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 № 4

на тему «Управление процессами и потоками (Windows). Порождение, завершение, изменение приоритетов процессов и потоков, исследование эффективности.»

Выполнил:

студент гр 153504

Шишков В.В.

Проверил:

Гриценко Н.Ю.

Минск 2023

**СОДЕРЖАНИЕ**

[1 Цель работы 3](#_Toc149132404)

[2 Теоретические сведения по потокам и процессам 4](#_Toc149132405)

[3 Реализация программного продукта 6](#_Toc149132406)

[4 Результат выполнения программы 8](#_Toc149132407)

[Заключение 11](#_Toc149132408)

[Список использованных источников 12](#_Toc149132409)

[Приложение А 13](#_Toc149132410)

# **1 ЦЕЛЬ РАБОТЫ**

Задачей данной лабораторной работы является разработка приложения для мониторинга и управления системной памятью в операционной системе Windows. Главными задачами работы являются отслеживание текущего потребления памяти различными процессами, возможность приостановки, возобновления и завершения процессов. Данное приложение будет обеспечивать более гибкий контроль ресурсов системы и повысит понимание механизмов управления памятью в операционной системе Windows.

# **2 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ ПО ПОТОКАМ**

Потоки и процессы представляют собой важные компоненты операционной системы Windows, позволяющие эффективно управлять выполняющимися задачами и ресурсами компьютера. Процесс можно рассматривать как изолированную среду, в которой выполняются приложения. В пределах одного процесса могут работать несколько потоков, которые представляют собой наименьшие исполнительные единицы в операционной системе. Каждый процесс имеет свою собственную область памяти, в то время как потоки внутри процесса разделяют общую область памяти процесса. Это позволяет потокам обмениваться данными и взаимодействовать друг с другом.

Потоки в Windows предоставляют параллельное выполнение задач, что позволяет эффективно использовать многоядерные процессоры. Один процесс может содержать несколько потоков, которые выполняются независимо друг от друга, улучшая отзывчивость и производительность программ. Win32 API предоставляет множество функций и механизмов для создания, управления и синхронизации потоков.

Функция CreateThread создает новый поток для процесса. Создаваемый поток должен указать начальный адрес кода, который должен выполнить новый поток. Как правило, начальный адрес — это имя функции, определенной в коде программы. Эта функция принимает один параметр и возвращает значение DWORD . Процесс может содержать несколько потоков, одновременно выполняющих одну и ту же функцию.[1]

Однако важно соблюдать осторожность при работе с потоками, так как неправильная синхронизация доступа к общей памяти может привести к гонкам данных и другим проблемам. Для обеспечения безопасности потоков в Win32 API существует множество механизмов синхронизации, таких как мьютексы, семафоры и критические секции.

Процессы в Windows изолируют друг друга, что обеспечивает надежность и безопасность работы операционной системы. Каждый процесс имеет свою собственную область памяти и независимые ресурсы. Взаимодействие между процессами может осуществляться через механизмы межпроцессного взаимодействия (IPC), такие как каналы, разделяемая память и сигналы.

Win32 API предоставляет функциональность для управления процессами, включая их создание, завершение, приоритеты выполнения и получение информации о них. Создание новых процессов выполняется с использованием функции “CreateProcess”, которая позволяет запустить новое приложение в новом процессе. Управление процессами в Win32 API позволяет определить их приоритет, что влияет на распределение ресурсов компьютера и уровень их выполнения. Для многих функций требуется дескриптор процесса. Чтобы получить дескриптор процесса для запущенного процесса, нужно передать его идентификатор процесса (полученный из EnumProcesses) в функцию OpenProcess . Также нужно вызвать функцию CloseHandle по завершении работы с дескриптором процесса.[2]

Итак, понимание работы с потоками и процессами в операционной системе Windows и использование соответствующих функций Win32 API позволяют разработчикам эффективно управлять задачами, ресурсами и процессами в системе, повышая производительность и надежность приложений.

# **3 РЕАЛИЗАЦИЯ ПРОГРАММНОГО ПРОДУКТА**

Для выполнения задачи было разработано приложение, представляющее собой диспетчер задач для операционной системы Windows. Это приложение позволяет пользователю отслеживать и управлять процессами, выполняющимися в системе.

