Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное   
учреждение высшего образования

«Национальный исследовательский

Нижегородский государственный университет им. Н.И. Лобачевского»

Институт информационных технологий математики и механики

**Отчет по учебной практике**

**на тему**

**Имитационное моделирование системы обслуживания потока заданий на ЭВМ**

**Выполнил**:студент группы 381703-2

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Долгополов Д.А.

Подпись

**Проверил**:

Доцент каф. МОСТ, к.т.н.

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

Подпись

Нижний Новгород

2018

Оглавление

[Введение 3](#_Toc531183272)

[Постановка задачи 4](#_Toc531183273)

[Руководство пользователя 5](#_Toc531183274)

[Руководство программиста 6](#_Toc531183275)

[Описание структуры программы 8](#_Toc531183276)

[Описание алгоритмов 8](#_Toc531183277)

[Заключение 9](#_Toc531183278)

# 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

Требуется разработать систему классов, обеспечивающих поддержку очереди и программных средств, производящих имитацию процесса обслуживания заданий в вычислительной системе.

При построении модели 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.

# Руководство пользователя

При запуске программы пользователь должен ввести входные данные: количество узлов, количество процессоров в каждом узле, количество ядер в каждом процессоре, частоту появления программ, количество тактов.

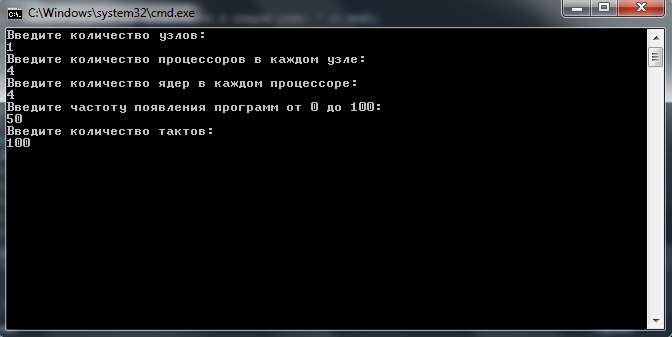


Рис. 1. Меню запуска программы

После ввода всех необходимых данных программа выводит количество всех возникших программ, количество программ, выполнявшихся в момент завершения, количество выполненных программ, количество невыполненных программ, количество программ, которым отказали в выполнении.

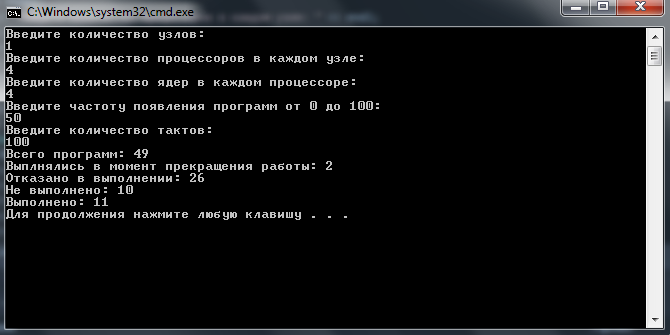


Рис. 2. Результат выполнения программы

# Руководство программиста

В программе реализованы классы под названием "**TQueue**", "**TProgram**", "**TClaster**".

Класс "**TQueue**" содержит следующие поля:

Приватные:

1. begin - переменная, содержащая значение индекса начала очереди.
2. end - переменная, содержащая значение индекса конца очереди.
3. count - переменная, содержащая значение количества элементов в очереди.
4. mem - массив, содержащий элементы класса "TProgram".

Класс "**TQueue**" содержит следующие методы:

Публичные:

1. Необходимый конструктор.
2. IsEmpty - метод, проверяющий очередь на пустоту.
3. IsFull - метод, проверяющий очередь на полноту.
4. Pop - метод, возвращающий первый элемент в очереди и смещающий индекс первого элемента.
5. FirstEl - метод, возвращающий первый элемент в очереди и не смещающий индекс первого элемента.
6. Put - метод, который помещает элемент в конец очереди.

