# Programming Assignment 4

- Anirudh Saikrishnan(CS22BTECH11004)

## Goal:

To solve the Readers-Writers problem (writer preference) and Fair Readers-Writers problem using Semaphores as discussed in the class in C++. You must implement these two
algorithms and compare each thread's average and worst-case time to access the critical section
(shared resources).

## Implementation:

The code is divided into multiple functions, including simple functions to calculate random CS Time, random Remaining time, Average wait time and Worst Case Wait time along with the main Reader and Writer functions. The implementation for the first 4 is mostly self explanatory.

### 1) Writer Preference RW Problem

#### a) Reader:

The `reader` function encapsulates the behavior of a reader thread within the readers-writers problem. Upon invocation, the thread enters a sequence of actions designed to ensure safe concurrent access to a shared resource. Firstly, it acquires the `readTry` semaphore to serialize access to the `readcount` variable, indicating the number of active readers. By incrementing `readcount`, the reader signals its intent to access the shared resource, potentially preventing subsequent writer threads from entering the critical section simultaneously. If it happens to be the first reader, it additionally obtains the `resource` semaphore, thereby securing exclusive access to the critical section to avoid write-write conflicts.

Within the critical section, the thread executes its read operation, emulated by the function `randCSTime(muCS)`, which introduces a random delay simulating the time required to perform the read task. Upon completing the reading task, the reader exits the critical section by releasing the `resource` semaphore and decrementing `readcount`. If no other readers are active, the semaphore is freed up for potential writer threads waiting to access the critical section.

Subsequently, the reader thread simulates execution outside the critical section by waiting for a random duration, as determined by the function `randRemTime(muREM)`. This step ensures that the reader's execution pattern aligns with realistic scenarios where threads alternate between critical and non-critical activities.

Finally, the function computes the duration of the reader's wait time and logs pertinent information, such as the entry and exit times of the critical section.

## b) Writer:

The `writer` function embodies the behavior of a writer thread within the readers-writers problem, executing a series of steps to maintain coordinated access to a shared resource while ensuring data consistency. Here's a detailed breakdown of its functionality:

Upon invocation, the writer thread begins by acquiring the `wmutex` semaphore, which serves to serialize access to the `writecount` variable. By incrementing `writecount`, the writer indicates its intent to access the critical section for writing operations. If it happens to be the first writer, it also acquires the `readTry` semaphore, thereby blocking subsequent reader threads from entering the critical section concurrently, ensuring exclusive access for writing.

Once the writer thread successfully acquires the necessary semaphores, it proceeds to acquire the `resource` semaphore, thereby securing exclusive access to the critical section, preventing read-read and write-read conflicts. Within this critical section, the writer executes its writing operation, emulated by the function `randCSTime(muCS)`, which introduces a random delay representing the time taken to perform the write task effectively.

After completing the writing operation, the writer releases the `resource` semaphore, signifying the end of its critical section access. Subsequently, it decrements `writecount` to reflect the completion of its writing task. If no other writers are active, it releases the `wmutex` semaphore, potentially allowing blocked reader threads to proceed.

Following the critical section, the writer thread simulates execution outside the critical section by waiting for a random duration, as determined by the function `randRemTime(muREM)`. This ensures that the writer's execution pattern aligns with realistic scenarios where threads alternate between critical and non-critical activities.

Finally, the function calculates the duration of the writer's wait time, from the request to enter the critical section until actual entry, providing valuable insights into synchronization delays and system efficiency. It logs pertinent information, including entry and exit times of the critical section, contributing to performance analysis and debugging efforts.

# 2) Fair RW Problem

## a) Reader:

The provided code presents a solution to the readers-writers problem, focusing on fairness through the use of semaphores in C++. In the `reader` function, upon invocation, the thread logs its request time, including its ID and request count, before accessing the critical section. This logging ensures a comprehensive understanding of thread behavior. To maintain fairness, a semaphore named `serviceQueue` is utilized, ensuring that requests are processed in a first-come, first-served manner. Within the critical section, the reader thread acquires the `rmutex` semaphore to prevent concurrent access to the `readcount` variable, incrementing it to signal its presence. If it happens to be the first reader, it further acquires the `resource` semaphore, thereby blocking writer threads from accessing the critical section simultaneously. After successful entry, the reader releases the `rmutex` semaphore, allowing other reader threads to update `readcount`. The execution within the critical section is emulated by introducing a random delay using the `randCSTime(muCS)` function.

Similarly, the `writer` function follows a similar pattern. Upon invocation, it logs its request time and ensures fairness through the `serviceQueue` semaphore. Subsequently, it acquires the `resource` semaphore to gain exclusive access to the critical section, preventing concurrent access from both readers and writers. Within the critical section, the writer executes its writing operation, simulated by a random delay introduced by `randCSTime(muCS)`. After completing its task, the writer releases the `resource` semaphore, allowing other threads to access the critical section. Both reader and writer functions then simulate execution outside the critical section by waiting for a random duration specified by `muREM`.

