**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра Вычислительной техники**

**ОТЧЕТ**

**по лабораторной работе №4**

**по дисциплине «Операционные системы»**

**Тема: «Межпроцессное взаимодействие»**

Студента гр. 3312 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Поляков А.И.

Преподаватель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Тимофеев А.В.

Санкт-Петербург

2025

**Цель работы.**

исследовать инструменты и механизмы взаимодействия процессов в Windows.

**Задание 4.2.**

Использование именованных каналов для реализации сетевого межпроцессного взаимодействия.

**Указания к выполнению.**

1. Создайте два консольных приложения с меню (каждая выполняемая функция и/или операция должна быть доступна по отдельному пункту меню), которые выполняют:

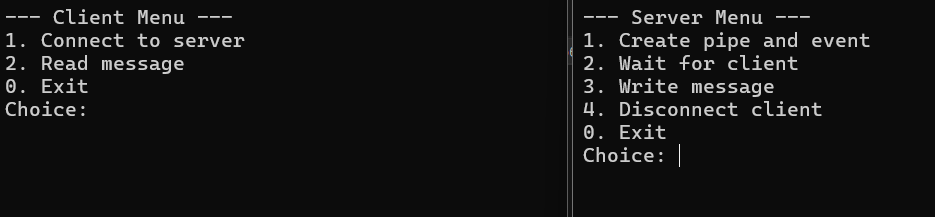
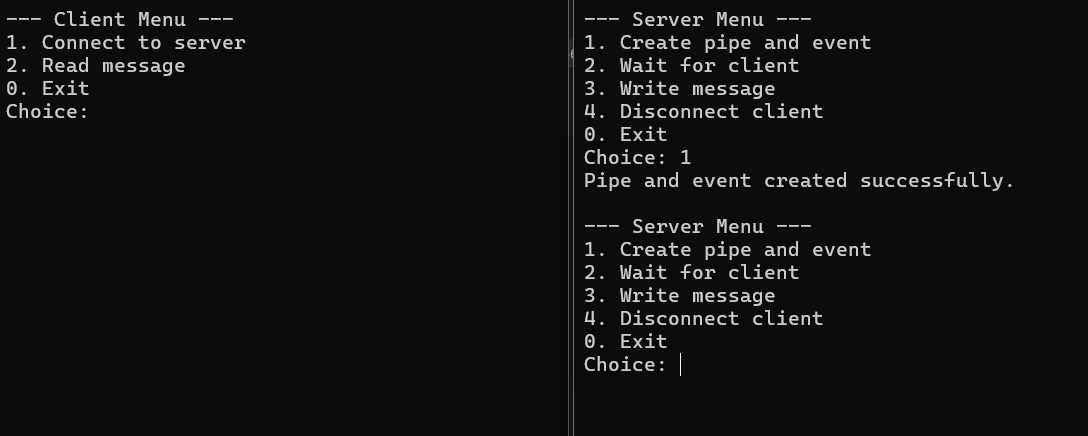
Приложение-сервер создает именованный канал (функция Win32  API – **CreateNamedPipe**), выполняет установление и отключение  соединения (функции Win32 API – **ConnectNamedPipe**,  **DisconnectNamedPipe**), создает объект «событие» (функция Win32  API – **CreateEvent**) осуществляет ввод данных с клавиатуры и их  асинхронную запись в именованный канал (функция Win32 API – **WriteFile**), выполняет ожидание завершения операции  вводавывода (функция Win32 API – **WaitForSingleObject**);

Приложение-клиент подключается к именованному каналу (функция Win32 API – **CreateFile**), в асинхронном режиме считывает содержимое из именованного канала файла (функция Win32 API – **ReadFileEx**) и отображает на экран.

2. Запустите приложения и проверьте обмен данных между процессами. Запротоколируйте результаты в отчет. Дайте свои комментарии в отчете относительно выполнения функций Win32 API.

3. Подготовьте итоговый отчет с развернутыми выводами по заданию.

**Пример выполнения программы**

****

На стороне сервера (справа):

Выбор 1. Create pipe and event

* Пользователь вызвал функцию CreatePipeAndEvent(), которая:

CreateNamedPipeW(...)

Создаёт именованный канал (named pipe) с именем:

L"\\\\.\\pipe\\MyPipe"

Этот путь указывает на локальную машину (\\\\.\\) и имя канала (MyPipe). Ключевые параметры:

* PIPE\_ACCESS\_OUTBOUND | FILE\_FLAG\_OVERLAPPED: позволяет запись и использование асинхронных операций.
* PIPE\_TYPE\_MESSAGE | PIPE\_READMODE\_MESSAGE | PIPE\_WAIT: работает в режиме сообщений, что обеспечивает чёткое разделение переданных блоков.
* Буферы ввода/вывода по 1024 байта.
* Один инстанс канала.

