Project 2 - “A Fork-Join Framework” Description

In project 2, our thread pool implementation for dynamic task parallelism focuses on the execution of fork/join tasks. In the following paragraph, we will describe our implementation process and thoughts as well as the strategies we used in the thread pool. Our implementation uses a work stealing strategy to spread out the work evenly such that it will have a better load balance. We have two mutexes to protect the global_queue and worker_list respectively, and each worker thread will have its own mutex to protect the local task queue.

struct thread_pool * thread_pool_new(int nthreads);
The thread_pool_new function allocates the memory for the pool object and initializes the global_queue, worker_list, mutexes, and a condition variable. Then it creates “n” worker threads by calling the pthread_create() function, and adds them to the worker_list. The new thread starts execution by calling start_routine() and arguments are passed in start_routine().

static void *start_routine(void *args);
This function will first identify the current worker thread from the worker_list, then pop out the tasks from the worker’s queue and execute them. If the current worker thread runs out of tasks, it will check the global queue for tasks and execute it if a task can be found. Otherwise, it will try to steal tasks from other workers’ queues (details are described below).

static struct list_elem *work_stealing(struct thread_pool *pool);
Our thread pool implementation has a work-stealing function which will be called if the worker thread has no task in its queue and the global queue is empty. Then it will steal tasks from the top of other workers’ deques.

struct future *thread_pool_submit(struct thread_pool *pool, fork_join_task_t task, void *data);
There are two different situations we need to consider when submitting a task to the pool, mainly internal and external task submission. We used a thread-local variable to distinguish these cases. If the task is submitted externally, it will be added to the global queue contained by the threadpool. If it is submitted internally, we will access the current worker’s deque and push the task to the front. In the list_push process, we have two different mutexes to protect the global queue and the worker’s queue.

void *future_get(struct future *future);
When the client wants to get the completed task affiliated with the future, our implementation will consider two cases. First, if the future has not been completed when future_get() is called, it will wait until it finishes. Second, if the future has not been scheduled yet, the current worker thread will grab the future from the queue and execute it.

void future_free(struct future *future);

This function will free the memory for a future instance allocated in thread_pool_submit()

void thread_pool_shutdown_and_destroy(struct thread_pool *pool);
When this function is called, it will shut down the pass in the thread pool. We have a “destroy” key to indicate the status of the threadpool, so it will be set to 1 in this case. Next, our program will loop through the worker list and join all the workers that are created. Then, we will remove the worker from the list and free them orderly. Finally, we will free the pool itself.