Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт №8 “Компьютерные науки и прикладная математика”

Кафедра №806 “Вычислительная математика и программирование”

Курсовой проект по курсу

«Операционные системы»

Группа: М8О-215Б-23

Студент: Лапенко К.А.

Преподаватель: Миронов Е.С.

Оценка: _____

Дата: 26.02.25

Москва, 2024

Постановка задачи

Вариант 5

Исследование 2 аллокаторов памяти: необходимо реализовать два алгоритма аллокации памяти и сравнить их по следующим характеристикам:

- Фактор использования
- Скорость выделения блоков
- Скорость освобождения блоков
- Простота использования аллокатора

Каждый аллокатор памяти должен иметь функции аналогичные стандартным функциям free и malloc (realloc, опционально). Перед работой каждый аллокатор инициализируется свободными страницами памяти, выделенными стандартными средствами ядра. Необходимо самостоятельно разработать стратегию тестирования для определения ключевых характеристик аллокаторов памяти. При тестировании нужно свести к минимуму потери точности из-за накладных расходов при измерении ключевых характеристик, описанных выше.
В отчете необходимо отобразить следующее:

- Подробное описание каждого из исследуемых алгоритмов
- Процесс тестирования
- Обоснование подхода тестирования
- Результаты тестирования
- Заключение по проведенной работе

Задание

Вариант 5. Морской бой. Общение между сервером и клиентом необходимо организовать при помощи memory map. Каждый игрок должен при запуске ввести свой логин. Для каждого игрока должна вестись статистика игр (сколько побед/поражений). Игрок может посмотреть свою статистику

Общий метод и алгоритм решения

Структура проекта и назначение файлов:

common.h

Назначение:
Определяет все общие константы, перечисления и структуры, используемые как сервером, так и клиентом. В данном файле задаются:

- Размеры игрового поля, количество игроков и игр.
- Типы клеток игрового поля (EMPTY, SHIP, MISS, HIT, DESTROYED).
- Типы кораблей и их количество.
- Структуры для описания корабля, игрового поля (GameBoard), игры (Game) и статистики игрока (PlayerStats).
- Определение структуры сообщения (Message) и общей памяти (SharedMemory), которая используется для обмена данными между процессами.

Это позволяет обеспечить единое понимание формата данных для всех компонентов системы .

server.cpp

Назначение:
Серверная часть приложения, ответственная за:

- Инициализацию разделяемой памяти, создание объекта памяти (shm_open, mmap) и семафоров для синхронизации (sem_open).
- Загрузку статистики игроков из файла (loadStats) и сохранение данных при завершении работы.
- Обработку сигналов (например, SIGINT) для корректного завершения работы и очистки ресурсов.
- Обработку различных типов сообщений, поступающих от клиентов, таких как:
  - Авторизация (LOGIN и LOGIN_RESPONSE).
  - Создание игры (CREATE_GAME и CREATE_GAME_RESPONSE).
  - Присоединение к игре (JOIN_GAME и JOIN_GAME_RESPONSE).
  - Запрос списка игр (LIST_GAMES, GAMES_LIST).
  - Расстановка кораблей (PLACE_SHIP, PLACE_SHIP_RESPONSE, SHIPS_READY и SHIPS_READY_RESPONSE).
  - Игровой процесс (MAKE_MOVE, MOVE_RESULT, GAME_STATUS).

Алгоритм работы сервера сводится к циклическому ожиданию сообщений от клиентов, их обработке и отправке соответствующих ответов, обновлению состояния игры и статистики .

client.cpp

Назначение:
Клиентская часть приложения, позволяющая пользователю:

- Подключиться к разделяемой памяти и семафорам, созданным сервером.
- Взаимодействовать с сервером, отправляя сообщения о своих действиях (авторизация, выбор опций меню, создание/присоединение к игре, ход в игре, расстановка кораблей).
- Отображать игровой интерфейс: вывод игрового поля, статистики, сообщений о состоянии игры и ходов.
- Обрабатывать ответы сервера и обновлять локальное состояние игры (например, отображать промахи, попадания, уничтоженные корабли).

Клиент взаимодействует с сервером через общий сегмент памяти, синхронизируясь с помощью семафоров (sem_post/sem_wait) для обмена сообщениями .

Алгоритм работы системы:

Инициализация сервера:

- Создание разделяемой памяти:
  Сервер создаёт объект разделяемой памяти посредством shm_open, устанавливает его размер через ftruncate и отображает его в адресное пространство с помощью mmap.
- Инициализация семафоров:
  Создаются семафоры для синхронизации работы между сервером и клиентами (SEM_CLIENT_READY и SEM_SERVER_READY).
- Загрузка статистики:
  Из файла статистики загружаются данные о игроках (число побед, поражений, текущий статус).

Обработка сообщений сервера

- Авторизация:
  При получении сообщения LOGIN сервер проверяет, зарегистрирован ли игрок, и отправляет ответ LOGIN_RESPONSE, устанавливая статус игрока (новый или уже активный).

- Создание/присоединение к игре:
  При получении запроса CREATE_GAME или JOIN_GAME сервер проверяет наличие игры с заданным именем, создаёт новую игру или присоединяет игрока, обновляет состояние игры и отправляет соответствующий ответ.
- Расстановка кораблей:
  Клиенты отправляют запросы на размещение кораблей (PLACE_SHIP), сервер проверяет корректность координат и обновляет игровое поле. После расстановки кораблей оба игрока уведомляются о готовности.
- Игровой процесс:
  Клиенты отправляют ходы (MAKE_MOVE), сервер проверяет попадание, обновляет игровое поле противника, определяет результат (промах, попадание, уничтожение корабля или победу) и отправляет MOVE_RESULT. Также сервер регулярно передаёт статус игры (GAME_STATUS) для синхронизации ходов.
- Завершение игры и обновление статистики:
  При достижении состояния GAME_OVER сервер обновляет статистику (увеличивает счетчик побед и поражений) и сохраняет данные в файле.

Работа клиента

- Подключение и авторизация:
  Клиент подключается к существующей разделяемой памяти и семафорам, отправляет запрос LOGIN и получает ответ от сервера.
- Выбор действий:
  Клиент предлагает пользователю меню, где можно создать новую игру, присоединиться к существующей игре, просмотреть статистику или выйти.
- Интерактивный игровой процесс:
  При создании или присоединении к игре клиент отображает игровое поле, предоставляет интерфейс для размещения кораблей, ожидания оппонента и совершения ходов. Клиент периодически отправляет запросы GAME_STATUS для получения обновлений.
- Обработка ответов:
  Клиент обрабатывает ответы сервера, обновляет локальное состояние игры, отображает результаты ходов и выводит информацию о состоянии игры и статистике.

Взаимодействие между файлами

- common.h
  Является ядром проекта: в нем определены все общие структуры и константы, которые используются как сервером, так и клиентом для формирования сообщений, хранения статистики и состояния игры. Это обеспечивает единообразие обмена данными между процессами.
- server.cpp
  Реализует серверную логику: принимает запросы от клиентов, обрабатывает их, обновляет игровое состояние, управляет статистикой и выполняет синхронизацию через разделяемую память и семафоры. Сервер постоянно работает в цикле, ожидая поступления сообщений и выполняя соответствующие действия.
- client.cpp
  Обеспечивает пользовательский интерфейс для клиента. Клиент отправляет запросы серверу (например, для авторизации, создания игры, совершения хода) и получает ответы, отображая их пользователю. Клиентская часть также занимается обработкой пользовательского ввода и обновлением локального отображения игрового процесса.

