Jordan Rowley
Computer Science
University of St. Andrews

W14-SP Project Report

Stacks and Thread Safety

The first section of the project involves developing and testing a generic, fixed-size stack implementation using dynamic memory allocation. My stack struct contains three attributes: maxSize, size, and array. The first of which will store the full size of the stack, the second will store the current size of the stack that contains elements, and the last of which is the array which will store the elements of the stack.

My constructor allocates memory for 2 int's and a number of void pointers which is the array. It will then initialise maxSize setting it to the size of the stack required and initialise size to zero before returning the new stack. The push function will first check if the stack is full, and if it is, it will of course fail and therefore return false. This will also occur if the element to be added is NULL. If neither of these conditions are satisfied, the push function can legally add an element to the stack and so the element is added to the array at the head and the size counter is incremented by one.

The pop function will first check if the stack is empty, in which case there is nothing to pop, and if it is empty, the function will fail and return false. If there are elements in the stack, the function will store the top element in a temporary variable, remove the element to be popped from the stack (by overwriting it to NULL), decrement the size by one, then return the value held by the temporary variable.

The rest of the functions are more simple. Stack_size will simply return the size attributes from the stack struct. Stack_isEmpty consists of an if statement which will check if size = 0 and if so, return true, otherwise it will return false. Stack_clear will iterate through every element in the stack overwriting it to NULL and decrementing the size until it is back at zero. Lastly, Stack_destroy will free the stack's malloc-ed memory.

As for testing my stack implementation, I wrote 7 tests in addition to the 2 tests provided. One to check the push function works as expected, one to check that the pop

function correctly pops elements from the stack, one to check that the stack size is properly updated, two to check that the isEmpty function works properly, one to check that Stack_clear works properly, and lastly one to see what happens when pop is called on an empty stack.

Now for the second part of the project, the thread-safe blocking stack. I'd like to note early that semaphores have deprecated functions on MacOS; I will touch upon this more later in the report. My blocking stack struct contains 6 attributes: the maxSize, size, and array as seen before but now also two semaphores named empty and full, and one mutex named mutex.

The constructor for the blocking stack is much like the contractor for the generic stack except it has memory allocated for the two semaphores and the mutex. All the simple functions as defined above work in the same way on the stack as on the blocking stack. The two functions that have changed are the push and pop functions.

The push function first checks if the element to be added is NULL, in which case the function returns false and exits. If the element to be added isn't NULL then the push function proceeds. the critical section of this code is adding the element to the stack and incrementing the size by one. To ensure that this critical section is thread-safe, pthread_mutex_lock is called on the blocking stack's mutex before this section and pthread_mutex_unlock is called after. This blocks multiple threads from accessing the critical section at once which guarantees that the threads are synchronised and that there is no race condition. Before the mutex lock, sem_wait is called on the 'empty' semaphore which contains the number of empty slots in the stack. This sem_wait function will decrement this semaphore by one. In the case that the stack is full, this semaphore will contain a negative value which will block the push function from adding an element to the stack until the 'empty' semaphore is updated to show that there is now space on the stack. After the mutex is unlocked we call sem_post on the 'full' semaphore which will increment 'full' and tell any waiting pop calls that an element has been added to the stack.

Similarly, in the pop function, we have a critical section enclosed by a mutex lock and then a mutex unlock to ensure that it can only be accessed by one thread at a time. This critical section consists of storing the element to be popped to a temporary variable, overwriting the element in the stack to NULL, and decrementing the size by one. Before the mutex lock we call sem_wait on the 'full' semaphore. This will decrement 'full' and if the stack was empty, this semaphore will now me negative causing the thread to wait until 'full' is increased by a sem_post call on 'full' when an element is pushed to the stack. After the mutex

unlock we call sem_post on ‘empty’ which will increment the ‘empty’ semaphore indicating to any waiting push threads that an empty space has opened on the stack.

Simplified, the ‘full’ semaphore stores how many spaces on the stack are filled and the ‘empty’ semaphore stores how many spaces on the stack are empty. The push function will check ‘empty’ to see if it can add to the stack and wait if it can’t yet. If there is space, the element is added and the ‘full’ semaphore is updated to indicate a space has been filled. The inverse is true for the pop function.

Before discussing my testing, I will elaborate on the previously mentioned deprecation of POSIX semaphore functions. MacOS doesn't support POSIX semaphores and therefore runtime warnings indicate that certain functions are deprecated which caused issues with blocking on my laptop. Since I have no access to the Linux operating system at home I could not run my tests properly. Due to this, I have tried to explain my logic in this report. I expect the program to run smoothly on a machine running Linux. MacOS does however support GCD semaphores but the specification expressed that POSIX should be used.

The tests I ran for the blocking stack were all also tests for the generic stack other than the final test. I wrote the final test to start two threads trying to pop from an empty stack. I expected them to wait until push was called and the semaphores were updated but this is when I ran into the deprecation issue and so the test did not run as expected. The test started the pop threads and then called push later and checked to see if the stack was empty after the threads had completed. This would be the expected outcome since once elements are pushed, the waiting pop threads would pop both elements off of the stack and it would be empty again. The test passed the isEmpty assertion but through debugging I discovered that the pop threads were executing before the push functions, returning NULL elements and causing the current stack size to drop below zero. Obviously this should never be the case and I expect that this will not happen on a machine that can correctly run POSIX functions.

I must note that, did the POSIX functions run, I would've written multiple tests to check the thread safety and the blocking. This would require creating a thread running the push function which takes in two arguments according to the interface in the .h file. Pthread_create can only pass one argument into a thread. I created a struct called argTuple in the TestBlockingStack.c file which has two attributes which correspond to the arguments expected by the push function. For the pthread_create function to pass in the correct arguments it would have to create a argTuple object and assign the attributes to the arguments. The push function’s arguments would have to change to a null pointer called args. Pthread_create could

then pass the push function a pointer to the argTuple object which, in the push function, could be dereferenced and the attributes accessed accordingly.

It was very frustrating encountering deprecation warnings and not being able to properly test my implementation. I have tried to make my report as comprehensive as possible to explain the logic behind my program as well as how it could be tested if I were able to run tests. I hope that it runs as expected on stacscheck.