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

Фроленко К.Ю.

Проверил: ассистент кафедры информатики Гриценко Н.Ю.

Минск 2024

СОДЕРЖАНИЕ

[1 Формулировка задачи 3](#_Toc179678267)

[2 Краткие теоритические сведения 4](#_Toc179678268)

[3 Описание функций программы 5](#_Toc179678269)

[3.1 Компонент server.cpp 5](#_Toc179678270)

[3.1.1 Функция CurrentTimestamp 5](#_Toc179678271)

[3.1.2 Функция HandleClient 5](#_Toc179678272)

[3.2 Компонент client.cpp 6](#_Toc179678273)

[3.2.1 Функция ClientThread 6](#_Toc179678274)

[3.2.2 Функция main 6](#_Toc179678275)

[4 Пример выполнения программы 7](#_Toc179678276)

[4.1 Запуск программы и процесс выполнения 7](#_Toc179678277)

[4.2 Описание работы и результатов 9](#_Toc179678278)

[Вывод 9](#_Toc179678279)

[Список использованных источников 10](#_Toc179678280)

[Приложение А (обязательное) 11](#_Toc179678281)

[Приложение Б (обязательное) 1](#_Toc179678284)4

1 ФОРМУЛИРОВКА ЗАДАЧИ

Целью моей лабораторной работы является разработка и детальный анализ многопользовательского сервера логирования, который демонстрирует использование именованных каналов для организации межпроцессного взаимодействия в операционной системе Windows. Исследование влияния различных механизмов IPC (Inter-Process Communication) на производительность и надежность информационных систем является актуальной задачей в сфере компьютерных наук. Особенно значимым это становится в условиях стремительного роста объемов данных и требований к скорости их обработки.

Цель работы заключается в том, чтобы исследовать и сравнить эффективность использования именованных каналов по сравнению с другими методами IPC, а также выявить преимущества мультиплексированного подхода при обработке входящих данных от множества клиентов. Результаты данного исследования помогут лучше понять, как выбор метода IPC может влиять на общую производительность и масштабируемость систем.

В рамках лабораторной работы предполагается выполнение следующих задач:

1 Реализация сервера логирования с использованием именованных каналов: на первом этапе необходимо создать и запустить сервер, который сможет принимать сообщения от множества клиентов асинхронно. Этот этап позволит оценить преимущества;

2 Обеспечение упорядоченной и синхронизированной записи в лог-файл: следующим шагом будет настройка механизма записи входящих сообщений с сохранением порядка их поступления и прикрепления к каждому сообщению временной метки и идентификатора источника. Этот метод будет анализироваться на предмет его надежности и способности обеспечить точную историю событий.

Эти этапы работы позволят не только глубоко изучить возможности и особенности межпроцессного взаимодействия в современных операционных системах, но и получить практический опыт в создании надежных и эффективных серверных приложений для обработки данных в многопользовательской среде. Разработанный сервер логирования может найти применение в различных областях, включая корпоративные информационные системы, телекоммуникации, финансовые технологии и другие сектора, где критически важна точность и скорость обработки больших объемов данных.

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

В рамках разработки многопользовательского сервера логирования и соответствующих клиентских приложений были реализованы следующие ключевые компоненты:

– компонент server.cpp;

– компонент client.cpp.

3.1 Компонент server.cpp

Компонент server.cpp реализует сервер логирования, который обрабатывает асинхронные запросы от множества клиентов. Основная задача сервера — принимать данные, записывать их в лог-файл и предоставлять информацию о времени поступления сообщений и идентификаторе источника.

В компоненте server.cpp реализованы следующие функции:

– функция CurrentTimestamp;

– функция HandleClient;

– функция main.

3.1.1 Функция CurrentTimestamp

Функция CurrentTimestamp предназначена для генерации текущей временной метки, которая используется для маркировки каждого сообщения, записываемого в лог-файл. Функция совершает следующие действия:

1 Получение текущего времени: используется функция time(nullptr) для получения текущего времени в секундах;

2 Преобразование в локальное время: с помощью функции localtime\_s, текущее время преобразуется в локальное, учитывая часовой пояс системы;

3 Форматирование времени: Строка времени форматируется в формат "YYYY-MM-DD HH:MM" с использованием функции strftime.

3.1.2 Функция HandleClient

Функция HandleClient обрабатывает входящие данные от подключенных клиентов и записывает их в лог-файл с временной меткой и идентификатором. Функция выполняет следующие действия:

1 Чтение данных из канала: данные читаются из именованного канала hPipe с использованием функции ReadFile.

2 Добавление временной метки: к каждому прочитанному сообщению добавляется временная метка, полученная вызовом CurrentTimestamp.

