Санкт-Петербургский Политехнический Университет Петра Великого
Институт компьютерных наук и технологий
Кафедра компьютерных систем и программных технологий

Операционные системы

Отчет по лабораторной работе №5
Управление процессами и потоками в Windows

Работу выполнил:
Черевичник Андрей
Группа: 43501/3
Преподаватель:
Малышев Игорь Алексеевич

Санкт-Петербург
2017

1 Цель работы

Изучить способы управления процессами и потоками в ОС семейства Windows.

2 Программа работы

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.
   1.2. Программа, получает имя конфигурационного файла из командной строки, открывает конфигурационный файл, читает строки и создает для запуска каждой команды отдельный процесс.
   1.3. Пусть программа получает имя конфигурационного файла из командной строки, открывает его с помощью fopen(), читает построчно функцией fgets(). 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.

2. Программа должна создавать два потока, выводящих в бесконечном цикле «1» и «2» соответственно. После создания дополнительных потоков, поток-родитель завершается. В данной программе после создания потоков, главный поток завершается, способов завершения может быть два: с помощью return и с помощью вызова функции ExitThread.
   2.2. 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.

3. 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. Предполагаем, так как приоритеты различны, то и время, отведенное на работу потокам различно (квант времени, выделяемый потокам, одинаков).
   3.2. Усложним задачу и дополним ее возможностью управлять классом приоритетов процесса. Код программы несколько изменим для получения более наглядного вывода результатов. Пусть программа по-прежнему создает 7 дополнительных потоков, со всеми возможными вариантами приоритета. Теперь в начале работы можно изменить класс приоритета процесса в целом. Каждый рабочий процесс выполняет увеличение связанного с ним счетчика (здесь типа int).
   3.3. Анализ поведения системных функций динамического управления приоритетами процессов и потоков. 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.:

4. Исследуйте результаты работы программы 3.1 и 3.2 в зависимости от того, какой приоритет назначается базовому потоку: above_normal, idle_priority class, high priority class, normal priority class и др.; своими экспериментальными данными заполните таблицу, с точным указанием для какой ОС и на каком отладочном комплексе проводились измерения.

5. Модифицируйте программу 3.2 для заполнения таблицы 2 текущими данными вашего эксперимента. Сделайте выводы.

6. С помощью утилит CPU Stress, позволяющих нагружать систему, и утилиты мониторинга ProcessExplorer() (или иных утилит) зафиксируйте динамическое изменение приоритетов, приведите результаты в отчете.

7. Создайте программу, демонстрирующую возможность наследования

   - дескриптора порождающего процесса,
   - дескрипторов открытых файлов,

   Для выполнения этого задания следует учесть, что по умолчанию наследование в Windows отключено и для возможности наследования, необходимо:

   (a) разрешить процессу-потомку наследовать дескрипторы,
   (b) сделать дескрипторы наследуемыми

3 Ход работы

3.1 Создание процессов

Рассмотрим создание одного процесса другим посредством функции WinAPI **CreateProcess**. При создании исполнительная система выполняет работу по организации окружения (среды исполнения процесса) и предоставлению необходимых ему ресурсов. Она выделяет новое адресное пространство и иные ресурсы для процесса, а также создает для него новый 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. Синтаксис команды **CreateProcess**:

```
BOOL CreateProcess( LPCTSTR lpApplicationName, // имя исполняемого модуля
          LPTSTR lpCommandLine, // командная строка
          LPSECURITY_ATTRIBUTES lpProcessAttributes, // атрибуты безопасности процесса
          LPSECURITY_ATTRIBUTES lpThreadAttributes, // атрибуты безопасности потока
          BOOL bInheritHandeles, // флаг наследования описателя
          DWORD dwCreationFlags, // флаги создания
          LPVOID lpEnvironment, // новый блок окружения
          LPCTSTR lpCurrentDirectory, // имя текущей директории
          LPSTARTUPINFO lpStartupInfo, // STARTUPINFO
          LPPROCESS_INFORMATION lpProcessInformation // PROCESS_INFORMATION
          )

```

CreateProcess() отводит место под объекты процесс и поток и возвращает значения их описателей (индексы в таблице) в структуре
`PROCESS_INFORMATION`.

Освободить выделенное место можно вызовом **CloseHandle**. При этом выполнение этого вызова не обязательно приведет к завершению процесса (только исчезнет ссылка на объект внутри вызвавшего процесса).

CreateProcess() возвращает ноль, если создание процесса прошло успешно.

Задание 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.

Исходный код программы (task_1.cpp), создающей процесс для запуска приложения notepad.exe и открытия блокнота с файлом без имени:

```
#include <windows.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char** argv[]) {
    printf("Program started\n");
    char LpCommandLine[] = "C:\\Windows\\notepad.exe";
    wchar_t wtext[30];
    mbstowcs(wtext, LpCommandLine, strlen(LpCommandLine) + 1);//Plus null
    LPWSTR ptr = wtext;
    STARTUPINFO startupInfo;
    PROCESS_INFORMATION processInfo; //информация о процессе
    ZeroMemory(&startupInfo, sizeof(STARTUPINFO));
    startupInfo.cb = sizeof(startupInfo);
    if (!CreateProcess(NULL, ptr, NULL, NULL, false, HIGH_PRIORITY_CLASS |
    ↪ CREATE_NEW_CONSOLE, NULL, NULL, &startupInfo, &processInfo)) {
        printf("Error creating process: %d\n", GetLastError());
        return -1;
    } else {
        printf("new process Handle: %d Handle of thread: %d\n", processInfo.dwProcessId,
    ↪ processInfo.dwThreadId);
        printf("Successfully created new process!\n");
    }
    CloseHandle(processInfo.hThread);
    CloseHandle(processInfo.hProcess);
    printf("Program finished\n");
    getchar();
    return 0;
}
```

Параметр lpApplicationName установлен NULL. В этом случае, имя модуля должно быть в строке lpCommandLine.

В данном случае будем использовать значения по умолчанию для атрибутов безопасности процесса и потока — NULL (параметры lpProcessAttributes и lpThreadAttributes) и FALSE для флага наследования (bInheritHandles).

