Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет   
информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Операционные системы и системное программирование

Отчёт

к лабораторной работе

на тему

Средства синхронизации и взаимного исключения (Windows). Изучение и использование средств синхронизации и взаимного исключения.

Студент: гр.153504

Пригожий К. А.

Проверил: Гриценко Н.Ю.

Минск 2023

**СОДЕРЖАНИЕ**

[1 Цель работы 3](#_Toc146635545)

[2 Теоретические сведения 4](#_Toc146635546)

[3 Результат выполнения программы 5](#_Toc146635547)

[Список использованных источников 7](#_Toc146635548)

[Приложение А 8](#_Toc146635549)

# 1 ЦЕЛЬ РАБОТЫ

Реализовать алгоритм философов обедающих, где философы (потоки) соревнуются за доступ к вилкам (ресурсам) с использованием мьютексов.

# 2 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Задача об обедающих философах — классический пример, используемый в информатике для иллюстрации проблем синхронизации при разработке параллельных алгоритмов и техник решения этих проблем.

Задача была сформулирована в 1965 году Эдсгером Дейкстрой как экзаменационное упражнение для студентов. В качестве примера был взят конкурирующий доступ к ленточному накопителю. Вскоре задача была сформулирована Энтони Хоаром в том виде, в каком она известна сегодня.

*Deadlock* — ситуация в многозадачной среде или СУБД, при которой несколько процессов находятся в состоянии бесконечного ожидания ресурсов, захваченных самими этими процессами.

Мьютекс – это синхронизационный объект, используемый для обеспечения взаимного исключения между потоками в многозадачных приложениях. Мьютексы позволяют только одному потоку одновременно получать доступ к критическому участку кода или ресурсам.

Для создания и управления мьютексами в *Win32* *API* используются функции, такие как *CreateMutex*, *WaitForSingleObject*, *ReleaseMutex*. Мьютексы могут использоваться как внутри одного процесса, так и между разными процессами[1].

Семафор - это синхронизационный объект, который может использоваться для управления доступом нескольких потоков к общим ресурсам. В отличие от мьютекса, семафор может разрешать доступ нескольким потокам одновременно, в зависимости от значения счетчика семафора.

В *Win32* *API* семафоры могут быть созданы с помощью функции *CreateSemaphore*, и потоки могут увеличивать или уменьшать значение счетчика семафора с помощью функций *ReleaseSemaphore* и *WaitForSingleObject.*[2]

*CreateThread* – это функция в *Win32* *API*, используемая для создания нового потока выполнения в текущем процессе. Она принимает указатель на функцию, которая будет выполняться в новом потоке, и аргументы, передаваемые этой функции.

После создания потока его выполнение начинается с указанной функции. *CreateThread* возвращает дескриптор нового потока, который может быть использован для управления и синхронизации потоками, например, с помощью мьютексов или семафоров.[3]

# 3 РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ ПРОГРАММЫ

Реализованная программа является консольным приложением имитирующим действия пяти философов сидящих за одним столом. Каждый философ может думать и есть. Философы могут есть только двумя вилками. В рамках задачи вилки – общий ресурс (мьютекс). Если каждый философ одновременно возьмет вилку, например левую от себя, возникнет такая ситуация, при которой невозможно будет взять вторую вилку.

Одно из решений данной проблемы звучит следующим образом. Философов тянущихся за вилкой можно изобразить в виде циклического графа, где каждый философ ждет, пока сосед освободит нужный ресурс (рисунок 1).

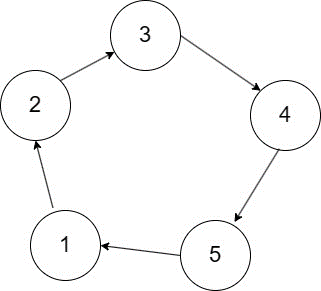


Рисунок 1 – Граф ожидающих друг друга философов

Решение заключается в изменении направления ожидания последнего философа в другую сторону (рисунок 2). Это значит, что последний философ будет брать не левую вилку первой, а вторую.

