Model Checking Futexes

Hugues Evrard[1] [0009−0001−7956−0371] and
Alastair F. Donaldson[2] [0000−0002−7448−7961]

[1] Google, France, hevrard@google.com
[2] Imperial College London, UK, alastair.donaldson@imperial.ac.uk

Abstract. The *futex* Linux system call enables implementing performant inter-thread synchronisation primitives, such as mutexes and condition variables. However, the futex system call is notoriously tricky to use correctly. In this case study, we use the Spin model checker to verify safety properties of a number of futex-based mutex and condition variable implementations. We show how model checking is able to detect bugs that affected real-world implementations, and confirm current implementations are correct. The Promela models we have developed are available as open source, and may be useful as teaching material for classes that cover futex-based synchronisation primitives, and as a template on how to perform formal verification on new synchronisation primitive designs.

Keywords: futex · mutual exclusion · condition variables · model checking · Promela/Spin

1 Introduction

The futex system call was introduced to the Linux kernel in the early 2000s in order to support efficient synchronisation primitives [9]. The name "futex" is derived from "**f**ast **u**serspace mu**tex**", because one of the most important use cases for the futex system call is the efficient implementation of mutexes, striking a balance between OS semaphores, whose manipulation always involves a system call even when contention is low, and spinlocks, which operate entirely in userspace but may lead to high CPU usage when contention is high.

When used in a careful and clever manner, futexes can enable efficient inter-thread and inter-process synchronisation. However, futexes are also notoriously tricky to use correctly. According to Drepper, in his aptly-titled paper "Futexes are Tricky" [7], a package authored by one of the inventors of the futex system call, containing user-level code demonstrating its use, turned out to be incorrect. Drepper describes why an early mutex implementation suffers from correctness problems, and presents two alternative implementations, arguing their correctness informally. In an article on futex-based condition variables [6], Denis-Courmont describes a number of flawed proposals for implementing condition variables, and a proposal that is argued to be correct under reasonable practical assumptions.

A limitation of these expositions of futex-based synchronisation primitives is that they are based on informal descriptions of how code snippets might behave in a concurrent context. The reader may not fully understand the (often subtle) arguments for

(in)correctness, and even if they do, it may be hard for them to imagine the consequences of alternative implementation choices.

In this case study, we investigate the use of the Promela language and Spin model checker [13] to express and analyse various proposals from [7] and [6] for futex-based mutexes and condition variables, respectively. Due to the ability of model checking to produce counterexamples, our Promela models of incorrect implementations lead to step-by-step traces that illustrate bug-triggering thread interleavings. This facility also aids in understanding why certain details of correct implementations are important, because one can change those details and inspect the counterexamples that arise as a result. In particular, we show that model checking can detect bugs that affected real-world implementations of mutexes and that it can confirm bugs in both naive and real-world implementations of condition variables. We also show that model checking aids in understanding the importance of certain intricacies of a futex-based mutex design.

The Promela models we have developed are available as open source, together with instructions on how to use Spin to analyse them [8]. We envisage that they may be useful as teaching material in classes that cover futex-based synchronisation primitives. In fact, our investigation into the application of model checking to this problem was inspired by the experience of one of the authors teaching about futex-based mutexes on a course at Imperial College London, and being dissatisfied with his informal correctness-related explanations. We also hope that our models will serve as a template on how to perform formal verification on new synchronisation primitive designs.

The rest of the paper is organised as follows. In Section 2 we provide necessary background on the futex system call. We explain how we have modelled this system call in Promela, to enable the modelling of synchronisation primitives that use it, in Section 3. Our Promela models of mutexes and condition variables rely on the modelling of various integer atomic operations, including operations that may overflow; we discuss these in Section 4. In Section 5 we work through examples of futex-based mutex implementations from Drepper's paper [7], explaining how we have modelled each mutex variant using Promela and presenting insights into our analysis of these models using Spin. In Section 6 we turn to condition variables, working through some implementation proposals from Denis-Courmont's article [6]. We discuss related work in Section 7 and conclude with a discussion of future directions in Section 8.

Throughout the paper we assume the reader is familiar with Promela and with basic operation of the Spin model checker. See [13] for a definitive reference.

2 The Futex System Call

The word *futex* is often used to designate three things: (1) a 32-bit addressable value also called the *futex word*, (2) the futex system call, and (3) mutex implementations based on the futex system call. In this section, we are concerned with (1) and (2), while (3) is discussed in Section 5.

Generally speaking, the futex system call enables threads to block depending on the value of a given memory word—the futex word—or to wake up threads that are

waiting in a queue associated with a futex word. In practice, a futex system call has the following form:[3]

```
long syscall(SYS_futex,
  uint32_t *addr, // pointer to the futex word
  int futex_op,   // operation: FUTEX_WAIT, FUTEX_WAKE, ...
  uint32_t val,   // plain value argument
  ...); // extra arguments for other operations
```

It is multiplexed via its `futex_op` argument, which refers to one of various operations. In this case study, we focus on the two basic operations: `FUTEX_WAIT` and `FUTEX_WAKE`, where only the `addr` and `val` arguments are relevant.

`FUTEX_WAIT`: the calling thread blocks and goes to sleep only if the value of the futex word addressed by `addr` is equal to the plain value argument `val`. This operation is atomic with respect to the futex word, which is typically in memory shared between threads. Similar to compare-and-exchange instructions on atomics, this call has a compare-and-block semantics: loading the futex word's value, comparing it to `val`, and blocking happen atomically and are totally ordered with respect to other concurrent operations on the futex word.

