CSC716 Fall 2015 Final Project

Multithreaded reader-writer problem

Lawrence Ferretti, Moon Lee, Steven Kuhfahl

**Introduction**

As one of classic problems of synchronization, readers-writers problem has been used to test nearly every new synchronization primitive. Readers-writers problem is that at a time only one writer can access the file or several readers can access the file concurrently. It has to ensure there are no writer and reader access the file at the same time. In this project, we will use three techniques-Monitor, Semaphore, Test and Set to simulate readers-writers problem. The goal of this project is to ensure mutual exclusive among readers and writer and between writers as well as deadlock.

Semaphore is done by Moon Lee, Monitor is done by Lawrence Ferretti, and Test and Set is done by Steven Kuhfahl.

**1. Semaphore**

Semaphore basically is a counter. In my program I used binary Semaphore, which is 0 and 1. Semaphore use wait() and signal() functions to block and unblock threads. It guarantees mutual exclusive. When it compares to mutex, since it doesn't have busy waiting, semaphore is more efficient than mutex. But when comapre to monitor, since shared data is a global variable, global variable can be modified in anywhere in program.

In this program, I used C++11 standard, and used mutex and condition variable to implement semaphore since C++ standard library does not define a semaphore type.

1.1 std::condition\_variable member functions

I used wait() and notify\_all() member functions in condition\_variable class defined in standard library. I used notify\_all() rather than notify\_one(), because program should provide possibility to run several readers run concurrently after a writer finishes accessing file.

The wait() function is implements as follws:

unique\_lock lck(mutex);

condition\_variable.wait(lck);

Wait() function always comes after obtaining an unique lock. At the moment of blocking the thread, it automatically calls lck.unlock() to unlock mutex, allowing other threads to continue. Once a thread being notified, the thread is unblocked and calls lck.lock() to lock mutex. The thread should then check the condition and resume waiting if the wake up was spurious.

1.2 Initialization

In the program, it will initialize three global variables as follows:

mutex mtx;

condition\_variable readwrite\_cv;

atomic<int> rw\_flag(0);

mutex and condition variable are used to implement semaphore. rw\_flag is used for check whether readers or a writer are accessing the file. I use atomic for rw\_flag here is because in reader() function there is rw\_flag++; operation which is not a single instruction. Thus it's possible that several threads modifying this variable at the same time. atomic type ensures the variable can be modified atomically.

1.3 Reader()

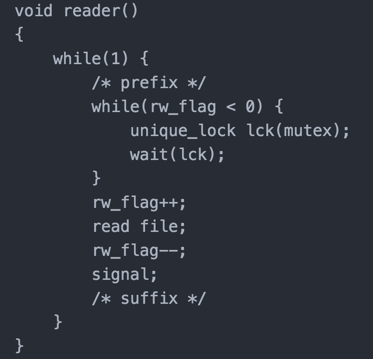


Figure 1. Reader() function

Figure 1 is pseudo code for reader() function. First it will check rw\_flag, if it is smaller than 0, that means a writer is accessing file. Thus the thread is being blocked. Once the thread is being notified by other thread (means other thread finishes accessing file), it will recheck rw\_flag value, it will execute further only if rw\_flag is not smaller than 0. Reader will increase rw\_flag by one in order to stop other writers accessing the file. While one reader accessing the file, other readers can also access the file because rw\_flag is set to bigger than 0 or equal to 0. After finishing reading file, it will decrease rw\_flag by one to let other writers can access the file. Then it will signal other waited threads to resume.

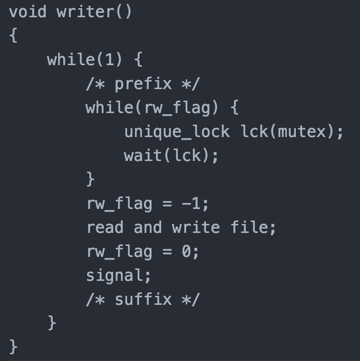


Figure 2. Writer() function

Figure 2 shows pseudo code for writer() function. First it will check rw\_flag, it it is bigger than 0 or smaller than 0, it will wait. When the thread is being notified, it will recheck rw\_flag and the thread will pass to next part of function only if rw\_flag is equal to 0. It will set rw\_flag as -1 before accessing file in order to stop other writers and readers accessing the file. After finishing accessing the file, it will set rw\_flag to 0 and signal waited threads to resume.

