Thread Pool (20 points)

This task introduces the concept of thread pools and explores the fundamentals of concurrent programming. The objective is to implement a thread pool in `C`, enabling efficient task management, resource allocation, and concurrent execution. By working on this assignment, you'll gain practical experience in designing and managing thread pools.

Introduction to Thread Pools

In the world of concurrent programming, managing and executing tasks efficiently is a critical challenge. This is where the concept of "Thread Pools" comes into play. A thread pool is a powerful mechanism for managing and reusing a group of worker threads, enabling them to execute tasks concurrently. Thread pools help streamline the allocation of tasks, ensuring that resources are utilized optimally and providing a structured approach to handle parallel processing.

Thread pools operate by maintaining a pool of reusable worker threads, which stand ready to execute tasks in parallel. When a task becomes available, it is assigned to an available worker thread, reducing the overhead of thread creation and destruction. This efficient approach optimizes resource utilization and provides an organized framework for parallel task execution, making it an essential tool in concurrent programming.

Problem Statement

In this assignment

- you are provided with the header file, `threadpool.h`, which defines the structures and functions necessary for building a thread pool;
- your task is to implement the missing functionalities in a file called `threadpool.c` based on the provided header file.

The functionality needed in `threadpool.c` includes creating and managing worker threads, safely executing jobs, and maintaining a job queue.

Note: Please read the header file before the instructions to understand the assignment better. You are also encouraged to read the `test.c` file to better understand what needs to be implemented.

To start preparing your solution, you are given three files:

- `threadpool.h`: The header file for the functions and needed structs for implementing the `threadpool.c`
- `test.c`: A test file written in `C` which will take your `threadpool.c` file and run 10 tests on it.
- `Makefile`: A Makefile is also provided for you which will help you in compiling and testing your assignment.

Makefile

Using only `make` command will compile the `threadpool.c` and `test.c` file and generate the test binary file.

```
make
```

After using `make` you can use `make run` to run the tests;

```
make run
```

You can also use the `make clean` command to remove the generated output files and compiled files:

```
make clean
```

threadpool.h

The `threadpool.h` header file defines the structures and function prototypes required for implementing a thread pool in C. This file is an essential part of the thread pool assignment, which aims to create a structured and efficient mechanism for managing and executing concurrent tasks.

Job Structure

The `Job` structure represents individual tasks to be executed within the thread pool. A job is an entity assigned with a function and the arguments required for running that function. We have two types of jobs: a) the jobs that can be executed in parallel b) jobs which we should make sure are run safely with among multiple threads

- `id`: A unique identifier for each job. (it's guaranteed that the tests will provide unique IDs)
- `function`: A pointer to the function to be executed. (learn more: pointers to functions)
- `args`: Arguments required for the function.
- `run_safely`: An integer argument that specifies whether the job should be executed safely using the `job_lock` mutex. (you have to run safely using `job_lock` if this variable is `1`)
- `should_free`: An integer argument that specifies whether the `*args` of the job should be freed after execution or not.
- `is_freed`: An integer argument that specifies whether the `*args` of the job is already freed or not.

Thread Structure

The `Thread` structure represents individual worker threads in the thread pool. It includes the following components:

- `id`: A unique identifier for each thread.
- `thread`: A `pthread_t` structure for managing the thread's attributes.

ThreadPool Structure

The `ThreadPool` structure encapsulates the overall thread pool and its components. It includes the following components:

- `jobs`: An array for storing the job queue.
- `job_count`: The number of jobs in the queue.
- `job_size`: The maximum size of the job queue.
- `front`: Pointer to the front of the job queue.
- `lock`: A mutex for ensuring safe access to the job queue.
- `job_lock`: A mutex for running jobs safely when specified.
- `jobs_available`: A semaphore for signaling job availability. (we'll use this in general to signal threads.)
- `threads`: An array of worker threads.
- `num_threads`: The number of worker threads in the pool.
- `stop_requested`: A flag used to request the termination of the thread pool.

WorkerInput Structure

The `WorkerInput` structure is a helper structure used for passing input to worker threads. It includes pointers to the `ThreadPool` and `Thread` to associate a specific thread with the pool.