`FUTEX_WAKE`: the calling thread wakes threads waiting on the futex identified by `addr`. It wakes `val` threads, or the amount of threads waiting on `addr`, whichever is smaller. There is no guarantee on which threads are woken up, or in which order threads are woken up.

The name "futex" is derived from **f**ast and **u**serspace because futex-based synchronisation primitive implementations (such as implementations of mu**tex**es) typically try first to synchronise using userspace atomic operations on a shared futex word, and only resort to futex system calls in case of contention. We see this pattern in the mutex implementations in Section 5.

3 Modelling the Futex System Call Variants

We model futexes in Promela as a `Futex` type, and two inline macros `futex_wait` and `futex_wake` to represent these variants of the general system call. Before covering these in detail, some general remarks about our modelling approach. To keep the state vector size under control, we use `byte` values virtually everywhere we would use `int` values in C: this is without loss of generality since, in our examples with a handful of threads, all interesting values are within [0, 255]. Threads are mapped to Promela's `proctype` and are identified by their `_pid` builtin variable. The total number of threads is a global constant that we use to dimension arrays. It is defined by a preprocessor macro, `NUM_THREADS`, so that it can be easily changed when invoking Spin (e.g. `spin -DNUM_THREADS=5 ...`):

```
#ifndef NUM_THREADS
#define NUM_THREADS 2
#endif // NUM_THREADS
```

Now, on to futexes: the `Futex` type contains a futex word, the list of threads that are waiting on this futex, and a counter of currently waiting threads:

[3] https://man7.org/linux/man-pages/man2/futex.2.html

```
typedef Futex {
  byte word; // Futex word
  bool wait[NUM_THREADS]; // Wait list: array of bool indexed by thread IDs,
                          // thread T is waiting iff wait[T] is true
  byte num_waiting; // Number of threads currently waiting
}
```

The wait list is modelled via an array indexed by thread IDs: this will prove convenient to wake up sleeping threads in a non-deterministic order. In a C program, each futex is identified by the address of its futex word; here each futex is identified by a variable of type `Futex` which is in global scope so that all threads can refer to it.

The `futex_wait` inline macro models the `FUTEX_WAIT` operation:

```
inline futex_wait(futex, val) {
  if
  :: d_step {
       futex.word == val ->
       printf("T%d futex_wait, value match: %d; sleep\n",
              _pid, futex.word);
       assert(!futex.wait[_pid]); // The thread must not be sleeping already
       futex.wait[_pid] = true;
       futex.num_waiting++;
     }
     d_step { !futex.wait[_pid] -> printf("T%d has woken\n", _pid); }
  :: d_step {
       else -> printf("T%d futex_wait, value mismatch: %d vs. %d; do not sleep\n",
                      _pid, futex.word, val);
     }
  fi
}
```

It takes as argument a variable of type `Futex`, and a plain value to compare to the futex word. If they are equal, the thread goes to sleep: we set its entry in the wait list, and increment the counter of waiting threads. An assertion checks that only non-sleeping threads may go to sleep. Then, the thread blocks until its wait list entry is set to false. If the value argument differs from the futex word, then the thread continues without blocking. Log messages prefixed by the ID of the executing thread are printed to ease the understanding of counterexamples.

The atomic compare-and-block semantics is achieved with the first `d_step` (line 3): this is a better choice than `atomic`, since all the statements in a `d_step` are treated as a single state change by Spin, thus reducing the search depth. It is safe to use `d_step` over `atomic` here since all contained statements are deterministic, there is no jump in or out the `d_step` scope, and there is no blocking statement in the middle of the scope. The `d_step` blocks at lines 11 and 12 guarantee that logging prints values related to the state in which a thread is woken up, or in which a value mismatch occurs, respectively (to avoid confusion due to log messages from other threads being interleaved).

The `futex_wake` inline macro models the `FUTEX_WAKE` operation:

```
inline futex_wake(futex, num_to_wake) {
  atomic {
    assert(!futex.wait[_pid]); // The waker must not be asleep
    byte num_woken = 0;
    do
    :: num_woken == num_to_wake || futex.num_waiting == 0 ->
       break
    :: else ->
       if
```

```
        :: futex.wait[0] -> futex.wait[0] = false; printf("T%d wakes T0\n", _pid)
        :: futex.wait[1] -> futex.wait[1] = false; printf("T%d wakes T1\n", _pid)
#if NUM_THREADS > 2
        :: futex.wait[2] -> futex.wait[2] = false; printf("T%d wakes T2\n", _pid)
#endif
#if NUM_THREADS > 3
        :: futex.wait[3] -> futex.wait[3] = false; printf("T%d wakes T3\n", _pid)
#endif
#if NUM_THREADS > 4
        :: futex.wait[4] -> futex.wait[4] = false; printf("T%d wakes T4\n", _pid)
#endif
#if NUM_THREADS > 5
#error "NUM_THREADS > 5, add more if branches in futex_wake"
#endif
        fi
        futex.num_waiting--;
        num_woken++;
    od
    printf("T%d woke up %d thread(s)\n", _pid, num_woken);
    num_woken = 0; // Reset to avoid state space explosion
  }
}
```

The `num_to_wake` argument indicates the number of threads to wake up, the local variable `num_woken` counts how many threads have been woken so far. Note that we cannot eliminate `num_woken` and instead decrement `num_to_wake` until it reaches zero since the macro argument `num_to_wake` may be a literal value, e.g. in a call such as `futex_wake(futex, 1)`. We enter a loop that wakes one thread per iteration, until the desired number of threads have been woken or there are no more threads to wake. When waking a thread, we use a nondeterministic `if` to pick one of the sleeping threads, which is then woken up by setting its entry in the futex wait list array to `false`.

