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 №4

## **Мережеве програмування у середовищі UNIX**

Тема: «TCP клієнт-сервер з мультиплексуванням введення-виведення»

Виконав:

Студент групи ІА-12

Оверчук Дмитро Максимович

Перевірив:

Сімоненко А.В.

Київ 2025

## Завдання на роботу

Розробити однопотоковий сервер, який виконує наступне:

1. Сервер підтримує аргументи командного рядка, визначені в лабораторній роботі No3. Та- кож сервер підтримує аргумент командного рядка, який визначає максимальну кількість клієнтів, з якими сервер може одночасно працювати. Сервер не приймає нові TCP з'єдна- ння після досягнення цього значення.
2. Сервер працює з клієнтами відповідно до користувальницького протоколу, визначеного в лабораторній роботі No3.
3. Сервер дозволяє одночасно працювати з кількома клієнтами за допомогою мультипле- ксування введення-виведення. Сервер послуговується системними викликами select() або poll() для мультиплексування введення-виведення.
4. Кількість даних, які сервер зчитує або відправляє одному клієнту під час виконання введення-виведення з ним, треба обмежити. Ця кількість задається в коді сервера кон-стантою, яка може мати значення 1 байт та більше. Тобто, якщо сервер отримав інформа- цію від ядра про можливість виконати введення або виведення для якогось дескриптора файлу сокета, тоді серверу дозволено відправити або отримати даних розміром не більше вказаної константи. Це обмеження дає змогу майже порівну розподіляти час роботи сервера для кожного клієнта, який потребує комунікації. Також невеликі значення цієї константи дозволяють імітувати проблеми з мережею та частково імітувати різну поведінку клієнтів.

Сервер не має завершувати своє виконання у випадку виникнення несистемної помилки.

Рекомендації для сервера такі самі, які були дані в лабораторній роботі No3.

## Код програми

### client

```
#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 <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;
}
```

server

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <sys/poll.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>
#include <sys/stat.h>
#include <dirent.h>
#include <limits.h>
#include <time.h>
#include "protocol.h"
#include "utils.h"

// Maximum data chunk to read/write in a single operation
#define MAX_IO_CHUNK 512
```

```
// Client connection states
typedef enum {
    STATE_READ_REQUEST,      // Reading file request from client
    STATE_SEND_RESPONSE,     // Sending file response to client
    STATE_WAIT_DECISION,     // Waiting for client's decision to receive file
    STATE_SEND_FILE,         // Sending file content to client
    STATE_DONE               // Client handled, connection can be closed
} ClientState;

// Client context structure
typedef struct {
    int socket;                             // Client socket descriptor
    ClientState state;                      // Current state
    time_t last_activity;                   // Time of last activity

    // Request handling
    FileRequestHeader request;              // Client request
    size_t request_bytes_read;              // How much of the request has been
read

    // Response handling
    FileResponseHeader response;            // Server response
    size_t response_bytes_sent;             // How much of the response has
been sent

    // Client decision handling
    ClientResponseHeader client_response;   // Client's decision response
    size_t decision_bytes_read;             // How much of the decision has
been read

    // File transfer
    int file_fd;                            // File descriptor
    uint64_t file_size;                     // Size of the file
```

```
    uint64_t file_bytes_sent;               // How much of the file has been
sent
    char filepath[PATH_MAX];                // Full path to the file


    char buffer[MAX_IO_CHUNK];              // Buffer for I/O operations
} ClientContext;


// Global variables
char server_directory[PATH_MAX];            // Directory to serve files from
int max_clients = 0;                        // Maximum number of clients
int active_clients = 0;                     // Number of active client
connections


// Function prototypes
void setup_server_socket(int *server_sock, const char *server_address, int
server_port);
void set_nonblocking(int socket_fd);
int accept_new_client(int server_sock, struct pollfd *fds, ClientContext
*contexts, int *nfds);
void handle_client_io(struct pollfd *fd, ClientContext *context);
void cleanup_client(struct pollfd *fds, ClientContext *contexts, int index, int
*nfds);
void process_read_request(struct pollfd *fd, ClientContext *context);
void process_send_response(struct pollfd *fd, ClientContext *context);
void process_wait_decision(struct pollfd *fd, ClientContext *context);
void process_send_file(struct pollfd *fd, ClientContext *context);


