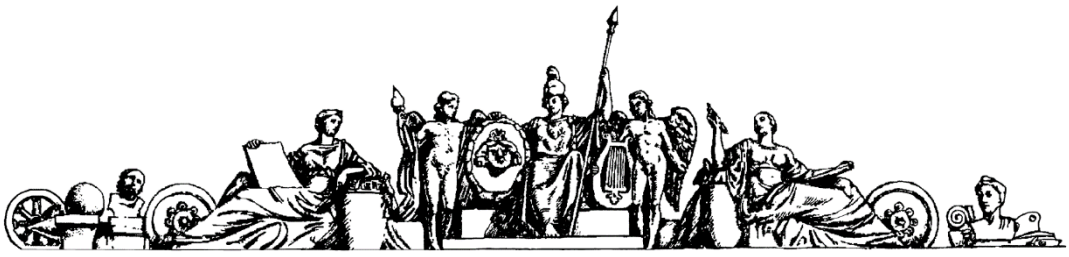

Министерство образования и науки Российской Федерации
Федеральное агентство по образованию
Государственное образовательное учреждение высшего профессионального образования
«Московский государственный технический университет имени Н.Э. Баумана»
(МГТУ им. Н.Э. Баумана)
Факультет «Робототехника и комплексная автоматизация» (РК)
Кафедра «Системы автоматизированного проектирования» (РК6)

Практикум по программированию №3.

Студент: Петраков Станислав

Группа: РК6-36Б

Преподаватель: Берчун Ю.В

Проверил:

Дата:

2020

Задание:

Требуется разработать программу, реализующую дискретно-событийное моделирование системы, рассмотренной в задании 2 домашнего задания №4. Обратите внимание, что все интервалы времени подчиняются законам распределений, носящим непрерывный характер. Поэтому категорически неверными является выбор целочисленных типов данных для моментов и интервалов времени, и тем более инкремент модельного времени с единичным шагом. Нужно реализовать именно переход от события к событию, как это сделано в GPSS и других проблемно-ориентированных системах. Для упрощения можно ограничиться использованием единственного потока случайных чисел для генерации всех необходимых случайных величин. Результатом работы программы должен быть лог-файл, содержащий записи типа: «В момент времени 12.345 транзакт с идентификатором 1 вошёл в модель», «В момент времени 123.456 транзакт с идентификатором 123 встал в очередь 1», «В момент времени 234.567 транзакт с идентификатором 234 занял устройство 2», «В момент времени 345.678 транзакт с идентификатором 345 освободил устройство 1», «В момент времени 456.789 транзакт с идентификатором 456 вышел из модели».

Алгоритм:

Создадим классы Transact, который имитирует транзакт, и State, который имитирует состояния.

В классе Transact хранятся:

- Приватная статическая целочисленная переменная _currID, который отвечает за идентификатор каждого объекта класса. Переменную следует увеличивать каждый раз на единицу как получаем новый номер, чтобы значения не повторялись;
- Приватная целочисленная переменная _ID, содержащая значение каждого экземпляра класса;
- Приватное число _timeNextEvent с плавающей запятой, обозначающее время следующего события;
- Целочисленная переменная _state, которая равна номеру текущего состояния;
- Конструктор Transact, который устанавливает значение через _currID, после увеличивая её на единицу, чтобы у каждого следующего объекта идентификатор отличался;
- Методы getID, getTimeNextEvent и getState, которые возвращают идентификатор, время и номер текущего состояния, и метод setTimeNextEvent и setState, которые устанавливают время и номер текущего состояния;

В классе State хранятся:

- Защищенные перемнные _currID и _ID, которые хранят идентификацию состояний;
- Номер следующего состояния _nextState, в которое должен перейти транзакт по завершению работы в данном состоянии;

- Конструктор, который устанавливает значение идентификатора _currID, и при этом увеличивает его на единицу, чтобы у каждого следующего объекта значение идентификатора отличался;
- Метод getID возвращает идентификатор и setNextState возвращает номер следующего состояния;
- Абстрактный метод useTransact, который должен быть реализован в каждой реализации состояния;

Создадим классы в программе: Queue, Channel, Generate, Terminate - имеющие аналоги в GPSS.

