

САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО

КАФЕДРА КОМПЬЮТЕРНЫХ СИСТЕМ И ПРОГРАММНЫХ ТЕХНОЛОГИЙ

Отчёт по лабораторной работе №5

Курс: «Операционные системы»

Тема: «Управление процессами и потоками в Windows»

Выполнил студент:

Бояркин Никита Сергеевич

Группа: 43501/3

Проверил:

Душутина Елена Владимировна

Санкт-Петербург
2017 г.

Содержание

1	Лабораторная работа №5	2
1.1	Цель работы	2
1.2	Программа работы	2
1.3	Характеристики системы	3
1.4	Ход работы	3
1.4.1	Глава 1. Порождение и запуск процессов	3
1.4.2	Глава 2. Создание потоков	8
1.4.3	Глава 3. Функции управления приоритетами процессов и потоков	12
1.4.4	Глава 4. Самостоятельные задания	20
1.5	Вывод	26
1.6	Список литературы	26

Лабораторная работа №5

1.1 Цель работы

- Исследовать возможность создания нескольких процессов с базовым потоком и нескольких потоков в рамках одного процесса.
- Проанализировать выделение процессорного ресурса потокам с изменением их приоритетов.

1.2 Программа работы

Глава 1. Порождение и запуск процессов

1. Программа после запуска должна создать новый процесс, с помощью функции `CreateProcess`. В новом процессе необходимо запустить любое приложение (например, `notepad.exe` или `calc.exe`). Для контроля можно вывести идентификаторы созданного процесса и потока, а затем завершить основную программу.
2. Программа, получает имя конфигурационного файла из командной строки, открывает конфигурационный файл, читает строки и создает для запуска каждой команды отдельный процесс.
3. Программа получает имя конфигурационного файла из командной строки. После прочтения каждой строки, если она не пуста, создается процесс, в командную строку которого пишется прочитанная строка. Если создать процесс не удалось, программа пробует читать конфигурационный файл дальше.

Глава 2. Создание потоков

1. Программа должна создавать два потока, выводящих в бесконечном цикле «1» и «2» соответственно. После создания дополнительных потоков, поток-родитель завершается.
2. Программа должна получать 2 параметра – количество создаваемых потоков и время жизни всего приложения. С интервалом в 1 сек каждый рабочий поток выводит о себе информацию и отслеживает состояние переменной, которая устанавливается в заданное значение по истечении времени жизни процесса.

Глава 3. Функции управления приоритетами процессов и потоков

1. Подготовить программу, в которой у каждого из потоков свой приоритет отличный от других. Все они выполняют одинаковую работу, например, увеличивают каждый свой счетчик. Накопленное значение счетчика, таким образом, отражает относительное суммарное время выполнения потока.
2. Дополнение программы возможностью управления классом приоритета процесса.
3. С помощью программы определить, назначается ли динамическое изменение приоритетов по умолчанию, на все ли потоки воздействует функция `SetProcessPriorityBoost()`, возможно ли разрешение отдельному потоку в процессе динамически изменять приоритет, если для процесса это запрещено.

Глава 4. Самостоятельные задания

1. Занести экспериментальные данные из предыдущей главы в таблицу с точным указанием операционной системы и отладочного комплекса.
2. С помощью соответствующих утилит зафиксировать динамическое изменение приоритетов.
3. Создайте программу, демонстрирующую возможность наследования.

1.3 Характеристики системы

Некоторая информация об операционной системе и ресурсах системы:

```
Операционная система: Windows 10 Корпоративная 2016 с долгосрочным обслуживанием 64-разрядная (10.0,  
Язык: русский (формат: русский)  
Изготовитель компьютера: MSI  
Модель компьютера: MS-7885  
BIOS: Default System BIOS  
Процессор: Intel(R) Core(TM) i7-6800K CPU @ 3.40GHz (12 CPUs), ~3.4GHz  
Память: 16384MB RAM  
Файл подкачки: 6918 МБ использовано, 11798 МБ свободно  
Версия DirectX: DirectX 12
```

Рис. 1.1

Информация о компиляторе:

```
1 Оптимизирующий  
2 компилятор Microsoft (R) C/C++ версии 19.00.24215.1 для x86  
3 (C) Корпорация Майкрософт (Microsoft Corporation). Все права защищены.
```

Информация о компоновщике:

```
1 Microsoft (R) Incremental Linker Version 14.00.24215.1  
2 Copyright (C) Microsoft Corporation. All rights reserved.
```

1.4 Ход работы

1.4.1 Глава 1. Порождение и запуск процессов

Порождение и запуск процессов осуществляется функцией `CreateProcess`, которая создает новый процесс: выделяет новое адресное пространство и иные ресурсы процессора, создает базовый поток. Когда новый процесс будет создан, старый процесс будет продолжать исполняться, используя старое адресное пространство, а новый будет выполняться в новом адресном пространстве с новым базовым потоком. После того, как исполнительная система создала новый процесс, она возвращает его описатель, а также описатель его базового потока. Сигнатура функции `CreateProcess`:

```
BOOL WINAPI CreateProcess(  
    _In_opt_ LPCTSTR lpApplicationName, // Имя исполняемого модуля  
    _Inout_opt_ LPTSTR lpCommandLine, // Командная строка  
    _In_opt_ LPSECURITY_ATTRIBUTES lpProcessAttributes, // Атрибуты безопасности процесса  
    _In_opt_ LPSECURITY_ATTRIBUTES lpThreadAttributes, // Атрибуты безопасности потока  
    _In_ BOOL bInheritHandles, // Флаг наследования описателя  
    _In_ DWORD dwCreationFlags, // Флаги создания  
    _In_opt_ LPVOID lpEnvironment, // Новый блок окружения  
    _In_opt_ LPCTSTR lpCurrentDirectory, // Имя текущей директории  
    _In_ LPSTARTUPINFO lpStartupInfo, // Информация при запуске  
    _Out_ LPPROCESS_INFORMATION lpProcessInformation // Вывод информации о процессе  
);
```

Десять параметров функции `CreateProcess` обеспечивают большую гибкость при использовании программистом, в простейшем случае для многих параметров можно использовать значения по умолчанию.

- *lpApplicationName* и *lpCommandLine* используются вместе для указания исполняемой программы и аргументов командной строки.
- *lpProcessAttributes*, *lpThreadAttributes* и *bInheritHandles* первые два – указатели на атрибуты безопасности для процесса и потока соответственно, последний параметр – флаг наследования (наследуются ли файловые дескрипторы и т.д.).

- *DwCreationFlags* может объединять в себе несколько флаговых значений, включая следующие:
 - *CREATE_SUSPENDED* — указывает на то, что основной поток будет создан в приостановленном состоянии и начнет выполняться лишь после вызова функции *ResumeThread*.
 - *DETACHED_PROCESS* и *CREATE_NEW_CONSOLE* — взаимоисключающие значения, которые не должны устанавливаться оба одновременно. Первый флаг означает создание нового процесса, у которого консоль отсутствует, а второй — процесса, у которого имеется собственная консоль. Если ни один из этих флагов не указан, то новый процесс наследует консоль родительского процесса;
 - *CREATE_NEW_PROCESS_GROUP* — указывает на то, что создаваемый процесс является корневым для новой группы процессов. Если все процессы, принадлежащие данной группе, разделяют общую консоль, то все они будут получать управляющие сигналы консоли (Ctrl-C или Ctrl-break).
 - В качестве флагов так же могут быть указаны приоритеты: *HIGH_PRIORITY_CLASS*, *IDLE_PRIORITY_CLASS*, *NORMAL_PRIORITY_CLASS* или *REALTIME_PRIORITY_CLASS*. Значение по умолчанию - *NORMAL_PRIORITY_CLASS*, но если порождающий процесс имеет приоритет *IDLE_PRIORITY_CLASS*, то и процесс-потомок также будет иметь этот приоритет.
- *lpEnvironment* используется для передачи нового блока переменных окружения порожденному процесс-потомку. Если NULL, то потомок использует то же окружение, что и родитель. Если не NULL, то *lpEnvironment* должен указывать на массив строк, каждая name=value.
- *lpCurrentDirectory* определяет полное путевое имя директории, в которой потомок будет выполняться. Если использовать NULL, то потомок будет использовать директорию родителя.
- *lpStartupInfo* указатель на структуру *STARTUPINFO*, которая устанавливает оконный режим терминала, рабочий стол, стандартные дескрипторы и внешний вид главного окна для нового процесса.
- *lpProcessInformation* указатель на структуру *PROCESS_INFORMATION*, которая принимает идентифицирующую информацию о новом процессе.

Таким образом, используя функцию *CreateProcess* создаем новый процесс для запуска приложения. Для контроля работы функции выводим на консоль идентификаторы созданного процесса и потока.

