**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра Вычислительной техники**

отчёт   
ЛАБОРАТОРНАЯ РАБОТА № 4

**по дисциплине «Операционные Системы»**

Тема: **Межпроцессное взаимодействие**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3311 |  | Шарпинский Д. А. |
| Преподаватель |  | Тимофеев А. В. |

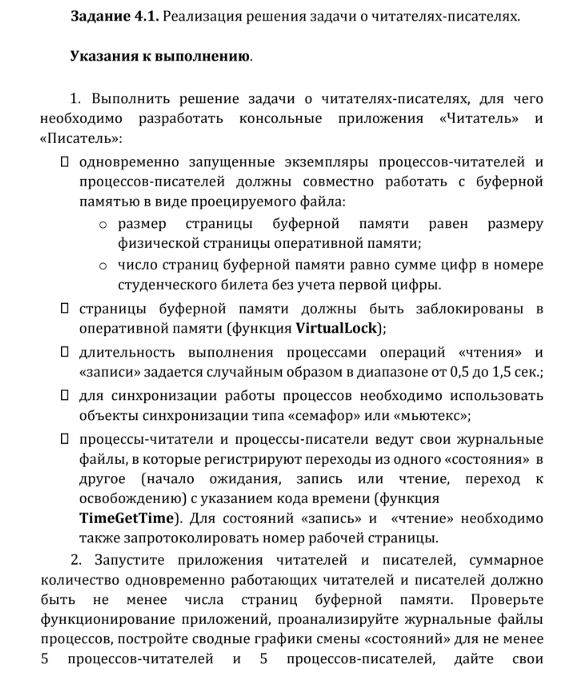
Санкт-Петербург

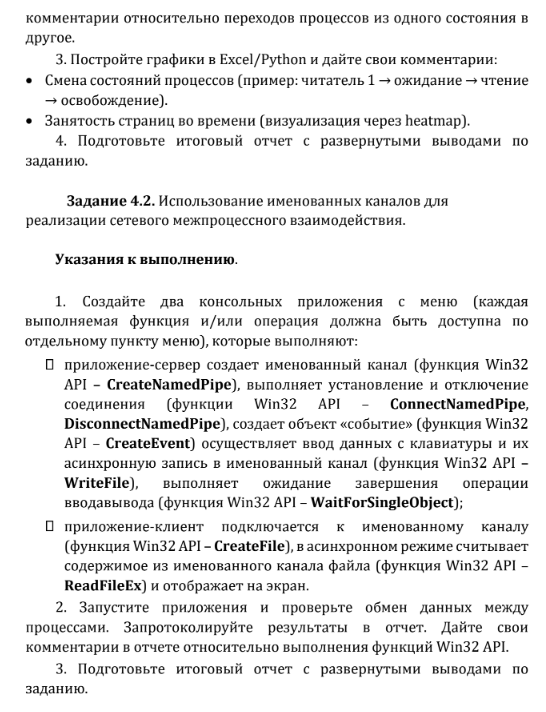
2025

# ЦЕЛЬ РАБОТЫ:

Исследовать инструменты и механизмы взаимодействия процессов в Windows.

# ЗАДАНИЯ НА ЛАБОРАТОРНУЮ РАБОТУ:





**Выполнение задание 4.1**

Задание 4.1 представляет из себя задачу о читателях-писателях. Поэтому были написаны два файла – reader.c и writer.c. Для их запуска был создан .bat файл, который reader.exe запускал 3 раза, а writer.exe 8 раз, передавая в качестве аргумента в main() порядковый номер процесса. Этот номер мы будем использовать позже.

Для простоты понимания я приведу аналогию с читальным залом: есть читальный зал, в который могут прийти либо читатели, либо писатели.

Читатель, приходя в зал, отмечается на входе и делает инкремент счетчика читателей. Читатель не может читать книгу (страницу), которую в данный момент читает другой читатель. Выходя, читатель делает декремент счетчика читателей. Также в обязанности первого читателя, когда тот прибудет в читальный зал, входит следующее: выставить табличку «писателям вход воспрещен». В обязанности последнего читателя, который покинет этот зал, входит убрать эту табличку.

