**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 *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 ) 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.

[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**

TT

**Design and Implementation: BlockingQueue**

TT

**Testing: BlockingQueue**

The testing of my implementation can be divided into different sections:

* Running my code with a default value of , using the Makefile and executing the TryStackFrames executable.
* Running my code with different values of , which I did in the *Tests.c* file.
* Using disassembly with the *objdump -d TryStackFrames | less* command.

I will now explain the different steps of my testing, and why the outputs are valid.

As explained in the README.md file, I compiled and ran my code with a default value of using the Makefile, and this is the obtained output:

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

*Figure 2: ./TryStackFrames output*

We can immediately see that the format of the output is identical to the one in the [System Specification](https://studres.cs.st-andrews.ac.uk/CS2002/Coursework/W08-C-Architecture/W08-C-Architecture.pdf). Furthermore, all 5 previously stated patterns hold. We can also label some registers and what they contain: in each stack frame, the registers printed on the third and fourth lines are used to store the result of the recursive factorial number (, initialised at 1) and respectively, as well as what the different stack frames represent. Please see *Figure 3* for a visual analysis of this output.

A screenshot of a computer

Description automatically generated

*Figure 3: Visual analysis of the ./TryStackFrames output*

Note that in each line, the first hexadecimal number corresponds to a register, and the second hexadecimal number is the value that is stored in this register. Also, the addresses in pink used for the different function calls are the return addresses of those functions.

I also implemented some extra testing in a separate test file called Tests.c: it contains 3 nearly identical copies of the *factorial* and *executeFactorial* functions, except that they don’t use *DEFAULT\_VALUE\_OF\_N* but other macro variables defined in the header file Tests.h. This is done to test the stack frames for different values of *n*, not just the default value of . I decided to implement three extra tests: one for a larger value of *n* (here ), one for a smaller value of *n* (here ), and one for . After running it with the Makefile, we obtain the following output:

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

*Figure 4: Extract of ./Tests output (first test for )*

A screenshot of a computer

Description automatically generated

*Figure 5: Extract of ./Tests output (second test for )*

A screenshot of a computer

Description automatically generated

*Figure 6: Extract of ./Tests output (third test for )*

The previously shown outputs follow the same format as the expected one in *Figure 2*. They also follow the same 5 patterns, and a visual analysis similar to the one done in *Figure 3* can be performed to analyse the results obtained here. One thing to notice is that there is no “last line” printed out in *Figure 6*: according to pattern 1, the number of stack frames printed out is equal to . Therefore, as we are testing here for , only two stack frames are printed out. And as the stack frames are outputted in “reverse order” (the *printStackFrames* call comes first, then the recursive *factorial* calls from 1 to *n*, then *executeFactorial*), the *printStackFrames* and *factorial(0, 1)* function calls (and their corresponding stack frames) will be printed out here.

Finally, I used the *objdump-d TryStackFrames | less* command to verify that my program is printing out the correct values, and this is an extract of the obtained output:

A screenshot of a computer

Description automatically generated

*Figure 7: Disassembly output for puts and printf*

A screenshot of a computer

Description automatically generated

*Figure 8: Disassembly output for factorial*

A screenshot of a computer

Description automatically generated

*Figure 9: Disassembly output for executeFactorial*

A screenshot of a computer

Description automatically generated

*Figure 10: Disassembly output for getBasePointer and getReturnAddress*

For each function, the 6-digit hexadecimal number in the first line corresponds to the function’s address. Therefore, the machine needs to use this address to call the function. We notice that every time the *callq* operation is used in *executeFactorial* (see *Figure 9*), the address that follows is the same as the first address in the other functions (see *Figures 7, 8* and *Figure 10*). Note that I didn’t include screenshots for *printStackFrameData*, *printStackFrames*, and the *main* function in TryStackFrames.c, but the previously explained statement also holds for these three functions.

**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.