State-Space Exploration for Concurrent Algorithms under Weak Memory Orderings (Preliminary Version)

Bengt Jonsson
UPMARC
Department of Information Technology, Uppsala University, Sweden
bengt@it.uu.se

Abstract

Several concurrent implementations of familiar data abstractions such as queues, sets, or maps typically do not follow locking disciplines, and often use lock-free synchronization to gain performance. Since such algorithms are exposed to a weak memory model, they are notoriously hard to get correct, as witnessed by many bugs found in published algorithms We outline a technique for analyzing correctness of concurrent algorithms under weak memory models, in which a model checker is used to search for correctness violations. The algorithm to be analyzed is transformed into a form where statements may be reordered according to a particular weak memory ordering. The transformed algorithm can then be analyzed by a model-checking tool, e.g., by enumerative state exploration. We illustrate the approach on a small example of a queue, which allows an enqueue operation to be concurrent with a dequeue operation, which we analyze with respect to the RMO memory model defined in SPARC v9.

1. Introduction

Shared-memory multiprocessors and multi-core chips are now ubiquitous. Programming such systems remains a huge challenge [19]. To make matters worse, most commonly used multiprocessor architectures use weak memory ordering models (see, e.g., [1]). For example, a processor may reorder loads and stores by the same thread if they target different addresses, or it may buffer stores in a local queue. To avoid exposing the programmer to these complications, programming guidelines recommend to employ a locking discipline to avoid race conditions which expose the particular weak memory model of the target platform. Such programs can be understood as using a sequentially consistent memory semantics [12], and can be reasoned about using standard interleaving semantics.

Concurrency libraries, e.g., the Intel Threading Building Blocks or the `java.util.concurrent` package, support the programmer by providing concurrent implementations of familiar data abstractions such as queues, sets, or maps. Implementations of such libraries typically do not follow locking disciplines, and can use lock-free synchronization for gaining performance (e.g., [13, 18]). Since these algorithms are exposed to a weak memory model, they are notoriously hard to get correct, as witnessed by many bugs found in published algorithms (e.g., [7, 14]). Implementations that use lock-free synchronization require explicit memory ordering fences to function correctly on weak memory models. Fences counteract the ordering relaxations by selectively enforcing memory order between preceding and succeeding instructions. A lack of fences leads to incorrect behavior, whereas an overzealous use of fences impacts performance. Unfortunately, fence placements are rarely published along with the algorithm.

This paper addresses the problem of verifying correctness of, or finding bugs in, concurrent algorithms which do not rely on explicit locking, as found in, e.g., lock-free implementations of common data structures. The absence of locks means that standard race-detection tools (e.g., [9, 16]) are of little use. Existing verification and testing techniques and tools must therefore be adapted to handle also weak memory models.

A large share of all verification, program analysis, and testing algorithms, can very roughly be thought of as performing an exploration of possible sequences of computation steps, starting from some initial state. There is of course a huge variation in how they cover the space of computations, and whether they store intermediate system configurations in order to avoid repeating already performed work. Some such exploration tools include the model checker SPIN [11], the backtracking testing/simulation tool VeriSoft [10], and most testing techniques (e.g., [17]). The goal of our work is to provide techniques to adapt them to handle weak memory orderings. In this paper, we consider the model checker SPIN. More precisely, we present a tech-

nique to adapt models analyzed by SPIN, which are naturally expressed for sequentially consistent memory models in the Promela modeling language, so that they also represent all possible computations under a weak ordering.

Some Related Work The research performed on the problem of analyzing concurrent algorithms correct under weak memory models is still limited. The work on CheckFence by Burckhardt and Alur [4] use a bounded model checkin approach rather than state-space exploration: they encode possible computations by a constraint system, and use a SAT solver to search for correctness violations. The work closest to ours is that by Park and Dill [15], who have developed an operational encoding of a shared memory with weak ordering constraints, in particular the RMO model used in SPARC v9, and used it to analyze simple synchronization examples from the SPARC architecture manual, using the model checker Murφ [6]. Their work only reports application to very small examples, our aim is to make a more efficient operational representation of the weak memory model, and to be able to analyze more complicated algorithms, such as, e.g., those considered by Burckhardt and Alur [4]. Some specific weak memory ordering has also been considered in program analysis work [8]. Burckhardt and Musuvathi [5] develops a run-time monitoring tool which checks whether concurrent executions are sequentially consistent, by maintaining vector clocks.

