Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інтегрованих інформаційних систем

Лабораторна робота №3

Мережеве програмування у середовищі UNIX

Тема: «Багатопроцесний ітеративний TCP клієнт-сервер»

Виконав:
Студент групи ІА-12
Оверчук Дмитро Максимович

Перевірив:
Сімоненко А.В.

Київ 2025

Завдання на роботу

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 ди- ректорії, шляхове ім'я якої вказано в аргументі його командного рядка, та відправляє кліє- нту 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 збі- гаються, 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 серер відправляє клієнту номер помилки та завершує з'єднан- ня. Розмір файлу не перевищує значення (2^64 - 1) байт (тобто є 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 го- товність 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 ви- значається 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 поча- ти 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 кі- лькості.
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.

Tips:

- 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).

Код програми

protocol.h

```
#ifndef PROTOCOL_H
#define PROTOCOL_H


#include <stdint.h>


// Версія протоколу
#define PROTOCOL_VERSION 1


// Максимальна довжина імені файлу (без нульового символу)
#define MAX_FILENAME_LENGTH 255


// Коди помилок
#define ERR_PROTOCOL_MISMATCH 1
#define ERR_INVALID_FILENAME 2
#define ERR_FILE_NOT_FOUND 3
```

```
#define ERR_FILE_ACCESS 4
#define ERR_INTERNAL 5

// Типи повідомлень
#define MSG_FILE_REQUEST     1
#define MSG_FILE_RESPONSE    2
#define MSG_READY_TO_RECEIVE 3
#define MSG_REFUSE_TO_RECEIVE 4

// Розмір блоку при передачі файлу (64КБ)
#define DEFAULT_CHUNK_SIZE (64 * 1024)

// Заголовок запиту від клієнта до сервера
typedef struct {
    uint8_t protocol_version;
    uint8_t message_type;
    uint16_t filename_length;  // Довжина імені файлу
    char filename[MAX_FILENAME_LENGTH + 1];  // +1 для термінатора '\0'
} FileRequestHeader;

// Заголовок відповіді від сервера до клієнта
typedef struct {
    uint8_t protocol_version;
    uint8_t message_type;
    uint8_t status;  // 0 = успіх, інакше код помилки
    uint64_t file_size;  // Розмір файлу у байтах
} FileResponseHeader;

// Заголовок відповіді клієнта серверу
typedef struct {
    uint8_t message_type;  // MSG_READY_TO_RECEIVE або MSG_REFUSE_TO_RECEIVE
} ClientResponseHeader;
```

```
#endif // PROTOCOL_H
```

utils.h

```
#ifndef UTILS_H
#define UTILS_H

#include <string.h>

// Function to validate a filename (no path separators or invalid characters)
static inline int validate_filename(const char *filename) {
    // Check for null or empty filename
    if (!filename || !*filename) {
        return 0;
    }

    // Check for path separators
    if (strchr(filename, '/') != NULL) {
        return 0;
    }

    // Check for "." or ".." which could be used to navigate directory structure
    if (strcmp(filename, ".") == 0 || strcmp(filename, "..") == 0) {
        return 0;
    }

    // Basic check for valid filename characters
    for (const char *p = filename; *p; p++) {
        if (*p <= 31 || *p == 127) {  // Control characters
            return 0;
        }

        // This is a simplified check - actual filesystem restrictions may vary
        if (strchr("<>:\"|?*\\", *p) != NULL) {
            return 0;
```

```
        }
    }

    return 1;
}

#endif // UTILS_H
```

client.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdint.h>
#include <errno.h>
#include <fcntl.h>
#include "protocol.h"

// Function to print error messages
void print_error(const char *message) {
    perror(message);
    exit(EXIT_FAILURE);
}

// Function to print error code description
void print_error_code(int code) {
    printf("Error: ");
    switch (code) {
        case ERR_PROTOCOL_MISMATCH:
            printf("Protocol version mismatch\n");
            break;
```

```
        case ERR_INVALID_FILENAME:
            printf("Invalid filename\n");
            break;
        case ERR_FILE_NOT_FOUND:
            printf("File not found\n");
            break;
        case ERR_FILE_ACCESS:
            printf("Access to file denied\n");
            break;
        case ERR_INTERNAL:
            printf("Internal server error\n");
            break;
        default:
            printf("Unknown error (code %d)\n", code);
    }
}

// Main function
int main(int argc, char *argv[]) {
    if (argc != 5) {
        fprintf(stderr, "Usage: %s <server_address> <server_port> <filename> <max_file_size>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    // Parse command line arguments
    const char *server_address = argv[1];
    int server_port = atoi(argv[2]);
    const char *filename = argv[3];
    uint64_t max_file_size = strtoull(argv[4], NULL, 10);

    // Validate filename length
    size_t filename_len = strlen(filename);
    if (filename_len == 0 || filename_len > MAX_FILENAME_LENGTH) {
```

```
        fprintf(stderr, "Error: Filename must be between 1 and %d characters\n",
MAX_FILENAME_LENGTH);
        exit(EXIT_FAILURE);
    }

    // Create socket
    int sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
        print_error("Socket creation failed");
    }

