Tom Morin
Sam Adams

Comp 3500: Project #2

1) <u>What functionality needs to be included in the critical sections of reader and writer threads?</u>

The critical section of the reader thread needs to read the value of the shared value and the writer thread is responsible for incrementing the shared value. Mutual exclusion needs to be enforced throughout the critical sections of reader and writer threads. It is important that no process is denied entry to the critical section if it isn't being used by another process. It's equally important that deadlock is avoided. Mutual exclusion ensures that a process will only stay in the critical process for a limited amount of time (avoiding a crash), and it is a generally good solution to the race condition problem we faced.

2) <u>Briefly describe the solution for the synchronization problem.</u>

In this synchronization problem, the second readers-writers problem to be specific, we implement a method referred to as writers-preference. This method ensures that the writer threads are given priority such that no writer thread will wait longer than necessary once it has entered the queue. Each reader is made to lock and release the rmutex semaphore, whereas only the first writer thread is responsible for locking the rmutex semaphore and each writer thread thereafter can access it when available. The last writer thread is then responsible for releasing the rmutex semaphore. If a reader thread has locked the rmutex semaphore, the next writer thread in the queue will have to wait for the reader thread to release it, and then the writer thread will promptly lock the semaphore for itself (because it has priority).

3) <u>Describe the entry section and exit section of the reader and writer threads.</u>

The entry/exit sections for both reader and writer threads function in essentially the same manner. The entry threads start with an indication that a reader/writer is trying to enter it's critical section. Then the entry section is locked (helping to avoid a race condition and handle synchronization issues). The reader or writer then makes itself known and checks whether it is next in line for the critical section: if so, it locks the critical section and releases the entry section. The exit section works similarly, starting with a process locking the exit section. The process then makes sure it's the last process to leave the critical section and releases the critical section if true. The process then exits and releases the exit section for the next process.

4) <u>What semaphore(s) are implemented?</u>

We implemented two semaphores titled *rmutex* and *wmutex*.

Tom Morin
Sam Adams

5) <u>What are the initialization value(s) for semaphore(s). Why? Provide reasoning.</u>

Both of our semaphores, *rmutex* and *wmutex*, are initialized to 1. The number used to initialize a semaphore is the number of initial permits made available in your program. Initializing a semaphore to 0 will block the proceeding functions, essentially starting the program with a lock on the critical section. Initializing the semaphore to 1, however, establishes that one thread will be permitted to enter the entry section/critical section at a time. The first call will be accepted immediately and all other calls will be blocked until the first one releases the section.

6) <u>Briefly describe the purpose of each semaphore(s) used?</u>

The *rmutex* is responsible for locking and releasing the critical section during reader processes. The *wmutex* is responsible for locking and releasing the critical section during writer processes. Both of these semaphores play large roles in the avoidance of race conditions.

7) <u>Are there any additional shared variables used? If yes, describe their purpose.</u>

No, we only used one shared variable (val) which is of type Int and initialized to 0.

8) <u>Are there any situations when the implemented solution approach doesn't work? Briefly describe with an example.</u>

Our program struggles to manage excessively large reader/writer thread counts. When testing large numbers of writer threads specifically, we've been receiving a segmentation fault. It is likely due to a slight error in our code that we have not been able to identify. The program still runs as intended when process counts are low, so we know we are on the right track.