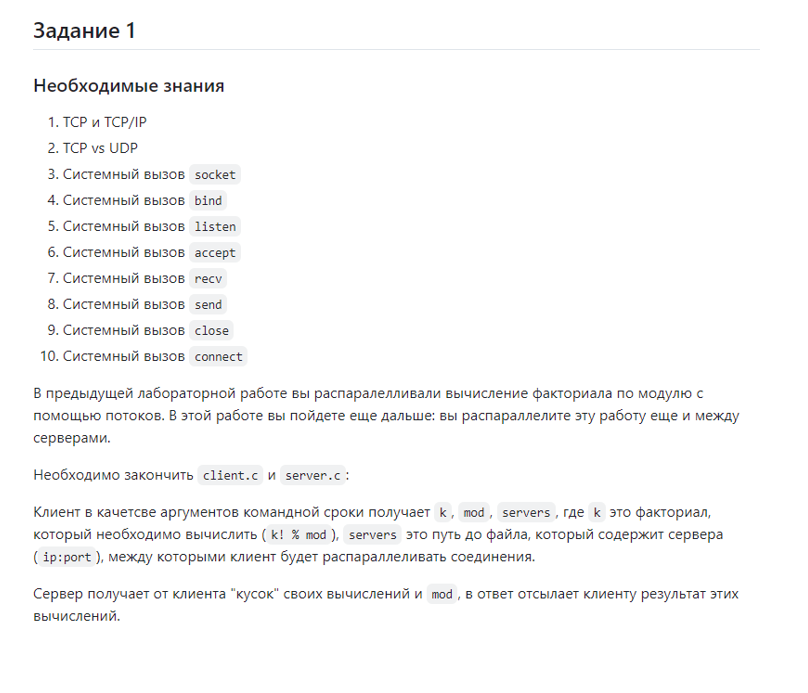
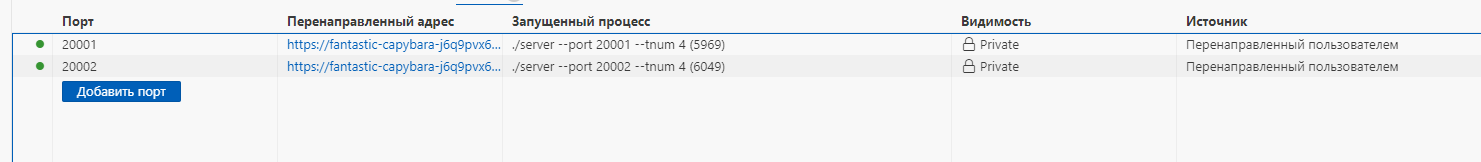
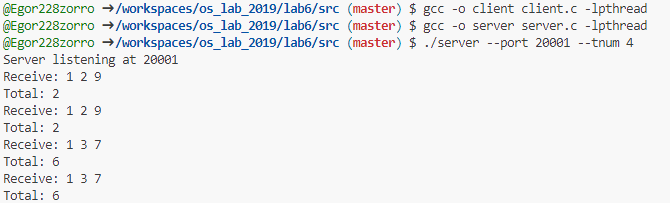
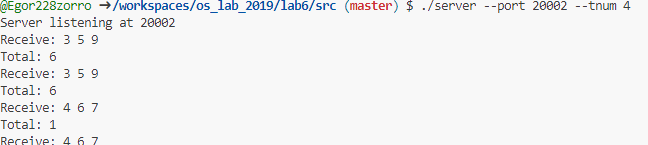
**Лабораторная работа 6**

