Due: Wednesday, March 25, 2015. 11:59pm.(Late days may be used.)

What to submit: Upload a tar ball using the p2 identifier that includes the following files:
- `id.txt` with SLO IDs in the format described for Project 1.
- `threadpool.c` with your code.
- `threadpool.pdf` with your project description. Use a suitable word processing program to produce the PDF file.
We will be using the provided `fjdriver.py` file to test your code.

We will use the machines of the rlogin cluster for testing and benchmarking, so make sure your code runs there.

1 Background

In 2001, Intel VP Patrick Gelsinger [3] warned in a now famous keynote given at the ISSCC 2001 conference that if processor power consumption trends for the Intel x86 line of processors continued at their then-existing trajectory, chip surface temperatures would reach the power density of a nuclear reactor by 2005, a rocket nozzle by 2010, and the surface of the sun by 2015. The answer was soon found in the move from single core to multiple core chips, opening the multicore era. Analysts soon projected that the number of cores per chips would soon double at a pace akin to Moore's law.

That change needed to be accompanied by a complete rethinking of how to write software. Researchers at Berkeley called the switch to parallel microprocessors nothing less than a "milestone" in the history of computing [1], and called for new human-centric programming models that make it easy to write scalable programs for the emerging multicore systems.

Fast forward 13 years to present day. The guardians of C++, after much deliberations, finally introduced support for multithreading in their language through the use of a `std::async` function[1]. The reference documentation on cppreference.com provides the example shown in Figure 1.

This toy example sums up the elements of a vector (all initialized to be 1) using a recursive divide-and-conquer approach. At each level of recursion, the array is subdivided into two equal parts, one of which is passed to `std::async` to be executed in a separate thread, the other part is recursively performed by the calling thread. `std::async` returns a handle of type `std::future`, which represents a reference to the result of a computation that is executed asynchronously. When the result is needed, a thread may invoke the `get()` method. `get()` will return the result, arranging for — or waiting for — its computation if necessary.

Compiling and running this program under g++ 4.7.2[2] one obtains the following output:

[1] You will not need to learn C++ for this project, I am just using it as a motivating example

[2] What about clang, George may ask. clang 3.4, which is the default clang on my brand new Ubuntu

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
#include <future>

template <typename RAIter>
int parallel_sum(RAIter beg, RAIter end)
{
    auto len = std::distance(beg, end);
    if(len < 1000)
        return std::accumulate(beg, end, 0);

    RAIter mid = beg + len/2;
    auto handle = std::async(std::launch::async,
                             parallel_sum<RAIter>, mid, end);
    int sum = parallel_sum(beg, mid);
    return sum + handle.get();
}

int main()
{
    std::vector<int> v(100000000, 1);
    std::cout << "The sum is " << parallel_sum(v.begin(), v.end())
              << '\n';
}
```

Figure 1: A parallel sum implementation in C++11. This is a slightly modified version of the example published at http://en.cppreference.com/w/cpp/thread/async. Instead of 10,000, this program is summing up a vector with 100,000,000 elements.

```
$ scl enable devtoolset-1.1 bash
$ g++ -O2 -std=c++0x cppasyncpsum.cc -o cppasyncpsum -pthread
$ ./cppasyncpsum
terminate called after throwing an instance of 'std::system_error'
  what():
```

Running it under strace, one can observe the `clone()` system call failing with `EGAIN`, *Resource temporarily unavailable*. Apparently, C++11's std::async is implemented by blindly spawning kernel-level threads (roughly 10^5 of them), without any regard to resource use.

This is where you come in.

In this project, you will create a framework that allows the parallel execution of divide-and-conquer algorithms such as the one shown in this example. To do that, you will create a thread pool implementation for dynamic task parallelism, which can execute so-called fork/join tasks. Your implementation should avoid excessive resource use to avoid

14.04 installation, will not even compile the example because it does not agree that the program constitutes valid C++ [URL]

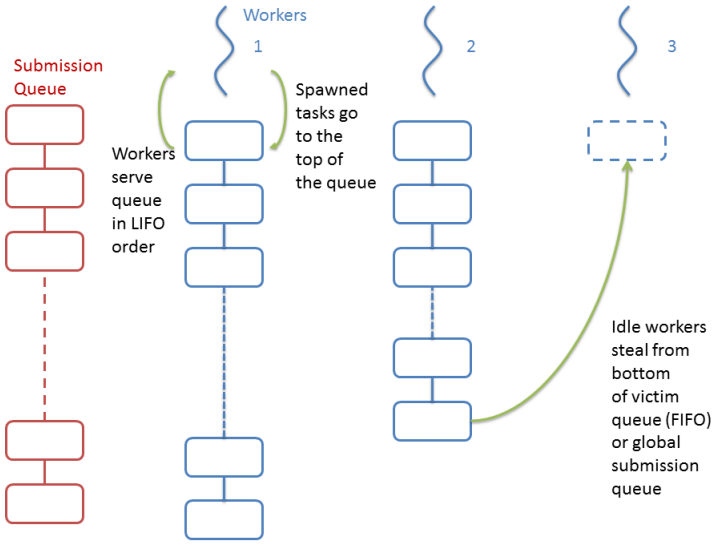

Figure 2: A work stealing thread pool. Worker threads execute tasks from their own deques in LIFO order. If they run out of work, they attempt to dequeue tasks from a global submission queue. Failing that, they attempt to steal tasks from the bottom of other workers queues.

crashes like the one shown above.

2 Thread Pools and Futures

Your thread pool should implement the following API:

```
/**
 * threadpool.h
 *
 * A work-stealing, fork-join thread pool.
 */

/*
 * Opaque forward declarations. The actual definitions of these
 * types will be local to your threadpool.c implementation.
 */
struct thread_pool;
struct future;

/* Create a new thread pool with no more than n threads. */
struct thread_pool * thread_pool_new(int nthreads);

/*
 * Shutdown this thread pool in an orderly fashion.
 * Tasks that have been submitted but not executed may or
 * may not be executed.
```

```
 *
 * Deallocate the thread pool object before returning.
 */
void thread_pool_shutdown_and_destroy(struct thread_pool *);

/* A function pointer representing a 'fork/join' task.
 * Tasks are represented as a function pointer to a
 * function.
 * 'pool' - the thread pool instance in which this task
 *          executes
 * 'data' - a pointer to the data provided in thread_pool_submit
 *
 * Returns the result of its computation.
 */
typedef void * (* fork_join_task_t) (struct thread_pool *pool, void * data);

/*
 * Submit a fork join task to the thread pool and return a
 * future.  The returned future can be used in future_get()
 * to obtain the result.
 * 'pool' - the pool to which to submit
 * 'task' - the task to be submitted.
 * 'data' - data to be passed to the task's function
 *
 * Returns a future representing this computation.
 */
struct future * thread_pool_submit(
        struct thread_pool *pool,
        fork_join_task_t task,
        void * data);

/* Make sure that the thread pool has completed the execution
 * of the fork join task this future represents.
 *
 * Returns the value returned by this task.
 */
void * future_get(struct future *);

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

2.1 Work Stealing

The are at least two common ways in which multiple threads can share the execution of dynamically created tasks: work sharing and work stealing. In a work sharing approach, tasks are submitted to a central queue from which all threads remove tasks. The drawback of this approach is that this central queue can quickly become a point of contention, particularly for applications that create many small tasks.

Instead, a work stealing approach is recommended [2] which has been shown to lead to

better load balancing and lower synchronization requirements. In a work stealing pool, each worker thread maintains its own local queue of tasks, as shown in Figure 1. Each queue is a double-ended queue (deque) which allows insertion and removal from both the top and the bottom. When a task spawns a new task, it is added to the top. Workers execute tasks by removing them from the top, thus in LIFO order. If a worker runs out of tasks, it checks a global submission queue for tasks. If a task can be found in it, it is executed. Otherwise, the worker attempts to steal tasks to work on from the bottom of other threads' queues.

2.2 Helping

A naive attempt to implement `future_get` would have the calling thread block if the task associated with that future has not been computed. However, doing so risks thread starvation: it is easily possible for all worker threads to be blocked on futures, leading to a deadlock because no worker threads are available to compute the tasks on which the workers are blocked!

Instead, worker threads that attempt to resolve a future that has not yet computed must help in their execution. For instance, if the future's task has not yet started executing, the worker could steal it and execute it itself. If it has started executing, the worker has a number of choices: it could wait for it to finish, or it could help executing tasks spawned by the task being joined, hoping to speed up its completion.