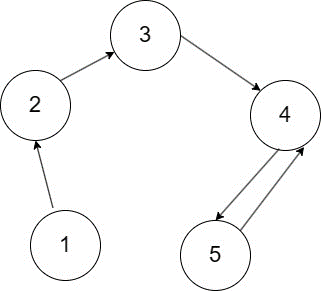


Рисунок 2 – Вывод при попытке припарковаться при наличии свободных мест

По прошествии 5 секунд машина выезжает с парковки и выводится сообщение (рисунок 3)

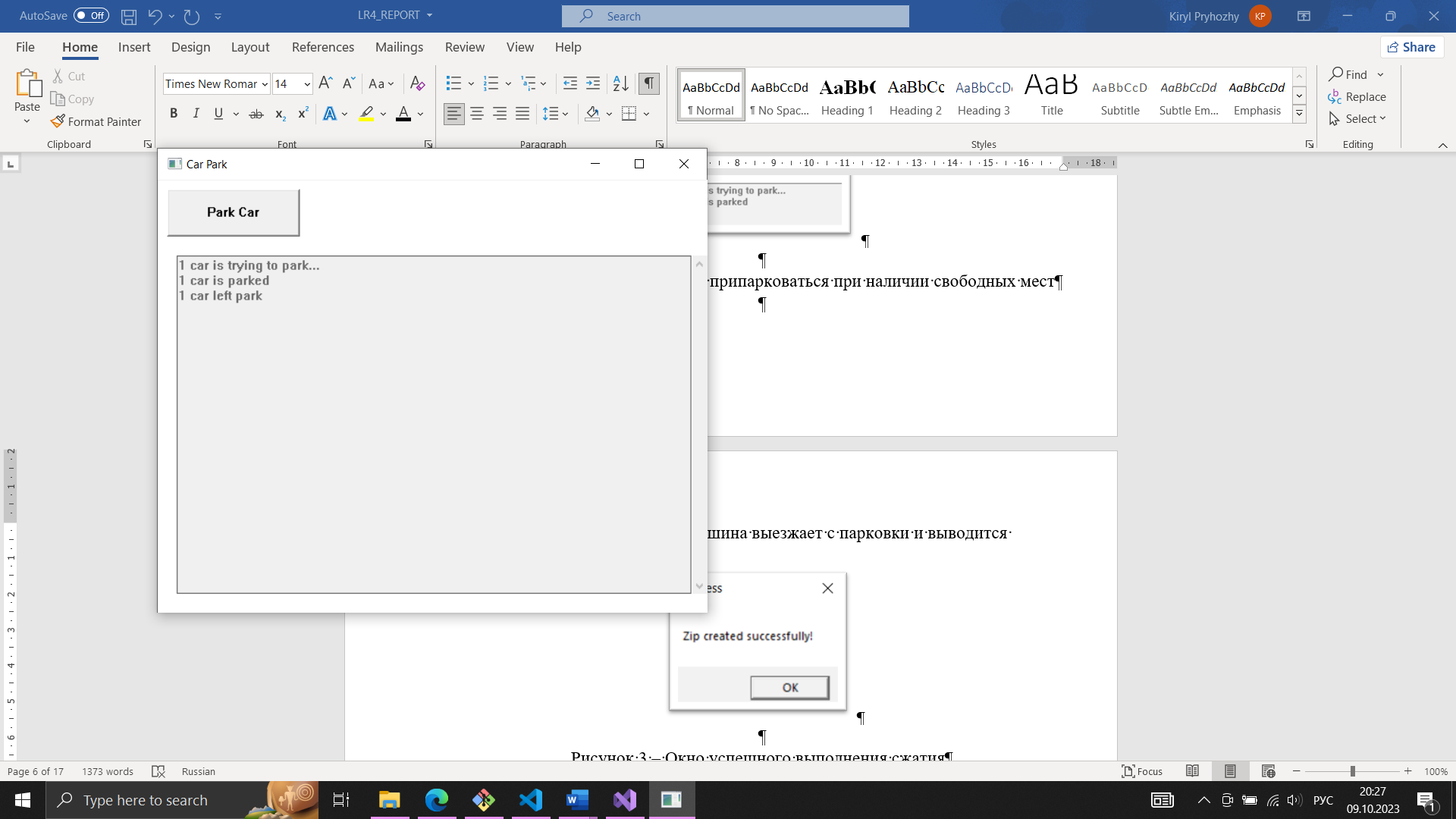


Рисунок 3 – Вывод при выезде машины из парковки

Если в настоящее время нету свободных парковочных мест, то выводится соответствующее сообщение(рисунок 4)*.*

