Producer-Consumer problem

In the Producer-Consumer problem, there are two main entities:

- The Producer, which generates data and places it into a shared buffer.

- The Consumer, which removes and processes data from the shared buffer.

The key challenge in this problem arises from the fact that both the Producer and the Consumer access a shared buffer concurrently. Without proper coordination, this can lead to concurrency and synchronization issues.

**Concurrency Issues:**

1. **Race Conditions:**
   * if two producers or two consumers access the buffer at the same time, they might overwrite each other's data or consume the same data twice.
2. **Data Inconsistency:**
   * Without proper control, both producers and consumers could access or modify shared data structures (like buffer pointers or counters) simultaneously, leading to inconsistent or corrupt data in the buffer.

**Synchronization Issues:**

1. **Buffer Overflow (Producer Overrun):**
   * If producers continue to add items when the buffer is full, without being aware of the buffer’s capacity, the system can experience **buffer overflow**, leading to loss of data or errors.
2. **Buffer Underflow (Consumer Starvation):**
   * If consumers attempt to take items from an empty buffer, they cannot proceed until the producer adds more items. Without synchronization, a consumer could try to consume from an empty buffer, causing **underflow** or unnecessary waiting.
3. **Deadlock:**
   * If both the producer and consumer are waiting for each other to release resources (e.g., the producer is waiting for the consumer to free buffer space, while the consumer is waiting for the producer to provide data), a **deadlock** situation can occur where neither can proceed.
4. **Livelock:**
   * This occurs when both producer and consumer are repeatedly trying to act but cannot make progress, e.g., they keep releasing and acquiring locks but never actually perform their respective operations.
5. **Starvation:**
   * If a scheduling mechanism or a lock prioritizes one party (either producers or consumers) more than the other, one might starve, i.e., producers might not get a chance to add items, or consumers might not get to consume items, leading to indefinite waiting for one party.

**Note: Mutex(lock variables), semaphores( wait and signal calls) , monitor (procedures and conditional variables in protected form)**

**A simple mutex-based solution for the bounded buffer Producer-Consumer problem** ensures mutual exclusion when producers and consumers access the shared buffer. Additionally, condition variables (or semaphores) are used to signal whether the buffer has space or items.

Key Components:

1. Buffer: Shared between producers and consumers, with a fixed size.

2. Mutex: Ensures mutual exclusion so only one thread (producer or consumer) can access the buffer at any time.

3. Condition Variables: Two condition variables are used:

- `not\_full`: Used by producers to wait if the buffer is full.

- `not\_empty`: Used by consumers to wait if the buffer is empty.

Pseudocode:

buffer[MAX\_SIZE]; // The buffer with a fixed size MAX\_SIZE

int count = 0; // Number of items currently in the buffer

mutex lock; // Mutex to ensure mutual exclusion

condition\_variable not\_full; // Condition variable to wait when buffer is full

condition\_variable not\_empty; // Condition variable to wait when buffer is empty

//Producer

function producer() {

while (true) {

item = produce\_item(); // Produce an item

// Lock the mutex before modifying the buffer

lock(mutex);

// Wait while the buffer is full

while (count == MAX\_SIZE) {

wait(not\_full, mutex); // Wait on 'not\_full' until space is available

}//while

// Add the item to the buffer

buffer[in] = item;

in = (in + 1) % MAX\_SIZE;

count++;

// Signal that the buffer is not empty (for consumers)

signal(not\_empty);

// Unlock the mutex

unlock(mutex);

}

}//producer

// Consumer

function consumer() {

while (true) {

// Lock the mutex before modifying the buffer

lock(mutex);

// Wait while the buffer is empty

while (count == 0) {

wait(not\_empty, mutex); // Wait on 'not\_empty' until an item is available

}

// Remove the item from the buffer

item = buffer[out];

out = (out + 1) % MAX\_SIZE;

count--;

// Signal that the buffer is not full (for producers)

signal(not\_full);

// Unlock the mutex

unlock(mutex);

// Consume the item

consume\_item(item);

}

}//consumer

Analysis:

- Blocking: Producers block when the buffer is full, and consumers block when the buffer is empty.

