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.**

Студент В. А. Сидорко

Преподаватель Н. Ю. Гриценко

Минск 2023

СОДЕРЖАНИЕ

[1 Цель работы 3](#_Toc146752068)

[2 Теоретические сведения 3](#_Toc146752069)

[3 Результат выполнения 6](#_Toc146752070)

[Заключение 7](#_Toc146752071)

[Список использованных источников 8](#_Toc146752072)

[Приложение А (обязательное) Листинг кода 9](#_Toc146752073)

1. **ЦЕЛЬ РАБОТЫ**

Изучить принципы управления процессами и потоками Windows. Научиться порождать, завершать, изменять приоритет потоков и процессов. функции API подсистемы памяти Win32, организацию и контроль асинхронных операций ввода-вывода, отображение файлов в память. Для достижения цели будет создано оконное приложение для отслеживания и управления процессами в системе, позволяющее создавать, приостанавливать, возобновлять и завершать процессы.

**2 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

Возможность управлять процессами и потоками является одной из важных функций *Win32 API*.

Каждый процесс предоставляет ресурсы, необходимые для выполнения программы. Процесс имеет виртуальное адресное пространство, исполняемый код, открытые дескрипторы для системных объектов, контекст безопасности, уникальный идентификатор процесса, переменные среды, класс приоритета, минимальный и максимальный размер рабочего набора и по крайней мере один поток выполнения. Каждый процесс запускается в одном потоке, часто называемого основным потоком, но может создавать дополнительные потоки из любого из его потоков.

Поток — это сущность в процессе, которую можно запланировать для выполнения. Все потоки процесса совместно используют его виртуальное адресное пространство и системные ресурсы. Кроме того, каждый поток поддерживает обработчики исключений, приоритет планирования, локальное хранилище потока, уникальный идентификатор потока и набор структур, которые система будет использовать для сохранения контекста потока до его планирования. Контекст потока включает набор, стек ядра, блок среды потока и стек пользователя в адресном пространстве процесса потока.

Для получения информации о процессах в данной работе была использована библиотека *Psapi.h*, предоставляющая следующие функции:

1 *EnumProcesses* – позволяет получить список идентификаторов всех запущенных процессов, заполняя ими массив значений *DWORD*.

2 *OpenProcess* – в качестве параметра принимает идентификатор процесса и возвращает дескриптор соответствующего процесса. По окончании работы с дескриптором процесса вызывается функция *CloseHandle*.

3 *EnumProcessModules* – используется для получения дескрипторов всех модулей, указанного процесса. Каждый процесс состоит из одного или нескольких модулей. Модуль – это исполняемый файл или библиотека *DLL*.

4 *GetProcessMemoryInfo* – в качестве входных данных принимает дескриптор процесса и заполняет структуру *PROCESS\_MEMORY\_COUNTERS* сведениями о статистике памяти для процесса. 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.

Для управления процессами и потоками была использована библиотека *processthreadsapi.h* и предоставляемые ей следующие функции:

1 *CreateProcess* – создает новый процесс и его основной поток.

2 *TerminateProcess* – завершает указанный процесс и все его потоки.

3 *SuspendThread* – приостанавливает указанный поток, увеличивает счетчик приостановок на единицу. Чтобы приостановить процесс, необходимо приостановить выполнение всех его потоков.

4 *ResumeThread* – уменьшает счетчик приостановок на единицу; если счетчик приостановок уменьшается до нуля, выполнение потока возобновляется. Чтобы возобновить процесс, необходимо возобновить выполнение всех его потоков.

1. **РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ**

В результате выполнения лабораторной работы было создано приложение для отслеживания и управления процессами в системе, позволяющее создавать, приостанавливать, возобновлять и завершать процессы (рисунок 1).

