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. 153503

Звягинцева В.А.

Проверил:

Гриценко Н. Ю.

Минск 2023

СОДЕРЖАНИЕ

1 Постановка задачи3

2 Краткие теоретические сведения4

2.1 Процессы4

2.2 Потоки4

3 Результаты выполнения лабораторной работы6

Вывод9

Список использованных источников10

Приложение А11

**1 ПОСТАНОВКА ЗАДАЧИ**

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

**2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

**2.1 Процессы**

Процесс в операционной системе *Windows* представляет собой выполняющуюся программу. Каждый процесс имеет своё собственное виртуальное адресное пространство, наборы ресурсов и инструкций. Операционная система управляет множеством процессов, обеспечивая их изоляцию и параллельное выполнение.

Для создания нового процесса в *Win*32 используется функция *CreateProcess*. Она принимает параметры, такие как путь к исполняемому файлу, командная строка, атрибуты безопасности и др. [2]

Процесс может быть завершён функцией *ExitProcess*, которая принимает код завершения. Также другой процесс может завершить процесс с использованием функции *TerminateProcess*. [2]

Процесс также может быть приостановлен и возобновлён. Для этого используются функции *SuspendProcess* и *ResumeProcess* соответсвенно. [2]

Каждый процесс имеет свой приоритет выполнения, который определяет, насколько процесс предпочтителен для выполнения относительно других процессов. Этот приоритет можно изменить с использованием функции *SetPriorityClass*. [2]

**2.2 Потоки**

Поток представляет собой наименьшую единицу выполнения внутри процесса. Процесс может иметь несколько потоков, каждый из которых выполняется параллельно.

Функция *CreateThread* используется для создания нового потока. Она принимает указатель на функцию, которая будет выполнена в созданном потоке. [2]

Поток завершается, когда функция, переданная при его создании, завершает своё выполнение. Также можно использовать функцию *ExitThread* для явного завершения потока. [2]

Каждый поток имеет свой приоритет выполнения внутри процесса. Приоритет потока можно изменить с использованием функции *SetThreadPriority*. [2]

В многозадачных сценариях потоки могут взаимодействовать между собой. Для предотвращения конфликтов и обеспечения синхронизации используются объекты синхронизации, такие как мьютексы и семафоры.

Мьютекс (*Mutex*) представляет собой механизм синхронизации, который используется для управления доступом к ресурсам, разделяемым несколькими потоками. Он гарантирует, что только один поток имеет доступ к ресурсу в определенный момент времени. Мьютексы широко используются для синхронизации доступа к общим ресурсам внутри процесса. Например, при доступе к общим структурам данных, файлам или другим критическим ресурсам. [3]

Семафор — это абстрактный объект, который используется для контроля доступа к общему ресурсу в многозадачной среде. В отличие от мьютекса, семафор может позволить нескольким потокам одновременно получать доступ к ресурсу. Он эффективен при управлении ресурсами, которые могут обслуживаться несколькими потоками параллельно. Например, при ограничении числа одновременно работающих потоков в системе.

*Windows* поддерживает как предпосылочную (*preemptive*), так и кооперативную многозадачность. Предпосылочная многозадачность означает, что операционная система может вытеснить текущий поток в любой момент и передать управление другому. [3]

**3 РЕЗУЛЬТАТЫ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ**

В ходе выполнения лабораторной работы было разработано приложение мониторинга ресурсов процессов операционной системы.

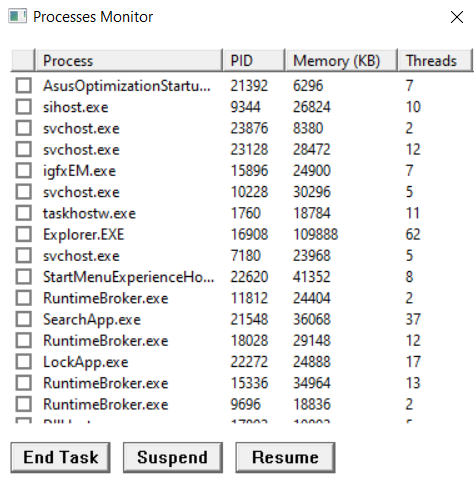


Рисунок 1. – Результат работы программы

В приложении пользователь может посмотреть активные процессы, узнать их идентификаторы, а также сколько рабочей памяти потребляет процесс. Также выводится информация о числе активных потоков.

Для того чтобы освободить занимаемую процессом память пользователь может снять задачу, для этого нужно выбрать нужный процесс и нажать кнопку *End Task*. После этого процесс будет завершён, а память освободиться.

На рисунке 2 представлена возможность выбора процесса для завершения.

