COMP3151 Assignment 1

Michael Cunanan (z5204816), Kenvin Yu (z5207857)

1 Summary

Our algorithm solves the 1 writer and $R > 0$ readers problem for a shared cyclic counter that cannot be atomically read or updated, and without the use of concurrency primitives. The counter consists of B bytes, where individual reads and writes to each byte is atomic and sequentially consistent. We define solved to mean that the algorithm is functionally correct, that is, reads will assume an actual value represented by the counter during the time interval in which the read operation was performed. The algorithm we present utilises 2 buffers that each represents the counter value and 2 parity bits. Readers and writers interact with these 2 buffers in opposing directions (more details later) and the parity bits are flipped whenever the counter value wraps around to 0. Our algorithm is implemented in Promela and Java, with the Java implementation being an almost-faithful port of the Promela code.

2 Assumptions

Our algorithm makes use of some assumptions.

- Reads and writes of individual bytes are atomic.
- We require that reads and writes to bytes are sequentially consistent, which is necessary to ensure that read and write operations take place in the specified order.
- Our algorithm requires the scheduler to have weak fairness in order to satisfy the property that reads will eventually complete.
- In the definition of functional correctness, the reader can assume any value of the counter in a time interval during which the read is performed. Our algorithm assumes that the start the time interval is defined as the first instance of time during the read operation where any shared data is read (including parity bits). Similarly, the end of the time interval is defined as the last instance of time where a shared data is read.

3 Algorithm Details

The shared variables used in this algorithm are

- Two byte arrays, each of length B, to hold two copies of the counter value, named **c** and **d**. The arrays have the most significant byte on the left (lowest index).
- Two parity bits, named p_1 and p_2.

The writer has the following procedure to perform for every write operation:

1. Read from either parity bits (p_1, p_2) and store in a local variable q. For example, $q \leftarrow p1$.
2. Read from any of the two arrays in any direction and store in a local variable **e**. For example, $\mathbf{e} \leftarrow \mathbf{c}$ from left to right. Note that there is only one writer so this step cannot conflict with another write operation.
3. Increment **e** by 1. In the case of a wrap-around, set **e** to the 0 value and flip the local parity bit q.
4. Write $p_2 \leftarrow q$.
5. Write $\mathbf{d} \leftarrow \mathbf{e}$ from left to right (most to least significant byte).
6. Write $\mathbf{c} \leftarrow \mathbf{e}$ from right to left (least to most significant byte).
7. Write $p_1 \leftarrow q$.

The readers have the following procedure to perform for every read operation:

1. Initialise a local byte array $\mathbf{v}$ of size B with zeroes.
2. Read $q_1 \leftarrow p_1$, where q_1 is a local variable. [Start of time interval for func. correctness]
3. Read $\mathbf{s} \leftarrow \mathbf{c}$ from left to right (most to least significant byte).
4. Read $\mathbf{t} \leftarrow \mathbf{d}$ from right to left (least to most significant byte).
5. Read $q_2 \leftarrow p_2$, where q_2 is a local variable. [End of time interval for func. correctness]
6. If $q_1 \neq q_2$, then $\mathbf{v}$ is the return value (which is 0).
7. Otherwise, reading from left to right, find the shortest non-matching prefix of $\mathbf{s}$ and $\mathbf{t}$. Denote these sub-arrays as $\mathbf{s}_{1,k}$ and $\mathbf{t}_{1,k}$ respectively where k is the inclusive index of the non-matching byte. If there is no non-matching prefix, i.e. $\mathbf{s} = \mathbf{t}$, then we will consider the entire array with $k = B$. Assign $\mathbf{v}_{1,k} = \mathbf{t}_{1,k}$ so that the bytes strictly after the k-th index for $\mathbf{v}$ are still 0. Return this $\mathbf{v}$.

Some of the steps (e.g Writer step 3, Reader step 7) are kept vague as they can be implemented very differently in different languages, though we require implementations of these steps to not block or use awaits. For more implementation details and comments, please see the accompanying Promela and Java files.

The intuition behind the opposing read/write directions is because if they were instead in the same direction, then the reader and writer may "leap-frog" each other which results in degenerate inter-leavings. With opposing directions, the read and writes do not have this behaviour. We applied this "opposing direction" idea to the read/write at both the array-level and the individual byte level, as well as for the parity bits. The parity bits allow us to simplify the problem of a cyclic counter to one with a monotonic counter.

4 Functional Correctness

4.1 Notation

We will use the variables defined in 3 Algorithm Details.

