int ticket[N] = { INT_MAX, ... };

// step 1, compute a ticket
int ticket[id] = 0; // set highest priority
Fence();
int max = 0; // O(N) search for largest ticket
for ( int j = 0; j < N; j += 1 ) {
    int v = ticket[j]; // could change so copy
    if ( v != INT_MAX && max < v ) max = v;
}
max += 1; // advance ticket
ticket[id] = max;
Fence();

// step 2, wait for ticket to be selected
for ( int j = 0; j < N; j += 1 ) // check other tickets
    while ( ticket[j] < max || // busy wait
        ( ticket[j] == max && j < id ) ) Pause();
CriticalSection( id );
ticket[id] = INT_MAX; // exit protocol

Figure 10. Hehner and Shyamasundar (bakery algorithm II).

Aside (instead of one) to indicate either a thread does not want into the CS or it is computing a ticket. A non-waiting thread is identified by the maximum possible ticket value, INT_MAX (∞), implying lowest priority, and all elements of the ticket array are initialized to this maximum value. Computing a ticket is identified by the minimum ticket value, implying highest priority.

For both ticket algorithms, there is no indefinite postponement (violation of rule 4), as the ticket priority, or alternatively, the priority of a thread’s position in the intent array ensures that on simultaneous arrival a tie is always broken. There is no starvation (violation of rule 5) because the ticket values are always increasing as long as there are waiting threads. Hence, if a thread exits the CS and tries to rush in again ahead of other waiting threads, it must generate a ticket value greater than any waiting thread because it examines all their ticket values. Because no thread retracts its intent in the entry protocol, this new maximal ticket value ensures the arriving thread cannot enter before any of the other waiting threads and establishes its bound with respect to any waiting threads, which is FCFS. However, while Lamport’s bakery algorithm is RW-safe, Hehner–Shyamasundar’s version is RW-unsafe because the assignment of max to the ticket can flicker so a reader may observe a value greater than max and incorrectly assume the thread has lower priority or is not selecting a ticket.

13. BURNS

Burns presents 4 \( N \)-thread software solution algorithms. The first algorithm [22, Figure 1, p. 170] is a simplification of Dijkstra’s algorithm collapsing the binary \( b \) and \( c \) arrays into a single array with three-state values, as performed in Knuth’s algorithm (Section 9).
volatile int flag[N] = { 0, ... }, turn = 0;

1    L0: flag[id] = 1;  // entry protocol
2    turn = id;  // RACE
3    Fence();
4    L1: if ( turn != id ) { // race losers
5       flag[id] = 0;  // retract intent
6       Fence();
7       L11: for ( j = 0; j < N; j += 1 )
8          if ( j != id && flag[j] != 0 ) {
9             Pause();
10           goto L11; // original: goto L1
11      }
12    goto L0;  // restart
13  } else {
14     // flag[id] = 1;
15     // Fence();
16    L2: if ( turn != id ) goto L1; // still winner ?
17       for ( j = 0; j < N; j += 1 )
18          if ( j != id && flag[j] != 0 ) goto L2;
19     }
20    CriticalSection( id );
21    flag[id] = 0;  // exit protocol

Figure 11. Burns Second $N$-thread algorithms.

The second algorithm [22, Figure 2, p. 170] appears in Figure 11 and uses only a two-state value as it is unnecessary to distinguish the particular location in the entry protocol. Two simplifications are made to this algorithm for the paper. First, line 9 in the original algorithm branches to line 4 (\texttt{goto L1}). However, it is unnecessary to branch to line 4 because if variable \texttt{turn} is not equal to \texttt{id} on the first entry at line 4, it is always not equal because there is no subsequent assignment by this thread to reset \texttt{turn} to its \texttt{id}. The same argument applies to setting \texttt{flag[id]} to 0, as it is not reset during the \texttt{then} clause. Hence, it is sufficient to restart the scan at line 7, so the new label \texttt{L11} is added and used as the target of the branch instead of label \texttt{L1}. Second, line 14 is superfluous because on entry to the \texttt{else} clause of the outer \texttt{if} statement, \texttt{flag[id]} has just been set to 1 on line 1. These changes produce a significant performance improvement over the original algorithm largely because two fence instructions are removed from the main scan-loops.

After declaring intent to enter, a thread performs a write race with other threads by assigning to variable \texttt{turn} (RW-unsafe). If a thread loses the race, it retracts its intent (Section 6) and scans until all other threads either finished racing (denoted by all intents being retracted) or one exits the CS (and retracts its intent). At this point, the thread restarts the entry protocol with another race. If a thread wins the race, it does not retract its intent and scans until it loses the race (another thread becomes the winner), or all other threads finish racing and it becomes the winner, or a thread exits the CS. At this point, the thread restarts the entry protocol with another race. If there is no new race winner, the other racers or thread in the CS eventually retract their intents allowing this thread to enter the CS.

However, this algorithm allows starvation. A new thread can arrive, race, and become the winner among a group of threads attempting entry. That is, there is no prioritization established among entering threads. This situation can happen an unbounded number of times preventing other threads from entering the CS.

The third algorithm [22, Figure 3, p. 171] is just a canonical restructuring of algorithm 2 to make it uniform for all threads.

The fourth algorithm [22, Figure 4, p. 174] is an attempt to remove starvation using only $N$ shared variables. However, we were unable to generate a working implementation for this algorithm. Model checking with Spin shows mutual exclusion failures even for $N = 2$, and unproductive spinning loops through the label \texttt{L2} for $N = 3$. It may be possible to fix the algorithm, but that work is beyond the scope of this paper.
volatile int Q[N + 1] = { 0, ... }, turns[N];

for ( int r = 1; r < N; r += 1 ) { // entry protocol, round
    Q[id] = r; // intent, current round
    turns[r] = id; // RACE
    Fence();
    L: for ( int k = 1; k <= N; k += 1 ) // find loser
        if ( k != id && Q[k] >= r && turns[r] == id ) { Pause(); goto L; }
    CriticalSection( id ); // exit protocol
    Q[id] = 0;
}

Figure 12. Peterson.

14. PETERSON

Figure 12 shows Peterson’s $N$-thread [6, Figure 3, p. 116] extension of his 2-thread write race version (Figure 4(b)). The entry protocol has each thread execute $N - 1$ rounds. In each round, all participants indicate intent by setting their current round number, where 0 means do not want in, and compete by racing to assign to variable $turn$ for that round. The value in variable $turn$ denotes the loser, $L$, of the round. As assignments occur during the round, the loser changes allowing up to $r - 1$ threads to progress to the next round, but only if there are no threads in later rounds.

The latter restriction ensures a thread racing alone makes progress as it loses each round, but there are no threads ahead of it. However, the restriction also means as soon as a progressing thread performs assignment $Q[id] = x$ at the next round, threads at the previous round cannot progress even if they did not lose the round because they see threads ahead of them. As a result, more threads wait at the lower rounds than are allowed at the higher rounds. A consequence of this behavior is that threads spend more time in larger competitions, which is more expensive, and explains why this algorithm has poor performance. This algorithm prevents livelock and starvation because the write race prevents a tie on simultaneous arrival, and newly arriving threads always create a new loser so the current loser becomes a winner and makes progress to the next round.

15. LAMPORT–FAST

Figure 13 shows Lamport’s fast $N$-thread [23, Figure 2, p. 5] mutual exclusion algorithm. Fast means, in the absence of contention, a process executes only a constant number of reads and writes, that is, the number of steps is not a function of the number of processes. Threads are numbered from 1 to $N$. Variable $y$ contains thread id 0, if and only if the CS is empty. The first write race on variable $x$ (line 2) is used to determine if there are simultaneous contending threads attempting entry to the CS and break the tie. After the first race, variable $y$ is checked to determine if the CS is in use or simultaneous contending threads still are determining a winner for entry. If either case is true, a thread retracts its intent and waits for the CS to empty and then restarts. A second write race on variable $y$ (line 10) is necessary to determine the winner among the group of simultaneous contending threads and stop more threads from entering the contending set. After the second race, there are two cases:

direct winner: The last thread assigning to x enters the CS (winner), and the losing threads retract intent and wait for all contending threads to do the same, including the winning thread that entered the CS before restarting.

indirect winner: If all contending threads delay before line 12, and a new thread changes x but cannot join the contending threads because y != 0, then all contending threads are directed into the if statement at line 12. The contending threads wait until they are all in the if and have retracted their intent, then the winner of the second race enters the CS and the others restart. The await on line 18 is superfluous as transferring back to start immediately enters the same await.

When the winning thread exits the CS, it resets y and retracts its intent, so threads at either await restart. The minimum sequence of actions for no contention is the operations: write b[id], write x, read y, write y, read x, write y, write b[id], which Lamport shows is optimal for no speed bound (i.e., no spinning based on time). Finally, Lamport states this algorithm has starvation as a thread can restart and win an unbounded number of times.