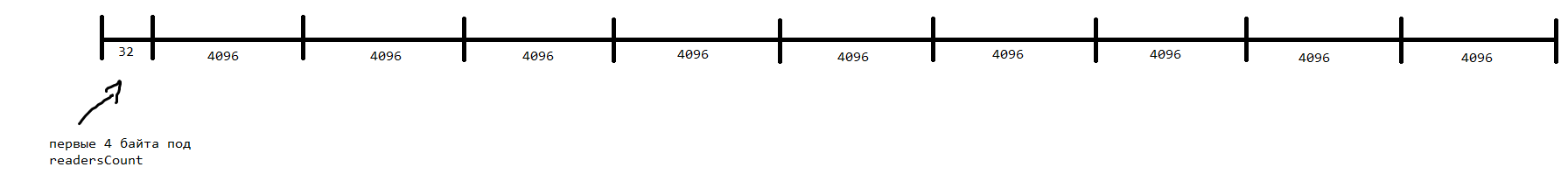
Писатель, в свою очередь, при входе в читальный зал, ждет, когда будет убрана табличка, а потом берет случайную из книжек (если книжка занята, то что ж – подождём), записывает туда hello\_NN\0, где NN – номер самого писателя, а после уходит.

Здесь:

* Читальный зал – шейред мемори
* Читатель – процесс reader.exe
* Писатель – процесс writer.exe
* Табличка на входе – мьютекс hMutex
* Счётчик писателей – первые 4 байта, которые я резервирую в нашей шейред мемори специально для этого счетчика.

Последнее хотелось бы разобрать подробнее:

Для реализации readersCount в reader.c я создал указатель типа LONG. Чтобы иметь доступ к актуальной информации я подписал его как volatile. И чтобы только 1 читатель мог обновлять этот счетчик в 1 момент времени я использовал семафор hReadersSemaphore (можно было и мьютекс, но просто разнообразия ради).



(Номер моего студенческого билета: 331131 => 3 + 1 + 1 + 3 + 1 = 9)

Теперь рассмотрим разделение ответственности между процессами.

Писатель:

1. Инициализирует память для шейред мемори.
2. Инициализирует мьютекс и семафор.
3. Отслеживает исключительно состояние мьютекса (до семафора ему дела нет).

Читатель:

1. Ничего не инициализирует, а лишь получает доступ к уже проинициализированным в памяти ячейками (т к читатели запускаются после писателей).
2. Обновляет счетчик читателей и, как было описано выше, обрабатывает 2 исключительные ситуации:
   1. Наш читатель – первый в зале и только в него пришел, а потому обязан выставить табличку (т е захватить мьютекс писателей)
   2. Наш читатель – последний в зале и уже уходит, а потому обязан убрать табличку (т е освободить мьютекс)
3. Читатели сами регулируют readersCount и семафор, чтобы отслеживать, сколько народу сейчас в читальном зале.

Каждый читатель и писатель в течение работы 5 раз приходит в читальный зал:

// main()

/\* other code \*/

for (int i = 0; i < 5; ++i) {

    performReadIteration();

}

Для того, чтобы писатели вообще могли пробиться в читальный зал было решено сделать 3 читателя и 8 писателей. Запуск производился через следующий .bat файл:

@echo off

setlocal enableextensions

REM === Создание директорий для логов ===

if not exist writers\_logs (

    mkdir writers\_logs

)

if not exist readers\_logs (

    mkdir readers\_logs

)

echo Starting writers...

for /L %%i in (0,1,7) do (

    start "" /B writer.exe %%i

)

REM — Ждём пару секунд, чтобы писатели успели создать shared memory

timeout /t 2 >nul

echo Starting readers...