- Synchronization: Mutexes prevent simultaneous access, and condition variables ensure that threads only proceed when appropriate (buffer full or empty conditions change).

- Circular Buffer: The use of `in` and `out` indices with modulo (`% MAX\_SIZE`) ensures that the buffer is treated as circular.

This solution effectively addresses concurrency and synchronization issues in the Producer-Consumer problem by ensuring mutual exclusion and proper signaling when the buffer is full or empty.

A **semaphore-based solution for the bounded buffer Producer-Consumer problem** is another common approach to ensure proper synchronization between producers and consumers when accessing the shared buffer.

Key Components:

1. Buffer: Shared between producers and consumers, with a fixed size.

2. Semaphores:

- `empty`: Counts the number of empty slots in the buffer (initialized to the size of the buffer), general semaphore

- `full`: Counts the number of items in the buffer (initialized to 0)., General semaphore

- `mutex`: Ensures mutual exclusion so that only one thread (either a producer or a consumer) accesses the buffer at a time, binary semaphore

Pseudocode:

Shared Variables:

buffer[MAX\_SIZE]; // The buffer with a fixed size MAX\_SIZE

int in = 0, out = 0; // Pointers for inserting/removing items

semaphore mutex = 1; // Semaphore for mutual exclusion

semaphore empty = MAX\_SIZE; // Semaphore for available buffer slots (starts with MAX\_SIZE)

semaphore full = 0; // Semaphore for filled buffer slots (starts with 0)

//Producer

function producer() {

while (true) {

item = produce\_item(); // Produce an item

// Wait for an empty slot in the buffer (decrement 'empty')

wait(empty);

// Lock the buffer (ensure mutual exclusion)

wait(mutex);

// Add the item to the buffer

buffer[in] = item;

in = (in + 1) % MAX\_SIZE; // Circular buffer

// Unlock the buffer (release mutual exclusion)

signal(mutex);

// Signal that a new item is available (increment 'full')

signal(full);

}

}

//Consumer

function consumer() {

while (true) {

// Wait for an item to be available in the buffer (decrement 'full')

wait(full);

// Lock the buffer (ensure mutual exclusion)

wait(mutex);

// Remove the item from the buffer

item = buffer[out];

out = (out + 1) % MAX\_SIZE; // Circular buffer

// Unlock the buffer (release mutual exclusion)

signal(mutex);

// Signal that an empty slot is available (increment 'empty')

signal(empty);

// Consume the item

consume\_item(item);

}

}

**Use of Semaphores:**

1. Semaphores:

- `empty`: Tracks how many empty slots are in the buffer. When the buffer is full, the producer will wait (block) on `empty` until the consumer consumes an item and frees up a slot.

- `full`: Tracks how many items are available in the buffer. If the buffer is empty, the consumer will wait (block) on `full` until the producer adds an item.

- `mutex`: Ensures mutual exclusion by allowing only one thread to access the shared buffer at a time.

2. Producer:

- The producer first waits on the `empty` semaphore. This ensures that it only proceeds if there is space in the buffer.

- It then waits on the `mutex` semaphore to gain exclusive access to the buffer and adds the produced item.

- After adding the item, it releases the `mutex` to allow other threads access and signals the `full` semaphore, indicating that there is a new item available for consumption.

3. Consumer:

- The consumer first waits on the `full` semaphore to ensure there is an item to consume.

- It then waits on the `mutex` semaphore to get exclusive access to the buffer and removes the item.

- After removing the item, it releases the `mutex` and signals the `empty` semaphore, indicating that there is now space in the buffer for the producer to add more items.

**Key Properties/features of the Semaphore Solution:**

1. Mutual Exclusion:

- Semaphore `mutex` ensures that only one thread (either a producer or consumer) accesses the buffer at any given time, preventing race conditions.

