

**МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ)**

Институт №8 «Компьютерные науки и прикладная математика»
Кафедра 806 «Вычислительная математика и программирование»

**Курсовая работа
по курсу «Операционные системы»**

Выполнила: Власова Е.Р.
Группа: М8О-208БВ-24
Преподаватель: Миронов Е.С.

Москва, 2025

Условие:

Необходимо написать консоль-серверную игру. Необходимо написать 2 программы: сервер и клиент. Сначала запускается сервер, а далее клиенты соединяются с сервером. Сервер координирует клиентов между собой. При запуске клиента игрок может выбрать одно из следующих действий (возможно больше, если предусмотрено вариантом): создать игру, введя ее имя; присоединиться к одной из существующих игр по имени игры.

Цель работы:

Целью является приобретение практических навыков в использовании знаний, полученных в течении курса и проведение исследования в выбранной предметной области.

Вариант:

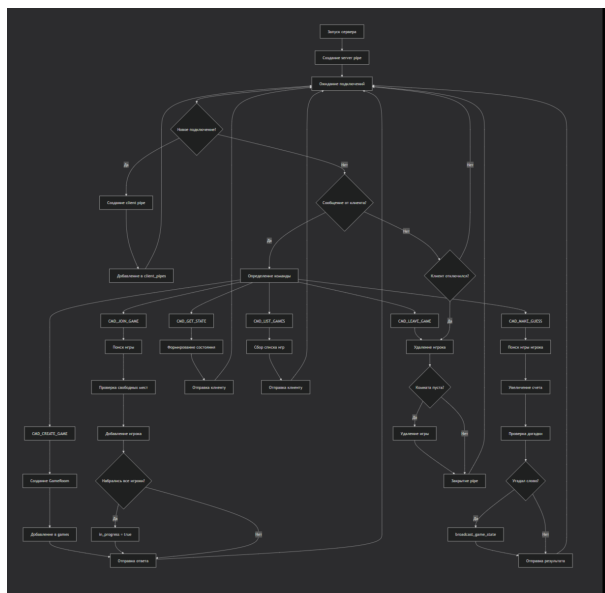
7

Задание:

«Быки и коровы» (угадывать необходимо слова). Общение между сервером и клиентом необходимо организовать при помощи pipe'ов. При создании каждой игры необходимо указывать количество игроков, которые будут участвовать. То есть угадывать могут несколько игроков. Если кто-то из игроков вышел из игры, то игра должна быть продолжена.

Метод решения:

Разработана клиент-серверная программа с использованием pipe'ов для межпроцессного взаимодействия. Сервер реализован как управляющий процесс, обрабатывающий несколько клиентов. Каждый клиент выполняется в отдельном процессе и обменивается сообщениями с сервером через именованные pipe'ы. Игровая логика инкапсулирована в классе GameLogic, обеспечивающем генерацию случайных слов, обработку догадок и расчет "быков и коров". Состояние игровых комнат хранится в std::map, обеспечивающем быстрый поиск по имени игры. Для синхронизации данных между клиентами реализован механизм рассылки обновлений состояния игры. Программа поддерживает создание игр с указанием количества игроков, присоединение к существующим играм,



угадывание слова несколькими участниками и продолжение игры при выходе отдельных игроков.

Описание программы:

Программа состоит из следующих файлов: `common.h`, `game-logic.cpp`, `pipe-manager.h`, `pipe-manager.cpp`, `server.h`, `server.cpp`, `client.h`, `client.cpp`, `main-server.cpp`, `main-client.cpp`.

Я использую следующие системные вызовы: `mkfifo()`, `open()`, `close()`, `read()`, `write()`, `unlink()`, `fork()`, `waitpid()`, `signal()`, `getpid()`, `time()`, `srand()`, `rand()`.

Выводы:

В этой лабораторной работе я создала многопользовательскую игру «Быки и коровы», где сервер и клиенты общаются через `pipe`. Основная идея была в том, чтобы разобраться, как процессы в Linux могут обмениваться данными. Сервер создаёт главный пайп, а каждый клиент при запуске делает свой личный пайп. Когда клиент хочет что-то сделать — создать игру или присоединиться — он пишет сообщение в общий пайп сервера. Сервер получает его, создаёт «клон»-процесс для обработки этого запроса и отправляет ответ в личный пайп клиента.

