



Міністерство освіти і науки України

Національний технічний університет України

“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Лабораторна робота №3

Мережеве програмування в середовищі Unix

Тема: Багатопроцесний ітеративний TCP клієнт-сервер

Виконав

студент групи ІП-11:

Панченко С. В.

Перевірив:

Сімоненко А.В.

ЗМІСТ

1 Мета лабораторної роботи.....	6
2 Завдання.....	7
3 Виконання.....	9
3.1 Ітеративний сервер.....	9
3.2 Паралельний сервер.....	9
3.3 Сервер з пулом процесів.....	10
ДОДАТОК А ТЕКСТИ ПРОГРАМНОГО КОДУ.....	11

1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ

Реалізувати простий мережевий сервіс передачі файлів за допомогою протоколу TCP.

2 ЗАВДАННЯ

Розробити клієнт та сервер, які виконують наступне:

1. Клієнт підтримує такі аргументи командного рядка: адреса сервера, порт сервера, ім'я файлу, максимальний розмір файлу.
2. Сервер підтримує такі аргументи командного рядка: адреса сервера, порт сервера, шляхове ім'я директорії.
3. Клієнт та сервер використовують транспортний протокол TCP для мережевого з'єднання.
4. Клієнт відправляє запит серверу з ім'ям файлу, яке вказано в аргументі його командного рядка. Сервер отримує запит від клієнта, шукає звичайний файл з вказаним ім'ям у директорії, шляхове ім'я якої вказано в аргументі його командного рядка, та відправляє клієнту вміст файлу. Клієнт записує отриманий вміст у звичайний файл.

Протокол рівня застосунку має наступні характеристики:

1. Протокол має версію. Якщо версії протоколів, які використовують клієнт та сервер не збігаються, тоді з'єднання між клієнтом та сервером треба завершити.
2. Ім'я файлу в запиті клієнта повинно складатися з символів ASCII, які дозволені для імені файлу в наявній ФС (літери, цифри, знаки пунктуації і т. ін.). Максимальна довжина імені файлу обмежена. Клієнт може надіслати які завгодно дані замість імені файлу, сервер має перевірити коректність цих даних. Клієнт має надіслати ім'я файлу, а не шляхове ім'я.
3. Сервер відправляє клієнту інформацію чи було знайдено файл з вказаним ім'ям та його розмір, у випадку помилки сервер відправляє клієнту номер помилки та завершує з'єднання. Розмір файлу не перевищує значення $(2^{64} - 1)$ байт (тобто ≤ 64 біт для значення розміру файлу в заголовку). У випадку помилки клієнт виводить інформацію про помилку та завершує з'єднання. Якщо розмір файлу перевищує вказаний максимальний розмір файлу в аргументі командного рядка клієнта, тоді клієнт відправляє серверу

повідомлення про відмову отримувати вміст файлу, інакше клієнт відправляє серверу повідомлення про готовність отримувати вміст файлу.

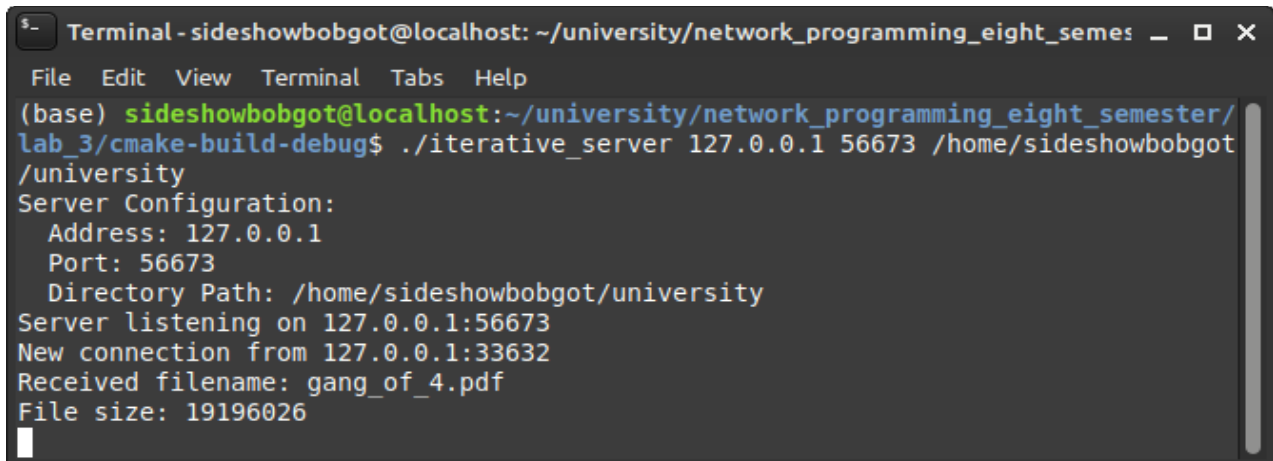
4. Якщо сервер отримав повідомлення від клієнта про готовність отримувати вміст файлу, тоді він відправляє вміст файлу частинами (тобто може потребуватися кілька викликів відповідного системного виклику для відправлення вмісту файлу). Розмір частини визначається в сервері константним значенням. Відправивши весь вміст файлу, сервер завершує з'єднання. Отримавши весь вміст файлу клієнт завершує з'єднання.