Для создания нового процесса (child) с высоким приоритетом в его собственном окне используем — `HIGH_PRIORITY_CLASS | CREATE_NEW_CONSOLE`.

Параметр lpEnvironment используется для передачи нового блока переменных окружения порожденному процессу-потомку (child). Если указано NULL —- то потомок использует то же окружение, что и родитель. Параметр lpCurrentDirectory установлен в (NULL). Это означает, что новый процесс создается с тем же самым текущим диском и каталогом, что и вызывающий процесс.

В структуре startupInfo устанавливаются оконный режим терминала, рабочий стол, стандартные дескрипторы и внешний вид главного окна для нового процесса.

В структуре processInfo хранятся: описатель вновь созданного процесса (hProcess), описатель его базового потока (hThread), глобальный идентификатор процесса (dwProcessId), глобальный идентификатор потока (dwThreadId).

Результат работы программы приведен на рисунке 1.

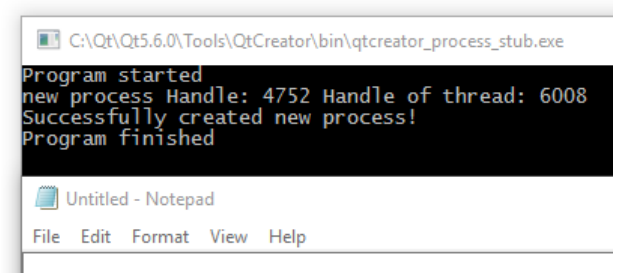

Рис. 1: Результат работы программы task_1.cpp

Задание Программа, получает имя конфигурационного файла из командной строки, открывает конфигурационный файл, читает строки и создает для запуска каждой команды отдельный процесс.

С целью упрощения кода сначала имя командного файла зададим прямо в программе и представим текст программы без обработки возможных ошибок. Создаем конфигурационный файл с именем "temp.txt"при помощи текстового редактора(например, notepad) и располагаем его в корневом каталоге диска C. Далее записываем в файл следующие строки:

```
C:\Windows\System32\NOTEPAD.EXE
C:\Windows\System32\CALC.EXE

```

Исходный код программы task_2.cpp:

```
#include <stdio.h>
#include <windows.h>
#include <assert.h>
#include <iostream>
#define MAX_LEN 200

int main(int argc, char* argv[]) {
    const char* frd = "C:\\temp.txt";
    FILE *f=fopen(frd,"r");
    if(f==NULL) {
        std::cout << "Coudn't open file" << std::endl;
        system("pause");
        return 1;
    }
    for (int i = 0; i<2; i++) {
        wchar_t* execString = (wchar_t*)calloc(MAX_LEN, sizeof(wchar_t));
        //выделение памяти
        fgetws(execString, MAX_LEN, f);
        // чтение строки из файла
        execString[wcslen(execString) - 1] = '\0';
        STARTUPINFO startupInfo;
        ZeroMemory(&startupInfo, sizeof(STARTUPINFO));
        startupInfo.cb = sizeof(startupInfo);
        PROCESS_INFORMATION processInfo;
        std::wcout << execString;
```

```
26          std::cout << std::endl;
27          if( !CreateProcess(NULL, execString, NULL, NULL, false, 0, NULL, NULL, &startupInfo,
        ↪  &processInfo) ) {
28              std::cout << "Error creating process: " << GetLastError() << std::endl;
29              return -1;
30          } else
31              printf("Process successfully created!\n");
32          free(execString);
33          CloseHandle(processInfo.hThread);
34          CloseHandle(processInfo.hProcess);
35      }
36      return 0;
37  }
```

Результат работы программы приведен на рисунке 2.

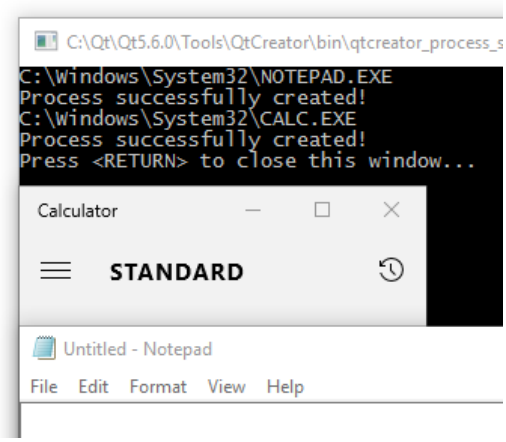

Рис. 2: Результат работы программы task_2.cpp

Доработаем программу. Пусть программа получает имя конфигурационного файла в качестве аргумента из командной строки, открывает его с помощью fopen(), читает построчно функцией fgetws(). 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.

Исходный код программы task_2_changed.cpp:

```
1  #include <stdio.h>
2  #include <windows.h>
3  #include <assert.h>
4  #include <iostream>
5  #define DEF_BUFLEN 200
6
7  int main(int argc, char* argv[]) {
8
9      printf("Program started\n");
10     if (argc < 2){
11         std::cout << "Input name of configuration file" << std::endl;
12         exit(-1);
13     }
14     const char* frd = argv[1];
15     FILE *f = fopen(frd, "r");
16     if(f==NULL){
17         std::cout << "error opening file " << argv[1] << std::endl;
18         exit(-2);
19     }
20     wchar_t commandLine[DEF_BUFLEN];
21     for (int i = 0; i<2; i++) {
22         wchar_t* execString = (wchar_t*)calloc(DEF_BUFLEN, sizeof(wchar_t));
23         //выделение памяти
24         fgetws(execString, DEF_BUFLEN, f);
25         // чтение строки из файла
26         execString[wcslen(execString) - 1] = '\0';
27         STARTUPINFO startupInfo;
28         ZeroMemory(&startupInfo, sizeof(STARTUPINFO));
29         startupInfo.cb = sizeof(startupInfo);
30         PROCESS_INFORMATION processInfo;
31         std::wcout << execString;
```

```
32          std::cout << std::endl;
33          if( !CreateProcess(NULL, execString, NULL, NULL, false, 0, NULL, NULL, &startupInfo,
    ↪  &processInfo) ) {
34              std::cout << "Error creating process: " << GetLastError() << std::endl;
35              return -1;
36          } else
37              printf("Process successfully created!\n");
38          free(execString);
39          CloseHandle(processInfo.hThread);
40          CloseHandle(processInfo.hProcess);
41      }
42      return 0;
43  }
```

Результат работы программы приведен на рисунке 3.