1. Программа, создающая новый процесс

Создадим программу, которая запускает новый процесс калькулятора, после чего завершает свою работу:

```

1 #include <iostream>
2 #include <windows.h>
3 #include <tchar.h>
4
5 // Задержка перед завершением процесса
6 const int DELAY = 5 * 1000;
7 // Путь к запускаемой программе
8 const char* PATH_TO_CALC = "C:\\Windows\\System32\\calc.exe";
9
10 int main() {
11     // Универсальная форма строки с командой
12     LPTSTR commandLine = _tcsdup(TEXT(PATH_TO_CALC));
13
14     STARTUPINFO startupInfo;
15     ZeroMemory(&startupInfo, sizeof(startupInfo));
16     startupInfo.cb = sizeof(startupInfo);
17
18     // Информация о процессе будет( получена после создания процесса)
19     PROCESS_INFORMATION processInformation;
20     // Пробуем создать процесс
21     if(!CreateProcess(nullptr, commandLine, nullptr, nullptr, false, HIGH_PRIORITY_CLASS |
22         CREATE_NEW_CONSOLE, nullptr, nullptr, &startupInfo, &processInformation)) {
23         std::cerr << "It's impossible to create process." << std::endl;
24         return 0x1;
25     }
26
27     std::cout << "New process was created, process id " << processInformation.dwProcessId <<
28         ", thread id " << processInformation.dwThreadId << "." << std::endl;

```

```

28 // Задержка перед завершением процесса
29 Sleep(DELAY);
30
31 CloseHandle(processInformation.hThread);
32 CloseHandle(processInformation.hProcess);
33
34 std::cout << "Program finished." << std::endl;
35
36 std::getchar();
37 return 0x0;
38 }

```

Программа успешно запустила процесс калькулятора:

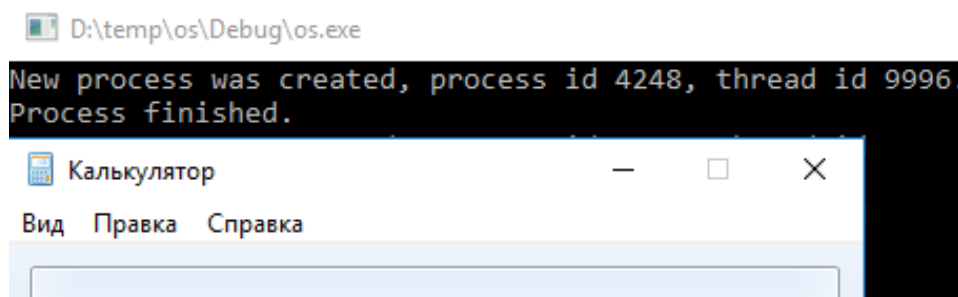


Рис. 1.2

2. Запуск процессов из конфигурационного файла

Программа открывает конфигурационный файл на чтение, построчно считывает содержимое и создает соответствующие процессы:

```

1 #include <iostream>
2 #include <fstream>
3 #include <string>
4 #include <windows.h>
5 #include <tchar.h>
6
7 // Путь к конфигурационному файлу
8 const char* PATH_TO_FILE = "D:\\ afiles\\student\\temp\\OS\\5\\p1.2.config";
9
10 int main() {
11     // Открываем конфигурационный файл на чтение
12     std::ifstream stream(PATH_TO_FILE);
13
14     // Если не удалось открыть файл
15     if(!stream.is_open()) {
16         std::cerr << "It's impossible to open configuration file." << std::endl;
17         return 0x1;
18     }
19
20     std::string line;
21     do {
22         line.clear();
23
24         // Считываем из файла построчно
25         std::getline(stream, line);
26         if(line.empty())
27             break;
28
29         // Универсальная форма строки с командой
30         LPTSTR commandLine = _tcsdup(TEXT(line.data()));
31
32         STARTUPINFO startupInfo;
33         ZeroMemory(&startupInfo, sizeof(startupInfo));

```

```

34     startupInfo.cb = sizeof(startupInfo);
35
36     // Информация о процессе будет( получена после создания процесса)
37     PROCESS_INFORMATION processInformation;
38     // Попробуем создать процесс
39     if (!CreateProcess(nullptr, commandLine, nullptr, nullptr, false, HIGH_PRIORITY_CLASS
40         | CREATE_NEW_CONSOLE, nullptr, nullptr, &startupInfo, &processInformation)) {
41         std::cerr << "It's impossible to create process." << std::endl;
42         continue;
43     }
44
45     std::cout << "New process was created, process id " << processInformation.dwProcessId
46     << ", thread id " << processInformation.dwThreadId << "." << std::endl;
47
48     CloseHandle(processInformation.hThread);
49     CloseHandle(processInformation.hProcess);
50 } while (!stream.eof());
51
52 // Очищаем считывающий поток
53 stream.close();
54
55 std::getchar();
56 return 0;
57 }

```

Содержимое конфигурационного файла:

```

1 C:\Windows\System32\calc.exe
2 C:\Windows\System32\notepad.exe
3 C:\Windows\System32\Taskmgr.exe

```

Результат работы программы:

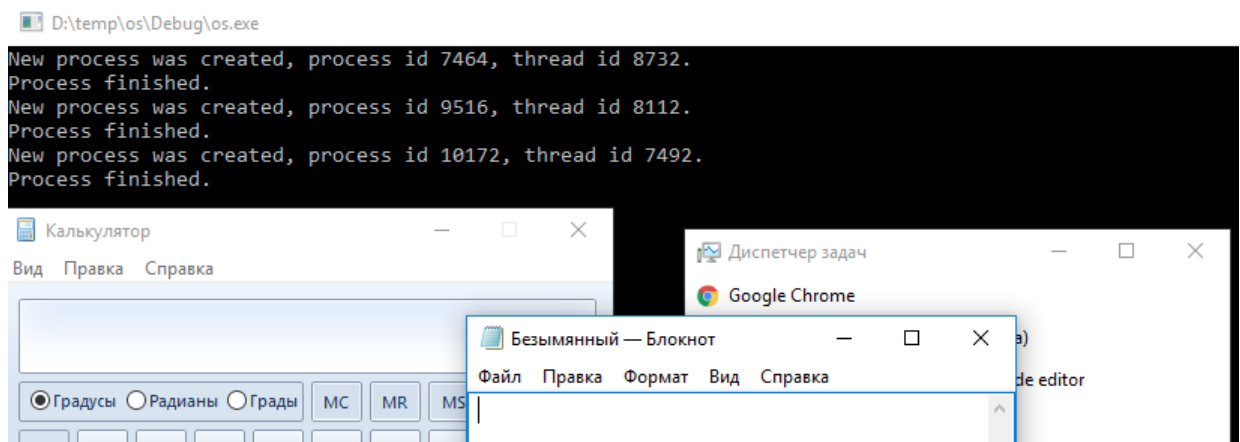


Рис. 1.3

Все три процесса, путь к которым был указан в конфигурационном файле были успешно запущены.

3. Доработка программы с игнорированием пустых строк и обработкой ошибок

Доработаем программу следующим образом: название конфигурационного файла теперь считывается из аргумента командной строки, программа продолжает работать при пустых строках пропуская их, если невозможно открыть процесс, то выводится сообщение об ошибке:

```

1 #include <iostream>
2 #include <fstream>
3 #include <string>
4 #include <windows.h>
5 #include <tchar.h>
6
7 int main(int argc, char** argv) {

```

```

8  if(argc < 2) {
9      std::cerr << "Wrong count of arguments." << std::endl;
10     return 0x1;
11 }
12
13 // Открываем конфигурационный файл на чтение
14 std::ifstream stream(argv[1]);
15
16 // Если не удалось открыть файл
17 if(!stream.is_open()) {
18     std::cerr << "It's impossible to open configuration file." << std::endl;
19     return 0x2;
20 }
21
22 std::string line;
23 do {
24     line.clear();
25
26     // Считываем из файла построчно
27     std::getline(stream, line);
28     if(line.empty()) {
29         std::cerr << "Skip empty line." << std::endl;
30         continue;
31     }
32
33     // Универсальная форма строки с командой
34     LPTSTR commandLine = _tcsdup(TEXT(line.data()));
35
36     STARTUPINFO startupInfo;
37     ZeroMemory(&startupInfo, sizeof(startupInfo));
38     startupInfo.cb = sizeof(startupInfo);
39
40     // Информация о процессе будет( получена после создания процесса)
41     PROCESS_INFORMATION processInformation;
42     // Пробуем создать процесс
43     if (!CreateProcess(nullptr, commandLine, nullptr, nullptr, false, HIGH_PRIORITY_CLASS
44         | CREATE_NEW_CONSOLE, nullptr, nullptr, &startupInfo, &processInformation)) {
45         std::cerr << "It's impossible to create process." << std::endl;
46         continue;
47     }
48
49     std::cout << "New process was created, process id " << processInformation.dwProcessId
50     << ", thread id " << processInformation.dwThreadId << "." << std::endl;
51
52     CloseHandle(processInformation.hThread);
53     CloseHandle(processInformation.hProcess);
54 } while(!stream.eof());
55
56 // Очищаем считывающий поток
57 stream.close();
58
59 std::getchar();
60 return 0x0;
61 }

