CPSC 213

Lab/Assignment 6

Done outside the labs. There will be some help in labs on Aug 7 and 9.

Due: Friday, Aug 9, 2013, 10:00pm

Objectives

Become familiar with some multithreaded programming using the POSIX `pthreads` library and some synchronization primitives to order operations between these threads.

Multithreaded Programming and Synchronization

The provided C program (`thread_drive.c`) enqueues a number of values in the `comp` array, and then creates a number of threads using `pthread_create`. Recall that `pthread_create` takes a function pointer to the function to execute (in this case `thread_work`) and a pointer to the argument (in this case a structure that contains the thread ID and the total number of threads created).

The `thread_work` function in `thread_work.c` dequeues an element from the array and prints it out. This is an example of self scheduling, where each thread grabs the next element to be processed when it is ready, rather than having a chunk of elements pre-assigned to it for processing by a master thread controller. Currently, however, there is no synchronization between the threads, so the ordering of processing amongst threads is non-deterministic; this can be tested by running the program multiple times and comparing the output. Furthermore, in the absence of synchronization, the same element could be processed more than once.

The goal is to synchronize accesses to the `comp` array, so that, if there are **`m`** threads running, the thread with ID **`i`** will only process the array elements **`m*n+i`**, where **`n = 0, 1, 2,`**..., and only after all previous elements have been processed. For instance, for an array with 10 elements, and 4 threads, each of the threads should process the following elements:

- Thread with ID 0 (`T0`) : 0, 4, 8
- Thread with ID 1 (`T1`) : 1, 5, 9
- Thread with ID 2 (`T2`) : 2, 6
- Thread with ID 3 (`T3`) : 3, 7

Also, `T0` should not start processing element 4 until `T3` has completed processing element 3, which, of course, should not start processing element 3 until `T2` is done with element 2, and so on. Since the number of elements and number of threads are provided as parameters, the program

should support multiple different values (the testing program may use different values from the ones provided for the assignment).

The synchronization between threads can be achieved using any synchronization primitives—feel free to use mutex or other synchronization primitives from the `pthread` library, or build your own locks using the spinlock definition that is provided in the thread_work.c file.

Files

The provided files (included in the lab6_code.zip archive):

- `thread_work.h`
- `thread_work.c`
- `thread_drive.c`
- `Makefile`

Building the Program

The provided files can be built simply by typing `make` from the command line on a Linux-based system and create an executable file `thread_drive` which can be run using `./thread_drive`.

The TAs will briefly discuss make files in the labs this week. If you want to learn more about make, you can check section 2 of ***The Stanford CS Education Library*** which is linked from the Resources page of the course web site.

You can also build the program without using makefiles. To create an executable `thread_drive`, you can run the following command

```
gcc -o thread_drive thread_work.c thread_drive.c -l pthread
```

In this, the `-o thread_drive` option assigns the name `thread_drive` to the executable file and the `-l pthread` option includes the pthread library, which is needed in this case.

Requirements

Modify the `thread_work` function in `thread_work.c`. This function should use thread synchronization to ensure that only the correct thread dequeues the value and prints it out. Synchronization can be achieved either by using synchronization primitives from the `pthread` library.

NOTE: Please do not change the format of the `printf` in `thread_work.c`, or print anything else from the files you hand in.

Deliverables

You must submit

- The assignment cover page filled in for this assignment.
- The modified files `thread_work.c` and `thread_work.h` (even if you did not make changes to the latter.)

Handin Instructions

Each group must submit only ONE copy of the assignment, using the department's handin tool.

The person who will submit the assignment should do the following:

- In your `cs213` directory create a directory directory **lab6**.
- Place all the files you have to submit in the lab6 directory.
- At the Unix command line execute
  > handin cs213 lab6
- Type **handin help** to see the options that are available for the handin tool.

Last Updated: 05/08/2013 10:28 PM