Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное   
учреждение высшего профессионального образования

«Нижегородский государственный университет им. Н.И. Лобачевского»

Институт информационных технологий математики и механики

**Отчет по лабораторной работе № 4**

**“Имитационное моделирование системы обслуживания потока заданий на ЭВМ (очереди)”**

**Выполнил**:студент группы 381703-2

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Сизов И.И

Подпись

**Научный руководитель**:

Доцент каф.МОСТ

Кандидат технических наук

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Сысоев А.В

Подпись

**Содержание**

Оглавление

[**Введение** 3](#_Toc531110062)

[**Постановка задачи** 4](#_Toc531110063)

[**Руководство пользователя** 5](#_Toc531110064)

[**Руководство программиста** 7](#_Toc531110065)

[**Описание структуры программы** 9](#_Toc531110066)

[**Описание алгоритмов** 10](#_Toc531110067)

[**Заключение** 12](#_Toc531110068)

[**Литература** 13](#_Toc531110069)

[**Приложение** 14](#_Toc531110070)

# **Введение**

Лабораторная работа направлена на практическое освоение динамической структуры данных Очередь. С этой целью в лабораторной работе изучаются различные варианты структуры хранения очереди и разрабатываются методы и программы решения задач с использованием очередей. В качестве области приложений выбрана тема эффективной организации выполнения потока заданий на вычислительных системах. Очередь характеризуется таким порядком обработки значений, при котором вставка новых элементов производится в конец очереди, а извлечение – из начала. Подобная организация данных широко встречается в различных приложениях. В качестве примера использования очереди предлагается задача разработки системы имитации однопроцессорной ЭВМ. Рассматриваемая в рамках лабораторной работы схема имитации является одной из наиболее простых моделей обслуживания заданий в вычислительной системе и обеспечивает тем самым лишь начальное ознакомление с проблемами моделирования и анализа эффективности функционирования реальных вычислительных систем.

# **Постановка задачи**

Для вычислительной системы (ВС) с одним процессором и однопрограммным

последовательным режимом выполнения поступающих заданий требуется разработать программную систему для имитации процесса обслуживания заданий в ВС. При построении модели функционирования ВС должны учитываться следующие основные моменты обслуживания заданий:

* генерация нового задания;
* постановка задания в очередь для ожидания момента освобождения процессора;
* выборка задания из очереди при освобождении процессора после обслуживания очередного задания.

По результатам проводимых вычислительных экспериментов система имитации должна выводить информацию об условиях проведения эксперимента (интенсивность потока заданий, размер очереди заданий, производительность процессора, число тактов имитации) и полученные в результате имитации показатели функционирования вычислительной системы, в т.ч.

* количество поступивших в ВС заданий;
* количество отказов в обслуживании заданий из-за переполнения очереди;
* среднее количество тактов выполнения заданий;
* количество тактов простоя процессора из-за отсутствия в очереди заданий для

обслуживания.

# **Руководство пользователя**

Данная программа написана в среде разработки Microsoft Visual Studio 2017 на языке С++ и является консольным приложением. При запуске приложения, будет предложено инициализировать параметры процессора, для дальнейших вычислений, см. Рис.1.