16. SZYMANSKI

Figure 14 shows Szymanski’s N-thread [24, Figure 2, p. 624] mutual exclusion algorithm. Note, there is a typographical error in the original paper on line E0 in the algorithm: and should be or:

E0: wait until ∀ j<i: flag[j]<2 & flag[j]>3 // incorrect operation
E0: wait until ∀ j<i: flag[j]<2 | flag[j]>3 // correct operation

The algorithm is based on a metaphor of two doors into and out of a waiting room, and threads proceed in groups through the waiting room toward the CS. Each thread shares a value containing five states:

0 => do not want in
1 => intent to enter
2 => in waiting room
3 ⇒ passed through door 1
4 ⇒ passed through door 2

Arriving threads wait if either door is in use, that is, any thread is currently passing through a door, which is indicated by states 3 or 4. When the doors are closed, a thread may open the first door by setting its state to 3, and proceed into the waiting room. Then, the thread checks if other threads are intending to enter (state 1). If there are intending threads, a thread sets its state to 2, delays in the waiting room, which correspondingly opens door 1 for another thread to make progress. Eventually, the group of arriving threads drains, and the last thread in the group does not delay in the waiting room; it opens door 2 (sets its state to 4) and races toward the CS. Once door 2 is open, that is, there is one or more threads with state 4, threads in the waiting room stop their delay and also race toward the CS. At this point, door 2 stays open (multiple state 4s), precluding any arriving threads from joining this group as door 1 cannot open. However, there is a problem if threads in this group complete the CS and retract intent before all threads in the waiting room see door 2 is open. For example, if the last thread of a group resets its state from 4 to 0 before threads in the waiting room see the 4, none of the threads in the waiting room make progress in this group and would become part of the next group. As a result, there is the potential for starvation as a thread could wait forever in the waiting room. One possible solution is to have each thread wait until all threads in the waiting room drain state from 2 to 4:

```
for (int j = 0; j < N; j += 1) // wait for all threads in waiting room
    await (flag[j] < 2 || flag[j] > 3); // to pass through door 2
```

An alternate solution appears in the exit protocol. The threads that move forward now decide on entry to the CS using their thread id as a priority, where threads id 0 is the highest priority (as in Section 6.2). That is, each thread in the group scans each higher priority thread until it exits the CS and resets its state to either 0 (do not want in) or eventually to 1 (intent to enter). In the exit protocol, each thread performs the checks for threads in the waiting room, rather than before. There are two advantages: only threads with lower priority need to be checked because higher priority threads have been scanned before entering the CS, and in a parallel environment, threads in the waiting room can make progress to door 2 while another thread is in the CS.
This algorithm has two interesting properties. First, it is the only algorithm with waiting in the exit protocol. Although we verified the waiting can be moved into the entry protocol with no negative performance effect, so there is no reason to deviate from the practice of having all waiting before the CS. Second, it uses only three bits for the five states to accomplish mutual exclusion with a linear wait: no process can enter its CS twice while another process is waiting. Hence, threads are not serviced in FCFS order but rather in groups where the group is serviced linearly from low to high thread id with some overtaking. Finally, Szymanski proves this algorithm is RW-safe [24, § 5], if it is changed to use three separate bit flags; however, nothing is shown about RW-safety for the rather different case in the actual algorithm, where the flag is implemented as an integer with five states.

17. MINIMAL BITS FOR FIRST-COME FIRST-SERVE

The following algorithms address the question of the minimum number of bits necessary to provide all five rules for $N$-thread mutual exclusion (Section 3) and guarantee service is FCFS. (As stated, commodity computers are not bit addressable, so this discussion is largely theoretical.) Lycklama and Hadzilacos [25, Figure 4, p. 569] present a RW-safe FCFS algorithm requiring only five shared bits per thread: $c, v, x, turn[0..1]$. The Lycklama and Hadzilacos algorithm uses a complex mechanism to cycle through four values using pairs of bits 00, 01, 11, 10 in a matrix $\text{turn}$; Figure 15(a) shows our simplification of Lycklama and Hadzilacos replacing the pairs of bits by the values 0, 1, 2, respectively, and cycling through these values using modulo arithmetic. As a result, our version theoretically uses only 4.5 bits per thread: $c[id], v[id], intents[id]$ and 1.5 bits to store the three values in $\text{turn}$. However, this simplified algorithm is RW-unsafe as it combines multiple states in an intent. Hesselink [16, Figure 4, p. 11] builds on Lycklama and Hadzilacos maintaining the RW-safe and FCFS properties but reduces the number of bits to 4: $cc$ and $\text{turn}[0..2]$. Finally, Aravind [26, Figure 3] presents a RW-unsafe FCFS algorithm but reduces the number of bits to 3: $\text{intents}$ and 2 bits to store the four values in $\text{turn}$; Figure 15(b) shows our version of this algorithm (and the one used in subsequent performance experiments), moving $\text{turn}[id] = 0$ after the CS rather than before as in the original. Both versions are correct, but we achieve a 20–40% performance improvement in our version, which requires further work to understand. Aravind [26, Figure 4] also presents a RW-safe FCFS using 4 bits: $\text{intents}$, $x$, and 2 bits in $\text{turn}$. Szymanski [27] presents a 4-bit RW-safe FCFS algorithm, but we did not have time to study and implement it.

Whereas prior algorithms provide all five rules for mutual exclusion directly, these algorithms use an indirect approach. They require an algorithm, like B–L (Section 6.2) or Dijkstra (Section 8), to provide rules 1–4, and preface this algorithm with code to establish FCFS among the threads, providing rule 5. What is interesting about this class of algorithms is the mechanism to order $N$ threads with so few bits. To provide FCFS, it is necessary to prevent the thread exiting the CS from cycling back to the start and barging ahead of waiting threads. A shared array of waiting states is used to indicate which threads are attempting entry. To start the FCFS, each thread makes a local copy of all the shared waiting states and then sets its shared state indicating it is in the waiting phase. The local copy captures a moment in time with respect to all waiting threads. Another thread making a copy either sees a thread before or after it sets it is waiting state, which establishes a temporal ordering between them. Then, each thread compares its copy with the current shared states to determine if it can proceed.

The temporal ordering is approximate because of simultaneous arrivals. For example, assume the initial condition with $\text{turn}$ initialized to 0, and then all threads arrive simultaneously, copy the 0 values in $\text{turn}$, and set $\text{turn}[id]$ to indicate each is in the waiting phase; hence, all of the threads proceed because $\text{copy}_j \neq \text{turn}_j$, so there is no differentiation in the FCFS. The threads proceed to the next algorithm where an arbitrary ordering for execution is selected, which is reasonable because the threads arrived (essentially) simultaneously. However, this scenario cannot repeat until the CS becomes unoccupied (no threads wants entry). If any thread from this group attempts reentry while the scenario is being processed, it is temporally behind this thread group and waits, starting a temporal ordering.
The possible values for the waiting state are analyzed for Aravind in Figure 15(b), as Lycklama and Hadzilacos in Figure 15(a) is more complicated. In Aravind, \( \tau_{urn}[j] \) is not attempting to enter; \( \tau_{urn}[j] \) is reset to 0 in the exit protocol. If the copy for \( \tau_{urn}[j] \) is 0, the thread is not waiting in FCFS and can be ignored; otherwise, the thread has set its waiting indication.

If the copy for \( \tau_{urn}[j] \) is equal to the current value of \( \tau_{urn} \), there has been no temporal change by the other thread, implying it indicated its waiting state before the copy was made, and hence, it is temporally ahead. Therefore, the checking thread busy waits for the other thread to make progress. If the copy for \( \tau_{urn}[j] \) is not equal to the current value of \( \tau_{urn} \), there has been a temporal change by the other thread, implying it indicated its waiting state after the copy was made, and hence, it is temporally behind. Therefore, the checking thread advances to check the next waiting thread.

After all waiting threads are checked, it implies a moment in time when all waiting threads have temporally arrived after the checking thread, so it is at the head of the FCFS list and may proceed.

A single bit for the waiting state, with 0 meaning not waiting and 1 waiting, is insufficient as a third state is necessary to distinguish between arriving before or after a copy is made. Hence, a thread must cycle its waiting state between 1 and 2 on each entry so other threads can see a difference after the copy (the values are unimportant). However, two waiting states is still insufficient. Assume two threads, fast (T0) and slow (T1), the starting conditions, and intents[0] is irrelevant for T0. Figure 16 shows the sequence of steps illustrating a failure. At the last step in the sequence, both threads enter the FCFS loop and T0 finds \( \text{copy}[1] == \tau_{urn}[1] \), while T1 finds \( \text{copy}[0] == \tau_{urn}[0] \), so both spin waiting for the other thread to change \( \tau_{urn} \) and there is livelock. This failure means at least three distinct states are needed. Aravind gives an informal proof to show that three states are sufficient [26, p. 1032], while Hesselink [16, § 4] gives a formal proof of this for a slightly different algorithm.

Finally, all three algorithms have subtle interactions across the FCFS and B–L algorithm boundaries. Clearly, certain states must be set and held during FCFS and B–L for correctness, but the expectation is that they can be reset between the algorithms. For example, in Aravind, it would seem
Figure 16. FCFS failure with 2 bits.

