Introduction

This project involved writing two C++ programs, Reader and Writer so that we can perform Inter-process communication (IPC) between two processes. The primary goal is to minimize the latency between the time the writer sends the message and the time the reader reads the message.

There are multiple IPC's that we can use to send messages between processes. Some of the IPC's we can use are:

1. Message Queue
2. Shared memory
3. TCP/IP
4. Shared files
5. Pipes

Since we want to minimize the latency, we need the transfer to happen over memory. So this weeds out TCP/IP and Shared Files. I first explored the simpler ones where the sharing of data is done by the operating system, just so that I get the feeling of how things are working. The Linux's Message Queue, is a "POSIX message queues allow processes to exchange data in the form of messages."

This was easy to set up and get working after seeing a couple of well-written examples online and with the Linux man page. I only ran into one problem which was the way I am counting my sent and received messages. I first tested with single-threaded (1-1 reader and writer) sending a million messages, the average latency between each message was around 700-1000 nano-seconds. Then I created an array of threads that do the same read and write but over multiple threads. This was also easy to implement because the OS handles synchronization. This had an average latency of 3500 - 4500 nano-seconds per message.

The problem I noticed was that after I read the message, I am not doing any processing of the data. So the more threads I added, the higher latency because of cache misses and race conditions. This is one of the reasons why multithread took longer because there is no work to be done, and it doesn't take advantage of multiple threads because they are all waiting. I knew at this point that if I added more threads, I would get higher latency on any form of ICP.

Since I wanted more direct access to the handling of sending and receiving messages, I looked into Shared Memory. This is a memory segment allocated by the OS so that multiple processes can read and write to the same segment without calling operating system functions. So, this gives us a better latency because of the less overhead and faster access to read and write, because it acts just like a pointer to a variable.

Shared Memory

Here I describe my implementation of the shared memory. Since our shared memory is just some memory segment, we have to order or add additional data to the memory so that we have read and write functionality.

1. The First Iteration

When I first started, I just wanted to understand how shared memory works. I knew from the beginning that I needed some sort of locking mechanism because I didn't want the writer to override the previous message without the reader reading the message. I also didn't want the reader to read the data multiple times.

After researching multiple locking systems, the one that I found most interesting was Semaphore. The idea behind this was that I would have a flag variable inside my shared memory, this variable would be an integer. So the process would look like this:

1. Writer: If the flag is 1, write the message and set the flag equal to 2.
2. Reader: If the flag is 2, read the message and set the flag to 1.

So this acted as a signal to both reader and writer on when to read or write. If the writer finished it would set the flag to 3, and the reader would stop reading. This worked great and I got an average latency of 150-200 nanoseconds between messages. The fastest I have gotten so far.

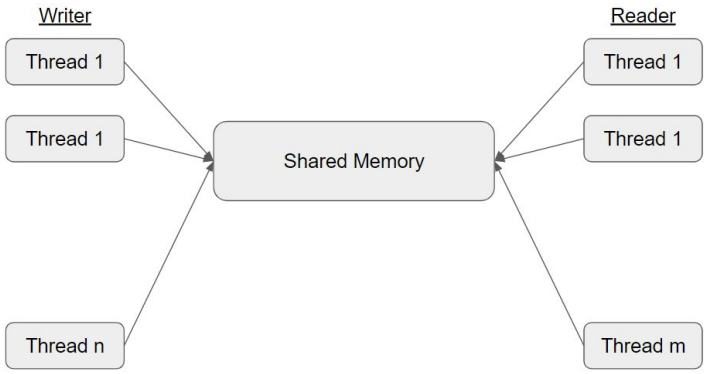
1. The Second Iteration

From the message queue, I knew that I needed to implement some kind of queue so that multiple messages could be put in and taken out to work on, instead of having a 1-1 relationship. The overview of this new design is here:

1. SpinLock: This was another locking system I found. Instead of raising a flag like before, we loop till we get a hold of the lock. Once we have the lock, we do work on the memory segment and release the lock. So that others can hold the lock.
2. Ring Buffer: This acted like a queue system, where we had a circular buffer. And reading and writing were done on its respective index which was stored in the memory. So, if our write index or read reaches the end, then we wrap back to the front.

Single Shared Memory

From this point on I pretty much had the type of Shared Memory I wanted. I had a locking system for multiple reading and writing and a queue to hold an array of messages. Now it was just a matter of writing and reading from the Shared Memory. So in my Single Shared Memory system, you can have n-writers or m-readers, whose values can be passed through the executable arguments. And the diagram looks like this:



Multiple factors influence the latency:

1. Number of threads for writing and reading
2. How my spinlock works (handling race condition)
3. The size of the shared memory (size of my queue or Ring Buffer)

The most important influences were the second and third factors. For the spinlock, it's important how we handle race conditions, and cache misses when trying to get hold of the lock. So, we can fix it by putting the thread to sleep or do busy work before all the threads race to get the lock. As for the size of the fixed queue, again should be smaller to avoid writers always getting the lock. But if we had to process our read, then a bigger queue would be best so that the writer can write more messages.