Список процессов загружается при запуске приложения с использованием функционала Win32 API. Полученные процессы отображаются в элементе списка. Пользователь имеет возможность выполнять различные действия, включая завершение процессов, обновление списка, приостановку и возобновление процессов, а также запуск новых процессов.

Реализация приложения для управления процессами и потоками в рамках данной лабораторной работы включала в себя важные шаги, связанные с взаимодействием с операционной системой Windows и использованием соответствующих API. В начале разработки, необходимо было получить список активных процессов в системе. Это достигается с помощью использования функции CreateToolhelp32Snapshot, которая создает снимок текущих процессов в системе. Затем используется структура PROCESSENTRY32 для перебора процессов и получения информации о каждом из них, включая имя исполняемого файла.

Для выполнения различных действий с процессами, таких как завершение, приостановка и возобновление, приложение использует функции, предоставляемые Win32 API. Например, функция TerminateProcess позволяет принудительно завершить процесс, а функции SuspendThread и ResumeThread могут приостанавливать и возобновлять выполнение отдельных потоков в процессе. Эти операции требуют правильного управления дескрипторами процессов и потоков.

Когда вызывается функция TerminateProcess, операционная система немедленно завершает указанный процесс без возможности выполнения завершающих операций и очистки ресурсов. Это может привести к утечке ресурсов и непредсказуемым последствиям для системы. Файлы, открытые процессом, могут остаться заблокированными, и данные могут быть повреждены. Поэтому принудительное завершение процесса следует использовать осторожно, только при крайней необходимости.

Когда процесс приостанавливается с использованием функции SuspendThread, все его потоки приостанавливают выполнение. Это может быть полезно для временного приостановления выполнения процесса, чтобы провести анализ его состояния или предотвратить выполнение определенных действий. Однако долгосрочное приостановление потоков может привести к блокировке системы или вызвать неответственность внутри самого приложения. Функция ResumeThread позволяет возобновить выполнение потоков после приостановки. Поэтому при работе с приостановкой и возобновлением процессов важно тщательно контролировать время их блокировки, чтобы избежать негативного воздействия на систему и стабильность работы приложений.

Для анализа процессов, приложение использует функции GetProcessTimes и GetProcessMemoryInfo, которые предоставляют информацию о времени выполнения и использовании памяти процесса. Полученные данные помогают пользователям оценить эффективность и потребление ресурсов каждого процесса в системе. Работа с потоками и процессами в данной лабораторной работе демонстрирует важные аспекты программирования под Windows и подчеркивает значимость корректного управления ресурсами и выполнения операций с процессами.

Разработанное приложение предоставляет возможность мониторинга и управления процессами в операционной системе Windows, что является важным инструментом для повышения эффективности и надежности системы. Оно демонстрирует использование Win32 API для выполнения задач, связанных с управлением процессами и потоками, и может быть полезным инструментом для системных администраторов и разработчиков при работе с процессами в Windows.

# **4 РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ ПРОГРАММЫ**

В данной лабораторной было создано приложение для мониторинга и управления процессами, в котором имеется возможность завершить, приостановить и возобновить процессы. Также можно исследовать их на потребление памяти и время работы. (рисунок 4.1).

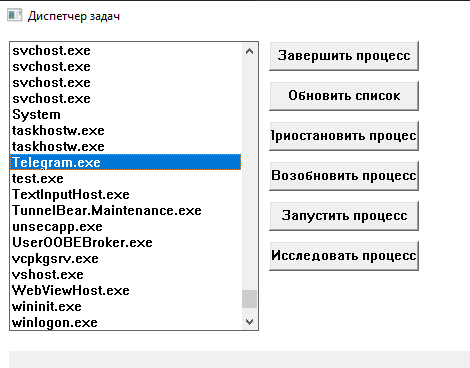


Рисунок 4.1 – Главное окно приложения

Есть возможность обновить список процессов на кнопку, есть возможность завершить процесс при надобности. На рисунке 4.1 виден процесс Telegram.exe, после нажатия на кнопку завершить процесс, в системе запущенного процесса больше нет. (рисунок 4.2).

