**Name:Predise Jessica Bai**

**Matric Number: UG-16-1680**

**400 Level**

**Assignment Question**

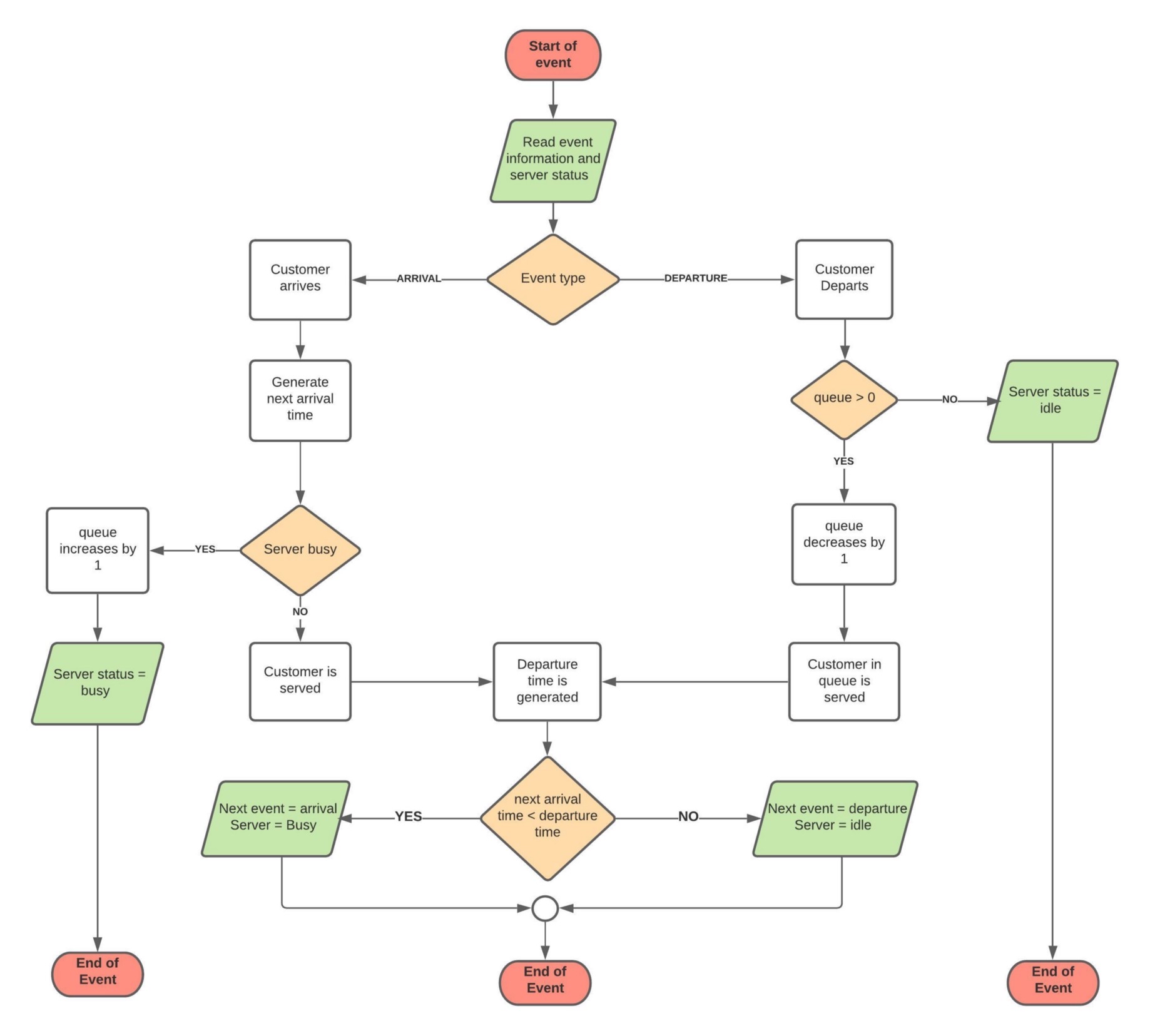
1. Implement of the stimulation model of a single server queue example in c++ or java programming language. Input of the program should be the initial condition.