3 Запись в лог-файл: модифицированное сообщение записывается в файл server.log с синхронизацией доступа через мьютекс log\_mutex.

4 Вывод сообщения: принятое сообщение выводится в стандартный вывод для информации.

3.2 Компонент client.cpp

Компонент client.cpp включает функции для отправки сгенерированных сообщений серверу для их последующего протоколирования. Эти функции обеспечивают установление соединения с сервером, отправку данных и обработку возможных ошибок при передаче.

В компоненте server.cpp реализованы следующие функции:

– функция ClientThread;

– функция main.

3.2.1 Функция ClientThread

Функция ClientThread предназначена для подключения к серверу и отправки серии сообщений. Функция совершает следующие действия:

1 Установление соединения: инициализирует подключение к серверу через именованный канал. В случае, если канал занят, функция ожидает его освобождения. Обрабатывает ошибки подключения и информирует пользователя о состоянии подключения.

2 Отправка сообщений: генерирует и отправляет пять последовательных сообщений. Каждое сообщение маркируется уникальным номером клиента и номером сообщения. В случае ошибки передачи, выводит соответствующее уведомление.

3 Завершение работы: закрывает соединение после отправки всех сообщений, управляя ресурсами системы корректно.

3.2.2 Функция main

Функция main служит входной точкой для клиентской программы и управляет запуском потоков для множественных клиентов. Функция совершает следующие действия:

1 Запрос количества клиентов: предоставляет пользователю возможность задать количество клиентов, которое определяет число потоков для создания.

2 Запуск потоков: инициирует потоки, каждый из которых запускает выполнение ClientThread.

3 Синхронизация потоков: ожидает завершения всех клиентских потоков, гарантируя, что все сообщения были отправлены перед закрытием программы.

4 ПРИМЕР ВЫПОЛНЕНИЯ ПРОГРАММЫ

4.1 Запуск программы и процесс выполнения

Программа начинается с запуска сервера, который инициализирует лог-файл и ожидает подключения клиентов. Запись данных клиентами и их логирование сервером происходит через именованный канал. На рисунке 4.1 представлены результаты выполнения всех операций на консоли сервера.

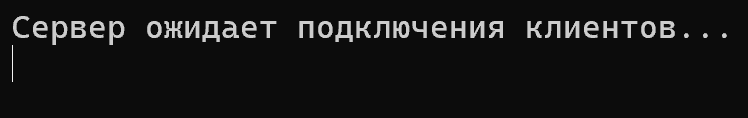


Рисунок 4.1 – Консоль с результатами выполнения сервера

Программа клиента инициируется с вводом количества клиентов. Каждый клиент отправляет серию сообщений серверу, которые затем логируются с временной меткой и идентификатором источника. На рисунке 4.2 показан вывод на консоли клиента с подтверждением отправленных сообщений.

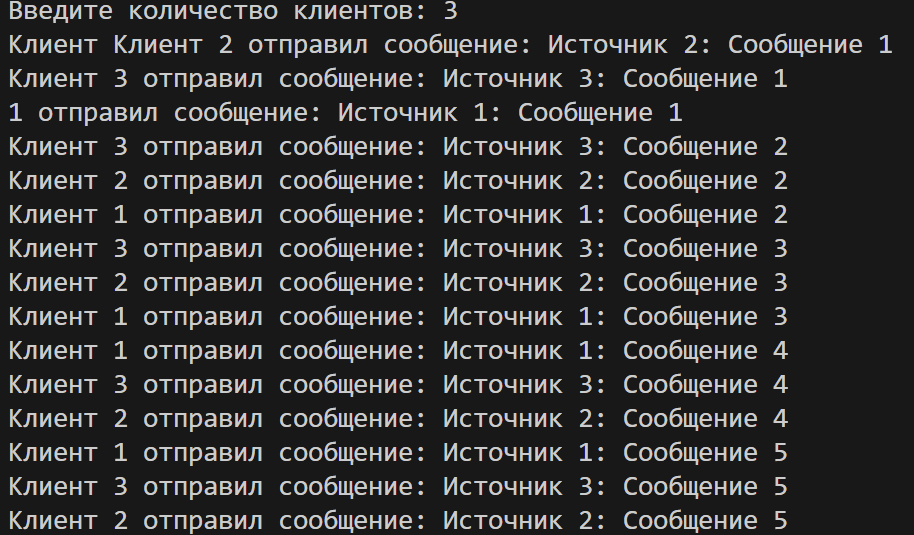


Рисунок 4.2 – Консоль с результатами выполнения клиента

Дополнительно, на рисунке 4.3 показано содержимое файла server.log, отображающее логированные сообщения от клиентов. Это подтверждает успешное протоколирование сообщений с указанием времени получения и источника каждого сообщения.

