CS 3214 Project 2 - Fork-Join Threadpool

Jiahui Huang - hjiahui7

Keren Chen - keren9

Abstract

This is project description for the fork-join threadpool project.

The project provides a smart threadpool which allows user program to be executed in parallel, which feature is especially useful when it comes to programs utilizing the “divide and conquer” concepts and recursive executions.

The threadpool is able to automatically organize and execute processes in specified number of threads. The interaction between workers and between works and threadpool is realized through the built in mutex and semaphore features of C. To find more information, please visit the manuals of “semaphore.h” and “pthread.h”.

Installation

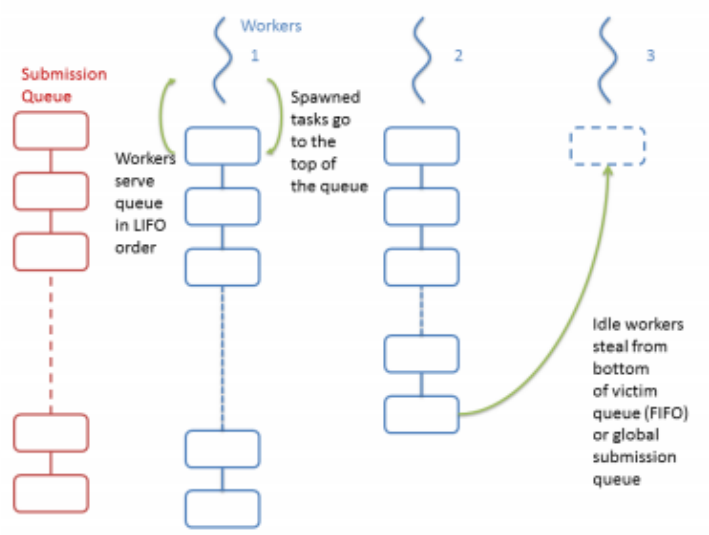
By assumption, users of threadpool should have their own copy of *threadpool.h* ready.

To incorporate the threadpool into any process, simply find the *threadpool.c* source file included in the tar file, and include it under the directory of your other source files. Do not forget to also include the *threadpool.h* header file in source files using functions from *threadpool.c* directly.

Then, follow the instructions provided in “How to Use” part to incorporate the threadpool.

How to Use

API and Implementation



As suggested by the visualization on the left (credit to the project2 spec and its author), the threadpool consist of a submission queue, and arbitrary number of worker threads (as well as supportive features not visible to users).

All futures submitted to the pool will go to the submission queue, and been transported to the queue of any idling worker at random.

Each worker thread will work on the futures in its queue, starting from the top. If the function called by a future spawns more futures (tasks), the tasks will be pushed into the top of the queue of the worker currently executing this future.

If a worker thread has finished all the futures in its queue, it will try request another future from the submission queue; if the submission queue is empty, the worker will try to steal futures from the bottom of other workers' queues.

The threadpool provides the following API:

Structs:

struct thread_pool;

A threadpool struct that contains submissions to be done, worker threads, and status information. Users need to initialize and keep track of their thread_pool structs as they need it to perform most interactions. A thread_pool must be closed after use.

struct future;

A struct that denotes a future, which is similar to a job. It contains a function to be or have been called, parameters for the function, values returned, the threadpool this future belongs to, and other information. Users must keep track of futures created in their submissions to their threadpools in order to retrieve the results.

A future can be in any of the 3 states: unstart, running, or completed.

Functions:

struct thread_pool * thread_pool_new(int nthreads);

Initialize a new thread_pool struct with specified number of worker threads.

*Param*s:

nthreads number of threads to be created

Returns:

The threadpool struct created

Implementations:

The threadpool uses mutex and semaphore to maintain the integrity in any theadpool. After setting up the basic parts, mutex lock is used, and worker threads are created through pthread_create() calling a supportive static function: **void *startWork(void *c)**. The pool mutex is unlocked once all workers are created.

The startWork function makes sure that a worker thread will continue working as long as there is any job remaining anywhere in the threadpool. It follows a straightforward logic: set up, and try to execute any remaining work repetitively.

In the setup part, the worker first stores its thread id as a local-to-thread integer for submit references. It then tried to lock and unlock the mutex of the worker's threadpool: mutex lock will not be available until the previous thread who have the mutex under control currently have unlocked it, and this operation makes sure the threadpool has done creation of worker threads.

In the execution part, as long as the threadpool it belongs to is still running, the worker will try to execute futures following the priority order of: futures in the worker's own queue, futures in the threadpool's submission queue, then any other worker's queue (the priority of worker queue is based on the worker sequence in the threadpool's worker list). Any future found will be transported to this workers' queue, and start running; after the future is done, the future's semaphore count will increase. Threadpool mutex lock/unlock is called through this process for integrity.

If there is no more future remaining, the threadpool's semaphore count will try to decrease

void thread_pool_shutdown_and_destroy(struct thread_pool *);

Savely close a thread_pool and clean up all the subordinates of it; futures in execution will not be guaranteed to finish.

*Param*s:

thread_pool * a pointer to the threadpool to be closed

Returns:

Implementations:

This function first set the threadpool's status to shut-down. Sem_post is then sent to all workers to wake them up. After that, all workers are popped from the threadpool's list and joined with the main thread (the threadpool). The main thread is cleaned and closed at last.

Status set and close individual worker parts are considered critical, and thus mutex locks are adopted for these sections.

struct future * thread_pool_submit(struct thread_pool *pool, fork_join_task_t task, void * data);

Initialize a new thread_pool struct with specified number of worker threads.

*Param*s:

thread_pool * a pointer to the threadpool to which the task is submitted

task the function called by this future

data task function params

Returns:

A future denoting this task for user to keep track of

Implementations:

A future struct is created to contain submitted task. If the task is submitted by an internal worker, the future will go into the top of the worker's queue; otherwise, it is stored at the pool submission queue. A helper function is deployed to check if the future's stored pid corresponds to any worker thread's pid to realize this feature

A huge mutex lock applies to the whole function.

void * future_get(struct future *curFuture);

Get the result of a task represented by a given future

*Param*s:

*curFuture the future of the task

Returns:

A result returned by executing the task.

Implementations:

Once this function is called, if the future has already been executed, the result will be returned directly; if the future is been executing by a worker, the program waits till it has finished running and returns the result; otherwise the main thread will execute the specified future itself and return a result.

As the whole transportation of future is critical, a mutex lock is applied to the pool till the future has finished executing.

void future_free(struct future *curFuture);

Free a specified future.

*Param*s:

*curFuture the future to be freed

Returns:

The threadpool struct created

Implementations:

The future's semaphore is destroyed and then the future is closed.

Usage

For more detailed descriptions of each function, refer to the comments in *threadpool.h* file.

Each time a user program wants to use a threadpool, they should follow the following format:

```
// first, create the threadpool
struct thread_pool * threadpool = thread_pool_new(nthreads);

//submit any task to the threadpool
struct future * future = thread_pool_submit(threadpool,
                          (fork_join_task_t) a_function_to_be_called,
                               &a_set_of_params_for_the_function);

//obtain and free the future
Int result = future_get(future);
future_free(future);

// must shut down the threadpool at last
thread_pool_shutdown_and_destroy(threadpool);
```