**«Санкт-Петербургский Государственный Политехнический Университет»**

Кафедра компьютерных систем и программных технологий

**Отчёт по лабораторной работе №5**

Тема: «Процессы в OC Windows»

Дисциплина: «Операционные системы»

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Выполнил студент гр.3530901/70203 | \_\_\_\_\_\_\_\_\_\_\_ | | | Иванов И.Д. |
| Преподав­атель, к.т.н., доц. | \_\_\_\_\_\_\_\_\_\_\_ | | | Душутина Е. В. |
|  |  | “\_\_\_”\_\_\_\_\_\_\_\_\_\_\_\_ 2019 г. | | |
|  |  | |  | |

**1. Цель работы**

Изучение принципов управления процессами в ОС Windows.

**2. Программа работы**

1.1. Создать программу, реализующую создание нового процесса с помощью функции CreateProcess. В новом процессе необходимо запустить любое приложение (например, notepad.exe).

1.2. Создать программу, реализующую создание процессов при работе с конфигурационным файлом. Она должна открывать конфигурационный файл, считывать оттуда строки и создавать отдельный процесс для запуска каждой команды.

1.3. Доработать программу из второго пункта. 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.

2.1. Разработать программу, которая создаёт два потока, выводящих в бесконечном цикле «1» и «2» соответственно. После создания дополнительных потоков, поток-родитель завершается.

2.2. Разработать программу, в которой время жизни процесса и порождаемых в нем потоков задается как параметр. Программа должна получать 2 параметра – количество создаваемых потоков и время жизни всего приложения. С интервалом в 1 секунду каждый рабочий поток выводит о себе информацию и отслеживает состояние переменной, которая устанавливается в заданное значение по истечении времени жизни процесса.

3.1. 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.

3.2. Усложнить задачу из п.3.1. и дополнить ее возможностью управлять классом приоритетов процесса.

3.3. 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.

4.1. Модифицировать программу из п.3.2. для заполнения таблицы зависимости уровня приоритета потока от класса приоритета процесса текущими данными вашего эксперимента. Сделать выводы.

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

**Система, на которой производится работа:**

C:\Windows\system32>ver

Microsoft Windows [Version 10.0.14393]

C:\Windows\system32>gcc --version

gcc (MinGW.org GCC-8.2.0-3) 8.2.0

Кол-во ядер: 4

**3. Выполнение работы**

**1.1. Создать программу, реализующую создание нового процесса с помощью функции CreateProcess. В новом процессе необходимо запустить любое приложение (например, notepad.exe).**

При создании нового процесса 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.

Функция ***CreateProcess( )*** имеет порядка десяти параметров, причем некоторые из них достаточно сложные и информационно емкие. 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 текущего каталога

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.

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[60];

strncpy(LpCommandLine,"C:\\WINDOWS\\system32\\notepad.exe",60);

STARTUPINFO startupInfo;

PROCESS\_INFORMATION processInfo; //информация о процессе

ZeroMemory(&startupInfo, sizeof(STARTUPINFO));

startupInfo.cb = sizeof(startupInfo);

if(!CreateProcess(NULL, LpCommandLine, 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;

}

В данном случае будем использовать значения по умолчанию для атрибутов безопасности процесса и потока – и false для флага наследования.

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). Т.к. указано NULL, потомок использует тоже 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 processInfo хранятся: описатель вновь созданного процесса (hProcess), описатель его базового потока (hThread), глобальный идентификатор процесса (dwProcessId), глобальный идентификатор потока (dwThreadId).

**Результат выполнения программы:**

C:\lab\_5\1\_1>task\_1.exe

Program started

new process Handle: 1788 Handle of thread:7484

Successfully created new process!

Program finished

В результате, был запущен блокнот, как того и требовало задание.

Модифицируем программу так, чтобы вместо блокнота запускался калькулятор.

**Текст модифицированной программы:**

#include <windows.h>

#include <stdio.h>

#include <string.h>

int main(int argc, char\*\* argv[])

{

printf("Program started\n");

char LpCommandLine[60];

strncpy(LpCommandLine,"C:\\WINDOWS\\system32\\calc.exe",60);

STARTUPINFO startupInfo;

PROCESS\_INFORMATION processInfo; //информация о процессе

ZeroMemory(&startupInfo, sizeof(STARTUPINFO));

startupInfo.cb = sizeof(startupInfo);

if(!CreateProcess(NULL, LpCommandLine, 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;

}

**Результат выполнения программы:**

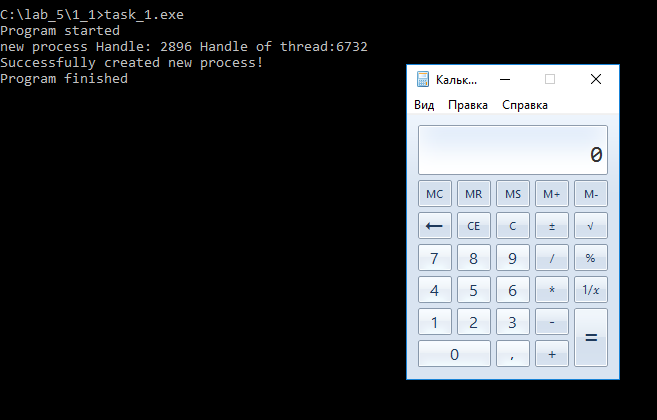


Рис.1. Результат выпаолнения программы *task\_1.exe*.