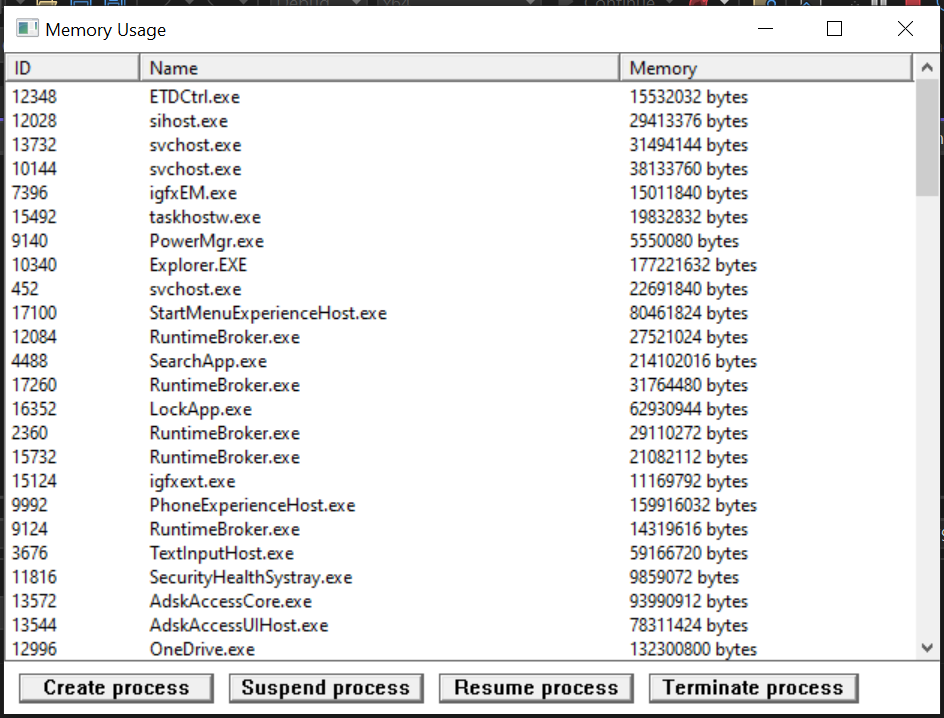


Рисунок 1 – Окно приложения

В окне приложения отражены идентификатор процесса (*ID*), его название (*Name*), и объем используемой памяти в байтах (*Memory*). Внизу окна расположены кнопки «*Create Process*», «*Suspend* *Process*», «*Resume Process*» и «*Terminate Process*», позволяющие создавать, приостанавливать, возобновлять и завершать процессы соответственно.

# ЗАКЛЮЧЕНИЕ

В ходе лабораторной работы были изучены и применены на практике знания о функциях управления процессами и потоками. В результате проделанной работы было создано оконное приложение для отслеживания и управления процессами в системе, позволяющее создавать, приостанавливать, возобновлять и завершать процессы.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Функции процессов и потоков [Электронный ресурс]. – Режим доступа : https://learn.microsoft.com/ru-ru/windows/win32/procthread/process-and-thread-functions.
2. Щупак, Ю.А. Win32 API. Разработка приложений для Windows /Ю.А. Щупак. – СПб : Питер, 2008. – 592 с.
3. Psapi, документация Microsoft [Электронный ресурс]. – Режим доступа : https://learn.microsoft.com/en-us/windows/win32/api/psapi/.
4. О процессах и потоках [Электронный ресурс]. – Режим доступа : https://learn.microsoft.com/ru-ru/windows/win32/procthread/about-processes-and-threads.

# ПРИЛОЖЕНИЕ А (обязательное) Листинг кода

Листинг 1 – Файл main.cpp

#pragma comment(lib, "comctl32.lib")

#include <Windows.h>

#include <psapi.h>

#include "commctrl.h"

#include <TlHelp32.h>

#define IDT\_TIMER1 1

WNDCLASS NewWindowClass(HINSTANCE hInst, LPCWSTR name, WNDPROC procedure);

