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.

Студент: гр.153502 Сидорова У.Ю.

Проверил: ассистент кафедры информатики Гриценко Н.Ю.

Минск 2023

**СОДЕРЖАНИЕ**

[1 Цель работы 3](#_Toc145669228)

[2 Теоретические сведения 3](#_Toc145669228)

[3 Результат работы программы 4](#_Toc145669229)

[Список использованных источников 6](#_Toc145669230)

[Приложение A 7](#_Toc145669231)

1 ЦЕЛЬ РАБОТЫ

Целью выполнения лабораторной работы является создание приложения на *Win32 API*, которое управляет процессами (*Windows*).

В качестве задачи необходимо разработать приложение для отслеживания и управления процессами в системе, позволяющее приостанавливать, возобновлять, создавать и завершать процессы.

**2** ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

В данной лабораторной работе подпрограммы, реализованные в библиотеке *Ntoskrnl.lib*. Эта библиотека *DLL* содержит фактические реализации подпрограмм системных служб, которые используются драйверами в режиме ядра, с помощью вызова точки входа *Nt* и *Zw*. С несколькими незначительными исключениями каждая точка входа в *Ntdll.dll* для подпрограммы *Nt* имеет соответствующую точку входа для подпрограммы *Zw* . Описание подпрограмм *Nt* , которые могут вызываться приложениями, описаны в заголовочных файлах *Winternl*, *winternl.h* [1]*.*

Функция *NtQuerySystemInformation* извлекает указанные сведения о системе. Используется с параметром *SystemProcessInformation.* Возвращает массив *SYSTEM\_PROCESS\_INFORMATION* структур, по одной для каждого процесса, выполняющегося в системе. Эти структуры содержат сведения об использовании ресурсов каждого процесса, включая количество потоков и дескрипторов, используемых процессом, пиковое использование файла подкачки и количество страниц памяти, выделенных процессом [2].

Для приостановки процесса используется функция *DebugActiveProcess.* Она озволяет отладчику подключаться к активному процессу и выполнять его отладку [3].

Для возобновления процесса вызывается функция *DebugActiveProcessStop.* Она останавливает отладку отладчика указанного процесса [4].

Создается процесс с помощью функции *CreateProcess.* Она создает новый процесс, который выполняется независимо от процесса создания. Для простоты эта связь называется отношением "родители-потомки". Если *CreateProcess* завершается успешно, он возвращает структуру *PROCESS\_INFORMATION*, содержащую дескрипторы и идентификаторы для нового процесса и его основного потока. Дескрипторы потоков и процессов создаются с правами полного доступа, хотя вы можете ограничить доступ, если указать дескрипторы безопасности. Если эти дескрипторы больше не нужны, их нужно закрыть с помощью функции *CloseHandle* [5].

Функция *TerminateProcess* завершает указанный процесс и все его потоки [6].

3 РЕЗУЛЬТАТ РАБОТЫ ПРОГРАММЫ

При запуске программы открывается оконное приложение (рисунок 1).

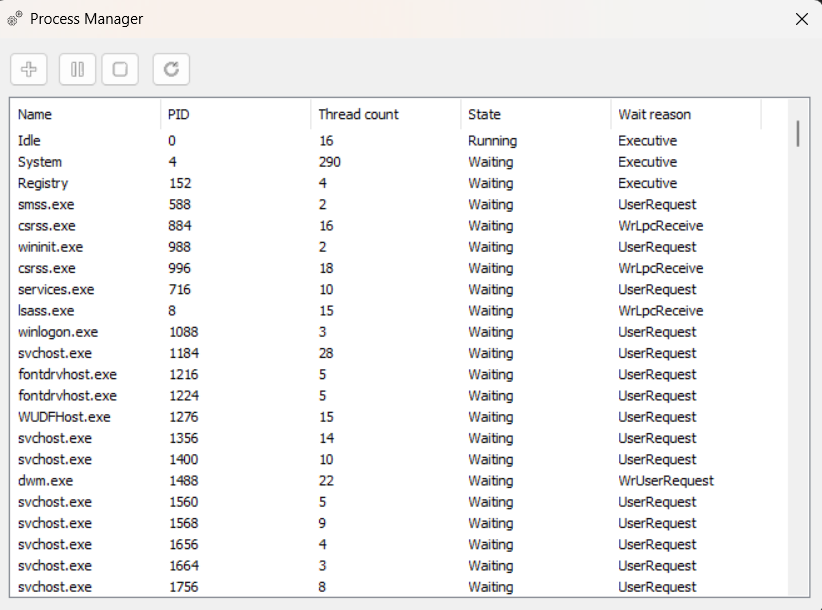


Рисунок 1 – Графический интерфейс программы

Функциональные возможности приложения:

– просмотр выполняемых процессов и информации о них;

При запуске программы в виде таблицы отображаются выполняемые процессы и дополнительная информация о них: имя, *Id*, количество потоков, состояние и причина ожидания процесса (рисунок 2).