С помощью команды `'strace'` я наблюдала за тем, как система работает изнутри: видела, как создаются пайпы, как процессы размножаются, как идут запись и чтение.

Такая система теоритически может стать основой для других приложений, где требуется координация между процессами: чаты, уведомления, совместная работа с данными. Теперь я лучше понимаю, как устроены многие серверные приложения и как они управляют множеством пользователей одновременно.

Исходная программа:

```
common.h
#pragma once

#include <string>
#include <vector>

const std::string SERVER_PIPE = "/tmp/bulls_cows_server";
const int MAX_PLAYERS = 15;

struct GameRoom {
    std::string name;
    std::string secret_word;
    int max_players;
    std::vector<std::string> players;
    std::vector<int> scores;
    bool in_progress;
};

struct GameResult {
    int bulls;
    int cows;
    bool guessed;
};

enum Command {
    CMD_CREATE_GAME,
    CMD_JOIN_GAME,
    CMD_MAKE_GUESS,
    CMD_GET_STATE,
    CMD_LIST_GAMES,
    CMD_LEAVE_GAME,
    CMD_RESPONSE,
    CMD_ERROR,
    CMD_INCOMPLETE
};

struct Message {
    Command cmd;
    std::string data;
    std::string player_name;
```

```

        std::string game_name;
};

game-logic.h
#pragma once

#include <string>
#include <vector>

#include "common.h"

const int WORD_LENGTH = 5;

class GameLogic {
private:
    static std::vector<std::string> words;

public:
    static void init();
    static std::string get_random_word();
    static bool is_valid_word(const std::string& word);
    static GameResult check_guess(const std
        ::string& secret, const std::string& guess);
};

```

```

game-lodic.cpp
#include "game_logic.h"
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <algorithm>
#include <cctype>

std::vector<std::string> GameLogic::words;

void GameLogic::init() {
    words = {
        "apple", "grape", "peach", "lemon", "melon",
        "bread", "table", "chair", "house", "tiger",
        "eagle", "smile", "water", "earth", "light",

```

```

        "music", "paper", "stone", "cloud", "river"
    };

    std::srand(std::time(nullptr));
}

std::string GameLogic::get_random_word() {
    if (words.empty()) {
        init();
    }

    int index = std::rand() % words.size();
    return words[index];
}

bool GameLogic::is_valid_word(const std::string& word) {
    if (word.length() != WORD_LENGTH) return false;

    for (char c : word) {
        if (!std::isalpha(c)) return false;
    }

    std::string lower = word;
    for (char& c : lower) c = std::tolower(c);

    for (const auto& w : words) {
        if (w == lower) return true;
    }

    return false;
}

GameResult GameLogic::check_guess(const
std::string& secret, const std::string& guess) {
    GameResult result{0, 0, false};

    if (secret.length() != guess.length()) {
        return result;
    }

    bool secret_used[WORD_LENGTH] = {false};

```

```

    bool guess_used[WORD_LENGTH] = {false};

    for (int i = 0; i < WORD_LENGTH; ++i) {
        if (secret[i] == guess[i]) {
            result.bulls++;
            secret_used[i] = true;
            guess_used[i] = true;
        }
    }

    for (int i = 0; i < WORD_LENGTH; ++i) {
        if (!guess_used[i]) {
            for (int j = 0; j < WORD_LENGTH; ++j) {
                if (!secret_used[j] && secret[j] == guess[i]) {
                    result.cows++;
                    secret_used[j] = true;
                    break;
                }
            }
        }
    }

    result.guessed = (result.bulls == WORD_LENGTH);
    return result;
}

```

pipe-manager.h

```
#pragma once
```

```
#include "common.h"
```

```
#include <string>
```

```
class PipeManager {
```

```
private:
```

```
    static std::string leftover_buffer;
```

```
public:
```

```
    static bool create_pipe(const std::string& name);
```

```
    static int open_pipe(const std::string& name, int mode);
```

```
    static bool send_message(int pipe_fd, const Message& msg);
```

```
    static Message receive_message(int pipe_fd);
```

```
    static void close_pipe(int fd);
```

```

        static std::string get_client_pipe(const std::string& client_name);
        std::vector<Message> receive_all_messages(int pipe_fd);
};

```

pipe-manager.cpp

```

#include "pipe_manager.h"
#include "common.h"
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <cstring>
#include <sstream>
#include <iostream>

