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 Сапроненко В. В.

Проверил: ассистент кафедры информатики Гриценко Н.Ю.

Минск 2024

**СОДЕРЖАНИЕ**

[1 Формулировка задачи 3](#_30j0zll)

[2 Краткие теоретические сведения 4](#_1fob9te)

[3 Описание функций программы 6](#_3znysh7)

[3.1 Запуск сервера и вывод сообщений 6](#_2et92p0)

[3.2 Подключение клиента 6](#_tyjcwt)

З[аключение 8](#_6hzpozfdz8ye)

С[писок использованных источников 9](#_1t3h5sf)

П[риложение А (обязательное) Исходный код программы 10](#_4d34og8)

# 1 ФОРМУЛИРОВКА ЗАДАЧИ

Лабораторная работа посвящена изучению подходов, системных объектов и функций, которые применяются для реализации многопользовательского взаимодействия и логирования данных. Основная цель — исследовать механизмы межпроцессного взаимодействия (IPC), организовать корректную передачу и обработку данных между процессами, а также изучить подходы к обеспечению надежности и упорядоченности данных в условиях многопользовательской среды.

В рамках работы необходимо реализовать программу «Сервер логирования», который будет обрабатывать сообщения от нескольких источников (клиентов). Задача сервера — принимать сообщения, записывать их в файл с соблюдением порядка поступления, добавлять временные метки, идентификаторы источников и форматировать данные согласно заданным правилам. В качестве задачи необходимо создать программу для логирования сообщений клиента и взаимодействие между окнами, используя каналы Windows и клиент-серверной модели. Задача включает в себя следующие этапы:

– создать программу сервера, создающую именованный канал;

– создать программу клиента, подключающегося к каналу;

– создание этапа инициализации клиентов;

– создание функционала для обмена сообщениями между клиентом и сервером

В результате выполнения этой лабораторной работы будут не только получены теоретические знания работы со средствами IPC, но и практический опыт в разработке приложений, которые могут эффективно использовать ресурсы компьютера. Понимание межпроцессной коммуникации станет необходимым для создания высокопроизводительных программных решений.

# 2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

В многозадачных операционных системах, где одновременно выполняется несколько процессов, важной задачей является обеспечение обмена данными между процессами или совместное их использование. Операционные системы создают изоляцию между процессами для защиты их памяти и данных, что в целом обеспечивает безопасность и устойчивость системы, но также создает трудности при необходимости обмена данными между процессами. Для этого разработаны механизмы межпроцессного взаимодействия (IPC, interprocess communication), которые позволяют процессам передавать информацию друг другу или работать с одними и теми же данными[1].

Изоляция процессов — это механизм операционной системы, который предотвращает доступ одного процесса к памяти и данным другого. Этот подход повышает устойчивость системы, так как ошибки и сбои одного процесса не затрагивают другие. Однако в случаях, когда процессы должны взаимодействовать, например, передавать данные или работать с общими ресурсами, изоляцию необходимо частично снять. Для преодоления этой изоляции ОС предоставляет специализированные механизмы IPC, которые обеспечивают безопасный обмен данными между процессами. Эти средства включают в себя различные объекты для передачи данных и организации совместного доступа.

При обмене данными между процессами встает проблема согласованного доступа. Это связано с тем, что одновременное изменение данных несколькими процессами может привести к ошибкам и несогласованности информации. Есть два основных подхода для решения этой проблемы: копирование данных или использование механизмов синхронизации.

Копирование данных — данные передаются от одного процесса к другому по принципу "сделай копию и отправь". Каждый процесс работает со своей копией данных, и изменения в одном процессе не затрагивают данные другого. Этот подход прост, но не всегда эффективен, так как занимает дополнительную память и может быть медленным при большом объеме данных.

Синхронизация — используется для предотвращения одновременного изменения одних и тех же данных разными процессами. Этот метод подразумевает использование механизмов взаимного исключения (мьютексов, семафоров и других). Процессы могут обращаться к данным по очереди, что обеспечивает целостность и согласованность данных[2].

Для организации обмена данными и совместного использования информации операционная система предоставляет несколько механизмов IPC:

Каналы (pipes) — это средства для организации потока данных между двумя процессами. Каналы позволяют передавать данные от одного процесса к другому в форме последовательного потока, который может быть либо однонаправленным (данные передаются только в одну сторону), либо двунаправленным (данные передаются в обе стороны). Каналы удобны для простой передачи данных, но не позволяют совместно использовать память.

Разделяемая память (shared memory) — это область памяти, к которой могут одновременно обращаться несколько процессов. Она создается одним процессом и предоставляется другим для совместного использования. Разделяемая память — самый быстрый метод передачи данных, так как исключает необходимость копирования, но требует синхронизации, чтобы избежать конфликтов при одновременном доступе.