- This ensures mutual exclusion over the critical section, which is the buffer.

2. Progress:

- The producer will block if the buffer is full (wait on `empty`), but it makes progress as soon as space becomes available (when the consumer signals `empty`).

- The consumer will block if the buffer is empty (wait on `full`), but it makes progress as soon as an item becomes available (when the producer signals `full`).

- The system avoids deadlock because both producers and consumers signal each other when buffer conditions change (empty/full), ensuring continuous progress.

3. Bounded Buffer:

- The `empty` semaphore starts at `MAX\_SIZE`, representing the total number of available slots in the buffer. As the producer adds items, `empty` decreases, and as the consumer removes items, `empty` increases.

- Similarly, the `full` semaphore starts at 0 and increases as items are added and decreases as items are removed. This keeps track of how many items are available in the buffer.

4. Fairness:

- This solution guarantees fairness: if a producer or consumer is waiting, it will eventually be able to proceed as long as the buffer conditions (full or empty) change. Starvation does not occur because the semaphores guarantee that each process gets its turn once the buffer has available space or items.

**A monitor-based solution for the bounded buffer Producer-Consumer problem** uses a high-level synchronization construct that combines mutual exclusion and condition synchronization in a structured way. Monitors encapsulate shared variables and methods that operate on those variables, ensuring that only one thread can be executing a monitor procedure at a time.

In this solution, the monitor manages the buffer and provides methods for producers and consumers to access the buffer in a safe manner.

Components of a Monitor-Based Solution:

1. Monitor: A high-level construct that encapsulates:

- A shared buffer and associated variables.

- Condition variables: Used to control when producers can add items to the buffer and when consumers can remove items.

- Mutex: Automatically enforced by the monitor to ensure mutual exclusion when accessing shared variables.

2. Condition Variables:

- not\_full: Used by producers to wait when the buffer is full.

- not\_empty: Used by consumers to wait when the buffer is empty.

// Monitor Structure

monitor BoundedBuffer {

condition not\_full; // Condition variable for buffer not being full

condition not\_empty; // Condition variable for buffer not being empty

item buffer[MAX\_SIZE]; // The shared buffer

int in = 0, out = 0, count = 0; // Pointers and count for the circular buffer

// Add an item to the buffer (Producer)

procedure add\_item(item x) {

if (count == MAX\_SIZE) {

wait(not\_full); // Wait until there is space in the buffer

}

buffer[in] = x; // Add the item to the buffer

in = (in + 1) % MAX\_SIZE; // Circular buffer

count++; // Update the count

signal(not\_empty); // Signal that buffer is no longer empty

} // procedure

// Remove an item from the buffer (Consumer)

procedure remove\_item() -> item {

if (count == 0) {

wait(not\_empty); // Wait until there is an item in the buffer

}

item x = buffer[out]; // Remove the item from the buffer

out = (out + 1) % MAX\_SIZE; // Circular buffer

count--; // Update the count

signal(not\_full); // Signal that buffer is no longer full

return x;

}

}

// Producer

function producer() {

while (true) {

item = produce\_item(); // Produce an item

// Add the item to the shared buffer

BoundedBuffer.add\_item(item);

}

}

// Consumer \

function consumer() {

while (true) {

// Remove the item from the shared buffer

item = BoundedBuffer.remove\_item();