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


    // Parse command line arguments
```

```
    const char *server_address = argv[1];
    int server_port = atoi(argv[2]);
    strncpy(server_directory, argv[3], sizeof(server_directory) - 1);
    max_clients = atoi(argv[4]);

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

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

    // Create and setup server socket
    int server_sock;
    setup_server_socket(&server_sock, server_address, server_port);

    // Set server socket to non-blocking mode
    set_nonblocking(server_sock);

    // Allocate memory for poll structures and client contexts
    // +1 for server socket
    struct pollfd *fds = malloc((max_clients + 1) * sizeof(struct pollfd));
    ClientContext *contexts = malloc((max_clients + 1) * sizeof(ClientContext));

    if (!fds || !contexts) {
        perror("Failed to allocate memory");
        exit(EXIT_FAILURE);
```

```
    }

    // Initialize server socket in poll structure
    fds[0].fd = server_sock;
    fds[0].events = POLLIN;
    int nfds = 1;

    printf("Multiplexing server started at %s:%d, serving files from '%s', max clients: %d\n",
           server_address, server_port, server_directory, max_clients);

    // Main server loop
    while (1) {
        // Wait for events on sockets
        int poll_result = poll(fds, nfds, -1);

        if (poll_result < 0) {
            if (errno == EINTR) {
                continue;  // Interrupted system call, try again
            }
            perror("Poll failed");
            break;
        }

        // Check for events on all file descriptors
        for (int i = 0; i < nfds; i++) {
            if (fds[i].revents == 0) {
                continue;  // No events on this descriptor
            }

            // Check for error conditions
            if (fds[i].revents & (POLLERR | POLLHUP | POLLNVAL)) {
                if (i == 0) {
                    // Error on server socket, critical error
```

```
                    fprintf(stderr, "Error on server socket, exiting\n");
                    exit(EXIT_FAILURE);
                } else {
                    // Error on client socket, close connection
                    printf("Socket error (fd=%d), closing connection\n",
fds[i].fd);
                    cleanup_client(fds, contexts, i, &nfds);
                    continue;
                }
            }

            // Handle incoming connection on server socket
            if (i == 0 && (fds[i].revents & POLLIN)) {
                if (active_clients < max_clients) {
                    if (accept_new_client(server_sock, fds, contexts, &nfds)) {
                        active_clients++;
                    }
                }
                continue;
            }

            // Handle client I/O
            if (fds[i].revents & (POLLIN | POLLOUT)) {
                handle_client_io(&fds[i], &contexts[i]);

                // Check if client is done and needs to be closed
                if (contexts[i].state == STATE_DONE) {
                    cleanup_client(fds, contexts, i, &nfds);
                    active_clients--;
                } else {
                    // Update last activity time
                    contexts[i].last_activity = time(NULL);
                }
            }
```

```
        }
    }

    // Cleanup
    close(server_sock);
    free(fds);
    free(contexts);

    return 0;
}

// Set up server socket
void setup_server_socket(int *server_sock, const char *server_address, int
server_port) {
    *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, max_clients) < 0) {
        perror("Listen failed");
        exit(EXIT_FAILURE);
    }
}

// Set socket to non-blocking mode
void set_nonblocking(int socket_fd) {
    int flags = fcntl(socket_fd, F_GETFL, 0);
    if (flags == -1) {
        perror("fcntl F_GETFL");
        exit(EXIT_FAILURE);
    }
    if (fcntl(socket_fd, F_SETFL, flags | O_NONBLOCK) == -1) {
        perror("fcntl F_SETFL O_NONBLOCK");
        exit(EXIT_FAILURE);
    }
}