Очереди сообщений (message queues) — это структуры данных, хранящие сообщения, которые процессы могут записывать и считывать. Очереди позволяют передавать блоки данных (сообщения) между процессами, причем порядок сообщений сохраняется. Этот метод удобен для асинхронного обмена данными, так как процессы могут обмениваться сообщениями без необходимости ожидания.

Эти механизмы позволяют эффективно организовать обмен данными и совместный доступ, решая проблемы изоляции и согласованного доступа в многозадачной среде.

Объекты синхронизации — это инструменты, которые обеспечивают согласованный доступ к общим ресурсам, когда несколько процессов или потоков работают одновременно. Они помогают предотвратить ситуации, когда процессы могут одновременно изменить одни и те же данные, что приводит к некорректным результатам или сбоям. Основная задача объектов синхронизации — координировать доступ к ресурсам, чтобы избежать так называемых гонок за ресурс или взаимных блокировок, когда один процесс блокирует ресурс, необходимый другому.

Одним из таких инструментов является мьютекс. Мьютекс действует как «взаимное исключение», разрешая только одному процессу или потоку владеть ресурсом в конкретный момент времени. Если ресурс занят, остальные процессы будут ожидать, пока мьютекс не освободится. Это гарантирует, что доступ к общим данным всегда остается согласованным и предотвращает некорректные изменения.

Семафоры также используются для управления доступом к ресурсам, но работают более гибко, чем мьютексы. Семафор отслеживает количество разрешений для использования ресурса, и если разрешение есть, процесс может его получить и начать работу. Семафоры могут быть как двоичными, так и счетными, что позволяет использовать их в ситуациях, где ресурс может поддерживать несколько одновременных пользователей.

События — ещё один тип объектов синхронизации, который позволяет процессам дожидаться наступления определенного состояния или сигнала перед выполнением дальнейших операций. Событие позволяет одному процессу уведомить другие о завершении какой-то операции или достижении условия, при котором можно продолжить выполнение[3].

# 3 ОПИСАНИЕ ФУНКЦИЙ ПРОГРАММЫ

Согласно формулировке задачи, были спроектированы следующие функции программы :

– запуск одного сервера и клиента;

– подключение клиента к серверу с заданием ID

– отправка сообщения с клиента к серверу;

## 3.1 Запуск сервера и вывод сообщений

Вывод сообщений на сервере производится при подключении клиентов (рисунок 3.1).

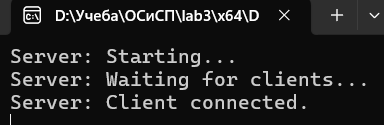


Рисунок 3.1 – Вывод сообщений о подключении

## 3.2 Подключение клиента

При подключении клиента происходит выбор ID (рисунок 3.2). Впоследствии клиент может писать сообщения для отправки на сервер.

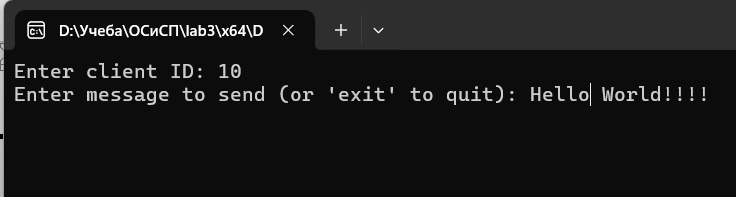


Рисунок 3.2 – Выбор ID клиента и ввод сообщения

Получение сервером сообщения показано на рисунке 3.3.



Рисунок 3.3 – Получение сообщения сервером

Таким образом, было создано программное обеспечение для программы логирования сообщений клиента. Была использована клиент-серверная система. Программа-сервер создает именованный канал и задает ему настройки работы, такие как тип данных «сообщения», дуплексный (двусторонний) режим работы, размер буфера, время ожидания и т. д. Далее производится подключение клиента через экземпляры канала. После подключения производится инициализация, после которого подключение клиентов более невозможно. В конце концов, производится запуск отправки и логирования сообщений.

Программа-клиент имеет простой интерфейс и несложный набор функций для полноценной работы. При запуске клиент автоматически подключается к серверу (если сервер был запущен ранее, иначе вывод ошибки и завершение программы) и дает пользователю выбрать ID. После инициализации можно писать сообщения, которые будут записываться в текстовый документ.

# 

# ЗАКЛЮЧЕНИЕ

В результате выполнения лабораторной работы был разработан «Сервер логирования», который обеспечивает прием и обработку сообщений от источника (процесса-клиента). Сервер реализует запись сообщений в файл с соблюдением порядка поступления, дополнением их временными метками, идентификаторами источников и форматированием в заданной структуре. Это позволило изучить и практически применить механизмы межпроцессного взаимодействия (IPC) в многопользовательской среде.

