Due: Monday, Apr 19, 2010. 11:59pm (no extensions).

What to submit: Upload a single file threadpool.c with your code. We will be using the provided check.py file to test your code.

You should do this exercise on the systemslab or rlogin machines.

Thread Pools and Futures

In this exercise, you are asked to practice the use of mutexes, condition variables, and semaphores by creating a partial implementation of a thread pool and futures. The thread pool and the futures should implement functionality similar to the Executor and Future classes provided in the Java standard libraries by providing the following API:

```c
/* Create a new thread pool with n threads. */
struct thread_pool * thread_pool_new(int nthreads);

/* Shutdown this thread pool. May or may not execute already queued tasks. */
void thread_pool_shutdown(struct thread_pool *);

/* A function pointer representing a ‘callable’ */
typedef void * (* thread_pool_callable_func_t) (void * data);

/* Submit a callable to thread pool and return future. */
* The returned future can be used in future_get() and future_free()
* /
struct future * thread_pool_submit(
    struct thread_pool *,
    thread_pool_callable_func_t callable,
    void * callable_data);

/* Make sure that thread pool has completed executing this callable, */
* then return result. */
void * future_get(struct future *);

/* Deallocate this future. Must be called after future_get() */
void future_free(struct future *);
```

The implement these functions, you will have to define private structures struct future and struct thread_pool in threadpool.c. A future should store a pointer to the function to be called, any data to be passed to that function, as well as the result (when available). You should use a semaphore to communicate whether a future’s result is available.

A thread pool should keep track of a FIFO work queue, implemented as a list of futures. As a simplification, you should implement a singly-linked list. A thread pool instance should keep track of the worker threads it has created. You should use a combination of one mutex and one condition variable in a monitor-like fashion to protect the member fields of a thread pool. This monitor synchronizes between threads adding tasks to the
workqueue and the worker threads removing and completing them. You will also need a flag to denote when the thread pool is shutting down.

**thread_pool_new()**. Create a new thread pool with \( n \) threads. The threads should be started eagerly, i.e., during the call to `thread_pool_new`. Each thread should use `pthread_cond_wait()` to wait for new futures to be enqueued in the thread pool’s work queue. If a future is enqueued, one worker thread should remove it and execute it, store the result and signal the completion of the future.

**thread_pool_submit()**. This function takes a function pointer and a data pointer. It should allocate a new future, initialize its semaphore, enqueue the future at the end of the work queue, and notify one worker thread via `pthread_cond_signal()`.

**future_get()**. This function shall wait for the future’s semaphore to be signaled, then return the future’s result.

**thread_pool_shutdown()**. This function will shut down the thread pool. Already executing futures must complete; queued futures may or may not complete. You should set a flag in the thread pool instance, then use `pthread_cond_broadcast()` to notify all worker threads of the impending shutdown. The calling thread must join all worker threads before returning. Do not use `pthread_cancel()` as this function does not ensure that currently executing futures run to completion.

**future_free()**. Frees the memory for a future instance allocated in `thread_pool_submit()`. This function is called by the client. Do not call it in your thread pool implementation.

**Requirements.**

- Start by making a copy of the `~cs3214/public_html/spring2010/exercises/thread1/-futures` directory. This directory contains all the files you need.

- You need to create `threadpool.c`. Do not change any of the other files! (If you do, such changes will not be taken into account when grading and you may fail the grading process.)

- Your code must compile without warnings. The Makefile enforces this via `-Werror`.

- You should not define any global or static variables.

- Your code must be thread-safe and should not produce any warnings when run under the Helgrind race condition checker.

The provided script ‘`check.py`’ will check for all of these requirements. We will grade your submission by running `./check.py`. We will award 50% partial credit if you pass all tests but do not pass Helgrind’s checker. A clean and correct solution can be created in less than 130 lines of code.