

|  |  |  |  |
| --- | --- | --- | --- |
| Федеральное государственное бюджетное  образовательное учреждение высшего образования  «НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ» | | | |
|  | | | |
|  | | | |
|  | | | |
|  | | | |
|  | | | |
|  | | | |
|  | | | |
|  | | | |
| 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. ГГОРОДНИЧЕВ 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.

1. Добавить в программу возможность запуска потоков с разными атрибутами (см. пример 2).
2. Добавить в программу возможность передавать в поток сразу несколько параметров (см. пример 3).
3. Добавить в функцию потока возможность вывода информации о всех параметрах потока, с которыми он был создан.
4. Разработать программу, которая обеспечивает параллельное

применение заданной функции к каждому элементу массива. Размер

массива, применяемая функция и количество потоков задаются динамически.

1. 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

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

#include <pthread.h>

**Arrayprocessing**

*arrayprocessing.c*

// 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 | | | | |
| 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 | | | | |

#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;

// 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

#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

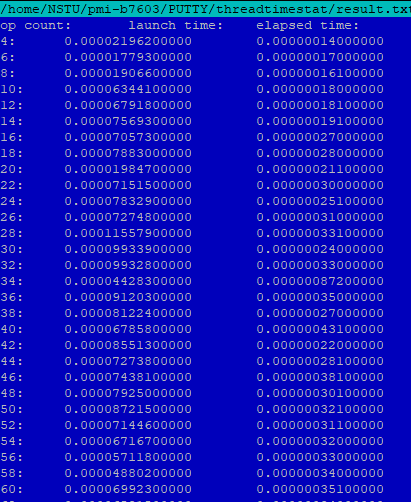
cleanTemp:

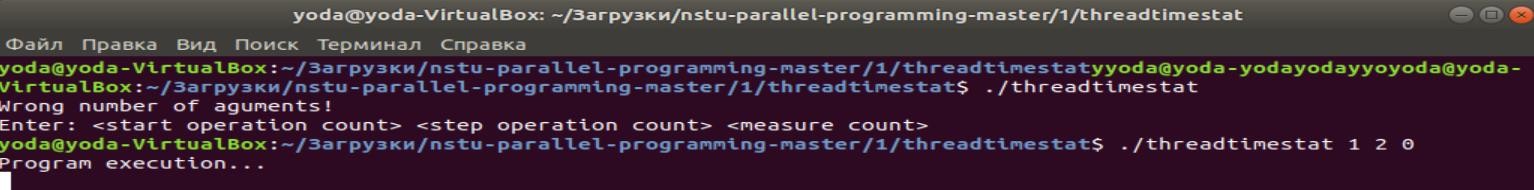
rm -rf \*.o

clean:

rm -rf $(TARGET)

Результаты работы второй задачи





# Задание для самостоятельной работы

## sender.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;

}

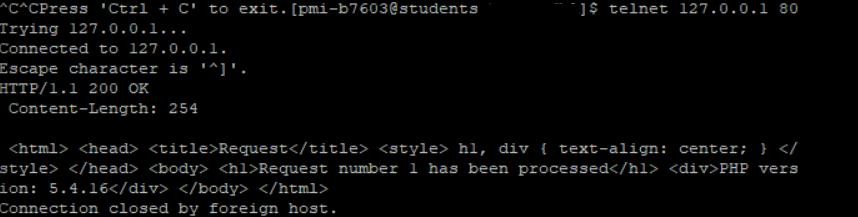
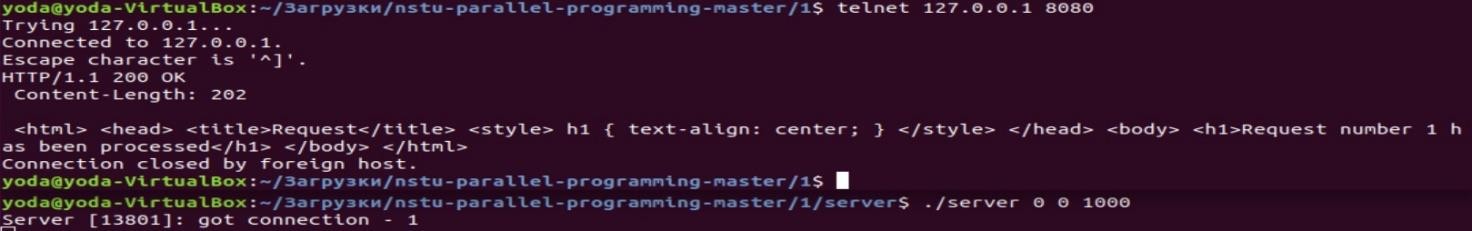
## Makefile

