CS2002 W14-SP - Stacks and Thread Safety

Jon Lewis (jon.lewis@st-andrews.ac.uk)

Due date: Wednesday 13th May, 21:00
Weighting: 25%

Objective

To gain experience with developing and testing thread-safe data types in C.

Learning Outcomes

By the end of this practical you should be capable of:

- constructing and using generic Stack implementations in C
- using dynamic memory allocation
- developing thread-safe data types in C using POSIX threads and synchronisation
- writing tests to ensure correct operation of your Stack implementations in C

Overview

In this practical you will write an array-based implementation of a fixed-size generic Stack in C. You will design and implement tests to check correct operation of your stack. You will then use/extend your first Stack implementation to provide a thread-safe, BlockingStack implementation and design and implement tests for your BlockingStack.

Getting started

To start with, you should download and decompress the `zip` file at

```
http://studres.cs.st-andrews.ac.uk/CS2002/Practicals/W14-SP/code.zip
```

Please note that the `zip` file contains a number of files in the `src` directory, some of which are blank or only partially implemented. **Take care that you do not accidentally decompress it again once you have worked on your assignment, thereby accidentally overwriting your own implementation.**

Stack Implementation

You are required to develop and test a generic, fixed-size stack implementation in accordance with the module interface in `Stack.h`. As usual, coding to an interface is important, so please don't change the Stack function signatures that are provided.

You are supplied with a `Stack.c` file containing the stack functions which do not do anything sensible yet and which you will have to implement. You are also provided with a `TestStack.c` file into which you should write tests to verify correct operation of your stack. The file contains two simple tests to show you how to write tests and call your test functions from the `main` function in `TestStack.c`.

You are supposed to use dynamic memory allocation and pointers to structs for your Stack data type and functions. You may find material in C_SP Lectures 7 - 10 useful, including the Person example L07-08/PersonStructDynamic. While you are supposed to implement this stack using a dynamically allocated array, you may also find the Linked List example L10/IntegerList interesting from the point of view of seeing how an integer list can be implemented in C using dynamic memory allocation. There are hints in the lecture how you might create a generic list instead.

You will have to add suitable members to the `struct Stack` definition in `Stack.h`. You should examine the function declarations and comments for each function in `Stack.h` to give you an insight into how you should implement your stack, what elements you should add to your `struct Stack`, and what the expected behaviour of the functions is.

For example, `Stack_push(Stack* this, void* element)` takes a `Stack*` argument (i.e. a pointer to a `Stack` struct) representing the stack on which to operate, and a generic (in the C sense) `void*` argument, representing the element that should be pushed onto the stack. As such, it should be possible to push a pointer to any value onto the stack. The code comment specifies that the function should return `true on success and false on push failure when element is NULL or stack is full`. As such, NULL values are not permitted in this stack and the `push` operation should return false in this case or if the stack is full.

The `Stack* new_Stack(int max_size)` "constructor" function takes a size parameter and returns a pointer to a new, suitably initialised `Stack` struct for which memory should have been allocated dynamically. As the maximum stack size is not known at compile time, you should also use dynamic memory allocation for the stack's `store`. You will have to allocate enough space for a dynamically allocated array of `max_size` `void*` elements.

As you develop your implementation, you should also write Stack tests (such as `pushOneElement()` for example) by writing functions in `TestStack.c` as indicated in the file and calling these by adding lines (such `runTest(pushOneElement);`) to the `main` function. Although your stack is designed to store non-NULL, generic `void *` pointers, it can also be used to store non-zero integers directly if you want, because an `int` can safely be cast to `void*` before pushing and cast back to `int` when popping.

There is a simple `assert` statement that you can make use of in tests and which is defined in "`myassert.h`". A `setup` function in `TestStack.c` is designed to create a new stack with some default size which can be used for each test. Similarly, a `teardown` function has been provided to destroy the test stack after each test. The `setup` and `teardown` functions are called by `runTest` before and after running each test.

Instructions on "Compiling Testing and Stacscheck" and some "General Tips" are given below after the "Thread-Safe Blocking Stack" section. Please do read these before starting, as they may help you avoid problems down the line.

Thread-Safe Blocking Stack

Once you have developed and tested your stack implementation and are sure that it works, you can move on. In this part, you are going to develop and test a thread-safe blocking version of your fixed-size stack in the associated files `BlockingStack.h`, `BlockingStack.c` and `TestBlockingStack.c`. The thread-safe BlockingStack has the same core operations as a Stack, but with some crucial differences in blocking when full/empty (on push/pop respectively) as outlined in the code comments in the module interface `BlockingStack.h`.

There are different ways to approach this task, one which involves re-use, where your `BlockingStack` essentially contains a `Stack` and the `BlockingStack_` functions add the thread-safety, the blocking aspects, and call the corresponding `Stack_` functions. Alternatively, you could design your `BlockingStack` as a completely independent module which does not have to call functions defined in `Stack.c`. There are probably pros and cons to each approach and you should consider these prior to starting this part.

