Project 2: Operating Systems

Dining/Drinking Philosophers

Due Date: as posted in Blackboard

Objective: This assignment will help you to understand thread programming and semaphores.

Part 1: The Classic Dining Philosophers Problem:

Five philosophers are seated around a circular table. Each philosopher has a plate in front of him. In the middle of the table is a bowl of spaghetti. The spaghetti is so slippery that a philosopher needs two forks to eat it. Between each pair of plates is one fork. The life of a philosopher consists of alternate periods of eating and thinking. When a philosopher gets hungry, he tries to pick up his left fork and his right fork, one at a time, in any order. No two philosophers may use the same fork at the same time. If the philosopher successfully acquires two forks, he proceeds to eat for a while. When he is done eating, he puts down his two forks and continues thinking. The only possible activities are eating and thinking.

The problem: devise a solution (algorithm) that will allow the philosophers to eat. The algorithm must satisfy mutual exclusion while avoiding deadlock and starvation.

Part 1 Specifications:

Your program should expand upon the code presented in the textbook for the Dining Philosophers problem. Your program should be able to handle n philosophers (where $1 < n \leq 15$), not just 5 philosophers. Your program should prompt the user for the number of philosophers to create. Your program should be implemented using threads (where you spawn off one thread for each philosopher), and thread semaphores. Your program should be deadlock and starvation free.

Program Output:

Your program should generate the following output to standard output (i.e., the display). When a philosopher begins an activity, you should print a message of the form:

```
Eating Activity
          1
01234567890
*  * *  * *
```

In this example, there are 11 philosophers, and philosophers 1, 2, 4, 6, 7, and 9 are thinking, and philosophers 0, 3, 5, 8, and 10 are eating. Note, you should never have the case where all philosophers are eating simultaneously. There are other conditions which should never occur as well (think about these conditions, as they will help you formulate the problem).

You should create a global *activity* array with one element per philosopher. A value of 0 means that the philosopher is engaged in thinking, and a value of 1 means that the philosopher is engaged in eating. You should create a semaphore *access_activity* which governs the reading and writing of data into this array. Each time a philosopher begins an activity, he should acquire this

semaphore, update the philosopher's element of the activity array, and then based on the values in the array, print out both activity displays in the format above.

As a longer example, consider the output lines:

```
Eating Activity
          1
01234567890
 *
 *   *
 *   * *
     * *
*    * *
* * * *
* *    *
* *    *   *
* *        *
* *   *    *
*     *    *
      *    *
      *
    * *
    * *     *
    * *   * *
    *     * *
          * *
          *
 *        *
 *
 *      *
 *   *  *
```

Each line of output shows a start/stop in activity for one philosopher.

You should use a random number generator to determine the amount of time that a philosopher should be engaged in thinking, as well as how much time he should be engaged in eating (from the point in time he enters the eating critical section). The minimum activity period for thinking or eating should be 1 second. The maximum period for eating should be 3 seconds, and the maximum period for thinking should be 10 seconds (not including the time it takes to acquire the semaphores in order to enter the critical section).

In addition, your program should run until every philosopher has had a chance to think and eat at least 5 times. After this condition is satisfied, your program can stop and terminate. Be sure to clean up any remaining threads before your process/program exits.

The Drinking Philosophers Problem:

Processes, called philosophers, are placed at the vertices of a finite undirected graph G with one philosopher at each vertex. Two philosophers are neighbors if and only if there is an edge between them in G.

A philosopher is in one of 3 states: (1) thinking, (2) thirsty, or (3) drinking. Associated with each edge in G is a bottle. A philosopher can drink only from bottles associated with his incident edges. A tranquil philosopher may become thirsty. A thirsty philosopher needs a nonempty set of bottles that he wishes to drink from. He may need different sets of bottles for different drinking sessions. On holding all needed bottles, a thirsty philosopher starts drinking; a thirsty philosopher remains thirsty until he gets all bottles he needs to drink. On entering the drinking state a philosopher remains in that state for a finite period, after which he becomes tranquil and goes back to thinking.

