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

ОТЧЁТ

к лабораторной работе №3

на тему

**ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ: ОБМЕН ДАННЫМИ**

Выполнил: студент гр.253504   
Севрюков С.И.

Проверил: ассистент кафедры информатики Гриценко Н.Ю.

Минск 2024

СОДЕРЖАНИЕ

[1 Формулировка задачи 3](#_Toc178641666)

[2 Краткие теоритические сведения 4](#_Toc178641667)

[3 Описание функций программы 5](#_Toc178641668)

[Заключение 7](#_Toc178641669)

[Список использованных источников 8](#_Toc178641670)

[Приложение А (обязательное) исходный код 9](#_Toc178641671)

# **1 ФОРМУЛИРОВКА ЗАДАЧИ**

Данная лабораторная работа посвящена изучению методов работы с обеспечением передачи данных между взаимодействующими процессами и совместной их обработки, рассмотреть типичные проблемы, возникающие при организации взаимодействия, и пути их решения.

В рамках задачи необходимо разработать приложение, которое выполняет сохранение сообщений, которые получает сервер от пользователей.

Для реализации задачи будут использоваться механизмы межпроцессного взаимодействия (*IPC*), такие как каналы и т.д.

Цель работы: подходы, системные объекты и функции для обеспечения передачи данных между взаимодействующими процессами и/или совместной их обработки. Типичные проблемы, возникающие при организации взаимодействия, и пути их решения.

# **2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

Преодоление изоляции процессов: в многозадачной системе процессы изолированы друг от друга для безопасности. Однако для обмена данными необходимо использовать механизмы межпроцессного взаимодействия (*IPC*), такие как каналы, разделяемая память и очереди сообщений.

Проблемы согласованного доступа к данным: основная проблема — это обеспечение согласованного и безопасного доступа к данным несколькими процессами. Решается через: копирование данных: данные передаются между процессами с созданием копий, предотвращая одновременные изменения; синхронизация (взаимное исключение): использование мьютексов, семафоров для контроля доступа к общим ресурсам.

Использование специализированных средств *IPC*:

1 Каналы: двусторонний обмен сообщениями между процессами.

2 Разделяемая память: несколько процессов могут одновременно читать и писать в одну область памяти.

3 Очереди сообщений: процессы могут отправлять и получать сообщения, управляемые системой для синхронизации.

# **3 ОПИСАНИЕ ФУНКЦИЙ ПРОГРАММЫ**

В данной программе реализуются функции для реализации многопользовательского вывода. Программа состоит из нескольких ключевых функций.

В серверной части:

1 *CreateNamedPipe*: создает именованный канал, где *L*"*\\\\*.*\\pipe\\ChatPipe*" — имя канала, *PIPE\_ACCESS\_DUPLEX* — доступ к каналу на чтение и запись, *PIPE\_TYPE\_MESSAGE* | *PIPE\_READMODE\_MESSAGE* | *PIPE\_WAIT* — тип канала, режим чтения и ожидания, *PIPE\_UNLIMITED\_INSTANCES* — неограниченное количество экземпляров, а также размеры буферов для чтения и записи.

2 *ConnectNamedPipe*: ожидает подключения клиента к именованному каналу. Входные параметры: *hPipe* — дескриптор канала,

3 *ReadFile*: читает данные из канала (сообщения от клиента). Входные параметры: *hPipe* — дескриптор канала, буфер для хранения полученных данных, размер буфера, *bytesRead* — количество считанных байтов, *nullptr* — указатель на структуру *OVERLAPPED* (не используется в синхронном режиме).

4 *WriteFile*: записывает данные в канал (если сервер решит отправить ответ). Входные параметры: *hPipe* — дескриптор канала, буфер с данными для записи, размер данных, *bytesWritten* — количество записанных байтов, *nullptr* — указатель на структуру *OVERLAPPED* (не используется).

5 *CloseHandle*: закрывает дескриптор канала после завершения работы с ним.

6 *std*::*ofstream* **(функция** *logMessage***)**: открывает файл для записи сообщений в лог. Входные параметры: имя файла для логирования, режим открытия файла (*std*::*ios*::*app* — добавление данных в конец файла), записывает строку в файл с текущим временем и текстом сообщения.

7 *std*::*chrono*::*system\_clock*::*now* и *std*::*chrono*::*system\_clock*::*to\_time\_t*: получение текущего времени системы для последующего логирования.

8 *std*::*strftime*: форматирует время в строку.

9 *std*::*thread*: создает новый поток для обработки клиента в функции *handleClient*, позволяя серверу одновременно обслуживать несколько клиентов.

В клиентской части:

1 *CreateFile*: открывает соединение с именованным каналом, предоставляя доступ на чтение и запись. Входные параметры: *L*"*\\\\*.*\\pipe\\ChatPipe*" — имя канала, *GENERIC\_READ* | *GENERIC\_WRITE* — доступ на чтение и запись, 0 — нет совместного доступа, *nullptr* — нет атрибутов безопасности, *OPEN\_EXISTING* — открытие существующего канала, 0 — отсутствуют дополнительные атрибуты, *nullptr* — отсутствие шаблона безопасности.