```

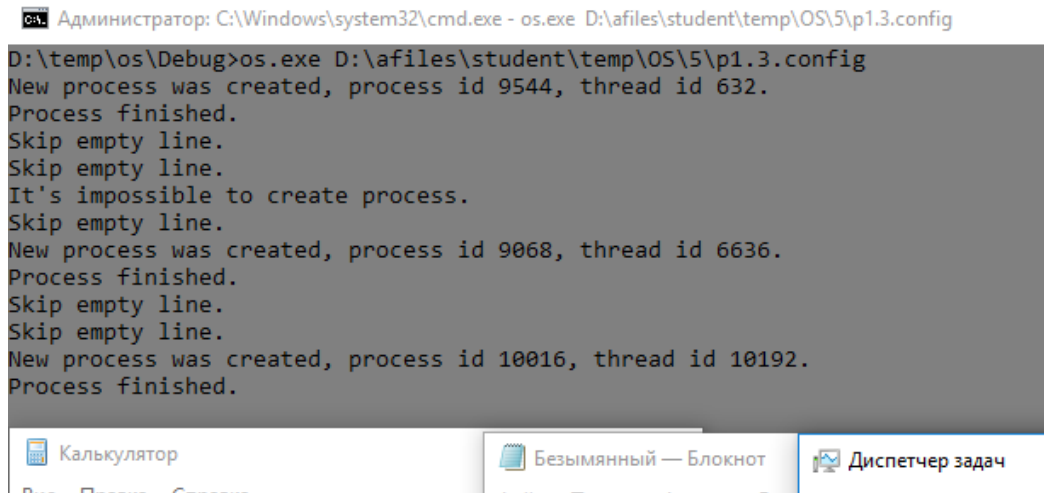
Содержимое конфигурационного файла с пропусками строк и неправильным путем:

```

1 C:\Windows\System32\calc.exe
2
3
4 C:\Windows\System32\IMPOSIBRU.exe
5
6 C:\Windows\System32\notepad.exe
7
8
9 C:\Windows\System32\Taskmgr.exe

```


Результат работы программы:



```
Администратор: C:\Windows\system32\cmd.exe - os.exe D:\afiles\student\temp\OS\5\p1.3.config
D:\temp\os\Debug>os.exe D:\afiles\student\temp\OS\5\p1.3.config
New process was created, process id 9544, thread id 632.
Process finished.
Skip empty line.
Skip empty line.
It's impossible to create process.
Skip empty line.
New process was created, process id 9068, thread id 6636.
Process finished.
Skip empty line.
Skip empty line.
New process was created, process id 10016, thread id 10192.
Process finished.
```

Рис. 1.4

Первый процесс калькулятора был успешно создан, после этого две пустых строки были пропущены. Далее был указан неправильный путь к файлу и была выведена ошибка. Остальные строки были обработаны схожим образом.

1.4.2 Глава 2. Создание потоков

Создание потоков производится посредством функции WinAPI `CreateThread`. Функция `CreateThread` имеет следующие аргументы:

```
HANDLE WINAPI CreateThread(
    _In_opt_ LPSECURITY_ATTRIBUTES lpThreadAttributes, // Указатель на структуру с атрибутами
    _In_     SIZE_T dwStackSize, // Размер стека нового потока в байтах
    _In_     LPTHREAD_START_ROUTINE lpStartAddress, // Указатель на функцию обработчика
    _In_opt_ LPVOID lpParameter, // Аргумент функции обработчика
    _In_     DWORD dwCreationFlags, // Флаг запуска
    _Out_opt_ LPDWORD lpThreadId // Получение идентификатора потока
);
```

- *lpThreadAttributes* - указатель на структуру с атрибутами защиты.
- *dwStackSize* - размер стека нового потока в байтах. Значению 0 этого параметра соответствует размер стека по умолчанию, равный размеру стека основного потока.
- *lpStartAddress* - указатель на функцию (принадлежащую контексту процесса), которая должна выполняться. Эта функция принимает единственный аргумент в виде указателя и возвращает 32-битовый код завершения. Этот аргумент может интерпретироваться потоком либо как переменная типа `DWORD`, либо как указатель.
- *lpParameter* - аргумент функции обработчика потока.
- *dwCreationFlags* - если значение этого параметра установлено равным 0, то поток запускается сразу же после вызова функции `CreateThread`. Установка значения `CREATE_SUSPENDED` приведет к запуску потока в приостановленном состоянии, из которого поток может быть переведен в состояние готовности путем вызова функции `ResumeThread`.
- *lpThreadId* - указатель на переменную типа `DWORD`, которая получает идентификатор нового потока. Если `NULL`, то идентификатор не возвращается. Если функция выполнялась успешно, то вернется описатель потока, если нет, вернется `NULL`.

1. Программа, создающая два потока

Разработаем программу, которая создает два потока, выводящие в бесконечном цикле цифры 1 и 2. Цифры передаются через атрибут функции обработчика:

```

1 #include <iostream>
2 #include <windows.h>
3
4 // Задержка перед каждым выводом потока в( миллисекундах)
5 const int DELAY = 500;
6
7 // Обработчик потока
8 DWORD WINAPI threadExecutor(LPVOID);
9
10 int main() {
11     std::cout << "Press \"Enter\" to exit." << std::endl;
12
13     // Число для передачи в поток идентификатор( первого потока)
14     static const int firstNumber = 1;
15     // Создаем первый поток
16     HANDLE thread = CreateThread(nullptr, NULL, threadExecutor, LPVOID(&firstNumber), NULL,
17         nullptr);
18     // Закрываем дескриптор потока не( завершает поток)
19     CloseHandle(thread);
20
21     // Число для передачи в поток идентификатор( второго потока)
22     static const int secondNumber = 2;
23     // Создаем первый поток
24     thread = CreateThread(nullptr, NULL, threadExecutor, LPVOID(&secondNumber), NULL,
25         nullptr);
26     // Закрываем дескриптор потока не( завершает поток)
27     CloseHandle(thread);
28
29     // Завершаем программу по нажатию
30     std::getchar();
31     return 0x0;
32 }
33
34 // Обработчик потока
35 DWORD WINAPI threadExecutor(LPVOID ptr) {
36     // Указатель на идентификатор потока был передан как параметр функции
37     const int threadId = *static_cast<int*>(ptr);
38
39     while(true) {
40         // Вывод организован функцией printf, так как потоки в C++ вызывают проблемы с синхронизацией
41         // изза endl
42         printf("%d\n", threadId);
43         // Задержка перед следующим выводом
44         Sleep(DELAY);
45     }
46 }

```

Результат работы программы:

```

1 Press "Enter" to exit.
2 1
3 2
4 1
5 2
6 2
7 1
8 2
9 1
10 1
11 2
12 2
13 1
14 1
15 2
16 2
17 1

```

18 2
19 1

Функция Sleep позволяет потоку отказаться от использования процессора и перейти из состояния выполнения в состояние ожидания, которое будет длиться в течение заданного промежутка времени. Например, выполнение задачи потоком может продолжаться в течение некоторого периода времени, после чего поток приостанавливается. По истечении периода ожидания планировщик вновь переводит поток в состояние готовности.

2. Программа, время жизни которой определяется параметром

Реализуем поставленную задачу с помощью таймера ожидания. Таймеры ожидания(waitable timers) – это объекты ядра, которые самостоятельно переходят в свободное состояние в определенное время или через регулярные промежутки времени. Чтобы создать ожидаемый таймер, достаточно вызвать функцию CreateWaitableTimer. Объекты таймера всегда создаются в занятом состоянии. Чтобы сообщить таймеру, в какой момент он должен перейти в свободное состояние, необходимо вызвать функцию SetWaitableTimer.

Программа получает два параметра аргументами командной строки: количество создаваемых потоков и время жизни приложения. С определенным интервалом каждый рабочий поток выводит свой идентификатор.

```
1 #include <iostream>
2 #include <windows.h>
3 #include <string>
4
5 // Аргумент для функции обработчика потока
6 struct Argument {
7     int number;
8     bool* interrupt;
9 };
10
11 // Количество создаваемых потоков по умолчанию
12 static const int DEFAULT_COUNT_OF_THREADS = 3;
13 // Время до завершения потока в( секундах)
14 static const int DEFAULT_TIME_TO_STOP = 5;
15 // Константа для таймера
16 static const __int64 TIMER_CONSTANT = -1 * 10000000;
17 // Задержка перед каждым выводом потока в( миллисекундах)
18 static const int DELAY = 1000;
19
20 // Обработчик потока
21 DWORD WINAPI threadExecutor(LPVOID);
22 // Вывод текущего времени
23 void printCurrentTime();
24
25 int main(int argc, char** argv) {
26     int countOfThreads = DEFAULT_COUNT_OF_THREADS;
27     int timeToStop = DEFAULT_TIME_TO_STOP;
28
29     // Получение аргументов командной строки
30     if(argc > 2) {
31         try {
32             countOfThreads = std::stoi(argv[1]);
33             timeToStop = std::stoi(argv[2]);
34         } catch(const std::invalid_argument& exception) {
35             std::cerr << exception.what() << std::endl;
36             return 0x1;
37         }
38     }
39
40     std::cout << "Count of threads – " << countOfThreads << ", time to stop – " <<
        timeToStop << "." << std::endl;
41
42     // Создаем таймер
43     HANDLE timer = CreateWaitableTimer(nullptr, false, nullptr);
44
45     // Устанавливаем временные характеристики таймера
46     __int64 endTimeValue = TIMER_CONSTANT * timeToStop;
```

```

47 LARGE_INTEGER endTimeStruct;
48 endTimeStruct.LowPart = DWORD(endTimeValue & 0xFFFFFFFF);
49 endTimeStruct.HighPart = LONG(endTimeValue >> 0x20);
50 SetWaitableTimer(timer, &endTimeStruct, NULL, nullptr, nullptr, false);
51
52 printCurrentTime();
53
54 bool interrupt = false;
55 for(int index = 0; index < countOfThreads; ++index) {
56     // Создаем указатель на структуру аргумента
57     Argument* argument = new Argument;
58     argument->number = index;
59     argument->interrupt = &interrupt;
60
61     // Создаем поток
62     HANDLE thread = CreateThread(nullptr, NULL, threadExecutor, LPVOID(argument), NULL,
63     nullptr);
64     CloseHandle(thread);
65 }
66
67 // Ожидаем таймер
68 WaitForSingleObject(timer, INFINITE);
69 CloseHandle(timer);
70 // Посылаем прерывание всем потокам
71 interrupt = true;
72
73 printCurrentTime();
74
75 printf("Press \"Enter\" to exit.\n");
76 std::getchar();
77
78 return 0x0;
79 }
80
81 void printCurrentTime() {
82     // Выводим текущее время
83     SYSTEMTIME now;
84     GetLocalTime(&now);
85     printf("Current local time %.2d:%.2d:%.2d\n", now.wHour, now.wMinute, now.wSecond);
86 }
87
88 // Обработчик потока
89 DWORD WINAPI threadExecutor(LPVOID ptr) {
90     // Указатель на идентификатор потока был передан как параметр функции
91     const Argument argument = *static_cast<Argument*>(ptr);
92
93     printf("Thread %d started.\n", argument.number);
94
95     while(!(*argument.interrupt)) {
96         // Вывод организован функцией printf, так как потоки в C++ вызывают проблемы с синхронизацией
97         // изза endl
98         printf("%d\n", argument.number);
99         // Задержка перед следующим выводом
100         Sleep(DELAY);
101     }
102
103     return 0x0;
104 }