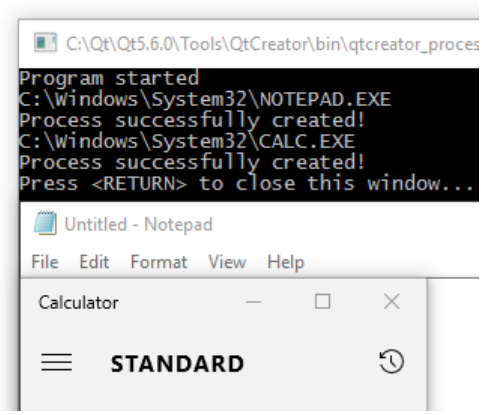

Рис. 3: Результат работы программы task_2_changed.cpp

Вызов процессов производится здесь без синхронизации. Процесс-родитель не дожидается создания процессов-потомков, а им после создания необходимо разбирать строку аргументов. Когда передается один неверный аргумент, процесс не создается (т.к. сразу проверяется наличие исполняемого файла с указанным именем). «Error 2» соответствует ошибка «Файл не найден». При передаче в commandLine процесса строки из нескольких слов процесс все равно создается (даже если аргументы не верны), но быстро завершается. При этом созданные процессы связаны с консолью процесса-родителя (аргумент DwCreationFlags в вызове CreateProcess равен 0).

3.2 Создание потоков

Создание потоков посредством функции WinAPI **CreateThread**:

```
1  HANDLE CreateThread(
2    LPSECURITY_ATTRIBUTES lpsa, //дескриптор защиты
3    DWORD dwStackSize, // начальный размер стека
4    LPTHREAD_START_ROUTINE lpStartAddr, //функция потока
5    LPVOID lpThreadParm, //параметр потока
6    DWORD dwCreationFlags, //опции создания
7    LPDWORD lpThreadId //идентификатор потока
8    )
9
```

Если функция выполнилась успешно, то вернется описатель потока, если нет, вернется NULL.

Задание Программа должна создавать два потока, выводящих в бесконечном цикле «1» и «2» соответственно. После создания дополнительных потоков, поток-родитель завершается.

В данной программе после создания потоков, главный поток завершается, способов завершения может быть два: с помощью return и с помощью вызова функции ExitThread.

Исходный код программы task_3.cpp:

```
1  #include <iostream>
2  #include <windows.h>
3
4  DWORD WINAPI threadHandler(LPVOID);
5
6  int main(int argc, char* argv[]) {
7      printf("Program started\n");
```

```
8       HANDLE t;
9       int number = 1;
10      t = CreateThread(NULL, 0, threadHandler, (LPVOID)number, 0, NULL);
11      CloseHandle(t);
12      number = 2;
13      t = CreateThread(NULL, 0, threadHandler, (LPVOID)number, 0, NULL);
14      CloseHandle(t);
15      //ExitThread(0); // разкомментировать для второго варианта
16      std::cout << "Program finished" << std::endl;
17      Sleep(1000);
18      return 0;
19  }
20  DWORD WINAPI threadHandler(LPVOID param) {
21      int number = (int)param;
22      for (;;) {
23          Sleep(300);
24          std::cout << number;
25      }
26      return 0;
27  }
```

Результат работы программы приведен на рисунках 4 – 5.

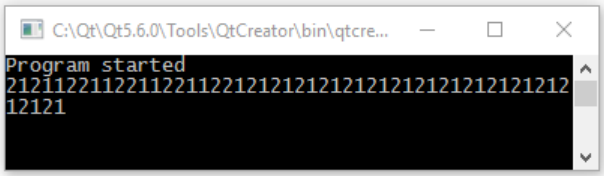

Рис. 4: Результат работы программы task3.cpp с завершением при помощи ExitThread(0)

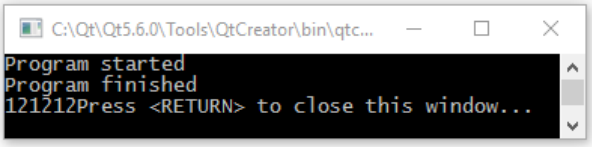

Рис. 5: Результат работы программы task3.cpp с завершением при помощи return

Очевидно, что в первом случае завершился только главный поток, а процесс – нет, а во втором - произошло завершение всего процесса.

Аналогичные результаты будут, если основной поток после создания потомков выполнит функцию **Sleep()** с неопределенным временем ожидания.

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.

Задание 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.

Возможны различные варианты решения данной задачи. Например, для подсчета времени можно использовать поток-координатор, вычисляющий момент завершения периода жизни с помощью функции **getTickCount()**, (сравнивая разницу текущего и стартового времени с заданным периодом жизни) или с помощью функции получения системного времени **GetSystemTime (&now)**. Другой способ (более рациональный) — использование таймера ожидания (Waitable Timer). Рассмотрим вариант на основе 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 «ожидаемый таймер» всегда создаются в занятом состоянии. Чтобы сообщить таймеру, в какой момент он должен перейти в свободное состояние, необходимо вызвать функцию **SetWaitableTimer**.

