Процессы в Windows

- В Windows под *процессом* понимается объект ядра, которому принадлежат системные ресурсы, используемые исполняемым приложением. Поэтому можно сказать, что в Windows процессом является исполняемое приложение.

- Каждый процесс в операционной системе Windows владеет следующими ресурсами:

- ✔ виртуальным адресным пространством;
- ✔ рабочим множеством страниц в реальной памяти;
- ✔ маркером доступа, содержащим информацию для системы безопасности;
- ✔ таблицей для хранения дескрипторов объектов ядра.

- Новый процесс в Windows создается вызовом функции **CreateProcess,** которая имеет следующий прототип:

```
BOOL CreateProcess(
LPCTSTR lpApplicationName, // имя исполняемого модуля
LPTSTR lpCommandLine,   // командная строка
LPSECURITY_ATTRIBUTES lpProcessAttributes, // защита процесса
LPSECURITY_ATTRIBUTES lpThreadAttributes, // защита потока
BOOL blnheritHandles,   // признак наследования дескриптора
DWORD dwCreationFlags,   // флаги создания процесса
LPVOID IpEnvironment, // блок новой среды окружения
LPCTSTR lpCurrentDirectory, // текущий каталог
LPSTARTUPINFO IpStartUpInfo, // вид главного окна
LPPROCESS_INFORMATION lpProcessInformation // информация о процессе
) ;
```

```
#include <conio.h>

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

  _cputs("I am created.");

  _cputs("\nMy name is: ");
  _cputs(argv[0]);

  for (i = 1; i < argc; ++i)
```

```
    _cprintf ("\n My %d parameter =  %s", i, argv[i]);

  _cputs("\nPress any key to finish.\n");
  _getch();

  return 0;
}
```

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

int main()
{
  char lpszAppName[] = "C:\\ConsoleProcess.exe";

  STARTUPINFO si;
  PROCESS_INFORMATION piApp;

  ZeroMemory(&si, sizeof(STARTUPINFO));
  si.cb = sizeof(STARTUPINFO);
```

```
// создаем новый консольный процесс
if (!CreateProcess(lpszAppName, NULL, NULL, NULL, FALSE,
    CREATE_NEW_CONSOLE, NULL, NULL, &si, &piApp))
{
  _cputs("The new process is not created.\n");
  _cputs("Check a name of the process.\n");
  _cputs("Press any key to finish.\n");
  _getch();
  return 0;
}

_cputs("The new process is created.\n");
```

```
    // ждем завершения созданного процесса
    WaitForSingleObject(piApp.hProcess, INFINITE);
    // закрываем дескрипторы этого процесса в текущем процессе
    CloseHandle(piApp.hThread);
    CloseHandle(piApp.hProcess);

    return 0;
}
```

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

int main()

{
  char lpszCommandLine[] = "C:\\ConsoleProcess.exe p1 p2 p3";

  STARTUPINFO si;
  PROCESS_INFORMATION piCom;

  ZeroMemory(&si, sizeof(STARTUPINFO));
  si.cb = sizeof(STARTUPINFO);
```

```
    // создаем новый консольный процесс
    CreateProcess(NULL, lpszCommandLine, NULL, NULL, FALSE,
        CREATE_NEW_CONSOLE, NULL, NULL, &si, &piCom);
    // закрываем дескрипторы этого процесса
    CloseHandle(piCom.hThread);
    CloseHandle(piCom.hProcess);

    _cputs("The new process is created.\n");
    _cputs("Press any key to finish.\n");
    _getch();

    return 0;
}
```

- При использовании параметра **lpCommandLine** система для запуска нового процесса осуществляет поиск требуемого exe-файла в следующей последовательности каталогов:

✔ каталог, из которого запущено приложение;

✔ текущий каталог родительского процесса;

✔ системный каталог Windows;

✔ каталог Windows;