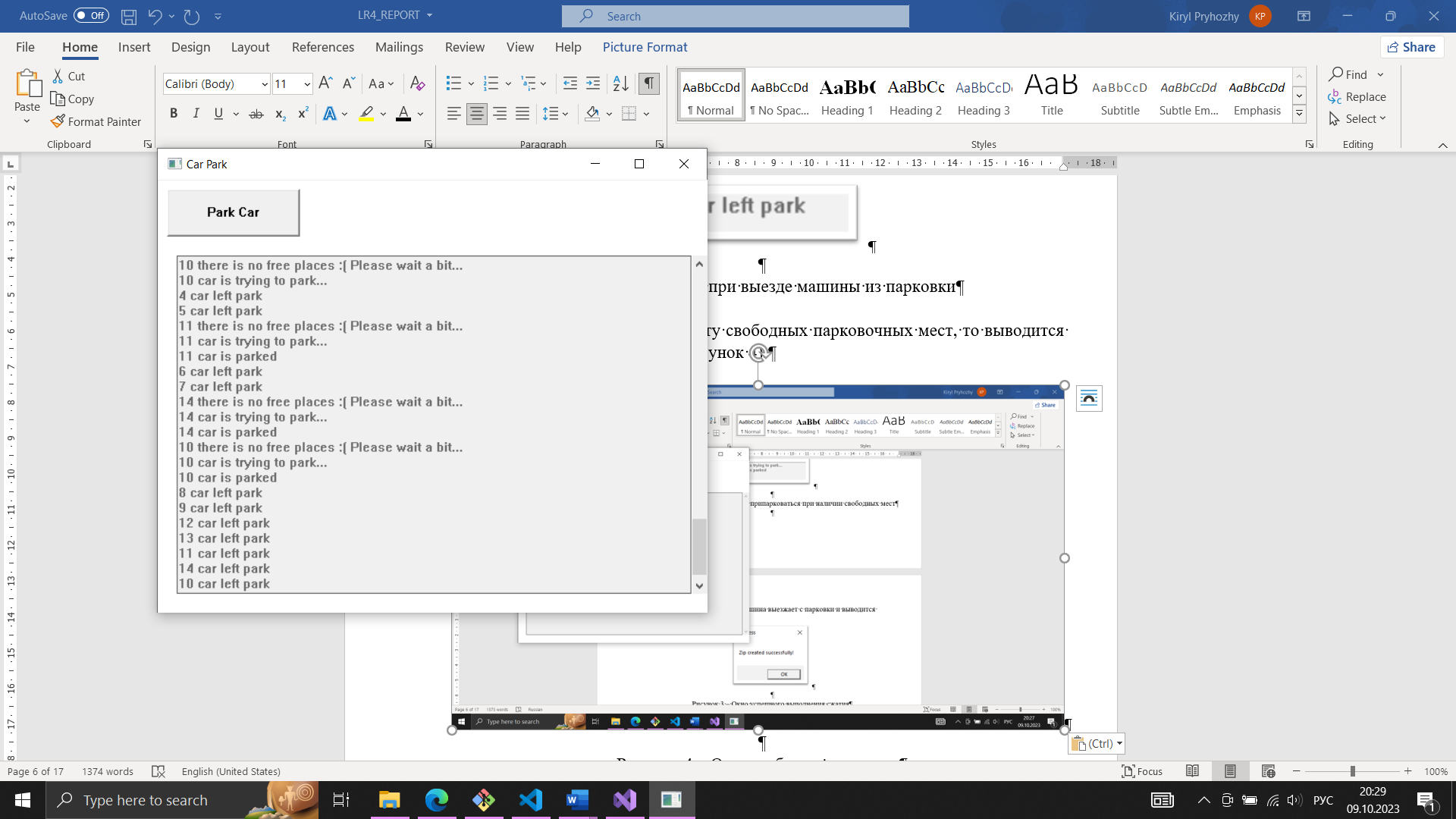


Рисунок 4 – Вывод при недостатке свободных мест

Для синхронизации в парковочных местах был использован семафор со счетчиком 8 и для корректного вывода сообщений использовался мьютекс, который позволял одновременно писать сообщения только одному потоку.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Using Mutex Objects - Win32 apps | Microsoft Learn [Электронный ресурс]. – Режим доступа: <https://learn.microsoft.com/en-us/windows/win32/sync/using-mutex-objects>

