Минобрнауки России  
Федеральное государственное автономное образовательное  
Учреждение высшего образования  
«Санкт-Петербургский государственный электротехнический  
Университет им. В.И. Ульянова (Ленина)»  
(СПГЭТУ «ЛЭТИ»)  
Факультет компьютерных технологий и информатики

Кафедра вычислительной техники  
  
  
  
**Отчет по лабораторной работе №4**

**Тема:«** **Межпроцессное взаимодействи»**

по дисциплине «Операционные системы»

Выполнил студент гр.9307 Брызгалова Е.А.

Принял: к.т.н., доцент Тимофеев А.В.

Санкт-Петербург  
2021 г.

Оглавление

[Цель: 3](#_Toc91672357)

[Исследовать инструментыи механизмы взаимодействия процессов в Windows. 3](#_Toc91672358)

[Задание 1.1. 3](#_Toc91672359)

[Указания к выполнению. 3](#_Toc91672360)

[Графики смены «состояний». 5](#_Toc91672361)

[**График занятости страниц буферной памяти** 6](#_Toc91672362)

[Выводы по заданию 1 6](#_Toc91672363)

[Задание 1.2 6](#_Toc91672364)

[Указания к выполнению 6](#_Toc91672365)

[Протокол работы программы 8](#_Toc91672366)

[Выводы по заданию 2 8](#_Toc91672367)

[Приложение 9](#_Toc91672368)

[Исходный код 9](#_Toc91672369)

# Цель:

# Исследовать инструментыи механизмы взаимодействия процессов в Windows.

# Задание 1.1.

## Указания к выполнению.

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 (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.

Пример журнала

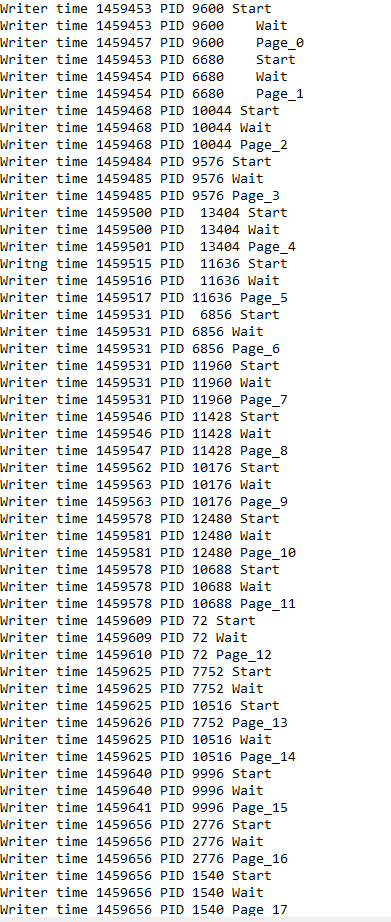


Рисунок 1. Журнал

## Графики смены «состояний».

Рисунок 1. Start

Рисунок 2. Wait

Рисунок 3.Read/Wright

Рисунок 4. Ready

Рисунок 5. Finish

График занятости страниц буферной памяти

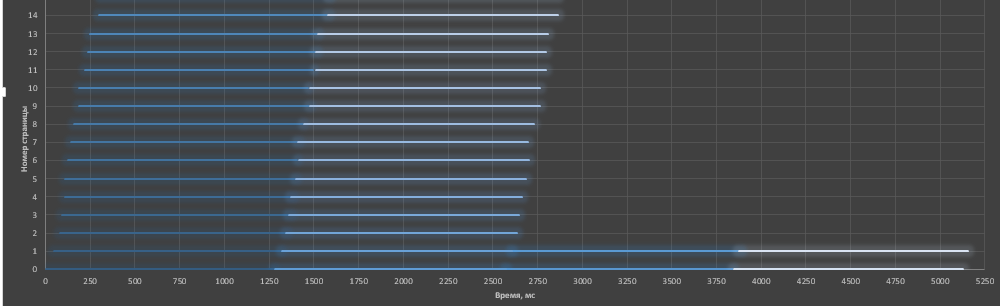


Рисунок 1. График

## Выводы по заданию 1

В ходе выполнения первого задания лабораторной работы было реализовано консольные приложения «Читатель» и «Писатель». Познакомились с семафорами и мьютексами. Семафор – неотрицательная целая переменная, которая может изменяться и проверяться только посредством двух примитивов: увеличения и, если возможно, уменьшения на 1. Мьютекс - упрощенная версия семафора - переменная, которая может находиться водном из двух состояний: блокированном или неблокированном.

