Operating System

CS 370

Spring 2017

Project: 4

Peterson Leader Election Using Threads

Due Date: Friday, March 17th 2017

Must be submitted before midnight

Total Points: 100

Teaching Assistant1: Pradip Singh Maharjan (maharp1@unlv.nevada.edu)
Office Hours:

TuTh:11:00 am to 1:00 pm

Teaching Assistant2: Jung Lee (leej26@unlv.nevada.edu)

Course Web Page : `http://osserver.cs.unlv.edu/moodle/`

Requirements:

- Must be implemented in C (not C++)
- Must compile on cardiac.cs.unlv.edu
- First submission of this project must compile as “gcc project4a.c -pthread” . Similarly, second or third submissions must compile as “gcc project4b.c -pthread” or “gcc project4c.c -pthread” respectively.
- Project must be submitted through website ONLY.
- Input must be from redirection

Purpose:

The purpose of this project is to gain more experience with multi threaded programming. You will use semaphores not only for critical sections but for synchronization. You will get to experience a distributed algorithm at work.

Description:

For this project you will implement Peterson’s distributed leader election algorithm. You are to use a thread for each node. Because this will be a shared memory system and the algorithm is designed for a message passing environment you will need to simulate this. This algorithm uses a unidirectional ring network so you will need to implement a channel to pass messages in one direction between each node.

Building the Channel:

You will need to implement a method to write to a channel and a method to read from the channel. There may be situations where a thread will read from a channel, but there is no data in the channel yet. In this situation you will need to make the read call block until data has been written to the channel. To achieve this blocking behavior you cannot use busy waiting! You are to use semaphores to synchronize the methods. Each channel is basically a queue of messages (FIFO) and for this algorithm must be able to hold at least n values (n = the number of nodes in the network). Because the queue part of the channel will be used by more than one thread at a time (one will read, one will write) the queue code will need to be in a critical section. Here is pseudo code for the read method:

```
int read(channel) {
  wait(synchronization_semaphore);
  wait(race_condition_semaphore);
  int value = remove value from the queue
  post(race_condition_semaphore);
  return value;
}
```

And here is the pseudo code for the write method:

```
void write(channel, value) {
  wait(race_condition_semaphore);
  put value on the queue
  post(race_condition_semaphore);
  post(synchronization_semaphore);
}
```

So to implement this algorithm you will need n channels (1, 2,. . . n) and n nodes (1, 2,. . . n) so node$_i$ will read from channel$_i$ and node$_{i-1}$ will write to channel$_i$. Remember that this is a ring network so node 1's left neighbor is node n. Each channel should at least contain a queue, semaphore for race condition and a semaphore for synchronization. I would suggest putting these in a struct.

Implementing the Algorithm:

Each node will start executing as an active node. Here is the pseudo code for one phase of an active node:

```
write temp uid
read one hop temp uid
write one hop temp uid

read two hop temp uid
if one hop temp uid == temp uid
 =>this node is the leader
else if one hope temp uid > two hop temp uid && one hope temp uid >
    temp uid
 =>this node continues to be an active node
 =>temp uid = one hope temp uid
else
 =>this node becomes a relay node
```

Here is the pseudo code for one phase of a relay node:

```
read temp uid
write temp uid
read temp uid
write temp uid
```

During the execution of the algorithm your program will need to print out some node data. Each active node must print the following at the beginning of each phase:

```
[phase number][node uid][node temp uid]
```

Once a node has become a relay node it does not need to print. The leader node needs to print when it is found to be the leader:

```
leader: node uid
```

Input Specification:

The input must be from redirection. The input file will contain one number per line. The first line will be the number of nodes in the ring network. This will be followed by a line for each node that specifies the uid. You can assume good input and no ids will be repeated.

Hints:

- Use a value of 1 in sem_init for race condition semaphores
- Use a value of 0 in sem_init for synchronization semaphores
- Use a channel struct

Sample Input/Output:

Here is the sample input

```
16
25
3
6
15
19
8
7
14
4
22
21
18
24
1
10
23
```

Here is the output to this input:

```
[1][25][25]
[1][3][3]
[1][6][6]
[1][15][15]
[1][19][19]
[1][8][8]
[2][8][19]
[1][7][7]
[1][14][14]
[1][4][4]
[2][4][14]
[1][10][10]
[1][1][1]
[1][24][24]
[1][23][23]
[2][3][25]
[1][22][22]
[1][18][18]
[1][21][21]
[2][21][22]
[2][1][24]
[3][8][25]
leader: 8
```

Point Breakdown:

• Reading the input correctly	[10 pts]
• Implementing the channels correctly (no busy waiting)	[25 pts]
• Outputting during each phase correctly	[25 pts]
• Finding the correct leader	[25 pts]
• Coding style	
– Proper and informative comments	[5 pts]
– Informative variable names	[5 pts]
– Code organization (methods are not too long)	[5 pts]

References:

Algorithm reference:

`http://www.cs.wustl.edu/~kjg/CS333_SP97/leader.html`