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

ОТЧЕТ

по лабораторной работе №4

по дисциплине «Операционные системы»

ТЕМА: МЕЖПРОЦЕССНОЕ ВЗАИМОДЕЙСТВИЕ

Студент гр. 9308 _____ Яловега Н.В.

Преподаватель _____ Тимофеев А.В.

Санкт-Петербург

2021

Введение

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

Задание 1: Реализация решения задачи о читателях-писателях.

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:
  - 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.

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 2: Использование именованных каналов для реализации сетевого межпроцессного взаимодействия.

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. Реализация решения задачи о читателях-писателях.

Для выполнения задания воспользуемся тремя программами: программа-читатель, программа-писатель и программа, запускающая первые две.

Программа для запуска создает все нужные файлы, проекцию файла для работы процессов и запускает сами процессы, затем ожидает остановки всех процессов и освобождает ресурсы.

Мьютекс (англ. *mutex*, от ***mut****ual* ***ex****clusion* — «взаимное исключение») — это базовый механизм синхронизации. Он предназначен для организации взаимоисключающего доступа к общим данным для нескольких потоков с использованием барьеров памяти.

Захват мьютекса: Поток запрашивает *монопольное использование* общих данных, защищаемых мьютексом. Дальше два варианта развития событий: происходит захват мьютекса этим потоком (и в этом случае ни один другой поток не сможет получить доступ к этим данным) или поток блокируется (если мьютекс уже захвачен другим потоком).

Освобождение мьютекса: Когда ресурс больше не нужен, текущий владелец должен вызвать функцию разблокирования, чтобы и другие потоки могли получить доступ к этому ресурсу. Когда мьютекс освобождается, доступ предоставляется одному из ожидающих потоков.

Семафор – это объект, который используется для контроля доступа нескольких потоков до общего ресурса. В общем случае это какая-то переменная, состояние которой изменяется каждым из потоков. Текущее состояние переменной определяет доступ к ресурсам.

Результаты работы программы представлены на рисунках 1-2.

