



**Министерство образования и науки Российской Федерации  
Федеральное государственное бюджетное образовательное  
учреждение  
высшего образования  
«Московский государственный технический университет  
имени Н.Э. Баумана  
(национальный исследовательский университет)»  
(МГТУ им. Н.Э. Баумана)**

---

Факультет «Робототехники и комплексной автоматизации»  
Кафедра «Системы автоматизированного проектирования» (РК-6)

**Отчет**

по лабораторной работе №1  
по курсу «Разработка программных систем»  
Вариант №5

Студент: Смирнова А. А.  
Группа: РК6-6  
Преподаватель: Федорук В. Г.

Дата: \_\_\_\_\_  
Подпись: \_\_\_\_\_

Москва, 2018

## Оглавление

Текст задания на лабораторную работу	3
Текстовое описание структуры программы	4
Реализованный способ взаимодействия процессов	5
Блок-схема программы	6
Пример результатов работы программы	7
Текст программы	8

## Текст задания на лабораторную работу

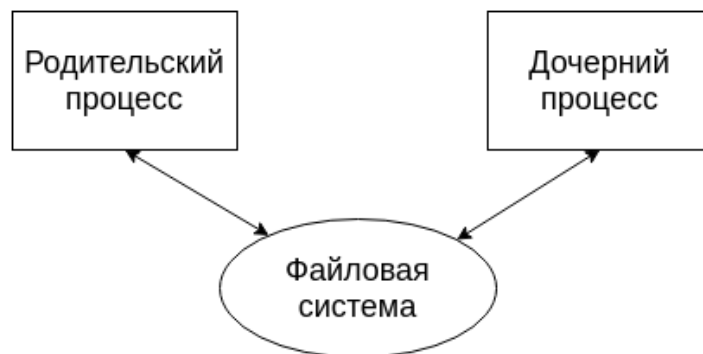
Составить программу, которая заданное число раз (для определенности 5) через определенный временной интервал (5 сек.) повторяет на экране запрос, ожидающий стандартный ввод. Процесс должен завершаться в случае корректного ответа на запрос или после исчерпывания заданного числа запросов.

## Текстовое описание структуры программы

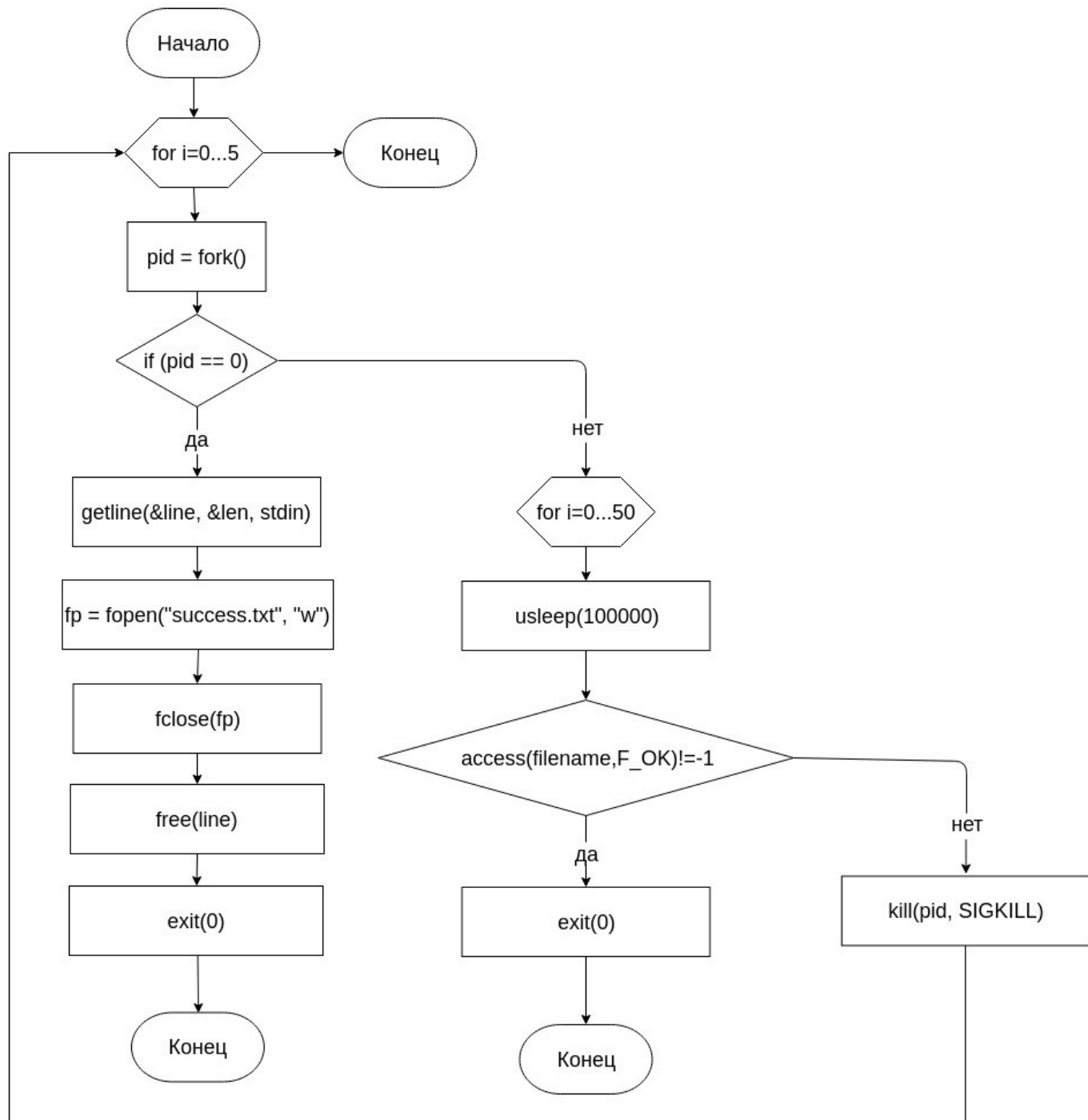
В начале программы с помощью системного вызова `fork()` родительский процесс создает дочерний. В случае успешного завершения системный вызов `fork` для дочернего процесса возвращает 0, а для процесса-родителя PID созданного процесса. С помощью PID процесса в программе определяется, является ли текущий процесс родительским или дочерним. В программе коммуникация между процессами организована с помощью файловой системы. Дочерний процесс ждет ввода от пользователя и в случае введения пользователем любой строки, создает в текущей директории файл `success.txt`. Родительский процесс каждые 0.1 секунду проверяет, есть ли в текущей директории файл `success.txt`. Если есть - то программа завершается, иначе, по истечению 5 секунд работы дочернего процесса, родительский процесс убивает его и с помощью `fork` создает еще один. Всего родительский процесс может максимум создать 5 дочерних, после этого программа завершится.

## Реализованный способ взаимодействия процессов

Взаимодействие между процессами осуществлялось с помощью разделяемой файловой системы.

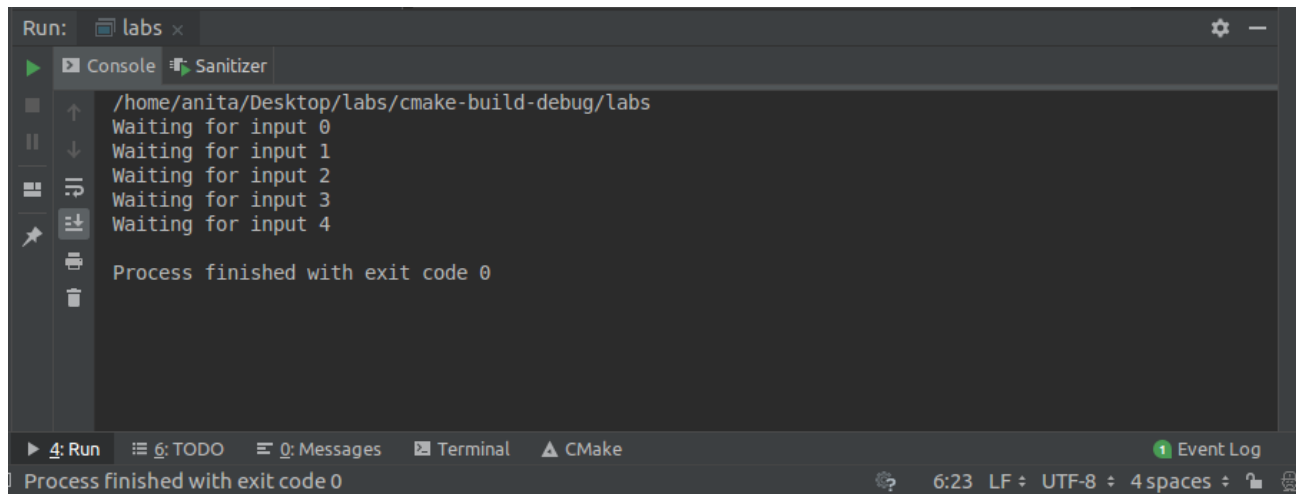


## Блок-схема программы



## Пример результатов работы программы

1. Пользовательский ввод отсутствовал. Было создано 5 дочерних процессов, после этого программа была завершена.



