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

Студент Н. С. Шмидт
Преподаватель Н. Ю. Гриценко

Минск 2023

СОДЕРЖАНИЕ

1 ЦЕЛЬ РАБОТЫ

1 Изучение основных концепций и технологий, связанных с управлением памятью и вводом-выводом в операционной системе Windows.

2 Изучение методов организации и контроля асинхронных операций ввода-вывода в Windows.

3 Изучение функций API подсистемы памяти Win32, включая их назначение и способы использования.

4 Разработка оконного приложения для мониторинга системной памятью, отображающее текущее потребление памяти различными процессами.

2 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Управление памятью и вводом-выводом (*I/O*) - важные аспекты операционных систем, включая *Windows*. В *Windows* операционная система предоставляет множество функций *API* для управления памятью и для ввода-вывода.

Виртуальная память: *Windows* предоставляет виртуальную память, которая позволяет приложениям использовать больше памяти, чем физически доступно. Виртуальная память разбита на страницы, и Windows отвечает за переключение и загрузку страниц в физическую память.

1 Функция *VirtualAlloc*: Выделяет блоки виртуальной памяти.

2 Функция *VirtualFree*: Освобождает виртуальную память.

3 Функция *VirtualProtect*: Изменяет защиту страниц памяти (например, чтение, запись, выполнение).

4 Функция *VirtualQuery*: Получает информацию о виртуальной памяти.

Memory mapping (отображение памяти) - это мощный механизм в *Windows API*, который позволяет приложениям работать с файлами и разделять данные между процессами, используя виртуальную память.

Memory mapping представлен в *Windows* через *Memory-Mapped Files* (*MMF*). Это позволяет отображать файлы или другие ресурсы, доступные в памяти, в адресное пространство приложения.

Основные функции *Windows API* для работы с *memory mapping* включают в себя:

1 *CreateFileMapping*: Создание или открытие отображаемого файла.

2 *MapViewOfFile* или *MapViewOfFileEx*: Отображение файла в адресное пространство процесса.

3 *UnmapViewOfFile*: Отмена отображения файла.

4 *FlushViewOfFile*: Сохранение изменений в отображенном файле.

5 *OpenFileMapping*: Открытие существующего отображения файла.

Преимущества *Memory Mapping*:

1 Быстрый доступ: *Memory mapping* обеспечивает быстрый доступ к данным в файле, так как операции ввода-вывода выполняются автоматически при доступе к данным в памяти.

2 Обмен данными: *Memory mapping* позволяет разным процессам разделять данные, так как несколько процессов могут отобразить один и тот же файл в память и обмениваться данными через него.

3 РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ

В результате выполнения лабораторной работы было создано приложение для ля мониторинга системной памятью, отображающее текущее потребление памяти различными процессами (рисунок 1).

