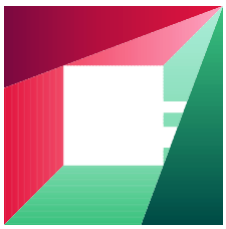
МИНИСТЕРСТВО НАУКИ
И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное
образовательное учреждение высшего образования
«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

НГТУ  ЭТИ

Кафедра прикладной математики

Лабораторная работа № 1

по дисциплине «Параллельное программирование»

ПРОГРАММИРОВАНИЕ НЕЗАВИСИМЫХ ПОТОКОВ

Бригада 6 — ПМИ-71 АНТОНОВ С.
ПМИ-71 АРНОЛЬД Э.
ПМИ-71 КАЙЛЬ Д.

Преподаватель — 4
ЩУКИН Г. А.
ГОРОДНИЧЕВ М. А.

Новосибирск

Задание

1. Изучить теоретическую часть из раздела 1 настоящего пособия.

2. Записать программу из примера 1. Скомпилировать ее и проверить корректность работы.

3. Доработать программу с целью поддержки n потоков.

4. Оценить стоимость запуска одного потока операционной системой. Изменяя количество операций (можно использовать любую

арифметическую операцию), которые исполняет функция потока,

определить такое их количество, чтобы порождение потока было

оправданным.

5. Добавить в программу возможность запуска потоков с разными

атрибутами (см. пример 2).

6. Добавить в программу возможность передавать в поток сразу несколько параметров (см. пример 3).

7. Добавить в функцию потока возможность вывода информации о

всех параметрах потока, с которыми он был создан.

8. Разработать программу, которая обеспечивает параллельное

применение заданной функции к каждому элементу массива. Размер

массива, применяемая функция и количество потоков задаются динамически.

9. Выполнить задание для самостоятельной работы (вариант согласовывается с преподавателем), по результатам подготовить отчет.

Задание для самостоятельной работы

1 Создать упрощенный HTTP-сервер, отвечающий на любой запрос клиента (например, браузера) строкой «Request number <номер запроса> has been processed», где под номером запроса понимается порядковый номер, присвоенный запросу сервером. Нумерация начинается с единицы. Доработать однопоточную версию сервера. Обработка каждого запроса выполняется в отдельном потоке: при получении запроса создается новый поток для его обработки, после отправки результата клиенту поток завершает свою работу. Соединение с клиентом закрывается сразу после обработки запроса.

2 Оценить производительность сервера с помощью утилиты ab, входящей в комплект поставки веб-сервера Apache.

3 Оценить максимальное количество потоков, с которым может работать сервер, для различных размеров стека по умолчанию (2 Мбайт, 1 Мбайт, 512 Кбайт).

4 Добавить в обработчик запроса от клиента запуск простейшего PHP-скрипта, возвращающего версию PHP (<?php echo phpversion();?>). Вернуть номер версии клиенту. Оценить изменение производительности сервера с помощью утилиты ab.

Из вышеперечисленных 1-8 заданий можно сформировать два:

1. Оценить стоимость запуска одного потока операционной системой. Изменяя количество операций (можно использовать любую арифметическую операцию), которые исполняет функция потока, определить такое их количество, чтобы порождение потока было оправданным.
2. Разработать программу, которая обеспечивает параллельное применение заданной функции к каждому элементу массива. Размер массива, применяемая функция и количество потоков задаются динамически.

Программа

Arrayprocessing

arrayprocessing.c

```
#include "arrayprocessing.h"
#include <string.h>
#include <pthread.h>


// Функция обработки ошибок
#define   throwErr(msg)   do   {   \
    fprintf(stderr,  "%s\n",  msg);  \
    exit(EXIT_FAILURE);              \
} while (0)

// Параметры потока (блок данных для обработки)
typedef struct Chunk {
    double* A;          // Указатель на массив
    size_t size;        // Размер массива

    func_ptr func;      // Функция обработки элемента
} Chunk;

static void* threadFunc(void* arg) {
    Chunk* chunk = (Chunk*)arg;

    for (size_t i = 0; i != chunk->size; ++i)
        chunk->A[i] = chunk->func(chunk->A[i]);

    pthread_exit(NULL);
}

// Обработка элементов массива
void arrayProcessing(dou-
ble* A, size_t size, func_ptr func, uint8_t threads_count) {
    // Создаём необходимое количество потоков
    pthread_t* threads = (pthread_t*)malloc(sizeof(pthread_t) * threads_count);
    if (threads == NULL)
```

```
        throwErr("Error: threads out of memmory!");

    Chunk* chunks = (Chunk*)malloc(sizeof(Chunk) * threads_count);
    if (chunks == NULL)
        throwErr("Error: chunks out of memmory!");

    // Определяем размер блоков и остаток по размеру массива
    size_t chunk_size = size / threads_count;
    uint8_t remainder = size % threads_count;

    int err = 0;
    size_t shift = 0;
    // Создаём потоки и разбиваем массив на блоки
    for (uint8_t i = 0; i != threads_count; ++i) {
        // Записываем в параметр потока ту часть массива,
        // которую он будет обрабатывать, и указатель на функцию обработки
        chunks[i] = (Chunk){A + shift, chunk_size + (i < remainder ? 1 : 0), func};

        shift += chunks[i].size;

        err = pthread_create(&threads[i], NULL, threadFunc, (void*)&chunks[i]);
        if (err != 0)
            throwErr("Error: cannot create a thread: ");
    }

    // Ожидаем завершения созданных потоков перед завершением работы программы
    for (uint8_t i = 0; i != threads_count; ++i) {
        err = pthread_join(threads[i], NULL);
        if (err != 0)
            throwErr("Error: cannot join a thread");
    }

    free(threads);
    free(chunks);
}
```

