// Train.cpp

#include "TrainCrew.cpp"

class Train

{

private:

float hoursTillArrival;

float waitTimeStart;

float totalTimeInQueue = 0;

float timeInSystem = 0;

int trainNum;

float unloadingTime;

bool startedUnloading = false;

bool hoggedOut = false;

float arrivalTime;

public:

TrainCrew trainCrew;

Train(float arrival, TrainCrew crew) : trainCrew(3)

{

hoursTillArrival = arrival;

trainCrew = crew;

}

void startWaitTime(float currentTime)

{

waitTimeStart = currentTime;

}

float getHoursTillArrival()

{

return hoursTillArrival;

}

float getWaitTimeStart()

{

return waitTimeStart;

}

void updateTotalTimeInQueue(float currentTime)

{

totalTimeInQueue = currentTime - waitTimeStart;

return;

}

float getTimeInQueue()

{

return totalTimeInQueue;

}

void updateTimeInSystem(float currentTime)

{

timeInSystem = currentTime - waitTimeStart;

return;

}

float getTimeInSystem()

{

return timeInSystem;

}

void giveTrainNum(int num)

{

trainNum = num;

}

int getTrainNum()

{

return trainNum;

}

void setUnloadingTime(float time)

{

unloadingTime = time;

}

float getUnloadingTime()

{

return unloadingTime;

}

void trainIsUnloading()

{

startedUnloading = true;

}

bool hasStartedUnloading()

{

return startedUnloading;

}

void trainIsHoggedOut()

{

hoggedOut = true;

}

bool isHoggedOut()

{

return hoggedOut;

}

void setArrivalTime(float now)

{

arrivalTime = now;

}

float getArrivalTime()

{

return arrivalTime;

}

};

struct TrainComp{

bool operator()(Train& a, Train& b)

{

return a.getArrivalTime() > b.getArrivalTime();

}

};

// Train Crew.cpp

class TrainCrew

{

private:

float hoursToWork;

float hoursTillArrival;

float hogOutTime = 0;

public:

TrainCrew(float hours)

{

hoursToWork = hours;

}

float remainingHours()

{

return hoursToWork;

}

void setNewRemainingHours(float time)

{

hoursToWork = time;

}

void callInNewCrew(float hours)

{

hoursTillArrival = hours;

}

float getHoursTillArrival()

{

return hoursTillArrival;

}

void setHogOutTime(float timeOfHogOut)

{

hogOutTime = timeOfHogOut;

}

float getHogOutTime()

{

return hogOutTime;

}

};

// main.cpp

#include <iostream>

#include <ctime>

#include <random>

#include <math.h>

#include <queue>

#include <functional>

#include <list>

#include <algorithm>

#include <map>

#include "train.cpp"

std::mt19937 randomGenerator(time(0));

std::uniform\_real\_distribution<float> trainArrival(0.0f, 1.0f);

std::uniform\_real\_distribution<float> unloadingTime(3.5f, 4.5f);

std::uniform\_real\_distribution<float> remainingWorkTime(6.0f, 11.0f);

std::uniform\_real\_distribution<float> replacementCrewTime(2.5f, 3.5f);

std::uniform\_real\_distribution<float> newCrewWorkTime(8.5f, 9.5f);

typedef void(\*trainEvent )(Train, float);

float now = 0.0;

bool loadingDockBusy = false;

std::map<std::string, float> loadingDockStats;

struct TrainEventComp

{

bool operator()(std::pair<float, std::pair<Train, trainEvent>> a, std::pair<float, std::pair<Train, trainEvent>> b)

{

return a.first > b.first;

}

};

typedef std::priority\_queue<Train, std::vector<Train>, TrainComp> TrainQueue;

typedef std::priority\_queue < std::pair<float, std::pair<Train, trainEvent>>, std::vector<std::pair<float, std::pair<Train, trainEvent>>>,

TrainEventComp> EventList;

TrainQueue trainQueue;

EventList eventList;

float generateTrainArrivalTime(float random, float arrivalRate)

