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»

Выполнил:

студент гр. 153504

Хрищанович А.К.

Проверил:

Гриценко Н.Ю.

Минск 2023

СОДЕРЖАНИЕ

[1 Постановка задачи 3](#_Toc146631498)

[2 Краткие теоретические сведения 4](#_Toc146631499)

[3 Результаты выполнения лабораторной работы 6](#_Toc146631500)

[Выводы 8](#_Toc146631501)

[Список использованных источников 9](#_Toc146631502)

[Приложение А (обязательное) Листинг кода 10](#_Toc146631503)

## 1 ПОСТАНОВКА ЗАДАЧИ

Целью выполнения данной лабораторной работы является создание многозадачного приложения, использующего многопоточность для обработки большого объема данных в параллель.

## 2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Win32 API (Windows API) представляет собой набор функций и интерфейсов, предоставляемых операционной системой Windows для разработки приложений. Этот мощный набор инструментов обеспечивает доступ к различным функциональным возможностям Windows, включая создание и управление окнами, обработку сообщений, работу с файлами и реестром, а также многие другие операции. Win32 API играет ключевую роль в разработке приложений для Windows и обеспечивает высокую степень контроля над поведением приложений.

Управление процессами и потоками в операционной системе Windows является важной частью разработки многозадачных приложений.

Создание нового процесса или потока выполняется с использованием функций Windows API, таких как CreateProcess для процессов и CreateThread для потоков. При порождении нового процесса происходит выделение отдельного адресного пространства памяти, файловых ресурсов и других системных ресурсов.

Процессы и потоки могут завершаться по разным причинам. Можно использовать функции, такие как ExitProcess для завершения процесса и ExitThread для завершения потока. Кроме того, процесс или поток могут быть завершены системой из-за ошибок или других обстоятельств.[1]

Windows предоставляет различные уровни приоритетов для процессов и потоков, такие как высокий, выше среднего, нормальный, ниже среднего и низкий. Функция SetPriorityClass позволяет устанавливать приоритеты для процессов, а функция SetThreadPriority для потоков. Например, существует возможность повысить приоритет критически важных задач для обеспечения их выполнения вовремя.[2]

Для изучения эффективности приложения можно использовать возможность подсчета времени работы каждого алгоритма в миллисекундах.

Управление процессами и потоками в Windows – это важный аспект разработки приложений, позволяющий создавать эффективные и отзывчивые многозадачные приложения, а также обеспечивать их стабильную работу.

Для выполнения данной лабораторной работы, были использованы следующие теоретические сведения и концепции:

1. В главной программе (проекте) создаются три процесса с помощью функции CreateProcess(). Каждый процесс выполняет свою задачу, описанную в трех отдельных исполняемых файлах: Process1.exe, Process2.exe и Process3.exe.

2. Для каждого процесса создаются отдельные структуры STARTUPINFO и PROCESS\_INFORMATION, которые используются для запуска и отслеживания процессов.

3. В третьем процессе (Process3.exe) устанавливается приоритет с помощью функции SetPriorityClass(). Приоритет выставляется как ABOVE\_NORMAL\_PRIORITY\_CLASS, что означает повышенный приоритет выполнения для этого процесса.

4. Для каждого процесса ожидается его завершение с использованием функции WaitForSingleObject(). Это гарантирует, что главная программа будет ждать завершения всех трех процессов, прежде чем продолжить выполнение.

5. После завершения всех трех процессов закрываются их дескрипторы с помощью CloseHandle().

6. Каждый из трех процессов (Process1.exe, Process2.exe и Process3.exe) выполняет разные вычислительные задачи и собирает статистику о времени выполнения с использованием функций QueryPerformanceCounter() и QueryPerformanceFrequency(). Результаты и время выполнения выводятся в консоль.

## 3 РЕЗУЛЬТАТЫ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

В ходе выполнения лабораторной работы было реализовано простое консольное многозадачное приложение, которое использует многопоточность для обработки списка, состоящего из чисел от 1 до 1000000. Родительская программа порождает три процесса, каждый из которых имеет свою функциональность. Первый процесс ведет подсчет суммы чисел списка, второй находит его максимальный элемент, а третий – минимальный элемент. На момент запуска программы наиболее высокий приоритет установлен у третьего процесса. Результат работы программы показан на рисунке 3.1.

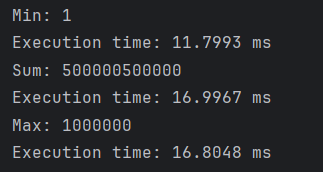


Рисунок 3.1 – Результат работы программы

Результат работы программы, при установке высшего приоритета у второго процесса, показан на рисунке 3.2.