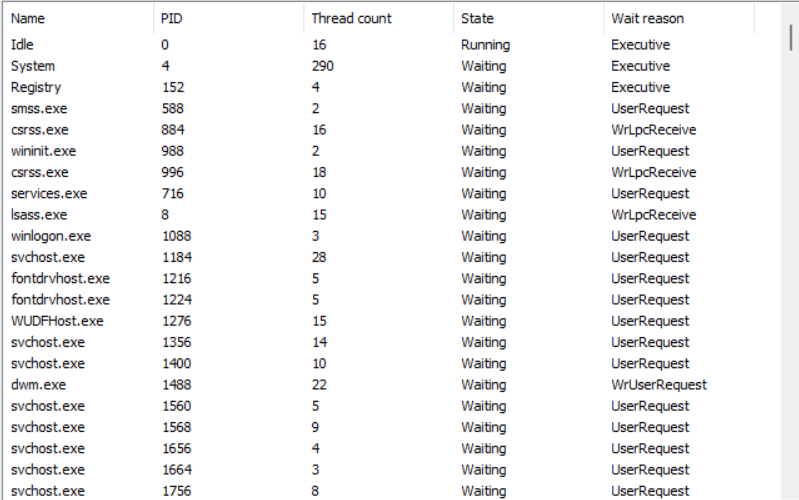


Рисунок 2 – Отображение выполняемых процессов

– обновление списка процессов;

Для обновления списка процессов необходимо нажать кнопку *refresh* (крайняя слева) в левом верхнем углу. Список автоматически обновляется после приостановки, возобновления и завершения процессов.

– приостановить процесс;

Для того, чтобы приостановить процесс необходимо выделить интересующий процесс и нажать кнопку *pause* (вторая слева) в верхнем левом углу(рисунок 3).

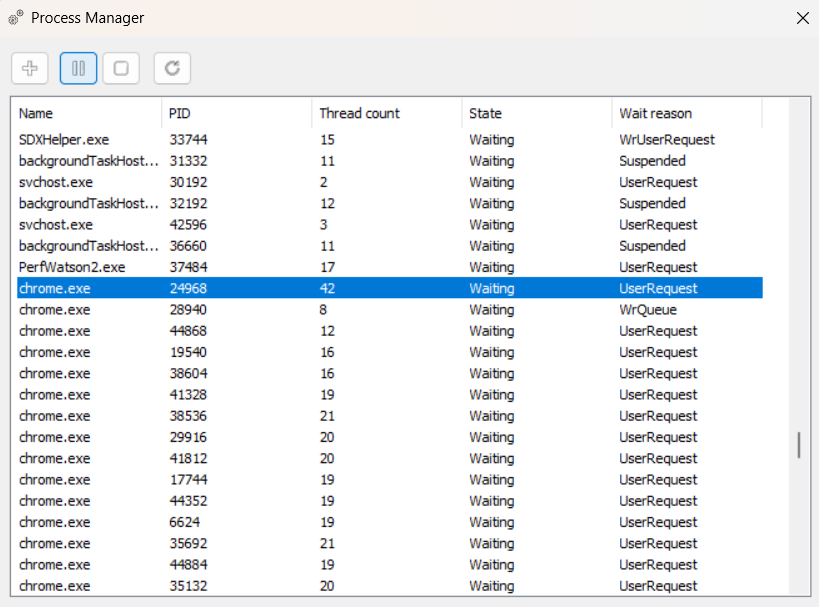


Рисунок 3 – Состояние до приостановки процесса

После нажатия кнопки *pause*, причина ожидания данного процесса изменится на *Suspended* (приостановленный), а сама кнопка, при выделении этого процесса, будет меняться на *play* (рисунок 4).

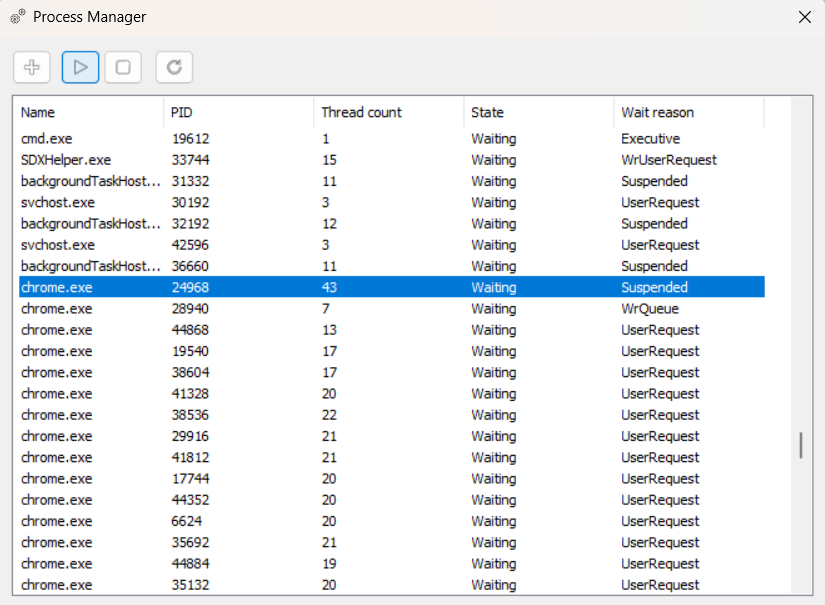


Рисунок 4 – Состояние после приостановки процесса

– возобновление процесса;

Для возобновления процесса необходимо выбрать процесс с причиной ожидания *Suspended*, и нажать кнопку *play* (вторая слева) в верхнем левом углу. После нажатия пропадет статус *Suspended* и, при нажатии на данный процесс, кнопка поменяется на *pause* (рисунок 5).