Исходный код программы task_4.cpp:

```
#include <stdio.h>
#include <windows.h>
DWORD WINAPI Thread1(LPVOID);

int stop; struct params {
    int num;
    bool* runflg;
};

int main(int argc, char* argv[]) {
    SYSTEMTIME now;
    int thrds;
    if (argc < 3)
        thrds = 2;
    else thrds = atoi(argv[1]);
    if (argc < 3)
        stop = 5000;
    else stop = atoi(argv[2]);
    DWORD targetThreadId;
    bool runFlag = true;
    __int64 end_time;
    LARGE_INTEGER end_time2; //создание и установка таймера
    HANDLE tm1 = CreateWaitableTimer(NULL, false, NULL);
    end_time = -1 * stop * 10000000;
    end_time2.LowPart = (DWORD) (end_time & 0xFFFFFFFF);
    end_time2.HighPart = (LONG) (end_time >> 32);
    SetWaitableTimer(tm1, &end_time2, 0,NULL, NULL, false);
    for (int i = 0; i < thrds; i++) {
        params* param = (params*)malloc(sizeof(params));
        param->num = i;
        param->runflg = &runFlag;
        HANDLE t1 = CreateThread(NULL, 0, Thread1, param, 0, &targetThreadId);
        CloseHandle(t1);
    }
    GetSystemTime(&now);
    printf("System Time %d %d %d\n", now.wHour, now.wMinute, now.wSecond);
    WaitForSingleObject(tm1, INFINITE);
    runFlag = false; //установка флага CloseHandle(tm1);
    GetSystemTime(&now);
    printf("System Time %d %d %d\n", now.wHour, now.wMinute, now.wSecond);
    system("pause");
    return 0;
}

DWORD WINAPI Thread1(LPVOID prm) {
    while (1) {
        params arg = *((params*)prm);
        Sleep(1000); printf("%d\n", arg.num);
        if (*(arg.runflg) == false) //проверка флага
            break;
    }
    return 0;
}
```

Результат работы программы приведен на рисунке 6

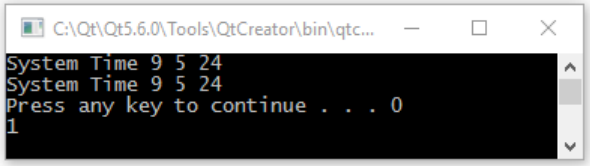

Рис. 6: Результат работы программы task4.cpp

3.3 Функции управления приоритетами процессов и потоков

Windows поддерживает шесть классов приоритета процесса: idle (простаивающий), below normal (ниже обычного), normal (обычный), above normal (выше обычного), high (высокий), real-time (реального времени).

Кроме того, Windows поддерживает семь относительных приоритетов потока: idle (простаивающий), lowest (низший), below normal (ниже обычного), normal (обычный), above normal (выше обычного), highest (высший) и time-critical (критичный по времени).

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:

1. после завершения операции ввода/вывода;
2. по окончании ожидания на каком-либо объекте исполнительной системы;
3. при нехватке процессорного времени и инверсии приоритетов.

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.

Системные функции обслуживаются с приоритетами реального времени. ОС никогда не меняет приоритет потоков с уровнями реального времени (от 16 до 31). Это ограничение позволяет сохранять целостность системы и обеспечивает необходимый уровень безопасности.

Задание 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.

Предполагаем, так как приоритеты различны, то и время, отведенное на работу потокам различно (квант времени, выделяемый потокам, одинаков).

Исходный код программы task_5.cpp:

```
#include <stdio.h>
#include <windows.h>

DWORD WINAPI Thread1(LPVOID);
int stop;
int sleep = 10000;
struct params {
    int num;
    bool* runflg;
};

long long counters[7] = { 0, 0, 0, 0, 0, 0, 0 }; //счетчики для потоков
int priority[7] = {THREAD_PRIORITY_IDLE,THREAD_PRIORITY_LOWEST,
THREAD_PRIORITY_BELOW_NORMAL,THREAD_PRIORITY_NORMAL,
THREAD_PRIORITY_ABOVE_NORMAL, THREAD_PRIORITY_HIGHEST,
THREAD_PRIORITY_TIME_CRITICAL}; //массив приоритетов в порядке возрастания

int main(int argc, char* argv[]) { //в командной строке аргументом задаем время жизни потоков
    stop = 10;
    DWORD targetThreadId;
    bool runFlag = true; //инициализация структур потокатаймера–
    __int64 end_time;
    LARGE_INTEGER end_time2;
    HANDLE tm1 = CreateWaitableTimer(NULL, false, NULL); //создание таймера
    end_time = -1 * stop * 10000000;
    end_time2.LowPart = (DWORD) (end_time & 0xFFFFFFFF);
    end_time2.HighPart = (LONG) (end_time >> 32);
    SetWaitableTimer(tm1, &end_time2, 0,NULL, NULL, false);//запуск таймера
    for (int i = 0; i < 7; i++) {
        params* param = (params*)malloc(sizeof(params));
        param->num = i;
        param->runflg = &runFlag;
        HANDLE t1 = CreateThread(NULL, 0, Thread1, param, 0, &targetThreadId); //порождение
    ↪ потока и
        SetThreadPriority(t1, priority[i]); //задание ему приоритета
        PBOOL ptr1 = (PBOOL)malloc(sizeof(BOOL));
```

```
36          //GetThreadPriorityBoost(t1, ptr1);
37          //SetThreadPriorityBoost(t1, true); проверка// динамического распределения приоритетов
38          CloseHandle(t1); //очистка памяти
39      }
40      WaitForSingleObject(tm1,INFINITE); //ожидание потока таймера
41      runFlag = false; //флаг завершения работы
42      CloseHandle(tm1);
43      printf("\n");
44      for (int i = 0; i < 7; i++) {
45          printf("%d - %ld\n",i, counters[i]); //вывод результатов
46      }
47      return 0;
48  }
49
50  DWORD WINAPI Thread1(LPVOID prm) {
51      while (1) {
52          //DWORD WINAPI thrdid = GetCurrentThreadId(); значение// идентификатора вызывающего
    ↪ потока
53          //HANDLE WINAPI handle = OpenThread(THREAD_QUERY_INFORMATION , false, thrdid);
    ↪ дескриптор// потока
54          //int WINAPI prio = GetThreadPriority(handle); приоритет// для определяемого потока
55          params arg = *((params*)prm);
56          counters[arg.num]++;
57          //if(prio != priority[arg.num])
58          //    printf("\nPriority of %d is %d %d changed\n", arg.num, priority[arg.num], prio
    ↪ ); выводится// когда динамическое распределение проиритетов включено
59          //Sleep(0);
60          if(*(arg.runflg) == false)
61              break;
62      }
63      return 1;
64  }
```

Результат работы программы приведен на рисунке 7

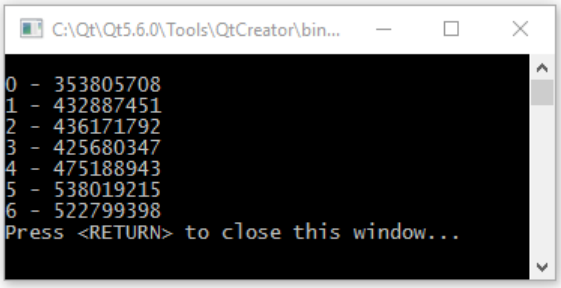

Рис. 7: Результат работы программы task_5.cpp

Слева — номер класса приоритета от низшего к высшему, а справа — значение счетчика, накопленное за все кванты, предоставленные потоку до истечения таймера. Очевидно, что потокам, у которых приоритет ниже, выделяется меньшее количество квантов времени для выполнения, и поэтому их счетчики соответственно меньше. При каждом запуске экспериментальные данные получаются различными, но пропорциональное соотношение между счетчиками потоков примерно сохраняется.