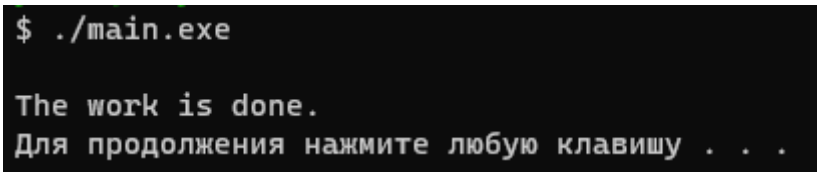


Рисунок 1: Основная программа

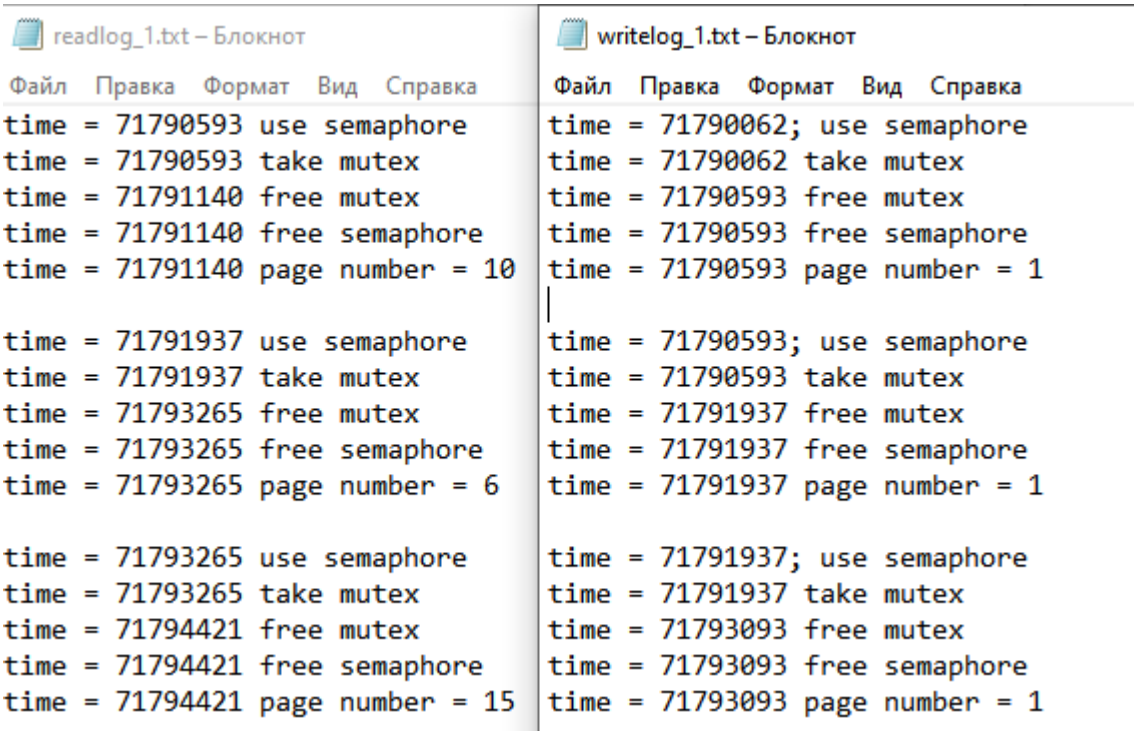
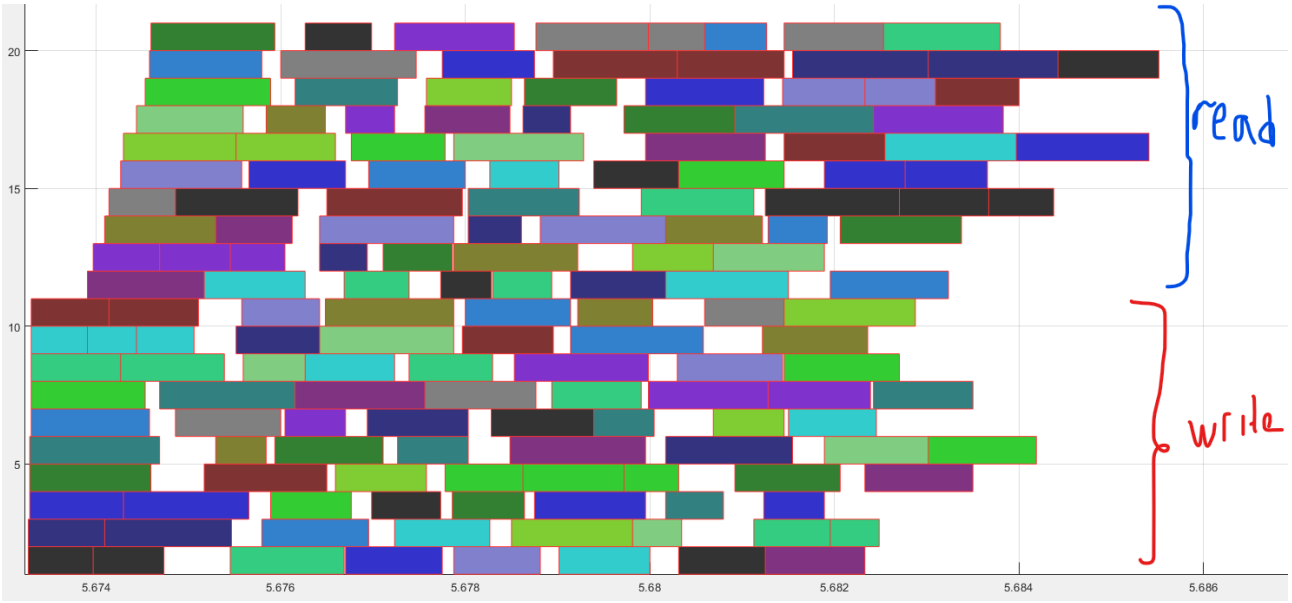


Рисунок 2: Файлы логов программы

Анализ результатов работы программы:

Построим график занятости страниц буферной памяти (проецируемого файла) во времени, используя данные из лог файлов.

Два семафора следят за пустыми и заполненными страницами. Для исключения гонок, будем считать, что запись в страницу и считывание из страницы является критическими секциями, взаимное исключение при доступе, к которым будем обеспечивать мьютексом для каждой страницы. В начале работы программы значение семафора для «чистых» страниц становится равно N, в то время как для «использованных» - 0. Таким образом писатели могут сразу приступать к работе – читатели ожидают появления «использованных» страниц. Писатель ожидает сигнала от семафора «чистых страниц» - так он понимает, что есть страница, которую он может записать. Затем писатель выполняет необходимые действия. Наконец, выполняется ReleaseSemaphore. Работа писателя завершается. Для читателей ситуация аналогичная, только ожидают они «использованные» страницы.



