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

ОТЧЁТ

к лабораторной работе №4

на тему

УПРАВЛЕНИЕ ПРОЦЕССАМИ И ПОТОКАМИ (WINDOWS). ПОРОЖДЕНИЕ, ЗАВЕРШЕНИЕ,

ИЗМЕНЕНИЕ ПРИОРИТЕТОВ ПРОЦЕССОВ И ПОТОКОВ, ИССЛЕДОВАНИЕ ЭФФЕКТИВНОСТИ.

Выполнил студент гр.153502 Богомолов М.А.

Проверил ассистент кафедры информатики Гриценко Н.Ю.

Минск 2023

**СОДЕРЖАНИЕ**

[1 Формулировка задачи 3](#_Toc148046360)

[2 Теоретические сведения 4](#_Toc148046361)

[3 Описание функций программы 5](#_Toc148046362)

[Список использованных источников 7](#_Toc148046363)

[Приложение А (обязательное) Листинг кода 8](#_Toc148046364)

# **1 ФОРМУЛИРОВКА ЗАДАЧИ**

Целью выполнения лабораторной работы является разработка приложение для отслеживания и управления процессами в системе, позволяющее приостанавливать, возобновлять и завершать процессы.

В качестве задачи необходимо реализовать возможность выбор процесса из списка всех запущенных и его дальнейшее завершение, приостановка, возобновление.

# **2 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

*Win32 API* предоставляет множество функций и возможностей для создания, управления и мониторинга процессов и потоков в операционной системе *Windows*.

Функция *TerminateProcess* (*processthreadsapi.h*) завершает указанный процесс и все его потоки [1]. То есть, передавая в эту функцию *ID* процесса типа *DWORD* и вызывая её, функция завершает каждый поток процесса с указанным *ID* по отдельности, тем самым завершая весь процесс целиком.

Для приостановки и возобновления работы процессов используется механизм, при котором каждый поток, принадлежащий указанному процессу, приостанавливается/возобновляется по отдельности, тем самым приостанавливая/возобновляя весь процесс целиком. *SuspendThread*() и *ResumeThread*() – функции для работы с потоками. Эти две функции позволяют приостанавливать и возобновлять выполнение задач(потоков) [2].

# **3 ОПИСАНИЕ ФУНКЦИЙ ПРОГРАММЫ**

Для того, чтобы завершить процесс, требуется выбрать его в элементе *ListView*, после чего необходимо нажать кнопку *Terminate Process*. Выбранный процесс завершится, а из списка текущих процессов исчезнет выбранный ранее процесс. Выбор процесса для в элементе *ListView* для операций с ним (Рисунок 1).