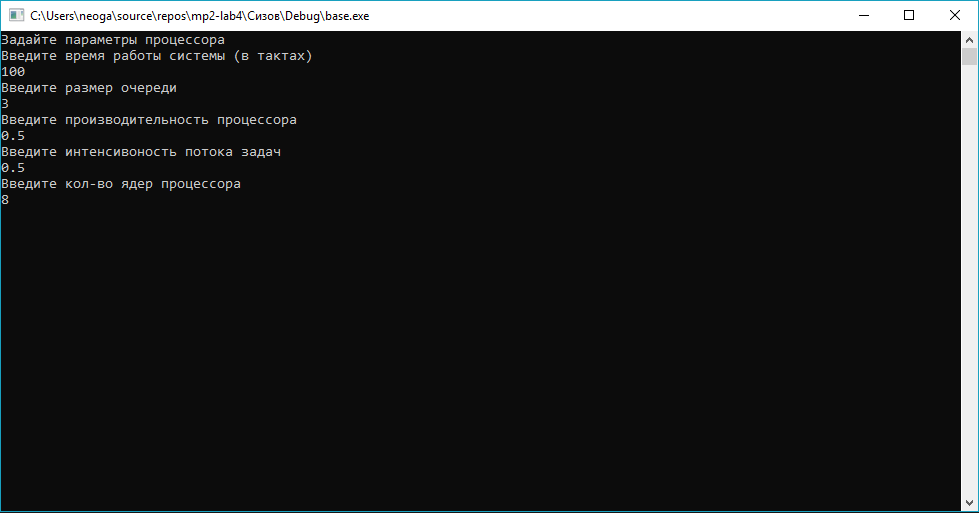


Рис.1. Инициализация переменных.

После инициализации значений произойдет, создание новой переменной класса TProc, в конструктор которой будут переданы наши значения.

Пример: TProc proc(tacts,sq,rate,intens,cores);

На следующем этапе, для запуска имитации работы процессора, нужно использовать метод RunJob().  
Для сбора статистики работы процессора используется отдельные класс названный Result.  
Чтобы вывести статистику работы процессора создайте переменную данного типа, указав в аргумент конструктора используемый процессор, и вызовите метод AllStatus().

Пример: Result r1(proc); r1.AllStatus();

В результате работы программы вычисляются такие значения как:

1. Общее кол-во задач.
2. Кол-во завершенных и незавершенных задач.
3. Среднее время выполнение программы (в тактах).
4. Загруженность ядер в результате выполнения вычислений.
5. Процент отклонённых задач.
6. Процент бездействия процессора.

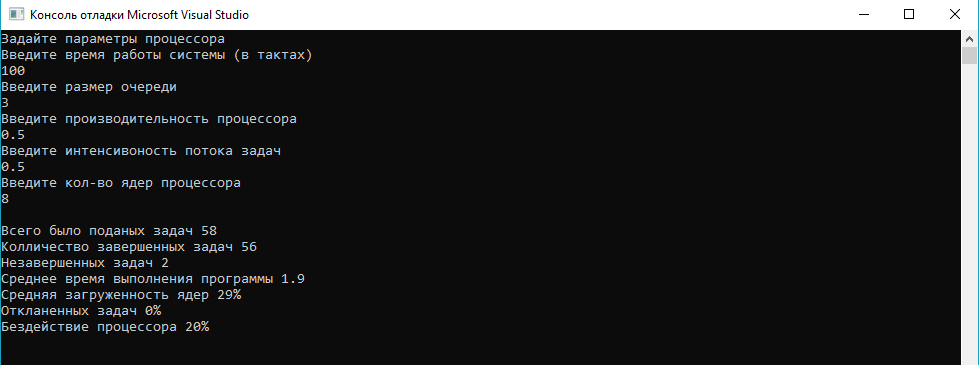


Рис.2. Вывод результата работы процессора.

# **Руководство программиста**

Программа написана при помощи трех классов TQueue, TProc, Result.

Класс TQueue содержит следующие приватные поля:

1. int size – Размер очереди
2. int datacount – Кол-во элементов находящихся в очереди
3. int first – Указатель на первый элемент очереди
4. int last – Указатель на последний элемент очереди
5. type \*pMem -Указатель шаблонного типа, отвечающий за хранение элементов в очереди.