arrayprocessing.h

```
#ifndef ARRAYPROCESSING_H
#define ARRAYPROCESSING_H

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

// Указатель на функцию обработки элемента массива
typedef double (*func_ptr)(double);

// Создание массива
static inline double* arrayCreate(size_t size) {
    double* A = (double*)malloc(sizeof(double) * size);
    return A;
```

```
}

// Инициализация массива случайными числами
static inline void arrayRandInit(double* A, size_t size) {
    for (size_t i = 0; i != size; ++i)
        A[i] = rand() % size;
}

// Копирование массива
static inline void arrayCopy(double* dest, double* src, size_t size) {
    for (size_t i = 0; i != size; ++i)
        dest[i] = src[i];
}

// Вывод массива на экран
static inline void arryPrint(double* A, size_t size) {
    for (size_t i = 0; i != size; ++i)
        printf("%g ", A[i]);
    printf("\n");
}

// Обработка элементов массива
void arrayProcessing(double* A, size_t size, func_ptr func, uint8_t threads_count);

#endif
```

arrayprocessing_main.c

```
#include "arrayprocessing.h"

#include <stdio.h>
#include <math.h>
#include <time.h>

#define ARGS_COUNT 6

#define RESULT_FILENAME "result.txt"

#define BILLION 1.0E+9

// Функция вычета разности между временными величинами
#define clocktimeDifference(start, stop)            \
    1.0 * (stop.tv_sec - start.tv_sec) +            \
    1.0 * (stop.tv_nsec - start.tv_nsec) / BILLION

// Функции обработки элементов массива
static double sqrFunc(double val) {
    return val * val;
}

static double expFunc(double val) {
    return exp(val);
```

```
}

static double revFunc(double val) {
    return 1 / val;
}

// Функция расчёта времени обработки элементов массива на указанном количестве пото
ков,
// с указанным начальным и конечным размером массива
static void resultOutput(FILE* fp, uint8_t threads_count,
                         size_t array_size_min, size_t array_size_max,
                         uint8_t func_num, size_t measure_count) {
    // Определение функции обработки элемента массива
    func_ptr array_func = sqrFunc;
    switch (func_num) {
        case 0: array_func = sqrFunc; break;
        case 1: array_func = expFunc; break;
        case 2: array_func = revFunc; break;
    }

    // Составление таблицы
    fprintf(fp, "size:\tthreads: 1\tthreads: 2\tthreads: 3\tthreads: 4\n");

    // Увеличение размера массива
    for (size_t array_size = array_size_min; array_size < array_size_max; ar-
ray_size *= 10) {
        // Создание исходного массива и массива для многократной обработки,
        // для обеспечения одинаковых данных на всех испытаниях
        double* A_src = arrayCreate(array_size);
        double* A = arrayCreate(array_size);

        // Инициализация исходного массива
        arrayRandInit(A_src, array_size);

        fprintf(fp, "%zu\t", array_size);

        for (uint8_t i = 0; i != threads_count; ++i) {
            // Копирование данных из исходного массива в массив обработки
            arrayCopy(A, A_src, array_size);

            // Повторяем замеры указанное число раз
            double elapsed_time = 0;
            for (size_t j = 0; j != measure_count; ++j) {
                struct timespec start, stop;
                clock_gettime(CLOCK_MONOTONIC, &start);

                arrayProcessing(A, array_size, array_func, i + 1);

                clock_gettime(CLOCK_MONOTONIC, &stop);

                elapsed_time += clocktimeDifference(start, stop);
```

```
            }

            fprintf(fp, "%lf\t", elapsed_time / measure_count);
        }

        fprintf(fp, "\n");

        free(A_src);
        free(A);
    }
}

int main(int argc, char* argv[]) {
    if (argc < ARGS_COUNT) {
        fprintf(stderr, "Wrong number of arguments!\n");
        fprintf(stderr, "Enter: <threads count> <array size min> "
                        "<array size max> <func number> <measure count>\n");
        fprintf(stderr, "(Func number: 0 - sqr, 1 - cube, 2 - revers)\n");
        exit(EXIT_FAILURE);
    }

    srand(time(NULL));

    uint8_t threads_count = atoi(argv[1]);
    size_t array_size_min = atoi(argv[2]);
    size_t array_size_max = atoi(argv[3]);
    uint8_t func_num = atoi(argv[4]);
    size_t measure_count = atoi(argv[5]);

    FILE* fp = fopen(RESULT_FILENAME, "w");

    printf("Program execution...\n");
    resultOutput(fp, threads_count, array_size_min,
                 array_size_max, func_num, measure_count);
    printf("Done.\n");

    fclose(fp);

    return 0;
}
```

Makefile

```
DEFINES = -D_POSIX_C_SOURCE -D_BSD_SOURCE
CFLAGS  = -std=c99 -O2 -g $(DEFINES)
LIBS    = -lpthread -lm
TARGET  = arrayprocessing

all: $(TARGET) cleanTemp
```

```
$(TARGET): $(TARGET)_main.o $(TARGET).o
    gcc $(CFLAGS) -o $(TARGET) $(TARGET)_main.o $(TARGET).o $(LIBS)

$(TARGET)_main.o: $(TARGET)_main.c $(TARGET).h
    gcc $(CFLAGS) -c $(TARGET)_main.c

$(TARGET).o: $(TARGET).c $(TARGET).h
    gcc $(CFLAGS) -c $(TARGET).c

cleanTemp:
    rm -rf *.o

clean:
    rm -rf $(TARGET)
```

