COMP3151 Assignment 1

Winston Lin (z5310789) and
Eloise Dossetor (z5020182)

July 16, 2021

1 Introduction

Did you know you can make any number of concurrent read operations of an integer variable wait-free in a read-writer scenario multi-threaded program? A quick search on google would probably suggest the read-copy-update (RCU) method. Although easy to implement, the RCU method has its limitations.

Firstly it has strict requirements on the features offered by the language runtime. The RCU method requires atomic increment/decrement, or an automatic garbage collection mechanism like the one offered by java or ARC in objc.

Secondly it requires atomic pointer/reference assignment to achieve the atomic replacement of the global counter variable; however, references in Java are machine words, so 4/8 bytes long on 32/64 bits machine respectively. This violates the assignment specification that only read/write of single byte are atomic. So unfortunately this method isn't feasible.

Our method is similar to the standard RCU in the sense that more than one copy of the variable is made, but by the reader instead of the writer.

2 Our Algorithm

2.1 What Does it Involve

This method requires:

- A Boolean array (named isEdited) of length n, where n is the number of readers. The Writer uses the bit map to notify all the reader processes that an update has occurred.
- $2n$ copies of the global counter.
  - n of these copies are a local archive of the counter stored by each reader. This local copy represents what the counter looks like the last time they met. The local archive is an important person, often exposed to the threat of assassination in the event that a second write operation started before the archive copy operation has finished.

- The second n copies are a decoy copy, which protects the local archive from reading the counter incorrectly. The decoy's purpose embodies that of a food taster. The decoy tries the food that was prepared for the archive (emperor), to confirm it is safe to eat.

2.2 How it Works

There is one writer only, so right away there is no writer contention and the otherwise necessary writer mutex locking. The writer modifies the global counter variable without any blocking. Then it will set all the bits in the 'isEdited' array to 1, every bit in the array corresponds to a reader process, and by setting the bit to 1, the writer is essentially telling the reader "your local copy of the counter is outdated, you need to read from the global counter again to obtain the latest value."

The reader starts by checking its exclusive isEdited bit, if it's switched on, the reader tries to copy the global counter into the decoy of its local copy then turn the bit off. However, because the counter is multiple bytes, the operation of copying the counter isn't atomic, meaning the writer can modify the counter meanwhile a reader is reading it, thus result in tempered byte(s). To prevent copying an incorrect version of the counter, a temporary decoy array is used. If the writers makes an edit during the read, isEdited will be flagged and counter is not copied over from the decoy array to the readers local copy of counter (See Figure 1). The reader retries the copy procedure until a complete, non-interrupted counter value is read into the decoy.

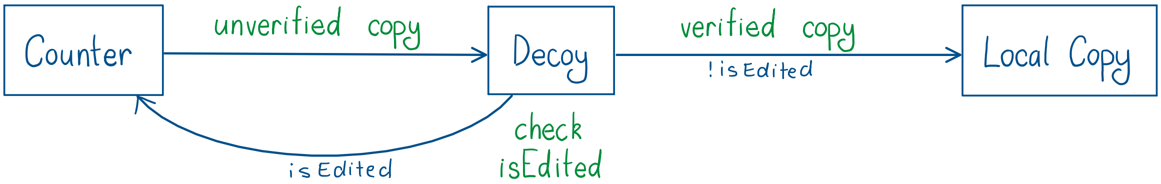


Figure 1: Visual depiction of decoy copy checking counter before passing it on to local copy.

If there is no writes to counter during the reading process, isEdited will not be flagged and thus the decoy array is a verified copy of counter and is transferred into the reader's local copy array.