Треба реалізувати наступні реалізації серверів:

1. Ітеративний сервер, який опрацьовує запити одного клієнта повністю, перед тим, як почати опрацьовувати запити наступного клієнта.
2. Паралельний сервер, який створює нові процеси для опрацювання запитів нових клієнтів. Сервер має обмеження на максимальну кількість дочірніх процесів, які опрацьовують запити клієнтів. Ця максимальна кількість вказується в аргументі командного рядка сервера. Сервер не приймає нові TCP з'єднання від клієнтів після досягнення цієї кількості.
3. Паралельний сервер, який заздалегідь створює нові процеси для опрацювання запитів клієнтів, кожний дочірній процес є ітеративним сервером, як у першому пункті. Кількість дочірніх процесів, які має створити сервер, вказується в аргументі командного рядка сервера. Сервер не має завершувати своє виконання у випадку виникнення несистемної помилки. Для перевірки коректності роботи програм рекомендується виводити повідомлення про дії в програмах (адреси, номери портів, вміст заголовків).

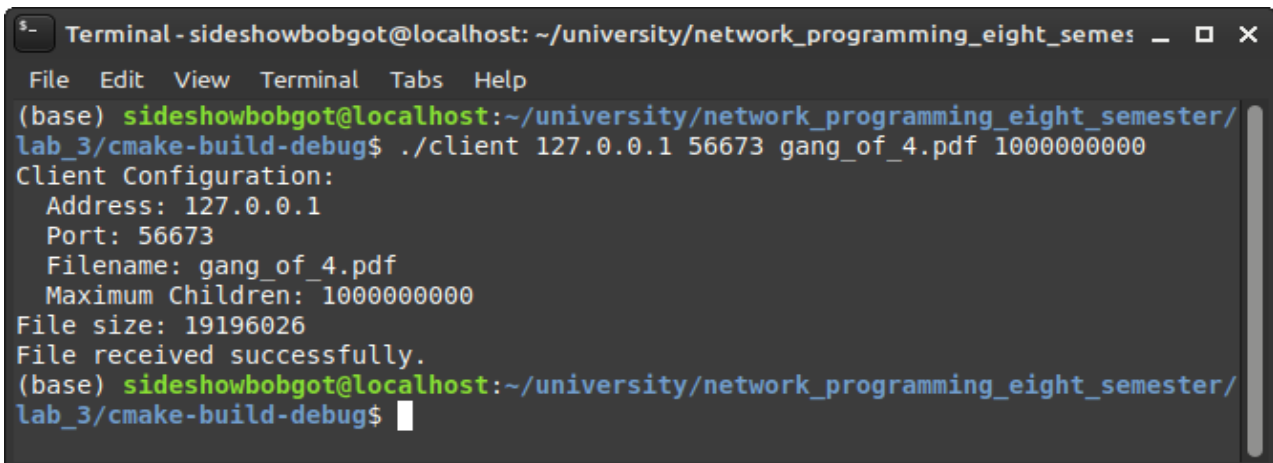
3 ВИКОНАННЯ

3.1 Ітеративний сервер



```
Terminal - sideshowbobgot@localhost: ~/university/network_programming_eight_semester
File Edit View Terminal Tabs Help
(base) sideshowbobgot@localhost:~/university/network_programming_eight_semester/
lab_3/cmake-build-debug$ ./iterative_server 127.0.0.1 56673 /home/sideshowbobgot/
university
Server Configuration:
  Address: 127.0.0.1
  Port: 56673
  Directory Path: /home/sideshowbobgot/university
Server listening on 127.0.0.1:56673
New connection from 127.0.0.1:33632
Received filename: gang_of_4.pdf
File size: 19196026
```