2. You simply need to write the code for the flow chat.

**ANSWER**

Below is a C++ CODE , A FLOW CHART THE NECESSARY TERMS OBSERVED IN THE SIMULATION PROGRAM INCLUDE; double Clock, MeanInterArrivalTime, MeanServiceTime, SIGMA, LastEventTime, TotalBusy, MaxQueueLength, SumResponseTime; long NumberOfCustomers, QueueLength, NumberInService, TotalCustomers, NumberOfDepartures, LongService; ALL THESE WILL BE THE NECESSARY TOOLS FOR IDENTIFYING A SINGLE SERVER SIMULATION

**FLOW CHART**



**C++ CODE**

|  |  |
| --- | --- |
|  | #include <iostream> |
|  | #include <math.h> |
|  | #include <queue> |
|  |  |
|  | double Clock, MeanInterArrivalTime, MeanServiceTime, SIGMA, LastEventTime, |
|  | TotalBusy, MaxQueueLength, SumResponseTime; |
|  | long NumberOfCustomers, QueueLength, NumberInService, |
|  | TotalCustomers, NumberOfDepartures, LongService; |
|  |  |
|  | double NumNormals; |
|  | double SaveNormal; |
|  |  |
|  | class Event { |
|  | friend bool operator<(const Event& e1, const Event& e2); |
|  | friend bool operator==(const Event& e1, const Event& e2); |
|  |  |
|  | public: |
|  | Event() {}; |
|  | enum EvtType { arrival, departure }; |
|  | Event(EvtType type, double etime) : |
|  | \_type(type), \_etime(etime){} |
|  | EvtType get\_type() { return \_type; } |
|  | double get\_time() { return \_etime; } |
|  | protected: |
|  | EvtType \_type; |
|  | double \_etime; |
|  | }; |
|  |  |
|  | bool operator <(const Event& e1, const Event& e2) { |
|  | return e2.\_etime < e1.\_etime; } |
|  |  |
|  | bool operator ==(const Event& e1, const Event& e2) { |
|  | if (e1.\_etime != e2.\_etime) return false; |
|  | if (e1.\_type == Event::departure) return true; |
|  | return false; |
|  | } |
|  |  |
|  | priority\_queue<Event> FutureEventList; |
|  | queue<Event> Customers; |
|  |  |
|  | double unif() { |
|  | #define RANGE 0x7fffffff |
|  | return (random()/(double)RANGE); |
|  | } |
|  |  |
|  | double expon(double mean) { |
|  | return -mean\*log( unif() ); |
|  | } |
|  |  |
|  | double normal(double mean, double sigma) { |
|  | #define PI 3.1415927 |
|  | double ReturnNormal; |
|  | // should we generate two normals? |
|  | if(NumNormals == 0 ) { |
|  | double r1 = unif(); |
|  | double r2 = unif(); |
|  | ReturnNormal = sqrt(-2\*log(r1))\*cos(2\*PI\*r2); |
|  | SaveNormal = sqrt(-2\*log(r1))\*sin(2\*PI\*r2); |
|  | NumNormals = 1; |
|  | } else { |
|  | NumNormals = 0; |
|  | ReturnNormal = SaveNormal; |
|  | } |
|  | return ReturnNormal\*sigma + mean ; |
|  | } |
|  |  |
|  | void ScheduleDeparture() { |
|  | double ServiceTime; |
|  | // get the job at the head of the queue |
|  | while( (ServiceTime = normal(MeanServiceTime, SIGMA)) < 0 ); |
|  | Event depart = Event(Event::departure, Clock+ServiceTime); |
|  | FutureEventList.push(depart); |
|  | NumberInService = 1; |
|  | QueueLength--; // the one going into service isn't waiting |
|  | } |
|  |  |
|  | void ProcessArrival(Event evt) { |
|  |  |
|  | Customers.push(evt); // push arrival onto the queue |
|  | QueueLength++; // increment number waiting |
|  |  |
|  | // if the server is idle, fetch the event, do statistics, |
|  | // and put into service |
|  | if( NumberInService == 0 ) { |
|  | ScheduleDeparture(); |
|  | } |
|  | else { |
|  | // server is busy |
|  | TotalBusy += (Clock - LastEventTime); |
|  |  |
|  | // adjust max queue length statistics |
