Министерство науки и высшего образования Российской Федерации

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

Капустин Григорий Константинович

**Проверил**:

Доцент кафедры МОСТ, к.т.н.

Сысоев А.B.

Нижний Новгород

2018

Содержание

[Введение 3](#_Toc531300819)

[Постановка задачи 4](#_Toc531300820)

[Руководство пользователя 5](#_Toc531300821)

[Руководство программиста 6](#_Toc531300822)

[Описание структуры программы 7](#_Toc531300823)

[Описание алгоритмов 8](#_Toc531300824)

[Заключение 9](#_Toc531300825)

[Литература 10](#_Toc531300826)

[Приложение 11](#_Toc531300827)

# 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.

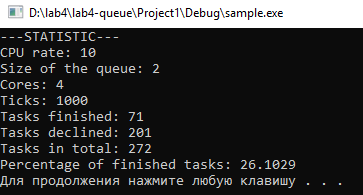
Очередь характеризуется таким порядком обработки значений, при котором вставка новых элементов производится в конец очереди, а извлечение – из начала. Подобная организация данных широко встречается в различных приложениях. В качестве примера использования очереди предлагается задача разработки системы 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

В рамках лабораторной работы ставится задача реализации программы, обеспечивающих поддержку очереди, и разработки программных средств, производящих имитацию однопроцессорной ЭВМ.

# Руководство пользователя

При запуске программы-примера (проект «sample») на экран будет выведен результат работы класса «процессор».



**Рис. 1. Вывод в консоль**

# Руководство программиста

В программе реализованы три класса: «очередь», «задача» и «процессор».

**Класс «Очередь» - шаблонный, имеющий следующие приватные поля:**

1. mem: массив элементов шаблонного типа,
2. size: хранит размер массива,
3. begin: переменная, хранящая индекс элемента, который считаем за начало очереди,
4. end: соответственно, конец очереди,
5. counter: хранит размер очереди.

**В классе «Очередь» имеются следующие методы:**

1. isEmpty: возвращает 1 – если очередь не содержит элементов, 0 – если есть элементы,
2. IsFull: возвращает 1 – если очередь полна, 0 – если не полон,
3. push\_back: в качестве аргумента принимает элемент шаблонного типа, добавляет его в очередь, ничего не возвращает,
4. getSize: возвращает актуальное количество элементов в очереди,

**Класс «Задача» имеет два приватных поля:**

1. intense: сложность задачи,
2. state: boolean переменная «состояние задачи»,

**Также содержатся следующие приватные методы:**

1. setState: устанавливает state = false,
2. tryToComplete: выполняется попытка сделать задачу, возвращает 1 в случае успеха, 0 – в случае неудачи,
3. getState: возвращает переменную «state»,

В header файле этого класса есть глобальная константа double MAX\_RANDOM, определяющая настройку «случайности» для метода tryToComplete.

**В классе «Процессор» присутствуют несколько приватных полей:**

1. procRate: частота симулируемого процессора,
2. numCores: число ядер,
3. ticks: количество тактов, необходимых выполнить за цикл работы,
4. tasksDone: счетчик успешных выполнений задач,
5. tasksLost: счетчик утерянных задач.
6. Queue<int> \*queue: очередь, необходимая для выполнения алгоритма
7. vector<JobStream> runTasks: поток задач, поступающий на процессор.

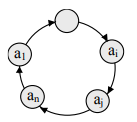
## Описание структуры программы

В решении содержатся:

1. проект code, содержащий классы:
   1. JobStream.h
   2. Proc.h
   3. Queue.h
2. проект sample, содержащий программу-пример.
   1. Sample\_main.cpp
3. проект test, включающий в себя юнит-тесты для класса «очередь»
   1. Test\_cases.cpp
   2. Test\_main.cpp

## Описание алгоритмов

**Очередь** – структура данных, добавление элементов в котором происходит по принципу «первым пришел - первым ушел» (first in, first out или FIFO). При реализации такого множества, важную роль играет выбор подходящего метода хранения данных, обеспечивающего эффективного использования памяти без перепаковок и без использования списков. Будем использовать кольцевой буфер.



**Рис. 2 схема структуры хранения вида кольцевой буфер.**

Следующие методы необходимы для реализации такой структуры данных:

1. void push\_back(T elem)

Добавляет шаблонный элемент в конец очереди, увеличиваем счетчик элементов, увеличиваем на единицу по модулю размера очереди индекс последнего элемента.

1. T pop()

Возвращается первый элемент, уменьшаем счетчик элементов, сдвигаем, опять же по модулю размера очереди, индекс первого элемента.

**Моделирование работы процессора** будет выполняться по довольно простой схеме:

1. Задаем фиксированное количество тактов, которые выполнит ЭВМ.
2. На каждом из них опрашивается состояние потока задач и сам процессор.
3. Если задача появилась, то добавляем её в очередь ожидания на выполнение, а если она переполнена – считаем, что задание утеряно.
4. В случае выполнения задания предпринимается попытка загрузки процессора новыми задачами, для этого они извлекаются из очереди ожидания.

# Заключение

Была изучена, реализована структура хранения данных очередь. Стало понятна область её область применения – в случаях, когда необходимо выполнять какие-то действия в порядке их поступления, что и было выполнено при моделировании работы процессора.