**1.2. Создать программу, реализующую создание процессов при работе с конфигурационным файлом. Она должна открывать конфигурационный файл, считывать оттуда строки и создавать отдельный процесс для запуска каждой команды.**

**Содержимое конфигурационного файла tmp.txt:**

C:\Windows\System32\notepad.exe C:\lab\_5\1\_2\Read\_me.txt

C:\Windows\System32\calc.exe

**Текст программы:**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <windows.h>

#include <assert.h>

#include <string.h>

#define MAX\_LEN 250

int main(int argc, char\* argv[])

{

const char\* frd = "C:\\lab\_5\\1\_2\\tmp.txt";

FILE \*f=fopen(frd,"r");

if(f==NULL) {

printf("Coudn't open file\n");

system("pause");

return 1;

}

for(int i=0; i<2; i++)

{

char\* execString = (char\*)calloc(MAX\_LEN, sizeof(char));

//выделение памяти

fgets(execString, MAX\_LEN, f); // чтение строки из файла

execString[strlen(execString) - 1] = '\0';

STARTUPINFO startupInfo;

ZeroMemory(&startupInfo, sizeof(STARTUPINFO));

startupInfo.cb = sizeof(startupInfo);

PROCESS\_INFORMATION processInfo;

printf("%s\n", execString);

if( !CreateProcess(NULL, execString, NULL, NULL, false, 0, NULL, NULL, &startupInfo, &processInfo) )

{

printf("Error creating process: %d\n", GetLastError());

system("pause");

return -1;

}

else

printf("Process successfully created!\n");

free(execString);

CloseHandle(processInfo.hThread);

CloseHandle(processInfo.hProcess);

}

system("pause");

return 0;

}

**Результат выполнения программы:**

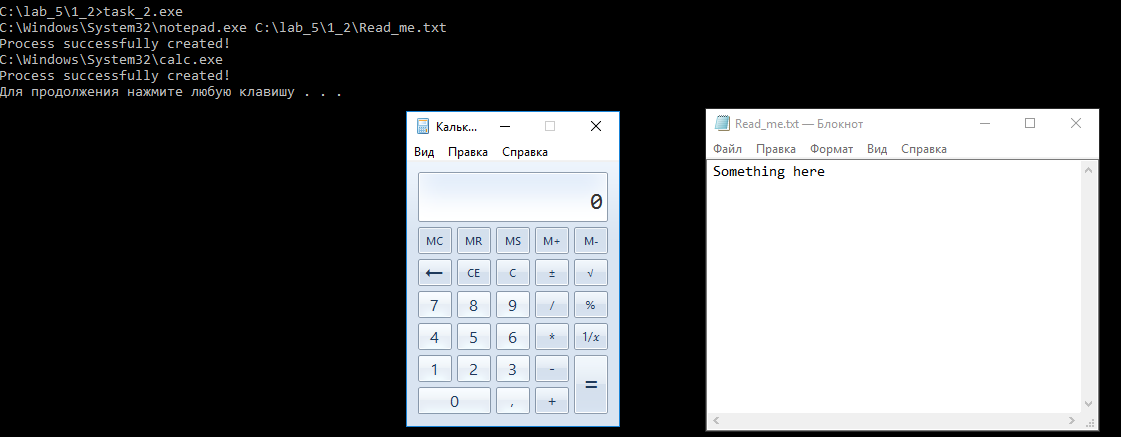
****

Рис.2. Результат выполнения программы *task\_2.exe*.

**1.3. Доработать программу из второго пункта. 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.**

**Содержимое конфигурационного файла tmp.txt:**

C:\Windows\System32\notepad.exe C:\lab\_5\1\_3\Read\_me.txt

C:\Windows\System32\does\_not\_exist.exe

C:\Windows\System32\calc.exe

**Текст программы:**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <windows.h>

#include <assert.h>

#define DEF\_BUFLEN 100

int main(int argc, char\* argv[]) {

printf("Program started\n");

if (argc < 2) {

printf("Input name of configuration file\n");

exit(999);

}

const char\* frd = argv[1];

FILE \*f = fopen(frd, "r"); // открываем конфигурационный файл на чтение

if(f==NULL) {

printf("error opening file %s\n",argv[1]);

exit(1000);

}

char commandLine[DEF\_BUFLEN]; // буфер для читаемой строки

STARTUPINFO StartupInfo;

PROCESS\_INFORMATION ProcessInformation;

while (!feof(f)) {

ZeroMemory(commandLine,DEF\_BUFLEN);

fgets(commandLine, DEF\_BUFLEN, f);

if(strlen(commandLine) <= 1) {

printf("skipping empty string\n");

continue;

}

commandLine[strlen(commandLine) - 1] = '\0';

ZeroMemory(&StartupInfo, sizeof(STARTUPINFO));

StartupInfo.cb = sizeof(STARTUPINFO);

printf("Try to create new process with command line '%s'\n", commandLine);

if( !CreateProcess(NULL, commandLine, NULL, NULL,false, 0,NULL, NULL, &StartupInfo, &ProcessInformation) ){

printf("Can't create new process.Error is: %d\nContinue with next line\n", GetLastError());

continue;

}

printf("new process Handle: %d Handle of thread: %d\n",

ProcessInformation.dwProcessId,ProcessInformation.dwThreadId);

CloseHandle(ProcessInformation.hThread);

CloseHandle(ProcessInformation.hProcess);

}

fclose(f);

printf("Program finished\n");

getchar();

return 0;