The whole macro body is contained in an `atomic` scope to prevent concurrent accesses to the futex internals. This time, `d_step` cannot be used due to the nondeterministic order in which threads are woken. At the end of the `atomic` scope, `num_woken` is reset to zero. This is vital to reduce state-space explosion: it prevents Spin from regarding otherwise identical states that differ only in the final value of `num_woken` as distinct, which would lead to Spin continuing its exhaustive search from each such state.

Relying on the non-deterministic selection of enabled `if` branches requires exactly `NUM_THREADS` branches: we use the C preprocessor to achieve this, supporting here up to five threads, with it being easy to support more threads by adding further `if` branches. For a really arbitrary number of threads, one could easily script the generation of these branches. We opt for the C preprocessor to keep the Promela code self-contained.

4 Modelling Atomic Operations and Overflow

The mutex and condition variable implementations rely on standard C/C++ atomic operations that we model in Promela. Atomic compare-and-exchange, `cmpxchg`, compares the value at a location with an `expected` value: if they match, the location is set to a `desired` value; otherwise it is left unchanged. Either way, the original location value is returned, here via a `result` parameter:

```
inline cmpxchg(location, expected, desired, result) { d_step {
  result = location; location = (location == expected -> desired : location)
}}
```

Atomic fetch-and-increment, `fetch_inc`, returns the current value of a location before incrementing it. To limit both state space explosion and counterexample length, we model overflow and wrapping on `byte` values with a tighter upper bound set to the total number of threads plus one, represented a constant, `MAX_BYTE_VALUE`. This is without loss of generality, since C/C++ atomic integers also wrap upon overflow. We define the `inc` macro to handle overflow, and use `d_step` to make `fetch_inc` atomic:

```
#define MAX_BYTE_VALUE (NUM_THREADS + 1)
#define inc(a)  (a == MAX_BYTE_VALUE -> 0 : a + 1)
inline fetch_inc(location, result) {
  d_step { result = location; location = inc(location) }
}
```

In a similar fashion, we define a `dec` macro that handles underflow, and a `fetch_dec` macro for atomic fetch-and-decrement. Some of the Promela models discussed later also make direct use of the `inc` macro when performing an increment in a local expression, rather than operating on a futex word.

5 Model Checking Futex-based Mutexes

We describe the usage scenario and properties for mutexes to which model checking is applied (Section 5.1), then the modelling and verification of the two main mutex implementations from [7] (Section 5.2, Section 5.3).

5.1 Model Checking Harness and Properties

We use the following harness to enable model checking of various futex-based mutex implementations:

```
byte num_threads_in_cs; // Number of threads in the critical section (CS)

active [NUM_THREADS] proctype Thread() {
  do
  :: lock();
     num_threads_in_cs++;
     num_threads_in_cs--;
     unlock();
  :: printf("T%d is done\n", _pid) -> break
  od
}

ltl safe_cs { [](num_threads_in_cs <= 1) } // Never more than one thread in CS
```

It uses an `active` proctype to launch `NUM_THREADS` threads, each of which uses the `lock()` and `unlock()` inline macros to repeatedly lock and unlock a shared mutex. Separate versions of these macros are provided for each mutex implementation discussed below. The macros assume that a global variable of type `Futex` is available. Global variable `num_threads_in_cs`, initialised to 0 by default, is used to record when threads enter and leave the critical section.

We consider model checking of two safety properties: (1) freedom from invalid end states (a built-in feature of Spin), which confirms that it is not possible for a thread to become blocked in a call to `futex_wait` when all other threads have terminated, and

(2) mutual exclusion, captured by the "safe critical section" linear temporal logic (LTL) property, `safe_cs`, which checks that the number of threads in the critical section never exceeds one.

5.2 Incorrect Futex-based Mutex

The following shows C++ code for a subtly incorrect futex-based mutex, adapted from [7, §4]. The futex word is the 32-bit atomic integer field `futex_word`. The intention is that the mutex is free if and only if `futex_word` has value 0.

```
class Mutex {
public:
  Mutex() : futex_word(0) {}
  void lock() {
    uint32_t old_value;
    while ((old_value = futex_word.fetch_add(1)) != 0)
      futex_wait(&futex_word, old_value + 1);
  }
  void unlock() {
    futex_word.store(0);
    futex_wake(&futex_word, 1);
  }

private:
  atomic<uint32_t> futex_word;
};
```

A thread attempts to lock the mutex by incrementing `futex_word` via a `fetch_add` , storing the previous value of the futex word in the local variable `old_value`. If this value is 0 then the thread has locked the mutex, by changing `futex_word` from 0 to 1, and can return from `lock`. Otherwise, the thread calls `futex_wait` with `old_value + 1`: if no other thread modifies the futex word in between the call to `fetch_add` and the call to `futex_wait`, this value will match the futex word and the thread will go to sleep until the lock becomes free. If another thread modifies the futex word before the call to `futex_wait`, then this call will not put the first thread to sleep so that the thread will immediately attempt to acquire the mutex again via another `fetch_add`.

Unlocking the mutex is simpler: `futex_word` is set to 0, and `futex_wake` is called so that one of the threads waiting on `futex_word`, if any, will be woken.

