## Design

This practical involved a two part design and implementation: first, a basic, not thread safe, fixed-size Stack; and second, a thread-safe BlockingStack which would have similar functionality to the initial stack, but would block threads when pushing/popping the stack when full/empty until the method can be ran successfully.

To design the Stack's store to be a fixed-size data structure holding generic pointers, I decided to use dynamic memory allocation to create a void* array (void**) which would initialise with enough space to hold a given maximum size number of void* pointers. As pointers are all the same size, this meant that all pointers could be held in this data structure making it generic, without the need to store the data itself onto the stack which would cause issues with different sized elements. This fit the specification well, although it does open up the risk for pointers to be pushed onto the stack from one scope and then popped out in another scope where the pointer no longer leads to the same memory location.

After designing and implementing the Stack to specification, I moved onto designing the BlockingStack. There were two main options for creating this thread-safe version of the stack: first would be to create a whole new, independent module, which would have similar, but thread-safe versions of the functions in the Stack; the second would be to integrate a Stack, with wrapper BlockingStack_ functions to ensure thread-safety.

I decided to go with the latter option. My reasoning for this lies in what resources the mutex actually needs to lock to make the implementation thread-safe. For the BlockingStack_ functions size, isEmpty, push, and pop, pretty much all the resources of the initial Stack need to be locked, and so re-writing the Stack functions would not really reduce the time the mutex is locked. Moreover, it would have involved a lot of redundant re-writing of code which can be avoided by calling the Stack functions from the BlockingStack. The final benefit is that the BlockingStack contains a Stack attribute, which can be accessed if threading is only needed for certain parts of the code to prevent unecessary blocking functionality where not needed, increasing the compatibility between the two.

Along with the Stack instance held in the BlockingStack, the other attributes I decided to hold in the struct are three semaphores for thread-safety. The first of which is a mutex implementation using a semaphore. I decided to do this instead of using the mutex themselves because it kept the code leaner as the only datatypes needed were Stack and sem_t. The mutex is required to prevent stack access from multiple threads, as I didn't want multiple threads pushing to the same stack at the same time and overwriting data.

The two other semaphores are used for keeping track of when the stack is empty or full. When the Stack is empty, the num_elements semaphore reaches 0, and so to pop any more elements off the stack, there must be a push call meaning the pop call will wait for that semaphore. The num_free semaphore means that push calls wait when the stack is full as the num_free reaches 0, and is released when a pop call is made. This works because in the push method, the num_free semaphore is waited on, and if the number is greater than 0, then the value is decremented and the code continues to locking the mutex and pushing the element. After pushing, the mutex is unlocked and a sem_post call is sent to the num_elements which enables another pop call to made, or prompts a waiting call to

resume. The pop method waits for the num_elements semaphore if it is 0, or decrements if/when it is greater and then locks the mutex and pops from the stack. After this completes, the mutex is unlocked and a sem_post call is made to num_free which enables another push call to be made later or prompts a waiting call to resume.

## Testing Methodology

I used a test-driven-development (TDD) methodology for this practical which is reflected in the order of the tests in the TestStack and TestBlockingStack source files.

Many of the BlockingStack tests mirror the Stack tests as the order of testing the foundational creation, size and isEmpty functionality before testing the more complex push and pop functions helped debug errors at each stage to detect which functions were not thread-safe.

## Testing and Implementation

My TDD for the Stack began with the utility functions which the further spec functionality relied upon. This meant checking that the new_Stack method returned a not NULL pointer. Then, I tested the Stack_size and Stack_isEmpty methods for that newly initialised Stack which tested their base functionality and made sure the Stack is initialised correctly.

After this, I added edge cases for the isEmpty and size methods for when a NULL Stack is given to prevent core dumps for bad parameters.

Then, I began adding tests to add the push and pop methods which used those size and isEmpty methods. I began with simple single push and push-pop tests which made sure pointers could be added and retrieved successfully. Then, I added in pushing and popping multiple pointers and in different orders.

It was here that I made the mistake of creating the store for the Stack as a void* instead of a void**. This caused issues when trying to dereference the values popped from the stack, and I was able to spot this issue quickly because of the TDD method.

I then made sure that the Stack could hold generic pointers as the specification required which passed once I added the void** fix for the previous tests.

As with the size and isEmpty methods, I made sure to add edge-case testing for NULL entries and NULL Stacks.

For the BlockingStack, I copied over the tests from the Stack tests and began working to integrate the Stack methods in a way which ensured thread-safety. This process was relatively simple given the procedure of gradually attaining the requirements in the order of testing and implementation given above.

However, there are two tests which do not appear in the BlockingStack tests which do appear in the Stack tests. These are the tests for pushOnFullStackFails and popEmptyStackReturnsNull tests. I could not include these because this would have caused the program to hang on waiting for the

thread blocked by the semaphore wait. I discuss this issue further in the further development section as I did not have time to implement a fix for this and the specification was not clear what the most desirable solution would be because it would depend on the use-case.

I wanted to make sure that my Blocking functions were thread-safe independently of which functions ran faster, and so I implemented two tests for each of the threading tests. These added a parameter to the thread functions which would simulate slow push or pop calls using the sleep method. First, I tested a slow push function, and then a slow pop function which made sure all calls were successful even if one thread ran much faster or slower than others.

## Further Development

One particular note for further development I would have liked to design would be to implement some kind of detection for when the semaphores used when pushing on a full stack or popping onto an empty stack block calls for too long. This issue arises when there is one of these blocking calls made, but the semaphore is never posted to and so the program hangs – e.g. if the stack is only pushed too and exceeds the max_size.

One idea for resolving this issue would be to have a timeout for the semaphore which would eventually return a failure code if this time is exceeded. This solution would impose it's own collection of issues however – as having the maximum wait time too short would cause some calls to unnecessarily fail, and too long causing poor performance. Also, if the program was intended to be a live service with constant push and pop calls, then we would not need to implement this.

An alternative solution would be to implement a signal to be sent once the program reaches it's end. This would terminate the hanging threads and let them return failure codes. However, similar to the above solution, this may cause issues if the program finishes before all of the pushing and pulling functions are successful – as it would mean that some calls are terminated unnecessarily.