CS 5352
Project 1 – Peer-to-peer communication using flooding
Due: 10/13/14 at 11:59 pm

Problem: Peer-to-peer based computing has been widely used at all levels such as to realize user-level applications, kernel-owned applications such as servers, and inter-connection between network routers (hardware). Broadcasting data is a common method of communication between peer-to-peer applications. Flooding is a simple method of realizing broadcasting. The purpose of this project is to implement message flooding for communication between peers. We assume a static configuration, i.e., all peers form a network at the beginning, each peer sets up connection with a few other peers, connections remain unchanged, and that no peer leaves in the middle. For a quick overview of the project, visualize a network of peers as an undirected, non-weighted graph with vertices representing the peers and edges representing communication links. Let vertex number 1 be called *init*, which is the only peer that generates messages. It generates a pair of integers called message and TTL and delivers the pair to a fixed peer (vertex number 2). Peer 2 picks up each pair in order, reduces its TTL by 1, and if it is not zero, it sends the message and the updated TTL to every peer connected to it directly. Other peers then do the same, i.e., they pick up each message in order of its arrival, reduce its TTL by 1, and if it is still not zero, they send the message to their neighbors other than the one who sent the message. If the initial TTL is large enough, the message would reach to all peers in the network; many peers may get the same message multiple times, however. In this project, peers are threads of a multi-threaded program, all running the same procedure. Each peer owns a queue and its neighbors send a message to it by putting it in this queue.

Goal: This project is designed to teach you several concepts and programming constructs such as multi-threaded programming, use of locks, simulation of concurrency on a single processor system using nanosleep(), queue management, dynamic memory allocation, etc. It is a complex project and hence, start early. The project description uses certain names of the data structures and procedures to describe design and implementation. **DO NOT USE THE SAME NAMES IN YOUR PROGRAMS.**

The project is divided into several stages to develop the project incrementally.

Stage1: Understanding the given multi-threaded program code

Create a folder "cs5352" in your HOME directory. Create a subfolder "proj1" in your "cs5352" directory for this project. Copy "~cs5352/projects/proj1/mt_demo.c". It contains core code of a multi-threaded program prepare to help you develop this project. This program contains two procedures main() and peer(). Main() defines a constant N and initializes a global table NW[N+2][N+2] with some random data, creates N threads, and then waits for them to finish. Since main() is also a thread, there are altogether N+1 threads. NW holds connectivity between peers, i.e., its i^{th} row gives the index of all neighbors connected to the i^{th} peer. Since Solaris assigns thread-id of 1 to main() and 0 is not used, row/column 0 and 1 of NW are essentially unused and N peer threads are assigned id from 2 to (N+1); it should explain why NW is of size (N+2)x(N+2). Every link between two threads is bi-directional. Main() also creates (N+2) lock variables, the last N for each of N threads. In a multi-threaded program, there is only one copy of the global variables, but there is a separate copy of local variables (those declared in the thread procedures), i.e., each thread is virtually a separate program and that only it can access its local data structures but communicates with other threads using global variables. Each thread runs the same procedure peer( ). A peer with thread-id i uses row NW[i] to determine its neighbors and saves the list of its neighbors in a local array initially.

On a single processor host, the CPU can execute only one thread at a time. In order to simulate concurrency between multiple threads, each thread should run for a short period, sleep for a short period, and repeat this cycle. While a thread is asleep, the CPU can run other threads that are not blocked. In mt_demo.c, threads use usleep() to sleep for a chosen number of micro-seconds. Selecting the proper sleep duration is important, because threads perform little computation in this project which can be easily completed in few micro-seconds.

Compile mt_demo.c using "-lthread" in the gcc command. It contains several print statements for debugging purpose. Follow the output and trace the execution of threads.

You can debug a multi-threaded program using the debugger, gdb, just the same way. Compile mt_demo.c with "-g" option and break at line 42, for example. You will see that the debugger reports three LWP (light weight process). You can use any gdb command on a thread.

Stage2: Reading network connectivity from the given input file.

Create a file stage2.c. Define N to 10. Declare a global variable r_num. Develop main() to perform the following.