- Denote $\mathbf{c}^{(i)}$ to mean the i-th version of $\mathbf{c}$ since initialisation, which is $\mathbf{c}^{(0)} = \mathbf{0}$.
- We can decompose the array as a B-tuple, $\mathbf{c} = (\mathbf{c}_1, \ldots, \mathbf{c}_B)$, noting that $\mathbf{c}_1$ is the most significant byte and $\mathbf{c}_B$ is the least significant byte.
- Denote sub-arrays as $\mathbf{c}_{j,k}$ which corresponds to $(\mathbf{c}_j, \mathbf{c}_{j+1}, \ldots, \mathbf{c}_k)$.
- For comparison between arrays, we will use $\leq_{\text{lex}}$.

Of course we can combine notation, so $\mathbf{c}^{(i)}_{j,k} = (\mathbf{c}^{(i)}_j, \mathbf{c}^{(i)}_{j+1}, \ldots, \mathbf{c}^{(i)}_k)$ means the j to k sub-array of the i-th version of $\mathbf{c}$.

We begin by proving some lemmas for later use.

4.2 Prefix Lemma

Suppose we have two B-tuples $\mathbf{x}$ and $\mathbf{y}$. We can use the aforementioned notation to define prefixes as $\mathbf{x}_{1,k}$ (the first k elements of $\mathbf{x}$. We also use the following definition of $\leq_{\text{lex}}$:

$$\begin{aligned} \mathbf{x} <_{\text{lex}} \mathbf{y} &\iff \exists i \in [1, B] : (\mathbf{x}_1 = \mathbf{y}_1) \wedge \cdots \wedge (\mathbf{x}_{i-1} = \mathbf{y}_{i-1}) \wedge (\mathbf{x}_i < \mathbf{y}_i) \\ \mathbf{x} \leq_{\text{lex}} \mathbf{y} &\iff (\mathbf{x} = \mathbf{y}) \vee (\mathbf{x} <_{\text{lex}} \mathbf{y}) \end{aligned}$$

We claim that if $\mathbf{x} \leq_{\text{lex}} \mathbf{y}$, then $\mathbf{x}_{1,n} \leq \mathbf{y}_{1,n}$ for all $n \in [1, B]$.

If $\mathbf{x} = \mathbf{y}$, then we are done. Otherwise, let the first index where $\mathbf{x}$ and $\mathbf{y}$ are different to be N, i.e.

$$\mathbf{x}_1 = \mathbf{y}_1,\ \mathbf{x}_2 = \mathbf{y}_2,\ \ldots,\ \mathbf{x}_{N-1} = \mathbf{y}_{N-1},\ \mathbf{x}_N < \mathbf{y}_N.$$

For any $k \in [1, N)$, we have $\mathbf{x_1} = \mathbf{y_1}, \ldots, \mathbf{x_k} = \mathbf{y_k}$, hence $\mathbf{x}_{1,k} = \mathbf{y}_{1,k}$, and therefore $\mathbf{x}_{1,k} \leq_{\text{lex}} \mathbf{y}_{1,k}$.

For any $k \in [N, B]$, we have $\mathbf{x}_1 = \mathbf{y}_1, \ldots, \mathbf{x}_{N-1} = \mathbf{y}_{N-1}, \mathbf{x}_N < \mathbf{y}_N$, which implies $\mathbf{x}_{1,k} \leq_{\text{lex}} \mathbf{y}_{1,k}$ by definition as $N \in [1, k]$.

Hence in either case, $\mathbf{x}_{1,k} \leq_{\text{lex}} \mathbf{y}_{1,k}$. This has the useful interpretation that if $\mathbf{x} \leq_{\text{lex}} \mathbf{y}$, then this is also true for their prefixes of matching lengths.

4.3 Ordering Lemma

Suppose we have two processes operating on bytes x and y:

```
P():                     | Q():
for $i \in \mathbb{N}$   |   read $x$
  $y \leftarrow y^{(i)}$ |   read $y$
  $x \leftarrow x^{(i)}$ |
```

where process P updates y and x with increasing version numbers. Suppose process Q reads in versions $x^{(a)}$ and $y^{(b)}$ for x and y respectively. We claim that $a \leq b$.

Because x was read by process Q to be $x^{(a)}$, this read must have occurred sometime after the assignment $x \leftarrow x^{(a)}$ (under the assumption that writing to a single byte is atomic, it cannot happen *during* the write). Note that $y \leftarrow y^{(a)}$ precedes $x \leftarrow x^{(a)}$ in P, so any further writes of y must attain a higher version number. Since Q reads y after x, then y cannot have a lower version number than the x which Q read in, which was version a. It follows that Q must have read a version of y with a version number that was greater than or equal to the version number of x that it read in. Hence $a \leq b$.