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

void UpdateProcessesInfo(HWND hWnd);

void CreateNewProcess(HWND hWnd);

void SuspendSelectedProcess(HWND hWnd);

void ResumeSelectedProcess(HWND hWnd);

void TerminateSelectedProcess(HWND hWnd);

void AddWidgets(HWND hWnd);

void CALLBACK TimeProc(UINT uID, UINT uMsg, DWORD dwUser, DWORD dw1, DWORD dw2, HWND hWnd);

HINSTANCE hInstance;

HWND hWndListView;

HWND hWndCreateProcessButton;

HWND hWndSuspendProcessButton;

HWND hWndResumeProcessButton;

HWND hWndTerminateProcessButton;

char szFileName[MAX\_PATH];

int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd) {

hInstance = hInst;

WNDCLASS MainClass = NewWindowClass(hInst, L"Main Window Class", WndProc);

if (!RegisterClass(&MainClass)) { return -1; }

MSG MainMessage = { 0 };

CreateWindow(L"Main Window Class", L"Memory Usage",

WS\_OVERLAPPEDWINDOW | WS\_VISIBLE, 100, 50, 640, 480, NULL, NULL, hInst, NULL);

while (GetMessage(&MainMessage, NULL, NULL, NULL)) {

TranslateMessage(&MainMessage);

DispatchMessage(&MainMessage);

}

}

WNDCLASS NewWindowClass(HINSTANCE hInst, LPCWSTR name, WNDPROC procedure) {

WNDCLASS windowClass = {};

windowClass.hInstance = hInst;

windowClass.lpszClassName = name;

windowClass.lpfnWndProc = procedure;

return windowClass;

}

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

{

switch (uMsg)

{

case WM\_COMMAND:

if (lParam == (LPARAM)hWndCreateProcessButton) {

CreateNewProcess(hWnd);

}

else if (lParam == (LPARAM)hWndSuspendProcessButton) {

SuspendSelectedProcess(hWnd);

}

else if (lParam == (LPARAM)hWndResumeProcessButton) {

ResumeSelectedProcess(hWnd);

}

else if (lParam == (LPARAM)hWndTerminateProcessButton) {

TerminateSelectedProcess(hWnd);

}

break;

case WM\_CREATE:

SetTimer(hWnd, IDT\_TIMER1, 1000, (TIMERPROC)UpdateProcessesInfo);

AddWidgets(hWnd);

UpdateProcessesInfo(hWnd);

break;

case WM\_CLOSE:

DestroyWindow(hWnd);

break;

case WM\_DESTROY:

KillTimer(hWnd, IDT\_TIMER1);

PostQuitMessage(0);

return 0;

}

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

}

void AddWidgets(HWND hWnd)

{

InitCommonControls();

hWndListView = CreateWindow(WC\_LISTVIEW, L"",

WS\_VISIBLE | WS\_BORDER | WS\_CHILD | LVS\_REPORT,

0, 0, 625, 406, hWnd, NULL, hInstance, 0);

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

LVCOLUMN lvcolId;

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

lvcolId.pszText = LPWCH(L"ID");

lvcolId.cx = 90;

ListView\_InsertColumn(hWndListView, 0, &lvcolId);

LVCOLUMN lvcolName;

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

lvcolName.pszText = LPWCH(L"Name");

ListView\_InsertColumn(hWndListView, 1, &lvcolName);

ListView\_SetColumnWidth(hWndListView, 1, 320);

LVCOLUMN lvcolMemory;

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

lvcolMemory.pszText = LPWCH(L"Memory");

ListView\_InsertColumn(hWndListView, 2, &lvcolMemory);