```

```

std::string PipeManager::leftover_buffer = "";

```

```

bool PipeManager::create_pipe(const std::string& name) {
    unlink(name.c_str());
    if (mkfifo(name.c_str(), 0666) == -1) {
        std::cerr << "Error creating pipe '" << name << "': "
                  << strerror(errno) << std::endl;
        return false;
    }
}

```

```

    return true;
}

```

```

int PipeManager::open_pipe(const std::string& name, int mode) {
    int fd = open(name.c_str(), mode);
    return fd;
}

```

```

std::string serialize_message(const Message& msg) {
    std::ostringstream oss;
    oss << static_cast<int>(msg.cmd) << "|"
        << msg.player_name << "|"
        << msg.game_name << "|"
        << msg.data;
    return oss.str();
}

```

```

Message deserialize_message(const std::string& str) {
    Message msg;
    size_t pos1 = str.find('|');
    size_t pos2 = str.find('|', pos1 + 1);
    size_t pos3 = str.find('|', pos2 + 1);

    msg.cmd = static_cast<Command>(std::stoi(str.substr(0, pos1)));
    msg.player_name = str.substr(pos1 + 1, pos2 - pos1 - 1);
    msg.game_name = str.substr(pos2 + 1, pos3 - pos2 - 1);
    msg.data = str.substr(pos3 + 1);

    return msg;
}

bool PipeManager::send_message(int pipe_fd, const Message& msg) {
    std::string data = serialize_message(msg);
    data += "\n";

    ssize_t written = write(pipe_fd, data.c_str(), data.length());
    return written == static_cast<ssize_t>(data.length());
}

Message PipeManager::receive_message(int read_fd) {
    char buffer[1024];

    size_t size = 0;
    if (read(read_fd, &size, sizeof(size)) != sizeof(size)) {
        return {CMD_ERROR, "", "", ""};
    }

    if (size == 0 || size > 1024) {
        return {CMD_ERROR, "", "", ""};
    }

    ssize_t bytes_read = read(read_fd, buffer, size);
    if (bytes_read != static_cast<ssize_t>(size)) {
        return {CMD_ERROR, "", "", ""};
    }

    buffer[bytes_read] = '\0';
    return deserialize_message(std::string(buffer, bytes_read));
}

```

```
}
```

```
std::vector<Message> PipeManager::receive_all_messages(int pipe_fd) {  
    std::vector<Message> messages;  
    while (true) {  
        Message msg = receive_message(pipe_fd);  
        if (msg.cmd == CMD_INCOMPLETE) {  
            break;  
        } else if (msg.cmd == CMD_ERROR) {  
            break;  
        } else {  
            messages.push_back(msg);  
        }  
    }  
    return messages;  
}
```

```
std::string PipeManager::get_client_pipe(const std::string& client_name) {  
    return "/tmp/client_" + client_name;  
}
```

```
void PipeManager::close_pipe(int fd) {  
    if (fd != -1) {  
        close(fd);  
    }  
}
```

```
server.h  
#pragma once
```

```
#include <vector>  
#include <map>
```

```
#include "common.h"
```

```
class Server {  
private:  
    std::map<std::string, GameRoom> games;  
    std::map<std::string, std::string> player_to_game;
```

```

    void handle_client(int client_fd, const std::string& client_pipe);
    void process_create_game(const
    Message& msg, int client_fd, const std::string& client_name);
    void process_join_game(const
    Message& msg, int client_fd, const std::string& client_name);
    void process_make_guess(const Message& msg,
    int client_fd, const std::string& client_name);
    void process_get_state(const
    Message& msg, int client_fd, const std::string& client_name);
    void process_list_games(int client_fd);
    void process_leave_game(const
    std::string& client_name);

    void broadcast_game_state(const std::string& game_name);
    void remove_player(const std::string& player_name);

public:
    void run();
};

```

server.cpp

```
#include "server.h"
```

```
#include <iostream>
```

```
#include <cstring>
```

```
#include <sstream>
```

```
#include <algorithm>
```