4.4 Bounded Lemma

Suppose we read some vector $\mathbf{c}$ while it was being updated. Call the read-in value $\mathbf{s}$, defined by

$$\mathbf{s} := \left(\mathbf{c}_1^{(i_1)}, \ldots, \mathbf{c}_B^{(i_B)}\right).$$

Suppose that $i_1 \leq i_2 \leq \cdots \leq i_B$ and that $\{\mathbf{c}^{(i)}\}$ is monotonically increasing, i.e.

$$\mathbf{c}^{(0)} \leq_{\text{lex}} \mathbf{c}^{(1)} \leq_{\text{lex}} \mathbf{c}^{(2)} \leq_{\text{lex}} \cdots$$

We claim that $\mathbf{s} \leq_{\text{lex}} \mathbf{c}^{(i_B)}$.

First, fix version i_B. We will try to maximise the expression $\left(\mathbf{c}_1^{(i_1)}, \ldots, \mathbf{c}_B^{(i_B)}\right)$ by choosing $i_1, i_2, \ldots, i_{B-1}$ under the constraint $i_1 \leq i_2 \cdots \leq i_B$, using lexicographic ordering $\leq_{\text{lex}}$. Note that $\mathbf{s} = \mathbf{c}^{(i_B)}$ is possible (with $i_1 = i_2 = \cdots = i_B$), so we should reject any choices for $i_1, \ldots, i_{B-1}$ that would result in $\mathbf{s} <_{\text{lex}} \mathbf{c}^{(i_B)}$. We proceed with induction.

We will first choose version i_1. By the Prefix Lemma, we know that $i_1 \leq i_B$ implies $\mathbf{c}_1^{(i_1)} \leq \mathbf{c}_1^{(i_B)}$ (prefixes of length 1). If we choose i_1 such that $\mathbf{c}_1^{(i_1)} < \mathbf{c}_1^{(i_B)}$, then we would have $\mathbf{s} <_{\text{lex}} \mathbf{c}^{(i_B)}$ and so we reject such a choice. It follows that we should choose i_1 such that $\mathbf{c}_1^{(i_1)} = \mathbf{c}_1^{(i_B)}$.

Assume we have chosen version i_k for $k \in [1, B)$, such that $\mathbf{c}_{1,k}^{(i_k)} = \mathbf{c}_{1,k}^{(i_B)}$.

Let us now choose version i_{k+1}. We know by the Prefix Lemma that $i_k \leq i_{k+1} \leq i_B$ implies

$$\mathbf{c}_{1,k}^{(i_k)} \leq \mathbf{c}_{1,k}^{(i_{k+1})} \leq \mathbf{c}_{1,k}^{(i_B)} \quad \text{and} \quad \mathbf{c}_{1,k+1}^{(i_{k+1})} \leq \mathbf{c}_{1,k+1}^{(i_B)}.$$

Under the assumption $\mathbf{c}_{1,k}^{(i_k)} = \mathbf{c}_{1,k}^{(i_B)}$ however, we can deduce that $\mathbf{c}_{1,k}^{(i_{k+1})} = \mathbf{c}_{1,k}^{(i_B)}$. Hence if we chose i_{k+1} such that $\mathbf{c}_{k+1}^{(i_{k+1})} > \mathbf{c}_{k+1}^{(i_B)}$, this would imply $\mathbf{c}_{1,k+1}^{(i_{k+1})} > \mathbf{c}_{1,k+1}^{(i_B)}$ which is a contradiction to the result from the Prefix Lemma. It follows that we must have $\mathbf{c}_{k+1}^{(i_{k+1})} \leq \mathbf{c}_{k+1}^{(i_B)}$. If we choose i_{k+1} such that $\mathbf{c}_{k+1}^{(i_{k+1})} < \mathbf{c}_{k+1}^{(i_B)}$, we would have $\mathbf{s} <_{\text{lex}} \mathbf{c}^{(i_B)}$, so we should reject such a choice and instead choose i_{k+1} such that $\mathbf{c}_{k+1}^{(i_{k+1})} = \mathbf{c}_{k+1}^{(i_B)}$, leading to $\mathbf{c}_{1,k+1}^{(i_{k+1})} = \mathbf{c}_{1,k+1}^{(i_B)}$.

Hence by induction we have shown that the 'best' choices for $i_1, i_2, \ldots i_{B-1}$ under the constraint $i_1 \leq \cdots \leq i_B$ leads to $\mathbf{c}_{1,k}^{(i_k)} = \mathbf{c}_{1,k}^{(i_B)}$ for all $k \in [1, B)$, which has the implication $\mathbf{c}_k^{(i_k)} = \mathbf{c}_k^{(i_B)}$ for all $k \in [1, B)$. Recalling that i_B was fixed, we have shown that the maximum value for $\mathbf{s}$ must be one that agrees with $\mathbf{c}^{(i_B)}$ on all entries, which can only be $\mathbf{c}^{(i_B)}$ itself. Hence $\mathbf{s} \leq_{\text{lex}} \mathbf{c}^{(i_B)}$.

