**Advanced Operating Systems Spring 2019,**

**Programing Assignment #3**

## DESCRIPTION

## PART 1 - LOGICAL CLOCKS

## DESCRIPTION

Write a C or C++ program named logical.c or logical.cpp (executable name should be logical) to implement Lamport's logical clocks using MPI. The screenshot from jumpshot should be saved as logical.pdf. You will implement a set of primitives to maintain the state of logical clocks. To show that your logical clocks work correctly, your program should read in a set of simulation parameters and then print the logical clocks associated with each event. It is recommended that a single manager process and a set of worker processes be implemented. The manager process will manage the simulation, and the worker processes will exchange messages and implement the logical clocks.

## INPUT TO THE PROGRAM

The input will be in the following format:

number of processes N
event type pid
event type pid1 pid2 msg
...
end

Process IDs in the input file will range from 1 to N (You need N+1 processors to run this simulation). There will be exactly one input event per line. There are two event types that may appear in the input: exec and send. An exec event has no second argument, and indicates that an execution event has taken place in a process. A send event denotes the sending of a message from one process to another. The message will be a string of printable ASCII characters delimited by quotes. The quotes should not be stored as part of the string.

Example file named logical.txt:
3
exec 1
send 1 2 "silly message"
end

Sample Execution:
mpirun -np 4 logical logical.txt

**PROGRAM OUTPUT**

For execution events, print that the event has occurred and the logical timestamp at the process. For a send event, print the relevant information at both the sender and receiver. After each event, you should output any logical clocks updated by the event, and only those. At the end of the run, print the logical clocks of all processes in the system in ascending rank order.

At the beginning of the simulation, output the number of processes as following:

There are N processes in the system

For each event, you should use the equivalent of the following printf() statements:

```
printf( "Execution event in process %d\n", pid );
printf( "Message sent from process %d to process %d: %s\n", pid1, pid2, msg );
printf( "Message received from process %d by process %d: %s\n", pid1, pid2, msg );
```

For any logical clock printed, use the following statement (or similar):

```
printf( "Logical time at process %d is %d\n", pid, logical_time );
```

**MISCELLANEOUS**

1. Start all clocks at time 0.
2. The clock delta, d, should be 1 for all processors.
3. Your MPI program will read the file name as command line argument and write the output to standard output.
4. The only MPI functions you are permitted to use are MPI_Init(), MPI_Finalize(), MPI_Comm_rank(), MPI_Comm_size(), MPI_Send(), MPI_Recv(), and MPI_Barrier().
5. You may use the MPI_Probe function if needed.
6. You may use an acknowledgement message to ensure that the current action (e.g. a message being sent between slave processes) has concluded before the master reads the next action.

**PART 2 - VECTOR CLOCKS**

**DESCRIPTION**

Write a C or C++ program named vector.c or vector.cpp (executable name should be aosproj2_1) to implement vector clocks using MPI. The screenshot from jumpshot should be saved as vector.pdf. You will implement a set of primitives to maintain the state of vector clocks. To show that your vector clocks work correctly, your program should read in a set of simulation parameters and then print the vector clocks associated with each event. It is recommended that a single manager process and a set of worker processes be implemented. The manager process will

manage the simulation, and the worker processes will exchange messages and implement the vector clocks.

**INPUT TO THE PROGRAM**

The input will be in the following format:

number of processes N
event type pid
event type pid1 pid2 msg
...
end

Process IDs in the input file will range from 1 to N (You need N+1 processors to run this simulation). There will be exactly one input event per line. There are two event types that may appear in the input: exec and send. An exec event has no second argument, and indicates that an execution event has taken place in a process. A send event denotes the sending of a message from one process to another. The message will be a string of printable ASCII characters delimited by quotes. The quotes should not be stored as part of the string.

Example file named vector.txt:
3
exec 1
send 1 2 "silly message"
end

Sample Execution:
mpirun -np 4 vector vector.txt

**PROGRAM OUTPUT**

For execution events, print that the event has occurred and the vector timestamp at the process. For a send event, print the relevant information at both the sender and receiver. After each event, you should output any vector clocks updated by the event, and only those. At the end of the run, print the vector clocks of all processes in the system in ascending rank order.

At the beginning of the simulation, output the number of processes as following:

There are N processes in the system

For each event, you should use the equivalent of the following printf() statements:

printf( "Execution event in process %d\n", pid );
printf( "Message sent from process %d to process %d: %s\n", pid1, pid2, msg );

printf( "Message received from process %d by process %d: %s\n", pid1, pid2, msg );

For any vector clock printed, use the following statement (or similar):

printf( "Vector time at process %d is %d\n", pid, vector_time );

**MISCELLANEOUS**

1. Start all clocks at time (0,0, ...0).
2. The clock delta, d, should be 1 for all processors.
3. Your MPI program will read the file name as command line argument and write the output to standard output.
4. The only MPI functions you are permitted to use are MPI_Init(), MPI_Finalize(), MPI_Comm_rank(), MPI_Comm_size(), MPI_Send(), MPI_Recv(), and MPI_Barrier().
5. You may use the MPI_Probe function if needed.
6. You may use an acknowledgement message to ensure that the current action (e.g. a message being sent between slave processes) has concluded before the master reads the next action.