Результат работы первой задачи

```
[pmi-b7603@students arrayprocessing]$ cat result.txt
size:   threads: 1      threads: 2      threads: 3      threads: 4
10      0.002087        0.000256        0.000347        0.000225
```

size	Поток 1	Поток 2	Поток 3	Поток 4
10	0,002087	0,000256	0,000347	0,000225
10	0.001667	0.000211		
10	0.001660	0.000200	0.000252	

Threadtimestat

threadtimestat_main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <float.h>

#include "threadtimestat.h"

// Количество аргументов командной строки
#define ARGS_COUNT 4

// Выходной файл для результато тестирования
```

```c
#define RESULT_FILENAME "result.txt"

static void* threadFunc(void* arg) {
    ThreadArg* thread_arg = (ThreadArg*)arg;

    double a = 17;
    double b = 5.125013;

    // Расчёт времени обработки указанного кол-ва операций умножения
    struct timespec start, stop;
    clock_gettime(CLOCK_MONOTONIC, &start);

    for (size_t i = 0; i != thread_arg->op_count; ++i)
        a *= b;

    clock_gettime(CLOCK_MONOTONIC, &stop);
    thread_arg->elapsed_time = clocktimeDifference(start, stop);

    pthread_exit(NULL);
}

// Функция поиска оптимального количества операций для порождения потока,
// с выводом всех шагов поиска
static void resultOutput(FILE* fp, size_t op_start, size_t op_step, size_t meas-
ure_count) {
    fprintf(fp, "op count:\tlaunch time:\telapsed time:\n");

    size_t op_count = op_start;
    ThreadStat min_time = (ThreadStat){DBL_MAX, DBL_MAX};
    // Пока время выполнения меньше времени запуска
    while (min_time.elapsed_time <= min_time.launch_time) {
        min_time = (ThreadStat){DBL_MAX, DBL_MAX};

        // Повторяем замеры указанное число раз и находим минимальное значение
        // для каждой величины (исключаем посторонние процессы)
        for (size_t j = 0; j != measure_count; ++j) {
            ThreadStat thread_stat = threadTimeStat(threadFunc, op_count);

            if (thread_stat.launch_time < min_time.launch_time)
                min_time.launch_time = thread_stat.launch_time;

            if (thread_stat.elapsed_time < min_time.elapsed_time)
                min_time.elapsed_time = thread_stat.elapsed_time;
        }

        fprintf(fp, "%zu:\t", op_count);
        fprintf(fp, "%.14lf\t", min_time.launch_time);
        fprintf(fp, "%.14lf\n", min_time.elapsed_time);

        // Увеличиваем количество операций
        op_count += op_step;
```

```
    }
}

int main(int argc, char *argv[]) {
    if (argc < ARGS_COUNT) {
        fprintf(stderr, "Wrong number of aguments!\n");
        fprintf(stderr, "Enter: <start operation count> "
                        "<step operation count> <measure count>\n");
        exit(EXIT_FAILURE);
    }

    size_t op_start = atoi(argv[1]);
    size_t op_step = atoi(argv[2]);
    size_t measure_count = atoi(argv[3]);

    FILE* fp = fopen(RESULT_FILENAME, "w");

    printf("Program execution...\n");
    resultOutput(fp, op_start, op_step, measure_count);
    printf("Done.\n");

    fclose(fp);

    return 0;
}
```

threadtimestat.h

```
#ifndef THREADTIMESTAT_H
#define THREADTIMESTAT_H

#include <time.h>
#include <pthread.h>

#define BILLION 1.0E+9

// Функция вычета разности между временными величинами
#define clocktimeDifference(start, stop)            \
    1.0 * (stop.tv_sec - start.tv_sec) +            \
    1.0 * (stop.tv_nsec - start.tv_nsec) / BILLION

// Указатель на потоковую функцию
typedef void* (*pthread_func)(void*);

// Статистика потока
typedef struct ThreadStat {
    double launch_time;     // Время запуска
    double elapsed_time;    // Время выполнения
} ThreadStat;
```

```c
// Параметры потока
typedef struct ThreadArg {
    // Входной аргумент
    size_t op_count;        // Количество операций

    // Выходной аргумент
    double elapsed_time;    // Время выполнения
} ThreadArg;

// Функция получения времени запуска, времени выполнения потока,
// при указанном количестве операций
ThreadStat threadTimeStat(pthread_func thread_func, size_t op_count);

#endif
```

threadtimestat.c

```c
#include "threadtimestat.h"

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>

// Функция обработки ошибок
#define   throwErr(msg)   do   {   \
    fprintf(stderr,  "%s\n",  msg);  \
    exit(EXIT_FAILURE);              \
} while (0)

// Функция получения времени запуска, времени выполнения потока,
// при указанном количестве операций
ThreadStat threadTimeStat(pthread_func thread_func, size_t op_count) {
    pthread_t thread;
    ThreadArg thread_arg = (ThreadArg){op_count, 0.0};
    int err = 0;