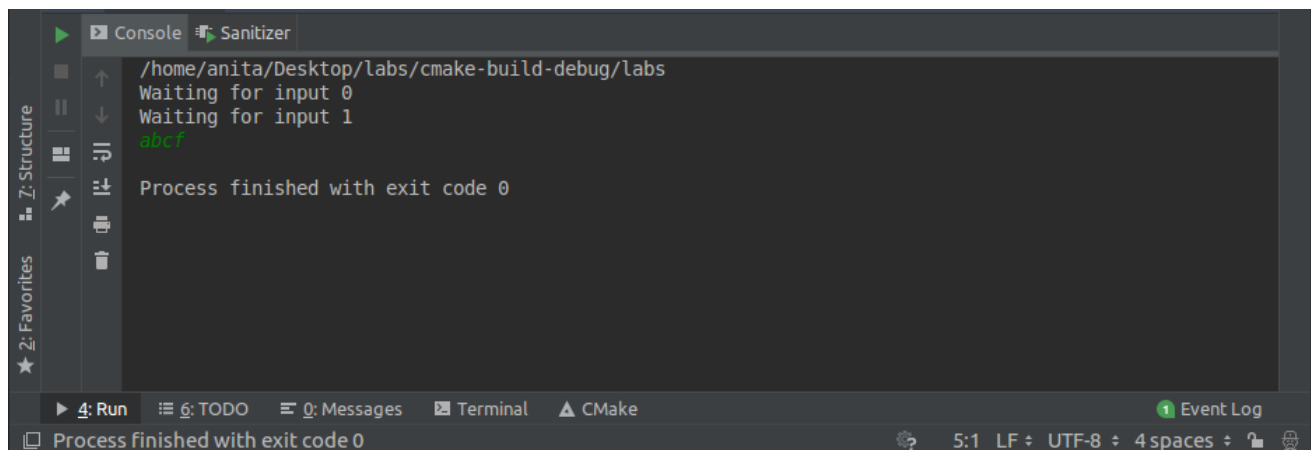
The screenshot shows a terminal window titled 'Run: labs x'. The console output is as follows:

```
/home/anita/Desktop/labs/cmake-build-debug/labs
Waiting for input 0
Waiting for input 1
Waiting for input 2
Waiting for input 3
Waiting for input 4

Process finished with exit code 0
```

The terminal interface includes a left sidebar with icons for Run, TODO, Messages, Terminal, CMake, and Event Log. The status bar at the bottom indicates 'Process finished with exit code 0', '6:23', 'LF', 'UTF-8', and '4 spaces'.

2. Был совершен пользовательский ввод. Было создано 2 дочерних процессов, после ввода строки программа была завершена.



The screenshot shows a terminal window titled 'Run: labs x'. The console output is as follows:

```
/home/anita/Desktop/labs/cmake-build-debug/labs
Waiting for input 0
Waiting for input 1
abc
Process finished with exit code 0
```

The terminal interface includes a left sidebar with icons for Run, TODO, Messages, Terminal, CMake, and Event Log. The status bar at the bottom indicates 'Process finished with exit code 0', '5:1', 'LF', 'UTF-8', and '4 spaces'.

## Текст программы

```
#define _GNU_SOURCE
#include <stdio.h>
#include <zconf.h>
#include <stdlib.h>
#include <signal.h>

// Значение таймаута в секундах
#define TIMEOUT 5

// Количество дочерних процессов
#define TRIES 5

int main() {
    const char *filename = "success.txt";
    // Удаление файла, если он был создан при предыдущей работе программы
    remove(filename);

    for (int i = 0; i < TRIES; i++) {
        // Создание дочернего процесса
        int pid = fork();

        // Если процесс дочерний
        if (pid == 0) {
            char *line = NULL;
            size_t len = 0;
            printf("Waiting for input %d\n", i);

            // Дочерний процесс ждет ввода
            getline(&line, &len, stdin);

            // Если ввод произошел - создается файл
            FILE *fp = fopen(filename, "w+");
            fclose(fp);

            // Очищение памяти под введенную строку
            free(line);

            // Процесс завершается
            exit(0);
        }
        // Если процесс родительский
        else {
            for (int j = 0; j < TIMEOUT * 10; j++) {

                // Родительский процесс засыпает на 0.1 секунду
                usleep(100000);

                // Проверяет, есть ли файл success.txt в текущий директории
                if (access(filename, F_OK) != -1) {
                    // Если файл есть, то пользовательский ввод был совершен - родительский
                    // процесс завершается
                    exit(0);
                }
            }
        }
    }
}
```



```
    }  
    }  
    // Файла в директории нет и таймаут прошел - завершение дочернего процесса  
    kill(pid, SIGKILL);  
    }  
}  
return 0;  
}
```



**Министерство образования и науки Российской Федерации  
Федеральное государственное бюджетное образовательное  
учреждение  
высшего образования  
«Московский государственный технический университет  
имени Н.Э. Баумана  
(национальный исследовательский университет)»  
(МГТУ им. Н.Э. Баумана)**

---

Факультет «Робототехники и комплексной автоматизации»  
Кафедра «Системы автоматизированного проектирования» (РК-6)

**Отчет**

по лабораторной работе №2  
по курсу «Разработка программных систем»  
Вариант №23

Студент: Смирнова А. А.

Группа: РК6-6

Преподаватель: Федорук В. Г.

Дата: \_\_\_\_\_

Подпись: \_\_\_\_\_

## Оглавление

Текст программы	8
Текст задания на лабораторную работу	12
Текстовое описание структуры программы	13
Реализованный способ взаимодействия потоков	14
Описание основных используемых структур данных	14
Блок-схема программы	15
Пример результатов работы программы	16

## Текст задания на лабораторную работу

Разработать программу, моделирующую в реальном времени работу поточной линии, состоящей из  $N$  станков и обрабатывающей заготовки  $M$  типов. Каждая заготовка последовательно проходит обработку на всех станках линии. Времена обработки заготовок каждого типа (в секундах) на каждом станке линии задаются в виде прямоугольной матрицы размером  $N \times M$  в конфигурационном файле `line.cnf`. Первые 2 строки этого файла содержат числа  $N$  и  $M$ . Программа реализуется  $N+1$  потоком управления. Корневой поток порождает  $N$  потоков-"станков", передавая им информацию о временах обработки заготовок разного типа. Далее этот поток читает со стандартного ввода последовательность номеров типов заготовок и передает ее на вход первого потока-"станка". Потоки-"станки" имитируют обработку заготовок с помощью функции `sleep()` и передают номера типов обработанных заготовок потокам-приемникам. Предусмотреть вывод информации о ходе обработки заготовок.

## Текстовое описание структуры программы

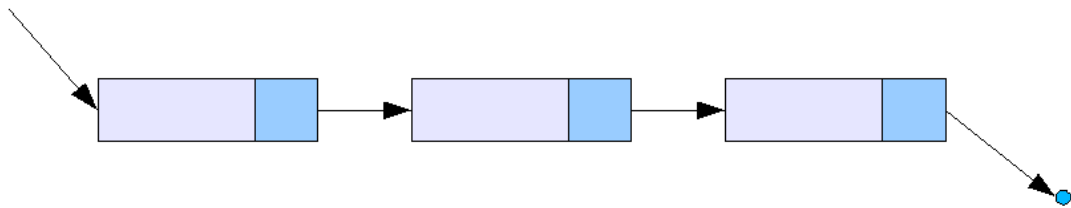
В программе используются 2 глобальных переменных - массив *times* и массив структур *queues*. Программа читает из файла *line.cnf* двумерный массив секунд *times*, содержащий время, нужное для обработки j-ой заготовки i-го станка. Далее создаются потоки для каждого станка и для каждого станка создается структура *ThreadContext*, которая хранит номер текущего потока, сколько всего было создано потоков и количество еще не обработанных заготовок для текущего потока. Для каждого потока вызывается функция *billet\_machine*, которая симулирует обработку детали. Логика *billet\_machine*: пока количество заготовок, которых надо обработать, больше нуля, поток делает *pop* из своей очереди задач. Если *pop* вернул NULL, это значит, что заготовок нет и поток засыпает на 0.1 секунду и продолжает ждать заготовок в цикле. Если *pop* вернул номер заготовки, то поток берет из массива *times* время для этой заготовки, обрабатывает ее (делает *sleep* на время обработки), уменьшает количество заготовок, которых нужно обработать этому станку. Если поток не последний, то он добавляет обработанную заготовку в очередь к следующему потоку. Когда поток обработал все заготовки, поток возвращает NULL и завершается.

