Operating Systems (234123) - Spring 2015

Home Assignment 3 – Wet

Due date: Friday, 29.05.2014, 12:30 noon

Teaching assistant in charge: Arthur Kiyanovski arthurk@cs.technion.ac.il

Important: the Q&A for the exercise will take place at a public forum Piazza only. Please note, the forum is a part of the exercise. Important clarifications/corrections will also be published in the FAQ on the site. A number of guidelines to use the forum:

- Read previous Q&A carefully before asking the question; repeated questions will probably go without answers
- Be polite, remember that course staff does this as a service for the students
- You're not allowed to post any kind of solution and/or source code in the forum as a hint for other students; In case you feel that you have to discuss such a matter, please come to the reception hour
- When posting questions regarding hw1, put them in the hw1 folder

Introduction:

In this exercise, you will implement a simple version of a thread pool.

From Wikipedia - "A thread pool is a design pattern where a number of threads are created to perform a number of tasks, which are usually organized in a queue. Typically, there are many more tasks than threads. As soon as a thread completes its task, it will request the next task from the queue until all tasks have been completed. The thread will then sleep until there are new tasks available."

Our thread pool is created with N threads – so it can handle at most N tasks at the same time.

The producer of a task calls tpInsertTask() to insert a new task into the queue of tasks inside the thread pool.

An enqueued task remains in the queue until one of the threads in the thread pool finishes its previous task, dequeues the next one from the queue and executes it.

If a thread in the thread pool finishes its task and there are no enqueued tasks in the queue – it will wait (**No busy waits here!**) on the queue until a task is enqueued.

Since this is not a course in Data Structures – we include on the course site an implementation of a queue (osqueue.h, osqueue.c) you can use for the purpose of this assignment.

Interface:

Your thread pool should support the following operations (which are defined in threadPool.h you receive with this assignment):

1. ThreadPool* tpCreate(int numOfThreads);

   Description

   - Creates a new thread pool.

   Parameters

   - numOfThreads – number of threads in the thread pool.

   Return Value

   - A pointer to a thread pool handling struct, which will be passed to all other functions that operate on a thread pool.

2. void tpDestroy(ThreadPool* threadPool, int shouldWaitForTasks);

   Description

   - Destroys the thread pool and frees all the allocated memory.
   - Once this operation starts no new tasks can be inserted to the thread pool.
   - Once this operation is still taking place no concurrent tpDestroy() are allowed on the same threadPool. A call to this function after the threadPool was destroyed is undefined and will not be tested.
   - if shouldWaitForTasks != 0 – waits for all tasks (both running and in queue at the moment the function was called – no new tasks can be inserted into the pool after this moment) to finish and only then returns.
   - if shouldWaitForTasks == 0 – lets all the tasks that are currently running by threads to finish but none of the still enqueued tasks will be started.

   Parameters

   - threadPool – the thread pool to destroy.
   - shouldWaitForTasks – indicates if the method waits until all tasks finish running (both enqueued and running) or only those already running.

3. int tpInsertTask(ThreadPool* threadPool, void (*computeFunc) (void *), void* param);

   Description

   - Inserts a task to the task queue of the thread pool

   Parameters

   - threadPool – the thread pool.
   - computeFunc – the function that will be run by the task.
   - param – a parameter to with which computeFunc will be run.

   Return Value

- 0 if succeeds, -1 if fails in case tpDestroy is in the process of being called on the threadPool.

Mission:

Your mission is to implement the above interface in a file named threadPool.c, so that you will have a working thread pool – as presented in the introduction. You will also need to add fields to struct thread_pool as you find necessary.

Important Notes:

1. All functions must be thread safe – meaning that if multiple threads (outside of the thread pool) call the same function in the thread pool interface concurrently - all operations will succeed. Unless of course the thread pool has been destroyed, in which case the behavior is undefined.
2. Regarding the calling to functions while tpDestroy is already underway – tpDestroy might take a long time to complete in case the tasks yet to finish are long. Therefore while tpDestroy is waiting for these tasks to finish, it is important:
    a. Not to let new tasks be inserted into the queue – which might cause tpDestroy to run forever!
    b. Not to let tpDestroy be called again by another thread.