2. Representing Weak Memory Models

In this section, we describe the principles for representing the memory model in this work. Abstractly, a memory model specifies how the program operations "see" the effects of other program operations through the memory system. The interesting part here is how load operations see store operations. More specifically, an execution consists of a set of load and store operations (plus memory barriers, to be explained later), which affect the "state" of the main memory. Each load sees the value of some store operation (or the initial value) to the same location. The hard part is to describe in a concise way which store operations can be seen.

We follow Burckhardt [3] (who in his turn follows previous work), and use

- a partial order $\prec$, called the program order, which is a total order on all operations of the same thread, and which does not order two operations of different threads,
- a total order $<_M$, called the memory order, which intuitively models the order in which operations reach the "main memory".

The fact that $<_M$ is a total order implies that we are aiming at modeling memory models with a global store order, i.e., such that the stores of a thread are seen in the same order by all other threads. Load operations of the thread that performs the store may see it earlier than other threads through the mechanism of store-load forwarding.

The orderings $\prec$ and $<_M$ are related by four axioms. For a load operation l, let $\mathit{seed}(l)$ be the store operations which stores the value that l loads. Let $S(l)$ be the set of store operations s which access the same address as l, such that either $s <_M l$ or $s \prec l$. The axioms are

(A1) whenever x and y are operations to the same address, y is a store, and $x \prec y$, then $x <_M y$,

(A2) $\mathit{seed}(l) \in S(l)$ for all loads l,

(A3) $\mathit{seed}(l)$ is the maximal element wrp. to $<_M$ in $S(l)$,

(A4) whenever $x \prec f \prec y$ for a fence operation f, and x and y match the type of the fence f (e.g., if f is a load-load fence, then x and y should both be load operations), then $x <_M y$.

We shall in particular consider the RMO memory model, defined by SPARC v9, which is also used by Park and Dill [15], which is nice because it preserves single-thread semantics. This is done by defining a dependency order, which constrains the order between data dependent operations of the same thread. For operations x and y of the same thread where x is a load, we say that $x <_d y$ if y loads a data register that is written by x, or if some control branch instruction between x and y is data dependent on x. Add the axiom

(A5) whenever $x \prec y$ and $x <_d y$, then $x <_M y$.

Roughly speaking, the RMO ordering differs in two respects from the natural sequential consistency model. First, operations of one thread may be reordered, but respecting fences and data dependencies. Second, while the global memory order is a merge of the (possibly reordered) local orderings as in sequential consistency, a load sees the latest store to the same location in the same thread, if it is later wrp. to $<_M$ than the latest preceding store in memory order.

Finally, let us consider locks. In the examples we have locks which are updated by `lock` and `unlock` operations. In this work, we assume that lock operations are atomic, and that

- For the `lock` operation, a fence is inserted to make sure that the `lock` operation precedes all the following instructions of the thread in memory order.
- For the `unlock` operation, a fence is inserted to make sure that the `unlock` operation succeeds the preceding instructions of the thread in memory order.

