Instructions: Implement a program that computes the centroid of the points stored in multiple files using a thread pool. The idea to solve this problem is illustrated in Fig. 1. We will have a queue of jobs and a pool of workers. Each worker will pull a job from the queue, execute the job, and return a result. The program launching the pool will specify the job that each worker will execute. Specifically, each job consists in opening a file containing points, load them into memory, accumulate them, and return the accumulated point and the number of points read from that file. The main program will launch the pool, and collect the accumulated points to calculate the final centroid. The main program will be given to you. This program assumes that the thread pool and queue work correctly and will invoke them to compute the centroid. To implement such a program, the project is divided into three parts. See below.

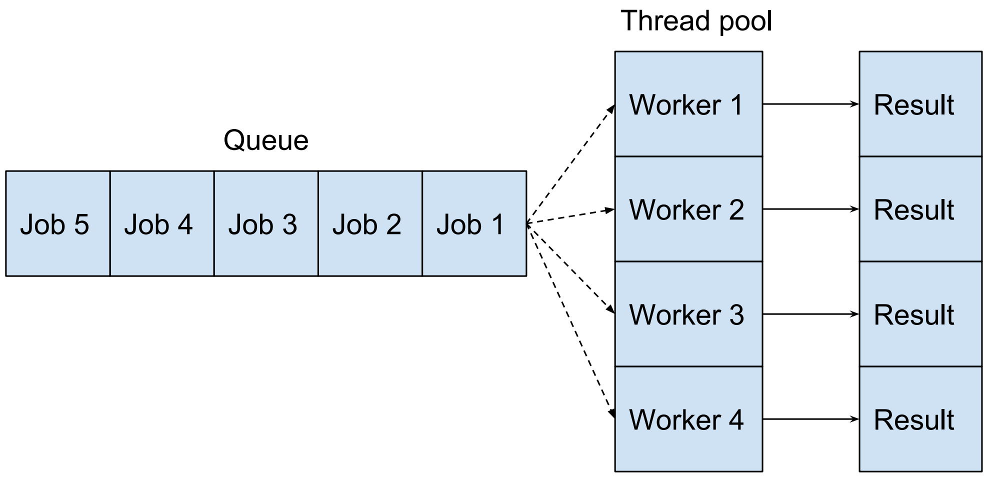

Figure 1: Overview of the components working together to produce results. The queue will store the jobs to be processed, while a worker in the thread pool will pull a job from the queue, and execute the job. The main program will collect each of the results of each job and will compute the centroid.

In summary, your solution must be comprised of four files:

1. **queue.h** This header file will include function prototypes and structures for a Queue.
2. **queue.c** Implementation of a Queue.
3. **thread_pool.h** This header file will include function prototypes and structures for a thread pool.
4. **thread_pool.c** Implementation of the thread pool.

Special Instructions:

- Download the file final_project.zip from e-campus. This file contains tests for the queue and threadpool. Also, the file contains the program that calculates the centroid using your queue and threadpool implementations as well as several files with the data points.
- For each file, add a comment at the top indicating your names, mix emails, and student ids.
- Run the tests for the queue and threadpool to make sure your implementation works correctly.

Due Date: No later than May 4th, 2017 11:59pm
Submission: Zip all your files in a zip-file named lastname1_lastname2.zip and submit it via e-campus. Replace lastname1 and lastname2 with the family names of the members in your team.

Part 1: Send email to instructor

(10 pts) Send email to instructor no later than Apr 7th. Send an email containing the two members of your team to the instructor. The email subject must be: “CS 350 - Final Project Team”.

Part 2: Queue

(45 pts) Implement a Queue. The goal of the queue is to store “jobs” for each of the threads in the threadpool. For this implementation create a structure for the Queue with the following members:

1. A pointer to a Job named first, which points to the first element in the queue.
2. A member that holds the size of the queue.

Your implementation must ensure the following functions:
Constructor:

1. Name: CreateQueue
2. Return type: A pointer to the created queue.

Destructor:

1. Name: DestroyQueue
2. Return type: Nothing.
3. Input: A pointer to the queue to destroy.
4. Note: Your destructor must destroy/deallocate all the jobs inserted in the queue.

Enqueue: Inserts a job to the queue.

1. Name: Enqueue
2. Return type: Nothing.
3. Input: A pointer to the queue.
4. Input: A pointer to the job to insert.
5. Note: Your implementation must assume that the job is allocated in the heap, and that the Queue will be the owner of such a job while in the queue.

Dequeue: Pops a job from the queue.

1. Name: Dequeue
2. Return type: A pointer to the job that was popped from the queue.
3. Input: A pointer to the queue.

GetQueueSize: Returns the size of the queue.

1. Name: GetQueueSize
2. Return type: An integer number indicating the size of the queue.
3. Input: A pointer to the queue.

Test your queue implementation by creating a binary using the test_queue.c and your queue implementation.

Part 3: Thread pool

(45 pts) Implement a thread pool. The goal of the queue is to have a number of workers (threads) that will execute the jobs enqueued. This implementation will use the pthread library. Create a structure for the ThreadPool with the following members:

1. A member to store the number of workers (threads).
2. A boolean variable that indicates if the thread pool is actve.
3. A pointer to the jobs-queue.
4. A pointer to the array of workers, i.e., an array of pthread_t.
5. A member named joinable to create joinable threads using the pthread library.

Your implementation must ensure the following functions:
Constructor:

1. Name: CreateThreadPool
2. Return: A pointer to the thread pool.
3. Input: A number specifying the number of workers.

Destructor:

1. Name: DestroyThreadPool
2. Return: Nothing
3. Input: A pointer to the ThreadPool

EnqueueJob: Inserts a job into the queue.

1. Name: EnqueueJob
2. Return: Nothing
3. Input: A pointer to the thread pool.
4. Input: A pointer to the job to insert into the queue.

ExecuteJobs: Launches the workers that will execute the jobs in a queue.

1. Name: ExecuteJobs
2. Return: Nothing
3. Input: A pointer to the thread pool.

IsThreadPoolActive:

1. Name: IsThreadPoolActive
2. Return: True when active, and False otherwise.
3. Input: A pointer to the thread pool.

GetNumberOfRemainingJobs:

1. Name: GetNumberOfRemainingJobs
2. Return: The number of remaining jobs in the queue.
3. Input: A pointer to the thread pool.

The queue implemented earlier must be adapted to handle synchronization. To do this, add a pthread_mutex_t member to the Queue structure, and adapt functions that can face race conditions.
Test your queue implementation by creating a binary using the test_thread_pool.c and your queue implementation. See the compilation comment in the source file.

Test program

To test the entire program, compile compute_centroid_in_parallel.c. See the compilation comment in the source file. The files to test this program are: points$X$.bin, where $X = 1, \ldots, 8$ and input_points.txt. When everything is working correctly, you should get the following centroid:

$$x = -0.000418, y = -0.000226, z = -0.000008 \tag{1}$$

Dependencies

The only dependency for this program is the pthreads library. Many systems (e.g., OSX, Linux) already have pthreads library installed. If using MinGW, make sure that library is installed. This project assumes a *NIX environment (e.g., OSX, Linux, or MinGW). Unfortunately, Windows cannot run the phtreads library since it is not a UNIX operating system. Options for windows users are: 1) install a virtual machine; or 2) use the shell servers.