Министерство науки и высшего образования РФ

Федеральное государственное автономное 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

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Николаев Д.Э

Подпись

**Научный руководитель**:

Доцент каф.МОСТ

Кандидат технических наук

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

Подпись

**Содержание**

[**Введение** 3](#_gjdgxs)

[**Постановка задачи** 4](#_30j0zll)

[**Руководство пользователя** 5](#_3znysh7)

[**Руководство программиста** 7](#_2et92p0)

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

[**Описание алгоритмов** 10](#_3dy6vkm)

[**Заключение** 12](#_1t3h5sf)

[**Литература** 1](#_4d34og8)2

[**Приложение** 1](#_2s8eyo1)3

# **Введение**

В сфере IT индустрии используются различные структуры и системы хранения данных для реализации поставленных задач. Одной из таких структур данных является Очередь. В качестве лабораторной работы выбрана простая модель многопоточной ЭВМ для имитации формирования и управления списком задач. Данная работа позволяет ознакомиться с одним из простейших способов управления данными высоко вычислительных систем.

**Постановка задачи**

В рамках данной работы требуется разработать реализацию, позволяющую построить список и распределить задачи, поступающие в вычислительную систему (далее ВС) с одним процессором и многопоточным последовательным режимом выполнения.

При построении модели функционирования ВС должны учитываться следующие основные моменты обслуживания заданий:

1. создание нового задания;
2. постановка задания в очередь для ожидания момента освобождения процессора;
3. 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.

1. количество поступивших в ВС заданий;
2. количество отказов в обслуживании заданий из-за переполнения очереди;
3. среднее количество тактов выполнения заданий;
4. 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.

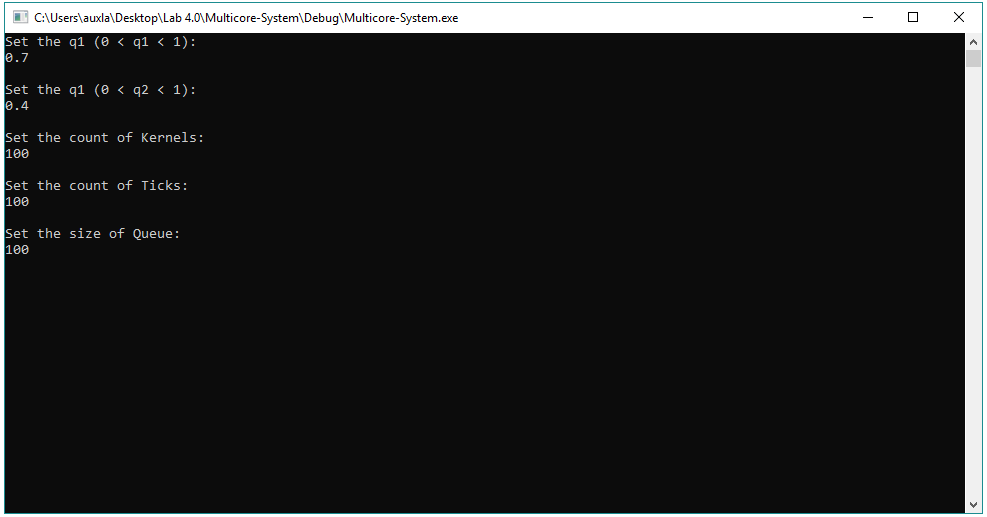


Рис.1. Инициализация

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

Пример: TaskManager manager(\_qTask, \_qKernel, \_kerCount, \_countTact, \_queueSize);

На следующем этапе, для запуска имитации работы процессора, нужно использовать метод StartManage().  
По окончанию работы метод выдаст статистический отчет о проделанной работе.

В результате работы программы вычисляются такие значения как:

1. Количество ядер системы;
2. Количество тактов;
3. Общее количество задач.
4. Количество отказов (задач, не попавших в очередь из-за переполнения);
5. Количество незавершенных задач на момент остановки программы;
6. Количество завершенных задач;