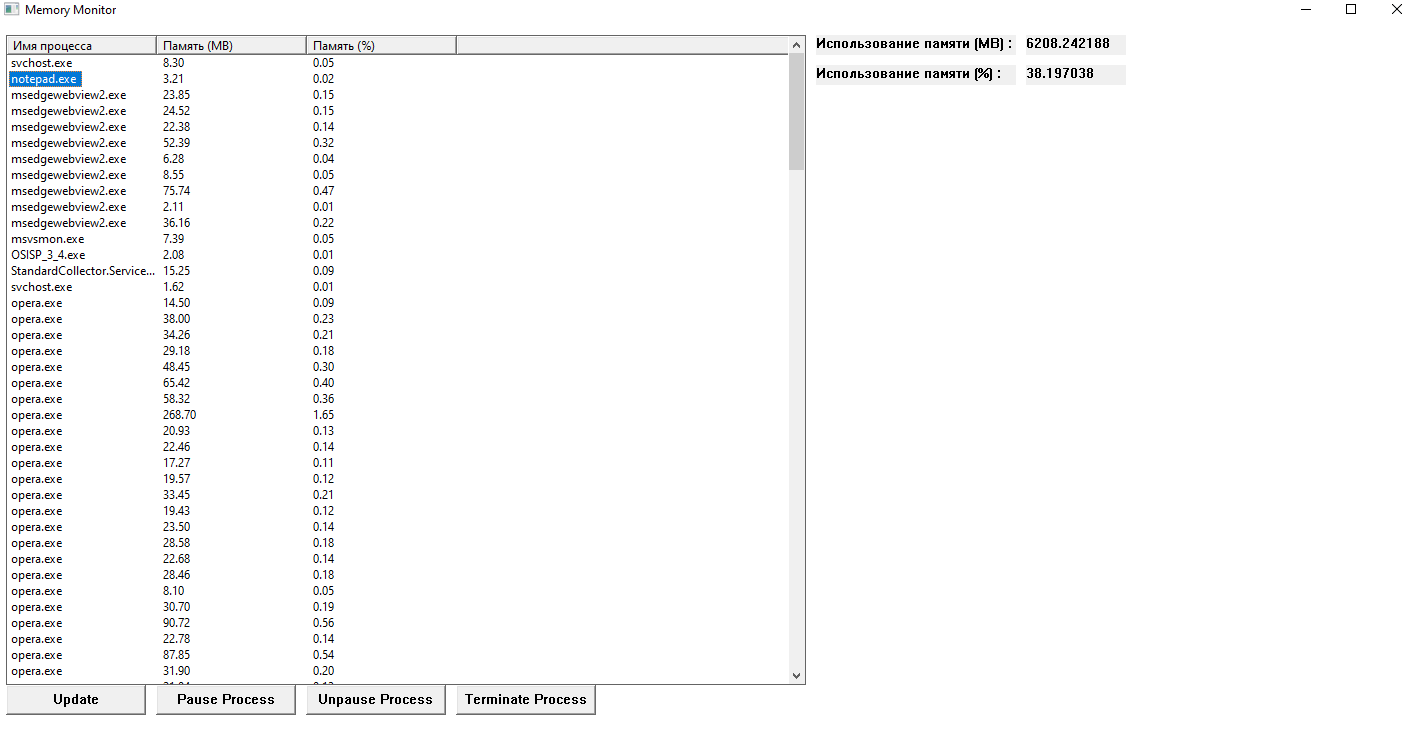


Рисунок 1 – Выбор процесса

Для примера, завершим процесс *notepad*.*exe* с помощью кнопки *Terminate Process*, чтобы обновить список запущенных процессов. Как можно заметить, поле *notepad*.*exe* исчезло из элемента *ListView* (Рисунок 2).

