University of Central Punjab

FOIT (Operating Systems)

GL- 11

Reader-Writer Problem

Objectives

Students will be able to solve a classical synchronization problem: reader-writer

Reader-Writer

The readers-writers problem is a classical one in computer science: we have a resource (*e.g.*, a database) that can be accessed by readers, who do not modify the resource, and writers, who can modify the resource. When a writer is modifying the resource, no-one else (reader or writer) can access it at the same time: another writer could corrupt the resource, and another reader could read a partially modified (thus possibly inconsistent) value.

This problem has been extensively studied since the '60s, and is often categorized into three variants:

1. Give readers priority: when there is at least one reader currently accessing the resource, allow new readers to access it as well. This can cause starvation if there are writers waiting to modify the resource and new readers arrive all the time, as the writers will never be granted access as long as there is at least one active reader.
2. Give writers priority: here, readers may starve.
3. Give neither priority: all readers and writers will be granted access to the resource in their order of arrival. If a writer arrives while readers are accessing the resource, it will wait until those readers free the resource, and then modify it. New readers arriving in the meantime will have to wait.

Unfortunately, only the solutions for the first two problems are easily found on the Internet. As of today, even the wikipedia page on the subject does not have an algorithm for what is called "the third readers-writers problem". However, this algorithm is very simple once you decompose the problem into smaller steps and properties.

The third readers-writers problem

We will build the solution using C code. We will use semaphores for mutual exclusion (mutex). Those semaphores, being used as locks, are all initialized in the released state (1 available place); those initializations will not appear in code snippets below but will be shown in comments.

For that, we want to have fair-queuing between readers and writers in order to prevent starvation. To achieve that, we will use a semaphore named **orderMutex** that will materialize the order

of arrival. This semaphore will be taken by any entity that requests access to the resource, and released as soon as this entity gains access to the resource:

```
semaphore orderMutex;      // Initialized to 1

void reader()
{
  P(orderMutex);           // Remember our order of arrival
  ...
  V(orderMutex);           // Released when the reader can access the
resource
  ...
}

void writer()
{
  P(orderMutex);           // Remember our order of arrival
  ...
  V(orderMutex);           // Released when the writer can access the
resource
}
```

Now, we can write the writer code as it is the most straightforward of both. The writer wants exclusive access to the resource. We will create a new semaphore named accessMutex that the writer will request before modifying the resource:

```
semaphore accessMutex;     // Initialized to 1
semaphore orderMutex;      // Initialized to 1

void reader()
{
  P(orderMutex);           // Remember our order of arrival
  ...
  V(orderMutex);           // Released when the reader can access the
resource
  ...
}

void writer()
{
  P(orderMutex);           // Remember our order of arrival
  P(accessMutex);          // Request exclusive access to the resource
  V(orderMutex);           // Release order of arrival semaphore (we have
been served)

  WriteResource();         // Here the writer can modify the resource at
will
```

```
  V(accessMutex);          // Release exclusive access to the resource
}
```

The reader code is a bit more complicated as multiple readers can simultaneously access the resource. We want the first reader to get access to the resource to lock it so that no writer can access it at the same time. Similarly, when a reader is done with the resource, it needs to release the lock on the resource if there are no more readers currently accessing it.

This is similar to a light switch in a dark room: the first person entering the room will turn the lights on, while the last person leaving the room will turn the lights off. However, it is much more simple if the room has only one door and only one person can enter or leave the room at the same time, to prevent someone from switching the lights off when someone enters at the same time using another door. This is why we will use a counter named **readers** representing the number of readers currently accessing the resource, as well as a semaphore named **readersMutex** to protect the counter against conflicting accesses.

```
semaphore accessMutex;      // Initialized to 1
semaphore readersMutex;     // Initialized to 1
semaphore orderMutex;       // Initialized to 1

unsigned int readers = 0;   // Number of readers accessing the resource

void reader()
{
  P(orderMutex);            // Remember our order of arrival

  P(readersMutex);          // We will manipulate the readers counter
  if (readers == 0)         // If there are currently no readers (we came
first)...
    P(accessMutex);         // ...requests exclusive access to the
resource for readers
  readers++;                // Note that there is now one more reader
  V(orderMutex);            // Release order of arrival semaphore (we have
been served)
  V(readersMutex);          // We are done accessing the number of readers
for now

  ReadResource();           // Here the reader can read the resource at
will

  P(readersMutex);          // We will manipulate the readers counter
  readers--;                // We are leaving, there is one less reader
```

```
  if (readers == 0)          // If there are no more readers currently
reading...
    V(accessMutex);          // ...release exclusive access to the resource
  V(readersMutex);           // We are done accessing the number of readers
for now
}

void writer()
{
  P(orderMutex);             // Remember our order of arrival
  P(accessMutex);            // Request exclusive access to the resource
  V(orderMutex);             // Release order of arrival semaphore (we have
been served)

  WriteResource();           // Here the writer can modify the resource at
will

  V(accessMutex);            // Release exclusive access to the resource
}
```

University of Central Punjab

FOIT (Operating Systems)
GL- 11
Reader-Writer Problem

Practice Tasks

Task1:

Implement & validate the algorithm of the 1st reader-writer problem.

Task2:

Modify the Task1 to maintain order in which the readers or writers arrive and make it starvation-free.

Task3:

Modify the Task1 such that Readers and Writers go their critical section in an alternative order. Not more than 5 readers can enter their critical sections.

If there are 8 readers and 3 writer threads, five readers go their critical section and when they all exit one writer enters its critical section. Upon exit of the writer, the next 3 reader threads then enter their critical sections.