Рисунок 3.1 - Робота ітеративного сервера



```
Terminal - sideshowbobgot@localhost: ~/university/network_programming_eight_semester
File Edit View Terminal Tabs Help
(base) sideshowbobgot@localhost:~/university/network_programming_eight_semester/
lab_3/cmake-build-debug$ ./client 127.0.0.1 56673 gang_of_4.pdf 1000000000
Client Configuration:
  Address: 127.0.0.1
  Port: 56673
  Filename: gang_of_4.pdf
  Maximum Children: 1000000000
File size: 19196026
File received successfully.
(base) sideshowbobgot@localhost:~/university/network_programming_eight_semester/
lab_3/cmake-build-debug$
```

Рисунок 3.2 - Робота клієнта з ітеративним сервером

3.2 Паралельний сервер

Запустимо паралельний сервер з максимальною кількості процесів, що дорівнює 2. Скопіюємо gang_of_4.pdf, optimizing_cpp.pdf, modern-cmake.pdf. У результаті побачимо повідомлення, що для modern-cmake.pdf немає вільних процесів.

```

Terminal - sideshowbobgot@localhost: ~/university/network_programming_eight_semester/lab_3/
File Edit View Terminal Tabs Help
(base) sideshowbobgot@localhost:~/university/network_programming_eight_semester/lab_3/c
make-build-debug$ ./parallel_server 127.0.0.1 56673 /home/sideshowbobgot/university 2
Server Configuration:
Address: 127.0.0.1
Port: 56673
Directory Path: /home/sideshowbobgot/university
Maximum Children: 2
Server listening on 127.0.0.1:56673
New connection from 127.0.0.1:47024
Working child processes: 1
Received filename: gang_of_4.pdf
File size: 19196026
New connection from 127.0.0.1:47040
Working child processes: 2
Received filename: optimizing_cpp.pdf
File size: 1798079
New connection from 127.0.0.1:53588
No processes available
Process 37256 exited
Process 37254 exited
[]

Terminal - sideshowbobgot@localhost: ~/university/network_programming_eight_seme
File Edit View Terminal Tabs Help
(base) sideshowbobgot@localhost:~/university/network_programming_eight_semester/
lab_3/cmake-build-debug$ ./client 127.0.0.1 56673 gang_of_4.pdf 1000000000
Client Configuration:
Address: 127.0.0.1
Port: 56673
Filename: gang_of_4.pdf
Maximum Children: 1000000000
File size: 19196026
File received successfully.
(base) sideshowbobgot@localhost:~/university/network_programming_eight_semester/
lab_3/cmake-build-debug$ []

Terminal - sideshowbobgot@localhost: ~/university/network_programming_eight_seme
File Edit View Terminal Tabs Help
(base) sideshowbobgot@localhost:~/university/network_programming_eight_semester/
lab_3/cmake-build-debug$ ./client 127.0.0.1 56673 optimizing_cpp.pdf 1000000000
Client Configuration:
Address: 127.0.0.1
Port: 56673
Filename: optimizing_cpp.pdf
Maximum Children: 1000000000
File size: 1798079
File received successfully.
(base) sideshowbobgot@localhost:~/university/network_programming_eight_semester/
lab_3/cmake-build-debug$ []

Terminal - sideshowbobgot@localhost: ~/university/network_programming_eight_seme
File Edit View Terminal Tabs Help
(base) sideshowbobgot@localhost:~/university/network_programming_eight_semester/
lab_3/cmake-build-debug$ ./client 127.0.0.1 56673 modern-cmake.pdf 1000000000
Client Configuration:
Address: 127.0.0.1
Port: 56673
Filename: modern-cmake.pdf
Maximum Children: 1000000000

