

Управление центральным процессором...

API Win32 для создания и завершения процессов

Порядок создания процесса

- Вызов функции *CreateProcess* ().
- Система создает объект ядра «процесс» с начальным значением счетчика числа его пользователей, равным 1.
- Затем система создает для нового процесса виртуальное адресное пространство и загружает в него код и данные как для исполняемого файла, так и для любых DLL (если таковые требуются).
- Далее система формирует объект ядра «поток» (со счетчиком, равным 1) для первичного потока нового процесса.
- Первичный поток начинает с исполнения стартового кода из библиотеки C/C++, который в конечном счете вызывает функцию *WinMain* или *main* в Вашей программе.
- Если системе удастся создать новый процесс и его первичный поток, функция *CreateProcess* () вернет TRUE.



Функция создания процесса

```
BOOL CreateProcess (  
    PCTSTR pszApplicationName, // имя исполняемого файла  
    PTSTR pszCommandLine, // командная строка  
    PSECURITY_ATTRIBUTES psaProcess, // атрибуты защиты процесса  
    PSECURITY_ATTRIBUTES psaThread, // атрибуты защиты потоков  
    BOOL bInheritHandles, // наследование дескрипторов  
    DWORD fdwCreate, // флаги  
    PVOID pvEnvironment, // блок памяти с переменными окружения  
    PCTSTR pszCurDir, // текущий диск и каталог для процесса  
    PSTARTUPINFO psiStartInfo, // используется Windows-функциями  
    // при создании нового процесса  
    PPROCESS_INFORMATION ppiProcInfo // инициализируемая  
    структура  
);
```



Результат выполнения *CreateProcess*

- ▣ *CreateProcess ()* возвращает TRUE до окончательной инициализации процесса. Это означает, что на данном этапе загрузчик ОС еще не искал все необходимые DLL.
- ▣ Если он не сможет найти хотя бы одну из DLL или корректно провести инициализацию, процесс завершится.
- ▣ Но, поскольку *CreateProcess ()* уже вернула TRUE, родительский процесс ничего не узнает об этих проблемах.



Параметры *CreateProcess*

- ❑ Параметры ***pszApplicationName*** и ***pszCommandLine***. Эти параметры определяют имя исполняемого файла, которым будет пользоваться новый процесс, и командную строку, передаваемую этому процессу.
- ❑ Параметры ***psaProcess***, ***psaThread***. Параметры ***psaProcess*** и ***psaThread*** позволяют определить нужные атрибуты защиты для объектов «процесс» и «поток» соответственно. Если передать NULL, то система будет использовать дескрипторы защиты по умолчанию.
- ❑ Параметр ***blInheritHandles*** позволяет разрешить (TRUE) дочернему процессу наследовать дескрипторы объектов родительского процесса с правами доступа как у оригиналов.
- ❑ Параметр ***fdwCreate*** определяет класс приоритета процесса и флаги, влияющие на то, как именно создается новый процесс. Флаги комбинируются булевым оператором OR.
- ❑ Параметр ***pvEnvironment*** указывает на блок памяти, хранящий строки переменных окружения, которыми будет пользоваться новый процесс. Если его значение – NULL, то дочерний процесс наследует строки переменных окружения от родительского процесса.
- ❑ Параметр ***pszCurDir*** позволяет родительскому процессу установить текущие диск и каталог для дочернего процесса. Если его значение – NULL, рабочий каталог нового процесса будет тем же, что и у приложения, его породившего.
- ❑ Параметр ***psiStartInfo*** указывает на структуру STARTUPINFO. Элементы структуры STARTUPINFO используются Windows-функциями при создании нового процесса.
- ❑ Параметр ***ppiProcInfo*** указывает на структуру типа PROCESS_INFORMATION, которую

Параметр *fdwCreate* (1)

- Параметр *fdwCreate* позволяет задать класс приоритета процесса:
 - IDLE_PRIORITY_CLASS
 - BELOW_NORMAL_PRIORITY_CLASS
 - NORMAL_PRIORITY_CLASS
 - ABOVE_NORMAL_PRIORITY_CLASS
 - HIGH_PRIORITY_CLASS
 - REALTIME_PRIORITY_CLASS



Параметр *fdwCreate* (2)

- ❑ Флаги **DEBUG_PROCESS** и **DEBUG_ONLY_THIS_PROCESS** позволяют родительскому процессу проводить отладку дочернего процесса, а также всех процессов, которые последним могут быть порождены (**DEBUG_PROCESS**).
- ❑ Флаг **CREATE_SUSPENDED** позволяет создать процесс и в то же время приостановить его первичный поток. Это позволяет родительскому процессу модифицировать содержимое памяти в адресном пространстве дочернего, изменить приоритет и т.д. Для разрешения выполнения приостановленного потока следует использовать функцию *ResumeThread ()*.
- ❑ Флаг **DETACHED_PROCESS** блокирует доступ дочернего процесса к консольному окну родительского процесса и сообщает системе, что вывод следует перенаправить в новое консольное окно.
- ❑ Флаг **CREATE_NEW_CONSOLE** приводит к созданию нового консольного окна для нового процесса.
- ❑ Флаг **CREATE_NO_WINDOW** позволяет создать процесс без пользовательского интерфейса.
- ▶ ❑ Флаг **CREATE_BREAKAWAY_FROM_JOB** позволяет процессу, включенному в задание, создать новый процесс, отделенный от этого задания.