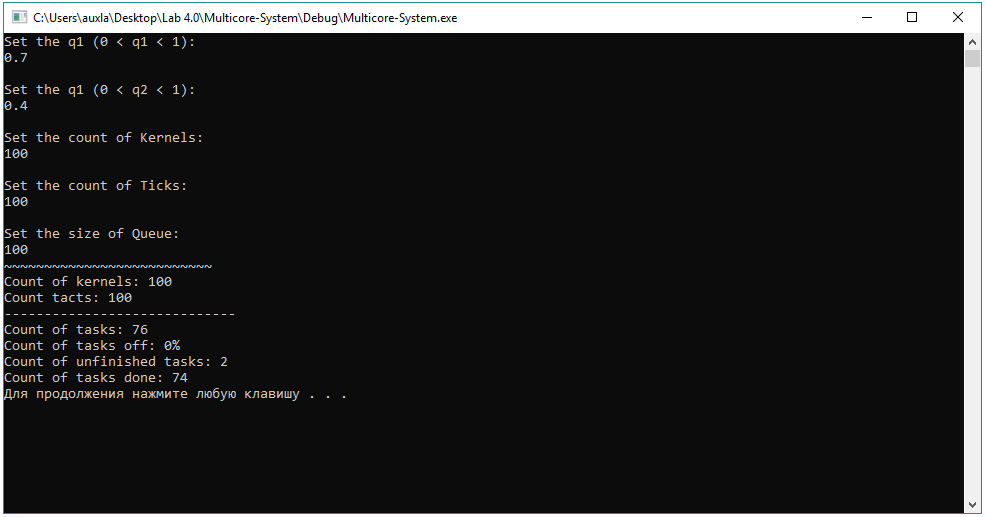


Рис.2. Вывод результата работы менеджера задач

# **Руководство программиста**

Программа написана при помощи трех классов TQueue, TaskManager, Kernel.

Шаблонный класс TQueue содержит следующие приватные поля:

1. Type\* pMem - указатель на динамический массив данных типа Type
2. size\_t size - количество элементов в очереди
3. int count - количество заполненных ячеек очереди
4. size\_t setPointer - индекс элемента для записи
5. size\_t getPointer - индекс элемента для чтения

Класс TQueue содержит следующие методы:

1. TQueue(int) - конструктор класса
2. void setSize(int) - устанавливает размер очереди
3. void Put(Type&) - добавляет элемент типа Type в очередь
4. Type Get() - возвращает элемент типа Type и удаляет его из очереди
5. bool isEmpty() - возвращает истину, если очередь пуста, иначе ложь
6. bool isFull() - возвращает истину, если очередь полна, иначе ложь
7. ~TQueue() - деструктор класса

Класс TaskManager содержит следующие приватные поля:

1. double q - частота генерации новых заданий
2. TQueue<size\_t> TaskList - очередь для задач
3. Kernel\* kernelArray - массив ядер для вычислений
4. size\_t kernelCount - количество ядер
5. size\_t startCountTacts - Общее количество тактов
6. size\_t countTacts - количество совершенных тактов
7. size\_t taskIndex - индекс последней сгенерированной задачи
8. size\_t taskOff - количество отказов от выполнения задач
9. size\_t taskDone - количество успешно выполненных задач

В классе TProc содержатся три метода:

1. TaskManager(double, double, size\_t, size\_t, size\_t) - конструктор класса TaskManager
2. void StartManage() - метод для запуска управления задачами

Класс Kernel является вспомогательным и хранит в себе следующие поля:

1. double q - частота скорости выполнения задач
2. bool ready - статус готовности последней назначенной задачи
3. size\_t taskCounter - количество задач, выполненных ядром

Класс Kernel содержит в себе следующие методы:

1. Kernel() - конструктор класса
2. void setParams(double) - метод для установки частоты ядра
3. void addTask(unsigned int) - метод для назначения новой задачи
4. unsigned int getTaskCount() - возвращает количество выполненных задач
5. bool isReady() - статус готовности ядра к новым задачам
6. bool isDone() - статус готовности последней назначенной задачи

## 

## **Описание алгоритмов**

**Опишем алгоритмы класса TQueue**

1. TQueue(int) – Конструктор инициализатор, с начальным значением размерности 10. Выделяем память для хранения элементов очереди, также присваивает некоторые значения по умолчанию.
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 count 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 барабанного буфера, следовательно, указатель getPointer должен высчитываться как getPointer = ++getPointer % size.
5. void Put(Type&) – Функция кладет в конец очереди элемент, двигая указатель last. Так как очередь реализована в виде барабанного буфера, то setPointer = ++setPointer % size;.

