Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Операционные среды и системное программирование

**ОТЧЕТ**

к лабораторной работе № 3

на тему «Управление памятью и вводом-выводом, расширенные возможности ввода-вывода Windows. Функции API подсистемы памяти

Win32. Организация и контроль асинхронных операций ввода-вывода. Отображение файлов в память.»

Выполнил:

студент гр 153504

Шишков В.В.

Проверил:

Гриценко Н.Ю.

Минск 2023

**СОДЕРЖАНИЕ**

[1 Цель работы 3](#_Toc149130709)

[2 Теоретические сведения по асинхронным операциям и win32api 4](#_Toc149130710)

[3 Реализация программного продукта 6](#_Toc149130711)

[4 Результат выполнения программы 7](#_Toc149130712)

[Заключение 9](#_Toc149130713)

[Список использованных источников 10](#_Toc149130714)

[Приложение А 11](#_Toc149130715)

# **1 ЦЕЛЬ РАБОТЫ**

Задачей этой лабораторной работы является разработка приложения для мониторинга и управления системной памятью, предоставляющее возможность отображения и анализа текущего потребления памяти различными процессами в операционной системе Windows с использованием расширенных функций управления памятью, асинхронных операций ввода-вывода и отображения файлов в память. Также целью является изучение функций API Win32, связанных с управлением системной памятью, вводом-выводом и отображением файлов в память.

# **2 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ ПО WIN32API**

Асинхронные операции ввода-вывода представляют собой важный аспект в мире программирования, позволяя эффективно управлять взаимодействием приложений с файлами и устройствами в асинхронном режиме. В контексте операционной системы Windows, асинхронные операции ввода-вывода играют важную роль в обеспечении отзывчивости приложений и оптимизации использования ресурсов. В асинхронном режиме приложение может выполнять любую функцию, которая включает значение контекста в качестве одного из своих параметров, и может продолжать выполнять другие команды или функции, пока приложение ожидает завершения задачи. Пока задача завершается, функция обратного вызова состояния, предоставляемая приложением, получает уведомление о ходе выполнения задачи и ее завершении. В настоящее время функция обратного вызова состояния может вызывать другие функции или выполнять любые другие необходимые задачи, зависящие от завершения задачи.[1]

Организация и контроль асинхронных операций ввода-вывода в Win32 API является важной составляющей разработки высокопроизводительных и отзывчивых приложений. Win32 API предоставляет набор механизмов и структур данных, которые позволяют разработчикам создавать и управлять асинхронными операциями ввода-вывода внутри приложений. Главная цель - дать приложению возможность эффективно использовать ресурсы, не блокируя главный поток выполнения и обеспечивая отзывчивость пользовательского интерфейса.

Одним из ключевых элементов организации асинхронных операций в Win32 API является использование объекта OVERLAPPED. Структура OVERLAPPED используется для отслеживания асинхронных операций, предоставляя информацию о состоянии операции, ее ходе и завершении. Это позволяет приложению мониторить и контролировать множество асинхронных операций одновременно, не блокируя выполнение других задач. Также используются функции API, такие как “ReadFile”, “WriteFile” и “GetOverlappedResult”, чтобы инициировать, управлять и завершать асинхронные операции.[2]

Контроль асинхронных операций ввода-вывода в Win32 API также включает в себя мониторинг и обработку событий, связанных с завершением операций. Это может быть достигнуто с использованием механизма событий или функции “WaitForSingleObject”, которые позволяют приложению блокировать выполнение до тех пор, пока асинхронная операция не завершится. Для управления несколькими асинхронными операциями существуют функции, такие как “WaitForMultipleObjects”. Этот контроль и координация операций являются важными аспектами создания эффективных и стабильных приложений, способных обрабатывать множество асинхронных задач одновременно.[3]

Отображение файлов в память представляет собой технику, которая позволяет работать с файлами, как если бы они были обычными областями памяти. Это устраняет необходимость в явном чтении и записи данных в файл, так как данные могут быть обрабатаны как массив байтов в оперативной памяти. Это оказывает положительное воздействие на производительность приложения, особенно при работе с крупными файлами или когда необходимо многократно обращаться к одним и тем же данным. Создание объекта отображения файла с последующим его отображением в адресное пространство процесса обеспечивает быстрый и удобный доступ к данным.