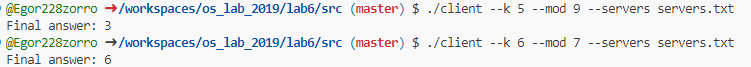
**Задание 1**



****







**client.c**

**#include <limits.h>**

**#include <stdbool.h>**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <string.h>**

**#include <unistd.h>**

**#include <getopt.h>**

**#include <netinet/in.h>**

**#include <pthread.h>**

**#include <sys/socket.h>**

**#include <arpa/inet.h>**

**struct Server {**

**char ip[64];**

**int port;**

**};**

**uint64\_t MultModulo(uint64\_t a, uint64\_t b, uint64\_t mod) {**

**uint64\_t result = 0;**

**a = a % mod;**

**while (b > 0) {**

**if (b % 2 == 1)**

**result = (result + a) % mod;**

**a = (a \* 2) % mod;**

**b /= 2;**

**}**

**return result % mod;**

**}**

**int read\_servers(const char \*path, struct Server \*\*servers) {**

**FILE \*file = fopen(path, "r");**

**if (file == NULL) {**

**fprintf(stderr, "Error: Cannot open file %s\n", path);**

**return -1;**

**}**

**size\_t count = 0;**

**size\_t capacity = 4;**

**\*servers = malloc(capacity \* sizeof(struct Server));**

**while (fscanf(file, "%63s %d", (\*servers)[count].ip, &(\*servers)[count].port) == 2) {**

**count++;**

**if (count >= capacity) {**

**capacity \*= 2;**

**\*servers = realloc(\*servers, capacity \* sizeof(struct Server));**

**}**

**}**

**fclose(file);**

**return count;**

**}**

**int send\_task(const struct Server \*server, uint64\_t begin, uint64\_t end, uint64\_t mod, uint64\_t \*result) {**

**int sock = socket(AF\_INET, SOCK\_STREAM, 0);**

**if (sock < 0) {**

**fprintf(stderr, "Error: Cannot create socket\n");**

**return -1;**

**}**

**struct sockaddr\_in server\_addr;**

**server\_addr.sin\_family = AF\_INET;**

**server\_addr.sin\_port = htons(server->port);**

**inet\_pton(AF\_INET, server->ip, &server\_addr.sin\_addr);**

**if (connect(sock, (struct sockaddr \*)&server\_addr, sizeof(server\_addr)) < 0) {**

**fprintf(stderr, "Error: Cannot connect to server %s:%d\n", server->ip, server->port);**

**close(sock);**

**return -1;**

**}**

**uint64\_t task[3] = {begin, end, mod};**

**if (send(sock, task, sizeof(task), 0) < 0) {**

**fprintf(stderr, "Error: Cannot send data to server\n");**

**close(sock);**

**return -1;**

**}**

**if (recv(sock, result, sizeof(\*result), 0) < 0) {**

**fprintf(stderr, "Error: Cannot receive data from server\n");**

**close(sock);**

**return -1;**

**}**

**close(sock);**

**return 0;**

**}**

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

**uint64\_t k = 0;**

**uint64\_t mod = 0;**

**char \*servers\_path = NULL;**

**while (true) {**

**int current\_optind = optind ? optind : 1;**

**static struct option options[] = {**

**{"k", required\_argument, 0, 'k'},**

**{"mod", required\_argument, 0, 'm'},**

**{"servers", required\_argument, 0, 's'},**

**{0, 0, 0, 0}};**

**int option\_index = 0;**

**int c = getopt\_long(argc, argv, "", options, &option\_index);**

**if (c == -1)**

**break;**

**switch (c) {**

**case 'k':**

**k = strtoull(optarg, NULL, 10);**

**break;**

**case 'm':**

**mod = strtoull(optarg, NULL, 10);**

**break;**

**case 's':**

**servers\_path = optarg;**

**break;**

**default:**

**fprintf(stderr, "Usage: %s --k <value> --mod <value> --servers <path>\n", argv[0]);**

**return 1;**

**}**

**}**

**if (k == 0 || mod == 0 || servers\_path == NULL) {**

**fprintf(stderr, "Usage: %s --k <value> --mod <value> --servers <path>\n", argv[0]);**

**return 1;**

**}**

**struct Server \*servers = NULL;**

**int servers\_count = read\_servers(servers\_path, &servers);**

**if (servers\_count <= 0) {**

**fprintf(stderr, "Error: No servers available\n");**

**return 1;**

**}**

**uint64\_t chunk\_size = k / servers\_count;**

**uint64\_t