<table>
<thead>
<tr>
<th>fast (T0)</th>
<th>slow (T1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-6 : turn[0] = 1</td>
<td>:</td>
</tr>
<tr>
<td>7-24 : CS (no contention)</td>
<td>:</td>
</tr>
<tr>
<td>:</td>
<td>2 : intents[1] = 1</td>
</tr>
<tr>
<td>:</td>
<td>3-5 : copy[0] = 1</td>
</tr>
<tr>
<td>25-26 : turn[0] = 0 (exit)</td>
<td>:</td>
</tr>
<tr>
<td>27-5 : copy[1] = 0</td>
<td>:</td>
</tr>
<tr>
<td>6 : turn[0] = 2</td>
<td>:</td>
</tr>
<tr>
<td>:</td>
<td>6 : turn[1] = 1</td>
</tr>
<tr>
<td>:</td>
<td>7 : intents[1] = 0 (allow T0 to progress)</td>
</tr>
<tr>
<td>7-24 : CS</td>
<td>:</td>
</tr>
<tr>
<td>25-26 : turn[0] = 0 (exit)</td>
<td>:</td>
</tr>
<tr>
<td>27-5 : copy[1] = 1</td>
<td>:</td>
</tr>
<tr>
<td>6 : turn[0] = 1 (only 2 states)</td>
<td>:</td>
</tr>
</tbody>
</table>

Figure 17. FCFS boundary failure.

<table>
<thead>
<tr>
<th>fast (T0)</th>
<th>slow (T1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-6 : turn[0] = 1</td>
<td>:</td>
</tr>
<tr>
<td>:</td>
<td>3-5 : copy[0] = 1</td>
</tr>
<tr>
<td>7-27-6 : turn[0] = 2 (cycle thru CS)</td>
<td>:</td>
</tr>
<tr>
<td>7-27-6 : turn[0] = 3 (cycle thru CS)</td>
<td>:</td>
</tr>
<tr>
<td>7-26 : turn[0] = 0 (exit)</td>
<td>:</td>
</tr>
<tr>
<td>:</td>
<td>6 : turn[1] = 1</td>
</tr>
<tr>
<td>27-5 : copy[1] = 1</td>
<td>:</td>
</tr>
<tr>
<td>6 : turn[0] = 1</td>
<td>:</td>
</tr>
</tbody>
</table>

turn is only needed in FCFS and intents only needed in B–L; however, intents brackets FCFS and turn is held across B–L. If the bracketing of the FCFS (doorway) is removed, that is, intents[1] == 0 throughout FCFS, then a failure occurs. Assume two threads, fast (T0) and slow (T1), and the starting conditions. Figure 17 shows the sequence of steps illustrating a failure. At the last step in the sequence, both threads enter the FCFS loop and T0 finds copy[1] == turn[1], while T1 finds copy[0] == turn[0], both spin waiting for the other thread to change turn, so there is livelock. The problem occurs because the fast thread rotates through its states, and hence, the CS, while the other thread is copying in FCFS. Holding intents[i] during copying, precludes multiple consecutive entries while a copy is in progress either in stage 1 or 2 of B–L.

18. TOURNAMENT

To simplify N-thread solutions and reduce the amount of scanning, a tournament approach is introduced. A tournament uses divide-and-conquer, where only one of every D threads progresses to the next round, and the others busy wait. A tournament solution is based on an auxiliary solution (MX) for D threads, where D is a small number > 1, usually D = 2. A D-ary tree is constructed with height \( \log_D N \), and each thread is assigned a starting position at one of the N leaves of the tree. To reach the CS, a thread repeatedly performs the entry protocol of MX, in competition with the siblings of its current node. When it wins the competition, it moves to the parent of its current node. When it reaches the root of the tree, it accesses the CS. After finishing the CS, a thread retraces its path from root to leaf (starting position) executing the exit protocols of MX at each node.
Because of discrete subdivisions \( (D) \), some mechanism must exist for non-powers of \( D \) players.

1. Create a minimal tree with only \( N \) start nodes so some paths from leaf to root are shorter than others.
2. Round \( N \) to the next power of \( D \) resulting in non-existent threads marked as not participating (do not want in). These extra entries must be tested but never cause contention.

Both approaches introduce unfairness because of the unbalanced competition, that is, some threads do not have to compete with as many threads. The performance results in Section 22 show the unfairness is small.

Figure 18(a) shows the maximal binary tree \( (D = 2) \) approach for \( N \) threads by rounding \( N \) to the next power of 2, and Figure 18(b) shows the minimal binary tree. For all trees, threads start at the leaves of the tree, and \( \lceil \log_2 N \rceil \) levels of internal nodes to implement the tournament. Each node is a 2-thread solution for mutual exclusion, such as Dekker, Kessels, or Peterson. Each thread is assigned to a particular leaf where it begins the mutual exclusion process. At each node, the 2-thread algorithm ensures an arriving thread is guaranteed to make progress because of rule 5 for the internal MX algorithm; that is, the loser at each node eventually becomes the winner and continues to the next level. Therefore, each thread eventually reaches the root of the tree and enters the CS. All tournament algorithms discussed allow unbounded overtaking because there is no synchronization among the nodes of the tree. That is, a fast thread can enter the CS arbitrarily often down one branch, while a slow thread is working down another branch. With a minimal binary tree, the tournament approach uses \( (N - 1)M \) bits, where \( (N - 1) \) is the number of tree nodes and \( M \) is the node size; if Peterson’s algorithm is used at each node, the node size can be three or four variables, for example, \( \text{turn} \), \( \text{intents}[2] \), and possibly the link fields to represent the tree.

In designing a tournament algorithm, there are five issues to be resolved:

1. choice of internal MX algorithm,
2. how to store/organize the shared variables for the MX,
3. how to relate the thread identity with the identity of its current node, and of the siblings and parent of the current node,
4. form of the tree, and
5. mechanism to retrace path from root to leaf (starting position) in the exit protocols.

Whether a tournament algorithm is RW-safe depends on the MX algorithm, as it communicates with opponents; the tree traversal is local to each thread. Note, for \( D > 2 \), one of the previous \( N \)-thread software solutions must be used in each of tree nodes (MX), versus one of the simple 2-thread solutions. For uniform memory access (UMA) architectures, the simple 2-thread solutions are usually the best approach, as there is no additional cost to access any of the shared data. For non-uniform memory access (NUMA) architectures, where access to shared state can increase significantly with

![Figure 18. N-thread tournament tree, N = 5, D = 2: (a) maximal and (b) minimal tree size.](image-url)
distance between CPUs, $D > 2$ can be more cache friendly as the $D$ group compete locally before moving to the next local node. A number of tournament algorithms are examined using different mechanisms to implement the five issues of a tournament algorithm.

18.1. Peterson–Fischer

Figure 19 shows Peterson–Fischer’s $N$-thread [4, p. 93] extension of their 2-thread read-race version (Section 7.2.1). This $N$-thread solution seems to be the first tournament algorithm:

A solution for $n$ processes is obtained by splitting them into two groups of $n/2$ processes each. In each group, the processes compete to enter a CS using recursively the solution for $n/2$ processes. The ‘winners’ of each group, that is the two processes which succeeded in entering their respective group’s CS, use the two process solution to determine which is allowed to enter the top-level CS. The groupings of the processes can be thought of as a tree with a process progressing from a leaf to the root, competing at each level of the tree against at most one of the processes in its brother’s subtree. [4, pp. 92–93]

```c
typedef union {
    uint32_t atom;  // ensure atomic assignment
    struct {
        uint16_t level;  // current level in tournament
        uint16_t state;  // intent to enter critical section
    } tuple;
} Tuple;

#define L(t) ((t).tuple.level)
#define R(s) ((s).tuple.state)
#define EQ(a, b) ((a).tuple.level == (b).tuple.level && (a).tuple.state == (b).tuple.state)
int bit( int i, int k ) { return ( i & (1 << (k - 1))) != 0; }  // return true if i is a winner
int min( int a, int b ) { return a < b ? a : b; }  // return true if i is a winner

int depth = ceil( log2( N ));  // maximal depth of binary tree
int width = 1 << depth;  // maximal width of binary tree
int mask = width - 1;  // 1 bits for masking

volatile Tuple Q[N] = { {0, 0}, ... };

uint32_t QMAX( int id, int k ) {
    int low = ((id >> (k - 1))^1)<< (k - 1);  // low (i.e., left) sibling
    int high = min( low | mask >> (depth - (k - 1)), N - 1 );  // high (i.e., right) sibling
    Tuple opp;
    for ( int i = low; i <= high; i += 1 ) {
        opp.atom = Q[i].atom;
        if ( (L(opp) >= k ) return opp.atom;
    }
    return (Tuple){.tuple = {0, 0}.atom;  // return to top-level
} // QMAX

Tuple opp;
for ( int k = 1; k <= depth; k += 1 ) {  // entry protocol, round
    opp.atom = QMAX( id, k );
    Fence();
    Q[id].atom = L(opp) == k ? (Tuple){.tuple = {k, bit(id,k) ^ R(opp)} }.atom : (Tuple){.tuple = {k, 1} }.atom;
    Fence();
    opp.atom = QMAX( id, k );
    Fence();
    Q[id].atom = L(opp) == k ? (Tuple){.tuple = {k, bit(id,k) ^ R(opp)} }.atom : Q[id].atom;
    Fence();
    wait: opp.atom = QMAX( id, k );
    Fence();
    if ( (L(opp) == k && (bit(id,k) ^ EQ(opp, Q[id]))) ) {  // L(opp) > k } { Pause(); goto wait; }
}

CriticalSection( id );
Q[id].atom = (Tuple){.tuple = {0, 0} }.atom;  // exit protocol

Figure 19. Peterson–Fischer tournament.
```
Interestingly, this algorithm is not a direct tree walk. Rather than create a connected tree to find the next opponent, a winner at any level has to perform a linear search among remaining opponents to find its next opponent. For \( N = 8 \), thread \( T_0 \) competes in 3 levels:

<table>
<thead>
<tr>
<th>level</th>
<th>opponents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2, 3</td>
</tr>
<tr>
<td>3</td>
<td>4, 5, 6, 7</td>
</tr>
</tbody>
</table>

At level 1, \( T_0 \) competes with \( T_1 \). At level 2, \( T_0 \) competes with the winner of \( T_2 \) and \( T_3 \) but must perform a linear search to determine which of the two progressed, and hence, which one raced with \( T_0 \) at level 2. Similar at level 3, \( T_0 \) searches among threads \( T_4 \), \( T_5 \), \( T_6 \), \( T_7 \), to find the winner that raced with \( T_0 \) at this level. The search turns space for the connected tree into time, as this algorithm only requires \( N \) storage by packing level and state into a single word (see later in the text). The routine \( QMAX \) performs the search at a specific level and returns an opponent for a thread, with the number of opponents increasing as the level increases. An opponent must be on the same or greater level as the calling thread to be of interest; otherwise, a null opponent is returned, which does not want in to the CS.

The data structure for the Peterson-Fischer algorithm is an array of \( N \) tuples, where each tuple contains a thread’s current tournament level (1 to \( \lfloor \log_2 N \rfloor \)) and race value (\( R \)).

These tuples must be read and written atomically, making the algorithm RW-unsafe. Hence, the two tuple values are half-words fields in a union with a word value, and all reads/writes are performed using the word field (\( \text{atom} \)). Unlike Peterson’s 2-thread read-race solution, the intent is stored separately in the level (0 \( \Rightarrow \) do not want in) so the race state is binary rather than ternary.

The entry protocol loops through the \( \lfloor \log_2 N \rfloor \) levels for each thread, performing a 2-thread read-race at each level. A thread starts by calling \( QMAX \) to find an opponent. For a race, it is necessary to distinguish the two opponents (left/right). However, the threads ids are no longer binary values, as for the 2-thread algorithm, making identification more difficult. Peterson observed that bit position \( k \) (level) is different for the two opponents because the subtrees of the binary tree are a power of 2, for example, for thread \( T_1 \’s \) opponents:

<table>
<thead>
<tr>
<th>thread id, base 10 and 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( k )</td>
</tr>
<tr>
<td>tree</td>
</tr>
<tr>
<td>level</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

At each level, the bit in \( T_1 \’s \) id at position \( k \) (red) is always the opposite of all of its opponents at that level, so \( \text{bit}(i.d, k) \) uniquely identifies a thread with respect to its opponent at level \( k \).

The entry protocol begins by calling \( QMAX \) to find an opponent (or null opponent) at level \( k \). If there is an opponent at the same level, \( L(\text{opp}) = k \), which implies intent, a race occurs as in the 2-thread approach, where the two opponents read from the other’s state, \( R(\text{opp}) \), and write to their own variable \( Q[i.d] \). The race exclusive-ors the local id, drawn from bit position \( k \) of a thread’s id, with the race state, which gives the same value as the race for id 0 and the opposite for id 1, which is like the \( \text{if} \) statement in the 2-thread solution. Otherwise, the thread declares its intent by setting its level to \( k \) (the race value is arbitrary). Like the 2-thread solution, if 2 threads move to the next level and search simultaneously, they do not see each other as they both have a level less than the current level. Hence, they both set \( Q[i.d] \) to \{k, 1\} because \( L(\text{opp}) \neq k \). The second search now finds both
opponents at level $k$ and they now race. The result of the second race is always sufficient, regardless of the result of the first race.

A third search is performed before the loser waits, but this is unnecessary, and the code:

```
wait; opp.atom = QMAX{id, k};
Fence();
if ( (L(opp) == k & & (bit(id, k) ^ EQ{opp, Q[id]})) | | L(opp) > k ) {Pause(); gotowait; }
```

is rewritten as follows in the implementation for this paper:

```
while ( L(opp) > k | | (L(opp) == k & & (bit(id, k) ^ EQ{opp, Q[id]})) ) {
    Pause();
    opp.atom = QMAX{ id, k };
}
```

which also eliminates the fence in the loop. A thread now waits if its opponent is at a higher level, that is, further along in the tree traversal, and any race result is ignored, or for a thread at the same level with which it raced but lost. It is necessary to call $\text{QMAX}$ and continue racing while waiting because the winning thread may exit the CS and race again by itself; at which point, the waiting thread detects it has won the race and continues, which is how fairness is established, that is, alternation on simultaneous arrival.

18.2. Lynch

Figure 20 shows Lynch’s tournament algorithm [28, § 10.5.3], which is a simplification of Peterson’s tournament. However, significant parts of this algorithm are written in prose and therefore left to our interpretation with respect to implementation. First, the intents and race variables are separated into two arrays rather than an array of structures, using slightly more storage than Peterson’s tournament: $N + 2^{\lceil \log_2 N \rceil}$. Second, it uses a write race rather than a read race. The code before the race computes the current location in the tree walk, the opponent at that level (left/right) and sets intent. The code after the race performs the same linear search for opponents as Peterson’s tournament algorithm and delays until a thread becomes a race winner.

18.3. Taubenfeld

Figure 21(a) shows Taubenfeld’s tournament algorithm [29, p. 38], which performs a direct tree walk, that is, there is no linear search to locate an opponent at each level. Each node in the
int depth = ceil( log2( N ) );  // maximal depth of binary tree
int width = 1 << depth;  // maximal width of binary tree
// triangular matrix, row 0: width, row 1: width/2, row 2: width/4, ... 
volatile int intents[depth][2] = {{ 0, ... }, { 0, ... }, ...};  // triangular matrix, row 0: width/2, row 1: width/4, row 2: width/8, ...
volatile int turns[depth][2];

node = id;
for ( int i = 0; i < depth; i += 1 ) {
    lr = node & 1;
    node = node >> 1;
    intents[i][2 * node + lr] = 1;  // declare intent
    turns[i][node] = lr;  // RACE
    Fence();
    while ( ! ( intents[i][2 * node + (1 - lr)] == 0 ||
                turns[i][node] == 1 - lr ) ) Pause();
}

CriticalSection( id );
for ( int i = depth - 1; i >= 0; i -= 1 ) {
    intends[i][id / (1 << i)] = 0;
}

(a) Taubenfeld

ri di = id;
for ( int i = 0; i < depth; i += 1 ) { // entry
    rift = ridi >> 1;
    intends[i][ridi] = 1;  // declare intent
    turns[i][ridi] = ridi;  // RACE
    Fence();
    while ( ( intents[i][ridi ^ 1] == 1 &&
            turns[i][ridi] == ridi ) ) Pause();
    ridi >>= 1;
}

CriticalSection( id );
for ( int i = depth - 1; i >= 0; i -= 1 ) {
    intends[i][id >> i] = 0;  // retract reverse order
}

(b) Buhr

Figure 21. Taubenfeld tournament: (a) original and (b) optimized.

Figure 22. Taubenfeld–Buhr data structure: \( T = 8 \).

tournament tree is a Peterson 2-thread write race algorithm, which is inlined. For non-powers of two threads, \( N \) is rounded up to the next power of 2 to create a maximal binary tree, and unused entries are marked as not participating. This approach results in a worst case of \( \log_2 N \) unnecessary checks (but no spinning) by at most one thread. For example, for \( T = 9 \), \( T_{1-8} \) compete on one half of the tree while \( T_9 \) competes alone on the other half until it reaches the root of the tree, which is essentially the depth of the tree, \( \log_2 N \). The intents and turns needed by the 2-thread Peterson algorithm are separate and stored as two minimal triangular matrices (Figure 22). The storage required for this algorithm is \( 2N + N \) for the triangular intent plus the turn matrices, respectively.