2 *WriteFile*: отправляет сообщение на сервер через именованный канал. Входные параметры: *hPipe* — дескриптор канала, буфер с данными для отправки, размер данных, *bytesWritten* — количество записанных байтов, *nullptr* — указатель на структуру *OVERLAPPED* (не используется).

3 *std*::*cin*.*getline*: читает ввод пользователя (имя и текст сообщения) с консоли. Входные параметры: буфер для хранения строки, максимальный размер буфера.

4 *CloseHandle*: закрывает дескриптор канала после завершения работы с ним.

Результат работы программы на рисунке 3.1.

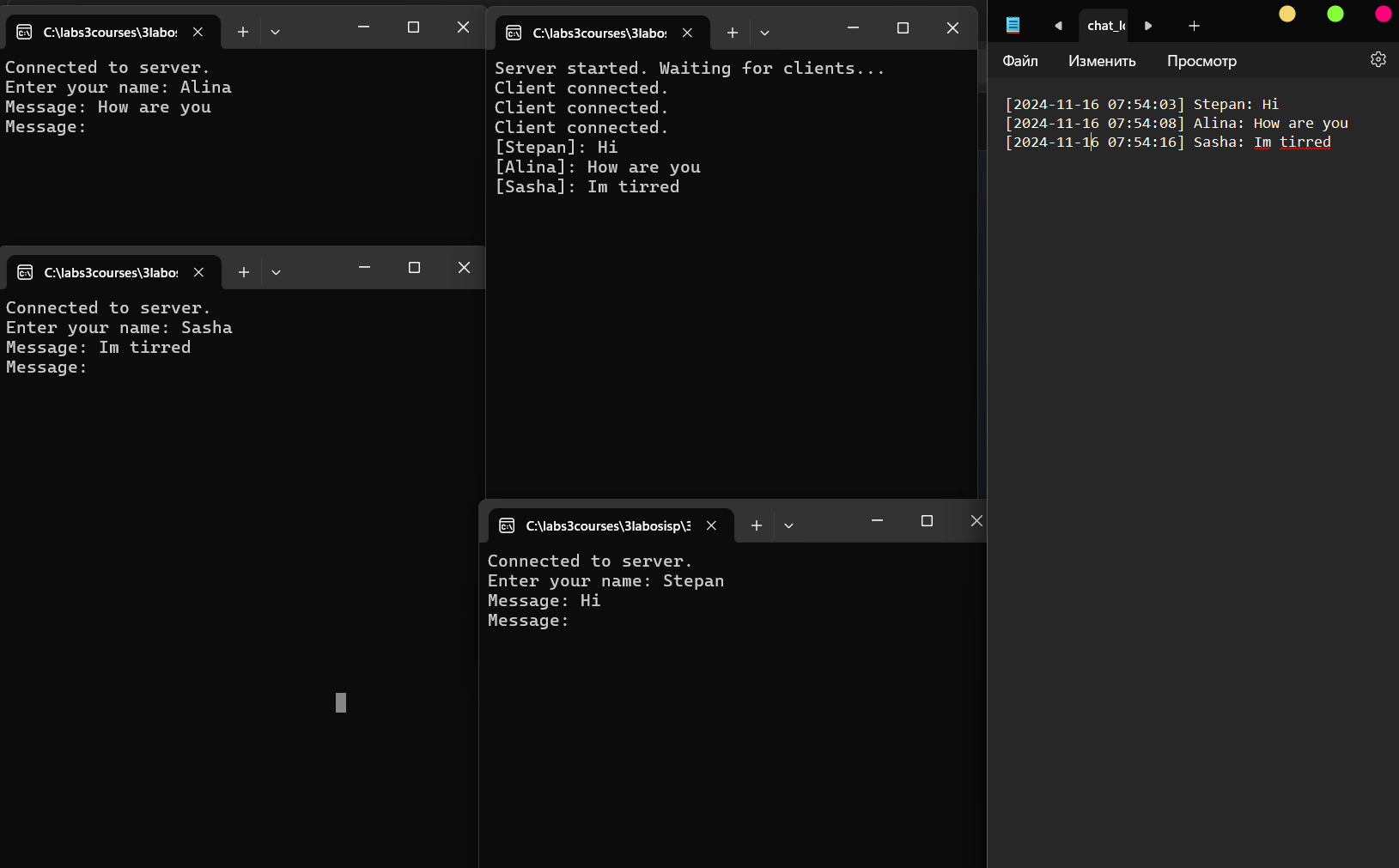


Рисунок 3.1 – Результат программы

На рисунке видно, что сервер получает сообщения от пользователей, а затем логирует их с отметкой времени и пользователем, который сообщение отправил.

# **ЗАКЛЮЧЕНИЕ**

В ходе разработки программы, обеспечивающей обмен сообщениями между клиентом и сервером с использованием *IPC* через именованные каналы, были продемонстрированы ключевые принципы работы с такими механизмами в операционной системе *Windows*. Использование именованных каналов для организации двухсторонней связи между процессами позволяет эффективно передавать данные между ними, что является важным элементом для создания многозадачных приложений.