Достоинства семафоров:

* простота
* отсутствие «активного ожидания»
* независимость от количества процессов

Недостатки:

* семафор не «привязывается» к конкретному синхроусловию или
* критическому ресурсу
* сложность доказательства корректности программы
* неправильное использование может привести к нарушению
* работоспособности системы (возможны тупиковые ситуации)

# Задание 1.2

## Указания к выполнению

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.

## Протокол работы программы

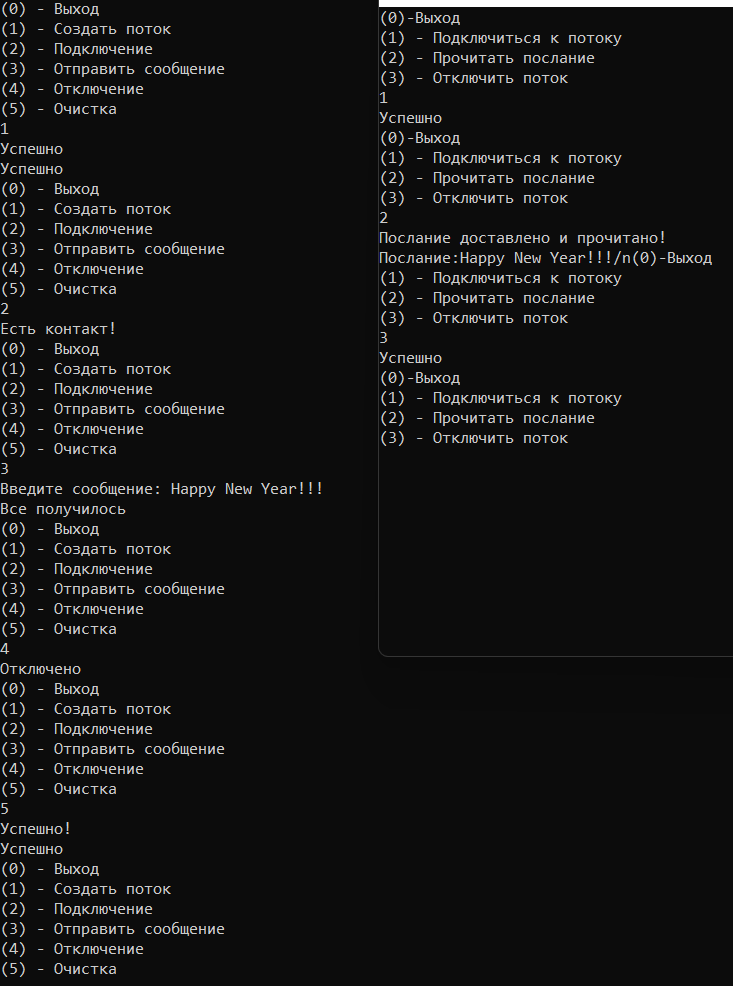


Рисунок 3. Протокол работы программы

## Выводы по заданию 2

В ходе выполнения второго задания лабораторной работы познакомились с именованными каналами. Именованные каналы являются одним из способов межпроцессного взаимодействия, они полезны для организации прямой связи между двумя процессами на одном ПК или в сети. Именованные каналы могут быть однонаправленным или двунаправленным (дуплексным). Поддерживают асинхронный обмен. Полезны для организации прямой связи между двумя процессами на одном ПК или в сети.

# Приложение

## Исходный код

Part2\_pice1.cpp

#include <windows.h>

#include <iostream>

#include <string>

using namespace std;

int main() {

setlocale(LC\_ALL, "Russian");

HANDLE p = INVALID\_HANDLE\_VALUE;

HANDLE e = INVALID\_HANDLE\_VALUE;

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

char mes[256];

string s;

int x = 1, y, i;

bool f, connection = false;

while (x != 0) {

cout << "(0) -Выход\n"<<"1 - Создать поток\n" << "2 - Подключится\n" <<"3 - Отправить сообщение\n" << "4 - Отключение\n" << "5 - Очистка\n" ;

cin >> x;

switch (x) {

case 1:

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

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

cout << "Успешно" << endl;

return -1;

} else

cout << "Успешно" << endl;

s = "\\\\.\\pipe\\pipe1";

p = CreateNamedPipeA(s.c\_str(), PIPE\_ACCESS\_DUPLEX | FILE\_FLAG\_OVERLAPPED,

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

PIPE\_UNLIMITED\_INSTANCES, 256, 256, 0, NULL);

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

cout << "Поток создан\n";

overlappedPipe.e = e;

} else

cout << "Ошибка\n";

break;

case 2:

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

if (connection)

cout << "Подключено\n";

else

cout << "Ошибка подключения\n";

break;

case 3:

if (!connection) cout << "Нет соединения" << endl;

else {

cout << "Введите послание: ";

cin >> ws;

getline(cin, s);

cout << endl;

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

mes[i] = '\0';

overlapped.e = e;

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

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

cout << "Успешно" << endl;

else

cout << "Слишком не разборчиво написано";

}

cout << endl;

break;

case 4:

connection = DisconnectNamedPipe((HANDLE)p);

if (connection)

cout << "Отключено\n";

else

cout << "Отключения не произошло\n";

connection = !connection;

break;

case 5:

if (p == INVALID\_HANDLE\_VALUE)

cout << "Ошибка" << endl;

else {

CloseHandle(p);

p == INVALID\_HANDLE\_VALUE;

cout << "Успешно" << endl;

}

if (e == INVALID\_HANDLE\_VALUE)

cout << "Ошибка" << endl;

else {

CloseHandle(e);

e = INVALID\_HANDLE\_VALUE;

cout << "Успешно" << endl;

}

Break;

}

}

}

Part2\_pice2.cpp

#include <windows.h>

#include <iostream>

#include <string>

using namespace std;

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

cout << "Послание доставлено и прочитано!" << endl;

}

int main() {

setlocale(LC\_ALL, "Russian");

HANDLE hPipe;

//HANDLE hEvent = INVALID\_HANDLE\_VALUE;

OVERLAPPED overlapped;

char \*mes;

string s;

int x = 1, y, i;

bool f, connection = false;

while (x != 0) {

cout <<"(0)-Выход\n" <<"(1) - Подключиться к потоку\n" << "(2) - Прочитать послание\n"<< "(3) - Отключить поток\n";

cin >> x;

switch (x) {

case 1:

s = "\\\\.\\pipe\\pipe1";

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

FILE\_FLAG\_OVERLAPPED, NULL);

if (hPipe != INVALID\_HANDLE\_VALUE)

cout << "Успешно\n";

else

cout << "Ошибка\n";

break;

case 2:

mes = new char[512];

overlapped = OVERLAPPED();

if (!ReadFileEx(hPipe, (LPVOID) mes, 512, &overlapped, ReadCallback)) cout << "Ужасный почерк. Ничего не разобрать\n";

else {

SleepEx(INFINITE, true);

cout << "Послание:" << mes<<"/n";

}

delete[] mes;

break;

case 3:

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

if (!CloseHandle(hPipe))

cout << "Ошибка при отключении\n" ;

else cout << "Успешно\n" ;

hPipe = INVALID\_HANDLE\_VALUE;

}

break;

}

}

}

Part1\_main.cpp

#include <iostream>

#include <fstream>

#include <sstream>

#include <Windows.h>

int main() {

int pageSize = 4096, page = 14, proces = 18;

HANDLE F = CreateFile("fileMapping", GENERIC\_ALL, FILE\_SHARE\_READ | FILE\_SHARE\_WRITE, NULL, OPEN\_ALWAYS, FILE\_ATTRIBUTE\_NORMAL, NULL);

SetFilePointer(F, pageSize \* page, 0, FILE\_BEGIN);

SetEndOfFile(F);

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

HANDLE hFileMapping = CreateFileMapping(F, NULL, PAGE\_READWRITE, 0, 0, "MappedFile");

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

HANDLE writeSemaphore = CreateSemaphore(NULL, page, page, "SemForWrite");

HANDLE readSemaphore = CreateSemaphore(NULL, 0, page, "SemForRead");

HANDLE loggerMutex = CreateMutex(NULL, FALSE, "MutexForLog");

HANDLE arrMutex[page];

std::stringstream nameMX;