3. Illustration of Technique

Our ambition is to consider examples, such as those taken from the thesis of Sebastian Burckhardt [3]. In this section, we use one of them, a two-lock queue, to illustrate how the technique presented in this paper, should work. The code for the queue, in C syntax, is the following, taken literally from [3].

```
 1 #include "lsl_protos.h"
 2
 3 /* ---- data types ---- */
 4
 5 typedef int value_t;
 6
 7 typedef struct node {
 8   struct node *next;
 9   value_t value;
10 } node_t;
11
12 typedef struct queue {
13   node_t *head;
14   node_t *tail;
15   lsl_lock_t headlock;
16   lsl_lock_t taillock;
17 } queue_t;
18
19 /* ---- operations ---- */
20
21 void init_queue(queue_t *queue)
22 {
23   node_t *dummy =
        lsl_malloc(sizeof(node_t));
24   dummy->next = 0;
25   dummy->value = 0;
26   queue->head = dummy;
27   queue->tail = dummy;
28   lsl_initlock(&queue->headlock);
29   lsl_initlock(&queue->taillock);
30 }
31
32 void enqueue(queue_t *queue, value_t val)
33 {
34   node_t *node = lsl_malloc(sizeof(node_t));
35   node->value = val;
36   node->next = 0;
37   lsl_lock(&queue->taillock);
38   lsl_fence("store-store");
39   queue->tail->next = node;
40   queue->tail = node;
41   lsl_unlock(&queue->taillock);
42 }
43
44 boolean_t dequeue
          (queue_t *queue, value_t *retvalue)
45 {
46   node_t *node;
47   node_t *new_head;
48   lsl_lock(&queue->headlock);
49   node = queue->head;
50   new_head = node->next;
51   if (new_head == 0) {
52     lsl_unlock(&queue->headlock);
53     return false;
54   }
55   lsl_fence("data-dependent-loads");
56   *retvalue = new_head->value;
57   queue->head = new_head;
58   lsl_unlock(&queue->headlock);
59   lsl_free(node);
60   return true;
61 }
```

The prefix `lsl` on some operations (for memory management and lock operations) means that they refere to particular definitions of these operations used in [3].

Generating an Analyzable Program In order to see which sequences of loads and stores are in principle generated by these operations, we transform the description into "high-level machine instructions", which are on the same level of abstraction as the above C pseudocode, but obeys the restriction that each statement induces at most one store or load operation. A store operation is typically of the form `*p = v` for some address `p` and value `v`. We allow both `p` and `v` to be locally computable expressions. Analogously, a load operation has the form `r = *p` for some local variable `r` (sometimes called register), and address `p`. The first transformation typically preserves most of the description, but breaks up statements that involve more than one store or load. In order to introduce offset calculations more explicitly, for a field `f` in a structure `struct`, we introduce `[f]` to denote the offset induced by `f`. Thus, if `structp` points to `struct`, then `structp + [f]` points to the field `f` in `struct`.

Let us first consider the `init_queue` operation. We transform the code into

```
   void init_queue(queue)
   {
1  node_t *dummy =
          lsl-malloc(sizeof(node_t));
2  *(dummy + [next]) = 0;
3  *(dummy + [value]) = 0;
4  *(queue + [head]) = dummy;
5  *(queue + [tail]) = dummy;
6  lsl-initlock(queue + [headlock]);
7  lsl-initlock(queue + [taillock]);
   return
   }
```

This is essentially the same as before. In order to infer which are possible orderings between statements, we should

find the data-dependencies between statements. The only ones in this function are that line 1 must precede lines 2, 3, 4, and 5, through the dependence on dummy.

Next we consider the enqueue operation. At first, we ignore the fence instruction at line 38. We transform the code as follows (e.g., breaking the statement at line 39 into two: one load and one store).

```
    void enqueue(queue, val)
    {
 1  node = lsl-malloc();
 2  *(node + [value]) = val;
 3  *(node + [next]) = 0;
 4  lsl-lock(queue + [taillock]);
 5  queuetail = *(queue + [tail]);
 6  *(queuetail + [next]) = node;
 7  *(queue + [tail]) = node;
 8  lsl-unlock(queue + [taillock]);
 9  return
    }
```

Our next job is to see which orderings in the program order are preserved in the RMO model. We see that dependencies arise as follows:

$$1 <_d 2 \quad 1 <_d 3 \quad 1 <_d 6 \quad 1 <_d 7 \quad 5 <_d 6 \quad 5 <_d 7$$

It remains to understand the ordering constraints imposed by the lock operation. These ensure that instruction 4 precede all following instructions, and that 8 succeed all preceding instructions. In total, we arrive at the following dependencies:

$$1 <_d 2 <_d 8 \quad 1 <_d 3 <_d 8 \quad 1 <_d 6 <_d 8 \quad 5 <_d 7$$
$$1 <_d 7 <_d 8 \quad 4 <_d 5 <_d 6 <_d 8 \quad 4 <_d 7 <_d 8$$

We can summarize these dependencies in the following diagram.