As a corollary, if instead $i_1 \geq i_2 \cdots \geq i_B$, then $\mathbf{s} \geq_{\text{lex}} \mathbf{c}^{(i_B)}$. This can be proved by repeating the above proof, but trying to minimize $\mathbf{s}$ instead.

4.5 Parity Bits and Overflow

As part of the algorithm, we keep two 'parity' bits p_1 and p_2 (read into q_1 and q_2 respectively) to make conclusions about 'overflow'.

We define overflow to be the act of the counter wrapping around the maximum value, back to an array of 0's. We claim that if the reader reads different values for q_1 and q_2, then the counter must have overflowed in between these reads. Specifically, a counter is acknowledged to have overflowed when p_2 is assigned a different value than before (since writing to p_2 is the start of the write).

Note that the writer is the only process that can change the parity bits p_1 and p_2. Specifically, each write operation updates in the order of p_2 then p_1, so that $p_1^{(i)} = p_2^{(i)}$. The reader reads these in the opposite order. This allows us to use the Ordering Lemma.

During a single read operation, the reader reads $q_1 \leftarrow p_1^{(a)}$ then $q_2 \leftarrow p_2(b)$. By the Ordering Lemma, we have that $a \leq b$ which means we know that the version did not decrease from reading p_1 to reading p_2. Now $p_1^{(a)} = p_2^{(a)}$ follows from how the parity bits are written and so we can just consider $p_2^{(a)}$ and $p_2^{(b)}$. By the definition of overflow, we can deduce that there were n overflows in the read time interval if and only if p_2 was flipped n times in the same time interval. When n is odd, then we have that $p_2^{(b)} \neq p_2^{(b)}$ since bits only have 2 states. Hence, we can deduce that $q_1 \neq q_2$ if and only if there is an odd number of overflows in the read time interval.

So, we have two cases:

- $q_1 \neq q_2$: Having unequal values means that in between reading p_1 and p_2, there were an odd number of overflows. In the smallest case, 1 overflow, it follows from the definition of an overflow that the counter at some point represented an entirely zero value. Hence, returning $\mathbf{0}$ is functionally correct. The same behaviour is also trivially correct in the case of an higher odd number.
- $q_1 = q_2$. Having equal values means that in between reading p_1 and p_2, there were an even number of overflows (possibly 0). If 2 or more overflows occurred during a read, then the counter must have looped through all possible values, hence any returned value makes the algorithm functionally correct. We use the remainder of this section to examine the case of 0 overflows. That is, we have reduced the problem to a monotonic counter.

4.6 Monotonic Counter Proof

As explained by the previous section, from now on we only need to consider the case of 0 overflows occurring during a read. If the counter does not overflow, then this is equivalent to the counter monotonically increasing during the period of the read. Recall that our counter was represented by two buffers $\mathbf{c}$ and $\mathbf{d}$, so we can represent this statement as

$$\mathbf{c}^{(0)} \leq_{\text{lex}} \mathbf{c}^{(1)} \leq_{\text{lex}} \mathbf{c}^{(2)} \dots$$
$$\mathbf{d}^{(0)} \leq_{\text{lex}} \mathbf{d}^{(1)} \leq_{\text{lex}} \mathbf{d}^{(2)} \dots$$

and at the end of each write, the buffers are assigned the same value so, $\mathbf{c}^{(i)} = \mathbf{d}^{(i)}$ for all i.

We read in the values from $\mathbf{c}$ and $\mathbf{d}$ into $\mathbf{s}$ and $\mathbf{t}$ respectively with the following result

$$\mathbf{s} := \left(\mathbf{c}_1^{(i_1)}, \dots, \mathbf{c}_B^{(i_B)}\right),$$
$$\mathbf{t} := \left(\mathbf{d}_1^{(j_1)}, \dots, \mathbf{d}_B^{(j_B)}\right)$$

where $i_1, \dots, i_B, j_1, \dots, j_B \geq 0$ are version numbers.

We defined the start of the reading interval to occur at the read of $q_1 \leftarrow p_1$. Suppose when we read p_1, $\mathbf{c}$ is at version i_0. Observe that the writer will write $\mathbf{c}_1$ then p_1, whereas the reader will read from p_1 then $\mathbf{c}_1$. By the Ordering

Lemma, this implies $i_0 \leq i_1$.