    // Connect to server
    struct sockaddr_in server_addr;
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(server_port);

    if (inet_pton(AF_INET, server_address, &server_addr.sin_addr) <= 0) {
        fprintf(stderr, "Invalid address/ Address not supported\n");
        close(sock);
        exit(EXIT_FAILURE);
    }

    printf("Connecting to server at %s:%d\n", server_address, server_port);
    if (connect(sock, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0)
{
        print_error("Connection failed");
    }
    printf("Connected to server\n");

    // Prepare file request
    FileRequestHeader request;
    memset(&request, 0, sizeof(request));
    request.protocol_version = PROTOCOL_VERSION;
```

```
    request.message_type = MSG_FILE_REQUEST;
    request.filename_length = filename_len;
    strncpy(request.filename, filename, MAX_FILENAME_LENGTH);

    printf("Sending file request for: %s\n", filename);

    // Send file request
    if (send(sock, &request, sizeof(request), 0) != sizeof(request)) {
        print_error("Failed to send file request");
    }

    // Receive server response
    FileResponseHeader response;
    ssize_t bytes_received = recv(sock, &response, sizeof(response), 0);
    if (bytes_received != sizeof(response)) {
        print_error("Failed to receive response header");
    }

    printf("Received response: protocol version %d, message type %d, status %d,
file size %lu\n",
           response.protocol_version, response.message_type, response.status,
response.file_size);

    // Check protocol version
    if (response.protocol_version != PROTOCOL_VERSION) {
        fprintf(stderr, "Error: Protocol version mismatch\n");
        close(sock);
        exit(EXIT_FAILURE);
    }

    // Check for error
    if (response.status != 0) {
        print_error_code(response.status);
        close(sock);
```

```
        exit(EXIT_FAILURE);
    }

    // Check file size against maximum
    ClientResponseHeader client_response;
    if (response.file_size > max_file_size) {
        printf("File size (%lu bytes) exceeds maximum allowed size (%lu bytes)\n",
               response.file_size, max_file_size);
        client_response.message_type = MSG_REFUSE_TO_RECEIVE;
        send(sock, &client_response, sizeof(client_response), 0);
        close(sock);
        exit(EXIT_FAILURE);
    }

    // Send ready to receive
    client_response.message_type = MSG_READY_TO_RECEIVE;
    printf("Sending ready to receive message\n");
    if (send(sock, &client_response, sizeof(client_response), 0) != sizeof(client_response)) {
        print_error("Failed to send ready message");
    }

    // Create output file
    int file_fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (file_fd < 0) {
        print_error("Failed to create output file");
    }

    // Receive file content
    printf("Receiving file content (%lu bytes)...\n", response.file_size);
    uint64_t total_received = 0;
    char buffer[4096];
```

```
    while (total_received < response.file_size) {
        bytes_received = recv(sock, buffer, sizeof(buffer), 0);
        if (bytes_received <= 0) {
            if (bytes_received == 0) {
                printf("Connection closed by server\n");
                break;
            } else {
                print_error("Error receiving file content");
            }
        }

        if (write(file_fd, buffer, bytes_received) != bytes_received) {
            print_error("Failed to write to output file");
        }

        total_received += bytes_received;
        printf("\rReceived: %lu/%lu bytes (%.1f%%)",
               total_received, response.file_size,
               (float)total_received / response.file_size * 100);
        fflush(stdout);
    }
    printf("\nFile transfer complete\n");

    // Close file and socket
    close(file_fd);
    close(sock);

    return 0;
}
```

iterative_server.c

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

```c
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <stdint.h>
#include <errno.h>
#include <ctype.h>
#include <dirent.h>
#include <limits.h>
#include "protocol.h"
#include "utils.h"

// Function to print error messages
void print_error(const char *message) {
    perror(message);
}

// Function to handle a client connection
void handle_client(int client_sock, const char *directory) {
    // Receive file request
    FileRequestHeader request;
    ssize_t bytes_received = recv(client_sock, &request, sizeof(request), 0);

    if (bytes_received != sizeof(request)) {
        printf("Error: Failed to receive file request (received %zd bytes, expected %zu)\n",
               bytes_received, sizeof(request));
        close(client_sock);
        return;
    }

    // Prepare response
```

```
    FileResponseHeader response;
    memset(&response, 0, sizeof(response));
    response.protocol_version = PROTOCOL_VERSION;
    response.message_type = MSG_FILE_RESPONSE;

    // Check protocol version
    if (request.protocol_version != PROTOCOL_VERSION) {
        printf("Error: Protocol version mismatch (client: %d, server: %d)\n",
               request.protocol_version, PROTOCOL_VERSION);
        response.status = ERR_PROTOCOL_MISMATCH;
        send(client_sock, &response, sizeof(response), 0);
        close(client_sock);
        return;
    }

    // Null-terminate filename to be safe
    request.filename[request.filename_length] = '\0';

    printf("Received file request for: %s\n", request.filename);

    // Validate filename
    if (!validate_filename(request.filename)) {
        printf("Error: Invalid filename: %s\n", request.filename);
        response.status = ERR_INVALID_FILENAME;
        send(client_sock, &response, sizeof(response), 0);
        close(client_sock);
        return;
    }