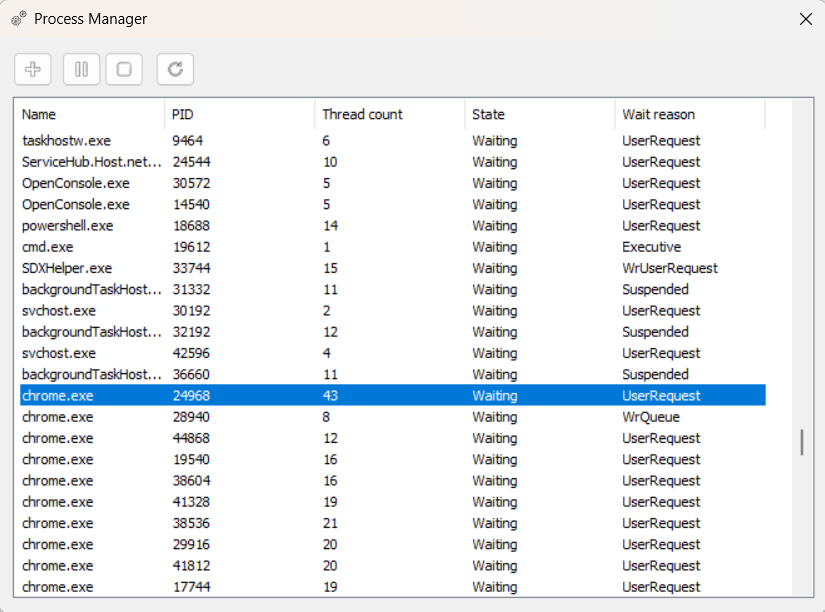


Рисунок 5 – Состояние после возобновления процесса

– завершение процесса;

Для завершения процесса необходимо выбрать процесс и нажать кнопку *stop* (третья слева) в верхнем левом углу. После нажатия процесс завершится и пропадет из списка процессов (рисунок 6, рисунок 7).

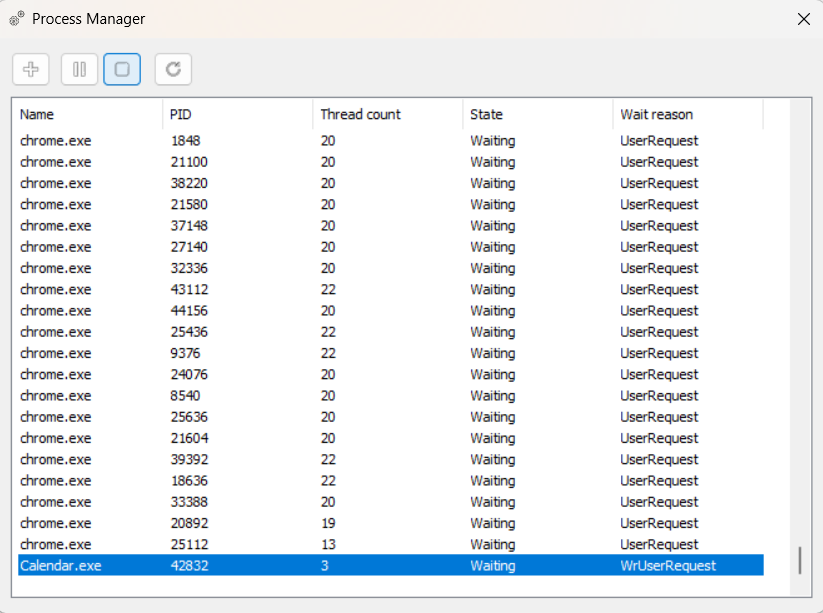


Рисунок 6 – Состояние до завершения процесса

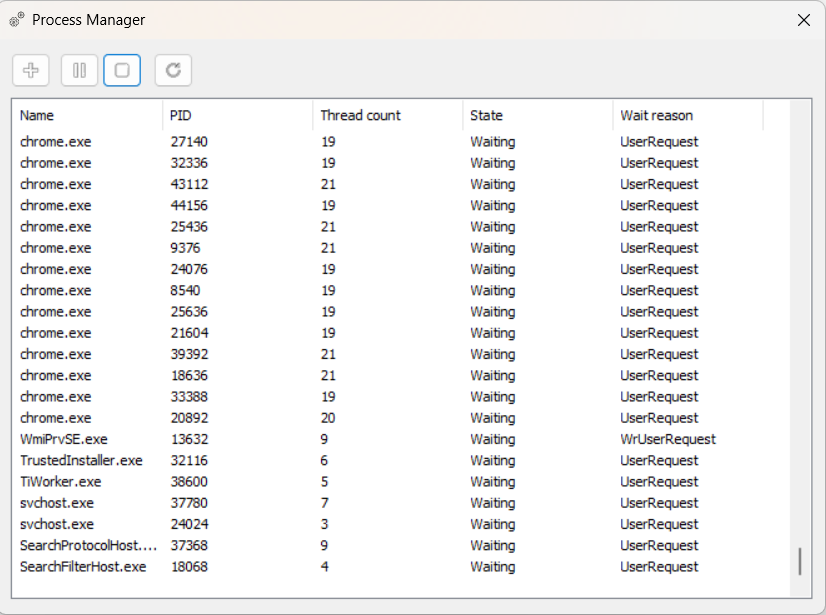


Рисунок 7 – Состояние после завершения процесса

– добавление процесса;

Для создания процесса необходимо выбрать процесс и нажать кнопку *add* (первая слева) в верхнем левом углу. После нажатия откроется файловый менеджер, ограниченный выбором файлов с расширением *exe* (рисунок 8).