3.4 Управление классом приоритетов процесса

Усложним задачу и дополним ее возможностью управлять классом приоритетов процесса. Программа по-прежнему создает 7 дополнительных потоков, со всеми возможными вариантами приоритета. Теперь в начале работы можно изменить класс приоритета процесса в целом. Каждый рабочий процесс выполняет увеличение связанного с ним счетчика (здесь типа int). После увеличения счетчика, поток отдает оставшуюся часть кванта времени остальным, с помощью вызова функции Sleep с параметром 0. Через заданное время рабочие потоки завершаются, а основной поток выводит результаты их работы в новом формате. Окончание работы происходит по сигналу от таймера. Заложена возможность задания произвольного количества потоков и времени жизни, отсчитываемого таймером.

Исходный код программы task_6.cpp:

```
1  #include <stdio.h>
2  #include <conio.h>
```

```
3  #include <windows.h>
4  #define DEF_THREADS 7
5  #define DEF_TTL 10
6  DWORD WINAPI threadHandler(LPVOID);
7  HANDLE initTimer(int sec);
8  int getPriorityIndex(DWORD prClass);
9  int isFinish = 0;
10 long counters[7] = { 0, 0, 0, 0, 0, 0, 0 };
11 int priorities[7] = { THREAD_PRIORITY_IDLE, THREAD_PRIORITY_LOWEST,
       ↪ THREAD_PRIORITY_BELOW_NORMAL, THREAD_PRIORITY_NORMAL,
12 THREAD_PRIORITY_ABOVE_NORMAL, THREAD_PRIORITY_HIGHEST, THREAD_PRIORITY_TIME_CRITICAL };
13 char charPrio[7][10] = { "IDLE", "LOWEST", "BELOW", "NORMAL", "ABOVE", "HIGHEST", "TIME_CRIT
       ↪ " };
14 char charProcPrio[6][10] = { "IDLE", "BELOW", "NORMAL", "ABOVE", "HIGH", "REAL-TIME" };
15 int procPriorities[6] = { IDLE_PRIORITY_CLASS, BELOW_NORMAL_PRIORITY_CLASS,
       ↪ NORMAL_PRIORITY_CLASS,
16 ABOVE_NORMAL_PRIORITY_CLASS, HIGH_PRIORITY_CLASS, REALTIME_PRIORITY_CLASS };
17 int priorityBoost[7] = { 0, 0, 0, 0, 0, 0, 0 };
18 int priorityChange[7] = { 0, 0, 0, 0, 0, 0, 0 };
19 int main(int argc, char* argv[])
20 {
21     int numThreads = DEF_THREADS;
22     int threadLive = DEF_TTL;
23     if (argc < 2)
24         printf("Using default numThreads = %d and default time to live = %d\n", numThreads,
       ↪ threadLive);
25     else if (argc < 3)
26         printf("Using default time to live = %d\n", threadLive);
27     else {
28         numThreads = atoi(argv[1]);
29         threadLive = atoi(argv[2]);
30         if (numThreads <= 0 || threadLive <= 0) {
31             printf("All arguments must be numbers!!!!\n");
32             exit(0);
33         }
34     }
35     HANDLE t = initTimer(threadLive);
36     HANDLE t1;
37     // установить приоритет процесса IDLEили( иной)
38     SetPriorityClass(GetCurrentProcess(), REALTIME_PRIORITY_CLASS);
39     for (int i = 0; i < numThreads; i++) {
40         t1 = CreateThread(NULL, 0, threadHandler, (LPVOID)i, 0, NULL);
41         SetThreadPriority(t1, priorities[i]);
42         SetThreadPriorityBoost(t1, true);
43         GetThreadPriorityBoost(t1, &priorityBoost[i]);
44         CloseHandle(t1);
45     }
46     WaitForSingleObject(t,INFINITE); // ОЖИДАТЬ ВСЕХ ПОТОКОВ –
       ↪ комментируемнекомментируем– строку в зависимости от поставленной задачи
47     CloseHandle(t);
48     isFinish = 1;
49     char hasBoost[4];
50     char wasChanged[4];
51     int priorIdx = getPriorityIndex(GetPriorityClass(GetCurrentProcess()));
52     printf("Result of work:\n");
53     printf("Process priority:%s\n", charProcPrio[priorIdx]);
54     printf("Priority\tHas Boost\tWas changed\tCounter\n");
55     for (int i = 0; i < 7; i++) {
56         priorityBoost[i] == 0 ? strcpy_s(hasBoost, "NO") : strcpy_s(hasBoost, "YES");
57         priorityChange[i] == 0 ? strcpy_s(wasChanged, "NO") : strcpy_s(wasChanged, "YES");
58         printf("%8s\t%9s\t%10s\t%7d\n", charPrio[i], hasBoost, wasChanged, counters[i]);
59     }
60     system("pause");
61     return 0;
62 }
63 DWORD WINAPI threadHandler(LPVOID prm) {
64     int myNum = (int)prm;
65     int priority = 0;
66     for (;;) {
67         ++counters[myNum];
68         priority = GetThreadPriority(GetCurrentThread());
69         if (priority != priorities[myNum])
70             priorityChange[myNum] = 1;
71         if (isFinish)
72             break;
73         Sleep(0);
```

```
74      }
75      return 0;
76  }
77  HANDLE initTimer(int sec) {
78      __int64 end_time;
79      LARGE_INTEGER end_time2;
80      HANDLE tm = CreateWaitableTimer(NULL, false, L"timer");
81      end_time = -1 * sec * 1000000;
82      end_time2.LowPart = (DWORD)(end_time & 0xFFFFFFFF);
83      end_time2.HighPart = (LONG)(end_time >> 32);
84      SetWaitableTimer(tm, &end_time2, 0, NULL, NULL, false);
85      return tm;
86  }
87  int getPriorityIndex(DWORD prClass) {
88      for (int i = 0; i < 6; ++i) {
89          if (procPriorities[i] == prClass)
90              return i;
91      }
92      return 0;
93  }
```

Результат работы программы приведен на рисунках 8 –9.