## Реализованный способ взаимодействия потоков

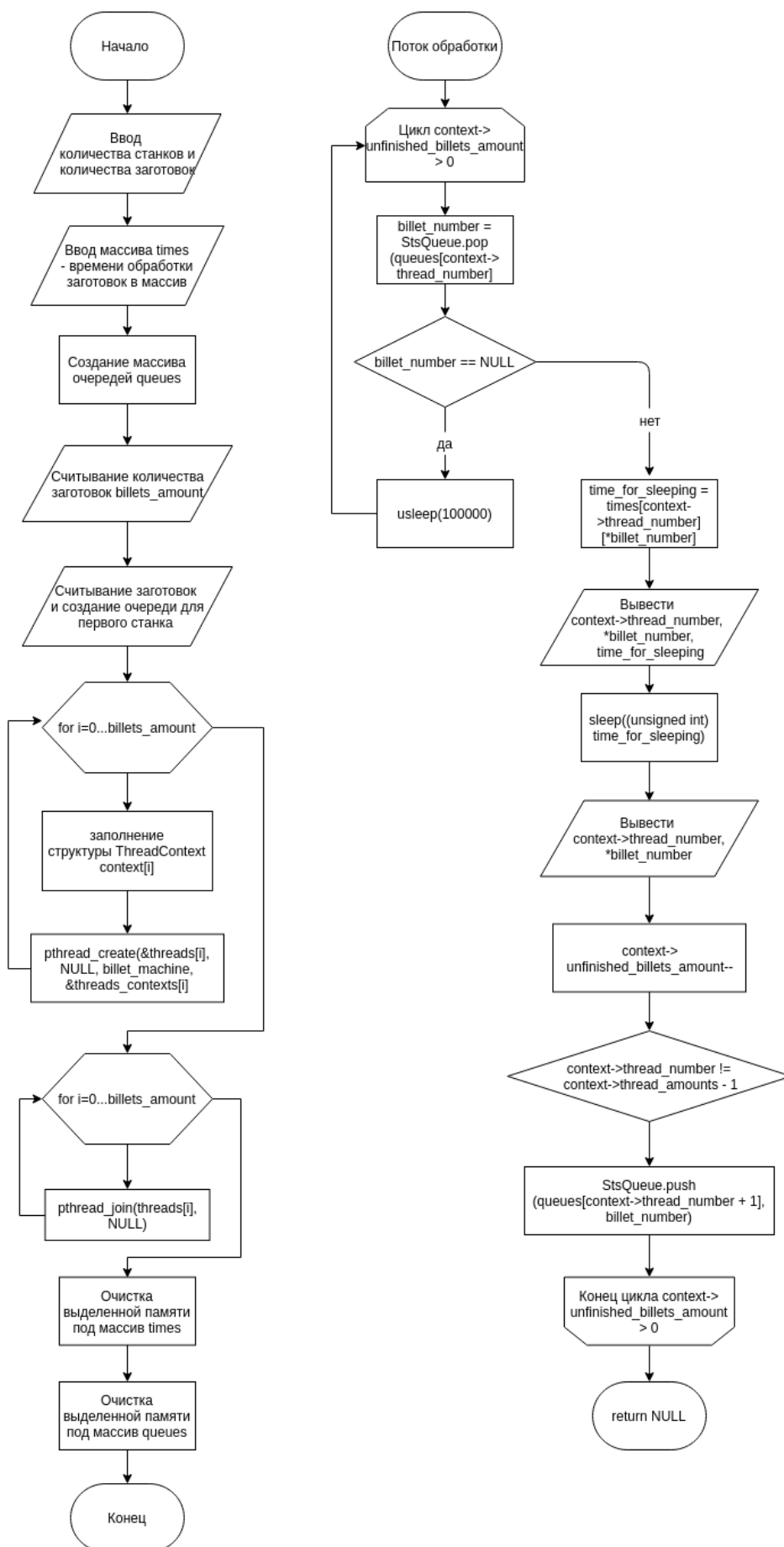
Потокобезопасная очередь в разделяемой памяти.

### Описание основных используемых структур данных

В качестве очереди задач для каждого потока использовался потокобезопасный линейный связный список, который представляет из себя структуру данных, состоящую из элементов одного типа, связанных между собой последовательно посредством указателей. Реализация списка представлена в файле `sts_queue.c`.