✔ каталоги, которые перечислены в переменной PATH среды окружения.

```
#include <windows.h>
#include <iostream.h>
int main() {
STARTUPINFO si;
PROCESS_INFORMATION pi;
// заполняем значения структуры STARTUPINFO по умолчанию
ZeroMemory (&si, sizeof(STARTUPINFO));
si.cb = sizeof(STARTUPINFO);
```

```
// запускаем процесс Notepad
if (!CreateProcess(
NULL, // имя не задаем
"Notepad.exe", // имя программы
NULL, // атрибуты защиты процесса устанавливаем по умолчанию
NULL, // атрибуты защиты первичного потока по умолчанию
FALSE, // дескрипторы текущего процесса не наследуются
0, //по умолчанию NORMAL_PRIORITY_CLASS
NULL, // используем среду окружения вызывающего процесса
NULL, // текущий диск и каталог, как и в вызывающем процессе
&si,   // вид главного окна - по умолчанию
&pi   // информация о новом процессе
)
)
```

```
{
cout « "The new process is not created." « endl
« "Check a name of the process." « endl;
return 0;
}
Sleep(1000); // немного подождем и закончим свою работу
// закроем дескрипторы запущенного процесса в текущем процессе
CloseHandie(pi.hThread);
CloseHandie(pi.hProcess);
return 0;
}
```

- Процесс может завершить свою работу вызовом функции **ExitProcess,** которая имеет следующий прототип:

```
VOID ExitProcess(
UINT uExitCode // код возврата из процесса
);
```

- Один процесс может быть завершен другим при помощи вызова функции **TerminateProcess,** которая имеет следующий прототип:

```
BOOL TerminateProcess(
HANDLE hProcess,   // дескриптор процесса
UINT uExitCode   // код возврата
);
```

Наследование дескрипторов

- Свойство наследования объекта означает, что если наследуемый объект создан или открыт в некотором процессе, то к этому объекту будут также иметь доступ все процессы, которые создаются этим процессом, т.е. являются его потомками. Свойство наследования объекта определяется его дескриптором, который также может быть *наследуемым* или *ненаследуемым.* Для того чтобы объект стал наследуемым, необходимо сделать наследуемым его дескриптор и наоборот.

- Не могут наследоваться следующие дескрипторы:

✔ дескриптор виртуальной памяти, который возвращает любая из функций LocalAlloc, GlobalAlloc, HeapCreate или HeapAlloc;

✔ дескриптор динамической библиотеки, который возвращает функция LoadLibrary.

```
#include <windows.h>
#include <conio.h>
int main(int argc, char *argv[])
{
HANDLE hThread;
char c;
// преобразуем символьное представление дескриптора в число
hThread = (HANDLE)atoi(argv[1]) ;
// ждем команды о завершении потока
while (true)
{
_cputs("Input 't' to terminate the thread: ");
c = _getch();
```

```
if (c == ‘t’)
{
_cputs("t\n") ;
break;
}
// завершаем поток
TerminateThread(hThread, 0);
// закрываем дескриптор потока
CloseHandle(hThread);
_cputs("Press any key to exit.\n");
_getch();
return 0;
}
```

```
#include <windows.h>
#include <conio.h>
volatile int count;
void thread()
{
for (;;)
{
count++;
Sleep(500);
_cprintf("count = %d\n", count);
}
}
int main()
{
char IpszComLine[80]; // для командной строки
STARTUPINFO si;
PROCESS_INFORMATION pi;
SECURITY_ATTRIBUTES sa;
HANDLE hThread;
DWORD IDThread;
```

```
_cputs("Press any key to start the count-thread.\n");
_getch();
// устанавливаем атрибуты защиты потока
sa.nLength = sizeof(SECURITY_ATTRIBUTES);
sa.IpSecurityDescriptor = NULL; // защита по умолчанию
sa.blnheritHandle = TRUE;// дескриптор потока наследуемый
// запускаем поток-счетчик
hThread = CreateThread(&sa, 0, thread, NULL, 0, &IDThread);
if (hThread == NULL)
return GetLastError();
// устанавливаем атрибуты нового процесса
ZeroMemory (&si, sizeof(STARTUPINFO));
si.cb=sizeof(STARTUPINFO);
```

```
// формируем командную строку
wsprintf(lpszComLine, "C:\\ConsoleProcess.exe %d", (int)hThread);
// запускаем новый консольный процесс
if (!CreateProcess(
NULL, // имя процесса
lpszComLine, // адрес командной строки
NULL, // атрибуты защиты процесса по умолчанию
NULL, // атрибуты защиты первичного потока по умолчанию
TRUE, // наследуемые дескрипторы текущего процесса наследуются новым процессом CREATE_NEW_CONSOLE, // новая консоль
NULL, // используем среду окружения процесса предка
NULL, // текущий диск и каталог, как и в процессе-предке
&si,// вид главного окна - по умолчанию
&pi// здесь будут дескрипторы и идентификаторы нового процесса и его первичного потока
)
)
```

```
{
_cputs("The new process is not created.\n");
_cputs("Press any key to finish.\n");
_getch();
return GetLastError();
}
// закрываем дескрипторы нового процесса
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
// ждем закрытия потока-счетчика
WaitForSingleObject(hThread, INFINITE);
_cputs("Press any key to exit.\n");
_getch();
// закрьгоаем дескриптор потока
CloseHandle(hThread);
return 0;
}
```