Если всё прошло успешно, дескриптор канала сохраняется в переменной hPipe.

CreateEventW(...)

Создаёт событие синхронизации, которое будет использоваться в OVERLAPPED-структуре для асинхронных операций, например, ожидания подключения клиента или завершения записи:

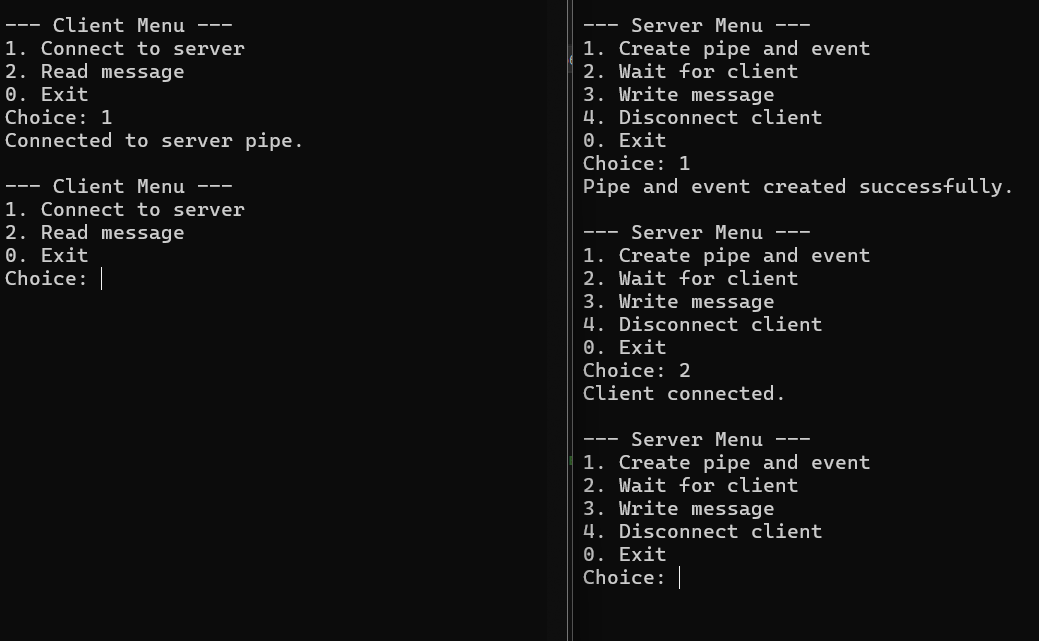
* TRUE — manual reset.
* FALSE — изначально не установленное состояние.

Если событие создано — оно привязывается к overlapped.hEvent.

На стороне клиента (слева):

Пока клиент просто запущен и ожидает действий от пользователя. Он ещё не подключён к каналу. Как видно:

* Отображено основное меню с опциями:
  + Подключиться к серверу.
  + Прочитать сообщение.
  + Выход.

****

После запуска клиента, была выбрана опция "1. Connect to server", в результате чего вызвана функция CreateFileW. Эта функция подключается к ранее созданному именованному каналу \\.\pipe\MyPipe. Клиент использует только чтение, что соответствует архитектуре.

На серверной стороне для ожидания подключения используется ConnectNamedPipe с асинхронной моделью через OVERLAPPED. В случае асинхронного ожидания используется WaitForSingleObject, ожидающий сигнала события hEvent.

Успешное подключение подтверждается сообщениями:

* Клиент: "Connected to server pipe."
* Сервер: "Client connected."

****

На стороне сервера:

Выбор 3. Write message

Выполняется функция WriteToClient(). Она отвечает за отправку текста, введённого пользователем, в канал (клиенту).

Ввод сообщения:

Пользователь ввёл:

Hello, World!

(13 байт, включая пробел и восклицательный знак)

Что делает код:

ResetEvent(hEvent);

Сбрасывает событие, чтобы обеспечить корректное ожидание завершения текущей асинхронной операции.

WriteFile(...)

Пытается асинхронно записать данные в канал:

WriteFile(

hPipe,

input.c\_str(),

input.size(),

nullptr,

&overlapped);

GetLastError() и ожидание:

Если WriteFile вернул FALSE, а GetLastError() == ERROR\_IO\_PENDING, то это означает, что операция ещё не завершилась, но выполняется асинхронно.

