Prof. Jingke Li (FAB 120-06, lij@pdx.edu); Class: TTh 10:00-11:20 @ UTS 205; Lab: W 12:30-13:50, 14:00-15:20.

Assignment 1: Programming with Pthreads

(Due Friday, 2/3/17)

This assignment is to practice multi-threaded programming using the Pthreads library. You'll develop a Pthreads program, compile and run it on the CS Linux system (`linuxlab.cs.pdx.edu`). For this assignment, CS415 and CS515 students will work on two different programs. This assignment carries a total of 10 points.

Download the file `assign1.zip` to your CS Linux account and unzip it. You'll see several program files: `task-queue.h`, `task-queue.c`, and `qsort.c`, plus a set of supporting files: a `Makefile`; two run scripts `run1` and `run2`, and two sample outputs `sample1.txt` and `sample2.txt`.

Task Queue Representation and Supporting Routines

Your program will be task queue based. The task queue representation is shown below:

```
typedef struct task_ task_t;
struct task_ {
  int low;        // value 1
  int high;       // value 2
  task_t *next;   // pointer to next task
};

struct queue_ {
  task_t *head;   // head pointer
  task_t *tail;   // tail pointer
  int limit;      // queue capacity (0 means unlimited)
  int length;     // number of tasks in queue
};
```

Each task holds a pair of integer values and a pointer to next task. A queue has two pointers and two integer fields.

There are four task queue routines:

```
extern task_t *create_task(int low, int high);
extern queue_t *init_queue(int limit);
extern int add_task(queue_t *queue, task_t *task);
extern task_t *remove_task(queue_t *queue);
```

The task queue implementation is available in the two included files, `task-queue.h` and `task-queue.c`. To use the code in your program, add an include line:

```
#include "task-queue.h"
```

[CS415 Students] Producer-Consumer Program

Implement a producer-consumer program; name it `prodcons-pthd.c`. You may use code samples from this week's lecture as a starting template. The program should have the following features:

- It takes an *optional* command-line argument, `numCons`, which represents the number of consumer threads. If this argument is not provided, the program defaults its value to 1.

```
linux> ./prodcons-pthd 10    // 10 consumer threads
linux> ./prodcons-pthd       // 1  consumer thread
```

- The task queue is set to have a fixed capacity of 20 tasks.
- The producer generates 100 total tasks. Each task is identified by an unique integer `i`, whose value is between 1 and 100. The value is stored in both the `low` and `high` fields of a task record.
- The consumer threads compete to remove tasks off the task queue, one at a time. Each consumer thread keeps track of how many tasks it has successfully obtained.
- The program terminates properly after all tasks are done.

Thread Termination

A challenging part of this program is the handling of the termination of consumer threads. The following are two possible approaches:

- The producer creates a bogus "termination" task and add `numCons` copies of it to the global queue. Upon receiving such a task, a consumer thread will termination itself by breaking out its infinite loop.
- Use a global count to keep track of the completed tasks. The consumer threads all participate in updating and monitoring the global count. When the count reaches the total number of tasks, all threads terminate.

You may use either of these approaches, or a totally different approach of your own.

0.0.1 Output Requirements

- Each producer/consumer thread should print out a starting message and an ending message, at the beginning and at the end of its execution, respectively:

```
Producer starting on core 3
Consumer[1] starting on core 5
Consumer[2] starting on core 2
...
Producer ending
Comsumer[2] ending
Comsumer[1] ending
...
```

  The starting message should contain the id of the CPU core it is running on. (There is no requirement on where threads should run, *i.e.* no need to control CPU affinity.) The consumer threads should include their own id in the messages.

- The program should print out a final message showing the distribution of tasks over threads, and the total number of tasks computed from the distribution data:

```
Task count across threads:
C[ 0]:14, C[ 1]:19, C[ 2]: 9, C[ 3]:21,
C[ 4]: 6, C[ 5]: 9, C[ 6]: 5, C[ 7]:17,
Total: 100
```

  The numbers inside the brackets are comsumer ids, and the numbers outside are the number of tasks each consumer thread has obtained. The total is the sum of the individual counts. (It should match the input parameter set at the beginning of the program.) The file `sample1.txt` contains sample outputs for this program.

[CS515 Students] Quicksort Program

Your task is to convert a sequential quicksort program to a parallel Pthreads program using a task queue. The provided sequential program, `qsort.c`, has a simple interface:

```
linux> ./qsort <N>
```

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

```c
void quicksort(int *array, int low, int high)
{
  if (high - low < MINSIZE) {
    bubblesort(array, low, high);
    return;
  }
  int middle = partition(array, low, high);
  if (low < middle)
    quicksort(array, low, middle-1);
  if (middle < high)
    quicksort(array, middle+1, high);
}
```

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

The Parallel Version

Name your Pthreads quicksort program `qsort-pthd.c`. It should have the following interface:

```
linux> ./qsort-pthd <N> [<numThreads>]
```

It reads in one or two command-line arguments. The first argument `<N>` 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 one thread (*i.e.* the main thread itself) as the default.

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

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

int main(int argc, char **argv) {
  // read in command-line arguments, N and numThreads;

  // initialize array, queue, and other shared variables

  // create numThreads-1 worker threads, each executes a copy
  // of the worker() routine; each copy has an integer id,
  // ranging from 1 to numThreads-1.
  for (long k = 1; k < numThreads; k++)
    pthread_create(&thread[k-1], NULL, (void*)worker, (void*)k);

  // the main thread also runs a copy of worker(), with the id 0
  worker(0);
  // the main thread waits for worker threads to join back
  for (long k = 1; k < numThreads; k++)
    pthread_join(thread[k-1], NULL);

  // verify the result
  verify_array(array, N);
}

void worker(long wid) {
  while (<termination condition> is not met) {
    task = remove_task();
    quicksort(array, task->low, task->high);
  }
}
```

Implementation Details

- **The Task Queue** — Use the provided task queue implementation. For this program, the queue's capacity is assumed to be unlimited. Hence when initializing the queue, use the value 0 for the limit.
- **The quicksort() Routine** — This routine is similar in structure to the sequential version in `qsort.c`. However, instead of recursing on the two smaller array segments, it places the first segment unto the task queue, and recurses only on the second one.
- **Synchronization** — Since the task queue is a shared resource, synchronization is needed. Specifically, (1) when adding and removing tasks from the queue, the operations need to be serialized to avoid race conditions; (2) when the queue is (temporarily) empty, a waiting/signaling mechanism is needed to coordinate the work.
- **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. Think about the two approaches discussed above in the Producer-Consumer section, and see which one fits this program better.
- **Result Verification** — Like in the sequential version, the `main()` routine should call `verify_array()` at the end to verify the sorting result.
- **Output Requirements** — Read the output requirements in the Producer-Consumer section above, and generate similar outputs. I.e., Have the `worker()` routine print out a starting message and an ending message, and have the program print out a final message showing work distribution and the total. The file `sample2.txt` contains sample outputs for this program.

[All Students]

Summary Report Write a short (one-page) summary (in pdf, MS Words, or plain text) covering your experience with this assignment: What issues did you encounter?, How did you resolve them? What do you think of your program's performance in regard to parallelism and work distribution? etc.

Extra Credits If you are interested in extra challenges, you can work on both programs. For CS415 students, completing both programs will earn you up to 4 extra points. For CS515 students, completing both programs will earn you up to 2 extra points.

Submission Make a zip file containing your program(s), two new sample outputs, and your write-up. Submit it through the Dropbox on the D2L site.