```

Результат работы программы по умолчанию:

```

1 Count of threads – 3, time to stop – 5.
2 Current local time 06:56:47
3 Thread 0 started.
4 Thread 1 started.
5 1
6 0

```

```

7 Thread 2 started .
8 2
9 0
10 1
11 2
12 0
13 1
14 2
15 0
16 1
17 2
18 0
19 1
20 2
21 Current local time 06:56:52
22 Press "Enter" to exit .

```

Результат работы программы с заданным количеством потоков и временем жизни:

```

1 D:\temp\os\Debug>os.exe 5 6
2 Count of threads – 5, time to stop – 6.
3 Current local time 06:58:28
4 Thread 0 started .
5 0
6 Thread 1 started .
7 Thread 2 started .
8 Thread 3 started .
9 Thread 4 started .
10 4
11 2
12 3
13 1
14 3
15 4
16 1
17 0
18 2
19 3
20 2
21 4
22 1
23 0
24 3
25 2
26 4
27 1
28 0
29 3
30 4
31 2
32 0
33 1
34 3
35 4
36 2
37 0
38 1
39 Current local time 06:58:34
40 Press "Enter" to exit .

```

1.4.3 Глава 3. Функции управления приоритетами процессов и потоков

Windows поддерживает шесть классов приоритетов процессов:

- *real-time* - наивысший возможный приоритет. Потоки в этом процессе обязаны немедленно реагировать на события, их исполнение может привести к полной блокировке системы и требует осторожности в

использовании этого класса.

- *high* - потоки быстрого реагирования на события.
- *above normal* - класс приоритета промежуточный между *normal* и *high*, введенный в версии Windows 2000.
- *normal* - потоки в этом процессе не предъявляют особых требований к выделению им процессорного времени.
- *below normal* - класс приоритета промежуточный между *normal* и *idle*, введенный в Windows 2000.
- *idle* - потоки в этом процессе выполняются, когда система не занята другой работой. Этот класс приоритета обычно используется для утилит, работающих в фоновом режиме.

Кроме того, Windows поддерживает семь относительных приоритетов потока:

- *time-critical* - поток выполняется с приоритетом 31 в классе real-time и с приоритетом 15 в других классах.
- *highest* - поток выполняется с приоритетом на два уровня выше обычного для данного класса.
- *above normal* - поток выполняется с приоритетом на один уровень выше обычного для данного класса.
- *normal* - поток выполняется с обычным приоритетом процесса для данного класса.
- *below normal* - поток выполняется с приоритетом на один уровень ниже обычного для данного класса.
- *lowest* - поток выполняется с приоритетом на два уровня ниже обычного для данного класса.
- *idle* - Поток выполняется с приоритетом 16 в классе real-time и с приоритетом 1 в других классах.

Относительный приоритет потока принимает значение от 0 (самый низкий) до 31 (самый высокий), но программист работает не с численными значениями, а с так называемыми «константными». Это обеспечивает определенную гибкость и независимость при изменении алгоритмов планирования, а они меняются практически с каждой новой версией ОС, а с ними, соответственно, могут измениться и соотношения приоритетов. Уровень приоритета формируется самой системой, исходя из класса приоритета процесса и относительного приоритета потока.

Динамическое повышение приоритета предназначено для оптимизации общей пропускной способности и реактивности системы, при этом выигрывает не каждое приложение в отдельности, а система в целом. Windows может динамически повышать значение текущего приоритета потока в одном из следующих случаев:

- После завершения операции ввода/вывода ОС временно динамически повышает приоритет потоков, предоставляя им больше шансов возобновить выполнение и обработать полученные данные. После динамического повышения приоритета поток в течение одного кванта выполняется с этим приоритетом. Следующий квант потоку выделяется с понижением приоритета на один уровень. Этот цикл продолжается до тех пор, пока приоритет не снизится до базового.
- По окончании ожидания на каком-либо объекте исполнительной системы (например, SetEvent, ReleaseSemaphore) приоритет потока увеличивается на один уровень.
- При инверсии приоритетов диспетчер настройки баланса просматривает очереди готовых потоков и ищет потоки, которые находились в состоянии готовности (Ready) более 3 секунд. Обнаружив такой поток, диспетчер повышает его приоритет до 15 и выделяет ему квант вдвое больше обычного. По истечении двух квантов приоритет потока снижается до исходного уровня.

Система повышает приоритет только тех потоков, базовый приоритет которых попадает в область динамического приоритета (dynamic priority range), т.е. находится в пределах 1-15. ОС не допускает динамического повышения приоритета прикладного потока до уровней реального времени (выше 15).

1. Программа с семью потоками с разными приоритетами

Каждый поток процесса имеет различный приоритет и инкриминирует собственный счетчик:

```
1 #include <iostream>
2 #include <windows.h>
3 #include <string>
4 #include <iomanip>
5
6 // Аргумент для функции обработчика потока
7 struct Argument {
8     int number;
9     bool* interrupt;
10 };
11
12 static const int COUNT_OF_PRIORITIES = 7;
13 // Количество создаваемых потоков по умолчанию
14 static const int DEFAULT_COUNT_OF_THREADS = COUNT_OF_PRIORITIES;
15 // Время до завершения потока в( секундах)
16 static const int DEFAULT_TIME_TO_STOP = 5;
17 // Константа для таймера
18 static const __int64 TIMER_CONSTANT = -1 * 10000000;
19
20 // Счетчики увеличиваемые каждым потоком
21 static __int64 counter[COUNT_OF_PRIORITIES] = { 0, 0, 0, 0, 0, 0, 0 };
22 // Значения приоритетов для каждого из потоков
23 static const int priority[COUNT_OF_PRIORITIES] = { THREAD_PRIORITY_IDLE,
24     THREAD_PRIORITY_LOWEST, THREAD_PRIORITY_BELOW_NORMAL, THREAD_PRIORITY_NORMAL,
25     THREAD_PRIORITY_ABOVE_NORMAL, THREAD_PRIORITY_HIGHEST, THREAD_PRIORITY_TIME_CRITICAL };
26 // Текстовые строки приоритетов для вывода
27 static const char* string[COUNT_OF_PRIORITIES] = { "THREAD_PRIORITY_IDLE", "
28     THREAD_PRIORITY_LOWEST", "THREAD_PRIORITY_BELOW_NORMAL", "THREAD_PRIORITY_NORMAL",
29     "THREAD_PRIORITY_ABOVE_NORMAL", "THREAD_PRIORITY_HIGHEST", "THREAD_PRIORITY_TIME_CRITICAL
30     " };
31
32 // Обработчик потока
33 DWORD WINAPI threadExecutor(LPVOID);
34
35 int main() {
36     std::cout << "Count of threads - " << DEFAULT_COUNT_OF_THREADS << ", time to stop - "
37         << DEFAULT_TIME_TO_STOP << "." << std::endl;
38
39     // Создаем таймер
40     HANDLE timer = CreateWaitableTimer(nullptr, false, nullptr);
41
42     // Устанавливаем временные характеристики таймера
43     __int64 endTimeValue = TIMER_CONSTANT * DEFAULT_TIME_TO_STOP;
44     LARGE_INTEGER endTimeStruct;
45     endTimeStruct.LowPart = DWORD(endTimeValue & 0xFFFFFFFF);
46     endTimeStruct.HighPart = LONG(endTimeValue >> 0x20);
47     SetWaitableTimer(timer, &endTimeStruct, NULL, nullptr, nullptr, false);
48
49     bool interrupt = false;
50     for(int index = 0; index < DEFAULT_COUNT_OF_THREADS; ++index) {
51         // Создаем указатель на структуру аргумента
52         Argument* argument = new Argument;
53         argument->number = index;
54         argument->interrupt = &interrupt;
55
56         // Создаем поток
57         HANDLE thread = CreateThread(nullptr, NULL, threadExecutor, LPVOID(argument), NULL,
58             nullptr);
59         // Задаем приоритет для каждого потока
60         SetThreadPriority(thread, priority[index]);
61         SetThreadPriorityBoost(thread, true);
62         CloseHandle(thread);
63     }
64 }
```

```

59
60 // Ожидаем таймер
61 WaitForSingleObject(timer, INFINITE);
62 CloseHandle(timer);
63 // Посылаем прерывание всем потокам
64 interrupt = true;
65
66 for(int index = 0; index < countOfThreads; ++index)
67     std::cout << std::left << std::setw(30) << std::setfill(' ') << string[index] << " -
    " <<
68     std::right << std::setw(30) << std::setfill(' ') << counter[index] << std
    ::endl;
69
70 printf("Press \"Enter\" to exit.\n");
71 std::getchar();
72
73 return 0x0;
74 }
75
76 // Обработчик потока
77 DWORD WINAPI threadExecutor(LPVOID ptr) {
78     // Указатель на идентификатор потока был передан как параметр функции
79     const Argument argument = *static_cast<Argument*>(ptr);
80
81     // Увеличиваем счетчик пока не пришло прерывание
82     while(!(*argument.interrupt))
83         ++counter[argument.number];
84
85     return 0x0;
86 }

