МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра Вычислительной техники

ОТЧЁТ

по лабораторной работе №4

по дисциплине «Операционные системы»

Тема: Потоки и процессы

Студент гр.2307 _____ Подберёзский А.Д.

Преподаватель _____ Тимофеев А.В.

Санкт-Петербург,
2024

Введение

Тема работы: Межпроцессорное взаимодействие

Цель работы: Исследовать инструменты и механизмы взаимодействия процессов в Windows.

Задание 4.1. Реализация решения задачи о читателях-писателях. Указания к выполнению.

1. Выполнить решение задачи о читателях-писателях, для чего необходимо разработать консольные приложения «Читатель» и «Писатель»:

- одновременно запущенные экземпляры процессов-читателей и процессов-писателей должны совместно работать с буферной памятью в виде проецируемого файла:
  - о размер страницы буферной памяти равен размеру физической страницы оперативной памяти;
  - о число страниц буферной памяти равно сумме цифр в номере студенческого билета без учета первой цифры.
- страницы буферной памяти должны быть заблокированы в оперативной памяти (функция VirtualLock);
- длительность выполнения процессами операций «чтения» и «записи» задается случайным образом в диапазоне от 0,5 до 1,5 сек.;
- для синхронизации работы процессов необходимо использовать объекты синхронизации типа «семафор» и «мьютекс»;
- процессы-читатели и процессы-писатели ведут свои журнальные файлы, в которые регистрируют переходы из одного «состояния» в другое (начало ожидания, запись или чтение, переход к освобождению) с указанием кода времени (функция TimeGetTime).

Для состояний «запись» и «чтение» необходимо также запротоколировать номер рабочей страницы.

2. Запустите приложения читателей и писателей, суммарное количество одновременно работающих читателей и писателей должно быть не менее числа страниц буферной памяти. Проверьте функционирование приложений, проанализируйте журнальные файлы процессов, постройте сводные графики смены «состояний» для не менее 5 процессов-читателей и 5 процессов-писателей, дайте свои комментарии относительно переходов процессов из одного состояния в другое. Постройте графики занятости страниц буферной памяти (проецируемого файла) во времени, дайте свои комментарии.

3. Подготовьте итоговый отчет с развернутыми выводами по заданию.

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 ввода-вывода (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.

Задание 1:

Результаты выполнения программы

Далее будут приведены примеры логов и графики состояния читателей и писателей, в которых различные состояния потокоа обозначены целыми числами:

0. Ожидание

1. Чтение записись

2. Освобождение

В файле логов следующие колонки:

1. Состояние читателя или писателя

2. Время

3. Страница (-1, если не должна быть указана)