}

**Результат выполнения программы:**

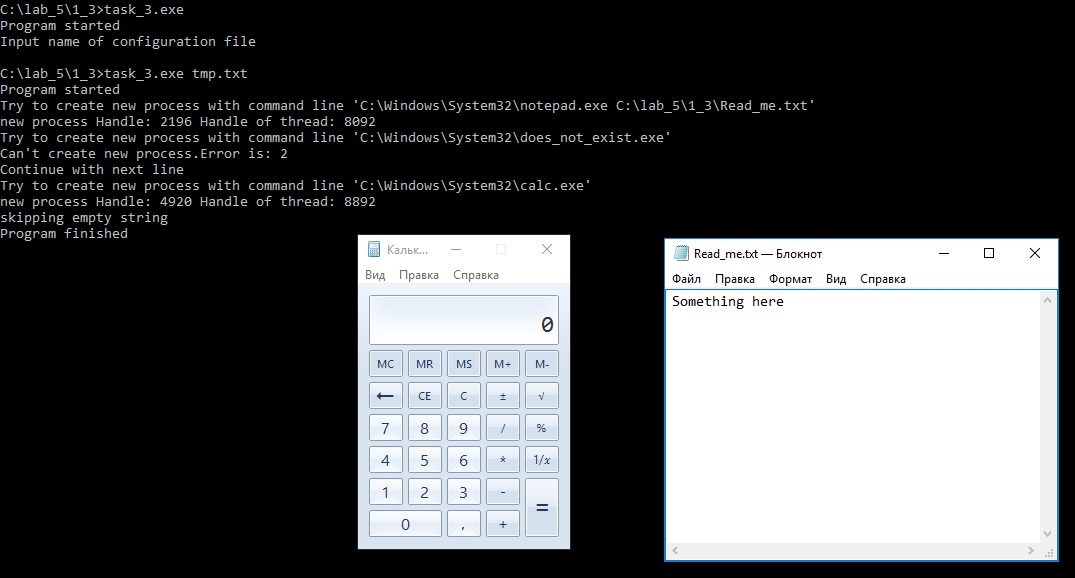


Рис.3. Результат выполнения программы *task\_3.exe*

При попытке запустить программу без аргументов выводится сообщение о том, что необходимо указать имя конфигурационного файла. При указании имени конфигурационного файла программа создаёт новые процессы, если это возможно. В том случае, если создать процесс не удаётся, программа сообщает об этом и пробует читать конфигурационный файл дальше.

**2.1. Разработать программу, которая создаёт два потока, выводящих в бесконечном цикле «1» и «2» соответственно. После создания дополнительных потоков, поток-родитель завершается.**

Потоки создаются посредством функции WinAPI **CreateThread**.

*Синтаксис CreateThread*:

HANDLE CreateThread(

LPSECURITY\_ATTRIBUTES lpsa, //дескриптор защиты

DWORD dwStackSize, // начальный размер стека

LPTHREAD\_START\_ROUTINE lpStartAddr, //функция потока

LPVOID lpThreadParm, //параметр потока

DWORD dwCreationFlags, //опции создания

LPDWORD lpThreadId //идентификатор потока

)

*lpsa* — 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.

*lpStartAddr* — 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.

Функция потока (ThreadFunc) имеет следующую сигнатуру:

DWORD WINAPI ThreadFunc(LPVOID)

*lpThreadParm* — параметр главной функции 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 выполнилась успешно, то вернется описатель потока, если нет, вернется NULL.

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

**Текст программы с завершением с помощью оператора return:**

#include <stdio.h>

#include <windows.h>

DWORD WINAPI threadHandler(LPVOID);

int main(int argc, char\* argv[])

{

printf("Program started\n");

HANDLE t;

int number = 1;

t = CreateThread(NULL, 0, threadHandler,(LPVOID)number, 0, NULL);

CloseHandle(t);

number = 2;

t= CreateThread(NULL, 0, threadHandler,(LPVOID)number, 0, NULL);

CloseHandle(t);

ExitThread(0); // разкомментировать для второго варианта

printf("Program finished\n");

system("pause");

//return 0; // закомментировать для второго варианта

}

DWORD WINAPI threadHandler(LPVOID param){

int number = (long) param;

for(;;) {

Sleep(1000);

printf("%d",number);

fflush(stdout);

}

return 0;

}

**Текст программы с завершением с помощью ExitThread:**

#include <stdio.h>

#include <windows.h>

DWORD WINAPI threadHandler(LPVOID);

int main(int argc, char\* argv[])

{

printf("Program started\n");

HANDLE t;

int number = 1;

t = CreateThread(NULL, 0, threadHandler,(LPVOID)number, 0, NULL);

CloseHandle(t);

number = 2;

t= CreateThread(NULL, 0, threadHandler,(LPVOID)number, 0, NULL);

CloseHandle(t);

ExitThread(0); // разкомментировать для второго варианта

printf("Program finished\n");

system("pause");

//return 0; // закомментировать для второго варианта

}

DWORD WINAPI threadHandler(LPVOID param){

int number = (long) param;

for(;;) {

Sleep(1000);

printf("%d",number);

fflush(stdout);

}

return 0;

}

**Результат выполнения программы:**

С использованием return:

C:\lab\_5\2\_1>task2\_1.exe

Program started

Program finished

Для продолжения нажмите любую клавишу . . . 121212121212121212

С использованием ExitThread:

C:\lab\_5\2\_1>task2\_1\_2.exe

Program started

1212121212121212121212121221^C

В результате, в первом случае произошло завершение всего процесса (возврат из функции main), во втором – завершился только главный поток, а процесс – нет.

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.

Длительность интервала ожидания указывается в милисекундах, и одним из ее возможных значений является INFINITE, что соответствует бесконечному периоду ожидания, при котором выполнение приостанавливается на неопределенное время.

Использование *Sleep()* для синхронизации является очень неудачным решением, т.к. данная функция говорит только то, что по завершению таймаута поток/процесс перейдет в состояние «готов к выполнению». Порядок передачи управления определяет планировщик, поэтому предсказать очередность выполнения потоков не представляется возможным.

