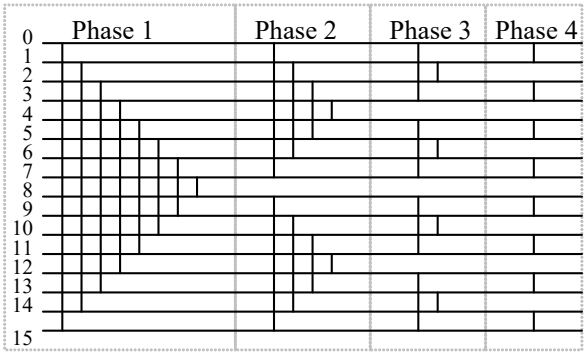
CS 5348 Operating Systems -- Project 2

The goal of this project is to let you get familiar with thread programming. This project is independent of Project 1 and subsequent projects. So, you can use and C++ (or C) and Pthread library or Java and Java Thread to implement this project. If you use Java, you will have to figure out how to use the Thread library by yourself. We will only support C++ and Pthread programming.

1 Overview

You will implement the balanced sorting algorithm. This algorithm requires log N stages, each of which consists of log N phases, to sort N items. For example, sorting 16 integers would require 4 stages, with 4 phases in each stage. During each phase, N/2 compare-exchange operations are made and these operations can be performed in parallel. The compare-exchange operation sorts the two items in ascending order. The following diagram illustrates the operations of the Balanced Sorting algorithm in one complete stage for sorting 16 items.



As shown in the figure, there are 4 phases in each stage. Each phase performs 8 compare-exchange operations (which can be done in parallel). The data pairs to be operated are:

Phase 1:

compare-exchange item 0 and item 15
compare-exchange item 1 and item 14
. . . . . .
compare-exchange item 7 and item 8

Phase 2:

compare-exchange item 0 and item 7
. . . . . .
compare-exchange item 3 and item 4
compare-exchange item 8 and item 15
. . . . . .
compare-exchange item 11 and item 12

. . . . . .

2 Balanced Sort with Parallel Threads

Your need to write a program to implement the balanced sort algorithm. The parallel operations are performed by threads. The outline of the algorithm is as follows.

```
read the number of integers in the input list, N;
while N ≠ 0 do
{   read N integer numbers and store them in an array;
    create and initialize the semaphores necessary for synchronization;
    create N/2 threads to sort the array using balanced sort algorithm;
    wait for all the threads to finish;
    print the array of sorted integers;
    read the number of integers in the input list, N;
}
```

The pseudo code for thread t (0 <= t < N/2) is given in the following. Note that, N is always 2^x for some positive number x.

```
repeat
  for p = 1 to (log N) do   // (log N phases)
    {   num_groups = 2^(p−1);   // you can start from 1, and set num_groups *= 2 after each phase
        group_size = N / num_groups;
        gindex = t / (group_size / 2);
        mindex = t % (group_size / 2);   // mindex: group member index
        group_start = gindex * group_size;
        group_end = (gindex + 1) * group_size – 1;
        data1 = group_start + mindex;
        data2 = group_end – mindex;
        compare-exchange the array items data1 and data2;
        loop for (data1+data2) iterations doing nothing (just to introduce different computation times);
    }
until sorted
```

Each thread performs one compare-exchange operation in each phase. The pairs of data to be sorted in each phase are given in the figure. For example, thread 0 can compare-exchange data items 0 and 15 in the first phase, 0 and 7 in the second phase, 0 and 3 in the third phase, and 0 and 1 in the fourth phase. In your program, there should be no sleep() call or any similar method to introduce a delay. Also, you should create threads only once to sort one list of input data (cannot kill the threads after each phase or after each stage).

3 Synchronization

You need to synchronize the threads at the beginning or the end of each phase in order to sort correctly. This is called the barrier synchronization, i.e., all threads should cross the same barrier at the beginning (or the end) of each phase before continuing. You should **only** use semaphores to achieve the goal. Try to use as few semaphores as possible, but the number of semaphores you use will not impact your grade.