    // Build full path
    char filepath[PATH_MAX];
    snprintf(filepath, sizeof(filepath), "%s/%s", directory, request.filename);
```

```
    // Check if the file exists and is a regular file
    struct stat file_stat;
    if (stat(filepath, &file_stat) != 0) {
        printf("Error: File not found: %s\n", filepath);
        response.status = ERR_FILE_NOT_FOUND;
        send(client_sock, &response, sizeof(response), 0);
        close(client_sock);
        return;
    }

    if (!S_ISREG(file_stat.st_mode)) {
        printf("Error: Not a regular file: %s\n", filepath);
        response.status = ERR_FILE_NOT_FOUND;
        send(client_sock, &response, sizeof(response), 0);
        close(client_sock);
        return;
    }

    // Get file size
    uint64_t file_size = file_stat.st_size;

    // Prepare success response
    response.status = 0;  // Success
    response.file_size = file_size;

    printf("Sending file response: status=%d, file_size=%lu\n",
           response.status, response.file_size);

    // Send response
    if (send(client_sock, &response, sizeof(response), 0) != sizeof(response)) {
        print_error("Failed to send file response");
        close(client_sock);
        return;
```

```
    }


    // Receive client's decision
    ClientResponseHeader client_response;
    bytes_received = recv(client_sock, &client_response,
sizeof(client_response), 0);


    if (bytes_received != sizeof(client_response)) {
        printf("Error: Failed to receive client response\n");
        close(client_sock);
        return;
    }


    if (client_response.message_type == MSG_REFUSE_TO_RECEIVE) {
        printf("Client refused to receive the file (too large)\n");
        close(client_sock);
        return;
    }


    if (client_response.message_type != MSG_READY_TO_RECEIVE) {
        printf("Error: Unexpected client response message type: %d\n",
client_response.message_type);
        close(client_sock);
        return;
    }


    printf("Client is ready to receive the file\n");


    // Open the file
    int file_fd = open(filepath, O_RDONLY);
    if (file_fd < 0) {
        print_error("Failed to open file");
        close(client_sock);
        return;
```

```
    }

    // Send file content in chunks
    char buffer[DEFAULT_CHUNK_SIZE];
    ssize_t bytes_read;
    uint64_t total_sent = 0;

    printf("Sending file content (%lu bytes)...\n", file_size);

    while ((bytes_read = read(file_fd, buffer, sizeof(buffer))) > 0) {
        ssize_t bytes_sent = send(client_sock, buffer, bytes_read, 0);
        if (bytes_sent != bytes_read) {
            print_error("Failed to send file content");
            close(file_fd);
            close(client_sock);
            return;
        }

        total_sent += bytes_sent;
        printf("\rSent: %lu/%lu bytes (%.1f%%)",
               total_sent, file_size,
               (float)total_sent / file_size * 100);
        fflush(stdout);
    }

    printf("\nFile transfer complete\n");

    // Close file and client connection
    close(file_fd);
    close(client_sock);
}

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

```
    if (argc != 4) {
        fprintf(stderr, "Usage: %s <server_address> <server_port> <directory>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    // Parse command line arguments
    const char *server_address = argv[1];
    int server_port = atoi(argv[2]);
    const char *directory = argv[3];

    // Check if directory exists
    DIR *dir = opendir(directory);
    if (!dir) {
        fprintf(stderr, "Error: Directory '%s' does not exist or is not accessible\n", directory);
        exit(EXIT_FAILURE);
    }
    closedir(dir);

    // Create socket
    int server_sock = socket(AF_INET, SOCK_STREAM, 0);
    if (server_sock < 0) {
        perror("Socket creation failed");
        exit(EXIT_FAILURE);
    }

    // Set socket option to reuse address
    int opt = 1;
    if (setsockopt(server_sock, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt)) < 0) {
        perror("Setsockopt failed");
        exit(EXIT_FAILURE);
    }
```

```
    // Bind socket to specified address and port
    struct sockaddr_in server_addr;
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(server_port);

    if (inet_pton(AF_INET, server_address, &server_addr.sin_addr) <= 0) {
        fprintf(stderr, "Invalid address/ Address not supported\n");
        exit(EXIT_FAILURE);
    }

    if (bind(server_sock, (struct sockaddr *)&server_addr, sizeof(server_addr))
< 0) {
        perror("Bind failed");
        exit(EXIT_FAILURE);
    }

    // Listen for connections
    if (listen(server_sock, 10) < 0) {
        perror("Listen failed");
        exit(EXIT_FAILURE);
    }

    printf("Iterative server started at %s:%d, serving files from '%s'\n",
           server_address, server_port, directory);

    // Main server loop
    while (1) {
        struct sockaddr_in client_addr;
        socklen_t client_len = sizeof(client_addr);

        printf("Waiting for connections...\n");

        // Accept a new connection
```

```
        int client_sock = accept(server_sock, (struct sockaddr *)&client_addr,
&client_len);
        if (client_sock < 0) {
            print_error("Accept failed");
            continue;  // Continue accepting connections
        }

        // Get client information
        char client_ip[INET_ADDRSTRLEN];
        inet_ntop(AF_INET, &client_addr.sin_addr, client_ip, INET_ADDRSTRLEN);
        int client_port = ntohs(client_addr.sin_port);

        printf("Accepted connection from %s:%d\n", client_ip, client_port);

        // Handle client in the same process (iterative server)
        handle_client(client_sock, directory);

        printf("Client connection handled and closed\n");
    }

    // Close server socket (never reached in this example)
    close(server_sock);

    return 0;
}
```