Similarly, we defined the end of a reading interval to occur at the read $q_2 \leftarrow p_2$. Suppose when we read p_2, $\mathbf{d}$ is at version j_0. Observe that the writer will write p_2 then $\mathbf{d}_1$, whereas the reader will read from $\mathbf{d}_1$ then p_2. By the Ordering Lemma, this implies $j_1 \leq j_0$.

Recall that we defined functional correctness as the reader 'returning' any value that the counter obtained during the reading interval. Since the counter was at version i_0 at the start of the read, at version j_0 at the end of the read, and the counter is monotonically increasing, we require our returned value $\mathbf{v}$ to satisfy the following inequality:

$$\mathbf{c}^{(i_0)} \leq_{\text{lex}} \mathbf{s} \leq_{\text{lex}} \mathbf{d}^{(j_0)}$$

Observe that for any $1 \leq k < B$, the writer will write $\mathbf{c}_{k+1}$ followed by $\mathbf{c}_k$, whereas the reader will read from $\mathbf{c}_k$ then $\mathbf{c}_{k+1}$. By the Ordering Lemma, this implies that $i_k \leq i_{k+1}$ for all $1 \leq k < B$ or equivalently $i_1 \leq i_2 \cdots \leq i_B$. By the Bounded Lemma, this implies $\mathbf{s} \leq_{\text{lex}} \mathbf{c}^{(i_B)}$.

Similarly, observe that for any $1 \leq k < B$, the writer will write $\mathbf{d}_k$ followed by $\mathbf{d}_{k+1}$, whereas the reader will read from $\mathbf{d}_{k+1}$ then $\mathbf{d}_k$. By the Ordering Lemma, this implies that $j_{k+1} \leq j_k$ for all $1 \leq k < B$ or equivalently $j_1 \geq j_2 \cdots \geq j_B$. By the corollary of the Bounded Lemma, this implies $\mathbf{t} \geq_{\text{lex}} \mathbf{d}^{(j_B)}$.

Lastly, notice that the writer will write $\mathbf{d}_B$ followed by $\mathbf{c}_B$, whereas the reader will read from $\mathbf{c}_B$ then $\mathbf{d}_B$. By the Ordering Lemma, this implies that $i_B \leq j_B$, and since the counter is monotonically increasing, we have $\mathbf{c}^{(i_B)} = \mathbf{d}^{(i_B)} \leq \mathbf{d}^{(j_B)}$.

We can combine these conclusions to get

$$\mathbf{s} \leq_{\text{lex}} \mathbf{c}^{(i_B)} \leq_{\text{lex}} \mathbf{d}^{(j_B)} \leq_{\text{lex}} \mathbf{t} \quad (*)$$

We proceed with two cases.

4.6.1 $\mathbf{s} = \mathbf{t}$

We claim that in this scenario, returning $\mathbf{v} := \mathbf{t}$ is functionally correct. From $(*)$,

$$\mathbf{s} = \mathbf{t} \implies \mathbf{s} = \mathbf{c}^{(i_B)} = \mathbf{d}^{(j_B)} = \mathbf{t}$$

Importantly, we can establish the following inequalities (under monotonicity of $\mathbf{c}$ and $\mathbf{d}$):

$$\mathbf{c}^{i_0} \leq_{\text{lex}} \mathbf{c}^{i_B} \leq_{\text{lex}} \mathbf{c}^{j_B} = \mathbf{t} = \mathbf{d}^{j_B} \leq_{\text{lex}} \mathbf{d}^{j_0}.$$

Hence $\mathbf{t}$ satisfies the requirements for functional correctness.

4.6.2 $\mathbf{s} < \mathbf{t}$

By $(*)$, we know that $\mathbf{s} \neq \mathbf{t} \implies \mathbf{s} <_{\text{lex}} \mathbf{t}$.

Let $N \in [1, B]$ be the index of the first mismatch between $\mathbf{s}$ and $\mathbf{t}$ which must exist under the assumption that they are not equal. Formally,

$$\mathbf{s}_1 = \mathbf{t}_1, \mathbf{s}_2 = \mathbf{t}_2, \ldots, \mathbf{s}_{N-1} = \mathbf{t}_{N-1}, \mathbf{s}_N < \mathbf{t}_N$$

Using the definitions of $\mathbf{s}_k$ and $\mathbf{t}_k$, we can restate each equality as $\mathbf{c}_k^{(i_k)} = \mathbf{d}_k^{(j_k)}$ for all $k \in [1, N)$, which can be further simplified to $\mathbf{c}_k^{(i_k)} = \mathbf{c}_k^{(j_k)}$ since $\mathbf{c}^{(i)} = \mathbf{d}^{(i)}$ for all i.

