CPSC 351, Operating Systems Concepts

Homework, Synchronization Thread Safe Stack

(*20 pts*) The provided C program contains an implementation of a stack using a singly linked list, and a client that uses the stack. An example of its usage is as follows:

```
Stack myStack = EmptyStack;  // define, create, and initialize an
...                          // empty stack called "myStack"
push(&myStack, 15);          // push the value 15 onto a stack called "myStack"
...
int value = pop(&myStack);   // pop a value from a stack called "myStack" and store
...                          // it in an object called "value"
```

Or, if you had several stacks, say four of them, and wanted to hold them in an array of stacks, it might look like this:

```
Stack myArrayOfStacks[] = {EmptyStack, EmptyStack, EmptyStack, EmptyStack};
unsigned const size     = sizeof(myArrayOfStacks)/sizeof(*myArrayOfStacks);
...
push(myArrayOfStacks,   15);  // push 15 onto the stack held in myArrayOfStacks[0]
push(myArrayOfStacks+2,  7);  // push  7 onto the stack held in myArrayOfStacks[2]
...
int value = pop(myArrayOfStacks+1);  // pop the value off the top of
                                     // the stack held in myArrayOfStacks[1]
```

The stack implementation provided works great in a single threaded environment but has race conditions and is not appropriate for a concurrent multithreaded environment. Your assignment is to fix that problem and to verify in a multithreaded program the problem has been fixed.

Submit through Canvas your source code and the output generated from running your program several times. See the note below about accumulating the output of multiple runs in output.txt.

Files provided:	Files to be delivered:
1. main.c	main.c (unchanged)
2. stack.h/stack.c	stack.h (modified)/stack.c (modified)
3. main-single.c.txt	output.txt

How to begin:

1) Study the provided program carefully, paying particular attention to:
   - *The stack's interface*. The stack's interface is contained in the header file stack.h. The `Stack` ADT contains two attributes (`_top` and `_size`) and three public functions, `push`, `pop`, and `isEmpty`. For simplicity, the stack only holds integers. The stack is unbounded, meaning nodes are created dynamically when elements are pushed onto the stack, and released when elements are popped from the stack. The definitions of both a stack node, and an empty stack are private, meaning they are declared in the header file stack.h but defined in the source file stack.c.
   - *How the stack is used*.
     - A single-threaded client of the `Stack` ADT is provided in the source file main-single.c.txt. A common real-world task you may face is updating a single-threaded program to a multi-threaded program. main-single.c.txt is that single-threaded program. Compare this with main.c to see the transformation steps required:
       - Identify the thread entry point function (`somethingReallyImportant` in this case). Notice how the function's signature has to change from passing multiple parameters to a single pointer parameter pointing to a new structure of parameters, and returning a pointer to the function's return value. This kind of thread-entry signature is required.
       - Notice the program must change from calling `somethingReallyImportant` directly to passing a pointer to `somethingReallyImportant` when creating a new thread.
       - Notice that after creating all the threads the program waits for all the threads to complete before exiting.

       main-single.c.txt is provided for reference only and there is nothing you need to do to this file other than understand how you get from this typical single-threaded starting point to multi-threaded solution. All of these transformation have been previously covered, but sometimes it helps to see another example.
     - A multi-threaded client of the `Stack` ADT is provided in the source file main.c. The only information available to the client about the stack is the interface (prototypes, structures, etc.) presented in stack.h. The client doesn't know (or care) how the stack is implemented.

       Function `main()` creates an array of stacks, calls a private helper function to do something really cleaver and important requiring that array of stacks, and finishes by printing some information about each stack. Many concurrent threads are created, each one executing `somethingReallyImportant` in that thread. In other words, `somethingReallyImportant` is executing in parallel many times over.

       The `somethingReallyImportant()` function takes two arguments: the stack array and the number of elements in the stack array. We simulate really cleaver and important work by repeatedly pushing random integers onto a randomly selected stack a random 67% of the time and popping from that randomly selected stack the other 33% of the time. Our goal here is to vigorously stress the design of the Stack ADT by altering the stack's content very frequently.
   - *How the stack is designed and implemented*. The stack's implementation is contained in the source file stack.c. As mentioned above, the definitions of `EmptyStack` and `StackNode` are defined here. Only the implementation of `Stack` needs to know these definitions. The clients of `Stack` do not.

