Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»
КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

ОТЧЕТ

по лабораторной работе № 5

по курсу: «Моделирование»

Тема Моделирование работы информационного центра

Студент Якуба Д. В.

Группа ИУ7-73Б

Оценка (баллы) _____

Преподаватель Рудаков И.В.

Москва, 2021

1. Задание

В информационный центр приходят клиенты через интервал времени 10 ± 2 минуты. Если все три имеющихся оператора заняты, клиенту отказывают в обслуживании. Операторы имеют разную производительность и могут обеспечивать обслуживание среднего запроса пользователя за 20 ± 5; 40 ± 10; 40 ± 20. Клиенты стремятся занять свободного оператора с максимальной производительностью. Полученные запросы сдаются в накопитель. Откуда выбираются на обработку. На первый компьютер запросы от 1 и 2-ого операторов, на второй – запросы от 3-его. Время обработки запросов первым и 2-м компьютером равны соответственно 15 и 30 мин. Промоделировать процесс обработки 300 запросов.

2. Теория

2.1 Концептуальная модель системы в терминах СМО

На рисунке 2.1 предоставлена концептуальная модель моделируемой системы в терминах СМО.

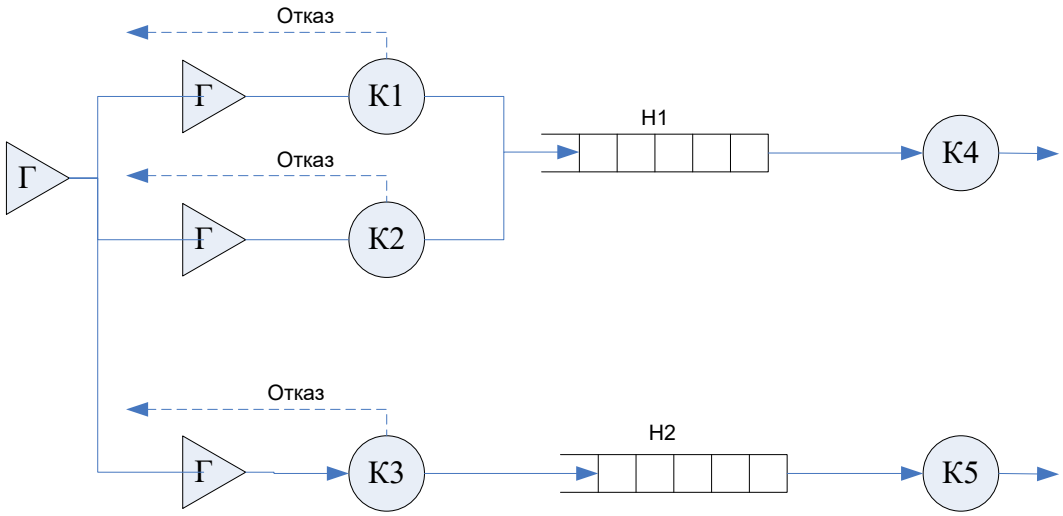

Рис. 2.1, концептуальная модель системы в терминах СМО

В процессе взаимодействия клиентов с информационным центром возможен:

1) Режим нормального обслуживания, т.е. клиент выбирает одного из свободных операторов, отдавая предпочтение тому у которого меньше номер.

2) Режим отказа в обслуживании клиента, когда все операторы заняты

2.2 Переменные и уравнения имитационной модели

Эндогенные переменные: время обработки задания i-ым оператором, время решения этого задания j-ым компьютером.

Экзогенные переменные: число обслуженных клиентов и число клиентов, получивших отказ.

Вероятность отказа в обслуживании клиента:

$$\mathrm{P}_{\text{отказа}} = \frac{C_{\text{отказанных}}}{C_{\text{отказанных}} + C_{\text{обслуженных}}}$$

где $C_{\text{отказанных}}$ – количество заявок, которым было отказано в обслуживании, $C_{\text{обслуженных}}$ – количество заявок, которые были обслужены.

3. Выполнение

Моделирование проводилось с использованием событийного принципа.

Важно отметить, что в силу того, что система подразбита на два «домена», вычисляемая вероятность отказа прямо не зависит от скорости обработки запросов компьютеров, которая позволяют лишь увеличить или уменьшить время моделирования системы, так как конечным условием моделирования является обработка 300 запросов. В качестве количества обслуженных заявок бралась сумма обработанных операторами заявок.

