Programming Assignment 1

Work done

by

Arun Prakash Themothy Prabu Vincnet

NET ID : AXT161330

SPRING'17 CS/TE6385

CS/CE 6352 Performance of Computer Systems and Networks

CONTENTS

No.	Chapter	Page
1	Project description	3
2	Graphs	6
	2.1 rho(ρ) vs Average Components E(N)	6
	2.2 rho(p) vs Blocking Probability	6
	2.3 rho(p) vs Average Time	7
	2.4 rho(p) vs Utilization	7
3	Appendices	8
	3.1 Appedix – Source Code	8

Chapter 1

Project Description

In this problem, you will implement an event-driven simulation of a queueing system. In an event-driven simulation, the system state is updated only when an event (e.g., an arrival or a departure) occurs, rather than being updated at periodic time intervals. When an event occurs, several steps must be taken to update the system state. The first step is to update the system time to the time at which the event occurred. The next step is to update any other state parameters, such as the number of customers in the queue. Finally, new events are generated based on the current event. Once the system state is updated, the simulation moves on to the next event in chronological order.

Queueing System Description

Two machines generate components that are sent to a processing center for packaging. The first machine generates components according to a Poisson process with rate γ components/minute. The second machine generates components according to a Poisson process with rate λ components/minute. If there are two or more components in the processing center, the first machine stops generating components. If there are K or more components in the processing center ($K \geq 2$), the second machine continues to generate components, but these components are discarded (they do not enter the queueing system). The processing center employs m workers who package the components. The time it takes each worker to package a component is exponentially distributed with an average packing time of 1 μ minutes.

Event-Driven Simulation

In the simulation of a Markovian queueing system, we need to consider two basic types of events: arrivals and departures.
When an arrival event occurs, we need to perform the following tasks:
• Update the system time to reflect the time of the current arrival.
• Increment the system size if the system is not at its full capacity and the arrival is not blocked.
• If there is an idle server that can take the arriving customer, then generate a departure event for the new arrival. The departure time will be the current system time plus an exponentially distributed length of time with parameter μ.

• Generate the next arrival event, if applicable. The time of the next arrival will be the current system time plus an exponentially distributed length of time.

When a departure event occurs, we must perform the following steps:

• Update the system time.
• Decrement the system size.
• If there are customers waiting in the queue, and if a server is available, then one customer will enter service when the departure occurs. Generate a departure event for this customer entering service.
• If applicable, generate an arrival event. (This step may not be necesary depending on how you implement the simulation).

Once the tasks associated with an event have been completed, the simulation should go on to the next event. In order to manage events, we can maintain an event list. An event list consists of a linked list whose elements are data structures which indicate the type of event and the time at which the event occurs. By sorting the event list in chronological order, the next event can be selected from the head of the event list. When a new event is generated, it is placed in the event list in the correct chronological sequence. Note that the event list is simply a data structure for keeping track of the timing of events in the simulation. The event list does not represent the state of the queueing system.

Collecting Performance Measures

When implementing the simulation, you will also need to maintain additional information in order to calculate performance measures for the system. In particular, you should determine the average number of jobs in the system, the average time a job spends in the system, and the blocking probability versus ρ, where ρ is defined as λ mμ . For each plot, the parameter ρ should range between 0.1 and 1.0, and each plot should contain at least ten data points, (i.e., ρ = 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0). The values of λ can be determined from the values of μ, ρ, and m, i.e., you will run the simulation for values of $\lambda = 0.1 \cdot m \cdot \mu$, $0.2 \cdot m \cdot \mu$, etc. For each data point, you should run the simulation for at least 100000 departures.

Experiments For the following experiments, do not hard-code values for K and μ into your program. Instead, the values should be entered as user inputs to the program. Additional cases may be tested during the grading of your program.

1. Let $\mu = 4$ components/minute and $\gamma = 5$ components/minute. Plot the average number of components in the system versus ρ for the case in which $m = 2$ and $K = 4$. (You may have your program output numerical values, and then create the plots using any standard plotting/graphing/spreadsheet program, such as MS Excel or Matlab.) Include in the same graph plots of the theoretical values of the expected number of customers in the system versus ρ. You may calculate the theoretical values by programming the appropriate equations into a computer program or by calculating the values by hand.