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

nameMX << "Mutex\_" << i;

arrMutex[i] = CreateMutex(NULL, FALSE, nameMX.str().c\_str());

}

HANDLE hChildProcess[2 \* proces];

HANDLE hThreadChild[2 \* proces];

std::fstream log;

log.open("Part1.log", std::fstream::app | std::fstream::out);

log.flush();

log.close();

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

STARTUPINFOA si;

ZeroMemory(&si, sizeof(STARTUPINFOA));

si.cb = sizeof(STARTUPINFOA);

PROCESS\_INFORMATION pi;

if (CreateProcess("Part1\_Write.exe", NULL, NULL, NULL, FALSE, NULL, NULL, NULL, &si, &pi))

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

hChildProcess[i] = pip.hProcess;

hThreadChild[i] = pip.hThread;

}

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

STARTUPINFOA si;

ZeroMemory(&si, sizeof(STARTUPINFOA));

si.cb = sizeof(STARTUPINFOA);

PROCESS\_INFORMATION pi;

if (CreateProcess("Part1\_Read.exe", NULL, NULL, NULL, FALSE, NULL, NULL, NULL, &si, &pi))

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

hChildProcess[proces+i] = pip.hProcess;

hThreadChild[procse+i] = pip.hThread;

}

WaitForMultipleObjects(2 \* procCount, hChildProcess, TRUE, INFINITE);

std::cout << "Ready" << std::endl;

std::cin.get();

for (int i = proces; i < 2 \* proces; i++) {

CloseHandle(hThreadChild[i]);

CloseHandle(hChildProcess[i]);

}

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

CloseHandle(arrMutex[i]);

}

CloseHandle(writeSemaphore);

CloseHandle(readSemaphore);

CloseHandle(hFileMapping);

}

else {

std::cout << "Ошибка" << std::endl;

std::cin.get();

}

CloseHandle(hFile);

}

else {

std::cout << "Ошибка" << std::endl;

std::cin.get();

}

return 0;

}

Part1\_read.cpp

#include <iostream>

#include <sstream>

#include <fstream>

#include <Windows.h>

#include <random>

#include <timeapi.h>

int main() {

int pageSize = 4096, pageCount = 14;

srand(time(NULL));

HANDLE F = OpenFileMapping(FILE\_MAP\_ALL\_ACCESS, FALSE, "MappedFile");

std::fstream logFile;

logFile.open("lab4\_1.log", std::fstream::app | std::fstream::out);

logFile.rdbuf()->pubsetbuf(NULL, 0);

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

void\* address = MapViewOfFile(hMappedFile, FILE\_MAP\_READ, 0, 0, 0);

HANDLE writeSemaphore = OpenSemaphore(SEMAPHORE\_ALL\_ACCESS, FALSE, "SemForWrite");

HANDLE readSemaphore = OpenSemaphore(SEMAPHORE\_ALL\_ACCESS, FALSE, "SemForRead");

HANDLE logFileMutex = OpenMutex(MUTEX\_ALL\_ACCESS, FALSE, "MutexForLog");

HANDLE arrMutex[pageCount];

std::stringstream nameMx;

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

nameMx << "Mutex\_" << i;

arrMutex[i] = OpenMutex(MUTEX\_ALL\_ACCESS, FALSE, nameMx.str().c\_str());

}

int ProccessId = GetCurrentProcessId();

WaitForSingleObject(logFileMutex, INFINITE);

logFile << "R\t" << ProccessId << "\tStart\t" << timeGetTime() << std::endl;

logFile.flush();

ReleaseMutex(logFileMutex);

VirtualLock(address, pageSize \* pageCount);

WaitForSingleObject(logFileMutex, INFINITE);

logFile << "R\t" << ProccessId << "\tWait\t" << timeGetTime() << std::endl;

logFile.flush();

ReleaseMutex(logFileMutex);

WaitForSingleObject(readSemaphore, INFINITE);

int index = WaitForMultipleObjects(pageCount, arrMutex, FALSE, INFINITE);

WaitForSingleObject(logFileMutex, INFINITE);

logFile << "R\t" << ProccessId << "\tPage\_" << index << "\t" << timeGetTime() << std::endl;

logFile.flush();

ReleaseMutex(logFileMutex);

unsigned int delay = (rand() % 1000) + 500;