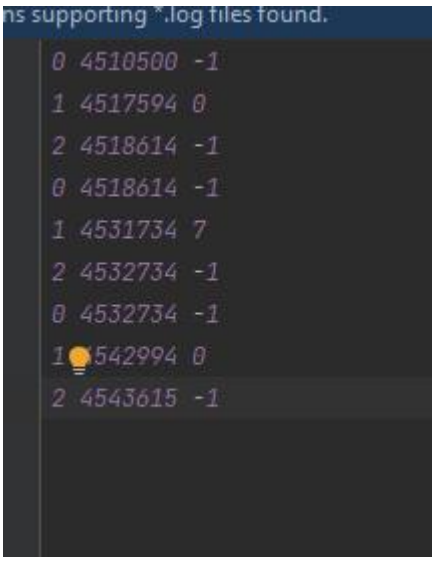

рис.1 Пример логов

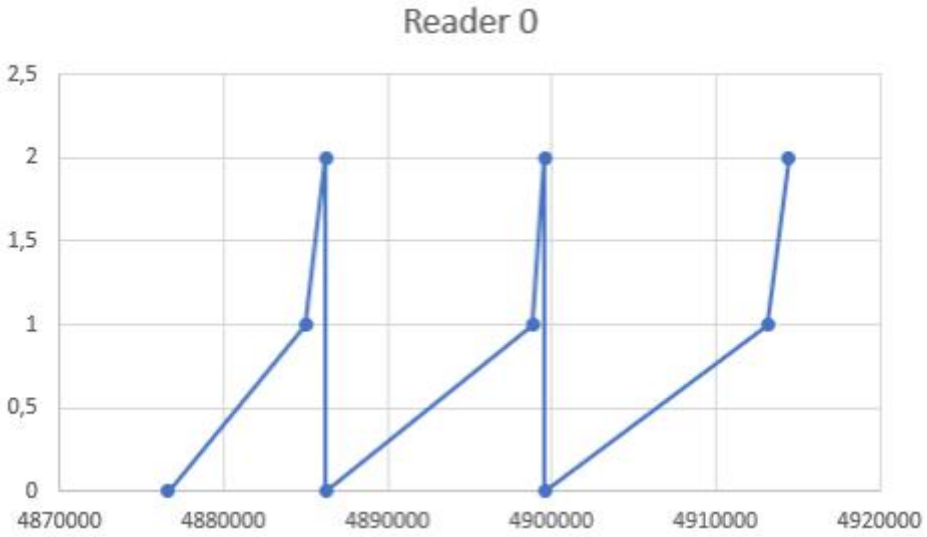

рис.2. График состояний одного из читателей

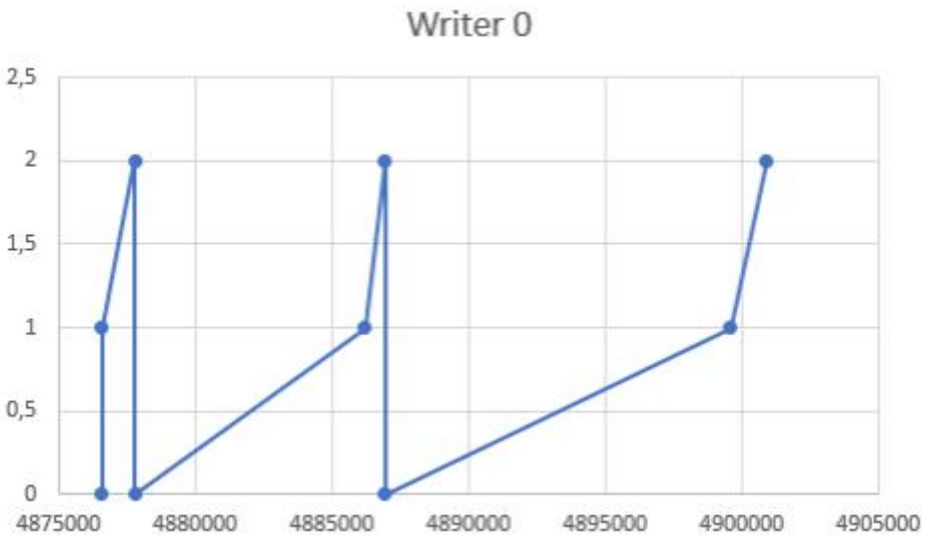

Рис.3 График состояний 1 из писателей

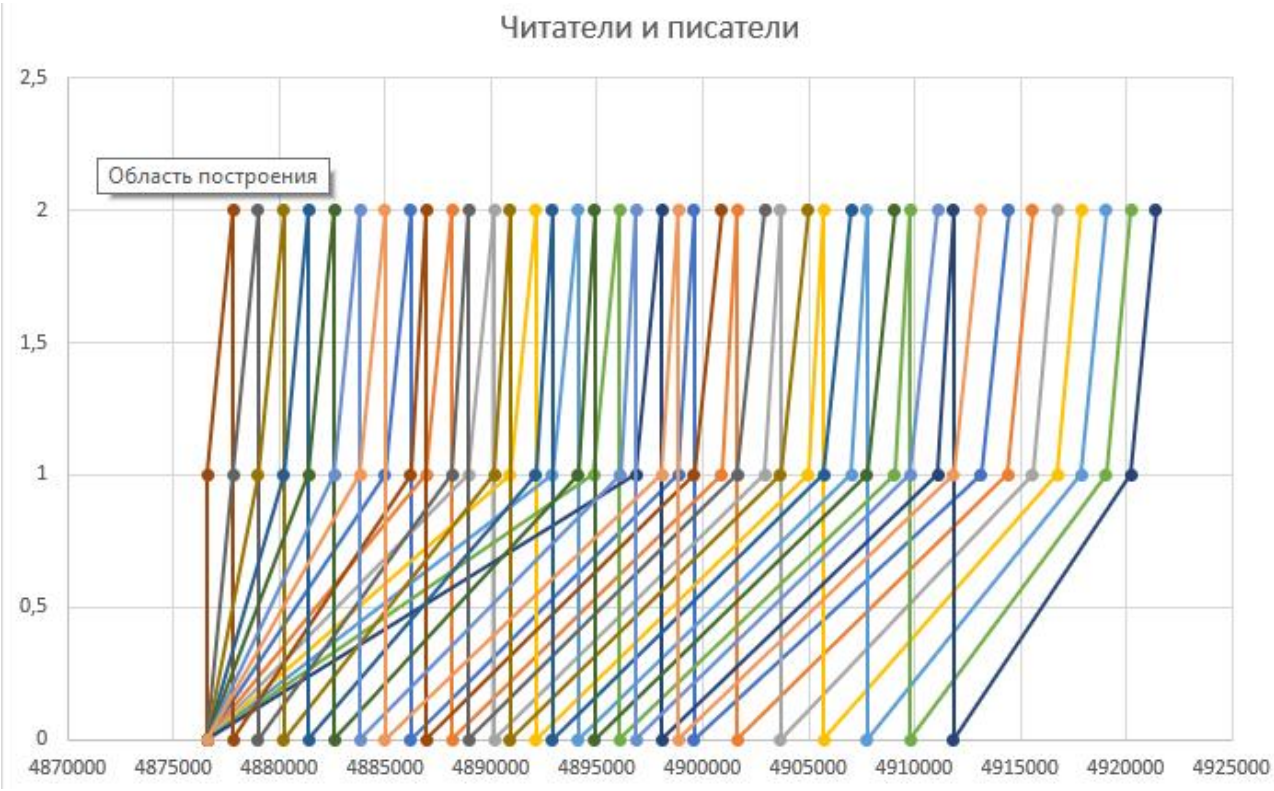

Рис.4 График состояний все писателей и читателей

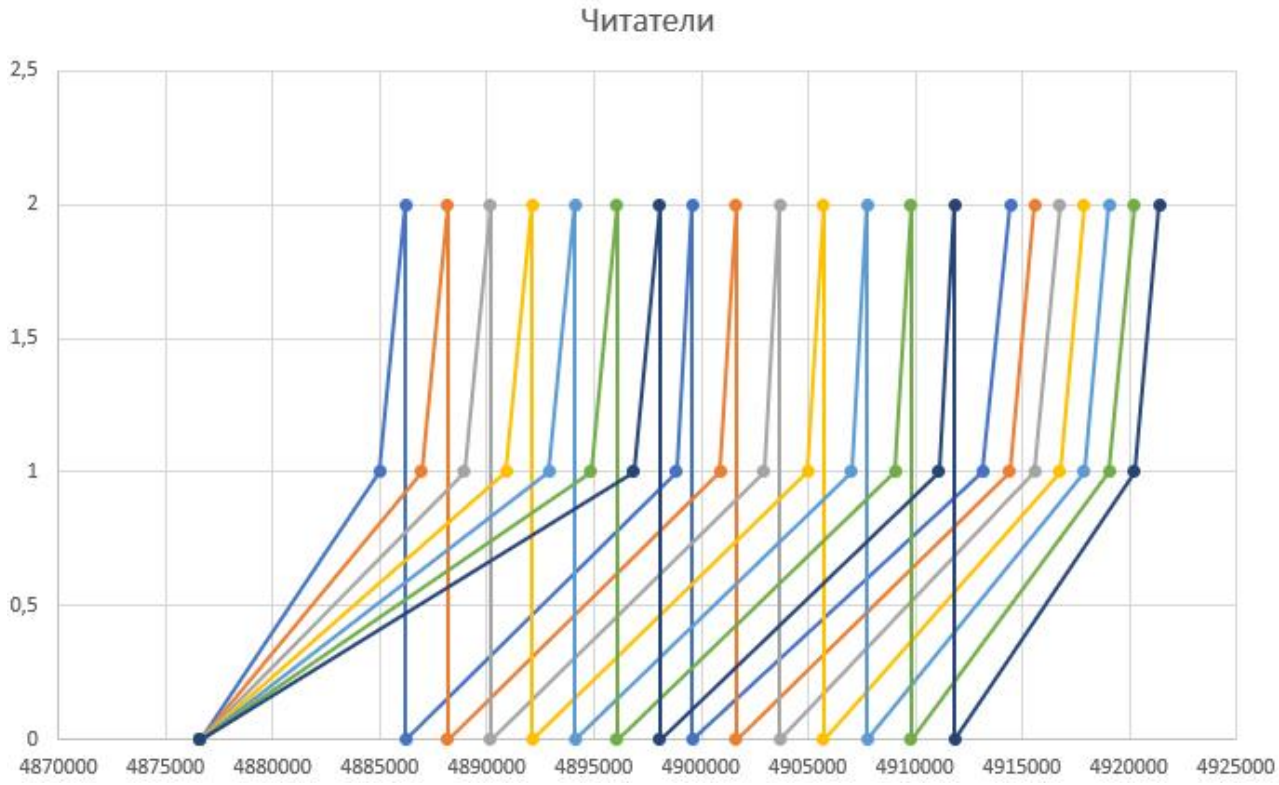

Рис.5 график состояний всех читателей

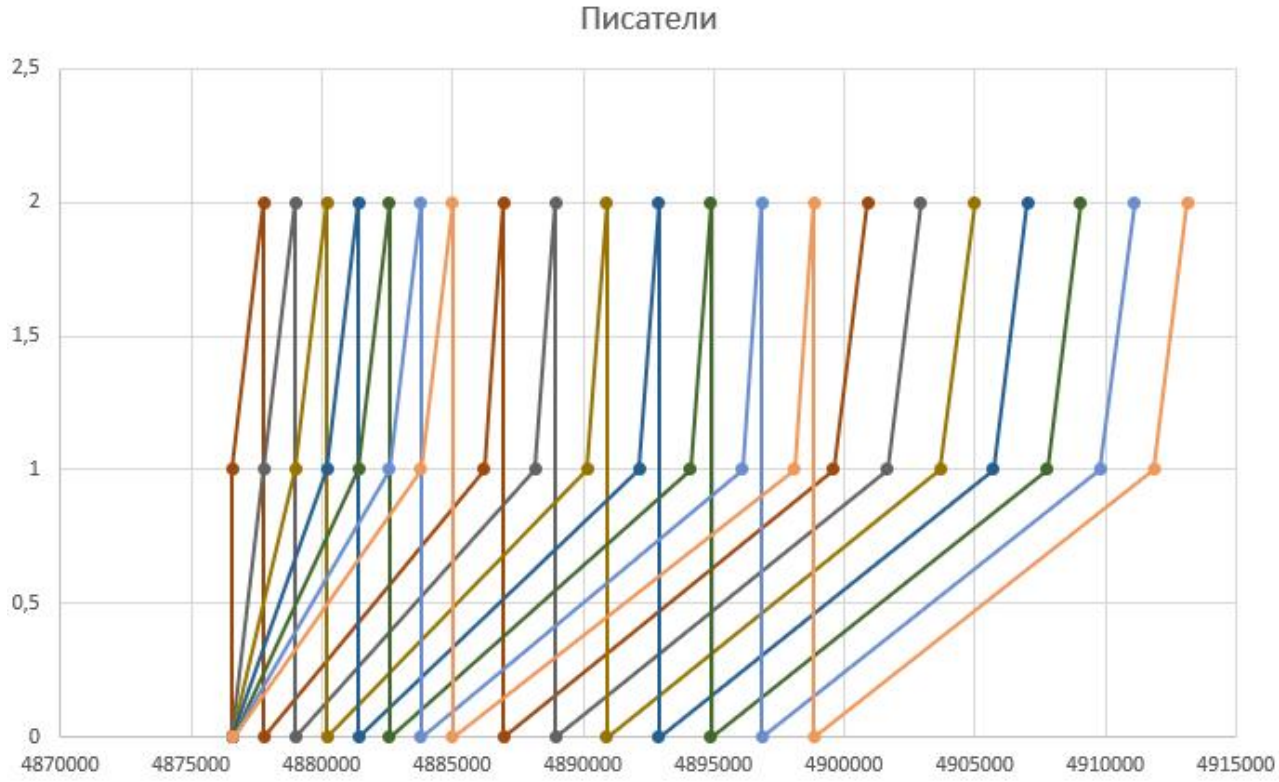

Рис.6 график состояний всех писателей

Выводы по первой части

Созданные консольные приложения "Читатель" и "Писатель" совместно взаимодействуют с буферной памятью в виде проецируемого файла. Размер страницы буферной памяти соответствует размеру физической страницы оперативной памяти, а количество страниц определяется суммой цифр в номере студенческого билета (без учета первой цифры). Страницы буферной памяти фиксируются в оперативной памяти с помощью функции VirtualLock. Длительность операций чтения и записи генерируется случайным образом в диапазоне от 0,5 до 1,5 секунд. Для синхронизации процессов используются семафоры и мьютексы, которые регулируют доступ к буферной памяти, гарантируя ее целостность и согласованность.