for /L %%i in (0,1,2) do (

    start "" /B reader.exe %%i

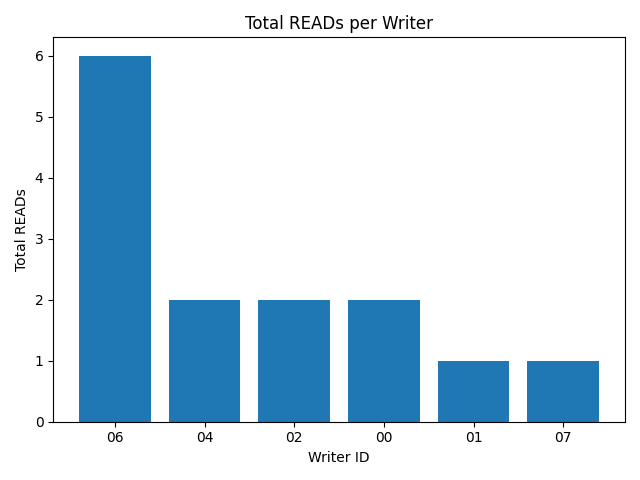
)

echo All processes launched.

endlocal

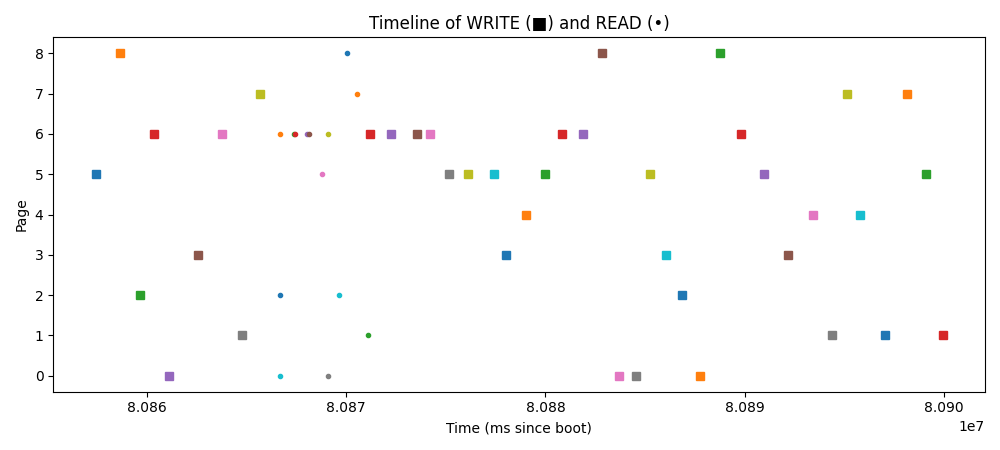
Теперь посмотрим на графики, которые были построены по логам.

Первый – это график наиболее «популярных» писателей:



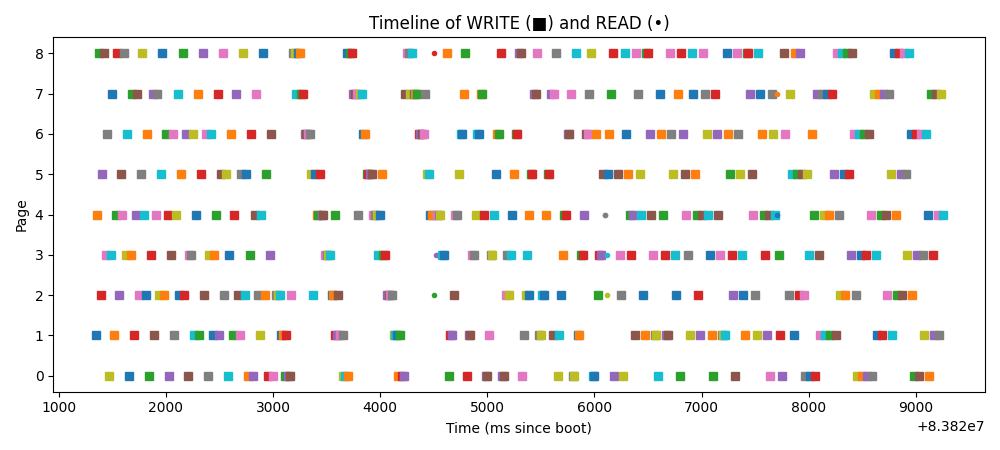
Как мы видим, писатель 6 оказался самым читаемым, а писатели 3 и 8 вообще никому не были интересны и получили 0 прочтений за все время (потому их нет на графике). Как я отслеживал, чью запись прочел какой-то читатель? Для этого я и оставлял в страницах подписи типа hello\_NN\0.