Drepper discusses a correctness issue triggered by an overflow of the futex word. Suppose several threads are contending to try to lock an already locked mutex. It is possible that while a given contending thread T1 is between the calls to `fetch_add` and `futex_wait`, another contending thread T2 calls `fetch_add` and modifies the futex word, such that T1 will not go to sleep and will itself call `fetch_add` again, preventing T2 from going to sleep. This can go on until the futex word wraps back to 0, in which case a contending thread might believe it can successfully lock the mutex.

This mutex design is modelled in Promela by the following inline macros:

```
inline lock() {
  byte old_value;
  do
  :: atomic {
       fetch_inc(futex.word, old_value);
       if
       :: old_value == 0 -> printf("T%d locks mutex\n", _pid); break
       :: else -> printf("T%d lock fail, old_value: %d\n", _pid, old_value);
```

```
9         fi
10      }
11      futex_wait(futex, inc(old_value))
12    od
13  }
14
15  inline unlock() {
16    d_step { futex.word = 0; printf("T%d unlocks mutex\n", _pid); }
17    futex_wake(futex, 1);
18  }
```

Here, we make use of `atomic` and `d_step` blocks to (a) ensure that print statements are executed atomically with the actions that they aim to document, and (b) limit state explosion by allowing interleavings only between operations that have inter-thread visibility: calls to `futex_wait`/`futex_wake`, and statements that manipulate the futex word. For example, it is vital that there is an interleaving point between `fetch_inc` at line 5 and `futex_wait` at line 11. However, there is no value in considering thread interleavings between the `fetch_inc` and the `if..fi` that immediately follows. These only involve a thread manipulating its local state. An interleaving point will cause needless state-space explosion which we have found Spin's partial order reduction does not completely alleviate.

With two threads, Spin quickly verifies the `safe_cs` property and confirms that all end states are valid. This is expected: the bug described above requires a race between multiple contending threads when the mutex is already held by a further thread. With three threads, Spin quickly reports a counterexample (minimised using Spin's iterative shortening algorithm) with the following messages:

T0 locks mutex
T1 lock fail, `old_value`: 1
T2 lock fail, `old_value`: 2
T1 `futex_wait`, value mismatch: 3 vs. 2; do not sleep
T1 lock fail, `old_value`: 3
T2 `futex_wait`, value mismatch: 4 vs. 3; do not sleep
T2 lock fail, `old_value`: 4
T1 `futex_wait`, value mismatch: 0 vs. 4; do not sleep
T1 locks mutex
assertion `num_threads_in_cs <= 1` violated

This nicely illustrates the problem where threads T1 and T2 repeatedly prevent one another from sleeping by each incrementing the futex word before the other can call `futex_wake`; "value mismatch: 0 vs. 4" shows the futex word wrapping from 4 to 0.

The "no invalid end states" property also fails, though with a longer counterexample. Here is a summary of the problem. Suppose that T0 holds the lock. T1 and T2 then get into a race, incrementing the futex word until T1 observed the word's old value to be 3 and T2 observed the word's old value to be 4, so that the word's *current* value is 0 (T2 having caused it to wrap-around). T1 is poised to call `futex_wait(4)`, and T2 is poised to call `futex_wait(0)`, but neither have done so yet.

At this point, T0 unlocks the mutex by setting the futex word to 0, wakes up no threads, and terminates. T1 calls `futex_wait(4)`, which immediately returns due to a value mismatch; T1 tries and succeeds to lock the mutex, then immediately releases it, waking up no threads, and terminates. T2 finally calls `futex_wait(0)`, and by now

the futex word value *is* 0, so T2 goes to sleep with no chance of being woken since all other threads have terminated.

As explained in [7], this problem affected real code. It is great that model checking can quickly expose it, with a clear counterexample.

5.3 Correct Futex-based Mutex

Drepper goes on to present the following more intricate mutex implementation compared with that of Section 5.2, which is claimed to be correct [7, §5]:

```
1  class Mutex {
2  public:
3    Mutex() : futex_word(0) {}
4    void lock() {
5      uint32_t old_value;
6      if ((old_value = cmpxchg(futex_word, 0, 1)) != 0)
7        do {
8          if (old_value == 2 || cmpxchg(futex_word, 1, 2) != 0)
9            futex_wait(&futex_word, 2);
10       } while ((old_value = cmpxchg(futex_word, 0, 2)) != 0);
11   }
12   void unlock() {
13     if (futex_word.fetch_sub(1) != 1) {
14       futex_word.store(0);
15       futex_wake(&futex_word, 1);
16     }
17   }
18
19 private:
20   atomic<uint32_t> futex_word;
21 };
```

We use *waiters* to refer to threads that are asleep due to having called `futex_wait`. In this implementation, the futex word can take on one of three values. A value of 0 means that the mutex is free, while values 1 and 2 mean that some thread, say T, holds the mutex. If the futex word is 1, a state referred as "locked, no waiters", then when T unlocks the mutex, T is not obliged to wake up any waiters. In contrast, if the futex word is 2, a state referred as "locked, waiters", then when T unlocks the mutex, T must call `futex_wake` to request that one waiter be woken.

In `lock`, a thread T first tries to lock the mutex by changing the value of the futex word from 0 to 1 via a `cmpxchg` at line 6. If T succeeds in doing this then it has locked the mutex and can return. In this case, we say that the thread has locked the mutex on the *fast path*.

