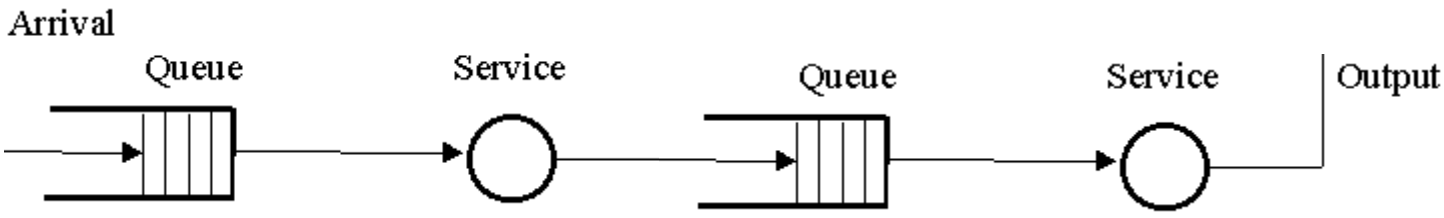
Luis Serra Garcia
997/SE
Full code in the end

QUEUING THEORY

DIGITAL SIMULATION



The system that is going to be simulated in this project consists of 2 concatenated systems. First of all, the customers arrive to the first queue, that has a finite length of K1, following a Poisson process (memoryless) with an arrival rate of λ elements/time unit. As the queue has a finite length, there is a blocking probability when a customer arrives to the system and finds the queue full. The first server has an exponential (memoryless) service time distribution with a service rate of μ1 elements/time unit. This first part of the system is analog to an M/M/1/K system following the Kendall's notation.

The elements exiting the first server enter the queue of a second server which has a length of K2. The output rate of the first server will determine the arrival rate to the second queue so we can't say that is memoryless. There will also be a blocking probability on the second queue. The second server has a μ2 service rate. Finally, after leaving the second server, the customers exit the whole system.

The simulation has been developed in the C++ programming language. The program consist of a class named eventList, where the system events are stored dynamically (this means that the memory is allocated only in the moment of creating the event). This is an event driven simulation and the time axis is divided into variable length intervals, depending on the arrival of customers. The class definition is the following:

```
class eventList{
  private:
    struct eventNode{
      double time;        //defines the time when the event happens
      char type;          //defines the event type: it can be arrival(element arrives to the first queue),
                          midService(element arrives to the second queue) or departure(element exists
                          the system)
      eventNode* next;  //pointer to the next event
      eventNode* prev;  //pointer to the previous event
    };
    eventNode* head;    //pointer to the first event of the list
  public:
```

```
    eventList();            // constructor
    ~eventList();           // destructor
    void clearList();       // function to erase all events from the list
    bool put(double t,char x,double timeC);
    void print();           // function to print the list
    char get(double& timeC);
};
```

The functions put and get are the most important. The function put inserts an event chronologically in the event list. Given the time of the event, the type and the current time of the system, the function inserts the event in the list. The time of the event must be greater that the current time of the system. This function returns true if the event is added or false if it is not.
With the get function, given the current time of the system, it extracts the first element of the list returning the event type. It also changes the current time of the system to the time set in the event that is extracted.

In the main part of the program, the program variables are first declared. After that the program asks the user to set the arrival rate to the system. In the event processing there are 3 separate procedures depending on the type of the event:

