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

Сачивко В.Г.

Проверил: Гриценко Н.Ю.

Минск 2023

**СОДЕРЖАНИЕ**

1 Цель работы 3

2 Теоретические сведения 4  
3 Результат выполнения программы 5

[Список использованных источников 7](#_TOC_250001)

[Приложение А 8](#_TOC_250000)

**1 ЦЕЛЬ РАБОТЫ**

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

**2 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

Управление процессами и потоками в операционных системах *Windows* представляет собой важный аспект для эффективного использования ресурсов компьютера и обеспечения многозадачности. Процесс представляет собой экземпляр выполняемой программы и имеет свое собственное адресное пространство и ресурсы. Поток, с другой стороны, является наименьшей единицей выполнения внутри процесса и существует в его контексте, используя его ресурсы.

Процессы могут быть созданы, выполняться, приостановлены, завершены и уничтожены. Они могут взаимодействовать друг с другом с использованием механизмов межпроцессного взаимодействия (*IPC*). Потоки позволяют многозадачным операционным системам эффективно использовать ресурсы, так как несколько потоков могут выполняться параллельно в пределах одного процесса.

Создание процесса осуществляется с использованием функции «*CreateProcess*», что позволяет порождать новые экземпляры программы. Создание потока внутри процесса выполняется с помощью функции «*CreateThread*». Процессы и потоки также могут быть завершены с использованием соответствующих функций, таких как «*TerminateProcess*» и «*ExitThread*». [1]

У процессов и потоков есть приоритеты, которые определяют их относительное право на использование процессорного времени. Высший приоритет дает процессу или потоку больше процессорного времени. Управление приоритетами позволяет эффективно распределять ресурсы и оптимизировать производительность системы.

*Windows* предоставляет различные инструменты для мониторинга процессов и потоков, такие как Диспетчер задач и Производительность. *API* *Windows* предоставляет функции для получения информации о процессах и потоках в коде, что полезно для исследования и оптимизации эффективности системы.

В целом, понимание управления процессами и потоками важно для разработчиков, поскольку это позволяет создавать стабильные, отзывчивые и эффективные приложения под *Windows*.

**3 РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ ПРОГРАММЫ**

При нажатии на кнопку «Обновить» появляется список процессов в виде таблицы, в которой указаны PID процесса, занимаемая память и название процесса (рисунок 1).