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

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

```
#include <fcntl.h>
```

```
#include "common.h"
```

```
#include "pipe_manager.h"
```

```
#include "game_logic.h"
```

```
void Server::run() {
```

```
    std::cout << "Server starting\n";
```

```
    if (!PipeManager::create_pipe(SERVER_PIPE)) {
```

```
        std::cerr << "Cannot create server pipe\n";
```

```

        return;
    }

    signal(SIGCHLD, SIG_IGN);

    std::cout << "Server ready. Waiting for connections\n";

    while (true) {
        int server_fd = PipeManager::open_pipe(SERVER_PIPE, O_RDONLY);
        if (server_fd == -1) {
            std::cerr << "Cannot open server pipe\n";
            break;
        }

        Message msg = PipeManager::receive_message(server_fd);
        PipeManager::close_pipe(server_fd);

        if (msg.cmd == CMD_ERROR) {
            continue;
        }

        pid_t pid = fork();

        if (pid == 0) {
            std::string client_pipe = PipeManager::get_client_pipe(msg.player_name);
            int client_fd = PipeManager::open_pipe(client_pipe, O_WRONLY);

            if (client_fd != -1) {
                handle_client(client_fd, client_pipe);
                PipeManager::close_pipe(client_fd);
            }

            exit(0);
        }
    }
}

void Server::handle_client(int client_fd, const std::string& client_pipe) {
    int client_read_fd = PipeManager::open_pipe(client_pipe, O_RDONLY );

    if (client_read_fd == -1) {

```

```

        return;
    }

    bool connected = true;
    std::string current_player_name;
    while (connected) {
        Message msg = PipeManager::receive_message(client_read_fd);

        if (msg.cmd == CMD_ERROR) {
            usleep(100000);
            continue;
        }

        current_player_name = msg.player_name;

        switch (msg.cmd) {
            case CMD_CREATE_GAME:
                process_create_game(msg, client_fd, msg.player_name);
                break;
            case CMD_JOIN_GAME:
                process_join_game(msg, client_fd, msg.player_name);
                break;
            case CMD_MAKE_GUESS:
                process_make_guess(msg, client_fd, msg.player_name);
                break;
            case CMD_GET_STATE:
                process_get_state(msg, client_fd, msg.player_name);
                break;
            case CMD_LIST_GAMES:
                process_list_games(client_fd);
                break;
            case CMD_LEAVE_GAME:
                process_leave_game(msg.player_name);
                connected = false;
                break;
            default:
                break;
        }
    }

    PipeManager::close_pipe(client_read_fd);
    if (!current_player_name.empty() &&

```

```

        player_to_game.find(current_player_name) != player_to_game.end()) {
            remove_player(current_player_name);
        }
    }

void Server::process_create_game(const Message
& msg, int client_fd, const std
::string& client_name) {
    if (games.find(msg.game_name) != games.end()) {
        PipeManager::send_message(client_fd, Mess
age{CMD_ERROR, "Game already exists", "", ""});
        return;
    }

    std::istringstream iss(msg.data);
    int max_players;
    iss >> max_players;

    if (max_players < 1 || max_players > MAX_PLAYERS) {
        PipeManager::send_message(client_fd, Message{CMD_ERROR, "Invalid number
of players", "", ""});
        return;
    }

    GameRoom room;
    room.name = msg.game_name;
    room.secret_word = GameLogic::get_random_word();
    room.max_players = max_players;
    room.players.push_back(client_name);
    room.scores.push_back(0);
    room.in_progress = false;

    games[msg.game_name] = room;
    player_to_game[client_name] = msg.game_name;

    std::string response = "Game created. Secret word: " + room.secret_word;
    PipeManager::send_message(client_fd, Message{CMD_RESPONSE, response, "", ""});
}

void Server::process_join_game(const Message& msg,

```

```

int client_fd, const std::string& client_name) {
    auto it = games.find(msg.game_name);
    if (it == games.end()) {
        PipeManager::send_message(client_fd, Message{CMD_ERROR,
            "Game not found", "", ""});
        return;
    }

    GameRoom& room = it->second;

    if (room.players.size() >= static_cast<size_t>(room.max_players)) {
        PipeManager::send_message(client_fd, Message{CMD_ERROR, "Game is full", "", ""});
        return;
    }

    if (std::find(room.players.begin(), room.players.end(), client_name) != room.players.end()) {
        PipeManager::send_message(client_fd, Message{CMD_ERROR, "Already in game", "", ""});
        return;
    }

    room.players.push_back(client_name);
    room.scores.push_back(0);
    player_to_game[client_name] = msg.game_name;

    if (room.players.size() == static_cast<size_t>(room.max_players)) {
        room.in_progress = true;
    }

    PipeManager::send_message(client_fd, Message{CMD_RESPONSE, "Joined game: " + msg.game_name, "", ""});
    broadcast_game_state(msg.game_name);
}

void Server::process_make_guess(const Message& msg, int client_fd, const std::string& client_name) {
    if (player_to_game.find(client_name) == player_to_game.end()) {