Код программы

client.cpp

```
#include <iostream>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include <semaphore.h>
#include <string>
#include <sstream>
#include <iomanip>
#include <cstdlib>
#include <cstring>
#include "common.h"

// Отображение игрового поля
void displayBoard(const CellState board[BOARD_SIZE][BOARD_SIZE], bool hideShips =
false) {
    std::cout << "  ";
    for (int x = 0; x < BOARD_SIZE; x++) {
        std::cout << " " << x;
    }
    std::cout << std::endl;

    for (int y = 0; y < BOARD_SIZE; y++) {
        std::cout << y << " ";
        for (int x = 0; x < BOARD_SIZE; x++) {
            char symbol;
            switch (board[y][x]) {
                case EMPTY:
                    symbol = '.';
                    break;
                case SHIP:
                    symbol = hideShips ? '.' : 'S';
                    break;
                case MISS:
                    symbol = 'o';
                    break;
                case HIT:
                    symbol = 'X';
                    break;
                case DESTROYED:
                    symbol = '#';
                    break;
                default:
                    symbol = '?';
            }
            std::cout << " " << symbol;
        }
        std::cout << std::endl;
    }
}

// Function to display boards horizontally (side by side)
void displayBoardsHorizontally(const CellState myBoard[BOARD_SIZE][BOARD_SIZE],
                               const CellState enemyBoard[BOARD_SIZE][BOARD_SIZE],
                               bool hideEnemyShips = true) {
    // Header
    std::cout << "      Your Board                Enemy Board      " << std::endl;

    // Column numbers
    std::cout << "  ";
    for (int x = 0; x < BOARD_SIZE; x++) {
        std::cout << " " << x;
    }
    std::cout << "      ";
```

```
    for (int x = 0; x < BOARD_SIZE; x++) {
        std::cout << " " << x;
    }
    std::cout << std::endl;

    // Board contents
    for (int y = 0; y < BOARD_SIZE; y++) {
        // First board row
        std::cout << y << " ";
        for (int x = 0; x < BOARD_SIZE; x++) {
            char symbol;
            switch (myBoard[y][x]) {
            case EMPTY: symbol = '.'; break;
            case SHIP: symbol = 'S'; break;
            case MISS: symbol = 'o'; break;
            case HIT: symbol = 'X'; break;
            case DESTROYED: symbol = '#'; break;
            default: symbol = '?';
            }
            std::cout << " " << symbol;
        }

        // Spacing between boards
        std::cout << "    ";

        // Second board row
        std::cout << y << " ";
        for (int x = 0; x < BOARD_SIZE; x++) {
            char symbol;
            switch (enemyBoard[y][x]) {
            case EMPTY: symbol = '.'; break;
            case SHIP: symbol = hideEnemyShips ? '.' : 'S'; break;
            case MISS: symbol = 'o'; break;
            case HIT: symbol = 'X'; break;
            case DESTROYED: symbol = '#'; break;
            default: symbol = '?';
            }
            std::cout << " " << symbol;
        }
        std::cout << std::endl;
    }
}

bool waitForOpponentShips(SharedMemory* sharedMem, sem_t* semClientReady, sem_t*
semServerReady,
                          std::string username, std::string gameName) {
    std::cout << "\nWaiting for your opponent to place their ships..." << std::endl;

    int pollCount = 0;
    const int MAX_POLLS = 300; // Ждем 5 минут

    while (pollCount < MAX_POLLS) {
        // Poll for game status
        sharedMem->message.type = Message::GAME_STATUS;
        strcpy(sharedMem->message.username, username.c_str());
        strcpy(sharedMem->message.gameName, gameName.c_str());

        sem_post(semClientReady);
        sem_wait(semServerReady);

        if (sharedMem->message.type == Message::GAME_STATUS) {
            // Игра началась? (все поставили корабли)
            if (sharedMem->message.gameState == PLAYER1_TURN ||
                sharedMem->message.gameState == PLAYER2_TURN) {
                std::cout << "\nYour opponent has finished placing ships!" << std::endl;
                std::cout << "Game is starting now..." << std::endl;
                return true;
```

```
            }

            // Check if the game has ended unexpectedly
            if (sharedMem->message.gameState == GAME_OVER) {
                std::cout << "\nGame has ended: " << sharedMem->message.data <<
std::endl;
                return false;
            }
        }

        // Мини анимашка ожидания
        if (pollCount % 5 == 0) {
            std::cout << "." << std::flush;
        }

        sleep(1); // Ждем секунду перед проверкой на соединение
        pollCount++;
    }

    std::cout << "\nWaited too long for opponent. You can check back later." <<
std::endl;
    return false;
}

// Функция для размещения кораблей
void placeShips(SharedMemory* sharedMem, sem_t* semClientReady, sem_t* semServerReady,
                std::string username, std::string gameName) {
    system("clear");
    std::cout << "\n====== Ship Placement ======\n" << std::endl;
    std::cout << "You need to place:\n";
    std::cout << "- " << BATTLESHIP_COUNT << " battleships (4 cells)\n";
    std::cout << "- " << CRUISER_COUNT << " cruisers (3 cells)\n";
    std::cout << "- " << DESTROYER_COUNT << " destroyers (2 cells)\n";
    std::cout << "- " << SUBMARINE_COUNT << " submarines (1 cell)\n";

    // Локальная копия доски для отображения
    CellState localBoard[BOARD_SIZE][BOARD_SIZE] = {};
    for (int y = 0; y < BOARD_SIZE; y++) {
        for (int x = 0; x < BOARD_SIZE; x++) {
            localBoard[y][x] = EMPTY;
        }
    }

    // Массив для отслеживания размещенных кораблей
    int shipsPlaced[5] = {0}; // 0 не используется, 1-4 - длины кораблей

    // Цикл размещения кораблей
    while (true) {
        std::cout << "\nCurrent board:" << std::endl;
        displayBoard(localBoard);

        std::cout << "\nRemaining ships:" << std::endl;
        std::cout << "- Battleships (4): " << BATTLESHIP_COUNT - shipsPlaced[4] <<
std::endl;
        std::cout << "- Cruisers (3): " << CRUISER_COUNT - shipsPlaced[3] << std::endl;
        std::cout << "- Destroyers (2): " << DESTROYER_COUNT - shipsPlaced[2] <<
std::endl;
        std::cout << "- Submarines (1): " << SUBMARINE_COUNT - shipsPlaced[1] <<
std::endl;

        // Проверяем, все ли корабли размещены
        if (shipsPlaced[1] == SUBMARINE_COUNT &&
            shipsPlaced[2] == DESTROYER_COUNT &&
            shipsPlaced[3] == CRUISER_COUNT &&
            shipsPlaced[4] == BATTLESHIP_COUNT) {

            // Отправляем серверу уведомление, что корабли готовы
```

```
            sharedMem->message.type = Message::SHIPS_READY;
            strcpy(sharedMem->message.username, username.c_str());
            strcpy(sharedMem->message.gameName, gameName.c_str());

            sem_post(semClientReady);
            sem_wait(semServerReady);

            if (sharedMem->message.type == Message::SHIPS_READY_RESPONSE) {
                std::cout << sharedMem->message.data << std::endl;
                break;
            } else {
                std::cerr << "Unexpected server response!" << std::endl;
                return;
            }
        }

        // Ввод данных для размещения корабля
        int shipLength;
        do {
            std::cout << "\nEnter ship length (1-4): ";
            std::string input;
            std::getline(std::cin, input);
            std::stringstream ss(input);
            if (!(ss >> shipLength) || shipLength < 1 || shipLength > 4) {
                std::cout << "Invalid length. Please enter a number between 1 and 4." <<
std::endl;
                shipLength = 0;
                continue;
            }

            // Проверяем, остались ли корабли этой длины
            if ((shipLength == 4 && shipsPlaced[4] >= BATTLESHIP_COUNT) ||
                (shipLength == 3 && shipsPlaced[3] >= CRUISER_COUNT) ||
                (shipLength == 2 && shipsPlaced[2] >= DESTROYER_COUNT) ||
                (shipLength == 1 && shipsPlaced[1] >= SUBMARINE_COUNT)) {
                std::cout << "You have already placed all ships of this length!" <<
std::endl;
                shipLength = 0;
            }
        } while (shipLength < 1 || shipLength > 4);

        // Получаем координаты
        int x, y;
        std::cout << "Enter coordinates (format: x y): ";
        std::string input;
        std::getline(std::cin, input);
        std::stringstream ss(input);
        if (!(ss >> x >> y) || x < 0 || x >= BOARD_SIZE || y < 0 || y >= BOARD_SIZE) {
            std::cout << "Invalid coordinates! Please try again." << std::endl;
            continue;
        }

        // Запрос ориентации (для кораблей длиннее 1)
        bool horizontal = true;
        if (shipLength > 1) {
            std::cout << "Orientation (h - horizontal, v - vertical): ";
            std::getline(std::cin, input);
            horizontal = (input != "v" && input != "V");
        }

        // Отправляем запрос на размещение корабля
        sharedMem->message.type = Message::PLACE_SHIP;
        strcpy(sharedMem->message.username, username.c_str());
        strcpy(sharedMem->message.gameName, gameName.c_str());
        sharedMem->message.x = x;
        sharedMem->message.y = y;
        sharedMem->message.shipLength = shipLength;
```

```
        sharedMem->message.shipHorizontal = horizontal;

        sem_post(semClientReady);
        sem_wait(semServerReady);

        if (sharedMem->message.type == Message::PLACE_SHIP_RESPONSE) {
            std::cout << sharedMem->message.data << std::endl;

            // Если корабль успешно размещен, обновляем локальную доску
            if (strstr(sharedMem->message.data, "successfully") != nullptr) {
                // Размещение на локальной доске
                for (int i = 0; i < shipLength; i++) {
                    int shipX = horizontal ? x + i : x;
                    int shipY = horizontal ? y : y + i;
                    localBoard[shipY][shipX] = SHIP;
                }

                // Обновляем счетчик размещенных кораблей
                shipsPlaced[shipLength]++;
            }

            system("clear");
        } else {
            std::cerr << "Unexpected server response!" << std::endl;
        }
    }
}

// Функция для игрового процесса
void playGame(SharedMemory* sharedMem, sem_t* semClientReady, sem_t* semServerReady,
              std::string username, std::string gameName, GameState initialState,
std::string opponent) {
    system("clear");