На графиках мы видим, что каждый читатель-писатель дожидается завершения всех прочих активных процессов, чтобы получить доступ к ресурсу, в данном случае симуляции записи или чтения виртуального файла. Читатели ждут чтобы хотя бы один писатель записал страницу файла.

Исходный код:

consts.hpp

```
#include <iostream>
#include <windows.h>
#include <ctime>
#include <windows.h>
#include <string>
#include <fstream>
#define WAITING 0
#define RW_OPERATION 1
#define REALISED 2
#define PATH_TO_LOGS
R"(C:\Users\Keltasar\CLionProjects\etu_2024_04_os_lab\lab4\part_1)"
const int pageSize = 4096,
        numberOfPages = 19,
        numberOfReaders = 10,
        numberOfWriters = 10,
        rwDelay_ms = 500,
        rwDelayDiv_ms = 1000;
const std::string mutexName = "IOMutex",
        mapName = "mapped_file",
        fileName = "file";
void logProcessEvent(int id, bool is_reader, int event_type, int page) {
    std::string processTypeName = "writer";
    if (is_reader) processTypeName = "reader";
    std::ofstream logFile(std::string(PATH_TO_LOGS) + processTypeName + "_" +
std::to_string(id) + ".txt", std::ios::app);
    if (logFile.is_open()) {
```

```
        DWORD time = GetTickCount();
        logFile << event_type << " " << time << " " << page << std::endl;
        logFile.close();
    }
}
void logProcessEvent(int id, bool is_reader, int event_type) {
    logProcessEvent(id, is_reader, event_type, -1);
}
```

writer.cpp

```
#include "consts.hpp"

int main(int argc, char* argv[]) {
    srand(time(nullptr));
    int id = strtol(argv[1], nullptr, 10);
    // Open handles to semaphores and mutex
    HANDLE writeSemaphores[numberOfPages], readSemaphores[numberOfPages];
    HANDLE ioMutex = OpenMutex(
            MUTEX_MODIFY_STATE | SYNCHRONIZE,
            false,
            mutexName.c_str());
    HANDLE mappedFile = OpenFileMapping(
            GENERIC_READ,
            false,
            mapName.c_str());
    LPVOID pointerToMappedContent = MapViewOfFile(mappedFile,
                                    FILE_MAP_WRITE,    0,    0,    pageSize    *
numberOfPages);
    for (int i = 0; i < numberOfPages; i++) {
```

```
    writeSemaphores[i] = OpenSemaphore(SEMAPHORE_MODIFY_STATE |
                                       SYNCHRONIZE,
                                       FALSE,
                                       std::to_string(i).c_str());
    readSemaphores[i] = OpenSemaphore(SEMAPHORE_MODIFY_STATE |
                                      SYNCHRONIZE,
                                      FALSE,
                                      std::to_string(i + numberOfPages).c_str());
}
for (int i = 0; i < 3; i++) {
    logProcessEvent(id, false, WAITING);
    DWORD pageNumber = WaitForMultipleObjects(
            numberOfPages,
            writeSemaphores,
            FALSE,
            INFINITE);
    WaitForSingleObject(
            ioMutex,
            INFINITE);
    logProcessEvent(id, false, RW_OPERATION, pageNumber);
    VirtualLock((char*)pointerToMappedContent + pageSize * pageNumber,
                pageSize);
    Sleep(rwDelay_ms + rand() % rwDelayDiv_ms);
    VirtualLock((char*)pointerToMappedContent + pageSize * pageNumber,
                pageSize);
    ReleaseMutex(ioMutex);
    ReleaseSemaphore(readSemaphores[pageNumber], 1, nullptr);
    logProcessEvent(id, false, REALISED);
```

```
    }
    for (int i = 0; i < numberOfPages; i++) {
        CloseHandle(writeSemaphores[i]);
        CloseHandle(readSemaphores[i]);
    }
    CloseHandle(ioMutex);
    CloseHandle(mappedFile);
    return 0;
}
```

