Санкт-Петербургский государственный политехнический университет

Факультет технической кибернетики

Кафедра компьютерных систем и программных технологий

Отчёт по лабораторной работе №5

«Управление процессами в ОС WINDOWS»

Работу выполнил студент группы № 4081/12

Дорофеев Юрий Владимирович

Работу принял преподаватель _____

Малышев Игорь Алексеевич

г. Санкт-Петербург

2012

Содержание работы

1. Цель работы

Используя функции *CreateProcess* и *CreateThread*, создайте:

- один и несколько процессов (каждый с базовым потоком)**;**
- несколько потоков в одном процессе.

2. Программа работы

2.1.Создание процесса для запуска notepad.exe

task_1.cpp:

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

int main(int argc, char** argv[])
{
     char LpCommandLine[60];
     strcpy(LpCommandLine, "C:\\WINDOWS\\system32\\notepad.exe temp.txt");
     STARTUPINFO startupInfo;
     PROCESS_INFORMATION processInfo;
     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("Process successfully created!\n");
    }
   CloseHandle(processInfo.hThread);
   CloseHandle(processInfo.hProcess);
   return 0;
}
```

Запустим программу:

```
E:\task_1\Debug>task_1.exe
Process successfully created!
```

В результате запускается *notepad.exe* и появляется информационное окно, в котором мы указываем - хотим ли мы создать файл *temp.txt*.

В данном случае будем использовать значения по умолчанию для атрибутов безопасности процесса и потока – *NULL* 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.

В структуре *processInfo* хранятся:

- описатель вновь созданного процесса (*hProcess*);
- описатель его базового потока (*hThread*);
- глобальный идентификатор процесса (*dwProcessId*);
- глобальный идентификатор потока (*dwThreadId*).

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

Используем текстовый редактор *notepad* для подготовки конфигурационного файла. Он содержит следующий перечень:

```
C:\\WINDOWS\\system32\\notepad.exe temp.txt
C:\\WINDOWS\\system32\\calc.exe
```

task_2.cpp:

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

int main(int argc, char* argv[])
{
	if (argc < 2)
	{
		printf("Configuration file not found\n");
		exit(-1);
	}
	FILE* f = fopen(argv[1], "r");
	while (!feof(f))
	{
		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());
                  return -1;
            }
            else printf("Process successfully created!\n");

            free(execString);
         CloseHandle(processInfo.hThread);
         CloseHandle(processInfo.hProcess);

      }
      return 0;
}
```

Запустим программу в командной строке и укажем конфигурационный файл:

```
      E:\task_2\Debug>task_2.exe temp.txt
      C:\\WINDOWS\\system32\\notepad.exe temp.txt
      Process successfully created!
      C:\\WINDOWS\\system32\\calc.exe
      Process successfully created!
```

В результате запускается блокнот и калькулятор.

2.3. Создадим два дополнительных потока в нашем процессе. Функция каждого из потоков в бесконечном цикле выводит сообщение о своем выполнении (цифры «1» и «2» соответственно). Базовый поток после создания этих двух дополнительных потоков в бесконечном цикле выводит сообщение о себе (символ «b»). Каждому из трех потоков квант времени на выполнение предоставляется поочередно.

task_3.cpp:

```
#include <Windows.h>
#include <fstream>
using namespace std;

DWORD WINAPI Thread1(PVOID p)
{
      while(1)
      {
            printf("1\n\n");
            Sleep(2000);
      }
```

```
	return 0;
}

DWORD WINAPI Thread2(PVOID p)
{
	while(1)
	{
		printf("2\n\n");
		Sleep(2000);
	}

	return 1;
}

int main (void)
{
	printf("The creation of two threads\n");
	DWORD ThreadID1;
	DWORD ThreadID2;
	CreateThread(NULL, 0, Thread1, NULL, HIGH_PRIORITY_CLASS |
CREATE_NEW_CONSOLE, &ThreadID1);
	CreateThread(NULL, 0, Thread2, NULL, HIGH_PRIORITY_CLASS |
CREATE_NEW_CONSOLE, &ThreadID2);

	while(1)
	{
		printf("b\n\n");
		Sleep(2000);
	}
	return 0;
}
```

Создание потока производится с помощью функции *CreateThread()*:

- Первый аргумент — атрибуты безопасности
- Второй аргумент — начальный размер стека потока
- Третий — точка входа для потока, в качестве неё передаётся адрес функции
- Четвёртый — указатель на аргументы функции в виде указателя на void (*mun LPVOID*)
- Пятый — флаги создания
- Шестой — адрес записи идентификатора потока.