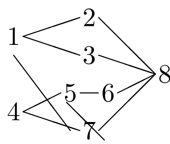

Figure 1. Dependencies in the procedure enqueue

We finally consider the function dequeue. A condensed pseudo-code is as follows

```
    void dequeue(queue, retvalue)
    {
 1  lsl-lock(queue + [headlock]);
 2  node = *(queue + [head]);
 3  new_head = *(node + [next]);
 4  if (new_head == 0) {
 5      lsl-unlock(queue + [headlock]);
 6      return(0);
    }
 7  tmp = *(new_head + [value]);
 8  *retvalue = tmp:
 9  *(queue + [head]) = new_head;
10  lsl-unlock(queue + [headlock]);
11  lsl-free(node);
12  return(1);
    }
```

Here, there are more data dependencies.

$$1 <_d 2 <_d 3 <_d 4 <_d 5$$
$$4 <_d 7 <_d 8 <_d 10 \quad 4 <_d 9 <_d 10 \quad 3 <_d 11$$

Lines 6 and 12 should be the last ones

Generating a Promela Model In order to use the SPIN model checker to analyze the queue implementation, we must produce a Promela Model, which executes the statements of the transformed program in any possible order consistent with the ordering. To to this, we must consider the following issues.

- Promela does not support dynamid heap data structures. Instead, we model, e.g., the queue structure as just a structure, and the nodes of type node_t by an array.
- Promela has only a few standard control constructs, therefore we should find an idiom for allowing all executions that are linearizations of a partial order. We can do this by a loop, which in each iteration checks whether the appropriate preceding statements have been executed in order to see whether some instruction is enabled. This scheme needs an array of flags to record which statements have already been executed.

A possible Promela model of the above queue for a particular test case is shown in Appendix A.

4. Experiments

We have so far only considered the example queue described in Section 3, to obtain some illustrative example. We ran exhaustive analyses using several different test harnesses that first perform an initialization using init_queue, and thereafter starts a number of threads, each of which performs a sequence of enqueue or dequeue operations. After this, we check that the sequence of values returned by the dequeue operation is

consistent with a normal sequentially consistent execution of these operations.

We denote test harnesses in a condensed notation (following [3]), using a sequence of e (for `enqueue`) and d (for `dequeue`) in each thread, and separating threads by |. For example, the test (ee | dd) has two threads, one with two `enqueue` operations, and one with two `dequeue` operations.

We first performed a simple test (e | d), which found a shortest counterexample in a few seconds, generating about 900 states. The problem is the obvious one, that the initialization of the new node in `enqueue` at line 2 can be delayed past the dequeueing of the same node, so that the dequeue operation read an uninitialized `value` field. This problem can be remedied by a store-store-fence between lines 3 and line 6 of `enqueue`, e.g., after the `lock` operation, as in the C pseudocode (line 38). This implies that line 6 can be completed only after lines 2 and 3. We modified the promela model accordingly, and reran the test, and the number of reachable states decreased to 250 with no violation of sequentially consistent semantics. The Promela model for this experiment is given in the appendix.

We thereafter subjected the model to the two largest tests of [3], namely (eeeee | ddddd), and (e | e | e | e | d | d). The first test completed by SPIN in less than one second, generating about $100,000$ states. The second test completed after 260 seconds, using state compression and between 1GB and 2GB of memory, generating a state space of $28,000,000$ states. Out of curiosity, we tried different values for the number of operations in the first test, and were able to make SPIN analyze two threads, each with 10 operations, in 143 seconds, generating around $37,000,000$ states. It seems that SPIN has problems handling a large number of threads, due to the many possible interleavings It seems that work on optimization is needed to make the approach scale to a larger number of threads.

5 Conclusions

We presented a technique for analyzing correctness of concurrent algorithms, under weak memory models. The algorithm to be analyzed is transformed into a partial ordering form, which satisfies exactly the ordering constraints imposed by the memory model under consideration. The transformed algorithm can then be analyzed by a model checking tool, such as SPIN.

We implemented the approach in the context of the SPIN model checker [11], by developing a transformation to Promela models, which follows a certain idiom to model execution under partial order constraints. We illustrated the approach by applying it to an example used in the thesis by Burckhardt [3]. the scalability of the approach by applying it to published synchronization algorithms and concurrent data structures.