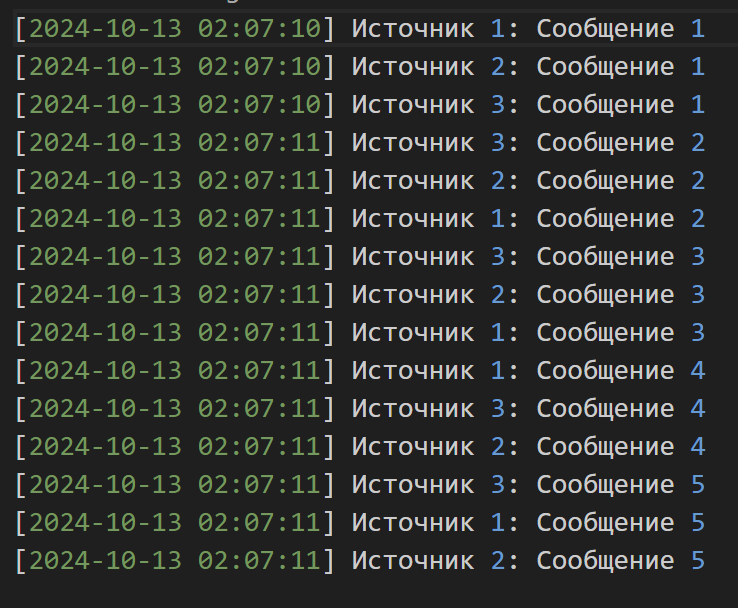


Рисунок 4.3 – Содержимое файла server.log

4.2 Описание работы и результатов

Программа демонстрирует эффективное взаимодействие между сервером и множеством клиентов через именованные каналы, где сервер обрабатывает запросы асинхронно. Каждый клиент отправляет сообщения, которые сервер принимает и записывает в лог-файл с указанием временной метки и идентификатора источника.

Система работает стабильно даже при одновременной обработке нескольких клиентов, подтверждая надёжность и производительность многопоточной архитектуры. Все сообщения корректно логируются в файл server.log, что демонстрирует успешное выполнение программы и правильное функционирование механизма логирования. Использование асинхронных методов позволяет серверу эффективно управлять входящими запросами без задержек и потерь данных.

# ВЫВОД

В ходе выполнения лабораторной работы была успешно реализована система многопользовательского сервера логирования, использующая именованные каналы для организации межпроцессного взаимодействия в операционной системе Windows. Программа продемонстрировала эффективность использования механизмов IPC для обеспечения надежной передачи данных между клиентами и сервером. Сервер логирования корректно обрабатывал поступающие запросы, записывая сообщения в лог-файл с указанием временной метки и идентификатора источника, что позволило точно регистрировать каждое сообщение.

Результаты работы показали, что использование именованных каналов в сочетании с асинхронным подходом к обработке запросов обеспечивает высокую производительность и устойчивость системы при работе с несколькими клиентами одновременно. Сервер поддерживает параллельную обработку входящих сообщений, что позволяет ему эффективно обслуживать множество клиентов без потерь данных и с минимальной задержкой. Асинхронная природа обработки запросов также позволила значительно повысить скорость выполнения операций по сравнению с синхронными подходами, так как сервер не блокировался при выполнении каждой отдельной операции.

Особое внимание было уделено синхронизации доступа к общим ресурсам, в частности к лог-файлу, что обеспечило корректность записей и отсутствие коллизий между клиентами при записи сообщений. Использование мьютексов позволило избежать возможных ошибок при многопоточной записи, гарантируя сохранение порядка поступления данных и их целостность.

Таким образом, разработанная система демонстрирует важные преимущества мультиплексирования и асинхронного взаимодействия в контексте межпроцессного обмена данными. Реализованный сервер логирования может служить основой для более сложных систем, требующих параллельной обработки данных и высокой производительности. Применение такого подхода актуально для различных областей, включая телекоммуникации, мониторинг систем и корпоративные приложения, где важна точность и надежность передачи данных между процессами.

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: 13.10.2024.

[2] Understanding IPC Mechanisms in Windows [Электронный ресурс]. – Режим доступа: https://www.codeproject.com/Articles/34073/Understanding-IPC-Mechanisms-in-Windows. – Дата доступа: 13.10.2024.

[3] Windows Programming: Inter-Process Communication [Электронный ресурс]. – Режим доступа: https://www.tenouk.com/ModuleJ.html. – Дата доступа: 13.10.2024.