    BUT – we will be checking only "safe cases" when it is certain that tpDestroy is still waiting for the jobs to finish. We will not try to insert tasks or call tpDestroy when the current tpDestroy is after all the tasks are finished – since at these final stages of destruction the state of the thread pool is not well defined. You are expected to minimize synchronization issues as much as possible.
3. You will need to manipulate functions – such as when you need to save them in the queue and later execute them by a thread in the thread pool. You do this using function pointers. The following webpage includes everything you need to know about function pointers - http://www.cprogramming.com/tutorial/function-pointers.html
4. The things you are required/allowed to change in the files we gave you are:
    In threadPool.h:
    a. Add #include <header.h> statements - for example you will probably need to add #include <pthread.h> if you want to add pthread related fields inside struct thread_pool.
    b. Add fields inside struct thread_pool
    c. Add type definitions of your own - other structs, enums etc…
5. The things you are absolutely not allowed to change in the files we gave you:
    Well, basically everything that isn't written in the previous point is not allowed.
    More specifically:
    a. **Do not** change any other files – osqueue.c, osqueue.h
    b. **Do not** change the signatures of the functions in threadPool.h – we are going to call these functions in our tests.
6. Busy waits are not allowed – use the synchronization primitives you learnt about in class.

7. Use locking in a fine grained manner only to protect as small critical sections as possible. Locking larger than necessary amounts of code will cause reduction in points.
8. Test your thread pool using multiple threads.
9. You should handle your memory allocations and frees – points will be reduced for segmentation faults and memory leaks.
10. You must implement this exercise in C. Try to stick to the ANSI C standard.
11. Use only standard POSIX threads and synchronization functions.
12. There is no need to make the osqueue concurrent. You can lock the whole queue upon enqueue and dequeue.
13. We only require you to submit the implementation of your thread pool, but not any test programs that use the data structure. Of course, this does not mean you will not need a test program for debugging!

Sanity Check:

On the course website with the assignment, you will find the following files (except for those already mentioned):

1. test.c – a very simple sanity test program.
2. Makefile – a makefile that creates a test program a.out from libthreadPool.a and test.c
   A few notes about this makefile –
   1. –L. – means that the linker should look for libraries in the current directory.
   2. –lthreadPool – means the linker should link to a library named libthreadPool.a.
   3. Note that the libraries must be after the source files and that the "–L." must be before the source files otherwise the make will not work.

What you need to do to see that you pass the sanity check is:

1. Extract your submission zip file to some folder FOLDER – using the command – "unzip file.zip -d destination_folder"
2. inside FOLDER run – make – this should successfully create libthreadPool.a
3. Create a folder TEST
4. Copy your libthreadPool.a, threadPool.h, osqueue.h into TEST
5. Copy **our** test.c, Makefile files into TEST.
6. In the TEST folder run – make – this should successfully create a.out.
7. Run a.out – you should see 5 rows of "hello" appear on screen and your program should terminate normally.

If you completed all the above instructions and it all worked – Great! you have all necessary source files in your zip (just don't forget submitters.txt).

Note that the above sanity check is by no means enough – it's just to make sure you are on the right track.

Submission:

You submission should consist of two parts: an electronic and a printed submission.

The printed submission should contain the documentation of your design.

In it we expect you to describe the way you used POSIX threads and synchronization primitives to achieve the goals of synchronization and concurrency.

You should provide an explanation (no formal proof required) why your implementation does not have deadlocks.

The electronic submission should contain the implementation of your threadPool – both threadPool.h with the new fields in struct thread_pool and the functions implementations in threadPool.c.

You should create a zip file containing all the files that you have used.

Make sure your zip file contains these files without any directories in it:

- All source and header files that are part of your implementation (at least threadPool.c, threadPool.h and osqueue.h, osqueue.c).
- A makefile named "Makefile" that creates a library against which we can link our test code.

  The library should be compiled as a static library.

  The library filename should be libthreadPool.a.

  It should include the implementation of the functions defined in the Interface section.

  If you want to create a static library to contain several object files you can do it in the following way:

  ar rcs libthreadPool.a. obj1.o obj2.o

- A file named submitters.txt which includes the ID, name and email of the participating students.

The following format should be used:

Bill Gates bill@t2.technion.ac.il 123456789

Linus Torvalds linus@gmail.com 234567890

Steve Jobs jobs@os_is_best.com 345678901

Important Note: Make the outlined zip structure exactly. In particular, the zip should contain files (no directories!). You can create the zip by running (inside VMware):

zip final.zip Makefile threadPool.h threadPool.c osqueue.h osqueue.c … submitters.txt

The zip should look as follows:

zipfile -+

+- Makefile

+- threadPool.h

+- threadPool.c

+- osqueue.h

+- osqueue.c

+- ...

+- submitters.txt

Good Luck!

The Course Staff