    // Расчёт времени запуска потока
    struct timespec launch_start, launch_stop;
    clock_gettime(CLOCK_MONOTONIC, &launch_start);

    err = pthread_create(&thread, NULL, thread_func, (void*)&thread_arg);
    if (err != 0)
        throwErr("Error: thread create error!");

    clock_gettime(CLOCK_MONOTONIC, &launch_stop);

    pthread_join(thread, NULL);
    if (err != 0)
        throwErr("Error: thread join error!");
```

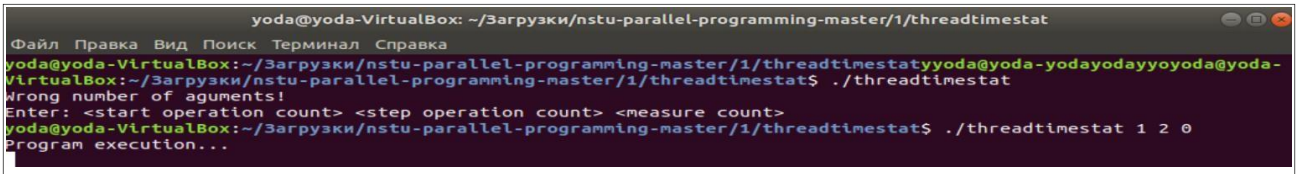
```
    return (ThreadStat){clocktimeDifference(launch_start, launch_stop),
                        thread_arg.elapsed_time};
}
```

Makefile

```
DEFINES = -D_POSIX_C_SOURCE -D_BSD_SOURCE
CFLAGS  = -std=c99 -O0 -g $(DEFINES)
LIBS    = -lpthread -lm
TARGET  = threadtimestat

all: $(TARGET) cleanTemp

$(TARGET): $(TARGET)_main.o $(TARGET).o
    gcc $(CFLAGS) -o $(TARGET) $(TARGET)_main.o $(TARGET).o $(LIBS)

$(TARGET)_main.o: $(TARGET)_main.c $(TARGET).h
    gcc $(CFLAGS) -c $(TARGET)_main.c

$(TARGET).o: $(TARGET).c $(TARGET).h
    gcc $(CFLAGS) -c $(TARGET).c

cleanTemp:
    rm -rf *.o

clean:
    rm -rf $(TARGET)
```

Результаты работы второй задачи

```
/home/NSTU/pmi-b7603/PUTTY/threadtimestat/result.tx
op count:       launch time:     elapsed time:
4:      0.00002196200000        0.00000014000000
6:      0.00001779300000        0.00000017000000
8:      0.00001906600000        0.00000016100000
10:     0.00006344100000        0.00000018000000
12:     0.00006791800000        0.00000018100000
14:     0.00007569300000        0.00000019100000
16:     0.00007057300000        0.00000027000000
18:     0.00007883000000        0.00000028000000
20:     0.00001984700000        0.00000021100000
22:     0.00007151500000        0.00000030000000
24:     0.00007832900000        0.00000025100000
26:     0.00007274800000        0.00000031000000
28:     0.00011557900000        0.00000033100000
30:     0.00009933900000        0.00000024000000
32:     0.00009932800000        0.00000033000000
34:     0.00004428300000        0.00000087200000
36:     0.00009120300000        0.00000035000000
38:     0.00008122400000        0.00000027000000
40:     0.00006785800000        0.00000043100000
42:     0.00008551300000        0.00000022000000
44:     0.00007273800000        0.00000028100000
46:     0.00007438100000        0.00000038100000
48:     0.00007925000000        0.00000030100000
50:     0.00008721500000        0.00000032100000
52:     0.00007144600000        0.00000031100000
54:     0.00006716700000        0.00000032000000
56:     0.00005711800000        0.00000033000000
58:     0.00004880200000        0.00000034000000
60:     0.00006992300000        0.00000035100000
```



Задание для самостоятельной работы

sender.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <signal.h>

#define ARGS_COUNT 2

// Шаблон команды для отправки HTTP запроса
#define COMMAND_TEMPLATE "telnet %s >/dev/null 2>&1"

// Внешняя переменная для приёма сигнала
volatile sig_atomic_t running = true;

// Функция отлова сигнала
static void signalHandler(int sig_num) {
    if (sig_num == SIGINT)
        running = false;
}

int main(int argc, char* argv[]) {
    if (argc < ARGS_COUNT) {
        fprintf(stderr, "Wrong number of arguments!\n");
        fprintf(stderr, "Enter: [host name [port]]\n");
        exit(EXIT_FAILURE);
    }

    signal(SIGINT, signalHandler);

    // Формирование команды
    size_t command_len = strlen(COMMAND_TEMPLATE) + strlen(argv[1]) + 1;
    char* command = (char*)malloc(sizeof(char*) * command_len);
    if (command == NULL) {
        fprintf(stderr, "Error: out of memmory!\n");
        exit(EXIT_FAILURE);
    }

    snprintf(command, command_len, COMMAND_TEMPLATE, argv[1]);

    printf("Sending requests to the address...\n", argv[1]);
    printf("Press \'Ctrl + C\' to exit.");
```

```
    // Исполнять без остановки системную команду, пока не будет послан сигнал SIGIN
T
    // (Нажаты клавиши Ctrl + C)
    while (running) {
        int ret = system(command);
        // Проверка перехвата сигнала системной командой
        if (WIFSIG-
NALED(ret) && (WTERMSIG(ret) == SIGINT || WTERMSIG(ret) == SIGQUIT))
            running = false;
    }

    free(command);

    return 0;
}
```

