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 Win 32. Организация и контроль асинхронных операций ввода-вывода. Отображение файлов в память»

Выполнил:

студент гр. 153504

Сивый А.А.

Проверил:

Гриценко Н.Ю.

Минск 2023

**СОДЕРЖАНИЕ**

[1 Цели работы 3](#_gjdgxs)

[2 Краткие теоретические сведения 4](#_30j0zll)

[3 Полученные результаты 5](#_1fob9te)

[Вывод 6](#_4q3p755gzys6)

[Список использованных источников 7](#_4q3p755gzys6)

[Приложение А (обязательное) Листинг кода 8](#_4q3p755gzys6)

**1 ЦЕЛИ РАБОТЫ**

1 Изучить расширенные возможности ввода-вывода Windows.

2 Изучить функции API подсистемы памяти Win32 и их использование.

3 Изучить организацию и контроль асинхронных операций ввода-вывода.

4 Создать приложение для мониторинга и управления системной памятью, отображающее текущее потребление памяти различными процессами.

**2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

Управление памятью и вводом-выводом – это важные аспекты операционной системы Windows. Они позволяют приложениям эффективно использовать ресурсы компьютера, такие как память и дисковое пространство.

Функции API подсистемы памяти Win 32 предоставляют возможность управления памятью в приложениях, работающих в операционной системе Windows.

Расширенные возможности ввода-вывода Windows предоставляют дополнительные функции для работы с устройствами ввода-вывода, такие как файловые операции, работа с сетью и асинхронные операции ввода-вывода.

Отображение файлов в память – это процесс загрузки содержимого файла в оперативную память компьютера. Это может быть полезно для ускорения доступа к файлу и уменьшения количества обращений к диску.

В данной лабораторной работе была использована библиотека Psapi.h. Она предоставляет функции для работы с процессами в операционной системе Windows. Она используется для получения информации о процессах, модулях и памяти, используемой процессами.

Функции, которые предоставляет библиотека Psapi.h:

1 EnumProcesses – используется для получения идентификаторов всех процессов в системе. Она принимает три параметра: указатель на массив, который будет содержать идентификаторы процессов, размер этого массива в байтах и указатель на переменную, которая будет содержать количество байт, необходимых для заполнения массива.

2 OpenProcess – используется для открытия существующего локального объекта процесса, возвращает дескриптор открытого процесса, если операция выполнена успешно, и NULL в противном случае.

3 EnumProcessModules – используется для получения дескрипторов всех модулей, загруженных в указанный процесс. Она принимает четыре параметра: дескриптор процесса, массив, который будет содержать дескрипторы модулей, размер этого массива в байтах и указатель на переменную, которая будет содержать количество байт, необходимых для заполнения массива.

4 GetProcessMemoryInfo – используется для получения информации о памяти, используемой указанным процессом. Она принимает три параметра: дескриптор процесса, указатель на структуру PROCESS\_MEMORY\_COUNTERS, которая будет содержать информацию о памяти, используемой процессом, и размер этой структуры в байтах.

Это основные функции, используемые в лабораторной работе для получения информации о памяти, используемой процессами.

**3 ПОЛУЧЕННЫЕ РЕЗУЛЬТАТЫ**

Была создана программа, которая позволяет наблюдать за всеми запущенными процессами в системе Windows (рисунок 3.1).

