CS 5/7343 Fall 2011

Programming Homework 3

The goal of this homework is to implement methods for deadlock avoidance.

Due : 11/23 (Tue) 11:59pm,. No extensions.

Application – Fine (Buffet) Dining Philosopher's problem

For this problem, our philosophers are now having a meal in a fine dining buffet restaurant (assume one exists). In this restaurant, there are a lot of rules and regulations. For example, you need various tools/utensils to eat different kind of food in a manner that is suitable for the restaurant. (For example, you need a soup spoon for soup, and a clam opener for lobsters, and chopsticks for rice etc.).

To abstract the problem, we assume that there is a total of k types of tools to be used. For type j, the restaurant has t_j copies of type j tools to be offered for the philosophers. Every philosopher, when he/she arrives, he/she knows what are the food they want to eat, and thus how many of each type of tool its need. (For each philosopher, we denote s_j to be the number of tools that a philosopher needs). Notice that for different philosophers, the value of s_i is different.

The restaurant has a set of m tables. When a philosopher arrives at the restaurant, it first informs the restaurant the number of tools he/she needs. However, the restaurant does not immediately give him/her the utensils. Instead, the philosopher will sit down on one of the tables, and then periodically request the tools that it needs. If there is no table available, the philosopher will be put on a waiting queue and will be seated to the first table available.

Once a philosopher is sat, he/she will start making requests for tools for eating at random times. Each time a philosopher can request at most two tools. Every time a request is made, the restaurant will determine whether the tools is available, and whether satisfying the request will potentially lead to a deadlock (using the deadlock avoidance algorithm mentioned in class). As a result, the restaurant may deny the request and tell the philosopher that the request the denied. The philosopher will then wait for a random amount of time before making the next request (may or may not be the same).

Once a philosopher obtained all the tools, he/she will spend some time eating his/her meal and then leaves. Once he/she leaves, the table becomes open and the restaurant can sit another philosopher on that table.

Program Structure

Your program structure should resemble that of program 2. The first thing your program should do is to read an input file (which filename should be provided via the command line). The file format will be as follows:

- The first line contains three positive number, separated by (any number of) spaces
    - The first number is the number of types of tools (k)
    - The second number is the number of tables (m)
    - The third number is the number of philosophers (n)
- The second line contains k numbers, denoting the available tools for each type by the restaurant. (You can assume all numbers are > 0, otherwise you should quit the program).

- Then the following n lines contain information about each philosopher. It contains the following fields, separated by (any number of spaces),
    - The first field is a string (at most 20 characters), contains the name of the philosopher
    - The next field is a number that contains the total number of tools that the philosopher needs (let call this number p)
    - Then there will p numbers, each number (between 0 and k-1, inclusive) denote a tool that the philosopher wants. Notice that the number can repeat (since a philosopher may request multiple copies of the same tool). The numbers should be read in and stored into a list (to represent the order of requests).

A sample input file is as follows:

5 3 6
9 8 7 5 8
John 7 0 1 3 2 0 1 1
Jack 3 4 4 1
Jane 5 3 2 3 0 4
Jenny 6 4 2 1 3 0 4
Jim 2 0 0
Joanna 4 1 2 4 2

The main process will be responsible for assigning tables to each philosopher, including assigning a table to a philosopher that is waiting when the table becomes available. The logic for the main program should looks like the following

Read in the input file
Initialization
Create the threads
Sit the first m philosophers
Signal each thread to start
While (there are philosopher not finished)

When a table becomes available
If there are still philosopher(s) waiting
assign the next waiting philosopher to that table
else
inform that table that no one else is coming in

Clean up if necessary
Output("All done")
Exit

Each thread is a table that will sit one philosopher at a time, the logic of each thread is as follows:

Initialization if necessary
Wait for the signal from the main process to start
Repeat

While not done
Repeat
Check the current time (using time()) [denote it as t]
If t is odd
Call Request() to request the next tool on the list
Else
Call Request() to request the next two tool on the list (if there is only one item on the list, then request that item only)
If (request is granted)
Update the tools that the philosopher got
Else
Move the requested item to the end of the list
Sleep for 1 sec + (t mod 1000) milliseconds
Until the philosopher obtains all the tools
Sleep for 2 seconds
Return all the tools back to the restaurant
Signal the main process that he is done eating and is leaving
Wait until either a new philosopher is seated, or the main process told the thread that there is no more philosopher waiting
Clean up if necessary
exit

There may be some functionality that is not included in the pseudo code, and you need to figure out how to fit those in.

Request() function

For requesting tools, you should implement a Request() function that will take in (at least) the following parameter: the philosopher that request the tools (you can identify him/her either by the philosopher himself/herself or by the table (thread) that he/she is in); and the tool(s) that he/she request (either 1 or 2). Your function should run the deadlock avoidance algorithm to check whether the lock should be granted. It should return a Boolean (or int) that indicate whether the request is successful or not.

Obviously, some data structures need to be updated if the request is granted, you need to decide whether you want to apply those changes inside or outside the function.

Output

Your program should provide the following output:

- Whenever a philosopher sit down on a table (thread), the system should output:
  - <Philosopher name> sits down at table < table number>
- Whenever a philosopher finishes and ready to leave (after sleeping for 2 seconds) output:
  - <Philosopher name> finishes, releasing <list of tools it releases>
    - The list should have k numbers, each number denoting the number of tools of type 0, 1, 2, … that it releases
- Whenever a philosopher request for tools, you should output

    - <Philosopher name> requests <list of tools it requests>
- If the request is granted, you should output
    - <Philosopher name> requests <list of tools it requests>, granted
- If the request is denied, you should output
    - <Philosopher name> requests <list of tools it requests>, denied

Extra Credit 1 (10 points) (Required for 7343 students, optional for 5343 students)

In this part, you would create a Request2() function to be used instead of Request(), where in this case Request2() can request any number of tools each time. Notice that the function will either grant the request for all the tools, or rejected it and grant none of the tools.

In the main loop for the threads, one should determine how many tools to request each time as follows:

- If t is odd, then still only request 1 item
- If t is even, let q be the number of tools on the list of the tools to be requested. Then the number of tools to request = 2 + (t div 25) mod (q – 1) [div is integer division

Extra Credit 2 (15 points)

This is build on top of extra credit 1. In this part, you would introduce pre-emption as part of the scheme.

If a philosopher request is denied, he/she must release one tool that he/she is holding back to the restaurant before continuing. The thread should sleep for 1 second, and then select a random tool to release before it continues.

Extra Credit 3 (10 points)

I would like you to investigate whether using pre-emption will help or hurt the running time of all the threads, and whether it increases or decreases the execution of the program

You should hardwire your program such that every request will request only 1 items. Then construct multiple input file. You should than run it on cases with or without extra credit 2 and calculate the total number of requests that is made, and the total number that is denied, and use it as the basis of performance of the system.

Since there are still some randomness involved, you should run each input file a multiple number of times and take the average and standard deviation as the measurement for comparison.

You should write a 1-2 page report to summarize your findings.

What to hand in

You should hand in ONLY, ONLY, ONLY the source code for your program, and your report for extra credit 3. You should have multiple versions of your program for each extra credit. (You can share code among the different versiosn e.g. have the same functions but have different main program, and specify what files to compile for each case).

If you submit more than that is needed (I do NOT want you to send me the WHOLE project tree, for instance), you will have points deducted.

Comment bonus (5 points)

The comment bonus for program 2 applies.