В данной работе, мы будем исследовать и применять эти техники в контексте создания приложения для мониторинга и управления системной памятью, с акцентом на асинхронных операциях ввода-вывода и отображении файлов в память. Это поможет нам глубже понять и использовать возможности операционной системы Windows для оптимизации работы с данными и ресурсами.

# **3 РЕАЛИЗАЦИЯ ПРОГРАММНОГО ПРОДУКТА**

В рамках данной лабораторной работы было разработано приложение “Memory Manager”, предназначенное для мониторинга и управления системной памятью, а также записи информации о процессах в файл. Реализация программного продукта базируется на использовании Windows API (Win32) и включает в себя основные компоненты, такие как графический интерфейс, сбор информации о процессах и асинхронное выполнение задач.

Основным компонентом приложения является оконный интерфейс, созданный с использованием Win32 API. Он позволяет пользователю взаимодействовать с приложением и выполнять различные действия. В окне приложения размещены элементы управления, такие как кнопки “Обновить” и “Записать в файл”, а также список для отображения информации о процессах.

Для мониторинга процессов приложение использует функции Windows API, такие как EnumProcesses, OpenProcess, EnumProcessModules и GetProcessMemoryInfo. Эти функции позволяют получить информацию о запущенных процессах, их модулях и использовании памяти. Полученные данные обрабатываются и отображаются в списке на графическом интерфейсе, обеспечивая пользователю информацию о текущем потреблении памяти различными процессами.

Для асинхронной записи информации о процессах в файл приложение использует механизм std::async. Этот механизм позволяет выполнять операции в фоновом режиме, не блокируя пользовательский интерфейс. Информация о процессах записывается в текстовый файл с именем "ProcessList.txt," который может быть создан и обновлен пользователем при помощи соответствующей кнопки на интерфейсе. Этот механизм обеспечивает отзывчивость приложения и позволяет пользователю в любой момент сохранить информацию о процессах.

Таким образом, приложение "Memory Manager" представляет собой реализацию мониторинга и управления системной памятью с использованием Win32 API. Оно обеспечивает пользователю доступ к данным о процессах и памяти, а также предоставляет возможность записи этой информации в файл для последующего анализа. Реализация включает в себя использование асинхронных операций, чтобы обеспечить высокую отзывчивость и удобство использования приложения.

# **4 РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ ПРОГРАММЫ**

В данной лабораторной было создано приложение для мониторинга системной памяти, которое отображает потребление памяти различными процессами. (рисунок 4.1).

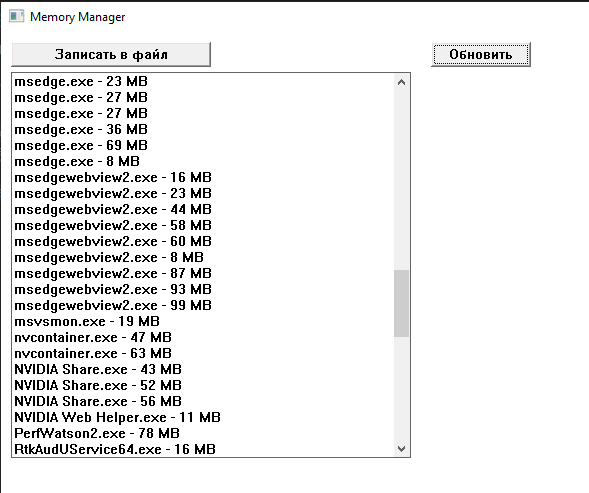


Рисунок 4.1 – Главное окно приложения

При нажатии на кнопку обновить происходит обновление списка процессов в данный момент времени. Как видно на скриншотах, потребление памяти в разные моменты времени немного отличается. (рисунок 4.2).

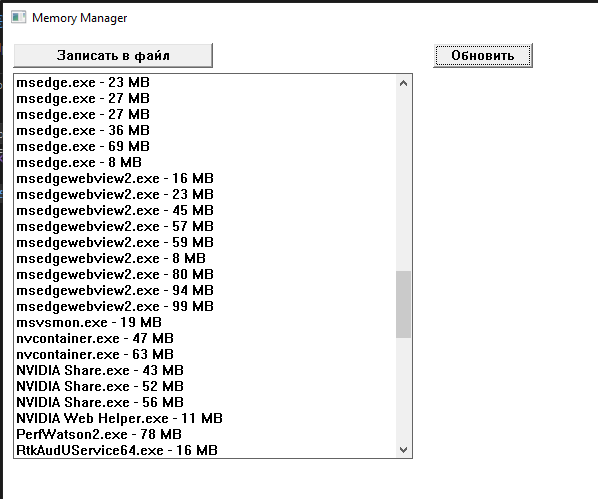


Рисунок 4.2 ­– Обновленный список процессов

Также в приложении есть возможность записать список процессов в файл. При нажатии на кнопку “Записать в файл” произойдет асинхронная запись в файл списка процессов и их потребление оперативной памяти. (рисунок 4.3).