На рисунках 3.1 – 3.2 предоставлены примеры работы реализованного приложения.

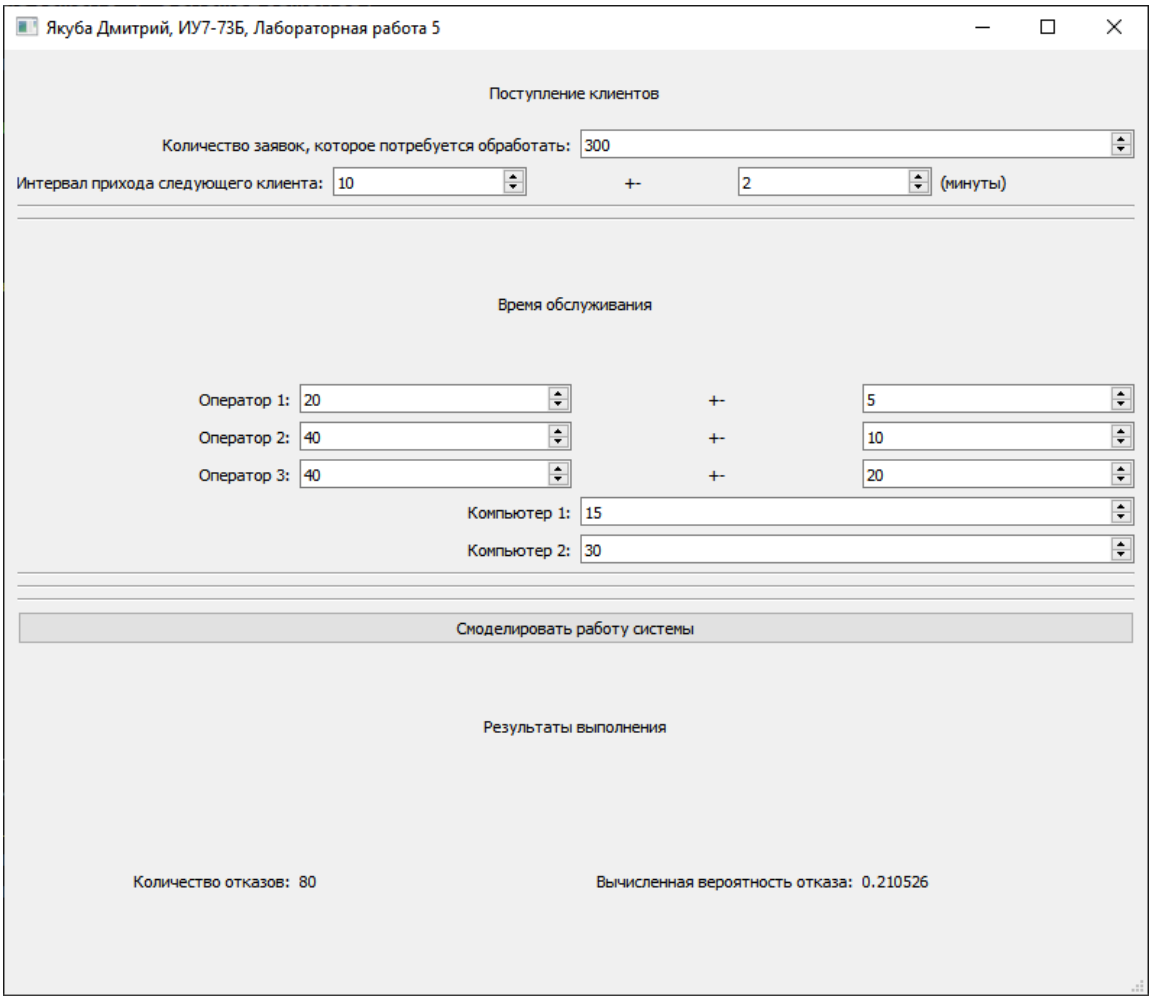

Рис. 3.1, пример работы реализованного приложения

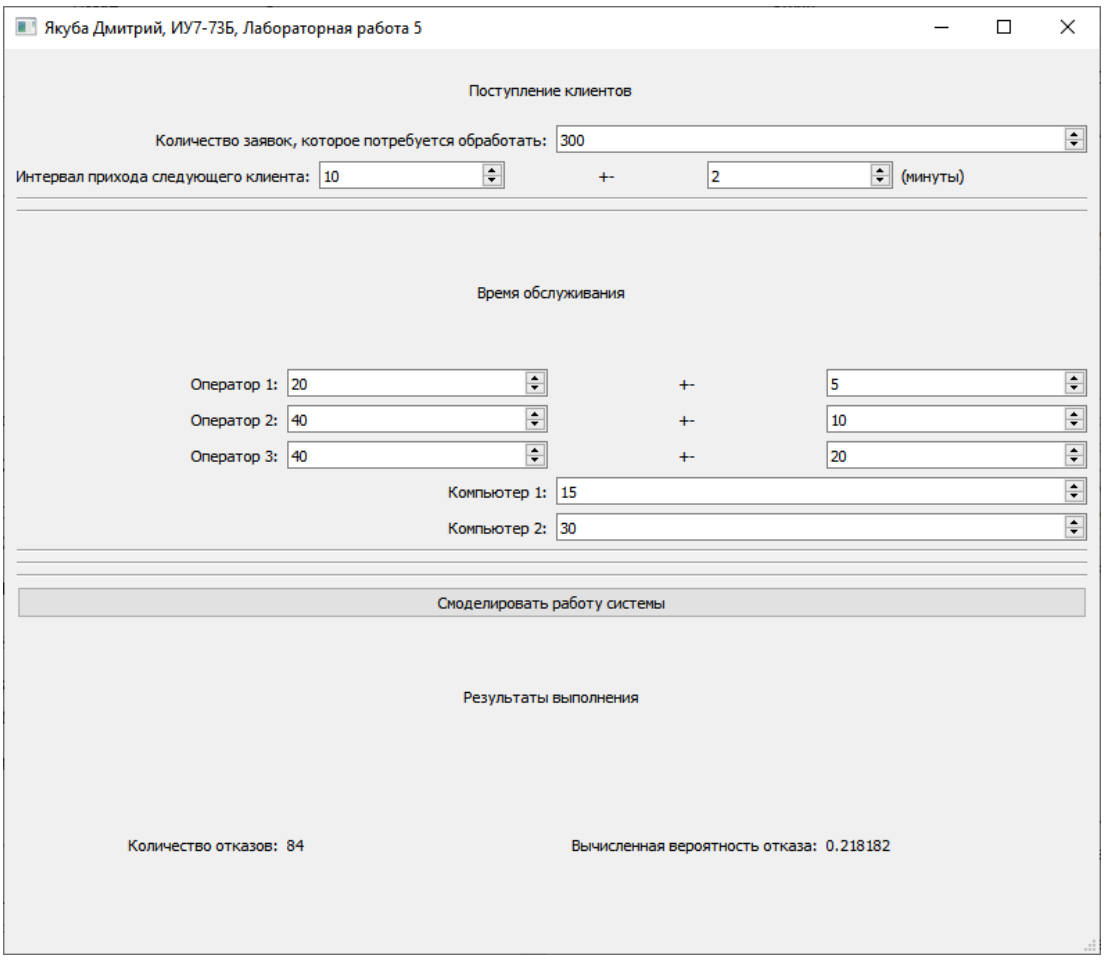

Рис. 3.2, пример работы реализованного приложения

На рисунке 3.3 предоставлено доказательство первого замечания. Как видно, вычисленная вероятность отказа остаётся примерно такой же, как и на предыдущих данных.