- Arrival (element arrives to first queue): if the first queue is already full, it increments the number of elements blocked in the first queue and adds the next arrival event using a random time taken form the exponential distribution of λ. If the firsts server is empty adds a mid service event using μ1 for the time. If the event wasn't blocked, it increases the number of elements in the first part of the system and adds an arrival event. An arrival event will only be added if the customers arrived to the first queue1 are less than 100,000.
- Midservice (element leaves first server and goes to the second queue): first, it decreases the number of elements in the first part of the system and increases the number of customers served by server 1. If the second queue is full, it increases the number of elements blocked in the second queue and, if there are more elements in queue, it adds another midservice event. If the second server is empty, it adds a departure event using μ2 for the time and if there are still elements in the first queue it adds the next midservice event.
- Departure (element leaves second server, leaving the system): first, decreases the number of elements in the second part of the system (queue2+server2) and it increases the total number of customers served by the whole system. If there are still customers in the second queue it adds another departure event using μ2 for the time.

```
case 'A':
    customersArrived++;
    s=s+(N1+N2)*(currentTime-previousTime);
    if (N1>lengthQueue1){
        blockedElements1++;
        if (customersArrived<customersToSimulate){
            x.put(currentTime+distributionA(generator),'A',currentTime);
        }
        break;
    }
    else if (N1==0){
        x.put(currentTime+distributionM(generator),'M',currentTime);
        previousUsageStart1=currentTime;// To calculate server  utilization rate
    }
    N1++;
    if (customersArrived<customersToSimulate){
        x.put(currentTime+distributionA(generator),'A',currentTime);
    }
    break;
case 'M':
```

```
        s=s+(N1+N2)*(currentTime-previousTime);
        a1=a1+(currentTime-previousUsageStart1);
        N1--;
        customersServed1++;
        if(N2>lengthQueue2){
          blockedElements2++;
          if (N1>0){
            x.put(currentTime+distributionM(generator),'M',currentTime);
            previousUsageStart1=currentTime;
          }
          break;
        }
        else if(N2==0){
          x.put(currentTime+distributionD(generator),'D',currentTime);
          previousUsageStart2=currentTime; // To calculate server  utilization rate
        }
        N2++;
        if (N1>0){
          x.put(currentTime+distributionM(generator),'M',currentTime);
          previousUsageStart1=currentTime;
        }
        break;
      case 'D':
        s=s+(N1+N2)*(currentTime-previousTime);
        N2--;
        customersServed2++;
        a2=a2+(currentTime-previousUsageStart2);
        if(N2>0){
          x.put(currentTime+distributionD(generator),'D',currentTime);
          previousUsageStart2=currentTime;
        }
        else if (N1==0 && N2==0 && customersArrived==customersToSimulate)
          isFinished=true;
        break;
```

This program calculates some parameters to determine the efficiency and the behavior of the system. It displays the blocked costumers on the queues, the blocking probabilities, the theoretical blocking probabilities, the throughput rate of the system (mean number of customers served per time unit), the mean sojourn time (mean time that a customer spends in the system), the mean number of elements in the system and the servers utilization rates.

With this we can see that when the arrival rate is inferior to the service rates, the blocking probability is really low however, the server utilization rate is also low. As we increase the arrival rate the utilization rate increases but the blocking probabilities increase too. When the arrival rate equals to the service rate, the theoretical blocking probability is not possible to calculate.

The program prints the results in the screen and also saves it on a txt document with the proper tabulations. It is really easy to put the results in a word table, just copy it from the txt, select it, press ad table and select convert text to table.

λ=0.5

Simulation nº	*Blocked elements in queue 1*	*Blocking Probability Queue 1*	*Theoretical BP Queue 1*	*Blocked elements in queue 2*	*Blocking Probability Queue 2*	*Theoretical BP Queue 2*	*Total Blocking Probability*	*Throughput rate*	*Mean nº of elements in the system*	*Mean Sojourn Time*	*Server 1 utilization rate*	*Server 2 utilization rate*
1	12	0.012%	0.018%	872	0.872%	0.798%	0.884%	0.497 elements/s	1.289	2.594 s	25,078%	49,869%
2	17	0.017%	0.018%	771	0.771%	0.780%	0.788%	0.495 elements/s	1.270	2.564 s	24,996%	49,422%
3	24	0.024%	0.018%	812	0.812%	0.802%	0.836%	0.498 elements/s	1.282	2.575 s	25,226%	49,660%
4	23	0.023%	0.018%	729	0.729%	0.783%	0.752%	0.496 elements/s	1.265	2.551 s	25,012%	49,387%
5	16	0.016%	0.018%	785	0.785%	0.800%	0.801%	0.498 elements/s	1.277	2.565 s	25,159%	49,541%
6	18	0.018%	0.018%	724	0.724%	0.798%	0.742%	0.498 elements/s	1.287	2.586 s	25,256%	49,771%
7	14	0.014%	0.018%	717	0.717%	0.808%	0.731%	0.499 elements/s	1.283	2.572 s	25,046%	49,827%
8	17	0.017%	0.018%	796	0.796%	0.788%	0.813%	0.496 elements/s	1.277	2.573 s	25,097%	49,603%
9	23	0.023%	0.018%	812	0.812%	0.796%	0.835%	0.497 elements/s	1.281	2.577 s	24,988%	49,905%
10	11	0.011%	0.018%	735	0.735%	0.761%	0.746%	0.493 elements/s	1.274	2.585 s	24,870%	49,420%