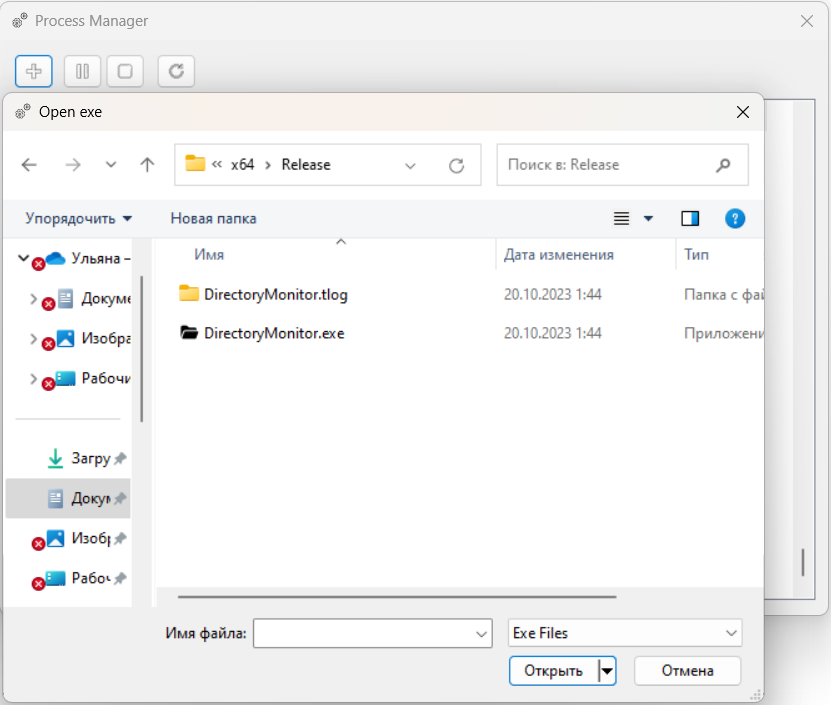


Рисунок 8 – Состояние после завершения процесса

После выбора нужного файла создастся процесс. Для его отображения необходимо обновить список процессов (рисунок 9, рисунок 10).

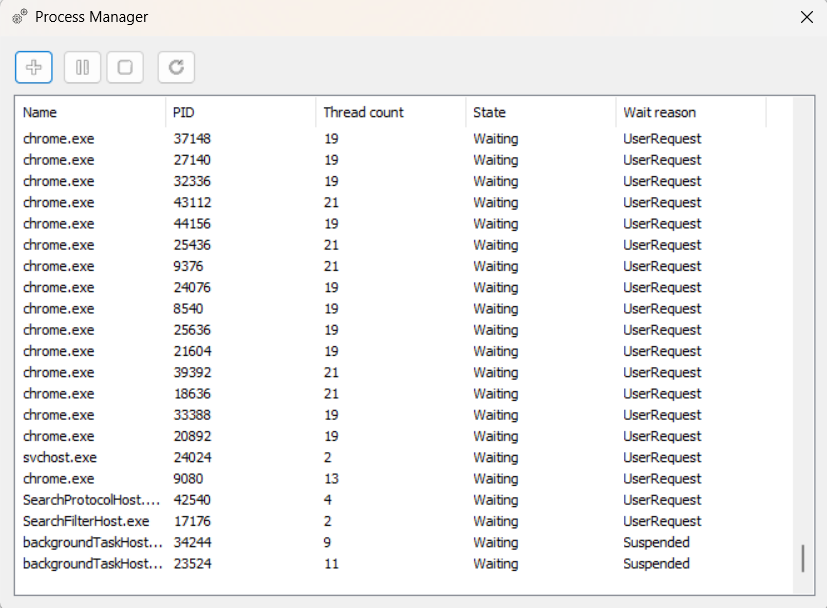


Рисунок 9 – Состояние до добавления процесса

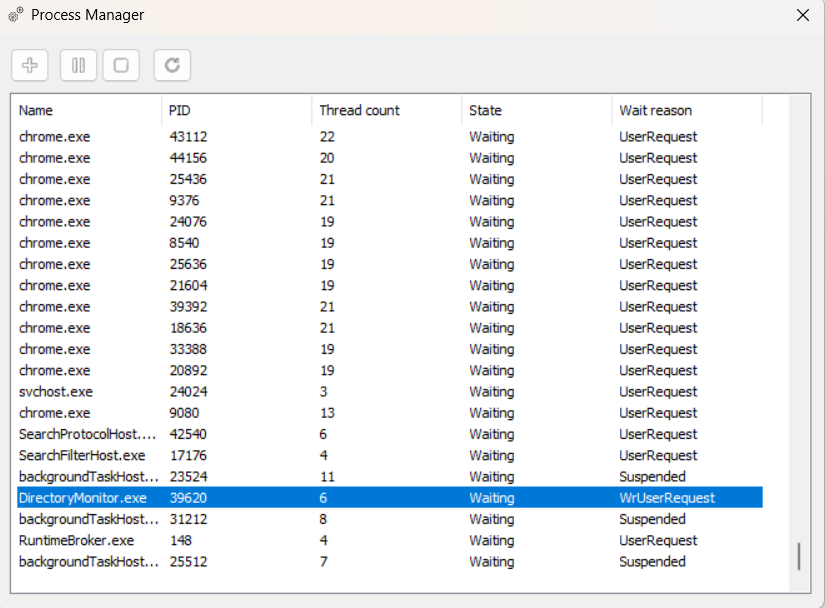


Рисунок 10 – Состояние после добавления процесса

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Библиотеки и заголовки [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://learn.microsoft.com/ru-ru/windows-hardware/drivers/kernel/libraries-and-headers> – Дата доступа 15.10.2023.