**2.2. Разработать программу, в которой время жизни процесса и порождаемых в нем потоков задается как параметр.** **Программа должна получать 2 параметра – количество создаваемых потоков и время жизни всего приложения. С интервалом в 1 секунду 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()***.

HANDLE CreateWaitableTimer(

PSECURITY\_ATTRIBUTES psa,

BOOL fManualReset,

PCTSTR pszName)

*PSECURITY\_ATTRIBUTES psa* – атрибуты безопасности (аналогичны как и для вызовов createThread, createProcess);

*BOOL fManualReset* – определяет тип ожидаемого таймера (со сбросом вручную или автосбросом). Когда освобождается таймер со сбросом вручную, то возобновляется выполнение всех потоков, ожидавших этот объект, а когда в свободное состояние переходит таймер с автосбросом – лишь один из потоков.

*PCTSTR pszName* – имя таймера, по которому можно получить его описатель (может быть равно NULL).

Объекты «ожидаемый таймер» всегда создаются в занятом состоянии. Чтобы сообщить таймеру, в какой момент он должен перейти в свободное состояние, необходимо вызвать функцию ***SetWaitableTimer()***.

BOOL SetWaitableTimer(

HANDLE hTimer,

const LARGE\_INTEGER \*pDueTime,

LONG lPeriod,

PTIMERAPCROUTINE pfnCompletionRoutine,

PVOID pvArgToCotnpletionRoutine,

BOOL fResume)

*HANDLE hTimer – описатель таймера;*

*const LARGE\_INTEGER \*pDueTime* – время первого срабатывания таймера ( используется специальная структура FILETIME);

*LONG lPeriod* – период повторений срабатывания таймера (если равен 0, то сработает 1 раз; считается в миллисекундах);

*BOOL fResume* – позволяет вывести компьютер из режима сна, когда таймер срабатывает (иначе – таймер перейдет в свободное состояние, но ожидавшие его потоки не получат процессорное время, пока компьютер не выйдет из режима сна).

Остальные параметры берутся по умолчанию.

Поток – координатор получает в качестве аргумента созданный таймер и запускает функцию WaitForSingleObject, которая возвращает управление, если объект освободился.

DWORD **WaitForSingleObject**(

HANDLE hObject,

DWORD dwMilliseconds)

*HANDLE hObject* – описатель объекта;

*DWORD dwMilliseconds* – сколько времени в миллисекундах мы готовы ждать освобождения (если используется константа INFINITE, то ожидать будем бесконечно).

Чтобы посмотреть, что возвращает функция, можно использовать *GetLastError*: WAIT\_OBJECT\_O – если послано уведомление; или WAIT\_TIMEOUT – истек тайм-аут.

**Текст программы:**

#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;

}

В этой программе базовый поток ожидает сигнала от таймера (по истечению заданного времени), и после этого устанавливает флаг runFlag, который анализируют другие потоки, и если его значение равно false, то они заканчивают свое выполнение. Для контроля выполнения в начале и в конце программы выводится системное время.

**Результат выполнения программы:**

C:\lab\_5\2\_2>task2\_2.exe 3 10

System Time 19 8 34

0

1

2

1

0

2

0

1

2

0

2

1

0

1

2

0

1

2

0

1

2

0

1

2

0

1

2

System Time 19 8 44

0

1

2

Для продолжения нажмите любую клавишу . . .

Использовать таймер с функциями WaitFor..Object можно не только в разных потоках, но и в разных процессах, т.е. *ожидающие таймеры могут служить средством межпроцессного взаимодействия.*

**3.1. 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.**

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

*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* – тоже потоки быстрого реагирования на события (этот класс присвоен Task Manager).

*Above normal* – класс приоритета промежуточный между normal и high, класс 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, класс введенный, начиная с 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, ОС 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), описания которых указаны в таблице 1.

Таблица 1.

|  |  |
| --- | --- |
| Относительный приоритет потока | Описание |
| 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:

1) после завершения операции ввода/вывода;

2) по окончании ожидания на каком-либо объекте исполнительной системы;

3) при нехватке процессорного времени и инверсии приоритетов.

Рассмотрим эти случаи

1. 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.

2. По окончании ожидания на каком-либо объекте исполнительной системы (например, *SetEvent, ReleaseSemaphore*) приоритет потока увеличивается на один уровень.

3. При инверсии приоритетов диспетчер настройки баланса просматривает очереди готовых потов 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).

*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.

*Для работы с приоритетами* используются следующие функции.

Функция ***SetThreadPriority()*** дает возможность установки базового уровня приоритета потока относительно класса приоритета его процесса.

BOOL SetThreadPriority(

*HANDLE hThread*, // дескриптор потока

int *nPriority* // уровень приоритета потока

);

Функция ***GetThreadPriorityBoost ()*** извлекает значение форсированного (динамически изменяемого) приоритета, который управляет состоянием заданного потока.

BOOL GetThreadPriorityBoost(

*HANDLE hThread*, // дескриптор потока

*PBOOL pDisablePriorityBoost* // состояние динамического изменения приоритета

);

Функция ***SetThreadPriorityBoost()*** разрешает/запрещает динамическое *изменение приоритетов отдельного потока*, не затрагивая остальные потоки.

BOOL SetThreadPriorityBoost(

*HANDLE hThread*, // дескриптор потока *BOOL DisablePriorityBoost* // состояние форсирования приоритета

);

Когда поток запускается в одном из классов динамического приоритета, система временно повышает (форсирует) приоритет потока, чтобы вывести его из состояния ожидания. Если вызывается функция *SetThreadPriorityBoost()* с параметром *DisablePriorityBoost****=***TRUE, приоритет потока не поднимается.