1.4 Problems

1.4.1 I rarely found some threads start to run while initializing reader and writer threads.

It looks like:

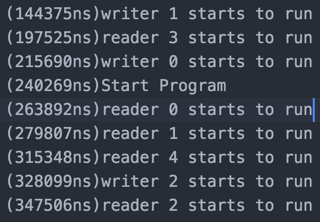


Figure 3. Some threads start to run while other threads are being initialized

We can see that writer 0, 1 and reader 3 start to run before all threads are initialized. "Start Program" is the point all threads are initialized.

Solution:

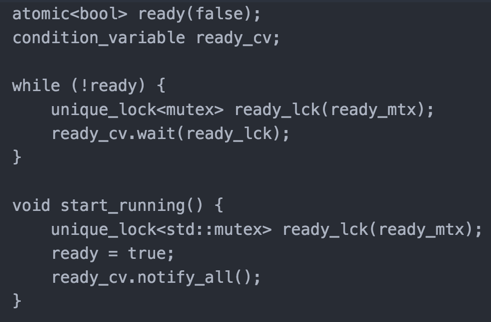


Figure 3. Solution for starting threads concurrently

As can be seen in Figure 3, I add another mutex named ready\_mtx to stop threads running while initializing reader and writer threads. In each reader and writer function, add a while loop to wait to run together. After all threads are initialized, execute start\_runing() function to notify all waited threads. It may not be necessary to make sure all threads start to run simultaneously. I added this part just in order to make sleep random numbers accurate.

1.4.2 I found printed messages are overlapped in Xcode output in Mac OS X. But it wasn't occurring in Ubuntu. It looks like following:

(924803550ns)Reader id: 0(Sleep for 315(( (9(9m9292i2424l4848l8181i2337s1705e5704c3335o4n6nnnsnsds)s)s)R)R)ceRe

2aea

daddeeerrr iiiddd::: 341(((SSSllleeeeeeppp fffooorrr 448784750 mmmiiilllllliiissseeecccooonnndddsss)))

It's because cout() function is not a single instruction. Several threads may call cout() function concurrently. It's necessary to make print\_message() function to run atomically.

Solution:

In order to guarantee printed messages are the same in different platform, I had to fix this issue. I solve this issue by adding another mutex named print\_mtx. In the print\_message() function, it try to get lock (print\_mtx) first. Once it gets lock, it will start to print. Modified print\_message() looks like:

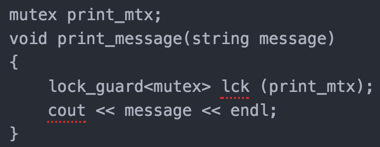


Figure 4. Solution for overlapping printing

1.4.3 In writer's while loop it has potential to cause race condition.

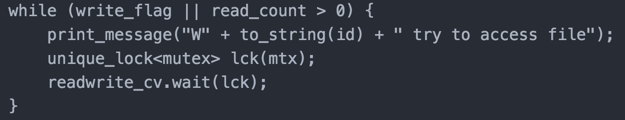


Figure 5. Potential race condition in writer() function

In case two writers are simultaneously executing while loop, since there are two comparison, they might not be executed atomically. Thus two writers would be enter critical section simultaneously.

Solution:

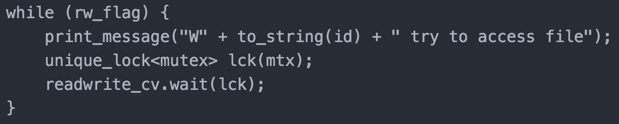


Figure 6. Solution for potential race condition in writer() function

As can be seen in Figure 6, I changed two variables to one. I found the condition check can be done by only one variable, that is to set to -1 for writing and increment for reading. Eliminating unnecessary variable not only make the code condense but also fix the potential race condition when checking two comparisons.

**Test Data**

We use the same number of readers and writers, and the same prefix, access, suffix time for

Monitor, Test and Set, Semaphore methods.