Класс TQueue содержит следующие методы:

1. TQueue(int \_size = 10) – Конструктор класса.
2. TQueue (const TQueue &q) – Конструктор копирования.
3. TQueue<type> &operator=(const TQueue &q) – Оператор присваивания.
4. bool operator == (const TQueue &q)const – Оператор сравнения(равенство).
5. bool operator != (const TQueue &q)const - Оператор сравнения(неравенство).
6. bool IsEmpty() – Метод проверки пустоты очереди.
7. bool IsFull() – Метод проверки полноты очереди.
8. type Get() – Функция выдачи первого элемента из очереди. Вынимает элемент, перемещая указатель first.
9. void Put(type elem) – Функция кладет в конец очереди элемент, перемещая указатель last.
10. type First() – Функция показывает верхний элемент очереди, не вынимая его.
11. type Last() – Метод возвращает последний элемент очереди.
12. int Size() – Возвращает размер очереди.
13. ~TQueue() – Деструктор.
14. friend ostream& operator <<(ostream &os, TQueue &a) – Выписывает очередь в правильном порядке.

Класс TProc содержит следующие приватные поля:

1. unsigned int all\_tacts - Кол-во тактов имитации системы.
2. unsigned int squeue - Очередь задач процессора.
3. double intens - Интенсивность потока задач.
4. double rate - Мощность процессора.
5. unsigned int all\_cores - Всего кол-во ядер процессора.
6. unsigned int free\_cores - Свободных ядер, остающихся в результате работы процессора.
7. int core\_load – Сумма загрузки ядер на каждом такте.
8. int c\_progs – Всего было создано программ.
9. int c\_uncomplete - Число незавершенных программ.
10. int c\_complete - Число завершенных программ.
11. int c\_ignore - Число отклонённых задач.
12. int c\_downtime – Сумма тактов простоя процессора.
13. int c\_execution - Все время выполнения всех программ.
14. vector<Task> jobs – Массив ,хранящий выполняемые задачи.
15. Task GetNewTask() – Метод генерации нового задания.
16. double Random(int min,int max) – Вспомогательный метод генерирующий случайное число в заданном диапазоне.
17. bool IsTask() – Метод проверки генерации задания(нужно ли создавать задание или нет).
18. bool IsComplete () - Метод проверки задания (выполнилось или нет).

В классе TProc содержатся три метода:

1. TProc() – Конструктор класса по умолчанию
2. TProc(unsigned int \_tacts, unsigned int \_squeue, double \_intens, double \_rate,int \_cores) – Конструктор инициализатор
3. void RunJob() – Метод запуска работы процессора, в котором происходит подсчет некоторых величин.

Класс Result является вспомогательным и хранит в себе следующие поля и методы:

TProc proc – Переменная хранящая процессора.

Result(TProc c1) :proc(c1) – Конструктор класса инициализирующий процессор.

unsigned int GetAllTacts() – Возвращает кол-во тактов работы системы

int GetPrograms() – Возвращает кол-во всех созданных задач.

int GetComplited() – Возвращает число завершённых задач.

int GetUnfinish() - Возвращает число незавершённых задач.

int GetIgnored() - Возвращает число отклонённых задач.

int GetDowntime() – Возвращает число тактов простоя.

int GetTimeExecution() – Возвращает общее число тактов работы всех программ.

int GetCoreLoad() – Возвращает общее число загрузки всех ядер

void AllStatus() – Выводит полную информацию работы процессора.

## **Описание структуры программы**

Программа содержит три основных проекта:

1. base,
2. base\_test,
3. gtest.

Проект base содержит в себе модули Queue.h, TaskFlow.h, Result.h. Класс TQueue объявлен в заголовочном файле и не требует дополнительных файлов с расширением cpp, так как является шаблонным. Модули TaskFlow.h, Result.h содержат в себе объявление классов TProc и Result соответственно. Реализация методов классов содержатся в отдельных файлах source именуемые taskflow.cpp и result.cpp.

Проект base\_test содержит набор необходимых тестов Google Test, проверяющих правильность реализации основных классов. Тесты написаны только для класса TQueue.

Проект gtest содержит необходимую структуру для работы тестов Google Test.

В целом, программа построена на использовании интуитивно понятного пользователю меню.