We should not make to firm conclusions about this approach from the limited amount of experiments conducted. For a better evaluation, the transformation should be automated; now it is by hand. For the particular example considered, the limitations, in terms scalability, appear comparable to the approach by Burckhardt. In our approach, we were able to perform slightly longer test cases, but on the other hand Burckhardt's approach is automated.

The work closest to ours, by Park and Dill [15], use a similar approach of letting a model checker examine all possible executions that are consistent with the memory model. Their work only reports application to very small examples. We have been able to show that the approach can also handle interesting concurrent algorithms.

An impression from the illustrating example is that increase in the number of interleavings as the number of threads grow will impose limits on the scalability in using a model checker in the way proposed in this paper. We can probably make scalability better by introducing a more generic model of the heap; now there will be many duplications of isomorphic heaps in the state space from our representation as an array. It may also be fruitful to consider approaches which are not so sensitive to this explosion, considering either static program analysis (e.g., as in [8] or parameterized infinite-state model checking (e.g., as in [2].

Acknowledgments Ke Jiang pointed out a problem in a previous version of the paper.

References

[1] S. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. IEEE Computer, 29(12):66–76, 1996.

[2] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In Emerson and Sistla, editors, Proc. 12th Int. Conf. on Computer Aided Verification, volume 1855 of Lecture Notes in Computer Science, pages 403–418. Springer Verlag, 2000.

[3] S. Burckhardt. Memory Model Sensitive Analysis of Concurrent Data Types. PhD thesis, Univ. of Pennsylvania, 2007.

[4] S. Burckhardt, R. Alur, and M. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. In PLDI 2007, pages 12–21, 2007.

[5] S. Burckhardt and M. Musuvathi. Effective program verification for relaxed memory models. In Computer-Aided Verification (CAV), pages 107–120, 2008.

[6] D. Dill. The mur*phi* verification system. In Proc. 8th Int. Conf. on Computer Aided Verification, volume 1102 of Lecture Notes in Computer Science, pages 390–393. Springer Verlag, 1996.

[7] S. Doherty, D. Detlefs, L. Groves, C. Flood, V. Luchangco, P. Martin, M. Moir, N. Shavit, and G. S. Jr. Dcas is not a silver bullet for nonblocking algorithm design. In SPAA

2004: Proceedings of the Sixteenth Annual ACM Symposium on Parallel Algorithms, June 27-30, 2004, Barcelona, Spain, pages 216–224, 2004.
[8] P. Ferrara. Static analysis via abstract interpretation of the happens-before memory model. In Proc. TAP 2008, 2nd Int. Conf. Tests and Proofs, Prato, Italy, volume 4966 of Lecture Notes in Computer Science, pages 116–133. Springer Verlag, April 2008.
[9] C. Flanagan and S. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. Science of Computer Programming, 71(2):89–109, 2008.
[10] P. Godefroid, B. Hammer, and L. Jagadeesan. Model checking without a model: An analysis of the heart-beat monitor of a telephone switch using verisoft. In Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis, pages 124–133, 1998.
[11] G. Holzmann. The model checker SPIN. IEEE Trans. on Software Engineering, SE-23(5):279–295, May 1997.
[12] L. Lamport. Time, clocks and the ordering of events in a distributed system. Communications of the ACM, 21(7):558–565, 1978.
[13] M. Michael. Scalable lock-free dynamic memory allocation. In PLDI 2004, pages 35–46, 2004.
[14] M. Michael and M. Scott. Correction of a memory management method for lock-free data structures. Technical Report TR599, University of Rochester, 1995.
[15] S. Park and D. Dill. An executable specification and verifier for relaxed memory order. IEEE Trans. on Computers, 48(2):227–235, 1999.
[16] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. ACM Trans. on Computer Systems, 14(4):391–411, Nov. 1997.
[17] K. Sen. Race directed random testing of concurrent programs. In PLDI 2008, pages 11–21, 2008.
[18] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. J. Parallel Distrib. Comput., 65(5):609–627, 2005.
[19] H. Sutter and J. Larus. Software and the concurrency revolution. ACM Queue, 3(7):54–62, 2005.