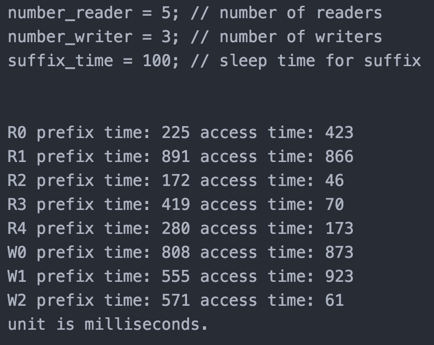


Figure 7. Test Data

**Program Output**

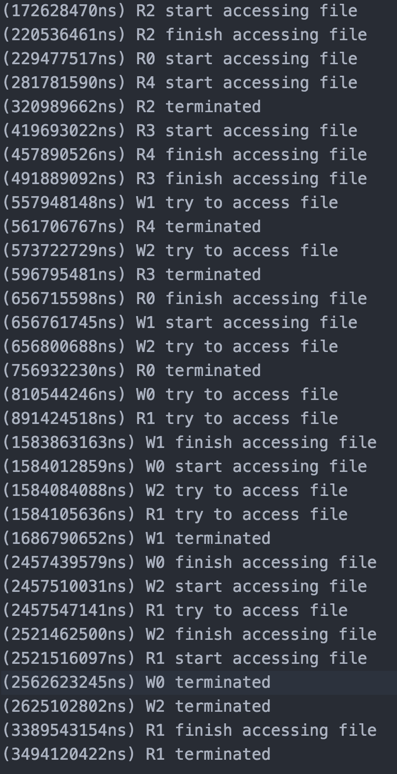


Figure 8. Output for Semaphore

**2. Test And Set**

Test and Set is a synchronization technique that is implemented by hardware instructions or by low level operating system methods. This method performs two instructions atomically. Atomic instructions are executed together, without being interrupted, or they are not performed at all. These atomic operations ensure that the Test and Set method is suitable for synchronizing multiple processes that might be trying to access the same common resource. This prevents race conditions from occurring when two processes try to access the shared space simultaneously, and receive unexpected or incorrect information as a result.

* 1. Implementation

Test and Set Method is implemented as a low level operating system or hardware instruction, but a possible implementation of the method is represented by the following code:

bool TS(bool lock)

{

if(lock) {return true;} //lock denied

else{

lock = true;

return false; //lock granted

}

}

In the above code, the test and set method can be seen to perform two instructions. The if statement tries check whether the lock is already set, and if so returns true, denying the lock to the process that requested it. If the lock is not already set, the function sets the lock, and returns false. These instructions are again performed atomically to prevent problems from occurring when multiple threads try to set the lock.

The Test and Set method is typically used within a program in a similar fashion to the following pseudocode:

Line1: Bool lock

Line2: Prefix i

Line3: TS(lock) {} //process waits in this loop until it returns false

Line4: Critical Section //process enters the CS

Line5: lock =0

Line6: suffix i

Here, we have declared a Boolean variable named lock in line 1. This is the lock used to prevent access to the shared resource. Line 3 executes the test and set method passing lock as a parameter. From the code shown earlier, we know that the test and set method returns true, if the lock is already set. Here on line 3 we can see that the process calling the test and set method must wait in an infinite loop while the method returns true. This means that the lock is held by another thread, and any process looking to use the shared resource must wait until it is released. Once, the lock is released, the next call to this method by a process will return false, setting the lock and preventing any other threads from entering the critical section concurrently. On line 5, the process has finished with the critical section and releases the lock, allowing the next process to enter the critical section.

* 1. Global Variables

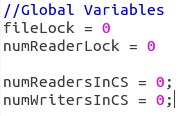


Figure 9: Global variables used in test and set synchronization program

Above, it can be seen that there are two locks being used in the program, which are used for the following:

fileLock: Used by reader and writer threads to lock the file and guarantee mutual exclusion

numReaderLock: Used by the reader threads when entering and exiting the critical section. This resource is protected with a test and set method to prevent a race condition from occurring when multiple reader threads try to increase or decrease the value at once. This lock is critical in allowing the last reader thread to release the lock on the file and allow writers a chance to write to the file.

The other two variables refer to the number of readers and the number of writers currently in the critical section.

* 1. Writer Program

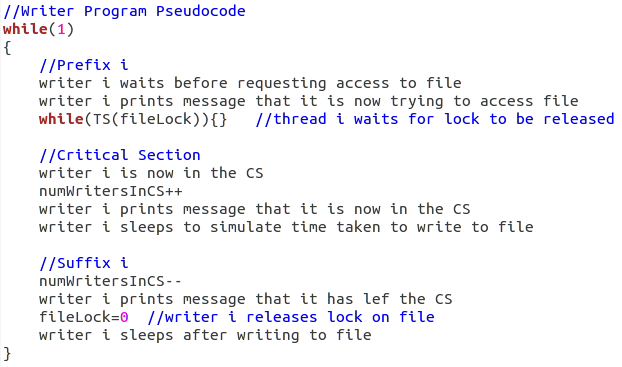


Figure 10: Writer program pseudocode

Above in figure 10, the Writer program is very simple to implement because only one writer thread is allow in the critical section at a time. The writer thread waits before requesting access to the file, then tries to set the lock. If the lock is already set, it waits. If the lock is not set, it sets the lock, enters the critical section, and increases the number of writers in the critical section. The writer thread sleeps in the critical section to simulate that it is actually writing to a file. After it is done it decreases the number of writers in the critical section, releases the lock on the file, and sleeps for a random period of time.