In the `main` function, semaphores are initialized, and input parameters are read from a file. The code spawns reader and writer threads, ensuring concurrent execution. After all threads complete execution, semaphores are destroyed. The function then calculates and writes average and worst-case wait times for both readers and writers, aiding in performance analysis and debugging efforts. This approach guarantees fairness by processing requests in a sequential manner, ensuring equitable access to the critical section for all threads, thereby mitigating issues such as starvation and thread priority inversion.

## b) Writer:

The `writer` function in the provided code governs the behavior of writer threads within the readers-writers problem, aiming to coordinate access to a shared resource, typically a critical section, while ensuring fairness and data integrity.

Upon invocation, the `writer` function initiates by logging the request time of the writer thread, including its identification and request count. This logging mechanism provides a chronological record of thread requests and their respective entries into the critical section, aiding in understanding the sequence of operations.

To maintain fairness in access, the function utilizes the `serviceQueue` semaphore, which regulates the order in which threads are granted access to the critical section. This ensures that threads are processed in a sequential manner, mitigating potential issues such as starvation and ensuring equitable access to the shared resource.

Within the critical section, the writer thread proceeds by acquiring both the `serviceQueue` and `resource` semaphores. The `serviceQueue` semaphore ensures fairness, while the `resource` semaphore provides exclusive access to the critical section, preventing concurrent access by other writer and reader threads.

Once inside the critical section, the writer executes its writing operation, simulated by a random delay specified by the `muCS` parameter, utilizing the `randCSTime(muCS)` function. This delay emulates the time required for the writer to perform its writing task within the critical section, ensuring that the shared resource is modified appropriately and consistently.

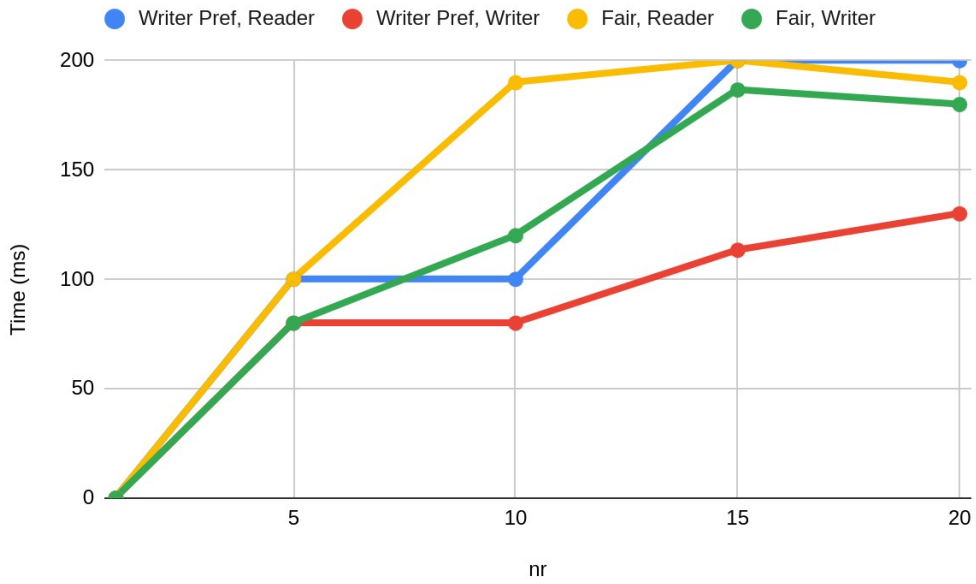
Upon completing its writing operation, the writer releases the `resource` semaphore, indicating the end of its critical section access. Subsequently, it releases the `serviceQueue` semaphore to allow subsequent threads to access the critical section in a fair and orderly manner.

Following its critical section execution, the writer simulates execution outside the critical section by waiting for a random duration specified by the `muREM` parameter, employing the `randRemTime(muREM)` function. This ensures that the writer's execution pattern aligns with realistic scenarios, where threads alternate between critical and non-critical activities.

Throughout its execution, the `writer` function diligently calculates the duration of the writer's wait time and logs pertinent information, such as entry and exit times from the critical section.