In the exit protocol, the thread leaving the CS must retract its intents in the reverse order acquired. For example, if \( T_4 \) enters the CS, its intents are set in matrix \( \text{intents} \) at elements \([0,4],[1,2],[2,1]\), respectively, during the entry protocol, and hence, these intents must be reset in the order \([2,1],[1,2],[0,4]\). If intents are not reset in reverse order in the exit protocol, there is a race between released threads and the thread retracting intents that walk the same path in the tree. For example, if \( T_4 \) first retracts its intent at \([0,4]\), its partner at that level, \( T_5 \), can now move to \([1,2]\) and set its intent, but that intent can be immediately reset by \( T_4 \) as it moves to \([1,2]\). When intents are retracted in reverse order, released threads move in the opposite direction in the tree from the releasing thread so there is no race.

Note, the algorithm presented in [29, p. 38] has an off-by-one-error in the exit protocol: line 11 of the algorithm \( \text{node} := [i/2^{\text{level}+1}] \) should be \( \text{node} := [i/2^{\text{level}}] \) [30]. An example of a failure
resulting from the ‘+1’ is at the start of the loop retracting intent where level begins with the value \( \log_2 n - 1 \). Substituting for the first iteration:

\[
\text{node} := \lfloor \frac{i}{2^{\log_2 n - 1 + 1}} \rfloor \\
:= \lfloor \frac{i}{2^{\log_2 n}} \rfloor \\
:= \lfloor \frac{i}{2^{\log_2 n}} \rfloor \\
:= \lfloor \frac{i}{n} \rfloor \\
:= 0, \text{ because } i < n
\]

shows a value of 1 can never be generated. However, for thread identifiers from \( n/2 \) to \( n \) (upper half of the threads), the first retraction is at position 1 of the intents for level \( \log_2 n - 1 \), that is, the right side of the two intents. Hence, the wrong intent is retracted. Other failure cases are possible because further loop iterations are incorrect, resulting in retraction of the wrong intents.

### 18.4. Taubenfeld–Buhr

Figure 21(b) shows a simplified version of the Taubenfeld algorithm created for this paper. The new algorithm has only three shifts and an exclusive-or. The only algorithmic difference is the value inserted into variable \( \tau \). The Taubenfeld algorithm computes a 0/1 for each pair of threads at each level and uses these values for the race, as in the Kessels algorithm. The Taubenfeld–Buhr algorithm computes the local id for each pair of threads at each level and uses these values for the race.

### 18.5. Kessels

Figure 23 shows Kessels’ tournament algorithm [5, pp. 140–141], which also performs a direct tree walk. Each node in the tournament tree is Kessels’ 2-thread read-race algorithm [5, § 2], where

```c
typedef struct {
    int Q[2], R[2];
} Token;
volatile Token t[N] = { {0, 0}, ... };

#define inv( c ) (1 - c)
#define plus( a, b ) (((a + b) & 1)

void binary_prologue( int c, volatile Token *t ) {
    t->Q[c] = 1;
    Fence();
    t->R[c] = plus( t->R[inv(c)], c );
    Fence();
    while ( t->Q[inv( c )] && t->R[c] == plus( t->R[inv( c )], c ) ) Pause(); // busy wait
}

void binary_epilogue( int c, volatile Token *t ) {
    t->Q[c] = 0;
}

int n, e[N];

while ( n > 1 ) {
    // entry protocol
    int lr = n & 1;
    n >>= 1;
    binary_prologue( lr, &l[n] );
    e[n] = lr;
}

CriticalSection( id );

for ( n = 1; n < N; n = n + n + e[n] ) {
    // exit protocol
    binary_epilogue( e[n], &l[n] );
}
```

Figure 23. Kessels.
each thread is either the left (0) or right (1) member of the arrays Q and R. Kessels’ algorithm uses a minimal binary tree and dynamically computes the walk for each thread (computations on variable n). Using the minimal binary tree also means there are only \( N - 1 \) nodes needed for the tree, instead of \( 2^{\log_2 N} \). The algorithm stores the binary tree in an array (like heap-sort implementations [31]); for example, for \( T = 5 \), Figure 24 shows the layout of the binary tree in the array (where the first element of the array is unused). Hence, threads 0 and 1 first compete at node 2, while threads 2 and 3 compete at node 4, and so on. The winner at node 1 enters the CS.

Each thread also has a sparse e array (only half the elements are used) to remember the left/right information at each node in the tree during the tree walk computed via \( n \) in the entry protocol. This information is necessary for reconstructing the reverse path through the tree during the exit protocol.

![Figure 24. Kessels data structure: \( T = 5 \).](image)

typedef struct {
    int Q[2], R, PAD; // even multiple of word size by padding
} Token;

#define inv( c ) (1 - c)

void binary_prologue( int c, volatile Token *t ) {
    t->Q[c] = 1;
    t->R = c;
    Fence();
    while ( t->Q[c] ) && t->R == c ) Pause(); // busy wait
}

volatile Token t[N];

typedef struct {
    int es; // left/right opponent
    volatile Token *ns; // pointer to path node from leaf to root
} Tuple;

Tuple states[N][[int]log2(N)]; // handle worst-case critical section
int levels[N] = { -1 }

for ( int id = 0; id < N; id += 1 ) { // precompute thread traversals
    t[id].Q[0] = t[id].Q[1] = 0;
    int start = N + id, level = Log2( start );
    levels[id] = level - 1;
    for ( int s = 0; start > 1; start >>= 1, s += 1 ) {
        states[id][s].es = start & 1;
        states[id][s].ns = &t[start >> 1];
    }
}

int s, level = levels[id];
Tuple *state = states[id];

for ( s = 0; s <= level; s += 1 ) // entry protocol
    binary_prologue( state[s].es, state[s].ns ); // table lookup

CriticalSection( id );

for ( s = level; s >= 0; s -= 1 ) // exit protocol, reverse order
    binary_epilogue( state[s].es, state[s].ns ); // table lookup

Figure 25. Peterson–Buhr.
for resetting intents. The storage for this algorithm is $4N$ words for the array of structures plus the $N$ element $e$ array in each thread.

18.6. Peterson–Buhr

Figure 26 shows the modifications made to Kessels algorithm to increase performance. The read race is replaced with a write race. This modification simplifies the algorithm and generates better performance. The performance gain largely results from the removal of a fence when switching from read to write race. Padding the structure to an even multiple of word size increased performance possibly eliminating some false sharing.

The next change is based on the observation that during a tournament, each thread always follows the same path from base to root in the tournament tree and is always a fixed left or right opponent at each level. As well, the number of levels a thread traversed in the minimal binary tree is known a priori. Therefore, this information can be precomputed at the start of a thread or at the start of the system, after which it is stable and immutable (stationary). As a result, the entry/exit protocols for each thread becomes a table lookup.

This change increases storage for the algorithm to $4N + 2N\log_2 N$ words for the static table of structures and an array for the levels of size $M$ for each CS allowing $M$ entries. However, static tables work for all CS less than or equal to the table size, so only one maximal-sized table is needed for the entire system. And because the table is constant, both compiler and hardware are free to perform best optimizations accessing it.

19. ARBITER

A final $N$-thread software solution uses an arbiter thread to control entry to the CS (Figure 26). However, this approach changes the definition of the problem because the $N$ threads no longer decide among themselves about entry to the CS; rather, the $N$ threads communicate with an arbiter to know when to enter the CS. In essence, the mutual exclusion problem among $N$ threads is converted into a synchronization problem between the $N$ threads and the arbiter.

In the entry protocol, a thread indicates its intent to enter and waits until the arbiter indicates the start of this thread’s turn. The exit protocol indicates to the arbiter the thread has exited the CS. The arbiter cycles around the $N$ intent flags for a thread wanting to enter. Once one is found, the arbiter retracts the thread’s intent and indicates it is this thread’s turn, and then waits for the thread to indicate it has exited the CS. Model checking with Spin shows livelock failure if assignment to intents is not atomic; hence, the algorithm is RW-unsafe.

```c
volatile int intents[N] = { 0, … }, serving[N] = { 0, … };

Worker
    intents[id] = 1;
    while ( serving[id] == 0 ) Pause(); // entry protocol
    CriticalSection( id );
    serving[id] = 0;
    // exit protocol

Arbiter
    int id = N;
    for ( ; ; ) { // force cycle to start at id=0
        for ( ; ; ) { // circular search => no starvation
            id = cycleUp( id, N );
            if ( intents[id] == 1 ) break; // want in ?
            Pause();
        }
        intents[id] = 0; // retract intent on behalf of worker
        serving[id] = 1; // wait for exit from critical section
        while ( serving[id] == 1 ) Pause(); // busy wait
    }
```

Figure 26. Arbiter.
No fences are required as it is safe to depend on eventual consistency. For example, take the fragment of the Arbiter at lines 10–11; without a fence, the fetch of serving[id] in the busy loop can continue to return the intent value sitting in the local store buffer after the worker has retracted intent. However, the store eventually is pushed to memory, and the load sees the retraction.

There is no indefinite postponement (violation of rule 4) because the arbiter never uses the CS, it only lets a thread in. There is no starvation (violation of rule 5) because the arbiter cycles through the \( N \) intent flags so there are at most \( N-1 \) threads ahead of any waiting thread.

