Counter

Summary:

In this report, we will introduce our approach to solve the unblocked read-writer problem. Firstly, two methods were considered, one is to store carry-over using flags and the other is to use caches to keep critical bytes. And we finally determined the cache version for submission because it's more stable and logically concise. This algorithm works consistently with different parameter combinations of (B, R, k). Also, the promela code passed eventually entry requirements and correctness check using LTL statements.

Problem:

The difficulty for the reader to read the correct counter value is to read every byte one by one while, in the meantime, the writer keeps writing the value and carry may occur. So the reader could end up reading parts of carried(or multi carried) with parts of not carried.

Algorithm:

The core of the algorithm is to make a 'snapshot' when a reader starts to read the counter value.
The reader will first claim its 'snapshot' holder (aka 'backup value' in the code). Then the reader will start to read from the lowest byte first to the highest. The reader will check the flag for each byte before reading it, and if the flag is true, the reader will read the corresponding byte in 'snapshot', if the flag is false, then the reader will read the original counter value and then check the flag again and decide whether to read the 'snapshot' or not. After all the bytes are read, the reader will drop the claim before starting the next turn.

The writer will write the lowest byte normally. When the carry happens, the writer will activate the current 'snapshot', and copy the lowest byte(255) to the snapshot and set the flag[0] to true first and every time a carry happens to a byte, the writer will first copy the current byte to the corresponding byte in 'snapshot', then set the corresponding flags to true. After all the byte is finished writing in a carry write, the writer will check if there are one or more readers currently claiming this active 'snapshot'. If so, the writer will then seal the current 'snapshot' by copying the whole value to it and, activate the next 'snapshot', then reset all the flags.

A few reasoning and hurdles:

Algorithmic

- If there are n readers, then there are most n claimed 'snapshots' at the same time, so the n+1(reason see below) backup array will be initialized at the beginning.
- The readers will always claim the current activated 'snapshot', even if the 'snapshot' is currently being written by the writer.
- The seal for the 'snapshot' was expensive in terms of time cost but considered necessary as the writer should not wait for the reader to finish reading so that in case the counter carried once or more before the read finished, the reading would still be eminently accurate.
- To protect a sealed 'snapshot', the writer will always choose to activate the next unclaimed 'snapshot' (there will be at least one unclaimed 'snapshot'). We use an array of size n to indicate how many current claims for each 'snapshot'.

The advantage of the algorithm is that it is accurate and lock-free. There's no live lock here since the reader will never re-check the previously read bytes. The disadvantage is obviously the time cost, as the sealing process is a significant amount of time-consuming, but we have implemented the following to reduce the chance for it to happen:

- The seal will only occur if carry happens.
- The writer will check if the 'snapshot' is worth sealing by looking at how many claims to the 'snapshot'.

Promela implementations

- 2-d arrays are not supported by Promela, so we choose to create separate variables and processes for each reader. Also, combination of if statements and arrays are used to replace 2-d array operations.
- Cyclic call is not supported by Promela. To solve this problem, we used a global value called cry (abbreviation of carry) to represent the value passed to carry() and use a loop to simulate the recursion process of carrying over.
- It's hard to verify that the read value is correct by directly comparing, so we used an array called $Saved$ to keep the data after carrying over. And we created separate arrays $Savedn$ to keep the value for comparison. Also, a flag called done is used to show that the reading process is done, so that the result can be used for verifying LTL statements.
    - Notice that, the $atomic$ expression used are ***only*** for verification, it's not a part of the algorithm.

Reproducing & testing:

Java

In the code implementation, for readability, R is named as READERS, B is named as SIZE and k is named as ROUNDS.

For JAVA code, firstly compile the code by running

$$javac\ *.java,$$

and then run the program by:

$$java\ Counter\ R\ B\ k.$$

The result is printed in the terminal with the read/written value.

Promela

For Promela, the value of R, B and k should be manually set in the code before running. Also, if a R other than 3 is desired to be tested, please add or delete $readern$ processes and $ReadValue$ arrays.

Simulation

The counting process can be simulated by clicking (Re)Run button in iSpin. While k is the value entered in the field “maximum number of steps” under Simulate/Replay tab. The value of counter and reader’s result can be inspected from the variables $CounterValue$ and $ReadValue$ in “variable values” section in the bottom left.

Verification

For verification, select “use claim” option and enter the name for the LTL statement that is going to be examined. Notice that, enforce weak fairness constraint should be toggled on.

The first statement $eventual\_reader$ is to ensure the eventually entry property and the second statement $correct\_value$ is used to check that the read value is correct according to the definition in spec.

Also, please remind that, if other R and B values are used to test, please modify LTL statements, and related #define correspondingly by following the instruction on comment.