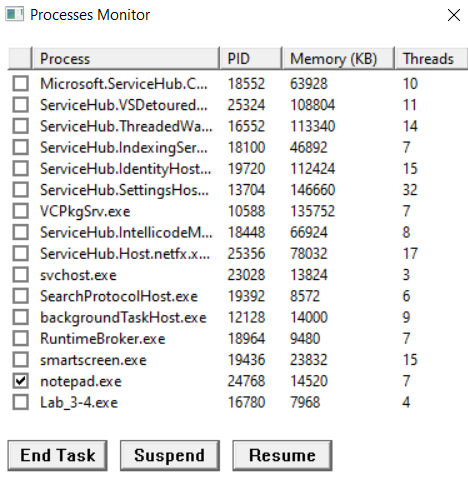


Рисунок 2. – Выбор процесса который нужно завершить

На рисунке 3 представлен результат завершения процесса.

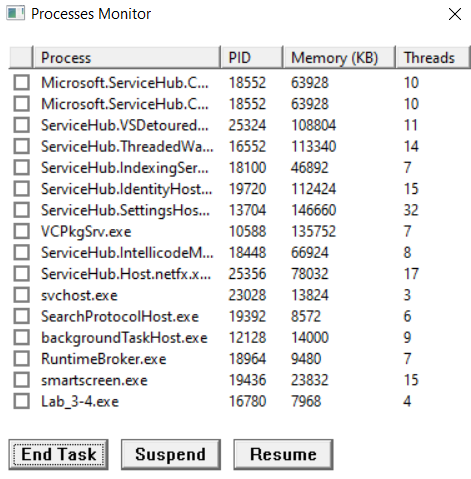


Рисунок 3. – Процесс был завершён

Для того чтобы приостановить процесс пользователь может нажать кнопку *Suspend*.

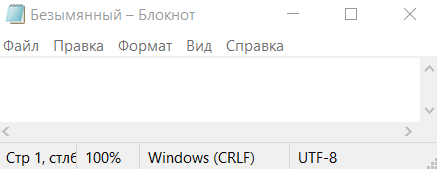


Рисунок 4. – Процесс notepad.exe приостановлен

На рисунке 4 мы можем наблюдать, что блокнот неактивен, так как его процесс приостановлен,

Для возобновления процесса нужно нажать кнопку *Resume.*

**ВЫВОД**

В результате выполнения лабораторной работы были изучены и освоены способы управления процессами и потоками (*Windows*). порождение, завершение, изменение приоритетов процессов и потоков, исследование эффективности на примере приложения позволяющего останавливать, возобновлять и завершать процессы.

**СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ**

1. Основы программирования для *Win32 API* [Электронный ресурс]. – Режим доступа: https://dims.karelia.ru/win32/.
2. Процессы и потоки [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/procthread/processes-and-threads.
3. Синхронизация процессов [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/sync/interprocess-synchronization.

**ПРИЛОЖЕНИЕ А**

**(обязательное)**

**Листинг кода**

**main.h**

#pragma once

#include <Windows.h>

#include <tchar.h>

#include <iostream>

#include <vector>

#include <string>

#include <future>

#include <thread>

#include <psapi.h>

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

#include <CommCtrl.h>

#pragma comment(lib, "Comctl32.lib")

#include <TlHelp32.h>

#define ID\_LISTVIEW 1000

#define ID\_BUTTON\_ENDTASK 1001

#define ID\_TIMER\_UPDATE 1002

#define ID\_BUTTON\_SUSPEND 1003

#define ID\_BUTTON\_RESUME 1004

**main.cpp**

#include "main.h"

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

void UpdateProcessList(HWND);

void EnumThreadsInProcess(DWORD, DWORD&);

void SuspendProcess(DWORD);

void ResumeProcess(DWORD);

HWND hwndListView = nullptr;

int index = -1;

int suspendIndex = -1;

int resumeIndex = -1;

std::future<void> asyncTask;

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

WNDCLASSEX wcex = { sizeof(WNDCLASSEX) };

wcex.lpfnWndProc = WndProc;

wcex.hInstance = hInstance;

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

wcex.lpszClassName = L"Processes Monitor";

RegisterClassEx(&wcex);

HWND hWnd = CreateWindow(L"Processes Monitor", L"Processes Monitor", WS\_OVERLAPPED | WS\_CAPTION | WS\_SYSMENU,

CW\_USEDEFAULT, CW\_USEDEFAULT, 400, 400, nullptr, nullptr, hInstance, nullptr);

if (!hWnd) {

return false;

}

ShowWindow(hWnd, nCmdShow);

UpdateWindow(hWnd);