```

Результат работы программы:

```

1 D:\temp\os\Debug>os.exe
2 Count of threads - 7, time to stop - 5.
3 THREAD_PRIORITY_IDLE - 1073623413
4 THREAD_PRIORITY_LOWEST - 920194997
5 THREAD_PRIORITY_BELOW_NORMAL - 1062454130
6 THREAD_PRIORITY_NORMAL - 1229730847
7 THREAD_PRIORITY_ABOVE_NORMAL - 1126257278
8 THREAD_PRIORITY_HIGHEST - 2117912869
9 THREAD_PRIORITY_TIME_CRITICAL - 2142065888
10 Press "Enter" to exit.

```

Можно заметить, что счетчики в общем случае увеличиваются с увеличением приоритета потока, что говорит о том, что процессорный ресурс действительно сильно привязан к приоритетам.

Результат работы программы при переводе ресурсов процессора на одно ядро:

```

1 D:\temp\os\Debug>start /affinity 1 os.exe
2 Count of threads - 7, time to stop - 5.
3 THREAD_PRIORITY_IDLE - 0
4 THREAD_PRIORITY_LOWEST - 0
5 THREAD_PRIORITY_BELOW_NORMAL - 0
6 THREAD_PRIORITY_NORMAL - 0
7 THREAD_PRIORITY_ABOVE_NORMAL - 0
8 THREAD_PRIORITY_HIGHEST - 0
9 THREAD_PRIORITY_TIME_CRITICAL - 2528652903
10 Press "Enter" to exit.

```

Как видно, при работе на одном процессоре, процессорный ресурс получает только поток с наивысшим приоритетом. Остальные потоки не выполняются.

2. Доработанная программа

Модифицируем программу, добавив возможность выбирать количество потоков и время жизни приложения. Также будем отслеживать изменился ли приоритет потока во время работы приложения.


```

1 #include <iostream>
2 #include <windows.h>
3 #include <string>
4 #include <iomanip>
5
6 // Аргумент для функции обработчика потока
7 struct Argument {
8     int number;
9     bool* interrupt;
10 };
11
12 static const int COUNT_OF_PRIORITIES = 7;
13 // Количество создаваемых потоков по умолчанию
14 static const int DEFAULT_COUNT_OF_THREADS = COUNT_OF_PRIORITIES;
15 // Время до завершения потока в( секундах)
16 static const int DEFAULT_TIME_TO_STOP = 5;
17 // Константа для таймера
18 static const __int64 TIMER_CONSTANT = -1 * 10000000;
19
20 // Счетчики увеличиваемые каждым потоком
21 static __int64 counter[COUNT_OF_PRIORITIES] = { 0, 0, 0, 0, 0, 0, 0 };
22 // Если был буст
23 static int boost[COUNT_OF_PRIORITIES] = { 0, 0, 0, 0, 0, 0, 0 };
24 // Показывает, изменился ли приоритет в процессе
25 static bool change[COUNT_OF_PRIORITIES] = { false, false, false, false, false, false, false };
26 // Значения приоритетов для каждого из потоков
27 static const int priority[COUNT_OF_PRIORITIES] = { THREAD_PRIORITY_IDLE,
28     THREAD_PRIORITY_LOWEST, THREAD_PRIORITY_BELOW_NORMAL, THREAD_PRIORITY_NORMAL,
29     THREAD_PRIORITY_ABOVE_NORMAL, THREAD_PRIORITY_HIGHEST, THREAD_PRIORITY_TIME_CRITICAL };
30 // Текстовые строки приоритетов для вывода
31 static const char* string[COUNT_OF_PRIORITIES] = { "THREAD_PRIORITY_IDLE", "
32     THREAD_PRIORITY_LOWEST", "THREAD_PRIORITY_BELOW_NORMAL", "THREAD_PRIORITY_NORMAL",
33     "THREAD_PRIORITY_ABOVE_NORMAL", "THREAD_PRIORITY_HIGHEST", "THREAD_PRIORITY_TIME_CRITICAL
34     " };
35
36 // Обработчик потока
37 DWORD WINAPI threadExecutor(LPVOID);
38
39 int main(int argc, char** argv) {
40     int countOfThreads = DEFAULT_COUNT_OF_THREADS;
41     int timeToStop = DEFAULT_TIME_TO_STOP;
42
43     // Получение аргументов командной строки
44     if(argc > 2) {
45         try {
46             countOfThreads = std::stoi(argv[1]);
47             timeToStop = std::stoi(argv[2]);
48             if(countOfThreads > DEFAULT_COUNT_OF_THREADS)
49                 return 0x1;
50         }
51         catch(const std::invalid_argument& exception) {
52             std::cerr << exception.what() << std::endl;
53             return 0x2;
54         }
55     }
56
57     std::cout << "Count of threads – " << countOfThreads << ", time to stop – " <<
58         timeToStop << "." << std::endl;
59
60     // Создаем таймер
61     HANDLE timer = CreateWaitableTimer(nullptr, false, nullptr);
62
63     // Устанавливаем временные характеристики таймера
64     __int64 endTimeValue = TIMER_CONSTANT * timeToStop;

```

```

61 LARGE_INTEGER endTimeStruct;
62 endTimeStruct.LowPart = DWORD(endTimeValue & 0xFFFFFFFF);
63 endTimeStruct.HighPart = LONG(endTimeValue >> 0x20);
64 SetWaitableTimer(timer, &endTimeStruct, NULL, nullptr, nullptr, false);
65
66 // Устанавливаем приоритет реального времени
67 SetPriorityClass(GetCurrentProcess(), REALTIME_PRIORITY_CLASS);
68
69 bool interrupt = false;
70 for(int index = 0; index < countOfThreads; ++index) {
71     // Создаем указатель на структуру аргумента
72     Argument* argument = new Argument;
73     argument->number = index;
74     argument->interrupt = &interrupt;
75
76     // Создаем поток
77     HANDLE thread = CreateThread(nullptr, NULL, threadExecutor, LPVOID(argument), NULL,
78     nullptr);
79     // Задаем приоритет для каждого потока
80     SetThreadPriority(thread, priority[index]);
81     SetThreadPriorityBoost(thread, true);
82     GetThreadPriorityBoost(thread, &boost[index]);
83     CloseHandle(thread);
84 }
85
86 // Ожидаем таймер
87 WaitForSingleObject(timer, INFINITE);
88 CloseHandle(timer);
89 // Посылаем прерывание всем потокам
90 interrupt = true;
91
92 // Выводим в виде таблицы
93 std::cout << std::left << std::setw(32) << std::setfill(' ') << "Priority" <<
94     std::left << std::setw(7) << std::setfill(' ') << "Boost" <<
95     std::left << std::setw(7) << std::setfill(' ') << "Changed" <<
96     std::right << std::setw(12) << std::setfill(' ') << "Counter" <<
97     std::endl;
98 for(int index = 0; index < countOfThreads; ++index)
99     std::cout << std::left << std::setw(32) << std::setfill(' ') << string[index] <<
100     std::left << std::setw(7) << std::setfill(' ') << ((boost[index] == 0) ? "No"
101     : "Yes") <<
102     std::left << std::setw(7) << std::setfill(' ') << ((!change[index]) ? "No" : "
103     Yes") <<
104     std::right << std::setw(12) << std::setfill(' ') << counter[index] <<
105     std::endl;
106
107 printf("Press \"Enter\" to exit.\n");
108 std::getchar();
109
110 return 0x0;
111 }
112
113 // Обработчик потока
114 DWORD WINAPI threadExecutor(LPVOID ptr) {
115     // Указатель на идентификатор потока был передан как параметр функции
116     const Argument argument = *static_cast<Argument*>(ptr);
117
118     // Увеличиваем счетчик пока не пришло прерывание
119     while(!(*argument.interrupt)) {
120         // Получаем текущий приоритет
121         int currentPriority = GetThreadPriority(GetCurrentThread());
122         // Если приоритет поменялся
123         if(currentPriority != priority[argument.number])
124             change[argument.number] = true;
125
126         ++counter[argument.number];
127     }
128 }