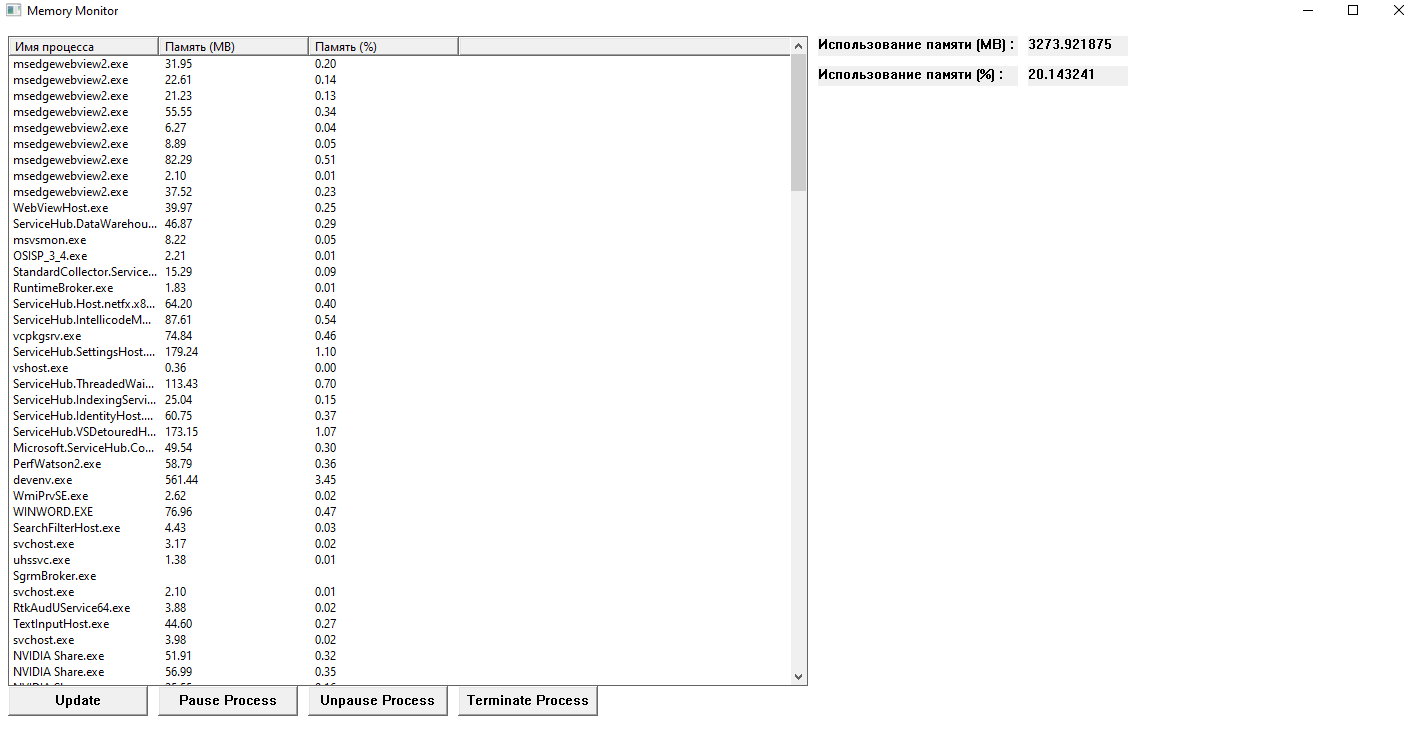
****

Рисунок 2 – Результат выполнения *Terminate Process*

Нажатие кнопки *Pause* *Process* для выбранного процесса, приостанавливает все его потоки.

Нажатие кнопки *Unpause* *Process* для выбранного процесса, восстанавливает все его потоки.

Нажатие кнопки *Pause Process* для процесса *notepad*.*exe* и сравнение с информацией о процессе *notepad*.*exe* в Диспетчере Задача *Windows* (Рисунок 3).