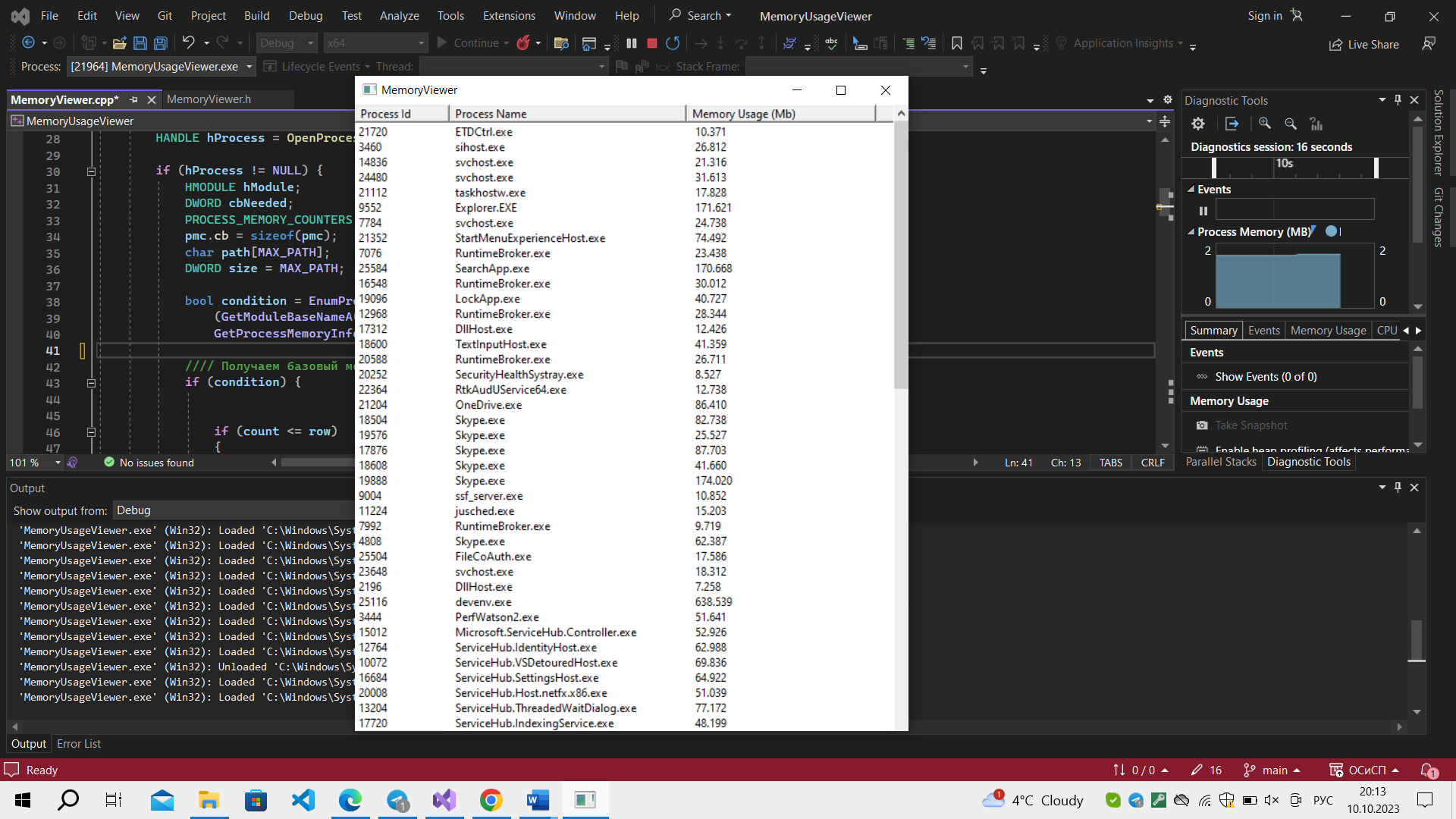


Рисунок 3.1 – Результат работы программы

В окне созданной программы можно наблюдать следующую информацию: идентификатор процесса (Process Id), название запущенного процесса (Process Name) и используемый объём памяти (Memory Usage) в мегабайтах. Так как используемый объём памяти постоянно меняется, программа ежесекундно обновляет информацию, поэтому, если запустить программу, после чего запустить какой-либо процесс, он сразу же отобразится в списке.

**ВЫВОД**

В результате выполнения работы было разработано приложение, которое позволяет управлять и наблюдать за памятью, используемой разными процессами Windows. Для этой цели был изучен теоретический материал по различным библиотекам.

**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

[1] Psapi, документация Microsoft [Электронный ресурс]. – Режим доступа: <https://learn.microsoft.com/en-us/windows/win32/api/psapi/>.

[2] Управления памятью Windows [Электронный ресурс]. – Режим доступа: learn.microsoft.com/ru-ru/windowshardware/drivers/kernel/managing-memory-for-drivers.

[3] Управление памятью и вводом-выводом в ОС Windows. [Электронный ресурс] – Режим доступа: https://studopedia.ru/25\_3449\_tema-upravlenie-pamyatyu-i-vvodomvivodom-v-os-Windows.html

**ПРИЛОЖЕНИЕ А**

**(обязательное)**

**Листинг кода**

#include <iostream>

#include <iomanip>

#include <Windows.h>

#include <Psapi.h>

#include <stdio.h>

#include <wchar.h>

#include <string>

#include "MemoryViewer.h"

int UpdateProcessList() {

DWORD processes[1024];

DWORD bytesReturned;

// Получаем идентификаторы запущенных процессов

if (!EnumProcesses(processes, sizeof(processes), &bytesReturned)) {

std::cout << "Error with call EnumProcesses" << std::endl;

return 1;

}

// Вычисление количества процессов

int numProcesses = bytesReturned / sizeof(DWORD);

// Выводим идентификаторы процессор

int row = 0;

int count = ListView\_GetItemCount(hList);

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

// Открываем процесс для получения дополнительной ифнормации

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

if (hProcess != NULL) {

HMODULE hModule;

DWORD cbNeeded;

PROCESS\_MEMORY\_COUNTERS pmc;

pmc.cb = sizeof(pmc);

char path[MAX\_PATH];

DWORD size = MAX\_PATH;

bool condition = EnumProcessModules(hProcess, &hModule, sizeof(hModule), &cbNeeded) &&

(GetModuleBaseNameA(hProcess, hModule, path, sizeof(path)) != 0) &&

GetProcessMemoryInfo(hProcess, &pmc, sizeof(pmc));