Server

server_main.c

```
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <unistd.h>

#include "server.h"

#define ARGS_COUNT 4

// Параметры ожидания для ждущего потока
#define TIME_MINUTE     60      // Минута
#define THREAD_WAIT_MIN 10      // Кол-во минут

// Препроцессор для формирования строковых констант
#define TEXT_QUOTE(...) #__VA_ARGS___

// Количество цифр десятичного числа
#define intDigitsCount(val) \
    ((val) == 0 ? 1 : (size_t)floor(log10(abs(val))) + 1)

// Шаблон для стандартной функции
static const char* RESPONSE_TEMPLATE = TEXT_QUOTE(
    HTTP/1.1 200 OK\r\n
    Content-Length: %lu\r\n\r\n
        <html>
            <head>
                <title>Request</title>
                <style>
                    h1 {
```

```
                    text-align: center;
                }
            </style>
        </head>
        <body>
            <h1>Request number %u has been processed</h1>
        </body>
    </html>\r\n
);

// Шаблон для возврата в версии PHP
static const char* RESPONSE_TEMPLATE_PHP = TEXT_QUOTE(
    HTTP/1.1 200 OK\r\n
    Content-Length: %lu\r\n\r\n
        <html>
            <head>
                <title>Request</title>
                <style>
                    h1, div {
                        text-align: center;
                    }
                </style>
            </head>
        <body>
            <h1>Request number %u has been processed</h1>
            <div>PHP version: %s</div>
        </body>
    </html>\r\n
);

// Стандартная потоковая функция
static void* threadFunc(void* arg) {
    ThreadParam* thread_param = (ThreadParam*)arg;

    size_t response_size = strlen(RESPONSE_TEMPLATE) +
                           intDigitsCount(thread_param->request_num);
    char* response = (char*)malloc(sizeof(char) * response_size);
    snprintf(response, response_size, RESPONSE_TEMPLATE, response_size,
             thread_param->request_num);

    clientWrite(thread_param->client_fd, response, response_size);
    clientClose(thread_param->client_fd);

    free(response);
    pthread_exit(NULL);
}

// Потоковая функция с вызовом интерпретатора PHP и возвратом версии PHP
static void* threadFuncPHP(void* arg) {
    ThreadParam* thread_param = (ThreadParam*)arg;
```

```
    enum {PHP_VERSION_LEN = 20};
    char php_version[PHP_VERSION_LEN];

    FILE* fp = popen("php -r \"echo phpversion();\"", "r");
    if (fscanf(fp, "%s", php_version) == 1) {
        pclose(fp);

        size_t response_size = strlen(RESPONSE_TEMPLATE_PHP) +
                               intDigitsCount(thread_param->request_num) + PHP_VER-
SION_LEN;
        char* response = (char*)malloc(sizeof(char) * response_size);
        snprintf(response, response_size, RESPONSE_TEMPLATE_PHP,
                 response_size, thread_param->request_num, php_version);

        clientWrite(thread_param->client_fd, response, response_size);
        free(response);
    }

    clientClose(thread_param->client_fd);
    pthread_exit(NULL);
}

// Потоковая функция не прекращающая работу
static void* threadFuncWait(void* arg) {
    ThreadParam* thread_param = (ThreadParam*)arg;
    clientClose(thread_param->client_fd);

    sleep(THREAD_WAIT_MIN * TIME_MINUTE);

    pthread_exit(NULL);
}

int main(int argc, char* argv[]) {
    if (argc < ARGS_COUNT) {
        fprintf(stderr, "Wrong number of arguments!\n");
        fprintf(stderr, "Enter: <func num> <stack size num> <clear pull>\n");
        fprintf(stderr, "(Func number: 0 - std, 1 - php, 2 - wait; "
                        "Stack size num: 0 - 512 KB, 1 - 1 MB, 2 - 2 MB)\n");
        exit(EXIT_FAILURE);
    }

    uint8_t func_num = atoi(argv[1]);
    uint8_t stack_size_num = atoi(argv[2]);
    size_t clear_pull = atol(argv[3]);

    pthread_func thread_func = threadFunc;
    switch (func_num) {
        case 0: thread_func = threadFunc;  break;case
        1: thread_func = threadFuncPHP; break;case 2:
        thread_func = threadFuncWait; break;
    }
```

```
    size_t stack_size = DEFAULT_STACK_SIZE;
    switch (stack_size_num) {
        case 0: stack_size = 512 * KB; break;
        case 1: stack_size = 1 * MB; break;
        case 2: stack_size = 2 * MB; break;
    }

    serverStart(thread_func, stack_size, clear_pull);

    return 0;
}
```

server.h

```
#ifndef SERVER_H
#define SERVER_H

#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>
#include <pthread.h>

#include "list.h"

#define SERVER_PORT 8080

// Единицы размеров для стека
#define KB 1024
#define MB 1024 * KB

// Размер стека потока по умолчанию
#define DEFAULT_STACK_SIZE   2 * MB
// Размер очереди подключений по умолчанию
#define DEFAULT_CLIENT_COUNT 128

// Указатель на потоковую функцию
typedef void* (*pthread_func)(void*);

// Открытые параметры потока
typedef struct ThreadParam {
    uint32_t request_num;   // Номер запроса (номер потока + 1)
    int client_fd;          // Дескриптор клиента
} ThreadParam;
// Отправить ответ клиенту
void clientWrite(int client_fd, char* response, size_t response_size);
// Закрытие соединения с клиентом
void clientClose(int client_fd);
// Запуск сервера с указанной функцией, размером стека и параметром очистки
```

```
// (Параметр очистки: 0 - не чистить, n - чистить через каждые n записей)
void serverStart(pthread_func thread_func, size_t stack_size, size_t clear_pull);

#endif
```