remaining = k % servers\_count;**

**uint64\_t total = 1;**

**for (int i = 0; i < servers\_count; i++) {**

**uint64\_t begin = i \* chunk\_size + 1;**

**uint64\_t end = (i + 1) \* chunk\_size;**

**if (i == servers\_count - 1) // Add remaining part to the last chunk**

**end += remaining;**

**uint64\_t result = 0;**

**if (send\_task(&servers[i], begin, end, mod, &result) < 0) {**

**fprintf(stderr, "Error: Task failed on server %s:%d\n", servers[i].ip, servers[i].port);**

**free(servers);**

**return 1;**

**}**

**total = MultModulo(total, result, mod);**

**}**

**printf("Final answer: %lu\n", total);**

**free(servers);**

**return 0;**

**}**

**server.c**

**#include <limits.h>**

**#include <stdbool.h>**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <string.h>**

**#include <unistd.h>**

**#include <getopt.h>**

**#include <netinet/in.h>**

**#include <pthread.h>**

**#include <sys/socket.h>**

**#include "multmodulo.h"**

**struct FactorialArgs {**

**uint64\_t begin;**

**uint64\_t end;**

**uint64\_t mod;**

**};**

**uint64\_t MultModulo(uint64\_t a, uint64\_t b, uint64\_t mod) {**

**uint64\_t result = 0;**

**a = a % mod;**

**while (b > 0) {**

**if (b % 2 == 1)**

**result = (result + a) % mod;**

**a = (a \* 2) % mod;**

**b /= 2;**

**}**

**return result % mod;**

**}**

**uint64\_t Factorial(const struct FactorialArgs \*args) {**

**uint64\_t ans = 1;**

**for (uint64\_t i = args->begin; i <= args->end; i++) {**

**ans = MultModulo(ans, i, args->mod);**

**}**

**return ans;**

**}**

**void \*ThreadFactorial(void \*args) {**

**struct FactorialArgs \*fargs = (struct FactorialArgs \*)args;**

**uint64\_t \*result = malloc(sizeof(uint64\_t));**

**\*result = Factorial(fargs);**

**return result;**

**}**

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

**int tnum = -1;**

**int port = -1;**

**while (true) {**

**static struct option options[] = {**

**{"port", required\_argument, 0, 0},**

**{"tnum", required\_argument, 0, 0},**

**{0, 0, 0, 0}};**

**int option\_index = 0;**

**int c = getopt\_long(argc, argv, "", options, &option\_index);**

**if (c == -1)**

**break;**

**switch (c) {**

**case 0:**

**if (strcmp(options[option\_index].name, "port") == 0) {**

**port = atoi(optarg);**

**} else if (strcmp(options[option\_index].name, "tnum") == 0) {**

**tnum = atoi(optarg);**

**}**

**break;**

**default:**

**fprintf(stderr, "Unknown argument\n");**

**return 1;**

**}**

**}**

**if (port == -1 || tnum == -1) {**

**fprintf(stderr, "Usage: %s --port <port> --tnum <threads>\n", argv[0]);**

**return 1;**

**}**

**int server\_fd = socket(AF\_INET, SOCK\_STREAM, 0);**

**if (server\_fd < 0) {**

**perror("socket");**

**return 1;**

**}**

**struct sockaddr\_in server;**

**server.sin\_family = AF\_INET;**

**server.sin\_port = htons(port);**

**server.sin\_addr.s\_addr = htonl(INADDR\_ANY);**

**if (bind(server\_fd, (struct sockaddr \*)&server, sizeof(server)) < 0) {**

**perror("bind");**

**return 1;**

**}**

**if (listen(server\_fd, 128) < 0) {**

**perror("listen");**

**return 1;**

**}**

**printf("Server listening at %d\n", port);**

**while (true) {**

**struct sockaddr\_in client;**

**socklen\_t client\_len = sizeof(client);**

**int client\_fd = accept(server\_fd, (struct sockaddr \*)&client, &client\_len);**

**if (client\_fd < 0) {**

**perror("accept");**

**continue;**

**}**

**uint64\_t args[3];**

**if (recv(client\_fd, args, sizeof(args), 0) <= 0) {**

**perror("recv");**

**close(client\_fd);**

**continue;**

**}**

**uint64\_t begin = args[0];**

**uint64\_t end = args[1];**

**uint64\_t mod = args[2];**

**printf("Receive: %lu %lu %lu\n", begin, end, mod);**

**pthread\_t threads[tnum];**

**struct FactorialArgs fargs[tnum];**

**uint64\_t chunk\_size = (end - begin + 1) / tnum;**

**uint64\_t total = 1;**

**for (int i = 0; i < tnum; i++) {**

**fargs[i].begin = begin + i \* chunk\_size;**

**fargs[i].end = (i == tnum - 1) ? end : (begin + (i + 1) \* chunk\_size - 1);**

**fargs[i].mod = mod;**

**if (pthread\_create(&threads[i], NULL, ThreadFactorial, &fargs[i]) != 0) {**

**perror("pthread\_create");**

**close(client\_fd);**

**return 1;**

**}**

**}**

**for (int i = 0; i < tnum; i++) {**

**uint64\_t \*result;**

**pthread\_join(threads[i], (void \*\*)&result);**

**total = MultModulo(total, \*result, mod);**

**free(result);**

**}**

**printf("Total: %lu\n", total);**

**if (send(client\_fd, &total, sizeof(total), 0) <= 0) {**

**perror("send");**

**}**

**close(client\_fd);**

**}**

**return 0;**

**}**

**servers.txt**

127.0.0.1 20001

127.0.0.1 20002

**Командная строка:**

./server --port 20001 --tnum 4

./server --port 20002 --tnum 4

./client --k 5 --mod 9 --servers servers.txt

**1.TCP и TCP/IP**

**TCP (Transmission Control Protocol)**

* TCP — это протокол транспортного уровня модели TCP/IP.
* Предоставляет надежную передачу данных: обеспечивает гарантированную доставку, контроль ошибок и порядок передачи.
* Примеры использования: HTTP, FTP, SMTP.

**TCP/IP**

* Стек протоколов, на основе которого работает интернет.
* Состоит из нескольких уровней:
  + **Прикладной уровень:** HTTP, FTP, DNS.
  + **Транспортный уровень:** TCP, UDP.
  + **Сетевой уровень:** IP (IPv4/IPv6).
  + **Канальный уровень:** Ethernet, Wi-Fi.

| **Характеристика** | **TCP** | **UDP** |
| --- | --- | --- |

|  |  |  |
| --- | --- | --- |
| **Тип соединения** | С установленным соединением | Без установления соединения |

|  |  |  |
| --- | --- | --- |
| **Надежность** | Гарантирует доставку | Не гарантирует доставку |

|  |  |  |
| --- | --- | --- |
| **Контроль порядка** | Сохраняет порядок данных | Порядок данных не гарантирован |

|  |  |  |
| --- | --- | --- |
| **Применение** | HTTP, FTP, почта | VoIP, видео, DNS-запросы |

|  |  |  |
| --- | --- | --- |
| **Скорость** | Медленнее из-за проверки ошибок | Быстрее, но менее надежно |

**2.вызов socket**

Создает файловый дескриптор для работы с сетевым соединением. Этот дескриптор будет представлять сокет, через который происходит обмен данными.

**Прототип:**

**int socket(int domain, int type, int protocol);**

**Параметры:**

**domain — указывает семейство протоколов:**

**AF\_INET — для IPv4.**

**AF\_INET6 — для IPv6.**

**AF\_UNIX — для локального взаимодействия в системе.**

**type — тип сокета:**

**SOCK\_STREAM — потоковый (TCP).**

**SOCK\_DGRAM — дейтаграммный (UDP).**

**protocol — конкретный протокол (обычно 0, если используется протокол по умолчанию).**

**3.вызов bind**

**Привязывает сокет к локальному адресу (IP + порт), чтобы сервер мог принимать соединения или получать данные.**

**Прототип:**

**int bind(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);**

**Параметры:**

**sockfd — дескриптор сокета.**

**addr — указатель на структуру с информацией об адресе (struct sockaddr).**

**addrlen — размер структуры адреса.**

**4.Вызов listen**

**Переводит сокет в режим ожидания входящих соединений (только для TCP).**

**Прототип:**

**int listen(int sockfd, int backlog);**

**Параметры:**

**sockfd — дескриптор сокета.**

**backlog — максимальное число ожидающих соединений в очереди.**

**5. Вызов accept**

**Принимает входящее соединение от клиента. Возвращает новый дескриптор сокета для общения с этим клиентом. (Нужен для :Создания нового сокета для общения с конкретным клиентом)**