Программа показала, что при реализации многозадачности с использованием потоков можно одновременно обслуживать несколько клиентов, не блокируя основной процесс сервера.

Логирование сообщений с временными метками помогает отслеживать активность и делает систему более прозрачной для анализа.

Таким образом, программа демонстрирует базовые принципы организации общения между процессами, использование многозадачности и синхронного ввода-вывода. Для более сложных и высоконагруженных систем, возможно, потребуется применение асинхронных методов или других подходов для улучшения производительности и масштабируемости. Тем не менее, полученные результаты подчеркивают важность правильного выбора технологий для реализации межпроцессного взаимодействия в зависимости от требований к системе и её нагрузке.

# **СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

[1] Interprocess communications [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/enus/windows/win32/ipc/interprocess-communications

# **ПРИЛОЖЕНИЕ А** **(обязательное)** **Исходный код**

#define UNICODE

#include <windows.h>

#include <iostream>

#include <string>

struct Message {

char sender[32];

char text[256];

};

int main() {

HANDLE hPipe;

Message message;

DWORD bytesWritten;

// Подключение к именованному каналу

hPipe = CreateFile(

L"\\\\.\\pipe\\ChatPipe",

GENERIC\_READ | GENERIC\_WRITE,

0,

nullptr,

OPEN\_EXISTING,

0,

nullptr);

if (hPipe == INVALID\_HANDLE\_VALUE) {

std::cerr << "Failed to connect to server. Error: " << GetLastError() << std::endl;

return 1;

}

std::cout << "Connected to server.\n";

std::cout << "Enter your name: ";

std::cin.getline(message.sender, sizeof(message.sender));

while (true) {

std::cout << "Message: ";

std::cin.getline(message.text, sizeof(message.text));

// Отправка сообщения на сервер

if (!WriteFile(hPipe, &message, sizeof(Message), &bytesWritten, nullptr)) {

std::cerr << "Failed to write to pipe. Error: " << GetLastError() << std::endl;

break;

}

}

CloseHandle(hPipe);

}

#define UNICODE

#include <windows.h>

#include <iostream>

#include <string>

#include <thread>

#include <vector>

#include <fstream>

#include <chrono>

#include <ctime>

// Структура для хранения сообщения

struct Message {

char sender[32];

char text[256];

};

// Функция для логирования сообщений в файл

void logMessage(const std::string& sender, const std::string& text) {

// Получаем текущее время

auto now = std::chrono::system\_clock::now();

std::time\_t time = std::chrono::system\_clock::to\_time\_t(now);

char timeStr[100];

std::strftime(timeStr, sizeof(timeStr), "%Y-%m-%d %H:%M:%S", std::localtime(&time));

// Открываем файл для добавления сообщений

std::ofstream logFile("chat\_log.txt", std::ios::app);

if (logFile.is\_open()) {

logFile << "[" << timeStr << "] " << sender << ": " << text << std::endl;

}

}

// Функция для обработки сообщений от клиента

void handleClient(HANDLE hPipe) {

Message message;

DWORD bytesRead;

while (true) {

// Чтение сообщения от клиента

if (ReadFile(hPipe, &message, sizeof(Message), &bytesRead, nullptr) && bytesRead > 0) {

std::string sender(message.sender);

std::string text(message.text);

// Выводим сообщение на экран

std::cout << "[" << sender << "]: " << text << std::endl;

// Логируем сообщение в файл

logMessage(sender, text);

}

else {

if ((GetLastError() == 109) || (GetLastError() == 0))

std::cout << "Client disconnected" << std::endl;

else

std::cerr << "Client disconnected or read error. Error: " << GetLastError() << std::endl;

break;

}

}

CloseHandle(hPipe);

}

int main() {

HANDLE hPipe;

std::vector<std::thread> clientThreads;

std::cout << "Server started. Waiting for clients...\n";

while (true) {

// Создаем именованный канал

hPipe = CreateNamedPipe(

L"\\\\.\\pipe\\ChatPipe",

PIPE\_ACCESS\_DUPLEX,

PIPE\_TYPE\_MESSAGE | PIPE\_READMODE\_MESSAGE | PIPE\_WAIT,

PIPE\_UNLIMITED\_INSTANCES,

sizeof(Message),

sizeof(Message),

0,

nullptr);

if (hPipe == INVALID\_HANDLE\_VALUE) {

std::cerr << "Failed to create named pipe. Error: " << GetLastError() << std::endl;

return 1;

}

// Ожидаем подключения клиента

if (!ConnectNamedPipe(hPipe, nullptr)) {

std::cerr << "Failed to connect to client. Error: " << GetLastError() << std::endl;

CloseHandle(hPipe);

continue;

}

std::cout << "Client connected.\n";

// Создаем поток для обработки клиента

clientThreads.emplace\_back(handleClient, hPipe);

}

// Ждем завершения всех потоков перед завершением работы сервера

for (auto& thread : clientThreads) {

if (thread.joinable()) {

thread.join();

}

}

return 0;

}