parallel_server.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

```c
#include <fcntl.h>
#include <sys/stat.h>
#include <stdint.h>
#include <errno.h>
#include <ctype.h>
#include <dirent.h>
#include <sys/wait.h>
#include <signal.h>
#include "protocol.h"
#include "utils.h"

// Global variables to track child processes
volatile int active_children = 0;
int max_children = 0;

// Function to print error messages
void print_error(const char *message) {
    perror(message);
}

// Signal handler for child process termination
void sigchld_handler(int sig) {
    int saved_errno = errno;
    while (waitpid(-1, NULL, WNOHANG) > 0) {
        active_children--;
    }
    errno = saved_errno;
}

// Function to handle a client connection
void handle_client(int client_sock, const char *directory) {
    // Receive file request
    FileRequestHeader request;
```

```
    ssize_t bytes_received = recv(client_sock, &request, sizeof(request), 0);

    if (bytes_received != sizeof(request)) {
        printf("Error: Failed to receive file request (received %zd bytes,
expected %zu)\n",
               bytes_received, sizeof(request));
        close(client_sock);
        return;
    }

    // Prepare response
    FileResponseHeader response;
    memset(&response, 0, sizeof(response));
    response.protocol_version = PROTOCOL_VERSION;
    response.message_type = MSG_FILE_RESPONSE;

    // Check protocol version
    if (request.protocol_version != PROTOCOL_VERSION) {
        printf("Error: Protocol version mismatch (client: %d, server: %d)\n",
               request.protocol_version, PROTOCOL_VERSION);
        response.status = ERR_PROTOCOL_MISMATCH;
        send(client_sock, &response, sizeof(response), 0);
        close(client_sock);
        return;
    }

    // Null-terminate filename to be safe
    request.filename[request.filename_length] = '\0';

    printf("[PID %d] Received file request for: %s\n", getpid(),
request.filename);

    // Validate filename
    if (!validate_filename(request.filename)) {
```

```
        printf("[PID %d] Error: Invalid filename: %s\n", getpid(),
request.filename);
        response.status = ERR_INVALID_FILENAME;
        send(client_sock, &response, sizeof(response), 0);
        close(client_sock);
        return;
    }

    // Build full path
    char filepath[PATH_MAX];
    snprintf(filepath, sizeof(filepath), "%s/%s", directory, request.filename);

    // Check if the file exists and is a regular file
    struct stat file_stat;
    if (stat(filepath, &file_stat) != 0) {
        printf("[PID %d] Error: File not found: %s\n", getpid(), filepath);
        response.status = ERR_FILE_NOT_FOUND;
        send(client_sock, &response, sizeof(response), 0);
        close(client_sock);
        return;
    }

    if (!S_ISREG(file_stat.st_mode)) {
        printf("[PID %d] Error: Not a regular file: %s\n", getpid(), filepath);
        response.status = ERR_FILE_NOT_FOUND;
        send(client_sock, &response, sizeof(response), 0);
        close(client_sock);
        return;
    }

    // Get file size
    uint64_t file_size = file_stat.st_size;

    // Prepare success response
```

```
    response.status = 0;  // Success
    response.file_size = file_size;

    printf("[PID %d] Sending file response: status=%d, file_size=%lu\n",
           getpid(), response.status, response.file_size);

    // Send response
    if (send(client_sock, &response, sizeof(response), 0) != sizeof(response)) {
        print_error("Failed to send file response");
        close(client_sock);
        return;
    }

    // Receive client's decision
    ClientResponseHeader client_response;
    bytes_received = recv(client_sock, &client_response,
sizeof(client_response), 0);

    if (bytes_received != sizeof(client_response)) {
        printf("[PID %d] Error: Failed to receive client response\n", getpid());
        close(client_sock);
        return;
    }

    if (client_response.message_type == MSG_REFUSE_TO_RECEIVE) {
        printf("[PID %d] Client refused to receive the file (too large)\n",
getpid());
        close(client_sock);
        return;
    }

    if (client_response.message_type != MSG_READY_TO_RECEIVE) {
        printf("[PID %d] Error: Unexpected client response message type: %d\n",
               getpid(), client_response.message_type);
```

```
        close(client_sock);
        return;
    }

    printf("[PID %d] Client is ready to receive the file\n", getpid());

    // Open the file
    int file_fd = open(filepath, O_RDONLY);
    if (file_fd < 0) {
        print_error("Failed to open file");
        close(client_sock);
        return;
    }

    // Send file content in chunks
    char buffer[DEFAULT_CHUNK_SIZE];
    ssize_t bytes_read;
    uint64_t total_sent = 0;

    printf("[PID %d] Sending file content (%lu bytes)...\n", getpid(),
file_size);

    while ((bytes_read = read(file_fd, buffer, sizeof(buffer))) > 0) {
