Threads

The goal here in this homework is to understand how the OS implements malloc and free on multi-threaded system and how we implement mutexes.

The final submission will include a stack server.

In this homework you may only use TBB objects (Test&Set, Compare & Swap) for sync. You must also implement your own multi-threaded malloc and free routines.

1. You should name all source files *.cpp and not *.c so that the compiler will use g++ to compile and not gcc. The reason for this is that the TBB library is written in c++ and not c. However, "syntax wise", there are no differences, and you can use regular c.

2. For your convenience – a c interface for compare and swap is attached at the end in the appendix (so that you wont have to know c++ to use tbb). However, it isn't that hard and I encourage you to use the tbb (C++) interface directly, instead of mine (Don't forget that I can have bugs too).

3. Practice: Implement a multi-threaded lock free stack using POSIX methods. (For this section only you may use conds, mutexes and the OS malloc) your stack must hold a variable length null terminated strings. Your application will work from the command line initially and network interface later and follow the following commands:

a. PUSH <text>\n

b. POP\n

c. TOP\n

4. The expected behaviour

a. PUSH will push <text> into the stack.

b. POP will pop text from the stack

c. TOP will display (output) the stack top.

d. Your output should begin with OUTPUT: prefix. These lines are the only lines that will be checked by us. (Your only command that generate output is the TOP command)

e. For the sake of clarity and simplicity. You may assume that all commands are only in uppercase. Text includes characters and can be capped at 1024 bytes. You may add additional commands for operation or debugging. (Such as command for quitting, displaying stack contents or clearing the stack.) Additionally, you may assume that all commands and text end with a \n (ASCII 10) and that this character will not be part of the text.

Threads

f. You may assume all input is legal and that TOP or POP are only given when the stack is in non-empty state but encouraged to give error message if you find illegal command. Use ERROR: <cause> prefix for errors.

g. We will ignore any other prefix for output. However, you are encouraged to use DEBUG: for debug output.

5. Start by implementing the stack so that it works with STDIN&STDOUT. (Single threaded for now.)

a. As the final implementation will work with sockets you are encouraged to use inheritance for inputs & outputs.

b. In this phase you can ignore syncing as you have only one thread conversing with the user. (So no syncing is necessary.) However, it is advised to add synchronisation into the code and start testing.

6. Practice: Merge the server implemented in ex 3 with the stack implemented in 5.

a. You now serve multiple client that transmit the stack command (instead of standard input)

b. You must implement locking now as you may receive multiple commands to the stack on different connections.

7. Practice: implement (using TBB) synchronisation primitives to replace the POSIX primitive you used in 6.

8. Practice: implement new methods to manipulate the heap (malloc, free) in multi threaded environment. Do not use POSIX primitives but the replacements you implemented in 6.
You must also implement calloc to support threads (as pthread will call calloc) realloc is nice to have but not required

9. Final submission: Merge the server implemented in 6 with the components developed in section 7+8. **You should submit both the server and client.**

10. Bonus (2 points): implement double ended queue instead of stack. Support the additional commands

a. ENQUEUE <text>\n: insert to the tail. (Otherwise same as PUSH)

b. DEQUEUE\n: remove the tail. (Otherwise same as POP)

11. Bonus (3 points): Implement the exercise in windows as well.

Use TBB in windows as well.

Threads

b. Use winsock to implement sockets.

c. Submit CMake or automake solution to support both platforms.

12. Additional instructions:

a. You are encouraged to implement inheritance for input and output, sync and malloc.

b. You must use dynamic memory allocation for the stack. Don't use arrays.

c. You should use brk(2) for malloc(3) implementation..

d. Test your work! You must develop at least one non-trivial tester. Submit it with your work.

e. The program should be built using a makefile and the target all (makefiles will be briefly explained in class). I.e. the program (both server and client) must be built when the user goes to the directory and types "make all".

13. To use this interface, add it to your code. You should use the atomic_counter type and pass it to the functions.

```
//CAS C INTERFACE {


typedef void * atomic_counter;


#include <tbb/tbb.h>


using namespace tbb;


/**
 * Return the new atomic counter on success or NULL on memory allocation error.
 * Also the counter is initialized to 0.
 *
```

```
 * @return The atomic counter
 */
static void *ose_four_allocate_atomic_counter(void)
{
    atomic<int> *ret = new atomic<int>();

    return ret;
}

/**
 * Free the atomic counter.
 *
 * @param atomic_ctr The counter to free.
 */
static void ose_four_free_atomic_counter(atomic_counter atomic_ctr)
{
    atomic<int> *ctr = (atomic<int> *)atomic_ctr;

    assert(ctr != NULL);

    delete ctr;
}

/**
```

```
 * Denote val as the value of the atomic_ctr.
 * If val==compare then val = swap is done.
 * That is if val equals compare then it is set to swap.
 * In any case the old val is returned.
 *
 * THIS ACTION IS ATOMIC!
 * That means that the compare and the swap are done atomically (i.e. no other change
 * to the value of the counter can occur between them - only before or after both
 * actions have been done).
 *
 * @param atomic_ctr The counter to (possibly) change and return the value of.
 * @param compare The value to base the change decision on.
 * @param swap The value to swap in case the compare equals the contents of the counter.
 *
 * @return The old value of atomic_ctr (before this function changed it, if it did).
 */
static int ose_four_compare_and_set_atomic_counter(
                atomic_counter atomic_ctr,
                int compare,
                int swap)
{
    atomic<int> *ctr = (atomic<int> *)atomic_ctr;

    assert(ctr != NULL);
```

```
    return ctr->compare_and_swap(compare, swap);
}

//} CAS C INTERFACE
```

14. this exercise like all other exercises will be checked for cheating using moss