На оси абсцисс показано время в миллисекундах, а по горизонтали — процессы. При этом в нижней половине — это писатели, а в верхней — читатели. Каждой странице соответствует свой цвет. Согласно варианту страниц должно быть 17, соответственно будет и 17 разных цветов.

Так как процессов больше, чем страниц, (в нашем случае количество страниц — 17, процессов-писателей — 10, процессов-читателей — 10), то некоторым нужно ждать (на графике есть пробелов между блоками).

Выводы по заданию

В данном задании были использованы такие объекты синхронизации, как семафоры и мьютексы, так как синхронизацию нужно выполнять между процессами разных потоков.

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

Для использования именованных каналов для реализации сетевого межпроцессного взаимодействия было создано две программы – сервер и клиент.

Сначала необходимо запустить программу-сервер и создать именованный канал. Далее программа ожидает подключение клиента. После подключения сервер может передавать сообщения клиенту.

Для установления соединения между программами используется именованный канал (named pipe).

Для создания именованного канала Pipe можно использовать функцию CreateNamedPipe. Канал может использоваться как для записи в него данных, так и для чтения.

После того как серверный процесс создал канал, он может перейти в режим соединения с клиентским процессом. Соединение со стороны сервера выполняется с помощью функции ConnectNamedPipe.

Результаты работы программы представлены на рисунках ниже.

```
$ ./server.exe
Enter message (:q to exit): Hello world!
Message was sent!
Enter message (:q to exit): Test os/lab4
Message was sent!
Enter message (:q to exit): Akjsdhf;kjasdhf;kjashdfk;jasdhfkjashdflkajshdfklajshfAkjsdhf;kjasdhf;kjashdfk;jasdhfkjashdfl
kajshdfklajshfAkjsdhf;kjasdhf;kjashdfk;jasdhfkjashdflkajshdfklajshfAkjsdhf;kjasdhf;kjashdfk;jasdhfkjashdflkajshdfklajshf
Akjsdhf;kjasdhf;kjashdfk;jasdhfkjashdflkajshdfklajshfAkjsdhf;kjasdhf;kjashdfk;jasdhfkjashdflkajshdfklajshfAkjsdhf;kjasdh
f;kjashdfk;jasdhfkjashdflkajshdfklajshfAkjsdhf;kjasdhf;kjashdfk;jasdhfkjashdflkajshdfklajshfAkjsdhf;kjasdhf;kjashdfk;jas
dhfkjashdflkajshdfklajshfAkjsdhf;kjasdhf;kjashdfk;jasdhfkjashdflkajshdfklajshfAkjsdhf;kjasdhf;kjashdfk;jasdhfkjashdflkaj
shdfklajshfAkjsdhf;kjasdhf;kjashdfk;jasdhfkjashdflkajshdfklajshfAkjsdhf;kjasdhf;kjashdfk;jasdhfkjashdflkajshdfklajshfAkj
sdhf;kjasdhf;kjashdfk;jasdhfkjashdflkajshdfklajshfAkjsdhf;kjasdhf;kjashdfk;jasdhfkjashdflkajshdfklajshfAkjsdhf;kjasdhf;k
jashdfk;jasdhfkjashdflkajshdfklajshfAkjsdhf;kjasdhf;kjashdfk;jasdhfkjashdflkajshdfklajshfAkjsdhf;kjasdhf;kjashdfk;jasdhf
kjashdflkajshdfklajshfAkjsdhf;kjasdhf;kjashdfk;jasdhfkjashdflkajshdfklajshfAkjsdhf;kjasdhf;kjashdfk;jasdhfkjashdflkajshd
fklajshfAkjsdhf;kjasdhf;kjashdfk;jasdhfkjashdflkajshdfklajshfAkjsdhf;kjasdhf;kjashdfk;jasdhfkjashdflkajshdfklajshfAkjsdh
f;kjasdhf;kjashdfk;jasdhfkjashdflkajshdfklajshf
Error: Message more than 1024 bytes.
Enter message (:q to exit): :q
Message was sent!
```

Рисунок 4: Программа-сервер

```
$ ./client.exe
Message: "Hello world!".
Message: "Test os/lab4".
Message: ":q".
```

Рисунок 5: Программа-клиент

Выводы по заданию

Для реализации сетевого межпроцессного взаимодействия был использован такой объект, как именованный канал.