λ=1

Simulation nº	*Blocked elements in queue 1*	*Blocking Probability Queue 1*	*Theoretical BP Queue 1*	*Blocked elements in queue 2*	*Blocking Probability Queue 2*	*Theoretical BP Queue 2*	*Total Blocking Probability*	*Throughput Rate*	*Mean nº of elements in the system*	*Mean Sojourn Time*	*Server 1 utilization rate*	*Server 2 utilization rate*
1	806	0.806%	0.787%	13228	13,335%	13,723%	14,034%	0.855 elements/s	3.885	4.543 s	49,475%	85,131%
2	816	0.816%	0.787%	13190	13,299%	13,776%	14,006%	0.857 elements/s	3.893	4.544 s	49,346%	85,242%
3	737	0.737%	0.787%	13735	13,837%	14,044%	14,472%	0.857 elements/s	3.955	4.617 s	49,574%	86,003%
4	841	0.841%	0.787%	14073	14,192%	14,113%	14,914%	0.855 elements/s	3.996	4.676 s	50,140%	86,309%
5	834	0.834%	0.787%	13915	14,032%	14,038%	14,749%	0.855 elements/s	3.928	4.596 s	49,806%	85,631%
6	860	0.860%	0.787%	13477	13,594%	13,790%	14,337%	0.854 elements/s	3.913	4.582 s	49,571%	85,342%
7	809	0.809%	0.787%	13439	13,549%	13,905%	14,248%	0.857 elements/s	3.930	4.587 s	49,484%	85,691%
8	819	0.819%	0.787%	13775	13,889%	13,934%	14,594%	0.854 elements/s	3.937	4.610 s	49,469%	85,774%
9	867	0.867%	0.787%	13318	13,434%	13,955%	14,185%	0.859 elements/s	3.921	4.565 s	49,559%	85,631%
10	911	0.911%	0.787%	13867	13,994%	14,074%	14,778%	0.856 elements/s	3.947	4.612 s	49,890%	85,657%

λ=2

Simulation nº	*Blocked elements in queue 1*	*Blocking Probability Queue 1*	*Theoretical BP Queue 1*	*Blocked elements in queue 2*	*Blocking Probability Queue 2*	*Theoretical BP Queue 2*	*Total Blocking Probability*	*Throughput Rate*	*Mean nº of elements in the system*	*Mean Sojourn Time*	*Server 1 utilization rate*	*Server 2 utilization rate*
1	14395	14,395%	inf%	36515	42,655%	42,618%	50,910%	0.983 elements/s	7.839	7.978 s	85,707%	98,727%
2	14062	14,062%	inf %	36759	42,774%	42,942%	50,821%	0.987 elements/s	7.834	7.941 s	85,630%	98,659%
3	14162	14,162%	inf %	36132	42,093%	42,543%	50,294%	0.991 elements/s	7.834	7.907 s	85,840%	98,623%
4	14360	14,360%	inf %	36320	42,410%	42,637%	50,680%	0.987 elements/s	7.832	7.934 s	86,054%	98,614%
5	14128	14,128%	inf %	36544	42,556%	42,725%	50,672%	0.986 elements/s	7.798	7.907 s	85,473%	98,665%
6	14544	14,544%	inf %	36210	42,373%	42,557%	50,754%	0.986 elements/s	7.856	7.966 s	85,951%	98,714%
7	14249	14,249%	inf %	36388	42,434%	42,742%	50,637%	0.989 elements/s	7.822	7.913 s	85,894%	98,685%
8	14546	14,546%	inf %	36511	42,726%	42,684%	51,057%	0.983 elements/s	7.809	7.948 s	85,694%	98,534%
9	14448	14,448%	inf %	36215	42,331%	42,339%	50,663%	0.983 elements/s	7.792	7.928 s	85,575%	98,443%
10	14423	14,423%	inf %	36014	42,084%	42,663%	50,437%	0.993 elements/s	7.808	7.861 s	85,650%	98,489%