Программная установка флага *priorityBoost* позволяет для каждого из потоков отдельно указать возможность динамического повышения его приоритета. По-умолчанию, *boost* потока и процесса «разрешено».

Рабочие потоки так же могут следить за своим приоритетом, при необходимости обновляя глобальную переменную priorityChange.

*По умолчанию* ОС разрешает *динамическое изменение приоритетов для всего процесса* (то есть для всех его потоков).

Функция ***SetProcessPriorityBoost()*** оказывает влияние на все потоки указанного процесса, но не препятствует дальнейшему разрешению/запрещению динамического изменения приоритетов отдельных потоков.

Иногда для передачи управления другому потоку используют команду *sleep(0),* напомним, в этом случае нужно учитывать, что прогнозировать, какой из потоков запустится, довольно сложно, это осуществляется на усмотрение планировщика из очереди готовых в соответствии со сложившейся ситуацией в системе на текущий момент.

**Текст программы:**

#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

if (argc < 2) stop = 5000;

else stop = atoi(argv[2]);

DWORD targetThreadId;

bool runFlag = true; //инициализация структур потокатаймера

\_\_int64 end\_time;

LARGE\_INTEGER end\_time2;

//создание 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);

// установить приоритет процесса IDLE(или иной)

SetPriorityClass(GetCurrentProcess(),IDLE\_PRIORITY\_CLASS);

//запуск таймера

SetWaitableTimer(tm1, &end\_time2, 0,NULL, NULL, false);

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));

GetThreadPriorityBoost(t1, ptr1);

SetThreadPriorityBoost(t1, true); //проверка динамического распределения приоритетов

CloseHandle(t1); //очистка памяти

}

WaitForSingleObject(tm1,INFINITE);//ожидание потока таймера

runFlag = false; //флаг завершения работы

CloseHandle(tm1);

printf("\n");

for (int i = 0; i < 7; i++) {

printf("%d - %ld\n",i, counters[i]); //вывод результатов

}

system("pause");

return 0;

}

DWORD WINAPI Thread1(LPVOID prm)

{

while(1) {

DWORD WINAPI thrdid = GetCurrentThreadId();

//значение идентификатора вызывающего потока

HANDLE WINAPI handle =

OpenThread(THREAD\_QUERY\_INFORMATION , false, thrdid);

//дескриптор потока

//приоритет для определяемого потока

int WINAPI prio = GetThreadPriority(handle);

params arg = \*((params\*)prm);

counters[arg.num]++;

if(prio != priority[arg.num])

printf("\nPriority of %d is %d %d changed\n", arg.num,

priority[arg.num], prio); //выводится, когда динамическое распределение

// приоритетов включено

Sleep(0);

if(\*(arg.runflg) == false)

break;

}

return 1;

}

**Результат выполнения программы:**

C:\lab\_5\3\_1>task3\_1.exe

0 - 6

1 - 3

2 - 21

3 - 16

4 - 14

5 - 5

6 - 1

Для продолжения нажмите любую клавишу . . .

В результате в консоль выводятся две колонки цифр: слева – номер класса приоритета от низшего к высшему, а справа – значение счетчика, накопленное за все кванты, предоставленные потоку до истечения таймера. В теории, потокам, у которых приоритет ниже, выделяется меньшее количество квантов времени для выполнения, и поэтому их счетчики должны быть меньше. В данном случае динамическое изменение приоритета не запрещено. При каждом запуске экспериментальные данные будут получаться различными, но пропорциональное соотношение между счетчиками потоков примерно сохраняется. Продемонстрируем это, запустив программу ещё несколько раз:

C:\lab\_5\3\_1>task3\_1.exe 0 - 5 1 - 11 2 - 13 3 - 12 4 - 7 5 - 1 6 - 1 Для продолжения нажмите любую клавишу . . .

C:\lab\_5\3\_1>task3\_1.exe

0 - 7

1 - 15

2 - 13

3 - 16

4 - 13

5 - 5

6 – 1

Видно что пропорциональность приоритетов сохраняется.

Задам в качестве аргументов время жизни потоков:

C:\lab\_5\3\_1>task3\_1.exe 7 6

0 - 11

1 - 55254

2 - 698454

3 - 794994

4 - 806907

5 - 811836

6 – 810760

C:\lab\_5\3\_1>task3\_1.exe 7 6

0 - 9

1 - 57688

2 - 750758

3 - 808999

4 - 816217

5 - 819334

6 - 819588

Для продолжения нажмите любую клавишу . . .

Таким образом, пример демонстрирует, как ОС распределяет время процессора между потоками в зависимости от их приоритетов.

**3.2. Усложнить задачу из п.3.1. и дополнить ее возможностью управлять классом приоритетов процесса.**

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). 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.

**Текст программы:**

#include <stdio.h>

#include <conio.h>

#include <windows.h>

#define DEF\_THREADS 7

#define DEF\_TTL 10

DWORD WINAPI threadHandler(LPVOID);

HANDLE initTimer(int sec);

int getPriorityIndex(DWORD prClass);

int isFinish = 0;

long counters[7] = {0,0,0,0,0,0,0};

int priorities[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};

char charPrio[7][10] = {"IDLE", "LOWEST", "BELOW", "NORMAL", "ABOVE", "HIGHEST", "TIME\_CRIT"};

char charProcPrio[6][10] = {"IDLE", "BELOW", "NORMAL", "ABOVE", "HIGH", "REAL-TIME"};

int procPriorities[6] = {IDLE\_PRIORITY\_CLASS,BELOW\_NORMAL\_PRIORITY\_CLASS,NORMAL\_PRIORITY\_CLASS,