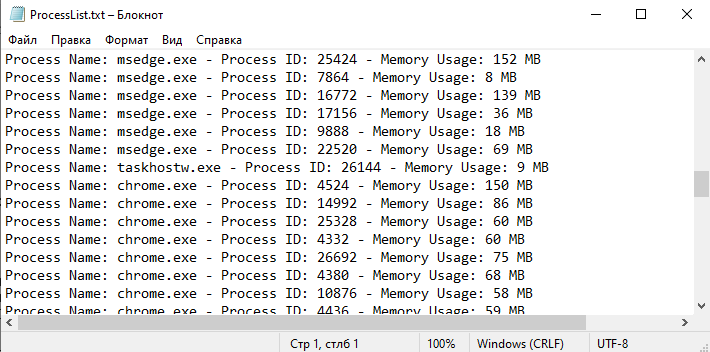


Рисунок 4.3 – Файл со списком процессов

# **ЗАКЛЮЧЕНИЕ**

В ходе выполнения лабораторной работы было исследованы и применены различные важные API-интерфейсы, которые предоставляет платформа Win32 для управления памятью, вводом-выводом и мониторинга системных ресурсов. Эти API-интерфейсы играют ключевую роль в разработке высокоэффективных и отзывчивых приложений под операционную систему Windows.

Один из важных аспектов, рассмотренных в лабораторной работе, связан с асинхронными операциями ввода-вывода. Был успешно применен механизм асинхронных операций с использованием стандартной библиотеки C++, что позволило выполнять операции ввода-вывода параллельно, не блокируя главный поток приложения. Использование API Win32, таких как CreateFile, ReadFile, и WriteFile, дало возможность эффективно управлять файлами и данными, обеспечивая высокую отзывчивость приложения.

Второй важный аспект связан с мониторингом системных ресурсов и отображением данных в память. Был активно использован Win32 API для сбора информации о работающих процессах и их потреблении оперативной памяти. С помощью библиотеки Windows API, такой как EnumProcesses, OpenProcess, и GetProcessMemoryInfo, возможно эффективно мониторить систему и представлять эту информацию на графическом интерфейсе приложения.

Заключая, полученные знания и практические навыки открывают новые перспективы для разработки более сложных приложений и инструментов для мониторинга и управления ресурсами операционной системы Windows. Эти API-интерфейсы остаются важными инструментами для разработчиков, стремящихся создавать высокопроизводительные приложения под Windows.

# **СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1. Асинхронная операция [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/wininet/asynchronous-operation
2. Асинхронные операции [Электронный ресурс]. – Режим доступа: http://www.vsokovikov.narod.ru/New\_MSDN\_API/Comm\_res/overlapped\_operations.htm
3. Рекомендации по асинхронным операциям [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/wua\_sdk/guidelines-for-asynchronous-wua-operations

# **ПРИЛОЖЕНИЕ А**

#include <Windows.h>

#include <psapi.h>

#include <tchar.h>

#include <vector>

#include <string>

#include <fstream>

#include <thread>

#include <future>

LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam);

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow) {

WNDCLASS wc = {};

wc.lpfnWndProc = WindowProc;

wc.hInstance = hInstance;

wc.lpszClassName = L"MemoryManagerWindowClass";

RegisterClass(&wc);

HWND hwnd = CreateWindowEx(0, L"MemoryManagerWindowClass", L"Memory Manager",

WS\_OVERLAPPEDWINDOW, CW\_USEDEFAULT, CW\_USEDEFAULT, 800, 600, NULL, NULL, hInstance, NULL);

if (hwnd == NULL) {

return 1;

}

ShowWindow(hwnd, nCmdShow);

MSG msg = {};

while (GetMessage(&msg, NULL, 0, 0)) {

TranslateMessage(&msg);