We claim that for all $k \in [1, N)$, $\mathbf{c}_k^{(i_k)} = \cdots = \mathbf{c}_k^{(i_B)} = \mathbf{c}_k^{(j_B)} = \cdots = \mathbf{c}_k^{(j_k)}$.

For $k = 1$, the Prefix Lemma implies that

$$\mathbf{c}_1^{(i_1)} \leq \mathbf{c}_1^{(i_2)} \leq \ldots \mathbf{c}_1^{(j_1)}$$

and because $\mathbf{c}_1^{(i_1)} = \mathbf{c}_1^{(j_1)}$, we have $\mathbf{c}_1^{(i_1)} = \mathbf{c}_1^{(i_2)} = \cdots = \mathbf{c}_1^{(j_1)}$.

Suppose that $\mathbf{c}_n^{(i_n)} = \mathbf{c}_n^{(i_n+1)} = \cdots = \mathbf{c}_n^{(i_B)} = \mathbf{c}_n^{(j_B)} = \cdots = \mathbf{c}_n^{(j_n)}$ is true for all $n \in [1,k]$ for some $k < N-1$. For $n = k+1$, we can use the Prefix Lemma to get

$$\mathbf{c}_{1,k+1}^{(i_{k+1})} \leq_{\text{lex}} \mathbf{c}_{1,k+1}^{(i_{k+1}+1)} \leq_{\text{lex}} \ldots \leq_{\text{lex}} \mathbf{c}_{1,k+1}^{(j_{k+1})}$$

By the induction hypothesis, we have

$$\begin{aligned} \mathbf{c}_1^{(i_1)} = \mathbf{c}_1^{(i_2)} = \cdots = \mathbf{c}_1^{(j_1)} &\implies \mathbf{c}_1^{(i_{k+1})} = \mathbf{c}_1^{(j_{k+1})} \\ \mathbf{c}_2^{(i_2)} = \mathbf{c}_2^{(i_3)} = \cdots = \mathbf{c}_2^{(j_2)} &\implies \mathbf{c}_1^{(i_{k+1})} = \mathbf{c}_1^{(j_{k+1})} \\ &\cdots \\ \mathbf{c}_k^{(i_k)} = \mathbf{c}_k^{(i_{k+1})} = \cdots = \mathbf{c}_k^{(j_k)} &\implies \mathbf{c}_k^{(i_{k+1})} = \mathbf{c}_k^{(j_{k+1})} \end{aligned}$$

Or in short, $\mathbf{c}_{1,k}^{(i_{k+1})} = \mathbf{c}_{1,k}^{(j_{k+1})}$.

We know however that $\mathbf{c}_{k+1}^{(i_{k+1})} = \mathbf{c}_{k+1}^{(j_{k+1})}$, therefore $\mathbf{c}_{1,k+1}^{(i_{k+1})} = \mathbf{c}_{1,k+1}^{(j_{k+1})}$ and

$$\mathbf{c}_{1,k+1}^{(i_{k+1})} = \mathbf{c}_{1,k+1}^{(i_{k+2})} = \cdots = \mathbf{c}_{1,k+1}^{(j_{k+1})}$$

and so $\mathbf{c}_{k+1}^{(i_{k+1})} = \mathbf{c}_{k+1}^{(i_{k+2})} = \ldots \mathbf{c}_{k+1}^{(j_{k+1})}$. This completes the induction.

The induction provides the fact that

$$\forall k \in [1,N) : \mathbf{c}_k^{i_k} = \mathbf{c}_k^{i_N} = \mathbf{c}_k^{j_N} = \mathbf{c}_k^{j_k}$$

which implies

$$\mathbf{s}_{1,N-1} = \mathbf{c}_{1,N-1}^{(i_N)} = \mathbf{d}_{1,N-1}^{(j_N)} = \mathbf{t}_{1,N-1}.$$

Recalling that $\mathbf{s}_N = \mathbf{c}_N^{i_N}$, $\mathbf{t}_N = \mathbf{d}_N^{j_N}$ and $\mathbf{s}_N < \mathbf{t}_N$, we get

$$\mathbf{s}_{1,N} = \mathbf{c}_{1,N}^{i_N} < \mathbf{d}_{1,N}^{j_N} = \mathbf{t}_{1,N}$$