[2] Функция NtQuerySystemInformation (winternl.h) [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://learn.microsoft.com/ru-ru/windows/win32/api/winternl/nf-winternl-ntquerysysteminformation?redirectedfrom=MSDN> – Дата доступа 15.10.2023.

[3] Функция DebugActiveProcess (debugapi.h) [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://learn.microsoft.com/ru-ru/windows/win32/api/debugapi/nf-debugapi-debugactiveprocess> – Дата доступа 15.10.2023.

[4] Функция DebugActiveProcessStop (debugapi.h) [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://learn.microsoft.com/ru-ru/windows/win32/api/debugapi/nf-debugapi-debugactiveprocessstop> – Дата доступа 15.10.2023.

[5] Создание процессов [Электронный ресурс]. – Электронные данные. – Режим доступа: [https://learn.microsoft.com/ru-ru/windows/win32/procthread/creating-processes](https://learn.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-waitforsingleobject) – Дата доступа 15.10.2023.

[6] Функция TerminateProcess (processthreadsapi.h) [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://learn.microsoft.com/ru-ru/windows/win32/api/processthreadsapi/nf-processthreadsapi-terminateprocess> – Дата доступа 15.10.2023.

ПРИЛОЖЕНИЕ А

**(обязательное)**

**Исходный код программы**

**Файл main.cpp**

#include "resource.h"

#include <windows.h>

#include <tlhelp32.h>

#include <tchar.h>

#include <winternl.h>

#include <strsafe.h>

#include <string>

#include <list>

#include "Helpers.h"

#include "SystemInformation.h"

#pragma comment(lib,"ntdll.lib")

#define IDT\_REFRESH\_TIMER 1001

struct CurrentState {

int CurrentPosition = 0;

DWORD CurrentPID = NULL;

BOOL IsSuspended = 0;

};

static HINSTANCE hAppInstance = NULL;

static HWND hMainWindow = NULL;

static CurrentState CurrentProcess;

static HANDLE hWatchThread = NULL;

static HANDLE hStopWatchingEvent = NULL;

INT\_PTR CALLBACK MainDialogProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam);

// Forward declarations:

BOOL GetProcessList(HWND hDirListView);

BOOL ListProcessModules(DWORD dwPID);

BOOL ListProcessThreads(DWORD dwOwnerPID);

BOOL PopulateProcessListAndStatus(HWND hDirListView);

void printError(const TCHAR\* msg);

void SuspendProcess(DWORD PID);

void ResumeProcess(DWORD PID);

void TerminateProcess(DWORD PID);

void CreateProcessFromPath(std::wstring path);

int WINAPI wWinMain(\_In\_ HINSTANCE hInstance, \_In\_opt\_ HINSTANCE hPrevInstance, \_In\_ PWSTR pCmdLine, \_In\_ int nCmdShow)

{

hAppInstance = hInstance;

hMainWindow = ::CreateDialogParam(hInstance, MAKEINTRESOURCE(IDD\_MAIN\_DIALOG), NULL, MainDialogProc, 0);

if (hMainWindow == NULL)

{

return -1;

}

HICON hIcon = LoadIcon(hInstance, MAKEINTRESOURCE(IDI\_PROCESSMANAGMENT));

SendMessage(hMainWindow, WM\_SETICON, ICON\_SMALL, (LPARAM)hIcon);

::ShowWindow(hMainWindow, nCmdShow);

::UpdateWindow(hMainWindow);

MSG msg = { };

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

{

if (!IsDialogMessage(hMainWindow, &msg))

{

::TranslateMessage(&msg);

::DispatchMessage(&msg);

}

}

return 0;

}

INT\_PTR CALLBACK MainDialogProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)

{

switch (uMsg)

{

case WM\_DESTROY:

::KillTimer(hwnd, IDT\_REFRESH\_TIMER);

::PostQuitMessage(0);

return TRUE;

case WM\_CLOSE:

::DestroyWindow(hwnd);

return TRUE;

case WM\_TIMER:

if (wParam == IDT\_REFRESH\_TIMER)

{

HWND hDirListView = ::GetDlgItem(hwnd, IDC\_PROCESS\_LIST);

ListView\_DeleteAllItems(hDirListView);

PopulateProcessListAndStatus(hDirListView);

}

return TRUE;

case WM\_NOTIFY:

{

switch (LOWORD(wParam))

{

case IDC\_PROCESS\_LIST:

if (((LPNMHDR)lParam)->code == NM\_CLICK)

{

HWND hDirListView = ::GetDlgItem(hwnd, IDC\_PROCESS\_LIST);

int iPos = ListView\_GetNextItem(hDirListView, -1, LVNI\_SELECTED);

LVITEMW lvi = { };

if (iPos != -1) {

ListView\_GetItem(hDirListView, &lvi);

WCHAR PID[MAX\_PATH] = { 0 };

ListView\_GetItemText(hDirListView, iPos, 1, PID, MAX\_PATH);