```

```

        PipeManager::send_message(client_fd, Message{CMD_ERROR, "Not in any game", "", ""});
        return;
    }

    std::string game_name = player_to_game[client_name];
    auto it = games.find(game_name);
    if (it == games.end()) {
        PipeManager::send_message(client_fd, Message{CMD_ERROR, "Game not found", "", ""});
        return;
    }

    GameRoom& room = it->second;

    if (!room.in_progress) {
        PipeManager::send_message(
            client_fd, Message{CMD_ERROR, "Game not started", "", ""});
        return;
    }

    if (!GameLogic::is_valid_word(msg.data)) {
        PipeManager::send_message(
            client_fd, Message{CMD_ERROR, "Invalid word", "", ""});
        return;
    }

    GameResult result = GameLogic::check_guess(room.secret_word, msg.data);

    auto player_it = std::find(room.players.begin(),
        room.players.end(), client_name);
    if (player_it != room.players.end()) {
        int index = std::distance(room.players.begin(), player_it);
        room.scores[index]++;
    }

    std::ostringstream oss;
    oss << "Bulls: " << result.bulls << ", Cows: " << result.cows;

    if (result.guessed) {
        oss << " You guessed the word!";
    }

```

```

    }

    PipeManager::send_message(client_fd, Message{CMD_RESPONSE, oss.str(), "", ""});

    if (result.guessed) {
        broadcast_game_state(game_name);
    }
}

void Server::process_get_state(const Message& msg,
int client_fd, const std::string& client_name) {
    if (player_to_game.find(client_name)
== player_to_game.end()) {
        PipeManager::send_message(client_fd,
        Message{CMD_ERROR, "Not in any game", "", ""});
        return;
    }

    std::string game_name = player_to_game[client_name];
    auto it = games.find(game_name);
    if (it == games.end()) {
        PipeManager::send_message(client_fd, Message{CMD_ERROR, "Game not found", "",
        return;
    }

    GameRoom& room = it->second;

    std::ostringstream oss;
    oss << "Game: " << room.
name << "\n";
    oss << "Players: " << r
oom.players.size() << "/" << room.max_players << "\n";
    oss << "Status: " << (room.in_progress ? "In progress" : "Waiting") << "\n";
    oss << "Players list:\n";

    for (size_t i = 0; i < room.players.size(); ++i) {
        oss << " " << room.players[i] << " - Score: " << room.scores[i] << "\n";
    }

    PipeManager::send_message(client_fd, Message{CMD_RESPONSE, oss.str(), "", ""});
}

```

```

void Server::process_list_games(int client_fd) {
    std::ostringstream oss;

    if (games.empty()) {
        oss << "No games available";
    } else {
        oss << "Available games:\n";
        for (const auto& pair : games) {
            const GameRoom& room = pair.second;
            oss << "  " << room.name << " ("
                << room.players.size() << "/" << room.max_players << ") - ";
            oss << (room.in_progress ? "In progress" : "Waiting") << "\n";
        }
    }

    PipeManager::send_message(client_fd, Message{CMD_RESPONSE, oss.str(), "", ""});
}

void Server::process_leave_game(const std::string& client_name) {
    remove_player(client_name);
}

void Server::broadcast_game_state(const std::string& game_name) {
    auto it = games.find(game_name);
    if (it == games.end()) return;

    GameRoom& room = it->second;

    for (const auto& player : room.players) {
        std::string client_pipe = PipeManager::get_client_pipe(player);
        int client_fd = PipeManager::open_pipe(client_pipe, O_WRONLY);

        if (client_fd != -1) {
            process_get_state(Message{CMD_GET_STATE, "", player, game_name},
                             client_fd, player);
            PipeManager::close_pipe(client_fd);
        }
    }
}