Второй график – это таймлайн наших станиц:



Видно, что на 6ую страницу пришлось наибольшее количество чтений и записей. Также видно, что сначала писатели пришли в читальный зал, сделали какое-то количество записей, потом ушли, пришли читатели, прочли всё, что писатели написали, а после ушли. Потом снова пришли писатели и принялись писать (правда читать уже было некому).

Я решил повторить замеры, но уже с 100 писателей и 2 читателями (но снизив время, которое они проводят за чтением в 100 раз):



Теперь видно, что промежутки пребывания читателей и писателей хотя бы чередуются.

**Промежуточный итог**

Был рассмотрен один из способов реализации ipc через шейред мемори. Писатели и читатели проецируют себе в vRAM 9 страниц и 4 байта из RAM (причем первый писатель еще и инициализирует их, заполняя нулями). Затем происходит описанный выше процесс посещения нашего читального зала (шейред мемори).

**Исходные коды**

**// reader.c**

#include <windows.h>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <mmsystem.h>

// задаем имя нашему буферу, мьютексу и семафору

#define SHM\_NAME L"SharedBuffer"

#define MUTEX\_NAME L"WritersMutex"

#define SEM\_NAME L"ReadersSemaphore"

// 331131 => 3 + 1 + 1 + 3 + 1 = 9

#define PAGE\_COUNT 9

#define PAGE\_SIZE 4096

#define LOG\_NAME\_FMT "readers\_logs/reader\_%02d.log"

// чтобы хранить количество читателей в данный момент, я выделил для этого 4 байта в нашей шейред мемори

// это нужно, чтобы понимать, является ли наш читатель первым или последним

#define READER\_COUNTER\_OFFSET 0

// ну а сами данные начинаются соответственно с оффсетом в 4 байта

#define BUFFER\_OFFSET 4

volatile LONG\* readersCount = NULL;

BYTE\* pageBase = NULL;

HANDLE hWritersMutex = NULL;

HANDLE hReadersSemaphore = NULL;

HANDLE hMap = NULL;

FILE\* logFile = NULL;

// через args в мейн я передаю номер читателя

int readerId = -1;

void logEvent(const char\* state, int pageIndex, DWORD timestamp) {

    if (logFile) {

        if (pageIndex >= 0)

            fprintf(logFile, "READER\_%02d %s PAGE\_%d %lu\n", readerId, state, pageIndex, timestamp);

        else

            fprintf(logFile, "READER\_%02d %s %lu\n", readerId, state, timestamp);

        fflush(logFile);

    }

}

void initLog(int id) {

    char filename[64];

    sprintf(filename, LOG\_NAME\_FMT, id);

    logFile = fopen(filename, "w");

    if (!logFile) {

        fprintf(stderr, "Failed to open log file.\n");

        exit(1);

    }

}

// читатели начинают свою работу уже после писателей, так что им не надо заполнять память, это уже сделано писателями

void initSharedObjects() {

    hMap = OpenFileMappingW(FILE\_MAP\_ALL\_ACCESS, FALSE, SHM\_NAME);

    if (!hMap) {

        fprintf(stderr, "OpenFileMapping failed: %lu\n", GetLastError());

        exit(1);

    }

    BYTE\* base = (BYTE\*)MapViewOfFile(hMap, FILE\_MAP\_ALL\_ACCESS, 0, 0, 0);

    if (!base) {

        fprintf(stderr, "MapViewOfFile failed: %lu\n", GetLastError());

        exit(1);

    }

    readersCount = (volatile LONG\*)(base + READER\_COUNTER\_OFFSET);

    pageBase = base + BUFFER\_OFFSET;

    // мьютекс и семафор тоже создаются писателями

    hWritersMutex = OpenMutexW(SYNCHRONIZE, FALSE, MUTEX\_NAME);

    if (!hWritersMutex) {

        fprintf(stderr, "OpenMutex failed: %lu\n", GetLastError());

        exit(1);