Unfortunately, this solution is impractical because the arbiter is continuously busy-waiting checking the intent flags even when there is no contention for a CS. For \( M \) CS, there are \( M \) arbiters spinning constantly, which can consume a substantial amount of the CPU resource. Any attempt to reduce the arbiter spinning, correspondingly slows entry into the CS. So the arbiter is largely impractical for software CS, but it does illustrate an interesting hardware behavior in one of the performance experiments (Section 22).

### 20. HARDWARE LOCK

In contrast to software solutions are locks built directly from atomic hardware instructions. It is important to understand the difference in performance between software and hardware locks, as it is often felt software solutions are significantly slower than hardware. For the contrast, three common hardware locks are examined: spinlock with test-and-set and exponential backoff [32], Mellor-Crummey and Scott (MCS) [33, Figure 6] and the pthread mutex-lock implementation on Linux and Solaris. As with software solutions, the hardware locks provide an entry and exit protocol contained in the lock and unlock routines, and are used as follows:

<table>
<thead>
<tr>
<th>Spinlock</th>
<th>MCS</th>
<th>pthread_mutex_t</th>
</tr>
</thead>
<tbody>
<tr>
<td>volatile int</td>
<td>lock CALIGN;</td>
<td>p = lock CALIGN;</td>
</tr>
<tr>
<td>spin_lock( &amp;lock );</td>
<td>MCS_node node CALIGN;</td>
<td>pthread_mutex_lock( &amp;lock );</td>
</tr>
<tr>
<td>CriticalSection( id );</td>
<td>mcs_lock( &amp;lock, &amp;node );</td>
<td>CriticalSection( id );</td>
</tr>
<tr>
<td>spin_unlock( &amp;lock );</td>
<td>mcs_unlock( &amp;lock, &amp;node );</td>
<td>pthread_mutex_unlock( &amp;lock );</td>
</tr>
</tbody>
</table>

The code for spinlock and MCS appears in Figures 27 and 28. For the SPARC architecture, a custom atomic fetch-and-store and compare-and-swap were used for MCS because the GCC atomic intrinsics insert unnecessary memory barriers. The spinlock has no bound on service; therefore, starvation is possible. MCS provides a bound on service by linking waiting threads into a list and servicing the list in FCFS order. The pthread mutex-lock is an interface with many possible implementations, and implementers have a fair degree of latitude, including direct interaction with the operating system. Hence, we had no control over the pthread mutex-lock as it is part of the dynamically loaded runtime-library provided by the operating system.

```c
void spin_lock( volatile int *lock ) {
    enum { SPIN_START = 4, SPIN_END = 64 * 1024, };,
    int spin = SPIN_START;

    for (;;) {
        if ( lock == 0 && __sync_lock_test_and_set( lock, 1 ) == 0 ) break;
        for ( int i = 0; i < spin; i <<= 1 ) Pause(); // exponential spin
        spin += spin;
        if ( spin > SPIN_END ) spin = SPIN_START; // prevent overflow
    } // spin_lock

void spin_unlock( volatile int *lock ) {
    __sync_lock_release( lock ); // safe release
} // spin_unlock
```

Figure 27. Spin lock using test-and-set and exponential backoff.
typedef struct mcs_node MCS_node;

typedef struct mcs_node {
    MCS_node *volatile next;
    volatile int spin;
} *MCS_node;

void mcs_lock( MCS_lock *lock, MCS_node *node ) {
    MCS_node *pred;
    node->next = NULL;
    pred = __sync_lock_test_and_set( lock, node ); // fetch-and-store
    if ( pred != NULL ) {
        // someone on list ?
        node->spin = 1;
        // mark as waiting
        pred->next = node;
        // add to list of waiting threads
        while ( node->spin == 1 ) Pause(); // busy wait on my spin variable
    }
} // mcs_lock

void mcs_unlock( MCS_lock *lock, MCS_node *node ) {
    if ( node->next == NULL ) { // no one waiting ?
        if ( __sync_bool_compare_and_swap( lock, node, NULL ) ) return; // not changed since last looked ?
        while ( node->next == NULL ) Pause(); // busy wait until my node is modified
    }
    node->next->spin = 0; // stop their busy wait
} // mcs_unlock

Figure 28. John M. Mellor-Crummey and Michael L. Scott (MCS) lock.

21. PERFORMANCE EXPERIMENT

Two performance experiments, maximal and minimal contention, are used to determine performance difference among the algorithms and to contrast performance of software solutions to the two commonly used hardware solutions. (The code to reproduce the experiments is publicly available [34], and the algorithm implementations are verified by the Relacy Race Detector [35].) For low-contention situations, most algorithms reduce to an $O(N)$ search to determine if another thread wants into the CS. However, some algorithms have only a $O(1)$ check for a single thread, making them ideal for very low-contention situations. As contention increases, the amount of search and retry in each algorithm determines its performance.

The maximal contention performance experiment has a test harness that creates $N$ pthread worker threads in the range 1–32, and then blocks for a fixed period, $t$, after which a global stop flag is set to indicate an experiment is over. The $N$ threads repeatedly attempt entry into the self-checking CS of Section 5 until the stop flag is set. During the $t$ seconds, each thread counts the number of times it enters the CS. The higher the aggregate count, the better an algorithm, as it is able to process more requests for the CS per unit time (throughput). When the stop flag is set, a worker thread stops entering the CS and atomically adds it subtotal entry-counter to a global total entry-counter. When the harness unblocks after $t$ seconds, it busy waits until all threads have noticed the stop flag and added their subtotal to the global counter, which is then stored. Five identical experiments are performed, each lasting 20 s. The median value of the five results is plotted. The thread identifiers are randomized across the processors to avoid inadvertent correlations between thread start-up order and geographic placement on the system, which can influence false sharing in the lock data structures.

The minimal contention performance experiment is similar to the maximal experiment, except only 1 pthread worker thread is created, but the algorithms are constructed for $N$ threads in the range 1–32. As a result, each algorithm creates the necessary data structures and executes the necessary looping as if $N$ threads are attempting entry to the CS. Because the single thread encounters no contention, this experiment measures the cost of fast access within an algorithm as $N$ increases. To ensure the single thread exercises all aspects of an algorithm, it is assigned different start-points on each access to the CS by randomly changing its thread id. The randomness is accomplished using approximately 64 pseudorandom thread ids, where 64 is divided by $N$ to get $R$ repetitions, for example, for $N = 5, R = 64/5 = 12$. Each of the 12 repetition is filled with five random value in the range, 0..$N-1$, without replacement, for example, 0 3 4 1 2. There are no consecutive thread
ids within a repetition, but there may be between repetitions. The thread cycles through this array of ids during an experiment. As for the maximal experiment, the higher the aggregate count, the better an algorithm, as it is able to process more requests for the CS per unit time.

The performance experiments were run on two different multicore hardware systems to determine if there is consistency across platforms:

1. Supermicro AS-2042G-6RF with four sockets, each containing an AMD Interlagos 6274 16 core 2.2G 16M 6400MT, equals 64 cores, running Linux v3.2.0-35, compiling with gcc 4.8.1
2. Oracle single-socket SPARC T2+ with eight cores, each core has two pipelines, and each pipeline supports (multiplexes) 4 logical processors, running Solaris 10, compiling with gcc 4.8.1

Because the x86 system has $4 \times 16$-core processors, NUMA effects with respect to accessing shared data occur. The SPARC T2+ is a single-node, UMA architecture.

The major difference between the two hardware architectures is communication cost, due to the single versus multinode architectures. To demonstrate this difference, the following program is run five times for 20 s each, and the median result is presented in Figure 29 for 1–32 threads-Cpus (i.e., situations with parallelism). All CPUs are accessing shared variable turn, so there is high cache-coherence traffic and hence a test of the communication among the processors on and off chip. As well, the fence represents the cost in all the algorithms of precluding the hardware from incorrectly reordering instructions. The higher the loop count per thread, the lower the communication cost as there are more iterations in unit time. The results for the AMD x86 show the communication cost is significant as performance drops by a factor of 3+ across the range of CPUs, that is, the system saturates almost immediately. (Note, the Y-axis is log scale to show the x86 decrease.) The results for the SPARC show the communication cost scales linearly up to 24 CPUs and then saturates similar to the x86 but with no drop in performance out to 32 CPUs.

This difference in communication cost has a significant effect on software solutions for mutual exclusion because the algorithms communicate heavily via shared memory. Therefore, when communication cost is high, placement of threads is crucial to reduce the cost of shared access, that is, pack cores than sockets. However, the default scheduling policies for both Linux and Solaris assumes low sharing among threads (which is normally the correct assumption) and hence scatter threads across sockets and cores to prevent inter-core resource competition, which causes increase caching and core heating, which drops turbo-boost. Because the single-socket SPARC has very low communication costs, thread placement is largely irrelevant; hence, Solaris placement policy does not affect testing the software algorithms on the SPARC T2+ (but would on other SPARC processors). Unfortunately, given the high communication costs on the AMD, the thread placement policy of Linux is the worst possible approach to demonstrate differences among the software algorithms and results in unusual behavior and significant jitter. Therefore, for the AMD experiments, threads
are explicitly pinned (using affinity) to cores and than sockets, which clears up irregularities in
the data and more closely represents single-socket processors on embedded systems, where these
algorithm might be used.