```

```

void Server::remove_player(const std::string& player_name) {
    auto game_it = player_to_game.find(player_name);
    if (game_it == player_to_game.end()) return;

    std::string game_name = game_it->second;
    player_to_game.erase(player_name);

    auto room_it = games.find(game_name);
    if (room_it == games.end()) return;

    GameRoom& room = room_it->second;

    auto player_it = std::find(room.players.begin(), room.players.end(), player_name)
    if (player_it != room.players.end()) {
        int index = std::distance(room.players.begin(), player_it);
        room.players.erase(player_it);
        room.scores.erase(room.scores.begin() + index);
    }

    if (room.players.empty()) {
        games.erase(room_it);
    } else {
        broadcast_game_state(game_name);
    }
}

```

client.h

```
#pragma once
```

```
#include <string>
```

```
#include "common.h"
```

```
class Client {
```

```
private:
```

```
    std::string player_name;
```

```
    std::string client_pipe;
```

```
    bool connected;
```

```
    bool setup_connection();
```

```
    void send_message(const Message& msg);
```

```
    Message receive_message();
```

```

        void cleanup();

public:
    Client(const std::string& name);
    ~Client();

    void run();
    void create_game();
    void join_game();
    void make_guess();
    void show_game_state();
    void list_games();
    void leave_game();
};

client.cpp,
#include "client.h"

#include <iostream>
#include <string>
#include <thread>
#include <chrono>
#include <cstring>
#include <csignal>
#include <fcntl.h>

#include "pipe_manager.h"
#include "game_logic.h"

Client::Client(const std::string& name) : player_name(name), connected(false) {
    client_pipe = PipeManager::get_client_pipe(player_name);
}

Client::~~Client() {
    cleanup();
}

bool Client::setup_connection() {
    if (!PipeManager::create_pipe(client_pipe)) {
        std::cerr << "Cannot create client pipe\n";
        return false;
    }
}

```

```

    }

    connected = true;
    return true;
}

void Client::send_message(const Message& msg) {
    int server_fd = PipeManager::open_pipe(SERVER_PIPE, O_WRONLY);
    if (server_fd == -1) {
        std::cerr << "Cannot connect to server\n";
        return;
    }

    PipeManager::send_message(server_fd, msg);
    PipeManager::close_pipe(server_fd);
}

Message Client::receive_message() {
    int client_fd = PipeManager::open_pipe(client_pipe, O_RDONLY);
    if (client_fd == -1) {
        return Message{CMD_ERROR, "", "", ""};
    }

    Message msg = PipeManager::receive_message(client_fd);
    PipeManager::close_pipe(client_fd);

    return msg;
}

void Client::cleanup() {
    if (connected) {
        unlink(client_pipe.c_str());
    }
}

void Client::run() {
    if (!setup_connection()) {
        return;
    }

    std::cout << "Connected as: " << player_name << "\n";
}

```

```

while (connected) {
    std::cout << "1. Create game\n";
    std::cout << "2. Join game\n";
    std::cout << "3. List games\n";
    std::cout << "4. Make guess\n";
    std::cout << "5. Show game state\n";
    std::cout << "6. Leave game\n";
    std::cout << "7. Exit\n\n";
    std::cout << "Choice: ";

    int choice;
    std::cin >> choice;
    std::cin.ignore();

    switch (choice) {
        case 1:
            create_game();
            break;
        case 2:
            join_game();
            break;
        case 3:
            list_games();
            break;
        case 4:
            make_guess();
            break;
        case 5:
            show_game_state();
            break;
        case 6:
            leave_game();
            break;
        case 7:
            connected = false;
            send_message(Message{CMD_LEAVE_GAME, "", player_name, ""});
            break;
        default:
            std::cout << "Invalid choice\n";
    }
}

```

```

    }

    std::cout << "Disconnected\n";
}

void Client::create_game() {
    std::string game_name;
    int max_players;

    std::cout << "Game name: ";
    std::getline(std::cin, game_name);

    std::cout << "Max players (1-" << MAX_PLAYERS << "): ";
    std::cin >> max_players;
    std::cin.ignore();

    send_message(Message{CMD_CREATE_GAME, std::to_string(max_players),
        player_name, game_name});

    Message response = receive_message();
    std::cout << response.data << "\n";
}

void Client::join_game() {
    std::string game_name;

    std::cout << "Game name: ";
    std::getline(std::cin, game_name);

    send_message(Message{CMD_JOIN_GAME, "",
        player_name, game_name});

    Message response = receive_message();
    std::cout << response.data << "\n";
}

void Client::list_games() {
    send_message(Message{CMD_LIST_GAMES, "", player_name, ""});

    Message response = receive_message();
    std::cout << response.data << "\n";
}