reader.cpp

```
#include "consts.hpp"
// Function to open a semaphore with error handling
HANDLE OpenSemaphoreWithErrorCheck(DWORD accessMode, BOOL
inheritHandle, const std::string& semaphoreName) {
    HANDLE semaphore = OpenSemaphore(accessMode, inheritHandle,
                                semaphoreName.c_str());
    if (semaphore == nullptr) {
        std::cerr << "Error opening semaphore: " << semaphoreName << std::endl;
        ExitProcess(1);
    }
    return semaphore;
}
int main(int argc, char* argv[]) {
    srand(time(nullptr));
    int id = strtol(argv[1], nullptr, 10);
    HANDLE writeSemaphores[numberOfPages], readSemaphores[numberOfPages];
    HANDLE ioMutex = OpenMutex(
            MUTEX_MODIFY_STATE | SYNCHRONIZE,
```

```
        false,
        mutexName.c_str());
    HANDLE mappedFile = OpenFileMapping(
        GENERIC_READ,
        false,
        mapName.c_str());
    LPVOID pointerToMappedContent =
MapViewOfFile(mappedFile,FILE_MAP_WRITE, 0, 0, pageSize * numberOfPages);
    for (int i = 0; i < numberOfPages; i++) {
      writeSemaphores[i] =
          OpenSemaphoreWithErrorCheck(SEMAPHORE_MODIFY_STATE |
                                      SYNCHRONIZE,
                                      FALSE,
                                      std::to_string(i));
      readSemaphores[i] =
          OpenSemaphoreWithErrorCheck(SEMAPHORE_MODIFY_STATE |
                                      SYNCHRONIZE,
                                      FALSE,
                                      std::to_string(i + numberOfPages));
    }
    for (int i = 0; i < 3; i++) {
      logProcessEvent(id, true, WAITING);
      DWORD pageNumber = WaitForMultipleObjects(
          numberOfPages,
          readSemaphores,
          FALSE,
          INFINITE);
      WaitForSingleObject(
```

```
            ioMutex,
            INFINITE);
        logProcessEvent(id, true, RW_OPERATION, pageNumber);
        VirtualLock((char*)pointerToMappedContent + pageSize * pageNumber,
                pageSize);
        Sleep(rwDelay_ms + rand() % rwDelayDiv_ms);
        VirtualLock((char*)pointerToMappedContent + pageSize * pageNumber,
                pageSize);
        ReleaseMutex(ioMutex);
        ReleaseSemaphore(writeSemaphores[pageNumber], 1, nullptr);
        logProcessEvent(id, true, REALISED);
    }
    for (int i = 0; i < numberOfPages; i++) {
        CloseHandle(writeSemaphores[i]);
        CloseHandle(readSemaphores[i]);
    }
    CloseHandle(ioMutex);
    CloseHandle(mappedFile);
    return 0;
}
```

dispatch.cpp

```
#include "consts.hpp"
#include <chrono>
#include <thread>
HANDLE CreateNewProcess(const std::string&, const std::string&);
int main() {
    HANDLE writeSemaphores[numberOfPages], readSemaphores[numberOfPages];
    HANDLE ioMutex = CreateMutex(
```

```
        nullptr,
        false,
        mutexName.c_str());
HANDLE fileHandle = CreateFile(
        fileName.c_str(),
        GENERIC_WRITE | GENERIC_READ,
        0, nullptr,
        CREATE_ALWAYS,
        FILE_ATTRIBUTE_NORMAL,
        nullptr);
HANDLE mapFile = CreateFileMapping(
        fileHandle,
        nullptr,
        PAGE_READWRITE,
        0,
        pageSize * numberOfPages,
        mapName.c_str());
HANDLE readers[numberOfReaders], writers[numberOfWriters];
// Map the file to the memory
LPVOID fileView = MapViewOfFile(
        mapFile,
        FILE_MAP_ALL_ACCESS,
        0,
        0,
        pageSize * numberOfPages);
for (int i = 0; i < numberOfPages; i++) {
    writeSemaphores[i] = CreateSemaphore(
            nullptr,
```

```
            1,
            1,
            std::to_string(i).c_str());
    readSemaphores[i] = CreateSemaphore(
            nullptr,
            0,
            1,
            std::to_string(i + numberOfPages).c_str());
}
// Create writer and reader processes
for (int i = 0; i < numberOfWriters; i++) {
    writers[i] = CreateNewProcess("lab4_1_write.exe", std::to_string(i));
}
for (int i = 0; i < numberOfReaders; i++) {
    readers[i] = CreateNewProcess("lab4_1_read.exe", std::to_string(i));
}
WaitForMultipleObjects(
        numberOfWriters,
        writers,
        true,
        INFINITE);
WaitForMultipleObjects(
        numberOfReaders,
        readers,
        true,
        INFINITE);
UnmapViewOfFile(fileView);
CloseHandle(mapFile);
```

```
    CloseHandle(fileHandle);
    for (int i = 0; i < numberOfPages; i++) {
        CloseHandle(writeSemaphores[i]);
        CloseHandle(readSemaphores[i]);
    }
    CloseHandle(ioMutex);
    return 0;
}
HANDLE CreateNewProcess(const std::string& exePath, const std::string& logName)
{
    // Create the process with specified path to the executable file and arguments
    STARTUPINFOA si;
    PROCESS_INFORMATION pi;
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));
    char buffer[512];
    strcpy_s(buffer, exePath.c_str());
    std::string commandLine = exePath + " " + logName;
    if (!CreateProcessA(
            exePath.c_str(),
            const_cast<char*>(commandLine.c_str()),
            nullptr,
            nullptr,
            false,
            0,
            nullptr,
            nullptr,
```

```
        &si,
        &pi)) {
        std::cerr << "Error creating process: " << GetLastError() << std::endl;
        ExitProcess(1);
    }
    CloseHandle(pi.hThread);
    return pi.hProcess;
}
```

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 ввода вывода (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.