2. Plot the average time spent in the system versus ρ with $\mu = 4$ components/minute, $\gamma = 5$ components/minute, $m = 2$ and $K = 4$. Plot theoretical values for the expected time spent in the system on the same graph.

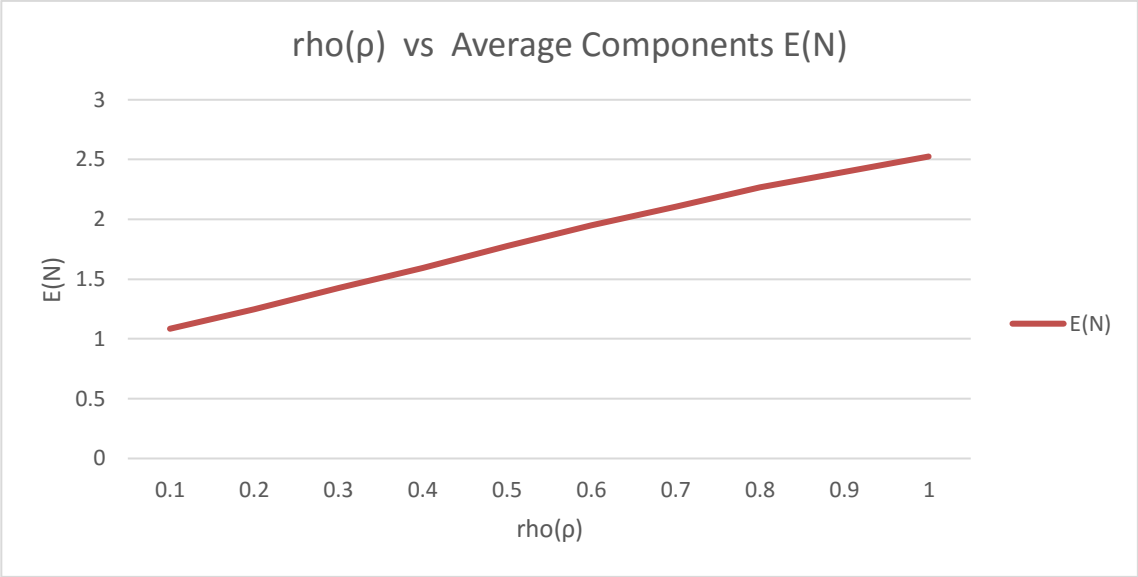
3. Plot the fraction of components that are discarded (blocked) versus ρ with $\mu = 4$ components/minute, $\gamma = 5$ components/minute, $m = 2$ and $K =$

4. Plot theoretical values for the blocking probability on the same graph. 4. Plot the total utilization of the system (all servers combined) versus ρ with the same parameters as above. Plot theoretical values for utilization on the same graph