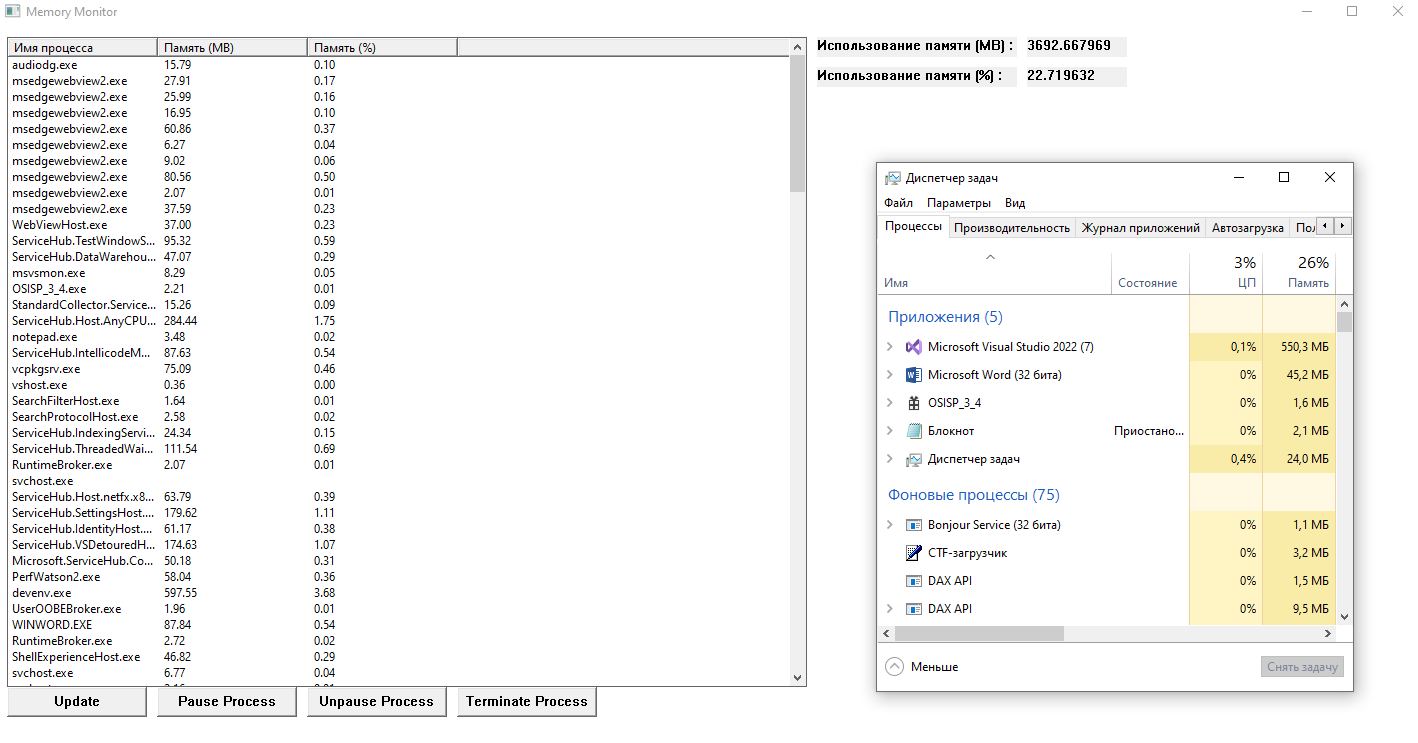


Рисунок 3 – Приостановка процесса *notepad.exe*

# **СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

[1] Функция TerminateProcess (processthreadsapi.h) [Электронный ресурс]. – Режим доступа: <https://learn.microsoft.com/ru-ru/windows/win32/api/processthreadsapi/nf-processthreadsapi-terminateprocess> – Дата доступа: 02.11.2023.

[2] Шаг 112 - SuspendThread() и ResumeThread() [Электронный ресурс]. – Режим доступа: <https://firststeps.ru/mfc/winapi/r.php?112> – Дата доступа: 02.11.2023.

# **ПРИЛОЖЕНИЕ А (обязательное) Листинг кода**

Листинг 1 – Файл OSISP\_3\_4.cpp

#include <windows.h>

#include <TlHelp32.h>

#include <CommCtrl.h>

#include <Psapi.h>

#include <stdio.h>

#include <thread>

#include <string>

#include <vector>

HINSTANCE hInst;

HWND hWnd;

HWND hListView;

HWND hUpdateButton;

HWND hPauseProcessButton;

HWND hUnpauseProcessButton;

HWND hTerminateProcessButton;

HWND hMemoryUsageMBLabel;

HWND hMemoryUsagePercentageLabel;

HWND hMemoryUsageMB;

HWND hMemoryUsagePercentage;

struct ProcessInfo {

DWORD dwProcessId;

};

std::vector<ProcessInfo> processInfoList;

#define IDC\_UPDATE\_BUTTON 1001

#define IDC\_PAUSE\_PROCESS\_BUTTON 1002

#define IDC\_UNPAUSE\_PROCESS\_BUTTON 1003

#define IDC\_TERMINATE\_PROCESS\_BUTTON 1004

void GetProcessList(HWND hListView);

void PauseProcess(HWND hListView);

void UnpauseProcess(HWND hListView);

void TerminateProcessC(HWND hListView);

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)

{

hInst = hInstance;

WNDCLASSEX wcex;

wcex.cbSize = sizeof(WNDCLASSEX);