The blocking property should not have to involve busy-waiting, so you will use semaphores for this bit. Thread-safety should be ensured by using mutexes. Please note that the BlockingStack itself and its operations should be thread-safe and satisfy the semantics of blocking when full/empty without the calling (client) code having to operate on any mutexes or semaphores directly. Client code should merely have to call the functions defined in the module interface. As such, you should consider where the mutex and semaphores should be stored for your `BlockingStack` object.

When designing, implementing and testing your blocking stack implementation, you should make sure you understand the material covered in C_SP Lectures 16 - 17 and may find it useful to study examples L16/threads2 (for creating and joining threads), L17/thr_broken_fixed5 (for use of mutexes in making access to `bowl` and `tap` thread-safe), L19/bounded_buffer and L19/bounded_buffer_named_sems (to see how mutexes and semaphores are used to provide thread-safe, blocking access to a shared integer array).

The tests you write for your blocking stack in `TestBlockingStack.c` should test the basic stack operations in a similar way to how your Stack was tested in the first part. However, you should also write one or more tests that create a number of threads (at least two threads) which access a single blocking stack. The threads might each execute either a suitable number of `BlockingStack_push` or `BlockingStack_pop` operations on the shared stack and the test might involve checking whether all values pushed onto the stack are also eventually popped. The L19 bounded buffer examples (and associated discussion in the lecture) may be of use here. You might also be able to demonstrate the blocking aspect visually in a different test/program by making the either the pushing or popping thread sleep occasionally such that the other thread has to block.

Compiling Testing and Stacscheck

You are supplied with a `Makefile` to help you build your stack implementations and tests. The following lines, executed from a Terminal window and from within the `src` directory, should permit you to build and run your tests over your stack implementations.

```
make
./TestStack
./TestBlockingStack
```

If you execute `make` on the code supplied in `code.zip` on StudRes, the compiler will generate a number of warnings, because the Stack functions are not implemented yet. As a result, executing `./TestStack` and `./TestBlockingStack` will show failures for the sample tests included as shown below.

```
palain:src jonl$ ./TestStack
Assertion failed in test newStackIsNotNull (TestStack.c line 76)
Assertion failed in test newStackSizeZero (TestStack.c line 84)
Stack Tests complete: 0 / 2 tests successful.

palain:src jonl$ ./TestBlockingStack
Assertion failed in test newStackIsNotNull (TestBlockingStack.c line 77)
Assertion failed in test newStackSizeZero (TestBlockingStack.c line 85)

BlockingStack Tests complete: 0 / 2 tests successful.
```

If you wish to build your project using `gcc` instead of `clang`, you can alter the first line in the supplied makefile from `CC = clang` to `CC = gcc`.

Stacscheck can be used to execute your own tests using the command

```
cd ~/CS2002/W14-SP
stacscheck /cs/studres/CS2002/Practicals/W14-SP/Tests
```

assuming `CS2002/W14-SP` is your assignment directory. We will run some of our own private tests over your implementations following submission.

General Tips

- Don't try to develop the whole implementation in one go as this will probably result in more bugs and lengthier, more painful debugging. Instead, it may be sensible to develop your implementation in stages using a TDD approach, writing tests as you go.
- At the start, you will have to choose a suitable struct to represent your stack and will have to ensure you can allocate memory for the stack and its elements. At that point you probably want to implement your `new_Stack` and `Stack_size` functions such that that you can get the two supplied tests to pass.
- The next step might be to implement `Stack_isEmpty` and to write a test for this function.
- Once your implementation passes tests for size and isEmpty, you can probably move on to the other operations: push, pop, clear, and destroy, writing suitable tests as you go.
- Only *free* things you have *malloc*-ed, or else bad things happen.
- Repeat the step-by-step development approach for your BlockingStack implementation.
- When using semaphores and mutexes for the thread-safe blocking stack, make sure you always initialise these properly.
- Always check the manual pages for the functions you call.
- Check POSIX function return values – checking the return values from mutex and semaphore functions is very important, as is writing suitable error and cleanup functions as shown in the L19 bounded buffer example – this is a good way to cut down on debugging time.

Deliverables

Hand in via MMS, by the deadline of 9pm on Wednesday of Week 14, a zip file containing:

- Your assignment directory with all your source code for implementation and tests.
- A PDF report describing your design and implementation, testing, any difficulties you encountered, how you tested your implementation. Take care to explain and justify your design and implementation decisions for stacks and tests in clarity and detail.

Marking Guidance

The submission will be marked according to the general mark descriptors at:

`https://studres.cs.st-andrews.ac.uk/CS2002/Assessment/descriptors.pdf`

A very good attempt achieving almost all required functionality, together with a clear report showing a good level of understanding, can achieve a mark of 14 - 16. This means you should produce very good code with very good decomposition and testing and provide clear explanations and justifications of design and implementation decisions in your report. To achieve a mark of 17 or above, you will need to implement all required functionality. Quality and clarity of design, implementation, testing, and your report are key at the top end.

Lateness

The standard penalty for late submission applies (Scheme B: 1 mark per 8 hour period, or part thereof):
`http://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties`

Good Academic Practice

As usual, I would remind you to ensure you are following the relevant guidelines on good academic practice as outlined at

`https://www.st-andrews.ac.uk/students/rules/academicpractice/`