Function Prototypes

The header file also provides function prototypes for various operations related to the thread pool:

- `thread_pool_init`: Initializes the thread pool with the specified number of worker threads and job queue size.
- `thread_pool_submit`: Submits a job to the thread pool for execution.
- `thread_pool_wait`: Waits for all worker threads to finish their tasks.
- `thread_pool_stop`: Requests the termination of the thread pool and all worker threads.
- `thread_pool_clean`: Cleans up resources used by the thread pool.
- `worker_thread`: The main function executed by worker threads for job execution.

By understanding and implementing the functions defined in this header file, you will be able to create a fully functional thread pool for concurrent task management in C.

Note: It is essential to implement the `threadpool.c` file according to the provided instructions to complete the assignment successfully. You can use the tests to ensure that everything is in place. Your codes will also be evaluated using hidden tests.

Submission

- Commiting your changes to your git repository is sufficient: you do not need to manually submit anywhere else.
- Only the `threadpool.c` file contained in your repository before deadline will be taken into account for grading.
- We require that you `commit` and `push` your changes to your git repository before deadline.

Note: You can write the code in any environment, but the tests should pass at least in the TinyCore VM. The `test.c`, `Makefile` and our own solution (which is not public) have been tested inside a TinyCore VM similar to the one you used for the previous assignments.

For debugging and verifying that all the resources have been cleaned, the following guides might be extremely useful:

- debugging guide -- https://web.stanford.edu/class/archive/cs/cs107/cs107.1242/resources/debugging.html
- the `gdb` debugger -- https://web.stanford.edu/class/archive/cs/cs107/cs107.1242/resources/gdb.html
- the `valgrind` memory leak checker -- https://web.stanford.edu/class/archive/cs/cs107/cs107.1242/resources/valgrind.html

Instructions for Implementing the solution in `threadpool.c`

In this section you can find instructions for each of the six functions you'll be implementing.

Initialize the Thread Pool (`thread_pool_init`):

- Initialize the variables and initialize jobs and threads queue using `malloc`.
- Initialize mutexes and the semaphore for thread synchronization using necessary functions.
- Create threads:
    - All of the threads will run `worker_thread` function
    - Use `WorkerInput` to pass the pointer of pool and pointer of thread to the worker function. You'll need these values in using those functions. (remember to cast to `void *`)
    - Use `pthread_create`.
    - You'll have to set thread_id for each thread. Keep in mind that threads have unique IDs from 0 to `num_threads` meaning that the first thread you create has the id of `0` and the last one has `num_threads - 1`

Submit Jobs to the Thread Pool (`thread_pool_submit`):

- Lock the job queue mutex.
- Check if the job queue is full. If it's full you'll have to print `job queue is full!\n` (This is important for passing the tests). Remember to free the `*args` of the job if `should_free` is true.
- Add the job to the correct position. (hint: use `job_count`)
- Signal that a job is available. (hint: use `sem_post`)
- Unlock the job queue mutex.

Worker Threads (`worker_thread`):

- Cast the `void*` argument to a `WorkerInput*` structure.
- Access the necessary information from the `WorkerInput` structure for getting access to the threadpool instance. You'll need this mainly for using the locks and also getting access to the jobs.
- Implement an infinite loop to keep the thread running. (threads will run until the termination is requested)
- Wait for signal on the `jobs_available` semaphore.
- After receiving the signal, check if the thread pool is requested to stop and exit the loop if requested. If it's not stopping, you'll proceed to the next steps and run a job.
- Lock the job queue mutex for access.

- Dequeue a job from the front of the queue.
- Unlock the job queue mutex.
- Execute the job, considering whether it should be run safely using `job_lock`. Remember to free the `*args` of the job if `should_free` is true.
- After the while loop, when the thread is exiting the `worker_thread` function, you should print `thread with id %d is finished.\n` (This is very important for passing the tests. You can use the `Thread` instance in the `WorkerInput` to get the id of the running thread).
- Remember to free the `WorkerInput`.

Stop the Thread Pool (`thread_pool_stop`):

- Set the `stop_requested` flag.
- Signal all worker threads to exit using `sem_post` and `job_available` semaphore

Wait for Worker Threads to Finish (`thread_pool_wait`):

- Use `pthread_join` to wait for all worker threads to finish their work.

Clean Up Resources (`thread_pool_clean`):

- Clean the `*args` of any job which `should_free` is 1 and `is_freed` is 0.
- Release the memory allocated for the thread pool and its components.
- Free the memory for the `threads` and `jobs` arrays.