Министерство образования Республики Беларусь
Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники»

Факультет компьютерных систем и сетей
Кафедра информатики
Дисциплина «Операционные среды и системное программирование»

ОТЧЕТ
к лабораторной работе №3
на тему:
«ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ: ОБМЕН ДАННЫМИ»
БГУИР 6-05-0612-02 67

Выполнил студент группы 353503
КОХАН Артём Игоревич

_____
(дата, подпись студента)

Проверил ассистент каф. информатики
Гриценко Никита Юрьевич

_____
(дата, подпись преподавателя)

Минск 2025

СОДЕРЖАНИЕ

1 ИНДИВИДУАЛЬНОЕ ЗАДАНИЕ

Приложение, демонстрирующее работу многозадачного комплекса с обменом (передачей) или совместным использованием данных несколькими процессами (можно и потоками, но доступ к данным не через массив в общем адресном пространстве). Анализ корректности (отсутствия коллизий). Оценка эффективности механизмов IPC. Специальных требований к приложениям не предъявляется; в частности, во многих случаях они могут быть не обязательно оконными, но также и консольными.

Реализация многопользовательского (с мультиплексированием) вывода. Процесс-сервер: ожидание данных (сообщений) от нескольких источников (клиентов); запись сообщений в файл с соблюдением порядка поступления, снабжая дополнительной информацией (временная метка, идентификатор источника и т.п.) и с соблюдением определенного формата. Опционально – дополнительные функции/возможности: управление файлом «журнала», периодическое архивирование, перезапись и т.п. Процессы-клиенты – достаточно тестово-демонстрационного функционала: передача сгенерированных сообщений серверу для их протоколирования. Варианты используемых IPC в качестве интерфейса сервера: именованные и неименованные каналы, почтовые ящики, очереди сообщений.

2 КРАТКИЕ ТЕОРИТИЧЕСКИЕ СВЕДЕНИЯ

В современных многозадачных операционных системах, таких как Windows, эффективное взаимодействие между процессами является ключевым аспектом для обеспечения надёжности и производительности систем. Основными механизмами межпроцессного взаимодействия (IPC) являются именованные каналы, почтовые ящики, очереди сообщений и разделяемая память. Каждый из этих механизмов предоставляет уникальные возможности для решения специфических задач, связанных с передачей данных между процессами [1].

Именованные каналы являются одним из наиболее мощных средств IPC в Windows, позволяя передавать данные между процессами, которые могут быть расположены как на одном компьютере, так и в сети. Они поддерживают двунаправленную связь, синхронные и асинхронные операции ввода-вывода, что делает их идеальными для создания надёжных серверов логирования, способных обрабатывать запросы от множества клиентов [2].

Почтовые ящики и очереди сообщений предоставляют абстракцию очереди FIFO (first-in, first-out), которая полезна в ситуациях, когда необходимо обеспечить порядок обработки сообщений. Эти механизмы часто используются в задачах, требующих строгой последовательности выполнения и гарантии доставки сообщений.

Разделяемая память позволяет разным процессам обращаться к одним и тем же данным в памяти без необходимости копирования, что может значительно увеличить скорость обмена данными между процессами, особенно при работе с большими объёмами данных [3].

В контексте разработки многопользовательского сервера логирования особенно актуальным является использование именованных каналов для мультиплексированного ввода-вывода. Мультиплексирование позволяет серверу обрабатывать множество входящих соединений в одном или нескольких потоках, что оптимизирует использование системных ресурсов и упрощает архитектуру приложения. Сервер логирования, использующий этот подход, может одновременно принимать данные от различных клиентов, регистрировать их в лог-файл с сохранением порядка поступления и снабжать каждую запись временной меткой и идентификатором источника. Это не только повышает производительность системы, но и облегчает последующий анализ логов.

В данной лабораторной работе мы исследуем, как различные методы IPC влияют на производительность системы, и как мультиплексирование и асинхронные операции ввода-вывода могут быть использованы для повышения эффективности многопользовательских приложений. Также будет рассмотрена роль синхронизации и взаимного исключения в предотвращении коллизий и обеспечении консистентности данных в мультипоточной среде.