ABOVE\_NORMAL\_PRIORITY\_CLASS,HIGH\_PRIORITY\_CLASS,REALTIME\_PRIORITY\_CLASS};

int priorityBoost[7] = {0,0,0,0,0,0,0};

int priorityChange[7] = {0,0,0,0,0,0,0};

int main(int argc, char\* argv[])

{

int numThreads = DEF\_THREADS;

int threadLive = DEF\_TTL;

if(argc < 2)

printf("Using default numThreads = %d and default time to live = %d\n",numThreads,threadLive);

else if(argc < 3)

printf("Using default time to live = %d\n",threadLive);

else {

numThreads = atoi(argv[1]);

threadLive = atoi(argv[2]);

if(numThreads <= 0 || threadLive <= 0) {

printf("All arguments must be numbers!!!!\n");

exit(0);

}

}

HANDLE t = initTimer(threadLive);

HANDLE t1;

// установить приоритет процесса IDLE(или иной)

SetPriorityClass(GetCurrentProcess(),REALTIME\_PRIORITY\_CLASS);

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

t1 = CreateThread(NULL, 0, threadHandler, (LPVOID)i, 0, NULL);

SetThreadPriority(t1, priorities[i]);

SetThreadPriorityBoost(t1,true);

GetThreadPriorityBoost(t1,&priorityBoost[i]);

CloseHandle(t1);

}

//WaitForSingleObject(t,INFINITE); // ОЖИДАТЬ ВСЕХ ПОТОКОВ -

//комментируем или не комментируем строку зависит от поставленной

//задачи

CloseHandle(t);

isFinish = 1;

char hasBoost[4];

char wasChanged[4];

int priorIdx = getPriorityIndex(GetPriorityClass(GetCurrentProcess()));

printf("Result of work:\n");

printf("Process priority:%s\n",charProcPrio[priorIdx]);

printf("Priority\tHas Boost\tWas changed\tCounter\n");

for (int i = 0; i < 7; i++) {

priorityBoost[i] == 0 ? strncpy(hasBoost,"NO",3) : strncpy(hasBoost,"YES",4);

priorityChange[i] == 0 ? strncpy(wasChanged,"NO",3) :

strncpy(wasChanged,"YES",4);

printf("%8s\t%9s\t%10s\t%7d\n",charPrio[i],hasBoost,wasChanged,counters[i]);

}

system("pause");

return 0;

}

DWORD WINAPI threadHandler(LPVOID prm) {

int myNum = (int)prm;

int priority = 0;

for(;;) {

++counters[myNum];

priority = GetThreadPriority(GetCurrentThread());

if(priority != priorities[myNum])

priorityChange[myNum] = 1;

if(isFinish)

break;

Sleep(0);

}

return 0;

}

HANDLE initTimer(int sec) {

\_\_int64 end\_time;

LARGE\_INTEGER end\_time2;

HANDLE tm = CreateWaitableTimer(NULL, false, "timer");

end\_time = -1 \* sec \* 10000000;

end\_time2.LowPart = (DWORD) (end\_time & 0xFFFFFFFF);

end\_time2.HighPart = (LONG) (end\_time >> 32);

SetWaitableTimer(tm, &end\_time2, 0,NULL, NULL, false);

return tm;

}

int getPriorityIndex(DWORD prClass) {

for(int i = 0; i < 6; ++ i) {

if(procPriorities[i] == prClass)

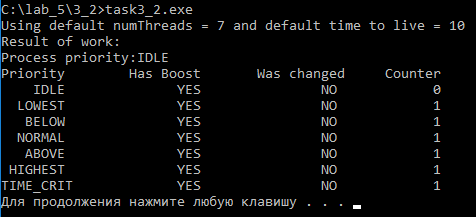
return i;

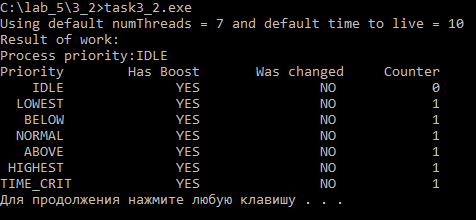
}

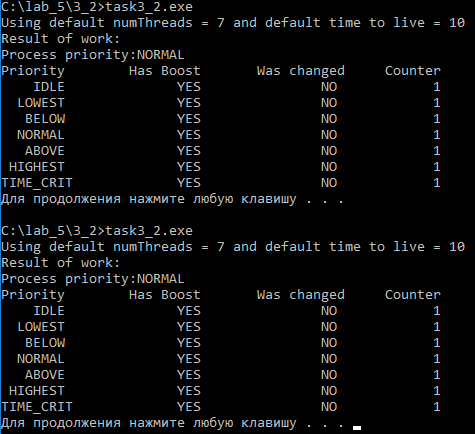
return 0;

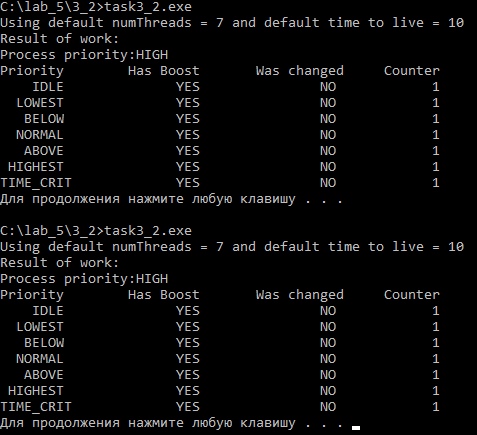
}

**Результат выполнения программы:**









В результате работы программы при использовании нескольких процессоров почти все потоки (кроме потоков с самым низким приоритетом) получают достаточное количество процессорного времени.