## **Описание алгоритмов**

**Опишем алгоритмы класса TQueue**

1. TQueue(int \_size = 10) – Конструктор инициализатор, с начальным значением размерности 10. Выделяем память для хранения элементов очереди, так же присваивает некоторые значения по умолчанию. Выдаст исключение если значение size будет меньше нуля или больше некоторой константы const int MAXQUEUE = 100.
2. bool IsEmpty() – Метод проверки пустоты очереди. Если значение переменной datacount будет равно нулю, значит очередь пуста, в этом случае вернет true, в ином случае false.
3. bool IsFull() – Метод проверки полноты очереди. Если значение переменной datacount будет равно переменной size, значит очередь полная, вернет в таком случае true, иначе вернет false.
4. type Get() – Метод возвращает первый элемент очереди, перемещая указатель first вперед на единицу. Данный класс реализован в виде кольцевого буфера, следовательно, указатель first должен высчитываться как first = first % size.
5. void Put(type elem) – Функция кладет в конец очереди элемент, двигая указатель last. Так как очередь реализована в виде кольцевого буфера, то last=(++last)%size.
6. type First() – Вернет первый элемент очереди, не двигая указателей. Данная функция используется только для просмотра элемента, без его “удаления”.
7. type Last() – Возвращает последний элемент из очереди.

**Опишем алгоритмы класса TProc**

1. double Random(int min,int max) – Функция создает случайное число в заданном диапазоне, используется в методах GetNewTask(), IsTask() и IsComplete().
2. Task GetNewTask() – Метод создания нового задания. Структура Task содержит в себе переменные ticks и cores. Значение cores генерируется случайно, а значение ticks задается по умолчанию как 1.
3. bool IsTask() – Функция проверяет нужно ли создавать новую задачу для процессора. Если случайное сгенерированное число будет меньше параметра intens то функция вернет значение true, в противном случае false.
4. bool IsComplete() – Проверка на выполнение задания. Если некоторое случайное число будет меньше, чем величина rate, тогда задание является выполненным, метод возвращает true. Иначе вернет false.
5. void RunJob() – Основной метод в данной программе, запускающий инициализацию самого процессора. В ходе выполнения метода происходит главный цикл, кол-во итераций которого равно значению переменной all\_tacts. Создается очередь типа Task для хранения задач. Происходит проверка на создание новой задачи. Если новая задача была создана, она заносится в очередь, если она не полная. Если очередь полная, программа записывается как отклонённая. Далее задается новый цикл, который проходит по всем задачам, которые находятся уже внутри процессора. Если какая-то из ранее запущенных задач была завершена, она помечается как завершенная, в процессоре освобождаются ядра. Если программа не завершена, ее время работы инкрементируется. После прохода по всем задачам процессора, происходит занесение новых задач, лежащих внутри очереди. Если в конце всех действий, описанных выше, процессор пустой, значит произошел такт простоя.

# **Заключение**

Данная лабораторная работа познакомила нас с эффективным методом хранения информации. Также программа позволяет имитировать работу центрального процессора с достаточно близкой точностью. Использование имитационного моделирования позволяет проводить изучение исследуемых объектов и явлений без проведения реальных (натурных) экспериментов.

# **Литература**

1. Столлингс, В. Структурная организация и архитектура компьютерных систем, 5-е изд.: Пер. с англ. — М.: Издательский дом «Вильямс», 2002. — 896 с.: ил. — Парал. тит. англ.
2. Брайан Керниган, Деннис Ритчи «Язык программирования Си».
3. Герберт Шилдт - Полный справочник по C

# **Приложение**

**Модуль Queue.h**

#include <iostream>

#include <string>

using namespace std;

const int MAXQUEUE = 100;

template <class type>

class TQueue

{

private:

int size;

int datacount;

int first;

int last;

type \*pMem;

public:

TQueue(int \_size = 10)

{

if (\_size > 0 && \_size <= MAXQUEUE)

{

size = \_size;

pMem = new type[size];

datacount = 0;

first = 0;

last = -1;

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

pMem[i] = {};

}

else throw "error of size";

}

TQueue (const TQueue &q)