Otherwise, T must contend for the mutex on the *slow path*, via the loop headed at line 7. The thread considers calling `futex_wait` to go to sleep and be notified when the mutex becomes free. Before this, at line 8, T checks whether the previous value of the futex word was already 2 ("locked, waiters"). If not, the previous value must have been 1 ("locked, no waiters"), so T attempts to change the value from 1 to 2 via another `cmpxchg`. Normally T will then call `futex_wait` at line line 9, but if the `cmpxchg` returns a previous value of 0 this indicates that the mutex has suddenly become free, in which case there is no point calling `futex_wait`; instead, T should try again to lock the mutex.

Once T returns from `futex_wait`, or if T decided not to perform this call due to observing the mutex to be free, it performs another `cmpxchg` to try to lock the mutex

at line 10. In contrast to line 6, here T attempts to change the futex word from 0 to 2 to record the fact that T had to contend for the mutex and so there may be some waiters. T leaves the loop only when the `cmpxchg` at line 10 returns 0: we say that T has locked the mutex *on the slow path*.

The `unlock` function is simpler: the futex word is atomically decremented and its old value is inspected (line 13). If the old value is 1, "locked, no waiters", then the futex word is now 0 so the mutex is properly unlocked, and the thread has no obligation to wake up waiters, so can return from `unlock`. Otherwise the old value must have been 2, "locked, waiters", so the thread must set the futex word to 0 (line 14) and call `futex_wake` to wake up one waiter, if any (line 15).

This mutex design is difficult to understand, and it is unlikely that a reader will gain a full understanding from a best-effort prose explanation such as the above, or the explanation given by Drepper [7]. Particularly subtle is the fact that the futex word can have value 1, "locked, no waiters", despite the fact that there *are* waiters, and conversely the mutex word can have value 2, "locked, waiters" even though there are *no* waiters. Reasoning informally that this mutex implementation is correct is difficult, hence why we decided to model it formally. Here are the Promela `lock` and `unlock` macros for this mutex implementation:

```
1  inline lock() {
2    byte old_value;
3    atomic {
4      cmpxchg(futex.word, 0, 1, old_value);
5      if
6      :: old_value == 0 -> printf("T%d locks mutex on fast path\n", _pid);
7         goto acquired_mutex
8      :: else -> printf("T%d fails to lock mutex on fast path\n", _pid)
9      fi
10   }
11   do
12   :: atomic {
13        if
14        :: old_value == 2
15        :: else -> assert(old_value == 1);
16           cmpxchg(futex.word, 1, 2, old_value)
17           if
18           :: old_value == 0 -> goto retry
19           :: else
20           fi
21        fi
22     }
23     futex_wait(futex, 2)
24 retry:
25     atomic {
26       cmpxchg(futex.word, 0, 2, old_value)
27       if
28       :: old_value == 0 -> printf("T%d locks mutex on slow path\n", _pid);
29          goto acquired_mutex
30       :: else -> printf("T%d fails to lock mutex on slow path\n", _pid)
31       fi
32     }
33   od
34   acquired_mutex:
35 }
36
37 inline unlock() {
38   byte old_value;
39   d_step {
40     fetch_dec(futex.word, old_value);
```

```
41      printf("T%d decrements futex word from %d to %d\n", _pid, old_value, futex.
        word);
42    }
43    if
44    :: d_step { old_value == 2 -> futex.word = 0; old_value = 0 }
45       futex_wake(futex, 1)
46    :: d_step { old_value == 1 -> old_value = 0 }
47    fi
48  }
```

As with the Promela code of Section 5.2 we use print statements for counterexample readability and use `atomic` and `d_step` so that threads only interleave after issuing visible operations. The Promela is a fairly straightforward reflection of the original C++, but the differences in the structured control flow constructs offered by the language led to us making use of Promela's `goto`.

Checking correctness Spin is able to rapidly verify the `safe_cs` property, as well as freedom from invalid end states (see Section 5.1) for our model of this mutex implementation for up to five threads. The results for checking `safe_cs` are summarised in Table 1, checking invalid end states leads to the same number of states and similar times, so they are omitted. Results were obtained using Spin version 6.5.2 on an AMD EPYC workstation running Linux 5.19, with C code generated by Spin compiled using GCC 12.2.0. The times shown are averages taken over 10 runs, and overall we observed a variance of less than 7%.

#Threads	#States	Time (s)
2	370	0.00
3	13058	0.01
4	356992	0.27
5	8680310	10.76

Table 1. State space sizes and times for Drepper's correct mutex.

Understanding bugs in incorrect variants Having a formal, checkable model makes it easy to experiment with the intricacies of this futex-based mutex implementation and understand why they are needed. We give two examples of changes to the mutex implementation that compromise its correctness in ways that might not seem immediately obvious. For each, we show that model checking quickly produces short, illuminating counterexample traces.

Bug 1: incorrect simplification. On line 8 of the C++ code on page 9, the conditions under which a thread calls `futex_wait` are rather complex and, as discussed by Drepper [7], some of this intricacy is for purposes of optimisation. One might wonder whether, from a correctness point of view, it would suffice for a thread that just failed to lock the mutex to set the futex word to 2 ("locked, waiters"), and call `futex_wait` in an attempt to go to sleep. This would amount to replacing lines 8 and 9 of the C++ code with:

```
futex_word.store(2);
futex_wait(&futex_word, 2);
```

This change does not lead to violations of the `safe_cs` property, but does lead to the possibility of lost waiters. Making corresponding adjustments to `lock()` in our

Promela model (including adding a print statement to log the storing of 2 to `futex_word` by a thread), Spin quickly produces the following counterexample when invoked on a 2-threaded configuration:

```
T0 locks mutex on fast path
T1 fails to lock mutex on fast path
T0 decrements futex word from 1 to 0
T0 is done
T1 sets futex.val to 2
T1 futex_wait, value match: 2; sleep
```

The problem is that between T1 observing the mutex to be unavailable and setting the futex word to 2, T0 unlocks the mutex, waking up no waiters, because there are none yet, and terminates. T1 then sets the futex word to 2, goes to sleep and is never woken.