    std::cout << "\n====== Game Started ======\n" << std::endl;
    std::cout << "You are playing against: " << opponent << std::endl;

    // Локальные копии досок для отображения
    CellState myBoard[BOARD_SIZE][BOARD_SIZE] = {};    // Моя доска
    CellState enemyBoard[BOARD_SIZE][BOARD_SIZE] = {}; // Доска противника

    // Инициализация пустыми клетками только для доски противника
    for (int y = 0; y < BOARD_SIZE; y++) {
        for (int x = 0; x < BOARD_SIZE; x++) {
            enemyBoard[y][x] = EMPTY;
        }
    }

    // Запрашиваем состояние доски
    sharedMem->message.type = Message::GAME_STATUS;
    strcpy(sharedMem->message.username, username.c_str());
    strcpy(sharedMem->message.gameName, gameName.c_str());

    sem_post(semClientReady);
    sem_wait(semServerReady);

    int playerIdx = -1;
    // Находим игру и определяем какой мы игрок
    for (int i = 0; i < sharedMem->gameCount; i++) {
        if (strcmp(sharedMem->games[i].name, gameName.c_str()) == 0) {
            if (strcmp(sharedMem->games[i].player1, username.c_str()) == 0) {
                // Мы игрок 1, копируем доску 1
                for (int y = 0; y < BOARD_SIZE; y++) {
                    for (int x = 0; x < BOARD_SIZE; x++) {
                        myBoard[y][x] = sharedMem->games[i].board1.cells[y][x];
                    }
                }
                playerIdx = 1;
```

```
                break;
            } else if (strcmp(sharedMem->games[i].player2, username.c_str()) == 0) {
                // Мы игрок 2, копируем доску 2
                for (int y = 0; y < BOARD_SIZE; y++) {
                    for (int x = 0; x < BOARD_SIZE; x++) {
                        myBoard[y][x] = sharedMem->games[i].board2.cells[y][x];
                    }
                }
                playerIdx = 2;
                break;
            }
        }
    }

    bool isPlayer1 = (playerIdx == 1);

    // Текущее состояние игры
    GameState gameState = initialState;
    bool isMyTurn = (gameState == PLAYER1_TURN && isPlayer1) ||
                    (gameState == PLAYER2_TURN && !isPlayer1);

    while (gameState != GAME_OVER) {
//         // Отображаем обе доски
//         std::cout << "\nYour board:" << std::endl;
//         displayBoard(myBoard);
//
//         std::cout << "\nEnemy board:" << std::endl;
//         displayBoard(enemyBoard, true);  // Скрываем корабли противника
        std::cout << std::endl;
        displayBoardsHorizontally(myBoard, enemyBoard);

        if (isMyTurn) {
            std::cout << "\nYour turn! Enter coordinates to fire (format: x y): ";
            std::string input;
            std::getline(std::cin, input);
            system("clear");

            // Обработка выхода из игры
            if (input == "quit" || input == "exit") {
                std::cout << "Exiting game..." << std::endl;
                break;
            }

            std::stringstream ss(input);
            int x, y;
            if (!(ss >> x >> y) || x < 0 || x >= BOARD_SIZE || y < 0 || y >= BOARD_SIZE)
{
                std::cout << "Invalid coordinates! Please try again." << std::endl;
                continue;
            }

            // Отправляем ход на сервер
            sharedMem->message.type = Message::MAKE_MOVE;
            strcpy(sharedMem->message.username, username.c_str());
            strcpy(sharedMem->message.gameName, gameName.c_str());
            sharedMem->message.x = x;
            sharedMem->message.y = y;

            sem_post(semClientReady);
            sem_wait(semServerReady);

            if (sharedMem->message.type == Message::MOVE_RESULT) {
                std::cout << sharedMem->message.data << std::endl;

                // Обновляем локальную доску противника в соответствии с результатом
                if (sharedMem->message.hitResult >= 0) {
                    switch (sharedMem->message.hitResult) {
```

```
                    case 0: // Промах
                        enemyBoard[y][x] = MISS;
                        isMyTurn = false;
                        break;
                    case 1: // Попадание
                        enemyBoard[y][x] = HIT;
                        break;
                    case 2: // Корабль уничтожен
                        // Фулл обновляем доску для отметки всего корября пореженным
                        for (int i = 0; i < sharedMem->gameCount; i++) {
                            if (strcmp(sharedMem->games[i].name, gameName.c_str())
== 0) {
                                // Кто мы?
                                const GameBoard& updatedBoard = isPlayer1 ?
sharedMem->games[i].board2 : sharedMem->games[i].board1;

                                // Берем только уничтоженные клетки
                                for (int boardY = 0; boardY < BOARD_SIZE; boardY++)
{
                                    for (int boardX = 0; boardX < BOARD_SIZE;
boardX++) {
                                        if (updatedBoard.cells[boardY][boardX] ==
DESTROYED) {
                                            enemyBoard[boardY][boardX] = DESTROYED;
                                        }
                                    }
                                }
                                break;
                            }
                        }
                        break;
                    case 3: // Победа
                        enemyBoard[y][x] = DESTROYED;
                        gameState = GAME_OVER;
                        std::cout << "\nCongratulations! You won the game!" <<
std::endl;
                        break;
                }
            }

            // Обновляем состояние игры
            gameState = sharedMem->message.gameState;
        } else {
            std::cerr << "Unexpected server response!" << std::endl;
        }
    } else {
        std::cout << "\nWaiting for opponent's move..." << std::endl;

        // Чекаем обновления игры пока ждем оппонента
        bool opponentMoved = false;
        while (!opponentMoved) {
            // Чекаем обновы
            sharedMem->message.type = Message::GAME_STATUS;
            strcpy(sharedMem->message.username, username.c_str());
            strcpy(sharedMem->message.gameName, gameName.c_str());

            sem_post(semClientReady);
            sem_wait(semServerReady);

            if (sharedMem->message.type == Message::GAME_STATUS) {
                GameState updatedState = sharedMem->message.gameState;

                // Наш ход?
                if ((updatedState == PLAYER1_TURN && isPlayer1) ||
                    (updatedState == PLAYER2_TURN && !isPlayer1)) {
                    isMyTurn = true;
                    opponentMoved = true;
```

```
                        gameState = updatedState;

                        // Обновляем доску на основе данных сервера
                        // Соединяем удары и нашу доску
                        for (int i = 0; i < sharedMem->gameCount; i++) {
                            if (strcmp(sharedMem->games[i].name, gameName.c_str()) == 0)
{
                                if (isPlayer1) {
                                    // Мы игрок 1 - копируем доску 1, которая содержит
удары противника
                                    for (int y = 0; y < BOARD_SIZE; y++) {
                                        for (int x = 0; x < BOARD_SIZE; x++) {
                                            myBoard[y][x] =
sharedMem->games[i].board1.cells[y][x];
                                        }
                                    }
                                } else {
                                    // Мы игрок 2 - копируем доску 2, которая содержит
удары противника
                                    for (int y = 0; y < BOARD_SIZE; y++) {
                                        for (int x = 0; x < BOARD_SIZE; x++) {
                                            myBoard[y][x] =
sharedMem->games[i].board2.cells[y][x];
                                        }
                                    }
                                }
                                break;
                            }
                        }
                        system("clear");
                        std::cout << "     Your opponent made a move. Your turn now!" <<
std::endl;
                    } else if (updatedState == GAME_OVER) {
                        gameState = GAME_OVER;
                        opponentMoved = true;

                        // Check if we lost by updating our board one last time
                        for (int i = 0; i < sharedMem->gameCount; i++) {
                            if (strcmp(sharedMem->games[i].name, gameName.c_str()) == 0)
{
                                if (isPlayer1) {
                                    for (int y = 0; y < BOARD_SIZE; y++) {
                                        for (int x = 0; x < BOARD_SIZE; x++) {
                                            myBoard[y][x] =
sharedMem->games[i].board1.cells[y][x];
                                        }
                                    }
                                } else {
                                    for (int y = 0; y < BOARD_SIZE; y++) {
                                        for (int x = 0; x < BOARD_SIZE; x++) {
                                            myBoard[y][x] =
sharedMem->games[i].board2.cells[y][x];
                                        }
                                    }
                                }
                                break;
                            }
                        }
                        system("clear");
                        std::cout << "😭 Game ended! Your opponent has won 😭" <<
std::endl;
                    }
                }

                if (!opponentMoved) {
                    sleep(1); // Ждем немного снова
                }
```

```
            }
        }
    }

    std::cout << "\nGame over!" << std::endl;
}

// Функция для получения и отображения статистики
void viewStats(SharedMemory* sharedMem, sem_t* semClientReady, sem_t* semServerReady,
std::string username) {
    sharedMem->message.type = Message::GET_STATS;
    strcpy(sharedMem->message.username, username.c_str());

    sem_post(semClientReady);
    sem_wait(semServerReady);

    if (sharedMem->message.type == Message::STATS_DATA) {
        system("clear");
        std::cout << "\n====== Player Statistics ======\n" << std::endl;
        std::cout << sharedMem->message.data << std::endl;
    } else {
        std::cerr << "Error retrieving statistics!" << std::endl;
