**Assignment 4: Producer-Consumer Problem**

**The bounded-buffer problem**

This project implements a solution to the producer-consumer problem involving a bounded buffer. One or more producer threads attempt to add an item to the buffer, while at the same time one or more consumer threads try to remove items from the buffer. If multiple threads attempt to access the buffer simultaneously, then a race condition may occur. Therefore, synchronization is essential to protecting the information in the buffer and effecting reliable behavior from the threads accessing the resource.

However, the use of synchronization tools alone, such as mutexes and semaphores, is not enough to ensure good program behavior. Synchronization must be carefully constructed to prevent deadlock among the threads, or their execution will stall indefinitely as they wait for locks to be freed, or for conditions to be satisfied that never are. The solution presented in this submission is in the pattern of a monitor: a class called Buffer which encapsulates the buffer*—*an array of integers—as well as a mutex and condition variables which ensure that only one thread at a time is manipulating the buffer.

**The monitor: class Buffer**

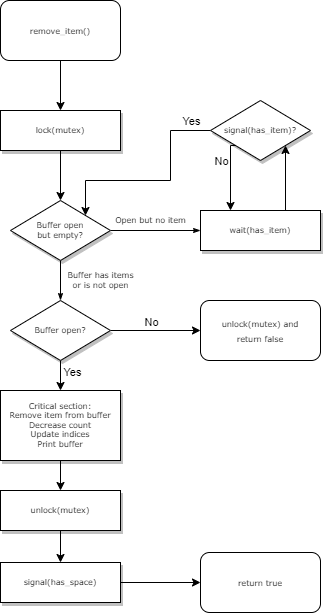
The Buffer class wraps an integer array with methods to insert and remove items in a thread-safe manner. The buffer’s members include one mutex lock to ensure one thread at a time is accessing the underlying array, which is required to prevent race conditions. The buffer also has two condition variables, has\_space and has\_items, which signal to the producer and consumer threads, respectively, when it is appropriate to perform their operations on the array.

The condition has\_space is necessary because the buffer is bounded: it has finite capacity. On the other hand, has\_items is needed because, in this implementation, the consumer thread does not know when it enters the buffer if the array is empty. In both cases, the signaling occurs entirely within the buffer.

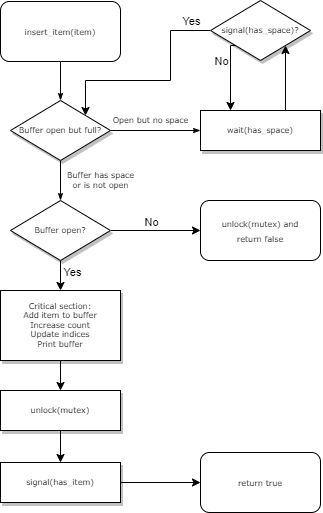
It is possible that the program may be implemented so that threads may check the state of the buffer before calling its modifying functions, but it would require accessor methods is\_full and is\_empty to be both protected by mutex, so that the correct size of the array contents is returned, and to happen strictly in sequence with a following modifying function call, so that between calls of the accessor and modifier functions, the array state does not change. Instead, this implementation puts the is\_full and is\_empty accessors in the private scope, to be called by the modifying methods only after the lock is acquired, ensuring that a thread can accurately judge the array state.

The diagrams below show the flow of the insert\_item and remove\_item methods.

The consumer function:



The producer function:



**Reader-writer preference**

It is possible for an application to prefer that, given a queue of threads waiting to access a resource, one type of operation be preferred and allowed ahead of another, disregarding the order in which they arrived. The program may privilege buffer reads before writes, where the first reading thread acquires a mutex, and only the last reading thread releases it. Likewise, all waiting writing threads may be allowed to execute before the buffer is read.

This implementation gives no preference to either reading or writing threads. The order in which threads access the buffer depends first on their acquisition of the mutex lock, then second whether their respective conditions are satisfied: has\_items for readers and has\_space for writers. The order of access does not necessarily follow the order that threads call the monitor functions.

**Thread Cancelation**

Because the program is intended to quit after a certain amount of time, the buffer must be able to signal to threads that no other changes may be made, whether the state of the array allows access or not. This is because as threads wait for a signal on their condition variable, they may become trapped waiting if the program stops and no further changes are made that validate the condition variable. For example, a consumer thread may be waiting for a producer to place an item into the buffer, but because the main thread has stopped and is no longer running producer threads, it is left indefinitely cycling through the while-condition-wait loop which precedes the critical sections of the modifying functions.

This implementation solves this hanging thread problem by introducing a Boolean variable open\_for\_work, which indicates whether the buffer is accepting modifications. The modifying functions, in addition to checking for is\_full and is\_empty, also check this open\_for\_work variable. The main thread closes the buffer by calling Buffer member function close, which changes open\_for\_work to false, and calls pthread\_cond\_broadcast on all signals to release any blocked threads and return them from their function calls. The threads are then joined by pthread\_join.

This deferred thread cancelation replaced an earlier version of the program in which the main thread called pthread\_cancel to terminate threads asynchronously. When pthread\_cancel is called, threads are terminated when they are at any *cancellation point.* Threads may be cancelled, for example, when they are waiting for a condition by pthread\_cond\_wait, which is defined as a cancellation point by the pthreads library, but not when they are in a critical section. Therefore, this asynchronous procedure generates reliable, predictable behavior. However, this was not in the sprit of the monitor pattern, in which the buffer should synchronize the threads accessing it, so the program was reformulated so that the buffer could signal threads to leave and call pthread\_exit elsewhere instead of being forcefully terminated.