server.c

```
#include "server.h"

#include <stdio.h>
#include <string.h>
#include <limits.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define SOCK_ERR -1

// Функция обработки ошибок
#define   throwErr(msg)   do   {   \
    fprintf(stderr, "%s\n", msg); \
    exit(EXIT_FAILURE);           \
} while (0)

// Проверка условия очистки данных: 0 - не чистить,
// n - чистить через каждые n записей
#define clearCheck(rec_num, clear_pull) \
    (clear_pull == 0 ? 0 : rec_num % clear_pull == 0)

// Все параметры потока
typedef struct ThreadInfo {
    // Открытые параметры
    uint32_t request_num;       // Номер запроса
    int client;                 // Дескриптор клиента

    // Закрытый параметр
    pthread_t thread;           // Идентификатор потока
} ThreadInfo;

// Отправить ответ клиенту
void clientWrite(int client, char* response, size_t response_size) {
    if (write(client, response, response_size) != response_size)
        fprintf(stderr, "Client write error!\n");
}

// Закрытие соединения с клиентом
void clientClose(int client) {
    shutdown(client, SHUT_WR);
    recv(client, NULL, 1, MSG_PEEK | MSG_DONTWAIT);
```

```
    close(client);
}

// Запуск сервера с указанной функцией, размером стека и параметром очистки
// (Параметр очистки: 0 - не чистить, n - чистить через каждые n записей)
void serverStart(pthread_func thread_func, size_t stack_size, size_t clear_pull) {
    struct sockaddr_in serv_addr;
    int err = 0;

    int serv_sock = socket(AF_INET, SOCK_STREAM, 0);
    if (serv_sock == SOCK_ERR)
        throwErr("Socket open error!");

    char opt = 1;
    setsockopt(serv_sock, SOL_SOCKET, SO_REUSEADDR | SO_REUSE-
PORT, &opt, sizeof(opt));

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(SERVER_PORT);

    err = bind(serv_sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
    if (err == SOCK_ERR)
        throwErr("Bind error!");

    err = listen(serv_sock, DEFAULT_CLIENT_COUNT);
    if (err == SOCK_ERR)
        throwErr("Listen error!");

    struct sockaddr_in client_addr;
    socklen_t sock_len = sizeof(client_addr);

    pthread_attr_t thread_attr;
    err = pthread_attr_init(&thread_attr);
    if (err != 0)
        throwErr("Error: cannot create thread attribute!");

    err = pthread_attr_setstacksize(&thread_attr, stack_size);
    if (err != 0)
        throwErr("Error: setting thread stack size failed!");

    // Инициализация структуры для хранения данных о поступающих соединениях
    List* threads_list = NULL;
    listInit(&threads_list, sizeof(ThreadInfo), NULL);

    pid_t serv_pid = getpid();

    // Приём входящих соединений в отдельных потоках
    uint32_t connection_count = 0;
    while (connection_count != UINT_MAX) {
        int client = accept(serv_sock, (struct sockaddr*)&client_addr, &sock_len);
```

```
        if (SOCK_ERR < client) {
            printf("Server [%d]: got connection - %u\n", serv_pid, ++connec-
tion_count);

            ThreadInfo thread_param = (ThreadInfo){connection_count, client};
            listPushBack(threads_list, (void*)&thread_param);

            err = pthread_create(&((ThreadInfo*)listBack(threads_list))->thread,
                                 &thread_attr, thread_func, list-
Back(threads_list));
            if (err != 0)
                throwErr("Error: cannot create a thread");
        }

        // Очистка данных завершённых потоков, в зависимости от условия
        // Проходим по всем потоками, если существуют, и посылаем сигнал
        if (clearCheck(connection_count, clear_pull) && !listIs-
Empty(threads_list)) {
            ListNode* threads_list_iter = threads_list->head;
            do {
                ListNode* check_item = threads_list_iter;
                threads_list_iter = threads_list_iter->next;

                // Проверяем по сигналу, завершил ли поток работу
                // Если завершил -- очищаем все данные связанные с потоком
                if (pthread_kill(((ThreadInfo*)check_item->data)->thread, 0)) {
                    pthread_join(((ThreadInfo*)check_item->data)->thread, NULL);
                    listDeleteNode(threads_list, check_item);
                }
            } while (threads_list_iter);
        }
    }

    pthread_attr_destroy(&thread_attr);

    listFree(&threads_list);
    close(serv_sock);
}
```

list.h

```
#ifndef LIST_H
#define LIST_H

#include <stddef.h>
#include <stdbool.h>

typedef void (*list_func_ptr)(void*);

typedef struct ListNode {
```

```
    void* data;

    struct ListNode* next;
    struct ListNode* prev;
} ListNode, Node;

typedef struct List {
    struct ListNode* head;
    struct ListNode* tail;

    size_t data_size;

    list_func_ptr free_func;
} List;

static inline bool listIsEmpty(const List* L) {
    return L->head == NULL || L->tail == NULL;
}

void listInit(List** L, size_t data_size, list_func_ptr free_func);

void listPushBack(List* L, void* data);
void listPushFront(List* L, void* data);

void listPopBack(List* L);
void listPopFront(List* L);

void* listBack(const List* L);
void* listFront(const List* L);

void listForEach(List* L, list_func_ptr proces_func);

void listDeleteNode(List* L, ListNode* node);

void listClear(List* L);
void listFree(List** L);

#endif
```

