**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра ВТ**

отчет

**по лабораторной работе №6**

**по дисциплине «Операционные системы»**

Тема: «Межпроцессное взаимодействие»

|  |  |  |
| --- | --- | --- |
| Студент гр. 7307 |  | Десятников Г.Е. |
| Преподаватель |  | Тимофеев А.В. |

Санкт-Петербург

2019

**Цель работы.**

Исследовать инструменты и механизмы взаимодействия процессов в ОС Windows.

**Задание 1.** Реализация решения задачи о читателях-писателях.

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:

– размер страницы буферной памяти равен размеру физической страницы оперативной памяти;

– число страниц буферной памяти равно сумме цифр в номере студенческого билета без учета первой цифры.

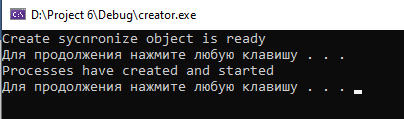
– страницы буферной памяти должны быть заблокированы в оперативной памяти;

– 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»;

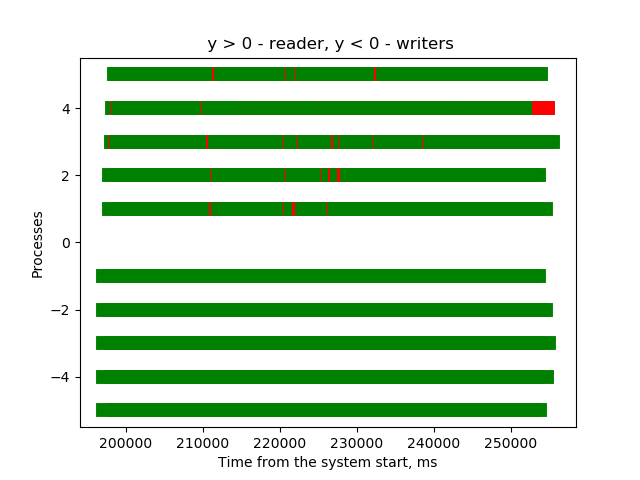
– процессы-читатели и процессы-писатели ведут свои журнальные файлы, в которые регистрируют переходы из одного «состояния» в другое (начало ожидания, запись или чтение, переход к освобождению) с указанием кода времени. Для состояний «запись» и «чтение» необходимо также запротоколировать номер рабочей страницы.

Для создания процессов, объектов синхронизации и памяти используется приложение-создатель. Приложение запускается создает все объекты для процессов и запускает сами процессы, затем ожидает нажатие кнопки для остановки всех процессов и освобождения ресурсов.



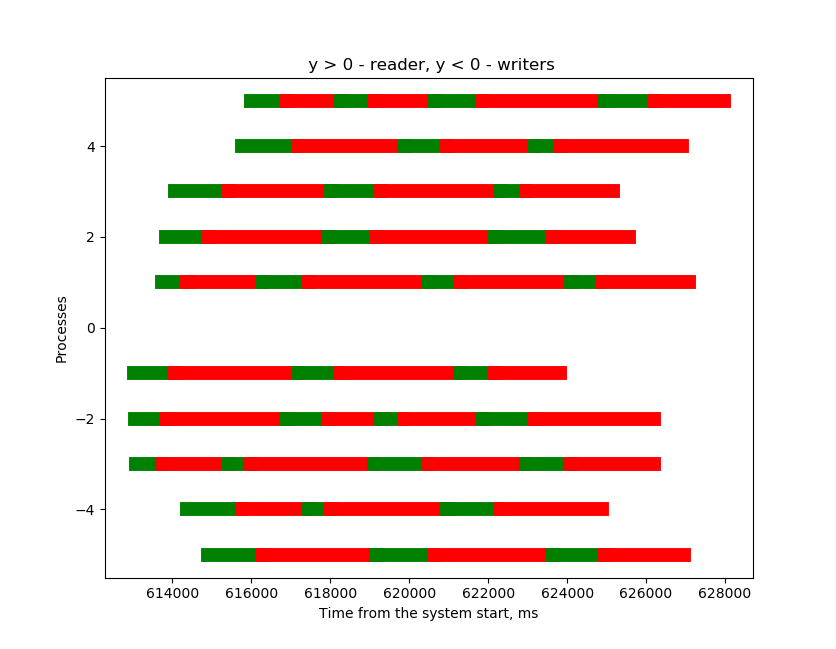
Число страниц буферной памяти: 7+ 3 + 0 + 7 + 2 + 0 = 19. Для реализации поставленной задачи о читателях-писателях на каждую страницу был использовано по два семафору на каждую страницу и два «мьютекса» для лог файлов. Половина семафоров разрешает чтение страницы памяти 1 читателю, а остальные писать 1 писателю. Так если свободный семафор со страницей забирает писатель, то после записи семафор страниц писателей остается закрытым,но освобождается семафор той же страницы для читателя, после закрытия и чтения читателем после открывает семафор писателя. Это гарантирует, что каждый писатель не будет писатель не будет писать в страницу которую не прочитал читатель, а читатель читать, прочитанный или не записанную страницу. Мьютексы гарантирует корректный доступ к лог файлам.