//// Получаем базовый модуль процесса

if (condition) {

if (count <= row)

{

LVITEM lvItem;

lvItem.mask = LVIF\_TEXT;

lvItem.iItem = row;

lvItem.iSubItem = 0;

lvItem.pszText = new wchar\_t[0];

ListView\_InsertItem(hList, &lvItem);

}

wchar\_t buffer[256];

swprintf\_s(buffer, L"%d", processes[i]);

ListView\_SetItemText(hList, row, 0, buffer);

int length = strlen(path) + 1;

int size = MultiByteToWideChar(CP\_UTF8, 0, path, length, NULL, 0);

MultiByteToWideChar(CP\_UTF8, 0, path, length, buffer, size);

ListView\_SetItemText(hList, row, 1, buffer);

swprintf\_s(buffer, L"% .3lf", (double)pmc.WorkingSetSize / (1024 \* 1024));

ListView\_SetItemText(hList, row, 2, buffer);

row++;

std::cout << "Process Identificator: " << processes[i] << " Process Name: " << path << " Memory usage: " << std::fixed << std::setprecision(2) << (double)pmc.WorkingSetSize / (1024 \* 1024) << " Mb" << std::endl;

}

// Закрываем дескриптор процесса

CloseHandle(hProcess);

}

}

for (int i = row; i < count; i++) {

ListView\_DeleteItem(hList, i);

}

return 0;

}

VOID CALLBACK TimerCallback(HWND hWnd, UINT uMsg, UINT\_PTR idEvent, DWORD dwTime);

int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hPrevInst, LPSTR lpCmdLine, int nCmdShow) {

hInstance = hInst;

WNDCLASSEX wc = { sizeof(WNDCLASSEX) };

wc.cbClsExtra = 0;

wc.cbWndExtra = 0;

wc.hbrBackground = (HBRUSH)(COLOR\_WINDOW + 1);

wc.hCursor = LoadCursor(NULL, IDC\_ARROW);

wc.hIcon = LoadIcon(NULL, IDI\_APPLICATION);

wc.hIconSm = LoadIcon(NULL, IDI\_APPLICATION);

wc.hInstance = hInstance;

wc.lpfnWndProc = WindowProc;

wc.lpszClassName = L"MyWindowClass";

wc.style = CS\_HREDRAW | CS\_VREDRAW;

if (!RegisterClassEx(&wc)) {

MessageBox(NULL, L"Register window class error!", L"Error!", MB\_ICONERROR | MB\_OK);

return 1;

}

CreateMainWindow();

if (hwnd == NULL) {

MessageBox(NULL, L"Creating main window error!", L"Error!", MB\_ICONERROR | MB\_OK);

return 1;

}

UpdateProcessList();

UINT\_PTR timerId = SetTimer(NULL, 0, interval, TimerCallback);

ShowWindow(hwnd, nCmdShow);

UpdateWindow(hwnd);

MSG msg;

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

TranslateMessage(&msg);

DispatchMessage(&msg);

}

KillTimer(NULL, timerId);

return 0;

}

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

switch (uMsg) {

case WM\_SIZE:

{

break;

}

case WM\_CLOSE:

DestroyWindow(hwnd);

break;

case WM\_DESTROY:

PostQuitMessage(0);

break;

default:

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

}

}

VOID CALLBACK TimerCallback(HWND hWnd, UINT uMsg, UINT\_PTR idEvent, DWORD dwTime){

UpdateProcessList();

}

void CreateMainWindow() {

hwnd = CreateWindow(\_T("MyWindowClass"), L"MemoryViewer",WS\_OVERLAPPEDWINDOW, 100, 100, 600, 700, NULL, NULL, hInstance, NULL);

hList = CreateWindow(WC\_LISTVIEW, \_T(""), WS\_CHILD | WS\_VISIBLE | LVS\_REPORT, 0, 0, 585, 700, hwnd, NULL, NULL, NULL);

ListView\_SetExtendedListViewStyle(hList, LVS\_EX\_FULLROWSELECT);

LVCOLUMN lvColumn;

lvColumn.mask = LVCF\_TEXT | LVCF\_WIDTH;

lvColumn.cx = 100;

lvColumn.pszText = const\_cast<wchar\_t\*>(L"Process Id");

ListView\_InsertColumn(hList, 0, &lvColumn);

LVCOLUMN lvColumn2;

lvColumn2.mask = LVCF\_TEXT | LVCF\_WIDTH;

lvColumn2.cx = 250;

lvColumn2.pszText = const\_cast<wchar\_t\*>(L"Process Name");

ListView\_InsertColumn(hList, 1, &lvColumn2);

LVCOLUMN lvColumn3;

lvColumn3.mask = LVCF\_TEXT | LVCF\_WIDTH;

lvColumn3.cx = 200;

lvColumn3.pszText = const\_cast<wchar\_t\*>(L"Memory Usage (Mb)");

ListView\_InsertColumn(hList, 2, &lvColumn3);

}