* 1. Reader Program

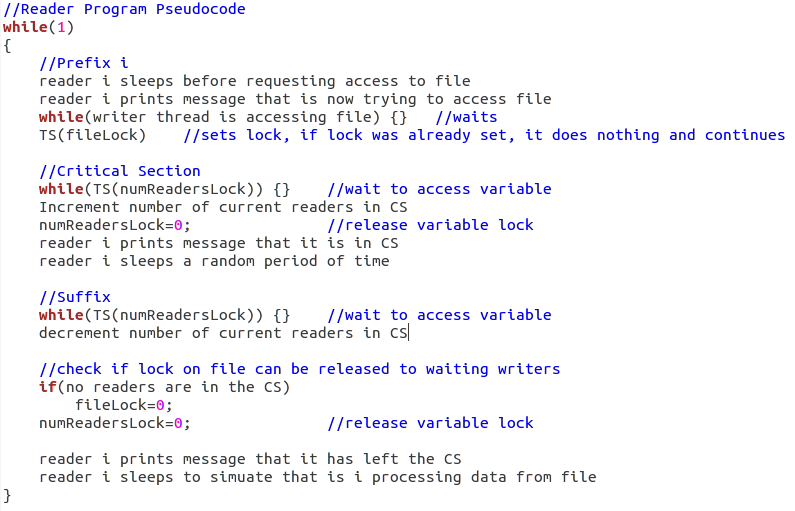


Figure 11: Reader program pseudocode

The reader thread program is more complicated to implement because multiple readers are allowed to access the file concurrently. First, the reader thread sleeps before trying to access the file. After this, it checks if there is a writer thread currently using the file. If so, it waits until it leaves. When there is no writer thread in the file, the reader thread sets the lock on the file. This lock is only set the by the first reader to enter the file. Next, the reader thread enter the critical section and increases the number of reader threads in the file. This variable is protected by a test and set instruction to prevent race condition from occurring. The reader thread then sleeps to simulate that it is reading data from the file. After it is done in the critical section, the reader decrements the number of readers in the file. This is again protected to prevent race conditions. If this reader thread was the last one out of the file, it releases the lock on the file so that writer threads waiting can enter. The thread then sleeps to simulate that the data it read is being processed.

* 1. Sample Output

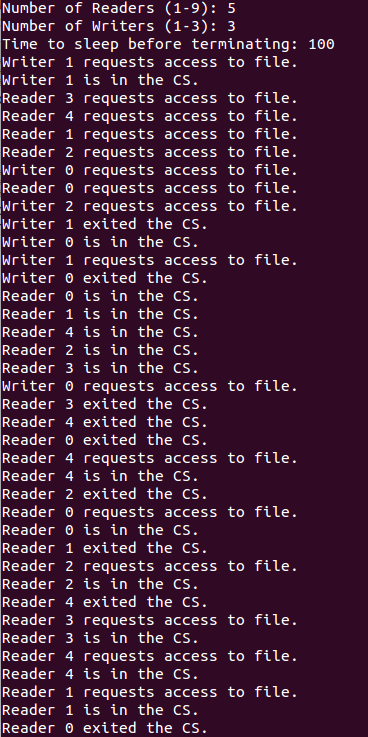


Figure 12: Sample output

The sample output above shows that only one writer thread is allowed to access the file at once. Further, no reader threads may access when a writer is holding the lock. It is also clear, that more than one reader thread can access the file at once. This sample output shows that mutual exclusion is guaranteed between writer and writer, and between reader and writer. The last half of the figure shows that as the number of reader threads increases, the writer threads are more likely to starve. This is expected because multiple reader threads are allowed to access the file at once. This can make it hard to writers to gain access.

**3. Monitor:**

Monitor is a similar concept to semaphore. This structure is used to ensure exclusive access to resources. In order to protect the data this structure only allows one thread to access at a time then update. This is similar to a bank vault the customer asks a teller to open the vault. The teller then allows the customer to go in alone open their specific box. If another person comes in to the bank then they have to wait until the person comes out. This helps keep control over the use of the global variables and important critical sections. Mutual Exclusion is enforced using monitor.