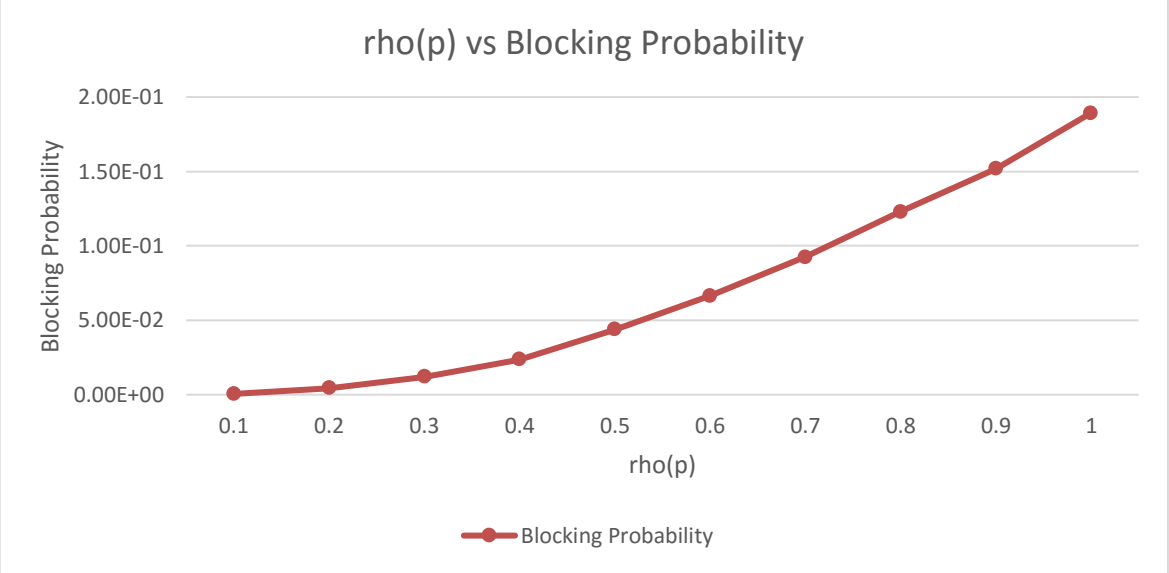
Chapter 3

Graphs :

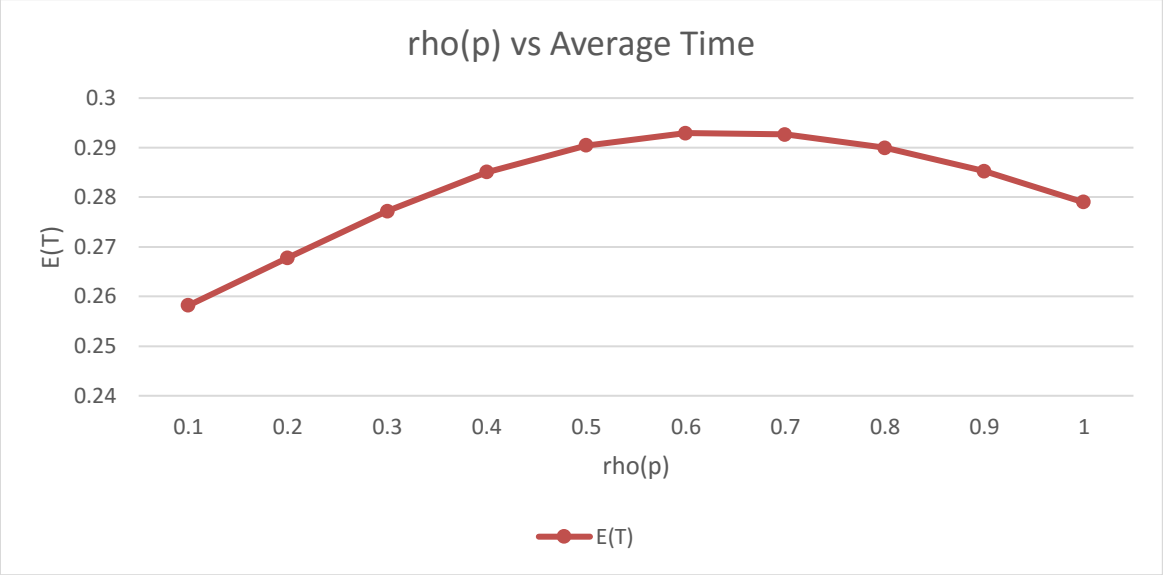
3.1 rho(ρ) vs Average Components E(N)



