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

ОТЧЕТ

по лабораторной работе №2

по дисциплине «Операционные системы»

ТЕМА: УПРАВЛЕНИЕ ПАМЯТЬЮ

Студент гр. 9308 _____ Яловега Н.В.

Преподаватель _____ Тимофеев А.В.

Санкт-Петербург

2021

Введение

Цель работы: исследовать механизмы управления виртуальной памятью Win32.

Задание:

1. Исследовать виртуальное адресное пространство процесса.

- Создайте консольное приложение с меню (каждая выполняемая функция и/или операция должна быть доступна по отдельному пункту меню), которое выполняет:

– получение информации о вычислительной системе (функция Win32 API – GetSystemInfo);

– определение статуса виртуальной памяти (функция Win32 API – GlobalMemoryStatus);

– определение состояния конкретного участка памяти по заданному с клавиатуры адресу (функция Win32 API – VirtualQuery);

– резервирование региона в автоматическом режиме и в режиме ввода адреса начала региона (функция Win32 API – VirtualAlloc);

– резервирование региона и передача ему физической памяти в автоматическом режиме и в режиме ввода адреса начала региона (функция Win32 API – VirtualAlloc);

– запись данных в ячейки памяти по заданным с клавиатуры адресам;

– установку защиты доступа для заданного (с клавиатуры) региона памяти и ее проверку (функция Win32 API – VirtualProtect);

– возврат физической памяти и освобождение региона адресного пространства заданного (с клавиатуры) региона памяти (функция Win32 API –VirtualFree).

- Запустите приложение и проверьте его работоспособность на нескольких наборах вводимых данных. Запротоколируйте результаты в отчет. Дайте

свои комментарии в отчете относительно выполнения функций Win32 API.

- Подготовьте итоговый отчет с развернутыми выводами по заданию.

2. Использование проецируемых файлов для обмена данными между процессами.

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

– приложение-писатель создает проецируемый файл (функцииWin32 API – CreateFile, CreateFileMapping), проецирует фрагмент файла в память (функции Win32 API – MapViewOfFile, UnmapViewOfFile), осуществляет ввод данных с клавиатуры и их запись в спроецированный файл;

– приложение-читатель открывает проецируемый файл (функция Win32 API – OpenFileMapping), проецирует фрагмент файла в память (функции Win32 API – MapViewOfFile, UnmapViewOfFile), считывает содержимое из спроецированного файла и отображает на экран.

- Запустите приложения и проверьте обмен данных между процессами, удостоверьтесь в надлежащем выполнении задания.Запротоколируйте результаты в отчет. Дайте свои комментарии в отчете относительно выполнения функций Win32 API.
- Подготовьте итоговый отчет с развернутыми выводами по заданию.

Задание 1. Исследование виртуального адресного пространства процесса.

Главное меню программы:

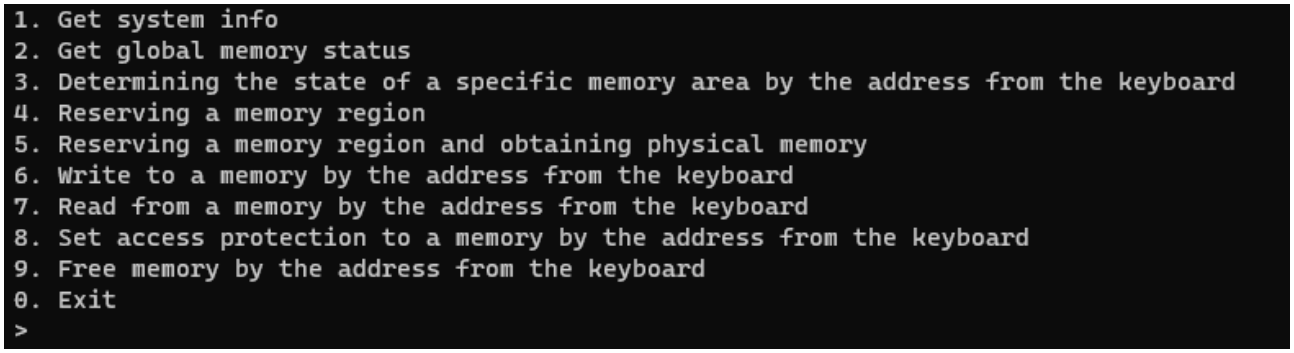

Рисунок 1: Главное меню

Получение информации о вычислительной системе:

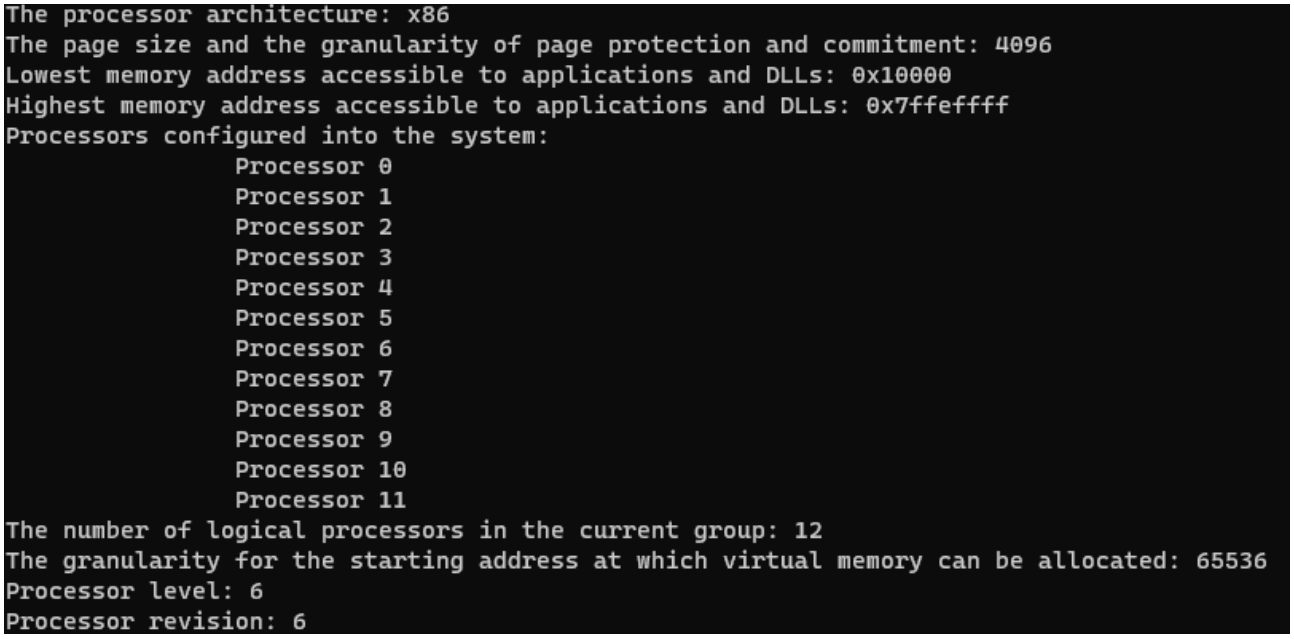

Рисунок 2: Получение информации о вычислительной системе

Определение статуса виртуальной памяти:

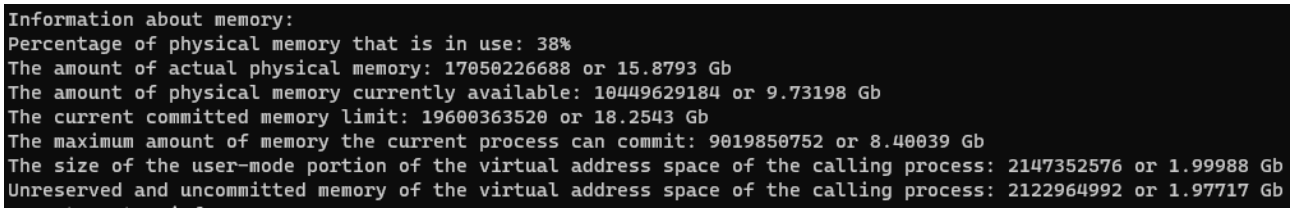

Рисунок 3: Определение статуса виртуальной памяти

Резервирование региона в автоматическом режиме:

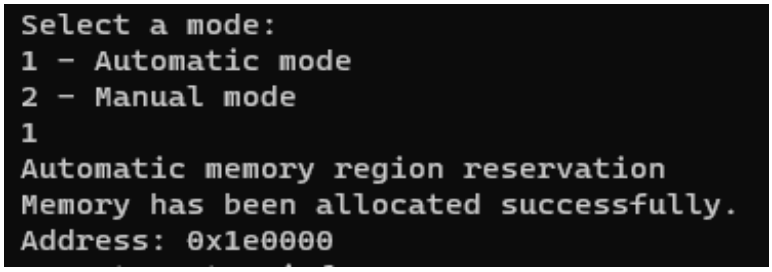

Рисунок 4: Резервирование в автоматическом режиме

Резервирование региона и передача ему физической памяти в режиме ввода адреса начала региона:

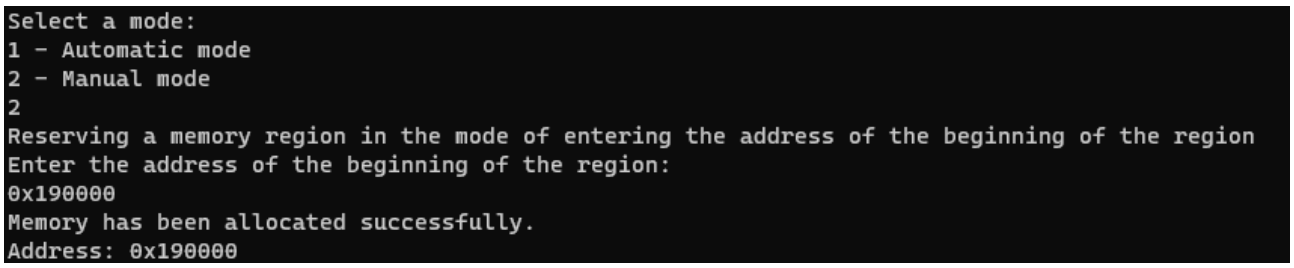

Рисунок 5: Резервирование региона и передача ему физической памяти в режиме ввода адреса начала