Класс "**TProgram**" содержит следующие поля:

Приватные:

1. f - вспомогательная переменная для случайной генерации.
2. tacts - переменная, содержащая значение количества тактов.
3. cores - переменная, содержащая значение количества ядер.

Класс "**TProgram**" содержит следующие методы:

Публичные

1. Необходимый конструктор и перегруженный оператор присваивания.
2. GetTacts - метод, возвращающий значение переменной tacts.
3. GetCores - метод, возвращающий значение переменной cores.

Класс "**TClaster**" содержит следующие поля:

Приватные:

1. f - вспомогательная переменная для случайной генерации.
2. run\_count, running, is\_running, run\_begining\_tact - вспомогательные переменные и массивы для выполняющихся программ.
3. q - переменная класса "TQueue".
4. free\_cores - переменная, содержащая количество свободный ядер.
5. core\_count - переменная, содержащая количество всех ядер.
6. tacts - переменная, содержащая количество тактов.
7. frequency - переменная, содержащая значение частоты появления программ.
8. running\_now, all\_progs, rejected\_progs, done\_progs, not\_done\_progs - переменные, соответственно содержащие количество программ, выполнявшихся на момент окончания тактов; всех программ; программ, отказавшим в выполнении; завершенных программ; незавершенных программ.

Класс "**TClaster**" содержит следующие методы:

Публичные:

1. Необходимый конструктор.
2. ProgAppear - метод, отвечающий за появление программ.
3. DoTact - метод, выполняющий необходимые действия в тактах.
4. GetAllProgs - метод, возвращающий значение переменной all\_progs.
5. GetRunnigProgs - метод, возвращающий значение переменной running\_now.
6. GetRejected - метод, возвращающий значение переменной rejected\_progs.
7. GetDone - метод, возвращающий значение переменной done\_progs.
8. GetNotDone - метод, возвращающий значение переменной not\_done\_progs.

# Описание структуры программы

Программа состоит из трех проектов: base, base\_test и gtest.

В проекте base содержаться:

1. Заголовочный файл "queue.h", в котором объявлен и реализован класс "TQueue".
2. Заголовочный файл "claster.h", в котором объявлен класс "TClaster".
3. Заголовочный файл "program.h", в котором объявлен и реализован класс "TProgram".
4. Файл "claster.cpp", в котором реализован класс "TClaster".
5. Файл "sample.cpp", в котором реализована функция main, представляющая пример пользовательского меню.

В проекте base\_test содержаться файлы "test\_main.cpp", "test\_claster.cpp", "test\_queue.cpp", "test\_program.cpp" в которых реализованы тесты для проверки работоспособности программы.

В проекте gtest содержаться необходимые файлы для поддержки работы Google Test.

# Описание алгоритмов

Алгоритмы реализации методов класса "TQueue":

1. Метод IsEmpty - сравнивает количество элементов count с 0.
2. Метод IsFull - сравнивает количество элементов count с максимальным размером.
3. Метод Pop - возвращает значение первого элемента и смещает индекс begin.
4. Метод FirstEl - возвращает значение первого элемента и не смещает индекс begin.
5. Метод Put - кладет элемент в конец очереди и смещает индекс end.

Алгоритмы реализации методов класса "TProgram":

1. Метод GetTacts - возвращает количество тактов tacts.
2. Метод GetCores - возвращает количество ядер cores.

Алгоритмы реализации методов класса "TClaster":

1. Метод ProgAppear - генерирует случайное число от 0 до 100 и сравнивает его с частотой, заданной пользователем. Возвращает true, если оно меньше, false - если больше.
2. Метод DoTact - каждый такт с первого по последний, заданный пользователем, выполняет: проверку на выполнение исполняющихся программ; если программа не появляется, то переходит к следующему такту, если появляется, то сравнивает необходимые ей ядра и количества всех ядер и отклоняет, если всех ядер будет недостаточно; если очередь полна, то отклоняет программу, если не полна - помещает ее туда; если первой программе в очереди не хватает свободных ядер, то переходит к следующему такту, если хватает - начинает выполнение программы и переходит к следующему такту.
3. Методы GetAllProgs, GetRunningNow, GetRejected, GetDone, GetNotDone - возвращают значение соответствующих переменных.