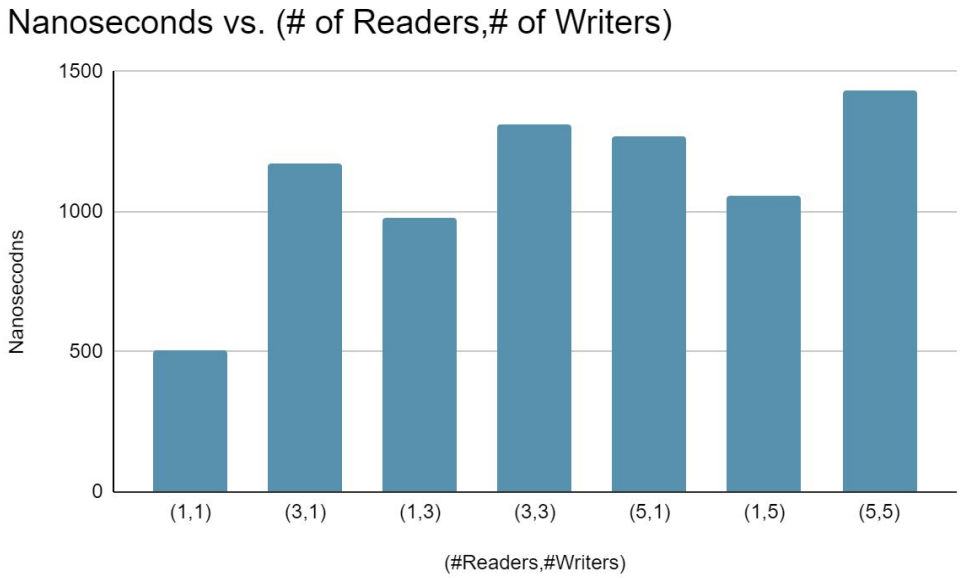


Fig 1. Relationship between the number of threads and latency for Single Shared Memory.

Multi-Shared Memory

The idea behind Multi-Shared Memory is to provide multiple shared memory so that both reader and writer can do their operations on multiple memory blocks simultaneously. By default there are 4 shared memories, this can be altered to handle more incoming data.

Previously, we had both read and write threads operate on the same buffer, but now we have each read or write thread randomly access one of many shared memory and operator accordingly on that shared memory. So multi or single have the same factors that influence its latency. The more shared memory you have, the higher latency because we might not access the written data.

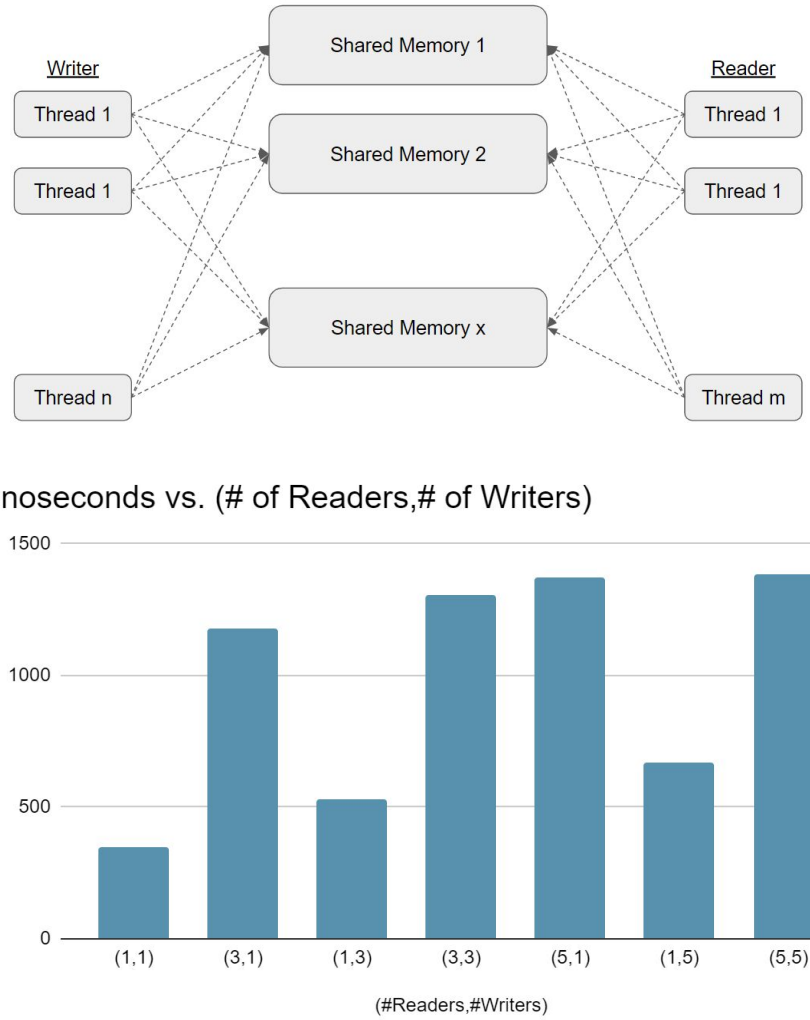


Fig 2. Relationship between the number of threads and latency for Multi-Shared Memory

One thing to note is that both in single and multi shared memory, the more writers with fewer readers results in lower latency. I think this is partly because there is no race condition or cache misses on the reader side, and on the writer, we have a queue we can write to. So we have to fine-tune our parameters in such a way that we get the lowest latency. Like the number of readers/writers, queue size, Spinlock (race condition), and the number of shared memory.