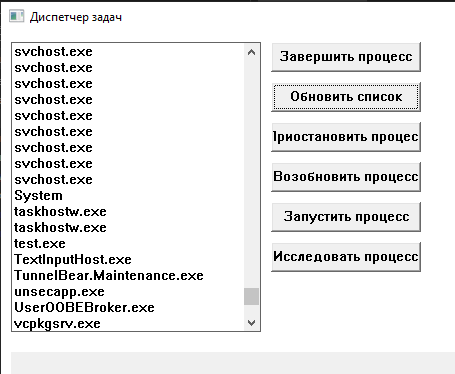


Рисунок 4.2 ­– Работа кнопки завершения процесса

Также можно приостановить процесс и возобновить его, чтобы потоки процесса перестали работать на время. Еще имеется возможность создать новый процесс. Для этого нужно нажать на соответствующую кнопку и выбрать исполняемый файл (.exe) для запуска процесса. (рисунок 4.3, рисунок 4.4).

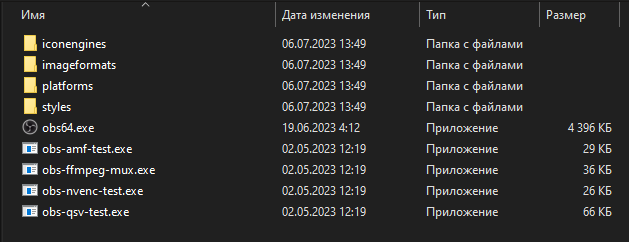


Рисунок 4.3 – Папка с исполняемым фалов obs64.exe

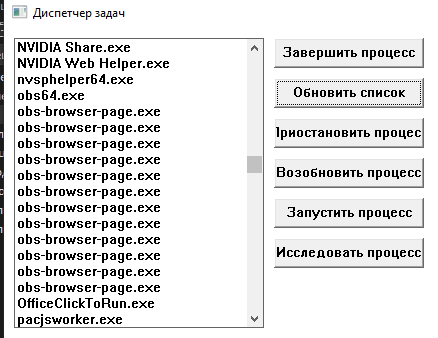


Рисунок 4.4 – Процесс obs64.exe появился в списке

Также имеется возможность исследовать процесс. При нажатии на кнопку исследовать процесс в поле снизу будет показано время работы процесса и его потребление памяти. (рисунок 4.5).

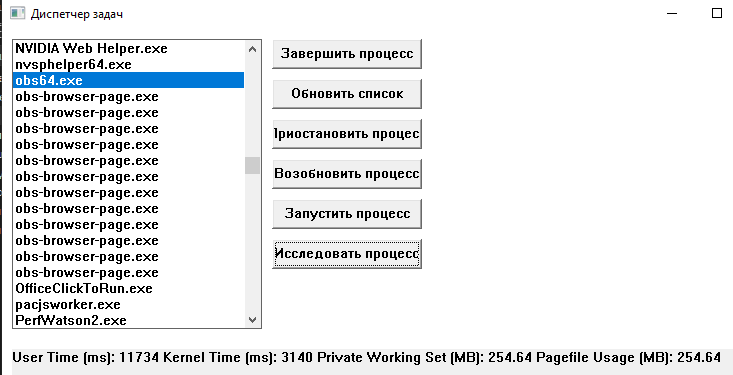


Рисунок 4.5 – Исследование процесса

# **ЗАКЛЮЧЕНИЕ**

В заключении данной лабораторной работы можно отметить, что она предоставила возможность изучить и применить различные аспекты управления процессами и потоками в операционной системе Windows. Были рассмотрены ключевые элементы работы с процессами, включая запуск, завершение, приостановку и возобновление исследуемых задач. Это позволило лучше понять механизмы взаимодействия приложений и операционной системы.

В ходе лабораторной работы было разработано приложение, способное анализировать эффективность процессов, отображая информацию о затрате процессами ресурсов и времени. Это приложение также позволяет пользователю взаимодействовать с процессами, выполняя действия, такие как завершение и приостановка. Подобные инструменты имеют важное значение для управления ресурсами системы и обеспечения ее стабильности.

Кроме того, лабораторная работа подчеркнула важность осторожного и осознанного подхода при работе с процессами и потоками, так как неправильное управление ими может вызвать нежелательные последствия. Отслеживание и анализ ресурсов, выделенных процессам, помогает в повышении эффективности системы и ее надежности. Полученный опыт и знания о работе с процессами и потоками в операционной системе Windows могут быть полезными в разработке и обслуживании приложений, а также в решении задач системного администрирования.

