# Work stealing Threadpool

Kidus Bekele, Jiaqi Ma

## How to run:

```
[kbekele@spruce tests]$ make
make: Nothing to be done for 'all'.
[kbekele@spruce tests]$ ../scripts/fjdriver.py -a
```

## Implementation:

Thread_pool_new

- This function takes in the number threads as an input and returns a thread pool object. Inside the function, memory is allocated to the pool and the different variables in the pool struct are initialized. Then a loop iterates until the desired number of threads are created. Pthread_create is called when creating those threads. A function named start routine is passed into it as one of the parameters.

Start_routine

- This is the main function that dictates what the worker thread does. A thread pool is passed into it as a parameter. It first stores the current calling thread in a thread local variable. Then it enters a large while loop where the main operations are caried out. The calling thread waits for a signal telling it that there is a task available. Then the worker tries to grab a future first from its own queue. If its own queue is empty then it checks the global queue. If the global queue is also empty, it moves on to the work stealing approach. This is done by looping through a list of threads and checking their queues. If a future is found then the calling thread steals the task. Finally the task is done. The loop is broken only when the pool is shutting down.

thread_pool_shutdown_and_destroy

- This function shuts down the Threadpool and deallocates memory. It takes in a Threadpool as a parameter. It signals that the pool is shutting down and lets go of the workers that are waiting. It then joins the threads then frees all the memory that was allocated.

thread_pool_submit

- This function takes in a thread pool, a fork join task and the data for the task. It first initializes a future object and all the variables that are in the future struct. Then if the submitted task is internal, it is added to the calling threads local queue. If its external, it is added to the global queue. It finally signals workers to get to work and returns a future.

Future_get

- This function makes sure the task of a future is executed, then returns the result. It implements work helping by performing the task if its not started. If the task is in progress, it waits till its done. This waiting is done by a semaphore, which gets posted to when a task is done. Finally the result is returned.

Future_free

- This function takes in a future then frees the allocated memory.

Optimization

- The optimization for this project is done through the use of locks. Specifically using separate locks for when accessing independent data. Meaning, the worker threads have individual locks for when their local queue is accessed. There is also a global queue lock that is held when accessing the global queue. Finally there is a pool lock that is used when updating some information found in the pool struct. This separate lock implementation helps by preventing unnecessary locking of data that is not used. Also, a work stealing implementation is used. This greatly helped in completing tasks quicker as there wont be any idle workers if there are any tasks.