    }

    hReadersSemaphore = OpenSemaphoreW(SYNCHRONIZE | SEMAPHORE\_MODIFY\_STATE, FALSE, SEM\_NAME);

    if (!hReadersSemaphore) {

        fprintf(stderr, "OpenSemaphore failed: %lu\n", GetLastError());

        exit(1);

    }

}

void cleanup() {

    if (logFile) fclose(logFile);

    if (pageBase) UnmapViewOfFile((LPCVOID)(pageBase - BUFFER\_OFFSET));

    if (hMap) CloseHandle(hMap);

    if (hWritersMutex) CloseHandle(hWritersMutex);

    if (hReadersSemaphore) CloseHandle(hReadersSemaphore);

}

void performReadIteration() {

    int pageIndex = rand() % PAGE\_COUNT;

    DWORD tStart = timeGetTime();

    logEvent("WAIT\_READ", -1, tStart);

    // захватываем семафор, который показывает, читает ли кто-то страницу

    WaitForSingleObject(hReadersSemaphore, INFINITE);

    if (InterlockedIncrement(readersCount) == 1) {

        // если мы пришли в читальный зал первыми, то надо выставить табличку (т е захватить мьютекс)

        WaitForSingleObject(hWritersMutex, INFINITE);

    }

    // все, мы обновили readersCount, можно отпускать семафор

    ReleaseSemaphore(hReadersSemaphore, 1, NULL);

    DWORD tRead = timeGetTime();

    logEvent("READ", pageIndex, tRead);

    BYTE\* page = pageBase + pageIndex \* PAGE\_SIZE;

    char buffer[16] = {0};

    memcpy(buffer, page, 9);

    fprintf(logFile, "READER\_%02d READ\_CONTENT PAGE\_%d \"%s\"\n", readerId, pageIndex, buffer);

    fflush(logFile);

    Sleep(5 + rand() % 10);

    DWORD tRelease = timeGetTime();

    logEvent("RELEASE", -1, tRelease);

    // все то же самое, но наоброт, теперь мы выходим из "читального зала"

    WaitForSingleObject(hReadersSemaphore, INFINITE);

    if (InterlockedDecrement(readersCount) == 0) {

        // а если были последними, то разрешаем записывать

        ReleaseMutex(hWritersMutex);

    }

    ReleaseSemaphore(hReadersSemaphore, 1, NULL);

}

int main(int argc, char\* argv[]) {

    if (argc < 2) {

        fprintf(stderr, "Usage: Reader.exe <reader\_id>\n");

        return 1;

    }

    readerId = atoi(argv[1]);

    srand((unsigned int)(time(NULL) + readerId));

    initLog(readerId);

    initSharedObjects();

    for (int i = 0; i < 5; ++i) {

        performReadIteration();

    }

    cleanup();

    printf("Reader %d finished\n", readerId);

    return 0;

}

**// writer.c**

#include <windows.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>

// все по аналогии с readers.c

#define SHM\_NAME     L"SharedBuffer"

#define MUTEX\_NAME   L"WritersMutex"

#define SEM\_NAME     L"ReadersSemaphore"

#define PAGE\_COUNT   9

#define PAGE\_SIZE    4096

#define TOTAL\_SIZE   (4 + PAGE\_COUNT \* PAGE\_SIZE)

#define LOG\_FMT      "writers\_logs/writer\_%02d.log"

#define COUNTER\_OFFSET 0

#define BUFFER\_OFFSET  4

BYTE\* pageBase = NULL;

FILE\* logFile = NULL;

HANDLE hMap = NULL;

HANDLE hMutex = NULL;

HANDLE hReadersSemaphore = NULL;

int writerId = -1;

void logEvent(const char\* state, int pageIndex, DWORD timestamp) {

    if (logFile) {

        if (pageIndex >= 0)

            fprintf(logFile, "WRITER\_%02d %s PAGE\_%d %lu\n", writerId, state, pageIndex, timestamp);

        else

            fprintf(logFile, "WRITER\_%02d %s %lu\n", writerId, state, timestamp);

        fflush(logFile);