Sleep((DWORD)delay);

WaitForSingleObject(logFileMutex, INFINITE);

logFile << "R\t" << ProccessId << "\tReady\t" << timeGetTime() << std::endl;

logFile.flush();

ReleaseMutex(logFileMutex);

ReleaseMutex(arrMutex[index]);

ReleaseSemaphore(writeSemaphore, 1, NULL);

VirtualUnlock(address, pageSize \* pageCount);

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

CloseHandle(arrMutex[i]);

}

WaitForSingleObject(logFileMutex, INFINITE);

logFile << "R\t" << ProccessId << "\tFinish\t" << timeGetTime() << std::endl;

logFile.flush();

ReleaseMutex(logFileMutex);

UnmapViewOfFile(address);

CloseHandle(hMappedFile);

CloseHandle(writeSemaphore);

CloseHandle(readSemaphore);

}

else {

logFile << "Error Read" << std::endl;

}

logFile.close();

return 0;

}

Part1\_write.cpp

#include <iostream>

#include <sstream>

#include <fstream>

#include <Windows.h>

#include <random>

#include <ctime>

#include <timeapi.h>

int main() {

int pageSize = 4096, pageCount = 14;

srand(time(NULL));

HANDLE hMappedFile = OpenFileMapping(FILE\_MAP\_ALL\_ACCESS, FALSE, "MappedFile");

std::fstream logFile;

logFile.open("lab4\_1.log", std::fstream::app | std::fstream::out);

logFile.rdbuf()->pubsetbuf(NULL, 0);

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

void\* address = MapViewOfFile(hMappedFile, FILE\_MAP\_READ, 0, 0, 0);

HANDLE writeSemaphore = OpenSemaphore(SEMAPHORE\_ALL\_ACCESS, FALSE, "SemForWrite");

HANDLE readSemaphore = OpenSemaphore(SEMAPHORE\_ALL\_ACCESS, FALSE, "SemForRead");

HANDLE loggerMutex = OpenMutex(MUTEX\_ALL\_ACCESS, FALSE, "MutexForLog");

HANDLE arrMutex[pageCount];

std::stringstream name;

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

name << "Mutex\_" << i;

arrMutex[i] = OpenMutex(MUTEX\_ALL\_ACCESS, FALSE, name.str().c\_str());

}

int ProccessId = GetCurrentProcessId();

WaitForSingleObject(loggerMutex, INFINITE);

logFile << "W\t" << ProccessId << "\tStart\t" << timeGetTime() << std::endl;

logFile.flush();

ReleaseMutex(loggerMutex);

VirtualLock(address, pageSize \* pageCount);

WaitForSingleObject(loggerMutex, INFINITE);

logFile << "Writer" <<"time"<< timeGetTime()<<"PID"<< "Wait";

logFile.flush();

ReleaseMutex(loggerMutex);

WaitForSingleObject(writeSemaphore, INFINITE);

int index = WaitForMultipleObjects(pageCount, arrMutex, FALSE, INFINITE);

WaitForSingleObject(loggerMutex, INFINITE);

logFile << "Writer" <<"time"<< timeGetTime()<<"PID"<< ProccessId << "Page\_" << index ;

logFile.flush();

ReleaseMutex(loggerMutex);

unsigned int delay = (rand() % 1000) + 500;

Sleep((DWORD)delay);

WaitForSingleObject(loggerMutex, INFINITE);

logFile << "Writer" <<"time"<< timeGetTime()<<"PID"<< ProccessId << "Ready"; logFile.flush();

ReleaseMutex(loggerMutex);

ReleaseMutex(arrMutex[index]);

ReleaseSemaphore(readSemaphore, 1, NULL);

VirtualUnlock(address, pageSize \* pageCount);

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

CloseHandle(arrMutex[i]);

}

WaitForSingleObject(loggerMutex, INFINITE);

logFile << "Writer" <<"time"<< timeGetTime()<<"PID"<< ProccessId << ProccessId << "\n" ;

logFile.flush();

ReleaseMutex(loggerMutex);

CloseHandle(loggerMutex);

UnmapViewOfFile(address);

CloseHandle(hMappedFile);

CloseHandle(writeSemaphore);

CloseHandle(readSemaphore);

}

else {

logFile << "Error Write" << std::endl;

}

logFile.close();

return 0;

}