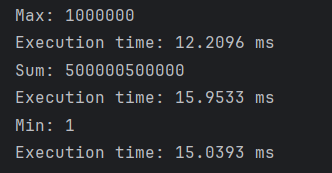


Рисунок 3.2 – Результат работы программы при высшем приоритете второго процесса

Результат работы программы без установки приоритетов показан на рисунке 3.3.

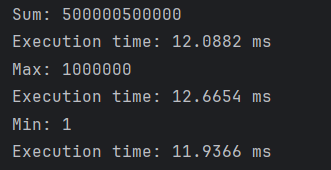


Рисунок 3.3 – Результат работы программы без установки приоритетов

## ВЫВОДЫ

В ходе выполнения данной лабораторной работы было создано простое многозадачное консольное приложение, которое выводит результат подсчетов трех процессов, работающих параллельно. Первый процесс подсчитывает сумму чисел списка, второй процесс находит максимальное число, третий процесс – минимальное число. Внутри кода можно изменить приоритет процессов. Так же в качестве исследования эффективности использован подсчет времени выполнения каждого процесса в миллисекундах для того, чтобы наглядно показать, сколько времени занимает прохождение каждого из алгоритма.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Щупак Ю. Win32 API. Разработка приложений для Windows. – СПб: Питер, 2008. – 592 с.: ип.
2. [Электронный ресурс]. – Режим доступа: [https://learn.microsoft.com/ru-ru/windows/win32/procthread/process-and-thread-functions – Дата доступа 08.10.2023](https://learn.microsoft.com/ru-ru/windows/win32/procthread/process-and-thread-functions%20–%20Дата%20доступа%2008.10.2023)
3. [Электронный ресурс]. – Режим доступа:  [[https://club.shelek.ru/viewart.php?id=71 – Дата доступа 10.10.2023](%20https://club.shelek.ru/viewart.php?id=71%20–%20Дата%20доступа%2010.10.2023)](https://stackoverflow.com/questions/17187265/how-to-group-radio-box-buttons-using-win32-api%20–%20Дата%20доступа%2023.09.2023)

## ПРИЛОЖЕНИЕ А

## (обязательное)

## Листинг кода

**Project4**

**main.cpp**

#include <vector>

#include <windows.h>

#include <iostream>

int main() {

// Создаем разные процессы для разных задач

STARTUPINFO si1, si2, si3;

PROCESS\_INFORMATION pi1, pi2, pi3;

ZeroMemory(&si1, sizeof(si1));

si1.cb = sizeof(si1);

ZeroMemory(&pi1, sizeof(pi1));

ZeroMemory(&si2, sizeof(si2));

si2.cb = sizeof(si2);

ZeroMemory(&pi2, sizeof(pi2));

ZeroMemory(&si3, sizeof(si3));

si3.cb = sizeof(si3);

ZeroMemory(&pi3, sizeof(pi3));

CreateProcess(NULL, // имя исполняемого файла

R"(D:\5 semester\C++\Process1\cmake-build-debug\Process1.exe)", // командная строка

NULL, // дескриптор процесса не наследуется

NULL, // дескриптор потока не наследуется

FALSE, // дескриптор процесса не наследуется

0, // флаги создания

NULL, // блок переменных окружения

NULL, // текущий каталог

&si1, // информация о запуске

&pi1); // информация о процессе

CreateProcess(NULL, // имя исполняемого файла

R"(D:\5 semester\C++\Process2\cmake-build-debug\Process2.exe)", // командная строка

NULL, // дескриптор процесса не наследуется

NULL, // дескриптор потока не наследуется

FALSE, // дескриптор процесса не наследуется

0, // флаги создания

NULL, // блок переменных окружения

NULL, // текущий каталог

&si2, // информация о запуске

&pi2); // информация о процессе

CreateProcess(NULL, // имя исполняемого файла

R"(D:\5 semester\C++\Process3\cmake-build-debug\Process3.exe)", // командная строка

NULL, // дескриптор процесса не наследуется

NULL, // дескриптор потока не наследуется

FALSE, // дескриптор процесса не наследуется

0, // флаги создания

NULL, // блок переменных окружения

NULL, // текущий каталог

&si3, // информация о запуске

&pi3); // информация о процессе

// УСТАНОВИТЬ ПРИОРИТЕТ ОТ 1 ДО 3

if (!SetPriorityClass(pi2.hProcess, ABOVE\_NORMAL\_PRIORITY\_CLASS)) {

std::cerr << "Failed to set priority." << std::endl;

}

// Ждем завершения процессов

WaitForSingleObject(pi1.hProcess, INFINITE);

WaitForSingleObject(pi2.hProcess, INFINITE);

WaitForSingleObject(pi3.hProcess, INFINITE);

// Закрываем дескрипторы процессов

CloseHandle(pi1.hProcess);