ListView\_SetColumnWidth(hWndListView, 2, 195);

hWndCreateProcessButton = CreateWindow(L"BUTTON", L"Create process",

WS\_TABSTOP | WS\_VISIBLE | WS\_CHILD | BS\_DEFPUSHBUTTON,

10, 414, 130, 20, hWnd, NULL, hInstance, NULL);

hWndSuspendProcessButton = CreateWindow(L"BUTTON", L"Suspend process",

WS\_TABSTOP | WS\_VISIBLE | WS\_CHILD | BS\_DEFPUSHBUTTON,

150, 414, 130, 20, hWnd, NULL, hInstance, NULL);

hWndResumeProcessButton = CreateWindow(L"BUTTON", L"Resume process",

WS\_TABSTOP | WS\_VISIBLE | WS\_CHILD | BS\_DEFPUSHBUTTON,

290, 414, 130, 20, hWnd, NULL, hInstance, NULL);

hWndTerminateProcessButton = CreateWindow(L"BUTTON", L"Terminate process",

WS\_TABSTOP | WS\_VISIBLE | WS\_CHILD | BS\_DEFPUSHBUTTON,

430, 414, 140, 20, hWnd, NULL, hInstance, NULL);

}

void UpdateProcessesInfo(HWND hWnd) {

// array of process ids

DWORD aProcesses[1024];

// number of bytes returned in process ids array

DWORD cbNeeded;

// number of process ids returned

DWORD cProcesses;

if (!EnumProcesses(aProcesses, sizeof(aProcesses), &cbNeeded)) {

MessageBox(hWnd, L"Failed to get process ids!", L"Error", MB\_OK);

return;

}

cProcesses = cbNeeded / sizeof(DWORD);

UINT count = 0;

DWORD dwTotalitems = ListView\_GetItemCount(hWndListView);

for (UINT i = 0; i < cProcesses; i++) {

if (aProcesses[i] != 0) {

TCHAR szProcessName[MAX\_PATH] = L"<unknown>";

HANDLE hProcess = OpenProcess(PROCESS\_QUERY\_INFORMATION | PROCESS\_VM\_READ,

FALSE, aProcesses[i]);

PROCESS\_MEMORY\_COUNTERS pmc;

WCHAR szInfo[MAX\_PATH];

WCHAR szWorkingSet[100];

if (hProcess != NULL) {

HMODULE hModule;

DWORD cbNeeded;

if (EnumProcessModules(hProcess, &hModule, sizeof(hModule), &cbNeeded)) {

GetModuleBaseName(hProcess, hModule, szProcessName,

sizeof(szProcessName) / sizeof(TCHAR));

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

wsprintf(szWorkingSet, TEXT("%lu bytes"), pmc.WorkingSetSize);

}

CloseHandle(hProcess);

if (count >= dwTotalitems) {

LVITEM lvItem;

lvItem.mask = LVIF\_TEXT;

lvItem.iItem = count;

lvItem.iSubItem = 0;

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

ListView\_InsertItem(hWndListView, &lvItem);

}

wchar\_t buffer[MAX\_PATH];

wsprintf(szInfo, TEXT("%lu"), aProcesses[i]);

ListView\_SetItemText(hWndListView, count, 0, szInfo);

wsprintf(szInfo, TEXT("%s"), szProcessName);

ListView\_SetItemText(hWndListView, count, 1, szInfo);

ListView\_SetItemText(hWndListView, count, 2, szWorkingSet);

count++;

}

}

}

if (count < dwTotalitems) {

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

ListView\_DeleteItem(hWndListView, i);

}

}

}

void CreateNewProcess(HWND hWnd) {

OPENFILENAME ofn;

ZeroMemory(&ofn, sizeof(ofn));

ofn.lStructSize = sizeof(ofn);

ofn.hwndOwner = hWnd;

ofn.lpstrFilter = L"Executable Files\0\*.exe\0";

ofn.lpstrFile = (LPWSTR)szFileName;

ofn.nMaxFile = MAX\_PATH;

ofn.Flags = OFN\_EXPLORER | OFN\_PATHMUSTEXIST | OFN\_HIDEREADONLY;

ofn.lpstrDefExt = L"exe";

if (GetOpenFileName(&ofn)) {

STARTUPINFO si;

PROCESS\_INFORMATION pi;

ZeroMemory(&si, sizeof(si));

si.cb = sizeof(si);