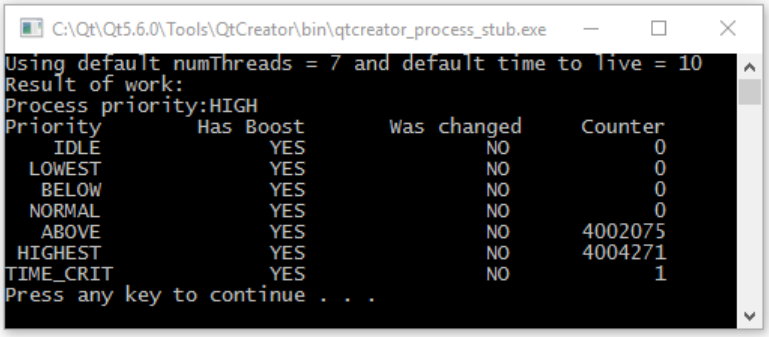

Рис. 8: Результат работы программы task6.cpp на однопроцессорной системе

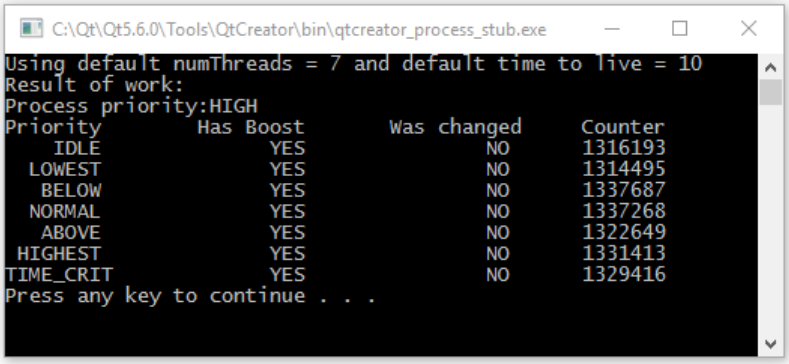

Рис. 9: Результат работы программы task6.cpp при использовании нескольких процессоров

3.5 Функций динамического управления приоритетами

Задача Анализ поведения системных функций динамического управления приоритетами процессов и потоков.

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.

Исходный код программы task_7.cpp:

```
1  #include <stdio.h>
2  #include <iostream>
3  #include <string>
```

```
4  #include <windows.h>
5
6  using namespace std;
7
8  void thread() {
9      while (true) { Sleep(2000); }
10 }
11
12 int main(int argc, char *argv[]) {
13     BOOL dynamic;
14     HANDLE processHandle, mainThread, secondThread;
15     DWORD secondID; processHandle = GetCurrentProcess();
16     mainThread = GetCurrentThread();
17     //create a second thread
18     secondThread = CreateThread(NULL,0, (LPTHREAD_START_ROUTINE) thread,NULL,NULL,&secondID)
       ↪ ;
19     //print default values
20     cout <<"0 - dynamic enable, 1-disabled." <<endl;
21     GetProcessPriorityBoost(processHandle,&dynamic);
22     cout <<"Process dynamically default is " <<dynamic <<endl;
23     GetThreadPriorityBoost(mainThread,&dynamic);
24     cout <<"Main thread dynamic default is " <<dynamic <<endl;
25     GetThreadPriorityBoost(secondThread,&dynamic);
26     cout <<"Second thread dynamic default is " <<dynamic <<endl <<endl;
27     cout <<"Change Second thread priority" <<endl;
28     if(!SetThreadPriorityBoost(secondThread,true)){ //dizable
29         cout <<"Error on thread change!" <<endl;
30         return 2;
31     }
32     GetProcessPriorityBoost(processHandle,&dynamic);
33     cout <<"Process dynamically default is " <<dynamic <<endl;
34     GetThreadPriorityBoost(mainThread,&dynamic);
35     cout <<"Main thread dynamic default is " <<dynamic <<endl;
36     GetThreadPriorityBoost(secondThread,&dynamic);
37     cout <<"Second thread dynamic default is " <<dynamic <<endl <<endl;
38     //change process dinamically
39     if(!SetProcessPriorityBoost(processHandle,true)) {
40         cout <<"Error on prir change!" <<endl;
41         return 1;
42     }
43     cout <<"After process dynamic drop:" <<endl;
44     GetProcessPriorityBoost(processHandle,&dynamic);
45     cout <<"Process is " <<dynamic <<endl;
46     GetThreadPriorityBoost(mainThread,&dynamic);
47     cout <<"Main thread is " <<dynamic <<endl;
48     GetThreadPriorityBoost(secondThread,&dynamic);
49     cout <<"Second thread is " <<dynamic <<endl <<endl;
50     //may be can change thread dynamic prio?
51     if(!SetThreadPriorityBoost(secondThread,false)){ //dizable
52         cout <<"We cannot change if process ban dynamic prio!!!" <<endl;
53         return 3;
54     } else {
55         cout <<"Thread dynamic prio changed, but process bans it!!!" <<endl;
56         GetProcessPriorityBoost(processHandle,&dynamic);
57         cout <<"Process is " <<dynamic <<endl;
58         GetThreadPriorityBoost(mainThread,&dynamic);
59         cout <<"Main thread is " <<dynamic <<endl;
60         GetThreadPriorityBoost(secondThread,&dynamic);
61         cout <<"Second thread is " <<dynamic <<endl;
62     }
63     return 0;
64 }
```

Результат работы программы приведен на рисунке 10. Из листингов видно, что 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 (SetProcessPriorityBoost), то изменение также будет запрещено и для всех потоков. Разрешение отдельному потоку в процессе динамически изменять приоритет возможно, если для процесса это запрещено.