Input: main() reads data from an input file passed as argv[1]. This file first describes the network layout and then gives the messages to be broadcast. Its first line contains the number of peers, say r_num (it is less than N=10). Next r_num number of lines give peer connectivity. Each line begins with a count, the number of neighbors of a peer, and then there is list of the neighboring peers. So, for example, if the k^{th} line contains 2, 3 and 7, it means that peer k has 2 neighbors, namely 3 and 7. Since main() (i.e., thread 1) is not connected to any peer, the second line contains 0 for the count. Next few lines give connectivity of peers 2 and up. Clearly, after reading R_num, main() runs in a loop 1 <= k <= R_num to read data on connectivity; in this loop, it first reads a count C and then runs an inner loop to find each neighbor. If x is a neighbor of k, main() sets NW[k][x] = NW[x][k] = 1, i.e., if peer x is connected to peer k, the later is also connected to the former. Since 0 and 1 are not connected to any one, NW[x][0] = NW[0][x] = NW[x][1] = NW[1][x] remains 0 for all x.

Each of the remaining lines of the input file contain a pair of integers representing a message and its TTL. The following is an example of an input file.

```
5        # number of peers (always < N)
0        # no peer is connected to thread 1
2 3 4    # peer 2 (thread 2) has two neighbors 3 and 4
1 5      # thread 3 (peer 3) is connected to one peer, peer 5 only
1 5      #thread 4 connected to one peer only
1  6     #peer 5 is connected to peer 6; no need to give connectivity for last peer separately
1000 3   # message = 1000 and TTL =3
2000 7   #message = 2000, TTL = 7
```

Stage3: Using queues for inter-thread communication

Assume that each peer owns a queue to receive messages from its neighbors. A queue is just a 1-D linked list of nodes. For purpose of this project, a node contains three data fields, sender's id, TTL, and a message.

```
typedef  struct MSG
          {unsigned int sender;
           unsigned int TTL;
           unsigned int msg;
           struct MSG *next;
           } node, *node_ptr;
```

Since queues should be accessible to neighboring threads, the head and tail of queues should be defined as global variables. main() declares two global arrays (defined below) and initialize them as empty lists.

```
node_ptr  Q_head[N+1], Q_tail[N+1];
```

Since the peers are numbered 2 to R_num+1, peer k owns k^{th} queue; 2 <= k <= R_num+1. Queue[0] and Queue[1] remain not used.

Copy stage2.c to stage3.c and expand it to build procedures: sendMSG(), enterQ(), and grabMSG( ). Their brief descriptions are as follows. Main() should also create and initialize a lock array.

sendMSG( )
{1. Allocate a new node dynamically.
2. Copy sender ID, TTL, and msg passed as arguments into the new node
3. enterQ( )
}

enterQ(node, qID)
{ 1. Lock qID
2. enter node in the queue of qID
3. unlock qID.
}

grabMSG(x)
{1. lock the queue x
2. delete the first node y of queue x
3. unlock queue
3. return y;
}

After creating the queue handlers, add the following to main() to test your queue handling routines.

```
sendMSG(1, 2, 1000, 1); //sender = 1, receiver=2, msg = 1000, TTL=1
sendMSG(1, 2, 2000, 2);
sendMSG(1, 4, 2000, 4);

If  ((y = grabMSG(2)) != NULL) printf (content of node y)
Else printf suitable error;
If  ((y = grabMSG(3)) != NULL) printf (content of node y)
Else printf error;
If  ((y = grabMSG(2)) != NULL) printf (content of node y)
Else printf suitable error;
```

Before exiting, main() tests each queue. If Q_head[i] != NULL, it prints a suitable one line message. Finally, it destroys the lock array.

Stage4: One round of propagation

Copy stage3.c to stage4.c. Remove the statements that insert and delete dummy messages above. After reading the connectivity data, main() should now create R_num threads and should go on to wait for threads to complete. Main() also opens a globally accessible Log file. Next, code peer() to implement the following.

peer( )
{1. myID ← argument.
2. Declare an array myNEIGHBORS[N+1] and copy NW[myID][*].
Print myID and myNEIGHBORS[myID]
3. usleep() //sleep little to let all other peers to run.
Return NULL;
}

Stage5: Flooding of a dummy message.

Copy stage4.c to stage5.c. Add "sendMSG(1,2, 2000, 3);" to main() after it has created all threads. Expand peer() as per the following.

```
peer( )
{1. myID ← argument.
2. Declare an array myNEIGHBORS[N+1] and copy NW[myID][*]
Print myID and myNEIGHBOR[myID] on terminal
3. usleep()

4. y = grabMSG( myID) // pick and remove a message from own queue
5. if (y != NULL)
  {fetch content of node y;
   If (TTL > 1)
    { TTL --;
      Came-from = sender of y;
      for each neighbor x other than sender of message
        { sendMSG();} //send a copy of message to x
          Print (came-from, myID, sent-to, msg, TTL) to Log file //note order of items
       }
   }
 }
8. free msg-node grabbed in step 4.
9. usleeps(t) // t micro-sec

10. Return NULL;
}
```

The above gives core of the flooding algorithm. A peer attempts to pick up the first item from its own queue. If it finds one, (i.e., y is not NULL), it propagates to all its neighbors skipping the one where the message came from, provided that TTL > 1.