В этом случае вызывается:

WaitForSingleObject(hEvent, INFINITE);

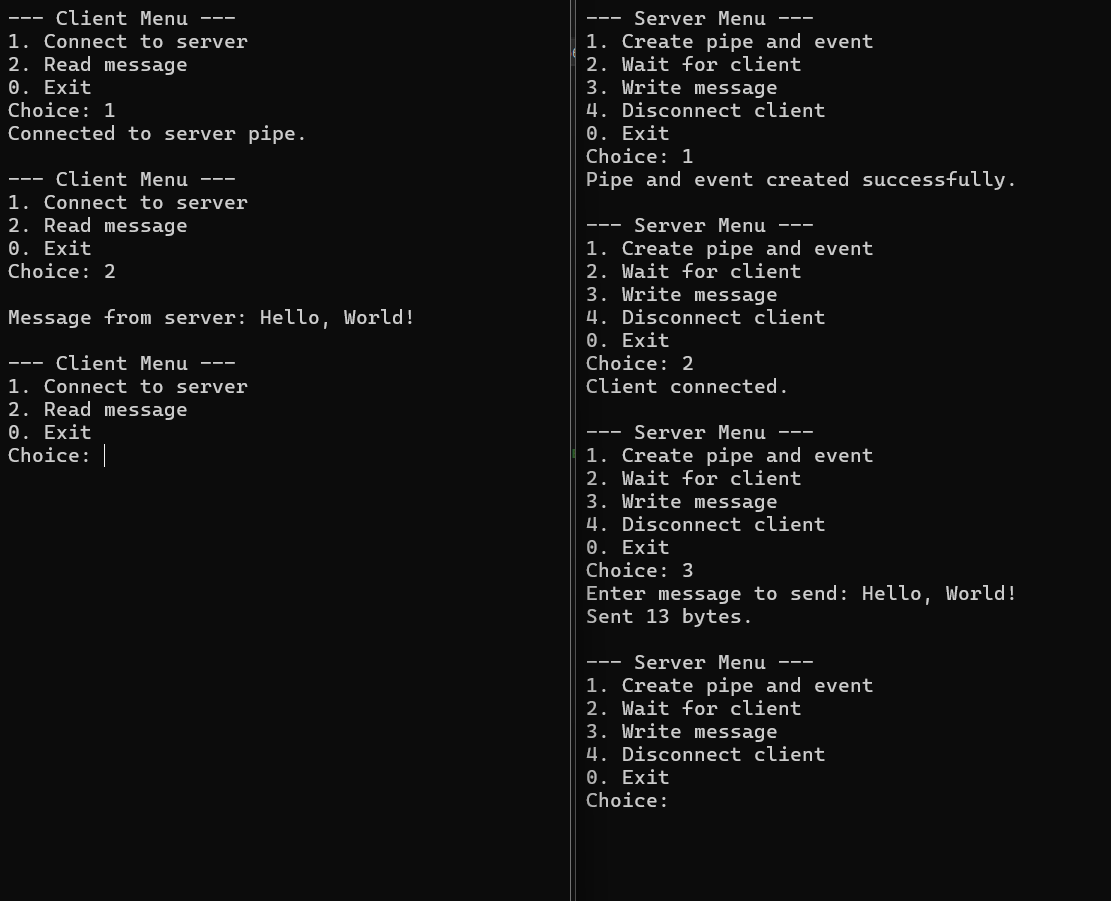
чтобы дождаться завершения.

Затем GetOverlappedResult(...) извлекает количество реально записанных байт:

GetOverlappedResult(hPipe, &overlapped, &bytesWritten, FALSE);

Успешная передача подтверждается сообщением:

"Sent 13 bytes."

****

На стороне клиента:

Выбор 2. Read message

Запускается функция ReadMessage():

Буфер и OVERLAPPED:

char buffer[1024] = {};

OVERLAPPED overlapped = {};

overlapped.hEvent = buffer;

Вместо дескриптора события, как обычно, в поле hEvent записывается указатель на буфер, чтобы потом этот буфер легко получить в колбэке ReadCompleted.

ReadFileEx(...)

Чтение происходит асинхронно:

ReadFileEx(

hPipe,

buffer,

sizeof(buffer),

&overlapped,

ReadCompleted);

Если чтение начинается успешно, программа переходит в "ожидание завершения" с помощью:

SleepEx(INFINITE, TRUE);

* TRUE в аргументе говорит системе: "разрешить выполнение асинхронных завершений (APC)", в частности, функцию ReadCompleted.