3.6 Самостоятельные задания

Исследуйте результаты работы программы task5.cpp и task6.cpp в зависимости от того, какой приоритет назначается базовому потоку; своими экспериментальными данными заполните таблицу с точ-

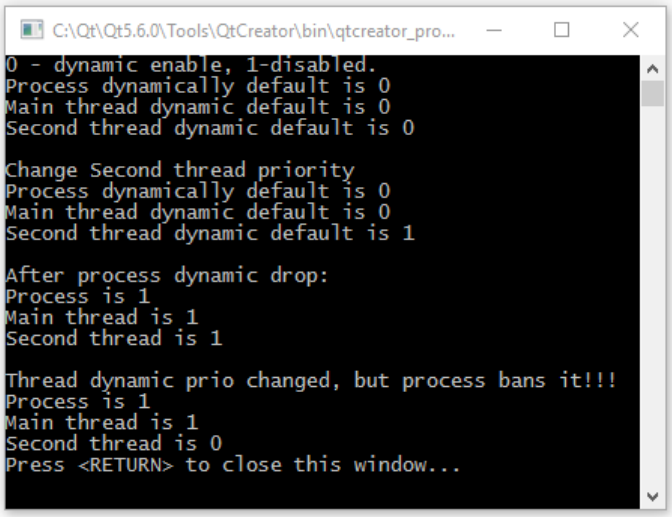

Рис. 10: Результат работы программы task7.cpp

ным указанием для какой ОС и на каком отладочном комплексе проводились измерения.

Исследование проводилось на ОС Windows 10.

	idle	below normal	normal	above normal	highest	time critical
idle	384889470	338750935	370475172	397654353	375165721	367254998
lowest	442457983	409933756	438973987	460370700	388419286	449369600
below normal	453232814	447691472	454739216	468485519	509029435	498882038
normal	471224827	499423558	463177844	463073975	578848964	481554700
above normal	461201143	441116575	468579495	421960961	364916750	468927130
highest	458434552	479824772	502420426	468361146	414371949	463562081
time critical	517431059	494322870	479882926	493014328	491354487	475320493

Таблица 1: Исследование программы task_5.cpp для различных значений приоритета базового потока

По результатам исследования можно сказать, что пропорциональное соотношение между счетчиками потоков примерно сохраняется при любых значениях приоритета базового потока. Однако, судя по результатам, значение приоритета базового потока несильно влияет на время, выделяемое процессу. Возможно, это происходит из-за того, что выполнение программы прерывается другими программами, имеющими больший приоритет.

Модифицируйте программу task5.cpp для заполнения таблицы 2 текущими данными вашего эксперимента. Сделайте выводы. По результатам исследования можно предположить, что в данной системе

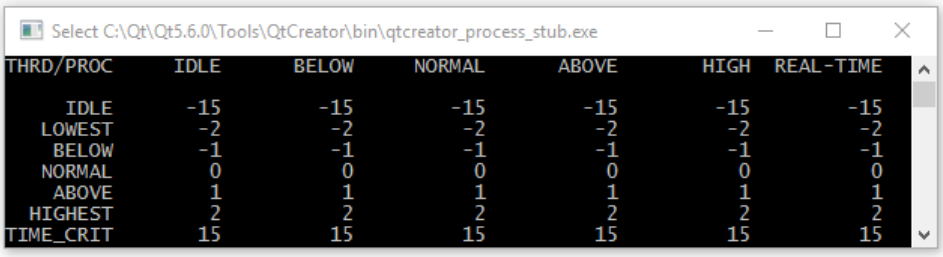

Рис. 11: Примерная зависимость уровня приоритета потока

приоритет потока не зависит от класса потока.

С помощью утилиты CPU Stress test, которая позволяет нагружать систему, и утилиты мониторинга ProcessExplorer() было зафиксировано динамическое изменение приоритетов. В разгруженной системе выполнение программ task5 task6 происходит слишком быстро, мы не успеваем зафиксировать их появления в утилите мониторинга. После запуска утилиты CPU Stress test время работы программ значительно увеличивается.

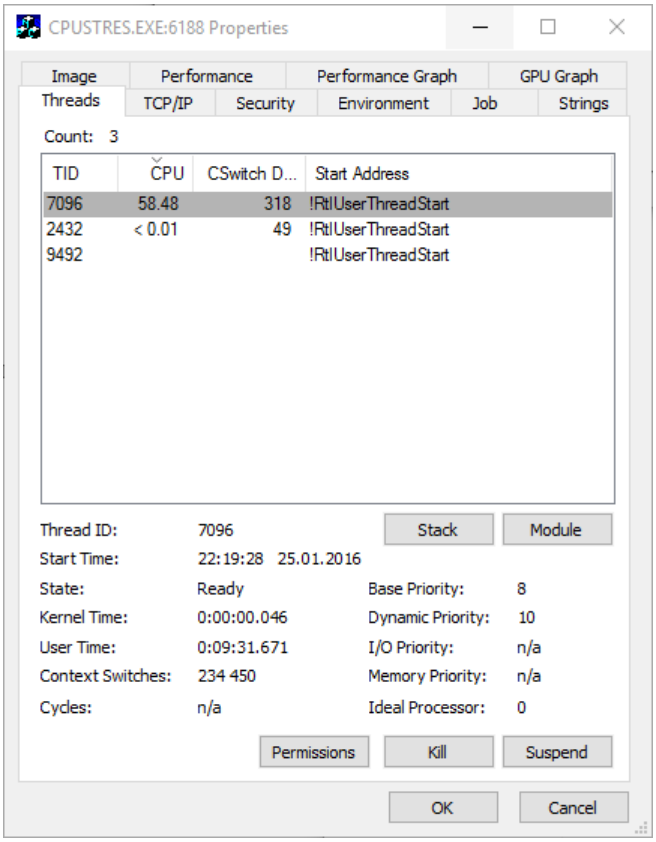

Рис. 12: Динамический приоритет потока в программе CPU Stress

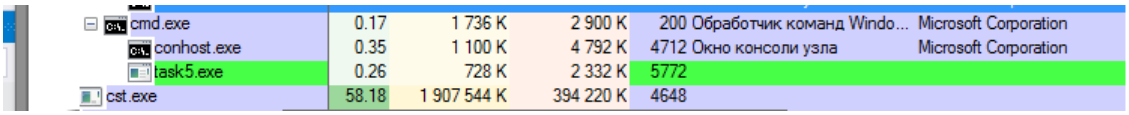

Рис. 13: Утилита мониторинга нагруженной системы

По рисунку видно, что большую часть процессорного времени занимает CPU Stress test (процесс cst.exe). Нашей программе выделяется очень мало процессорного времени.