    }

}

void initLog(int id) {

    char filename[64];

    sprintf(filename, LOG\_FMT, id);

    logFile = fopen(filename, "w");

    if (!logFile) {

        fprintf(stderr, "Failed to open log file\n");

        exit(1);

    }

}

void initSharedMemory() {

    hMap = CreateFileMappingW(

        INVALID\_HANDLE\_VALUE,

        NULL,

        PAGE\_READWRITE,

        0,

        TOTAL\_SIZE,

        SHM\_NAME

    );

    if (!hMap) {

        fprintf(stderr, "CreateFileMapping failed: %lu\n", GetLastError());

        exit(1);

    }

    BYTE\* base = (BYTE\*)MapViewOfFile(hMap, FILE\_MAP\_ALL\_ACCESS, 0, 0, TOTAL\_SIZE);

    if (!base) {

        fprintf(stderr, "MapViewOfFile init failed: %lu\n", GetLastError());

        exit(1);

    }

    if (GetLastError() != ERROR\_ALREADY\_EXISTS) {

        // если мы первые, то надо заполнить память

        memset(base, 0, TOTAL\_SIZE);

    }

    pageBase = base + BUFFER\_OFFSET;

}

void initSyncObjects() {

  hMutex = CreateMutexW(NULL, FALSE, MUTEX\_NAME);

  if (!hMutex) {

      fprintf(stderr,"CreateMutex failed:%lu\n",GetLastError());

      exit(1);

  }

  hReadersSemaphore = CreateSemaphoreW(

      NULL,

      1,             // initial count

      1,             // max count

      SEM\_NAME

  );

  if (!hReadersSemaphore) {

      fprintf(stderr,"CreateSemaphore failed:%lu\n",GetLastError());

      exit(1);

  }

}

void cleanup() {

    if (logFile) fclose(logFile);

    if (pageBase) UnmapViewOfFile(pageBase - BUFFER\_OFFSET);

    if (hMap) CloseHandle(hMap);

    if (hMutex) CloseHandle(hMutex);

}

void performWriteIteration() {

    int pageIndex = rand() % PAGE\_COUNT;

    DWORD tWait = GetTickCount();

    logEvent("WAIT\_WRITE", -1, tWait);

    // ждем, пока все читатели покинут "читальный зал"

    WaitForSingleObject(hMutex, INFINITE);

    DWORD tWrite = GetTickCount();

    logEvent("WRITE", pageIndex, tWrite);

    BYTE\* target = pageBase + pageIndex \* PAGE\_SIZE;

    char content[16] = {0};

    sprintf(content, "hello\_%02d", writerId);

    memcpy(target, content, strlen(content) + 1);

    Sleep(5 + rand() % 10);

    DWORD tRelease = GetTickCount();

    logEvent("RELEASE", -1, tRelease);

    // покидаем "читальный зал" сами

    ReleaseMutex(hMutex);

}

int main(int argc, char\* argv[]) {

    if (argc < 2) {

        fprintf(stderr, "Usage: Writer.exe <writer\_id>\n");

        return 1;

    }

    writerId = atoi(argv[1]);

    srand((unsigned int)(time(NULL) + writerId));

    initLog(writerId);

    initSharedMemory();

    initSyncObjects();

    for (int i = 0; i < 5; ++i) {

        performWriteIteration();

    }

    cleanup();

    printf("Writer %d finished\n", writerId);

    return 0;

}

**Выполнение задания 4.2**

В данном задании требовалось реализовать IPC через именованные каналы (named pipes). Данная технология чем-то напоминает вебсокет: тоже есть сервер и неограниченное количество клиентов. Создаётся канал (объект кернел, который хранится в памяти) и у этого канала есть 2 буфера – клиентский и серверный. В эти буферы записываются сообщения от клиента и сервера соответственно. Если буфер переполнен, то в случае синхронной записи процессу придется ждать, пока буфер освободится. Буфер освобождается тогда, когда из него читают информацию. В случае асинхронной записи этот процесс не блокирует ввод/вывод. Из буфера можно читать информацию – синхронно и асинхронно. При синхронном чтении ввод и вывод блокируется, а при асинхронном – нет.

