Solution pseudocode

- Semaphore operations on variable S are now defined as,

```
wait(S)
{
S.value--;
        if (S.value < 0)
          {
           add this process to S.List;
           block();  //suspends, waiting state
           }
}
signal(S)
{
S.value++;
          if (S.value <= 0)
            {
             remove a process P from S.List;
             wakeup(P); //resumes, ready state
             }
 }
```

The structure of a writer's process :

```
while (true)
    {
   wait(rw_mutex); // any writers or readers?
// writing is performed
   signal(rw_mutex); // enable others
    }
```

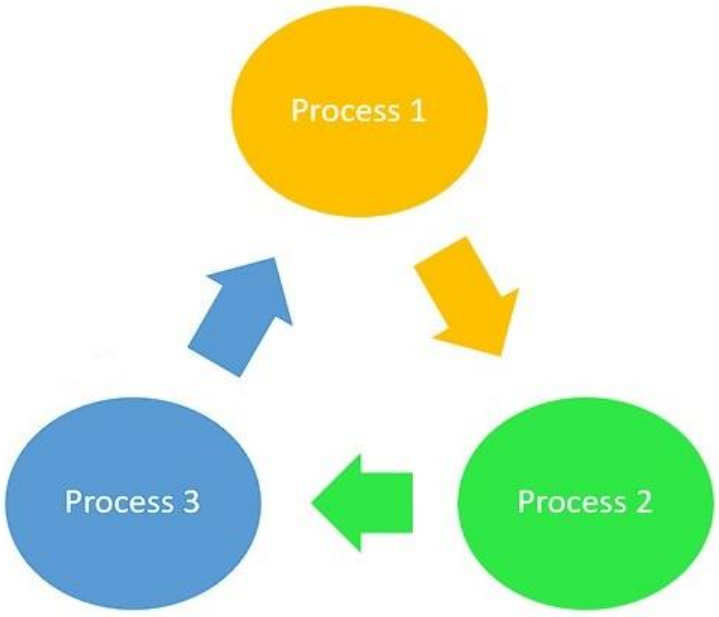
The structure of a reader process :

```
do {
wait(mutex);
    read_count++;
            if (read_count == 1)  // Only the first reader blocks writers
                wait(rw_mutex);  // block writers
signal(mutex);
                        /* reading is performed */
        wait(mutex); // ensure mutual exclusion read
count--;  // reader done
  if (read_count == 0)
    signal(rw_mutex);  //Only the last reader unblocks the writers If no more readers in
         signal(mutex); // release lock for other readers
 } while (true);
```

Examples of Deadlock (Circular wait) ?

One process want to read file is waiting for the resource, which writer is held by the second process, which is also waiting for the resource held by the third process etc. This will continue until the last process is waiting for a resource held by the first process. This creates a circular chain.

For example, Process READ is allocated Resource WRITE as it is requesting Resource READ. In the same way, Process WRITE is allocated Resource READ, and it is requesting Resource WRITE. This creates a circular wait loop



Circular wait example

How did solve deadlock ?

Theoretically, the solution to the readers-writers problem is as follows:

A writer should have exclusive access to the object in question when writing, meaning that no other readers nor writers should access the object during writing.

A reader has non-exclusive access to the object — meaning that multiple readers can operate at the same time.

The result

No reader is ever kept waiting — unless a writer is in control of the object.

deadlock-free (that is when each member of a group is waiting for another member of the group to take an action, resulting in a locked resource).

Examples of starvation :

the starvation of the Writer: a Writer thread does not have a chance to execute while any number of Readers continuously entering and leaving the working area.

How did solve starvation :

To avoid this problem the following commonly known solution is proposed.

Initialisation	Reader	Writer
in = Semaphore(1) mx = Semaphore(1) wrt = Semaphore(1) ctr = Integer(0)	- Wait in - Wait mx - if (++ctr)==1, then Wait wrt - Signal mx - Signal in [Critical section] - Wait mx - if (--ctr)==0, then Signal wrt - Signal mx	- Wait in - Wait wrt [Critical section] - Signal wrt - Signal in

This solution is simple and fast enough. However the penalty in comparison to the previous one is that the Reader must lock two mutexes to enter the working area. If the working area is fast and assuming that mutex locking is a heavy system call, there would be a benefit of having an algorithm which allows locking one mutex on entering the working area and one on exiting .

Explanation for real world application :

Rader-Writer

- Banking system : read account balances versus ubdate

- Reading and writing from the same file

how did apply the problem :

By using Semaphore , by placing a lock on the file when entering and leaving the lock when exiting

Reader-Writer

What if we have a constraint on a resource — something like a file. So we want to access this file as we run processes .

Now imagine we represent our tasks through threads that access these resources. multi thread reading while another writing is waiting .

To put the problem succinctly — what should happen when one thread wants to write to a resource while another thread wishes to read from the same resource ?

This is known as the readers–writers problem.