3 ОПИСАНИЕ ФУНКЦИЙ ПРОГРАММЫ

3.1 Описание LogServer

Программа LogServer представляет собой серверную часть системы логирования, которая работает как многопользовательский сервер, принимающий сообщения от нескольких клиентов через именованные каналы Windows. Сервер запускается и сразу создает именованный канал с фиксированным именем "\\.\pipe\LogServerPipe", после чего переходит в режим ожидания подключений. Когда клиент подключается к каналу, сервер создает новый экземпляр канала для обработки этого клиента и продолжает ожидать следующие подключения. Каждому подключившемуся клиенту присваивается уникальный последовательный идентификатор.

Основной цикл сервера постоянно создает новые экземпляры каналов и ожидает на них подключений через функцию ConnectNamedPipe. При успешном подключении сервер запускает обработку клиента в том же потоке, что обеспечивает последовательную обработку клиентов. В процессе обработки сервер читает сообщения от клиента, добавляет к каждому сообщению временную метку с точностью до миллисекунд и идентификатор клиента, затем записывает эту информацию в файл "server_log.txt" и одновременно выводит в консоль. Формат временной метки включает год, месяц, день, часы, минуты, секунды и миллисекунды для точного отслеживания времени поступления сообщений.

Сервер отвечает клиенту подтверждением "ACK: Message received" для каждого полученного сообщения, создавая таким образом простой протокол обмена. Когда клиент отправляет специальное сообщение "DISCONNECT", сервер корректно завершает соединение с этим клиентом, освобождает ресурсы и возвращается в режим ожидания новых подключений. Сервер также обрабатывает различные ошибки, включая разрыв соединения, ошибки чтения/записи, и обеспечивает устойчивую работу даже при некорректном завершении работы клиентов. Все операции записи в лог-файл выполняются с немедленным сбросом буфера, что гарантирует сохранность данных даже в случае аварийного завершения работы сервера.

3.2 Описание LogClient

Программа LogClient является клиентской частью системы и предоставляет интерфейс для отправки сообщений серверу логирования. Клиент запускается с интерактивным меню, где пользователь может выбрать один из трех режимов работы: выход из программы, демонстрация работы одного клиента или демонстрация работы нескольких клиентов одновременно. При выборе демонстрационного режима создаются экземпляры клиентов, которые подключаются к серверу через тот же именованный канал "\\.\pipe\LogServerPipe".

Каждый клиент работает по stateless-модели, где для каждого сообщения устанавливается новое соединение с сервером. Процесс отправки сообщения начинается с подключения к каналу через CreateFile, после чего клиент настраивает канал в режим работы с сообщениями с помощью SetNamedPipeHandleState. Затем сообщение отправляется серверу через WriteFile, и клиент ожидает ответное подтверждение, читая его через ReadFile. После получения ответа или обработки ошибки клиент закрывает соединение и освобождает ресурсы.

В демонстрационных режимах клиенты автоматически отправляют серию тестовых сообщений с интервалом в 2 секунды. Сообщения имеют формат "Hello from client X - message Y", где X – идентификатор клиента, а Y – номер сообщения. После отправки всех тестовых сообщений клиент отправляет сообщение "DISCONNECT" для информирования сервера о завершении работы. В многоклиентском режиме программа создает несколько потоков, каждый из которых работает со своим экземпляром клиента, при этом реализована синхронизация вывода в консоль через мьютекс для предотвращения перемешивания сообщений от разных потоков.

Клиент включает механизм повторных попыток подключения при занятости канала – при получении ошибки ERROR_PIPE_BUSY клиент ожидает 1 секунду и повторяет попытку, делая до 3 попыток перед окончательным отказом. Также клиент корректно обрабатывает различные ошибки сети и подключения, различая нормальное отключение сервера (ERROR_BROKEN_PIPE, ERROR_PIPE_NOT_CONNECTED) и реальные ошибки соединения. Все операции клиента сопровождаются информативными сообщениями в консоли, что позволяет отслеживать процесс работы и диагностировать возможные проблемы.