Функции, которые вызываются при запуске потока, выводят каждую секунду соответствующий им символ. Запустим программу:

```
	E:\task_3\Debug>task_3.exe
	The creation of two threads
	b
	1
	2
	2
	b
	1
	1
	2
	b
	<...>
```

На консоль выводится информация о выполнении каждого потока.

2.4. Изменим программу так, чтобы после создания двух дополнительных потоков, базовый поток заканчивал работу.

task_4.cpp:

```
#include <Windows.h>
#include <fstream>
using namespace std;

DWORD WINAPI Thread1(PVOID p)
{
	while(1)
	{
		printf("1\n");
		Sleep(1000);
	}
	return 0;
}

DWORD WINAPI Thread2(PVOID p)
{
	while(1)
	{
		printf("2\n");
		Sleep(1000);
	}
	return 1;
}

int main (void)
{
	printf("The creation of two threads\n");
	DWORD ThreadID1;
	DWORD ThreadID2;
	CreateThread(NULL, 0, Thread1, NULL, HIGH_PRIORITY_CLASS |
CREATE_NEW_CONSOLE, &ThreadID1);
	CreateThread(NULL, 0, Thread2, NULL, HIGH_PRIORITY_CLASS |
CREATE_NEW_CONSOLE, &ThreadID2);
	//Sleep(2000);
	return 2;
}
```

Запустим программу:

```
	E:\task_4\Debug>task_4.exe
	The creation of two threads
	E:\task_4\Debug>
```

Базовый поток выводит сообщение о создании 2 дополнительных потоках, затем создает их и завершает свою работу. Созданные потоки, не успевают вывести информацию о своем выполнении, так как после завершения базового потока, завершаются все остальные потоки, созданные им.

2.5. Изменим программу так, чтобы после создания двух дополнительных потоков, базовый поток вызывал функцию Sleep.

task_4.cpp:

```
#include <Windows.h>
#include <fstream>
using namespace std;

DWORD WINAPI Thread1(PVOID p)
{
	while(1)
	{
		printf("1\n");
		Sleep(1000);
	}
	return 0;
}

DWORD WINAPI Thread2(PVOID p)
{
	while(1)
	{
		printf("2\n");
		Sleep(1000);
	}
	return 1;
}

int main (void)
{
	printf("The creation of two threads\n");
	DWORD ThreadID1;
	DWORD ThreadID2;
	CreateThread(NULL, 0, Thread1, NULL, HIGH_PRIORITY_CLASS |
CREATE_NEW_CONSOLE, &ThreadID1);
	CreateThread(NULL, 0, Thread2, NULL, HIGH_PRIORITY_CLASS |
CREATE_NEW_CONSOLE, &ThreadID2);
	Sleep(2000);
	return 2;
}
```

Запустим программу:

```
	E:\task_4\Debug>task_4.exe
	The creation of two threads
	1
	2
	1
	2
	E:\task_4\Debug>
```

В этом случае созданные потоки выводят информацию о своем выполнении в течение времени работы базового потока.

2.6. Создадим в цикле несколько дополнительных потоков в процессе. В качестве аргументов командной строки зададим число создаваемых потоков и время их существования.

task_6.cpp:

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

DWORD WINAPI Thread1(LPVOID);

int main(int argc, char* argv[])
{
            int num;
            if (argc < 2) num = 2;        //если параметр не задан, то созд. 2
потока
            else num = atoi(argv[1]);

             DWORD targetThreadId;
            for (int i = 0; i < num; i++) {
                  int* param = (int*)malloc(sizeof(int));
                  *param = i;
                  HANDLE t = CreateThread(NULL, 0, Thread1, param, 0,
&targetThreadId); //созд. потока и передача параметра
                  CloseHandle(t);
            }
            while(1) {
                  Sleep(1000);
                  printf("b\n");
            }
        return 0;
}

DWORD WINAPI Thread1(LPVOID prm) {
      while(1) {
                  int arg = *((int*)prm);

                  Sleep(1000);
                  printf("%d\n", arg);
      }
      return 0;
}
```

Запустим программу с различными параметрами:

```
      E:\task_6\Debug>task_6.exe
      0
      b
      1
      0
      1
      b
      ^C
      E:\task_6\Debug>task_6.exe 3
      0
      2
      b
```