You need to initialize your semaphores properly. However, we would like to have control over your semaphores. So you need to read in your semaphore initialization values from an input file "**sema.init**". In the beginning of "sema.init" should be the number of semaphores M. Following M should be the initialization value of each semaphore you used. In case some of your semaphore initialization values is determined by N, then you can put x in sema.init, where $x = \frac{1}{y} + z, y \geq 2$, if $\frac{N}{y} + z$ is the initialization value

you used. For example, if your initialization value is $N/2$, then you put ½ in sema.init. If your initialization value is $\frac{N}{2}+1$, then you put 1½ in sema.init. If your initialization value is $\frac{N}{2}-1$, then you put $-½$ in sema.init.

In your *DesignDoc* file, you should explain the purpose of each semaphore and what the initialization value should be (listed in the same order as those in sema.init). If you use an array of semaphores, then all semaphore in the array should have the same initialization value. You only need to put one value for the entire array in "sema.init". Correspondingly, in your *DesignDoc* file, you need to discuss clearly about the semaphore array, including the size of the array, whether the size is dependent to N, the purpose of the array, and what the initialization value should be. Note that you need to use a fixed number of semaphores (semaphore array should have only one initialization value in sema.init) so that the initialization input can be fixed.

4 Input and Output

The input lists of integers to be sorted are given in a file, one number in each line. In each list, the first number is the number of integers to be read in (N), and the N lines following that are all the numbers in the list. There may be many lists in the file. The last N will be 0 to indicate the termination of the system. You can assume that there will be no errors in the input file. Also, N will always be 2^x for some integer x, $x > 0$ (or $x = 0$ for termination). Note: the sema.init file has been defined in Section 3.

Your program should be able to print output in two modes, the **observation mode** and the **regular mode**. In the regular mode, you only need to print the initial list before the threads start and the sorted list after all threads terminate. If N >= 8, you should print 8 numbers per line; otherwise, all numbers in one line. In the observation mode, besides the required print out of the regular mode, you also need to print out the list of integers being sorted after each **phase** (same printing rule as above). Also, when each thread passes the barrier (synchronization point) each time, you need to print out a message "Thread t finished stage i phase j", where t is the thread number. In both modes, after finishing sorting and printing for each number list, you need to print out a line "--------------------------------" to separate the sorting printouts for different number lists. Though it is not important, let thread number starts from 0 and both stage number and phase number start from 1.

The input file name and the printing mode should be the command line input to your program (first command line input is the input file name and second is the printing mode). The print mode can be "**-r**" representing the regular mode or "**-o**" representing the observation mode.

5 Divide Project into Phases

In the first phase, you can try to create 2 threads, let each thread print a message. The messages should contain information differentiating the threads by their thread id. Then you can use one semaphore to block the threads, i.e., each thread, after printing the message, can wait on the semaphore. The main thread can sleep for a period of time (like 10 seconds) and come back to signal the semaphore twice to release the two threads. The two threads after being released can print some messages again and finish. The main thread should wait till all threads are done and then terminate. You should see that two messages are printed first, then after 10 seconds, two additional messages are printed. If you have problem with message ordering, you can try to flush the output to make sure the print is done instantaneously. If the ordering is still a problem, then your semaphore may have problem. You can initialize your semaphore to different values and see what happens.

After the first phase, you can start to implement the entire program. Your final program should run correctly on CS platforms.

6 Project Submission

You need to submit your program **before** midnight (11:59pm) of the due date (check the web page for the due date) through the elearning system. Your submission should be a tar file or a zip file including the following

- ✓ All source code files making up your solutions to this assignment.
- ✓ The semaphore initialization file "**sema.init**".
- ✓ The *Makefile* that generates the executable from your source code files. Your executables have to be named "**sort.exe**".
- ✓ The *DesignDoc* file named "**design.doc**" that contains the description of the major features of your program that is not specified in the project specification. For this project, you definitely need to discuss in this document how you use semaphore to achieve the desired synchronization. You also need to explain what is the purpose of each semaphore and its initialization value in the order they appear in "sema.init".