Далее представлен график работы 5 процессов читателей и писателей для 19 страниц.

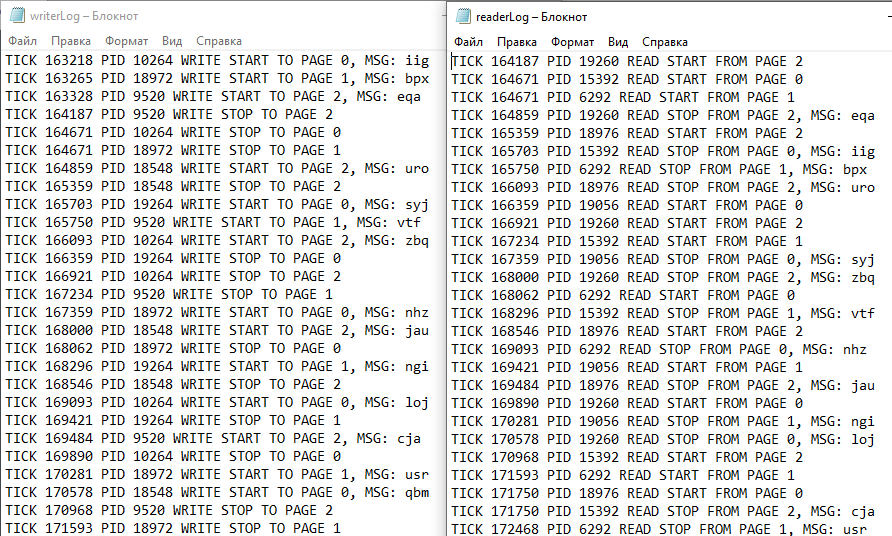


Как видно на графике работа писателей происходит непрерывно, что означает, что писателям хватает свободных страниц. Графики читателей сдвинуты от начала, из-за отсутствия записанных страниц. Виднеются простои в работе читателей, что связано отсутствием поступающих страниц от писателей.

Запустим процессы при 3 страницах.



Как видно на графике, при конце обработки 3 страниц писателями, эти страницы поступают к читателям. Писатели ожидают страницы после чтения читателями.



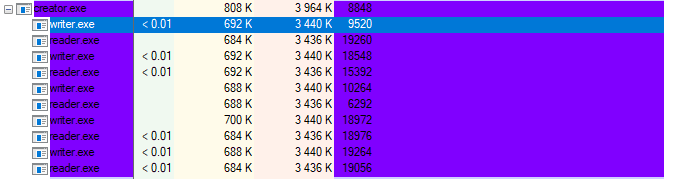


Рисунок 1.3. Информация об одном из потоков

Далее выведем объекты процесса-создателя. На рисунке видны созданные им процессы и их основные потоки.