2.3 Pseudo-code

Main

```
set multiple-byte Counter to 0
set isEdited Boolean array to false
```

Reader

```
$r1$ :  For R rounds or R = 0(infinite loop):
$r2$ :  While isEdited is true:
$r3$ :      set isEdited to false
$r4$ :      copy each byte in Counter into Decoy array
$r5$ :      If edit flag is still false (no edits have been made):
$r6$ :          copy each byte in Decoy array into LocalCopy array
$r7$ :      Else, do not copy, the Counter has been altered (back to $r2$)
```

Writer

```
$w1$ :   For R rounds or R = 0(infinite loop):
$w2$ :       set index to final byte in Counter
$w3$ :       set carry flag to be false
$w4$ :       If last byte if full (all bits == 1):
$w5$ :           set the the last byte to 0
$w6$ :           index– (decrement index)
$w7$ :           set the carry flag to true
$w8$ :       Else:
$w9$ :           If carry flag is true, set it to false
$w10$ :          increment Counter byte at this index

$w11$ :      If carry is true and index is $\geq 0$, go to $w4$
$w12$ :      While carry is true and index $> 0$
$w13$ :          set each element in isEdited array to true
```

2.4 Assumptions

- under weak fairness, even if writes repeatedly interrupt the copy process and causes the reader to retry, reads complete eventually.

  If the reader is allowed to read, it eventually succeeds. doesn't suffer starvation.

2.5 Limitations

- main trade-off is in memory usage, there are $2n$ copies of the counter.
- extra work for read operations.
- Reader abandons entire local copy and retries if writer has made an edit.
- in the scenario where the writer sets the 'isEdited' flag, writer sees the flag, turns off the bit, and finishes the read operation, then writer proceeds to modifies the counter. Now the flag is off and value is changed since the reader last sees it. If writer doesn't write again, reader will not update the local copy.

3 Proof of Correctness

3.1 Spin

3.1.1 Deadlock freedom

To verify deadlock freedom, we used spin safety + invalid end state-options, other options untouched. (See Appendix B).

3.1.2 Counter correctness

A counter value v read by a reader r is considered correct if v is actually assumed by c during the time interval while the counter is being read by r.

To verify this we used a ghost variable named “helper” for each reader process to obtain a correct copy of the counter atomically using the “atomic ” Spin statement, and then after the read has completed, check if this saved helper variable is the same as the program retrieved value.

This assertion was verified using spin safety + assertion violation options, other options untouched (See Appendix B).

3.1.3 starvation freedom

Under weak fairness, reads complete eventually even if writes subside. To verify this, we used the a LTL formula:

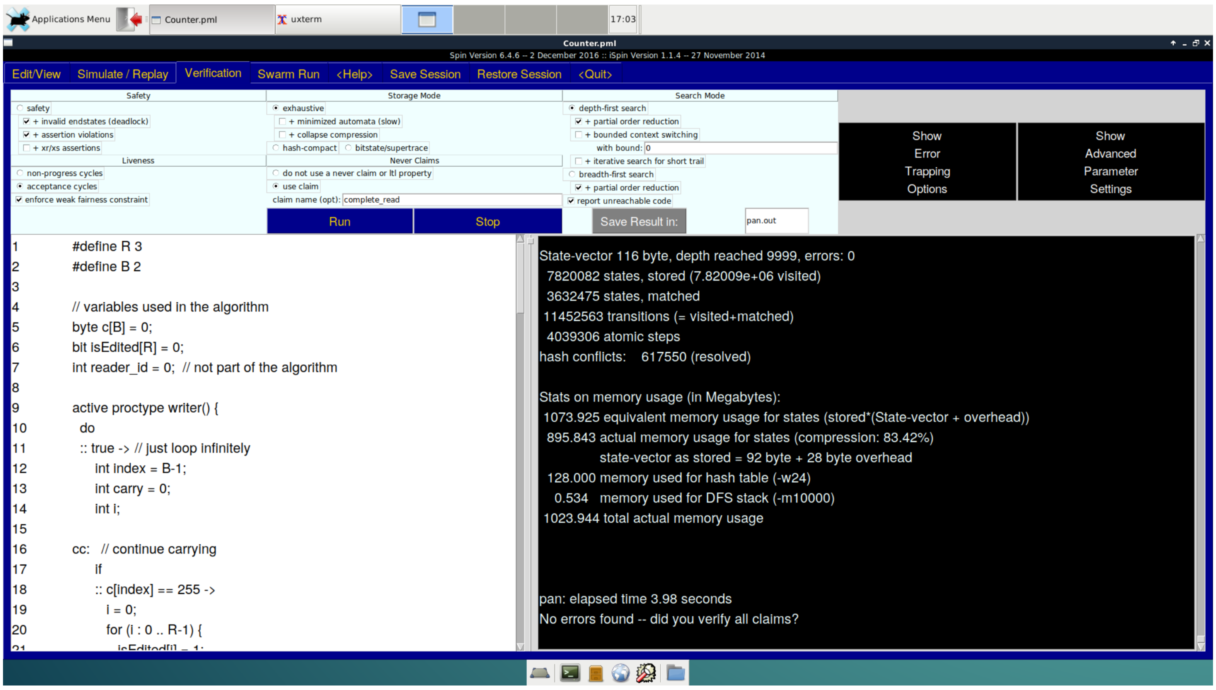
$$\text{ltl eventual\_entry } \{\ \Box\ ((\text{reader}[1]\text{sr}) \implies \Diamond\ (\text{reader}[1]\text{rc}))\}$$