# Заключение

Данная лабораторная работа познакомила нас с такой структурой данных как очередь. С помощью очереди можно задать порядок First in - First out, который эффективно применяется для организации выполнения потока заданий на вычислительных системах.

# Литература

1. «C++. Руководство для начинающих», Герберт Шилдт.
2. «Лабораторный практикум. Учебно-методическое пособие», Барышева И.В., Мееров И.Б., Сысоев А.В., Шестакова Н.В. Под редакцией Гергеля В.П.

# Приложение

**"program.h"**

#ifndef \_\_PROGRAM\_H\_\_

#define \_\_PROGRAM\_H\_\_

#include <time.h>

#include <random>

using namespace std;

class TProgram

{

int f;

int tacts;

int cores;

public:

TProgram()

{

srand(time(0));

tacts = rand() % 20 + 1;

srand(time(0) + f);

f += 100;

cores = rand() % 10 + 1;

}

TProgram operator= (const TProgram&p)

{

this->tacts = p.tacts;

this->cores = p.cores;

return \*this;

}

int GetTacts()

{

return tacts;

}

int GetCores()

{

return cores;

}

};

#endif

**"queue.h"**

#include "program.h"

const int MaxMemSize = 10;

class TQueue

{

int begin;

int end;

int count;

TProgram mem[MaxMemSize];

public:

TQueue():begin(-1), end(0), count(0)

{}

bool IsEmpty()

{

return count == 0;

}

bool IsFull()

{

return count == MaxMemSize;

}

TProgram Pop()

{

if (IsEmpty())

throw 123;

count--;

return mem[(begin++) % MaxMemSize];

}

TProgram FirstEl()

{

if (IsEmpty())

throw 123;

return mem[begin % MaxMemSize];

}

void Put(TProgram &v)

{

if (IsFull())

throw 123;

count++;

mem[end] = v;

end = (end + 1) % MaxMemSize;

}

};

**"claster.h"**

#include <vector>

#include <string>

#include <time.h>

#include <random>

#include "program.h"

#include "queue.h"

const int MaxProgramsRunnig = 1000;

using namespace std;

class TClaster

{

int f;

int run\_count = 0; //| Вспомогательные переменные для запущенных программ

TProgram running[MaxProgramsRunnig]; //|

bool is\_running[MaxProgramsRunnig]; //|

int run\_beginig\_tact[MaxProgramsRunnig]; //|

TQueue q;

double free\_cores;

double core\_count;

int tacts;

int frequency;

int running\_now = 0;

int all\_progs = 0; //|Кол-во всех программ

int rejected\_progs = 0; //|Кол-во отказов от выполнения

int done\_progs = 0; //|Кол-во сделанных программ

int not\_done\_progs = 0; //|Кол-во несделаных программ

public:

TClaster(double node = 2, double processor = 2, double core = 2, int \_tacts = 0, int \_freaquency = 50)

{

frequency = \_freaquency;

tacts = \_tacts;

core\_count = node \* processor \* core;

free\_cores = core\_count;

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

is\_running[i] = false;

}

bool ProgAppear();

void DoTact();

int GetAllProgs();

int GetRunningNow();

int GetRejected();

int GetDone();

int GetNotDone();

};

**"claster.cpp"**

#include "Claster.h"

bool TClaster::ProgAppear()

{

int r;

srand(time(0) + f);

f += 10;

r = rand() % 100;

if (r < frequency)

return true;

else

return false;

}

void TClaster::DoTact()

{

int temp = 1;

while (temp != tacts + 1)

{

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

{

if (is\_running[i] == true)

if (temp - run\_beginig\_tact[i] == running[i].GetCores())

{

is\_running[i] = false;

free\_cores += running[i].GetCores();

run\_count--;

running\_now--;

done\_progs++;