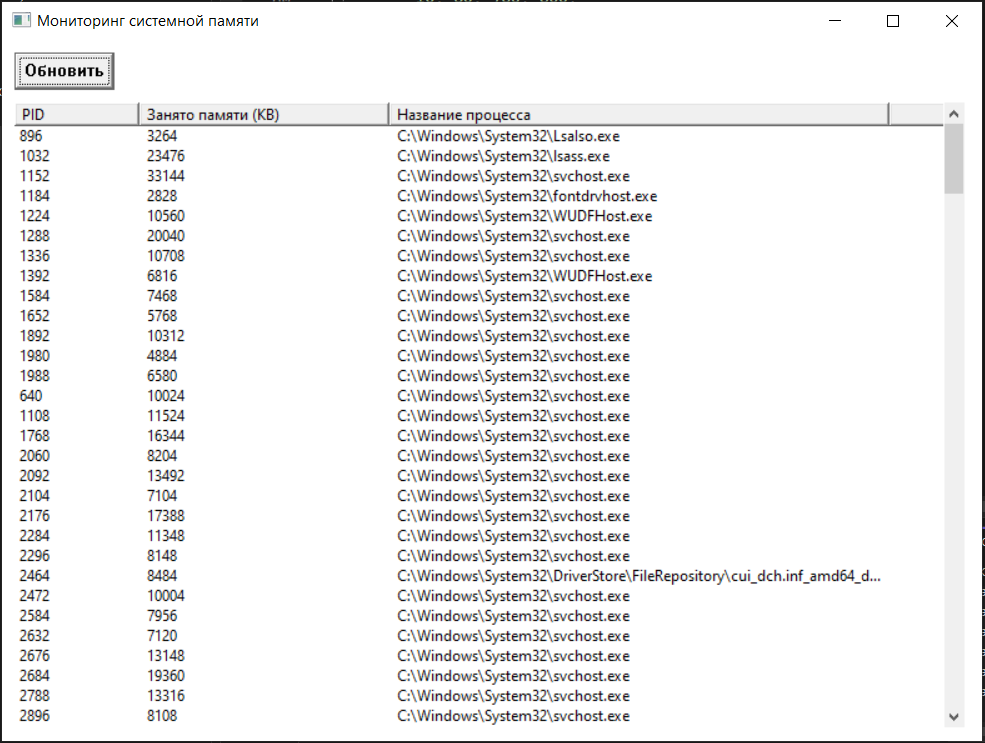


Рисунок 1 – Список процессов

Пользователь может завершить, приостановить или возобновить процесс. Для этого необходимо нажать правой кнопкой по определенному процессу в таблице и выбрать нужное (рисунок 2).

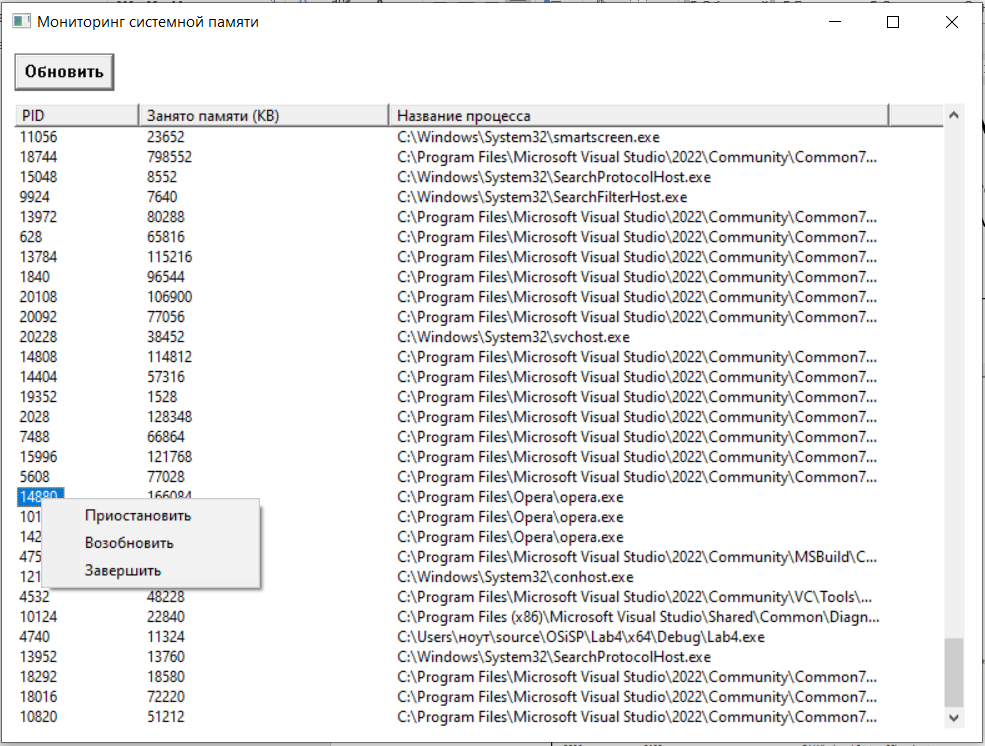


Рисунок 2 – Выбор действия

**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

[1] Функция TerminateProcess[Электронный ресурс]. – Режим доступа: <https://learn.microsoft.com/ruru/windows/win32/api/processthreadsapi/nf-processthreadsapi-terminateprocess>

**ПРИЛОЖЕНИЕ А**

**Исходный код программы**

**Файл Lab4.cpp**

#include <windows.h>

#include <psapi.h>

#include <commctrl.h>

#include <iostream>

#include <string>

#include <vector>

#include <string>

#include <tlhelp32.h>

#include <tchar.h>

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

#pragma comment(lib, "kernel32.lib")

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

void ListProcesses(HWND hwnd);

void TerminateSelectedProcess(DWORD processId);

void SuspendResumeSelectedProcess(DWORD processId, bool suspend);

std::vector<std::wstring> processListTextLines;

HMENU hMenu;

int WINAPI wWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PWSTR pCmdLine, int nCmdShow) {

INITCOMMONCONTROLSEX iccex;

iccex.dwSize = sizeof(INITCOMMONCONTROLSEX);

iccex.dwICC = ICC\_LISTVIEW\_CLASSES;

