Министерство науки и высшего образования 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 лабораторной работе №4**

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

**Выполнил**:

студент группы 382006-1

Расчётнов А.С.

**Проверил**:

доцент кафедры МОСТ, к.т.н.,

Сысоев А.В.

Нижний Новгород

2021

Содержание

[Введение 3](#_Toc529541653)

[1. Постановка задачи 4](#_Toc529541654)

[2. Руководство пользователя 5](#_Toc529541655)

[3.1. Описание структуры программы 6](#_Toc529541656)

[3.2. Описание алгоритмов 12](#_Toc529541657)

[4. Результаты экспериментов 13](#_Toc529541658)

[Заключение 14](#_Toc529541659)

[Литература 15](#_Toc529541660)

[Приложение 16](#_Toc529541661)

# 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 несколькими процессорами 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;

• количество выполненных заданий;

• количество тактов простоя процессора из-за отсутствия в очереди заданий для обслуживания.

• количество невыполненных заданий.

• средняя загруженность кластера.

# Руководство пользователя

При открытии программы пользователь видит окно с приглашением к вводу параметров кластера (рис.1).

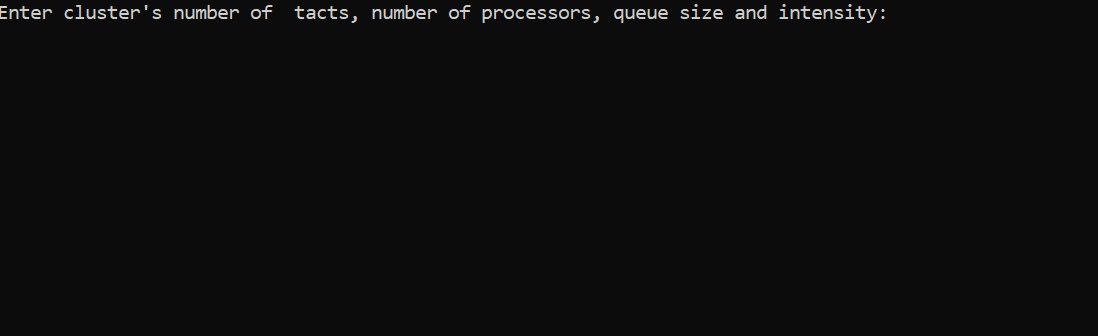


Рис.1. Стартовое окно

Необходимо последовательно ввести число тактов, количество процессоров, размер очереди и интенсивность появления заданий, после чего программа сымитиурет работу кластера и выдаст статистику, которая включает в себя количество сгенерированных заданий, количество отказов из-за переполнения очереди, количество выполненных заданий, количество заданий, находящихся в очереди, на момент завершения работы кластера, количество тактов простоя и среднюю загруженность кластера.

1. **Руководство программиста**

## Описание структуры программы

Программа реализована на языке программирования C++ с использованием шести пользовательских классов: Cluster, Processor, Queue, Statistics, Task, TasksStreamSimulator. Исходные файлы с реализацией методов классов расположены в директории srs, заголовочные файлы – в include. В папке samples расположен исходный код программы-сэмпла.

Поля класса Cluster:

* «vector<Processor> processors»

Данное поле хранит вектор процессоров кластера пользовательского класса Processor.

* «number\_of\_processors»

Поле хранит число процессоров кластера.

Методы класса Cluster:

* «Cluster(int \_number\_of\_processors)»

Метод является конструктором класса, поле number\_of\_processors инициализируется значением параметра \_number\_of\_processors, поле processors инициализируется процессорами, создающимися конструктором по умолчанию класса Processor.

* «int free\_processors()»

Метод возвращает количество свободных процессоров кластера.

Входных данных не имеет.

Выходные данные: переменная типа int – количество свободных процессоров кластера.

* «void update()»

Метод обновляет информацию о занятости процессоров класса.

Входных данных не имеет.

Выходных данных не имеет.

* «void give(int number, int tacts)»

Метод выдаёт кластеру заданию на выполнение.

Входные данные: переменная number типа int – количество процессоров, требующихся для выполнения задания, переменная tacts типа int – число тактов выполнения задания.

Выходных данных не имеет.

Поля класса Processor:

* «int busy\_tacts»

Поле хранит количество тактов, на которые процессор занят.

* «bool free»

Поле хранит информацию о том, свободен ли процессор в данный момент.

Методы класса Processor:

* «Processor(int \_busy\_tacts, bool \_free)»

Метод является конструктором класса, поле busy\_tacts инициализируется параметров \_busy\_tacts, поле free инициализируется параметром \_free.