```

```

124 }
125
126 return 0x0;
127 }

```

Результат работы программы:

```

1 D:\temp\os\Debug>os.exe 7 8
2 Count of threads – 7, time to stop – 8.
3 Priority          Boost   Changed   Counter
4 THREAD_PRIORITY_IDLE      Yes    No       11675197
5 THREAD_PRIORITY_LOWEST    Yes    No       18851680
6 THREAD_PRIORITY_BELOW_NORMAL Yes    No       18885593
7 THREAD_PRIORITY_NORMAL    Yes    No       19039509
8 THREAD_PRIORITY_ABOVE_NORMAL Yes    No       16039789
9 THREAD_PRIORITY_HIGHEST    Yes    No       19199869
10 THREAD_PRIORITY_TIME_CRITICAL Yes    No       11314114
11 Press "Enter" to exit.

```

Видим, что приоритет не изменился ни у одного потока.

Результат работы программы при переводе ресурсов процессора на одно ядро:

```

1 D:\temp\os\Debug>start /affinity 1 os.exe 7 10
2 Count of threads – 7, time to stop – 10.
3 Priority          Boost   Changed   Counter
4 THREAD_PRIORITY_IDLE      Yes    No        0
5 THREAD_PRIORITY_LOWEST    Yes    No        0
6 THREAD_PRIORITY_BELOW_NORMAL Yes    No        0
7 THREAD_PRIORITY_NORMAL    Yes    No        0
8 THREAD_PRIORITY_ABOVE_NORMAL Yes    Yes        3
9 THREAD_PRIORITY_HIGHEST    Yes    Yes      188516803
10 THREAD_PRIORITY_TIME_CRITICAL Yes    Yes      188855934
11 Press "Enter" to exit.

```

Как видно из результатов на одном процессоре, из-за включенной возможности динамического изменения приоритетов операционной системой, были изменены три приоритета, что повлияло на результаты.

3. Анализ поведения системных функций динамического управления приоритетами

С помощью программы определим, назначается ли динамическое изменение приоритетов по умолчанию, на все ли потоки воздействует функция `SetProcessPriorityBoost`, возможно ли разрешение отдельному потоку в процессе динамически изменять приоритет, если для процесса это запрещено.

```

1 #include <iostream>
2 #include <windows.h>
3
4 static const int SLEEP_DURATION = 1000;
5
6 // Обработчик потока
7 DWORD WINAPI threadExecutor(LPVOID);
8 // Вывод информации о динамическом изменении приоритетов
9 void printPriorityBoostInfo(HANDLE mainProcess, HANDLE mainThread, HANDLE sideThread);
10 // Расшифровка результата
11 const char* getStringByBool(BOOL value);
12
13 int main() {
14     // Получение дескриптора текущего процесса
15     HANDLE mainProcess = GetCurrentProcess();
16
17     // Получение дескриптора текущей нити
18     HANDLE mainThread = GetCurrentThread();
19
20     // Создание второго потока
21     HANDLE sideThread = CreateThread(nullptr, NULL, threadExecutor, nullptr, NULL, nullptr)
22     ;
23
24     // Вывод информации о динамическом изменении приоритетов

```

```

24 std::cout<< "Default priorities" << std::endl;
25 printPriorityBoostInfo(mainProcess, mainThread, sideThread);
26
27 // Повышаем приоритет второго потока
28 if(!SetThreadPriorityBoost(sideThread, true)) {
29     std::cerr << "It's impossible to change thread priority." << std::endl;
30     return 0x1;
31 }
32
33 // Вывод информации о динамическом изменении приоритетов
34 std::cout << std::endl << "Side thread priority has been changed" << std::endl;
35 printPriorityBoostInfo(mainProcess, mainThread, sideThread);
36
37 // Повышаем приоритет процесса
38 if(!SetProcessPriorityBoost(mainProcess, true)) {
39     std::cout << "It's impossible to change process priority." << std::endl;
40     return 0x2;
41 }
42
43 // Вывод информации о динамическом изменении приоритетов
44 std::cout << std::endl << "Process priority has been changed" << std::endl;
45 printPriorityBoostInfo(mainProcess, mainThread, sideThread);
46
47 // Понижаем приоритет второго потока
48 if(!SetThreadPriorityBoost(sideThread, false)) {
49     std::cerr << "It's impossible to change thread priority." << std::endl;
50     return 0x3;
51 }
52
53 // Вывод информации о динамическом изменении приоритетов
54 std::cout << std::endl << "Side thread priority has been changed" << std::endl;
55 printPriorityBoostInfo(mainProcess, mainThread, sideThread);
56
57 std::cout << "Press \"Enter\" to exit.\n" << std::endl;
58 std::getchar();
59
60 return 0;
61 }
62
63 DWORD WINAPI threadExecutor(LPVOID) {
64     while(true)
65         Sleep(SLEEP_DURATION);
66 }
67
68 void printPriorityBoostInfo(HANDLE mainProcess, HANDLE mainThread, HANDLE sideThread) {
69     BOOL result;
70
71     GetProcessPriorityBoost(mainProcess, &result);
72     std::cout << "Process dynamic: " << getStringByBool(result) << std::endl;
73
74     GetThreadPriorityBoost(mainThread, &result);
75     std::cout << "Main thread dynamic: " << getStringByBool(result) << std::endl;
76
77     GetThreadPriorityBoost(sideThread, &result);
78     std::cout << "Side thread dynamic: " << getStringByBool(result) << std::endl;
79 }
80
81 const char* getStringByBool(BOOL value) {
82     return ((value == 0) ? "Dynamic enable": "Disabled");
83 }

```

Результат работы программы:

```

1 D:\temp\os\x64\Debug>os.exe
2 Default priorities
3 Process dynamic: Dynamic enable
4 Main thread dynamic: Dynamic enable

```

```

5 Side thread dynamic: Dynamic enable
6
7 Side thread priority has been changed
8 Process dynamic: Dynamic enable
9 Main thread dynamic: Dynamic enable
10 Side thread dynamic: Disabled
11
12 Process priority has been changed
13 Process dynamic: Disabled
14 Main thread dynamic: Disabled
15 Side thread dynamic: Disabled
16
17 Side thread priority has been changed
18 Process dynamic: Disabled
19 Main thread dynamic: Disabled
20 Side thread dynamic: Dynamic enable
21
22 Press "Enter" to exit.

```

По умолчанию для процессов и потоков динамическое изменение приоритетов разрешено. Потом для второго потока было запрещено изменение динамического приоритета с помощью функции `SetThreadPriorityBoost`. Если запретить динамическое изменение приоритетов для процесса функцией `SetProcessPriorityBoost`, то изменение также будет запрещено и для всех потоков. Разрешение отдельному потоку в процессе динамически изменять приоритет возможно, если для процесса это запрещено.

1.4.4 Глава 4. Самостоятельные задания

1. Занесение экспериментальных данных в таблицу

Для заполнения таблицы, была доработана программа из предыдущей главы. Теперь программа не только создает семь потоков с различными приоритетами, но и варьирует значения приоритета самого процесса:

```

1 #include <iostream>
2 #include <windows.h>
3 #include <string>
4 #include <iomanip>
5
6 // Аргумент для функции обработчика потока
7 struct Argument {
8     int number;
9     bool* interrupt;
10 };
11
12 static const int COUNT_OF_PROCESS_PRIORITIES = 6;
13 static const int COUNT_OF_THREAD_PRIORITIES = 7;
14 // Количество создаваемых потоков по умолчанию
15 static const int DEFAULT_COUNT_OF_THREADS = COUNT_OF_THREAD_PRIORITIES;
16 // Время до завершения потока в( секундах)
17 static const int DEFAULT_TIME_TO_STOP = 5;
18 // Константа для таймера
19 static const __int64 TIMER_CONSTANT = -1 * 10000000;
20
21 // Счетчики увеличиваемые каждым потоком
22 static __int64 counter[COUNT_OF_THREAD_PRIORITIES] = { 0, 0, 0, 0, 0, 0, 0 };
23 // Если был буст
24 static int boost[COUNT_OF_THREAD_PRIORITIES] = { 0, 0, 0, 0, 0, 0, 0 };
25 // Показывает, изменился ли приоритет в процессе
26 static bool change[COUNT_OF_THREAD_PRIORITIES] = { false, false, false, false, false, false, false };
27 // Значения приоритетов для процесса
28 static const int processPriority[COUNT_OF_PROCESS_PRIORITIES] = { IDLE_PRIORITY_CLASS,
29     BELOW_NORMAL_PRIORITY_CLASS, NORMAL_PRIORITY_CLASS, ABOVE_NORMAL_PRIORITY_CLASS,
30     HIGH_PRIORITY_CLASS, REALTIME_PRIORITY_CLASS };
31 // Текстовые строки приоритетов для вывода
32 static const char* processString[COUNT_OF_PROCESS_PRIORITIES] = { "IDLE_PRIORITY_CLASS",
33     "BELOW_NORMAL_PRIORITY_CLASS", "NORMAL_PRIORITY_CLASS", "ABOVE_NORMAL_PRIORITY_CLASS",