Демонстрация работы программы

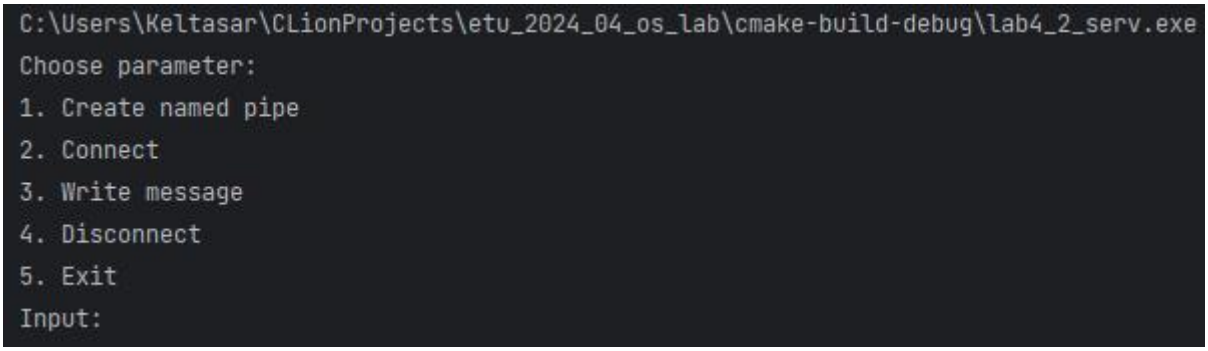

Рис.1 – server menu

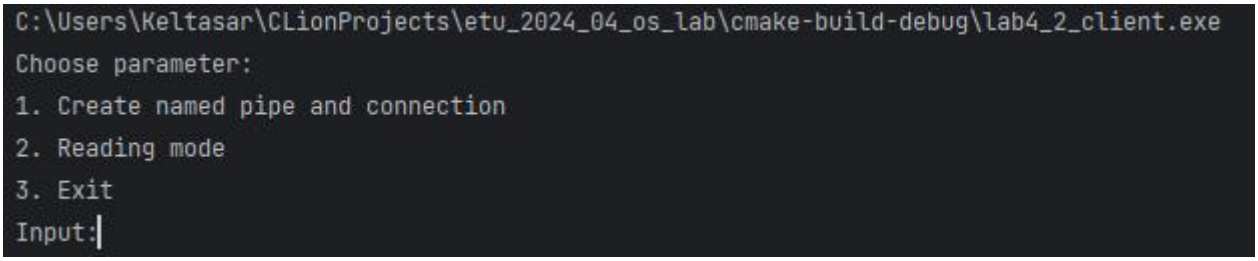

Рис.2 – client menu

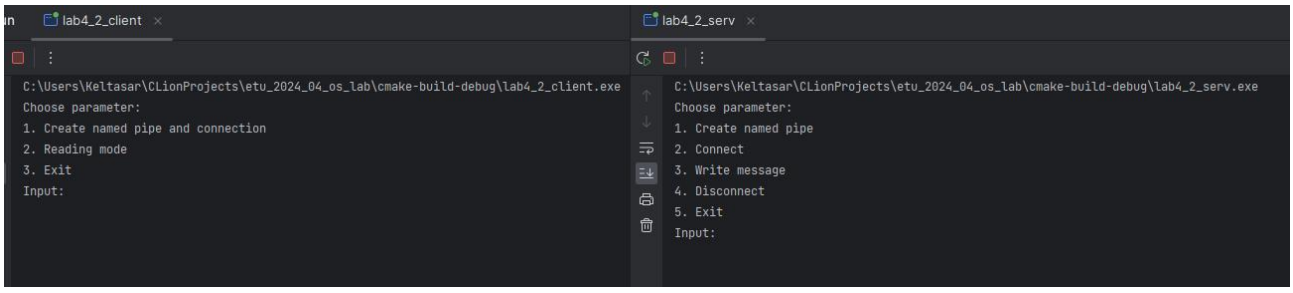

Рис.3 – ожидание соединения.

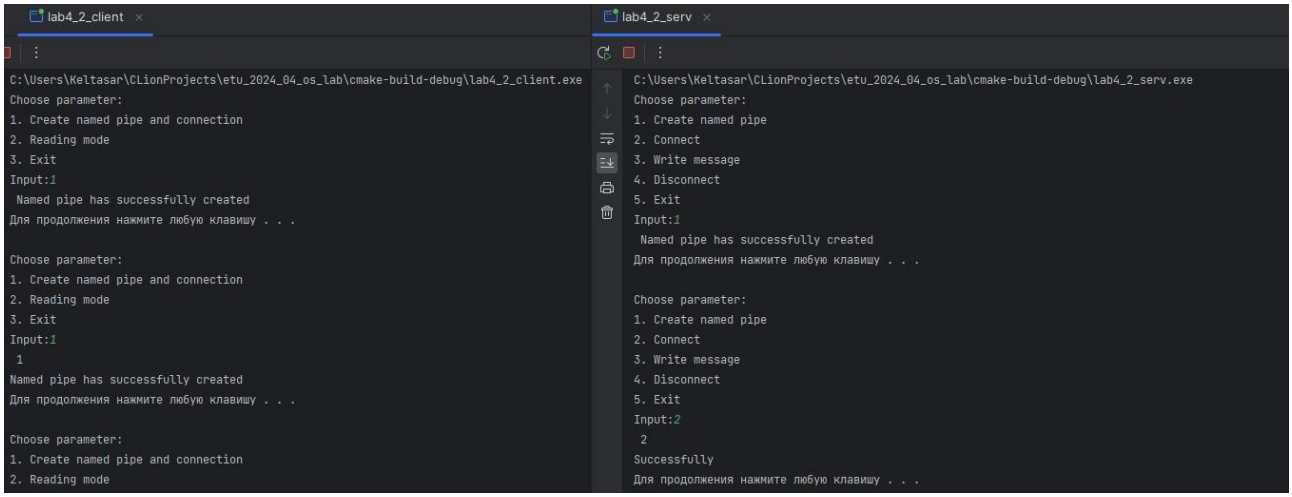

Рис.4 – успешное соединение.