4 ПРИМЕР ВЫПОЛНЕНИЯ ПРОГРАММЫ

4.1 Запуск программы и процесс выполнения LogServer

При запуске программы LogServer в консоли отображается сообщение "=== Log Server Application ===", после чего сервер инициализируется и выводит "Log Server started. Waiting for clients...". Сервер создает файл server_log.txt в текущей директории и записывает в него начальную строку с временной меткой о запуске сервера. Затем сервер переходит в состояние ожидания, периодически выводя "Waiting for client connection...". Когда клиент подключается, в консоли появляется сообщение "Client 1 connected successfully!" и сервер начинает обработку сообщений. В реальном времени отображаются все поступающие сообщения в формате "[2024-01-15 14:30:25.123] [Client:1] Текст сообщения". При отключении клиента выводится "Client 1 disconnected" и сервер возвращается в режим ожидания следующего подключения. Все операции дублируются в файл лога с сохранением временных меток и идентификаторов клиентов.

```
=== Log Server Application ===
Log Server started. Waiting for clients...
[2025-11-01 17:16:45.315] [Client:SERVER] Log server started
Waiting for client connection...
Client 1 connected successfully!
[2025-11-01 17:16:46.503] [Client:1] Client connected
[2025-11-01 17:16:46.504] [Client:1] Hello from client 1 - message 1
Client 1 disconnected
[2025-11-01 17:16:46.504] [Client:1] Client disconnected
Waiting for client connection...
Client 2 connected successfully!
[2025-11-01 17:16:48.506] [Client:2] Client connected
[2025-11-01 17:16:48.506] [Client:2] Hello from client 1 - message 2
Client 2 disconnected
[2025-11-01 17:16:48.507] [Client:2] Client disconnected
Waiting for client connection...
Client 3 connected successfully!
[2025-11-01 17:16:50.509] [Client:3] Client connected
[2025-11-01 17:16:50.509] [Client:3] Hello from client 1 - message 3
Client 3 disconnected
[2025-11-01 17:16:50.510] [Client:3] Client disconnected
Waiting for client connection...
Client 4 connected successfully!
[2025-11-01 17:16:52.512] [Client:4] Client connected
Client 4 disconnected normally
[2025-11-01 17:16:52.513] [Client:4] Client disconnected
Waiting for client connection...
```

Рисунок 4.1 – Результат выполнения программы LogServer

4.2 Запуск программы и процесс выполнения LogClient

При запуске LogClient отображается меню с тремя вариантами: "0 - Exit", "1 - Single client demo", "2 - Multiple clients demo". После выбора режима 1 (один клиент) программа создает клиента с ID=1 и выводит "=== Client 1 started ===". Клиент начинает отправлять сообщения с интервалом 2 секунды, для каждого сообщения отображая "Client 1 sent: Hello from client 1 - message

X" и получая ответ сервера "Client 1 received: ACK: Message received". После отправки трех тестовых сообщений клиент отправляет "DISCONNECT" и завершает работу с выводом "=== Client 1 finished ===". В режиме 2 (несколько клиентов) одновременно запускаются два клиента с ID=1 и ID=2, которые работают параллельно в разных потоках. Сообщения от разных клиентов выводятся в консоль без перемешивания благодаря синхронизации через мьютекс. Можно наблюдать, как клиенты поочередно подключаются к серверу, отправляют сообщения и получают подтверждения.

```
=== Log Client Demo ===
0 - Exit
1 - Single client demo
2 - Multiple clients demo
Choose mode: 1
=== Client 1 started ===
Client 1 sent: Hello from client 1 - message 1
Client 1 received: ACK: Message received
Client 1 sent: Hello from client 1 - message 2
Client 1 received: ACK: Message received
Client 1 sent: Hello from client 1 - message 3
Client 1 received: ACK: Message received
Client 1 sent: DISCONNECT
Client 1: Server disconnected normally
=== Client 1 finished ===
```

Рисунок 4.2 – Результат выполнения программы LogClient

4.3 Описание работы и результатов

Работа программы демонстрирует корректную реализацию многопользовательской системы логирования с использованием именованных каналов Windows. Сервер успешно обрабатывает multiple клиентов, присваивая каждому уникальный идентификатор и записывая все сообщения в лог-файл с точными временными метками. Клиенты могут работать как поодиночке, так и параллельно, обеспечивая надежную доставку сообщений и получение подтверждений от сервера. В ходе выполнения видно, что сервер поддерживает очередь подключений - когда несколько клиентов пытаются подключиться одновременно, они обслуживаются последовательно. Результатом работы является файл server_log.txt, содержащий полную историю всех сообщений от всех клиентов с детальной информацией о времени их получения, что полностью соответствует поставленной задаче создания сервера логирования с коллективным доступом.

ВЫВОД

В ходе выполнения лабораторной работы была успешно разработана распределенная система логирования, состоящая из серверной и клиентской частей, которая демонстрирует организацию межпроцессного взаимодействия через именованные каналы в среде Windows. Программа реализует модель "клиент-сервер", где несколько клиентских процессов могут одновременно отправлять сообщения на центральный сервер для их протоколирования.