|  | if( MaxQueueLength < QueueLength ) { |
|  | MaxQueueLength = QueueLength; |
|  | }} |
|  | // schedule the next arrival |
|  | Event next\_arrival(Event::arrival, |
|  | Clock+expon(MeanInterArrivalTime)); |
|  | FutureEventList.push(next\_arrival); |
|  | LastEventTime = Clock; |
|  | } |
|  |  |
|  | void ProcessDeparture(Event evt) { |
|  | // get the customer description |
|  | Event finished = Customers.front(); |
|  | Customers.pop(); |
|  |  |
|  | // if there are customers in queue then schedule |
|  | // the departure of the next one |
|  | if( QueueLength ) ScheduleDeparture(); |
|  | else NumberInService = 0; |
|  |  |
|  | // measure the response time and add to the sum |
|  | double response = (Clock - finished.get\_time()); |
|  | SumResponseTime += response; |
|  | if( response > 4.0 ) LongService++; // record long service |
|  | TotalBusy += (Clock - LastEventTime); // we were busy |
|  |  |
|  | NumberOfDepartures++; // one more gone |
|  | LastEventTime = Clock; |
|  |  |
|  | } |
|  |  |
|  | void ReportGeneration() { |
|  | double RHO = TotalBusy/Clock; |
|  | double AVGR = SumResponseTime/TotalCustomers; |
|  | double PC4 = ((double)LongService)/TotalCustomers; |
|  |  |
|  | cout << "SINGLE SERVER QUEUE SIMULATION - GROCERY STORE CHECKOUT COUNTER \n" |
|  | << endl; |
|  | cout << "\tMEAN INTERARRIVAL TIME " |
|  | << MeanInterArrivalTime << endl; |
|  | cout << "\tMEAN SERVICE TIME " |
|  | << MeanServiceTime << endl; |
|  | cout << "\tSTANDARD DEVIATION OF SERVICE TIMES " |
|  | << SIGMA << endl; |
|  | cout << "\tNUMBER OF CUSTOMERS SERVED " |
|  | << TotalCustomers << endl << endl ; |
|  |  |
|  | cout << "\tSERVER UTILIZATION " |
|  | << RHO << endl; |
|  | cout << "\tMAXIMUM LINE LENGTH " |
|  | << MaxQueueLength << endl; |
|  | cout << "\tAVERAGE RESPONSE TIME " |
|  | << AVGR << " MINUTES" << endl; |
|  | cout << "\tPROPORTION WHO SPEND FOUR \n" |
|  | << "\t MINUTES OR MORE IN SYSTEM " |
|  | << PC4 << endl; |
|  | cout << "\tSIMULATION RUNLENGTH " |
|  | << Clock << " MINUTES " << endl; |
|  | cout << "\tNUMBER OF DEPARTURES " |
|  | << TotalCustomers << endl; |
|  | } |
|  |  |
|  | void Initialization() { |
|  | // initialize global variables |
|  | Clock = 0.0; |
|  | QueueLength = 0; |
|  | NumberInService = 0; |
|  | LastEventTime = 0.0; |
|  | TotalBusy = 0; |
|  | MaxQueueLength = 0; |
|  | SumResponseTime = 0.0; |
|  | NumberOfDepartures = 0; |
|  | NumNormals = 0; |
|  | LongService = 0; |
|  |  |
|  | // create first arrival event |
|  | Event evt(Event::arrival, expon(MeanInterArrivalTime)); |
|  | FutureEventList.push(evt); |
|  | } |
|  |  |
|  |  |
|  | void main(int argc, char\* argv[]) { |
|  |  |
|  | // assign values to input variables |
|  | MeanInterArrivalTime = 4.5; |
|  | MeanServiceTime = 3.2; |
|  | SIGMA = 0.6; |
|  | TotalCustomers = 1000; |
|  |  |
|  | long seed = atoi(argv[1]); |
|  | srandom(seed); |
|  |  |
|  | // Initialize the simulation |
|  | Initialization(); |
|  |  |
|  | // Loop until first "TotalCustomers" have departed |
|  | while(NumberOfDepartures < TotalCustomers ) { |
|  |  |
|  | // get the next event, and remove from the event list |
|  | Event evt = FutureEventList.top(); |
|  | FutureEventList.pop(); |
|  | Clock = evt.get\_time(); |
|  | if(evt.get\_type() == Event::arrival ) ProcessArrival(evt); |
|  | else ProcessDeparture(evt); |
|  | } |
|  | ReportGeneration(); |
|  | } |