Создайте программу, демонстрирующую возможность наследования

- дескриптора порождающего процесса,
- дескрипторов открытых файлов,

Для выполнения этого задания следует учесть, что по умолчанию наследование в Windows отключено и для возможности наследования, необходимо:

1. разрешить процессу-потомку наследовать дескрипторы,
2. сделать дескрипторы наследуемыми

Напишем программу, которая создает файл, в который будут записываться данные родительским и дочерним процессом. Программа родитель:

```
1  #define _CRT_SECURE_NO_WARNINGS
2  #include <stdio.h>
3  #include <windows.h>
4  #include <assert.h>
5  #include <stdlib.h>
6  int main(void){
7    TCHAR cmdLine[100];
8    STARTUPINFO si;
9    SECURITY_ATTRIBUTES sa;
10   PROCESS_INFORMATION pi;
11
```

```
12    HANDLE hFile;
13    char DataBuffer[] = "I'm your father!\n";
14    DWORD dwBytesToWrite = (DWORD)strlen(DataBuffer);
15    DWORD dwBytesWritten = 0;
16    printf("\n");
17
18    system("pause");
19    sa.nLength = sizeof(SECURITY_ATTRIBUTES);
20    sa.lpSecurityDescriptor = NULL;
21    sa.bInheritHandle = TRUE;
22
23    hFile = CreateFile(TEXT("file.txt"),        // name of the write
24      GENERIC_WRITE,          // open for writing
25      0,                      // do not share
26      &sa,                    // default security
27      CREATE_NEW,             // create new file only
28      FILE_ATTRIBUTE_NORMAL,  // normal file
29      NULL);                  // no attr. template
30    if (hFile == INVALID_HANDLE_VALUE)
31    {
32      printf("Error #%d", GetLastError());
33      return -1;
34    }
35    BOOL bErrorFlag = WriteFile(
36      hFile,                  // open file handle
37      DataBuffer,             // start of data to write
38      dwBytesToWrite,         // number of bytes to write
39      &dwBytesWritten,        // number of bytes that were written
40      NULL);                  // no overlapped structure
41
42    if (FALSE == bErrorFlag)
43    {
44      printf("Terminal failure: Unable to write to file. Error #%d\n", GetLastError());
45    }
46    wsprintf(cmdLine, TEXT("D:\ConsoleApplication1.exe %d"), (int)hFile);
47    ZeroMemory(&si, sizeof(STARTUPINFO));
48    si.cb = sizeof(STARTUPINFO);
49    if (!CreateProcess(NULL,cmdLine,NULL,NULL,TRUE,CREATE_NEW_CONSOLE,NULL, NULL,&si,&pi)){
50      printf("Error creating process: %d\n", GetLastError());
51      system("pause");
52      return GetLastError();
53    }
54    CloseHandle(pi.hThread);
55    CloseHandle(pi.hProcess);
56    system("pause");
57    CloseHandle(hFile);
58    return 0;
59  }
```

Программа ребёнок:

```
1  #define _CRT_SECURE_NO_WARNINGS
2  #include <stdio.h>
3  #include <windows.h>
4  #include <assert.h>
5  #include <stdlib.h>
6  int main(int argc, char *argv[]){
7    HANDLE hFile;
8    char DataBuffer[] = "I'm your son\n";
9    DWORD dwBytesToWrite = (DWORD)strlen(DataBuffer);
10   DWORD dwBytesWritten = 0;
11   hFile = (HANDLE)atoi(argv[1]);
12   BOOL bErrorFlag = WriteFile(
13     hFile,                  // open file handle
14     DataBuffer,             // start of data to write
15     dwBytesToWrite,         // number of bytes to write
16     &dwBytesWritten,        // number of bytes that were written
17     NULL);                  // no overlapped structure
18
19   if (FALSE == bErrorFlag)
20   {
21     printf("Terminal failure: Unable to write to file. Error #%d\n", GetLastError());
22   }
23   if (CloseHandle(hFile) == 0)
24     printf("Error %d\n", GetLastError());
25   system("pause");
26   return 0;
```

```
27 }
```

Содержимое файла, созданного в текущей директории представлено на рисунке 14. Наличие вывода обеих программ говорит об успешной передаче дескрипторов открытых файлов

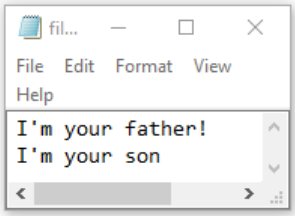

Рис. 14: Утилита мониторинга нагруженной системы

4 Вывод

В данной работе было изучено управление процессами и потоками в ОС Windows.

В отличие от систем Unix, в системе Windows первичными являются потоки, а не процессы. Поток — системный объект операционной системы, реально выполняющийся в процессоре. Процесс — абстрактная структура, определяющая единое адресное пространство и контекст одного или нескольких взаимосвязанных потоков. При создании любого процесса всегда создается первичный поток.

Функции управления процессами и потоками имеют большое количество аргументов, причем некоторые из них достаточно сложные и информационно емкие. Это позволяет производить гибкую настройку параметров создаваемых процессов, однако при первом ознакомлении оказывается достаточно сложным.

Важным аспектом приоритетов является связь приоритета процесса с приоритетом потока и не только ручное регулирование приоритета, а так же и динамическое, осуществляемое операционной системой с целью достижения большей общей производительности.