[2] Using Semaphore Objects - Win32 apps | Microsoft Learn [Электронный ресурс]. – Режим доступа: <https://learn.microsoft.com/en-us/windows/win32/sync/using-semaphore-objects>

[3] Creating Threads - Win32 apps | Microsoft Learn [Электронный ресурс]. – Режим доступа: <https://learn.microsoft.com/en-us/windows/win32/procthread/creating-threads>

# ПРИЛОЖЕНИЕ А

**(обязательное)**

**Исходный код программы**

**Файл main.h**

#pragma once

#include <Windows.h>

#include <string>

// constants

const wchar\_t APPLICATION\_NAME[] = L"Car Park";

const wchar\_t MAIN\_WINDOW\_CLASS\_NAME[] = L"MainWindowClassName";

const int MAIN\_WINDOW\_INITIAL\_WIDTH = 600;

const int MAIN\_WINDOW\_INITIAL\_HEIGHT = 500;

const int PARK\_PLACES\_COUNT = 8;

// buttons params

const int BUTTON\_WIDTH = 140;

const int BUTTON\_HEIGHT = 50;

// identifiers

#define IDC\_PARK\_CAR\_BUTTON 1

// variables

HWND editControl;

HANDLE semaphore;

HANDLE mutex;

int id = 0;

// functions

DWORD WINAPI ParkCar(LPVOID lpParam);

HWND InstantiateMainWindow(HINSTANCE hInstance);

WNDCLASSEX CreateMainWindowClass(HINSTANCE hInstance);

void PrintMessage(std::string message);

**Файл main.cpp**

#include "main.h"

int WINAPI wWinMain(HINSTANCE hInstance, HINSTANCE, PWSTR szCmdLine, int #include "main.h"

int WINAPI wWinMain(HINSTANCE hInstance, HINSTANCE, PWSTR szCmdLine, int nCmdShow)

{

WNDCLASSEX mainWindowClass = CreateMainWindowClass(hInstance);

if (not RegisterClassEx(&mainWindowClass)) {

MessageBoxA(NULL, "Main window hasn't registered", "Error", MB\_OK);

return EXIT\_FAILURE;

}

HWND mainWindow = InstantiateMainWindow(hInstance);

HWND archiveButton = CreateWindow(

L"BUTTON", L"Archive",

WS\_TABSTOP | WS\_VISIBLE | WS\_CHILD,

10, 10, BUTTON\_WIDTH, BUTTON\_HEIGHT,

mainWindow,

(HMENU)IDC\_ARCHIVE\_BUTTON,

hInstance,

NULL

);

HWND extractButton = CreateWindow(

L"BUTTON", L"Extract",

WS\_TABSTOP | WS\_VISIBLE | WS\_CHILD,

250, 10, BUTTON\_WIDTH, BUTTON\_HEIGHT,

mainWindow,

(HMENU)IDC\_EXTRACT\_BUTTON,

hInstance,

NULL

);

ShowWindow(mainWindow, nCmdShow);

UpdateWindow(mainWindow);

MSG message;

while (GetMessage(&message, NULL, 0, 0) > 0)

{

TranslateMessage(&message);

DispatchMessage(&message);

}

return message.wParam;

}

LRESULT CALLBACK WindowProc(HWND hWindow, UINT message, WPARAM wParam, LPARAM lParam)

{

switch (message) {

case WM\_COMMAND: {

switch (LOWORD(wParam)) {

case IDC\_ARCHIVE\_BUTTON: {

bool isPicked = PickFolder();

if (isPicked) {

std::wstring folderPath(selectedFolder);

std::wstring zipFilePath = std::wstring(selectedFolder).append(L".zip");

AddFolderToZip(folderPath.c\_str(), zipFilePath.c\_str());

MessageBoxA(hWindow, "Zip created successfully!", "Success", MB\_OK);

}

break;

}

case IDC\_EXTRACT\_BUTTON: {

bool isPicked = PickZipToExtract();

if (isPicked) {

std::string outputFolderPath(selectedZip.begin(), selectedZip.end());

size\_t dotPos = outputFolderPath.find\_last\_of(".");

std::string filePath(selectedZip.begin(), selectedZip.end());

if (dotPos != std::string::npos) {

outputFolderPath = outputFolderPath.substr(0, dotPos);