InitCommonControlsEx(&iccex);

WNDCLASS wc = {};

wc.lpfnWndProc = WindowProc;

wc.hInstance = hInstance;

wc.lpszClassName = L"MemoryMonitorApp";

RegisterClass(&wc);

HWND hwnd = CreateWindowEx(

0,

L"MemoryMonitorApp",

L"Мониторинг системной памяти",

WS\_OVERLAPPEDWINDOW,

CW\_USEDEFAULT, CW\_USEDEFAULT, 800, 600,

NULL, NULL, hInstance, NULL);

if (hwnd == NULL) {

return 0;

}

HMENU hMenu = CreatePopupMenu();

AppendMenu(hMenu, MF\_STRING, 101, L"Приостановить");

AppendMenu(hMenu, MF\_STRING, 102, L"Возобновить");

AppendMenu(hMenu, MF\_STRING, 103, L"Завершить");

SetMenu(hwnd, hMenu);

CreateWindow(

L"BUTTON",

L"Обновить",

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

10, 10, 80, 30,

hwnd, (HMENU)1, hInstance, NULL);

HWND hList = CreateWindowEx(

0,

WC\_LISTVIEW,

L"",

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

10, 50, 760, 500,

hwnd,

(HMENU)2,

GetModuleHandle(NULL),

NULL);

DWORD dwStyle = GetWindowLong(hList, GWL\_STYLE);

dwStyle |= LVS\_REPORT;

SetWindowLong(hList, GWL\_STYLE, dwStyle);

LVCOLUMN lvc;

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

lvc.iSubItem = 0;

wchar\_t pidText[] = L"PID";

lvc.pszText = pidText;

lvc.cx = 100;

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

lvc.iSubItem = 1;

wchar\_t memoryText[] = L"Занято памяти (KB)";

lvc.pszText = memoryText;

lvc.cx = 200;

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

lvc.iSubItem = 2;

wchar\_t processText[] = L"Название процесса";

lvc.pszText = processText;

lvc.cx = 400;

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

ShowWindow(hwnd, nCmdShow);

MSG msg;

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

TranslateMessage(&msg);

DispatchMessage(&msg);

}

processListTextLines.clear();

return 0;

}

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

HWND hList;

int selectedIdx;

switch (uMsg) {

case WM\_CREATE: {

hMenu = CreatePopupMenu();

AppendMenu(hMenu, MF\_STRING, 101, L"Приостановить");

AppendMenu(hMenu, MF\_STRING, 102, L"Возобновить");

AppendMenu(hMenu, MF\_STRING, 103, L"Завершить");

break;

}

case WM\_COMMAND:

switch (LOWORD(wParam)) {

case 1:

ListProcesses(hwnd);

break;

case 101: // Приостановить

hList = GetDlgItem(hwnd, 2);

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

if (selectedIdx != -1) {

WCHAR buffer[256];

ListView\_GetItemText(hList, selectedIdx, 0, buffer, 256);

std::wstring wstr(buffer);

std::string str(wstr.begin(), wstr.end());

DWORD processId = std::stoi(str);

SuspendResumeSelectedProcess(processId, true);

}

break;

case 102: // Возобновить

hList = GetDlgItem(hwnd, 2);

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

if (selectedIdx != -1) {

WCHAR buffer[256];

ListView\_GetItemText(hList, selectedIdx, 0, buffer, 256);

std::wstring wstr(buffer);

std::string str(wstr.begin(), wstr.end());

DWORD processId = std::stoi(str);

SuspendResumeSelectedProcess(processId, false);

}

break;

case 103: // Завершить

hList = GetDlgItem(hwnd, 2);

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

if (selectedIdx != -1) {

WCHAR buffer[256];

ListView\_GetItemText(hList, selectedIdx, 0, buffer, 256);

std::wstring wstr(buffer);

std::string str(wstr.begin(), wstr.end());

DWORD processId = std::stoi(str);

TerminateSelectedProcess(processId);

}

break;

}

break;

case WM\_CONTEXTMENU:

if ((HWND)wParam == GetDlgItem(hwnd, 2)) {

POINT pt;

GetCursorPos(&pt);

TrackPopupMenu(hMenu, TPM\_LEFTALIGN | TPM\_RIGHTBUTTON, pt.x, pt.y, 0, hwnd, NULL);