## Блок-схема программы



## Пример результатов работы программы

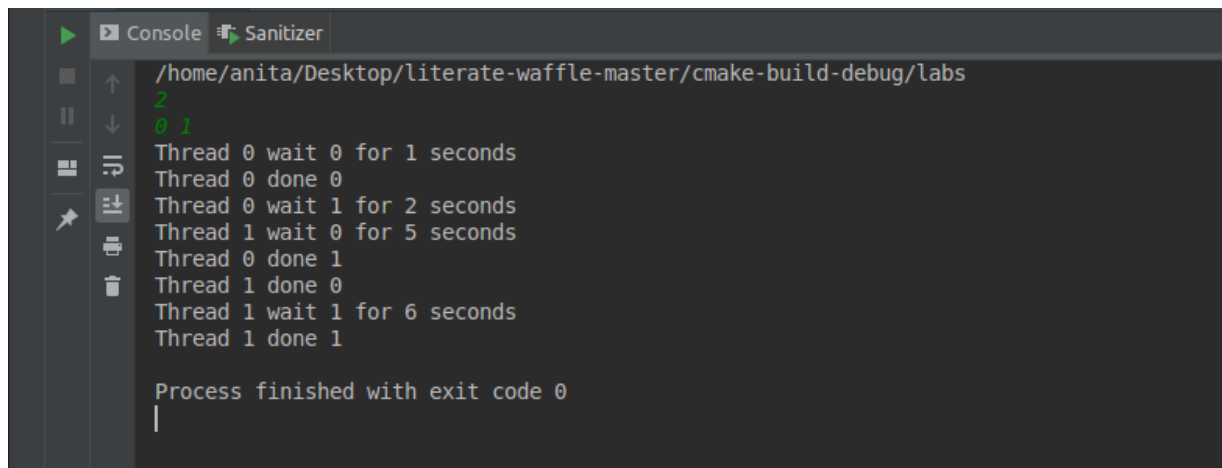
line.cnf

3 4

1 2 3 4

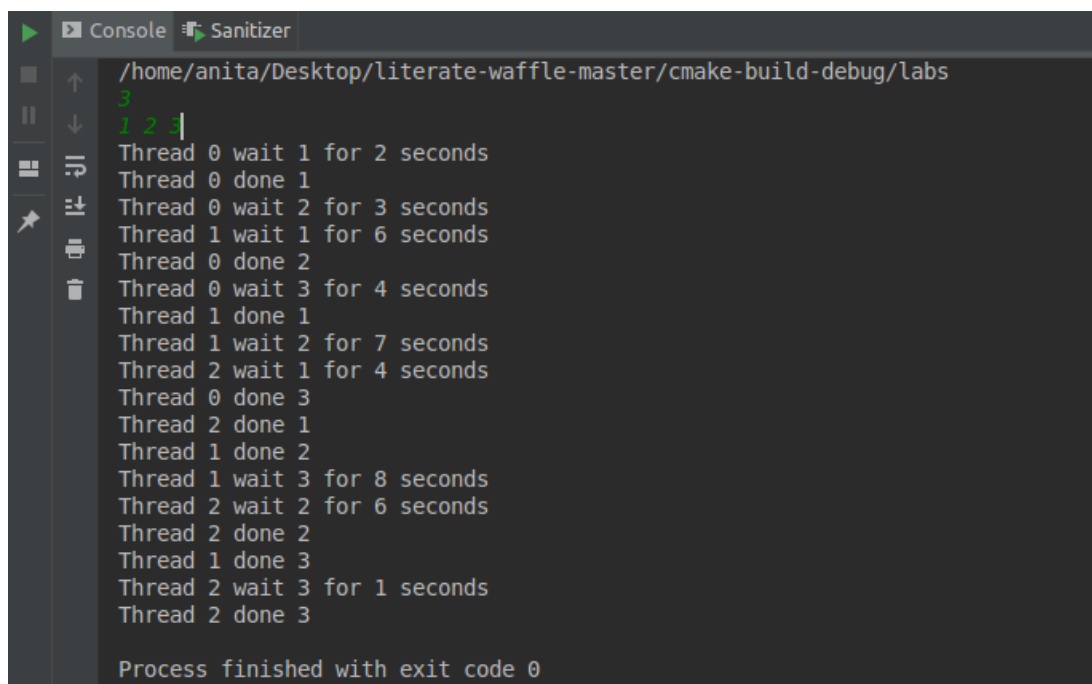
5 6 7 8

2 4 6 1



```
Console Sanitizer
/home/anita/Desktop/literate-waffle-master/cmake-build-debug/labs
2
0 1
Thread 0 wait 0 for 1 seconds
Thread 0 done 0
Thread 0 wait 1 for 2 seconds
Thread 1 wait 0 for 5 seconds
Thread 0 done 1
Thread 1 done 0
Thread 1 wait 1 for 6 seconds
Thread 1 done 1

Process finished with exit code 0
```



```
Console Sanitizer
/home/anita/Desktop/literate-waffle-master/cmake-build-debug/labs
3
1 2 3
Thread 0 wait 1 for 2 seconds
Thread 0 done 1
Thread 0 wait 2 for 3 seconds
Thread 1 wait 1 for 6 seconds
Thread 0 done 2
Thread 0 wait 3 for 4 seconds
Thread 1 done 1
Thread 1 wait 2 for 7 seconds
Thread 2 wait 1 for 4 seconds
Thread 0 done 3
Thread 2 done 1
Thread 1 done 2
Thread 1 wait 3 for 8 seconds
Thread 2 wait 2 for 6 seconds
Thread 2 done 2
Thread 1 done 3
Thread 2 wait 3 for 1 seconds
Thread 2 done 3

Process finished with exit code 0
```



## Текст программы

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <zconf.h>
#include "sts_queue.h"

// Структура, которая передается каждому потоку
typedef struct ThreadContext {
    // Номер текущего потока
    int thread_number;
    // Сколько было создано всего потоков
    int thread_amounts;
    // Количество еще не обработанных заготовок
    int unfinished_billets_amount;
} ThreadContext;

// Массив, хранящий время обработки j заготовок для каждого i-го станков
int **times = NULL;
// Массив "задач" = номеров заготовок для каждого станка
StsHeader **queues;

void *billet_machine(void *threadContext) {
    // преобразование к типу ThreadContext
    ThreadContext *context = (ThreadContext *) threadContext;

    // Пока остались необработанные заготовки
    while (context->unfinished_billets_amount > 0) {
        int *billet_number = StsQueue.pop(queues[context->thread_number]);

        // Если в очереди нет заготовок
        if (billet_number == NULL) {
            // Поток засыпает на 0.1 секунду
            usleep(100000);

            // Поток продолжает в цикле ждать появление заготовок в очереди
            continue;
        }
        // Если в очереди появилась заготовка
        // Получение времени, нужного для обработки заготовки
        int time_for_sleeping = times[context->thread_number][*billet_number];
        printf("Thread %d wait %d for %d seconds\n", context->thread_number, *billet_number,
            time_for_sleeping);

        // Обработка заготовки
        sleep((unsigned int) time_for_sleeping);
        printf("Thread %d done %d\n", context->thread_number, *billet_number);

        // Заготовка обработана - уменьшается количество заготовок, которых нужно
        // обработать станку
        context->unfinished_billets_amount--;

        // Если поток не последний
        if (context->thread_number != context->thread_amounts - 1) {
            // Добавление обработанной заготовки в очередь к следующему потоку
            StsQueue.push(queues[context->thread_number + 1], billet_number);
        }
    }
    return NULL;
}
```

```

int main() {
    FILE *fp = fopen("/home/anita/Desktop/labs/lab2/line.cnf", "r");
    int cols_amount = 0, rows_amount = 0;

    // rows_amount - количество станков, cols_amount - количество заготовок
    fscanf(fp, "%d%d", &rows_amount, &cols_amount);

    // Выделение памяти под двумерный массив times
    times = (int **) malloc(rows_amount * sizeof(int *));
    for (int i = 0; i < rows_amount; i++)
        times[i] = (int *) malloc(cols_amount * sizeof(int));

    // Считывание времени обработки заготовок в массив
    for (int i = 0; i < rows_amount; i++) {
        for (int j = 0; j < cols_amount; j++) {
            fscanf(fp, "%d", &times[i][j]);
        }
    }

    // Заккрытие файла
    fclose(fp);

    // Считывание количества заготовок
    int billets_amount = 0;
    fscanf(stdin, "%d", &billets_amount);

    // Создание массива очередей (каждому станку соответствует своя очередь)
    queues = (StsHeader **) malloc(billets_amount * sizeof(StsHeader *));
    for (int i = 0; i < billets_amount; i++)
        queues[i] = StsQueue.create();

    // Считывание заготовок и создание очереди для 1-го станка
    int billets[billets_amount];
    for (int i = 0; i < billets_amount; i++) {
        fscanf(stdin, "%d", &billets[i]);
        StsQueue.push(queues[0], &billets[i]);
    }

    // Массив потоков
    pthread_t threads[billets_amount];
    // Массив контекстов(информации, нужной для) для каждого потока
    ThreadContext threads_contexts[billets_amount];

    // Создание потока и контекста, содержащего информацию о потоке
    for (int i = 0; i < billets_amount; i++) {
        threads_contexts[i].thread_amounts = billets_amount;
        threads_contexts[i].thread_number = i;
        threads_contexts[i].unfinished_billets_amount = billets_amount;
        if (pthread_create(&threads[i], NULL, billet_machine, &threads_contexts[i])) {
            fprintf(stderr, "Error creating thread\n");
            return 1;
        }
    }

    // Join потоков для того, чтобы дождаться завершения всех потоков
    for (int i = 0; i < billets_amount; i++) {
        if (pthread_join(threads[i], NULL)) {
            fprintf(stderr, "Error joining thread\n");
            return 2;
        }
    }

    // Очистка выделенной памяти под массив times

```

```

    for (int i = 0; i < rows_amount; i++)
        free(times[i]);
    free(times);

    // Очистка выделенной памяти под массив queues
    for (int i = 0; i < billets_amount; i++)
        StsQueue.destroy(queues[i]);
    free(queues);
    return 0;
}

```

## sts\_queue.h

```

#ifndef STS_QUEUE_H
#define STS_QUEUE_H

typedef struct StsHeader StsHeader;

typedef struct {
    StsHeader* (* const create)();
    void (* const destroy)(StsHeader *handle);
    void (* const push)(StsHeader *handle, void *elem);
    void* (* const pop)(StsHeader *handle);
} _StsQueue;

extern _StsQueue const StsQueue;

```

## sts\_queue.c

```

#include "sts_queue.h"
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <printf.h>

typedef struct StsElement {
    void *next;
    void *value;
} StsElement;

struct StsHeader {
    StsElement *head;
    StsElement *tail;
    pthread_mutex_t *mutex;
};

static StsHeader* create();
static StsHeader* create() {
    StsHeader *handle = malloc(sizeof(*handle));
    handle->head = NULL;
    handle->tail = NULL;

    pthread_mutex_t *mutex = malloc(sizeof(*mutex));
    handle->mutex = mutex;

    return handle;
}

static void destroy(StsHeader *header);
static void destroy(StsHeader *header) {

```

```

free(header->mutex);
free(header);
header = NULL;
}

```

```

static void push(StsHeader *header, void *elem);
static void push(StsHeader *header, void *elem) {
    // Создание нового элемента
    StsElement *element = malloc(sizeof(*element));
    element->value = elem;
    element->next = NULL;

```

```

pthread_mutex_lock(header->mutex);
// Если список пустой
if (header->head == NULL) {
    header->head = element;
    header->tail = element;
} else {
    // Добавление элемента
    StsElement* oldTail = header->tail;
    oldTail->next = element;
    header->tail = element;
}
pthread_mutex_unlock(header->mutex);
}

```

```

static void* pop(StsHeader *header);
static void* pop(StsHeader *header) {
    pthread_mutex_lock(header->mutex);
    StsElement *head = header->head;

```