There are two conditional variables used in monitor they are similar to semaphore they are Wait() and Signal() . The wait operations allow the process to block execution until a certain condition is true. Then the monitor is unlocked to allow other processes to access the resource. Signal() is used to let other processes know that the resource is available. If several of these processes are in the waiting queue then it will pick one unless there is priority.

3.1 Monitor Implementation.

The basic idea is to lock out all other resources and hold this until the process completes or it cannot finish so it gives up its resources. The following is the abstract of monitor.

P1 {Acquire (lock) P2 {Acquire (lock)

If (head == null) if (head == null)

Signal (lock) Wait (lock)

Else head = Next item item = next item

Release (lock) Else (head = item)

Release (Lock)

Figure 13: Pseudo code for Monitor

3.2 Global Variables

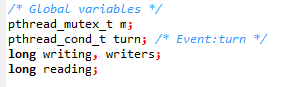


Figure 14:Global Variables

These are the structures used for the locking. In this case using mutex using pthreads will be the locking mechanisms to ensure that mutual exclusion is preserved. The condition which has been added Pthread\_cond\_t\_turn which will prevent the process from accessing the resources more than once before each has had an opportunity.

3.3 Reader function

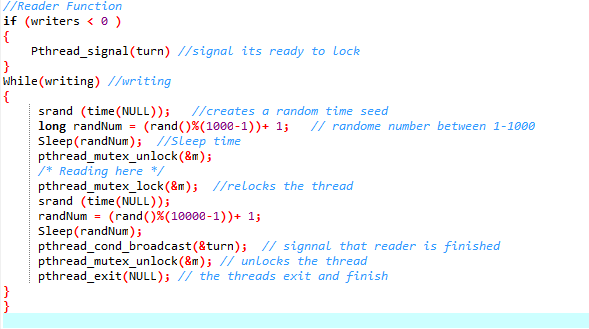
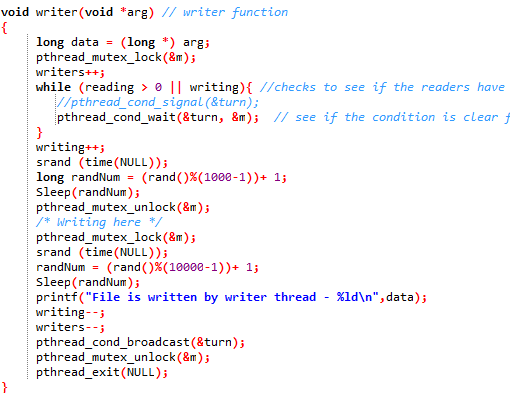


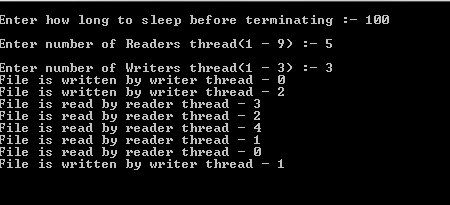
Figure 15: Reader Function

This function will holds the resource and blocks out all other threads even if they are a reader thread, but this will hold the thread in reader meaning that once a reader has it they will access one at a time but then release it to another reader and once those finish then it will give them back to writer.

3.4 Writer Function:



3.5 Output with test data



As seen in the output we can see that each of the threads has one chance in the critical section. If this program was running you would see that there is a significant time between readers and writers. This will be since each of the different functions is holding on to the critical section.

**Conclusion**

After comparing three techniques for multithreading-Monitor, Semaphore, Test and Set methods, we found that all three methods guarantee mutual exclusion between writer thread and writer thread, and between writer thread and reader thread. Multiple readers, however, are allowed to access the file at once which could lead to starvation of the writer threads as the number of reader threads increases. Deadlock is avoided by using semaphores and atomic locks to control when writer and reader threads are allowed to access the shared file. Race conditions are avoided by using locks and semphores to protect shared data that could otherwise be manipulated by multiple threads at the same time. Overall, we found each of the three methods of synchronization to provide valid solutions to the critical section problem.

We think Monitor is the best synchronization technique for producer-consumer problem. Because Test and Set has busy waiting which consume lots of CPU cycles. Semaphore uses global variable which can be modified in anywhere inside program. Monitor is also adapting Semaphore but since all shared data and member functions are encapsulated, only instances of Monitor can modify shared data. Thus Monitor provides both efficiency and secure.

**References**

1. Semaphore

http://en.cppreference.com/w/cpp/thread/condition\_variable

https://channel9.msdn.com/Events/GoingNative/2013/rand-Considered-Harmful

2. Test And Set

http://www.cplusplus.com/reference/atomic/atomic\_flag/test\_and\_set/