Определение состояния конкретного участка памяти (по адресу):

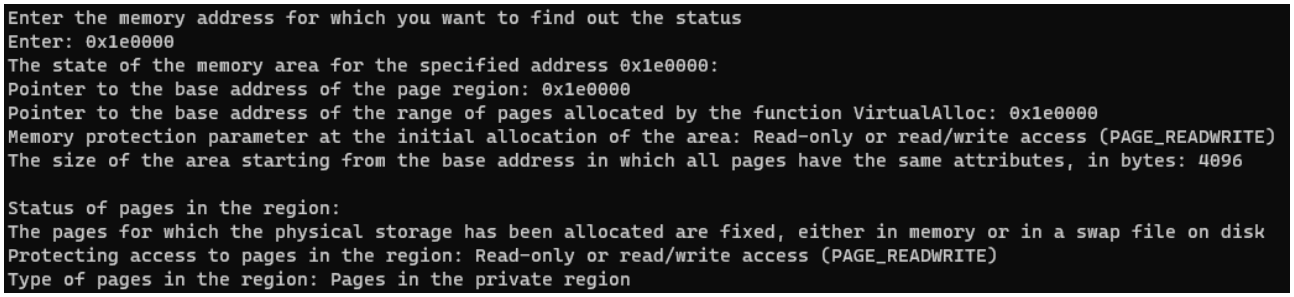

Рисунок 6: Определение состояния конкретного участка памяти по адресу 0x1e0000

Запись данных в ячейки памяти по заданным с клавиатуры адресам:

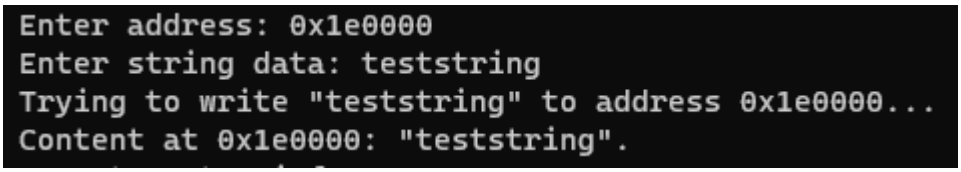

Рисунок 7: Запись данных в ячейки памяти по адресу 0x1e0000

Чтение данных из ячейки памяти по заданным с клавиатуры адресам:

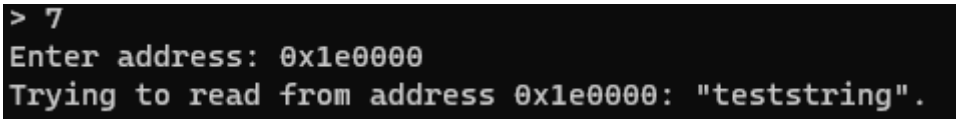

Рисунок 8: Чтение данных из ячейки памяти по адресу 0x1e0000

Установка защиты доступа для заданного региона памяти и её проверка:

```
Enter address: 0x1e0000
Choose memory protection constant:
1. PAGE_EXECUTE
2. PAGE_EXECUTE_READ
3. PAGE_EXECUTE_READWRITE
4. PAGE_EXECUTE_WRITECOPY
5. PAGE_NOACCESS
6. PAGE_READONLY
7. PAGE_READWRITE
8. PAGE_WRITECOPY
6
New protection level: Read-only access (PAGE_READONLY)

Old protection level:
Read-only or read/write access (PAGE_READWRITE)
```

Рисунок 9: Установка защиты доступа для 0x1e0000 и её проверка

Возврат физической памяти и освобождение региона адресного пространства заданного региона памяти:

```
> 9
Enter address: 0x1e0000
Success
```

Рисунок 10: Возврат физической памяти и освобождение 0x1e0000

Выводы по заданию

Виртуальное адресное пространство (ВАП) для определённого процесса — набор адресов виртуальной памяти, который можно использовать этому процессу. ВАП у каждого процесса приватное и не может быть доступно другим процессам без соответствующего разрешения. Виртуальный адрес — это не адрес в физической памяти. Для каждого процесса существует таблица страниц, за которую отвечает система. С помощью этой таблицы система преобразует виртуальные адреса в соответствующие физические.

Как работает виртуальная память:

- занимаемая процессором память разбивается на несколько страниц
- логический адрес, к которому обращается процесс, динамически транслируется в физический адрес;
- если страница не находится в физической памяти, необходимо совершить подкачку с диска;

Виртуальное адресное пространство сильно больше, чем реально доступная физическая память. Резервирование региона и передача ему физической памяти работает правильно, если регион уже не был до этого зарезервирован, при этом нужно учитывать, что указанный адрес округлится до ближайшего числа, "кратного" размеру страницы (в данном случае 4096). Чтобы автоматически зарезервировать регион (и передать ему физическую память) нужно передать первым аргументом NULL в функцию VirtualAlloc. Изменение защиты поменяет начало этого региона памяти, но при этом указатель на "base address of a range of pages allocated by the VirtualAlloc" не изменится. Доступ к региону можно будет иметь только в соответствии с параметрами доступа, иначе поведение программы непредсказуемо. При освобождении памяти необходимо учитывать, что освобождаться будет регион, который был изначально выделен функцией VirtualAlloc.

