Prof. Jingke Li (FAB 120-06, li@cs.pdx.edu); Classes: TTh 10:00-11:50 @ FAB 40-06; Office Hours: TTh 12:00-13:00.

Programming Project 1
(Due Thursday, 4/26/12)

This project is to practice multi-threaded programming with the use of the Pthread library. Your task is to write a Pthread quicksort program based on a given sequential version. For this project, the requirements are the same for all students.

The Sequential Version

A sequential quicksort program, `qsort_seq.c`, is provided to you as a reference. The program has a simple interface:

```
% qsort_seq N
```

where `N` is an integer representing the size of the integer array to be sorted. The program starts off with command-line processing to get the value of `N`. It then allocates an array of size `N`, with randomly permuted values from 1 through `N`. After that, it follows the standard quicksort algorithm:

```
void quicksort(int low, int high)
{
  if (high - low < MIN_SIZE) {
    bubble_sort(low, high);
    return;
  }
  int middle = partition(low, high);
  quicksort(low, middle-1);
  quicksort(middle+1, high);
}
```

It partitions the array (or array segment) into two smaller segments, and recurses down on them. When a segment is smaller than the pre-defined threshold value (`MIN_SIZE`, currently set at 10), it switches to use bubble-sort to finish the sorting.

The Parallel Version

The Pthread parallel program you are going to write is task-queue based, and should be called `qsort_queue.c`. It has the following interface:

```
% qsort_queue N [num_threads]
```

It reads in one or two command-line arguments. The first argument `N` still represents the array size, while the second (optional) argument represents the number of threads to use. When the second argument is omitted, the program uses the default of one thread. The following is a sketch of the structure of the program:

```
// A global array of size N contains the integers to be sorted.
// A global task queue is initialized with the given sort range [0,N-1].

main()
{
  read in command-line arguments, N and num_threads;
  create (num_threads-1) threads, each will execute the worker() routine;
  initialize the array and the task queue;
  execute worker();               // 'main' is also a member of the thread pool
  wait for other threads to join;
  verify results;
}
```

```
// Every thread executes this routine, including 'main'
worker()
{
  while (<termination condition> is not met) {
    [low,high] <- queue.delete();  // remove a task from the queue
    quicksort(low, high);
  }
}
```

The `quicksort()` routine is similar to that of the sequential program, except that instead of recurses on the two smaller array segments, it places the first segment unto the task queue, and recurses only on the second one.

For this project, you are *required* to follow the outline given here, and use as much existing code from the sequential program as possible.

Implementation Details

There are three areas of details, all very important for this project. Spend time thinking carefully what your design should be for each case, before starting the actual coding.

- **The Task Queue** — The task queue can be implemented as a linked list, with two pointers, `queue_head` and `queue_tail`. Tasks are added at the tail and removed from the head. Each task node should contain a pair of integers, representing the lower and upper bounds of an array segment. Define a C `struct` for this purpose.
- **Synchronization** — Since the task queue is a shared resource, synchronization is needed. It is needed for two reasons: (1) Adding and removing tasks from the queue needs to be serialized to avoid race conditions; (2) When the queue is (temporarily) empty, a waiting/waking-up mechanism needs to be used.
- **Termination Condition** — By default, each thread will keep looking for new tasks to work on. How should the program terminate? The task queue being empty is a necessary condition, but not sufficient, since new tasks may still be added to it. One possible solution is to keep track of how many array elements have been sorted. Use a global counter for this purpose. Every time an array segment is sorted, update the counter. Once all `N` elements are sorted, we know that the program can terminate. Implementing a termination condition can be a little tricky, especially if there are other waiting/waking-up synchronizations in the program.

Program Development

You need to develop and run this program on a linux/unix platform using the `gcc` compiler. The following are your options:

- Use one of the CS Linux Lab machines. You can either go to the lab, or access them remotely through `linuxlab.cs.pdx.edu`. To compile Pthread programs on these machines, you need to include the `-pthread` flag with `gcc`.
- Use one of the CS Unix (Sun Solaris) machines. You can only access these machines remotely, through `unix.cs.pdx.edu`. A separate machine, `cepheus.cs.pdx.edu`, is also available. This is an 8-core Sun server (which supports 64 hardware threads) dedicated to this course. On these unix machines, the Pthread library is included in `gcc` by default, so no need to use any flag.
- Use your own PC. If you have a Linux PC or have a linux environment (*e.g.* `cygwin`) installed on your PC, then you can also compile and run your program on your own PC.

Once you have your program compiled and running, you need to verify that the sorting result is correct. There is a block of code for this purpose at the end of the provided sequential program. You should copy this block of code to your program.

In addition to result correctness, you also need to verify that the program really is using multiple threads to do the sorting work. The following scheme can be used for this purpose: Include a `rank` parameter to the worker routine (0≤`rank`<`num_threads`) and add some debugging statements to the routine.

```
    // Every thread executes this routine, including 'main'
    worker(long rank)
    {
#ifdef DEBUG
      printf("Thread %ld starts ...\n", rank);
#endif
      while (<termination condition> is not met) {
        [low,high] <- queue.delete();  // remove a task from the queue
        quicksort(low, high);
#ifdef DEBUG
        printf("%ld.", rank);
#endif
      }
    }
```

When compiled with the flag `-DDEBUG`, a debug version can be built. The following is a sample output:

```
% ./qsort_queue 100 3
Quicksort with 3 threads (queue version), array_size = 100
Thread 1 starts ...
Thread 2 starts ...
Thread 0 starts ...
1.2.0.1.2.0.1.2.0.1.2.0.1.2.0.
...
Sorting result verified (cnt=100)!
```

It shows that all three threads indeed ran (and in an interleaved pattern).

Project Requirements

This project will be graded on the following criteria:

- **Correct Result** — Every of your program's run should end with the message `"Sorting result verified (cnt=...)!"`.
- **Correct Multi-Threading** — Test your program with various parameter combinations. Save an execution log of your program, with `N` taking values of 1000, 10000, and 100000; and `num_threads` taking values of 1, 2, 4, and 8. For the multi-threaded runs, your log should show threads interleaving.
- **Free of Synchronization Bugs** — Debug your program to make sure that it always terminates properly. If you observe threads hanging in some runs, you need to debug your synchronization design.
- **Reporting** — Write a one-page project summary. Cover the following aspects:
  - the synchronization design
  - the termination condition design
  - lessons you've learned

What to Turn In

Make a `zip` or `tar` file containing your source program, the log file, and the summary. Use the Dropbox on the D2L site to submit.