Класс Queue (аналог QUERRY в GPSS) представляет очередь. В классе хранятся:

- Приватный лист _transacts для указателей на объекты класса Transfer;
- Конструктор Queue;
- Публичный метод addTransact, добавляющий указатель на транзакты в список, hasTransact, проверяющий нахождение транзакта в списке, getSize, возвращающий число находящихся в списке указателей на транзакты, ejectTransact, возвращающий первый по списку указатель на транзакт, при этом удалив его из списка;

Класс Channel (аналог OPERATE в GPSS) является наследником State, обрабатывает транзакты. В классе хранятся:

- Приватные целочисленные переменные _MIN и _MAX для генератора случайных чисел;
- Приватная булевая переменная-флаг _isOccupied, значение в котором соответствует тому, обслуживается ли в настоящее время транзакт;
- Приватный объект класса Queue _queue, который представляет собой очередь к опреатору;
- Приватный указатель на транзакт _occupied, который в настоящее время находится в обработке оператором;
- Конструктор Channel, по умолчанию вызывающий конструктор базового класса, который устанавливает значения _MIN и _MAX, флаг _isOccupied равный false и указатель равный NULL;
- Метод isOccupied, который возвращает значение флага _isOccuped и метод getQueueSize, возвращающий текущую длину очереди;
- Метод useTransact, который, в зависимости от текущего состояния оператора (загруженность, наличие очереди, освобождение оператора) совершает действие обработки транзакта;

Класс Generate (аналог GENERATE в GPSS), являющийся публичным наследником класса State. В классе хранятся:

- Приватная численная перемнная с плавающей запятой _time, отвечающая за время появления каждого последующего транзакта, изначально равна 0;
- Приватные целочисленные переменные _MIN и _MAX для генератора случайных чисел.
- Конструктор Generate, вызывающий конструктор базового класса State и устанавливающий значение _time равный нулю;

- Публичный метод getTime возвращает время, к которому прибавляется случайное число в диапазоне от _MIN до _MAX;
- Реализация публичного абстрактного метода useTransact, который меняет текущее состояние у транзакта, который появился, и выводит соответствующее информационное уведомление;

Класс Terminate (аналог TERMINATE в GPSS), являющийся публичным наследником класса State, уничтожает покидающих систему транзактов. В классе хранятся:

- Конструктор Terminate, вызывающий конструктор базового класса;
- Реализация публичного метода useTransact, который очищает память от транзакта через delete;

В функции main объявляем начальную инициализацию для переменных r1, g1, b1, равные количеству шариков определённых цветов из домашнего задания №1, генератор рандомных чисел, файл, в котором будет записан результат работы программы, два объекта класса Channel, один объект классов Generate и Terminate. Создаём список всех указателей на транзакты. Для прохождения по списку нам понадобится итератор, а для завершения работы – флаг endFlag.
Через основной цикл, реализуем систему. Используем while с инверсией значения флага endFlag. В начале цикла определяем наименьшее время среди всех существующих транзактов и запоминаем его. Проходим по списку и сравниваем время транзакта с наименьшим. Если они совпадают, то вызываем switch для значения текущего состояния транзакта: 0 - включаем в систему посредством Generate и резервируем следующий транзакт для системы, при обнаружении завершающего транзакта установливаем флаг endFlag в значение true; 1 - выбираем оператор для последующей обработки, если какой-то из операторов свободен, то транзакт направляется к нему, если оба оператора заняты, то длины очередей операторов сравниваются в поиске наименьшей величины, тогда транзакт отправляется к оператору с наименьшей очередью, если операторы идентичны по занятости, выбирается первый оператор; 2 и 3 – обрабатываем транзакт во 2 и 3 оперетарах; 4 – транзакт закончил свою обработку и его необходимость удалить его из списка транзактов.