{

return (-arrivalRate)\*log(random);

}

void LeaveStation(Train train, float currentTime)

{

train.updateTimeInSystem(currentTime);

return;

}

void EndUnloading(Train train, float currentTime)

{

eventList.push(std::make\_pair(currentTime, std::make\_pair(train, LeaveStation)));

loadingDockBusy = false;

return;

}

void CrewHogsOut(Train train, float timeLeftToUnload)

{

if (train.hasStartedUnloading())

{

std::cout << "Crew hogged out during unloading; ";

float hoursTillNextCrew = replacementCrewTime(randomGenerator);

loadingDockStats["hogged out"] += hoursTillNextCrew;

eventList.push(std::make\_pair(now + hoursTillNextCrew + train.getUnloadingTime(), std::make\_pair(train, EndUnloading)));

}

else if (!train.hasStartedUnloading() && (trainQueue.top().getTrainNum() != train.getTrainNum()))

{

eventList.push(std::make\_pair(now + train.trainCrew.remainingHours(), std::make\_pair(train, CrewHogsOut)));

std::cout << "Crew hogged out inside queue; ";

}

return;

}

void StartUnloading(Train train, float unloadingTime)

{

train.trainIsUnloading();

// Update crew's remaining hours since train has arrived at station

if (train.trainCrew.getHogOutTime() == 0)

train.trainCrew.setNewRemainingHours(train.trainCrew.remainingHours() - (now - train.getArrivalTime()));

else

train.trainCrew.setNewRemainingHours(train.trainCrew.remainingHours() - (now - train.trainCrew.getHogOutTime()));

std::cout << "crew with " << (train.trainCrew.remainingHours() - (now - train.getArrivalTime())) << "h before hogout; ";

if (train.trainCrew.remainingHours() >= unloadingTime)

eventList.push(std::make\_pair(now + unloadingTime, std::make\_pair(train, EndUnloading)));

else

{

float timeLeftToUnload = train.getUnloadingTime() - train.trainCrew.remainingHours();

train.setUnloadingTime(timeLeftToUnload);

eventList.push(std::make\_pair(now + train.trainCrew.remainingHours(), std::make\_pair(train, CrewHogsOut)));

}

return;

}

void ExitQ(Train train, float currentTime)

{

trainQueue.pop();

train.updateTotalTimeInQueue(currentTime);

eventList.push(std::make\_pair(currentTime, std::make\_pair(train, StartUnloading)));

return;

}

void EnterQ(Train train, float currentTime)

{

train.setArrivalTime(currentTime);

trainQueue.push(train);

if (trainQueue.size() == 1 && loadingDockBusy == false)

{

eventList.push(std::make\_pair(currentTime, std::make\_pair(train, ExitQ)));

return;

}

return;

}

void Train\_Arrival(Train train, float nextTrainComingIn)

{

train.startWaitTime(now);

eventList.push(std::make\_pair(now + train.trainCrew.remainingHours(), std::make\_pair(train, CrewHogsOut)));

eventList.push(std::make\_pair(now, std::make\_pair(train, EnterQ)));

TrainCrew nextTrainCrew(remainingWorkTime(randomGenerator));

Train nextTrain(nextTrainComingIn, nextTrainCrew);

eventList.push(std::make\_pair(now + nextTrain.getHoursTillArrival(), std::make\_pair(nextTrain, Train\_Arrival)));

return;

}

int main(int argc, char\* argv[])

{

float inter\_arrivalTimeRate = atof(argv[1]);

float totalTime = atof(argv[2]);

// Statistical variables

int totalTrainsServed = 0;

int trainNum = -1;

std::list<int> trainsInStation; // Used to check if a train is still in the station, in case it hogs out after it leaves

int trainCurrentlyUnloading = -1;

std::vector<float> overallTimeInSystem;

std::vector<float> overallTimeInQueue;

loadingDockStats["busy"] = 0.0;

loadingDockStats["idle"] = 0.0;

loadingDockStats["hogged out"] = 0.0;

std::vector<float> trainArrivalTimes;

int counter = -1;

int maxQueue = 0;

int totalHogOuts = 0;