22. PERFORMANCE ANALYSIS

Figures 30 and 31 show the entry count results from the maximal performance experiment for each
algorithm on the x86 and SPARC architectures, respectively. Figures 32 and 33 plot the relative
standard deviation, $rc_v = \frac{g}{\mu} \times 100$, for the maximal performance experiment, which is a percentage
of the coefficient of variation ($c_v$) representing a normalized measure of dispersion of fairness for
each algorithm. If an algorithm is perfectly fair, then the count values for each thread are essentially
equal (modulo small differences at start-up and close-down), resulting in a $rc_v$ of essentially zero.
The more entry-counts differ, the higher the percentage of unfairness. The graph is in log scale
to spread out the curves into three ranges: 1–10%, 10–100%, and 100–1000%. Figures 34 and 35
show the entry count results from the minimal performance experiment, that is, an access with
no contention, 1 thread, but $N = 1..32$. As $N$ increases, more steps are required by most of the
algorithm even when only one thread is accessing the CS. For example, Lamport's fast algorithm in
$O.1/ with 7 accesses, but the tournament algorithms take $O.(log N). Many of the other algorithms
have one or more $O(N)$ searches and retries.

The hardware algorithms are examined first. For maximal contention on x86, the spinlock is 3–4
times faster than MCS. This result is a surprise, but because this paper is about software locks, we
did not pursue the reason for this large difference. For maximal contention on SPARC, the spinlock
and MCS lock are similar. Both Linux and Solaris implementations of pthread locks use a combi-
nation of hardware and software to achieve high performance for low contention. For example,
a thread spins briefly in the application via test-and-set, and if the lock is not acquired, the thread
is descheduled and blocks in the kernel. Hence, for maximal contention on x86 and SPARC, the
 pthread lock has lower performance than the two other hardware locks due to kernel entry. For max-
imal contention fairness, spinlock and pthreads do not provide eventual entry (starvation) and hence
are unfair in the range of 1–30% on the x86 and 30–300% on the SPARC. MCS is FCFS, which
is 0% unfairness and does not appear on the graph. For minimal contention on x86 and SPARC,
the hardware algorithms are constant as $N$ increases because there is only a small, fixed number
of instructions for the single thread. Spinlock performs best, while MCS and pthreads interchange
positions on the x86 and SPARC.

In general, for maximal/minimal contention on both architectures, the software algorithms divide
into three groups:

1. lots of scanning and retries: Peterson, Eisenberg, Peterson–Fischer, De Bruijn, Knuth, Dijkstra,
Burns, Lamport–Fast, Szymanski, Lycklama–Buhr, and Aravind;
2. fewer retries: Lamport bakery, Hehner, and Lynch; and
3. minimal scanning and retries: Taubenfeld, Taubenfeld–Buhr, Kessels, and Peterson–Buhr.

For maximal contention on x86, Peterson–Fischer is an anomaly on the x86 as it is in the lower
group, when it should be in the middle group with Lynch, as it is on the SPARC. The reason for the
shift is the complex mechanism to do the read-race, which causes additional scanning; the scanning
induces coherence traffic, which is more expensive on x86 than on the SPARC. Similarly, Hehner is an
anomaly on the SPARC as it is in the lower group, when it should be in the middle group with Lynch,
as it is on the x86. We do not understand why Hehner is performing poorly on the SPARC. For
maximal contention on x86 and SPARC, the four tournament algorithms, Taubenfeld, Taubenfeld–
Buhr, Kessels, and Peterson–Buhr, perform close or equal to the hardware MCS algorithm (and
Spinlock on SPARC). As well, several software algorithms exceed the performance of the pthread
lock across almost the entire range of processors.

For maximal contention fairness on x86 and SPARC, algorithms with a doorway protocol are
essentially perfectly fair, such as Arbiter, DeBruijn, Eisenberg, Hehner, Knuth, Lamport, Peterson,
Szymanski, Lycklama, and Aravind, and hence are in the range 0–1%, which does not appear on
the graph. For tournament algorithms, the $rc_v$ is zero for powers of 2 and rises and falls in between
these values depending on the non-power of 2 mechanism used. Kessels and Peterson–Buhr have a smooth, small amount of unfairness because of the minimal binary tree and hence are in the range 10–30%. Taubenfeld and Taubenfeld–Buhr have more unfairness right after a power of 2, which then diminishes, because of increasing $N$ to the next power of 2 and hence are in the range 10–100%. Both Peterson tournament and Lynch perform a linear search that results in more non-deterministic unfairness because the search always starts from low to high and hence are in the range 20–200%. (Note, in Figure 35, curves Peterson–Buhr and Kessels are virtually the same, as are Taubenfeld–Buhr, Taubenfeld, Lynch, and Peterson–Fischer, so some curves appear not to be present in the graph.) Algorithms that do not preclude starvation (rule 5), such as Dijkstra, Burns, and Lamport–Fast, have varying amounts of unfairness, depending on the non-determinism during an experiment and hence are in the range 10–400%. RetractIntent is extremely unfair: the highest priority thread essentially enters the CS continuously and does all the counting and hence is in the range 100–1000%.
For minimal contention on x86 and SPARC, Lamport–Fast (Section 15) is constant with only 2 reads and 5 writes. All other algorithms experience some drop as \(N\) increases because of increased data structure size resulting in more scanning. Surprisingly, several algorithms are faster than Lamport–Fast for \(N \leq 4\), and both RetractIntent and Burns2 perform exceptionally well exceeding Lamport–Fast for large values of \(N\) on the x86 and medium values on the SPARC. Given that the thread id is randomized for Fast experiments, there is no short path for either RetractIntent or Burns2; both experiments have to execute an \(O(N)\) search through all the intents. Currently, we have no explanation for this phenomenon.

Conspicuous by its absence is any significant NUMA effects on the x86 after \(N = 16\). At the scale of the graph, there are little or no discernible drops after \(N = 16\), except for the slight drop by the Taubenfeld and Taubenfeld–Buhr algorithms. One final architectural anomaly between x86 and SPARC occurs with the Arbiter algorithm. Communication costs among cores for the SPARC
Figure 32. Relative standard deviation, maximal contention: $N = 1..32$, x86, 20 s, measure of fairness among threads in an algorithm, where 0% is perfect fairness. Algorithms not shown have less than 1% unfairness.

Papers presenting a new mutual exclusion algorithm rarely have a performance comparison with existing, related algorithms. The only two papers similar to this paper, with a comprehensive analysis, are Yang et al. [36] and Zhang et al. [37]. Both of these papers use first-generation multiprocessor computers: BBN TC2000, Sequent Symmetry, Kendall Square Research. Therefore,
** Figure 33. Relative standard deviation, maximal contention: \( N = 1..32 \), SPARC, 20 s, measure of fairness among threads in an algorithm, where 0% is prefect fairness. Algorithms not shown have less than 1% unfairness.

Yang’s paper examines four representative mutual exclusion algorithms: Lamport–Fast (Section 15), Styer [38, Figure 1] tree based**, Peterson–Fischer tree based (Section 18.1), and Anderson [32, Table V] hardware fetch-and-add queue-based, MCS hardware queue-based (Section 20). It also presents two new tournament outlines in Figure 1 [36, p. 54] and Figure 3 [36, p. 57]. We refer to these as outlines instead of algorithms because nothing is made explicit on issues 2, 4, and 5 mentioned in Section 18.

---

** This algorithm attempts to combine the benefits of a tournament with Lamport–Fast. However, our attempt to implement the algorithm found two independent errors both resulting in mutual exclusion violations. It appears both Lemmas 3.1 and 3.2 of [38] have errors.
2. The issue of shared variable storage is ignored for the MX at each tree node.
4. There is no detailed discussion of the tree. The range of threads is split in two parts of size $N/2$ and $(N + 1)/2$, but the method of recursion is left to the reader. In this way, it avoids the problem of locating the shared variables for the internal nodes. On the other hand, they clearly suggests the tree should be nearly balanced.
5. While it is made clear an exiting thread must traverse its acquire path in reverse order, there no discussion on how this is accomplished.

Yang et al. implemented a version of their algorithm and used it to obtain experimental results [36, Figure 5 p. 58], but the actual algorithm/implementation is not presented. We implemented a version of Yang Figure 1, as did Zhang [37, Figure 5, p. 31]. The two algorithm are different, reflecting the fact the Yang outline allows many possible algorithms, and hence, different implementations. Because there is no specific algorithm, and we do not have the implementation used by Yang, this
work does not appear in our performance analysis. A similar argument applies to the other Yang outline in [36, Figure 3 p. 57].