// Accept new client connection
int accept_new_client(int server_sock, struct pollfd *fds, ClientContext
*contexts, int *nfds) {
```

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

    // Accept connection
    int client_sock = accept(server_sock, (struct sockaddr *)&client_addr,
&client_len);
    if (client_sock < 0) {
        if (errno == EAGAIN || errno == EWOULDBLOCK) {
            return 0;  // No pending connections
        }
        perror("Accept failed");
        return 0;
    }

    // Set client socket to non-blocking mode
    set_nonblocking(client_sock);

    // 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 (fd=%d)\n", client_ip, client_port,
client_sock);

    // Initialize client context
    int index = *nfds;
    fds[index].fd = client_sock;
    fds[index].events = POLLIN;  // Initially, wait for client request

    memset(&contexts[index], 0, sizeof(ClientContext));
    contexts[index].socket = client_sock;
    contexts[index].state = STATE_READ_REQUEST;
    contexts[index].last_activity = time(NULL);
```

```
    contexts[index].file_fd = -1;

    (*nfds)++;
    return 1;
}

// Handle client I/O based on current state
void handle_client_io(struct pollfd *fd, ClientContext *context) {
    switch (context->state) {
        case STATE_READ_REQUEST:
            process_read_request(fd, context);
            break;
        case STATE_SEND_RESPONSE:
            process_send_response(fd, context);
            break;
        case STATE_WAIT_DECISION:
            process_wait_decision(fd, context);
            break;
        case STATE_SEND_FILE:
            process_send_file(fd, context);
            break;
        case STATE_DONE:
            // Nothing to do, will be cleaned up in main loop
            break;
    }
}

// Process reading client request
void process_read_request(struct pollfd *fd, ClientContext *context) {
    // Calculate how many bytes we still need to read
    size_t bytes_to_read = sizeof(FileRequestHeader) - context-
>request_bytes_read;

    // Limit by MAX_IO_CHUNK
```

```
    if (bytes_to_read > MAX_IO_CHUNK)
        bytes_to_read = MAX_IO_CHUNK;

    // Read from socket
    char *buffer_ptr = (char *)&context->request + context->request_bytes_read;
    ssize_t bytes_read = read(context->socket, buffer_ptr, bytes_to_read);

    if (bytes_read < 0) {
        if (errno == EAGAIN || errno == EWOULDBLOCK) {
            return;  // No data available, try again later
        }
        perror("Read error");
        context->state = STATE_DONE;
        return;
    }

    if (bytes_read == 0) {
        printf("Client closed connection during request read\n");
        context->state = STATE_DONE;
        return;
    }

    context->request_bytes_read += bytes_read;

    // If we've read the entire request
    if (context->request_bytes_read == sizeof(FileRequestHeader)) {
        // Prepare response
        memset(&context->response, 0, sizeof(FileResponseHeader));
        context->response.protocol_version = PROTOCOL_VERSION;
        context->response.message_type = MSG_FILE_RESPONSE;

        // Check protocol version
        if (context->request.protocol_version != PROTOCOL_VERSION) {
```

```
            printf("Error: Protocol version mismatch (client: %d, server:
%d)\n",
                   context->request.protocol_version, PROTOCOL_VERSION);
            context->response.status = ERR_PROTOCOL_MISMATCH;
            context->state = STATE_SEND_RESPONSE;
            fd->events = POLLOUT;
            return;
        }

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

        printf("Received file request from fd=%d for: %s\n", context->socket,
context->request.filename);

        // Validate filename
        if (!validate_filename(context->request.filename)) {
            printf("Error: Invalid filename: %s\n", context->request.filename);
            context->response.status = ERR_INVALID_FILENAME;
            context->state = STATE_SEND_RESPONSE;
            fd->events = POLLOUT;
            return;
        }

        // Build full path
