CS450/550 Operating Systems

Project 2b: Concurrent / Multi-thread Programming For Odd-Even Transposition Sorting

Due with a demo on November 8, 2011 (Class Time)

Assignment Overview

For this assignment, you are to design and implement a C/C++ program which sorts an array of integers into ascending order using POSIX threads and the *odd-even transposition sort algorithm.* Please read the specifications and requirements *carefully*.

Assignment Specifications

1) The program will accept the name of a text file as an argument on the command line. The program will read a series of values from the file and store them in an array, display the contents of the array (before sorting), sort the contents of the array using the odd-even transposition sort algorithm, and display the contents of the array (after sorting).

2) Valid data files will contain *N* integer values, where N is ***even***, is more than 0, and is less than 20. The data file will contain one number per line. And, the data file will contain at least one blank line in the end. The program should work even if there are more than one blank line in the end.

3) The program will create **no more than N/2 threads** to perform the steps in the odd-even transposition sort (in case your code needs more than N/2, you have to specify it in the project report explicitly, and, you will lose some credits (but still get something)).

4) All program output will be sent to the standard output stream, and should be appropriately labeled and formatted.

5) The program will perform appropriate error checking with good programming convention.

Assignment Deliverables

The deliverables for this assignment include the following files:

proj02b.c --the source code file(s) for your solution
proj02b.h --the interface file(s) for your solution
makefile --a make file which produces "proj02b"
proj02bReport.doc – project description aside, it should also include the workload percentages between teammates, agreed & signed by all project teammates.

Be sure to use the specified file names, and to submit your files for grading to plama@uccs.edu. You should make sure the program is working in your laptops or ENG 138 machines. Please also have a **hard copy** of all deliverable files on the due time for grading reference.

Read the following useful notes.

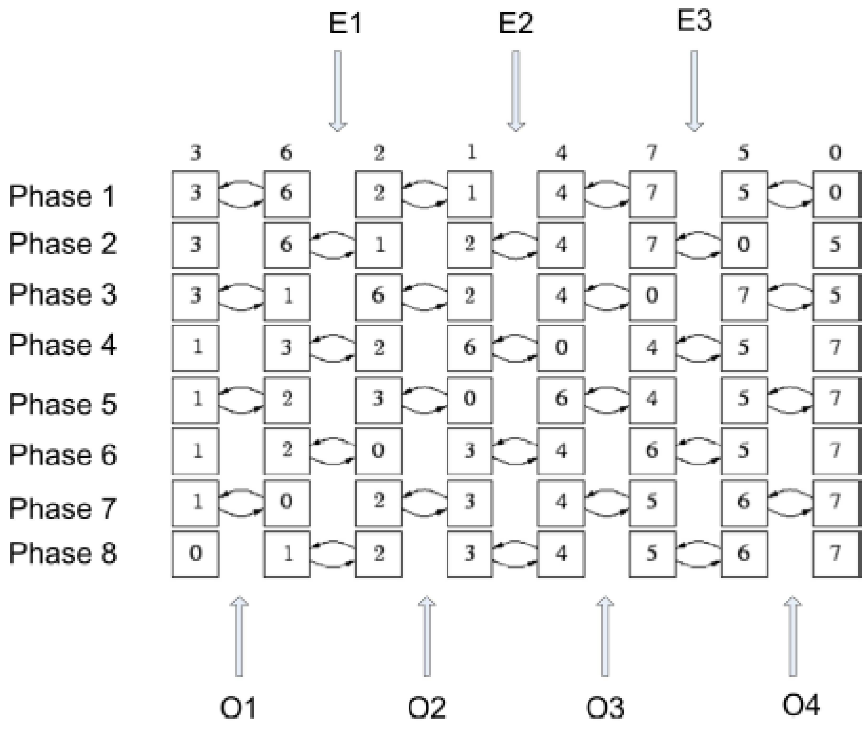
a) Odd-Even Transposition Sort Algorithm

Consider the concurrent odd-even transposition sort of an even-sized array. For 8 elements, 4 threads operate concurrently during the odd phase, and 3 threads operate concurrently during the even phase. Assume that threads, which execute during the odd phase, are named O1, O2, O3, and O4 and threads, which execute during the even phase, are named E1, E2 and E3.

During an odd phase, each odd-phase thread compares and exchanges a pair of numbers. Thread O1 compares the first pair of numbers, thread O2 compares the second pair of numbers, thread O3 compares the third pair, and O4 compares the fourth pair of numbers.

During an even phase, the first and last elements of the array are not processed; thread E1 compares the second and third array elements, thread E2 compares the fourth and fifth array elements, and thread E3 compares the sixth and seventh elements.

The algorithm runs for a total of 8 phases (i.e., equal to the number of values in the data file), as illustrated below.



Thus, as noted above, you will be required to limit the number of threads created for the sort processing. For an even-sized array of N elements, during the entire execution of the algorithm, only N/2 threads will be created. During the odd phases, all N/2 threads will be used. And, during the even phases, (N/2)-1 threads will be used. The program will run for N phases.

b) POSIX Threads and Semaphores

You can refer to http://www.llnl.gov/computing/tutorials/pthreads/ for more information on using POSIX threads. In this project, you might need the following calls from the POSIX library.