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

wcex.lpfnWndProc = WndProc;

wcex.cbClsExtra = 0;

wcex.cbWndExtra = 0;

wcex.hInstance = hInstance;

wcex.hIcon = LoadIcon(hInstance, IDI\_APPLICATION);

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

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

wcex.lpszMenuName = NULL;

wcex.lpszClassName = L"MyMemoryMonitorApp";

wcex.hIconSm = LoadIcon(wcex.hInstance, IDI\_APPLICATION);

RegisterClassEx(&wcex);

INITCOMMONCONTROLSEX icex;

icex.dwICC = ICC\_LISTVIEW\_CLASSES;

InitCommonControlsEx(&icex);

hWnd = CreateWindow(L"MyMemoryMonitorApp", L"Memory Monitor", WS\_OVERLAPPEDWINDOW, CW\_USEDEFAULT, 0, CW\_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);

hListView = CreateWindowEx(0, WC\_LISTVIEW, L"", WS\_VISIBLE | WS\_CHILD | WS\_BORDER | LVS\_REPORT | LVS\_SINGLESEL,

10, 10, 800, 650, hWnd, NULL, hInst, NULL);

LV\_COLUMN lvc;

lvc.mask = LVCF\_FMT | LVCF\_WIDTH | LVCF\_TEXT;

lvc.fmt = LVCFMT\_LEFT;

lvc.cx = 150;

lvc.pszText = const\_cast<LPWSTR>(L"Имя процесса");;

ListView\_InsertColumn(hListView, 0, &lvc);

lvc.cx = 150;

lvc.pszText = const\_cast<LPWSTR>(L"Память (MB)");

ListView\_InsertColumn(hListView, 1, &lvc);

lvc.cx = 150;

lvc.pszText = const\_cast<LPWSTR>(L"Память (%)");

ListView\_InsertColumn(hListView, 2, &lvc);

hMemoryUsageMBLabel = CreateWindow(L"STATIC", L"Использование памяти (MB) :",

WS\_VISIBLE | WS\_CHILD, 820, 10, 200, 20, hWnd, NULL, hInst, NULL);

hMemoryUsagePercentageLabel = CreateWindow(L"STATIC", L"Использование памяти (%) :",

WS\_VISIBLE | WS\_CHILD, 820, 40, 200, 20, hWnd, NULL, hInst, NULL);

hMemoryUsageMB = CreateWindow(L"STATIC", L"",

WS\_VISIBLE | WS\_CHILD, 1030, 10, 100, 20, hWnd, NULL, hInst, NULL);

hMemoryUsagePercentage = CreateWindow(L"STATIC", L"",

WS\_VISIBLE | WS\_CHILD, 1030, 40, 100, 20, hWnd, NULL, hInst, NULL);

hUpdateButton = CreateWindow(L"BUTTON", L"Load", WS\_CHILD | WS\_VISIBLE | BS\_PUSHBUTTON,

10, 660, 140, 30, hWnd, (HMENU)IDC\_UPDATE\_BUTTON, hInst, NULL);

hPauseProcessButton = CreateWindow(L"BUTTON", L"Pause Process", WS\_CHILD | WS\_VISIBLE | BS\_PUSHBUTTON,

160, 660, 140, 30, hWnd, (HMENU)IDC\_PAUSE\_PROCESS\_BUTTON, hInst, NULL);

hUnpauseProcessButton = CreateWindow(L"BUTTON", L"Unpause Process", WS\_CHILD | WS\_VISIBLE | BS\_PUSHBUTTON,

310, 660, 140, 30, hWnd, (HMENU)IDC\_UNPAUSE\_PROCESS\_BUTTON, hInst, NULL);

hTerminateProcessButton = CreateWindow(L"BUTTON", L"Terminate Process", WS\_CHILD | WS\_VISIBLE | BS\_PUSHBUTTON,

460, 660, 140, 30, hWnd, (HMENU)IDC\_TERMINATE\_PROCESS\_BUTTON, hInst, NULL);

if (!hWnd)

{

return FALSE;