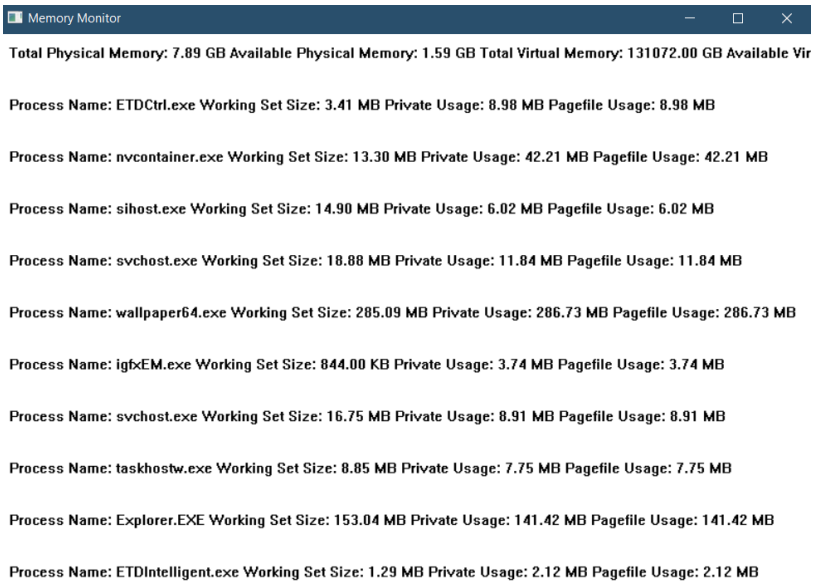

Рисунок 1 – Окно приложения

Данное приложение выполняет асинхронный мониторинг системной памяти с обновлением окна раз в 2 секунды (рисунок 2).

Рисунок 2 – Изменение значений потребления памяти через

Приложение показывает общее и доступное количество физической и виртуальной памяти устройства, а также информацию о процессах, такую как:

1 Название процесса
2 Объем физической памяти, выделенной для процесса
3 Количество частной памяти, используемой процессом
4 Количество памяти процесса, которое было выгружено в файл подкачки (*pagefile*) на диске

ЗАКЛЮЧЕНИЕ

В результате лабораторной работы были изучены основные принципы работы с вводом-выводом: управление памятью, контроль асинхронных операций. Было создано оконное приложение для мониторинга системной памятью, отображающее текущее потребление памяти различными процессами.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Справочник по программированию для API Win32 [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/api/.

[2] Сопоставление файлов [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/memory/file-mapping.

[3] Основы программирования для Win32 API [Электронный ресурс]. – Режим доступа: https://dims.karelia.ru/win32/.