* «bool is\_free()»

Метод проверяет, свободен ли процессор в данный момент.

Входных данных не имеет.

Выходные данные: переменная типа bool – значение поля free.

* «void decrease\_tacts()»

Метод уменьшает на единицу число тактов, на которых процессор занят (поле busy\_tacts).

Входных данных не имеет.

Выходных данных не имеет.

* «void set\_status()»

Метод устанавливает в нуль или единицу поле класса free, в зависимости от того, занят ли сейчас процессор или свободен.

Входных данных не имеет.

Выходных данных не имеет.

* «int get\_busy\_tacts()»

Метод возвращает количество тактов, на которые процессор занят.

Входных данных не имеет.

Выходные данные: переменная типа int – значение поля busy\_tacts.

* «void set\_busy\_tacs(int tacts)»

Метод устанавливает количество тактов, на которые процессор занят.

Входные данные: переменная tacts типа int – количество тактов, на которые процессор занят.

Выходных данных не имеет.

Поля класса TQueue:

* «int queue\_size»

Поле хранит размер очереди.

* «int top»

Поле хранит значение верхней границы очереди.

* «int bottom»

Поле хранит значение нижней границы очереди.

* «T\* data»

Поле хранит содержимое очереди произвольного типа T.

Методы класса TQueue:

* «TQueue(int \_queue\_size)»

Метод является конструктором класса, поле queue\_size инициализируется параметром \_queue\_size, затем выделяется память размера queue\_size для содержимого очереди, поля top и bottom по умолчанию инициализируются значениями -1 и 0.

* «int size()»

Метод возвращает размер очереди.

Входных данных не имеет.

Выходные данные: переменная типа int – значение поля queue\_size.

* «T front()»

Метод возвращает значение нижнего элемента очереди.

Входных данных не имеет.

Выходные данные: переменная типа T – содержимое нижнего элемента очереди.

* «T pop()»

Метод возвращает значение нижнего элемента очереди и удаляет его оттуда.

Входных данных не имеет.

Выходные данные: переменная типа T – содержимое нижнего элемента очереди.

* «void push(const T& object)»

Метод добавляет элемент в конец очереди.

Входные данные: константная ссылка на тип T – помещаемый в очередь элемент.

Выходных данных не имеет.

* «int current\_size()»

Метод возвращает текущее количество объектов в очереди.

Входных данных не имеет.

Выходные данные: переменная типа int – количество объектов в очереди.

* «bool is\_empty()»

Метод проверяет очередь на пустоту.

Входных данных не имеет.

Выходные данные: переменная типа bool, содержащая нуль, если очередь пуста и единицу в противном случае.

* «bool is\_full()»

Метод проверяет очередь на полноту.

Входных данных не имеет.

Выходные данные: переменная типа bool, содержащая единицу, если очередь полна и нуль в противном случае.

Поля класса Statistics:

* «int generated\_tasks»

Поле хранит количество появившихся заданий.

* «int not\_generated\_tasks»

Поле хранит количество заданий, не помещённых в очередь на исполнения из-за переполнения очереди.

* «int tasks\_in\_queue»

Поле хранит количество заданий, оставшихся в очереди на момент завершения работы кластера.

* «int completed\_tasks»

Поле хранит количество завершённых заданий.

* «int not\_completed\_tasks»

Поле хранит количество невыполненных заданий (задания, которые на момент завершения работы кластера, выполнялись).

* «int did\_nothing»

Поле хранит количество тактов простоя кластера.

* «double average\_workload»

Поле хранит среднюю загруженность кластера.

Методы класса Statistics:

* «int get\_generated\_tasks()»

Метод возвращает количество появившихся заданий.

Входных данных не имеет.

Выходные данные: переменная типа int – количество сгенерированных заданий.

* «int not\_generated\_tasks()»

Метод возвращает количество не помещённых в очередь заданий из-за её переполнения.

Входных данных не имеет.

Выходные данные: переменная типа int - количество не помещённых в очередь заданий из-за её переполнения.

* «int get\_completed\_tasks()»

Метод возвращает количество выполненных заданий.

Входных данных не имеет.

Выходные данные: переменная типа int – количество выполненных заданий.

* «int get\_not\_completed\_tasks()»

Метод возвращает количество невыполненных заданий.

Входных данных не имеет.

Выходные данные: переменная типа int – количество невыполненных заданий.

* «int get\_did\_nothing()»

Метод возвращает количество тактов простоя кластера.