Далее представлены результат работы программы handle для первых писателя и читателя соответственно. На них видно по 3 семафора для чтения и записи.



Далее представлена информация об объекте семафора двух процессов. Видно что информация совпадает.

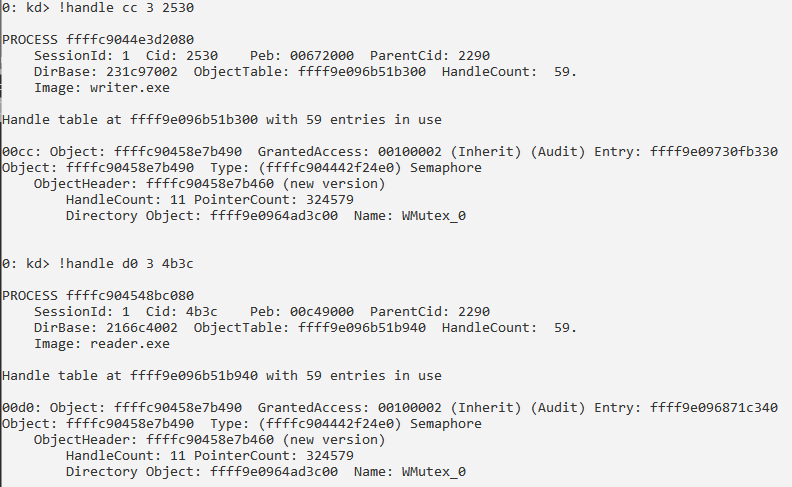


Рисунок 1.5.

**Вывод:** Если количество открытых процессов-читателей и процессов- писателей в сумме не превосходит количество используемых страниц, то когда расходуется минимальное количество времени на ожидание свободных страниц, так как всегда найдется хотя бы одна. Если же процессов больше, то с каждым новым процессом время ожидания будет увеличиваться.

**Задание 2**. Использование именованных каналов для реализации сетевого межпроцессного взаимодействия.

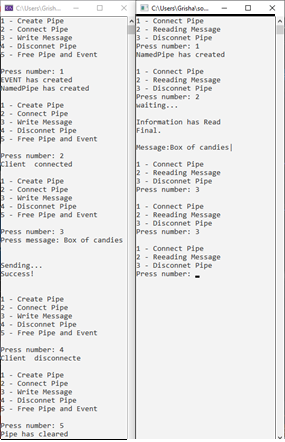
Создать два консольных приложения с меню (каждая выполняемая функция и/или операция должна быть доступна по отдельному пункту меню), которые выполняют:

– приложение-сервер создает именованный канал, выполняет установление и отключение соединения, создает объект «событие» и 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.

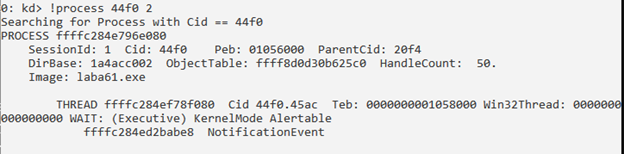
Запустим оба приложения и проверим их работоспособность.

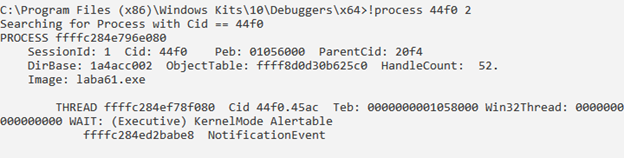


PID клиента: 44D016 =1761610;

PID сервера: 44F016 =1764810 .

Сравним количество дескрипторов приложений, до и после открытия каналов.





После запуска количество объектов процесса-сервера равно 50, но после

выполнения первого пункта меню счетчик увеличивается на 2. Так происходит из-за создания канала и отслеживаемого события.