Программа-сервер создаёт именованный канал с заданным названием и ожидает подключения клиента.

Затем запускается клиент и находит по названию нужный канал – соединение установлено.

По каналу сервер передаёт сообщение клиенту, которое выводится на экран. В любой момент можно отключить соединение и завершить работу программ.

Также можно заметить, что с помощью этих каналов можно синхронизировать процессы.

Вывод

Были исследованы инструменты и механизмы взаимодействия процессов в Windows.

Приложения

Задание 1

main.cpp

```
#include <iostream>
#include <string>
#include <windows.h>
#include <cstdlib>
#include <ctime>
#include <vector>

const int PAGESIZE = 4096;
const int PAGECOUNT = 17;
const int N = PAGECOUNT - 1;
const int WRITERCOUNT = 10;
const int READERCOUNT = 10;
const std::string MUTEXNAME ="mutex_a";

int main()
{
   srand(time(nullptr));
   int fileSize = PAGECOUNT * PAGESIZE;
   HANDLE freeSem = CreateSemaphore(NULL, N, N, "freeSem");
   HANDLE usedSem = CreateSemaphore(NULL, 0, N, "usedSem");
   std::vector<HANDLE> pageMutex;

   for(int i = 0; i < PAGECOUNT; i++)
   {
      std::string mutexName = MUTEXNAME + std::to_string(i);
      pageMutex.push_back(CreateMutexA(nullptr, false, mutexName.c_str()));
   }
```

```
    HANDLE fHandle = CreateFileA("file.txt", GENERIC_READ |
GENERIC_WRITE,
        FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, CREATE_ALWAYS,
FILE_ATTRIBUTE_NORMAL, NULL);

    SetFilePointer(fHandle, fileSize, 0, FILE_BEGIN);
    SetEndOfFile(fHandle);

    HANDLE mapHandle = CreateFileMapping(fHandle, NULL,
PAGE_READWRITE, 0, 0, (LPCTSTR)"filemap");

    LPVOID mapView = MapViewOfFile(mapHandle, FILE_MAP_ALL_ACCESS,
0, 0, fileSize);

    VirtualLock(mapView, fileSize);

    HANDLE processHandles[WRITERCOUNT+READERCOUNT];

    for (int i = 0; i < WRITERCOUNT; ++i)
    {
        std::string logName = "writelog_" + std::to_string(i + 1) + ".txt";

        STARTUPINFO sysInfo;
        PROCESS_INFORMATION procInfo;
        SECURITY_ATTRIBUTES secureAttr = { sizeof(secureAttr), NULL, TRUE };

        ZeroMemory(&sysInfo, sizeof(sysInfo));

        HANDLE outHandle = CreateFile((LPCTSTR)logName.c_str(),
GENERIC_WRITE, FILE_SHARE_WRITE, &secureAttr,
            CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
```

```
    sysInfo.cb = sizeof(sysInfo);
    sysInfo.hStdOutput = outHandle;
    sysInfo.hStdError = NULL;
    sysInfo.hStdInput = NULL;
    sysInfo.dwFlags |= STARTF_USESTDHANDLES;

    ZeroMemory(&procInfo, sizeof(procInfo));

    int mainProcess = CreateProcess((LPCTSTR)"writer.exe",
        NULL,
        NULL,
        NULL,
        TRUE,
        0,
        NULL,
        NULL,
        &sysInfo,
        &procInfo);

    if (mainProcess)
        processHandles[i] = procInfo.hProcess;
    // Sleep(1);
}

for (int i = 0; i < READERCOUNT; ++i)
{
    std::string fname = "readlog_" + std::to_string(i + 1) + ".txt";

    STARTUPINFO sysInfo;
    PROCESS_INFORMATION procInfo;
    SECURITY_ATTRIBUTES secureAttr = { sizeof(secureAttr), NULL, TRUE };
```

```
        ZeroMemory(&sysInfo, sizeof(sysInfo));

        sysInfo.hStdOutput = CreateFile((LPCTSTR)fname.c_str(), GENERIC_WRITE,
FILE_SHARE_WRITE, &secureAttr,
            CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

        sysInfo.cb = sizeof(sysInfo);
        sysInfo.hStdError = NULL;
        sysInfo.hStdInput = NULL;
        sysInfo.dwFlags |= STARTF_USESTDHANDLES;
        ZeroMemory(&procInfo, sizeof(procInfo));

        int createProcess = CreateProcess((LPCTSTR)"reader.exe",
            NULL,
            NULL,
            NULL,
            TRUE,
            0,
            NULL,
            NULL,
            &sysInfo,
            &procInfo);

        if (createProcess != 0)
            processHandles[WRITERCOUNT + i] = procInfo.hProcess;

    }

    WaitForMultipleObjects(WRITERCOUNT+READERCOUNT, processHandles,
true, INFINITE);
```

```
    for (int i = 0; i < WRITERCOUNT+READERCOUNT; ++i)
        CloseHandle(processHandles[i]);

    CloseHandle(mapHandle);
    UnmapViewOfFile(mapView);
    CloseHandle(fHandle);
    for(int i = 0; i < PAGECOUNT; i++)
        CloseHandle(pageMutex[i]);
    CloseHandle(freeSem);
    CloseHandle(usedSem);

    // std::cout << GetTickCount()<<e
    std::cout << std::endl << "The work is done." << std::endl;
    system("pause");
}
```