Solaris schedules threads to run in random order. If thread-2 is not scheduled first, no other peers would have anything in their queue. How long should steps 4-9 run before the peer returns to main()? It is a BIG question. I came up with a simple but temporary solution. Declare a global variable num_entries_queues. enterQ() increases it by 1 when it adds an entry in ANY queue and grabMSG() decreases by 1, when it deletes an entry from any queue. It is my temporary answer to the BIG question. Therefore **every peer should run steps 4-9 in a while loop as long as there is an entry in <u>any</u> queue. By adding this loop, you also solve the problem what if peer 2 does not get to run first.**

Testing: The log file should give the items shown in the print statement only. From the log file, verify manually that (a) all messages come from a neighbor of myID, and (b) no message is sent back to the peer it came from.

Stage6: Multi-message flooding algorithm.

Copy stage5.c to stage6.c. Replace the sendMSG() statement in main() by a loop. In this loop, main() should read the rest of the input file, one line at a time. Each line contains a message and TTL. Main() sends each message by deposing it to the queue of peer 2. After sending a message, it sleeps a little to give the message time to propagate to all peers. Rename the log file as per the stage; e.,g., stage5.c should produce file, "stage5_log".

<u>Remove all print statements that **you added for debugging** after you have completed the program.</u>

Run your program with Inp1, Inp2, and Inp3 input files. These are stored with mt_demo.c. Log **items should be separated by spaces only and aligned.** You can sort the Log by any column. For example, if you want to list messages sent by peers in ascending order, you can run "sort –k 2,3 stage3_Log". **This is how I will test the output**.

Stage7: Simulating distributed peers

Copy stage6.c to stage7.c. By creating the global variable num_entries_queues, we assumed that all peers always have globally shared memory. In distributed computing environment, it is not true. Therefore delete this variable and all references to this variable.

How do we determine if all messages have been delivered by every thread to their neighbors? There is no direct answer. An indirect answer to this question is that when a thread goes to grab a message and it finds the queue empty and that it happens for num_EMPTY rounds consecutively, the thread should get out of the loop and return to main(). The following code describes this logic:

```
Num_rounds_no_msg = 0;
While (num_rounds_no_msg < num_EMPTY)
{
        If  (my Q_head is NULL) num_rounds_no_msg ++;
        Else
        While ( …)
        { grab next msg and broadcast to neighbors, if TTL > 1;
          Num_rounds_no_msg = 0;
        }
        usleep( )
}
```

This stage should produce stage7_log file. Test this stage with input file Inp4 and Inp5.
Each peer should print at the end the average number of rounds the queue became empty before finding a message. Also print the smallest number of num_EMPTY for which all messages get propagated.

Submission: Remove all source files except stage3.c, stage5.c, stage6.c and stage7.c. Each stage should produce an appropriate log file. Some sample log files are provided. **Make sure that you implement the logic of a stage in that stage only. It would help me grade your program easily.**

Copy readMe from the same folder and edit it to to tell me what works. Follow the project submission method to submit your project.

Grade: Stage3: 40%, Stage5: 40%, Stage: 15%, stage7: 5%. No credit if a stage does not compile.

IT IS VERY IMPORTANT THAT EVERYONE WORKS INDIVIDUALLY. YOU MAY NOT SHOW YOUR CODE TO OTHERS. Plagiarized programs shall receive zero credit. You can discuss the assignment with others, but create the source file and debug it independently. Student programs may be submitted to the moss software for plagiarize detection.

POST PROJECT DISCUSSION

1. Scheduling is dynamic. Hence, the same program for the same input need not produce exactly the same output every time. It means that your log for a stage need not be exactly the same every time.
2. Is it possible that peers start returning after the num_entries becomes 0, but the main() has yet to finish up broadcasting all messages. If so, how? What is your remedy?
3. Is it possible that peer 2 grabbed the last MSG (num_entries = 0), but it is yet to broadcast that MSG to its neighbor and neighbors start returning due to num_entries = 0? If so, what is the remedy?

4. Stage 7 asks for specific statistics. It should be gathered and printed by the program.
5. Delete all print statement you added for debugging. Leave only those print statements that produce log. Test print statement should be complete so that it identifies the source of information also.
6. readMe discusses tests for each stage. Your output should show that you realized it.