}

}

if (ProgAppear())

{

TProgram p;

all\_progs++;

if (p.GetCores() > core\_count)

{

rejected\_progs++;

temp++;

continue;

}

else

if (q.IsFull())

{

rejected\_progs++;

temp++;

continue;

}

else

q.Put(p);

if (q.FirstEl().GetCores() > free\_cores)

{

temp++;

continue;

}

else

{

TProgram tmp = q.Pop();

running[run\_count] = tmp;

is\_running[run\_count] = true;

run\_beginig\_tact[run\_count] = temp;

run\_count++;

running\_now++;

free\_cores = free\_cores - tmp.GetCores();

}

}

else

{

temp++;

continue;

}

temp++;

}

while (!q.IsEmpty())

{

not\_done\_progs++;

q.Pop();

}

}

int TClaster::GetAllProgs()

{

return all\_progs;

}

int TClaster::GetRunningNow()

{

return running\_now;

}

int TClaster::GetRejected()

{

return rejected\_progs;

}

int TClaster::GetDone()

{

return done\_progs;

}

int TClaster::GetNotDone()

{

return not\_done\_progs;

}

**"sample.cpp"**

#include "Claster.h"

#include <iostream>

using namespace std;

void main()

{

setlocale(LC\_ALL, "Russian");

int tacts, nodes, processors, cores, freaquency;

cout << "Введите количество узлов: " << endl;

cin >> nodes;

cout << "Введите количество процессоров в каждом узле: " << endl;

cin >> processors;

cout << "Введите количество ядер в каждом процессоре: " << endl;

cin >> cores;

cout << "Введите частоту появления программ от 0 до 100: " << endl;

cin >> freaquency;

cout << "Введите количество тактов: " << endl;

cin >> tacts;

TClaster clast(nodes, processors, cores, tacts, freaquency);

clast.DoTact();

cout << "Всего программ: " << clast.GetAllProgs() << endl;

cout << "Выплнялись в момент прекращения работы: " << clast.GetRunningNow() << endl;

cout << "Отказано в выполнении: " << clast.GetRejected() << endl;

cout << "Не выполнено: " << clast.GetNotDone() << endl;

cout << "Выполнено: " << clast.GetDone() << endl;

}

**"test\_claster.cpp"**

#include "claster.cpp"

#include <gtest.h>

TEST(TClaster, can\_create\_claster)

{

ASSERT\_NO\_THROW(TClaster q);

}

TEST(TClaster, not\_losing\_programs)

{

TClaster q;

q.DoTact();

EXPECT\_EQ(q.GetAllProgs(), q.GetDone() + q.GetNotDone() + q.GetRejected() + q.GetRunningNow());

}

**"test\_program.cpp"**

#include "program.h"

#include <gtest.h>

TEST(TProgram, can\_assign\_program)

{

TProgram f;

TProgram q;

ASSERT\_NO\_THROW(f = q);

}

**"test\_queue.cpp"**

#include "queue.h"

#include <gtest.h>

TEST(TQueue, can\_create\_queue)

{

ASSERT\_NO\_THROW(TQueue q);

}

TEST(TQueue, cant\_pop\_empty\_queue)

{

TQueue q;

ASSERT\_ANY\_THROW(q.Pop());

}

TEST(TQueue, can\_check\_empty\_queue)

{

TQueue q;

EXPECT\_TRUE(q.IsEmpty());

}

TEST(TQueue, can\_check\_full\_queue)

{

TQueue q;

TProgram v;

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

q.Put(v);

EXPECT\_TRUE(q.IsFull());

}

TEST(TQueue, cant\_put\_in\_full\_queue)

{

TQueue q;

TProgram v;

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

q.Put(v);

ASSERT\_ANY\_THROW(q.Put(v));

}

TEST(TQueue, cant\_get\_first\_element\_of\_empty\_queue)

{

TQueue q;

ASSERT\_ANY\_THROW(q.FirstEl());

}