Zhang’s paper examines four representative mutual exclusion algorithms: Lamport retract intent (Section 2(b)), Lamport–Fast (Section 15), FCFS based (Section 17), and binary tree based [36]. It also presents three new algorithms: $d$-ary tree, fast 2-ary tree, and fast.

- The $d$-ary tree algorithm uses a maximal $d$-ary tree with B–L (Section 6.2) for MX at each node of the tree. Because B–L is used, there can be starvation at any level of the tree. The algorithm stores the $d$-ary tree in a square rather than triangular matrix, so more than half of the memory is wasted because it is a tree.
- The 2-thread algorithm in Figure 16 (page 38), used by the fast 2-ary tree, has a typographical error on line u9, `if(rival!=-1)`, which we believe should read
Table I. Algorithm summary: only shared storage is listed.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Location</th>
<th>FCFS</th>
<th>Starve-free</th>
<th>RW-safe</th>
<th>Shared (words)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dekker</td>
<td>Figure 3</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>$3$</td>
</tr>
<tr>
<td>Peterson/Kessels R-race</td>
<td>Figure 4(a)</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>$2 / 4$</td>
</tr>
<tr>
<td>Peterson W-race</td>
<td>Figure 4(b)</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>$3$</td>
</tr>
<tr>
<td>B–L</td>
<td>Figure 2</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>$N$</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>Figure 5</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>$2N + 3$</td>
</tr>
<tr>
<td>Knuth/De Bruijn</td>
<td>Figure 6</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>$N + 1$</td>
</tr>
<tr>
<td>Eisenberg–McGuire</td>
<td>Figure 8</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>$N + 1$</td>
</tr>
<tr>
<td>Lamport bakery</td>
<td>Figure 9</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>$2N$</td>
</tr>
<tr>
<td>Hehner–Shyamasundar</td>
<td>Figure 10</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>$N$</td>
</tr>
<tr>
<td>Burns2</td>
<td>Figure 11</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>$N + 1$</td>
</tr>
<tr>
<td>Peterson</td>
<td>Figure 12</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>$2N + 1$</td>
</tr>
<tr>
<td>Lampport–Fast</td>
<td>Figure 13</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>$N + 3$</td>
</tr>
<tr>
<td>Szymanski</td>
<td>Figure 14</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>$N$</td>
</tr>
<tr>
<td>Lycklama–Hadzilacos (simp)</td>
<td>Figure 15(a)</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>$4N$</td>
</tr>
<tr>
<td>Aravind (3-bits)</td>
<td>Figure 15(b)</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>$2N$</td>
</tr>
<tr>
<td>Peterson–Fischer</td>
<td>Figure 19</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>$N$</td>
</tr>
<tr>
<td>Lynch</td>
<td>Figure 20</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>$N + 2\log_2 N$</td>
</tr>
<tr>
<td>Taubenfeld/Tauben–Buhr</td>
<td>Figure 21</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>$3N$</td>
</tr>
<tr>
<td>Kessels</td>
<td>Figure 23</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>$4N$</td>
</tr>
<tr>
<td>Peterson–Buhr</td>
<td>Figure 25</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>$5N + 2N \log_2 N$</td>
</tr>
<tr>
<td>Arbiter</td>
<td>Figure 26</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>$2N$</td>
</tr>
</tbody>
</table>

if (rival != -1) c[rival]=0; and on line v10, c[rival]=0, which we believe should read if (rival != -1) c[rival]=0;

- The fast 2-ary algorithm in Figure 17 (page 39) is the same as the $d$-ary except the declare-intent data structure is divided into two parts so checking the intent variable is separate from the spin variable to reduce cache contention. As well, the algorithm gives one branch into each tree node higher priority than the other branch, that is, one branch retracts its intent and other does not. The algorithm has a typographical error on line 35, $c[rival]=0$, which we believe should read $c[j][rival]$. As well, the program has a subscript error on line 9 for odd values of $N$. For thread id $N-1$, the expression $1 + 1$ generates a subscript of $N$, which is beyond the end of array $x$. The paper only tests even values of $N$ so the subscript problem did not occur in their experiments. The correction is to dimension array $x$ to $N + 1$ for odd values of $N$ and initialize the last element to ‘don’t want in’.

- The fast algorithm in Figure 19 (page 40) attempts to provide high performance for low contention. However, we were unable to implement it. Model checking of the fast algorithm gives an index out of bounds error in line 24. There are two ways to repair this: either dimension wait[$N+1$] or replace the assignment to wait[i]=0. In both cases, this does not help as further testing for assert violations reveals mutual exclusion failure and the algorithm deadlocks. As a result, the performance graphs, Figures 20 and 21 of [37], show results for an irreproducible algorithm.

We implemented Zhang’s version of Yang (ZY), and the fast 2-ary with corrections and $d$-ary. (The previous graphs are already very dense, so the Zhang performance results are just discussed without showing the lines.) Because these are tournament algorithms, their performance was similar to the other tournament algorithms. However, because they use B–L for MX, performance was less than similar tournament algorithms using Kessels and Peterson for MX. The ZY algorithm has similar performance to Taubenfeld. The fast 2-ary did not improve on the ZY algorithm as occurred in the Zhang results [37, Figure 15, p. 38]. The $d$-ary tree is an idea we did not explore in the tournament section. Having more than two threads at each node of the tree helps flatten the tree but means more contention at each node. Clearly, there is an interesting trade-off here. As Zhang suggests and illustrates on the older architectures, the larger MX nodes can improve performance on NUMA architectures. Selecting the best MX algorithm is also interesting; one of the high-performance fast 2-ary tournament algorithm can be used locally in a node, embedded in...
a larger \( d \)-ary tree distributed across processor chips. Preliminary testing with \( d = 2, 4, 8, 16, 32 \) shows significant jitter at transitions across nodes. For example, with \( d = 4 \) and \( N = 5 \), the fifth thread in alone in the second node with no competitors. Up to a point, unfairness translates to improved performance, which causes an upward spike that diminishes as the second node fills with increasing \( N \). We did not have time to fully investigate \( d \)-ary trees and leave this for future work.

24. CONCLUSION

Generating the implementations in this paper was both difficult and time consuming. Many of the original papers present the algorithm as a fait accomplis with little intuition or explanation of how the algorithm works, followed immediately by a series of complex proofs showing lack of livelock and starvation. As well, many small errors were found in the algorithms during implementation. To be fair, no low-cost medium-sized multiprocessor computers existed when most of the original algorithms were created, so the ability to do extensive testing and analysis did not exist. For this paper, thousands of CPU hours were consumed testing the algorithms and making adjustments to understand their behavior.

For others attempting to build a high-performance software solution for mutual exclusion, we offer this advice. Use a tournament approach, the tournament must use a minimal binary tree, and use Peterson’s 2-thread write race at each node in the tree because it takes advantage of existing assignment atomicity. The better these three components are implemented, the faster the performance. This formula was used to create the final algorithm in this paper, Peterson–Buhr, and it immediately created one of the fastest software solutions. However, tournament approaches are slightly unfair for non-powers of 2.

The most interesting result from the experiments is how well some of the software solutions performed in comparison to hardware solutions on both architectures. This result is surprising, as we started this work believing software solutions would be an order of magnitude slower than hardware. Hence, on hardware without atomic instructions, a good software solution for mutual exclusion can be used with minimal performance degradation, even with significant contention. However, all software solutions restrict access to \( N \) threads for a CS because of the need to have intent and fairness storage for each arriving thread. While \( N \) for a particular CS can be made large initially or modified dynamically to handle a varying number of threads, both approaches can have a negative effect on performance. Hardware atomic instructions can handle an arbitrary number of arriving threads at a CS without fairness using only a small amount of fixed state in the lock. To achieve fairness, additional state is necessary, like the per thread linked-list node in MCS to order waiting threads, but the node storage can be allocated statically as part of the thread’s storage (because a thread can only block on one lock at a time) or dynamically on the thread’s stack as part of the entry protocol with \( O(1) \) cost. Trying to use the same storage trick with software solutions is largely impossible because chaining the storage onto a search data structure at the CS requires an atomic action that does not exist. Nevertheless, the results in this paper show the lack of atomic hardware instructions on a processor should not deter implementers of embedded systems with small, fixed thread counts from using preemptive threading.

Future work is to understand why a software lock can equal or outperform any hardware lock, and why the pthread locks tested had such unusual behavior across the processor range at high contention. This work requires many experiments examining hardware performance counters to determine the particular part of the processor system that is causing the behavior. As well, \( d \)-ary tree tournament algorithms need further study. We also estimate there are another 20–30 software solutions for mutual exclusion in the literature that should be tested for correctness and performance. We plan to add 10+ algorithms and make our testing package more robust for researchers and industry usage. The plan is for a long-term algorithm repository with a standard performance test, which should help compare future work.

ACKNOWLEDGEMENTS

Peter Buhr was funded by the Natural Sciences and Engineering Research Council of Canada.
REFERENCES

30. Taubenfeld G. Personal communication 2012.