    }
}

// Функция для получения списка доступных игр
std::string getGamesList(SharedMemory* sharedMem, sem_t* semClientReady, sem_t*
semServerReady, std::string username) {
    sharedMem->message.type = Message::LIST_GAMES;
    strcpy(sharedMem->message.username, username.c_str());

    sem_post(semClientReady);
    sem_wait(semServerReady);

    if (sharedMem->message.type == Message::GAMES_LIST) {
        return sharedMem->message.data;
    } else {
        return "Error retrieving games list!";
    }
}

int main() {
    // Открываем существующий объект памяти
    int fd = shm_open(MMF_NAME, O_RDWR, 0666);
    if (fd == -1) {
        std::cerr << "Error opening shared memory. Is the server running?" << std::endl;
        return 1;
    }

    // Отображаем память
    SharedMemory* sharedMem = (SharedMemory*)mmap(NULL, MMF_SIZE,
                                  PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (sharedMem == MAP_FAILED) {
        std::cerr << "Error mapping shared memory: " << strerror(errno) << std::endl;
        close(fd);
        return 1;
    }

    // Открываем существующие семафоры
    sem_t* semClientReady = sem_open(SEM_CLIENT_READY, 0);
    sem_t* semServerReady = sem_open(SEM_SERVER_READY, 0);

    if (semClientReady == SEM_FAILED || semServerReady == SEM_FAILED) {
        std::cerr << "Error opening semaphores: " << strerror(errno) << std::endl;
        munmap(sharedMem, MMF_SIZE);
        close(fd);
        return 1;
```

```
    }

    std::cout << "====== Welcome to Sea Battle ======\n" << std::endl;

    // Авторизация
    std::string username;
    std::cout << "Please enter your username: ";
    std::getline(std::cin, username);

    if (username.empty() || username.length() > 63) {
        std::cerr << "Invalid username! It must be between 1 and 63 characters." <<
std::endl;
        return 1;
    }

    // Отправляем запрос авторизации
    sharedMem->message.type = Message::LOGIN;
    strncpy(sharedMem->message.username, username.c_str(),
sizeof(sharedMem->message.username) - 1);
    sharedMem->message.username[sizeof(sharedMem->message.username) - 1] = '\0';
    strcpy(sharedMem->message.data, "Login request");

    // Уведомляем сервер
    sem_post(semClientReady);

    // Ждем ответа от сервера
    sem_wait(semServerReady);

    // Проверяем ответ на авторизацию
    if (sharedMem->message.type == Message::LOGIN_RESPONSE) {
        if (strcmp(sharedMem->message.data, "Already online") == 0) {
              std::cout << "Player is already online" << std::endl;
              exit(0);
        }
        std::cout << sharedMem->message.data << std::endl;
    } else {
        std::cerr << "Unexpected server response during login!" << std::endl;
        munmap(sharedMem, MMF_SIZE);
        close(fd);
        sem_close(semClientReady);
        sem_close(semServerReady);
        return 1;
    }

    // Основной игровой цикл
    std::string input;
    bool running = true;

    while (running) {
        std::cout << "\nOptions:\n";
        std::cout << "1. Create a new game\n";
        std::cout << "2. Join an existing game\n";
        std::cout << "3. View your statistics\n";
        std::cout << "4. Exit\n";
        std::cout << "Enter your choice (1-4): ";

        std::getline(std::cin, input);

        if (input == "1") {
            // Создание новой игры
            std::cout << "Enter game name: ";
            std::string gameName;
            std::getline(std::cin, gameName);

            if (gameName.empty() || gameName.length() > 63) {
                std::cout << "Invalid game name! It must be between 1 and 63
characters." << std::endl;
```

```
                continue;
            }

            // Отправляем запрос на создание игры
            sharedMem->message.type = Message::CREATE_GAME;
            strncpy(sharedMem->message.data, gameName.c_str(),
sizeof(sharedMem->message.data) - 1);
            sharedMem->message.data[sizeof(sharedMem->message.data) - 1] = '\0';
            strcpy(sharedMem->message.username, username.c_str());

            // Уведомляем сервер
            sem_post(semClientReady);

            // Ждем ответа от сервера
            sem_wait(semServerReady);

            if (sharedMem->message.type == Message::CREATE_GAME_RESPONSE) {
                system("clear");
                std::cout << "Server response: " << sharedMem->message.data <<
std::endl;

                if (sharedMem->message.gameState == WAITING_FOR_PLAYER) {
                    if (strcmp(sharedMem->message.data, "Game with this name already
exists!") == 0) {
                        continue;
                    }
                    if (strcmp(sharedMem->message.data, "Maximum number of games
reached!") == 0) {
                        continue;
                    }
                    std::string gameName = sharedMem->message.gameName;
                    std::cout << "Waiting for an opponent to join..." << std::endl;

                    // Ждем пока оппонент присоединится
                    int pollCount = 0;
                    const int MAX_POLLS = 600; // 10 minutes maximum wait time at 1
second intervals
                    bool opponentJoined = false;

                    while (pollCount < MAX_POLLS && !opponentJoined) {
                        // Чекаем статус игры
                        sharedMem->message.type = Message::GAME_STATUS;
                        strcpy(sharedMem->message.username, username.c_str());
                        strcpy(sharedMem->message.gameName, gameName.c_str());

                        sem_post(semClientReady);
                        sem_wait(semServerReady);

                        if (sharedMem->message.type == Message::GAME_STATUS) {
                            // Оппонент подсоединился? - ставим корабли
                            if (sharedMem->message.gameState == PLACING_SHIPS) {
                                opponentJoined = true;
                                std::cout << "\nAn opponent has joined! Moving to ship
placement phase..." << std::endl;

                                // Подсоединяемся к игре, чтобы начать ставить корабли
                                sharedMem->message.type = Message::JOIN_GAME;
                                strcpy(sharedMem->message.username, username.c_str());
                                strcpy(sharedMem->message.gameName, gameName.c_str());

                                sem_post(semClientReady);
                                sem_wait(semServerReady);

                                if (sharedMem->message.type ==
Message::JOIN_GAME_RESPONSE) {
                                    std::string opponentName =
sharedMem->message.opponent;
```

```
                                    // Ставим корабли
                                    placeShips(sharedMem, semClientReady,
semServerReady, username, gameName);

                                    // Ждем пока оппонент поставит корабли
                                    if (waitForOpponentShips(sharedMem, semClientReady,
semServerReady, username, gameName)) {
                                        // Оба поставили - начинаем битву
                                        playGame(sharedMem, semClientReady,
semServerReady, username, gameName,
                                                 sharedMem->message.gameState,
opponentName);
                                    }
                                }
                            }
                        }

                        // Снова мини анимашка
                        if (pollCount % 5 == 0) {
                            std::cout << "." << std::flush;
                        }

                        sleep(1); // Ждем секунду
                        pollCount++;
                    }

                    if (!opponentJoined) {
                        std::cout << "\nWaited too long for an opponent. Returning to
main menu." << std::endl;
                    }
                }
            } else {
                std::cerr << "Unexpected server response!" << std::endl;
            }
        }
        else if (input == "2") {
            // Получаем список игр
            std::string gamesList = getGamesList(sharedMem, semClientReady,
semServerReady, username);
            std::cout << "\n" << gamesList << std::endl;

            std::cout << "Enter game name to join (or 'back' to return): ";
            std::string gameName;
            std::getline(std::cin, gameName);

            if (gameName == "back") {
                continue;
            }

            if (gameName.empty()) {
                std::cout << "Game name cannot be empty!" << std::endl;
                continue;
            }

            // Запрос на подсоединение
            sharedMem->message.type = Message::JOIN_GAME;
            strcpy(sharedMem->message.username, username.c_str());
            strcpy(sharedMem->message.gameName, gameName.c_str());

            sem_post(semClientReady);
            sem_wait(semServerReady);

            if (sharedMem->message.type == Message::JOIN_GAME_RESPONSE) {
                std::cout << sharedMem->message.data << std::endl;
                std::string opponentName = sharedMem->message.opponent;
```

```
                if (sharedMem->message.gameState == PLACING_SHIPS) {
                    // Ставим корабли
                    placeShips(sharedMem, semClientReady, semServerReady, username,
gameName);

                    // Игра готова или ждем оппонентов?
                    if (waitForOpponentShips(sharedMem, semClientReady, semServerReady,
username, gameName)) {
                        // Корабли поставлены - начинаем!
                        playGame(sharedMem, semClientReady, semServerReady, username,
gameName,
                                 sharedMem->message.gameState, opponentName);
                    }
                }
            } else {
                std::cerr << "Unexpected server response!" << std::endl;
            }
        }  else if (input == "3") {
            // Просмотр статистики
            viewStats(sharedMem, semClientReady, semServerReady, username);

        } else if (input == "4") {
            std::cout << "Thank you for playing. Goodbye!" << std::endl;
            running = false;

        } else {
            std::cout << "Invalid option. Please try again." << std::endl;
        }
    }

    // Освобождаем ресурсы
    sem_close(semClientReady);
    sem_close(semServerReady);
    munmap(sharedMem, MMF_SIZE);
    close(fd);

    return 0;
}
```