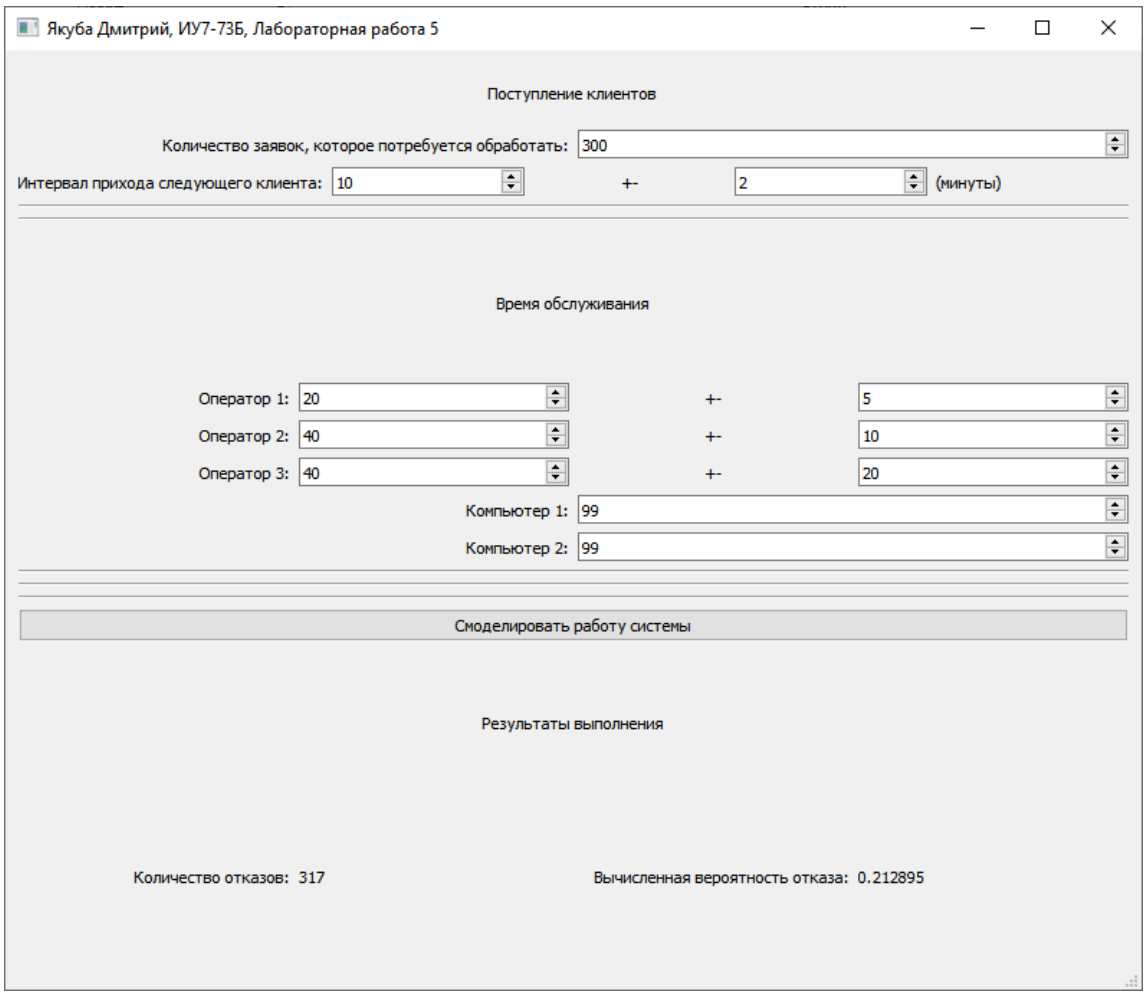

Рис. 3.3, значительное уменьшение скорости обработки запросов компьютерами

4. Листинг

В данном разделе предоставлены используемые для работы приложения классы (используемый ЯП – C++).

```
double currentTime;

class Processor
{
public:
    virtual ~Processor() {}

    virtual bool getRequest() = 0;
};

class RequestGenerator
{
public:
    using Generator = std::function<double()>;

    RequestGenerator(Generator generator_)
    {
        numberOfGeneratedRequests = 0;
        generator = generator_;
```

```
        receivers = std::vector<Processor *>();
        timeOfNextEvent = 0;
    }

    virtual ~RequestGenerator() {}

    void subscribeReceiver(Processor *receiver) {
receivers.push_back(receiver); }

    double getNextTime() { return generator(); }

    Processor *sendRequest()
    {
        numberOfGeneratedRequests++;
        for (auto &&receiver : receivers)
        {
            if (receiver->getRequest())
                return receiver;
        }

        return nullptr;
    }

public:
    int numberOfGeneratedRequests;

private:
    Generator generator;
    std::vector<Processor *> receivers;

public:
    double timeOfNextEvent;
};

class RequestProcessor : public RequestGenerator, public Processor
{
public:
    using Generator = RequestGenerator::Generator;

    RequestProcessor(
        Generator generator, int maxOfQueue_ = 0)
    : RequestGenerator(generator), Processor()
    {
        numberOfRequestsInQueue = 0;
        numberOfProcessedRequests = 0;
        maxOfQueue = maxOfQueue_;
    }

    ~RequestProcessor() override {}

    void process()
    {
        if (numberOfRequestsInQueue > 0)
        {
            numberOfProcessedRequests++;
            numberOfRequestsInQueue--;
            sendRequest();
        }

        timeOfNextEvent = numberOfRequestsInQueue ? currentTime +
getNextTime() : 0.0;
    }
```

```
    bool getRequest() override
    {
        if (maxOfQueue == 0 || numberOfRequestsInQueue < maxOfQueue)
        {
            numberOfRequestsInQueue++;
            if (numberOfRequestsInQueue == 0)
            {
                timeOfNextEvent = currentTime + getNextTime();
            }

            return true;
        }
        else
        {
            return false;
        }
    }

public:
    int numberOfRequestsInQueue;
    int numberOfProcessedRequests;

private:
    int maxOfQueue;
};

Results doSimulate(const SimulationParameters &parameters)
{
    RequestGenerator generatorOfClients([=]() {
        return getUniformReal(parameters.client.timeOfCome -
parameters.client.timeDelta,
            parameters.client.timeOfCome + parameters.client.timeDelta);
    });

    RequestProcessor operator1(
        [=]() {
            return getUniformReal(
                parameters.operator1.timeOfService -
parameters.operator1.timeDelta,
                parameters.operator1.timeOfService +
parameters.operator1.timeDelta);
        },
        1);
    RequestProcessor operator2(
        [=]() {
            return getUniformReal(
                parameters.operator2.timeOfService -
parameters.operator2.timeDelta,
