Overview

In this third and final CS2002 Practical, I was asked to write an array-based implementation of a dynamically-chosen fixed-size generic Queue in C. I also had to design and implement tests to make sure that my implementation worked. I then had to use/extend my first Queue implementation to provide a thread-safe, blocking queue implementation, as well as design and implement tests for my BlockingQueue.

I'm proud to say that I've achieved the previously established goals:

- I've implemented a generic array-based Queue.
- I've implemented a thread-safe BlockingQueue.
- I've tested both implementations.

Design and Implementation: Queue

I implemented a dynamically-chosen fixed-size generic Queue in C as a struct in the Queue.c file. The struct is defined in the Queue.h header file. A Queue struct has 5 attributes:

- *arr*: The queue, represented as an array of *void** elements.
- *capacity*: The queue's maximum capacity.
- *size*: The number of currently enqueued elements.
- *front*: The index of the element at the front of the queue.
- *rear*: The index of the element at the back of the queue.

The queue's implementation is made of the 5 "typical" queue operations (enqueueing an element to the back of the queue, dequeuing an element from the front of the queue, getting the size of the queue, checking if the queue is empty, and clearing the queue), as well as a "constructor" function to initialise the Queue and a destroy function that frees the memory used.

The "constructor" function *new_Queue* takes a single integer, *max_size*, as argument. It creates a new Queue for at most *max_size void** elements, initialises all the struct's attributes, and returns a pointer to a new Queue on success and *NULL* on failure. The pointer to the new Queue, *this*, is created using *malloc*: if it has not been initialised properly, *this* should automatically be *NULL*. *arr* is also initialised as an empty array of $max\_size + 1$ *void** elements (the extra space is for the empty character) using *malloc*. The queue's *capacity* is set to the inputted *max_size*, both *size* and *front* are set to 0, and *rear* is set to -1. Note that I had to explicitly call the queue using the *(*this)* syntax to modify this specific queue's attributes. All I had to do is then return the pointer *this*.

Queue_enq takes a pointer to a queue (*this*) and a *void** element (*element*) as arguments, and enqueues the given *void** element at the back of this Queue. It returns *true* on success and *false* on failure when either *element* is *NULL* or the queue is full. Hence, I used an *if* statement to check if either this Queue is full (if $size = capacity$) or *element* is *NULL*. If not, then I first increased this Queue's *rear* by 1 modulo its *capacity*, added the *element* to the back of the queue (added it to this Queue's *arr* at position *rear*), incremented this Queue's *size* by one, and returned *true*.

Queue_deq only takes a pointer to a queue (*this*) as argument, and dequeues an element from the front of this Queue. It returns the dequeued *void** element on success or *NULL* if queue is empty. Hence, I used an *if* statement to check if this Queue is empty, using the

Queue_isEmpty Boolean function. If not, then I first accessed the front of the queue (the element at index *front* in this Queue's *arr*), increased this Queue's *front* by 1 modulo its *capacity*, decreased this Queue's *size* by 1, and returned the dequeued element.

Queue_size takes a pointer to a queue (*this*) and returns *size*, the number of elements enqueued to this Queue.

Queue_isEmpty takes a pointer to a queue (*this*) and returns *true* if the queue is empty (a.k.a. its size is equal to 0), *false* if not.

Queue_clear takes a pointer to a queue (*this*) and clears it, returning it to an empty state. A queue in an empty state has its *size* and *front* set to 0, and its *rear* set to -1.

Finally, *Queue_destroy* takes a pointer to a queue (*this*) and destroys it by freeing all memory used. This Queue uses memory to store elements in the array *arr*, as well as for itself. Therefore, I freed the array, and freed this Queue.

[More explanations on the functionality of my queue, and the use of modulo]

I decided to implement a circular queue to facilitate the dequeuing. Indeed, without the circular property, after dequeuing the front of the queue, all other elements need to be "pushed forwards" one spot. However, I do not have to go through this thanks to the *front* and *rear* attributes. For the following examples, suppose the queue has maximum capacity 5. When initialised, the *front* of the queue is set to 0, and the *rear* to -1, as shown in Figure 1:

front				
NULL	NULL	NULL	NULL	NULL

Figure 1: Circular queue when initialised (rear is not indicated as its value is -1)

After enqueueing an element (here, e1), the *front* stays the same, and the *rear* is increased by 1: enqueueing an element does not affect the front of the queue, but the rear of the queue is now 0, as shown in Figure 2.