CloseHandle(pi2.hProcess);

CloseHandle(pi3.hProcess);

return 0;

}

**Process1**

**main.cpp**

#include <iostream>

#include <vector>

#include <windows.h>

DWORD WINAPI ThreadFunction(LPVOID lpParam) {

// Код для выполнения внутри потока

std::vector<int>\* numbers = static\_cast<std::vector<int>\*>(lpParam);

long long sum = 0;

for (int num : \*numbers) {

sum += num;

}

std::cout << "Sum: " << sum << std::endl;

return 0;

}

int main() {

// Измеряем начало выполнения участка кода

LARGE\_INTEGER startTime, endTime, frequency;

QueryPerformanceFrequency(&frequency);

QueryPerformanceCounter(&startTime);

const size\_t numNumbers = 1000000;

std::vector<int> numbers(numNumbers);

for (size\_t i = 0; i < numNumbers; ++i) {

numbers[i] = i + 1;

}

// Создаем поток

HANDLE hThread = CreateThread(NULL, 0, ThreadFunction, &numbers, 0, NULL);

// Дожидаемся завершения потока

WaitForSingleObject(hThread, INFINITE);

// Закрываем дескриптор потока

CloseHandle(hThread);

// Измеряем окончание выполнения участка кода

QueryPerformanceCounter(&endTime);

// Вычисляем время выполнения в миллисекундах

double executionTime = static\_cast<double>(endTime.QuadPart - startTime.QuadPart) / static\_cast<double>(frequency.QuadPart) \* 1000.0;

std::cout << "Execution time: " << executionTime << " ms" << std::endl;

return 0;

}

**Process2**

**main.cpp**

#include <iostream>

#include <vector>

#include <windows.h>

DWORD WINAPI ThreadFunction(LPVOID lpParam) {

// Код для выполнения внутри потока

std::vector<int>\* numbers = static\_cast<std::vector<int>\*>(lpParam);

int max = (\*numbers)[0];

for (int num : \*numbers) {

if (num > max) {

max = num;

}

}

std::cout << "Max: " << max << std::endl;

return 0;

}

int main() {

// Измеряем начало выполнения участка кода

LARGE\_INTEGER startTime, endTime, frequency;

QueryPerformanceFrequency(&frequency);

QueryPerformanceCounter(&startTime);

const size\_t numNumbers = 1000000;

std::vector<int> numbers(numNumbers);

for (size\_t i = 0; i < numNumbers; ++i) {

numbers[i] = i + 1;

}

// Создаем поток

HANDLE hThread = CreateThread(NULL, 0, ThreadFunction, &numbers, 0, NULL);

// Дожидаемся завершения потока

WaitForSingleObject(hThread, INFINITE);

// Закрываем дескриптор потока

CloseHandle(hThread);

// Измеряем окончание выполнения участка кода

QueryPerformanceCounter(&endTime);

// Вычисляем время выполнения в миллисекундах

double executionTime = static\_cast<double>(endTime.QuadPart - startTime.QuadPart) / static\_cast<double>(frequency.QuadPart) \* 1000.0;

std::cout << "Execution time: " << executionTime << " ms" << std::endl;

return 0;

}

**Process3**

**main.cpp**

#include <iostream>

#include <vector>

#include <windows.h>

DWORD WINAPI ThreadFunction(LPVOID lpParam) {

// Код для выполнения внутри потока

std::vector<int>\* numbers = static\_cast<std::vector<int>\*>(lpParam);

int min = (\*numbers)[0];

for (int num : \*numbers) {

if (num < min) {

min = num;

}

}

std::cout << "Min: " << min << std::endl;

return 0;

}

int main() {

// Измеряем начало выполнения участка кода

LARGE\_INTEGER startTime, endTime, frequency;

QueryPerformanceFrequency(&frequency);

QueryPerformanceCounter(&startTime);

const size\_t numNumbers = 1000000;

std::vector<int> numbers(numNumbers);

for (size\_t i = 0; i < numNumbers; ++i) {

numbers[i] = i + 1;

}

// Создаем поток

HANDLE hThread = CreateThread(NULL, 0, ThreadFunction, &numbers, 0, NULL);

// Дожидаемся завершения потока

WaitForSingleObject(hThread, INFINITE);

// Закрываем дескриптор потока

CloseHandle(hThread);

// Измеряем окончание выполнения участка кода

QueryPerformanceCounter(&endTime);

// Вычисляем время выполнения в миллисекундах

double executionTime = static\_cast<double>(endTime.QuadPart - startTime.QuadPart) / static\_cast<double>(frequency.QuadPart) \* 1000.0;

std::cout << "Execution time: " << executionTime << " ms" << std::endl;

return 0;

}