server.cpp

```
#include <iostream>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include <semaphore.h>
#include <signal.h>
#include <fstream>
#include <cstring>
#include <vector>
#include <ctime>
#include <cstdlib>
#include "common.h"

// Global variables to store player data
PlayerStats g_players[MAX_PLAYERS];
int g_playerCount = 0;


// Глобальные переменные для обработки сигналов
SharedMemory* g_sharedMem = nullptr;
int g_shm_fd = -1;
sem_t* g_semClientReady = nullptr;
sem_t* g_semServerReady = nullptr;

// Загрузка статистики из файла
void loadStats() {
```

```
    std::ifstream file(STATS_FILE, std::ios::binary);
    if (!file) {
        std::cout << "Stats file not found, starting with empty database." << std::endl;
        g_playerCount = 0;
        return;
    }

    file.read(reinterpret_cast<char*>(&g_playerCount), sizeof(int));

    if (g_playerCount > MAX_PLAYERS) {
        std::cerr << "Warning: Corrupt stats file or too many players. Resetting." <<
std::endl;
        g_playerCount = 0;
        return;
    }

    for (int i = 0; i < g_playerCount; i++) {
        file.read(reinterpret_cast<char*>(&g_players[i]), sizeof(PlayerStats));
        g_players[i].active = false;
        g_players[i].inGame = false;
    }

    std::cout << "Loaded " << g_playerCount << " player records." << std::endl;
    file.close();
}

// Сохранение статистики в файл
void saveStats() {
    std::ofstream file(STATS_FILE, std::ios::binary);
    if (!file) {
        std::cerr << "Error: Cannot open stats file for writing!" << std::endl;
        return;
    }

    file.write(reinterpret_cast<const char*>(&g_playerCount), sizeof(int));

    for (int i = 0; i < g_playerCount; i++) {
        file.write(reinterpret_cast<const char*>(&g_players[i]), sizeof(PlayerStats));
    }

    std::cout << "Saved " << g_playerCount << " player records." << std::endl;
    file.close();
}

// Поиск игрока по имени
int findPlayer(const char* username) {
    for (int i = 0; i < g_playerCount; i++) {
        if (strcmp(g_players[i].username, username) == 0) {
            return i;
        }
    }
    return -1;
}

// Добавление нового игрока
int addPlayer(const char* username) {
    if (g_playerCount >= MAX_PLAYERS) {
        return -1; // max players reached
    }

    int idx = g_playerCount++;
    strncpy(g_players[idx].username, username, sizeof(g_players[idx].username) - 1);
    g_players[idx].username[sizeof(g_players[idx].username) - 1] = '\0';
    g_players[idx].wins = 0;
    g_players[idx].losses = 0;
    g_players[idx].active = true;
    g_players[idx].inGame = false;
```

```
    g_players[idx].currentGame[0] = '\0';

    return idx;
}

// Поиск игры по имени
int findGame(SharedMemory* sharedMem, const char* gameName) {
    for (int i = 0; i < sharedMem->gameCount; i++) {
        if (strcmp(sharedMem->games[i].name, gameName) == 0 &&
sharedMem->games[i].active) {
            return i;
        }
    }
    return -1;
}

// Создание новой игры
int createGame(SharedMemory* sharedMem, const char* gameName, const char* playerName) {
    if (sharedMem->gameCount >= MAX_GAMES) {
        return -1; // достигнут максимум игр
    }

    // Проверяем, не занято ли это имя
    if (findGame(sharedMem, gameName) != -1) {
        return -2; // игра с таким именем уже существует
        }

    int idx = sharedMem->gameCount++;
    strncpy(sharedMem->games[idx].name, gameName, sizeof(sharedMem->games[idx].name) -
1);
    sharedMem->games[idx].name[sizeof(sharedMem->games[idx].name) - 1] = '\0';

    strncpy(sharedMem->games[idx].player1, playerName,
sizeof(sharedMem->games[idx].player1) - 1);
    sharedMem->games[idx].player1[sizeof(sharedMem->games[idx].player1) - 1] = '\0';

    sharedMem->games[idx].player2[0] = '\0';
    sharedMem->games[idx].state = WAITING_FOR_PLAYER;
    sharedMem->games[idx].winner = 0;
    sharedMem->games[idx].active = true;

    // Очищаем игровые поля
    sharedMem->games[idx].board1.clear();
    sharedMem->games[idx].board2.clear();

    // Обновляем статус игрока
    int playerIdx = findPlayer(playerName);
    if (playerIdx != -1) {
        g_players[playerIdx].inGame = true;
        strncpy(g_players[playerIdx].currentGame, gameName,
                sizeof(g_players[playerIdx].currentGame) - 1);
        g_players[playerIdx].currentGame[sizeof(g_players[playerIdx].currentGame) - 1] =
'\0';
    }

    return idx;
}

// Подсоединение к игре
bool joinGame(SharedMemory* sharedMem, const char* gameName, const char* playerName) {
    int gameIdx = findGame(sharedMem, gameName);
    if (gameIdx == -1) {
        return false; // Игры не найдено
    }

    // Special case: создатель присоединяется в своей же игре
    if (strcmp(sharedMem->games[gameIdx].player1, playerName) == 0 &&
```

```
        sharedMem->games[gameIdx].state == PLACING_SHIPS) {
        return true; // Allow player1 to join their own game for ship placement
        }

    // Если игрка не в состоянии ожидания или игрок хочет подключится сам к себе - стоп
    if (sharedMem->games[gameIdx].state != WAITING_FOR_PLAYER) {
        return false;
        }

    // Подсоединяем игрока к игре
    strncpy(sharedMem->games[gameIdx].player2, playerName,
sizeof(sharedMem->games[gameIdx].player2) - 1);
    sharedMem->games[gameIdx].player2[sizeof(sharedMem->games[gameIdx].player2) - 1] =
'\0';

    // Состояние игры - расстановка корабле
    sharedMem->games[gameIdx].state = PLACING_SHIPS;

    // Обновляем статус игрока
    int playerIdx = findPlayer(playerName);
    if (playerIdx != -1) {
        g_players[playerIdx].inGame = true;
        strncpy(g_players[playerIdx].currentGame, gameName,
                sizeof(g_players[playerIdx].currentGame) - 1);
        g_players[playerIdx].currentGame[sizeof(g_players[playerIdx].currentGame) - 1] =
'\0';
    }

    return true;
}


// Размещение корабля на поле
bool placeShip(GameBoard& board, int x, int y, int length, bool horizontal) {
    // Проверка выхода за границы поля
    if (x < 0 || y < 0 || x >= BOARD_SIZE || y >= BOARD_SIZE) {
        return false;
    }

    if (horizontal) {
        if (x + length > BOARD_SIZE) return false;
    } else {
        if (y + length > BOARD_SIZE) return false;
    }

    // Проверка пересечения с другими кораблями (включая соседние клетки)
    for (int i = -1; i <= length; i++) {
        for (int j = -1; j <= 1; j++) {
            int checkX = horizontal ? x + i : x + j;
            int checkY = horizontal ? y + j : y + i;

            if (checkX >= 0 && checkX < BOARD_SIZE && checkY >= 0 && checkY <
BOARD_SIZE) {
                if (board.cells[checkY][checkX] == SHIP) {
                    return false;
                }
            }
        }
    }

    // Размещаем корабль на поле
    if (board.shipsPlaced >= TOTAL_SHIPS) {
        return false; // все корабли уже размещены
    }

    board.ships[board.shipsPlaced].x = x;
    board.ships[board.shipsPlaced].y = y;
```

```
    board.ships[board.shipsPlaced].length = length;
    board.ships[board.shipsPlaced].horizontal = horizontal;
    board.ships[board.shipsPlaced].hits = 0;

    // Отмечаем клетки на поле
    for (int i = 0; i < length; i++) {
        if (horizontal) {
            board.cells[y][x + i] = SHIP;
        } else {
            board.cells[y + i][x] = SHIP;
        }
    }

    board.shipsPlaced++;
    return true;
}

// Проверка, что все корабли размещены
bool areAllShipsPlaced(const GameBoard& board) {
    int expected[5] = {0, SUBMARINE_COUNT, DESTROYER_COUNT, CRUISER_COUNT,
BATTLESHIP_COUNT};
    int actual[5] = {0}; // Индекс - длина корабля

    for (int i = 0; i < board.shipsPlaced; i++) {
        if (board.ships[i].length >= 1 && board.ships[i].length <= 4) {
            actual[board.ships[i].length]++;
        }
    }

    for (int i = 1; i <= 4; i++) {
        if (actual[i] != expected[i]) {
            return false;
        }
    }

    return true;
}


// Обработка хода игрока
int processMove(GameBoard& opponentBoard, int x, int y) {
    if (x < 0 || y < 0 || x >= BOARD_SIZE || y >= BOARD_SIZE) {
        return -1; // недопустимые координаты
    }

    // Уже стреляли в эту клетку
    if (opponentBoard.cells[y][x] == MISS || opponentBoard.cells[y][x] == HIT ||
        opponentBoard.cells[y][x] == DESTROYED) {
        return -2;
    }

    // Промах
    if (opponentBoard.cells[y][x] == EMPTY) {
        opponentBoard.cells[y][x] = MISS;
        return 0;
    }

    // Попадание
    if (opponentBoard.cells[y][x] == SHIP) {
        opponentBoard.cells[y][x] = HIT;

        // Проверяем, какой корабль поражен
        for (int i = 0; i < opponentBoard.shipsPlaced; i++) {
            Ship& ship = opponentBoard.ships[i];
            bool hit = false;

            for (int j = 0; j < ship.length; j++) {
```

```
                int shipX = ship.horizontal ? ship.x + j : ship.x;
                int shipY = ship.horizontal ? ship.y : ship.y + j;

                if (shipX == x && shipY == y) {
                    ship.hits++;
                    hit = true;
                    break;
                }
            }

            if (hit) {
                // Проверяем, уничтожен ли корабль
                if (ship.isDestroyed()) {
                    // Помечаем все клетки корабля как уничтоженные
                    for (int j = 0; j < ship.length; j++) {