MSG msg;

while (GetMessage(&msg, nullptr, 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\_CREATE: {

asyncTask = std::async(std::launch::async, UpdateProcessList, hwndListView);

// Создать ListView для отображения списка процессов

hwndListView = CreateWindow(WC\_LISTVIEW, L"", WS\_VISIBLE | WS\_CHILD | LVS\_REPORT,

10, 10, 480, 300, hWnd, (HMENU)ID\_LISTVIEW, nullptr, nullptr);

HWND hwndButton = CreateWindow(L"BUTTON", L"End Task", WS\_TABSTOP | WS\_VISIBLE | WS\_CHILD | BS\_DEFPUSHBUTTON,

10, 325, 80, 25, hWnd, (HMENU)ID\_BUTTON\_ENDTASK, GetModuleHandle(nullptr), nullptr);

HWND hwndButtonSuspend = CreateWindow(L"BUTTON", L"Suspend", WS\_TABSTOP | WS\_VISIBLE | WS\_CHILD | BS\_DEFPUSHBUTTON,

100, 325, 80, 25, hWnd, (HMENU)ID\_BUTTON\_SUSPEND, GetModuleHandle(nullptr), nullptr);

HWND hwndButtonResume = CreateWindow(L"BUTTON", L"Resume", WS\_TABSTOP | WS\_VISIBLE | WS\_CHILD | BS\_DEFPUSHBUTTON,

190, 325, 80, 25, hWnd, (HMENU)ID\_BUTTON\_RESUME, GetModuleHandle(nullptr), nullptr);

// Инициализировать ListView

LVCOLUMN lvColumn = { 0 };

lvColumn.mask = LVCF\_TEXT | LVCF\_WIDTH | LVCF\_SUBITEM;

lvColumn.pszText = const\_cast<LPWSTR>(L""); // Привести к типу LPWSTR

lvColumn.cx = 20;

ListView\_InsertColumn(hwndListView, 0, &lvColumn);

lvColumn.pszText = const\_cast<LPWSTR>(L"Process");

lvColumn.cx = 150;

ListView\_InsertColumn(hwndListView, 1, &lvColumn);

lvColumn.pszText = const\_cast<LPWSTR>(L"PID");

lvColumn.cx = 50;

ListView\_InsertColumn(hwndListView, 2, &lvColumn);

lvColumn.pszText = const\_cast<LPWSTR>(L"Memory (KB)");

lvColumn.cx = 90;

ListView\_InsertColumn(hwndListView, 3, &lvColumn);

lvColumn.pszText = const\_cast<LPWSTR>(L"Threads");

lvColumn.cx = 60;

ListView\_InsertColumn(hwndListView, 4, &lvColumn);

ListView\_SetExtendedListViewStyle(hwndListView, LVS\_EX\_CHECKBOXES | LVS\_EX\_FULLROWSELECT);

// Обновить список процессов

UpdateProcessList(hwndListView);

break;

}

case WM\_COMMAND:

switch (LOWORD(wParam)) {

case ID\_BUTTON\_ENDTASK:

// Пройти по всем элементам ListView и найти выделенный

for (int i = 0; i < ListView\_GetItemCount(hwndListView); ++i) {

if (ListView\_GetCheckState(hwndListView, i)) {

index = i;

break;

}

}

if (index != -1) {

// Получить PID выбранного процесса

wchar\_t buffer[256];

ListView\_GetItemText(hwndListView, index, 2, buffer, sizeof(buffer));

DWORD pid = \_wtoi(buffer);

// Отправить сигнал завершения процесса

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

if (hProcess != NULL) {

TerminateProcess(hProcess, 0);

CloseHandle(hProcess);

// Удалить элемент из ListView

ListView\_DeleteItem(hwndListView, index);

// Обновить список процессов

UpdateProcessList(hwndListView);

}

}

break;

case ID\_BUTTON\_SUSPEND:

for (int i = 0; i < ListView\_GetItemCount(hwndListView); ++i) {

if (ListView\_GetCheckState(hwndListView, i)) {

suspendIndex = i;

break;

}

}

if (suspendIndex != -1) {

wchar\_t buffer[256];

ListView\_GetItemText(hwndListView, suspendIndex, 2, buffer, sizeof(buffer));

DWORD pid = \_wtoi(buffer);

// Приостановить процесс

SuspendProcess(pid);

}

break;

case ID\_BUTTON\_RESUME:

for (int i = 0; i < ListView\_GetItemCount(hwndListView); ++i) {

if (ListView\_GetCheckState(hwndListView, i)) {

resumeIndex = i;

break;

}

}

if (resumeIndex != -1) {

wchar\_t buffer[256];

ListView\_GetItemText(hwndListView, resumeIndex, 2, buffer, sizeof(buffer));