```

```

}

void Client::make_guess() {
    std::string guess;

    std::cout << "Your guess (" << WORD_LENGTH << " letters): ";
    std::getline(std::cin, guess);

    if (guess.length() != WORD_LENGTH) {
        std::cout << "Word must be " << WORD_LENGTH << " letters\n";
        return;
    }

    send_message(Message{CMD_MAKE_GUESS, guess, player_name, ""});

    Message response = receive_message();
    std::cout << response.data << "\n";
}

void Client::show_game_state() {
    send_message(Message{CMD_GET_STATE, "", player_name, ""});

    Message response = receive_message();
    std::cout << response.data << "\n";
}

void Client::leave_game() {
    send_message(Message{CMD_LEAVE_GAME, "", player_name, ""});
    std::cout << "Left the game\n";
}

main-server.cpp
#include "server.h"

int main() {
    Server server;
    server.run();
    return 0;
}

main-client.cpp.

```

```

#include "client.h"
#include <iostream>

int main(int argc, char* argv[]) {
    if (argc != 2) {
        std::cout << "Usage: " << argv[0] << " <player_name>\n";
        return 1;
    }

    Client client(argv[1]);
    client.run();

    return 0;
}

```

Strace

```

motorola5000@DESKTOP:~$ strace -f -e trace=process,file,signal ./server

```

```

execve("./server", [ "./server" ], 0x7ffc89d4b780 /* 37 vars */) = 0

```

```

rt_sigaction(SIGCHLD, {s
_handler=SIG_IGN, sa_mask=[], sa_flags=SA_RESTORER|S
A_RESTART,
sa_restorer=0x7f1234567890}, NULL, 8) = 0

```

```

mknodat(AT_FDCWD, "/tmp/bulls_cows_server", S_IFIFO|0666) = 0

```

```

openat(AT_FDCWD, "/tmp/bulls_cows_server", O_RDONLY) = 3

```

```

write(1, "Server starting\n", 16)      = 16

```

```

write(1, "Server ready. Waiting for connections\n", 38) = 38

```

```

read(3, "0|Alice|MyGame|3\n", 1024) = 16

```

```

clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|
CLONE_CHILD_SETTID|SIGCHLD, child_tidptr=0x7f8c12345a10)
= 12345

```

```

[pid 12345] --- Создан процесс для Алисы ---

```

```

[pid 12345]

```

```

[pid 12345] mknodat(AT_FDCWD, "/tmp/client_Alice", S_IFIFO|0666) = 0

```

```

[pid 12345]

```

```

[pid 12345] openat(AT_FDCWD, "/tmp/client_Alice", O_WRONLY) = 4

```

```

[pid 12345]

```

```

[pid 12345] write(4, "\2|Game created. Secret word: apple\
n", 36) = 36

```

```

[pid 12345] close(4) = 0
[pid 12345] exit_group(0) = ?
read(3, <unfinished ...>
read(3, "1|Bob|MyGame|\n", 1024) = 14
clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|
CLONE_CHILD_SETTID
|SIGCHLD, child_tidptr=0x7f8c12345a10) = 12346
[pid 12346] --- Создан процесс для Боба ---

[pid 12346]
[pid 12346] openat(AT_FDCWD, "/tmp/client_Bob", O_WRONLY) = 4
[pid 12346] write(4, "\2|Joined game: MyGame\n", 22) = 22
[pid 12346]
[pid 12346] openat(AT_FDCWD, "/tmp/client
_Alice", O_WRONLY) = 5
[pid 12346] write(5, "\2|Game: MyGame\nPla
yers: 2/3\nStatus: Waiting\n...", 95) = 95
[pid 12346]
[pid 12346] write(4, "\2|Game: My
Game\nPlayers: 2/3\nStatus: Waiting\n...", 95) = 95
[pid 12346] close(4) = 0
[pid 12346] close(5) = 0
[pid 12346] exit_group(0) = ?

read(3, <unfinished ...>

```