При использовании одного процессора потоки с самым высоким приоритетом «отбирают» процессорное время у потоков с более низким, т.е. наблюдается «ресурсное голодание». В ходе эксперимента у всех потоков была включена возможность динамического изменения приоритетов операционной системой (но фактически она не использовалась).

**3.3. 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.**

**Текст программы:**

#include <stdio.h>

#include <string>

#include <windows.h>

void thread() {

while(true) {

Sleep(2000);

}

}

int main(int argc,char \*argv[]) {

BOOL dynamic;

HANDLE processHandle, mainThread, secondThread;

DWORD secondID;

processHandle = GetCurrentProcess();

mainThread = GetCurrentThread();

//create a second thread

secondThread = CreateThread(NULL,0,(LPTHREAD\_START\_ROUTINE) thread,NULL,NULL,&secondID);

//print default values

printf("0 - dynamic enable, 1-disabled.\n");

GetProcessPriorityBoost(processHandle,&dynamic);

printf("Process dynamically default is %d \n",dynamic);

GetThreadPriorityBoost(mainThread,&dynamic);

printf("Main thread dynamic default is %d \n" ,dynamic);

GetThreadPriorityBoost(secondThread,&dynamic);

printf("Second thread dynamic default is %d\n",dynamic);

printf("Change Second thread priority\n");

if(!SetThreadPriorityBoost(secondThread,true)){

//dizable

printf("Error on thread change!\n");

return 2;

}

GetProcessPriorityBoost(processHandle,&dynamic);

printf("Process dynamically default is %d\n" ,dynamic);

GetThreadPriorityBoost(mainThread,&dynamic);

printf("Main thread dynamic default is %d\n" ,dynamic);

GetThreadPriorityBoost(secondThread,&dynamic);

printf("Second thread dynamic default is %d\n\n" ,dynamic);

//change process dinamically

if(!SetProcessPriorityBoost(processHandle,true)) {

printf("Error on prir change!\n");

return 1;

}

printf("After process dynamic drop:\n");

GetProcessPriorityBoost(processHandle,&dynamic);

printf("Process is %d\n" , dynamic);

GetThreadPriorityBoost(mainThread,&dynamic);

printf("Main thread is %d\n" , dynamic);

GetThreadPriorityBoost(secondThread,&dynamic);

printf("Second thread is %d\n\n" , dynamic);

//may be can change thread dynamic prio?

if(!SetThreadPriorityBoost(secondThread,false)){ //dizable

printf("We cannot change if process ban dynamic prio!!!\n");

return 3;

} else {

printf("Thread dynamic prio changed, but process bans it!!!\n");

GetProcessPriorityBoost(processHandle,&dynamic);

printf("Process is %d\n" , dynamic);

GetThreadPriorityBoost(mainThread,&dynamic);

printf("Main thread is %d\n" , dynamic);

GetThreadPriorityBoost(secondThread,&dynamic);

printf("Second thread is %d\n" , dynamic);

}

system("pause");

return 0;

}

**Результат выполнения программы:**

C:\lab\_5\3\_3>task3\_3.exe

0 - dynamic enable, 1-disabled.

Process dynamically default is 0

Main thread dynamic default is 0

Second thread dynamic default is 0

Change Second thread priority

Process dynamically default is 0

Main thread dynamic default is 0

Second thread dynamic default is 1

After process dynamic drop:

Process is 1

Main thread is 1

Second thread is 1

Thread dynamic prio changed, but process bans it!!!

Process is 1

Main thread is 1

Second thread is 0

Для продолжения нажмите любую клавишу . . .

Изначально для процесса и обоих потоков разрешено динамическое изменение приоритетов.

Затем для второго потока было запрещено динамическое изменение приоритета с помощью функции *SetThreadPriorityBoost*.

Если же запретить динамическое изменение приоритетов для процесса, используя аналогичную функцию *SetProcessPriorityBoost*, то динамическое изменение приоритетов будет запрещено ещё и для всех потоков. Разрешение отдельному потоку в процессе динамически изменять приоритет возможно, если это запрещено для процесса.

**4.1. Модифицировать программу из п.3.2. для заполнения таблицы зависимости уровня приоритета потока от класса приоритета процесса текущими данными вашего эксперимента. Сделайте выводы.**

**Текст программы:**

#include <stdio.h>

#include <windows.h>

DWORD WINAPI Thread1(LPVOID);

int stop;

int sleep = 10000;

struct params {

int num;

bool\* runflg;

};

int getPriorityIndex(DWORD prClass);

char charProcPrio[6][10] = { "IDLE", "BELOW", "NORMAL", "ABOVE", "HIGH", "REAL-TIME" };

char charPrio[7][10] = { "IDLE", "LOWEST", "BELOW", "NORMAL", "ABOVE", "HIGHEST", "TIME\_CRIT" };

int procPriorities[6] = { IDLE\_PRIORITY\_CLASS, BELOW\_NORMAL\_PRIORITY\_CLASS, NORMAL\_PRIORITY\_CLASS,

ABOVE\_NORMAL\_PRIORITY\_CLASS, HIGH\_PRIORITY\_CLASS, REALTIME\_PRIORITY\_CLASS };

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

SetPriorityClass(GetCurrentProcess(), ABOVE\_NORMAL\_PRIORITY\_CLASS); //смена класса

int priorIdx = getPriorityIndex(GetPriorityClass(GetCurrentProcess()));

printf("Process priority:%s\n", charProcPrio[priorIdx]);

stop = 10;

DWORD targetThreadId;

bool runFlag = true; //инициализация структур потока-таймера

\_\_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

SetThreadPriority(t1, priority[i]); //задание ему приоритета

PBOOL ptr1 = (PBOOL)malloc(sizeof(BOOL));