DispatchMessage(&msg);

}

return 0;

}

void UpdateMemoryInfo(HWND hwnd) {

DWORD processes[1024];

DWORD cbNeeded;

if (EnumProcesses(processes, sizeof(processes), &cbNeeded)) {

int numProcesses = cbNeeded / sizeof(DWORD);

std::vector<std::pair<std::wstring, DWORD>> processInfo;

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

HANDLE hProcess = OpenProcess(PROCESS\_QUERY\_INFORMATION | PROCESS\_VM\_READ, FALSE, processes[i]);

if (hProcess != NULL) {

TCHAR szProcessName[MAX\_PATH] = TEXT("<unknown>");

HMODULE hModule;

DWORD cbNeeded;

if (EnumProcessModules(hProcess, &hModule, sizeof(hModule), &cbNeeded)) {

GetModuleBaseName(hProcess, hModule, szProcessName, sizeof(szProcessName) / sizeof(TCHAR));

}

PROCESS\_MEMORY\_COUNTERS pmc;

if (GetProcessMemoryInfo(hProcess, &pmc, sizeof(pmc))) {

processInfo.push\_back(std::make\_pair(szProcessName, pmc.WorkingSetSize));

}

CloseHandle(hProcess);

}

}

SendMessage(hwnd, LB\_RESETCONTENT, 0, 0);

for (const auto& info : processInfo) {

std::wstring processData = info.first;

processData += L" - ";

processData += std::to\_wstring(info.second / (1024 \* 1024));

processData += L" MB";

SendMessage(hwnd, LB\_ADDSTRING, 0, reinterpret\_cast<LPARAM>(processData.c\_str()));

}

}

}

std::future<void> SaveProcessListAsync() {

return std::async(std::launch::async, []() {

std::wofstream fileStream(L"ProcessList.txt");

if (fileStream.is\_open()) {

DWORD processes[1024];

DWORD cbNeeded;

if (EnumProcesses(processes, sizeof(processes), &cbNeeded)) {

int numProcesses = cbNeeded / sizeof(DWORD);

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

HANDLE hProcess = OpenProcess(PROCESS\_QUERY\_INFORMATION | PROCESS\_VM\_READ, FALSE, processes[i]);

if (hProcess != NULL) {

TCHAR szProcessName[MAX\_PATH] = TEXT("<unknown>");

HMODULE hModule;

DWORD cbNeeded;

if (EnumProcessModules(hProcess, &hModule, sizeof(hModule), &cbNeeded)) {

GetModuleBaseName(hProcess, hModule, szProcessName, sizeof(szProcessName) / sizeof(TCHAR));

}

PROCESS\_MEMORY\_COUNTERS pmc;

if (GetProcessMemoryInfo(hProcess, &pmc, sizeof(pmc))) {

fileStream << L"Process Name: " << szProcessName << L" - Process ID: " << processes[i]

<< L" - Memory Usage: " << pmc.WorkingSetSize / (1024 \* 1024) << L" MB" << std::endl;

}

CloseHandle(hProcess);

}

}

}

fileStream.close();

}

});

}

LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

static HWND hListBox;

static DWORD processes[1024];

static int numProcesses = 0;

switch (uMsg) {

case WM\_CREATE:

hListBox = CreateWindowEx(0, L"LISTBOX", L"Process Memory Usage",

WS\_CHILD | WS\_VISIBLE | WS\_BORDER | LBS\_STANDARD, 10, 40, 400, 400, hwnd, (HMENU)1, NULL, NULL);

UpdateMemoryInfo(hListBox);

CreateWindow(L"BUTTON", L"Обновить", WS\_CHILD | WS\_VISIBLE, 430, 10, 100, 25, hwnd, (HMENU)2, NULL, NULL);

CreateWindow(L"BUTTON", L"Записать в файл", WS\_CHILD | WS\_VISIBLE, 10, 10, 200, 25, hwnd, (HMENU)4, NULL, NULL);

break;

case WM\_COMMAND:

switch (LOWORD(wParam)) {

case 2:

UpdateMemoryInfo(hListBox);

break;

case 4:

SaveProcessListAsync();

break;

}

break;

case WM\_DESTROY:

PostQuitMessage(0);

break;

default:

return DefWindowProc(hwnd, uMsg, wParam, lParam);

}

return 0;

}