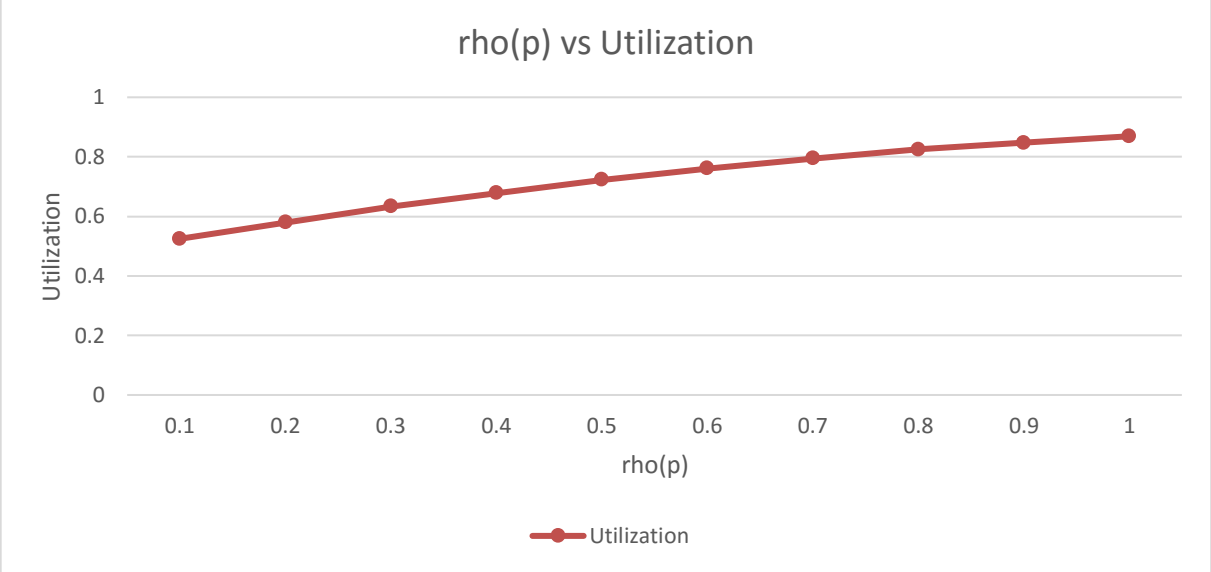
B. rho(p) vs Blocking Probability



3.3 .rho vs Average Time



3.4 rho(p) vs Utilization



Appendices

Appendix A – Source Code

Event.Java :

```
package PcsnProj1;

public class Event {

	int type; /* Type of Event 1-Arrival of machine1 comp
				2-Arrival of machine2 comp
				0-Departure */
	double timeStamp; // Timestamp of Event

	Event(double time, int t){
	  timeStamp = time; // Initialization
	  type = t;
	}
}
```

EventList.Java :

```
package PcsnProj1;

import java.util.LinkedList;
import java.util.List;
import java.util.Stack;

public class EventList {

	public static LinkedList<Event> eventList = new LinkedList<Event>();

	public static void listInsert( double timeStamp, int type){


	// System.out.println(timeStamp + "  " + type);
	  int eventAdded = 0; // to indicate if event is added to list
	  Event evnt = new Event(timeStamp, type); //creating a Event

	  if(eventList.isEmpty()){ // if the event list is empty
		eventList.add(evnt);
		//System.out.println("Arrival Event Added to List");
	  }
	  else // Insert event in the correct chronological position
```

```
        {
                int iterator = 0;
                while(iterator < eventList.size()){

                        Event temp = eventList.get(iterator);
                        if(temp.timeStamp < evnt.timeStamp){ //if arrived event has greater time
stamp
                                ++iterator ;
                        }
                        else{
                                eventList.add(iterator, evnt); //if current event has smaller
timestamp it is added to the list
                                eventAdded = 1; // set the event added status to success
                                break;
                        }
                }
            if(eventAdded != 1) // checking if the event the event is added
                eventList.addLast(evnt); /* the event has larger timestamps than
                                 all the available events in the list*/
              //  System.out.println("Arrival Event Added to List");

        }

      }

      public static Event listGetEvent(){

        if(eventList.isEmpty()) // checking if the list is empty
                return null;
        else
        {
                Event evnt = eventList.getFirst(); //Remove the element with the smallest
timeStamp
                eventList.removeFirst();
                return evnt; //return event
        }

      }


}
```

