Week 8 – LAQ's

Instructions

Devise an algorithm for reader-writers problem of synchronization. Explain the solution in detail

The Reader-Writers problem is a classic synchronization problem that involves managing access to a shared resource (like a database) by multiple readers and writers. The challenge is to ensure that:

1. Multiple readers can read the shared resource simultaneously.
2. Only one writer can write to the shared resource at a time.
3. When a writer is writing, no readers should be allowed to read.
4. The system should avoid starvation for both readers and writers.

Types of Reader-Writers Problems

There are two variations of the Reader-Writers problem:

1. First-Come, First-Served (FCFS): This version allows readers to starve writers if there are continuous readers.
2. Writer Priority: This version allows writers to have priority over readers, preventing reader starvation.

Algorithm for the Reader-Writers Problem

Here, we will present a solution for the First-Come, First-Served (FCFS) version of the Reader-Writers problem.

Data Structures

We will use the following data structures:

- Shared Resource: The resource that readers and writers access.

- Semaphore mutex: A binary semaphore to protect access to the shared variable readcount.
- Semaphore wrt: A binary semaphore to ensure mutual exclusion for writers.
- Integer readcount: A counter to keep track of the number of active readers.

Initialization

```
mutex = 1; // Binary semaphore for mutual exclusion
wrt = 1;   // Binary semaphore for writers
readcount = 0; // Number of active readers
```

Reader Process

```
Reader() {
   while (true) {
      wait(mutex); // Enter critical section to update readcount
      readcount++; // Increment the number of readers
      if (readcount == 1) {
         wait(wrt); // If this is the first reader, wait for writers
      }
      signal(mutex); // Exit critical section

      // Read the shared resource
      ReadResource();
```

```
        wait(mutex); // Enter critical section to update readcount
        readcount--; // Decrement the number of readers
        if (readcount == 0) {
            signal(wrt); // If this is the last reader, signal writers
        }
        signal(mutex); // Exit critical section
    }
}
```

Writer Process

```
Writer() {
    while (true) {
        wait(wrt); // Wait for access to the shared resource

        // Write to the shared resource
        WriteResource();

        signal(wrt); // Release access to the shared resource
    }
}
```

Explanation of the Algorithm

1. Reader Process:

- Each reader first enters a critical section protected by the mutex semaphore to update the readcount.
- If the readcount becomes 1 (indicating that this is the first reader), it waits on the wrt semaphore to block writers from accessing the resource.
- After updating the readcount, the reader exits the critical section.
- The reader then reads the shared resource.
- After reading, the reader re-enters the critical section to decrement the readcount.
- If the readcount becomes 0 (indicating that this is the last reader), it signals the wrt semaphore, allowing writers to access the resource.
- Finally, the reader exits the critical section.

2. Writer Process:

- Each writer waits on the wrt semaphore to gain exclusive access to the shared resource.
- Once it has access, the writer writes to the shared resource.
- After writing, the writer signals the wrt semaphore to allow other writers or readers to access the resource.

Advantages of This Solution

- Fairness: The algorithm ensures that readers and writers are treated fairly, preventing starvation of either group.
- Concurrency: Multiple readers can access the resource simultaneously, improving performance when reads are more frequent than writes.

Potential Issues

- Starvation: In the FCFS version, if there are continuous readers, writers may starve. This can be mitigated by implementing a writer priority mechanism.
- Complexity: The implementation of semaphores and critical sections can introduce complexity in the code.