# Classic Problems of Synchronization

So far, we have discussed three approaches to solve synchronization problems. To verify and test a proposed synchronization scheme, several classical synchronization problems are used. We will describe a few of these problems (producer-consumer, reader-writer, and dining philosophers) and how semaphores are used to resolve issues of mutual exclusion and synchronization in concurrent processing environments.

## Readers-Writers Problem

A classic example of database applications, where a database is shared among several concurrent processes. Some processes may want only to read the database, whereas others may want to update the database. We can describe these processes as readers and writers. Readers can access shared data simultaneously, with no adverse effects on data integrity. Whereas writers cannot access shared data simultaneously and if allowed will have adverse effects on data. To ensure data integrity, the writers have exclusive access to the shared database while writing to the database. The readers-writers problem is considered a typical class of synchronization problem, where multiple read operations, and exclusive write operations are possible.

## Readers-Writers Problem with Reader's Priority

In simple readers-writers problem, readers have priority; no reader be kept waiting unless a writer has already obtained permission to use shared data. Since multiple read operations are allowed for different readers, it means, no reader should wait for other readers to finish simply because a writer is waiting. We describe the solution of readers-writers problem with readers priority by defining following shared data structure:

```
semaphore rwsyn = 1;
semaphore mutex = 1;
int read-count = 0;
```

The semaphores rwsyn and mutex are initialized to 1; read-count is initialized to 0.
The mutex semaphore is used to exclusively update variable read-count, which keeps track of how many processes are currently reading the data. The semaphore rwsyn is common to both reader and writer processes. The semaphore rwsyn is to ensure exclusive writing function by the writers. It is also used by the first or last reader that enters or exits the critical section (reading data).
The code for reader and writer processes is listed below:

```
Semaphores: mutex= rwsyn = 1; int: read-count=0;
Reader()                          Writer()
wait(mutex);                       wait(rwsyn);
read-count++;                        <Write Unit>
if (read-count == 1)               signal(rwsyn);
        wait(rwsyn);
signal(mutex);
  <Read Unit>
wait(mutex);
read-count--;
if (read-count == 0)
        signal(rwsyn);
signal(mutex);
```

If we trace above code of reader and writer process, we can notice, while one reader in read section, multiple readers can join in the read section. If a writer is waiting (i.e. blocked on rwsyn semaphore, when the last reader is leaving the read unit, it will signal the waiting writer (decrements value of read-count and wakeup writer process) to procced for writing. While multiple readers are reading, a writer may starve to perform write operation. Similarly, if a writer is in the write unit and n readers are waiting, then one reader is queued on rwsyn, and n-1 readers are queued on mutex. We can also notice, when a writer executes signal(rwsyn), either waiting readers resume the execution or a single waiting writer. The selection is made by the scheduler by having a priority queue (reader's priority over writer) for processes blocked on rwsyn semaphore. This can also be achieved by having another semaphore to enforce priority of readers, *will leave for you as an exercise to modify the code and trace to verify your code.*

In readers-writers problem with readers priority it may starve writers by postponing them indefinitely, while readers are active. In situations where writer process is frequently updating data, a variant of readers-writers problem with writer's priority is considered.

**Readers-Writers Problem with Writer's Priority**

It has same basic requirement of multiple read operations and exclusive write operations. To ensure writers priority code of reader and writer processes is modified to satisfy following requirements:

- Once a writer is ready, that writer performs its write operation as soon as possible.
- If a writer is waiting to access, no new readers may start reading.

We describe the solution of readers-writers problem with writer's priority by defining following shared data structure:

```
Semaphores:
        x = 1;          /* for mutual exclusion of shared data of Readers */
        y = 1;          /* for mutual exclusion of shared data of Writers */
        wsem = 1;       /* to synchronize writer by readers */
        rsem = 1;       /* to synchronize first reader by writers; */
        z = 1;          /* to queue all readers while writer is writing */

Integers:
        read_count = 0;         /* to keep count of readers */
        write_count = 0;        /* to keep count of writers */
```

The code for reader and writer processes is listed below:

```
Reader( )
wait(z)
wait(rsem)
wait(x)
read_count++;
if (readcount == 1)
        wait(wsem)
signal(x)
signal(rsem)
signal(z)
<Read Unit>
wait(x)
read_count--;
if (readcount == 0)
       signal(wsem)
signal(x)
```

```
Writer( )
wait(y)
write_count++;
if (write_count == 1 )
          wait(rsem)
signal(y)
wait(wsem)
    <Write Unit>
signal(wsem)
wait(y)
write_count--;
 if (write_count == 0 )
       signal(rsem)
signal(y)
```

We describe summarily, reader and writer processes states, you can trace the code and verify it by noting values of different semaphores:

Readers only: wsem set

Writers only: wsem, rsem set, and writers queue on wsem

Reader(s) in C.S. /* Read Unit */
wsem set by reader, rsem set by writer,
writers queue on wsem, one reader on rsem, other readers queue on z

Writer in C.S /* Write Unit */
wsem, rsem set by writer, writers queue on wsem, one reader on rsem, other readers queue on z

When writers have priority, although the shared data will be updated but readers will not have timely access to it and readers miss some data, whereas when readers have priority, it may starve writers by postponing them indefinitely; while readers are active. To overcome this a strategy is proposed where both readers and writers compete in finite time:

- A new reader should not start if there is a writer waiting (prevent starvation of writers).
- All readers waiting at the end of a write operation should have priority over the next writer (prevent starvation of readers)