Because $j_N \leq j_1 \leq j_0$, we have

$$\begin{aligned} \mathbf{d}^{(j_0)} &\geq_{\text{lex}} \mathbf{d}^{(j_N)} \\ &= \left(\mathbf{d}_1^{(j_N)}, \ldots, \mathbf{d}_N^{(j_N)}, \mathbf{d}_{N+1}^{(j_N)}, \ldots, \mathbf{d}_B^{(j_N)}\right) && \text{(by definition of } \mathbf{d}^{(j_N)}\text{)} \\ &\geq_{\text{lex}} \left(\mathbf{d}_1^{(j_N)}, \ldots, \mathbf{d}_N^{(j_N)}, 0, \ldots, 0\right) \\ &= (\mathbf{t}_1, \ldots, \mathbf{t}_N, 0, \ldots, 0) := \mathbf{w} \end{aligned}$$

where have effectively 'rounded down' $\mathbf{t}$ to $\mathbf{w}$ by keeping only the first N entries. We claim that returning $\mathbf{v} := \mathbf{w}$ is functionally correct.

We have already shown that $\mathbf{w} \leq_{\text{lex}} \mathbf{d}^{(j_0)}$. Consider the prefixes $\mathbf{c}_{1,N}^{(i_N)} <_{\text{lex}} \mathbf{t}_{1,N}$. By the definition of $<_{\text{lex}}$, no matter how both N-tuples are extended (to the right) into B-tuples, we will always maintain a strict inequality. Therefore

$$\mathbf{t}_{1,N} >_{\text{lex}} \mathbf{c}_{1,N}^{(i_N)} \implies \mathbf{w} >_{\text{lex}} \mathbf{c}^{(i_N)} \geq_{\text{lex}} \mathbf{c}^{(i_0)}$$

Putting these inequalities together gets

$$\mathbf{c}^{(i_0)} <_{\text{lex}} \mathbf{w} \leq_{\text{lex}} \mathbf{d}^{(j_0)}.$$

Hence returning $\mathbf{v} := \mathbf{w}$ is functionally correct.

4.7 Readers not interfering other readers

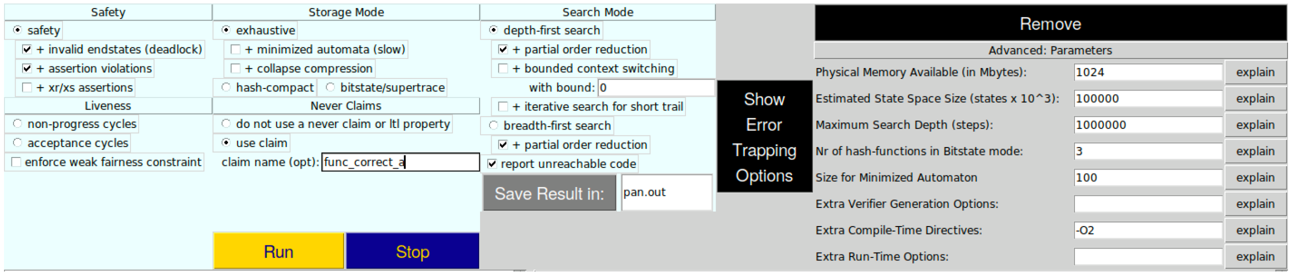
We have so far shown that the program is functionally correct in the case of $R = 1$ reader.

Note that there is still only one writer and this writer is the only process that can write to shared memory. Hence the introduction of additional readers will not affect the functional correctness of the first reader, and vice versa.

5 Promela Verification

We used shared ghost/auxiliary variables to assist in our verification attempts. Our best-effort verification results with our Promela model are as follows:

- Functional correctness of reads was partially verified for small values of R and B (see more under Limitations). We only verified functional correctness for the problematic case of when there is no overflow event during the read interval. The verification options are shown below and the claim, `func_correct_a`, can be changed appropriately.



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): func_correct_a

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

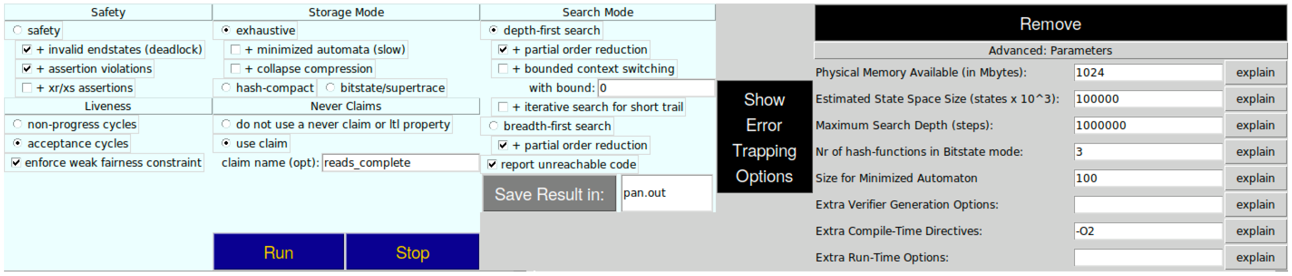
Save Result in: pan.out

Advanced: Parameters

Physical Memory Available (in Mbytes):	1024
Estimated State Space Size (states x 10^3):	100000
Maximum Search Depth (steps):	1000000
Nr of hash-functions in Bitstate mode:	3
Size for Minimized Automaton	100
Extra Verifier Generation Options:	
Extra Compile-Time Directives:	-O2
Extra Run-Time Options:	

