Due: Wednesday, Nov 16, 2011. 11:59pm. (Late days may be used.)

What to submit: Upload a tar ball using the p5 identifier that includes the following files:
- `threadpool.c` with your code.
- `speedup.pdf` with your analysis. Use a suitable word processing program to produce the PDF file.

We will be using the provided `check.py` file to test your code.

You should do this project on the machines of the rlogin cluster.

1 Thread Pools and Futures

In this project, you will 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 defining the following API:

```
/* 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. You may use the provided list implementation (known to you from projects 3 and 4). 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 the threads adding tasks to the workqueue and the worker threads that are 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, execute it, store the result and signal the completion of the future. You will need to write your own function (which each thread executes) that implements this functionality.

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. This call does not deallocate the future.

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()` because 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.

- Import the directory ~cs3214/threadlab/threadlab-fall2011 into your SVN. This directory contains all the files you need. Use of SVN is required.
- 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 you 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.

- You should not define any global functions other than the ones asked for - use static functions as necessary.
- Your implementation must not use busy waiting or otherwise consume excessive amounts of CPU time.
- 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. A clean and correct solution can be created in less than 130 lines of code.

Minimum Requirements. The minimum requirements for this project include a working thread pool implementation, as signaled by check.py prior to running the Helgrind checker. Passing the Helgrind checker is not required to meet minimum requirements.

2 Observing Parallel Speedup

Many divide-and-conquer algorithms are easily parallelized. An example of such an algorithm is quicksort: after pivot selection and partitioning, the two recursive calls can be executed in parallel by submitting them as tasks to a thread pool and waiting for the futures obtained in return. If the partitions are too small, however, it is better to apply a conventional, sequential quicksort implementation.

If your implementation of futures is race condition free and passes helgrind, the test script will link it with a parallel implementation of quicksort and test the resulting program. You should study the quicksort example to see how thread pools and futures can be used in an actual implementation.

For this project, you are asked to study parallel speedup. Ideal parallel speedup would decrease a program's execution time by a factor of N when run on N processors. Actual achievable speedup is less.

Run quicksort for $300,000,000$ integers, fixing the seed for the random number generator at 42. Consider (at least) 1, 2, 4, 6, 8, 12, and 16 threads. For each number of threads, determine (using trial-and-error) the parallel recursion depth that yields the greatest speedup. For instance, for 8 threads, using a parallel recursion depth of 7, you would be issuing this command: `./quicksort -s 42 -d 7 -n 8 300000000`
After determining the optimal parallel recursion depth (separately for each number of threads), run each experiment three times and report the average. Plot your results, showing the number of processors on the x-axis and the obtained (average) speedup factor on the y-axis. Does quicksort scale well using this implementation on the multicore machines of our rlogin cluster? Include a discussion of your results.

Perform these experiments on an unloaded machine on the rlogin cluster. Unloaded means that ’uptime’ should report a load average close to 0, so that all processors are available for your experiment. Coordinate with other students by avoiding running your benchmarks if you notice that other students are running theirs; use the forum or email if necessary.

3 Extra Credit: A Faster Parallel Sort.

Create a parallel sort algorithm, with or without your thread pool, that beats your speedup numbers obtained in the previous section.