Bug 2: incorrect `cmpxchg`. On line 10 of the C++ code on page 9, when a thread attempts to lock the mutex on the slow path it tries to change the value of the futex word from 0 to 2, in contrast to the fast path, where a value change from 0 to 1 is attempted (line 6). A reasonable question is: is it essential that the slow path changes the futex word to 2? Adapting the `lock()` implementation in Promela so that the slow path changes the futex word to 1 instead of 2, and applying Spin to a two-threaded configuration leads to successful verification. But with three threads, although `safe_cs` still successfully verifies, Spin quickly reports a counterexample demonstrating an invalid end state:

```
T0 locks mutex on fast path
T1 fails to lock mutex on fast path
T1 futex_wait, value match: 2; sleep
T2 fails to lock mutex on fast path
T2 futex_wait, value match: 2; sleep
T0 decrements futex word from 2 to 1
T0 wakes T2
T0 woke up 1 thread(s)
T0 is done
T2 has woken
T2 locks mutex on slow path
T2 decrements futex word from 1 to 0
T2 is done
```

The counterexample illustrates a situation where threads T1 and T2 go to sleep due to T0 holding the mutex. When the mutex becomes free, T0 wakes up T2, and T0 terminates. T2 then succeeds in locking the mutex on the slow path, but does *not* set the futex word to 2 in the process. As a result, when T2 unlocks the mutex it is not obliged to wake up any waiters, so T1 remains asleep. T2 then terminates, so that T1 becomes a "lost waiter".

This concrete example sheds light on why it is *essential* that the `cmpxchg` used to lock the mutex on the slow path changes the futex word to the "locked, waiters" state: this ensures that if there are additional waiters, the thread that succeeds in locking the mutex on the slow path is guaranteed to wake up one of them. Here model checking facilitates experimenting with design variations, and quickly produces counterexamples that clearly illustrate defects.

6 Model Checking Futex-based Condition Variables

Condition variables (`cv`) synchronise threads via three operations: `cv_wait`, `cv_signal` and `cv_broadcast`. The `cv_wait` operation takes a locked mutex as an argument. It atomically unlocks the mutex and puts the calling thread to sleep. Once the thread

is woken up, it locks the mutex again before returning. The `cv_signal` operation wakes up one thread chosen non-deterministically among the sleeping ones, while `cv_broadcast`, which we ignore hereafter for the sake of conciseness, wakes up all sleeping threads.

The `cv_wait` operation is atomic in the sense that by the time another thread locks the mutex, the first thread is in the list of threads sleeping on the condition variable. In particular, consider a pair of threads T0 and T1; first T0 holds the mutex and calls `cv_wait`, then T1 locks the mutex and calls `cv_signal`: the signal from T1 cannot be *lost*, i.e. it must wake up T0.

6.1 Model Checking Harness and Properties

Like for `lock` and `unlock` in Section 5, our harness makes use of to-be-defined macros `cv_wait` and `cv_signal`, and is designed to have threads loop on calling these two operations while always being able to reach termination. In terms of verification, here we pay special attention to make sure the harness can enable catching *lost signal* bugs by checking freedom from invalid end states.

First, condition variables are used in association with a mutex whose internals are irrelevant, so we define a simple mutex Promela implementation where a mutex is a global boolean variable, the default value of which is `false`. Locking involves blocking until its value is `false` before atomically setting it to `true`, while unlocking simply involves setting it back to `false`:

```
bool mutex;
inline mutex_lock() { d_step { !mutex -> mutex = true } }
inline mutex_unlock() { mutex = false }
```

The harness consists of a condition variable used by a single *signaller* thread and one or more *waiter* threads. The waiters call `cv_wait` an arbitrary number of times before terminating. The signaller calls `cv_signal` until all waiters are done, then it terminates. In order to catch *lost signal* bugs, we also make sure the signaller has an execution path where `cv_signal` is called only the necessary number of times to match calls to `cv_wait`, but no more.

To model all this, we start with a constant representing the number of waiters, and a couple of global variables to count the minimum number of signals that are needed and how many threads have terminated, before declaring the waiter threads:

```
#define NUM_WAITERS (NUM_THREADS - 1)
byte num_signals_req; // Number of signals required
byte num_done; // Number of terminated waiter threads

active[NUM_WAITERS] proctype Waiter() {
  do
  :: mutex_lock() ->
     num_signals_req++;
     printf("T%d calls cv_wait()\n", _pid);
     cv_wait();
     printf("T%d returns from cv_wait()\n", _pid);
     mutex_unlock()
  :: break
  od
  num_done++;
}
```

Each waiter loops on either locking the mutex, incrementing `num_signals_req`, calling `cv_wait` and then unlocking the mutex; or exiting the loop and incrementing `num_done` before terminating. Thus, each waiter may do an arbitrary number of calls to `cv_wait` before terminating.

