Project: Simple Thread Pool Manager

In this assignment, you will implement a fixed thread-pool framework using `pthreads`. Thread pools are used to manage a set of threads that are then used execute tasks in parallel. Thread pools are used when there is enough parallelism during the lifetime of a program, that can be expressed as a repeated set of tasks. Each task should be long enough to make up for the overhead of thread creation. The number of worker threads remains constant in a fixed thread pool.

A thread-pool mechanism provides the following advantages over a system which dynamically generates new threads based on the work to be done [1]:

a. Unlike traditional threads, worker threads persist after their assigned task is completed and wait for the next task. This helps mitigate the overhead required for repeated thread creation and cleanup.
b. A limit on the number of threads that can be active at one time helps the application degrade gracefully. For example, consider a web server application where each HTTP request is handled by a separate thread. If the application simply creates a new thread for every new HTTP request, and the system receives more requests than it can handle immediately, the application will suddenly stop responding to *all* requests. With a fixed number of threads, the application will not be servicing HTTP requests as quickly as they come in, but it will be servicing them at a sustainable rate.

However, all design decisions have tradeoffs. A fixed thread-pool also has some disadvantages as well:

a. Inability to scale the number of threads as the rate of incoming work increases.
b. Worker threads consume resources (mainly memory) when they are blocked waiting for a new set of tasks.

Recall that a thread-pool framework incorporates the following ideas:

- A manager, with worker threads, whereby the manager is responsible for queueing work.
- Worker threads (also referred to as task threads) that perform the actual task, such as serving web requests, computing π, and so on.
- Task threads can have thread local work queues or a shared work queue. In case of a shared work queue, there needs to be a thread-safe mechanism to acquire tasks from the shared queue.
- The manager enqueues work in the work queue(s) and signals the threads. It then waits for all the task threads to finish their work.
- After the task threads complete the task, they signal the manager.
- The manager can then enqueue another set of tasks to the signaling thread.

Implementation

In this project, the focus is on the implementation of the framework itself. Therefore, the tasks performed by the individual threads are artificially simple. Your program will consist of –

a. A manager thread that spawns of *N* worker threads (*N* > 4)
b. Task-threads that block on a monitor [3] until the manager thread signals that work is available for them to perform. To implement this, use one (monitor, condition) variable combination. The semantics are similar to the 'sense-reversal centralized barrier' concept covered in chapter 9 of the Solihin text. Instead of the spin-wait, you should have the threads block on a monitor and have them woken up by the manager thread when it has queued up work for them.
c. Each task-thread has a local double-ended queue, or deque. This data structure can be implemented as an array with two index variables, i.e. *top* and *bottom*, with both initialized to zero. The deque is a fixed-sized array with 2048 entries.
d. For this part of the assignment, the deque will essentially be treated as an array (*bottom* index is not used). The next assignment will use the deque more effectively.
e. The manager thread should enqueue a set of tasks, and wait for all the task threads to finish the assigned task in the current batch of tasks before enqueuing the next set of tasks. Use the appropriate synchronization between the manager thread and the task-threads to achieve this.
f. The manager *'enqueues*' the task for each task-thread by initializing each task-thread's task-queue, i.e., by resetting the array index and the array entries to zero. It then signals the task-threads to start executing the tasks and waits for them to finish the tasks.
g. The task for each thread is to write its thread ID [4] in each of its own deque slots. The state of the deque when the task is finished will be:
    a. deque.top = 2048
    b. deque.array[0..2047] = thread-id
h. This constitutes as one batch of tasks.
i. Once all the task threads have finished their assigned tasks in the current batch, the manager can enqueue the next batch of tasks, i.e., step-f
j. Three such batches of tasks need to be performed, with simple statistics being printed out each time:
    1. loop count
    2. number of elements written
    3. time taken (CPU time)
k. The main threads and the task threads should then exit gracefully.

Submission

A working implementation would satisfy the functional requirements defined above and be deadlock and livelock free. Program crashes are also not acceptable.

Your implementation should compile under `grendel.ece.ncsu.edu` using the following command-line arguments:

```
gcc -g proj.c -Wall -lpthread -lrt
```

References:

1. http://en.wikipedia.org/wiki/Thread_pool_pattern
2. http://download.oracle.com/javase/tutorial/essential/concurrency/pools.html
3. https://computing.llnl.gov/tutorials/pthreads/
4. https://computing.llnl.gov/tutorials/pthreads/man/pthread_self.txt