Колбэк ReadCompleted:

Когда данные приходят, вызывается:

void CALLBACK ReadCompleted(...) {

std::cout << "\nMessage from server: " << (char\*)lpOverlapped->hEvent << std::endl;

}

* Тут и происходит дешифровка hEvent — мы получаем указатель на буфер, в который пришли данные, и печатаем сообщение:

"Message from server: Hello, World!"

****

После завершения обмена данными сервер вызывает функцию DisconnectNamedPipe, которая разрывает соединение с клиентом, оставляя именованный канал доступным для повторного подключения.

Клиент, не зная о разрыве, выполняет ReadFileEx, однако чтение завершается неудачно. Причиной является отсутствие соединения с сервером: дескриптор существует, но передача данных невозможна.

Вызов ReadFileEx возвращает FALSE, и клиент сообщает об ошибке. Такой результат демонстрирует корректное поведение API в условиях завершения соединения и подчеркивает важность проверки состояния канала перед чтением.

Текст программы

#include "PIWinAPI.h"

#include "Omp.h"

int main()

{

int threadsCount[] = { 1,2,4,8,12,16 };

for (auto i : threadsCount)

{

std::cout << "Threads: " << i << std::endl;

WinAPIThreads(i);

}

std::cout << std::endl << std::endl;

for (auto i : threadsCount)

{

std::cout << "Threads: " << i << std::endl;

OMP(i);

}

}

*PIWinAPI.срр*

#include "PIWinAPI.h"

constexpr int N = 10000000;

constexpr int STUDENT\_ID = 331223;

constexpr int BLOCK\_SIZE = 10 \* STUDENT\_ID;

double pi = 0.0;

LONG current\_index = 0;

int total\_blocks = (N + BLOCK\_SIZE - 1) / BLOCK\_SIZE;

HANDLE hMutex;

int thread\_count;

DWORD WINAPI PiWorker(LPVOID lpParam)

{

int thread\_id = \*(int\*)lpParam;

while (true)

{

int block\_index = InterlockedIncrement(&current\_index) - 1;

if (block\_index >= total\_blocks) break;

int start = block\_index \* BLOCK\_SIZE;

int end = min(N, start + BLOCK\_SIZE);

double local\_sum = 0.0;

for (int i = start; i < end; ++i)

local\_sum += ((i % 2 == 0 ? 1.0 : -1.0) / (2 \* i + 1));

WaitForSingleObject(hMutex, INFINITE);

pi += local\_sum;

ReleaseMutex(hMutex);

}

return 0;

}

int WinAPIThreads(int thread\_count)

{

std::vector<HANDLE> threads;

std::vector<int> thread\_ids;

hMutex = CreateMutex(NULL, FALSE, NULL);

auto start\_time = std::chrono::high\_resolution\_clock::now();

for (int i = 0; i < thread\_count; ++i)

{

thread\_ids.push\_back(i);

HANDLE hThread = CreateThread(NULL, 0, PiWorker, &thread\_ids[i], 0, NULL);

threads.push\_back(hThread);

}

WaitForMultipleObjects(thread\_count, threads.data(), TRUE, INFINITE);

for (auto h : threads)

CloseHandle(h);

CloseHandle(hMutex);

auto end\_time = std::chrono::high\_resolution\_clock::now();

std::cout.precision(15);

std::cout << "Computed PI = " << 4 \* pi << std::endl;

std::cout << "Time: " << std::chrono::duration<double>(end\_time - start\_time).count() << " seconds" << std::endl;

return 0;

}

*ОMP.срр*

#include "Omp.h"

#include <omp.h>

#include <iostream>

constexpr int N = 10000000;

constexpr int STUDENT\_ID = 331223;

constexpr int BLOCK\_SIZE = 10 \* STUDENT\_ID;

int OMP(int thread\_count)

{

double pi = 0.0;

double start = omp\_get\_wtime();

#ifdef \_OPENMP

#pragma omp parallel for num\_threads(thread\_count) schedule(static) reduction(+:pi)

for (int i = 0; i < N; ++i)

pi += (i % 2 == 0 ? 1.0 : -1.0) / (2 \* i + 1);

#else

std::cerr << "OpenMP is not enabled. Please enable OpenMP in your compiler settings.\n";

return -1;

#endif

double end = omp\_get\_wtime();

std::cout.precision(15);

std::cout << "Computed PI = " << 4 \* pi << "\n";

std::cout << "Time: " << (end - start) << " seconds\n";

#pragma omp parallel

{

#pragma omp single

std::cout << "Used threads: " << omp\_get\_num\_threads() << std::endl;

}

return 0;

}