        snprintf(context->filepath, sizeof(context->filepath), "%s/%s",
                server_directory, context->request.filename);

        // Check if the file exists and is a regular file
        struct stat file_stat;
        if (stat(context->filepath, &file_stat) != 0) {
            printf("Error: File not found: %s\n", context->filepath);
            context->response.status = ERR_FILE_NOT_FOUND;
            context->state = STATE_SEND_RESPONSE;
```

```
            fd->events = POLLOUT;
            return;
        }

        if (!S_ISREG(file_stat.st_mode)) {
            printf("Error: Not a regular file: %s\n", context->filepath);
            context->response.status = ERR_FILE_NOT_FOUND;
            context->state = STATE_SEND_RESPONSE;
            fd->events = POLLOUT;
            return;
        }

        // Get file size
        context->file_size = file_stat.st_size;

        // Prepare success response
        context->response.status = 0;  // Success
        context->response.file_size = context->file_size;

        printf("Sending file response to fd=%d: status=%d, file_size=%lu\n",
               context->socket, context->response.status, context->response.file_size);

        // Change state and update poll events
        context->state = STATE_SEND_RESPONSE;
        fd->events = POLLOUT;
    }
}

// Process sending response to client
void process_send_response(struct pollfd *fd, ClientContext *context) {
    // Calculate how many bytes we still need to send
    size_t bytes_to_send = sizeof(FileResponseHeader) - context->response_bytes_sent;
```

```
    // Limit by MAX_IO_CHUNK
    if (bytes_to_send > MAX_IO_CHUNK)
        bytes_to_send = MAX_IO_CHUNK;

    // Write to socket
    char *buffer_ptr = (char *)&context->response + context->response_bytes_sent;
    ssize_t bytes_sent = write(context->socket, buffer_ptr, bytes_to_send);

    if (bytes_sent < 0) {
        if (errno == EAGAIN || errno == EWOULDBLOCK) {
            return;  // Buffer full, try again later
        }
        perror("Write error");
        context->state = STATE_DONE;
        return;
    }

    context->response_bytes_sent += bytes_sent;

    // If we've sent the entire response
    if (context->response_bytes_sent == sizeof(FileResponseHeader)) {
        // If response indicated an error, we're done
        if (context->response.status != 0) {
            context->state = STATE_DONE;
            return;
        }

        // Wait for client's decision
        context->state = STATE_WAIT_DECISION;
        fd->events = POLLIN;
    }
}
```

```
// Process waiting for client decision
void process_wait_decision(struct pollfd *fd, ClientContext *context) {
    // Calculate how many bytes we still need to read
    size_t bytes_to_read = sizeof(ClientResponseHeader) - context->decision_bytes_read;

    // Limit by MAX_IO_CHUNK
    if (bytes_to_read > MAX_IO_CHUNK)
        bytes_to_read = MAX_IO_CHUNK;

    // Read from socket
    char *buffer_ptr = (char *)&context->client_response + context->decision_bytes_read;
    ssize_t bytes_read = read(context->socket, buffer_ptr, bytes_to_read);

    if (bytes_read < 0) {
        if (errno == EAGAIN || errno == EWOULDBLOCK) {
            return;  // No data available, try again later
        }
        perror("Read error");
        context->state = STATE_DONE;
        return;
    }

    if (bytes_read == 0) {
        printf("Client closed connection during decision read\n");
        context->state = STATE_DONE;
        return;
    }

    context->decision_bytes_read += bytes_read;

    // If we've read the entire decision
```

```
    if (context->decision_bytes_read == sizeof(ClientResponseHeader)) {
        if (context->client_response.message_type == MSG_REFUSE_TO_RECEIVE) {
            printf("Client fd=%d refused to receive the file (too large)\n",
context->socket);
            context->state = STATE_DONE;
            return;
        }

        if (context->client_response.message_type != MSG_READY_TO_RECEIVE) {