where sr stands for “start read” and rc stands for “read complete”.

The LTL formula was verified in spin using the liveness + acceptance cycle + enforce weak fairness constraint (See Appendix A).

Appendices

A Spin verification of liveness properties



Safety
○ safety
☑ + invalid endstates (deadlock)
☑ + assertion violations
☐ + xr/xs assertions

Liveness
○ non-progress cycles
● acceptance cycles
☑ enforce weak fairness constraint

Storage Mode
● exhaustive
☐ + minimized automata (slow)
☐ + collapse compression
○ hash-compact ○ bitstate/supertrace

Never Claims
○ do not use a never claim or ltl property
● use claim
claim name (opt): complete_read

Search Mode
● depth-first search
☑ + partial order reduction
☐ + bounded context switching
with bound: 0
☐ + iterative search for short trail
○ breadth-first search
☑ + partial order reduction
☑ report unreachable code

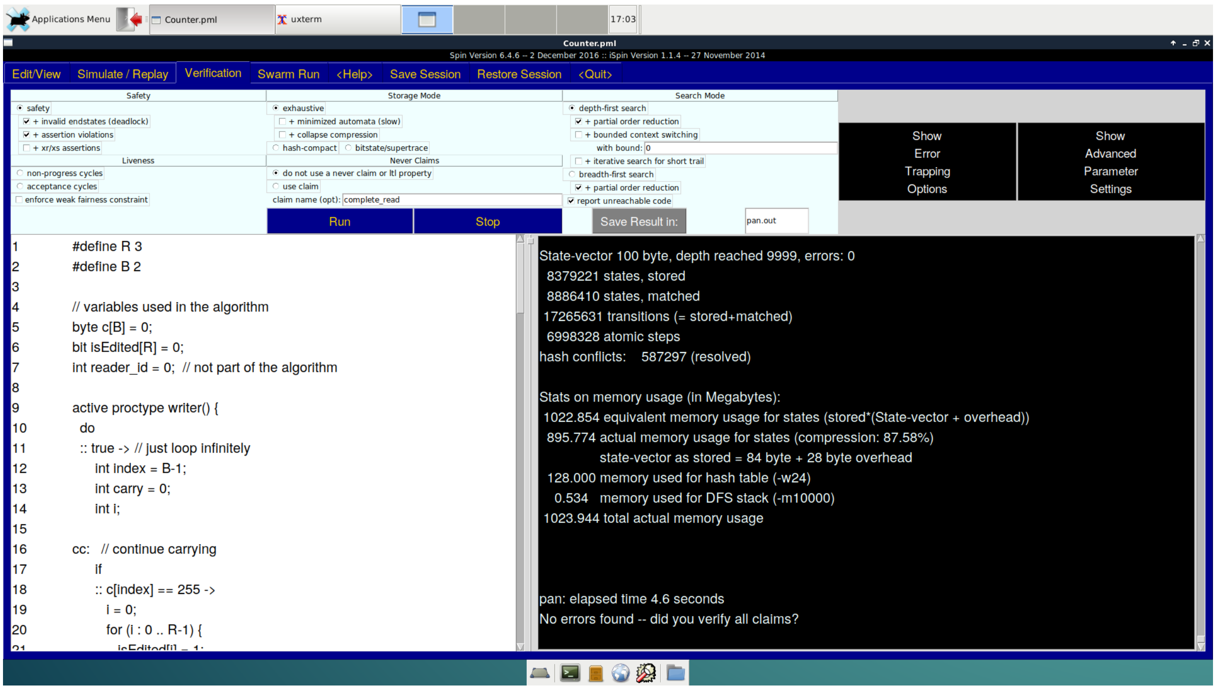
Save Result in: pan.out

```
#define R 3
#define B 2

// variables used in the algorithm
byte c[B] = 0;
bit isEdited[R] = 0;
int reader_id = 0;  // not part of the algorithm

active proctype writer() {
  do
  :: true -> // just loop infinitely
      int index = B-1;
      int carry = 0;
      int i;

cc:   // continue carrying
      if
      :: c[index] == 255 ->
         i = 0;
         for (i : 0 .. R-1) {
            isEdited[i] = 1;
```

```
State-vector 116 byte, depth reached 9999, errors: 0
 7820082 states, stored (7.82009e+06 visited)
 3632475 states, matched
11452563 transitions (= visited+matched)
 4039306 atomic steps
hash conflicts:    617550 (resolved)

Stats on memory usage (in Megabytes):
 1073.925 equivalent memory usage for states (stored*(State-vector + overhead))
  895.843 actual memory usage for states (compression: 83.42%)
         state-vector as stored = 92 byte + 28 byte overhead
 128.000 memory used for hash table (-w24)
   0.534   memory used for DFS stack (-m10000)
 1023.944 total actual memory usage



pan: elapsed time 3.98 seconds
No errors found -- did you verify all claims?
```