consume\_item(item); // Consume the item

}

}

```

Explanation of the Monitor-Based Solution:

1. Monitor:

- The monitor encapsulates the shared buffer (`buffer[]`), the `in` and `out` pointers for the circular buffer, and the `count` variable to track the number of items.

- Mutual exclusion is inherently managed by the monitor. Only one thread (either a producer or a consumer) can execute a monitor procedure at a time, ensuring safe concurrent access.

2. Condition Variables:

- not\_full: The producer will wait on this condition if the buffer is full. When the consumer removes an item and signals the producer, the producer can add a new item to the buffer.

- not\_empty: The consumer will wait on this condition if the buffer is empty. When the producer adds an item and signals the consumer, the consumer can remove the item from the buffer.

3. Producer:

- When the producer wants to add an item, it first checks if the buffer is full (i.e., `count == MAX\_SIZE`). If the buffer is full, the producer waits on the `not\_full` condition.

- After the consumer consumes an item and signals the producer (via `signal(not\_full)`), the producer resumes, adds the item to the buffer, and then signals the consumer (via `signal(not\_empty)`) that the buffer is no longer empty.

4. Consumer:

- When the consumer wants to remove an item, it first checks if the buffer is empty (i.e., `count == 0`). If the buffer is empty, the consumer waits on the `not\_empty` condition.

- After the producer produces an item and signals the consumer (via `signal(not\_empty)`), the consumer resumes, removes the item from the buffer, and then signals the producer (via `signal(not\_full)`) that the buffer is no longer full.

Analysis of the Solution:

1. Mutual Exclusion:

- The monitor guarantees that only one thread can be inside the `add\_item` or `remove\_item` procedures at a time.

- This ensures mutual exclusion, as it prevents race conditions when accessing the shared buffer. The `mutex` is implicitly managed by the monitor, so no additional locking mechanism is required.

2. Condition Synchronization:

- Blocking: The producer is blocked if the buffer is full, and the consumer is blocked if the buffer is empty. Condition variables (`not\_full` and `not\_empty`) ensure that producers and consumers only proceed when it is safe to do so.

- Signaling: Producers signal consumers when an item is added (buffer not empty), and consumers signal producers when an item is consumed (buffer not full). This ensures that producers and consumers are awakened when their conditions change.

3. Deadlock-Free:

- The solution is deadlock-free because producers and consumers only wait when necessary (on full or empty conditions). Once the buffer state changes (due to item addition or removal), the corresponding waiting process is signaled to continue.

4. Fairness and Starvation:

- Fairness: No producer or consumer will starve, as each process waiting on the condition variable (`not\_full` or `not\_empty`) will be signaled as soon as the required condition is satisfied.

- No Starvation: Every producer will eventually add its item to the buffer when space is available, and every consumer will eventually consume an item when one is available.

Key Advantages of the Monitor-Based Solution:

- Encapsulation: The monitor encapsulates the shared data and synchronization logic, making it easier to reason about the correctness and safety of the solution.

- Simplicity: The code is simpler and more structured compared to solutions using semaphores or explicit locks. The monitor automatically handles mutual exclusion.

- Condition Synchronization: Condition variables within the monitor ensure that producers and consumers are blocked only when necessary, providing efficient synchronization.

**Compare and contrast producer consumer solutions using - 1. Mutex, 2. Semaphore 3. Monitor**

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **Mutex-Based Solution** | **Semaphore-Based Solution** | **Monitor-Based Solution** |
| **Mutual Exclusion** | Yes (with manual control) | Yes (with counting) | Yes (automatic) |
| **Condition Management** | Manual (using condition variables) | Direct (using counts) | Built-in (using condition variables) |
| **Complexity** | Moderate (requires extra management) | Higher (requires understanding of counts) | Lower (encapsulated management) |
| **Deadlock Risk** | Moderate (manual handling) | Possible (if not managed properly) | Lower (built-in safeguards) |
| **Flexibility** | Limited (single resource focus) | High (can manage multiple resources) | Moderate (best for single resource) |
| **Language Support** | Widely supported | Widely supported | Language-specific support |

### Summary:

* **Mutex** solutions are simple and effective for basic mutual exclusion but can become complicated when managing conditions.