1. int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void*(*start_routine, void*), void *arg)

This call is used to create a new thread, with attributes specified by attr , within a process.
If attr is NULL, the default attributes are used. If the attributes specified by attr are
modified later, the thread's attributes are not affected. Upon successful completion,
pthread_create stores the ID of the created thread in the location referenced by thread .
The thread is created executing start_routine with arg as its sole argument.
If the start_routine returns, the effect is as if there was an implicit call to
pthread_exit using the return value of start_routine as the exit status

2. int pthread_join(pthread_t thread, void **status)

This call suspends processing of the calling thread until the target thread completes. thread must be a member of the current process and it cannot be a detached thread.. This function will not block processing of the calling thread if the target thread has already terminated.

3. int pthread_cancel(pthread_t target_thread)

This call requests that target_thread be canceled. You can also refer to http://www.gnu.org/software/libc/manual/html_node/POSIX-Threads.html to get more information on using semaphores (especially, sem_init(), sem_post(), sem_wait(),sem_destroy()) in your program.

3.1 sem_init(sem_t *sem, int pshared, unsigned int val)

This call initializes the semaphore referred by sem with val. If the pshared argument has a non-zero value, then the semaphore is shared between processes; in this case, any process that can access the semaphore sem can use sem for performing sem_wait(),sem_trywait(), sem_post(), and sem_destroy() operations.

3. 2. int sem_post(sem_t *sem)

This call is equivalent to the signal(S) operation discussed in the class. This call unlocks the semaphore referenced by sem by performing a semaphore signal operation on that semaphore. If the semaphore value resulting from this operation is positive, then no threads were blocked waiting for the semaphore to become unlocked; the semaphore value is simply incremented.

3.3. int sem_wait(sem_t *sem)

This call is equivalent to wait(S) operation discussed in the class. This call locks the semaphore referenced by sem by performing a semaphore lock operation on that semaphore. If the semaphore value is currently zero, then the calling thread will not return from the call to sem_wait until it either locks the semaphore or the call is interrupted by a signal.

3.4. int sem_destroy(sem_t *sem)

This call is used to destroy the unnamed semaphore indicated by sem.

c) Sample Files

For illustrations on how to use the POSIX function calls and semaphores, refer to http://www.llnl.gov/computing/tutorials/pthreads/, and the three example files in C on the project Web site.

To compile these files, you need to give "-lpthread" argument to the compiler.

(For example, gcc –lpthread p1.c or g++ -lpthread p1.c)

d) **Useful links**

- **http://www.llnl.gov/computing/tutorials/pthreads/**
    - **Many examples and examples with bugs**
- **http://yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html**
- **GDB for debugging: http://www.cs.cmu.edu/~gilpin/tutorial/#1**
- **DDD for debugging:**
- **http://www.gnu.org/manual/ddd/html_mono/ddd.html**

e) Project2b notes

* **you need to specify your approach in the project report**.

1. When creating a thread using the POSIX thread library, the startup routine of the thread might NOT be a Pthread class member function. If you use a class member function as the startup routine, the compiler may report errors.

2. For an array of size N, **your program should create no more than N/2 threads. Otherwise, specify it explicitly in the project report.**

3. You may try to sort 4 numbers first, then extend it to more general case, say 20 numbers. You will get partial credits if your code can work for 4 numbers sorting.

4. Performance is a very important metric of the coding quality and convention.

5. Your program should execute the odd and even phases of the sorting algorithm inside the threads. Each thread may execute the following:

Implementation Approach 1:

(Use 2N semaphores, one for each data element in Odd phase and even phase)

```
for(j = 1; j <= N; j++) {
   • Wait for semaphores controlling the corresponding data element in the corresponding
   phase (for example, thread 1 will wait for semaphores O1 and O2 in odd phase, E2 and
   E3 in even phase)
   • Do the compare and exchange operation as specified by phase j; in even phases, one
   of the threads (N/2th thread) will do no compare and exchange operation;
   • Signal semaphores controlling the corresponding data element in the subsequent
   phase (for example, thread 1 will signal semaphores E1 and E2 in odd phase, O2 and
   O3 in even phase).
   * Although the N/2th thread in even phase does not perform the compare and
   exchange operation, it has to signal some semaphore for the threads in odd phase to
   proceed. You have to be careful in determining which semaphores they will signal.
}
```

Implementation Approach 2:

(Use 3 semaphores, 1 – for counting the number of threads that has finished executing the phase, 1 for waiting to start odd phase, and the last one for waiting to start even phase)

```
for (j = 1; j <= N; j++) {
    • Do the compare and exchange operation as specified by phase j; in even phases, one
       of the threads (N/2th thread) will do no compare and exchange operation;
    • synchronization to wait for other threads in the phase to complete;
}
```

Implementation Approach 3:

You can have own approach. But no more than use of N/2 threads, and should allow those threads to run at the maximum possible concurrency.

You should try use semaphores. If you do not use semaphores, which is feasible, you may lose 10% of the project value.

5. Whenever a thread finishes executing the compare and exchange operation, it should wait until all the threads in that phase are complete (i.e., synchronization to wait for other threads in the phase to complete). You cannot use pthread_join for this purpose. Instead, you may use semaphores.

6. A thread will finish running ONLY after it has finished all its phases. At this point, it will terminate. Note that you are creating the N/2 threads only once. You should NOT create them in each phase (i**n case you have to do so, you need to specify it in the project report and some points could be lost**).

   After creating the threads, the main program should wait until all the threads terminate. This should be done using pthread_join.