The signaller thread is slightly more complex:

```
1  active proctype Signaller() {
2    do
3    :: num_signals_req > 0 ->
4       mutex_lock();
5       printf("T%d must signal, num_signals_req=%d\n", _pid, num_signals_req);
6       cv_signal();
7       num_signals_req--;
8       mutex_unlock()
9    :: else ->
10      if
11      :: true ->
12         mutex_lock();
13         printf("T%d signals without need\n", _pid);
14         cv_signal();
15         num_signals_req = (num_signals_req > 0 -> num_signals_req - 1 : 0);
16         mutex_unlock()
17      :: true -> printf("T%d won't signal until needed\n", _pid);
18         if
19         :: num_signals_req > 0 -> assert(num_done < NUM_WAITERS)
20         :: num_done == NUM_WAITERS -> assert(num_signals_req == 0); break
21         fi
22      fi
23   od
24 }
```

It loops on either detecting that a signal is required (line 3), in which case it locks the mutex, signals, decrements `num_signals_req` and unlocks the mutex (lines 4–8); or it sees that no signal is required (line 9). In this case, it non-deterministically decides to either call `cv_signal` even though there is no apparent need for it (lines 12–16), or to block until either a signal is needed (line 19), or all waiters are done in which case it breaks out of the loop to terminate (line 20). The `if` branches starting with `true` (lines 11 and 17) model the "internal" decision of the signaller. In particular, once it has decided to block, it must not signal again unless it detects the need for a signal.

On the one hand, this harness enables the signaller to produce an arbitrary number of signals, even if no waiter is currently waiting for a signal. On the other hand—and this is crucial to detect lost signal bugs—when the signaller sees that no signal is needed, it may decide to stop signalling until either a signal is needed, or all waiters are done. This ensures that each call to `cv_wait` is matched by at least one call to `cv_signal`, but potentially no more than the strictly needed number of signals. In the execution path where there is only one signal per wait, if any signal is lost this will lead to a scenario where (a) some waiter is stuck in the `cv_wait` call at line 10, and (b) the signaller is blocked at line 19 because no signals are currently required. Thus the lost signal will lead to the model checker reporting an invalid end state.

The rest of this section covers a couple of futex-based implementations of `cv_wait` and `cv_signal`, as presented in [6]. Each implementation requires a single futex, which is always declared as a global variable named `futex`.

6.2 Take 1: Naive and Incorrect

We start with a naive approach, from the "Simple but very wrong" section in [6]:

```
1  class CondVar {
2  public:
3    CondVar() : futex_word(0) {}
4    void cv_wait(mutex &m) {
5      m.unlock();
6      futex_wait(&futex_word, 0);
7      m.lock();
8    }
9    void cv_signal() { futex_wake(&futex_word, 1); }
10
11 private:
12   atomic<uint32_t> futex_word;
13 };
```

The `cv_wait` operation unlocks the mutex before calling `futex_wait` with a plain value of 0 (the initial value of the futex word) to put the thread to sleep. Upon waking up, it locks the mutex again before returning. The `cv_signal` operation just calls `futex_wake` to wake up one of the sleeping threads.

This is modelled in Promela using the following macros:

```
1  inline cv_wait() {
2    mutex_unlock();
3    futex_wait(futex, 0);
4    mutex_lock();
5  }
6
7  inline cv_signal() { futex_wake(futex, 1) }
```

Invoking Spin on the harness with this version leads to an invalid end state error. Spin produces a counterexample that illustrates the issue: after the mutex is unlocked in `cv_wait` (line 2), the signaller thread might call `cv_signal` and thus `futex_wake` before the waiter calls `futex_wait` (line 3); the signal is lost. In this case, if the signaller decides to block until another signal is needed, then the waiter thread has no chance to be woken up: the system is in deadlock.

6.3 Take 2: Bionic, Unlikely yet Possible Deadlock

Our second take, dubbed "Sequence counter, close but no cigar" in [6], mimics Android's Bionic libc [1] approach to implement condition variables, where `cv_signal` increments the futex word to avoid deadlocks seen in take 1:

```
1  class CondVar {
2  public:
3    CondVar() : futex_word(0) {}
4    void cv_wait(mutex &m) {
5      uint32_t old_value = futex_word;
6      m.unlock();
7      futex_wait(&futex_word, old_value);
8      m.lock();
9    }
10   void cv_signal() {
11     futex_word.fetch_add(1);
12     futex_wake(&futex_word, 1);
13   }
14
```

```
15 private:
16   atomic<uint32_t> futex_word;
17 };
```

In `cv_wait`, the value of the futex word is saved in `old_value` before releasing the mutex, then `futex_wait` is called with `old_value`. In `cv_signal`, the futex word is incremented by 1, with a possible overflow, before calling `futex_wake`. This avoids the deadlock situation encountered in Section 6.2: if `cv_signal` is executed between unlocking the mutex (line 6) and calling `futex_wait` (line 7) in `cv_wait`, the futex word value will be different from the value used in the call to `futex_wait` which thus will not block.

This is modelled in Promela using the following macros:

```
1  inline cv_wait() {
2    byte val = futex.word;
3    mutex_unlock();
4    futex_wait(futex, val);
5    mutex_lock();
6  }
7
8  inline cv_signal() {
9    futex.word = inc(futex.word);
10   futex_wake(futex, 1);
11 }
```

However, Spin still reports a possible deadlock: if between lines 3 and 4, `cv_signal` is called enough times to overflow the futex word and bring it back to the `old_value` saved in line 2, then the call to `futex_wait` does block, and we reach a deadlock. This issue is documented in Bionic, with an acknowledgement that it would be extremely unlikely to arise in practice: with a 32-bit futex word, we would need *exactly* 2^{32} calls to `cv_signal` in a row, while `cv_wait` is between lines 3 and 4, to trigger the deadlock.

