CS 655 : Distributed Simulation

Time Warp Simulator - Code description in detail

by Omkar Deshmukh 260463896

The code consists of following files

main.cpp

event.h

Queueclass.h

stackclass.h

heapoverall.h

exponentialrv.h

Queueclass.cpp

stackclass.cpp

heapoverall.cpp

exponentialrv.cpp

3 main data structures have been used in this code.

- **Min Heap**
- **Queue**
- **Stack**

Min Heap is the data structure which stores the time stamps to be used in the code. This data structure is represented with a class and coded in heapoverall.cpp

Queue structure is used at many places. All LPs have an input queue. LP1, LP2 and LP4 have an output queue too. Queue is made of events. This data structure is represented with a class and coded in Queueclass.cpp

Stacks are implemented for states of LP3 and LP4 as well as output of LP4. They are used in the rollback and the annihilation process. Stacks are represented with a class and coded in stackclass.cpp

A message or event has following fields:

1. Timestamp - refers to the timestamp on the message

2. Original source - refers to the LP from which the message originated - LP1 OR LP2

3. Pointer to next message

4. Pointer to past message

In main.cpp, there are three main parts. The first part concerns the declaration of global variables and include statements. The second part has definition of Logical Processor class which has functions for all the LPs. The third part has the main() function.

In the first part, class objects for all the queues, stacks and heap are defined. Processing times of all 4 processors are set here.

The second part has definition of Logical Processor class and member function for different LPs

Functionality of LP1:

while(1) loop is present to show that LP is always working.

When LP1 wants to access the heap to pick up a message, it checks if the corresponding mutex is being used. If it is being used, LP1 waits. When the mutex is free, it is an indication that the heap is not being used by any other thread. When LP1 is done accessing the heap, it sets the mutex to be free so that other threads can access the heap using appropriate mutex.

The accessed element from the heap is turned into an event and put in the input queue of LP1. From the input queue, it is moved to the output queue of LP1. At this point, the timestamp of the event is increased to reflect the processing time of LP1.

Using a mutex for the input queue of LP3, LP1 transfers its events in output queue to the input queue of LP3. The working of mutex here is same as explained above. If the input queue of LP3 is being accessed by some other process, the mutex will not be free and LP1 will wait. When the mutex becomes free, LP1 will pass the events in its output queue to the input queue of LP3.

Appropriate care is taken to see that a queue is de-queued only when it is not empty.

Functionality of LP2:

Functionality of LP2 is same as that of LP1. It picks up a timestamp from the heap using mutex. The time stamp is converted into an event and stored in the input queue. It's passed to the output queue and then goes to the input queue of LP3 using a mutex. Timestamp of the underlying event is suitably increased to reflect the processing time of LP2.

Functionality of LP3:

A while(1) loop is present to represent that the LP is always working.

The LP has a while loop to check if its input queue is empty. As soon as it finds that the input queue is not empty, it breaks out of the while loop to act on the input queue.

A mutex is used to work with the input queue. If the mutex is free, an event is picked from the front of the input queue. This event is then compared with the top event at the stack of states for LP3.

If the stack is empty or if the new event has bigger timestamp than the event at the top, LP3 functions normally. It puts the new element at the top of the stack and appropriately loads the output stack. It then passes on the timestamp and origin of this event to other processor using MPI.

If the new event has a timestamp lower than the timestamp, the roll back takes place!

In the roll back, a temporary stack is created for both the events in the state stack as well as the output stack. Using a simple while loop, all events in state stack with timestamps bigger than the timestamp of the new event are removed. Corresponding events in the output stack are removed too. Removed events from the state stack are put back in the input queue of LP3. Removed events from the output stack are forwarded to the other processor as anti-messages. This is followed by a regular processing of the new event that we earlier picked up from the input queue of LP3. This event is put on the top of the modified stack and appropriately added to the output stack. Its timestamp and origin are then sent to the other processor using MPI.

Functionality of Physical Processor two

receives the messages sent over MPI by LP3. The task of Physical processor two is to receive these messages, convert them into event data structure and pass them onto LP4. It does so inside a while(1) loop and using mutex for the input queue of LP4. Again, the mutex works in the same way as mentioned before.

Functionality of LP4:

LP4 has an input queue, a state stack and an output queue. The processing time of LP4 is set globally as mentioned before. LP4 waits for its input queue to be non empty. Once it is non empty, LP4 uses a mutex to access the input queue. It looks at all the events in the input queue to search for anti messages. If no anti messages are found, it de-queues an event from the input queue and processes it normally - puts on the stack and the output queue by increasing the timestamp appropriately.

If there are anti messages present in the input queue, LP4 annihilates them by searching for their corresponding messages in the input queue or the stack. Key observation here is that an anti-message can't come ahead of its message. Hence, any incoming anti-message can be annihilated. Annihilation takes place either in the input queue or the state stack. This is carried out till all anti messages in the input queue are annihilated. At the end of this process, if there are any normal messages left in the queue, they are again processed normally.

main()

In main(), MPI is initialized and processor ranks are determined. For processor 0, threads are defined for each of LP1 , LP2 and LP3. These threads are then started and joined. Heap is also initiated in processor 0. Number of elements in the heap is a global variable. For processor 1, threads are defined for each of LP4 and PP Two. These threads are then started and joined. MPI is finished at the end of the function.

Termination of code: A global variable is set when the input queue of LP4 is empty and the number of stacked states of LP4 = initial number of elements in the heap. This variable then helps in terminating each thread.

Miscellaneous:

Sleeping time of all processors is random. It is arrived at finding the difference between two rand() calls. This helps in bounding the total value of the sleeping time.

Inter arrival time of events is exponential. In the code, I have modeled this an exponential random variable with mean 2 and minimum value 1.

The file exponentialrv.cpp is responsible for finding numbers distributed exponentially. This is done using a uniform random variable.

Let U represent a uniform random variable between 0 and 1. X represent an exponential random variable with mean m.

Then, X is given by that value of CDF of X which makes it equal to a given value of U.

$x = -\log(1-u)/m$