}

ExtractZip(filePath.c\_str(), outputFolderPath.c\_str());

MessageBoxA(hWindow, "Files extracted successfully!", "Success", MB\_OK);

}

break;

}

}

return 0;

}

case WM\_CLOSE:

DestroyWindow(hWindow);

return 0;

case WM\_DESTROY:

PostQuitMessage(0);

return 0;

default:

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

}

}

WNDCLASSEX CreateMainWindowClass(HINSTANCE hInstance)

{

WNDCLASSEX wc{};

wc.cbSize = sizeof(WNDCLASSEX);

wc.lpfnWndProc = WindowProc;

wc.hInstance = hInstance;

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

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

wc.hIconSm = LoadIcon(NULL, IDI\_APPLICATION);

wc.lpszClassName = MAIN\_WINDOW\_CLASS\_NAME;

return wc;

}

HWND InstantiateMainWindow(HINSTANCE hInstance)

{

return CreateWindowEx(

WS\_EX\_CLIENTEDGE,

MAIN\_WINDOW\_CLASS\_NAME, APPLICATION\_NAME, //APPLICATION\_NAME,

WS\_OVERLAPPEDWINDOW,

CW\_USEDEFAULT, CW\_USEDEFAULT, MAIN\_WINDOW\_INITIAL\_WIDTH, MAIN\_WINDOW\_INITIAL\_HEIGHT,

NULL, NULL, hInstance, NULL

);

}

bool PickFolder()

{

HRESULT hr = CoInitialize(NULL);

if (FAILED(hr)) {

return false;

}

IFileOpenDialog\* pfd;

hr = CoCreateInstance(CLSID\_FileOpenDialog, NULL, CLSCTX\_INPROC\_SERVER, IID\_PPV\_ARGS(&pfd));

if (FAILED(hr)) {

CoUninitialize();

return false;

}

SetFileDialogOptions(pfd);

BrowseFolder(pfd);

pfd->Release();

CoUninitialize();

return true;

}

void SetFileDialogOptions(IFileOpenDialog\* pfd)

{

DWORD dwOptions;

pfd->GetOptions(&dwOptions);

pfd->SetOptions(dwOptions | FOS\_PICKFOLDERS);

}

void BrowseFolder(IFileOpenDialog\* pfd)

{

HRESULT hr = pfd->Show(NULL);

if (FAILED(hr))

return;

IShellItem\* result;

hr = pfd->GetResult(&result);

if (FAILED(hr))

return;

PWSTR selectedFolderPath;

hr = result->GetDisplayName(SIGDN\_DESKTOPABSOLUTEPARSING, &selectedFolderPath);

if (FAILED(hr))

return;

OutputDebugString(selectedFolderPath);

selectedFolder = selectedFolderPath;

}

void AddFileToZip(const wchar\_t\* zipFileName, const wchar\_t\* sourceFileName)

{

struct zip\* archive;

struct zip\_source\* source;

archive = zip\_open(WcharToString(zipFileName).c\_str(), ZIP\_CREATE, NULL);

if (archive == NULL) {

perror("Failed to open zip archive");

return;

}

source = zip\_source\_file(archive, WcharToString(sourceFileName).c\_str(), 0, 0);

if (source == NULL) {

perror("Failed to create zip source from file");

zip\_close(archive);

return;

}

std::wstring filePath = sourceFileName;

size\_t found = filePath.find(selectedFolder + L"\\");

if (found != std::wstring::npos) {

filePath.erase(found, selectedFolder.length() + 1);

}

if (zip\_file\_add(archive, WcharToString(filePath.c\_str()).c\_str(), source, 0) < 0) {

perror("Failed to add file to zip archive");

zip\_source\_free(source);

zip\_close(archive);

return;

}

zip\_close(archive);

}

void AddFolderToZip(const wchar\_t\* folderPath, const wchar\_t\* zipFileName) {

WIN32\_FIND\_DATA findFileData;

wchar\_t searchPattern[MAX\_PATH];

swprintf(searchPattern, sizeof(searchPattern) / sizeof(searchPattern[0]), L"%s\\\*", folderPath);

HANDLE hFind = FindFirstFile(searchPattern, &findFileData);

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

do {

if (!(findFileData.dwFileAttributes & FILE\_ATTRIBUTE\_DIRECTORY)) {

wchar\_t filePath[MAX\_PATH];

swprintf(filePath, sizeof(filePath) / sizeof(filePath[0]), L"%s\\%s", folderPath, findFileData.cFileName);