            printf("Error: Unexpected client response message type: %d\n",
                   context->client_response.message_type);
            context->state = STATE_DONE;
            return;
        }

        printf("Client fd=%d is ready to receive the file\n", context->socket);

        // Open the file
        context->file_fd = open(context->filepath, O_RDONLY);
        if (context->file_fd < 0) {
            perror("Failed to open file");
            context->state = STATE_DONE;
            return;
        }

        // Change state to send file and update poll events
        context->state = STATE_SEND_FILE;
        context->file_bytes_sent = 0;
        fd->events = POLLOUT;
    }
}

// Process sending file content
void process_send_file(struct pollfd *fd, ClientContext *context) {
```

```
    // If we've sent the entire file
    if (context->file_bytes_sent >= context->file_size) {
        printf("\nFile transfer complete for client fd=%d\n", context->socket);
        // Close file descriptor and mark as done
        if (context->file_fd >= 0) {
            close(context->file_fd);
            context->file_fd = -1;
        }
        context->state = STATE_DONE;
        return;
    }

    // Read from file into buffer (limited by MAX_IO_CHUNK)
    size_t bytes_to_read = MAX_IO_CHUNK;
    if (context->file_size - context->file_bytes_sent < bytes_to_read) {
        bytes_to_read = context->file_size - context->file_bytes_sent;
    }

    ssize_t bytes_read = read(context->file_fd, context->buffer, bytes_to_read);

    if (bytes_read <= 0) {
        if (bytes_read < 0) {
            perror("File read error");
        }
        // End of file or error
        context->state = STATE_DONE;
        close(context->file_fd);
        context->file_fd = -1;
        return;
    }

    // Send buffer to socket
    ssize_t bytes_sent = write(context->socket, context->buffer, bytes_read);
```

```
if (bytes_sent < 0) {
    if (errno == EAGAIN || errno == EWOULDBLOCK) {
        // Rewind file pointer as we couldn't send data
        if (lseek(context->file_fd, -bytes_read, SEEK_CUR) < 0) {
            perror("lseek error");
            context->state = STATE_DONE;
            close(context->file_fd);
            context->file_fd = -1;
        }
        return;  // Buffer full, try again later
    }
    perror("Write error");
    context->state = STATE_DONE;
    close(context->file_fd);
    context->file_fd = -1;
    return;
}

if (bytes_sent < bytes_read) {
    // We couldn't send all bytes, rewind file pointer
    off_t rewind_offset = bytes_read - bytes_sent;
    if (lseek(context->file_fd, -rewind_offset, SEEK_CUR) < 0) {
        perror("lseek error");
        context->state = STATE_DONE;
        close(context->file_fd);
        context->file_fd = -1;
        return;
    }
}

// Update bytes sent
context->file_bytes_sent += bytes_sent;
```

```
    // Print progress (only for significant changes to avoid console flooding)
    static int last_percent[FD_SETSIZE] = {0};
    int current_percent = (int)((double)context->file_bytes_sent / context->file_size * 100);

    if (current_percent > last_percent[context->socket] ||
        context->file_bytes_sent >= context->file_size) {
        printf("\rClient fd=%d: Sent %lu/%lu bytes (%d%%)",
               context->socket, context->file_bytes_sent, context->file_size, current_percent);
        fflush(stdout);
        last_percent[context->socket] = current_percent;
    }
}

// Clean up client resources and remove from poll array
void cleanup_client(struct pollfd *fds, ClientContext *contexts, int index, int *nfds) {
    // Close file descriptor if open
    if (contexts[index].file_fd >= 0) {
        close(contexts[index].file_fd);
    }

    // Close socket
    close(fds[index].fd);
    printf("Closed connection for client fd=%d\n", fds[index].fd);

    // Remove from poll array by shifting remaining descriptors
    (*nfds)--;
    for (int i = index; i < *nfds; i++) {
        fds[i] = fds[i + 1];
        contexts[i] = contexts[i + 1];
    }
}
```