Figure 2: Spin verification of liveness properties with acceptance cycles and enforced weak fairness, and R=3, B=2

B Spin verification of safety properties



Safety
● safety
☑ + invalid endstates (deadlock)
☑ + assertion violations
☐ + xr/xs assertions

Liveness
○ non-progress cycles
○ acceptance cycles
☐ enforce weak fairness constraint

Storage Mode
● exhaustive
☐ + minimized automata (slow)
☐ + collapse compression
○ hash-compact ○ bitstate/supertrace

Never Claims
● do not use a never claim or ltl property
○ use claim
claim name (opt): complete_read

Search Mode
● depth-first search
☑ + partial order reduction
☐ + bounded context switching
with bound: 0
☐ + iterative search for short trail
○ breadth-first search
☑ + partial order reduction
☑ report unreachable code

Save Result in: pan.out

```
#define R 3
#define B 2

// variables used in the algorithm
byte c[B] = 0;
bit isEdited[R] = 0;
int reader_id = 0;  // not part of the algorithm

active proctype writer() {
  do
  :: true -> // just loop infinitely
      int index = B-1;
      int carry = 0;
      int i;

cc:   // continue carrying
      if
      :: c[index] == 255 ->
         i = 0;
         for (i : 0 .. R-1) {
            isEdited[i] = 1;
```

```
State-vector 100 byte, depth reached 9999, errors: 0
 8379221 states, stored
 8886410 states, matched
17265631 transitions (= stored+matched)
 6998328 atomic steps
hash conflicts:    587297 (resolved)

Stats on memory usage (in Megabytes):
 1022.854 equivalent memory usage for states (stored*(State-vector + overhead))
  895.774 actual memory usage for states (compression: 87.58%)
         state-vector as stored = 84 byte + 28 byte overhead
 128.000 memory used for hash table (-w24)
   0.534   memory used for DFS stack (-m10000)
 1023.944 total actual memory usage



pan: elapsed time 4.6 seconds
No errors found -- did you verify all claims?
```

Figure 3: Spin verification of Deadlock freedom and assertions with R=3, B=2.