        ssize_t bytes_sent = send(client_sock, buffer, bytes_read, 0);
        if (bytes_sent != bytes_read) {
            print_error("Failed to send file content");
            close(file_fd);
            close(client_sock);
            return;
        }

        total_sent += bytes_sent;
        printf("[PID %d] \rSent: %lu/%lu bytes (%.1f%%)",
               getpid(), total_sent, file_size,
```

```
                (float)total_sent / file_size * 100);
        fflush(stdout);
    }

    printf("\n[PID %d] File transfer complete\n", getpid());

    // Close file and client connection
    close(file_fd);
    close(client_sock);
}

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

    // Parse command line arguments
    const char *server_address = argv[1];
    int server_port = atoi(argv[2]);
    const char *directory = argv[3];
    max_children = atoi(argv[4]);

    if (max_children <= 0) {
        fprintf(stderr, "Error: max_children must be a positive integer\n");
        exit(EXIT_FAILURE);
    }

    // Check if directory exists
    DIR *dir = opendir(directory);
    if (!dir) {
        fprintf(stderr, "Error: Directory '%s' does not exist or is not
accessible\n", directory);
```

```
        exit(EXIT_FAILURE);
    }
    closedir(dir);

    // Set up signal handler for child process termination
    struct sigaction sa;
    sa.sa_handler = sigchld_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;
    if (sigaction(SIGCHLD, &sa, NULL) == -1) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }

    // Create socket
    int server_sock = socket(AF_INET, SOCK_STREAM, 0);
    if (server_sock < 0) {
        perror("Socket creation failed");
        exit(EXIT_FAILURE);
    }

    // Set socket option to reuse address
    int opt = 1;
    if (setsockopt(server_sock, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt)) <
0) {
        perror("Setsockopt failed");
        exit(EXIT_FAILURE);
    }

    // Bind socket to specified address and port
    struct sockaddr_in server_addr;
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(server_port);
```

```
    if (inet_pton(AF_INET, server_address, &server_addr.sin_addr) <= 0) {
        fprintf(stderr, "Invalid address/ Address not supported\n");
        exit(EXIT_FAILURE);
    }

    if (bind(server_sock, (struct sockaddr *)&server_addr, sizeof(server_addr))
< 0) {
        perror("Bind failed");
        exit(EXIT_FAILURE);
    }

    // Listen for connections
    if (listen(server_sock, 10) < 0) {
        perror("Listen failed");
        exit(EXIT_FAILURE);
    }

    printf("Parallel server started at %s:%d, serving files from '%s', max
children: %d\n",
           server_address, server_port, directory, max_children);

    // Main server loop
    while (1) {
        // If we've reached the maximum number of children, wait
        while (active_children >= max_children) {
            printf("Max children reached (%d), waiting for a child to
exit...\n", max_children);
            sleep(1);  // Wait a bit to avoid busy-waiting
        }

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

        printf("Waiting for connections (active children: %d/%d)...\n",
```

```
                active_children, max_children);

        // Accept a new connection
        int client_sock = accept(server_sock, (struct sockaddr *)&client_addr,
&client_len);
        if (client_sock < 0) {
            if (errno == EINTR) {
                // Interrupted by signal, try again
                continue;
            }
            print_error("Accept failed");
            continue;  // Continue accepting connections
        }

        // Get client information
        char client_ip[INET_ADDRSTRLEN];
        inet_ntop(AF_INET, &client_addr.sin_addr, client_ip, INET_ADDRSTRLEN);
        int client_port = ntohs(client_addr.sin_port);

        printf("Accepted connection from %s:%d\n", client_ip, client_port);

        // Fork a child process to handle the client
        pid_t pid = fork();

        if (pid < 0) {
            print_error("Fork failed");
            close(client_sock);
            continue;
        }

        if (pid == 0) {
            // Child process
            close(server_sock);  // Child doesn't need the server socket
```

```
            printf("[PID %d] Child process created to handle client %s:%d\n",
                   getpid(), client_ip, client_port);

            handle_client(client_sock, directory);

            printf("[PID %d] Client connection handled and closed\n", getpid());
            exit(EXIT_SUCCESS);
        } else {
            // Parent process
            active_children++;
            printf("Child process created (PID: %d), active children: %d/%d\n",
                   pid, active_children, max_children);
            close(client_sock);  // Parent doesn't need the client socket
        }
    }

    // Close server socket (never reached in this example)
    close(server_sock);

    return 0;
}
```