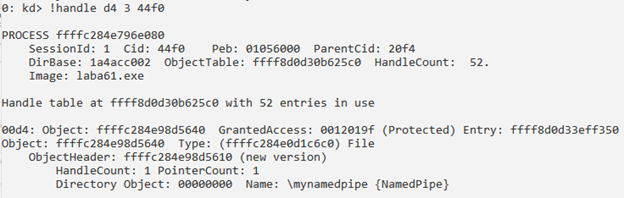


После запуска количество объектов процесса-сервера равно 50, но после

выполнения первого пункта меню счетчик увеличивается на 1. Так происходит из-за открытия именованного канала.

Выведем с помощью утилиты Handle список объектов данного процесса и найдем подробную информацию о именованном канале с помощью livekd.

Для процесса-сервера:



Для процесса-клиента:





Видно, что имена и типы объектов для процесса-клиента и процессасервера совпадают, а остальные поля отличаются.

**Вывод:** Именованные каналы являются одним из способов межпроцессного взаимодействия. Именованные каналы поддерживают полную дуплексную связь по сети и несколько экземпляров сервера, связь на основе сообщений и олицетворение клиента, позволяющее подключающимся процессам использовать собственные наборы разрешений на удаленных серверах.

Приложение 1

#include <iostream>

#include <windows.h>

#include <string>

#include <fstream>

using namespace std;

const int PAGESIZE = 4096;

const int PAGE\_COUNT = 19;

const LPCSTR MAP\_FILE = "D:\\NIR\_dir\\os\\os\\creator\\creator\\map\_file.txt";

const string WLOG\_FILE = "D:\\NIR\_dir\\os\\os\\creator\\creator\\writer.txt";

const string RLOG\_FILE = "D:\\NIR\_dir\\os\\os\\creator\\creator\\reader.txt";

const string WPATH = "D:\\NIR\_dir\\os\\os\\creator\\Debug\\writer.exe"; //writer

const string RPATH = "D:\\NIR\_dir\\os\\os\\creator\\Debug\\reader.exe"; //reader

const int PROCESS\_COUNT = 5;

int main()

{

HANDLE WMutex[PAGE\_COUNT];

HANDLE RMutex[PAGE\_COUNT];

for (int i = 0; i < PAGE\_COUNT; i++)

{

WMutex[i] = CreateSemaphoreA(NULL, 1, 1, ("WMutex\_" + to\_string(i)).c\_str());

RMutex[i] = CreateSemaphoreA(NULL, 0, 1, ("RMutex\_" + to\_string(i)).c\_str());

WaitForSingleObject(RMutex[i], INFINITY);

if (WMutex[i] == NULL || RMutex[i] == NULL) {

cerr << "Error in Creating Mutex" << endl;

return -1;

}

}

HANDLE WLogMutex = CreateMutexA(NULL, false, "WLogMutex");

HANDLE RLogMutex = CreateMutexA(NULL, false, "RLogMutex");

HANDLE MapFileHandle = CreateFileA(MAP\_FILE, GENERIC\_ALL, 0, NULL, CREATE\_ALWAYS, FILE\_ATTRIBUTE\_NORMAL, NULL);

HANDLE MapFileMapping = CreateFileMappingA(MapFileHandle, NULL, PAGE\_READWRITE, 0, PAGESIZE \* PAGE\_COUNT, "map\_file");

ofstream ofs;

ofs.open(WLOG\_FILE, std::ofstream::out | std::ofstream::trunc);

ofs.close();

ofs.open(RLOG\_FILE, std::ofstream::out | std::ofstream::trunc);

ofs.close();

cout << "Clear to execute, press to launch the connection" << endl;

system("pause");

PROCESS\_INFORMATION piW[PROCESS\_COUNT];

PROCESS\_INFORMATION piR[PROCESS\_COUNT];

BOOL t, p;

for (int i = 0; i < PROCESS\_COUNT; i++)

{

STARTUPINFOA siW;

ZeroMemory(&siW, sizeof(siW));

siW.cb = sizeof(siW);

ZeroMemory(&(piW[i]), sizeof(piW[i]));

STARTUPINFOA siR;