Текст программы:

Transact.h

```
#pragma once
#ifndef TRANSACTH

class Transact
{
private:
   static unsigned long long int _currID;
   unsigned long long _ID;
```

```
    double _timeNextEvent;
    unsigned int _state;
public:
    Transact();
    unsigned long long int getID();
    double getTimeNextEvent();
    unsigned int getState();
    void setTimeNextEvent(double input);
    void setState(unsigned int input);
};

#define TRANSACTH
#endif // !TRANSACTH
```

Transact.cpp

```
#include "Transact.h"
unsigned long long int Transact::_currID = 0;

Transact::Transact()
{
    _ID = _currID;
    _currID++;
    _state=0;
    _timeNextEvent = 0;
}

unsigned long long int Transact::getID()
{
    return _ID;
}

double Transact::getTimeNextEvent()
{
    return _timeNextEvent;
}

unsigned int Transact::getState()
{
    return _state;
}

void Transact::setTimeNextEvent(double input)
{
    _timeNextEvent = input;
```

```
}

void Transact::setState(unsigned int input)
{
    _state = input;
}
```

State.h

```
#pragma once
#ifndef STATEH
#define STATEH

#include "Transact.h"
#include <fstream>
class State
{
protected:
    static unsigned long long int _currID;
    unsigned long long int _ID;
    unsigned int _nextState;
public:
    State();
    unsigned long long int getID();
    void setNextState(unsigned int input);
    virtual void useTransact(Transact& transact, std::ofstream& file) = 0;
};

#endif // !STATEH
```

State.cpp

```
#include "State.h"
unsigned long long int State::_currID = 0;
State::State()
{
    _ID = _currID;
    _currID++;
    _nextState = 0;
}

unsigned long long int State::getID()
{
    return _ID;
}
```

```
void State::setNextState(unsigned int input)
{
    _nextState = input;
}
```

Channel.h

```
#pragma once
#ifndef CHANNELH
#define CHANNELH

#include "State.h"
#include "Queue.h"

class Channel : public State
{
private:
    unsigned long long int _MIN;
    unsigned long long int _MAX;
    bool _isOccupied;
    Transact* _occupied;
    Queue _queue;
public:
    Channel(unsigned long long int min, unsigned long long int max);
    int getQueueSize();
    bool isOccupied();
    void useTransact(Transact& transact, std::ofstream& file);

};

#endif // !CHANNELH
```

Channel.cpp

```
#include "Channel.h"

Channel::Channel(unsigned long long int min, unsigned long long int max) : State(),
_queue()
{
    _occupied = NULL;
    _isOccupied = false;
    _MIN = min;
    _MAX = max;
}
```

```
int Channel::getQueueSize()
{
   return _queue.getSize();
}

bool Channel::isOccupied()
{
   return _isOccupied;
}

void Channel::useTransact(Transact& transact, std::ofstream& file)
{
   if (!_isOccupied)
   {
   file << "At " << transact.getTimeNextEvent() << " transact with ID " << transact.getID()
<< " has start  processing at " << _ID <<" operator."<< std::endl;
   transact.setTimeNextEvent(transact.getTimeNextEvent() + (double)(_MIN + ((unsigned
long long int)rand() % ((unsigned long long int)_MAX - 1)) + (double)rand() /
RAND_MAX));
   _occupied = &transact;
   _isOccupied = true;
   }
   else if (&transact == _occupied)
   {
   file << "At " << transact.getTimeNextEvent() << " transact with ID " << transact.getID()
<< " has ended processing at " << _ID <<" operator."<< std::endl;
   transact.setState(_nextState);
   if (_queue.getSize() > 0)
   {
   Transact* next = _queue.ejectTransact();
   file << "At " << next->getTimeNextEvent() << " transact with ID " << next->getID() <<
" has start processing at " << _ID << " operator from queue." << std::endl;
   next->setTimeNextEvent(next->getTimeNextEvent() + (double)(_MIN + ((unsigned
long long int)rand() % ((unsigned long long int)_MAX - 1)) + (double)rand() /
RAND_MAX));
   _occupied = next;
   }
   else
   {
   _isOccupied = false;
   }
   }
   else
   {
```

```
    if (!_queue.hasTransact(transact))
    {
    _queue.addTransact(&transact);
    file << "At " << transact.getTimeNextEvent() << " transact with ID " << transact.getID()
<< " has gone to queue " << _ID <<"."<< std::endl;
    }
    transact.setTimeNextEvent(_occupied->getTimeNextEvent());
    }
}
```