                        int shipX = ship.horizontal ? ship.x + j : ship.x;
                        int shipY = ship.horizontal ? ship.y : ship.y + j;
                        opponentBoard.cells[shipY][shipX] = DESTROYED;
                    }

                    // Проверяем, все ли корабли уничтожены
                    if (opponentBoard.allShipsDestroyed()) {
                        return 3; // победа
                    }
                    return 2; // корабль уничтожен
                }
                return 1; // попадание
            }
        }
    }

    // Не должны сюда добраться, но на всякий случай
    return 0;
}

// Обработчик сигнала для корректного завершения
void signalHandler(int sig) {
    if (sig == SIGINT) {
        std::cout << "\nReceived SIGINT. Saving data and cleaning up..." << std::endl;

        if (g_sharedMem) {
            saveStats(); // Updated to not use sharedMem
            munmap(g_sharedMem, MMF_SIZE);
        }

        // Rest of the handler remains the same
        if (g_semClientReady) sem_close(g_semClientReady);
        if (g_semServerReady) sem_close(g_semServerReady);

        sem_unlink(SEM_CLIENT_READY);
        sem_unlink(SEM_SERVER_READY);

        if (g_shm_fd != -1) close(g_shm_fd);
        shm_unlink(MMF_NAME);

        exit(0);
    }
}

// Расчет процента побед
float calculateWinRate(int wins, int losses) {
    int total = wins + losses;
    if (total == 0) {
        return 0.0f;
    }
    return (float)wins * 100.0f / (float)total;
}
```

```
// Функция для красивого вывода
void centerText(char* buffer, const char* text, size_t width) {
    size_t textLen = strlen(text);
    if (textLen >= width) {
        // If text is longer than width, just copy it
        strcpy(buffer, text);
    } else {
        // Calculate padding
        size_t padding = (width - textLen) / 2;
        // Use sprintf to center the text
        sprintf(buffer, "%*s%s%*s", (int)padding, "", text, (int)(width - textLen -
padding), "");
    }
}

int main() {
    // Инициализируем генератор случайных чисел
    srand(static_cast<unsigned int>(time(nullptr)));

    // На всякий случай чистим
    shm_unlink(MMF_NAME);
    sem_unlink(SEM_CLIENT_READY);
    sem_unlink(SEM_SERVER_READY);


    // Установка обработчика сигнала
    signal(SIGINT, signalHandler);
    std::cout << "Sigint handler initalized" << std::endl;

    std::cout << "Initializing shared memory..." << std::endl;
    // Создаем объект в разделяемой памяти
    g_shm_fd = shm_open(MMF_NAME, O_CREAT | O_RDWR, 0666);
    if (g_shm_fd == -1) {
        std::cerr << "Error creating shared memory: " << strerror(errno) << std::endl;
        return 1;
    }

    // Устанавливаем размер
    if (ftruncate(g_shm_fd, MMF_SIZE) == -1) {
        std::cerr << "Error setting shared memory size: " << strerror(errno) <<
std::endl;
        close(g_shm_fd);
        shm_unlink(MMF_NAME);
        return 1;
    }

    // Отображаем в память
    g_sharedMem = (SharedMemory*)mmap(NULL, MMF_SIZE,
                                PROT_READ | PROT_WRITE, MAP_SHARED, g_shm_fd, 0);
    if (g_sharedMem == MAP_FAILED) {
        std::cerr << "Error mapping shared memory: " << strerror(errno) << std::endl;
        close(g_shm_fd);
        shm_unlink(MMF_NAME);
        return 1;
    }

    // Ставим все в нули
    memset(&g_sharedMem->message, 0, sizeof(Message));
    g_sharedMem->gameCount = 0;

    // Безопасно инициализируем массивы
    for (int i = 0; i < MAX_PLAYERS; i++) {
        memset(&g_players[i], 0, sizeof(PlayerStats));
    }
    for (int i = 0; i < MAX_GAMES; i++) {
        memset(&g_sharedMem->games[i], 0, sizeof(Game));
```

```
    }
    std::cout << "Shared memory initalized" << std::endl;

    // Загружаем статистику и игры
    loadStats();
    // loadGames(g_sharedMem);
    std::cout << "Stats downloaded" << std::endl;

    std::cout << "Initializing semaphores..." << std::endl;
    // Создаем семафоры для синхронизации
    g_semClientReady = sem_open(SEM_CLIENT_READY, O_CREAT, 0666, 0);
    if (g_semClientReady == SEM_FAILED) {
        std::cerr << "Error creating client semaphore: " << strerror(errno) <<
std::endl;
        munmap(g_sharedMem, MMF_SIZE);
        close(g_shm_fd);
        shm_unlink(MMF_NAME);
        return 1;
    }

    g_semServerReady = sem_open(SEM_SERVER_READY, O_CREAT, 0666, 0);
    if (g_semServerReady == SEM_FAILED) {
        std::cerr << "Error creating server semaphore: " << strerror(errno) <<
std::endl;
        sem_close(g_semClientReady);
        sem_unlink(SEM_CLIENT_READY);
        munmap(g_sharedMem, MMF_SIZE);
        close(g_shm_fd);
        shm_unlink(MMF_NAME);
        return 1;
    }
    std::cout << "Initializing semaphores complete" << std::endl;

//     // Чистим все ожидающие сигналы на семафорах
//     while (sem_trywait(g_semClientReady) == 0) {
//         // Пустой цикл для очищения семафора
//     }
//
//     while (sem_trywait(g_semServerReady) == 0) {
//         // Пустой цикл для очищения семафора
//     }

    std::cout << "\nSea Battle Server started. Press Ctrl+C to save and exit." <<
std::endl;

    // Основной цикл сервера
    while (true) {
        // Ожидаем сообщение от клиента
        sem_wait(g_semClientReady);

        // Обрабатываем различные типы сообщений
        switch (g_sharedMem->message.type) {
            case Message::LOGIN:
                {
                    std::string username = g_sharedMem->message.username;
                    std::cout << "Login request from: " << username << std::endl;

                    int playerIdx = findPlayer(username.c_str());
                    bool isNewUser = (playerIdx == -1);
                    bool isAlreadyActive = (g_players[playerIdx].active == true);

                    if (isNewUser) {
                        playerIdx = addPlayer(username.c_str());
                        std::cout << "New player registered: " << username << std::endl;
                    } else {
                        g_players[playerIdx].active = true;
```

```
                    g_players[playerIdx].inGame = false; // Reset game status on
login

                    std::cout << "Returning player: " << username
                              << " (W:" << g_players[playerIdx].wins
                              << "/L:" << g_players[playerIdx].losses << ")" <<
std::endl;
                }

                // Form response
                g_sharedMem->message.type = Message::LOGIN_RESPONSE;
                g_sharedMem->message.newUser = isNewUser;

                if (isNewUser) {
                    strcpy(g_sharedMem->message.data, "Registration successful!");
                } else if (isAlreadyActive) {
                    strcpy(g_sharedMem->message.data, "Already online");
                } else {
                    sprintf(g_sharedMem->message.data,
                            "Welcome back, %s! Your stats: %d wins, %d losses",
                            username.c_str(),
                            g_players[playerIdx].wins,
                            g_players[playerIdx].losses);
                }
            }
            break;

        case Message::CREATE_GAME:
            {
                std::string gameName = g_sharedMem->message.data;
                std::string username = g_sharedMem->message.username;

                std::cout << "Create game request: " << gameName << " from " <<
username << std::endl;

                int gameIdx = createGame(g_sharedMem, gameName.c_str(),
username.c_str());
                g_sharedMem->message.type = Message::CREATE_GAME_RESPONSE;

                if (gameIdx == -1) {
                    strcpy(g_sharedMem->message.data, "Maximum number of games
reached!");
                } else if (gameIdx == -2) {
                    strcpy(g_sharedMem->message.data, "Game with this name already
exists!");
                } else {
                    sprintf(g_sharedMem->message.data,
                            "Game '%s' created successfully! Waiting for
opponent...",
                            gameName.c_str());
                    g_sharedMem->message.gameState = WAITING_FOR_PLAYER;
                    strcpy(g_sharedMem->message.gameName, gameName.c_str());
                }
            }
            break;

        case Message::LIST_GAMES:
            {
                std::cout << "List games request from " <<
g_sharedMem->message.username << std::endl;

                // Создаем список доступных игр
                g_sharedMem->message.type = Message::GAMES_LIST;

                std::string gamesList = "Available games:\n";
                bool foundGames = false;
```

```
                    for (int i = 0; i < g_sharedMem->gameCount; i++) {
                        if (g_sharedMem->games[i].active) {
                            // Игры в статусе ожидания
                            if (g_sharedMem->games[i].state == WAITING_FOR_PLAYER &&
                                strcmp(g_sharedMem->games[i].player1,
g_sharedMem->message.username) != 0) {
                                gamesList += "- ";
                                gamesList += g_sharedMem->games[i].name;
                                gamesList += " (created by ";
                                gamesList += g_sharedMem->games[i].player1;