FULL CODE

```
#include <iostream>
#include <string.h>
#include <math.h>
#include <random>
#include <time.h>
#include <iomanip>
#include <fstream>
using namespace std;


class eventList{
   private:
      struct eventNode{
         double time;
         char type;
         eventNode* next;
         eventNode* prev;

      };
      eventNode* head;
   public:
      eventList();
      ~eventList();
      void clearList();
      bool put(double t,char x,double timeC);
      void print();
      char get(double& timeC);
};

eventList::eventList(){
   head=NULL;
}

eventList::~eventList(){
   clearList();
}

void eventList::clearList(){
   eventNode* aux=head;

   while (aux!=NULL){
      head=head->next;
      delete aux;
      aux=head;
   }
}
```

```
bool eventList::put(double t,char x,double timeC){
   if(t<=timeC)
      return false;
   eventNode* newNode=new eventNode;
   newNode->time=t;
   newNode->type=x;

   if (head==NULL){
      head=newNode;
      head->next=NULL;
      head->prev=NULL;
      return true;
   }

   if (head->time>t){
      newNode->next=head;
      newNode->prev=NULL;
      head=newNode;
      head->next->prev=head;
      return true;
   }

   eventNode* temp=head;
   while(temp->next!=NULL){
      if(temp->next->time>t){
         newNode->next=temp->next;
         temp->next=newNode;
         newNode->prev=temp;
         newNode->next->prev=newNode;
         return true;
      }
      temp=temp->next;
   }
   temp->next=newNode;
   newNode->next=NULL;
   newNode->prev=temp;
   return true;
}

void eventList::print(){
   if(head==NULL){
      cout<<"The queue is empty\n\n";
      return;
   }
   eventNode* temp=head;

   cout<<"The queue has the following elements: \n\n";

   while(temp!=NULL){
      cout<<"Time="<<temp->time<<"\nEvent type="<<temp->type<<endl<<endl;
      temp=temp->next;
   }
```

```
}

char eventList::get(double& timeC){
   if(head==NULL)
      return '0';
   eventNode* aux=head;
   char aux2=head->type;
   timeC=head->time;
   head=head->next;
   delete aux;
   return aux2;
}


int main(){
   double customersServed1=0;
   double customersServed2=0;
   double currentTime=0;
   int N1=0; //elements in the first part of the system(queue1 + server1)
   int N2=0; //elements in the second part of the system(queue2 + server2)
   double lengthQueue1, lengthQueue2;
   double blockedElements1=0;
   double blockedElements2=0;
   double s = 0.0; // To calculate mean number of elements in the system
   double a1 = 0.0; // To calculate server 1 utilization rate
   double a2 = 0.0; // To calculate server 2 utilization rate
   double previousTime;
   double previousUsageStart1;
   double previousUsageStart2;
   double lambda,mu,mu2,p1,p2;
   eventList x;
   int customersToSimulate=100000;
   int customersArrived=0;
   bool isFinished=false;
   int replications=10;
   int auxReplications=0;
   ofstream myfile;
   myfile.open ("simulationResults.txt");

   do{
      cout<<"Enter the mean arrival rate (in elements/time unit):\n";
      cin>>lambda;
   }while (lambda<0);

   while(auxReplications<replications){
      mu=2;
      mu2=1;
      lengthQueue1=5;
      lengthQueue2=5;
      p1=lambda/mu;
```

```
        std::default_random_engine generator;
        generator.seed(time(NULL)+auxReplications);
        std::exponential_distribution<double> distributionA(lambda);
        std::exponential_distribution<double> distributionM(mu);
        std::exponential_distribution<double> distributionD(mu2);

        x.put(1,'A',currentTime);

        while (!isFinished){
          previousTime=currentTime; //used to calculate the mean number of customers in the
system
          switch(x.get(currentTime)){
          case 'A':
            customersArrived++;
            s=s+(N1+N2)*(currentTime-previousTime);
            if (N1>lengthQueue1){
              blockedElements1++;
              if (customersArrived<customersToSimulate){
                x.put(currentTime+distributionA(generator),'A',currentTime);
              }
              break;
            }
            else if (N1==0){
              x.put(currentTime+distributionM(generator),'M',currentTime);