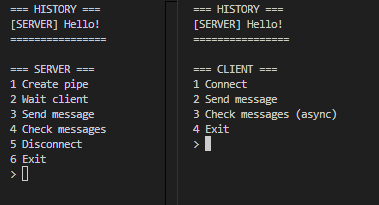
Для того, чтобы начать общение нужно:

1. На сервере создать канал – CreateNamedPipe.
2. Позволить серверу именованного канала ожидать подключения клиентского процесса к экземпляру именованного канала - ConnectNamedPipe.
3. На клиенте нужно подключиться к этому каналу, используя CreateFile.

После этого клиент и сервер могут отправлять друг другу сообщения (если канал был создан в дуплексном режиме). Реализация выглядит подобным образом:

|  |  |  |
| --- | --- | --- |
| Сервер создает канал: | Сервер разрешает подключение: | Клиент подключается |
|  |  |  |

После отправки сообщения на одном из приложений необходимо прочесть их на другом:



Для наглядности на сервере реализовано синхронное чтение, а на клиенте – асинхронное. Важно уточнить, что для асинхронного чтения в цикл внутри main() была добавлена строка

SleepEx(0, TRUE);

Благодаря ней поток переводится в alertable state на 0 миллисекунд, благодаря чему может обработать коллбек, переданный в асинхронную функцию чтения ReadFileEx.

**Промежуточный итог**

В данном задании был реализован другой механизм IPC – через named pipes. Данный механизм значительно проще shared memory в реализации, поскольку не требуется ручного отслеживания состояний и синхронизации. Теперь все, что требуется – так это написать коллбек для чтения.

**Исходные коды**

// server.c

#include <windows.h>

#include <stdio.h>

#include <string.h>

#define PIPE\_NAME "\\\\.\\pipe\\ChatPipe"

// максимальный размер сообщения

#define BUF 512

// макимальное количество сообщений в чате

#define MAX\_HISTORY 1024

HANDLE hPipe = NULL;

int clientOn = 0;

char history[MAX\_HISTORY][BUF];

int histCount = 0;

void addToHistory(const char \*tag, const char \*msg) {

    if (histCount >= MAX\_HISTORY) return;

    \_snprintf\_s(history[histCount], BUF, \_TRUNCATE, "%s %s", tag, msg);

    ++histCount;

}

void printHistory(void) {

    puts("=== HISTORY ===");

    for (int i = 0; i < histCount; ++i) puts(history[i]);

    puts("================\n");

}

void createPipe(void) {

    if (hPipe) CloseHandle(hPipe);

    hPipe = CreateNamedPipe(

        PIPE\_NAME,

        PIPE\_ACCESS\_DUPLEX,

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

        1, BUF, BUF, 0, NULL);

    puts(hPipe == INVALID\_HANDLE\_VALUE ? "CreateNamedPipe error" : "Pipe created.");

}

void waitClient(void) {

    if (!hPipe) {

        puts("Create pipe first.");

        return;

    }

    puts("Waiting client...");

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

        clientOn = 1;

        puts("Client connected.");

    }

    else puts("Connect error.");

}

void sendMessage(void) {

    if (!clientOn) {

        puts("No client.");

        return;

    }

    char buf[BUF];

    printf("Message: "); fgets(buf, BUF, stdin);

    buf[strcspn(buf, "\n")] = 0;

    DWORD wr;

    if (WriteFile(hPipe, buf, (DWORD)strlen(buf) + 1, &wr, NULL))

        addToHistory("[SERVER]", buf);

    else

        puts("Write error.");

}

void checkMessage(void) {

    if (!clientOn) {

        puts("No client.");

        return;

    }

    char buf[BUF] = {0};

    DWORD rd = 0;

    if (ReadFile(hPipe, buf, BUF, &rd, NULL)) {

        printf("[CLIENT] %s\n", buf);

    }

}

void disconnect(void) {

    if (hPipe) {

        DisconnectNamedPipe(hPipe);

        clientOn = 0;

        puts("Disconnected.");