DWORD pid = \_wtoi(std::wstring(PID).c\_str());

if (pid >= 0) {

CurrentProcess.CurrentPID = pid;

CurrentProcess.CurrentPosition = iPos;

WCHAR watingReason[MAX\_PATH] = { 0 };

ListView\_GetItemText(hDirListView, iPos, 4, watingReason, MAX\_PATH);

if (wcscmp(watingReason, ThreadWaitReasonValueNames[5]) == 0) {

Helpers::SetButtonIcon(hAppInstance, hwnd, IDC\_PLAY\_PAUSE\_BUTTON, IDI\_PLAY\_ICON);

}

else {

Helpers::SetButtonIcon(hAppInstance, hwnd, IDC\_PLAY\_PAUSE\_BUTTON, IDI\_PAUSE\_ICON);

}

}

}

}

return TRUE;

}

break;

}

case WM\_COMMAND:

{

switch (LOWORD(wParam))

{

case IDC\_PLAY\_PAUSE\_BUTTON:

{

HWND hDirListView = ::GetDlgItem(hwnd, IDC\_PROCESS\_LIST);

LVITEMW lvi = { };

ListView\_GetItem(hDirListView, &lvi);

if (CurrentProcess.IsSuspended) {

ResumeProcess(CurrentProcess.CurrentPID);

Helpers::SetButtonIcon(hAppInstance, hwnd, IDC\_PLAY\_PAUSE\_BUTTON, IDI\_PAUSE\_ICON);

}

else {

SuspendProcess(CurrentProcess.CurrentPID);

Helpers::SetButtonIcon(hAppInstance, hwnd, IDC\_PLAY\_PAUSE\_BUTTON, IDI\_PLAY\_ICON);

}

CurrentProcess.IsSuspended = !CurrentProcess.IsSuspended;

PopulateProcessListAndStatus(::GetDlgItem(hwnd, IDC\_PROCESS\_LIST));

}

break;

case IDC\_STOP\_BUTTON:

{

TerminateProcess(CurrentProcess.CurrentPID);

CurrentProcess = {};

PopulateProcessListAndStatus(::GetDlgItem(hwnd, IDC\_PROCESS\_LIST));

}

break;

case IDC\_ADD\_BUTTON:

{

std::wstring filePath = Helpers::BrowseForFile(hAppInstance, hwnd, TEXT("Open exe"),

TEXT("Exe Files\0\*.exe\0"), TEXT("Exe files"), false);

CreateProcessFromPath(filePath);

PopulateProcessListAndStatus(::GetDlgItem(hwnd, IDC\_PROCESS\_LIST));

}

break;

case IDC\_REFRESH\_BUTTON:

{

PopulateProcessListAndStatus(::GetDlgItem(hwnd, IDC\_PROCESS\_LIST));

// GetProcessList(::GetDlgItem(hwnd, IDC\_PROCESS\_LIST));

}

break;

}

}

break;

case WM\_INITDIALOG:

{

Helpers::SetButtonIcon(hAppInstance, hwnd, IDC\_PLAY\_PAUSE\_BUTTON, IDI\_PAUSE\_ICON);

Helpers::SetButtonIcon(hAppInstance, hwnd, IDC\_ADD\_BUTTON, IDI\_ADD\_ICON);

Helpers::SetButtonIcon(hAppInstance, hwnd, IDC\_STOP\_BUTTON, IDI\_STOP\_ICON);

Helpers::SetButtonIcon(hAppInstance, hwnd, IDC\_REFRESH\_BUTTON, IDI\_REFRESH\_ICON);

LVCOLUMNW lvc;

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

lvc.cx = 120;

lvc.fmt = LVCFMT\_LEFT;

static WCHAR szDirListColumnString[5][64] = { L"Name", L"PID", L"Thread count", L"State", L"Wait reason"};

HWND hDirListView = ::GetDlgItem(hwnd, IDC\_PROCESS\_LIST);

ListView\_SetExtendedListViewStyle(hDirListView, LVS\_EX\_AUTOSIZECOLUMNS | LVS\_EX\_FULLROWSELECT);

for (auto column = 0; column < 5; ++column)

{

lvc.pszText = szDirListColumnString[column];

if (ListView\_InsertColumn(hDirListView, column, &lvc) == -1)

{

return FALSE;

}

}

PopulateProcessListAndStatus(hDirListView);

//::SetTimer(hwnd, IDT\_REFRESH\_TIMER, 5000, NULL);

}

break;

}

return FALSE;

}

BOOL PopulateProcessListAndStatus(HWND hDirListView)

{

ListView\_DeleteAllItems(hDirListView);

NTSTATUS status;

PVOID buffer;

PSYSTEM\_PROCESS\_INFO spi;

buffer = VirtualAlloc(NULL, 1024 \* 1024, MEM\_COMMIT | MEM\_RESERVE, PAGE\_READWRITE);

if (!buffer)

{

printf("\nError: Unable to allocate memory for process list (%d)\n", GetLastError());

return -1;

}

spi = (PSYSTEM\_PROCESS\_INFO)buffer;

if (!NT\_SUCCESS(status = NtQuerySystemInformation(SystemProcessInformation, spi, 1024 \* 1024, NULL)))

{

printf("\nError: Unable to query process list (%#x)\n", status);

VirtualFree(buffer, 0, MEM\_RELEASE);

return -1;