Результаты тестирования системы показали корректную работу многопользовательского доступа к серверу логирования. Сервер успешно обрабатывает сообщения от нескольких клиентов, присваивая каждому подключению уникальный идентификатор и обеспечивая сохранение всей поступающей информации в единый лог-файл с временными метками. Использование именованных каналов доказало свою эффективность для организации надежного обмена сообщениями между процессами.

Практическая значимость работы заключается в получении опыта разработки распределенных систем с использованием механизмов IPC Windows, понимании принципов организации многопользовательских сервисов и реализации протоколов прикладного уровня для обмена данными между процессами. Разработанная система может служить основой для создания более сложных распределенных приложений, требующих централизованного сбора и анализа логов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Microsoft Documentation: Named Pipes [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/en-us/windows/win32/ipc/named-pipes. – Дата доступа: 31.10.2025.

[2] Understanding IPC Mechanisms in Windows [Электронный ресурс]. – Режим доступа: https://www.codeproject.com/Articles/34073/Understanding-IPC-Mechanisms-in-Windows. – Дата доступа: 31.10.2025.

[3] Windows Programming: Inter-Process Communication [Электронный ресурс]. – Режим доступа: https://www.tenouk.com/ModuleJ.html. – Дата доступа: 31.10.2025.

ПРИЛОЖЕНИЕ А
(справочное)
Исходный код

Листинг А.1 – Реализация LogServer.cpp

```
#include <iostream>
#include <windows.h>
#include <string>
#include <fstream>
#include <chrono>
#include <iomanip>
#include <sstream>

#define PIPE_NAME L"\\\\.\\pipe\\LogServerPipe"
#define BUFFER_SIZE 4096

class LogServer {
private:
    std::ofstream logFile;
    int clientCounter;

    std::string GetCurrentTimestamp() {
        auto now = std::chrono::system_clock::now();
        auto time_t = std::chrono::system_clock::to_time_t(now);
        auto ms = std::chrono::duration_cast<std::chrono::milliseconds>(
            now.time_since_epoch()) % 1000;

        std::tm timeInfo;
        localtime_s(&timeInfo, &time_t);

        std::stringstream ss;
        ss << std::put_time(&timeInfo, "%Y-%m-%d %H:%M:%S");
        ss << "." << std::setfill('0') << std::setw(3) << ms.count();
        return ss.str();
    }

    void WriteToLog(const std::string& clientId, const std::string& message) {
        std::string timestamp = GetCurrentTimestamp();
        std::string logEntry = "[" + timestamp + "] [Client:" + clientId + "]
" + message + "\n";

        logFile << logEntry;
        logFile.flush();

        std::cout << logEntry;
    }

    HANDLE CreateNamedPipeInstance() {
        return CreateNamedPipe(
            PIPE_NAME,                      // pipe name
            PIPE_ACCESS_DUPLEX,             // read/write access
            PIPE_TYPE_MESSAGE |             // message type pipe
            PIPE_READMODE_MESSAGE |         // message-read mode
            PIPE_WAIT,                      // blocking mode
            PIPE_UNLIMITED_INSTANCES,       // max. instances
            BUFFER_SIZE,                    // output buffer size
            BUFFER_SIZE,                    // input buffer size
            NMPWAIT_USE_DEFAULT_WAIT,       // client time-out
            NULL                            // default security attribute
        );
```

```
    }

    void HandleClient(HANDLE hPipe, const std::string& clientId) {
        char buffer[BUFFER_SIZE];
        DWORD bytesRead;
        BOOL success;

        while (true) {
            success = ReadFile(
                hPipe,                      // handle to pipe
                buffer,                     // buffer to receive data
                BUFFER_SIZE - 1,            // size of buffer
                &bytesRead,                 // number of bytes read
                NULL                        // not overlapped I/O