```
1
b
0
1
2
```

2.7. Время, которое процесс и потоки в нем должны существовать, зададим как параметр, используя для этого системное время. По истечении этого времени процесс и потоки должны быть завершены.

2.7.1. Один из возможных вариантов решения – выделение отдельного потока–координатора для проверки текущего времени, т.е. после того, как рабочие потоки созданы и выполняют свою работу, поток координатор проверяет текущее время, чтобы определить пороговое значение таймера для завершения. Если оно еще не достигнуто, координатор засыпает на 1 сек, а потом просыпается и проверяет время снова. Если достигнуто – поток-координатор устанавливает глобальную переменную runFlag=FALSE и заканчивается.

task_7_1.cpp:

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

DWORD WINAPI Thread1(LPVOID);
DWORD WINAPI Coordinator(LPVOID);        //поток-координатор

int stop;                                                    //кол-во секунд

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

int main(int argc, char* argv[])
{
            int thrds;
            if (argc < 2) thrds = 2;
            else thrds = atoi(argv[1]);

            if (argc < 2) stop = 5;
            else stop = atoi(argv[2]);

             DWORD targetThreadId;
             bool runFlag = true;

             for (int i = 0; i < thrds; i++)
             {
                  params* param = (params*)malloc(sizeof(params));
                  param->num = i;                        //кол-во потоков
                  param->runflg = &runFlag;
                  HANDLE t1 = CreateThread(NULL, 0, Thread1, param, 0,
&targetThreadId);
                  CloseHandle(t1);
            }
            params* param = (params*)malloc(sizeof(params));
            param->num = -1;
```

```
            param->runflg = &runFlag;
            HANDLE t = CreateThread(NULL, 0, Coordinator, param, 0,
&targetThreadId);
            while(1) {
                  Sleep(1000);
                  printf("b\n");
                  if(!runFlag)
                        break;
            }
        return 0;
}

DWORD WINAPI Thread1(LPVOID prm) {
      while(1) {
                  params arg = *((params*)prm);
                  Sleep(1000);
                  printf("%d\n", arg.num);
                  if(*(arg.runflg) == false)
                        break;
      }
      ExitThread(NULL);
}

DWORD WINAPI Coordinator(LPVOID prm) {
      int i = 0;
      while(1) {
                  params arg = *((params*)prm);
                  Sleep(1000);
                  i++;
                  if(i >= stop) {                          //проверка времени
                        *(arg.runflg) = false;
                        break;
                  }
      }
      ExitThread(NULL);
}
```

Поток-координатор устанавливает флаг *flg*, который при выполнении проверяют остальные потоки и в случае, если он установлен, завершаются. Время жизни в секундах задаётся вторым аргументом командной строки:

```
      E:\task_7_1\Debug>task_7_1.exe 3 2
      1
      b
      0
      2
      0
      2
      1
      b
```

Как видно, было создано 3 потока (0, 1 и 2) и они работали 2 секунды.

2.7.2. Второй способ решения – использование ожидаемого таймера (*WaitableTimer*).

task_7_2.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[])
{
            int thrds;
            if (argc < 2) thrds = 2;
            else thrds = atoi(argv[1]);

            if (argc < 2) 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);
            }
            WaitForSingleObject(tm1,INFINITE);
            runFlag = false;              //установка флага
            CloseHandle(tm1);

        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;
      }
      ExitThread(NULL);
}
```

В этой программе базовый поток ожидает сигнала от таймера и после этого устанавливает флаг *runFlag*, который так же анализируют остальные потоки.

Запустим программу:

```
E:\task_7_2\Debug>task_7_2.exe 3 3
0
2
1
2
0
1
```

Потоки до завершения счёта таймера не успели вывести символы, а были завершены.

2.8. Разработаем программу, которая позволит изменять класс приоритета процесса и приоритеты потоков этого процесса. Проанализируем и приведем экспериментальные данные, как значение приоритета влияет на выделение процессорного времени.

Приоритеты потоков:

task_8_1.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};
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};

int main(int argc, char* argv[])
{
            int thrds;
            if (argc < 2) thrds = 2;
            else thrds = atoi(argv[1]);

            if (argc < 2) stop = 5000;
            else stop = atoi(argv[2]);

             DWORD targetThreadId;
             bool runFlag = true;
              __int64 end_time;
```

```
            LARGE_INTEGER end_time2;

            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 < 7; i++) {
                  params* param = (params*)malloc(sizeof(params));
                  param->num = i;
                  param->runflg = &runFlag;
                  HANDLE t1 = CreateThread(NULL, 0, Thread1, param, 0,
&targetThreadId);
                  SetThreadPriority(t1, priority[i]);            //задание
приоритета
                  PBOOL ptr1 = (PBOOL)malloc(sizeof(BOOL));

                  GetThreadPriorityBoost(t1, ptr1);
                  SetThreadPriorityBoost(t1, true); //отключение динамич.
форсажа

                  //printf("%d", *ptr1);
                  CloseHandle(t1);
            }
            WaitForSingleObject(tm1,INFINITE);
            runFlag = false;
            CloseHandle(tm1);
            printf("\n");
            for (int i = 0; i < 7; i++) {
                  printf("%d - %ld\n",i,  counters[i]);
            }
        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
                  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);
                  Sleep(0);
                  if(*(arg.runflg) == false)
                        break;
      }
      ExitThread(NULL);
}
```

