Лабораторная работа 6

Задание 1

Задание 1

Необходимые знания

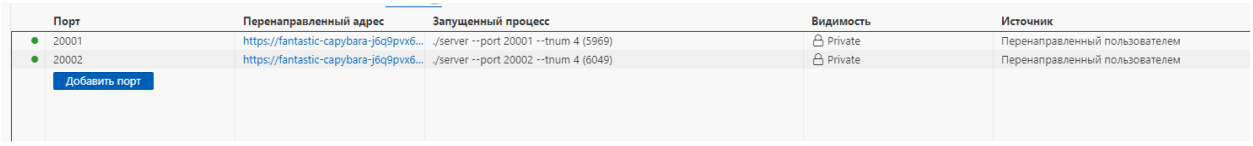
1. TCP и TCP/IP
2. TCP vs UDP
3. Системный вызов `socket`
4. Системный вызов `bind`
5. Системный вызов `listen`
6. Системный вызов `accept`
7. Системный вызов `recv`
8. Системный вызов `send`
9. Системный вызов `close`
10. Системный вызов `connect`

В предыдущей лабораторной работе вы распаралелливали вычисление факториала по модулю с помощью потоков. В этой работе вы пойдете еще дальше: вы распараллелите эту работу еще и между серверами.

Необходимо закончить `client.c` и `server.c`:

Клиент в качетсве аргументов командной сроки получает `k`, `mod`, `servers`, где `k` это факториал, который необходимо вычислить (`k! % mod`), `servers` это путь до файла, который содержит сервера (`ip:port`), между которыми клиент будет распараллеливать соединения.

Сервер получает от клиента "кусок" своих вычислений и `mod`, в ответ отсылает клиенту результат этих вычислений.




```
@Egor228zorro ➜/workspaces/os_lab_2019/lab6/src (master) $ gcc -o client client.c -lpthread
@Egor228zorro ➜/workspaces/os_lab_2019/lab6/src (master) $ gcc -o server server.c -lpthread
@Egor228zorro ➜/workspaces/os_lab_2019/lab6/src (master) $ ./server --port 20001 --tnum 4
Server listening at 20001
Receive: 1 2 9
Total: 2
Receive: 1 2 9
Total: 2
Receive: 1 3 7
Total: 6
Receive: 1 3 7
Total: 6
```

```
@Egor228zorro ➜/workspaces/os_lab_2019/lab6/src (master) $ ./server --port 20002 --tnum 4
Server listening at 20002
Receive: 3 5 9
Total: 6
Receive: 3 5 9
Total: 6
Receive: 4 6 7
Total: 1
Receive: 4 6 7
```

```
@Egor228zorro ➜/workspaces/os_lab_2019/lab6/src (master) $ ./client --k 5 --mod 9 --servers servers.txt
Final answer: 3
@Egor228zorro ➜/workspaces/os_lab_2019/lab6/src (master) $ ./client --k 6 --mod 7 --servers servers.txt
Final answer: 6
```

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

Задание 2

Создать makefile для программ клиента и сервера

```
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
```

```
@Egor228zorro ➜/workspaces/os_lab_2019/lab6/src (master) $ make -f makifile_6
gcc -c client.c -I. -lpthread
gcc -o client client.o -I. -lpthread
gcc -c server.c -I. -lpthread
gcc -o server server.o -I. -lpthread
```

Задание 3

Задание 3

Найти дублирующийся код в двух приложениях и вынести его в библиотеку. Добавить изменения в makefile.

```
@Egor228zorro ➜/workspaces/os_lab_2019/lab6/src (master) $ make -f makifile_6
gcc -c client.c -I. -lpthread
gcc -c multmodulo.c -I. -lpthread
gcc -o client client.o multmodulo.o -I. -lpthread
gcc -c server.c -I. -lpthread
gcc -o server server.o multmodulo.o -I. -lpthread
```

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