```

Рисунок 3.3 - Робота паралельного сервера та клієнтів

3.3 Сервер з пулом процесів

Аналогічно до пункту 3.2 запускаємо сервер та клієнти. Як бачимо, пул процесів з двох елементів опрацьовує три запити.

```

Terminal - sideshowbobgot@localhost: ~/university/network_programming_eight_semester/lab_3/
File Edit View Terminal Tabs Help
(base) sideshowbobgot@localhost:~/university/network_programming_eight_semester/lab_3/c
make-build-debug$ ./pool_server 127.0.0.1 56673 /home/sideshowbobgot/university 2
Server Configuration:
Address: 127.0.0.1
Port: 56673
Directory Path: /home/sideshowbobgot/university
Maximum Children: 2
Server listening on 127.0.0.1:56673
Created 2 child processes
New connection from 127.0.0.1:43178
Received filename: gang_of_4.pdf
File size: 19196026
New connection from 127.0.0.1:43184
Received filename: optimizing_cpp.pdf
File size: 1798079
New connection from 127.0.0.1:37766
Received filename: modern-cmake.pdf
File size: 1017294
[]

Terminal - sideshowbobgot@localhost: ~/university/network_programming_eight_seme
File Edit View Terminal Tabs Help
(base) sideshowbobgot@localhost:~/university/network_programming_eight_semester/
lab_3/cmake-build-debug$ ./client 127.0.0.1 56673 gang_of_4.pdf 1000000000
Client Configuration:
Address: 127.0.0.1
Port: 56673
Filename: gang_of_4.pdf
Maximum Children: 1000000000
File size: 19196026
File received successfully.
(base) sideshowbobgot@localhost:~/university/network_programming_eight_semester/
lab_3/cmake-build-debug$ []

Terminal - sideshowbobgot@localhost: ~/university/network_programming_eight_seme
File Edit View Terminal Tabs Help
(base) sideshowbobgot@localhost:~/university/network_programming_eight_semester/
lab_3/cmake-build-debug$ ./client 127.0.0.1 56673 optimizing_cpp.pdf 1000000000
Client Configuration:
Address: 127.0.0.1
Port: 56673
Filename: optimizing_cpp.pdf
Maximum Children: 1000000000
File size: 1798079
File received successfully.
(base) sideshowbobgot@localhost:~/university/network_programming_eight_semester/
lab_3/cmake-build-debug$ []

Terminal - sideshowbobgot@localhost: ~/university/network_programming_eight_seme
File Edit View Terminal Tabs Help
(base) sideshowbobgot@localhost:~/university/network_programming_eight_semester/
lab_3/cmake-build-debug$ ./client 127.0.0.1 56673 modern-cmake.pdf 1000000000
Client Configuration:
Address: 127.0.0.1
Port: 56673
Filename: modern-cmake.pdf
Maximum Children: 1000000000
File size: 1017294
File received successfully.
(base) sideshowbobgot@localhost:~/university/network_programming_eight_semester/
lab_3/cmake-build-debug$ []

```

Рисунок 3.3 - Робота сервера з пулом процесів та клієнтів

ДОДАТОК А ТЕКСТИ ПРОГРАМНОГО КОДУ

Тексти програмного коду
(Найменування програми (документа))

Жорсткий диск
(Вид носія даних)

(Обсяг програми (документа), арк.)

Студента групи ІП-11 4 курсу
Панченка С. В


```
// ./lab_3/pool_server.c
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <unistd.h>
```

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

```
#include <sys/wait.h>
```

```
#include <signal.h>
```

```
#include <iterative_server_utils/iterative_server_utils.h>
```

```
#include <parallel_server_utils/parallel_server_utils.h>
```

```
#define MAX_BACKLOG 10
```

```
volatile sig_atomic_t keep_running = 1;
```

```
static void handle_sigint(const int) {
```

```
    keep_running = 0;
```

```
}
```

```
static void child_process(const int listenfd, const char* const dir_path) {
```

```
    while (keep_running) {
```

```
        const int connfd = accept_connection(listenfd);
```

```
        if (connfd < 0) {
```

```
            // printf("No connectiosdfsdfsdfsfn\n");
```

```
            continue;
```

```
        }
```

```
        handle_client(connfd, dir_path);
```

```
        close(connfd);
```

```
    }
```

```
}
```

```

static void run_server(const ParallelServerConfig* const config) {
    const int listenfd = create_and_bind_socket(config->config.port, config-
>config.address);

    pid_t pids[config->max_children];
    for (int i = 0; i < config->max_children; i++) {
        pids[i] = fork();
        if (pids[i] < 0) {
            warn_err("fork()");
            exit(EXIT_FAILURE);
        }
        if (pids[i] == 0) {
            child_process(listenfd, config->config.dir_path);
            exit(EXIT_SUCCESS);
        }
    }

    printf("Created %d child processes\n", config->max_children);

    signal(SIGINT, handle_sigint);

    while (keep_running) {
        sleep(1);
    }

    printf("Shutting down...\n");

    for (int i = 0; i < config->max_children; i++) {
        kill(pids[i], SIGTERM);
    }
}

```

```

    for (int i = 0; i < config->max_children; i++) {
        waitpid(pids[i], NULL, 0);
    }

    close(listenfd);
    printf("Server shut down successfully\n");
}

int main(const int argc, char *argv[]) {
    const ParallelServerConfig config = handle_cmd_args(argc, argv);
    run_server(&config);
    return EXIT_SUCCESS;
}

// ./lab_3/iterative_server.c

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <signal.h>
#include <iterative_server_utils/iterative_server_utils.h>

static void print_config(const IterativeServerConfig *config) {
    printf("Server Configuration:\n");
    printf("  Address: %s\n", config->address);
    printf("  Port: %d\n", config->port);
    printf("  Directory Path: %s\n", config->dir_path);
}