list.c

```
#include "list.h"

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define throwErr(msg) do { \
    fprintf(stderr, "%s\n", msg); \
```

```
    exit(EXIT_FAILURE); \
} while (0)

void listInit(List** L, size_t data_size, list_func_ptr free_func) {
    (*L) = (List*)malloc(sizeof(List));
    if ((*L) == NULL)
        throwErr("Error: out of memory to initialize list!");

    (*L)->head = (*L)->tail = NULL;

    (*L)->data_size = data_size;
    (*L)->free_func = free_func;
}

void listPushBack(List* L, void* data) {
    ListNode* new_node = (ListNode*)malloc(sizeof(ListNode));
    if (new_node == NULL)
        throwErr("Error: out of memory to initialize node!");

    new_node->data = malloc(L->data_size);
    if (new_node->data == NULL)
        throwErr("Error: out of memory to initialize node data!");

    memcpy(new_node->data, data, L->data_size);
    new_node->next = new_node->prev = NULL;

    if (listIsEmpty(L))
        L->head = L->tail = new_node;
    else {
        new_node->prev = L->tail;
        L->tail->next = new_node;
        L->tail = new_node;
    }
}

void listPushFront(List* L, void* data) {
    ListNode* new_node = (ListNode*)malloc(sizeof(ListNode));
    if (new_node == NULL)
        throwErr("Error: out of memory to initialize node!");

    new_node->data = malloc(L->data_size);
    if (new_node->data == NULL)
        throwErr("Error: out of memory to initialize node data!");

    memcpy(new_node->data, data, L->data_size);
    new_node->next = new_node->prev = NULL;


    if (listIsEmpty(L))
        L->head = L->tail = new_node;
    else {
```

```
        L->head->prev = new_node;
        new_node->next = L->head;
        L->head = new_node;
    }
}

void listPopBack(List* L) {
    if (!listIsEmpty(L)) {
        if (L->free_func)
            L->free_func(L->tail->data);

        ListNode* new_tail = L->tail->prev;

        free(L->tail);

        L->tail = new_tail;
    }
}

void listPopFront(List* L) {
    if (!listIsEmpty(L)) {
        if (L->free_func)
            L->free_func(L->head->data);

        ListNode* new_head = L->head->next;

        free(L->head);

        L->head = new_head;
    }
}

void* listBack(const List* L) {
    return listIsEmpty(L) ? NULL : L->tail->data;
}

void* listFront(const List* L) {
    return listIsEmpty(L) ? NULL : L->head->data;
}

void listForEach(List* L, list_func_ptr proces_func) {
    ListNode* iter = L->head;

    do {
        proces_func(iter->data);

        iter = iter->next;
    } while (iter);
}

void listDeleteNode(List* L, ListNode* node) {
```

```
    if (!listIsEmpty(L)) {
        if (node != L->head)
            node->prev->next = node->next;
        else
            L->head = L->tail = NULL;

        if (L->free_func)
            L->free_func(node->data);

        free(node);
    }
}

void listClear(List* L) {
    if (!listIsEmpty(L)) {
        ListNode* iter = L->head;

        do {
            ListNode* delete_node = iter;

            iter = iter->next;

            if (L->free_func)
                L->free_func(delete_node->data);

            free(delete_node);
        } while (iter);

        L->head = L->tail = NULL;
    }
}

void listFree(List** L) {
    listClear((*L));
    free((*L));

    (*L) = NULL;
}
```

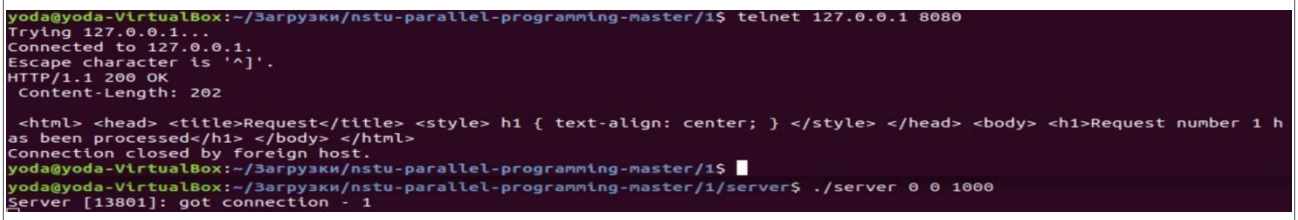
Makefile

```
DEFINES = -D_POSIX_C_SOURCE -D_BSD_SOURCE
CFLAGS  = -std=c99 -O2 -g $(DEFINES)
LIBS    = -lpthread -lm
TARGET  = server

all: $(TARGET) cleanTemp

list.o: list.c list.h
    gcc $(CFLAGS) -c list.c
```