```

```

32 "HIGH_PRIORITY_CLASS", "REALTIME_PRIORITY_CLASS" };
33 // Значения приоритетов для потоков
34 static const int threadPriority[COUNT_OF_THREAD_PRIORITIES] = { THREAD_PRIORITY_IDLE,
35     THREAD_PRIORITY_LOWEST, THREAD_PRIORITY_BELOW_NORMAL, THREAD_PRIORITY_NORMAL,
36     THREAD_PRIORITY_ABOVE_NORMAL, THREAD_PRIORITY_HIGHEST, THREAD_PRIORITY_TIME_CRITICAL };
37 // Текстовые строки приоритетов для вывода
38 static const char* threadString[COUNT_OF_THREAD_PRIORITIES] = { "THREAD_PRIORITY_IDLE", "
39     THREAD_PRIORITY_LOWEST", "THREAD_PRIORITY_BELOW_NORMAL", "THREAD_PRIORITY_NORMAL",
40     "THREAD_PRIORITY_ABOVE_NORMAL", "THREAD_PRIORITY_HIGHEST", "THREAD_PRIORITY_TIME_CRITICAL
41     " };
42
43 // Обработчик потока
44 DWORD WINAPI threadExecutor(LPVOID);
45
46 int main(int argc, char** argv) {
47     int countOfThreads = DEFAULT_COUNT_OF_THREADS;
48     int timeToStop = DEFAULT_TIME_TO_STOP;
49
50     // Получение аргументов командной строки
51     if(argc > 2) {
52         try {
53             countOfThreads = std::stoi(argv[1]);
54             timeToStop = std::stoi(argv[2]);
55             if(countOfThreads > DEFAULT_COUNT_OF_THREADS)
56                 return 0x1;
57         }
58         catch(const std::invalid_argument& exception) {
59             std::cerr << exception.what() << std::endl;
60             return 0x2;
61         }
62     }
63
64     std::cout << "Count of threads – " << countOfThreads << ", time to stop – " <<
65         timeToStop << "." << std::endl;
66
67     for(int processIndex = 0; processIndex < COUNT_OF_PROCESS_PRIORITIES; ++processIndex) {
68
69         for(int zeroingIndex = 0; zeroingIndex < COUNT_OF_THREAD_PRIORITIES; ++zeroingIndex)
70         {
71             counter[zeroingIndex] = 0;
72             boost[zeroingIndex] = 0;
73             change[zeroingIndex] = false;
74         }
75
76         std::cout << "Process priority is " << processString[processIndex] << "." << std::
77             endl << std::endl;
78
79         // Создаем таймер
80         HANDLE timer = CreateWaitableTimer(nullptr, false, nullptr);
81
82         // Устанавливаем временные характеристики таймера
83         __int64 endTimeValue = TIMER_CONSTANT * timeToStop;
84         LARGE_INTEGER endTimeStruct;
85         endTimeStruct.LowPart = DWORD(endTimeValue & 0xFFFFFFFF);
86         endTimeStruct.HighPart = LONG(endTimeValue >> 0x20);
87         SetWaitableTimer(timer, &endTimeStruct, NULL, nullptr, nullptr, false);
88
89         // Устанавливаем приоритет реального времени
90         SetPriorityClass(GetCurrentProcess(), processPriority[processIndex]);
91
92         bool interrupt = false;
93         for(int index = 0; index < countOfThreads; ++index) {
94             // Создаем указатель на структуру аргумента
95             Argument* argument = new Argument;
96             argument->number = index;
97             argument->interrupt = &interrupt;
98         }
99     }
100 }

```

```

92 // Создаем поток
93 HANDLE thread = CreateThread(nullptr, NULL, threadExecutor, LPVOID(argument), NULL,
94 nullptr);
95 // Задаем приоритет для каждого потока
96 SetThreadPriority(thread, threadPriority[index]);
97 SetThreadPriorityBoost(thread, true);
98 GetThreadPriorityBoost(thread, &boost[index]);
99 CloseHandle(thread);
100 }
101
102 // Ожидаем таймер
103 WaitForSingleObject(timer, INFINITE);
104 CloseHandle(timer);
105 // Посылаем прерывание всем потокам
106 interrupt = true;
107
108 // Выводим в виде таблицы
109 std::cout << std::left << std::setw(32) << std::setfill(' ') << "Priority" <<
110 std::left << std::setw(7) << std::setfill(' ') << "Boost" <<
111 std::left << std::setw(7) << std::setfill(' ') << "Changed" <<
112 std::right << std::setw(12) << std::setfill(' ') << "Counter" <<
113 std::endl;
114 for(int index = 0; index < countOfThreads; ++index)
115 std::cout << std::left << std::setw(32) << std::setfill(' ') << threadString[index]
116 <<
117 std::left << std::setw(7) << std::setfill(' ') << ((boost[index] == 0) ? "No" : "
Yes") <<
118 std::left << std::setw(7) << std::setfill(' ') << ((!change[index]) ? "No" : "Yes")
119 <<
120 std::right << std::setw(12) << std::setfill(' ') << counter[index] <<
121 std::endl;
122 }
123
124 printf("Press \"Enter\" to exit.\n");
125 std::getchar();
126
127 return 0x0;
128 }
129
130 // Обработчик потока
131 DWORD WINAPI threadExecutor(LPVOID ptr) {
132 // Указатель на идентификатор потока был передан как параметр функции
133 const Argument argument = *static_cast<Argument*>(ptr);
134
135 // Увеличиваем счетчик пока не пришло прерывание
136 while(!(*argument.interrupt)) {
137 // Получаем текущий приоритет
138 int currentPriority = GetThreadPriority(GetCurrentThread());
139 // Если приоритет поменялся
140 if(currentPriority != threadPriority[argument.number])
141 change[argument.number] = true;
142
143 ++counter[argument.number];
144 }
145
146 return 0x0;
147 }

```

Результат работы программы:

```

1 Count of threads – 7, time to stop – 5.
2 Process priority is IDLE_PRIORITY_CLASS.
3
4 Priority           Boost  Changed   Counter

```

```

5  THREAD_PRIORITY_IDLE          Yes    No      8790427
6  THREAD_PRIORITY_LOWEST        Yes    No      9057098
7  THREAD_PRIORITY_BELOW_NORMAL  Yes    No      9253933
8  THREAD_PRIORITY_NORMAL        Yes    No      9597097
9  THREAD_PRIORITY_ABOVE_NORMAL  Yes    No      9196167
10 THREAD_PRIORITY_HIGHEST        Yes    No      9581430
11 THREAD_PRIORITY_TIME_CRITICAL  Yes    No      10261306
12
13 Process priority is BELOW_NORMAL_PRIORITY_CLASS.
14
15 Priority          Boost    Changed    Counter
16 THREAD_PRIORITY_IDLE          Yes    No      9494646
17 THREAD_PRIORITY_LOWEST        Yes    No      9575198
18 THREAD_PRIORITY_BELOW_NORMAL  Yes    No      9350859
19 THREAD_PRIORITY_NORMAL        Yes    No      9847109
20 THREAD_PRIORITY_ABOVE_NORMAL  Yes    No      9968149
21 THREAD_PRIORITY_HIGHEST        Yes    No      9966572
22 THREAD_PRIORITY_TIME_CRITICAL  Yes    No      10262892
23
24 Process priority is NORMAL_PRIORITY_CLASS.
25
26 Priority          Boost    Changed    Counter
27 THREAD_PRIORITY_IDLE          Yes    No      8520514
28 THREAD_PRIORITY_LOWEST        Yes    No      9783950
29 THREAD_PRIORITY_BELOW_NORMAL  Yes    No      9150265
30 THREAD_PRIORITY_NORMAL        Yes    No      10332312
31 THREAD_PRIORITY_ABOVE_NORMAL  Yes    No      10489061
32 THREAD_PRIORITY_HIGHEST        Yes    No      9308324
33 THREAD_PRIORITY_TIME_CRITICAL  Yes    No      11359137
34
35 Process priority is ABOVE_NORMAL_PRIORITY_CLASS.
36
37 Priority          Boost    Changed    Counter
38 THREAD_PRIORITY_IDLE          Yes    No      8494694
39 THREAD_PRIORITY_LOWEST        Yes    No      7868407
40 THREAD_PRIORITY_BELOW_NORMAL  Yes    No      9519697
41 THREAD_PRIORITY_NORMAL        Yes    No      10915955
42 THREAD_PRIORITY_ABOVE_NORMAL  Yes    No      11329135
43 THREAD_PRIORITY_HIGHEST        Yes    No      10557174
44 THREAD_PRIORITY_TIME_CRITICAL  Yes    No      10245337
45
46 Process priority is HIGH_PRIORITY_CLASS.
47
48 Priority          Boost    Changed    Counter
49 THREAD_PRIORITY_IDLE          Yes    No      5434433
50 THREAD_PRIORITY_LOWEST        Yes    No      11784753
51 THREAD_PRIORITY_BELOW_NORMAL  Yes    No      11762219
52 THREAD_PRIORITY_NORMAL        Yes    No      11385041
53 THREAD_PRIORITY_ABOVE_NORMAL  Yes    No      11612825
54 THREAD_PRIORITY_HIGHEST        Yes    No      10978057
55 THREAD_PRIORITY_TIME_CRITICAL  Yes    No      7753954
56
57 Process priority is REALTIME_PRIORITY_CLASS.
58
59 Priority          Boost    Changed    Counter
60 THREAD_PRIORITY_IDLE          Yes    No      7666557
61 THREAD_PRIORITY_LOWEST        Yes    No      10013229
62 THREAD_PRIORITY_BELOW_NORMAL  Yes    No      5649789
63 THREAD_PRIORITY_NORMAL        Yes    No      9850837
64 THREAD_PRIORITY_ABOVE_NORMAL  Yes    No      12064692
65 THREAD_PRIORITY_HIGHEST        Yes    No      12126302
66 THREAD_PRIORITY_TIME_CRITICAL  Yes    No      12746440
67
68 Press "Enter" to exit.