```

```

static IterativeServerConfig handle_cmd_args(const int argc, char **argv) {
    if (argc != 4) {
        fprintf(stderr, "Usage: %s <server_address> <server_port> <directory_path>\n",
argv[0]);
        exit(EXIT_FAILURE);
    }

    const IterativeServerConfig config = {
        .address = argv[1],
        .port = atoi(argv[2]),
        .dir_path = argv[3],
    };

    print_config(&config);

    return config;
}

volatile sig_atomic_t keep_running = 1;

static void handle_sigint(const int) {
    keep_running = 0;
}

static void run_server(const IterativeServerConfig *config) {
    const int listenfd = create_and_bind_socket(config->port, config->address);
    signal(SIGINT, handle_sigint);
    while (keep_running) {
        const int connfd = accept_connection(listenfd);
        if (connfd < 0) {

```

```

        continue;
    }
    handle_client(connfd, config->dir_path);
    close(connfd);
}

close(listenfd);
}

int main(const int argc, char *argv[]) {
    const IterativeServerConfig config = handle_cmd_args(argc, argv);
    run_server(&config);
    return EXIT_SUCCESS;
}

// ./lab_3/CMakeLists.txt

cmake_minimum_required(VERSION 3.10)
project(lab_3 C)

set(CMAKE_C_STANDARD 11)

set(strict_compiler_options -Wall -Wextra -Werror)

add_library(protocol INTERFACE protocol/protocol/protocol.h)
target_include_directories(protocol INTERFACE protocol)

add_library(iterative_server_utils STATIC
    iterative_server_utils/iterative_server_utils/iterative_server_utils.h
    iterative_server_utils/iterative_server_utils.c
)

```

```
target_include_directories(iterative_server_utils PUBLIC iterative_server_utils)
target_compile_options(iterative_server_utils PRIVATE ${strict_compiler_options})
target_link_libraries(iterative_server_utils PUBLIC protocol)
```

```
add_library(parallel_server_utils STATIC
    parallel_server_utils/parallel_server_utils/parallel_server_utils.h
    parallel_server_utils/parallel_server_utils.c
)
target_include_directories(parallel_server_utils PUBLIC parallel_server_utils)
target_compile_options(parallel_server_utils PRIVATE ${strict_compiler_options})
target_link_libraries(parallel_server_utils PUBLIC iterative_server_utils)
```

```
add_executable(client client.c)
target_link_libraries(client PRIVATE protocol)
target_compile_options(client PRIVATE ${strict_compiler_options})
```

```
add_executable(iterative_server iterative_server.c)
target_link_libraries(iterative_server PRIVATE iterative_server_utils)
target_compile_options(iterative_server PRIVATE ${strict_compiler_options})
```

```
add_executable(parallel_server parallel_server.c)
target_link_libraries(parallel_server PRIVATE parallel_server_utils)
target_compile_options(parallel_server PRIVATE ${strict_compiler_options})
```

```
add_executable(pool_server pool_server.c)
target_link_libraries(pool_server PRIVATE parallel_server_utils)
target_compile_options(pool_server PRIVATE ${strict_compiler_options})
```

```
// ./lab_3/parallel_server.c
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <stdint.h>

#include <parallel_server_utils/parallel_server_utils.h>
#include <iterative_server_utils/iterative_server_utils.h>

static void handle_child_process(const int listenfd, const int connfd, const char*
const dir_path) {
    if (close(listenfd) < 0)
        exit_err("close()");
    handle_client(connfd, dir_path);
    close(connfd);
    exit(EXIT_SUCCESS);
}

static void handle_parent_process(const int connfd, int* active_children) {
    if (close(connfd) < 0)
        warn_err("close()");
    (*active_children)++;
}

volatile sig_atomic_t keep_running = 1;

static void handle_sigint(const int) {
    keep_running = 0;
}
```

```

static void reap_child_processes(int* active_children) {
    while (1) {
        const pid_t pid = waitpid(-1, NULL, WNOHANG);
        if (pid == -1)
            break;
        if (pid > 0) {
            printf("Process %jd exited\n", (intmax_t)pid);
            (*active_children)--;
        }
    }
    if (errno != ECHILD)
        exit_err("waitpid()");
}