preforked_server.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <stdint.h>
#include <errno.h>
```

```c
#include <ctype.h>
#include <dirent.h>
#include <sys/wait.h>
#include <signal.h>
#include "protocol.h"
#include "utils.h"

// Global variables for signal handling
volatile sig_atomic_t terminate = 0;  // Flag to terminate child processes

// Function to print error messages
void print_error(const char *message) {
    perror(message);
}

// Signal handler for termination signals
void termination_handler(int sig) {
    terminate = 1;
}

// Function to handle a client connection
void handle_client(int client_sock, const char *directory) {
    // Receive file request
    FileRequestHeader request;
    ssize_t bytes_received = recv(client_sock, &request, sizeof(request), 0);

    if (bytes_received != sizeof(request)) {
        printf("[PID %d] Error: Failed to receive file request (received %zd bytes, expected %zu)\n",
               getpid(), bytes_received, sizeof(request));
        close(client_sock);
        return;
    }
```

```
    // Prepare response
    FileResponseHeader response;
    memset(&response, 0, sizeof(response));
    response.protocol_version = PROTOCOL_VERSION;
    response.message_type = MSG_FILE_RESPONSE;

    // Check protocol version
    if (request.protocol_version != PROTOCOL_VERSION) {
        printf("[PID %d] Error: Protocol version mismatch (client: %d, server: %d)\n",
               getpid(), request.protocol_version, PROTOCOL_VERSION);
        response.status = ERR_PROTOCOL_MISMATCH;
        send(client_sock, &response, sizeof(response), 0);
        close(client_sock);
        return;
    }

    // Null-terminate filename to be safe
    request.filename[request.filename_length] = '\0';

    printf("[PID %d] Received file request for: %s\n", getpid(), request.filename);

    // Validate filename
    if (!validate_filename(request.filename)) {
        printf("[PID %d] Error: Invalid filename: %s\n", getpid(), request.filename);
        response.status = ERR_INVALID_FILENAME;
        send(client_sock, &response, sizeof(response), 0);
        close(client_sock);
        return;
    }

    // Build full path
    char filepath[PATH_MAX];
```

```
    snprintf(filepath, sizeof(filepath), "%s/%s", directory, request.filename);

    // Check if the file exists and is a regular file
    struct stat file_stat;
    if (stat(filepath, &file_stat) != 0) {
        printf("[PID %d] Error: File not found: %s\n", getpid(), filepath);
        response.status = ERR_FILE_NOT_FOUND;
        send(client_sock, &response, sizeof(response), 0);
        close(client_sock);
        return;
    }

    if (!S_ISREG(file_stat.st_mode)) {
        printf("[PID %d] Error: Not a regular file: %s\n", getpid(), filepath);
        response.status = ERR_FILE_NOT_FOUND;
        send(client_sock, &response, sizeof(response), 0);
        close(client_sock);
        return;
    }

    // Get file size
    uint64_t file_size = file_stat.st_size;

    // Prepare success response
    response.status = 0;  // Success
    response.file_size = file_size;

    printf("[PID %d] Sending file response: status=%d, file_size=%lu\n",
           getpid(), response.status, response.file_size);

    // Send response
    if (send(client_sock, &response, sizeof(response), 0) != sizeof(response)) {
        print_error("Failed to send file response");
```

```
        close(client_sock);
        return;
    }

    // Receive client's decision
    ClientResponseHeader client_response;
    bytes_received = recv(client_sock, &client_response,
sizeof(client_response), 0);

    if (bytes_received != sizeof(client_response)) {
        printf("[PID %d] Error: Failed to receive client response\n", getpid());
        close(client_sock);
        return;
    }

    if (client_response.message_type == MSG_REFUSE_TO_RECEIVE) {
        printf("[PID %d] Client refused to receive the file (too large)\n",
getpid());
        close(client_sock);
        return;
    }

    if (client_response.message_type != MSG_READY_TO_RECEIVE) {
        printf("[PID %d] Error: Unexpected client response message type: %d\n",
               getpid(), client_response.message_type);
        close(client_sock);
        return;
    }

    printf("[PID %d] Client is ready to receive the file\n", getpid());

    // Open the file
    int file_fd = open(filepath, O_RDONLY);
    if (file_fd < 0) {
```

```
        print_error("Failed to open file");
        close(client_sock);
        return;
    }

    // Send file content in chunks
    char buffer[DEFAULT_CHUNK_SIZE];
    ssize_t bytes_read;
    uint64_t total_sent = 0;

    printf("[PID %d] Sending file content (%lu bytes)...\n", getpid(),
file_size);

    while ((bytes_read = read(file_fd, buffer, sizeof(buffer))) > 0) {
        ssize_t bytes_sent = send(client_sock, buffer, bytes_read, 0);
        if (bytes_sent != bytes_read) {
            print_error("Failed to send file content");
            close(file_fd);
            close(client_sock);
            return;
        }

        total_sent += bytes_sent;
        printf("[PID %d] \rSent: %lu/%lu bytes (%.1f%%)",
               getpid(), total_sent, file_size,
               (float)total_sent / file_size * 100);
        fflush(stdout);
    }

    printf("\n[PID %d] File transfer complete\n", getpid());

    // Close file and client connection
    close(file_fd);
    close(client_sock);
```

```
}


// Function for child process to handle client connections
void child_process(int server_sock, const char *directory) {
    // Set up signal handler for termination
    struct sigaction sa;
    sa.sa_handler = termination_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sigaction(SIGTERM, &sa, NULL);


    printf("[PID %d] Child process started, waiting for connections\n",
getpid());


    while (!terminate) {
        struct sockaddr_in client_addr;
        socklen_t client_len = sizeof(client_addr);