TrainCrew train0crew(remainingWorkTime(randomGenerator));

float arrival = generateTrainArrivalTime(trainArrival(randomGenerator), inter\_arrivalTimeRate);

Train train0(arrival, train0crew);

eventList.push(std::make\_pair(now + train0.getHoursTillArrival(), std::make\_pair(train0, Train\_Arrival)));

loadingDockStats["idle"] += arrival;

while (now < totalTime)

{

bool validEvent = true;

std::pair<float, std::pair<Train, trainEvent>> nextEvent = eventList.top();

eventList.pop();

// Double hog out: Old hog out event hasn't started unloading and new hog out has started unloading

if ((nextEvent.second.second == CrewHogsOut) && !(nextEvent.second.first.hasStartedUnloading()) &&

(trainCurrentlyUnloading == nextEvent.second.first.getTrainNum()))

validEvent = false;

// Train hogs out but it has already left the station

if ((!eventList.empty()) && (nextEvent.second.second == CrewHogsOut) &&

(std::find(trainsInStation.begin(), trainsInStation.end(), nextEvent.second.first.getTrainNum()) == trainsInStation.end()))

validEvent = false;

if (validEvent)

{

now += nextEvent.first - now;

std::cout << "Time " << now << ": ";

if (nextEvent.second.second == Train\_Arrival)

{

trainNum++;

nextEvent.second.first.giveTrainNum(trainNum);

trainsInStation.push\_back(nextEvent.second.first.getTrainNum());

std::cout << "Train " << trainNum << " arrival; crew with " << nextEvent.second.first.trainCrew.remainingHours() <<

"h before hogout; ";

float nextTrainArrival = generateTrainArrivalTime(trainArrival(randomGenerator), inter\_arrivalTimeRate);

trainArrivalTimes.push\_back(nextTrainArrival + now);

nextEvent.second.second(nextEvent.second.first, nextTrainArrival);

}

else if (nextEvent.second.second == EnterQ)

{

nextEvent.second.second(nextEvent.second.first, now);

std::cout << "Train " << nextEvent.second.first.getTrainNum() << " entering queue; ";

}

else if (nextEvent.second.second == ExitQ)

{

nextEvent.second.second(nextEvent.second.first, now);

std::cout << "Train " << nextEvent.second.first.getTrainNum() << " leaving the queue; ";

counter++;

}

else if (nextEvent.second.second == StartUnloading)

{

trainCurrentlyUnloading = nextEvent.second.first.getTrainNum();

float unloadingtime = unloadingTime(randomGenerator);

nextEvent.second.first.setUnloadingTime(unloadingtime);

loadingDockStats["busy"] += unloadingtime;

std::cout << "Train " << nextEvent.second.first.getTrainNum() << " entering the dock for " << unloadingtime << "h of unloading; ";

nextEvent.second.second(nextEvent.second.first, unloadingtime);

loadingDockBusy = true;

}

else if (nextEvent.second.second == EndUnloading)

{

nextEvent.second.second(nextEvent.second.first, now);

if (!trainQueue.empty())

{

// If train next in queue is hogged out, wait until new crew arrives before scheduling its exit queue

if (trainQueue.top().isHoggedOut())

{

if ((trainQueue.top().trainCrew.getHoursTillArrival() + now) <= totalTime)

loadingDockStats["idle"] += trainQueue.top().trainCrew.getHoursTillArrival();

eventList.push(std::make\_pair(trainQueue.top().trainCrew.getHogOutTime() + trainQueue.top().trainCrew.getHoursTillArrival(),

std::make\_pair(trainQueue.top(), ExitQ)));

}

else

eventList.push(std::make\_pair(now, std::make\_pair(trainQueue.top(), ExitQ)));

}

std::cout << "Train " << nextEvent.second.first.getTrainNum() << " finishes unloading; ";

}

else if (nextEvent.second.second == LeaveStation)