# **СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1. Сведения о процессе [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/psapi/process-information
2. Создание потоков [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/procthread/creating-threads

# **ПРИЛОЖЕНИЕ А**

#include <Windows.h>

#include <TlHelp32.h>

#include <tchar.h>

#include <psapi.h>

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

int currentScrollPos = 0;

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

WNDCLASSEX wc = { sizeof(WNDCLASSEX), CS\_HREDRAW | CS\_VREDRAW, WindowProc, 0L, 0L, GetModuleHandle(NULL), NULL, NULL, NULL, NULL, \_T("TaskManager"), NULL };

RegisterClassEx(&wc);

HWND hwnd = CreateWindow(wc.lpszClassName, \_T("Диспетчер задач"), WS\_OVERLAPPEDWINDOW, 100, 100, 800, 700, NULL, NULL, wc.hInstance, NULL);

ShowWindow(hwnd, nCmdShow);

MSG msg;

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

TranslateMessage(&msg);

DispatchMessage(&msg);

}

return static\_cast<int>(msg.wParam);

}

void AnalyzeProcessEfficiency(DWORD processId, HWND resultText) {

HANDLE hProcess = OpenProcess(PROCESS\_QUERY\_INFORMATION | PROCESS\_VM\_READ, FALSE, processId);

if (hProcess != NULL) {

FILETIME createTime, exitTime, kernelTime, userTime;

if (GetProcessTimes(hProcess, &createTime, &exitTime, &kernelTime, &userTime)) {

ULONGLONG userTimeMS = userTime.dwHighDateTime \* 1000000000LL + userTime.dwLowDateTime;

ULONGLONG kernelTimeMS = kernelTime.dwHighDateTime \* 1000000000LL + kernelTime.dwLowDateTime;

TCHAR result[256];

\_stprintf\_s(result, \_T("User Time (ms): %lld \n"), userTimeMS / 10000);

SendMessage(resultText, EM\_REPLACESEL, 0, (LPARAM)result);

\_stprintf\_s(result, \_T("Kernel Time (ms): %lld \n"), kernelTimeMS / 10000);

SendMessage(resultText, EM\_REPLACESEL, 0, (LPARAM)result);

}

PROCESS\_MEMORY\_COUNTERS\_EX pmc;

if (GetProcessMemoryInfo(hProcess, (PROCESS\_MEMORY\_COUNTERS\*)&pmc, sizeof(pmc))) {

TCHAR result[256];

\_stprintf\_s(result, \_T("Private Working Set (MB): %.2f \n"), static\_cast<double>(pmc.PrivateUsage) / (1024 \* 1024));

SendMessage(resultText, EM\_REPLACESEL, 0, (LPARAM)result);

\_stprintf\_s(result, \_T("Pagefile Usage (MB): %.2f\n"), static\_cast<double>(pmc.PagefileUsage) / (1024 \* 1024));

SendMessage(resultText, EM\_REPLACESEL, 0, (LPARAM)result);

}

/\* User Time(время пользователя в миллисекундах) : Это время, в течение которого процесс был в режиме пользователя(т.е., выполнял свой код вне ядра операционной системы)

Kernel Time(время ядра в миллисекундах) : Это время, в течение которого процесс выполнял код внутри ядра операционной системы.

Это включает в себя время, затраченное на обращение к системным ресурсам, таким как дисководы и файловые операции.

Private Working Set(МБ) : Это объем оперативной памяти, используемый процессом в МБ.

Этот объем включает в себя физическую память, зарезервированную и используемую только этим процессом.

Pagefile Usage(МБ) : Это объем памяти, используемый процессом в файлах подкачки операционной системы(pagefile) в МБ.

Эта память используется, когда физическая оперативная память исчерпана.\*/

CloseHandle(hProcess);

}

}

void LaunchProcess(const TCHAR\* exePath) {

PROCESS\_INFORMATION pi;

STARTUPINFO si;

ZeroMemory(&si, sizeof(si));

si.cb = sizeof(si);

if (CreateProcess(exePath, NULL, NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi)) {

CloseHandle(pi.hProcess);

CloseHandle(pi.hThread);

}

}

DWORD FindProcessId(const TCHAR\* processName) {

PROCESSENTRY32 pe32;

pe32.dwSize = sizeof(PROCESSENTRY32);