                                gamesList += ")\n";
                                foundGames = true;
                                }
                        }
                    }

                    if (!foundGames) {
                        gamesList += "No games available. Create your own game!\n";
                    }

                    strncpy(g_sharedMem->message.data, gamesList.c_str(),
sizeof(g_sharedMem->message.data) - 1);
                    g_sharedMem->message.data[sizeof(g_sharedMem->message.data) - 1] =
'\0';
                }
                break;

            case Message::JOIN_GAME:
                {
                    std::string gameName = g_sharedMem->message.gameName;
                    std::string username = g_sharedMem->message.username;

                    std::cout << "Join game request: " << gameName << " from " <<
username << std::endl;

                    bool joined = joinGame(g_sharedMem, gameName.c_str(),
username.c_str());
                    g_sharedMem->message.type = Message::JOIN_GAME_RESPONSE;

                    if (!joined) {
                        strcpy(g_sharedMem->message.data,
                               "Could not join game. It may not exist, already started,
or you created it.");
                        g_sharedMem->message.gameState = GAME_OVER; // Для индикации
клиенту об ошибке
                    } else {
                        sprintf(g_sharedMem->message.data,
                                "Successfully joined game '%s'! Place your ships.",
                                gameName.c_str());

                        // Находим игру для получения информации о состоянии
                        int gameIdx = findGame(g_sharedMem, gameName.c_str());
                        if (gameIdx != -1) {
                            g_sharedMem->message.gameState =
g_sharedMem->games[gameIdx].state;
                            strcpy(g_sharedMem->message.gameName, gameName.c_str());

                            // Ставим нужного оппонент
                            if (strcmp(g_sharedMem->games[gameIdx].player1,
username.c_str()) == 0) {
                                // Player 1 is joining, so opponent is player 2
                                strcpy(g_sharedMem->message.opponent,
g_sharedMem->games[gameIdx].player2);
                            } else {
                                // Player 2 is joining, so opponent is player 1
                                strcpy(g_sharedMem->message.opponent,
g_sharedMem->games[gameIdx].player1);
```

```
                        }
                    }
                }
            }
            break;

            case Message::GAME_STATUS:
            {
                std::string gameName = g_sharedMem->message.gameName;
                std::string username = g_sharedMem->message.username;

                // std::cout << "Game status request from " << username << " for
game " << gameName << std::endl;

                int gameIdx = findGame(g_sharedMem, gameName.c_str());
                g_sharedMem->message.type = Message::GAME_STATUS;

                if (gameIdx == -1) {
                    strcpy(g_sharedMem->message.data, "Game not found!");
                    g_sharedMem->message.gameState = GAME_OVER;
                    break;
                }

                // Возвращаем текущее состояние игры
                g_sharedMem->message.gameState = g_sharedMem->games[gameIdx].state;

                // Чей ход
                bool isPlayer1 = (strcmp(g_sharedMem->games[gameIdx].player1,
username.c_str()) == 0);
                bool isPlayer2 = (strcmp(g_sharedMem->games[gameIdx].player2,
username.c_str()) == 0);

                if (!isPlayer1 && !isPlayer2) {
                    strcpy(g_sharedMem->message.data, "You are not a participant in
this game!");
                    break;
                }

                // Для ждущего отправляем инфу о последнем ходе
                if ((g_sharedMem->games[gameIdx].state == PLAYER1_TURN && isPlayer2)
||
                    (g_sharedMem->games[gameIdx].state == PLAYER2_TURN &&
isPlayer1)) {

                    // In a real implementation, we would store and retrieve the
last move's coordinates and result
                    // For now, we'll use defaults
                    g_sharedMem->message.x = -1;
                    g_sharedMem->message.y = -1;
                    g_sharedMem->message.hitResult = -1;
                    strcpy(g_sharedMem->message.data, "Waiting for opponent's
move");
                    } else {
                        sprintf(g_sharedMem->message.data, "It's your turn in game
%s", gameName.c_str());
                    }
            }
            break;

            case Message::PLACE_SHIP:
            {
                std::string gameName = g_sharedMem->message.gameName;
                std::string username = g_sharedMem->message.username;
                int x = g_sharedMem->message.x;
                int y = g_sharedMem->message.y;
                int length = g_sharedMem->message.shipLength;
                bool horizontal = g_sharedMem->message.shipHorizontal;
```

```
                    std::cout << "Place ship request from " << username << " in game "
<< gameName
                              << " at (" << x << "," << y << "), length " << length
                              << (horizontal ? " horizontal" : " vertical") <<
std::endl;

                    int gameIdx = findGame(g_sharedMem, gameName.c_str());
                    g_sharedMem->message.type = Message::PLACE_SHIP_RESPONSE;

                    if (gameIdx == -1) {
                        strcpy(g_sharedMem->message.data, "Game not found!");
                        break;
                    }

                    // Определяем номер игрока
                    bool isPlayer1 = (strcmp(g_sharedMem->games[gameIdx].player1,
username.c_str()) == 0);
                    bool isPlayer2 = (strcmp(g_sharedMem->games[gameIdx].player2,
username.c_str()) == 0);

                    if (!isPlayer1 && !isPlayer2) {
                        strcpy(g_sharedMem->message.data, "You are not a participant in
this game!");
                        break;
                    }

                    // Проверяем, что игра в фазе расстановки кораблей
                    if (g_sharedMem->games[gameIdx].state != PLACING_SHIPS) {
                        strcpy(g_sharedMem->message.data, "Game is not in the ship
placement phase!");
                        break;
                    }

                    // Выбираем соответствующую доску
                    GameBoard& board = isPlayer1 ? g_sharedMem->games[gameIdx].board1 :
g_sharedMem->games[gameIdx].board2;

                    // Проверяем, что осталось место для корабля
                    int shipsOfLength[5] = {0}; // Индекс - длина корабля
                    for (int i = 0; i < board.shipsPlaced; i++) {
                        shipsOfLength[board.ships[i].length]++;
                    }

                    bool canPlaceShip = false;
                    if (length == BATTLESHIP && shipsOfLength[BATTLESHIP] <
BATTLESHIP_COUNT) {
                        canPlaceShip = true;
                    } else if (length == CRUISER && shipsOfLength[CRUISER] <
CRUISER_COUNT) {
                        canPlaceShip = true;
                    } else if (length == DESTROYER && shipsOfLength[DESTROYER] <
DESTROYER_COUNT) {
                        canPlaceShip = true;
                    } else if (length == SUBMARINE && shipsOfLength[SUBMARINE] <
SUBMARINE_COUNT) {
                        canPlaceShip = true;
                    }

                    if (!canPlaceShip) {
                        strcpy(g_sharedMem->message.data, "You have placed all ships of
this type!");
                        break;
                    }

                    // Размещаем корабль
                    bool placed = placeShip(board, x, y, length, horizontal);
```

```
                    if (!placed) {
                        strcpy(g_sharedMem->message.data, "Cannot place ship at this
position!");
                    } else {
                        sprintf(g_sharedMem->message.data, "Ship of length %d placed
successfully!", length);

                        // Проверяем, все ли корабли размещены
                        if (areAllShipsPlaced(board)) {
                            strcat(g_sharedMem->message.data, " All ships are now
placed!");
                        }
                    }

                    // Отправляем обновленное количество размещенных кораблей
                    g_sharedMem->message.shipLength = board.shipsPlaced;
                }
                break;

            case Message::SHIPS_READY:
            {
                std::string gameName = g_sharedMem->message.gameName;
                std::string username = g_sharedMem->message.username;

                std::cout << "Ships ready notification from " << username << " in game "
<< gameName << std::endl;

                int gameIdx = findGame(g_sharedMem, gameName.c_str());
                g_sharedMem->message.type = Message::SHIPS_READY_RESPONSE;

                if (gameIdx == -1) {
                    strcpy(g_sharedMem->message.data, "Game not found!");
                    break;
                }

                // Определяем номер игрока