        // Accept a new connection
        int client_sock = accept(server_sock, (struct sockaddr *)&client_addr,
&client_len);
        if (client_sock < 0) {
            if (errno == EINTR) {
                // Interrupted by signal, check if we need to terminate
                if (terminate) {
                    break;
                }
                continue;
            }
            print_error("Accept failed");
            continue;  // Continue accepting connections
        }


        // Get client information
```

```
        char client_ip[INET_ADDRSTRLEN];
        inet_ntop(AF_INET, &client_addr.sin_addr, client_ip, INET_ADDRSTRLEN);
        int client_port = ntohs(client_addr.sin_port);

        printf("[PID %d] Accepted connection from %s:%d\n", getpid(), client_ip,
client_port);

        // Handle the client
        handle_client(client_sock, directory);

        printf("[PID %d] Client connection handled and closed\n", getpid());
    }

    printf("[PID %d] Child process terminating\n", getpid());
    exit(EXIT_SUCCESS);
}

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

    // Parse command line arguments
    const char *server_address = argv[1];
    int server_port = atoi(argv[2]);
    const char *directory = argv[3];
    int num_children = atoi(argv[4]);

    if (num_children <= 0) {
        fprintf(stderr, "Error: num_children must be a positive integer\n");
        exit(EXIT_FAILURE);
    }
```

```
    // Check if directory exists
    DIR *dir = opendir(directory);
    if (!dir) {
        fprintf(stderr, "Error: Directory '%s' does not exist or is not
accessible\n", directory);
        exit(EXIT_FAILURE);
    }
    closedir(dir);

    // Create socket
    int server_sock = socket(AF_INET, SOCK_STREAM, 0);
    if (server_sock < 0) {
        perror("Socket creation failed");
        exit(EXIT_FAILURE);
    }

    // Set socket option to reuse address
    int opt = 1;
    if (setsockopt(server_sock, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt)) <
0) {
        perror("Setsockopt failed");
        exit(EXIT_FAILURE);
    }

    // Bind socket to specified address and port
    struct sockaddr_in server_addr;
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(server_port);

    if (inet_pton(AF_INET, server_address, &server_addr.sin_addr) <= 0) {
        fprintf(stderr, "Invalid address/ Address not supported\n");
        exit(EXIT_FAILURE);
```

```
    }

    if (bind(server_sock, (struct sockaddr *)&server_addr, sizeof(server_addr))
< 0) {
        perror("Bind failed");
        exit(EXIT_FAILURE);
    }

    // Listen for connections
    if (listen(server_sock, 10) < 0) {
        perror("Listen failed");
        exit(EXIT_FAILURE);
    }

    printf("Pre-forked server started at %s:%d, serving files from '%s',
creating %d child processes\n",
           server_address, server_port, directory, num_children);

    // Store child PIDs
    pid_t *child_pids = malloc(num_children * sizeof(pid_t));
    if (!child_pids) {
        perror("Memory allocation failed");
        exit(EXIT_FAILURE);
    }

    // Pre-fork child processes
    for (int i = 0; i < num_children; i++) {
        pid_t pid = fork();

        if (pid < 0) {
            perror("Fork failed");
            exit(EXIT_FAILURE);
        }
```

```
        if (pid == 0) {
            // Child process
            free(child_pids);  // Child doesn't need this array
            child_process(server_sock, directory);
            // Should not reach here
            exit(EXIT_SUCCESS);
        } else {
            // Parent process
            child_pids[i] = pid;
            printf("Child process created (PID: %d, %d/%d)\n", pid, i + 1,
num_children);
        }
    }

    printf("All child processes created. Parent process is now waiting.\n");

    // Set up signal handler for parent to pass signals to children
    struct sigaction sa;
    sa.sa_handler = termination_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sigaction(SIGINT, &sa, NULL);
    sigaction(SIGTERM, &sa, NULL);

    // Wait for termination signal
    while (!terminate) {
        sleep(1);
    }

    printf("Parent process received termination signal. Shutting down...\n");

    // Signal all children to terminate
    for (int i = 0; i < num_children; i++) {
        printf("Sending SIGTERM to child process %d\n", child_pids[i]);
```

```
            kill(child_pids[i], SIGTERM);
        }

        // Wait for all children to exit
        printf("Waiting for all child processes to exit...\n");
        for (int i = 0; i < num_children; i++) {
            int status;
            waitpid(child_pids[i], &status, 0);
            printf("Child process %d exited with status %d\n", child_pids[i],
                   WIFEXITED(status) ? WEXITSTATUS(status) : -1);
        }

        free(child_pids);
        close(server_sock);
        printf("Server shutdown complete\n");