```

// Если список пустой
if (head == NULL) {
    pthread_mutex_unlock(header->mutex);
    return NULL;
} else {
    // Rewire
    header->head = head->next;

```

```

// Получение указателя на головной элемент и очистка памяти под удаленный элемент
void *value = head->value;
free(head);

pthread_mutex_unlock(header->mutex);
return value;
}
}

```

```

void print(StsHeader *header);
void print(StsHeader *header) {
    pthread_mutex_lock(header->mutex);
    StsElement *head = header->head;

    if (head == NULL) {
        pthread_mutex_unlock(header->mutex);
    } else {
        while(head->head != NULL) {
            printf("%d", *((int*)head->value));
            head->head = head->next;
        }
    }
    pthread_mutex_unlock(header->mutex);
}

```

```
}
```

```
_StsQueue const StsQueue = {  
  create,  
  destroy,  
  push,  
  pop  
};
```



**Министерство образования и науки Российской Федерации  
Федеральное государственное бюджетное образовательное  
учреждение  
высшего образования  
«Московский государственный технический университет  
имени Н.Э. Баумана  
(национальный исследовательский университет)»  
(МГТУ им. Н.Э. Баумана)**

---

Факультет «Робототехники и комплексной автоматизации»  
Кафедра «Системы автоматизированного проектирования» (РК-6)

**Отчет**

по лабораторной работе №3  
по курсу «Разработка программных систем»  
Вариант №5

Студент: Смирнова А. А.

Группа: РК6-6

Преподаватель: Федорук В. Г.

Дата: \_\_\_\_\_

Подпись: \_\_\_\_\_

## Оглавление

Текст задания на лабораторную работу	24
Описание прикладного протокола сетевого взаимодействия	25
Блок-схема программы	26
Примеры результатов работы программы	28
Текст программы	29

## Текст задания на лабораторную работу

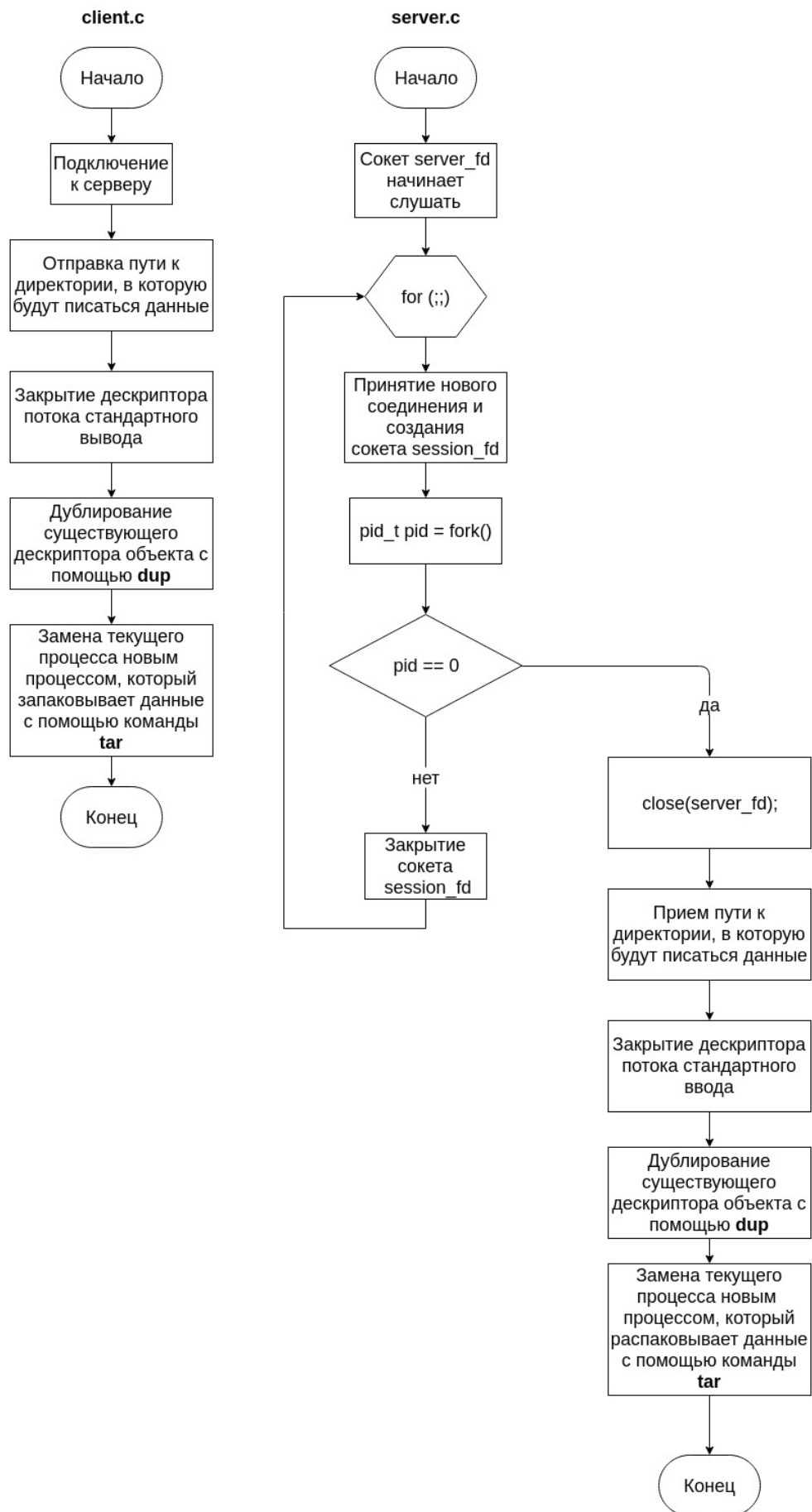
Разработать клиент-серверное приложение копирования файла (или поддерева файловой системы) с узла-клиента на узел-сервер в указанный каталог (аналог стандартной UNIX-команды `rsc`). Команда, выполняемая на стороне клиента, имеет следующий вид: `cprem path.to.src.file host@path.to.dst.dir`.



## Описание прикладного протокола сетевого взаимодействия

- 1) Устанавливается соединение
- 2) Посылается директория, в которую нужно прислать содержимое архива
- 3) Посылается содержимое архива
- 4) Закрывается соединение

## Блок-схема программы



*Пояснения:*

На клиенте используется функция `close(1)` для того, чтобы закрыть дескриптор потока стандартного вывода. Далее вызывается функция `dup`, которой как аргумент подается файловый дескриптор сокета. Новый дескриптор, возвращаемый вызовом `dup`, является самым низким нумерованным дескриптором, который в настоящее время не используется процессом, то есть это файловый дескриптор 1. Далее функция `execvp` из семейства функций `exec` заменяет текущий образ процесса новым образом процесса, в котором выполняется отправление результата архивирования в файловый дескриптор 1, то есть в файловый дескриптор сокета. Логика сервера построена аналогично с помощью функций `close`, `dup`, `execvp`.

## Примеры результатов работы программы

Содержимое директории test, которое будет передаваться на сервер:

```
anita@anita-laptop: ~/Desktop/test
anita@anita-laptop:~/Desktop/test$ ls
test1.txt  test2.txt
```

Содержимое директории server до отправки данных

```
anita@anita-laptop: ~/Desktop/server
anita@anita-laptop:~/Desktop/server$ ls
anita@anita-laptop:~/Desktop/server$
```

Работа клиента - выведены файлы, которые были переданы

```
cprem_server x cprem x
Console Sanitizer
/home/anita/Desktop/literate-waffle/lab3/cmake-build-debug/cprem /home/anita/Desktop/test 127.0.0.1@/home/anita/Desktop/server
test/
test/test2.txt
test/test1.txt
Process finished with exit code 0
```

Работа сервера - выведено сообщение о том, в какую директорию были присланы файлы и названия файлов, которые были переданы.

```
Run: cprem_server x cprem x
Console Sanitizer
/home/anita/Desktop/literate-waffle/lab3/cmake-build-debug/cprem_server
Recieved destination directory /home/anita/Desktop/server, len 26
test/
test/test2.txt
test/test1.txt
```

Содержание директории server после работы программы.