     Functions `push`, `pop`, and `isEmpty` are standard implementations of adding and removing a node to and from the front of a singly linked list. After dynamically allocating memory for a node, `push` populates the node with a copy (that's important) of the client's data then links the node into the list at the front. The `pop` function returns the data that was removed and releases the related dynamically allocated memory. Unlike more robust implementations, our version of `pop` will simply return a default value if requested to pop an empty stack. Both functions update the stack's size.

Function `isEmpty` simply returns true if the stack contains no elements, and false otherwise.

Each public function (`push`, `pop`, and `isEmpty`) verifies the integrity of the stack when called. This is primarily for the developer as an aid to detecting a corrupt data structure during testing, and typically disabled for releasable builds. In our case, the size of the stack is compared to the number of nodes in the stack. If all goes well, the two are equal. But you may be surprised at how often walking the list uncovers attempts to dereference null pointers, attempts to access memory you shouldn't, or discovers the number of nodes and the size don't match.

2) Build, execute, and study the provided source and header files. Make sure you understand everything before making changes. Using the Build.sh script from your first homework, compile, link, and execute your program.

```
./Build.sh stack
./stack | tee output.txt
```

```
stack[0] (16460):    732,  2668,  1628,  1076,  3972
stack[1] (16710):     65,  3845,  2161,  2525,  1201
stack[2] (16697):   1066,  1914,  1114,   746,  4062
stack[3] (16634):   4079,  2771,  3403,  3199,  3255
```

Your results will be similar, but not the same

```
Program completed successfully
```

It should not work out of the box. Remember you have a multi-threaded client (`main`) using a single-threaded stack. To convince yourself the stack's implementation indeed works for a single thread, you can temporarily modify the number of threads from 200 to 1 around line 76 in main.c. Change:

```
#define THREAD_COUNT 200u
```

to

```
#define THREAD_COUNT 1u
```

Don't forget to restore the number of threads to 200 after you're convinced.

Use this baseline as a point of departure. Rebuild and re-execute often as you make changes. Hint: you can accumulate the output of multiple runs in the `outout.txt` file by adding the -a option to tee, like this

```
./stack | tee -a output.txt
```

Fix the problem:

1. Modify Stack to protect shared resources from concurrent access

The most important thing here is to identify what resource(s) (exactly!) are being shared - by name and address. Hint, it's not the source code, and it's not "the stack". If an operating system is (among other things) a resource manager, what resource (exactly!) is being managed here?

Once you answer that question, identify the smallest chunks of code that must access the resource atomically. Place those chunks of code in critical sections, each guarded by the resource's corresponding mutex. Identify the absolute smallest chunk of code possible, but no smaller. Only code that actually accesses the resource should be in the critical section. Remember, critical sections should be small, lightweight, and quick.

To do all that, you need to

a. define a mutex for each shared resource (see above, what resource is shared?). Not just one, one for every stack. Do this by adding a `pthread_mutex_t` attribute to the `Stack` structure in stack.h.

   You'll also want to update the `EmptyStack` object in stack.c to include initializing this new attribute.

   See https://linux.die.net/man/3/pthread_mutex_init. In cases where default mutex attributes are appropriate, the macro PTHREAD_MUTEX_INITIALIZER can be used to initialize mutexes that are statically allocated. The effect shall be equivalent to dynamic initialization by a call to pthread_mutex_init() with parameter attr specified as NULL, except that no error checks are performed.

b. Use `pthread_mutex_lock` and `pthread_mutex_unlock` to encapsulate critical sections. Take care not to recursively take the same mutex before releasing it (deadlock). Function `pop` calls function `isEmpty` and is a common place where deadlock occurs in a poorly designed implementation. Hint: resource management responsibility (i.e. critical section) is placed only on the implementors of Stack and never placed on the clients of Stack (i.e. nothing in main.c changes). Also, if a function takes a lock, remember to release the lock before returning.

c. The remaining details are left to the student. Good luck!

Verifying the integrity of the stack takes a long time, so give your program a minute or two to execute before declaring deadlock and terminate. If you're interested, use the timer program from last homework to measure it. Also, using "-g0 -O3 -DNDEBUG" instead of "-g3 -O0" at the compile command disables the integrity checks and creates an optimized (vice debug) program.