```

```

static void run_server(const ParallelServerConfig *config) {
    const int listenfd = create_and_bind_socket(config->config.port, config-
>config.address);
    int active_children = 0;
    signal(SIGINT, handle_sigint);

    while(keep_running) {
        reap_child_processes(&active_children);

        if (active_children >= config->max_children) {
            printf("No processes available\n");
            sleep(1);
            continue;
        }

        const int connfd = accept_connection(listenfd);

```



```
if (connfd < 0)
```

```
    continue;
```

```
pid_t pid = fork();
```

```
if (pid < 0) {
```

```
    warn_err("fork()");
```

```
} else if (pid == 0) {
```

```
    handle_child_process(listenfd, connfd, config->config.dir_path);
```

```
} else {
```

```
    handle_parent_process(connfd, &active_children);
```

```
}
```

```
}
```

```
reap_child_processes(&active_children);
```

```
close(listenfd);
```

```
}
```

```
int main(const int argc, char *argv[]) {
```

```
    const ParallelServerConfig config = handle_cmd_args(argc, argv);
```

```
    run_server(&config);
```

```
    return EXIT_SUCCESS;
```

```
}
```

```
// ./lab_3/client.c
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <string.h>
```

```
#include <unistd.h>
```

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

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

```

#include <stdbool.h>

#include <protocol/protocol.h>

typedef struct {
    const char *address;
    int port;
    const char *filename;
    uint32_t max_file_size;
} ClientConfig;

static void print_config(const ClientConfig *config) {
    printf("Client Configuration:\n");
    printf("  Address: %s\n", config->address);
    printf("  Port: %d\n", config->port);
    printf("  Filename: %s\n", config->filename);
    printf("  Maximum Children: %u\n", config->max_file_size);
}

static ClientConfig handle_cmd_args(const int argc, char **argv) {
    if (argc != 5) {
        fprintf(stderr, "Usage: %s <server_address> <server_port> <filename>
<max_file_size>\n", argv[0]);
        exit(1);
    }

    const ClientConfig config = {
        .address = argv[1],
        .port = atoi(argv[2]),
        .filename = argv[3],
        .max_file_size = atol(argv[4])
    };

```

```

    print_config(&config);

    return config;
}

static int create_and_bind_socket(const int port, const char* const address) {
    const int sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock == -1) {
        perror("Socket creation failed");
        exit(1);
    }

    struct sockaddr_in server_addr;
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(port);
    if (inet_pton(AF_INET, address, &server_addr.sin_addr) <= 0) {
        perror("Invalid address");
        exit(1);
    }

    if (connect(sock, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
        perror("Connection failed");
        exit(1);
    }

    return sock;
}

static void send_receive_check_protocol_version(const int sock) {
    const int protocol_version = 1;

```

```
send(sock, &protocol_version, sizeof(protocol_version), 0);
```

```
bool match;
```

```
recv(sock, &match, sizeof(match), 0);
```

```
if(!match) {
```

```
    printf("Protocol version mismatch");
```

```
    exit(1);
```

```
}
```

```
}
```

```
static void send_filename_length(const int sock, const char* const filename) {
```

```
    const send_info_t send_info = {strlen(filename)};
```

```
    send(sock, &send_info, sizeof(send_info), 0);
```

```
    send(sock, filename, send_info.file_name_length, 0);
```

```
}
```

```
static file_size_t receive_file_info(const int sock) {
```

```
    file_existence_t file_existence;
```

```
    recv(sock, &file_existence, sizeof(file_existence), 0);
```

```
    if (file_existence == FILE_NOT_FOUND) {
```

```
        printf("File not found on server.\n");
```

```
        close(sock);
```

```
        exit(1);
```

```
}
```

```
    file_size_t file_size;
```

```
    recv(sock, &file_size, sizeof(file_size), 0);
```

```
    printf("File size: %lu\n", file_size.size);
```

```
    return file_size;
```

```
}
```

```

static size_t send_receive_readiness(const int sock, const file_size_t file_info, const
uint32_t max_file_size) {
    const file_receive_readiness_t readiness = file_info.size > max_file_size ?
REFUSE_TO_RECEIVE : READY_TO_RECEIVE;
    send(sock, &readiness, sizeof(readiness), 0);
    if(readiness == REFUSE_TO_RECEIVE) {
        printf("File size exceeds maximum allowed size.\n");
        close(sock);
        exit(1);
    }
    size_t chunk_size;
    recv(sock, &chunk_size, sizeof(chunk_size), 0);
    return chunk_size;
}