Main Program – EventDriven.Java :

```
package PcsnProj1;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Random;
import java.util.Scanner;

public class EventDriven {

	static double seed = 1111.0;


	public static void main(String[] args) throws FileNotFoundException{


	   Scanner scan;
	if (args.length > 0) { // if the input file is specified
	   File inputFile = new File(args[0]); // read the file
	   scan = new Scanner(inputFile); // get a scanner of the input file
	} else { // the input file must be provided
	   System.out.println("provide the location of the config file as the first input argument.");
	   scan = null;
	   System.exit(1);
	}


	   double delta; //production rate of Machine1
	   double lamda; //production rate of Machine2
	   double mu = 0; //processing rate of a single worker

	   int m = 0; // number of Servers
	   int K = 0; // capacity of the queue

	   //Read the inputs from the config File
	   delta = scan.nextDouble();
	   mu = scan.nextDouble();
	   m = scan.nextInt();
	   K = scan.nextInt();
	   int typeArrivalDelta = 1; //type 1 for arrival of component of machine1
	   int typeArrivalLamda = 2; //type 2 for arrival of component of machine2
	   int typeDeparture = 0; //type 0 for departure
	   double rho = 0.1; //initialization of rho
```

```
        //For each value of rho until 1
        while(rho <=1){
        lamda = (rho*m*mu); //calculate the production rate of machine 2

         System.out.println("\n For System Parameters rho = " + rho + " delta(Machine 1 Rate) =
" + delta + " lamda(Machine 2 Rate) = " + lamda + " m = " + m + " k = " + K);
         rho += 0.1;
         double Rate;
        int noOfComponents = 0;
        int noOfDepartures = 0;

         int systemSize = 0 ; // components in service + wait Queue
         int queueSize = 0; // components waiting for service
         double sysClock = 0.0;
         int noOfServerAvailable = 2; // No of workers available for packaging
        Event currentEvnt ;

        double timeStamp = exponentialRv(delta); //Production by machine 1
        EventList.listInsert(timeStamp, typeArrivalDelta); //Arrival of component from machine1

        timeStamp = exponentialRv(lamda);//Production by machine 2
        EventList.listInsert(timeStamp, typeArrivalLamda); // Arrival of component from
machine2

        //Initialization of System parameters
        int iter = 0;
        int totalArrival = 0;
        int totalEntered = 0;
        int blocked = 0;
        double E_N = 0.0;
        double Utilization = 0;

        //For 100000 Departures
        while(noOfDepartures < 100000){


                currentEvnt = EventList.listGetEvent(); // get event from the event List
                double prevClock = sysClock; //set the prev clock
                sysClock = currentEvnt.timeStamp; //update the system clock

                 //Calculate Utilization
                Utilization = calcUtilization(Utilization,systemSize,sysClock,prevClock);

                // If Event type is Arrival from machine 1
                if(currentEvnt.type == typeArrivalDelta){

                        E_N += (systemSize*(sysClock-prevClock));
```

```
                    if(systemSize < 2){ // checking if the system size is less than 2
                        ++systemSize ; //incrementing the system size
                        ++totalEntered ; //incrementing the arrival

                       /*checking if the server is available for processing*/
                       if(noOfServerAvailable >0 && systemSize>0){
                    EventList.listInsert(sysClock+exponentialRv(mu),
typeDeparture);/*Creating a departure Event if server is available */
                                        --noOfServerAvailable ;// setting server to busy
                                }

                       //if server is not available
                else{
                        queueSize++ ;//component added to queue for processing
                }
                        /*Generating Arrival Event*/
                           Rate = delta ; //Machine 1 generates components
                 timeStamp = sysClock + exponentialRv(Rate);
                 EventList.listInsert(timeStamp, typeArrivalDelta);// Event added to the list
                        }

                        else{
                                Rate = delta ;
                 timeStamp = sysClock + exponentialRv(Rate);
                 EventList.listInsert(timeStamp, typeArrivalDelta);
                        }


        }

        // If Event type is Arrival from Machine 2
          if(currentEvnt.type == typeArrivalLamda){

            E_N += (systemSize*(sysClock-prevClock));

            /*checking if the system capacity is full*/
             if(systemSize < K){
                    ++systemSize ; // updating the system size
                    ++totalEntered ; // updating the total arrival

                    /*checking if the server is available for processing*/
                    if(noOfServerAvailable >0 && systemSize >0){
                        EventList.listInsert(sysClock+exponentialRv(mu), typeDeparture);
                                --noOfServerAvailable ;
                        }
                //if server is not available
```

```
                else{
                        queueSize++ ;//component added to queue for processing
                }

                  Rate = lamda ; //Machine 2 generates components
            timeStamp = sysClock + exponentialRv(Rate);
            EventList.listInsert(timeStamp, typeArrivalLamda);

             }