```
anita@anita-laptop: ~/Desktop/server
anita@anita-laptop:~/Desktop/server$ ls
test
anita@anita-laptop:~/Desktop/server$ ls test
test1.txt  test2.txt
anita@anita-laptop:~/Desktop/server$
```

## Текст программы

### client.c

```
#include <libgen.h>
#include <dirent.h>
#include "errno.h"
#include "netdb.h"
#include "stdlib.h"
#include "string.h"
#include "unistd.h"

#include "helpers.h"

int main(int argc, char **argv) {
    die_if(argc != 3, "incorrect usage: waiting for cprem path.to.src.file
    host[:port]@path.to.dst.dir");

    // Преобразование host@path.to.dst.dir -> host, path.to.dst.dir
    char *token = NULL, *sep_ptr = NULL, *host_and_dst_path = NULL;
    host_and_dst_path = sep_ptr = strdup(argv[2]);
    token = strsep(&sep_ptr, "@");
    die_if(token == NULL, "invalid destination argument, should be host[:port]@path.to.dst.dir");
    char *hostname_and_port = strdup(token);
    token = strsep(&sep_ptr, "@");
    die_if(token == NULL, "invalid destination argument, should be host[:port]@path.to.dst.dir");
    char dst_path[DST_DIR_SIZE];
    strncpy(dst_path, token, DST_DIR_SIZE);

    // Преобразование hostname:port -> hostname, port
    sep_ptr = hostname_and_port;
    token = strsep(&sep_ptr, ":");
    die_if(token == NULL, "invalid destination argument, should be host[:port]@path.to.dst.dir");
    char *hostname = strdup(hostname_and_port);
    token = strsep(&sep_ptr, ":");
    char *port = token == NULL ? NULL : strdup(token);

    // Преобразование /path/to/src/file -> /path/to/src, file
    char *file_path_cpy = strdup(argv[1]);
    char *file_path_cpy2 = strdup(argv[1]);
    char *file_name = basename(file_path_cpy);
    char *dir_name = dirname(file_path_cpy2);

    // Подключение к серверу
    struct addrinfo *addr = resolve_addrinfo(hostname, port == NULL ? DEFAULT_PORT :
    port);
    int client_fd = socket(addr->ai_family, addr->ai_socktype, addr->ai_protocol);
    die_if(client_fd < 0, "failed to create socket %s", strerror(errno));
    int connect_res = connect(client_fd, addr->ai_addr, addr->ai_addrlen);
    die_if(connect_res < 0, "failed to connect to server socket %s", strerror(errno));

    // Отправка пути к директории, в которую будут писаться данные
    int n = write(client_fd, dst_path, strlen(dst_path));
    die_if(n < 0, "failed to write to socket %s", strerror(errno));
```

```

// Смена директории на ту, из которой нужно передать данные
int chdir_res = chdir(dir_name);
die_if(chdir_res == -1, "failed to chdir %s", strerror(errno));

// Заккрытие дескриптора потока стандартного вывода
close(1);

// Дублирование существующего дескриптора объекта
int dup_res = dup(client_fd);

// Теперь дескриптор сокета является дескриптором 1, т.е. дескриптором вывода
die_if(dup_res == -1, "failed to dup %s", strerror(errno));

freeaddrinfo(addr);
free(host_and_dst_path);
free(hostname_and_port);
free(hostname);
free(port);
free(file_path_cpy);
free(file_path_cpy2);

// Замена текущего образа процесса новым образом процесса с командой tar
char *tar_argv[] = {"tar", "-czv", file_name};
execvp(tar_argv[0], tar_argv);
die_if(dup_res == -1, "failed to exec tar %s", strerror(errno));

return 0;
}

```

## server.c

```

#include "errno.h"
#include "fcntl.h"
#include "netdb.h"
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
#include "unistd.h"
#include "helpers.h"

static void handle_session(int session_fd) {
    char dst_dir[DST_DIR_SIZE];

    // Чтение директории, в которую будут писаться данные
    int read_count = read(session_fd, dst_dir, DST_DIR_SIZE);
    die_if(read_count < 0, "failed to read from socket %s", strerror(errno));
    dst_dir[read_count] = '\0';
    printf("Recieved destination directory %s, len %d\n", dst_dir, read_count);

    int chdir_res = chdir(dst_dir);
    die_if(chdir_res == -1, "failed to chdir %s", strerror(errno));

    // Замена дескриптора стандартного ввода процесса на дескриптор сокета
    close(0);
}

```

```

int dup_res = dup(session_fd);
die_if(dup_res == -1, "failed to dup %s", strerror(errno));

// Замена текущего образа процесса новым образом процесса с командой tar
char *argv[2] = {"tar", "-xzv"};
execvp(argv[0], argv);
die_if(dup_res == -1, "failed to exec tar %s", strerror(errno));
}

int main(int argc, char **argv) {
    // Сокет начинает слушать
    struct addrinfo *addr = resolve_addrinfo(argc > 1 ? argv[1] : "0", argc > 2 ? argv[2] :
DEFAULT_PORT);
    int server_fd = socket(addr->ai_family, addr->ai_socktype, addr->ai_protocol);
    die_if(server_fd == -1, "failed to create socket %s", strerror(errno));
    int reuseaddr = 1;
    int setsockopt_res = setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &reuseaddr,
sizeof(reuseaddr));
    die_if(setsockopt_res != 0, "failed to set reuseaddr %s", strerror(errno));
    int bind_res = bind(server_fd, addr->ai_addr, addr->ai_addrlen);
    die_if(bind_res != 0, "failed to bind socket %s", strerror(errno));
    int listen_res = listen(server_fd, SOMAXCONN);
    die_if(listen_res != 0, "failed to listen for connections %s", strerror(errno));
    freeaddrinfo(addr);

    // Прием входящих запросов
    for (;;) {
        int session_fd = accept(server_fd, 0, 0);
        die_if(session_fd == -1, "failed to accept connection %s", strerror(errno));

        pid_t pid = fork();
        die_if(pid == -1, "failed to create child process %s", strerror(errno));
        if (pid == 0) {
            close(server_fd);

            // Обработка сессии
            handle_session(session_fd);
            close(session_fd);
            exit(0);
        } else {
            close(session_fd);
        }
    }
}

```

## helper.c

```
#include <netdb.h>
#include <stdbool.h>
#include "helpers.h"

#include "stdio.h"
#include "stdlib.h"
#include "string.h"
#include "zconf.h"

// Обработка ошибок и вывод сообщения об ошибке
void die_if(bool fail, const char *format, ...) {
    if (!fail)
        return;

    va_list vars;
    va_start(vars, format);
    vfprintf(stderr, format, vars);
    fprintf(stderr, ".\n");
    va_end(vars);
    exit(1);
}

// Заполнение структуры addrinfo
struct addrinfo *resolve_addrinfo(const char *hostname, const char *port) {
    struct addrinfo hints;
    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_protocol = 0;
    hints.ai_flags = AI_PASSIVE | AI_ADDRCONFIG;