## protocol

```
#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

```
#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
```

## Опис програми

Програма представляє собою однопотоковий мультиплексований сервер для передачі файлів. Розглянемо детальніше особливості його реалізації:

## Архітектурні рішення

Сервер використовує **мультиплексування вводу-виводу** замість створення окремих потоків/процесів для кожного клієнта. Це дозволяє одному потоку виконання обробляти багато з'єднань одночасно. У реалізації використано системний виклик poll(), який відстежує стан усіх сокетів.

## Основні компоненти сервера:

1. **Структура ClientContext** - зберігає стан взаємодії з кожним клієнтом:

- поточний стан обміну (читання запиту, відправка відповіді тощо)
- буфер для введення-виведення
- стан передачі файлу і метадані
- інформація про активність клієнта

2. **Кінцевий автомат станів клієнта** реалізовано через enum ClientState:
   - STATE_READ_REQUEST - читання запиту на файл
   - STATE_SEND_RESPONSE - відправка відповіді з метаінформацією
   - STATE_WAIT_DECISION - очікування рішення клієнта
   - STATE_SEND_FILE - передача файлу
   - STATE_DONE - завершення обробки

## Особливості реалізації

### *1. Обмеження на розмір операцій вводу-виводу*

Ключовою особливістю є константа MAX_IO_CHUNK (512 байт), яка обмежує розмір даних для операцій читання/запису за одну ітерацію. Це забезпечує:

- **Справедливий розподіл ресурсів** - сервер не "зависає" на обробці великого файлу для одного клієнта
- **Імітацію мережевих умов** - дозволяє тестувати роботу з повільними з'єднаннями
- **Рівномірне обслуговування клієнтів** - кожен клієнт отримує обмежену порцію уваги сервера

### *2. Неблокуючий ввід-вивід*

Усі сокети налаштовуються в неблокуючий режим через функцію set_nonblocking(). Це дозволяє функціям читання/запису повертати керування негайно, навіть якщо дані недоступні.

### *3. Обмеження кількості клієнтів*

Сервер підтримує параметр max_clients, який обмежує кількість одночасних підключень. При досягненні ліміту нові з'єднання не приймаються.

### *4. Системний виклик poll()*

Центральна частина мультиплексування - використання poll() для моніторингу кількох файлових дескрипторів. Сервер відстежує готовність дескрипторів до операцій читання або запису, а також помилки.

### *5. Поетапна обробка протоколу*

Обробка протоколу розбита на окремі функції згідно зі станами клієнта: - process_read_request() - обробка вхідних запитів на файл - process_send_response() - відправка метаінформації про файл - process_wait_decision() - очікування рішення клієнта - process_send_file() - відправка вмісту файлу частинами

## Алгоритм роботи сервера

1. **Ініціалізація**:
   - Розбір параметрів командного рядка
   - Створення сокету та прив'язка до адреси і порту
   - Встановлення сокету в режим прослуховування
   - Налаштування неблокуючого режиму
2. **Основний цикл**:
   - Очікування подій на всіх дескрипторах через poll()
   - Обробка нових підключень (якщо не досягнуто ліміту)
   - Обробка готових для вводу-виводу дескрипторів
   - Обробка помилок і закриття з'єднань
   - Оновлення структур даних і звільнення ресурсів для завершених з'єднань
3. **Обробка клієнтів**:
   - Прийом запиту на файл частинами (до MAX_IO_CHUNK за раз)
   - Перевірка існування файлу і підготовка відповіді
   - Відправка відповіді з метаінформацією (частинами)
   - Очікування підтвердження клієнта
   - Передача вмісту файлу невеликими блоками

Як видно з прикладів використання, сервер здатен одночасно обслуговувати декілька клієнтів, обмежуючи їх максимальну кількість і розподіляючи ресурси процесора та мережі між ними. Це дозволяє ефективно передавати файли навіть при роботі з декількома клієнтами одночасно.

## Приклади використання програми

### Запуск сервера

```
dymytryke@dymytryke:/mnt/c/Users/dymyt/OneDrive/Документы/save/Учеба/4_ку
pc/Netw.UNIX/4/bin$ ./multiplexing_server 127.0.0.1 12345 ../test_files/ 3
Multiplexing server started at 127.0.0.1:12345, serving files from '../test_files/', max
clients: 3
```

## Робота з одним клієнтом

### Клієнт

```
dymytryke@dymytryke:/mnt/c/Users/dymyt/OneDrive/Документы/save/Учеба/4_курс/Netw.UNIX/4/bin$ ./client 127.0.0.1 12345 baseball-cap-collection.zip 1200000000
Connecting to server at 127.0.0.1:12345
Connected to server
Sending file request for: baseball-cap-collection.zip
Received response: protocol version 1, message type 2, status 0, file size 13142141
Sending ready to receive message
Receiving file content (13142141 bytes)...
Received: 13142141/13142141 bytes (100.0%)
File transfer complete
```

### Сервер

```
Accepted connection from 127.0.0.1:47430 (fd=4)
Received file request from fd=4 for: baseball-cap-collection.zip
Sending file response to fd=4: status=0, file_size=13142141
Client fd=4 is ready to receive the file
Client fd=4: Sent 13142141/13142141 bytes (100%)
File transfer complete for client fd=4
Closed connection for client fd=4
```

## Робота з декількома клієнтами

### Скрипт для одночасного запуску 5-ти клієнтів

```
#!/bin/bash

./client 127.0.0.1 12345 AnyDesk.exe 100000000000000 &
./client 127.0.0.1 12345 FigmaSetup.exe 100000000000000 &
./client 127.0.0.1 12345 baseball-cap-collection.zip 100000000000000 &
./client 127.0.0.1 12345 junos-vsrx-12.1X47-D10.4-domestic.ova 100000000000000 &
./client 127.0.0.1 12345 n3467.pdf 1000 &
```

### Клієнт