}

EnableWindow(hPauseProcessButton, FALSE);

EnableWindow(hUnpauseProcessButton, FALSE);

EnableWindow(hTerminateProcessButton, FALSE);

ShowWindow(hWnd, nCmdShow);

UpdateWindow(hWnd);

MSG msg;

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

{

TranslateMessage(&msg);

DispatchMessage(&msg);

}

return (int)msg.wParam;

}

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)

{

switch (message)

{

case WM\_COMMAND:

{

int wmId = LOWORD(wParam);

int wmEvent = HIWORD(wParam);

switch (wmId)

{

case IDC\_PAUSE\_PROCESS\_BUTTON:

{

std::thread(PauseProcess, hListView).detach();

break;

}

case IDC\_UNPAUSE\_PROCESS\_BUTTON:

{

std::thread(UnpauseProcess, hListView).detach();

break;

}

case IDC\_UPDATE\_BUTTON:

{

SetWindowText(hUpdateButton, L"Update");

std::thread(GetProcessList, hListView).detach();

EnableWindow(hPauseProcessButton, TRUE);

EnableWindow(hUnpauseProcessButton, TRUE);

EnableWindow(hTerminateProcessButton, TRUE);

break;

}

case IDC\_TERMINATE\_PROCESS\_BUTTON:

{

std::thread(TerminateProcessC, hListView).detach();

break;

}

}

}

break;

case WM\_PAINT:

{

PAINTSTRUCT ps;

HDC hdc = BeginPaint(hWnd, &ps);

EndPaint(hWnd, &ps);

return 0;

}

break;

case WM\_DESTROY:

PostQuitMessage(0);

break;

default:

return DefWindowProc(hWnd, message, wParam, lParam);

}

return 0;

}

void GetProcessList(HWND hListView)

{

double AllMemoryUsageMB = 0;

double AllMemoryUsagePercentage = 0;

ListView\_DeleteAllItems(hListView);

processInfoList.clear();

HANDLE hProcessSnap;

PROCESSENTRY32 pe32;

hProcessSnap = CreateToolhelp32Snapshot(TH32CS\_SNAPPROCESS, 0);

if (hProcessSnap == INVALID\_HANDLE\_VALUE)

{

return;

}

pe32.dwSize = sizeof(PROCESSENTRY32);

if (Process32First(hProcessSnap, &pe32))

{

do

{

LVITEM lvi;

lvi.mask = LVIF\_TEXT;

lvi.iItem = 0;

lvi.iSubItem = 0;

lvi.pszText = pe32.szExeFile;

processInfoList.push\_back({ pe32.th32ProcessID });

ListView\_InsertItem(hListView, &lvi);

PROCESS\_MEMORY\_COUNTERS\_EX pmc;

HANDLE hProcess = OpenProcess(PROCESS\_QUERY\_INFORMATION | PROCESS\_VM\_READ, FALSE, pe32.th32ProcessID);

if (hProcess)

{

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

{

lvi.iSubItem = 1;

double memoryInMB = (double)(pmc.PrivateUsage / (1024.0 \* 1024.0)); // Рабочий набор в мегабайтах

WCHAR szMemoryUsage[64];

swprintf\_s(szMemoryUsage, L"%.2lf", memoryInMB);

AllMemoryUsageMB += memoryInMB;

lvi.pszText = szMemoryUsage;

ListView\_SetItem(hListView, &lvi);

lvi.iSubItem = 2;

MEMORYSTATUSEX memInfo;

memInfo.dwLength = sizeof(MEMORYSTATUSEX);

GlobalMemoryStatusEx(&memInfo);

double memoryUsagePercentage = (memoryInMB / (memInfo.ullTotalPhys / (1024.0 \* 1024.0))) \* 100.0; // Использование памяти в процентах

WCHAR szMemoryUsagePercentage[64];

swprintf\_s(szMemoryUsagePercentage, L"%.2lf", memoryUsagePercentage);

AllMemoryUsagePercentage += memoryUsagePercentage;

lvi.pszText = szMemoryUsagePercentage;