**Прототип:**

**int accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen);**

**Параметры:**

**sockfd — дескриптор сокета, который слушает соединения.**

**addr — структура для хранения адреса клиента.**

**addrlen — указатель на размер структуры адреса.**

**6. Вызов recv**

**Получает данные из сокета.**

**Прототип:**

**ssize\_t recv(int sockfd, void \*buf, size\_t len, int flags);**

**Параметры:**

**sockfd — дескриптор сокета.**

**buf — буфер для записи данных.**

**len — размер буфера.**

**flags — дополнительные опции (обычно 0).**

**7. Вызов send Отправляет данные через сокет.( Позволяет передавать данные между клиентом и сервером.)**

**Прототип:**

**ssize\_t send(int sockfd, const void \*buf, size\_t len, int flags);**

**Параметры:**

**sockfd — дескриптор сокета.**

**buf — буфер с данными для отправки.**

**len — количество отправляемых байт.**

**flags — дополнительные опции (обычно 0).**

**8. Вызов close**

**Закрывает сокет, освобождая ресурсы.(** (**Освобождает дескриптор сокета и закрывает соединение.)**

**Прототип:**

**int close(int sockfd);**

**9.Вызов connect**

**Устанавливает соединение с удаленным сервером (только для TCP).** **Клиенту необходимо вызвать connect, чтобы установить соединение с сервером.**

**Прототип:**

**int connect(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);**

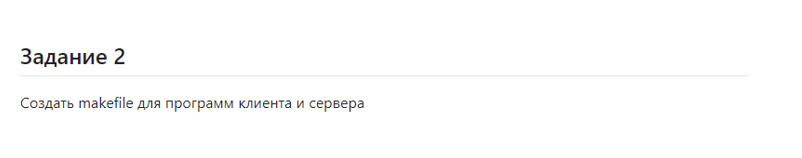
**Параметры:**

**sockfd — дескриптор сокета.**

**addr — структура с адресом сервера.**

**addrlen — размер структуры адреса.**

**Задание 2**

  
**CC = gcc**

**CFLAGS = -I. -lpthread**

**all: client server**

**client: client.o**

**$(CC) -o client client.o $(CFLAGS)**

**server: server.o**

**$(CC) -o server server.o $(CFLAGS)**

**client.o: client.c**

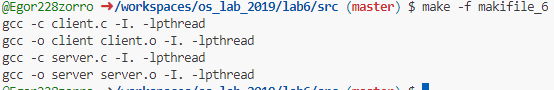
**$(CC) -c client.c $(CFLAGS)**

**server.o: server.c**

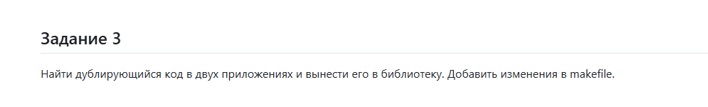
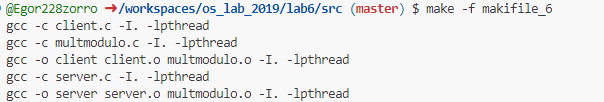
**$(CC) -c server.c $(CFLAGS)**

**clean:**

**rm -f \*.o client server**



**Задание 3**

**multmodulo.h:**

**#ifndef MULTMODULO\_H**

**#define MULTMODULO\_H**

**#include <stdint.h>**

**uint64\_t MultModulo(uint64\_t a, uint64\_t b, uint64\_t mod);**

**#endif // MULTMODULO\_H**

**multmodulo.c:**

**#include "multmodulo.h"**

**uint64\_t MultModulo(uint64\_t a, uint64\_t b, uint64\_t mod) {**

**uint64\_t result = 0;**

**a = a % mod;**

**while (b > 0) {**

**if (b % 2 == 1)**

**result = (result + a) % mod;**

**a = (a \* 2) % mod;**

**b /= 2;**

**}**

**return result % mod;**

**}**

**makefile\_6:**

**CC = gcc**

**CFLAGS = -I. -lpthread**

**all: client server**

**client: client.o multmodulo.o**

**$(CC) -o client client.o multmodulo.o $(CFLAGS)**

**server: server.o multmodulo.o**

**$(CC) -o server server.o multmodulo.o $(CFLAGS)**

**client.o: client.c multmodulo.h**

**$(CC) -c client.c $(CFLAGS)**

**server.o: server.c multmodulo.h**

**$(CC) -c server.c $(CFLAGS)**

**multmodulo.o: multmodulo.c multmodulo.h**

**$(CC) -c multmodulo.c $(CFLAGS)**

**clean:**

**rm -f \*.o client server**

**ссылка на githab: https://github.com/Egor228zorro/os\_lab\_2019/tree/master/lab6/src**