DWORD pid = \_wtoi(buffer);

// Возобновить процесс

ResumeProcess(pid);

}

break;

}

break;

case WM\_CLOSE:

if (asyncTask.valid())

asyncTask.get();

DestroyWindow(hWnd);

break;

case WM\_DESTROY:

PostQuitMessage(0);

break;

default:

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

}

return 0;

}

void UpdateProcessList(HWND hwndListView) {

// Очистить текущий список процессов

ListView\_DeleteAllItems(hwndListView);

// Получить список процессов

DWORD processIds[1024], cbNeeded, cProcesses;

if (EnumProcesses(processIds, sizeof(processIds), &cbNeeded)) {

// Вычислить количество процессов

cProcesses = cbNeeded / sizeof(DWORD);

// Для каждого процесса

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

// Открыть процесс

HANDLE hProcess = OpenProcess(PROCESS\_QUERY\_INFORMATION | PROCESS\_VM\_READ, FALSE, processIds[i]);

if (hProcess != nullptr) {

// Получить модуль процесса

HMODULE hMod;

DWORD cbNeeded;

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

// Получить базовое имя модуля

TCHAR szProcessName[MAX\_PATH];

GetModuleBaseName(hProcess, hMod, szProcessName, sizeof(szProcessName) / sizeof(TCHAR));

// Получить использование памяти

PROCESS\_MEMORY\_COUNTERS pmc;

if (GetProcessMemoryInfo(hProcess, &pmc, sizeof(pmc))) {

DWORD pid = GetProcessId(hProcess);

DWORD threadCount = 0;

EnumThreadsInProcess(pid, threadCount);

// Получить временные данные процесса

FILETIME creationTime, exitTime, kernelTime, userTime;

GetProcessTimes(hProcess, &creationTime, &exitTime, &kernelTime, &userTime);

// Добавить элемент в список

LVITEM lvItem = { 0 };

lvItem.mask = LVIF\_TEXT;

lvItem.iItem = i;

lvItem.iSubItem = 0;

int itemIndex = ListView\_InsertItem(hwndListView, &lvItem);

// Установить радиокнопку в 0 колонку

ListView\_SetCheckState(hwndListView, itemIndex, 0, FALSE);

// Установить имя процесса в 1 колонку

ListView\_SetItemText(hwndListView, itemIndex, 1, szProcessName);

// Создать буфер для хранения текста

wchar\_t buffer[256];

\_itow\_s(pmc.WorkingSetSize / 1024, buffer, 10);

// Преобразовать PID в строку

std::wstring pidString = std::to\_wstring(pid);

LPWSTR pidText = const\_cast<LPWSTR>(pidString.c\_str());

ListView\_SetItemText(hwndListView, itemIndex, 2, pidText); // PID

ListView\_SetItemText(hwndListView, itemIndex, 3, buffer); // Установить размер в KB в 3 колонку

std::wstring threadCountString = std::to\_wstring(threadCount);

LPWSTR threadCountText = const\_cast<LPWSTR>(threadCountString.c\_str());

ListView\_SetItemText(hwndListView, itemIndex, 4, threadCountText); // Установить Treads

}

}

// Закрыть дескриптор процесса

CloseHandle(hProcess);

}

}

}

}

void EnumThreadsInProcess(DWORD processId, DWORD& threadCount) {

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

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

return;

}

THREADENTRY32 threadEntry;

threadEntry.dwSize = sizeof(THREADENTRY32);

if (Thread32First(hThreadSnapshot, &threadEntry)) {

do {

if (threadEntry.th32OwnerProcessID == processId) {

threadCount++;

}

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

}

CloseHandle(hThreadSnapshot);

}

void SuspendProcess(DWORD processId) {

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

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

THREADENTRY32 threadEntry;

threadEntry.dwSize = sizeof(THREADENTRY32);

if (Thread32First(hSnapshot, &threadEntry)) {

do {

if (threadEntry.th32OwnerProcessID == processId) {

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

if (hThread != nullptr) {

SuspendThread(hThread);

CloseHandle(hThread);

}

}

} while (Thread32Next(hSnapshot, &threadEntry));

}

CloseHandle(hSnapshot);

}

}

void ResumeProcess(DWORD processId) {

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

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

THREADENTRY32 threadEntry;

threadEntry.dwSize = sizeof(THREADENTRY32);

if (Thread32First(hSnapshot, &threadEntry)) {

do {

if (threadEntry.th32OwnerProcessID == processId) {

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

if (hThread != nullptr) {

ResumeThread(hThread);

CloseHandle(hThread);

}

}

} while (Thread32Next(hSnapshot, &threadEntry));

}

CloseHandle(hSnapshot);

}

}