    struct addrinfo *addr = NULL;
    int err = getaddrinfo(hostname, port, &hints, &addr);
    die_if(err != 0, "failed to resolve server socket address (err=%d)", err);
    return addr;
}
```





**Министерство образования и науки Российской Федерации  
Федеральное государственное бюджетное образовательное  
учреждение  
высшего образования  
«Московский государственный технический университет  
имени Н.Э. Баумана  
(национальный исследовательский университет)»  
(МГТУ им. Н.Э. Баумана)**

---

Факультет «Робототехники и комплексной автоматизации»  
Кафедра «Системы автоматизированного проектирования» (РК-6)

**Отчет**

по лабораторной работе №4  
по курсу «Разработка программных систем»  
Вариант №1

Студент: Смирнова А. А.  
Группа: РК6-6  
Преподаватель: Федорук В. Г.

Дата: \_\_\_\_\_  
Подпись: \_\_\_\_\_

## Оглавление

Текст задания на лаб. работу	35
Описание структуры программы и реализованных способов взаимодействия процессов	36
Описание основных используемых структур данных	37
Блок-схема программы	40
Примеры результатов работы программы	41
Текст программы	42

## Текст задания на лаб. работу

Разработать средствами MPI параллельную программу решения (численного интегрирования) одномерной **нестационарной** краевой задачи методом конечных разностей с использованием **явной** вычислительной схемы.

Дан цилиндрический стержень длиной  $L$  и площадью поперечного сечения  $S$ . Цилиндрическая поверхность стержня теплоизолированная. На торцевых поверхностях стержня слева и справа могут иметь место граничные условия первого и второго родов. Распределение поля температур по длине стержня описывается уравнением теплопроводности

$$\frac{dT}{dt} = aT \cdot \frac{d^2 T}{dx^2} + gT, \text{ где}$$

$aT = \lambda / (CT \cdot \rho)$  - коэффициент температуропроводности;

$\lambda$  - коэффициент теплопроводности среды;

$CT$  - удельная теплоемкость единицы массы;

$\rho$  - плотность среды;

$gT = GT / (CT \cdot \rho)$  - приведенная скорость превращения тепловой энергии в другие виды энергии, в нашем случае  $gT = 0$ .

## Описание структуры программы и реализованных способов взаимодействия процессов

В начале программы используются базовые функции MPI:

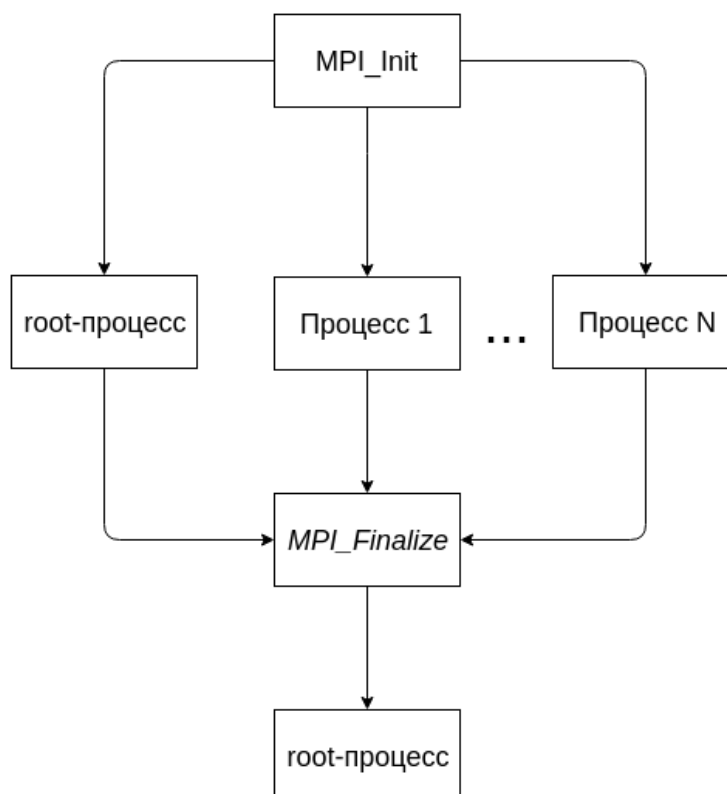
MPI\_Init – инициализация коммуникационных средств MPI.

MPI\_Comm\_size – определение общего количества параллельных процессов в группе осуществляется функцией.

MPI\_Comm\_rank – определение идентификатора (номера) процесса.

В родительском процессе выделяется память под одномерный массив T с граничными условиями первого рода. Затем с помощью функции MPI\_Scatter() данные распределяются по другим процессам. В начале вычислительного цикла процессы обмениваются данными между соседними полосами с помощью функции MPI\_Sendrecv(). Подробное описание функции содержится в следующем пункте.

После этого каждый процесс занимается вычислениями соответствующих ему полос. После выхода из вычислительного цикла данные собираются в родительский процесс с помощью функции MPI\_Gather(). Структура работы программы показана ниже.



В вычислительном цикле в родительском процессе происходит вывод производится в файл .dat для программы Gnuplot. Также программа

записывает файл `my_graph` с параметрами построения графика и анимации, который используется для запуска Gnuplot. Программа завершается функцией `MPI_Finalize` – нормальное завершение обменов в MPI.

## Описание основных используемых структур данных

Изначально создается одномерный массив  $T$  размером  $X$ , где  $X$  – количество узлов балки. В это массиве задаются начальные условия. Далее значения из этого массива распределяются по процессам с помощью функции `MPI_Scatter()`.

Функция `MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)` разбивает сообщение из буфера отправки процесса `root` на равные части размером `sendcount` и посылает  $i$  часть в буфер приема процесса с номером  $i$ . Процесс `root` использует оба буфера (буфер отправки и буфер приема), поэтому в вызываемой им подпрограмме все параметры являются существенными. Остальные процессы с коммунитором `comm` являются только получателями, поэтому для них параметры, которые определяют буфер отправки, не являются существенными.

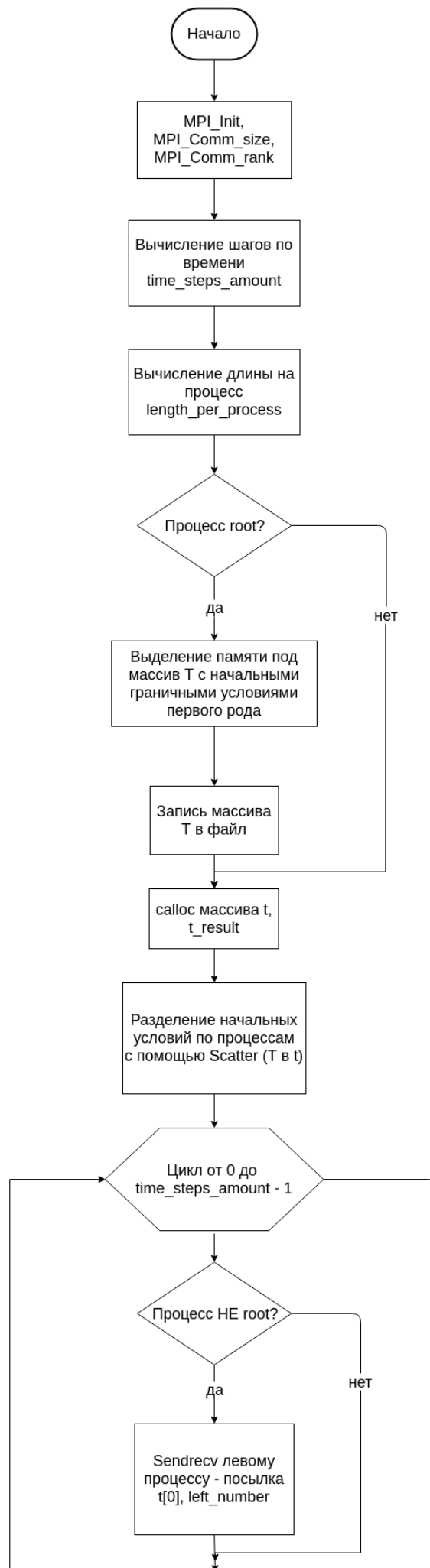
В результате работы Scatter-а исходные данные из массива  $T$  попадают в массивы `double t` размера `length_per_process`, где `length_per_process = X / total`, `total` – число процессов. Массивы  $t$  уникальны и хранятся в адресном пространстве своего процесса. Массивы  $t$  хранят соответствующие процессу полосы температур. После того, как каждый процесс получил свой массив  $t$ , вычисляется массив `t_result` по формуле:  $T_{j+1}^i = a_T * (T_{j+1}^i - 2 * T_j^i + T_{j-1}^i) * h_t / h_x^2 + T_j^i$

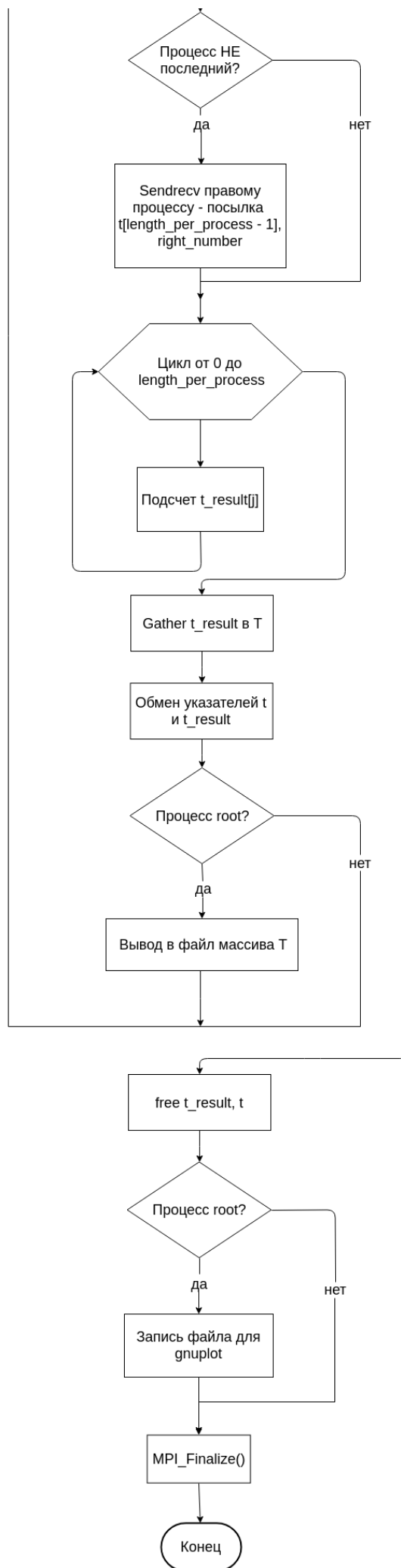
Данная формула использовалась в коде, как:

$$t\_result[j] = Lamb * (t[j + 1] - 2 * t[j] + t[j - 1]) * dT + t[j]$$

После вычислений данные из массивов `t_result` собираются в массив  $T$  с помощью функции `MPI_Gather()`. Эта функция производит сборку блоков данных, посылаемых всеми процессами группы, в один массив процесса с номером `root`. Длина блоков предполагается одинаковой. Объединение происходит в порядке увеличения номеров процессов-отправителей. То есть данные, посланные процессом  $i$  из своего буфера `sendbuf`, помещаются в  $i$ -ю порцию буфера `recvbuf` процесса `root`.

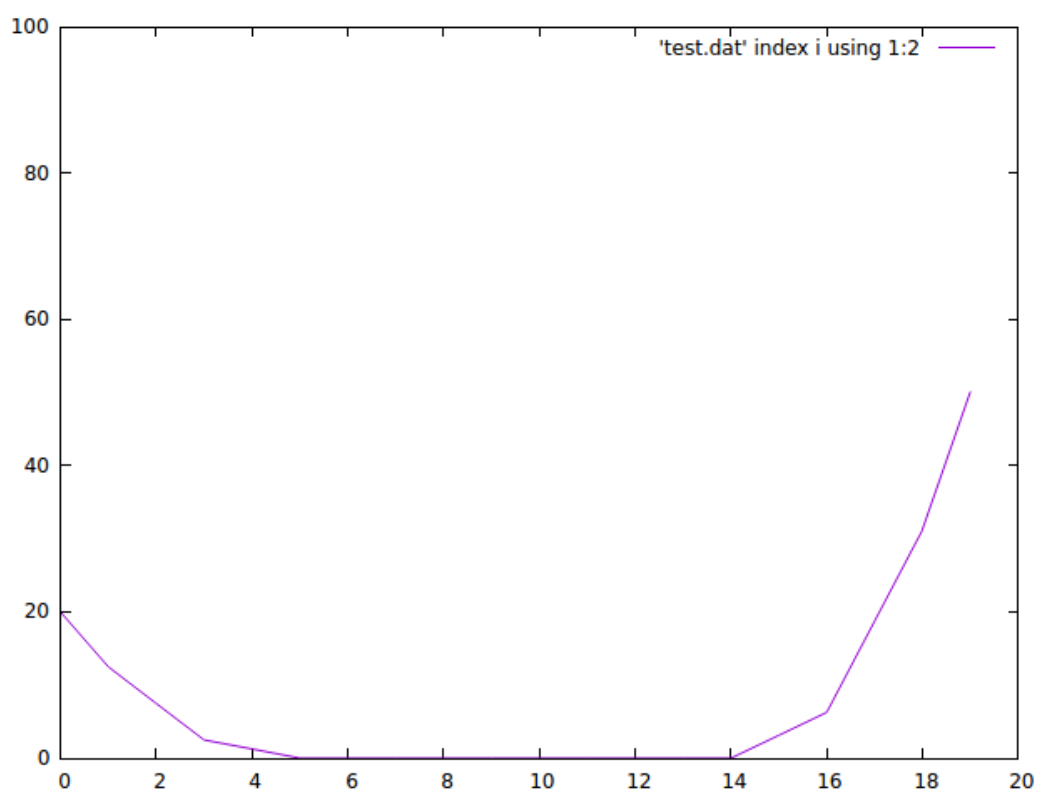
## Блок-схема программы







## Примеры результатов работы программы



## Текст программы

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

// Длина стержня
#define L 20
// Параметр теплопроводности
#define Lamb 0.5
// Временные узлы
#define TIME 10
// Шаг по времени
#define dT 1

// Граничные условия первого рода
#define LEFT_CONDITION 20
#define RIGHT_CONDITION 50

int main(int argc, char **argv) {
    remove("test.dat");
    remove("my_graph");

    // myrank - номер процесса
    // total - число процессов
    int myrank, total;

    // Инициализация
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &total);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    // Число шагов по времени
    int time_steps_amount = TIME / dT;

    // Длины на процесс
    int length_per_process = L / total;

    // Текущая полоса
    double *T = NULL;

    // В нулевом процессе
    if (!myrank) {
        // Выделение памяти под исходные данные
        T = (double *) calloc(L, sizeof(double));

        // Граничные условия первого рода
        T[0] = LEFT_CONDITION;
        T[L - 1] = RIGHT_CONDITION;

        FILE *file = fopen("test.dat", "a");
        for (int j = 0; j < L; j++) {
            fprintf(file, "%d %f\n", j, T[j]);
        }
        fprintf(file, "\n\n");
    }
}
```

```

    fclose(file);
}

double *t = (double *) calloc(length_per_process, sizeof(double));
double *t_result = (double *) calloc(length_per_process, sizeof(double));

// Разделение начальных условий по процессам
MPI_Scatter((void *) T,
            length_per_process,
            MPI_DOUBLE,
            (void *) t,
            length_per_process,
            MPI_DOUBLE,
            0,
            MPI_COMM_WORLD);

for (int i = 0; i < time_steps_amount - 1; i++) {
    // Номера процессов соседних полос (слева от текущей полосой и справа)
    int left = myrank - 1;
    int right = myrank + 1;

    // Значение числа из соседней полосы (справа и слева), необходимое для
    // подсчета температуры в текущей полосе
    double left_number = 0, right_number = 0;

    MPI_Status s1, s2;

    // У первой полосы нет соседней левой, есть только правая
    if (myrank != 0) {
        MPI_Sendrecv(&t[0],
                    1,
                    MPI_DOUBLE,
                    left,
                    0,
                    &left_number,
                    1,
                    MPI_DOUBLE,
                    left,
                    MPI_ANY_TAG,
                    MPI_COMM_WORLD,
                    &s2);
    }
    // У последней полосы нет соседней правой, есть только левая
    if (myrank != total - 1) {
        MPI_Sendrecv(&t[length_per_process - 1], // Адрес начала
                                                             // Адрес начала
                                                             // Число посылаемых элементов
                                                             // Тип посылаемых элементов
                                                             // Номер процесса-получателя в группе
                                                             // Идентификатор сообщения
                                                             // Адрес начала расположения
                                                             // Максимальное число принимаемых
                                                             // тип элементов принимаемого сообщения;
                                                             // номер процесса-отправителя
        MPI_SENDRECV,
        1,
        MPI_DOUBLE,
        right,
        0,
        &right_number,
        1,
        MPI_DOUBLE,
        right,
        MPI_ANY_TAG,
        MPI_COMM_WORLD,
        &s1);
    }
}