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:

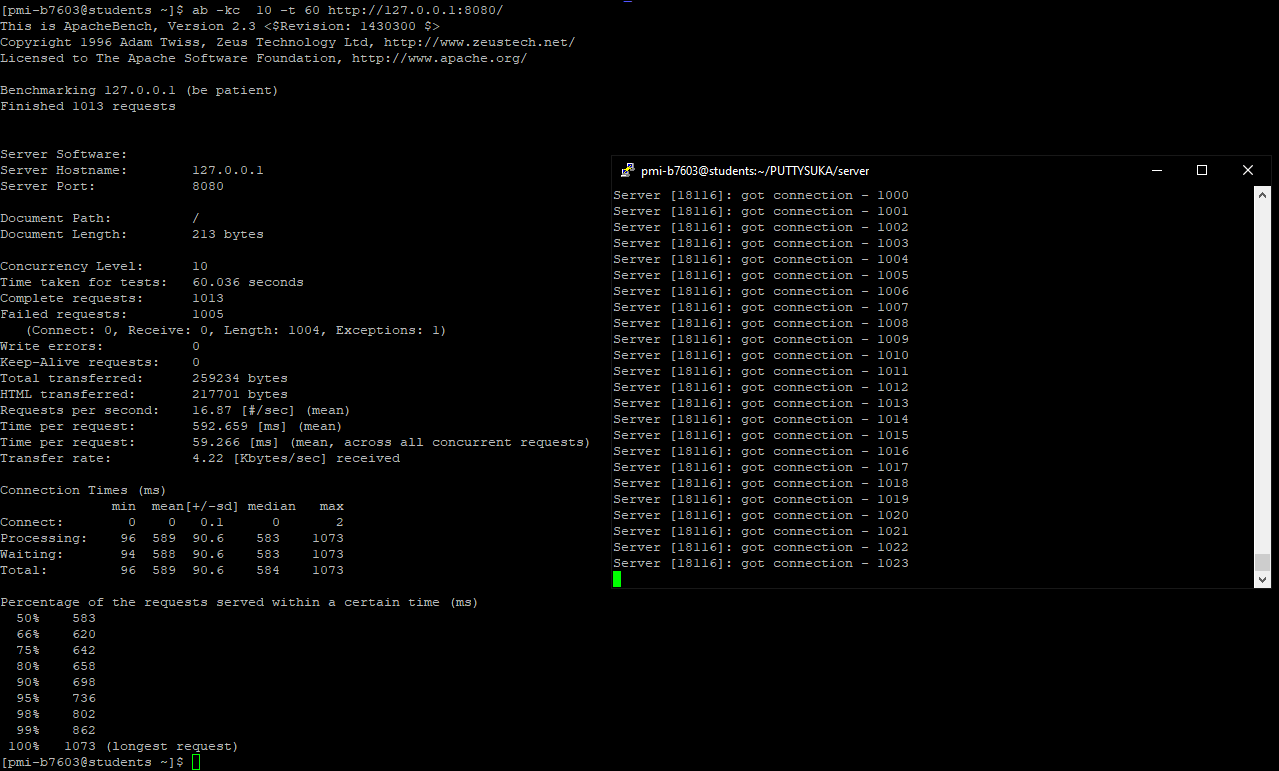
rm -rf $(TARGET)

# 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.