            );

            if (!success || bytesRead == 0) {
                if (GetLastError() == ERROR_BROKEN_PIPE) {
                    std::cout << "Client " << clientId << " disconnected" <<
std::endl;
                    WriteToLog(clientId, "Client disconnected");
                }
                else {
                    std::cerr << "ReadFile failed, GLE=" << GetLastError() <<
std::endl;
                }
                break;
            }

            buffer[bytesRead] = '\0';
            std::string message(buffer);

            if (message == "DISCONNECT") {
                std::cout << "Client " << clientId << " disconnected normally"
<< std::endl;
                WriteToLog(clientId, "Client disconnected");
                break;
            }

            WriteToLog(clientId, message);

            // Send response back to client
            std::string response = "ACK: Message received";
            DWORD bytesWritten;
            BOOL fSuccess = WriteFile(
                hPipe,                      // handle to pipe
                response.c_str(),           // buffer to write from
                response.length() + 1,     // number of bytes to write, include
NULL
                &bytesWritten,              // number of bytes written
                NULL                        // not overlapped I/O
            );

            if (!fSuccess) {
                std::cerr << "WriteFile failed, GLE=" << GetLastError() <<
std::endl;
                break;
            }
        }

        FlushFileBuffers(hPipe);
        DisconnectNamedPipe(hPipe);
        CloseHandle(hPipe);
    }
```

```
public:
    LogServer() : clientCounter(0) {
        logFile.open("server_log.txt", std::ios::app);
        if (!logFile.is_open()) {
            std::cerr << "ERROR: Cannot open log file!" << std::endl;
        }
        else {
            std::string timestamp = GetCurrentTimestamp();
            logFile << "\n=== Log Server started at " << timestamp << " ===\n";
            logFile.flush();
        }
    }

    ~LogServer() {
        if (logFile.is_open()) {
            logFile << "=== Log Server stopped ===\n";
            logFile.close();
        }
    }

    void Run() {
        std::cout << "Log Server started. Waiting for clients..." << std::endl;
        WriteToLog("SERVER", "Log server started");

        while (true) {
            HANDLE hPipe = CreateNamedPipeInstance();

            if (hPipe == INVALID_HANDLE_VALUE) {
                std::cerr << "ERROR: CreateNamedPipe failed, GLE=" <<
GetLastError() << std::endl;
                Sleep(2000);
                continue;
            }

            std::cout << "Waiting for client connection..." << std::endl;

            BOOL connected = ConnectNamedPipe(hPipe, NULL);
            if (connected) {
                clientCounter++;
                std::string clientId = std::to_string(clientCounter);
                std::cout << "Client " << clientId << " connected successfully!"
<< std::endl;
                WriteToLog(clientId, "Client connected");

                HandleClient(hPipe, clientId);
            }
            else {
                DWORD error = GetLastError();
                if (error == ERROR_PIPE_CONNECTED) {
                    clientCounter++;
                    std::string clientId = std::to_string(clientCounter);
                    std::cout << "Client " << clientId << " connected (already
connected)!" << std::endl;
                    WriteToLog(clientId, "Client connected");

                    HandleClient(hPipe, clientId);
                }
                else {
                    std::cerr << "ConnectNamedPipe failed, GLE=" << error <<
std::endl;
                    CloseHandle(hPipe);
                }
            }
```

```
        }
    }
};

int main() {
    std::cout << "=== Log Server Application ===" << std::endl;

    LogServer server;
    server.Run();

    return 0;
}
```