                parameters.operator2.timeOfService +
parameters.operator2.timeDelta);
        },
        1);
    RequestProcessor operator3(
        [=]() {
            return getUniformReal(
                parameters.operator3.timeOfService -
parameters.operator3.timeDelta,
                parameters.operator3.timeOfService +
parameters.operator3.timeDelta);
        },
        1);
```

```
    RequestProcessor computer1([=]() { return
parameters.computer1.timeOfService; });
    RequestProcessor computer2([=]() { return
parameters.computer2.timeOfService; });

    generatorOfClients.subscribeReceiver(&operator1);
    generatorOfClients.subscribeReceiver(&operator2);
    generatorOfClients.subscribeReceiver(&operator3);

    operator1.subscribeReceiver(&computer1);
    operator2.subscribeReceiver(&computer1);
    operator3.subscribeReceiver(&computer2);

    std::array<RequestGenerator *, 6> schemeElements{
        &generatorOfClients, &operator1, &operator2, &operator3, &computer1,
&computer2};

    int numberOfDenials = 0;
    generatorOfClients.timeOfNextEvent = generatorOfClients.getNextTime();

    while (computer1.numberOfProcessedRequests +
computer2.numberOfProcessedRequests <
           parameters.client.amountOfProccessedNeeded)
    {
        qDebug() << "In operator1: " << operator1.numberOfRequestsInQueue;
        qDebug() << "In operator2: " << operator2.numberOfRequestsInQueue;
        qDebug() << "In operator3: " << operator3.numberOfRequestsInQueue;
        qDebug() << "In computer1: " << computer1.numberOfRequestsInQueue
                 << computer1.numberOfProcessedRequests <<
computer1.timeOfNextEvent;
        qDebug() << "In computer2: " << computer2.numberOfRequestsInQueue
                 << computer2.numberOfProcessedRequests <<
computer2.timeOfNextEvent;
        currentTime = generatorOfClients.timeOfNextEvent;
        for (auto &&element : schemeElements)
        {
            if (element->timeOfNextEvent != 0 && element->timeOfNextEvent <
currentTime)
            {
                currentTime = element->timeOfNextEvent;
            }
        }

        for (auto &&element : schemeElements)
        {
            if (currentTime == element->timeOfNextEvent)
            {
                RequestProcessor *processor = dynamic_cast<RequestProcessor
*>(element);
                if (processor)
                {
                    processor->process();
                }
                else
                {
                    Processor *catchProcessor =
generatorOfClients.sendRequest();
                    if (!catchProcessor)
                    {
                        numberOfDenials++;
                    }

                    generatorOfClients.timeOfNextEvent =
```

```
                        currentTime + generatorOfClients.getNextTime();
                }
            }
        }
    }

    return {numberOfDenials,
        static_cast<double>(numberOfDenials) /
            (static_cast<double>(numberOfDenials) +
operator1.numberOfProcessedRequests +
                operator2.numberOfProcessedRequests +
                operator3.numberOfProcessedRequests)};
}
```