HANDLE hProcessSnap = CreateToolhelp32Snapshot(TH32CS\_SNAPPROCESS, 0);

if (Process32First(hProcessSnap, &pe32)) {

do {

if (\_tcsicmp(pe32.szExeFile, processName) == 0) {

CloseHandle(hProcessSnap);

return pe32.th32ProcessID;

}

} while (Process32Next(hProcessSnap, &pe32));

}

CloseHandle(hProcessSnap);

return 0;

}

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

switch (uMsg) {

case WM\_CREATE: {

HWND processList = CreateWindow(L"LISTBOX", NULL, WS\_CHILD | WS\_VISIBLE | WS\_VSCROLL | LBS\_STANDARD, 10, 10, 250, 300, hwnd, (HMENU)1, NULL, NULL);

PROCESSENTRY32 pe32;

pe32.dwSize = sizeof(PROCESSENTRY32);

HANDLE hProcessSnap = CreateToolhelp32Snapshot(TH32CS\_SNAPPROCESS, 0);

if (Process32First(hProcessSnap, &pe32)) {

do {

SendMessage(processList, LB\_ADDSTRING, 0, (LPARAM)pe32.szExeFile);

} while (Process32Next(hProcessSnap, &pe32));

}

CloseHandle(hProcessSnap);

HWND killButton = CreateWindow(\_T("BUTTON"), \_T("Завершить процесс"), WS\_CHILD | WS\_VISIBLE, 270, 10, 150, 30, hwnd, (HMENU)2, NULL, NULL);

HWND refreshButton = CreateWindow(\_T("BUTTON"), \_T("Обновить список"), WS\_CHILD | WS\_VISIBLE, 270, 50, 150, 30, hwnd, (HMENU)3, NULL, NULL);

HWND suspendButton = CreateWindow(\_T("BUTTON"), \_T("Приостановить процесс"), WS\_CHILD | WS\_VISIBLE, 270, 90, 150, 30, hwnd, (HMENU)4, NULL, NULL);

HWND resumeButton = CreateWindow(\_T("BUTTON"), \_T("Возобновить процесс"), WS\_CHILD | WS\_VISIBLE, 270, 130, 150, 30, hwnd, (HMENU)5, NULL, NULL);

HWND createProcessButton = CreateWindow(\_T("BUTTON"), \_T("Запустить процесс"), WS\_CHILD | WS\_VISIBLE, 270, 170, 150, 30, hwnd, (HMENU)6, NULL, NULL);

HWND analyzeButton = CreateWindow(\_T("BUTTON"), \_T("Исследовать процесс"), WS\_CHILD | WS\_VISIBLE, 270, 210, 150, 30, hwnd, (HMENU)7, NULL, NULL);

HWND resultText = CreateWindow(L"EDIT", NULL, WS\_CHILD | WS\_VISIBLE | WS\_VSCROLL | WS\_HSCROLL | ES\_MULTILINE | ES\_AUTOVSCROLL | ES\_AUTOHSCROLL | ES\_READONLY, 10, 320, 800, 150, hwnd, (HMENU)8, NULL, NULL);

break;

}

case WM\_COMMAND: {

int wmId = LOWORD(wParam);

if (wmId == 2) {

HWND processList = GetDlgItem(hwnd, 1);

int selectedIndex = SendMessage(processList, LB\_GETCURSEL, 0, 0);

if (selectedIndex != LB\_ERR) {

TCHAR processName[260];

SendMessage(processList, LB\_GETTEXT, selectedIndex, (LPARAM)processName);

int currentScrollPos = SendMessage(processList, LB\_GETTOPINDEX, 0, 0);

HANDLE hProcess = OpenProcess(PROCESS\_TERMINATE, FALSE, FindProcessId(processName));

if (hProcess != NULL) {

TerminateProcess(hProcess, 0);

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

PROCESSENTRY32 pe32;

pe32.dwSize = sizeof(PROCESSENTRY32);

HANDLE hProcessSnap = CreateToolhelp32Snapshot(TH32CS\_SNAPPROCESS, 0);

if (Process32First(hProcessSnap, &pe32)) {

do {

SendMessage(processList, LB\_ADDSTRING, 0, (LPARAM)pe32.szExeFile);

} while (Process32Next(hProcessSnap, &pe32));

}

CloseHandle(hProcessSnap);

CloseHandle(hProcess);