Листинг А.2 – Реализация LogClient.cpp

```
#include <iostream>
#include <windows.h>
#include <string>
#include <thread>
#include <chrono>
#include <mutex>

#define PIPE_NAME L"\\\\.\\pipe\\LogServerPipe"
#define BUFFER_SIZE 4096

// Мьютекс для синхронизации вывода
std::mutex cout_mutex;

void SafePrint(const std::string& message) {
    std::lock_guard<std::mutex> lock(cout_mutex);
    std::cout << message << std::endl;
}

class LogClient {
private:
    int clientId;

    bool ConnectWithRetry() {
        for (int attempt = 0; attempt < 3; attempt++) {
            HANDLE hPipe = CreateFile(
                PIPE_NAME,
                GENERIC_READ | GENERIC_WRITE,
                0,
                NULL,
                OPEN_EXISTING,
                0,
                NULL
            );

            if (hPipe != INVALID_HANDLE_VALUE) {
                return true;
            }

            DWORD error = GetLastError();
            if (error == ERROR_PIPE_BUSY) {
                SafePrint("Client " + std::to_string(clientId) + ": Pipe busy,
retrying in 1s... (" +
                    std::to_string(attempt + 1) + "/3)");
                Sleep(1000);
            }
            else {
                SafePrint("Client " + std::to_string(clientId) + ": Could not
open pipe. GLE=" + std::to_string(error));
```

```
                return false;
            }
        }
        SafePrint("Client " + std::to_string(clientId) + ": Failed to connect
after 3 attempts");
        return false;
    }

public:
    LogClient(int id) : clientId(id) {}

    bool SendMessage(const std::string& message) {
        HANDLE hPipe = CreateFile(
            PIPE_NAME,
            GENERIC_READ | GENERIC_WRITE,
            0,
            NULL,
            OPEN_EXISTING,
            0,
            NULL
        );

        if (hPipe == INVALID_HANDLE_VALUE) {
            SafePrint("Client " + std::to_string(clientId) + ": Could not open
pipe. GLE=" + std::to_string(GetLastError()));
            return false;
        }

        // Режим сообщений
        DWORD dwMode = PIPE_READMODE_MESSAGE;
        BOOL fSuccess = SetNamedPipeHandleState(
            hPipe,
            &dwMode,
            NULL,
            NULL
        );

        if (!fSuccess) {
            SafePrint("Client    "    +    std::to_string(clientId)    +    ":
SetNamedPipeHandleState failed. GLE=" + std::to_string(GetLastError()));
            CloseHandle(hPipe);
            return false;
        }

        // Отправляем сообщение
        DWORD cbWritten;
        fSuccess = WriteFile(
            hPipe,
            message.c_str(),
            message.length() + 1,
            &cbWritten,
            NULL
        );

        if (!fSuccess) {
            SafePrint("Client  "  +  std::to_string(clientId)  +  ":  WriteFile
failed. GLE=" + std::to_string(GetLastError()));
            CloseHandle(hPipe);
            return false;
        }

        SafePrint("Client " + std::to_string(clientId) + " sent: " + message);

        // Читаем ответ сервера
```

```
        char buffer[BUFFER_SIZE];
        DWORD cbRead;
        fSuccess = ReadFile(
            hPipe,
            buffer,
            BUFFER_SIZE - 1,
            &cbRead,
            NULL
        );

        if (!fSuccess) {
            DWORD error = GetLastError();
            if     (error     ==     ERROR_BROKEN_PIPE     ||     error     ==
ERROR_PIPE_NOT_CONNECTED) {
                SafePrint("Client  "  +  std::to_string(clientId)  +  ":  Server
disconnected normally");
            }
            else {
                SafePrint("Client " + std::to_string(clientId) + ": ReadFile
failed. GLE=" + std::to_string(error));
            }
        }
        else {
            buffer[cbRead] = '\0';
            SafePrint("Client " + std::to_string(clientId) + " received: " +
buffer);
        }

        CloseHandle(hPipe);
        return true;
    }
};

void DemoClient(int clientId) {
    LogClient client(clientId);
    SafePrint("=== Client " + std::to_string(clientId) + " started ===");

    // Разные клиенты начинают с разной задержкой
    std::this_thread::sleep_for(std::chrono::milliseconds(100 * clientId));

    for (int i = 1; i <= 3; i++) {
        std::string message = "Hello from client " + std::to_string(clientId)
+ " - message " + std::to_string(i);

        if (!client.SendMessage(message)) {
            SafePrint("Client " + std::to_string(clientId) + " failed to send
message");
            break;
        }

        std::this_thread::sleep_for(std::chrono::seconds(2));
    }

    // Отправляем сообщение о дисконнекте
    client.SendMessage("DISCONNECT");
    SafePrint("=== Client " + std::to_string(clientId) + " finished ===");
}

int main() {
    while (true) {
        std::cout << "=== Log Client Demo ===" << std::endl;
        std::cout << "0 - Exit" << std::endl;
        std::cout << "1 - Single client demo" << std::endl;
        std::cout << "2 - Multiple clients demo" << std::endl;
```

```
        std::cout << "Choose mode: ";

        std::string choice;
        std::cin >> choice;

        if (choice.size() != 1) {
            std::cout << "Invalid choice!" << std::endl;
            continue;
        }

        switch (choice[0]) {
        case '0':
            return 0;
        case '1':
            DemoClient(1);
            break;
        case '2': {
            std::thread client1(DemoClient, 1);
            std::thread client2(DemoClient, 2);

            client1.join();
            client2.join();
            break;
        }
        default:
            std::cout << "Invalid choice!" << std::endl;
        }
    }
}
```