Задание 2. Использование проецируемых файлов для обмена данными между процессами.

Для этой цели были реализованы 2 программы: программа-писатель и программа-читатель. Первой должна отработать программа-писатель, которая произведёт запись в спроецированный файл, после этого программа-читатель может считать данные из этого спроецированного файла. Пример работы:

Запустим программу-писатель. Введём имя файла. Введём имя отображения. Введём данные, который будут записаны в файл.

```
Enter file name: test
Enter map name: testmap
Trying to write to 0x150000:
Corrupti quia facilis commodi tempora magni voluptatibus ut. Ullam nobis consequatur qui ut officiis adipisci quisquam.
Qui sed ipsum iste. Eveniet autem et aliquam optio itaque amet.Ut id culpa exercitationem debitis est quae. Est laudanti
um ea dignissimos. Et vel asperiores natus ad.Inventore alias quas dolores id laudantium incidunt nihil. Recusandae dele
ctus et enim delectus. Qui vitae iste ad dolorem sint consequatur alias. Deserunt quis dolor laudantium voluptatem. Ulla
m esse molestiae perspiciatis rerum qui ipsa ut omnis.Pariatur molestias quos nam deleniti mollitia laborum rerum. Dicta
 debitis vitae consectetur vero. Autem nesciunt quia ut ad. Sit quo rerum at sapiente saepe est ratione.Officiis quis vo
luptate debitis voluptatem dolorem. Maiores repellendus rem modi mollitia rem voluptatibus saepe. Occaecati ipsa fugiat
sed. Et et magnam suscipit nam tempora tempore asperiores.
Run reader program to continue...
To exit from writer program enter 1.
>
```

Рисунок 11: Работа программы-писателя

Теперь запустим программу-читатель. Введём имя отображения (оно должно совпадать с именем отображения из программы-писателя). Выведем информацию, введённую в программе-писателе.

```
Enter map name: testmap
Trying to read from 0x1a0000:
Corrupti quia facilis commodi tempora magni voluptatibus ut. Ullam nobis consequatur qui ut officiis adipisci quisquam.
Qui sed ipsum iste. Eveniet autem et aliquam optio itaque amet.Ut id culpa exercitationem debitis est quae. Est laudanti
um ea dignissimos. Et vel asperiores natus ad.Inventore alias quas dolores id laudantium incidunt nihil. Recusandae dele
ctus et enim delectus. Qui vitae iste ad dolorem sint consequatur alias. Deserunt quis dolor laudantium voluptatem. Ulla
m esse molestiae perspiciatis rerum qui ipsa ut omnis.Pariatur molestias quos nam deleniti mollitia laborum rerum. Dicta
 debitis vitae consectetur vero. Autem nesciunt quia ut ad. Sit quo rerum at sapiente saepe est ratione.Officiis quis vo
luptate debitis voluptatem dolorem. Maiores repellendus rem modi mollitia rem voluptatibus saepe. Occaecati ipsa fugiat
sed. Et et magnam suscipit nam tempora tempore asperiores.
```

Рисунок 12: Работа программы-читателя

Выводы по заданию

Проецируемый файл — это способ работы с файлами, при котором этому проецируемому файлу ставится в соответствие определённый участок памяти. Чтение из этого выделенного участка памяти приводит к чтению из отображаемого файла и запись по адресам этого участка памяти приводит к записи в проецируемый файл. С помощью проецируемого файла можно организовать передачу данных между процессами.

Различие между виртуальной памятью и проецируемыми файлами состоит в том, что в последнем случае физическая память не выделяется из системного страничного файла, а берется из файла, уже находящегося на диске.

Имя объекта отображения должно быть уникальным и известным обеим программам.

Адреса проекций в программе-писателе и программе-читателе различаются. Происходит это потому, что каждая из программ проецирует созданный объект на своё ВАП и используют эту часть адресного пространства как разделяемую область данных.

Вывод

В ходе выполнения лабораторной работы были исследованы некоторые механизмы управления виртуальной памятью Win32, в частности были изучены некоторые способы работы с виртуальным адресным пространством процесса и был использован проецируемый файл для обмена данными между двумя разными процессами.

Приложения

task1.cpp

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

void protectInfo(DWORD);
void freeVirtualMem();
void setVirtualProtect();
void writeToRegion();
void readRegion();
void memoryReserveAndCommit(SYSTEM_INFO);
void memoryReserve(SYSTEM_INFO);
void virtualQuery();
void getGlobalMemStatus();
void getSystemInfo(SYSTEM_INFO);

int main()
{
	int menuItem;
	SYSTEM_INFO sysInfo;
	GetSystemInfo(&sysInfo);
	do {
		std::cout << "1. Get system info\n";
		std::cout << "2. Get global memory status\n";
		std::cout << "3. Determining the state of a specific memory area by the address from
the keyboard\n";
		std::cout << "4. Reserving a memory region\n";
		std::cout << "5. Reserving a memory region and obtaining physical memory\n";
		std::cout << "6. Write to a memory by the address from the keyboard\n";
		std::cout << "7. Read from a memory by the address from the keyboard\n";
		std::cout << "8. Set access protection to a memory by the address from the
keyboard \n";
		std::cout << "9. Free memory by the address from the keyboard \n";
		std::cout << "0. Exit\n";
		std::cout << "> ";
		std::cin >> menuItem;
		switch (menuItem)
		{
		case 1:
			getSystemInfo(sysInfo);
			break;
		case 2:
			getGlobalMemStatus();
			break;
		case 3:
			virtualQuery();
			break;
		case 4:
			memoryReserve(sysInfo);
			break;
		case 5:
```