Каждый поток увеличивает соответствующий ему счётчик в каждый квант времени. С помощью функции *Sleep(0)* процесс принудительно завершает свой квант времени.

```
      E:\task_8_1\Debug>task_8_1.exe 3 3
      0 - 6
```

```
1 - 2
2 - 2488
3 - 306444
4 - 308320
5 - 412239
6 - 411094
```

Очевидно, что потоки с меньшим приоритетом увеличивают свой счётчик меньшее число раз, так как им квант времени выделяется реже. Динамическое изменение приоритета по умолчанию включено (это выясняется с помощью вызова *GetThreadPriorityBoost (t1, ptr1)* и чтения *ptr1*). (*True* в этой переменной значит, что динамическое изменение приоритетов выключено, *false* - включено).

Изменение приоритета процесса:

task_8_2.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};
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};

int main(int argc, char* argv[])
{
            int thrds;
            if (argc < 2) thrds = 2;

            else thrds = atoi(argv[1]);
            if (argc < 2) stop = 5000;
            else stop = atoi(argv[2]);

            DWORD targetThreadId;
            bool runFlag = true;
            __int64 end_time;
            LARGE_INTEGER end_time2;
            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);
```

```
            DWORD WINAPI processid = GetCurrentProcessId();//идентификатор
вызывающего процесса
            HANDLE WINAPI prHandle = OpenProcess(PROCESS_QUERY_INFORMATION |
PROCESS_SET_INFORMATION, false, processid);//открытый дескриптор заданного
процесса
            DWORD WINAPI prioProcess = GetPriorityClass(prHandle);     //класс
приоритета заданного процесса
            printf("Process priority before: %d\n", prioProcess);

            SetPriorityClass(prHandle, NORMAL_PRIORITY_CLASS);
      //установка класса приоритета для заданного процесса

            prioProcess = GetPriorityClass(prHandle);
            printf("Process priority after: %d\n", prioProcess);

             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);
                  SetThreadPriority(t1, priority[i]);
                  PBOOL ptr1 = (PBOOL)malloc(sizeof(BOOL));
                  GetThreadPriorityBoost(t1, ptr1);
                  SetThreadPriorityBoost(t1, true);

                  //printf("%d", *ptr1);
                  CloseHandle(t1);
            }
            WaitForSingleObject(tm1,INFINITE);
            runFlag = false;
            CloseHandle(tm1);
            printf("\n");
            for (int i = 0; i < 7; i++) {
                  printf("%d - %ld\n",i,  counters[i]);
            }
        return 0;
}

DWORD WINAPI Thread1(LPVOID prm) {
      while(1) {
                  DWORD WINAPI thrdid = GetCurrentThreadId();
                  HANDLE WINAPI handle = OpenThread(THREAD_QUERY_INFORMATION ,
false, thrdid);
                  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);
                  Sleep(0);
                  if(*(arg.runflg) == false)
                        break;
      }
      ExitThread(NULL);
}
```

Запустим программу:

```
E:\task_8_2\Debug>task_8_2.exe 3 3
Process priority before: 32
Process priority after: 32
0 - 49
1 - 26
2 - 3778
3 - 298768
4 - 333242
5 - 333924
6 - 316774
```

Исходя из вывода программы ясно, что приоритет по умолчанию для процесса 32, то есть *NORMAL_PRIORITY_CLASS.*

Изменим класс приоритета на *ABOVE_NORMAL_PRIORITY_CLASS*:

```
E:\task_8_2\Debug>task_8_2.exe 3 3
Process priority before: 32
Process priority after: 32768
0 - 80
1 - 21
2 - 360
3 - 338865
4 - 321591
5 - 340427
6 - 322690
```

Изменим класс приоритета на *HIGH_PRIORITY_CLASS*:

```
E:\task_8_2\Debug>task_8_2.exe 3 3
Process priority before: 32
Process priority after: 128
0 - 13
1 - 31
2 - 21
3 - 341157
4 - 338144
5 - 342042
6 - 338745
```

Потоки в этом процессе немедленно реагируют на события.

3. Выводы

1. В WinAPI есть функции *CreateProcess()* и *CreateThread()* с большим числом параметров, что даёт большую гибкость при создании потоков и процессов.

2. При создании процесса функции *CreateProcess()* и *CreateThread()* возвращают описатель созданного процесса или потока. Он необходим для совершения любых операций над процессом или потоком.

3. С помощью объектов класса *WaitableTimer* можно управлять поведением потоков или других объектов используя временные задержки.

4. В ОС *Windows* есть средства задания приоритета процессов и потоков. Приоритеты могут задаваться вручную или изменяться динамически с помощью механизма динамических приоритетов.