However, helping must be done carefully: worker threads should not attempt to help by stealing tasks who, in order to complete, rely on the results of tasks whose execution has already been started by the current worker. This can happen if the current worker thread started a task but has not yet completed it because it may be in the middle of joining a task it has spawned. To avoid that, you could adopt a technique such as leap frogging [4], which keeps track of the depth of each task in the computation graph and provides a rule that allows or disallow stealing.

For the purposes of this assignment, we assume a fully-strict model. A fully-strict model requires that tasks join tasks they spawn (in order words, every call to submit a task has a matching call to `future_get()` within the same function invocation.) All our tests will be fully strict computations.

2.3 Implementation

Except for constraints imposed by the API and outside resource availability, you have near absolute freedom in how to implement your thread pool. Numerous strategies for stealing, helping, blocking, and signaling are possible, each with different trade-offs.

You will need to design a synchronization strategy to protect the data structures you use, such as flags representing the execution state of each tasks, the local queues, and the

global submission queue, and others. You will need a signaling strategy to achieve that worker threads learn about the availability of tasks in the global queue or in other threads' queues.

2.4 Basic Strategy

A basic strategy would be to use locks, condition variables, and the provided list implementation (known to you from Project 1), which allows constant-time insertion and removal of list elements.

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 will have to define appropriate variables to record the state of a future, such as whether its execution has started, is in progress, or has completed, as well as possibly which queues the future is in to keep track of stealing.

A thread pool should keep track of a global submission queue, as well as of the worker threads it has started. Each worker thread requires its own queue. You will also need a flag to denote when the thread pool is shutting down.

thread_pool_submit(). You should allocate a new future in this function and submit it to the pool. Since the same API is used for external submissions (from threads that are not part of the pool) and internal submissions (from threads that are part of the pool), you will need to use a thread-local variable to distinguish those cases. The thread local variable could be used to quickly look up the worker thread for internal submissions.

future_get(). The calling thread may have to help in completing the future being joined, as described in Section 2.2.

thread_pool_shutdown_and_destroy(). This function will shut down the thread pool. Already executing futures should complete; queued futures may or may not complete.

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; instead, use a flag and an appropriate signaling strategy.

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.

3 Additional Notes

3.1 Grading

Grading will be based on a combination of factors, including

- Correctness. We expect your code to produce the correct result. In particular, it should complete test programs when we restrict the number of worker threads to be 1.
- Thread Safety. Your code must not contain race conditions. You should run the code using the Helgrind race condition checker. If Helgrind flags any warnings, you should address them. If you believe Helgrind's warnings are spurious because you are making use of advanced synchronization facilities that trigger false positives, provide an rigorous proof.
- Speedup. For each of the benchmarks we provide, we will measure the speedup obtained using your thread pool.
- Resource Use. Outside of dedicated high-performance computing (HPC) fork-join implementations are required to coexist with other code and share computational resources. We will measure the amount of CPU time used by your implementation; the lower, the better.

3.2 Honor Code

As usual, all work submitted must be yours. You may not reuse code from any implementations you may find online without the instructor's permission (and the permission of the author, if applicable). If in doubt, you must ask. Otherwise, the collaboration policy described in the syllabus applies.

3.3 Minimum Requirements.

The minimum requirements for this project include a working thread pool implementation that can execute a specific set of parallel programs correctly. `fjdriver.py` will flag that you have met minimum requirements.

3.4 Running Experiments

Please run the `fjdriver.py` script on the rlogin machine. You can find hardware specifications for the cluster machines at `http://rlogin.cs.vt.edu/more.html`.

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.

3.5 Additional Requirements.

- Fork the directory https://git.cs.vt.edu/cs3214/threadlab-spring-2015 into your GIT account. This directory contains all the files you need. Use of version control, e.g., GIT, 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.

The submission check script may impose additional requirements to simplify automatic grading. Please work with teaching staff on any questions you encounter.

Good Luck!

References

[1] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[2] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999.

[3] P.P. Gelsinger. Microprocessors for the new millennium: Challenges, opportunities, and new frontiers. In *Solid-State Circuits Conference, 2001. Digest of Technical Papers. ISSCC. 2001 IEEE International*, pages 22–25, Feb 2001.

[4] David B. Wagner and Bradley G. Calder. Leapfrogging: A portable technique for implementing efficient futures. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '93, pages 208–217, New York, NY, USA, 1993. ACM.