              previousUsageStart1=currentTime;// To calculate server  utilization rate
            }
            N1++;
            if (customersArrived<customersToSimulate){
              x.put(currentTime+distributionA(generator),'A',currentTime);
            }
            break;
          case 'M':
            s=s+(N1+N2)*(currentTime-previousTime);
            a1=a1+(currentTime-previousUsageStart1);
            N1--;
            customersServed1++;
            if(N2>lengthQueue2){
              blockedElements2++;
              if (N1>0){
                x.put(currentTime+distributionM(generator),'M',currentTime);
                previousUsageStart1=currentTime;
              }
              break;
            }
            else if(N2==0){
              x.put(currentTime+distributionD(generator),'D',currentTime);
              previousUsageStart2=currentTime; // To calculate server  utilization rate
            }
            N2++;
            if (N1>0){
              x.put(currentTime+distributionM(generator),'M',currentTime);
              previousUsageStart1=currentTime;
```

```
            }
            break;
        case 'D':
            s=s+(N1+N2)*(currentTime-previousTime);
            N2--;
            customersServed2++;
            a2=a2+(currentTime-previousUsageStart2);
            if(N2>0){
                x.put(currentTime+distributionD(generator),'D',currentTime);
                previousUsageStart2=currentTime;
            }
            else if (N1==0 && N2==0 && customersArrived==customersToSimulate)
                isFinished=true;
            break;
        }
    }

    p2=(customersServed1/currentTime)/mu2;
    cout<<"Blocked elements in queue1: "<<blockedElements1<<endl;
    double dropRate1=blockedElements1/(customersArrived)*100;
    cout<<"Blocking probability in queue1: "<<dropRate1<<"%\n";
    double TdropRate1=100*((1.001-p1)*pow(p1,lengthQueue1+1))/(1.001-
pow(p1,lengthQueue1+2));
    cout<<"Theoretical blocking probability in queue1: "<<TdropRate1<<"%\n\n";
    cout<<"Blocked elements in queue2: "<<blockedElements2<<endl;
    double dropRate2=blockedElements2/(blockedElements2+customersServed2)*100;
    cout<<"Blocking probability in queue2: "<<dropRate2<<"%\n";
    double TdropRate2=100*((1.001-p2)*pow(p2,lengthQueue2+1))/(1.001-
pow(p2,lengthQueue2+2));
    cout<<"Theoretical blocking probability in queue2: "<<TdropRate2<<"%\n\n";
    double
totalDropRate=(blockedElements1+blockedElements2)/(customersToSimulate)*100;
    cout<<"Total blocking probability: "<<totalDropRate<<"%\n\n";
    double Trate=customersServed2 / currentTime;
    cout<<"Throughput Rate: "<<Trate<<" elements/s\n\n";
    double meanElements=s / currentTime;
    cout<<"Mean number of elements in the system: "<<meanElements<<endl<<endl;
    double sojournT=meanElements/Trate;
    cout<<"Mean Sojourn time: "<<sojournT<<"s\n\n";
    double uRate1=a1/currentTime*100;
    cout<<"Server 1 utilization rate: "<<uRate1<<"%\n\n";
    double uRate2=a2/currentTime*100;
    cout<<"Server 2 utilization rate: "<<uRate2<<"%\n\n";
    if (auxReplications==0){
        myfile<<"Simulation nº\tBlocked elements in queue 1\tBlocking Probability Queue
1\tTheoretical BP Queue 1\tBlocked elements in queue 2\t";
        myfile<<"Blocking Probability Queue 2\tTheoretical BP Queue 2\tTotal Blocking
Probability\tThroughput Rate\t";
        myfile<<"Mean nº of elements in the system\tMean Sojourn Time\tServer 1 utilization
rate\tServer 2 utilization rate\n";
    }
```

```
        myfile<<std::fixed << std::setprecision(3)
<<auxReplications+1<<"\t"<<blockedElements1<<"\t"<<dropRate1<<"%\t"<<TdropRate1<<"%\
t"<<blockedElements2<<"\t"<<dropRate2<<"%\t"<<TdropRate2;
        myfile<<"%\t"<<totalDropRate<<"%\t"<<Trate<<"
elements/s\t"<<meanElements<<"\t"<<sojournT<<" s\t"<<uRate1<<"%\t"<<uRate2<<"%\n";
        N1=0;
        N2=0;
        s=0;
        customersServed1=0;
        customersServed2=0;
        blockedElements1=0;
        blockedElements2=0;
        a1=0;
        a2=0;
        currentTime=0;
        customersArrived=0;
        x.clearList();
        isFinished=false;
        auxReplications++;

    }
    myfile.close();
    return 0;
}
```