ПРИЛОЖЕНИЕ А
(обязательное)
Листинг кода

Листинг 1 – Файл main.cpp

```
#include <windows.h>
#include <psapi.h>
#include <tchar.h>
#include <string>
#include <vector>
#include <iomanip>
#include <sstream>
#include <future>
#include <chrono>

HINSTANCE hInst;
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
void MonitorMemoryUsage(HWND hwnd, HDC hdc);
std::wstring FormatBytes(DWORDLONG bytes);
bool exitFlag = false;

std::wstring FormatBytes(DWORDLONG bytes)
{
    std::wstringstream ss;

    if (bytes >= 1024 * 1024 * 1024)
    {
        ss << std::fixed << std::setprecision(2) <<
static_cast<double>(bytes) / (1024 * 1024 * 1024) << L" GB";
    }
    else if (bytes >= 1024 * 1024)
    {
        ss << std::fixed << std::setprecision(2) <<
static_cast<double>(bytes) / (1024 * 1024) << L" MB";
    }
    else if (bytes >= 1024)
    {
        ss << std::fixed << std::setprecision(2) <<
static_cast<double>(bytes) / 1024 << L" KB";
    }
    else
    {
        ss << bytes << L" B";
    }
    return ss.str();
}

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR
lpCmdLine, int nCmdShow)
{
    hInst = hInstance;
    WNDCLASSEX wc = { sizeof(WNDCLASSEX), CS_CLASSDC, WndProc, 0L, 0L,
GetModuleHandle(NULL), NULL, NULL, NULL, NULL, _T("MemoryMonitor"), NULL };
    RegisterClassEx(&wc);
    HWND hwnd = CreateWindow(wc.lpszClassName, _T("Memory Monitor"),
WS_OVERLAPPEDWINDOW, 100, 100, 800, 600, NULL, NULL, wc.hInstance, NULL);

    if (hwnd == NULL)
```

```
    {
        MessageBox(NULL, _T("Window Creation Failed!"), _T("Error"),
MB_ICONEXCLAMATION | MB_OK);
        return 0;
    }

    ShowWindow(hwnd, nCmdShow);
    UpdateWindow(hwnd);

    std::future<void> monitorFuture;

    monitorFuture = std::async(std::launch::async, [hwnd]() {
        while (!exitFlag) {
            HDC hdc = GetDC(hwnd);
            MonitorMemoryUsage(hwnd, hdc);
            ReleaseDC(hwnd, hdc);
            std::this_thread::sleep_for(std::chrono::seconds(2));
        }
        });

    SetTimer(hwnd, 1, 2000, NULL);

    MSG msg;
    while (GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    KillTimer(hwnd, 1);
    UnregisterClass(wc.lpszClassName, wc.hInstance);
    return 0;
}

LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    HDC hdc;
    switch (msg)
    {
    case WM_CLOSE:
        PostQuitMessage(0);
        break;

    case WM_TIMER:
        hdc = GetDC(hwnd);
        MonitorMemoryUsage(hwnd, hdc);
        ReleaseDC(hwnd, hdc);
        break;

    case WM_PAINT:
    {
        PAINTSTRUCT ps;
        hdc = BeginPaint(hwnd, &ps);
        MonitorMemoryUsage(hwnd, hdc);
        EndPaint(hwnd, &ps);
    }
    break;

    default:
        return DefWindowProc(hwnd, msg, wParam, lParam);
```

```
    }
    return 0;
}

void MonitorMemoryUsage(HWND hwnd, HDC hdc)
{
    MEMORYSTATUSEX memInfo;
    memInfo.dwLength = sizeof(memInfo);
    GlobalMemoryStatusEx(&memInfo);
    std::wstring memoryInfo = L"Total Physical Memory: " +
FormatBytes(memInfo.ullTotalPhys) + L"\n";
    memoryInfo += L" Available Physical Memory: " +
FormatBytes(memInfo.ullAvailPhys) + L"\n";
    memoryInfo += L" Total Virtual Memory: " +
FormatBytes(memInfo.ullTotalVirtual) + L"\n";
    memoryInfo += L" Available Virtual Memory: " +
FormatBytes(memInfo.ullAvailVirtual) + L"\n";

    DWORD processIds[1024], bytesReturned;
    if (EnumProcesses(processIds, sizeof(processIds), &bytesReturned)) {
        DWORD numProcesses = bytesReturned / sizeof(DWORD);
        std::vector<std::wstring> processInfoList;

        for (DWORD i = 0; i < numProcesses; i++) {
            HANDLE hProcess = OpenProcess(PROCESS_QUERY_INFORMATION |
PROCESS_VM_READ, FALSE, processIds[i]);
            if (hProcess) {
                TCHAR processName[MAX_PATH];
                if (GetModuleBaseName(hProcess, NULL, processName, MAX_PATH))
{
                    std::wstring processInfo = L"Process Name: " +
std::wstring(processName) + L"\n";
                    PROCESS_MEMORY_COUNTERS_EX pmc;

                    if (GetProcessMemoryInfo(hProcess,
(PROCESS_MEMORY_COUNTERS*)&pmc, sizeof(pmc))) {
                        processInfo += L" Working Set Size: " +
FormatBytes(pmc.WorkingSetSize) + L"\n";
                        processInfo += L" Private Usage: " +
FormatBytes(pmc.PrivateUsage) + L"\n";
                        processInfo += L" Pagefile Usage: " +
FormatBytes(pmc.PagefileUsage) + L"\n";
                    }
                    processInfoList.push_back(processInfo);
                }
                CloseHandle(hProcess);
            }
        }

        RECT clientRect;
        GetClientRect(hwnd, &clientRect);
        FillRect(hdc, &clientRect, (HBRUSH)(COLOR_WINDOW + 1));
        int y = 10;
        TextOut(hdc, 10, y, memoryInfo.c_str(),
static_cast<int>(memoryInfo.size()));
        y += 50;

        for (const auto& processInfo : processInfoList) {
            TextOut(hdc, 10, y, processInfo.c_str(),
static_cast<int>(processInfo.size()));
            y += 50;
```

```
            }

            ReleaseDC(hwnd, hdc);
        }
    }
```