Входных данных не имеет.

Выходные данные: переменная типа int – количество тактов простоя кластера.

* «int get\_average\_workload»

Метод возвращает среднюю загруженность кластера.

Входных данных не имеет.

Выходные данные: переменная типа int – средняя загруженность кластера.

Поля класса Task:

* «int number»

Поле хранит номер задания.

* «int execute\_duraion»

Поле хранит количество тактов для выполнения задания.

* «int processors\_needed»

Поле хранит количество необходимых процессоров для выполнения задания.

Методы класса Task:

* «Task(int \_number, int \_execute\_duration, int \_processors\_needed)»

Метод является конструктором класса, поле number инициализируется параметром \_number, поле execute\_duration инициализируется значением параметра \_execute\_duration, поле processors\_needed инициализируется значением параметра \_processors\_needed.

* «int get\_number()»

Метод возвращает номер задания (значение поля number).

Входных данных не имеет.

Выходные данные: переменная типа int – номер задания.

* «int get\_duration()»

Метод возвращает количество тактов для выполнения задания (значение поля execute\_duration)

Входных данных не имеет.

Выходные данные: переменная типа int – количество тактов для выполнения задания.

* «int get\_processor\_needed()»

Метод возвращает число необходимых для выполнения задания процессоров.

Входных данных не имеет.

Выходные данные: переменная типа int – количество необходимых процессоров для выполнения задания.

Поля класса TasksStreamSimulator:

* «int number\_of\_tacts»

Поле хранит число тактов работы кластера.

* «int number\_of\_processors»

Поле хранит количество процессоров кластера.

* «int queue\_size»

Поле хранит размер очереди, в которую помещаются сгенерированные задания.

* «double intensity»

Поле хранит число в диапазоне от нуля до единицы, характеризующее интенсивность появления заданий. Чем больше число, тем больше заданий будут появляться, чем меньше число, тем меньше заданий будут появляться.

Методы класса TasksStreamSimulator:

* «TasksStreamSimulator(int \_number\_of\_tacts, int \_number\_of\_processors, \_int \_queue\_size, double \_intensity»

Метод является конструктором класса, все поля инициализируются входными параметрами.

* «double get\_intensity()»

Метод возвращает значение интенсивности появления заданий (поле intensity).

Входных данных не имеет.

Выходные данные: переменная типа int – значение интенсивности появления заданий.

* «void begin()»

Метод непосредственно запускает симуляцию работы кластера.

Входных данных не имеет.

Выходных данных не имеет.

## Описание алгоритмов

* «begin()»

Метод выполняет имитацию работы кластера. Создаются очередь, в которую попадают появившиеся задания и сам кластер. Появление заданий происходит за счёт генератора случайных чисел, который генерирует вещественное число в диапазоне от 0 до 1. В случае если число больше заданной наперёд интенсивности появления заданий, то задание генерируется (может сгенерироваться не одно задание, а несколько (см. далее)), если меньше – не генерируется. Далее с помощью того же генератора чисел, генерируется целое число в диапазоне от 1 до 3 – число появившихся заданий на текущем такте. Далее выполняется цикл для каждого такта, в котором задания помещаются в очередь, в том случае, если она не полна. В этом же цикле происходит распределение заданий кластером по следующему принципу: из очереди берётся задание, если у кластера есть необходимые ресурсы для его выполнения на текущем такте (свободные процессоры), то задание из очереди помещается на выполнение, и берётся следующее. Процесс происходит до тех пор, пока у кластера есть свободные ресурсы, чтобы выполнить текущее задание в очереди. Если кластер не может выполнить какое-то задание на текущем такте, то он будет ждать, пока необходимое число процессоров освободится, чтобы поместить это задание на выполнение, не беря при этом на выполнение менее требовательные задания из очереди. По ходу итераций цикла собирается статистика о работе класса. Симуляция работы кластера заканчивается, после того как цикл будет выполнен для каждого такта.

* «set\_status()»

Метод устанавливает статус процессора. Процессор может быть свободным или занятым. Метод проверяет значение поля busy\_tacts и на основе этого значения устанавливает значение поля free. Если значение поля busy\_tacts не равно нулю, то процессор занят, если равно нулю – процессор свободен.

* «update()»

Метод обновляет информацию о свободных процессорах кластера после каждого такта. После каждого такта, значение поля busy\_tacts для каждого процессора кластера уменьшается на единицу, в случае если оно не было нулевым (т.е. процессор на предыдущем такте работал и к следующему такту обязан освободиться), после чего происходит выполнение метода set\_status(), который обновляет информацию о свободных процессорах кластера.