Part 2 Specifications:

Part 2 of your program should expand upon the classic dining philosophers solution. Your project should read in a matrix representing a graph. This matrix will contain a 1 in a cell for philosophers connected by an edge (and hence a bottle), and a 0 otherwise. Again, your program should be able to handle n philosophers (where $1 < n \leq 15$), not just 5 philosophers. The size of the matrix will indicate how many philosophers there are. Your program should be implemented using threads (where you spawn off one thread for each philosopher), and thread semaphores. Your program should be deadlock and starvation free. As with the forks in the dining philosophers problem, no two philosophers should be holding the same bottle at the same time (which means you will need a semaphore for each bottle).

Program Input:

You should prompt for the filename of the file containing the bottle matrix. This input file will be similar to the following:

```
0  1  0  0  1  1
   0  0  0  1  1
      0  0  0  0
         0  1  0
            0  1
               0
```

which represents the matrix:

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Note: the matrix is symmetrical and the diagonal should always be 0s since a philosopher is not allowed to share a bottle with himself. In this example, there are 6 philosophers. Philosopher 1 shares a single distinct bottle with philosopher 2, another single distinct bottle with philosopher 5, and another single distinct bottle with philosopher 6. Likewise, philosopher 2 shares a bottle with philosopher 1, another with philosopher 5 and another with philosopher 6. You can identify the bottles based upon the philosopher numbers for the philosophers that share it. The bottle identifier should be an ordered pair (i, j), where $i < j$. The bottles are ordered according to the following rule:

if (i_1, j_1) and (i_2, j_2) are two distinct bottles (in other words $i_1 \neq i_2$ or $j_1 \neq j_2$), then

(i_1, j_1) < (i_2, j_2) if $i_1 < i_2$, or if $i_1 = i_2$ and $j_1 < j_2$
(i_1, j_1) > (i_1, j_2) otherwise

Program Outputs:
Your program should generate output as follows. Each time a philosopher changes state to drinking, the thread for that philosopher should obtain a semaphore controlling access to the terminal, and then print a message similar to:

Philosopher 4: drinking from bottles (1, 4), (2, 4), (4, 6).

This indicates that philosopher 4 has acquired bottles that is shares with philosophers 1, 2, and 6, and is proceeding to drink.

When a philosopher is done drinking, it should again obtain a semaphore controlling access to the terminal, and then print a message similar to:

Philospher 4: putting down bottles (1, 4), (2, 4), (6, 4).

You should use a random number generator to determine the amount of time that a philosopher should be engaged in thinking, as well as how much time he should be engaged in drinking (from the point in time he enters the drinking critical section). The minimum activity period for thinking or drinking should be 2 seconds. The maximum period for drinking should be 5 seconds, and the maximum period for thinking should be 10 seconds (not including the time it takes to acquire the semaphores in order to enter the critical section). In addition, random number generators should be used to non-deterministically select the set of bottles from which a philosopher will drink each time he changes to the drinking state.

As before, your program should run until every philosopher has had a chance to think and drink at least 5 times. After this condition is satisfied, your program can stop and terminate. Be sure to clean up any remaining threads before your process/program exits.

Randomly selecting bottles to use
You can use the following algorithm to randomly select a set of bottles a philosopher will want to drink from during each drinking session. At the beginning of the thirsty state, the philosopher will do the following:

1. put the bottle identifiers for bottles he shares with others in a list of size *n* (where *n* is the number of bottles he shares with other philosophers) by randomly putting the bottle identifiers in the slots of the list (to make it simple, you can use linear probing when collisions occur)

2. in a loop, randomly select a value *m* between 1 and *n* (where *n* is the number of bottles he shares with other philosophers), and then selecting the bottle at position *m* in the list of bottles. Repeat this loop *r* times, but ignore duplicate selections.

Notes:

- To compile your program you should use something like:

  g++ prog3.cpp –lpthread

- Your program should not place unnecessary limits on the philosophers.
- Your program should ensure that no deadlocks occur and that no starvation occurs either.
- Programming style matters. I will take off points if your program exhibits poor programming style.
- Project submissions should be the same as before.