ZeroMemory(&siR, sizeof(siR));

siR.cb = sizeof(siR);

ZeroMemory(&(piR[i]), sizeof(piR[i]));

t = CreateProcessA(WPATH.c\_str(), NULL, NULL, NULL, false,

0, NULL, NULL, &siW, &(piW[i]));

if (!t) {

cerr << "Error in CreateProcessA(WPATH)" << GetLastError() << endl;

return -1;

}

p = CreateProcessA(RPATH.c\_str(), NULL, NULL, NULL, false,

0, NULL, NULL, &siR, &(piR[i]));

if (!p) {

cerr << "Error in CreateProcessA(RPATH)" << endl;

return -1;

}

}

cout << "Connection established, press something to end process of reading/writting" << endl;

system("pause");

CloseHandle(MapFileHandle);

CloseHandle(MapFileMapping);

for (int i = 0; i < PROCESS\_COUNT; i++)

{

CloseHandle(piW[i].hProcess);

CloseHandle(piW[i].hThread);

CloseHandle(piR[i].hProcess);

CloseHandle(piR[i].hThread);

}

for (auto& i : WMutex)

CloseHandle(i);

for (auto& i : RMutex)

CloseHandle(i);

cout << "Final" << endl;

return 0;

}

Приложение 2

#include <iostream>

#include <fstream>

#include <windows.h>

#include <string>

#include <ctime>

using namespace std;

const int PAGESIZE = 4096;

const int PAGE\_COUNT = 19;

const std::string RLOG\_FILE = "D:\\NIR\_dir\\os\\os\\creator\\creator\\reader.txt";

int main()

{

srand(GetCurrentProcessId());

HANDLE WMutex[PAGE\_COUNT];

HANDLE RMutex[PAGE\_COUNT];

for (int i = 0; i < PAGE\_COUNT; i++) {

WMutex[i] = OpenSemaphoreA(SEMAPHORE\_MODIFY\_STATE | SYNCHRONIZE, false, ("WMutex\_" + to\_string(i)).c\_str());

RMutex[i] = OpenSemaphoreA(SEMAPHORE\_MODIFY\_STATE | SYNCHRONIZE, false, ("RMutex\_" + to\_string(i)).c\_str());

}

HANDLE RLogMutex = OpenMutexA(MUTEX\_MODIFY\_STATE | SYNCHRONIZE, false, "RLogMutex");

ofstream Log;

Log.open(RLOG\_FILE, fstream::out | fstream::app);

HANDLE Mapping = OpenFileMappingA(GENERIC\_READ, false, "map\_file");

void\* OwnMap = MapViewOfFile(Mapping, FILE\_MAP\_READ, 0, 0, PAGESIZE \* PAGE\_COUNT);

VirtualLock(OwnMap, PAGESIZE \* PAGE\_COUNT);

while (1) {

char msg[PAGESIZE];

DWORD PageToRead = WaitForMultipleObjects(PAGE\_COUNT, RMutex, false, INFINITE) - WAIT\_OBJECT\_0;

WaitForSingleObject(RLogMutex, INFINITE);

Log << "TICK " << (GetTickCount() % 1000000) << " PID " << GetCurrentProcessId() << " READ START FROM PAGE " << PageToRead << endl;

ReleaseMutex(RLogMutex);

memcpy(msg, (void\*)((intptr\_t)OwnMap + (PageToRead \* PAGESIZE)), PAGESIZE);

Sleep(500 + (rand() % 1000));

WaitForSingleObject(RLogMutex, INFINITE);

Log << "TICK " << (GetTickCount() % 1000000) << " PID " << GetCurrentProcessId() << " READ STOP FROM PAGE " << PageToRead << ", MSG: " << msg[0] << msg[1] << msg[2] << endl;

ReleaseMutex(RLogMutex);

ReleaseSemaphore(WMutex[PageToRead], 1, NULL);

}

system("pause");

return 0;