ZeroMemory(&pi, sizeof(pi));

if (CreateProcess(ofn.lpstrFile, NULL,

NULL, NULL, FALSE, NULL, NULL, NULL, &si, &pi)) {

CloseHandle(pi.hProcess);

CloseHandle(pi.hThread);

}

else {

MessageBox(hWnd, L"Failed to create new process.", L"Error",

MB\_OK | MB\_ICONERROR);

}

}

}

void SuspendSelectedProcess(HWND hWnd) {

UINT iItem = ListView\_GetNextItem(hWndListView, -1, LVNI\_SELECTED);

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

ListView\_GetItemText(hWndListView, iItem, 0, (LPWSTR)text, sizeof(text));

if (iItem < 0) {

MessageBox(hWnd, L"Please select a process to suspend.",

L"No process selected", MB\_OK | MB\_ICONERROR);

return;

}

DWORD dwProcessId = wcstoul(text, NULL, 10);

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

THREADENTRY32 te32;

te32.dwSize = sizeof(THREADENTRY32);

if (Thread32First(hTreadSnapshot, &te32)) {

do {

if (te32.th32OwnerProcessID == dwProcessId) {

// suspend thread

HANDLE hThread = OpenThread(THREAD\_SUSPEND\_RESUME,

FALSE, te32.th32ThreadID);

if (hThread == NULL) {

MessageBox(hWnd, L"Failed to suspend process.",

L"Error", MB\_OK | MB\_ICONERROR);

break;

}

SuspendThread(hThread);

CloseHandle(hThread);

}

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

}

else {

MessageBox(hWnd, L"Failed to suspend process.",

L"Error", MB\_OK | MB\_ICONERROR);

}

CloseHandle(hTreadSnapshot);

}

void ResumeSelectedProcess(HWND hWnd) {

UINT iItem = ListView\_GetNextItem(hWndListView, -1, LVNI\_SELECTED);

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

ListView\_GetItemText(hWndListView, iItem, 0, (LPWSTR)text, sizeof(text));

if (iItem < 0) {

MessageBox(hWnd, L"Please select a process to resume.",

L"No process selected", MB\_OK | MB\_ICONERROR);

return;

}

DWORD dwProcessId = wcstoul(text, NULL, 10);

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

THREADENTRY32 te32;

te32.dwSize = sizeof(THREADENTRY32);

if (Thread32First(hTreadSnapshot, &te32)) {

do {

if (te32.th32OwnerProcessID == dwProcessId) {

HANDLE hThread = OpenThread(THREAD\_SUSPEND\_RESUME,

FALSE, te32.th32ThreadID);

if (hThread == NULL) {

MessageBox(hWnd, L"Failed to resume process.",

L"Error", MB\_OK | MB\_ICONERROR);

break;

}

ResumeThread(hThread);

CloseHandle(hThread);

}

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

}

else {

MessageBox(hWnd, L"Failed to resume process.",

L"Error", MB\_OK | MB\_ICONERROR);

}

CloseHandle(hTreadSnapshot);

}

void TerminateSelectedProcess(HWND hWnd) {

UINT iItem = ListView\_GetNextItem(hWndListView, -1, LVNI\_SELECTED);

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

ListView\_GetItemText(hWndListView, iItem, 0, (LPWSTR)text, sizeof(text));

if (iItem < 0) {

MessageBox(hWnd, L"Please select a process to terminate.",

L"Process not selected", MB\_OK | MB\_ICONERROR);

return;

}

DWORD processId = wcstoul(text, NULL, 10);

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

if (hProcess == NULL) {

MessageBox(hWnd, L"Failed to open process.", L"Error", MB\_OK | MB\_ICONERROR);

}

else {

TerminateProcess(hProcess, -1);

WaitForSingleObject(hProcess, NULL);

CloseHandle(hProcess);

}

}

void CALLBACK TimeProc(UINT uID, UINT uMsg, DWORD dwUser, DWORD dw1, DWORD dw2, HWND hWnd) {

UpdateProcessesInfo(hWnd);

}