Readers/Writers Problem

Problem

You want to have a "critical section" where multiple writers are allowed, as well as multiple readers, but at no time should readers and writers be allowed in at the same time. You want a deadlock free, starvation free solution.

Solution

Below is code for the reader processes. The writer process is analogous. Just replace "r" for "w" and vice-versa. As a reminder, `r_active` is the number of active readers (readers in critical section, `r_waiting` is the number of readers waiting to go into the critical section. `r_sem` is the semaphore that waiting readers sleep on. The writers have analogous variables. `lock` is a mutex semaphore used by both readers and writers.

```
 1   P( lock ); // lock is a mutex and is initializes to 1.
 2    if ( nw_active + nw_waiting == 0 )
 3     {
 4        nr_active++; // Notify we are active
 5        V( r_sem );  // Allow ourself to get through
 6     }
 7   else
 8     nr_waiting++;  // We are waiting
 9   V( lock );
10   P( r_sem );  // Readers will wait here, if they must wait.
11
12   READING...
13
14   P( lock );
15   nr_active--;
16   if (( nr_active == 0 ) && ( nw_waiting > 0 )) // If we are the last reader
17     {
18        while ( nw_waiting > 0 )  // Allow all waiting writers to enter
19        {
20           V( w_sem );  // wake a writer;
21           w_active++;  // one more active writer
22           w_waiting--; // one less waiting writer.
23        }
24      }
25   V( lock );
```

Here is an intuition of how the algorithm behaves. You have a room, maybe it has windows so you can look inside. Just outside this room, there are two benches. One for readers, one for writers. Suppose you are a writer, and are coming in. You look inside the room. If there are readers in the room, you sit on the bench. If there are writers in the room, but there are readers sitting on the bench to go in, you will sit on a bench. Basically, this will prevent starvation. You will be polite and allow the reader to have a chance to go in. If you a writer waiting to enter, and there are writers in the room as well as readers on the bench, then the last writer to exit will let all waiting readers into the room. Then, the last reader to exit will let all the writers in. Some invariants that hold:

- Safety conditions: Readers and writers can not be active at the same time.
    - `nr_active > 0 -> nw_active == 0`
    - `nw_active > 0 -> nr_active == 0`
- Other conditions.
    - `nr_active > 0 & nr_waiting > 0 -> nw_waiting > 0`

    - `nw_active > 0 & nw_waiting > 0 -> nr_waiting > 0`
- Readers can't be both active and waiting unless a writer is waiting.

So, basically, if a writer is waiting, then either readers are in the room, and the last reader will let the writer in, or writers are in the room, in which case we waited because readers were waiting, and thus the last writer will let in the readers, and the last reader will let us in.

The last writer will check if there are any readers waiting, and if so, let all readers in. So, as the writers are in the room, if there are more readers and writers coming in, then all the readers that are on the bench as the last writer leaves are the ones sent in the room.

There is the same for the flip case (i.e., substitute readers for writers and vice versa in the above, and what I said still holds).

Note that this really makes no assumptions about processes being blocked in FIFO order. In the last while loop, we basically wake up all waiting writers. There might be some simplification (can't think of one though) were process woken in the order they went to sleep, but I believe this algorithm makes no such assumption, and in general, it should make no such assumptions.

More discussion

In line 20, we wake up any sleeping writers. Will we violate safety conditions (i.e., will readers and writers be in the critical section). If we are at line 20, then line we executed line 16, which means that there are no active readers. The process that is executing code in line 20 must have been the last reader, and since it is already in this part of the code, it has already completed executing the critical section (reading). But, perhaps another reader will somehow sneak into the critical section. Let's argue this can't happen.

Because the current process is at line 20, then another reader must be excluded from lines 1-9. That's because the current process hold "lock", which is a mutex variable, and thus no other process can be in between `P(lock)` and `V(lock)`. So, a reader that is attempting to sneak in is either just before line 1, or just after line 19.

If a reader (call it "R") is just before line 1, then the current process will eventually set `w_active` to be greater than 1, and this happens before the reader can get to line 2 (because of the mutex lock). When "R" enters this section, it will test line 2, then see that there is a waiting process, and eventually go to sleep at line 10. So, "R" won't get to read while the writers are writing.

Otherwise, "R" is just after line 9. If it expects to get into the critical section, then it must have executed line 5, which means it must have executed line 4. It must have done so before the current process got past line 14 (again, because of the mutex). Once the current process reaches line 16, `nr_active` will not be zero, and hence, it will not have woken up the sleeping writers. So, this is a rough argument for why a reader can't sneak into the critical section while we wake the writers.