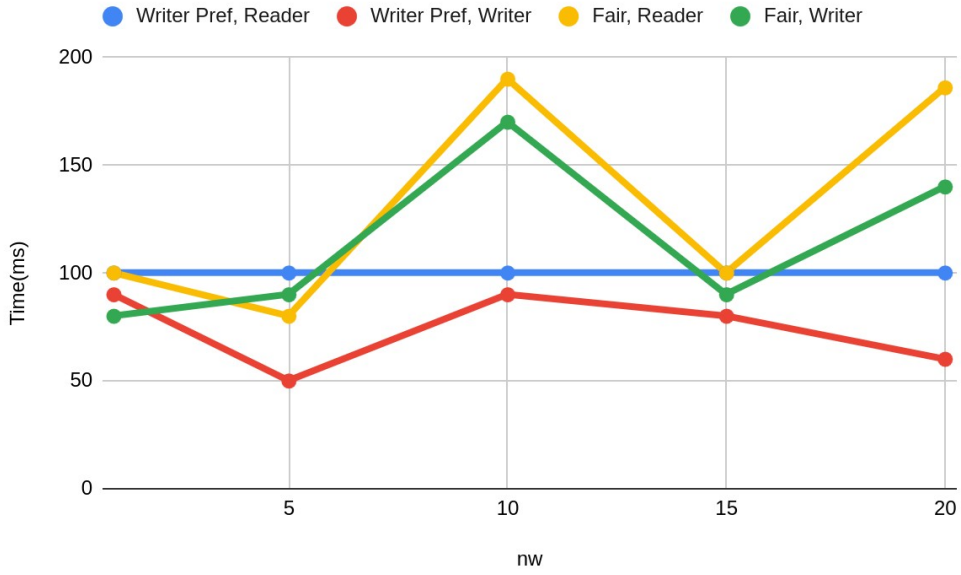
# Graphs & Analysis

## 1) Average Time, nw Fixed



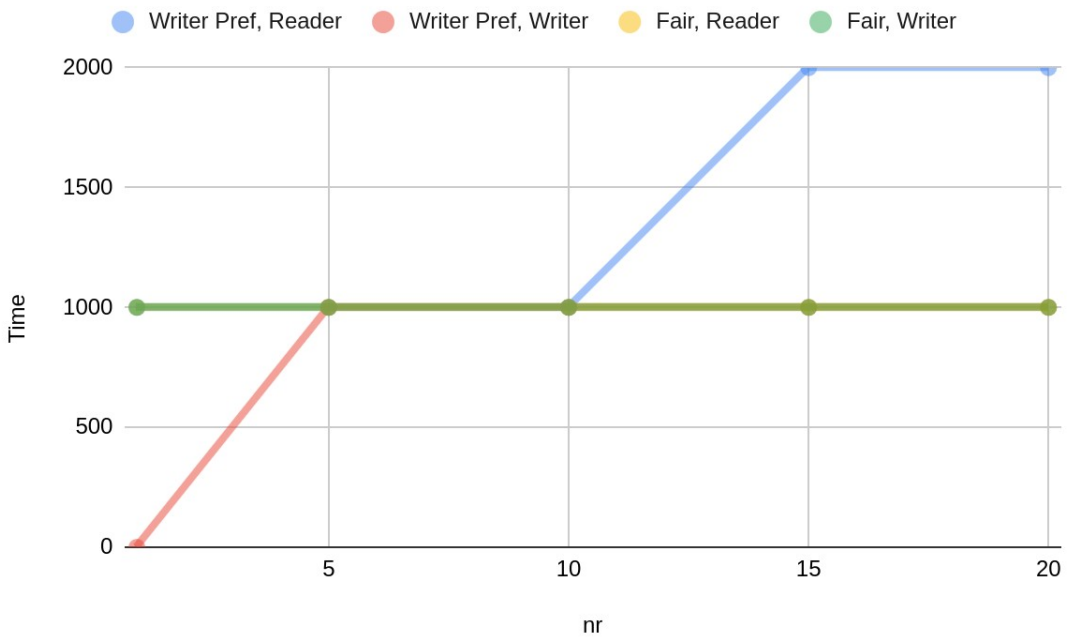

As expected, average wait time is significantly lower for writers in the writer preference algorithm while its more or less around the same level for the other 3

## 2) Average Time, nr Fixed



Again, as expected, writer time in writer preference is consistently lower. I am not completely sure why there is such high variance for the fair rw case and why the reader average is consistenly 100 ms though.

# 3) Worst case, nw fixed




Interestingly, for the fair case, the reader and writer graphs completely overlap! The writer threads for the writer preference case also overlap with these lines starting from nr=5.
It is difficult to say exactly why the trends are what they are because there is an element of randomness associated with the worst case analysis due to the involvement of the two exponential random variables in our code.

# 4) Worst Case, nr fixed

A graph feels unhelpful here since for all cases, irrespective of nw, worst case wait time was always 1000ms. I'm at a loss as to why this was the case but this result was consistent over a large number of tests. Once again, the only element I can think of attributing this to is the element of randomness.