}

LVITEMW lvi = { };

lvi.mask = LVIF\_PARAM | LVIF\_IMAGE | LVIF\_TEXT;

lvi.iSubItem = 0;

WCHAR pszText[] = L"Idle";

lvi.pszText = pszText;

while (spi->pi.NextEntryOffset)

{

if (spi->pi.ImageName.Buffer != NULL)

lvi.pszText = spi->pi.ImageName.Buffer;

ListView\_InsertItem(hDirListView, &lvi);

WCHAR PIDString[256];

wsprintfW(PIDString, L"%d", spi->pi.UniqueProcessId);

ListView\_SetItemText(hDirListView, lvi.iItem, 1, PIDString);

WCHAR ThreandString[256];

wsprintfW(ThreandString, L"%d", spi->pi.NumberOfThreads);

ListView\_SetItemText(hDirListView, lvi.iItem, 2, ThreandString);

if (spi->ti->ThreadState < 8) {

ListView\_SetItemText(hDirListView, lvi.iItem, 3, (LPWSTR)ThreadStateValueNames[spi->ti->ThreadState]);

}

else {

WCHAR StateString[256];

wsprintfW(StateString, L"%d", spi->ti->ThreadState);

ListView\_SetItemText(hDirListView, lvi.iItem, 3, StateString);

}

if (spi->ti->WaitReason < 38) {

ListView\_SetItemText(hDirListView, lvi.iItem, 4, (LPWSTR)ThreadWaitReasonValueNames[spi->ti->WaitReason]);

}

else {

WCHAR ReasonString[256];

wsprintfW(ReasonString, L"%d", spi->ti->WaitReason);

ListView\_SetItemText(hDirListView, lvi.iItem, 4, ReasonString);

}

++lvi.iItem;

spi = (PSYSTEM\_PROCESS\_INFO)((LPBYTE)spi + spi->pi.NextEntryOffset);

}

VirtualFree(buffer, 0, MEM\_RELEASE);

return 0;

}

void SuspendProcess(DWORD PID)

{

::DebugActiveProcess(PID);

}

void ResumeProcess(DWORD PID) {

::DebugActiveProcessStop(PID);

}

void TerminateProcess(DWORD PID) {

const auto explorer = OpenProcess(PROCESS\_TERMINATE, false, PID);

TerminateProcess(explorer, 0);

CloseHandle(explorer);

}

void CreateProcessFromPath(std::wstring path) {

STARTUPINFO info = { sizeof(info) };

PROCESS\_INFORMATION processInfo;

if (CreateProcess(path.data(), NULL, NULL, NULL, TRUE, 0, NULL, NULL, &info, &processInfo)) {

CloseHandle(processInfo.hProcess);

CloseHandle(processInfo.hThread);

}

}

**Файл Helpers.h**

#pragma once

#include <windows.h>

#include <string>

namespace Helpers

{

bool SetButtonIcon(HINSTANCE hInstance, HWND hDlg, int buttonID, int iconID);

std::wstring BrowseForFile(HINSTANCE hInstance, HWND hParentWindow, LPCWSTR title,

LPCWSTR filter, LPCWSTR defExtention, bool existingOnly);

std::wstring BrowseFolder(HWND hParentWindow, LPCWSTR title, LPCWSTR startFromPath);

void SetControlEnabled(HWND hDlg, int controlID, bool enabled);

std::wstring FileTimeToString(FILETIME ftime);

}

**Файл Helpers.cpp**

#include "Helpers.h"

#include <codecvt>

#include <shlobj.h>

#include <sstream>

#include <iomanip>

namespace Helpers

{

bool SetButtonIcon(HINSTANCE hInstance, HWND hDlg, int buttonID, int iconID)

{

HICON hIcon = static\_cast<HICON>(::LoadImage(hInstance, MAKEINTRESOURCE(iconID), IMAGE\_ICON,

16, 16, LR\_DEFAULTCOLOR));

if (hIcon == NULL)

{

return FALSE;

}

::SendDlgItemMessage(hDlg, buttonID, BM\_SETIMAGE, (WPARAM)IMAGE\_ICON, (LPARAM)hIcon);

return TRUE;

}

std::wstring BrowseForFile(HINSTANCE hInstance, HWND hParentWindow, LPCTSTR title,

LPCTSTR filter, LPCTSTR defExtention, bool existingOnly)

{

static const int MaxFilePath = 1024;

OPENFILENAME of = {};

of.lStructSize = sizeof(OPENFILENAME);

of.hwndOwner = hParentWindow;

of.hInstance = hInstance;

of.lpstrFilter = filter;

of.lpstrDefExt = defExtention;

of.lpstrTitle = title;

of.Flags = OFN\_EXPLORER;

if (existingOnly)

{

of.Flags |= OFN\_PATHMUSTEXIST | OFN\_FILEMUSTEXIST;

}

TCHAR selectedFilePath[MaxFilePath] = { 0 };

of.lpstrFile = selectedFilePath;

of.nMaxFile = MaxFilePath;

BOOL ok = ::GetOpenFileName(&of);

if (ok)

{

return selectedFilePath;

}

return std::wstring();

}

static int CALLBACK BrowseCallbackProc(HWND hwnd, UINT uMsg, LPARAM lParam, LPARAM lpData)

{

if (uMsg == BFFM\_INITIALIZED)

{

SendMessage(hwnd, BFFM\_SETSELECTION, TRUE, lpData);

}

return 0;