{

size = q.size;

pMem = new type[size];

datacount = q.datacount;

first = q.first;

last = q.last;

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

pMem[i] = q.pMem[i];

}

TQueue<type> &operator=(const TQueue &q)

{

if (this != &q)

{

if (size != q.size)

{

size = q.size;

pMem = new type[size];

}

datacount = q.datacount;

first = q.first;

last = q.last;

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

pMem[i] = q.pMem[i];

}

else return \*this;

}

bool operator == (const TQueue &q)const

{

if (size != q.size)

return false;

else

{

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

{

if (pMem[i] != q.pMem[i])

return false;

}

return true;

}

}

bool operator != (const TQueue &q)const

{

return!(\*this == q);

}

bool IsEmpty()

{

if (datacount == 0)

return true;

else return false;

}

bool IsFull()

{

if (datacount == size)

return true;

else return false;

}

type Get()

{

if (IsEmpty())

throw "Queue is empty";

first = first % size;

type re = pMem[first];

pMem[first++] = {};

datacount--;

return re;

}

void Put(type elem)

{

if (IsFull())

throw"Queue is full";

datacount++;

last=(++last)%size;

pMem[last] = elem;

}

type First()

{

int t\_first=first;

t\_first %= size;

return pMem[t\_first];

}

type Last()

{

return pMem[last];

}

~TQueue()

{

delete[] pMem;

}

int Size()

{

return datacount;

}

friend ostream& operator <<(ostream &os, TQueue &a)//выводит очередь в правильно порядке

{

for (int i = 0; i <a.datacount; i++)

{

a.first = a.first % a.size;

os << a.GetFirst()<<" ";

a.first++;

}

return os;

}

};

**Модуль TaskFlow.h**

#include "Queue.h"

#include "time.h"

#include <cstdlib>

#include <random>

#include <chrono>

#include <vector>

struct Task

{

int ticks;

int cores;

};

class TProc

{

private:

unsigned int all\_tacts;//кол-во тактов имитации

unsigned int squeue;//очередь процессора

double intens;//интенсивность потока задач

double rate;//мощность процессора

unsigned int all\_cores;//всего кол-во ядер

unsigned int free\_cores;//свободных ядер

int core\_load;

int c\_progs;// всего программ

int c\_uncomplete;//число незавершенных задач

int c\_complete;//завершенных програм

int c\_ignore;//число откланенных задач

int c\_downtime;//такты простоя

int c\_execution;//все время выполнения всех программ

vector<Task> jobs;

Task GetNewTask();

double Random(int min,int max);//рандом от 0 до 1 (неплохой рандом)

bool IsTask();

bool IsComplete();

public:

TProc();

TProc(unsigned int \_tacts, unsigned int \_squeue, double \_intens, double \_rate,int \_cores);

void RunJob();

friend class Result;

};

**Модуль taskflow.cpp**

#include "TaskFlow.h"

#include <iostream>

double TProc::Random(int min, int max)

{

std::mt19937\_64 rng;

uint64\_t timeSeed = std::chrono::high\_resolution\_clock::now().time\_since\_epoch().count();

std::seed\_seq ss{ uint32\_t(timeSeed & 0xffffffff), uint32\_t(timeSeed >> 32) };

rng.seed(ss);

std::uniform\_real\_distribution<double> unif(min, max);

return(unif(rng));

}

TProc::TProc() : c\_progs(0), c\_uncomplete(0), c\_complete(0), c\_ignore(0), c\_downtime(0), c\_execution(0),core\_load(0)

{

all\_tacts = 100;

squeue = 3;

intens = 0.5;

rate = 0.5;

all\_cores = 8;

free\_cores = 8;

}

TProc::TProc(unsigned int \_tacts, unsigned int \_squeue, double \_intens, double \_rate, int \_cores) : c\_progs(0), c\_uncomplete(0), c\_complete(0), c\_ignore(0), c\_downtime(0), c\_execution(0),core\_load(0)