Queue.h

```
#pragma once
#ifndef QUEUEH
#define QUEUEH

#include <list>
#include "Transact.h"
#include <iterator>
class Queue
{
private:
    std::list <Transact*> _transacts;

public:
    Queue();
    int getSize();
    void addTransact(Transact*);
    Transact* ejectTransact();
    bool hasTransact(Transact& transact);

};

#endif // !QUEUEH
```

Queue.cpp

```
#include "Queue.h"

Queue::Queue()
{
}

int Queue::getSize()
{
```

```
    return _transacts.size();
}

void Queue::addTransact(Transact* input)
{
    _transacts.push_back(input);
}

Transact* Queue::ejectTransact()
{
    Transact* ejected = _transacts.front();
    _transacts.pop_front();
    return ejected;
}

bool Queue::hasTransact(Transact& transact)
{
    std::list <Transact*> ::iterator iter = _transacts.begin();
    for (; iter != _transacts.end(); iter++)
    {
    if ((**iter).getID() == transact.getID())
    return true;
    }
    return false;
}
```

Generate.h

```
#pragma once
#ifndef GENERATEH
#define GENERATEH

#include "State.h"
#include <cstdlib>

class Generate : public State
{
private:
    double _time;
    unsigned long long int _MAX;
    unsigned long long int _MIN;
public:
    Generate(unsigned long long int, unsigned long long int);
    double getTime();
    void useTransact(Transact& transact, std::ofstream& file);
```

```
};

#endif // !GENERATEH
```

Generate.cpp

```
#include "Generate.h"

Generate::Generate(unsigned long long int min, unsigned long long int max) :State()
{
   _time = 0;
   _MIN = min;
   _MAX = max;
}

double Generate::getTime()
{
   _time += (double)(_MIN + ((unsigned long long int)rand() % ((unsigned long long
int)_MAX - 1)) + (double)rand() / RAND_MAX);
   return _time;
}

void Generate::useTransact(Transact& transact, std::ofstream& file)
{
   transact.setState(_nextState);
   file << "At " << transact.getTimeNextEvent() << " transact with ID " << transact.getID()
<< " generated." << std::endl;
}
```

Terminate.h

```
#pragma once
#ifndef TERMANATEH
#define TERMANATEH

#include "State.h"
class Terminate : public State
{
public:
   Terminate();
   void useTransact(Transact& transact, std::ofstream& file);
};

#endif // !TERMANATEH
```

Terminate.cpp

```
#include "Terminate.h"

Terminate::Terminate() : State()
{
}

void Terminate::useTransact(Transact& transact, std::ofstream& file)
{
   file << "At " << transact.getTimeNextEvent() << " transact with ID " << transact.getID()
<< " leave." << std::endl;
   delete &transact;
}
```