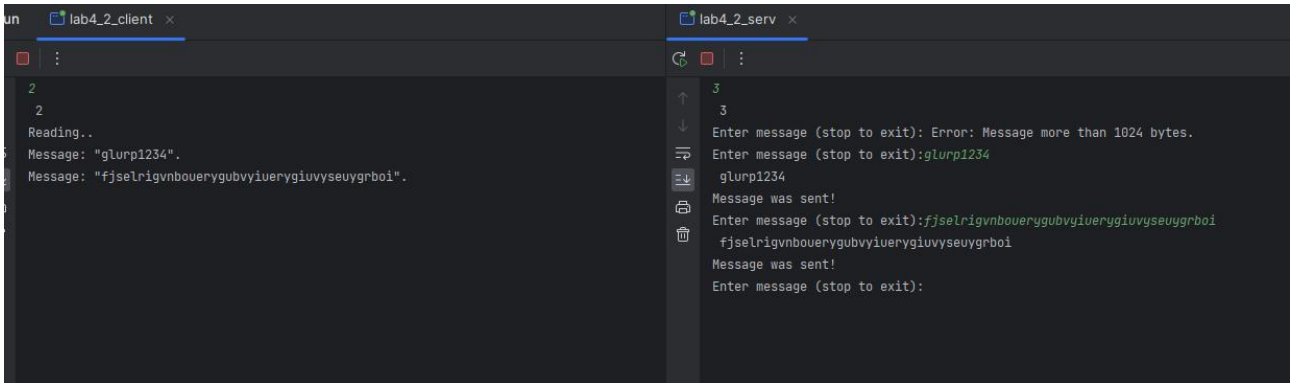

Рис. 5 – отправка сообщения.

Описание работы программ

Программа 1: client

1. **Отображение меню**: Программа выводит меню с опциями для создания соединения с именованным каналом и для чтения данных из канала.
2. **Создание соединения с именованным каналом**:
    - Функция **creationPipe** ожидает появление именованного канала и подключается к нему, используя **CreateFileA**. Канал открыт для чтения данных.
    - Если соединение успешно, выводится соответствующее сообщение.
3. **Режим чтения данных**:
    - Функция **readingMode** выполняет непрерывное чтение данных из канала до тех пор, пока не будет получено сообщение "stop".
    - Операция чтения выполняется асинхронно с помощью функции **ReadFileEx**, которая использует функцию обратного вызова **FileIOCompletionRoutine** для уведомления о завершении чтения.
    - Программа блокируется с помощью **SleepEx**, ожидая завершения чтения.
    - Полученные сообщения выводятся на экран.
4. **Главная функция**:
    - Обрабатывает пользовательский ввод для выполнения выбранных действий (создание соединения, чтение данных) до тех пор, пока пользователь не выберет опцию выхода.

Программа 2: server

1. **Отображение меню**: Программа выводит меню с опциями для создания именованного канала, подключения к нему, записи сообщений и отключения.

2. **Создание именованного канала**:
    - Функция **pipeCreation** создает именованный канал с помощью **CreateNamedPipeA**.
    - Если канал успешно создан, выводится соответствующее сообщение.
3. **Подключение к именованному каналу**:
    - Функция **ConnectNamedPipe** используется для ожидания подключения клиента к каналу.
    - Если подключение успешно, выводится соответствующее сообщение.
4. **Режим записи сообщений**:
    - Функция **writingPipe** позволяет пользователю вводить сообщения, которые отправляются через канал.
    - Вводимые сообщения копируются в буфер и отправляются с использованием функции **WriteFile**.
    - Программа ждет завершения операции записи с помощью **WaitForSingleObject**.
    - Если вводится сообщение "stop", программа прекращает запись и завершает работу.
5. **Отключение и завершение работы**:
    - Функция **DisconnectNamedPipe** разрывает соединение с клиентом.
    - Закрывается дескриптор события, используемый для асинхронных операций.
6. **Главная функция**:
    - Обрабатывает пользовательский ввод для выполнения выбранных действий (создание канала, подключение, запись сообщений, отключение) до тех пор, пока пользователь не выберет опцию выхода.

Теоретическая информация

Именованный канал (Named Pipe) в Windows — это механизм межпроцессного взаимодействия (IPC), который позволяет обмениваться данными между различными процессами. Эти процессы могут находиться на одном компьютере или на разных компьютерах в сети. Именованные каналы поддерживают двустороннюю передачу данных и могут быть асинхронными, что позволяет продолжать выполнение других задач, пока ожидаются операции чтения или записи.

Основные преимущества именованных каналов:

1. **Двусторонняя связь**: Именованные каналы позволяют передавать данные в обоих направлениях, что делает их подходящими для сложных взаимодействий между процессами.
2. **Локальная и сетевая связь**: Именованные каналы могут использоваться как для связи процессов на одном компьютере, так и для связи между процессами на разных компьютерах в сети.
3. **Асинхронные операции**: Поддержка асинхронного ввода-вывода позволяет процессам продолжать выполнение других задач во время ожидания завершения операций с каналом.
4. **Безопасность**: Интеграция с механизмами безопасности Windows позволяет устанавливать права доступа к каналам, обеспечивая защиту данных.
5. **Простота использования**: API именованных каналов в Windows делает их использование относительно простым для разработчиков, не требуя глубоких знаний о сетевых протоколах или управлении буферами.

Как клиент подключается к каналу

из лекции:

Клиенты производят подключение к каналу посредством вызова функции Create File ().

Далее сервер и клиенты могут обмениваться данными с помощью функций ReadFile () и WriteFile ().

Клиентский процесс может отключиться от канала в любой момент с помощью функции CloseHandle ().