```


Занесем эти данные в результирующую таблицу:

Thread \ Process	Idle	Below Normal	Normal	Above Normal	High	Realtime
Idle	8790427	9494646	8520514	8494694	5434433	7666557
Lowest	9057098	9575198	9783950	7868407	11784753	10013229
Below Normal	9253933	9350859	9150265	9519697	11762219	5649789
Normal	9597097	9847109	10332312	10915955	11385041	9850837
Above Normal	9196167	9968149	10489061	11329135	11612825	12064692
Highest	9581430	9966572	9308324	10557174	10978057	12126302
Time Critical	10261306	10262892	11359137	10245337	7753954	12746440
Сумма	65737458	68465425	68943563	68930399	70711282	70117846

Рис. 1.5

Можно заключить, что класс приоритета процесса действительно влияет на процессорное время, отводимое каждому процессу. Значение в графе "Сумма" в общем случае увеличивается с увеличением класса приоритета процесса.

2. Фиксация динамического изменения приоритетов

С помощью утилиты мониторинга процессов зафиксируем изменение приоритетов:

os.exe	5404	Выполняется	nikita	58	Низкий	11	os.exe
--------	------	-------------	--------	----	--------	----	--------

Рис. 1.6

os.exe	9092	Выполняется	nikita	59	Ниже сре...	11	os.exe
--------	------	-------------	--------	----	-------------	----	--------

Рис. 1.7

os.exe	8420	Выполняется	nikita	57	Выше сре...	11	os.exe
--------	------	-------------	--------	----	-------------	----	--------

Рис. 1.8

os.exe	8428	Выполняется	nikita	57	Реальног...	11	os.exe
--------	------	-------------	--------	----	-------------	----	--------

Рис. 1.9

При изменении значения класса приоритета процесса, динамически изменяется числовое значение колонки "Приоритет". Более высокому классу приоритета процесса соответствует большее числовое значение приоритета, что является противоположной ситуацией относительно ОС Unix, где меньшее число означало больший приоритет при диспетчеризации.

3. Наследование

Разработаем программу, которая создает файл, записывает в него строку и запускает процесс потомок с передачей ему дескриптора файла. Процесс потомок принимает файловый дескриптор аргументом командной строки, открывает файл и дописывает в конец другую строку.

Родительский процесс:

```

1 #include <stdio.h>
2 #include <windows.h>
3 #include <stdlib.h>
4 #include <iostream>
5
6 // Имя файла
7 static const char* FILENAME = "result.out";
8 // Строка для записи
9 static const char* PARENT_BUFFER = "Parent message\n";

```

```

10
11 static const char* COMMAND_LINE = "p4.3.c.exe %d";
12
13 static const int COMMAND_BUFFER_SIZE = 100;
14
15 int main() {
16     // Задаем атрибуты для файла
17     SECURITY_ATTRIBUTES securityAttributes;
18     ZeroMemory(&securityAttributes, sizeof(SECURITY_ATTRIBUTES));
19     securityAttributes.nLength = sizeof(SECURITY_ATTRIBUTES);
20     securityAttributes.lpSecurityDescriptor = nullptr;
21     securityAttributes.bInheritHandle = TRUE;
22
23     // Создаем новый файл
24     HANDLE createFile = CreateFile(TEXT(FILENAME), GENERIC_WRITE, NULL, &securityAttributes
25     , CREATE_NEW, FILE_ATTRIBUTE_NORMAL, nullptr);
26     if(createFile == INVALID_HANDLE_VALUE) {
27         std::cerr << "It's impossible to create file." << std::endl;
28         return 0x1;
29     }
30
31     // Записываем строку в файл
32     DWORD countOfBytesWritten;
33     BOOL writeFile = WriteFile(createFile, PARENT_BUFFER, DWORD(strlen(PARENT_BUFFER)), &
34     countOfBytesWritten, nullptr);
35     if(writeFile == FALSE) {
36         std::cerr << "It's impossible to write file." << std::endl;
37         return 0x2;
38     }
39
40     // Запись команды для создания процесса
41     TCHAR cmdLine[COMMAND_BUFFER_SIZE];
42     wsprintf(cmdLine, TEXT(COMMAND_LINE), createFile);
43
44     // Задаем информацию для создания процесса
45     STARTUPINFO startupInfo;
46     ZeroMemory(&startupInfo, sizeof(STARTUPINFO));
47     startupInfo.cb = sizeof(STARTUPINFO);
48
49     PROCESS_INFORMATION processInformation;
50
51     // Создаем новый процесс
52     if(!CreateProcess(nullptr, cmdLine, nullptr, nullptr, TRUE, CREATE_NEW_CONSOLE, nullptr
53     , nullptr, &startupInfo, &processInformation)) {
54         std::cerr << "It's impossible to create process." << std::endl;
55         return 0x3;
56     }
57
58     CloseHandle(processInformation.hThread);
59     CloseHandle(processInformation.hProcess);
60
61     printf("Press \"Enter\" to exit.\n");
62     std::getchar();
63
64     CloseHandle(createFile);
65     return 0x0;
66 }

```

Процесс-потомок:

```

1 #include <stdio.h>
2 #include <windows.h>
3 #include <string>
4 #include <iostream>
5
6 // Строка для записи
7 static const char* CHILD_BUFFER = "Child message\n";

```

```

8
9 int main(int argc, char *argv[]) {
10     // Получаем название файла для записи из командной строки
11     HANDLE filename = HANDLE(std::stoi(argv[1]));
12
13     DWORD countOfBytesWritten;
14
15     // Записываем строку в файл
16     BOOL writeFile = WriteFile(filename, CHILD_BUFFER, DWORD(strlen(CHILD_BUFFER)), &
17         countOfBytesWritten, nullptr);
18     if(writeFile == FALSE) {
19         std::cerr << "It's impossible to write file." << std::endl;
20         return 0x1;
21     }
22
23     CloseHandle(filename);
24     return 0x0;
25 }

```

Результат был записан файл:

```

1 Parent message
2 Client message

```

Результат записи родительским процессом и процессом потомком полностью совпал с ожидаемым, что говорит об успешном завершении эксперимента.

1.5 Вывод

Используя функцию `CreateProcess`, можно создать новый процесс для запуска другого приложения из текущего. Большое количество параметров функции обеспечивают гибкость при создании программ и порождении новых процессов.

Используя функцию `CreateThread`, можно создавать новые потоки в рамках одного процесса, тем самым распараллеливая вычисления. Как и функция `CreateProcess`, функция `CreateThread` имеет большое количество параметров, предоставляя широкие возможности по созданию потоков.

Такие системные объекты, как таймеры ожидания, могут быть использованы для синхронизации потоков в ОС семейства Windows.

Windows поддерживает шесть классов приоритета процесса, а так же 7 возможных приоритетов потока в рамках класса приоритета процесса. В ОС существует динамическое повышение приоритета, предназначенное для оптимизации общей пропускной способности системы: операционная система может автоматически повышать значение текущего приоритета после завершения операции ввода/вывода, по окончании ожидания на каком-либо системном объекте, при нехватке процессорного времени.

1.6 Список литературы

- `CreateProcess` function [Электронный ресурс]. — URL: [https://msdn.microsoft.com/ru-ru/library/windows/desktop/ms682425\(v=vs.85\).aspx](https://msdn.microsoft.com/ru-ru/library/windows/desktop/ms682425(v=vs.85).aspx) (дата обращения 06.01.2017).
- `CreateThread` function [Электронный ресурс]. — URL: [https://msdn.microsoft.com/ru-ru/library/windows/desktop/ms682453\(v=vs.85\).aspx](https://msdn.microsoft.com/ru-ru/library/windows/desktop/ms682453(v=vs.85).aspx) (дата обращения 06.01.2017).
- Scheduling Priorities [Электронный ресурс]. — URL: [https://msdn.microsoft.com/en-us/library/windows/desktop/ms685100\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms685100(v=vs.85).aspx) (дата обращения 06.01.2017).