        return 0;
}
```

Опис програми

1. Відмінності між серверами

Кожна реалізація сервера має свої переваги та недоліки, їх ефективність залежить від характеру навантаження:

- **Ітеративний сервер** – обробляє лише одного клієнта за раз. Він простий у реалізації (відсутнє розділення на процеси), займає мінімум ресурсів і виключає накладні витрати на створення процесів. Однак при одночасних запитах інші клієнти чекають завершення поточної передачі, що значно знижує продуктивність у сценаріях з багатьма клієнтами. Такий сервер підходить для одиночних запитів або відлагодження, але погано масштабується під навантаженням.
- **Паралельний сервер з fork** – підтримує багатозадачність, створюючи окремий процес на кожного клієнта. Це дозволяє обслуговувати відразу декілька клієнтів (до заданого ліміту), ефективно використовуючи багатоядерність CPU. Паралельний підхід значно швидший для багатокористувацької роботи, оскільки, наприклад, один клієнт може

отримувати великий файл, тоді як інший у той самий час вже підключений і теж отримує свій файл. Недоліком є витрати на створення і завершення процесів при кожному з'єднанні: fork копіює процес, що потребує часу і пам'яті. При великій кількості коротких запитів ці витрати можуть скласти помітну долю. Введення обмеження на максимальну кількість одночасних процесів запобігає вичерпуванню ресурсів, але якщо ліміт занадто малий, то при піковому навантаженні зайві клієнти чекатимуть, хоча й паралельно обслуговуючи групу клієнтів.

- **Преформований сервер** – поєднує паралельність з відсутністю накладних витрат на кожне нове підключення. Наперед створений пул процесів дозволяє відразу приймати підключення, що особливо ефективно при високому і постійному навантаженні (наприклад, сервер, до якого одночасно звертаються десятки клієнтів). Цей сервер має стабільний рівень використання ресурсів: фіксована кількість процесів постійно зайнята очікуванням або обробкою, і не витрачається час на запуск/завершення процесу для кожного клієнта. У порівнянні з динамічним fork-підходом, префорк дає виграш у продуктивності при великому потоці запитів. Якщо ж клієнтські запити надходять рідко, преформований сервер все одно тримає запущені процеси (деякі можуть простоювати), тобто може використовувати більше пам'яті, ніж паралельний сервер з fork, який у стані спокою має лише один процес. Також преформована модель складніша в реалізації, адже потребує механізмів керування пулом (коректне завершення дітей тощо), але в даній лабораторній роботі це реалізовано через пересилання сигналів SIGTERM усім дочірнім процесам при зупинці сервера.

2. Протокол взаємодії

Клієнт і сервер обмінюються повідомленнями за заздалегідь визначеним протоколом прикладного рівня, який має чітко визначений формат структур даних:

- **Запит файлу (FileRequest):**
  - **protocol_version** (1 байт) – версія протоколу (у даній реалізації використовується версія 1).
  - **message_type** (1 байт) – код повідомлення, що позначає запит на файл (MSG_FILE_REQUEST).
  - **filename_length** (2 байти) – фактична довжина назви файлу.

- **filename** – рядок ASCII, що містить лише назву файлу (без шляхів), обмежений до 255 символів плюс термінатор.

- **Відповідь на запит (FileResponse):**
  - **protocol_version** (1 байт) – версія протоколу, що має збігатися з версією клієнта.
  - **message_type** (1 байт) – код повідомлення, що позначає відповідь (MSG_FILE_RESPONSE).
  - **status** (1 байт) – статус виконання запиту (0 – успіх; інші значення – коди помилок).
  - **file_size** (8 байт) – розмір файлу у байтах (значущий лише при успішному запиті).
- **Відповідь клієнта (ClientResponse):**
  - **message_type** (1 байт) – повідомлення, яке сигналізує про готовність приймати файл (MSG_READY_TO_RECEIVE) або про відмову (MSG_REFUSE_TO_RECEIVE).
- **Передача файлу:**
  Після успішного обміну заголовками і підтвердження готовності клієнта, сервер передає вміст файлу у вигляді послідовних блоків байтів (розмір блоку визначено за замовчуванням, наприклад, 64 КБ). Завершення передачі відзначається закриттям TCP-з'єднання.

3. Обробка помилок

Система передбачає обробку декількох типових помилок:

- **Невідповідність версії протоколу:**
  Якщо версії клієнта і сервера не співпадають, сервер відправляє повідомлення з кодом помилки і закриває з'єднання, а клієнт повідомляє про невідповідність.

- **Некоректне ім'я файлу:**
  При виявленні заборонених символів або недопустимого формату імені (наприклад, містить шлях або є порожнім) сервер негайно повертає код помилки, що сигналізує про недопустиме ім'я файлу, і припиняє обробку запиту.

- **Файл не знайдено або недоступний:**
  Якщо файл не існує або сервер не може отримати до нього доступ, повертається відповідний код помилки, і з'єднання закривається. У цьому випадку клієнт повідомляє про те, що файл не знайдено.

- **Відмова клієнта приймати файл:**
  Якщо розмір файлу перевищує встановлений ліміт, клієнт відразу відправляє повідомлення про відмову приймати файл, і з'єднання завершується, що запобігає марній передачі даних.
- **Помилки мережі та системні помилки:**
  Якщо при мережевих операціях або роботі з файловою системою виникають помилки, сервер обробляє їх для конкретного з'єднання (надсилаючи повідомлення про помилку або закриваючи сокет), але продовжує свою роботу для інших клієнтів. Таким чином, і клієнт, і сервер завжди отримують зрозумілі повідомлення про помилки і коректно завершають сесію.

Приклади використання програми