writer.cpp

```
#include <iostream>
#include <string>
#include <ctime>
#include <cstdlib>
#include <windows.h>
#include <vector>
const std::string MUTEXNAME ="mutex_a";

int main()
{
   srand(time(nullptr) + GetCurrentProcessId()*197);
   HANDLE freeSem = OpenSemaphore(SYNCHRONIZE |
SEMAPHORE_MODIFY_STATE, FALSE, (LPCTSTR)"freeSem");
   HANDLE usedSem = OpenSemaphore(SYNCHRONIZE |
SEMAPHORE_MODIFY_STATE, FALSE, (LPCTSTR)"usedSem");
   HANDLE mutex;
   std::string mutexName = "mutex_a";
   DWORD waitResult;
   HANDLE hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
   DWORD written = 0;
   std::string outputString = "";
   HANDLE mappingHandle = OpenFileMapping(FILE_MAP_READ |
FILE_MAP_WRITE, FALSE, (LPCTSTR)"filemap");
   if (mappingHandle)
   {
      for (int i = 0; i < 8; ++i)
      {
         LONG page = -1;
         WaitForSingleObject(freeSem, INFINITE);
         outputString = "time = " + std::to_string(GetTickCount()) + "; use semaphore\
n";
         outputString = std::to_string(GetTickCount()) + "\n";
         WriteFile(hStdout, outputString.data(), outputString.length(), &written,
NULL);

         do {
            page += 1;
            mutexName = MUTEXNAME + std::to_string(page);
            mutex = OpenMutexA(SYNCHRONIZE, false, mutexName.c_str());
            waitResult = WaitForSingleObject(mutex, 0);
         } while (waitResult == WAIT_TIMEOUT);

         outputString = "time = " + std::to_string(GetTickCount()) + "; take mutex\n";
         WriteFile(hStdout, outputString.data(), outputString.length(), &written,
NULL);
```

```
            Sleep(rand() % 1000 + 500); // fake write

            if (ReleaseMutex(mutex))
            {
                outputString = "time = " + std::to_string(GetTickCount()) + "; free mutex\
n";
                WriteFile(hStdout, outputString.data(), outputString.length(), &written,
NULL);
            }
            else
            {
                std::string str = std::to_string(GetLastError()) + " code\n";
                WriteFile(hStdout, str.data(), str.length(), &written, NULL);
            }

            if (ReleaseSemaphore(usedSem, 1, nullptr))
            {
                outputString = "time = " + std::to_string(GetTickCount()) + "; free
semaphore\n";
                WriteFile(hStdout, outputString.data(), outputString.length(), &written,
NULL);
                std::string str = "time = " + std::to_string(GetTickCount()) + "; page
number = " + std::to_string(page + 1) + "\n\n";
                std::string str = std::to_string(GetTickCount()) + " " + std::to_string(page +
1) + "\n\n";
                WriteFile(hStdout, str.data(), str.length(), &written, NULL);
            }
            else
            {
                std::string str = std::to_string(GetLastError()) + " code\n";
                WriteFile(hStdout, str.data(), str.length(), &written, NULL);
            }
        }
    }
    else
    {
        WriteFile(hStdout, "Mapping creation failed\n", strlen("Mapping creation failed\
n"), &written, NULL);
    }

    CloseHandle(hStdout);

    return 0;
}
```