- Our algorithm cannot deadlock, though this is a trivial result as there are no blocking statements. This was verified in conjunction with the prior safety property as a built-in option to `spin`.

- Under weak fairness, reads will always complete eventually. Intuitively, this is because the reading operation has no blocking statements and all loops will eventually terminate as they are all for iterating over arrays. The verification options used are shown.



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): reads_complete

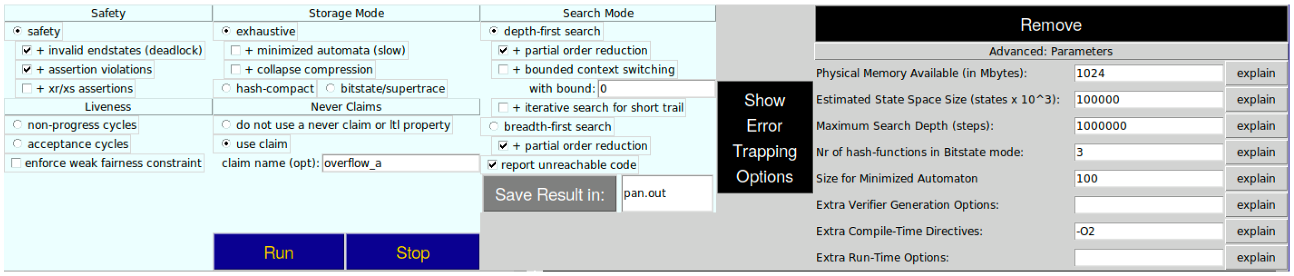
Search Mode
- (●) depth-first search
- [✓] + partial order reduction
- [ ] + bounded context switching
- with bound: 0
- [ ] + iterative search for short trail
- ( ) breadth-first search
- [✓] + partial order reduction
- [✓] report unreachable code

Save Result in: pan.out

Advanced: Parameters

Physical Memory Available (in Mbytes):	1024
Estimated State Space Size (states x 10^3):	100000
Maximum Search Depth (steps):	1000000
Nr of hash-functions in Bitstate mode:	3
Size for Minimized Automaton	100
Extra Verifier Generation Options:	
Extra Compile-Time Directives:	-O2
Extra Run-Time Options:	

- Whenever the reader reads differing values for $q_1 \leftarrow p_1$ and $q_2 \leftarrow p_2$, then we can guarantee that an overflow event occurred during the read time interval (in fact, there would be an odd number of overflows but we do not verify this). The verification options used are shown below and the claim, `overflow_a`, can be changed appropriately.



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): overflow_a

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

Advanced: Parameters

Physical Memory Available (in Mbytes):	1024
Estimated State Space Size (states x 10^3):	100000
Maximum Search Depth (steps):	1000000
Nr of hash-functions in Bitstate mode:	3
Size for Minimized Automaton	100
Extra Verifier Generation Options:	
Extra Compile-Time Directives:	-O2
Extra Run-Time Options:	

Note: Running the model under Simulate is very slow with “Track Data Values” enabled.

6 Limitations

There are a couple limitations that we encountered during the implementation of our algorithm in Java and Promela.

- Our attempts at Promela verification were all depth-limited. We have tried to address this by increasing the maximum depth or introducing an artificial maximum for each byte, but this does not solve the issue. Hence, our Promela verification attempts were best-effort.

- We manually verified the the Promela model only for small integers of R and B because each combination would require slightly different LTL formulas.

- Functional correctness was only partially verified in Promela due to growing technical complexity.

- Java only has signed bytes, which our implementation only makes use of the 0-127 range, whereas Promela has bytes that range over 0-255. This just means that the Java implementation of the cyclic counter will range over $0 \ldots 2^{7B-1}$ instead of $0 \ldots 2^{8B-1}$, though functional correctness still holds.
- Java does not have a bit datatype, so we represented the parity bits each with `int`. This should not be an issue as Java's primitive `int` type is atomic to read and write to.

7 Acknowledgements

We would like acknowledge Leslie Lamport's "Concurrent Reading and Writing (1977)" for the main inspiration of our algorithm. The proofs are our own work.