```

```

static void receive_file(
    const int sock,
    const file_size_t file_info,
    const char* const filename,
    const size_t chunk_size
) {
    // Receive and save file content
    FILE *file = fopen(filename, "wb");
    if (!file) {
        perror("Failed to open file for writing");
        close(sock);
        exit(1);
    }

    uint64_t received = 0;
    while (received < file_info.size) {

```

```

char buffer[chunk_size];
const int bytes = recv(sock, buffer, sizeof(buffer), 0);
if (bytes <= 0) break;
fwrite(buffer, 1, bytes, file);
received += bytes;
}

```

```

fclose(file);
close(sock);

```

```

if (received == file_info.size) {
    printf("File received successfully.\n");
} else {
    printf("Error: Incomplete file transfer.\n");
}
}

```

```

int main(const int argc, char *argv[]) {
    const ClientConfig config = handle_cmd_args(argc, argv);
    const int sock = create_and_bind_socket(config.port, config.address);
    send_receive_check_protocol_version(sock);
    send_filename_length(sock, config.filename);
    const file_size_t file_size = receive_file_info(sock);
        const size_t chunk_size = send_receive_readiness(sock, file_size,
config.max_file_size);
    receive_file(sock, file_size, config.filename, chunk_size);
    return EXIT_SUCCESS;
}

```

```
// ./lab_3/protocol/protocol/protocol.h
```

```
#pragma once
```

```
#include <stdint.h>
```

```
typedef enum {  
    READY_TO_RECEIVE,  
    REFUSE_TO_RECEIVE,  
} file_receive_readiness_t;
```

```
typedef enum {  
    FILE_FOUND,  
    FILE_NOT_FOUND,  
} file_existence_t;
```

```
typedef struct {  
    uint16_t file_name_length;  
} send_info_t;
```

```
typedef struct {  
    uint64_t size;  
} file_size_t;
```

```
struct error_info {  
    uint32_t error_code;  
};
```

```
// ./lab_3/parallel_server_utils/parallel_server_utils.c
```

```
#include <parallel_server_utils/parallel_server_utils.h>
```

```
#include <stdio.h>
```

```

#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

static void print_config(const ParallelServerConfig *config) {
    printf("Server Configuration:\n");
    printf("  Address: %s\n", config->config.address);
    printf("  Port: %d\n", config->config.port);
    printf("  Directory Path: %s\n", config->config.dir_path);
    printf("  Maximum Children: %d\n", config->max_children);
}

ParallelServerConfig handle_cmd_args(const int argc, char **argv) {
    if (argc != 5) {
        fprintf(stderr, "Usage: %s <server_address> <server_port> <directory_path>
<max_children>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    const ParallelServerConfig config = {
        .config.address = argv[1],
        .config.port = atoi(argv[2]),
        .config.dir_path = argv[3],
        .max_children = atoi(argv[4])
    };

    print_config(&config);

    return config;
}

```



```
// ./lab_3/parallel_server_utils/parallel_server_utils/parallel_server_utils.h
```

```
#pragma once
```

```
#include <iterative_server_utils/iterative_server_utils.h>
```

```
typedef struct {  
    IterativeServerConfig config;  
    int max_children;  
} ParallelServerConfig;
```

```
ParallelServerConfig handle_cmd_args(int argc, char **argv);
```

```
// ./lab_3/iterative_server_utils/iterative_server_utils.c
```

```
#include <iterative_server_utils/iterative_server_utils.h>
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <string.h>
```

```
#include <unistd.h>
```

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

```
#include <sys/stat.h>
```

```
#include <dirent.h>
```

```
#include <sys/types.h>
```

