Due: Friday, Oct 23, 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. Please see Section 3.3 for more information.

1 Background

In 2001, Intel VP Patrick Gelsinger [5] 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. As a response, manufacturers soon started packing CPU cores on chips, resulting in lower power consumption and heat dissipation. This trend started what we now refer to as the multicore era. Analysts projected that the number of cores per chips would soon double at a pace akin to Moore's law.

This architectural 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 to present day. The guardians of C++, after much deliberation, 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, which here are initialized to 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, whereas the other part is recursively performed by the calling thread. `std::async` returns a handle of type `std::future`, which represents a reference to a result of a computation that is executed asynchronously. When the computation's result is needed, a thread may invoke the future's `get()` method. `get()` will return the result, arranging for — or waiting for — its computation as necessary.

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

```
#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.

Compiling and running this program under g++ 4.8.2 one obtains the following output:

```
$ scl enable devtoolset-2 bash
$ g++ -O2 -std=c++0x cppasyncpsum.cc -o cppasyncpsum -pthread
$ ./cppasyncpsum
terminate called after throwing an instance of 'std::system_error'
  what():
```

When running this program under strace one can observe the `clone()` system call failing with `EGAIN`, *Resource temporarily unavailable*. The reason for this failure is that C++11's `std::async` is implemented by blindly spawning kernel-level threads (roughly 10^5 of them), without any regard to the amount of resources used by those threads. The same is true when using threads in Java (via `java.langThread`), making the use of threads difficult for all but the most simple scenarios.

This is where you come in.

In this project, you will create a fork/join framework that allows the parallel execution of divide-and-conquer algorithms such as the one shown in the example in Figure 1 in

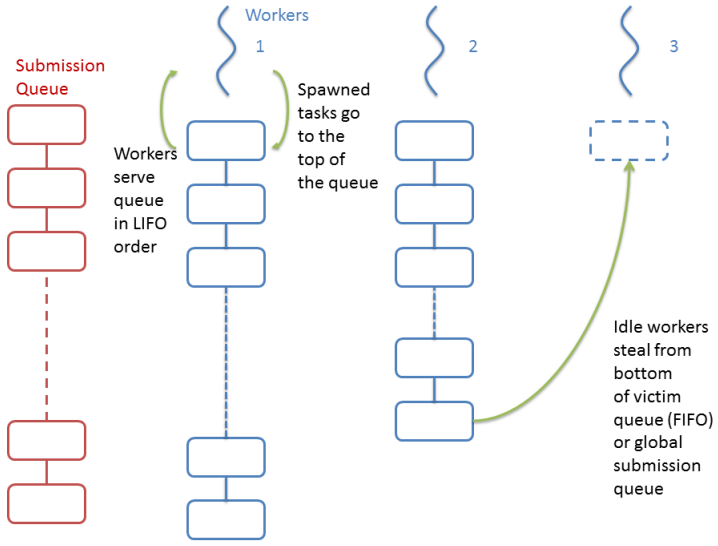

Figure 2: A work stealing thread pool. Worker threads execute tasks from their own deques in LIFO order. If they run out of work, they attempt to dequeue tasks from a global submission queue. Failing that, they attempt to steal tasks from the bottom of other workers queues.

a resource-efficient manner. To that end, you will create a thread pool implementation for dynamic task parallelism, focusing on the execution of so-called fork/join tasks. Your implementation should avoid excessive resource use to avoid crashes like the one seen in this example.

2 Thread Pools and Futures

Your fork-join 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

There 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 draw-

back 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 2. 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 following a 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 at implementing `future_get` would have the calling thread block if the task associated with that future has not yet been computed. "Blocking" here means to wait on a synchronization device such as a semaphore until it is signaled by the thread computing the future. However, this approach 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 been computed must help in their execution. If the future's task has not yet started executing, the worker should steal it and execute it itself. If it has started executing, the worker has two 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.

For the purposes of this assignment, we assume a fully-strict model as defined in [2]. A fully-strict model requires that tasks join tasks they spawn — in other 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, which encompass a wide range of parallel computations.

Restricting ourselves to fully-strict computation for this project simplifies helping because it is always safe for workers intending to help to steal any task as long as they steal from the bottom of any other worker's queue. Safety here refers to the absence of execution deadlock.

2.3 Implementation

Except for constraints imposed by the API and resource availability, you have complete 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 possibly 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.

2.5 Advanced Strategies/Extra Credit

Real-world fork/join implementations employ a number of optimizations designed to minimize per-task synchronization overhead. For instance, a crucial optimization is to

speed up the common case of adding an element to or removing it from the top of a worker's queue. The THE protocol based on Djikstra's mutual exclusion algorithm [3] may be used, which is further described in [4] and [6].