```
                        memoryReserveAndCommit(sysInfo);
                        break;
                case 6:
                        writeToRegion();
                        break;
                case 7:
                        readRegion();
                        break;
                case 8:
                        setVirtualProtect();
                        break;
                case 9:
                        freeVirtualMem();
                        break;
                case 0:
                        break;
                default:
                        std::cout << "Error. Unknown menu item" << std::endl;
                        break;

                }
        }
        while (menuItem != 0);

        return 0;
}

void getSystemInfo(SYSTEM_INFO sysInfo)
{
   GetSystemInfo(&sysInfo);

   std::cout << "The processor architecture: ";
        if (sysInfo.wProcessorArchitecture == PROCESSOR_ARCHITECTURE_AMD64)
                std::cout << "x64 (AMD or INTEL)" << std::endl;
        if (sysInfo.wProcessorArchitecture == PROCESSOR_ARCHITECTURE_IA64 )
                std::cout << "Intel Itanium Processor Family (IPF)" << std::endl;
        if (sysInfo.wProcessorArchitecture == PROCESSOR_ARCHITECTURE_INTEL)
                std::cout << "x86" << std::endl;
        if (sysInfo.wProcessorArchitecture == PROCESSOR_ARCHITECTURE_ARM)
                std::cout << "ARM" << std::endl;
        if (sysInfo.wProcessorArchitecture == PROCESSOR_ARCHITECTURE_UNKNOWN)
                std::cout << "Unknown architecture" << std::endl;

   std::cout << "The page size and the granularity of page protection and commitment: " <<
sysInfo.dwPageSize << std::endl;
   std::cout << "Lowest memory address accessible to applications and DLLs: " <<
sysInfo.lpMinimumApplicationAddress << std::endl;
   std::cout << "Highest memory address accessible to applications and DLLs: " <<
sysInfo.lpMaximumApplicationAddress << std::endl;
   std::cout << "Processors configured into the system: \n";
        for (DWORD i = 0; i < sysInfo.dwNumberOfProcessors; i++)
        {
                if (sysInfo.dwActiveProcessorMask & (1 << i))
```

```
                        std::cout << "\t\tProcessor " << i << std::endl;
        }
    std::cout << "The number of logical processors in the current group: " <<
(sysInfo.dwNumberOfProcessors) << std::endl;

    std::cout << "The granularity for the starting address at which virtual memory can be allocated: "
<< sysInfo.dwAllocationGranularity << std::endl;
    std::cout << "Processor level: " << (sysInfo.wProcessorLevel) << std::endl;
    std::cout << "Processor revision: " << (sysInfo.wProcessorLevel) << std::endl;
}

void getGlobalMemStatus()
{
    MEMORYSTATUSEX memStatus;
        memStatus.dwLength = sizeof(memStatus);
        GlobalMemoryStatusEx(&memStatus);
        std::cout << "Information about memory:" << std::endl;
        std::cout << "Percentage of physical memory that is in use: " <<
memStatus.dwMemoryLoad << "%"<< std::endl;
        std::cout << "The amount of actual physical memory: " << memStatus.ullTotalPhys <<" or
"<< (LONGLONG)memStatus.ullTotalPhys/1024.0/1024.0/1024.0<<" Gb\n";
        std::cout << "The amount of physical memory currently available: " <<
memStatus.ullAvailPhys << " or " << (LONGLONG)memStatus.ullAvailPhys / 1024.0 / 1024.0 /
1024.0 << " Gb\n";
        std::cout << "The current committed memory limit: " << memStatus.ullTotalPageFile << "
or " << (LONGLONG)memStatus.ullTotalPageFile / 1024.0 / 1024.0 / 1024.0 << " Gb\n";
        std::cout << "The maximum amount of memory the current process can commit: " <<
memStatus.ullAvailPageFile << " or " << (LONGLONG)memStatus.ullAvailPageFile / 1024.0 /
1024.0 / 1024.0 << " Gb\n";
        std::cout << "The size of the user-mode portion of the virtual address space of the calling
process: " << memStatus.ullTotalVirtual << " or " << (LONGLONG)memStatus.ullTotalVirtual /
1024.0 / 1024.0 / 1024.0 << " Gb\n";
        std::cout << "Unreserved and uncommitted memory of the virtual address space of the
calling process: " << memStatus.ullAvailVirtual << " or " <<
(LONGLONG)memStatus.ullAvailVirtual / 1024.0 / 1024.0 / 1024.0 << " Gb\n";
}

void virtualQuery()
{
    SIZE_T S;
        MEMORY_BASIC_INFORMATION MBI;
        S = sizeof(MBI);
        LPVOID adr = NULL;
        std::cout << "Enter the memory address for which you want to find out the status" <<
std::endl;
        std::cout << "Enter: 0x";
        std::cin >> adr;
        if (adr != NULL)
        {
                S = VirtualQuery(adr, &MBI, S);
                if (S != 0) {
                        std::cout << "The state of the memory area for the specified address " <<
std::hex << adr << ":" << std::endl;
```