**Опишем алгоритмы класса TaskManager**

1. TaskManager  
    (  
    double \_qTask,   
    double \_\_qKernel,  
    size\_t \_kerCount,  
    size\_t \_countTact,  
    size\_t \_queueSize  
    )  
    – Конструктор инициализатор, установка параметров для вычислений.
2. StartManage() – метод запускает симуляцию модели менеджера процессов. Запускается цикл, длинной в \_countTact. Каждый такт происходит вызов генератора случайных чисел. Если данное число получилось меньше, чем \_qTask, то это сигнализирует о появлении новой задачи для ВС. Задача попадает в очередь TaskList. Если очередь полна, происходит отказ. Количество невыполненных задач увеличивается. Затем запрашивается статус готовности всех ядер. Если ядро выполнило задачу, то количество выполненных задач увеличивается, а ядро становится готовым к новым инструкциям.   
   После этого, если есть свободные ядра, задачи поступают к ним на обработку.  
   Цикл завершает итерацию.

# **Заключение**

Программа, написанная в ходе лабораторной работы, позволяет запускать симуляцию модели работы центрального процессора с достаточно близкой точностью. Использование данной модели позволяет производить необходимые предварительные исследования без создания натуральной действующей модели ЦП.

# **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**

**TQueue**

#define \_\_throwif\_\_(expression,notification) if(expression){throw notification;}

template<class Type>

class TQueue {

Type\* pMem;

size\_t size = 0;

int count = 0;

size\_t setPointer = 0;

size\_t getPointer = 0;

public:

TQueue(int \_size = 0) {

if (\_size > 0) {

setSize(\_size);

}

}

void setSize(int \_size) {

\_\_throwif\_\_(\_size <= 0);

size = \_size;

pMem = new Type[size];

}

void Put(Type& v) {

\_\_throwif\_\_(count == size);

pMem[setPointer] = v;

setPointer = ++setPointer % size;

++count;

}

Type Get() {

\_\_throwif\_\_(count == 0);

Type \_temp = pMem[getPointer];

getPointer = ++getPointer % size;

--count;

return \_temp;

}

bool isEmpty() {

return (count == 0);

}

bool isFull() {

return (size == count);

}

~TQueue() {

delete[] pMem;

}

};

**TaskManager и Kernel**

#include <cassert>

#define RAND\_MAX 32767.00

class Kernel {

double q;

bool ready = true;

size\_t taskCounter = 0;

public:

Kernel() {}

void setParams(double \_q) {

q = \_q;

}

void addTask(unsigned int \_index) {

ready = false;

++taskCounter;

}

unsigned int getTaskCount() {

return taskCounter;

}

bool isReady() {

return ready;

}

bool isDone() {

bool \_temp = ((rand() / RAND\_MAX) < q) && !ready;

if (\_temp) {

ready = true;

}

return \_temp;

}

};

class TaskManager {

double q;

TQueue<size\_t> TaskList;

Kernel\* kernelArray;

size\_t kernelCount = 0;

size\_t startCountTacts = 0;

size\_t countTacts = 0;

size\_t taskIndex = 0;

size\_t taskOff = 0;

size\_t taskDone = 0;

public:

TaskManager(double \_qTask, double \_\_qKernel, size\_t \_kerCount, size\_t \_countTact, size\_t \_queueSize) :q(\_qTask) {

srand(0);

kernelCount = \_kerCount;

countTacts = \_countTact;

TaskList.setSize(\_queueSize);

startCountTacts = \_countTact;

kernelArray = new Kernel[\_kerCount];

for (size\_t i = 0; i < \_kerCount; i++) {

kernelArray[i].setParams(\_\_qKernel);

}

}

void StartManage() {

size\_t i;

double \_q;

while (countTacts) {

\_q = rand() / RAND\_MAX;

if (\_q < q) {

if (!TaskList.isFull()) {

TaskList.Put(taskIndex);

++taskIndex;

}

else {

++taskOff;

}

}

for (i = 0; i < kernelCount; i++) {

if (kernelArray[i].isDone()) {

++taskDone;

}

if (kernelArray[i].isReady()){

if (!TaskList.isEmpty()) {

kernelArray[i].addTask(TaskList.Get());

}

}

}

countTacts--;

}

taskIndex += taskOff;

std::cout << "Count of kernels: " << kernelCount << std::endl;

std::cout << "Count tacts: " << startCountTacts << std::endl;

std::cout << "-----------------------------" << std::endl;

std::cout << "Count of tasks: " << taskIndex << std::endl;

std::cout << "Count of tasks off: " << ((double)taskOff / (double)taskIndex) \* 100 << "%" << std::endl;

std::cout << "Count of unfinished tasks: " << taskIndex - taskDone << std::endl;

std::cout << "Count of tasks done: " << taskDone << std::endl;

}

};