A second possible extension would be to support computations that are not fully-strict (although the computational graph would still need to be assumed to by acyclic - if the computation has inherently cyclic dependencies, it is impossible to schedule it.) If the computation is not fully-strict, that is, if futures could be passed among tasks, a deadlock situation could arise where one worker steals a task that, in order to complete, requires the results of a task whose execution has been started by the stealing thread, but not yet finished. Systems such as CILK [4] avoid this by using a technique known as continuation stealing [7] in which it is possible for other worker threads to continue (and complete) a spawning task. However, continuation stealing requires compiler support since another thread would need access to the task's local variables. Systems that exploit child stealing, such as the thread pool you are building in this assignment, have to impose constraints on stealing for non-strict computations.[2] A technique such as leap frogging [8] could be used, which keeps track of the depth of each task in the computation graph and provides a rule that allows or disallow stealing.

If you implement any of these strategies, be sure to discuss it in your project description so that TAs may award extra credit if warranted.

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. Since you are writing a concurrent program, the results may vary between runs if your implementation is incorrect; we will run your code multiple times and expect it to complete correctly each time we run it. You should perform similar stress testing.

  We also expect your code to be correct when we restrict the number of threads in the pool to be 1, which requires a correct implementation of helping.

- 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 a rigorous proof.

- Speedup. For each of the benchmarks we provide, we will measure the speedup obtained using your thread pool. The `fjdriver.py` script will compile your thread

[2] Read Robison's Primer [URL] to learn more about child vs continuation stealing.

pool, link it with our tests, and benchmark it. It will then prepare a file you may upload to the scoreboard (via `fjpostresults.py`) to compare your results to those of others.[3]

The scoreboards are unofficial in that your final grade will be determined when the TAs check and benchmark your code. However, we will use the scoreboard as a yardstick to determine high-performing and low-performing implementations. In particular, if you see that for a particular test some implementations provide speedup that is a multiple of what your implementation provides, you may conclude that your implementation may impose unnecessary serialization or have other bottleneck factors you should try to address.

For grading, we will award credit for

- **Meeting Minimum Requirements,** which for this project include a working thread pool implementation that can execute a specific set of parallel programs correctly. `fjdriver.py` will flag whether you have met minimum requirements, but keep in mind that during grading, we will run the required tests multiple times and expect them to pass every time.
- **Robustness,** as measured by the ability to successfully and reliably complete a number of more complex applications within a test-specific timeout.
- **Performance,** as measured by the speedup obtained for more complex applications/tests.

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 Running Experiments

We will use the machines of the rlogin cluster for testing, so make sure your code runs there when invoking `fjdriver.py`. For the final scalability testing, we will use 2 AMD machines with 64 cores each (machine names are fir.rlogin and sourwood.rlogin). Please do not use those two machines until after you reliably pass all tests; they should be reserved for optimizing the scalability of your implementation only.

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

[3] We will have two scoreboards, one for regular 20-core rlogin machines, and one for the 64-core machines. See Section 3.3.

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. `fjdriver.py` will output a message and wait if run on a machine with a non-zero load average.

3.4 Additional Requirements.

- The use of git is required as in project 1.
- The upstream repository is `https://git.cs.vt.edu/cs3214-staff/threadlab`
- After forking the repository, be sure to set access to private. Not doing so is a potential honor code violation.
- All code for this project must be contained in `threadpool.c`, mainly to simplify testing. We also believe that the complexity of this assignment, at least in its basic form, should not necessitate the use of multiple source files.
- 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.
- Updates to these requirements may be posted on the website or the Forum forum (in a 'pinned' post at the top of the Forum board).

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] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569–, September 1965.

[4] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 212–223, 1998.

[5] 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.

[6] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[7] Arch Robison. A primer on scheduling fork-join parallelism with work stealing, 2014. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3872.pdf

[8] 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.