{

totalTrainsServed++;

nextEvent.second.first.updateTimeInSystem(now);

trainsInStation.remove(nextEvent.second.first.getTrainNum());

overallTimeInSystem.push\_back(nextEvent.second.first.getTimeInSystem());

overallTimeInQueue.push\_back(nextEvent.second.first.getTimeInQueue());

if (trainQueue.empty() && ((now + trainArrivalTimes[counter]) <= totalTime) && !trainArrivalTimes.empty())

{

loadingDockStats["idle"] += trainArrivalTimes[counter];

// counter++;

}

std::cout << "Train " << nextEvent.second.first.getTrainNum() << " departing; ";

}

else if (nextEvent.second.second == CrewHogsOut)

{

totalHogOuts++;

// Train has not started unloading

if (!nextEvent.second.first.hasStartedUnloading())

{

// There is a train ahead unloading and this train is next in queue

if (loadingDockBusy && (trainQueue.top().getTrainNum() == nextEvent.second.first.getTrainNum()))

{

trainQueue.pop();

nextEvent.second.first.trainIsHoggedOut();

nextEvent.second.first.trainCrew.setHogOutTime(now);

nextEvent.second.first.trainCrew.callInNewCrew(replacementCrewTime(randomGenerator));

trainQueue.push(nextEvent.second.first);

}

// Train ahead is unloading but this train is not next in queue

else if (loadingDockBusy && (trainQueue.top().getTrainNum() != nextEvent.second.first.getTrainNum()))

{

// Pop trains in trainQueue until we find the train that hogged out. Update it, then push everything back in

std::vector<Train> differentTrains;

for (int i = 0; i < trainQueue.size(); i++)

{

if (trainQueue.top().getTrainNum() == nextEvent.second.first.getTrainNum())

{

trainQueue.pop();

TrainCrew newcrew(newCrewWorkTime(randomGenerator));

nextEvent.second.first.trainCrew = newcrew;

nextEvent.second.first.trainCrew.setHogOutTime(now);

trainQueue.push(nextEvent.second.first);

}

else

{

differentTrains.push\_back(trainQueue.top());

trainQueue.pop();

}

}

for (int i = 0; i < differentTrains.size(); i++)

trainQueue.push(differentTrains[i]);

}

}

std::cout << "Train " << nextEvent.second.first.getTrainNum() << ": ";

nextEvent.second.second(nextEvent.second.first, now);

}

std::cout << "Q=" << trainQueue.size() << std::endl;

}

// If next event is beyond 7200 hrs, break out of the while loop

if (eventList.top().first >= totalTime)

break;

if (trainQueue.size() > maxQueue)

maxQueue = trainQueue.size();

}

std::cout << std::endl;

std::cout << totalTrainsServed << " trains served" << std::endl;

float max = overallTimeInSystem[0];

float totalTimeInSystem = 0;

float totalTimeInQueue = 0;

for (int i = 0; i < overallTimeInSystem.size(); i++)

{

if (overallTimeInSystem[i] > max)

max = overallTimeInSystem[i];

totalTimeInSystem += overallTimeInSystem[i];

}

float averageTimeInSystem = totalTimeInSystem / overallTimeInSystem.size();

std::cout << "Average time in system: " << averageTimeInSystem << std::endl;

std::cout << "Maximum time in system: " << max << std::endl;

for (int i = 0; i < overallTimeInQueue.size(); i++)

totalTimeInQueue += overallTimeInQueue[i];

float averageTimeInQueue = totalTimeInQueue / overallTimeInQueue.size();

std::cout << "Average time in queue: " << averageTimeInQueue << std::endl;

std::cout << "Max train queue size: " << maxQueue << std::endl;

//std::cout << "Loading dock spent busy: " << (loadingDockStats["busy"] / totalTime) << "%" << std::endl;

//std::cout << "Loading dock spent idle: " << loadingDockStats["idle"] << " hours" << std::endl;

std::cout << "Loading dock spent hogged out: " << (loadingDockStats["hogged out"]/totalTime) << "%" << std::endl;

return 0;

}