}

std::wstring BrowseFolder(HWND hParentWindow, LPCWSTR title, LPCWSTR startFromPath)

{

WCHAR path[MAX\_PATH];

BROWSEINFO bi = { 0 };

bi.hwndOwner = hParentWindow;

bi.lpszTitle = title;

bi.ulFlags = BIF\_RETURNONLYFSDIRS | BIF\_NEWDIALOGSTYLE;

bi.lpfn = BrowseCallbackProc;

bi.lParam = (LPARAM)startFromPath;

LPITEMIDLIST pidl = SHBrowseForFolder(&bi);

if (pidl != 0)

{

SHGetPathFromIDListW(pidl, path);

IMalloc\* imalloc = 0;

if (SUCCEEDED(SHGetMalloc(&imalloc)))

{

imalloc->Free(pidl);

imalloc->Release();

}

return std::wstring(path);

}

return std::wstring();

}

void SetControlEnabled(HWND hDlg, int controlID, bool enabled)

{

:: EnableWindow(GetDlgItem(hDlg, controlID), enabled);

}

std::wstring FileTimeToString(FILETIME ftime)

{

SYSTEMTIME utc;

::FileTimeToSystemTime(std::addressof(ftime), std::addressof(utc));

std::wostringstream os;

os << std::setfill<wchar\_t>('0') << std::setw(4) << utc.wYear << '-' << std::setw(2) << utc.wMonth

<< '-' << std::setw(2) << utc.wDay << ' ' << std::setw(2) << utc.wHour

<< ':' << std::setw(2) << utc.wMinute << ':' << std::setw(2) << utc.wSecond;

return os.str();

}

}

**Файл SystemInformation.h**

#pragma once

#include <winternl.h>

typedef struct \_SYSTEM\_PROCESS\_INFO

{

SYSTEM\_PROCESS\_INFORMATION pi;

SYSTEM\_THREAD\_INFORMATION ti[1];

} SYSTEM\_PROCESS\_INFO, \* PSYSTEM\_PROCESS\_INFO;

typedef enum

{

ThreadStateInitialized,

ThreadStateReady,

ThreadStateRunning,

ThreadStateStandby,

ThreadStateTerminated,

ThreadStateWaiting,

ThreadStateTransition,

ThreadStateDeferredReady

} THREAD\_STATE;

const WCHAR\* ThreadStateValueNames[] =

{

L"Initialized",

L"Ready",

L"Running",

L"Standby",

L"Terminated",

L"Waiting",

L"Transition",

L"DeferredReady"

};

typedef enum

{

ThreadWaitReasonExecutive,

ThreadWaitReasonFreePage,

ThreadWaitReasonPageIn,

ThreadWaitReasonPoolAllocation,

ThreadWaitReasonDelayExecution,

ThreadWaitReasonSuspended,

ThreadWaitReasonUserRequest,

ThreadWaitReasonWrExecutive,

ThreadWaitReasonWrFreePage,

ThreadWaitReasonWrPageIn,

ThreadWaitReasonWrPoolAllocation,

ThreadWaitReasonWrDelayExecution,

ThreadWaitReasonWrSuspended,

ThreadWaitReasonWrUserRequest,

ThreadWaitReasonWrEventPair,

ThreadWaitReasonWrQueue,

ThreadWaitReasonWrLpcReceive,

ThreadWaitReasonWrLpcReply,

ThreadWaitReasonWrVirtualMemory,

ThreadWaitReasonWrPageOut,

ThreadWaitReasonWrRendezvous,

ThreadWaitReasonWrKeyedEvent,

ThreadWaitReasonWrTerminated,

ThreadWaitReasonWrProcessInSwap,

ThreadWaitReasonWrCpuRateControl,

ThreadWaitReasonWrCalloutStack,

ThreadWaitReasonWrKernel,

ThreadWaitReasonWrResource,

ThreadWaitReasonWrPushLock,

ThreadWaitReasonWrMutex,

ThreadWaitReasonWrQuantumEnd,

ThreadWaitReasonWrDispatchInt,

ThreadWaitReasonWrPreempted,

ThreadWaitReasonWrYieldExecution,

ThreadWaitReasonWrFastMutex,

ThreadWaitReasonWrGuardedMutex,

ThreadWaitReasonWrRundown,

ThreadWaitReasonMaximumWaitReason

} THREAD\_WAIT\_REASON;

const WCHAR\* ThreadWaitReasonValueNames[] =

{

L"Executive",

L"FreePage",

L"PageIn",

L"PoolAllocation",

L"DelayExecution",

L"Suspended",

L"UserRequest",

L"WrExecutive ",

L"WrFreePage",

L"WrPageIn",

L"WrPoolAllocation",

L"WrDelayExecution",

L"WrSuspended",

L"WrUserRequest",

L"WrEventPair",

L"WrQueue",

L"WrLpcReceive",

L"WrLpcReply",

L"WrVirtualMemory",

L"WrPageOut",

L"WrRendezvous",

L"WrKeyedEvent",

L"WrTerminated",

L"WrProcessInSwap",

L"WrCpuRateControl",

L"WrCalloutStack",

L"WrKernel",

L"WrResource",

L"WrPushLock",

L"WrMutex",

L"WrQuantumEnd",

L"WrDispatchInt",

L"WrPreempted",

L"WrYieldExecution",

L"WrFastMutex",

L"WrGuardedMutex",

L"WrRundown",

L"MaximumWaitReason"

};