REPORT

Explanation on how it works

The starve-free solution works on this method : Any number of readers can simultaneously read the data. A writer will make its presence known once it has started waiting by setting `writer\_waiting` to true.

Once a writer has come, no new process that comes after it can start reading. This is ensured by the writer acquiring the `in\_sem` semaphore and not leaving it until the end of its process. This ensures that any new process that comes after this (be it reader or writer) will be queued up on `in\_sem`. Now, from the looks of it, it might seem like this is a method that is biased towards writer and may lead to starvation of readers. However, this is where the FIFO nature of semaphore comes into picture.

Suppose processes come as `RRRWRWRRR`. Now, by our method the first three readers will first read. Then, when the next process which is a writer takes the `in\_sem` semaphore, it won't give it up until it's done. Now, suppose by the time this writer is done writing, the next writer has already arrived. However, it will be queued up on the `in\_sem` semaphore. Thus, it won't get to start writing before the process before it, the reader process has completed. Thus, the reader process will get done and then the writer process will again block any more processes from starting and so on.

To explain the writer process further, in case there's no other reader, it doesn't wait at all. Otherwise, it sets `writer\_waiting` to true and waits on the `writer\_sem` semaphore. Also, to make the `writer\_sem` semaphore synchronizable across both the process where one process only executes `wait()` and other only executes `signal()`, we initialize it to `0`.

To explain the reader process further, it firsts increments the `num\_started` then reads the data and then increments the `num\_completed`. After this, it checks both these variables are equal. If they are and a writer is waiting (found from `writer\_waiting`), it signals the `writer\_sem` semaphore and finishes.

Thus, it will be pretty clear how we manage to make a starve-free solution for the readers-writers problem with a FIFO semaphore. We can say that all processes will be handled in a FIFO manner. Readers will allow other readers in the queue to start reading with them but writers will block all other processes waiting in the queue from executing before it finishes. In this way, we can implement a reader-writer solution where no process will have to indefinitely wait leading to starvation.

* orderMutex cannot be part of a [deadlock](http://en.wikipedia.org/wiki/Deadlock) since at the time it is requested no other semaphore is ever held by the calling process, so we can forget about it in our deadlock analysis (this property holds if every process only calls reader() or writer() once, but it works in the general case).
* readers is always strictly greater than 0 if any reader is currently executing lines 15 to 20 (inclusive): the variable is unconditionally incremented at line 14 and unconditionally decremented at line 21 without any other modification, and no modification ever occurs without an exclusive lock granted by readersMutex so the readers integrity is guaranteed.
* accessMutex is taken at line 13 just before readers goes from 0 to 1. It is released at line 23 just after readers goes back to 0. Detection of those raising or falling edges is guaranteed by the use of the readersMutex lock. As seen above, it means that no reader will be executing lines 14 to 22 (inclusive) without accessMutex being taken by a reader (possibly gone now), especially line 18 which represents resource access.
* accessMutex can be requested when readersMutex is held (on line 13), and readersMutex can be requested when accessMutex is held (on line 20). However, the first situation (line 13) only happens when readers is equal to 0 (first reader to get access to the resource), which can never be the case if another reader is currently trying to execute line 20 as seen above, so those two potentially deadlocking reservations can never occur at the same time. Thus those two semaphores cannot interact and cause a deadlock.
* The resource is never accessed by a writer without accessMutex being taken in an unconditional and exclusive way. Since we have seen above that accessMutex is always collectively taken by readers before they access the resource, the resource is properly protected.
* Fair access is guaranteed through the orderMutex which is taken upon arrival and released only when access to the resource has been granted.

References

- [arXiv:1309.4507](https://arxiv.org/abs/1309.4507)

- Abraham Silberschatz, Peter B. Galvin, Greg Gagne - Operating System Concepts