reader.cpp

```
#include <iostream>
#include <string>
#include <ctime>
#include <cstdlib>
#include <windows.h>
#include <vector>
const std::string MUTEXNAME ="mutex_a";


int main()
{
   srand(time(nullptr) + GetCurrentProcessId()*197);
   HANDLE freeSem = OpenSemaphore(SYNCHRONIZE |
SEMAPHORE_MODIFY_STATE, FALSE, (LPCTSTR)"freeSem");
   HANDLE usedSem = OpenSemaphore(SYNCHRONIZE |
SEMAPHORE_MODIFY_STATE, FALSE, (LPCTSTR)"usedSem");
   HANDLE mutex;
   DWORD waitResult;
   HANDLE hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
   DWORD written = 0;
   std::string outputString = "";

   HANDLE mapHandle = OpenFileMapping(FILE_MAP_READ |
FILE_MAP_WRITE, FALSE, "filemap");
   if (mapHandle)
   {
      for (int i = 0; i < 8; ++i)
      {
         LONG page = -1;

         WaitForSingleObject(usedSem, INFINITE);
         outputString = "time = " + std::to_string(GetTickCount()) + "; use semaphore\
n";
         outputString = std::to_string(GetTickCount()) + "\n";
         WriteFile(hStdout, outputString.data(), outputString.length(), &written,
NULL);

         do {
            page += 1;
            std::string mutexName = MUTEXNAME + std::to_string(page);
            mutex = OpenMutexA(SYNCHRONIZE, false, mutexName.c_str());
            waitResult = WaitForSingleObject(mutex, 0);
         } while (waitResult == WAIT_TIMEOUT);

         outputString = "time = " + std::to_string(GetTickCount()) + "; take mutex\n";
```

```
            WriteFile(hStdout, outputString.data(), outputString.length(), &written,
NULL);

            Sleep(rand() % 1000 + 500); // fake read

            if (ReleaseMutex(mutex))
            {
                outputString = "time = " + std::to_string(GetTickCount()) + "; free mutex\
n";
                WriteFile(hStdout, outputString.data(), outputString.length(), &written,
NULL);
            }
            else
            {
                std::string errorString = std::to_string(GetLastError()) + " code\n";
                WriteFile(hStdout, errorString.data(), errorString.length(), &written,
NULL);
            }

            if (ReleaseSemaphore(freeSem, 1, nullptr))
            {
                outputString = "time = " + std::to_string(GetTickCount()) + "; free
semaphore\n";
                WriteFile(hStdout, outputString.data(), outputString.length(), &written,
NULL);
                std::string str = "time = " + std::to_string(GetTickCount()) + "; page
number = " + std::to_string(page + 1) + "\n\n";
                std::string str = std::to_string(GetTickCount()) + " " + std::to_string(page +
1) + "\n\n";

                WriteFile(hStdout, str.data(), str.length(), &written, NULL);
            }
            else
            {
                std::string errorString = std::to_string(GetLastError()) + " code\n";
                WriteFile(hStdout, errorString.data(), errorString.length(), &written,
NULL);
            }
        }
    }
    else
    {
        WriteFile(hStdout, "Mapping creation failed\n", strlen("Mapping creation failed\
n"), &written, NULL);
    }
```

```
    CloseHandle(hStdout);

    return 0;
}
```

Задание 2

server.cpp

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

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

int main()
{
   size_t i;
   HANDLE 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 != INVALID_HANDLE_VALUE)
   {
      if(ConnectNamedPipe(hPipe, NULL))
      {
         OVERLAPPED over;
         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;
        }

        DisconnectNamedPipe(hPipe);
        CloseHandle(over.hEvent);
    }
    else
    {
        std::cout << "Error: Can't connect to pipe." << std::endl;
        return GetLastError();
    }

    CloseHandle(hPipe);
}
else
{
    std::cout << "Error: Can't create pipe." << std::endl;
    return GetLastError();
}

return 0;
}
```

client.cpp

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

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

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

int main()
{
   WaitNamedPipeA(PIPE_NAME.c_str(), NMPWAIT_WAIT_FOREVER);

   HANDLE hPipe = CreateFileA(PIPE_NAME.c_str(), GENERIC_READ, 0,
NULL, OPEN_EXISTING, FILE_FLAG_OVERLAPPED |
FILE_FLAG_NO_BUFFERING, NULL);

   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;
        return GetLastError();
    }

    return 0;
}
```