```
                        std::cout << "Pointer to the base address of the page region: " <<
MBI.BaseAddress << std::endl;
                        std::cout << "Pointer to the base address of the range of pages allocated by
the function VirtualAlloc: " << MBI.AllocationBase << std::endl;
                        std::cout << "Memory protection parameter at the initial allocation of the
area: ";
                        protectInfo(MBI.AllocationProtect);
                        std::cout << "The size of the area starting from the base address in which all
pages have the same attributes, in bytes: " << std::dec <<(LONGLONG)MBI.RegionSize <<
std::endl << std::endl;
                        std::cout << "Status of pages in the region: \n";
                        switch (MBI.State)
                        {
                        case MEM_COMMIT:
                                std::cout << "The pages for which the physical storage has been
allocated are fixed, either in memory or in a swap file on disk" << std::endl;
                                break;
                        case MEM_FREE:
                                std::cout << "There are free pages that are inaccessible to the calling
process and available for allocation" << std::endl;
                                break;
                        case MEM_RESERVE:
                                std::cout << "Reserved pages where a range of the virtual address
space of the process is reserved without allocating any physical storage" << std::endl;
                                break;
                        }
                        std::cout << "Protecting access to pages in the region: ";
                        protectInfo(MBI.Protect);
                        std::cout << "Type of pages in the region: ";
                        switch (MBI.Type)
                        {
                        case MEM_IMAGE:
                                std::cout << "The pages in the region are projected into the image
representation of the section" << std::endl;
                                break;
                        case MEM_MAPPED:
                                std::cout << "The page in the region is projected into the section
view" << std::endl;
                                break;
                        case MEM_PRIVATE:
                                std::cout << "Pages in the private region" << std::endl;
                                break;
                        }
                }
                else std::cerr << "Error: " << GetLastError();
        }
        else std::cout << "ADR==NULL" << std::endl;
}

void memoryReserve(SYSTEM_INFO si)
{
        int q;
        std::cout << "Select a mode:" << std::endl;
```

```
	std::cout << "1 - Automatic mode" << std::endl;
	std::cout << "2 - Manual mode" << std::endl;
	std::cin >> q;
	if (q == 1 || q == 2)
	{
		void *address = NULL;
		if (q == 2)
		{
			std::cout << "Reserving a memory region in the mode of entering the address
of the beginning of the region" << std::endl;
			std::cout << "Enter the address of the beginning of the region:" << std::endl;
			std::cin >> address;
			address = VirtualAlloc(address, si.dwPageSize, MEM_RESERVE,
PAGE_READWRITE);
		}
		else
		{
			std::cout << "Automatic memory region reservation" << std::endl;
			address = VirtualAlloc(NULL, si.dwPageSize, MEM_RESERVE,
PAGE_READWRITE);
		}
		if (address != NULL)
		{
			VirtualAlloc(address, si.dwPageSize, MEM_COMMIT,
PAGE_READWRITE);
			std::cout << "Memory has been allocated successfully." << std::endl;
			std::cout << "Address: " << address << std::endl;
		}
		else std::cout << "Error VirtualAlloc" << std::endl;
	}
	else std::cout << "You need to enter 1 or 2." << std::endl;
}
void memoryReserveAndCommit(SYSTEM_INFO si)
{
   int q;
	std::cout << "Select a mode:" << std::endl;
	std::cout << "1 - Automatic mode" << std::endl;
	std::cout << "2 - Manual mode" << std::endl;
	std::cin >> q;
	if (q == 1 || q == 2)
	{
		void *address = NULL;
		if (q == 2)
		{
			std::cout << "Reserving a memory region in the mode of entering the address
of the beginning of the region" << std::endl;
			std::cout << "Enter the address of the beginning of the region:" << std::endl;
			std::cin >> address;
			address = VirtualAlloc(address, si.dwPageSize, MEM_RESERVE |
MEM_COMMIT, PAGE_READWRITE);
		}
		else
		{
```

```
                        std::cout << "Automatic memory region reservation" << std::endl;
                        address = VirtualAlloc(NULL, si.dwPageSize, MEM_RESERVE |
MEM_COMMIT, PAGE_READWRITE);
                }
                if (address != NULL)
                {
                        std::cout << "Memory has been allocated successfully." << std::endl;
                        std::cout << "Address: " << address << std::endl;
                        }
                else std::cout << "Error VirtualAlloc" << std::endl;
        }
        else std::cout << "You need to enter 1 or 2." << std::endl;
}

void writeToRegion()
{
   LPVOID addr;
   std::string toWrite;
   std::cout << "Enter address: ";
   std::cin >> addr;
        getchar();
   std::cout << "Enter string data: ";
   getline(std::cin, toWrite);