Код программы

Client.cpp

```
#include <windows.h>
#include <iostream>

const size_t BUFFER_SIZE = 1024;
const std::string PIPE_NAME("\\\\.\\pipe\\lab4");
const char* EXIT_STR = "stop";
HANDLE hPipe;
size_t callback;


void show_menu(){
    std::cout << "Choose parameter:" << std::endl;
    std::cout << "1. Create named pipe and connection" << std::endl;
    std::cout << "2. Reading mode" << std::endl;
    std::cout << "3. Exit" << std::endl;
    std::cout << "Input: ";
```

```
}



void CALLBACK FileIOCompletionRoutine(DWORD dwErrorCode, DWORD
dwNumberOfBytesTransfered, LPOVERLAPPED lpOverlapped)
{
    ++callback;
}

void readingMode() {
    std::cout << "Reading.." << std::endl;
    if (hPipe != INVALID_HANDLE_VALUE) {
        OVERLAPPED over;
        size_t offset_i = 0;
        over.Offset = offset_i;
        over.OffsetHigh = offset_i >> 31;

        char buffer[BUFFER_SIZE];
        buffer[0] = '\0';

        while (strcmp(buffer, EXIT_STR) != 0) {
            callback = 0;

            ZeroMemory(buffer, BUFFER_SIZE);

            ReadFileEx(hPipe, buffer, BUFFER_SIZE, &over,
FileIOCompletionRoutine);
            SleepEx(-1, TRUE);
```

```
            std::cout << "Message: \"" << buffer << "\". " << std::endl;
        }
        CloseHandle(hPipe);
    } else {
        std::cout << "Error: Can't connect to the pipe." << std::endl;

    }
}
void creationPipe(){
    WaitNamedPipeA(PIPE_NAME.c_str(), NMPWAIT_WAIT_FOREVER);

    hPipe = CreateFileA(PIPE_NAME.c_str(), GENERIC_READ, 0, NULL, OPEN_EXISTING,
FILE_FLAG_OVERLAPPED | FILE_FLAG_NO_BUFFERING, NULL);
    std::cout << "Named pipe has successfully created" << std::endl;
}

int main()
{
   int choice;
   do{

       show_menu();
       std::cin>>choice;
       switch (choice) {
           case 1:
               creationPipe();
               system("pause");
               break;
           case 2:
```

```
                readingMode();
                system("pause");
                break;
        }
    }while (choice != 3);


    return 0;
}
```

Server.cpp

```
#include <windows.h>
#include <iostream>

const size_t BUFFER_SIZE = 1024;
const std::string PIPE_NAME("\\\\.\\pipe\\lab4");
const char* EXIT_STR = "stop";
HANDLE hPipe;
OVERLAPPED over;
void show_menu(){
    std::cout << "Choose parameter:" << std::endl;
    std::cout << "1. Create named pipe" << std::endl;
    std::cout << "2. Connect" << std::endl;
    std::cout << "3. Write message" << std::endl;
    std::cout << "4. Disconnect" << std::endl;
    std::cout << "5. Exit" << std::endl;
    std::cout << "Input: ";
}
void pipeCreation(){
```

```
    hPipe = CreateNamedPipeA(PIPE_NAME.c_str(),
                             PIPE_ACCESS_OUTBOUND | FILE_FLAG_OVERLAPPED |
WRITE_DAC,
                             PIPE_TYPE_MESSAGE | PIPE_WAIT,
                             1, 0, 0, 0, NULL);
    if (hPipe){
        std::cout<<"Named pipe has successfully created"<<std::endl;

    }
    else std::cout<<"Creation failed"<<std::endl;
}

void writingPipe() {
    size_t i;

    over.hEvent = CreateEvent(NULL, false, false, NULL);

    char buffer[BUFFER_SIZE];
    std::string string_buffer;
    while (strcmp(buffer, EXIT_STR) != 0) {
        ZeroMemory(buffer, 0);
        std::cout << "Enter message (" << EXIT_STR << " to exit): ";
        getline(std::cin, string_buffer);

        if (string_buffer.length() - 1 > BUFFER_SIZE) {
            std::cout << "Error: Message more than " << BUFFER_SIZE << " bytes."
<< std::endl;
            continue;
        } else {
```

```
            // CopyMemory(buffer, string_buffer.c_str(), string_buffer.length()
* sizeof(char));
            for (i = 0; i < string_buffer.length(); ++i)
                buffer[i] = string_buffer[i];
            buffer[i] = '\0';
        }

        WriteFile(hPipe, buffer, strlen(buffer) + 1, NULL, &over);
        WaitForSingleObject(over.hEvent, INFINITE);
        std::cout << "Message was sent!" << std::endl;
    }
}

int main()
{

    int choice;
    do {

        show_menu();
        std::cin >> choice;
        switch (choice) {
            case 1:
                pipeCreation();
                system("pause");
                break;
            case 2:
                if (ConnectNamedPipe(hPipe, NULL)) {
                    std::cout << "Successfully" << std::endl;
```

```
            } else {
                std::cout << "Error: Can't connect to pipe." << std::endl;
                return GetLastError();
            }
            system("pause");
            break;
        case 3:
            writingPipe();
            system("pause");
            break;
        case 4:
            DisconnectNamedPipe(hPipe);
            CloseHandle(over.hEvent);
            system("pause");
            break;
        default:
            std::cout << "Try again" << std::endl;
    }
}while(choice!=5);



return 0;
}
```

Выводы по второй части

На этой лабораторной работе я исследовал инструменты и механизмы взаимодействия процессов в Windows и использовал именованный канал для реализации сетевого межпроцессного взаимодействия. Были написаны приложения server и client, где приложение-сервер создает именованный канал выполняет установление и отключение соединения, создает объект «событие» осуществляет ввод данных с клавиатуры и их асинхронную запись в именованный канал, выполняет ожидание завершения операции ввода- вывода; приложение-клиент подключается к именованному каналу, в асинхронном режиме считывает содержимое из именованного канала файла и отображает на экран.сервер