front *rear*				
e1	NULL	NULL	NULL	NULL

Figure 2: Circular queue after enqueueing e1

This makes sense, as for a queue of one element, the front and back of the queue are the same. Now, enqueueing more elements will keep on incrementing the *rear*, as shown in Figure 3:

front				*rear*
e1	e2	e3	e4	e5

Figure 3: Circular queue after successively enqueueing e2, e3, e4 and e5

Similarly, dequeuing elements will also keep incrementing the *front* while keeping the *rear* unchanged, as shown in Figure 4:

			front	*rear*
e1	e2	e3	e4	e5

Figure 4: Circular queue after successively dequeuing e1, e2, and e3

At this point, our array is full, but our queue isn't (as we've just dequeued 3 elements): this is where the modulo comes in. When asked to enqueue another element (here e6), the *rear* is increased by 1, which would make it greater than the size of the array. This is why I set it to the value of *rear* modulo *capacity*: in our example, when *rear* is increased to 5, the modulo brings it back down to 0. Therefore, the old value e1 that has already been dequeued, is now replaced by the newly enqueued e6, as shown in Figure 5:

rear			*front*	
e6	e2	e3	e4	e5

Figure 5: Circular queue after enqueueing e6

The back of the queue is the front of the array: this is what makes my queue circular. Similarly, after dequeuing e4, the value of *front* would be 4. Therefore, dequeuing e5 would increase its value to 5. Hence, I used the same modulo as before to reset *front* to 0, as shown in Figure 6:

front *rear*				
e6	e2	e3	e4	e5

Figure 6: Circular queue after successively dequeuing e6 and e6

Testing: Queue

To test the implementation of my Queue, I wrote 21 unit tests in TestQueue.c. Most tests were written before implementing the code in Queue.c as per the Test-Driven Development method. Each test has a single assertion, as well as a documentation, which makes them very easy to understand. After running all tests (see README.md) and implementing all functions in Queue.c, all 21 tests passed.

Design and Implementation: BlockingQueue

I implemented a thread-safe Blocking Queue in C as a struct in the BlockingQueue.c file. The struct is defined in the BlockingQueue.h header file. A BlockingQueue struct has 6 attributes:

- *queue*: The blocking queue, represented as a Queue object.
- *capacity*: The blocking queue's maximum capacity.
- *mutex_enq* and *mutex_deq*: The mutexes used to enqueue and dequeue elements respectively.
- *sem_enq* and *sem_deq*: The semaphores used before enqueueing and dequeuing elements respectively. The value of *sem_enq* corresponds to the number of empty spaces in the queue (how many more elements can we enqueue before making the queue full), and the value of *sem_deq* corresponds to the number of available elements to be dequeued (how many more elements can we dequeue before making the queue empty).

The blocking queue's implementation is made of the same 7 previously explained Queue functions, as well as one extra *exit_error* function that prints out a message and terminates the code if an error is detected. The main difference between a Queue and a BlockingQueue is that *BlockingQueue_enq* and *BlockingQueue_deq* are thread-safe, using mutexes and semaphores.

The "constructor" function *new_BlockingQueue* does the same thing as *new_Queue*. It initialises all 6 attributes accordingly. The *queue* is initialised with *new_Queue* and both mutexes with *pthread_mutex_init*. The *sem_enq* semaphore is set to 20 using *sem_init*, and the *sem_deq* semaphore is set to 0. I also tested that those mutexes and semaphores were initialised properly: both *pthread_mutex_init* and *sem_init* have a POSIX return value of 0 if the mutex/semaphore has been initialised properly. And 0 in C is interpreted as *false*. Therefore, throughout BlockingQueue.c, I used an *if* statement every time I called any *pthread_mutex_* or *sem_* function: if the mutex/semaphore has not been modified properly, the function will return a value different than 0, which is interpreted as *true* in C, so an error has been detected hence *exit_error* will be called.

BlockingQueue_enq takes a pointer to a blocking queue (*this*) and a *void** element (*element*) as arguments, and enqueues the given *void** element at the back of this BlockingQueue if there is enough space. It returns *true* on success and *false* on failure when *element* is *NULL*. If the queue is full, the function will block the calling thread until there is space in the queue. To block the queue, I used the semaphore *sem_enq*: when the *BlockingQueue_enq* function is called, I immediately decrement the value of *sem_enq* by 1 using *sem_wait* (and an *if* statement to make sure it worked). *sem_enq* being initialised to this BlockingQueue's maximum capacity, it will reach 0 after *capacity* calls of *BlockingQueue_enq* (when the queue is full). Therefore, if the semaphore *sem_enq* has value 0 before enqueueing, *sem_wait* will block the calling thread until *sem_enq*'s value is increased. If there is enough space, I then locked the *mutex_enq* mutex to make the enqueueing thread-safe, called the *Queue_enq* function to enqueue the given *void** element, and unlocked the mutex after enqueueing. Then, if any element has been enqueued (if *Queue_enq* returned true), I incremented the *sem_deq* semaphore by 1, as there is now one more element that can be dequeued. Finally, the function returns the Boolean returned by *Queue_enq* (*false* if the *void** element is *NULL*, *true* otherwise).

