* 1. Initialising to 0 allows one thread to have to wait for a certain event before executing a block (condition synchronisation). Initialising to 1 allows for mutual exclusion of a critical section. Initialising to an arbitrary n allows for allocation of a fixed number, n, of resources.



* 1. If two threads each perform an action where both actions need to be completed before either thread can continue, each thread can have a semaphore corresponding to the message of its action being completed. Ideally, each thread would complete its action, signal its semaphore, and then wait on the other thread’s semaphore. However, if the threads wait first and then signal, they can create a deadlock.
  2. If there was no guard semaphore, then two producer/consumer threads could enter their respective critical sections and access the same index in the buffer before that index gets updated. For example, two producers could put an item at the same index before either one updates the index.
  3. // TODO
  4. If wait()’s operations are non-atomic, it might check the integer, determining that it is greater than 0, and then the execution context might switch away. By the time it switches back, the integer could be 0, at which point it could be decremented to -1 (or indeed roll over to the maximum integer) and this thread and many others who wait on this semaphore in future would execute their code when they should still be waiting.

If signal()’s operations are non-atomic, then it may think that there is a thread waiting, but then the execution context switches away, and then when it switches back the waiting thread has woken up. Then the thread will try to dequeue an element from a now-empty queue which could cause memory corruption.



* + 1. Case 1: The buffer is empty and so in == out. In this case the consumer will wait on items
    2. Case 2: The buffer is full and so in == out. In this case the producer will wait on spaces
    3. Case 3: The buffer is partially full and so in != out. In this case it is safe for consumers to remove items from the buffer at the same time that producers are adding items to the buffer until one of the other two cases is reached. This is because separate indices are used for in and out.
  1. The semaphore queue will arise if there are more of one type of thread than the other e.g., a queue of consumers forming because there aren’t enough producers to keep up with the demand, or a queue of producers forming because the consumers aren’t creating spaces fast enough. This is a bad indication because it suggests that resources are not being allocated evenly.

The item queue will arise when there are more producers than consumers. This is again a bad indication because it suggests that too many resources are being allocated to producer threads.

* 1. It would likely be preferable to implement the MRU policy using a native library because this would be more efficient than writing application-level code on top of a library. Furthermore, the application code would have to include support for all of the edge-cases of the unknown/uncontrollable thread library, making it even less efficient and more likely to include bugs.



* 1. // TODO
  2. Since the buffers are shared between producers and consumers, they would both have to be methods on the same monitor which is conceptually confusing. This also eliminates the possibility of a producer and a consumer running concurrently which decreases the parallelism of the solutions.
  3. ‘Fairly’ could mean that no thread has to wait too much longer than any other thread in the queue. It could mean that there is no starvation (i.e., every thread eventually makes progress). It could mean that Threads which will take longer to execute aren’t prioritised over those which will execute quickly. It could mean that threads which were enqueued first get executed first.

We cannot tell whether the code in figure 1 implements round robin because it depends on how the signal() method is implemented. All we know is that if there are threads waiting one will be woken up, but we don’t have any information about how the system picks one from the queue.

Round robin is a scheduling system by which each of the threads in the execution queue are allotted a quantum of time to run, after which if they have not terminated, they are pre-empted and sent to the back of the queue. This does not include threads which are blocked, so when a consumer holds the guard semaphore it will not be pre-empted by another consumer thread until it has signalled that semaphore.

// TODO

* 1. The MRU policy might be helpful in a scenario where completing tasks is time sensitive, and so if a task is taking too long to complete, it likely isn’t worth completing. The MRU policy is helpful here because it prioritises threads that were scheduled most recently and so more tasks will be completed in a short time.

Priority-scheduling would be helpful in scenarios where some threads might be doing work which is important to the system but might take a long time to execute. Priority-scheduling allows the application to have control over which threads will be executed first, allowing important tasks to be completed quickly.

Round-robin scheduling is useful in scenarios where threads are all roughly equally important, but you don’t know in advance how long each one is going to take to execute. Round robin is useful here because it prevents any one thread from having to wait too long for its next quantum of execution.

* 1. This is likely to be a low proportion because this critical section occurs after the item has been produced (which is presumably the method which might take a long time to execute) so there is only a small window of time during which this priority inversion can occur.