```

```

        MPI_ANY_TAG,                // идентификатор принимаемого
сообщения
        MPI_COMM_WORLD,            // коммунитор области связи
        &s1);                      // атрибуты принятого сообщения.
    }

    for (int j = 0; j < length_per_process; j++) {
        // Левая граница
        if (j == 0 && !myrank) {
            t_result[j] = LEFT_CONDITION;
        }
        // Правая граница
        else if (j == length_per_process - 1 && myrank == total - 1) {
            t_result[j] = RIGHT_CONDITION;
        }
        // Остальные случаи
        else {
            double current_left = t[j - 1];
            double current_right = t[j + 1];
            t_result[j] = Lamb * (current_right - 2 * t[j] + current_left) * dT + t[j];
        }
    }
    MPI_Gather((void *) (t_result),
               length_per_process,
               MPI_DOUBLE,
               (void *) (T),
               length_per_process,
               MPI_DOUBLE,
               0,
               MPI_COMM_WORLD);
    double *tmp = t_result;
    t_result = t;
    t = tmp;

    if (!myrank) {
        FILE *file = fopen("test.dat", "a");
        for (int j = 0; j < L; j++) {
            fprintf(file, "%d %f\n", j, T[j]);
        }
        fprintf(file, "\n\n");
        fclose(file);
    }
    free(t);
    free(t_result);

    if (!myrank) {
        FILE *file2 = fopen("my_graph", "w");
        fprintf(file2, "set yrange [0:100]\n");
        fprintf(file2, "do for [i=0:%d]{\n", length_per_process - 1);
        fprintf(file2, "plot 'test.dat' index i using 1:2 with lines\n");
        fprintf(file2, "pause 0.5}\n pause -1");
        fclose(file2);
    }
    MPI_Finalize();
}

```