   std::cout << "Trying to write \"" << toWrite << "\" to address " << addr << "... " << std::endl;

   if (addr != NULL)
   {
      SIZE_T resf;
      MEMORY_BASIC_INFORMATION memi;
      SIZE_T sizeOf_memi = sizeof(MEMORY_BASIC_INFORMATION);
      resf = VirtualQuery(addr, &memi, sizeOf_memi);
      if(!resf)
      {
         std::cout << "An error occurred when receiving information about a range of pages in the
virtual address space of the calling process. Error: " << GetLastError() << std::endl;
         return;
      }
      if(memi.State != MEM_COMMIT)
      {
         std::cout << "Memory " << addr << " state is not MEM_COMMIT" << std::endl;
         return;
      }
      if(!(memi.Protect == PAGE_EXECUTE_READWRITE ||
         memi.Protect == PAGE_EXECUTE_WRITECOPY ||
         memi.Protect == PAGE_READWRITE ||
         memi.Protect == PAGE_WRITECOPY))
      {
         std::cout << "No access to write to " << addr << std::endl;
         return;
      }

      std::string::size_type n = toWrite.size();
```

```
        char* addr_i = (char*)addr;
        for(std::string::size_type i = 0; i < n; ++i, ++addr_i)
            *addr_i = toWrite[i];
        *addr_i = '\0';

        std::cout << "Content at " << addr << ": \"";
        addr_i = (char*)addr;
        for(; *addr_i != '\0'; ++addr_i)
            std::cout << *addr_i;
        std::cout << "\". " << std::endl;
    }
    else
        std::cout << "addr == NULL. " << std::endl;
}

void readRegion()
{
    LPVOID addr;
    std::cout << "Enter address: ";
    std::cin >> addr;
        getchar();

    if(addr != NULL)
    {
        SIZE_T resf;
        MEMORY_BASIC_INFORMATION memi;
        SIZE_T sizeOf_memi = sizeof(MEMORY_BASIC_INFORMATION);
        resf = VirtualQuery(addr, &memi, sizeOf_memi);
        if(!resf)
        {
            std::cout << "An error occurred when receiving information about a range of pages in the
virtual address space of the calling process. Error: " << GetLastError() << std::endl;
            return;
        }
        if(memi.State != MEM_COMMIT)
        {
            std::cout << "Memory " << addr << " state is not MEM_COMMIT" << std::endl;
            return;
        }
        if(!(memi.Protect == PAGE_EXECUTE_READ ||
                             memi.Protect == PAGE_EXECUTE_READWRITE ||
                             memi.Protect == PAGE_EXECUTE_WRITECOPY ||
                             memi.Protect == PAGE_READONLY ||
                             memi.Protect == PAGE_READWRITE ||
                             memi.Protect == PAGE_WRITECOPY
                            ))
        {
            std::cout << "No access to read from " << addr << std::endl;
            return;
        }

        std::cout << "Trying to read from address " << addr << ": \"";
        char* addr_i = (char*)addr;
```

```
        for(; *addr_i != '\0'; ++addr_i)
            std::cout << *addr_i;
        std::cout << "\". " << std::endl;
    }
    else
        std::cout << "Address is NULL. " << std::endl;
}

DWORD getProtect() {
    std::cout << "Choose memory protection constant:" << std::endl;
    std::cout << "1. PAGE_EXECUTE" << std::endl;
    std::cout << "2. PAGE_EXECUTE_READ" << std::endl;
    std::cout << "3. PAGE_EXECUTE_READWRITE" << std::endl;
    std::cout << "4. PAGE_EXECUTE_WRITECOPY" << std::endl;
    std::cout << "5. PAGE_NOACCESS" << std::endl;
    std::cout << "6. PAGE_READONLY" << std::endl;
    std::cout << "7. PAGE_READWRITE" << std::endl;
    std::cout << "8. PAGE_WRITECOPY" << std::endl;

    int x;
    std::cin >> x;

    DWORD level;

    switch (x) {
    case 1:
        level = PAGE_EXECUTE;
        break;
    case 2:
        level = PAGE_EXECUTE_READ;
        break;
    case 3:
        level = PAGE_EXECUTE_READWRITE;
        break;
    case 4:
        level = PAGE_EXECUTE_WRITECOPY;
        break;
    case 5:
        level = PAGE_NOACCESS;
        break;
    case 6:
        level = PAGE_READONLY;
        break;
    case 7:
        level = PAGE_READWRITE;
        break;
    case 8:
        level = PAGE_WRITECOPY;
        break;
    }

    return level;
}
```

```
void setVirtualProtect()
{
   LPVOID address = NULL;

   DWORD oldLevel;
   DWORD newLevel;

   std::cout << "Enter address: 0x";
   std::cin >> address;

   if (address != NULL) {
      newLevel = getProtect();
      std::cout << "New protection level: ";
      protectInfo(newLevel);
      std::cout << std::endl;
      if (VirtualProtect(address, sizeof(DWORD), newLevel, &oldLevel))
      {
         std::cout << "Old protection level:" << std::endl;
         protectInfo(oldLevel);
      }
      else std::cout << "Error: " << GetLastError() << std::endl;