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

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

```
#include <stdbool.h>
```

```
#include <protocol/protocol.h>
```

```
void exit_err(const char *msg) {
    perror(msg);
    exit(EXIT_FAILURE);
}
```

```
void warn_err(const char *msg) {
    perror(msg);
}
```

```
static bool receive_send_check_protocol_version(const int client_sock) {
    int client_protocol_version = 0;
    recv(client_sock, &client_protocol_version, sizeof(client_protocol_version), 0);
    const int server_protocol_version = 1;
    const bool protocol_version_match = client_protocol_version ==
server_protocol_version;
    send(client_sock, &protocol_version_match, sizeof(protocol_version_match), 0);
    if(!protocol_version_match) {
        printf("Protocol version mismatch.\n");
        close(client_sock);
        return false;
    }
    return true;
}
```

```
#define MAX_FILENAME_LENGTH 256
```

```
static bool check_filename(
    const int client_sock,
    const send_info_t send_info,
    const char* const dir_path,
    char* filepath,
```

```

    const size_t file_path_max_length
) {
    char filename[MAX_FILENAME_LENGTH];
    recv(client_sock, filename, send_info.file_name_length, 0);
    printf("Received filename: %s\n", filename);
    filename[send_info.file_name_length] = '\0';

    snprintf(filepath, file_path_max_length, "%s/%s", dir_path, filename);

    struct stat st;
    if (stat(filepath, &st) == -1 || !S_ISREG(st.st_mode)) {
        const file_existence_t file_existence = FILE_NOT_FOUND;
        send(client_sock, &file_existence, sizeof(file_existence), 0);
        close(client_sock);
        return false;
    }
    const file_existence_t file_existence = FILE_FOUND;
    send(client_sock, &file_existence, sizeof(file_existence), 0);
    const file_size_t file_size = {.size = st.st_size};
    printf("File size: %lu\n", file_size.size);
    send(client_sock, &file_size, sizeof(file_size), 0);
    return true;
}

const size_t CHUNK_SIZE = 4096;

static bool check_file_readiness(const int client_sock) {
    file_receive_readiness_t file_receive_readiness;
    recv(client_sock, &file_receive_readiness, sizeof(file_receive_readiness), 0);
    if (file_receive_readiness == REFUSE_TO_RECEIVE) {
        printf("Client refused to receive file.\n");
    }
}

```

```

        close(client_sock);
        return false;
    }
    send(client_sock, &CHUNK_SIZE, sizeof(CHUNK_SIZE), 0);
    return true;
}

void handle_client(
    const int client_sock,
    const char* const dir_path
) {
    receive_send_check_protocol_version(client_sock);

    send_info_t send_info;
    recv(client_sock, &send_info, sizeof(send_info), 0);

    char filepath[PATH_MAX];
    if(!check_filename(client_sock, send_info, dir_path, filepath, sizeof(filepath))) {
        return;
    }
    if(!check_file_readiness(client_sock)) {
        return;
    }

    FILE *file = fopen(filepath, "rb");
    if (!file) {
        perror("Failed to open file");
        close(client_sock);
        return;
    }

```

```

char buffer[CHUNK_SIZE];
size_t bytes_read;
while ((bytes_read = fread(buffer, 1, sizeof(buffer), file)) > 0) {
    printf("Process %d read bytes: %lu\n", getpid(), bytes_read);
    send(client_sock, buffer, bytes_read, 0);
}

```

```

fclose(file);
close(client_sock);
}

```

```
#define MAX_BACKLOG 10
```

```

int create_and_bind_socket(const int port, const char* const address) {
    const int listenfd = socket(AF_INET, SOCK_STREAM | SOCK_NONBLOCK,
IPPROTO_TCP);
    if (listenfd < 0)
        exit_err("socket()");

```

// Проблема була у тому, що після перезапуску сервера я не міг перевикористати той самий сокет.

// Як пояснила документація: треба явно прописати перевикористання.

```

{
    const int optval = 1;
    if (setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &optval,
sizeof(optval)) < 0)
        exit_err("setsockopt(SO_REUSEADDR)");
}

```

```

struct sockaddr_in srv_sin4 = {0};
srv_sin4.sin_family = AF_INET;

```

```

    srv_sin4.sin_port = htons(port);
    srv_sin4.sin_addr.s_addr = inet_addr(address);

    if (bind(listenfd, (struct sockaddr *)&srv_sin4, sizeof(srv_sin4)) < 0)
        exit_err("bind()");

    if (listen(listenfd, MAX_BACKLOG) < 0)
        exit_err("listen()");

    printf("Server listening on %s:%d\n", address, port);
    return listenfd;
}

int accept_connection(const int listenfd) {
    struct sockaddr_in cln_sin4;
    socklen_t addrlen = sizeof(cln_sin4);
    const int connfd = accept(listenfd, (struct sockaddr *)&cln_sin4, &addrlen);
    if (connfd < 0) {
        return -1;
    }

    printf("New connection from %s:%d\n", inet_ntoa(cln_sin4.sin_addr),
ntohs(cln_sin4.sin_port));
    return connfd;
}

// ./lab_3/iterative_server_utils/iterative_server_utils/iterative_server_utils.h

#pragma once

typedef struct {
    const char *address;

```

```
    int port;  
    const char *dir_path;  
} IterativeServerConfig;
```

```
void exit_err(const char *msg);  
void warn_err(const char *msg);  
void handle_client(int client_sock, const char* dir_path);  
int create_and_bind_socket(int port, const char* address);  
int accept_connection(int listenfd);
```