# Литература

1. Бьерн Страуструп. «Язык программирования С++»,
2. Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн, «Introduction to Algorithms »,
3. Барышева И.В., Мееров И.Б., Сысоев А.В., Шестакова Н.В. Под редакцией Гергеля В.П. «Лабораторный практикум. Учебно-методическое пособие».

# Приложение

**Модуль “queue.h”**

#pragma once

const int MAX\_SIZE = 100000;

template <class T>

class queue

{

int size = -1;

int begin=0;

int end= 0;

int counter = 0;

T\* mem;

public:

queue(int size)

{

if (size< 0 || size > MAX\_SIZE)

throw("error");

this->size = size;

mem = new T[size];

}

bool isFull()

{

return counter == size;

}

bool isEmpty()

{

return counter == 0;

}

void push\_back(T elem)

{

if (isFull())

throw("error");

mem[end] = elem;

counter++;

end++;

end %= size;

}

T pop()

{

if (isEmpty())

throw("error");

T tmp = mem[begin];

counter--;

begin++;

begin %= size;

return tmp;

}

int getSize()

{

return size;

}

~queue()

{

delete[] mem;

}

};

**Модуль “JobStream.h”**

# #pragma once

#include <time.h>

#include <cstdlib>

const double MAX\_RANDOM = 1000.0;

class JobStream

{

double intense;

bool state = true;

public:

JobStream(double \_intense) :

intense(\_intense) { }

void setState()

{

state = false;

}

bool tryToComplete()

{

return state = (rand() / MAX\_RANDOM) < intense;

}

bool getState()

{

return state;

}

};

**Модуль “Proc.h”**

#pragma once

#include "JobStream.h"

#include "queue.h"

#include <iostream>

#include <vector>

using namespace::std;

class proc

{

double procRate;

int numCores;

int ticks;

int tasksDone = 0;

int tasksLost = 0;

queue<int> \*list;

vector<JobStream> runTasks;

public:

proc(double rate, double tasksIntense, int cores, int ticksToBeDone, int size) :

procRate(rate),numCores(cores), ticks(ticksToBeDone)

{

srand(time(NULL));

list = new queue<int>(size);

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

runTasks.push\_back(JobStream(tasksIntense));

}

void start()

{

int \_ticks = ticks;

while (\_ticks)

{

if ((rand() / MAX\_RANDOM) < procRate)

if (!list->isFull())

list->push\_back(0);

else

tasksLost++;

for (auto& c : runTasks)

if (c.getState())

if (!list->isEmpty())

{

c.setState();

list->pop();

}

else;

else

if (c.tryToComplete())

tasksDone++;

\_ticks--;

}

}

void print()

{

int totalTasks = tasksDone + tasksLost;

double percentage = double(tasksDone) / totalTasks \* 100;

cout << "---STATISTIC---" << endl;

cout << "CPU rate: " << procRate << endl;

cout << "Size of the queue: " << list->getSize() << endl;

cout << "Cores: " << numCores << endl;

cout << "Ticks: " << ticks << endl;

cout << "Tasks finished: " << tasksDone << endl;

cout << "Tasks declined: " << tasksLost << endl;

cout << "Tasks in total: " << totalTasks << endl;

cout << "Percentage of finished tasks: " << percentage << endl;

}

~proc()

{

delete list;

}

};

**Модуль “sample\_main.cpp”**

#include "Proc.h"

int main(int argc, char \*\*argv)

{

proc h(10.0, 0.8, 4, 1000, 2);

h.start();

h.print();

system("pause");

}

**Модуль “test\_cases.cpp”**

#include <gtest.h>

#include "queue.h"

TEST(queue, can\_create\_queue)

{

ASSERT\_NO\_THROW(queue<int> bf(10));

}

TEST(queue, cant\_create\_queue\_with\_negative\_size)

{

ASSERT\_ANY\_THROW(queue<int> bf(-10));

}

TEST(queue, cant\_create\_queue\_with\_too\_large\_size)

{

ASSERT\_ANY\_THROW(queue<int> bf(1000000));

}

TEST(queue, can\_push\_back)

{

queue<int> bf(2);

ASSERT\_NO\_THROW(bf.push\_back(10));

}

TEST(queue, can\_pop)

{

queue<int> bf(2);

bf.push\_back(10);

EXPECT\_EQ(10, bf.pop());

}

TEST(queue, cant\_push\_back\_into\_full\_queue)

{

queue<int> bf(1);

bf.push\_back(10);

EXPECT\_ANY\_THROW(bf.push\_back(100));

}

TEST(queue, cant\_pop\_into\_empty\_queue)

{

queue<int> bf(1);

EXPECT\_ANY\_THROW(bf.pop());

}

TEST(queue, can\_get\_size)

{

queue<int> bf(1);

EXPECT\_EQ(1, bf.getSize());

}

TEST(queue, isFull\_is\_correct)

{

queue<int> bf(1);

bf.push\_back(0);

EXPECT\_TRUE(bf.isFull());

}

TEST(queue, isEmpty\_is\_correct)

{

queue<int> bf(1);

EXPECT\_TRUE(bf.isEmpty());

}