}

Приложение 3

#include <iostream>

#include <fstream>

#include <windows.h>

#include <string>

#include <ctime>

using namespace std;

const int PAGESIZE = 4096;

const int PAGE\_COUNT = 19;

const string WLOG\_FILE = "D:\\NIR\_dir\\os\\os\\creator\\creator\\writer.txt";

int main()

{

srand(GetCurrentProcessId());

HANDLE WMutex[PAGE\_COUNT];

HANDLE RMutex[PAGE\_COUNT];

for (int i = 0; i < PAGE\_COUNT; i++) {

WMutex[i] = OpenSemaphoreA(SEMAPHORE\_MODIFY\_STATE | SYNCHRONIZE, false, ("WMutex\_" + to\_string(i)).c\_str());

RMutex[i] = OpenSemaphoreA(SEMAPHORE\_MODIFY\_STATE | SYNCHRONIZE, false, ("RMutex\_" + to\_string(i)).c\_str());

}

HANDLE WLogMutex = OpenMutexA(MUTEX\_MODIFY\_STATE | SYNCHRONIZE, false, "WLogMutex");

HANDLE Mapping = OpenFileMappingA(GENERIC\_WRITE, false, "map\_file");

void\* OwnMap = MapViewOfFile(Mapping, FILE\_MAP\_WRITE, 0, 0, PAGESIZE \* PAGE\_COUNT);

VirtualLock(OwnMap, PAGESIZE \* PAGE\_COUNT);

ofstream Log;

Log.open(WLOG\_FILE, fstream::out | fstream::app);

while (1)

{

char msg[PAGESIZE];

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

msg[i] = 'a' + rand() % 19;

DWORD PageToWrite = WaitForMultipleObjects(PAGE\_COUNT, WMutex, false, INFINITE) - WAIT\_OBJECT\_0;

WaitForSingleObject(WLogMutex, INFINITE);

Log << "TICK " << (GetTickCount() % 1000000) << " PID " << GetCurrentProcessId() << " WRITE START TO PAGE " << PageToWrite << ", MSG: " << msg[0] << msg[1] << msg[2] << endl;

ReleaseMutex(WLogMutex);

memcpy((void\*)((intptr\_t)OwnMap + (PageToWrite \* PAGESIZE)), msg, PAGESIZE);

Sleep(500 + (rand() % 1000));

WaitForSingleObject(WLogMutex, INFINITE);

Log << "TICK " << (GetTickCount() % 1000000) << " PID " << GetCurrentProcessId() << " WRITE STOP TO PAGE " << PageToWrite << endl;

ReleaseMutex(WLogMutex);

ReleaseSemaphore(RMutex[PageToWrite], 1, NULL);

}

system("pause");

return 0;

}

Приложение 4

#include <windows.h>

#include <iostream>

#include <string>

using namespace std;

int main()

{

HANDLE hPipe = INVALID\_HANDLE\_VALUE;

HANDLE hEvent = INVALID\_HANDLE\_VALUE;

OVERLAPPED overlapped = OVERLAPPED(), overlappedPipe = OVERLAPPED();

char\* mes;

string s;

int x = 1, y, i;

bool f, connection = false;

while (x != 0) {

cout << "1 - Create Pipe" << endl;

cout << "2 - Connect Pipe" << endl;

cout << "3 - Write Message " << endl;

cout << "4 - Disconnet Pipe" << endl;

cout << "5 - Free Pipe and Event" << endl;

cout << "\nPress number: ";

cin >> x;

switch (x) {

case 1:

hEvent = CreateEventA(NULL, false, false, NULL);

if (hEvent == INVALID\_HANDLE\_VALUE)

cout << "Error creating EVENT" << endl;

else

cout << "EVENT has created" << endl;

s = "\\\\.\\pipe\\mynamedpipe";

hPipe = CreateNamedPipeA(s.c\_str(), PIPE\_ACCESS\_DUPLEX | FILE\_FLAG\_OVERLAPPED, PIPE\_TYPE\_MESSAGE | PIPE\_READMODE\_MESSAGE | PIPE\_WAIT, PIPE\_UNLIMITED\_INSTANCES, 512, 512, 0, NULL);