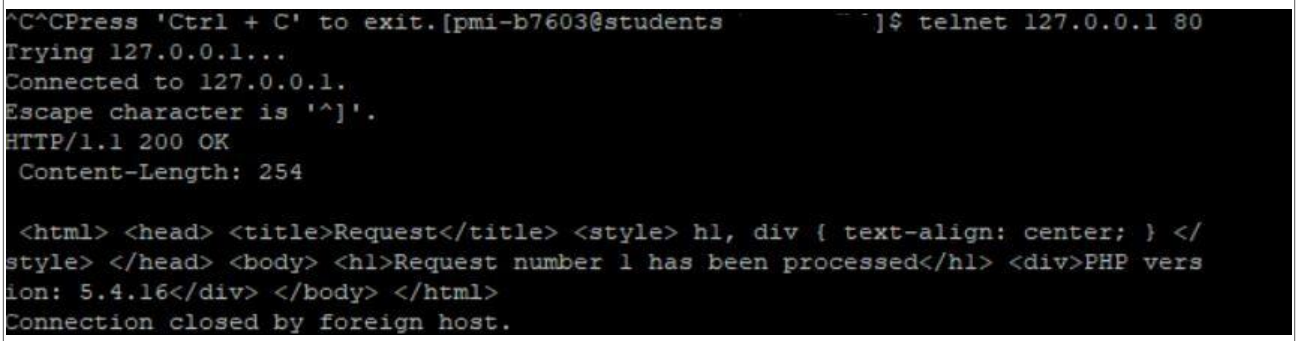
```
$(TARGET): $(TARGET)_main.o $(TARGET).o list.o
    gcc $(CFLAGS) -o $(TARGET) $(TARGET)_main.o $(TARGET).o list.o $(LIBS)

$(TARGET)_main.o: $(TARGET)_main.c $(TARGET).h
    gcc $(CFLAGS) -c $(TARGET)_main.c

$(TARGET).o: $(TARGET).c $(TARGET).h list.h
    gcc $(CFLAGS) -c $(TARGET).c

cleanTemp:
    rm -rf *.o

clean:
    rm -rf $(TARGET)
```

Результат работы задачи

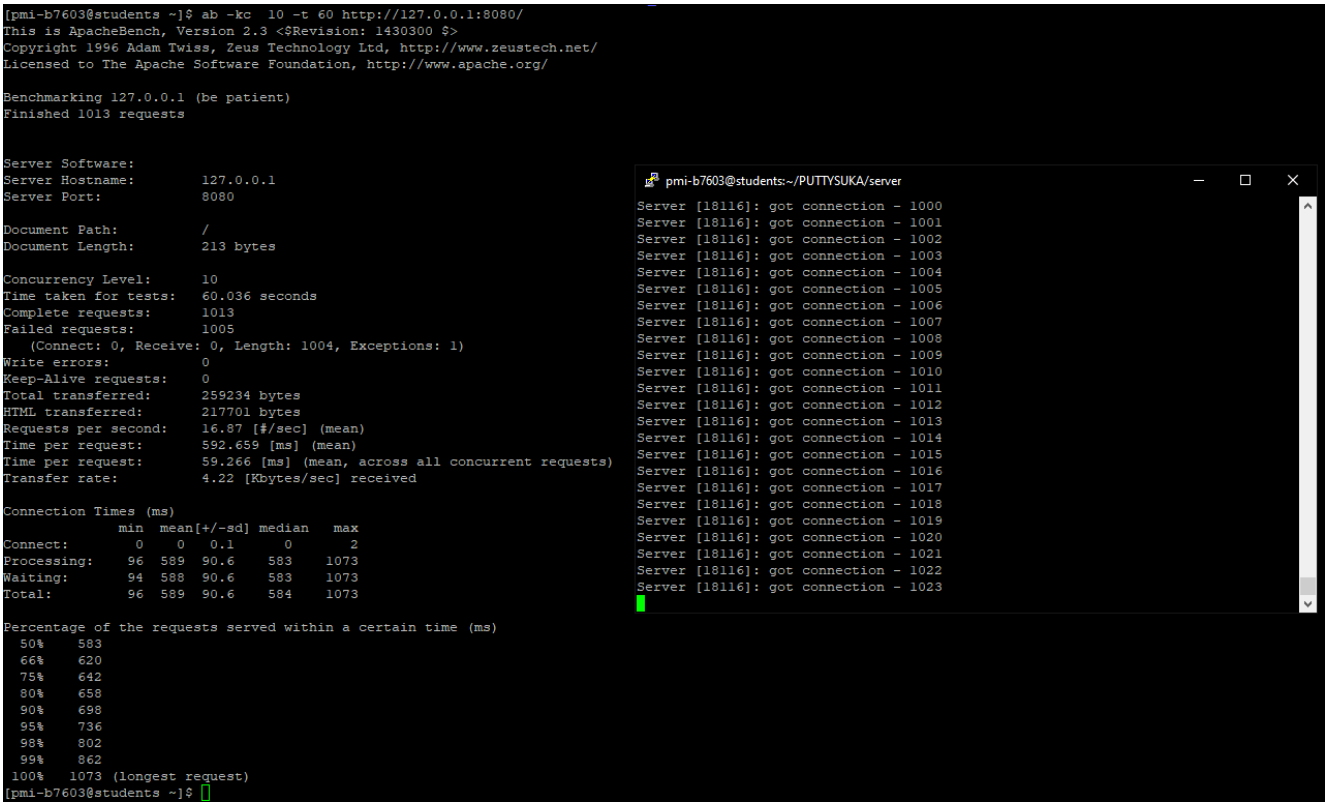
std



Php



Оценка при помощи Apache



Вывод

Мы познакомились с базовыми возможностями многопоточного программирования, научились работать с потоками, не имеющими информационных зависимостей.