ListView\_SetItem(hListView, &lvi);

}

CloseHandle(hProcess);

}

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

std::wstring AllMemoryUsageMBAsString = std::to\_wstring(AllMemoryUsageMB);

SetWindowText(hMemoryUsageMB, AllMemoryUsageMBAsString.c\_str());

std::wstring AllMemoryUsagePercentageAsString = std::to\_wstring(AllMemoryUsagePercentage);

SetWindowText(hMemoryUsagePercentage, AllMemoryUsagePercentageAsString.c\_str());

}

CloseHandle(hProcessSnap);

}

void PauseProcess(HWND hListView) {

int selectedIdx = ListView\_GetNextItem(hListView, -1, LVNI\_SELECTED);

if (selectedIdx != -1) {

if (selectedIdx < processInfoList.size()) {

DWORD dwProcessId = processInfoList[processInfoList.size() - selectedIdx - 1].dwProcessId;

HANDLE hThreadSnapshot = CreateToolhelp32Snapshot(TH32CS\_SNAPTHREAD, 0);

if (hThreadSnapshot != INVALID\_HANDLE\_VALUE) {

THREADENTRY32 te32;

te32.dwSize = sizeof(THREADENTRY32);

if (Thread32First(hThreadSnapshot, &te32)) {

do {

if (te32.th32OwnerProcessID == dwProcessId) {

HANDLE hThread = OpenThread(THREAD\_SUSPEND\_RESUME, FALSE, te32.th32ThreadID);

if (hThread != NULL) {

SuspendThread(hThread);

CloseHandle(hThread);

}

}

} while (Thread32Next(hThreadSnapshot, &te32));

}

CloseHandle(hThreadSnapshot);

}

}

else

{

return;

}

}

else {

return;

}

MessageBox(NULL, L"Процесс успешно приостановлен", L"OK", MB\_ICONINFORMATION);

GetProcessList(hListView);

}

void UnpauseProcess(HWND hListView) {

int selectedIdx = ListView\_GetNextItem(hListView, -1, LVNI\_SELECTED);

if (selectedIdx != -1) {

if (selectedIdx < processInfoList.size()) {

DWORD dwProcessId = processInfoList[processInfoList.size() - selectedIdx - 1].dwProcessId;

HANDLE hThreadSnapshot = CreateToolhelp32Snapshot(TH32CS\_SNAPTHREAD, 0);

if (hThreadSnapshot != INVALID\_HANDLE\_VALUE) {

THREADENTRY32 te32;

te32.dwSize = sizeof(THREADENTRY32);

if (Thread32First(hThreadSnapshot, &te32)) {

do {

if (te32.th32OwnerProcessID == dwProcessId) {

HANDLE hThread = OpenThread(THREAD\_SUSPEND\_RESUME, FALSE, te32.th32ThreadID);

if (hThread != NULL) {

ResumeThread(hThread);

CloseHandle(hThread);

}

}

} while (Thread32Next(hThreadSnapshot, &te32));

}

CloseHandle(hThreadSnapshot);

}

}

else

{

return;

}

}

else

{

return;

}

MessageBox(NULL, L"Процесс успешно возобновлён", L"OK", MB\_ICONINFORMATION);

GetProcessList(hListView);

}

void TerminateProcessC(HWND hListView) {

int selectedIdx = ListView\_GetNextItem(hListView, -1, LVNI\_SELECTED);

if (selectedIdx != -1) {

if (selectedIdx < processInfoList.size()) {

DWORD dwProcessId = processInfoList[processInfoList.size() - selectedIdx - 1].dwProcessId;

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

if (hProcess != NULL) {

if (TerminateProcess(hProcess, 0)) {

}

else {

return;

}

CloseHandle(hProcess);

}

else {

return;

}

}

else {

return;

}

}

else {

return;

}

MessageBox(NULL, L"Процесс успешно завершён", L"OK", MB\_ICONINFORMATION);

GetProcessList(hListView);

}