                bool isPlayer1 = (strcmp(g_sharedMem->games[gameIdx].player1,
username.c_str()) == 0);
                bool isPlayer2 = (strcmp(g_sharedMem->games[gameIdx].player2,
username.c_str()) == 0);

                if (!isPlayer1 && !isPlayer2) {
                    strcpy(g_sharedMem->message.data, "You are not a participant in this
game!");
                    break;
                }

                // Проверяем, что игра в фазе расстановки кораблей
                if (g_sharedMem->games[gameIdx].state != PLACING_SHIPS) {
                    strcpy(g_sharedMem->message.data, "Game is not in the ship placement
phase!");
                    break;
                }

                // Проверяем, все ли корабли размещены
                GameBoard& board = isPlayer1 ? g_sharedMem->games[gameIdx].board1 :
g_sharedMem->games[gameIdx].board2;

                if (!areAllShipsPlaced(board)) {
                    strcpy(g_sharedMem->message.data, "You haven't placed all your ships
yet!");
                    break;
                }

                // Проверяем, готовы ли оба игрока
```

```
                GameBoard& otherBoard = isPlayer1 ? g_sharedMem->games[gameIdx].board2 :
g_sharedMem->games[gameIdx].board1;

                if (areAllShipsPlaced(otherBoard)) {
                    // Оба игрока готовы, начинаем игру
                    g_sharedMem->games[gameIdx].state = PLAYER1_TURN;
                    strcpy(g_sharedMem->message.data, "Both players are ready! Game
starts now.");
                    g_sharedMem->message.gameState = PLAYER1_TURN;

                    // Указываем, чей сейчас ход
                    if (isPlayer1) {
                        strcat(g_sharedMem->message.data, " It's your turn!");
                        strcpy(g_sharedMem->message.opponent,
g_sharedMem->games[gameIdx].player2);
                    } else {
                        strcat(g_sharedMem->message.data, " Waiting for opponent's
move.");
                        strcpy(g_sharedMem->message.opponent,
g_sharedMem->games[gameIdx].player1);
                    }
                } else {
                    // Ждем второго игрока
                    strcpy(g_sharedMem->message.data, "\nYour ships are ready! Waiting
for your opponent...");
                    g_sharedMem->message.gameState = PLACING_SHIPS;

                    // Указываем оппонента
                    if (isPlayer1) {
                        strcpy(g_sharedMem->message.opponent,
g_sharedMem->games[gameIdx].player2);
                    } else {
                        strcpy(g_sharedMem->message.opponent,
g_sharedMem->games[gameIdx].player1);
                    }
                }
            }
            break;

            case Message::MAKE_MOVE:
            {
                std::string gameName = g_sharedMem->message.gameName;
                std::string username = g_sharedMem->message.username;
                int x = g_sharedMem->message.x;
                int y = g_sharedMem->message.y;

                std::cout << "Move request from " << username << " in game " << gameName
                          << " at (" << x << "," << y << ")" << std::endl;

                int gameIdx = findGame(g_sharedMem, gameName.c_str());
                g_sharedMem->message.type = Message::MOVE_RESULT;

                if (gameIdx == -1) {
                    strcpy(g_sharedMem->message.data, "Game not found!");
                    break;
                }

                // Определяем номер игрока
                bool isPlayer1 = (strcmp(g_sharedMem->games[gameIdx].player1,
username.c_str()) == 0);
                bool isPlayer2 = (strcmp(g_sharedMem->games[gameIdx].player2,
username.c_str()) == 0);

                if (!isPlayer1 && !isPlayer2) {
                    strcpy(g_sharedMem->message.data, "You are not a participant in this
game!");
                    break;
```

```
                }

                // Проверяем, чей сейчас ход
                if ((g_sharedMem->games[gameIdx].state == PLAYER1_TURN && !isPlayer1) ||
                    (g_sharedMem->games[gameIdx].state == PLAYER2_TURN && !isPlayer2)) {
                    strcpy(g_sharedMem->message.data, "It's not your turn!");
                    break;
                }

                // Выполняем ход
                GameBoard& targetBoard = isPlayer1 ? g_sharedMem->games[gameIdx].board2
: g_sharedMem->games[gameIdx].board1;
                int result = processMove(targetBoard, x, y);

                if (result == -1) {
                    strcpy(g_sharedMem->message.data, "Invalid coordinates!");
                    break;
                } else if (result == -2) {
                    strcpy(g_sharedMem->message.data, "You already fired at this
position!");
                    break;
                }

                // Обрабатываем результат хода
                g_sharedMem->message.hitResult = result;

                    if (result == 0) {
                        centerText(g_sharedMem->message.data, "❌ Miss! ❌", 54);
                        // Переход хода к другому игроку
                        g_sharedMem->games[gameIdx].state = isPlayer1 ? PLAYER2_TURN :
PLAYER1_TURN;
                        g_sharedMem->message.gameState =
g_sharedMem->games[gameIdx].state;
                    } else if (result == 1) {
                        centerText(g_sharedMem->message.data, "💥 Hit! 💥", 54);
                        // Игрок продолжает ход после попадания
                        g_sharedMem->message.gameState =
g_sharedMem->games[gameIdx].state;
                    } else if (result == 2) {
                        centerText(g_sharedMem->message.data, "🔥 Ship destroyed! 🔥",
54);
                        // Игрок продолжает ход после уничтожения корабля
                        g_sharedMem->message.gameState =
g_sharedMem->games[gameIdx].state;
                    } else if (result == 3) {
                        // Победа - все корабли уничтожены
                        centerText(g_sharedMem->message.data, "🌟 Victory! All enemy
ships destroyed! 🌟", 30);
                        g_sharedMem->games[gameIdx].state = GAME_OVER;
                        g_sharedMem->games[gameIdx].winner = isPlayer1 ? 1 : 2;
                        g_sharedMem->message.gameState = GAME_OVER;

                    // Обновляем статистику игроков
                    int winnerIdx = findPlayer(username.c_str());
                    int loserIdx = findPlayer(isPlayer1 ?
g_sharedMem->games[gameIdx].player2 : g_sharedMem->games[gameIdx].player1);

                    if (winnerIdx != -1) {
                        g_players[winnerIdx].wins++;
                        g_players[winnerIdx].inGame = false;
                        g_players[winnerIdx].currentGame[0] = '\0';
                    }

                    if (loserIdx != -1) {
                        g_players[loserIdx].losses++;
                        g_players[loserIdx].inGame = false;
                        g_players[loserIdx].currentGame[0] = '\0';
```

```
                    }
                }
            }
            break;

            case Message::GET_STATS:
                {
                    std::string username = g_sharedMem->message.username;
                    std::cout << "Stats request from " << username << std::endl;

                    int playerIdx = findPlayer(username.c_str());
                    g_sharedMem->message.type = Message::STATS_DATA;

                    if (playerIdx == -1) {
                        strcpy(g_sharedMem->message.data, "Player not found!");
                    } else {
                        sprintf(g_sharedMem->message.data,
                                "Statistics for %s:\nWins: %d\nLosses: %d\nWin rate:
%.1f%%",
                                username.c_str(),
                                g_players[playerIdx].wins,
                                g_players[playerIdx].losses,
                                calculateWinRate(g_players[playerIdx].wins,
g_players[playerIdx].losses));
                    }
                }
                break;

            default:
                std::cout << "Received unknown message type: " <<
g_sharedMem->message.type << std::endl;
                g_sharedMem->message.type = Message::ERROR;
                strcpy(g_sharedMem->message.data, "Unknown command");
                break;
        }

        // Уведомляем клиента, что ответ готов
        sem_post(g_semServerReady);
    }

    saveStats();
    // saveGames(g_sharedMem);
    munmap(g_sharedMem, MMF_SIZE);
    sem_close(g_semClientReady);
    sem_close(g_semServerReady);
    sem_unlink(SEM_CLIENT_READY);
    sem_unlink(SEM_SERVER_READY);
    close(g_shm_fd);
    shm_unlink(MMF_NAME);

    return 0;
}
```

Заключение

1. Программа демонстрирует практическое применение разделяемой памяти и семафоров для организации межпроцессного взаимодействия.
2. Серверная часть отвечает за централизованное управление состоянием игры, статистикой и синхронизацией между игроками.
3. Клиентская часть предоставляет интерактивный интерфейс для участия в игре и взаимодействия с сервером.
4. Совместное использование общих структур, определённых в common.h, гарантирует согласованность данных и корректную обработку сообщений между сервером и клиентом