GetThreadPriorityBoost(t1, ptr1);

SetThreadPriorityBoost(t1, true); //проверка динамического распределения приоритетов

CloseHandle(t1); //очистка памяти

}

WaitForSingleObject(tm1,INFINITE); //ожидание потока таймера

runFlag = false; //флаг завершения работы

CloseHandle(tm1);

printf("\n");

for (int i = 0; i < 7; i++) {

printf("%d - %ld\n",i, counters[i]); //вывод результатов

}

system("pause");

return 0;

}

DWORD WINAPI Thread1(LPVOID prm) {

while (1) {

DWORD WINAPI thrdid = GetCurrentThreadId(); //идентификатор вызывающего потока

HANDLE WINAPI handle = OpenThread(THREAD\_QUERY\_INFORMATION , false, thrdid); //дескриптор потока

int WINAPI prio = GetThreadPriority(handle); //приоритет для определяемого потока

params arg = \*((params\*)prm);

counters[arg.num]++;

if(prio != priority[arg.num])

printf("\nPriority of %d is %d %d changed\n", arg.num, priority[arg.num], prio); //при динамическом распределении приоритетов

Sleep(0);

if(\*(arg.runflg) == false){

printf("%s, %d\n",charPrio[arg.num],prio);

break;

}

}

return 1;

}

int getPriorityIndex(DWORD prClass) {

for (int i = 0; i < 6; ++i) {

if (procPriorities[i] == prClass)

return i;

}

return 0;

}

**Результат выполнения программы:**

C:\lab\_5\3\_1>task3\_1\_2.exe

Process priority:ABOVE

HIGHEST, 2

ABOVE, 1

0 - 4791

1 - 85280

2 - 1279262

3 - 1357222

4 - 1367816

5 - 1371029

6 - 1372240

BELOW, -1

LOWEST, -2

IDLE, -15

TIME\_CRIT, 15

NORMAL, 0

Для продолжения нажмите любую клавишу . . .

C:\lab\_5\3\_1>task3\_1\_2.exe

Process priority:IDLE

ABOVE, 1

HIGHEST, 2

TIME\_CRIT, 15

0 - 10352

1 - 95098

2 - 1273280

3 - 1351727

4 - 1364671

5 - 1369017

6 - 1368872

IDLE, -15

NORMAL, 0

BELOW, -1

LOWEST, -2

Для продолжения нажмите любую клавишу . . .

C:\lab\_5\3\_1>task3\_1\_2.exe

Process priority:BELOW

HIGHEST, 2

TIME\_CRIT, 15

0 - 10132

1 - 74160

2 - 1291304

3 - 1360272

4 - 1369752

5 - 1372402

6 - 1372644

BELOW, -1

LOWEST, -2

IDLE, -15

ABOVE, 1

NORMAL, 0

Для продолжения нажмите любую клавишу . . .

C:\lab\_5\3\_1>task3\_1\_2.exe

Process priority:NORMAL

NORMAL, 0

BELOW, -1

0 - 5506

1 - 85910

2 - 1293715

3 - 1361345

4 - 1370926

5 - 1373811

6 - 1373470

ABOVE, 1

LOWEST, -2

IDLE, -15

TIME\_CRIT, 15

HIGHEST, 2

Для продолжения нажмите любую клавишу . . .

C:\lab\_5\3\_1>task3\_1\_2.exe

Process priority:HIGH

TIME\_CRIT, 15

HIGHEST, 2

NORMAL, 0

0 - 37

1 - 95047

2 - 1308570

3 - 1355580

4 - 1363671

5 - 1367861

6 - 1366912

BELOW, -1

LOWEST, -2

IDLE, -15

ABOVE, 1

Для продолжения нажмите любую клавишу . . .

C:\lab\_5\3\_1>task3\_1\_2.exe

Process priority:HIGH

TIME\_CRIT, 15

HIGHEST, 2

NORMAL, 0

0 - 217

1 - 108046

2 - 1293989

3 - 1350392

4 - 1360226

5 - 1365952

6 - 1364345

LOWEST, -2

IDLE, -15

ABOVE, 1

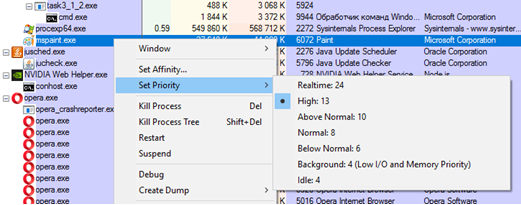
BELOW, -1

Для продолжения нажмите любую клавишу . . .

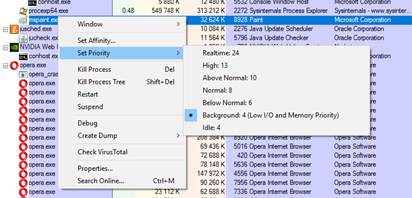
По результатам программы можно сделать вывод о том, что уровень приоритета потока не зависит от класса приоритета процесса в моей ОС Windows 10.

**4.2. С помощью утилит CPU Stress, позволяющих нагружать систему, и утилиты мониторинга ProcessExplorer() (или иных утилит) зафиксировать динамическое изменение приоритетов, привести результаты в отчете.**

Приоритет, когда программа запущенна и в ней ведётся работа:

****

Приоритет, когда программа не используется и свёрнута:



**4. Вывод**

В ходе выполнения данной лабораторной работы были получены навыки управления процессами и потоками в ОС Windows, изучено задание приоритетов процессам и потокам вручную, а также динамически. Также, была рассмотрена зависимость распределения процессорного времени от приоритета потока и приоритета процесса, из которого поток был запущен.