}

break;

case WM\_DESTROY:

processListTextLines.clear();

PostQuitMessage(0);

break;

default:

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

}

return 0;

}

void TerminateSelectedProcess(DWORD processId) {

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

if (hProcess != NULL) {

TerminateProcess(hProcess, 0);

CloseHandle(hProcess);

}

}

void SuspendResumeSelectedProcess(DWORD processId, bool suspend) {

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

if (hProcess != NULL) {

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

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

THREADENTRY32 threadEntry;

threadEntry.dwSize = sizeof(THREADENTRY32);

if (Thread32First(hThreadSnapshot, &threadEntry)) {

do {

if (threadEntry.th32OwnerProcessID == processId) {

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

if (hThread != NULL) {

if (suspend) {

SuspendThread(hThread);

}

else {

ResumeThread(hThread);

}

CloseHandle(hThread);

}

}

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

}

CloseHandle(hThreadSnapshot);

}

CloseHandle(hProcess);

}

}

void ListProcesses(HWND hwnd) {

HWND hList = GetDlgItem(hwnd, 2);

// Очищаем предыдущий список процессов

ListView\_DeleteAllItems(hList);

std::vector<DWORD> processes(1024);

DWORD bytesNeeded;

if (!EnumProcesses(processes.data(), processes.size() \* sizeof(DWORD), &bytesNeeded)) {

DWORD error = GetLastError();

std::cerr << "EnumProcesses не удалось выполнить с ошибкой: " << error << "\n";

std::cerr << "Требуемый размер буфера: " << bytesNeeded << " байт\n";

return;

}

DWORD processCount = bytesNeeded / sizeof(DWORD);

processes.resize(processCount);

// Очищаем и заново выделяем память для строк

processListTextLines.clear();

size\_t totalStringLength = processCount \* 3; // 3 строки на каждый процесс

processListTextLines.resize(totalStringLength);

LVITEM lvi = { 0 };

lvi.mask = LVIF\_TEXT;

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

HANDLE process = OpenProcess(PROCESS\_QUERY\_LIMITED\_INFORMATION | PROCESS\_VM\_READ, FALSE, processes[i]);

if (process != NULL) {

PROCESS\_MEMORY\_COUNTERS\_EX pmc = {};

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

TCHAR szProcessName[MAX\_PATH] = TEXT("<unknown>");

DWORD dwSize = sizeof(szProcessName) / sizeof(TCHAR);

if (QueryFullProcessImageName(process, 0, szProcessName, &dwSize)) {

processListTextLines[i \* 3] = std::to\_wstring(processes[i]);

processListTextLines[i \* 3 + 1] = std::to\_wstring(pmc.WorkingSetSize / 1024);

processListTextLines[i \* 3 + 2] = szProcessName;

lvi.iItem = i;

lvi.iSubItem = 0;

lvi.pszText = const\_cast<LPWSTR>(processListTextLines[i \* 3].c\_str());

int itemIndex = ListView\_InsertItem(hList, &lvi);

lvi.iSubItem = 1;

lvi.pszText = const\_cast<LPWSTR>(processListTextLines[i \* 3 + 1].c\_str());

ListView\_SetItemText(hList, itemIndex, 1, lvi.pszText);

lvi.iSubItem = 2;

lvi.pszText = const\_cast<LPWSTR>(processListTextLines[i \* 3 + 2].c\_str());

ListView\_SetItemText(hList, itemIndex, 2, lvi.pszText);

UpdateWindow(hList);

}

else {

DWORD error = GetLastError();

std::cerr << "QueryFullProcessImageName не удалось выполнить с ошибкой: " << error << "\n";

std::cerr << "ID процесса: " << processes[i] << "\n";

}

}

else {

DWORD error = GetLastError();

std::cerr << "GetProcessMemoryInfo не удалось выполнить с ошибкой: " << error << "\n";

std::cerr << "ID процесса: " << processes[i] << "\n";

}

CloseHandle(process);

}

else {

DWORD error = GetLastError();

std::cerr << "OpenProcess не удалось выполнить с ошибкой: " << error << "\n";

std::cerr << "ID процесса: " << processes[i] << "\n";

}

// Обновим lvi перед следующей вставкой

ZeroMemory(&lvi, sizeof(lvi));

lvi.mask = LVIF\_TEXT;

}

}