Процесс разработки продемонстрировал эффективность использования выбранного механизма IPC (например, именованных каналов или очередей сообщений) для организации передачи данных. Были успешно решены задачи синхронизации и координации доступа к общим ресурсам, что предотвратило возможные конфликты и обеспечило согласованный доступ. Дополнительно была реализована поддержка некоторых функций управления журналом, таких как архивирование или перезапись, что улучшило функциональность системы.

Во время работы также выявлены ключевые особенности и ограничения используемых IPC. Например, необходимость тщательного управления потоками данных для предотвращения потери сообщений и обеспечения высокой производительности. Однако, несмотря на эти сложности, созданное приложение показало стабильность работы и корректность протоколирования.

Таким образом, выполненная лабораторная работа позволила углубить понимание работы IPC, продемонстрировала их применимость для решения задач многопользовательского взаимодействия и предоставила практический опыт разработки систем логирования с высоким уровнем надежности и согласованности.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Interprocess communication [Электронный ресурс]. – Режим доступа: http://learn.microsoft.com/en-us/windows/win32/ipc/interprocess-communications

[2] Threading issues in multithreading [Электронный ресурс]. – Режим доступа: https://www.geeksforgeeks.org/threading-issues/#

[3] Synchronization objects [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/en-us/windows/win32/sync/synchronization-objects

# ПРИЛОЖЕНИЕ А

# (обязательное)

# Исходный код программы

#include <iostream>

#include <string>

#include <windows.h>

#define PIPE\_NAME L"\\\\.\\pipe\\LogPipe"

#define BUFFER\_SIZE 1024

void sendMessages(HANDLE pipe, const std::string& clientId) {

while (true) {

std::string message;

std::cout << "Enter message to send (or 'exit' to quit): ";

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

if (message == "exit") {

break;

}

std::string fullMessage = clientId + " " + message;

DWORD bytesWritten;

if (!WriteFile(pipe, fullMessage.c\_str(), (DWORD)(fullMessage.size() + 1), &bytesWritten, NULL)) {

std::cerr << "Error: Could not write to pipe. Code: " << GetLastError() << "\n";

break;

}

}

}

int main() {

Sleep(1000);

HANDLE pipe = CreateFile(

PIPE\_NAME,

GENERIC\_WRITE,

0,

NULL,

OPEN\_EXISTING,

0,

NULL

);

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

std::cerr << "Error: Could not open pipe. Code: " << GetLastError() << "\n";

return 1;

}

std::string clientId;

std::cout << "Enter client ID: ";

std::cin >> clientId;

std::cin.ignore();

sendMessages(pipe, clientId);

CloseHandle(pipe);

return 0;

}

#include <iostream>

#include <fstream>

#include <string>

#include <windows.h>

#include <chrono>

#include <sstream>

#include <iomanip>

#define PIPE\_NAME L"\\\\.\\pipe\\LogPipe"

#define BUFFER\_SIZE 1024

void logMessage(const std::string& message, const std::string& clientId) {

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

if (logFile.is\_open()) {

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

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

char timestamp[20];

std::strftime(timestamp, sizeof(timestamp), "%Y-%m-%d %H:%M:%S", std::localtime(&now\_time));

logFile << "[" << timestamp << "] [Client " << clientId << "] " << message << "\n";

logFile.close();

}

}

int main() {

HANDLE pipe;

char buffer[BUFFER\_SIZE];

DWORD bytesRead;

std::cout << "Server: Starting...\n";

pipe = CreateNamedPipe(

PIPE\_NAME,

PIPE\_ACCESS\_INBOUND,

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

PIPE\_UNLIMITED\_INSTANCES,

BUFFER\_SIZE,

BUFFER\_SIZE,

0,

NULL

);

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

std::cerr << "Error: Could not create named pipe. Code: " << GetLastError() << "\n";

return 1;

}

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

if (ConnectNamedPipe(pipe, NULL) || GetLastError() == ERROR\_PIPE\_CONNECTED) {

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

while (true) {

if (ReadFile(pipe, buffer, BUFFER\_SIZE - 1, &bytesRead, NULL) && bytesRead > 0) {

buffer[bytesRead] = '\0'; // Завершаем строку

std::string message(buffer);

// Извлечение идентификатора клиента

std::string clientId = message.substr(0, message.find(' '));

std::string logMessageContent = message.substr(message.find(' ') + 1);

logMessage(logMessageContent, clientId);

std::cout << "Server: Logged message from " << clientId << "\n";

}

else {

DWORD error = GetLastError();

if (error == ERROR\_BROKEN\_PIPE) {

std::cerr << "Server: Client disconnected. Waiting for a new connection...\n";

// Закрываем старый канал и ждем нового клиента

DisconnectNamedPipe(pipe);

ConnectNamedPipe(pipe, NULL);

}

else if (error == ERROR\_NO\_DATA) {

// Клиент временно не отправляет данные, ждем

Sleep(100);

}

else {

std::cerr << "Error: Could not read from pipe. Code: " << error << "\n";

break;

}

}

}

}

CloseHandle(pipe);

return 0;

}