```
dymytryke@dymytryke:/mnt/c/Users/dymyt/OneDrive/Документы/save/Учеба/4_курс/Netw.UNIX/4/bin$ Received response: protocol version 1, message type 2, status 0, file size 13142141
Sending ready to receive message
```

Received: 512/5634880 bytes (0.0%)Receiving file content (13142141 bytes)...
Received response: protocol version 1, message type 2, status 0, file size 118362136
Sending ready to receive message
Received: 8704/5634880 bytes (0.2%)Receiving file content (118362136 bytes)...
Received: 5634880/5634880 bytes (100.0%)
File transfer complete
Received: 5635584/13142141 bytes (42.9%)Received response: protocol version 1, message type 2, status 0, file size 238397440
Sending ready to receive message
Received: 5636096/118362136 bytes (4.8%)Receiving file content (238397440 bytes)...
Received: 13142141/13142141 bytes (100.0%)
File transfer complete
Received: 13143040/118362136 bytes (11.1%)Received response: protocol version 1, message type 2, status 0, file size 4309251
File size (4309251 bytes) exceeds maximum allowed size (1000 bytes)
Received: 118362136/118362136 bytes (100.0%)
File transfer complete
Received: 238397440/238397440 bytes (100.0%)
File transfer complete

**Сервер**

dymytryke@dymytryke:/mnt/c/Users/dymyt/OneDrive/Документы/save/Учеба/4_курс/Netw.UNIX/4/bin$ stdbuf -oL ./multiplexing_server 127.0.0.1 12345 ../test_files/ 3 | ts
Mar 25 08:49:48 Multiplexing server started at 127.0.0.1:12345, serving files from '../test_files/', max clients: 3
Mar 25 08:49:52 Accepted connection from 127.0.0.1:43086 (fd=4)
Mar 25 08:49:52 Accepted connection from 127.0.0.1:43082 (fd=5)
Mar 25 08:49:52 Received file request from fd=4 for: AnyDesk.exe
Mar 25 08:49:52 Sending file response to fd=4: status=0, file_size=5634880
Mar 25 08:49:52 Accepted connection from 127.0.0.1:43084 (fd=6)
Mar 25 08:49:52 Received file request from fd=5 for: baseball-cap-collection.zip
Mar 25 08:49:52 Sending file response to fd=5: status=0, file_size=13142141
Mar 25 08:49:52 Client fd=4 is ready to receive the file
Mar 25 08:49:52 Received file request from fd=6 for: FigmaSetup.exe
Mar 25 08:49:52 Sending file response to fd=6: status=0, file_size=118362136
Mar 25 08:49:52 Client fd=5 is ready to receive the file
Mar 25 08:49:52 Client fd=6 is ready to receive the file
Client fd=4: Sent 5634880/5634880 bytes (100%)
Mar 25 08:49:56 File transfer complete for client fd=4

Mar 25 08:49:56 Closed connection for client fd=4
Mar 25 08:49:56 Accepted connection from 127.0.0.1:43104 (fd=4)
Mar 25 08:49:56 Received file request from fd=4 for: junos-vsrx-12.1X47-D10.4-domestic.ova
Mar 25 08:49:56 Sending file response to fd=4: status=0, file_size=238397440
Mar 25 08:49:56 Client fd=4 is ready to receive the file
Client fd=5: Sent 13142141/13142141 bytes (100%)
Mar 25 08:50:03 File transfer complete for client fd=5
Mar 25 08:50:03 Closed connection for client fd=5
Mar 25 08:50:03 Accepted connection from 127.0.0.1:43112 (fd=5)
Mar 25 08:50:03 Received file request from fd=5 for: n3467.pdf
Mar 25 08:50:03 Sending file response to fd=5: status=0, file_size=4309251
Mar 25 08:50:03 Client fd=5 refused to receive the file (too large)
Mar 25 08:50:03 Closed connection for client fd=5
Client fd=6: Sent 118362136/118362136 bytes (100%)
Mar 25 08:50:58 File transfer complete for client fd=6
Mar 25 08:50:58 Closed connection for client fd=6
Client fd=4: Sent 238397440/238397440 bytes (100%)
Mar 25 08:51:25 File transfer complete for client fd=4
Mar 25 08:51:25 Closed connection for client fd=4

**Demo:**

Video