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

cout << "NamedPipe has created\n";

if (hEvent != INVALID\_HANDLE\_VALUE)

overlappedPipe.hEvent = hEvent;

else

cout << "Error create NamedPipe\n";

}

else

cout << "Error create NamedPipe\n";

break;

case 2:

connection = ConnectNamedPipe((HANDLE)hPipe, NULL);

if (connection) cout << "Client connected\n";

else cout << "Client DIDN'T connecte\n";

break;

case 4:

connection = DisconnectNamedPipe((HANDLE)hPipe);

if (connection)

cout << "Client disconnecte\n";

else

cout << "Client DIDN'T disconnecte\n";

connection = !connection;

break;

case 3:

if (!connection) cout << "No connection" << endl;

else {

cout << "Press message: ";

cin >> ws;

getline(cin, s);

cout << endl;

mes = new char[512];

for (i = 0; i < s.length(); ++i) mes[i] = s[i];

mes[i] = '\0';

overlapped.hEvent = hEvent;

WriteFile(hPipe, mes, 512, NULL, &overlapped);

cout << "\nSending...\n";

if (WaitForSingleObject(hEvent, 2000) == WAIT\_OBJECT\_0)

cout << "Success!" << endl;

else

cout << "Writing crashed";

delete[]mes;

}

cout << endl;

break;

case 5:

if (hPipe == INVALID\_HANDLE\_VALUE)

cout << "Pipe hasn't cleared" << endl;

else {

CloseHandle(hPipe);

hPipe == INVALID\_HANDLE\_VALUE;

cout << "Pipe has cleared" << endl;

}

if (hEvent == INVALID\_HANDLE\_VALUE)

cout << "EVENT hasn't cleared" << endl;

else {

CloseHandle(hEvent);

hEvent = INVALID\_HANDLE\_VALUE;

cout << "EVENT has cleared" << endl;

}

break;

}

cout << endl;

}

}

Приложение 5

#include <windows.h>

#include <iostream>

#include <string>

using namespace std;

void WINAPI ReadCallback(DWORD dwErrorCode, DWORD dwNumberOfBytesTransfered, LPOVERLAPPED lpOverlapped) {

cout << "Information has Read" << endl;

}

int main()

{

HANDLE hPipe = INVALID\_HANDLE\_VALUE;

OVERLAPPED overlapped;

char\* mes;

string s;

int x = 1, y, i;

bool f, connection = false;

while (x != 0) {

cout << "1 - Connect Pipe" << endl;

cout << "2 - Reeading Message" << endl;

cout << "3 - Disconnet Pipe" << endl;

cout << "Press number: ";

cin >> x;

switch (x) {

case 1:

s = "\\\\.\\pipe\\mynamedpipe";

hPipe = CreateFileA(s.c\_str(), GENERIC\_READ | GENERIC\_WRITE, 0, NULL, OPEN\_EXISTING, FILE\_FLAG\_OVERLAPPED, NULL);

if (hPipe != INVALID\_HANDLE\_VALUE)

cout << "NamedPipe has created\n";

else

cout << "Error create NamedPipe\n";

break;

case 2:

mes = new char[512];

overlapped = OVERLAPPED();

if (!ReadFileEx(hPipe, (LPVOID)mes, 512, &overlapped, ReadCallback)) cout << "reading no";

else

{

cout << "waiting..." << endl << endl;

SleepEx(INFINITE, true);

cout << "Final." << endl << endl;

cout << "Message:" << mes << endl;

}

delete[] mes;

break;

case 3:

if (hPipe != INVALID\_HANDLE\_VALUE)

{

if (!CloseHandle(hPipe))

cout << "error closing hPipe" << endl;

else cout << "hPipe has closed" << endl;

hPipe = INVALID\_HANDLE\_VALUE;

}

break;

}

cout << "\n";

}

}