main.cpp

```
#include <iostream>
#include <fstream>
#include <list>
#include <iterator>

#include "Channel.h"
#include "Generate.h"
#include "Queue.h"
#include "State.h"
#include "Terminate.h"
#include "Transact.h"

using namespace std;

int main()
{
   //Start init
   unsigned long long int r1 = 11;
   unsigned long long int g1 = 10;
   unsigned long long int b1 = 11;
   std::cout << "Enter random seed: ";
   unsigned int rnd=20;
   //std::cin >> rnd;
   std::cout << "Enter time of simulation: ";
   int endTime=60;
```

```
//std::cin >> endTime;
std::ofstream file;
file.open("result.txt");
if (!file.is_open())
{
std::cout << "Cannot use file" << std::endl;
return 1;
}
srand(rnd);

//Create generator transacts
Generate generate(0, r1 + g1 + b1);
generate.setNextState(1);

//Create channels
Channel* channels[2];
channels[0] = new Channel(r1, r1 + g1 + b1);
channels[0]->setNextState(4);
channels[1] = new Channel(g1, r1 + g1 + b1);
channels[1]->setNextState(4);

//Create TERMINATOR
Terminate Terminator;

// Create list of all transacts
std::list <Transact*> transacts;
Transact* end = new Transact();
end->setState(0);
end->setTimeNextEvent(endTime);
transacts.push_front(end);
Transact* start = new Transact();
start->setState(0);
start->setTimeNextEvent(generate.getTime());
transacts.push_front(start);
list <Transact*> ::iterator iter;

//Start main cycle

bool isEnd = false;
while (!isEnd)
{
double minTime = (*transacts.begin())->getTimeNextEvent();
for (iter = transacts.begin(); iter != transacts.end(); ++iter)
{
```

```
    minTime = minTime > (*iter)->getTimeNextEvent() ? (*iter)->getTimeNextEvent() :
minTime;
    }
    for (iter = transacts.begin(); iter != transacts.end(); ++iter)
    {
    if ((*iter)->getTimeNextEvent() == minTime)
    {
    bool flag;
    switch ((*iter)->getState())
    {
    case 0:
    if ((**iter).getID() == (*end).getID())
    {
    isEnd = true;
    }
    else
    {
    generate.useTransact(**iter, file);
    Transact* newTrans = new Transact();
    newTrans->setTimeNextEvent(generate.getTime());
    transacts.push_back(newTrans);
    }
    break;
    case 1:
    flag = false;
    for (int i = 0; i < sizeof(channels) / sizeof(*channels); ++i) {
    if (!(channels[i]->isOccupied())) {
    (*iter)->setState(i + 2);
    channels[i]->useTransact(**iter, file);
    flag = true;
    break;
    }
    }
    if (!flag)
    {
    int min = channels[0]->getQueueSize();
    int minIndex = 0;
    for (int i = 0; i < sizeof(channels) / sizeof(*channels); ++i)
    {
    if (channels[i]->getQueueSize() < min) {
    minIndex = i;
    min = channels[i]->getQueueSize();
    }
    }
    (*iter)->setState(minIndex + 2);
```

```
    channels[minIndex]->useTransact(**iter, file);
    }
    break;
    case 2:
    channels[0]->useTransact(**iter, file);
    break;
    case 3:
    channels[1]->useTransact(**iter, file);
    break;
    case 4:
    Terminator.useTransact(**iter, file);
    iter = transacts.erase(iter);
    break;
    }
    }
    }
    }

    //Close file
    file.close();
    return 0;
}
```

Результат работы:

```
At 10.7959 transact with ID 1 generated.
At 10.7959 transact with ID 1 has start  processing at 1 operator.
At 11.5574 transact with ID 2 generated.
At 11.5574 transact with ID 2 has start  processing at 2 operator.
At 22.5526 transact with ID 1 has ended processing at 1 operator.
At 22.5526 transact with ID 1 leave.
At 34.7391 transact with ID 2 has ended processing at 2 operator.
At 34.7391 transact with ID 2 leave.
At 36.756 transact with ID 3 generated.
At 36.756 transact with ID 3 has start  processing at 1 operator.
At 49.6404 transact with ID 4 generated.
At 49.6404 transact with ID 4 has start  processing at 2 operator.
```