{

if (\_squeue > MAXQUEUE)

throw;

if (\_intens > 1 || \_intens <= 0 || \_rate > 1 || \_rate <= 0)

throw;

all\_tacts = \_tacts;

squeue = \_squeue;

intens = \_intens;

rate = \_rate;

all\_cores = \_cores;

free\_cores = \_cores;

}

Task TProc::GetNewTask()

{

Task tmp;

tmp.cores = Random(1 , 4);

tmp.ticks = 1;

return tmp;

}

bool TProc::IsTask()

{

double q1 = Random(0, 1);//q1 в методичке

if (q1 <= intens) //если меньше значит генерируем новое задание

return true;

else

return false;

}

bool TProc::IsComplete()

{

double q2 = Random(0, 1);//q2 в методичке

if (q2 <= rate) //значит завершено задание

return true;

else

return false;

}

void TProc::RunJob()

{

TQueue <Task> queueTasks(squeue);

for (int i = 0; i < all\_tacts; i++)

{

if (IsTask())

{

c\_progs++;

Task tmp = GetNewTask();

if (tmp.cores > all\_cores || queueTasks.IsFull())//программа запросила больше ядер чем вообще существует у процессора

c\_ignore++;

else if (!queueTasks.IsFull())

queueTasks.Put(tmp);

}

for (int j = 0; j < jobs.size(); j++)//смотрим какие программы завершились в процессоре

{

if (IsComplete())

{

free\_cores += jobs[j].cores;

c\_execution += jobs[j].ticks;

c\_complete++;

jobs.erase(jobs.begin() + j);

j--;

}

else

jobs[j].ticks++;

}

while (queueTasks.First().cores <= free\_cores && !queueTasks.IsEmpty())//если возможно то записываем новую программу на этом такте из очереди

{

free\_cores -= queueTasks.First().cores;

jobs.push\_back(queueTasks.Get());

}

core\_load = core\_load + (all\_cores - free\_cores);

if (jobs.empty())

c\_downtime++;

}

c\_uncomplete = jobs.size() + queueTasks.Size();

}

**Модуль Result.h**

#include "TaskFlow.h"

class Result

{

TProc proc;

public:

Result(TProc c1) :proc(c1) {}

unsigned int GetAllTacts();

int GetPrograms();

int GetComplited();

int GetUnfinish();

int GetIgnored();

int GetDowntime();

int GetTimeExecution();

int GetCoreLoad();

void AllStatus();

};

**Модуль result.cpp**

#include "Result.h"

unsigned int Result :: GetAllTacts()

{

return proc.all\_tacts;

}

int Result::GetPrograms()

{

if (proc.c\_progs == 0)//чтобы не было деления на ноль

return 1;

return proc.c\_progs;

}

int Result::GetComplited()

{

if (proc.c\_complete == 0)//аналогично

return 1;

return proc.c\_complete;

}

int Result::GetUnfinish()

{

return proc.c\_uncomplete;

}

int Result::GetIgnored()

{

return proc.c\_ignore;

}

int Result::GetDowntime()

{

return proc.c\_downtime;

}

int Result::GetTimeExecution()

{

return proc.c\_execution;

}

int Result::GetCoreLoad()

{

return proc.core\_load;

}

void Result::AllStatus()

{

cout.precision(2);

cout << "Всего было поданых задач " << GetPrograms() << endl;

cout << "Колличество завершенных задач " << GetComplited() << endl;

cout << "Незавершенных задач " << GetUnfinish()<< endl;

cout << "Среднее время выполнения программы " << double(GetTimeExecution()) / GetComplited()<< endl;

cout << "Средняя загруженность ядер " << double(GetCoreLoad())\*100/ (GetAllTacts()\*proc.all\_cores)<< "%" << endl;

cout << "Откланенных задач " << (double(GetIgnored()) / GetPrograms())\*100<< "%" << endl;

cout << "Бездействие процессора " << (double(GetDowntime())/GetAllTacts())\*100<< "%" << endl;

cout << endl;

}