    }

}

int main(void) {

    while(1) {

        system("cls");

        printHistory();

        puts("=== SERVER ===");

        puts("1 Create pipe");

        puts("2 Wait client");

        puts("3 Send message");

        puts("4 Check messages");

        puts("5 Disconnect");

        puts("6 Exit");

        printf("> ");

        int c;

        scanf("%d", &c);

        getchar();

        switch (c) {

            case 1: createPipe(); break;

            case 2: waitClient(); break;

            case 3: sendMessage(); break;

            case 4: checkMessage(); break;

            case 5: disconnect(); break;

            case 6: return 0;

        }

        getchar();

    }

}

// client.c

#include <windows.h>

#include <stdio.h>

#include <string.h>

#define PIPE\_NAME "\\\\.\\pipe\\ChatPipe"

#define BUF 512

#define MAX\_HISTORY 1024

HANDLE hPipe = INVALID\_HANDLE\_VALUE;

char history[MAX\_HISTORY][BUF];

int histCount = 0;

void addToHistory(const char \*tag, const char \*msg) {

    if (histCount >= MAX\_HISTORY) return;

    \_snprintf\_s(history[histCount], BUF, \_TRUNCATE, "%s %s", tag, msg);

    ++histCount;

}

void printHistory(void) {

    puts("=== HISTORY ===");

    for (int i = 0; i < histCount; ++i) puts(history[i]);

    puts("================\n");

}

VOID CALLBACK ReadCompleted(

    DWORD dwErrorCode,

    DWORD dwNumberOfBytesTransfered,

    LPOVERLAPPED lpOverlapped

);

void connectToServer(void) {

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

        puts("Already connected.");

        return;

    }

    hPipe = CreateFile(

        PIPE\_NAME,

        GENERIC\_READ | GENERIC\_WRITE,

        0,

        NULL,

        OPEN\_EXISTING,

        FILE\_FLAG\_OVERLAPPED,

        NULL

    );

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

        puts("Connect error.");

    } else {

        puts("Connected.");

    }

}

void sendMessage(void) {

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

      puts("Not connected.");

      return;

    }

    char buf[BUF];

    printf("Message: ");

    fgets(buf, BUF, stdin);

    buf[strcspn(buf, "\n")] = 0;

    DWORD wr;

    if (WriteFile(hPipe, buf, (DWORD)strlen(buf) + 1, &wr, NULL)) {

        addToHistory("[CLIENT]", buf);

    } else {

        puts("Write error.");

    }

}

void startAsyncRead(void) {

    static OVERLAPPED ov = { 0 };

    static char buf[BUF] = { 0 };

    ov.Pointer = buf;

    // Запускаем асинхронное чтение с callback

    if (!ReadFileEx(

            hPipe,

            buf,

            BUF,

            &ov,

            ReadCompleted

        )) {

        printf("ReadFileEx error: %lu\n", GetLastError());

    }

}

void checkMessage(void) {

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

        puts("Not connected.");

        return;

    }

    startAsyncRead();

    puts("Async read started.");

}

// Функция-обработчик завершения чтения

VOID CALLBACK ReadCompleted(

    DWORD dwErrorCode,

    DWORD dwNumberOfBytesTransfered,

    LPOVERLAPPED lpOverlapped

) {

    if (dwErrorCode == 0 && dwNumberOfBytesTransfered > 0) {

        char\* buf = (char\*)lpOverlapped->Pointer;

        printf("[SERVER] %s\n", buf);

        addToHistory("[SERVER]", buf);

    } else {

        printf("Read error or no data: code=%lu bytes=%lu\n",

               dwErrorCode, dwNumberOfBytesTransfered);

    }

}

int main(void) {

    while(1) {

        system("cls");

        printHistory();

        puts("=== CLIENT ===");

        puts("1 Connect");

        puts("2 Send message");

        puts("3 Check messages (async)");

        puts("4 Exit");

        printf("> ");

        int c;

        scanf("%d", &c);

        getchar();

        switch (c) {

            case 1: connectToServer(); break;

            case 2: sendMessage(); break;

            case 3: checkMessage(); break;

            case 4: return 0;

        }

        getchar();

        SleepEx(0, TRUE);

    }

}