# Результаты экспериментов

Пример работы программы программы-сэмпла при входных значениях 1000, 64, 50, 0.2 (рис.2).

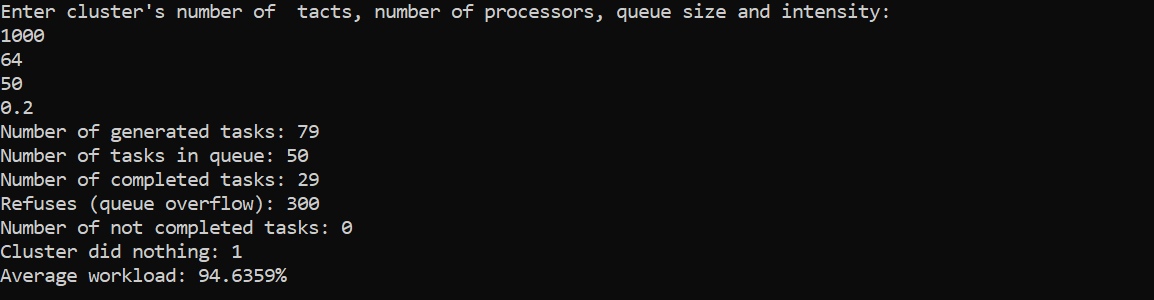


Рис.2. Пример работы программы

# Заключение

В результате выполнения лабораторной работы, был реализован пользовательский класс очереди TQueue, с помощью которого была создана программа, имитирующая выполнение потока заданий на кластере.

# Литература

# Бьёрн Страуструп. «Язык программирования C++»:

<http://www.8361.ru/6sem/books/Straustrup-Yazyk_programmirovaniya_c.pdf>.

1. Барышева И.В., Мееров И.Б., Сысоев А.В., Шестакова Н.В.. Лабораторный практикум (по программе "Алгоритмы и структуры данных "):

<http://www.lib.unn.ru/students/020302.html>.

# Приложение

**TasksStreamSimulator.cpp**

void TasksStreamSimulator::begin() {  
 int counter\_of\_tasks = 0;  
 int tasks\_per\_tact;  
 int sum\_of\_busy\_processors\_per\_tact = 0;  
 TQueue<Task> call\_queue(queue\_size);  
 Cluster cluster(number\_of\_processors);  
 for (int i = 0; i < number\_of\_tacts; i++) {  
 if (generate\_double(0, 1) <= intensity) {  
 counter\_of\_tasks++;  
 tasks\_per\_tact = generate\_int(1, 3);  
 for (int i = 0; i < tasks\_per\_tact; i++)  
 if(![call\_queue.is](https://vk.com/away.php?utf=1&to=http%3A%2F%2Fcall_queue.is)\_full()) {  
 call\_queue.push(Task(counter\_of\_tasks, generate\_int(1, 3), generate\_int(1, 4)));  
 generated\_tasks++;  
 }  
 else  
 not\_generated\_tasks++;  
 }  
 if (call\_queue.front().get\_processors\_needed() <= [cluster.free](https://vk.com/away.php?utf=1&to=http%3A%2F%2Fcluster.free)\_processors())  
 while (call\_queue.front().get\_processors\_needed() <= [cluster.free](https://vk.com/away.php?utf=1&to=http%3A%2F%2Fcluster.free)\_processors() && ![call\_queue.is](https://vk.com/away.php?utf=1&to=http%3A%2F%2Fcall_queue.is)\_empty()) {  
 cluster.give(call\_queue.front().get\_processors\_needed(), call\_queue.front().get\_duration());  
 if (call\_queue.front().get\_duration() + i < number\_of\_tacts)  
 completed\_tasks++;  
 else  
 not\_completed\_tasks++;  
 call\_queue.pop();  
 }  
 if ([cluster.free](https://vk.com/away.php?utf=1&to=http%3A%2F%2Fcluster.free)\_processors() == number\_of\_processors)  
 did\_nothing++;  
 sum\_of\_busy\_processors\_per\_tact += (number\_of\_processors - [cluster.free](https://vk.com/away.php?utf=1&to=http%3A%2F%2Fcluster.free)\_processors());  
 }  
 average\_workload = static\_cast<double>(sum\_of\_busy\_processors\_per\_tact) / (static\_cast<double>(number\_of\_tacts) \* static\_cast<double>(number\_of\_processors));  
 tasks\_in\_queue = call\_queue.current\_size();  
}