* **Semaphore** solutions offer more flexibility and are suitable for managing multiple resources but may introduce complexity and potential misuse.
* **Monitor** solutions provide a higher level of abstraction, encapsulating state and synchronization together, which simplifies programming but requires language support.

The Readers-Writers problem is a classic synchronization problem that arises when multiple threads (readers and writers) try to access a shared resource (like a file or a database). The problem is to ensure safe concurrent access while maintaining efficiency.

Problem Setup:

- Readers: Multiple readers can read the shared resource simultaneously, as reading does not modify the resource.

- Writers: A writer needs exclusive access to the resource because writing can modify it. If a writer is writing, no other writer or reader should access the resource.

Concurrency and Synchronization Issues:

1. Race Condition:

- If a reader or writer does not acquire locks correctly before accessing the shared resource, a race condition can occur, leading to inconsistent or corrupted data.

- For example, if a reader is reading data while a writer is modifying it, the reader may get inconsistent data.

2. Mutual Exclusion:

- Writers need mutual exclusion to prevent simultaneous write operations, which could lead to data corruption.

- Readers don’t need mutual exclusion from each other, but they need to ensure that no writer is modifying the resource when they are reading.

3. Deadlock:

- Deadlock can occur if readers and writers are blocked waiting for each other in a circular dependency.

- For example, a writer is waiting for readers to finish, while readers are waiting for the writer to release the resource.

4. Starvation:

- Starvation occurs when one group (readers or writers) is continually given priority, preventing the other group from accessing the resource.

- Writer Starvation: If the system gives continuous priority to readers, writers may starve (never get a chance to write). This is known as Reader Priority.

- Reader Starvation: If writers are given priority over readers, readers may starve if new writers keep coming. This is called Writer Priority.

5. Fairness:

- A fair solution must ensure that both readers and writers get a reasonable opportunity to access the resource.

- Fairness is about avoiding starvation and ensuring that neither readers nor writers are indefinitely delayed.

Concurrency and synchronization issues in readers-writers problem

Solutions to Readers-Writers Problem:

There are three main variations of solutions based on different priorities:

1. First Readers-Writers Problem (Reader Priority):

- Readers are given priority over writers.

- Multiple readers can read simultaneously, but a writer must wait until all readers are done before writing.

- Issue: Writers may starve if there is a continuous stream of readers.

2. Second Readers-Writers Problem (Writer Priority):

- Writers are given priority over readers.

- Once a writer is waiting, no new readers are allowed to start reading until the writer has finished.

- Issue: Readers may starve if there is a continuous stream of writers.

3. Third Readers-Writers Problem (Fair Solution):

- Ensures fairness by allowing both readers and writers to access the resource without starving either group.

- A common approach is to alternate between readers and writers in a fair manner. This prevents one group from starving the other.

Example of Concurrency and Synchronization Issues:

1. Race Condition:

- without proper synchronized access to the shared resource, data can be corrupted. For example, a reader might read partially updated data if a writer is allowed to write at the same time.

2. Deadlock:

- Deadlock in the Readers-Writers problem occurs when a writer waits for all readers to release the resource, but new readers keep arriving, preventing the writer from ever acquiring the lock. This leads to a cycle where the writer is permanently blocked, and readers continue accessing the resource.

3. Starvation:

- Reader Starvation Example: If new writers are given priority over readers, the readers may never get a chance to access the resource.

- Writer Starvation Example: If there are too many readers constantly reading, writers may have to wait indefinitely, which can lead to writer starvation.

Mutex solution for fair readers-writers problem

//Initialize:

mutex = CreateMutex() // Mutex for synchronizing read\_count

wrt = CreateSemaphore(1) // Semaphore for writer access

read\_count = 0 // Number of active readers

//Reader

function Reader(){

while true:

Lock(mutex) // Lock to update read\_count

read\_count = read\_count + 1 // Increment the number of readers