SendMessage(processList, LB\_SETTOPINDEX, currentScrollPos, 0);

}

}

}

else if (wmId == 3) {

HWND processList = GetDlgItem(hwnd, 1);

currentScrollPos = SendMessage(processList, LB\_GETTOPINDEX, 0, 0);

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

PROCESSENTRY32 pe32;

pe32.dwSize = sizeof(PROCESSENTRY32);

HANDLE hProcessSnap = CreateToolhelp32Snapshot(TH32CS\_SNAPPROCESS, 0);

if (Process32First(hProcessSnap, &pe32)) {

do {

SendMessage(processList, LB\_ADDSTRING, 0, (LPARAM)pe32.szExeFile);

} while (Process32Next(hProcessSnap, &pe32));

}

CloseHandle(hProcessSnap);

SendMessage(processList, LB\_SETTOPINDEX, currentScrollPos, 0);

}

else if (wmId == 4) {

HWND processList = GetDlgItem(hwnd, 1);

int selectedIndex = SendMessage(processList, LB\_GETCURSEL, 0, 0);

if (selectedIndex != LB\_ERR) {

TCHAR processName[260];

SendMessage(processList, LB\_GETTEXT, selectedIndex, (LPARAM)processName);

int currentScrollPos = SendMessage(processList, LB\_GETTOPINDEX, 0, 0);

HANDLE hProcess = OpenProcess(PROCESS\_SUSPEND\_RESUME, FALSE, FindProcessId(processName));

if (hProcess != NULL) {

SuspendThread(hProcess);

CloseHandle(hProcess);

}

}

}

else if (wmId == 5) {

HWND processList = GetDlgItem(hwnd, 1);

int selectedIndex = SendMessage(processList, LB\_GETCURSEL, 0, 0);

if (selectedIndex != LB\_ERR) {

TCHAR processName[260];

SendMessage(processList, LB\_GETTEXT, selectedIndex, (LPARAM)processName);

int currentScrollPos = SendMessage(processList, LB\_GETTOPINDEX, 0, 0);

HANDLE hProcess = OpenProcess(PROCESS\_SUSPEND\_RESUME, FALSE, FindProcessId(processName));

if (hProcess != NULL) {

ResumeThread(hProcess);

CloseHandle(hProcess);

}

}

}

else if (wmId == 6) {

OPENFILENAME ofn;

TCHAR szFileName[MAX\_PATH] = { 0 };

ZeroMemory(&ofn, sizeof(ofn));

ofn.lStructSize = sizeof(ofn);

ofn.hwndOwner = hwnd;

ofn.lpstrFilter = \_T("Исполняемые файлы (\*.exe)\0\*.exe\0");

ofn.lpstrFile = szFileName;

ofn.nMaxFile = MAX\_PATH;

ofn.Flags = OFN\_EXPLORER | OFN\_FILEMUSTEXIST;

if (GetOpenFileName(&ofn)) {

LaunchProcess(szFileName);

HWND processList = GetDlgItem(hwnd, 1);

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

PROCESSENTRY32 pe32;

pe32.dwSize = sizeof(PROCESSENTRY32);

HANDLE hProcessSnap = CreateToolhelp32Snapshot(TH32CS\_SNAPPROCESS, 0);

if (Process32First(hProcessSnap, &pe32)) {

do {

SendMessage(processList, LB\_ADDSTRING, 0, (LPARAM)pe32.szExeFile);

} while (Process32Next(hProcessSnap, &pe32));

}

CloseHandle(hProcessSnap);

}

}

else if (wmId == 7) {

HWND resultText = GetDlgItem(hwnd, 8);

SetWindowText(resultText, L"");

HWND processList = GetDlgItem(hwnd, 1);

int selectedIndex = SendMessage(processList, LB\_GETCURSEL, 0, 0);

if (selectedIndex != LB\_ERR) {

TCHAR processName[260];

SendMessage(processList, LB\_GETTEXT, selectedIndex, (LPARAM)processName);

DWORD processId = FindProcessId(processName);

if (processId != 0) {

HWND resultText = GetDlgItem(hwnd, 8);

AnalyzeProcessEfficiency(processId, resultText);

}

}

}

break;

}

case WM\_DESTROY: {

PostQuitMessage(0);

break;

}

default:

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

}

return 0;

}