BlockingQueue_deq only takes a pointer to a blocking queue (*this*) as argument, and dequeues an element from the front of this BlockingQueue. If the queue is empty, the function will block the calling thread until an element can be dequeued. Otherwise, it returns the dequeued *void** element. I followed a logic similar to the one used in *BlockingQueue_enq* to block the calling thread here, using the semaphore *sem_deq*: when the *BlockingQueue_deq* function is called, I immediately decrement the value of *sem_deq* by 1 using *sem_wait*. *sem_deq* being initialised at 0, *sem_wait* will block the calling thread until the value of *sem_deq* has been increased in the *BlockingQueue_enq* function. Therefore, one cannot dequeue from a BlockingQueue until an element has been enqueued, and one cannot enqueue to a full BlockingQueueue until an element has been dequeued. If the thread is not blocked, I then locked the *mutex_deq* mutex to make the dequeuing thread-safe, called the *Queue_deq* function to dequeue a *void** element from the front of the queue, unlocked the mutex after dequeuing, incremented the *sem_enq* semaphore by 1 (as there is now one more element that can be enqueued), and finally returned the dequeued *void** element.

The *BlockingQueue_size* and *BlockingQueue_isEmpty* functions are very simple: all they do is call the corresponding Queue function and return the same value.

BlockingQueue_clear takes a pointer to a blocking queue (*this*) and clears it, returning it to an empty state. There are two steps to clearing a BlockingQueue:

1. Clearing this BlockingQueue's *queue* object.
2. Resetting the semaphores to their original values.

Clearing the *queue* object was easy, all I had to do was to call the *Queue_clear* function. To reset the semaphores, I first had to access their current values using *sem_getvalue*. I then used two *for* loops, one to increase the value of *sem_enq* to this BlockingQueue's *capacity*, and the other to decrease the value of *sem_deq* to 0 (and using *if* statements every time to check the POSIX return values).

BlockingQueue_destroy takes a pointer to a blocking queue (*this*) and destroys it by freeing all memory used, as well as destroying all mutexes and semaphores used. This BlockingQueue uses memory for its Queue object as well as for itself. Therefore, I destroyed the Queue using *Queue_destroy*, the mutexes using *pthread_mutex_destroy*, the semaphores using *sem_destroy*, and freed the memory used by this BlockingQueue.

Finally, *exit_error* takes a pointer to a blocking queue (*this*) and an error message as a string (*char**). It prints out the error message, destroys this BlockingQueue using *BlockingQueue_destroy* and terminates the code. This method is similar to the *exit_err* function in the L17/bounded_buffer.c example. Note that this method is the only method that is not tested in TestBlockingQueue.c.

Testing: BlockingQueue

To test the implementation of my BlockingQueue, I wrote 22 unit tests and 2 helper functions in TestBlockingQueue.c. Again, most tests were written before implementing the code in BlockingQueue.c as per the Test-Driven Development method. All tests except *enqAndDeqAfterClearingFromFull* (new test), *enqFullQueue* and *deqFromEmpty* (modified) are identical to their homonyms in TestQueue.c. After running all tests (see README.md) and implementing all functions in BlockingQueue.c, all 22 tests passed. I will now explain the purpose and implementation of the three new/different tests.

enqFullQueue and *deqFromEmpty* are still used to check the behaviour of enqueueing to a full queue and dequeuing from an empty queue respectively. However, these functions (for blocking queues) no longer return *false* when full and *NULL* when empty respectively: the thread-safe implementation now makes them wait for the required conditions to be fulfilled. I decided then to use create two threads for each test: one thread will try enqueueing to a full queue/dequeuing from an empty queue, while the other will dequeue/enqueue an element. If my thread-safe implementation works, then the first thread in each test case will be blocked until the second thread finishes. To make sure that the first thread in each test case waits properly, I used the *sleep* function to force the program to sleep for 1 second before creating the second thread. I then accessed the return value of the function called in the first thread and checked for its validity. In *enqFullQueue*, the *BlockingQueue_enq* function called in the first thread does return *true*, and in *deqFromEmpty*, the *BlockingQueue_deq* function called in the first thread does return the element enqueued in the second thread (0).

The goal of *enqAndDeqAfterClearingFromFull* is to test if both semaphores have been properly reset in *BlockingQueue_clear*. If not, then *sem_enq* would still be equal to 0 after clearing a full queue, therefore blocking the thread when asked to enqueue a new element. Therefore, I decided to fill up a queue, clear it, and then try to enqueue a new element.

Conclusion

All in all, I have found this practical easier than the previous two. This wasn't the first time I had to implement and test a dynamically-chosen fixed-size generic queue object. Furthermore, I believe that I am getting more familiar with the C language. I did find implementing the thread-safe blocking queue challenging at first, as I did not really understand what mutexes and semaphores were. However, after completing the extra exercises and reviewing the lecture material, I quickly figured it out and was able to finish the practical.

If I had more time, I would've loved to do the same exercise, but this time with stacks. I also would've tried to transform this fixed-size implementation to a more generic version, where the user can define the queue's maximum size through the command line.