if read\_count == 1: // If this is the first reader

Wait(wrt) // Wait for writer access (lock for writing)

Unlock(mutex) // Unlock after updating read\_count

// Critical Section (Reading)

Read from shared resource // Perform read operation

Lock(mutex) // Lock to update read\_count

read\_count = read\_count - 1 // Decrement the number of readers

if read\_count == 0: // If this is the last reader

Signal(wrt) // Signal that writers can access

Unlock(mutex) // Unlock after updating read\_count

}//reader

//Writer

function Writer(){

while true:

Wait(wrt) // Wait for exclusive access to write

// Critical Section (Writing)

Write to shared resource // Perform write operation

Signal(wrt) // Release exclusive access

}//writer

### Key Points:

1. **Mutex (mutex)**: protects the read\_count variable to ensure correct updates to the reader count
2. **Semaphore (wrt)**: Controls access for writers, ensuring mutual exclusion when a writer is writing.

An analysis of how the solution handles different combinations of readers and writers:

**1. Reader-Reader (Multiple Readers):**

* Multiple readers are allowed to read the shared resource concurrently.
* **Fairness**: Readers can read concurrently because they only lock the mutex when incrementing or decrementing read\_count. The first reader locks the writers out by calling Wait(wrt) on the semaphore, ensuring that no writers can enter while readers are accessing the shared resource. Subsequent readers do not need to block other readers.
* **Result**: Fair and efficient. Multiple readers can read at the same time, and no reader is blocked if there is already a reader in the critical section.

**2. Reader-Writer (Simultaneous Readers and Writers):**

* **Fairness**: When a writer is waiting, readers that arrive after the writer must wait until the writer has completed its task. The first reader arriving will lock the writer out by calling Wait(wrt). However, once the last reader finishes and read\_count becomes 0, the writer will proceed by receiving the Signal(wrt).
* **Result**: Fair to both readers and writers. Writers can access the resource once all active readers finish, ensuring no starvation for writers, and readers can read concurrently until the first writer arrives.

**3. Writer-Writer (Multiple Writers):**

* Writers have exclusive access to the shared resource.
* **Fairness**: The semaphore wrt ensures mutual exclusion for writers. If multiple writers are waiting, the semaphore will allow only one writer to enter the critical section at a time. Other writers will be blocked until the currently active writer calls Signal(wrt).
* **Result**: Fair and prevents writer-writer starvation. Writers are serialized, ensuring only one writer is in the critical section at a time.

**4. Writer-Reader (Simultaneous Writers and Readers):**

* **Fairness**: If a writer is accessing the resource, readers must wait. Similarly, if readers are already in the critical section, a writer must wait until all readers have finished reading. Writers get access to the shared resource when there are no active readers (i.e., read\_count == 0).
* **Result**: Fair to both readers and writers. The writer has to wait for readers to finish, but once the writer gains access, it blocks all future readers until it finishes writing.

**Analysis of Fairness:**

* **No starvation**: The solution avoids both reader and writer starvation. Writers wait for ongoing readers to finish but will not be indefinitely delayed because once the last reader finishes, the writer gets exclusive access.
* **Mutual Exclusion**: Writers always get exclusive access to the critical section.
* **Reader Priority**: Readers are given priority in the sense that multiple readers can access the resource simultaneously, while writers have to wait if there are any active readers. However, once a writer starts, no new readers can access the resource until the writer finishes.

**Possible Edge Cases:**

* **Writer starvation in case of frequent readers**: While writers aren't starved in this implementation, there is a slight delay for writers if readers continuously arrive. However, this delay is minimal due to the condition that only when read\_count == 0 can a writer access the resource.

**Solution for Fair Readers-Writers Problem Using Semaphores**

Initialize:

mutex = CreateSemaphore(1) // Mutex to protect read\_count

wrt = CreateSemaphore(1) // Semaphore for writer access

read\_count = 0 // Number of active readers

read\_lock = CreateSemaphore(1) // Semaphore for reader access

//Reader

function Reader(){

while true:

Wait(read\_lock) // Wait to enter reader section

Wait(mutex) // Lock to update read\_count

read\_count = read\_count + 1 // Increment reader count

if read\_count == 1: // If this is the first reader

Wait(wrt) // Wait for writer access