Appendix

In this appendix, we show the Promela model, used to analyze the example of Section 3 for the test case (e|d).

```
#define TRUE   1
#define FALSE  0
#define UNDEF  255

#define IF if ::
#define FI :: else fi

#define FOR(i,l,h)    i = l ; do :: i < h ->
#define ROF(i,l,h)    ; i++ :: i >= h -> break od

#define HEAPSIZE    8
#define INITQUEUESIZE 9
#define ENQUEUESIZE 9
#define DEQUEUESIZE 12
#define RETVALSIZE 2

#define malloc(X)  X = cur ; cur++
#define free(X)    \
        atomic{ next[X] = UNDEF ;  value[X] = UNDEF}

byte   next[HEAPSIZE]; /* model of the heap */
byte   value[HEAPSIZE];
byte   cur = 0;

byte   head = UNDEF;  /* the queue structure */
byte   tail = UNDEF;
bit    headlock = 1;
bit    taillock = 1;

                  /* stores output from dequeue */
byte   retval[RETVALSIZE];

byte   i = 0 ;

proctype initqueue() {
    bit  done[INITQUEUESIZE];
    byte dummy ;
    do
    :: atomic{!done[1] ->
           malloc(dummy) ; done[1] = TRUE}
    :: atomic{!done[2] && done [1] ->
           next[dummy] = UNDEF ; done[2] = TRUE}
    :: atomic{!done[3] && done [1] ->
           value[dummy] = UNDEF ; done[3] = TRUE}
    :: atomic{!done[4] && done [1] ->
           head = dummy ; done[4] = TRUE}
    :: atomic{!done[5] && done [1] ->
           tail = dummy ; done[5] = TRUE}
    :: atomic{!done[6] && done [1] ->
           headlock = 1 ; done[6] = TRUE}
    :: atomic{!done[7] && done [1] ->
           taillock = 1 ; done[7] = TRUE}
    :: atomic{done[1] && done [2] && done [3] &&
              done [4] && done [5] && done [6] &&
              done [7] ->
           break}
    od
}

proctype enqueue(byte val) {
```

```
    bit  done[ENQUEUESIZE];
    byte node, queuetail;
    do
    :: atomic{!done[1] ->
           malloc(node) ; done[1] = TRUE}
    :: atomic{!done[2] && done [1] ->
           value[node] = val ; done[2] = TRUE}
    :: atomic{!done[3] && done [1] ->
           next[node] = UNDEF ; done[3] = TRUE}
    :: atomic{!done[4] && taillock == 1 ->
           taillock = 0 ; done[4] = TRUE}
    :: atomic{!done[5] && done[4] ->
           queuetail = tail ; done[5] = TRUE}
    :: atomic{!done[6] && done[2] &&
              done[3] && done[5] ->
           next[queuetail] = node ; done[6] = TRUE}
    :: atomic{!done[7] && done[1] &&
              done[4] && done[5] ->
           tail = node ; done[7] = TRUE}
    :: atomic{!done[8] && done[2] && done[3] &&
              done[6] && done[7] ->
           taillock = 1 ; done[8] = TRUE ; break}
    od
}

proctype dequeue(byte rv) {
    bit  done[DEQUEUESIZE];
    byte node, new_head, tmp;

    atomic{headlock == 1 -> headlock = 0};
    node = head;
    new_head = next[node];
    if
    :: atomic{ new_head == UNDEF ->
        headlock = 1 ; retval[rv] = 0}
    :: new_head != UNDEF ->
        do
        :: atomic{!done[7] ->
               tmp = value[new_head] ; done[7] = TRUE}
        :: atomic{!done[8] && done [7] ->
               retval[rv] = tmp ; done[8] = TRUE}
        :: atomic{!done[9] ->
               head = new_head ; done[9] = TRUE}
        :: atomic{!done[10] && done [8] && done [9] ->
               headlock = 1 ; done[10] = TRUE}
        :: atomic{!done[11] && done [10] ->
               free(node) ; done[11] = TRUE ; break}
        od
    fi
}

init{
    atomic{FOR(i,0,HEAPSIZE)
              next[i] = UNDEF ; value[i] = UNDEF
           ROF(i,0,HEAPSIZE)
    } ;
    run initqueue();
    timeout -> atomic{run enqueue(4) ; run dequeue(0)} ;
    timeout -> assert(retval[0] == 0 || retval[0] == 4)
}
```