- Функция для изменения свойств дескрипторов имеет прототип

```
BOOL SetHandleInformation(
HANDLE hObject,     // дескриптор объекта
DWORD dwMask,       // флаги, которые изменяем
DWORD dwFlags       // новые значения флагов
);
```

- В случае успешного завершения эта функция возвращает ненулевое значение, в противном случае — **FALSE.**

Псевдодескрипторы процессов

- *Псевдодескриптор текущего процесса* отличается от настоящего дескриптора процесса тем, что он может использоваться только текущим процессом и не может наследоваться другими процессами. Псевдодескриптор процесса не нужно закрывать после его использования. Из псевдодескриптора процесса можно получить настоящий дескриптор процесса: для этого псевдодексриптор нужно продублировать, вызвав функцию **DuplicateHandle**.

Обслуживание потоков

- Приоритеты потоков в Windows определяются относительно приоритета процесса, в контексте которого они исполняются, и изменяются от 0 (низ ший приоритет) до 31 (высший приоритет). Приоритет процессов устанав ливается при их создании функцией **CreateProcess**, используя параметр **dwCreationFiags.**

- **IDLE_PRIORITY_CLASS** — КЛАСС ФОНОВЫХ ПРОЦЕССОВ;
- **BELOW_NORMAL_PRIORITY_CLASS** — КЛАСС ПРОЦЕССОВ НИЖЕ НОРМАЛЬНЫХ;
- **NORMAL_PRIORITY_CLASS** — КЛАСС НОРМАЛЬНЫХ ПРОЦЕССОВ;
- **ABOVE_NORMAL_PRIORITY_CLASS** — КЛАСС ПРОЦЕССОВ ВЫШЕ НОРМАЛЬНЫХ;
- **HIGH_PRIORITY_CLASS** — КЛАСС ВЫСОКОПРИОРИТЕТНЫХ ПРОЦЕССОВ;
- **REAL_TIME_PRIORITY_CLASS** — КЛАСС ПРОЦЕССОВ РЕАЛЬНОГО ВРЕМЕНИ

- *Фоновые процессы* выполняют свою работу, когда нет активных пользовательских процессов. Обычно эти процессы следят за состоянием системы.
- *Процессы с нормальным приоритетом* — это обычные пользовательские процессы. Этот приоритет также назначается пользовательским процессам по умолчанию.
- *Процессы с высоким приоритетом* это такие пользовательские процессы, от которых требуется более быстрая реакция на некоторые события, чем от обычных пользовательских процессов. Эти процессы должны содержать небольшой программный код и выполняться очень быстро, чтобы не замедлять работу системы. Обычно такие приоритеты имеют другие системы, работающие на платформе операционных систем Windows.
- К последнему типу процессов относятся *процессы реального времени.* Работа таких процессов обычно происходит в масштабе реального времени и связана с реакцией на внешние события. Эти процессы должны работать непосредственно с аппаратурой компьютера.

- Приоритет процесса можно изменить при помощи функции **setPriorityClass**, которая имеет следующий прототип:

```
BOOL SetPriorityClass(
HANDLE hProcess,   // дескриптор процесса
DWORD dwPriorityClass // приоритет
);
```

- При успешном завершении функция **SetPriorityClass** возвращает ненулевое значение, в противном случае значение — **FALSE**. Параметр **dwPriorityClass** этой функции должен быть равен одному из флагов, которые приведены выше.

Таблица 1 - *Базовые приоритеты потоков*

	Real time	High	Above normal	Normal	Below normal	Idle
Time critical	31	15	15	15	15	15
Highest	26	15	12	10	8	6
Above normal	25	14	11	9	7	5
Normal	24	13	10	8	6	4
Below normal	23	12	9	7	5	3
Lowest	22	11	8	6	4	2
Idle	16	1	1	1	1	1