Signal(mutex) // Unlock after updating read\_count

Signal(read\_lock) // Allow other readers to enter

// Critical Section (Reading)

Read from shared resource // Perform read operation

Wait(mutex) // Lock to update read\_count

read\_count = read\_count - 1 // Decrement reader count

if read\_count == 0: // If this is the last reader

Signal(wrt) // Signal that writers can access

Signal(mutex) // Unlock after updating read\_count

}//reader

//Writer

function Writer(){

while true:

Wait(wrt) // Wait for exclusive access to write

// Critical Section (Writing)

Write to shared resource // Perform write operation

Signal(wrt) // Release exclusive access

}//writer

**Semaphores**:

* + mutex: Protects access to the read\_count variable.
  + wrt: Controls access for writers, ensuring that only one writer can write at a time.
  + read\_lock: Ensures that readers can enter the section safely without interfering with each other.

### Fairness & Performance Summary:

* **Reader-Reader Fairness**: Multiple readers can read the shared resource concurrently. The use of read\_lock prevents contention among readers when they try to increment or decrement read\_count. The design ensures fair access for all readers without starving any particular reader.
* **Reader-Writer Fairness**: Writers are not starved, as they will eventually access the shared resource once all current readers finish. The use of wrt ensures that new readers cannot start reading while a writer is waiting, which promotes fairness between readers and writers.
* **Writer-Writer Fairness**: Writers access the shared resource one at a time in a fair manner due to the wrt semaphore. No writer is starved, and they get serialized access.
* **Writer-Reader Fairness**: Writers have to wait for readers to finish, and once they begin writing, no readers can enter until the writer finishes. This promotes a fair balance between readers and writers.

### Starvation and Livelock Considerations:

* **No Starvation**: Both readers and writers get a fair share of access to the shared resource. Readers can access concurrently, and writers get their exclusive access once readers are done. The use of semaphores ensures that no group (readers or writers) will be indefinitely delayed.
* **No Livelock**: The solution prevents livelock since the read\_lock semaphore ensures that readers can't endlessly preempt writers. Writers are guaranteed eventual access after all readers finish.

**Solution for Fair Readers-Writers Problem Using Monitors**

Monitor ReadersWriters:

Condition canRead = CreateCondition()

Condition canWrite = CreateCondition()

int read\_count = 0 // Number of active readers

bool writing = false // Indicates if a writer is active

Procedure StartRead(){

while writing: // If a writer is active, wait

wait(canRead)

read\_count = read\_count + 1 // Increment reader count

}//startread

Procedure EndRead(){

read\_count = read\_count - 1 // Decrement reader count

if read\_count == 0: // If last reader finished

signal(canWrite) // Signal any waiting writers

}// EndRead

Procedure StartWrite(){

while read\_count > 0 or writing: // Wait if there are readers or a writer is active

wait(canWrite)

writing = true // Set writing to true (exclusive access)

} //StartWrite

Procedure EndWrite(){

writing = false // Writing is finished

signal(canRead) // Signal waiting readers

signal(canWrite) // Signal any waiting writers

}// EndWrite

//Reader

function Reader(){

while true:

StartRead() // Request to read

// Critical Section (Reading)

Read from shared resource // Perform read operation

EndRead() // Finish reading

}//writer

//Writer

function Writer(){

while true:

StartWrite() // Request to write

// Critical Section (Writing)

Write to shared resource // Perform write operation

EndWrite() // Finish writing

}//reader

### Summary:

1. **Monitor**: Encapsulates the shared resource, protecting it with conditions and state variables.
   * canRead: Condition variable to manage readers waiting for access.
   * canWrite: Condition variable to manage writers waiting for access.
   * read\_count: Tracks the number of active readers.
   * writing: Boolean flag indicating if a writer is currently writing.
2. **Reader Logic**:
   * Before starting to read, a reader checks if a writer is active. If so, it waits on canRead.
   * After incrementing read\_count, the reader performs its critical section (reading).
   * Once done, it decrements read\_count and signals any waiting writers if it’s the last reader.
3. **Writer Logic**:
   * Before writing, a writer checks if there are active readers or if another writer is active. If so, it waits on canWrite.
   * The writer then performs its critical section (writing).
   * Once finished, it signals both waiting readers and writers, allowing for fair access.