             // If the system capacity is full the event is blocked
             else{
                 blocked++;
                 Rate = lamda ; //Machine 2 generating components
               timeStamp = sysClock + exponentialRv(Rate);
               EventList.listInsert(timeStamp, typeArrivalLamda);// Event added to the list
             }


          }


           //If Event Type is Departure
                 if(currentEvnt.type == typeDeparture){

                        E_N += (systemSize*(sysClock-prevClock));
                 --systemSize ; // Updating the system size - Decrementing it by 1
                 ++noOfDepartures ; // incrementing the number of departures by 1
              ++noOfServerAvailable; // making a server available

              //checking if any component is waiting in the queue for service
              if(queueSize>0){

                 //Checking if the Server is available
              if(noOfServerAvailable >0){
              timeStamp = sysClock + exponentialRv(mu);
                        EventList.listInsert(timeStamp, typeDeparture); //Generating departure
Event
                    --noOfServerAvailable ; //making the server busy
                    --queueSize; // updating the queue
                 }
              }
                 }
      //  System.out.println("System SIze : " + systemSize + " Total Arrivals : " + (totalArrival) + "
Total Entered = " + totalEntered  + " Departures : " + noOfDepartures );

        }
```

```
        double AvgN = E_N/sysClock; // calculating the Average Number of Components
        double AvgT = E_N/(totalEntered) ; //calculating the Average Time Spent in system by
Components
        double Blocked = (double)blocked /totalEntered; //calculating the blocking probability
        double util = Utilization/sysClock; // calculating the utilization
        /*Display System Parameters*/

        System.out.println("\n Program Values ");
        System.out.println("E[N] = " + AvgN );
        System.out.println("Blocking Prob = " + Blocked);
        System.out.println("E[T] = " + AvgT);
        System.out.println("Utilization = " + util);
        calcTheory(lamda,delta,mu,K); // function call to calculate theoretical values
      }
}
      public static void calcTheory(double lamda, double delta, double mu ,int K){

        int k = K+1;// number of states
        double p[] = new double[k+1]; //probability of each state

        p[0]= 1; // Inital p0 set to 1 for computaion
        p[1] = ((delta+lamda)/mu)*p[0] ; //calculate p1 using formula
        p[2] = ((delta+lamda)/(2*mu))*p[1]; // calculate p2 using formula

        /*for every state compute p */
        for(int i=3 ;i<k ;++i){
          p[i] = p[i-1]* (lamda/(2*mu));
        }
          double temp = 1;

        for(int i=1; i<k; ++i){
                temp += p[i];
        }

        /* Compute the probability of states*/

          p[0] = 1/temp;
          p[1] = ((delta+lamda)/mu)*p[0] ;
          p[2] = ((delta+lamda)/(2*mu))*p[1];

          for(int i=3 ;i<k ;++i){
                  p[i] = p[i-1]* (lamda/(2*mu));
          }

          double AvgN =0; //theoretical value of average number of components
          double lambdaEff; //theoretical value of lamda effective
          double AvgT =0; //theoretical value of average time spent in system by components
```

```
            double Blocking;// theoretical value of blocking probability
            double lambdaEffective = p[0]*(lamda+delta)+p[1]*(lamda+delta);// compute lamda
Effective

            for(int i=2; i<k ;++i){
                    lambdaEffective += p[i]*(lamda) ;
            }

            for(int i=0; i<=4 ;++i){
            AvgN += i*p[i];// compute theoretical Expected number of customers
            }

            AvgT = AvgN/lambdaEffective; //calculate theoretical value of average time spent in
system by components                                                                    //
Theoretical Time Spent in the system

            Blocking = (lamda*p[K]*AvgT)/ (AvgN+(lamda*p[K]*AvgT));// calculate theoretical
Blocking Probability

            double utilization = 0.5*p[1]; //Initialization of Util with p1

            for(int i=2 ;i<k ;++i){
                    utilization += p[i] ;// calculate utilization
            }

            /*Display the theoretical Values */

            System.out.println("\n Theoretical Calculations : ");
            System.out.println("E[N] = " + AvgN );
            System.out.println("Blocking Prob = " + Blocking);
            System.out.println("E[T] = " + AvgT);
            System.out.println("Utilization = " + utilization);

}
        /*function to generate uniform random variable */
        static double uniformrv(){

    int k = 16807;
    int m = 2147483647;
    seed = ((k*seed) % m);
        double r = seed / m;
        return r;
        }

        /*function to generate exponential random variable */
        static double exponentialRv(double rate)
        {
```

```
            double expRV;
            expRV = ((-1) / rate) * Math.log(uniformrv());
            return(expRV);
        }

        /*function to calculate Utilization*/
        static double calcUtilization(double util,int systemSize ,double sysClock,double
prevClock){
            double Utilization = util ;
            if(systemSize == 1){
                    Utilization += 0.5 * (sysClock-prevClock);
            }
            else if(systemSize >= 2){
                    Utilization += 1 * (sysClock-prevClock);
            }
            return Utilization;
        }
}
```