AddFileToZip(zipFileName, filePath);

}

else if (wcscmp(findFileData.cFileName, L".") != 0 && wcscmp(findFileData.cFileName, L"..") != 0) {

wchar\_t subFolderPath[MAX\_PATH];

swprintf(subFolderPath, sizeof(subFolderPath) / sizeof(subFolderPath[0]), L"%s\\%s", folderPath, findFileData.cFileName);

AddFolderToZip(subFolderPath, zipFileName);

}

} while (FindNextFile(hFind, &findFileData) != 0);

FindClose(hFind);

}

}

void ExtractZip(const char\* zipFileName, const char\* outputFolder) {

struct zip\* archive;

archive = zip\_open(zipFileName, ZIP\_RDONLY, NULL);

if (!archive) {

return;

}

int numEntries = zip\_get\_num\_entries(archive, 0);

if (numEntries < 0) {

zip\_close(archive);

return;

}

for (int i = 0; i < numEntries; i++) {

struct zip\_stat entryStat;

if (zip\_stat\_index(archive, i, 0, &entryStat) == 0) {

std::string entryName(entryStat.name);

std::string outputPath;

if (entryName.find("\\") == 0) {

outputPath = std::string(outputFolder) + entryStat.name;

}

else {

outputPath = std::string(outputFolder) + '\\' + entryStat.name;

}

// Ensure the directory exists

size\_t lastSlashPos = outputPath.find\_last\_of("\\");

if (lastSlashPos != std::string::npos) {

std::string dirPath = outputPath.substr(0, lastSlashPos);

SHCreateDirectoryExA(NULL, dirPath.c\_str(), NULL);

}

zip\_file\* zipFile = zip\_fopen\_index(archive, i, 0);

if (zipFile) {

FILE\* outputFile;

if (fopen\_s(&outputFile, outputPath.c\_str(), "wb") == 0) {

char buffer[1024];

int bytesRead;

while ((bytesRead = zip\_fread(zipFile, buffer, sizeof(buffer))) > 0) {

fwrite(buffer, 1, bytesRead, outputFile);

}

fclose(outputFile);

}

zip\_fclose(zipFile);

}

else if (entryStat.name[strlen(entryStat.name) - 1] == '\\') {

// This entry is a directory

CreateDirectoryA(outputPath.c\_str(), NULL);

}

}

}

zip\_close(archive);

}

bool PickZipToExtract()

{

HRESULT hr = CoInitializeEx(NULL, COINIT\_APARTMENTTHREADED | COINIT\_DISABLE\_OLE1DDE);

if (FAILED(hr)) {

MessageBox(NULL, L"COM Initialization Failed", L"Error", MB\_ICONERROR);

return false;

}

IFileDialog\* pFileDialog = nullptr;

hr = CoCreateInstance(CLSID\_FileOpenDialog, NULL, CLSCTX\_INPROC\_SERVER, IID\_PPV\_ARGS(&pFileDialog));

if (FAILED(hr)) {

MessageBox(NULL, L"Failed to create File Open Dialog", L"Error", MB\_ICONERROR);

CoUninitialize();

return false;

}

COMDLG\_FILTERSPEC fileTypes[] = { L"Zip Archives", L"\*.zip" };

pFileDialog->SetFileTypes(ARRAYSIZE(fileTypes), fileTypes);

hr = pFileDialog->Show(NULL);

if (SUCCEEDED(hr)) {

IShellItem\* pItem;

hr = pFileDialog->GetResult(&pItem);

if (SUCCEEDED(hr)) {

PWSTR filePath;

hr = pItem->GetDisplayName(SIGDN\_FILESYSPATH, &filePath);

if (SUCCEEDED(hr)) {

std::wstring selectedFilePath(filePath);

selectedZip = selectedFilePath;

CoTaskMemFree(filePath);

}

pItem->Release();

}

}

pFileDialog->Release();

CoUninitialize();

if (FAILED(hr)) {

MessageBox(NULL, L"File Dialog Operation Failed", L"Error", MB\_ICONERROR);

return false;

}

return true;

}

std::string WcharToString(const wchar\_t\* data)

{

std::wstring wideString(data);

return std::string(wideString.begin(), wideString.end());

}