   }
   else std::cout << "Address is NULL" << std::endl;
   std::cout << std::endl << std::endl;
}

void freeVirtualMem()
{
   LPVOID address = NULL;
   std::cout << "Enter address: 0x";
   std::cin >> address;

   if (VirtualFree(address, 0, MEM_RELEASE))
      std::cout << "Success" << std::endl << std::endl;
   else std::cerr << "Error: " << GetLastError();

   std::cout << std::endl << std::endl;
}

void protectInfo(DWORD Pro)
{
         switch (Pro)
         {
         case 0:
                  std::cout << "No access" << std::endl;
                  break;
         case PAGE_EXECUTE:
                  std::cout << "Execute access (PAGE_EXECUTE)" << std::endl;
                  break;
         case PAGE_EXECUTE_READ:
                  std::cout << "Execute or read-only access (PAGE_EXECUTE_READ)" <<
std::endl;
```

```
                break;
        case PAGE_EXECUTE_READWRITE:
                std::cout << "Execute, read-only, or read/write access
(PAGE_EXECUTE_READWRITE)" << std::endl;
                break;
        case PAGE_EXECUTE_WRITECOPY:
                std::cout << "Execute, read-only, or copy-on-write access
(PAGE_EXECUTE_WRITECOPY)" << std::endl;
                break;
        case PAGE_NOACCESS:
                std::cout << "Disables all access to the committed region of pages
(PAGE_NOACCESS)" << std::endl;
                break;
        case PAGE_READONLY:
                std::cout << "Read-only access (PAGE_READONLY)" << std::endl;
                break;
        case PAGE_READWRITE:
                std::cout << "Read-only or read/write access (PAGE_READWRITE)" << std::endl;
                break;
        case PAGE_WRITECOPY:
                std::cout << "Read-only or copy-on-write access (PAGE_WRITECOPY)" <<
std::endl;
                break;
        }
        if ((Pro & PAGE_GUARD) != 0)
        {
                std::cout << "Pages in the region become guard pages (PAGE_GUARD)" <<
std::endl;
        }
        if ((Pro & PAGE_NOCACHE) != 0)
        {
                std::cout << "Pages to be non-cachable (PAGE_NOCACHE)" << std::endl;
        }
        if ((Pro & PAGE_WRITECOMBINE) != 0)
        {
                std::cout << "Pages to be write-combined (PAGE_WRITECOMBINE)" <<
std::endl;
        }
}
```

task2_writer.cpp

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

int main()
{
   HANDLE hFile = NULL;
   std::string fileName;
   std::string mapName;
   LPVOID addrMap = NULL;
   HANDLE hMap = NULL;

   std::cout << "Enter file name: ";
   std::cin >> fileName;

   hFile = CreateFileA(fileName.c_str(), GENERIC_WRITE | GENERIC_READ, 0, NULL,
CREATE_ALWAYS, 0, NULL);
   if(hFile == INVALID_HANDLE_VALUE || hFile == NULL)
   {
      std::cout << "Error: " << GetLastError() << std::endl;
      return GetLastError();
   }

   std::cout << "Enter map name: ";
   std::cin >> mapName;

   hMap = CreateFileMapping(hFile, NULL, PAGE_READWRITE, 0, 128, mapName.c_str());
   if(hMap == NULL)
   {
      std::cout << "Error: " << GetLastError() << std::endl;
      CloseHandle(hFile);
      return GetLastError();
   }

   addrMap = MapViewOfFile(hMap, FILE_MAP_WRITE, 0, 0, 0);
   if(addrMap == NULL)
   {
      std::cout << "Error: " << GetLastError() << std::endl;
      CloseHandle(hFile);
      return GetLastError();
   }

   std::cout << "Trying to write to " << addrMap;
   std::cout << ": \n";
   getchar();
   std::string toWrite;
   getline(std::cin, toWrite);

   CopyMemory(addrMap, toWrite.c_str(), toWrite.length() * sizeof(char));

   std::cout << "Run reader program to continue... \n";
```

```
    unsigned exit;
    do
    {
        std::cout << "To exit from writer program enter 1. \n>";
        std::cin >> exit;
    }
    while(exit != 1);

    CloseHandle(hFile);
    UnmapViewOfFile(addrMap);
    return 0;
}
```

task2_reader.cpp

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

int main()
{
   HANDLE MapHandle = nullptr;
   LPVOID addrMap = NULL;
   std::string mapName;

   std::cout << "Enter map name: ";
   std::cin >> mapName;

   MapHandle = OpenFileMappingA(FILE_MAP_READ, FALSE, mapName.c_str());
   if(MapHandle == NULL)
   {
      std::cout << "Error: " << GetLastError() << std::endl;
      return GetLastError();
   }

   addrMap = MapViewOfFile(MapHandle, FILE_MAP_READ, 0, 0, 0);
   if(addrMap == NULL)
   {
      std::cout << "Error: " << GetLastError() << std::endl;
      CloseHandle(MapHandle);
      return GetLastError();
   }

   std::cout << "Trying to read from " << addrMap << ": \n";
   std::cout << (char*)addrMap << std::endl << std::endl;

   CloseHandle(MapHandle);
   UnmapViewOfFile(addrMap);
   return 0;
}
```