Such issues are hard to foresee at design time. Model checking is valuable in illustrating rare risks of deadlocks, and evaluating their acceptability in practice.

7 Related Work

There is a significant literature on formal verification of inter-process communication primitives. Bogunovic *et al.* verified mutual exclusion algorithms with SMV [4], with an analysis of liveness and fairness. Mateescu and Serwe analysed 27 different shared-memory mutual exclusion protocols with CADP for both correctness and performance [15,16]. Bar-David and Taubenfeld used model checking techniques to automatically discover mutual exclusion algorithms [2]. More recently, Kokologiannakis and Vafeiadis developed a specific dynamic partial order reduction (DPOR) technique to better handle the barrier synchronisation primitive [14]. In terms of using model checking for education, Hamberg and Vaandrager wrote about their experience using UPPAAL in a course on operating systems [12].

We are not aware of formal verification of futex-based synchronisation primitives. Futexes are primarily a Linux system call [10,11]. Besides the two reference publications from Franke *et al.* [9] and Drepper [7], Benderski wrote a good introduction on the topic [3]. Note that the futex system call itself has suffered from bugs that affected userspace applications, such as the Java VM [17].

8 Future Directions

We have presented a case study of modelling a series of futex-based implementations of mutexes and condition variables in Promela, and using Spin to verify safety properties. An immediate extension would be to consider fairness to enable verifying liveness properties, like the absence of starvation. We can also explore additional futex-based synchronisation primitives, for instance barriers.

To create an educational resource that would require little model checking expertise, we can think of doing verification directly on C implementation by using a C model checker, like CBMC [5]. We can even envision extracting C models from various C standard library implementations (e.g. glibc), to verify designs actually used in widespread libraries. Finally, it would be interesting to verify the implementation of the futex system call implementation itself in the Linux kernel and other OSes that have adopted futexes (e.g. OpenBSD).

References

1. Android: Bionic C library, pthread_cond implementation (2023), https://android.googlesource.com/platform/bionic/+/refs/tags/android-13.0.0_r24/libc/bionic/pthread_cond.cpp, last accessed 2023-01-10
2. Bar-David, Y., Taubenfeld, G.: Automatic discovery of mutual exclusion algorithms. In: Fich, F.E. (ed.) Distributed Computing, 17th International Conference, DISC 2003, Sorrento, Italy, October 1-3, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2848, pp. 136–150. Springer (2003). https://doi.org/10.1007/978-3-540-39989-6_10, https://doi.org/10.1007/978-3-540-39989-6_10
3. Benderski, E.: Basics of futexes (2018), https://eli.thegreenplace.net/2018/basics-of-futexes/, last accessed 2023-01-10
4. Bogunovic, N., Pek, E.: Verification of mutual exclusion algorithms with smv system. In: The IEEE Region 8 EUROCON 2003. Computer as a Tool. vol. 2, pp. 21–25. IEEE (2003)
5. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004). Lecture Notes in Computer Science, vol. 2988, pp. 168–176. Springer (2004)
6. Denis-Courmont, R.: Condition variable with futex (2020), https://www.remlab.net/op/futex-condvar.shtml, last accessed 2023-01-10
7. Drepper, U.: Futexes are tricky (2011), https://www.akkadia.org/drepper/futex.pdf, last accessed 2023-01-10
8. Evrard, H., Donaldson, A.: Model checking futexes: Code examples (2022), https://github.com/mc-imperial/modelcheckingfutexes, last accessed 2022-01-16
9. Franke, H., Russell, R., Kirkwood, M.: Fuss, futexes and furwocks: Fast userlevel locking in Linux. In: Ottawa Linux Symposium 2002. pp. 479–495 (2002), https://www.kernel.org/doc/ols/2002/ols2002-pages-479-495.pdf, last accessed 2022-01-10
10. Futex manual page section 2 (system calls) (2023), https://man7.org/linux/man-pages/man2/futex.2.html, last accessed 2023-01-16
11. Futex manual page section 7 (miscellaneous) (2023), https://man7.org/linux/man-pages/man7/futex.7.html, last accessed 2023-01-16
12. Hamberg, R., Vaandrager, F.: Using model checkers in an introductory course on operating systems. ACM SIGOPS Operating Systems Review **42**(6), 101–111 (2008)
13. Holzmann, G.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional, 1st edn. (2011)
14. Kokologiannakis, M., Vafeiadis, V.: Bam: Efficient model checking for barriers. In: International Conference on Networked Systems. pp. 223–239. Springer (2021)
15. Mateescu, R., Serwe, W.: A study of shared-memory mutual exclusion protocols using cadp. In: International Workshop on Formal Methods for Industrial Critical Systems. pp. 180–197. Springer (2010)
16. Mateescu, R., Serwe, W.: Model checking and performance evaluation with CADP illustrated on shared-memory mutual exclusion protocols. Sci. Comput. Program. **78**(7), 843–861 (2013). https://doi.org/10.1016/j.scico.2012.01.003, https://hal.inria.fr/hal-00671321/en
17. Mechanical sympathy email group, discussion titled linux futex_wait() bug (2015), https://groups.google.com/g/mechanical-sympathy/c/QbmpZxp6C64, last accessed 2023-01-16