ПРИЛОЖЕНИЕ А

(обязательное)

Исходный код компонента server.cpp

#include <bits/stdc++.h>

#include <windows.h>

using namespace std;

#define PIPE\_NAME R"(\\.\pipe\LogPipe)"

bool running = true;

mutex log\_mutex;

ofstream logfile;

string CurrentTimestamp()

{

    time\_t now = time(nullptr);

    tm localTime;

    localtime\_s(&localTime, &now);

    char buffer[64];

    strftime(buffer, sizeof(buffer), "%Y-%m-%d %H:%M:%S", &localTime);

    return buffer;

}

void HandleClient(HANDLE hPipe)

{

    char buffer[1024];

    DWORD bytesRead;

    while (true)

    {

        BOOL success = ReadFile(hPipe, buffer, sizeof(buffer) - 1, &bytesRead, NULL);

        if (!success || bytesRead == 0)

        {

            break;

        }

        buffer[bytesRead] = '\0';

        string timestamp = CurrentTimestamp();

        {

            lock\_guard<mutex> lock(log\_mutex);

            logfile << "[" << timestamp << "] " << buffer << endl;

            logfile.flush();

        }

        cout << "Получено сообщение: " << buffer << endl;

    }

    CloseHandle(hPipe);

}

int main()

{

    logfile.open("server.log", ios::out);

    if (!logfile.is\_open())

    {

        cerr << "Ошибка при открытии файла журнала." << endl;

        return 1;

    }

    HANDLE hPipe;

    vector<thread> client\_threads;

    cout << "Сервер ожидает подключения клиентов..." << endl;

    while (running)

    {

        hPipe = CreateNamedPipeA(

            PIPE\_NAME,

            PIPE\_ACCESS\_DUPLEX,

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

            PIPE\_UNLIMITED\_INSTANCES,

            1024, 1024,

            0,

            NULL);

        if (hPipe == INVALID\_HANDLE\_VALUE)

        {

            cerr << "Ошибка при создании именованного канала: " << GetLastError() << endl;

            return 1;

        }

        BOOL connected = ConnectNamedPipe(hPipe, NULL) ? TRUE : (GetLastError() == ERROR\_PIPE\_CONNECTED);

        if (connected)

        {

            client\_threads.emplace\_back(HandleClient, hPipe);

        }

        else

        {

            CloseHandle(hPipe);

        }

    }

    for (auto &t : client\_threads)

    {

        if (t.joinable())

        {

            t.join();

        }

    }

    logfile.close();

    return 0;

}

ПРИЛОЖЕНИЕ Б

(обязательное)

Исходный код компонента client.cpp

#include <bits/stdc++.h>

#include <windows.h>

#define ll long long

using namespace std;

#define PIPE\_NAME R"(\\.\pipe\LogPipe)"

void ClientThread(ll source\_id)

{

    HANDLE hPipe;

    DWORD bytesWritten;

    while (true)

    {

        hPipe = CreateFileA(

            PIPE\_NAME,

            GENERIC\_WRITE,

            0,

            NULL,

            OPEN\_EXISTING,

            0,

            NULL);

        if (hPipe != INVALID\_HANDLE\_VALUE)

        {

            break;

        }

        if (GetLastError() != ERROR\_PIPE\_BUSY)

        {

            cerr << "Клиент " << source\_id << ": Не удалось подключиться к серверу." << endl;

            return;

        }

        if (!WaitNamedPipeA(PIPE\_NAME, 1000))

        {

            cerr << "Клиент " << source\_id << ": Сервер не отвечает." << endl;

            return;

        }

    }

    for (ll i = 0; i < 5; ++i)

    {

        string message = "Источник " + to\_string(source\_id) + ": Сообщение " + to\_string(i + 1);

        BOOL result = WriteFile(

            hPipe,

            message.c\_str(),

            static\_cast<DWORD>(message.size()),

            &bytesWritten,

            NULL);

        if (!result)

        {

            cerr << "Клиент " << source\_id << ": Ошибка при отправке сообщения." << endl;

        }

        else

        {

            cout << "Клиент " << source\_id << " отправил сообщение: " << message << endl;

        }

        this\_thread::sleep\_for(chrono::milliseconds(100));

    }

    CloseHandle(hPipe);

}

int main()

{

    ll num\_clients;

    cout << "Введите количество клиентов: ";

    cin >> num\_clients;

    vector<thread> threads;

    for (ll i = 0; i < num\_clients; ++i)

    {

        threads.emplace\_back(ClientThread, i + 1);

    }

    for (auto &t : threads)

    {

        t.join();

    }

    return 0;

}