Завершение процессов

- Существует 4 гипотетических варианта завершения процессов:
 - входная функция первичного потока возвращает управление (рекомендуемый способ);
 - один из потоков процесса вызывает функцию *ExitProcess ()* (нежелательный способ);
 - поток другого процесса вызывает функцию *TerminateProcess ()* (тоже нежелательно);
 - все потоки процесса сами «умирают» (большая редкость).
- Явный вызов *ExitProcess ()* и *TerminateProcess ()* – распространенная ошибка, которая мешает правильной очистке ресурсов.



Действия при возврате управления входной функцией первичного потока

- любые C++-объекты, созданные данным потоком, уничтожаются соответствующими деструкторами;
- система освобождает память, которую занимал стек потока;
- система устанавливает код завершения процесса – его и возвращает Ваша входная функция первичного потока (оператор *return*) ;
- счетчик пользователей данного экземпляра объекта ядра типа «процесс» уменьшается на 1.



Функция *ExitProcess*

- Процесс завершается, когда один из его потоков вызывает *ExitProcess* ().

`VOID ExitProcess (UINT fuExitCode);`

- Эта функция завершает процесс с кодом завершения *fuExitCode*.



Функция *TerminateProcess*

- Функция *TerminateProcess* () позволяет завершить произвольный процесс с дескриптором *hProcess* с кодом завершения *fuExitCode*.

BOOL *TerminateProcess*
(HANDLE *hProcess*, UINT *fuExitCode*);

- Функция *TerminateProcess* () является асинхронной, т.е. она сообщает системе, что Вы хотите завершить процесс, но к тому времени, когда она вернет управление, процесс может быть еще не уничтожен.
- Чтобы узнать момент завершения процесса, используйте *WaitForSingleObject* () или аналогичную функцию, передав ей дескриптор завершаемого процесса.

Когда все потоки процесса «уходят»

- Обнаружив, что в процессе не исполняется ни один поток, операционная система немедленно завершает его.
- При этом код завершения процесса приравнивается коду завершения последнего потока.



Действия при завершении процесса

1. Выполнение всех потоков в процессе прекращается.
2. Все User- и GDI-объекты, созданные процессом, уничтожаются, а объекты ядра закрываются (если их не использует другой процесс).
3. Код завершения процесса меняется со значения `STILL_ACTIVE` на код, переданный в *ExitProcess* или *TerminateProcess*.
4. Объект ядра «процесс» переходит в свободное (незанятое) состояние.
5. Счетчик объекта ядра «процесс» уменьшается на 1.

`BOOL GetExitCodeProcess`
`(HANDLE hProcess, PDWORD pdwExitCode);`



Управление динамическими приоритетами потоков процесса

```
BOOL SetProcessPriorityBoost(  
    HANDLE hProcess,          // дескриптор процесса  
    BOOL DisablePriorityBoost // состояние форсированного  
    // приоритета  
);  
  
BOOL GetProcessPriorityBoost(  
    HANDLE hProcess,          // дескриптор процесса  
    PBOOL pDisablePriorityBoost // состояние форсированного  
    // приоритета  
);
```

- Для успешного выполнения функций процесс должен иметь привилегию **PROCESS_SET_INFORMATION**.



Самостоятельная работа

- Построение дерева запущенных процессов
 - <http://www.rsdn.ru/article/qna/baseserv/enumproc.xml>
- Как найти родителя процесса
 - <http://forum.sources.ru/index.php?showtopic=209024>
- Завершение дерева процессов
 - <http://www.rsdn.ru/article/qna/baseserv/killproc.xml>



Управление центральным процессором...

API Win32 для управления потоками

Функция создания потока

```
HANDLE CreateThread (  
    PSECURITY_ATTRIBUTES psa, // атрибуты защиты потока  
    SIZE_T cbStack, // размер начального стека  
    PTHREAD_START_ROUTINE pfnStartAddr, // адрес функции  
        // потока  
    PVOID pvParam, // аргументы для вызова функции  
    ПОТОКА  
    DWORD dwCreate, // параметры создания потока  
    PDWORD pdwThreadId // идентификатор потока  
);
```




Параметры *CreateThread*

- ▣ **Параметр *psa*** позволяет задать атрибуты защиты для создаваемого потока. Если передать NULL, то система будет использовать дескриптор защиты по умолчанию. Чтобы дочерние процессы смогли наследовать дескриптор этого потока, определите структуру SECURITY_ATTRIBUTES и инициализируйте ее элемент *hInheritHandle* значением TRUE.
- ▣ **Параметр *cbStack*** определяет, какую часть виртуального адресного пространства процесса поток сможет использовать под свой стек.
- ▣ **Параметр *pfnStartAddr*** определяет адрес функции потока, с которой должен будет начать работу создаваемый поток, а **параметр *pvParam*** идентичен параметру *pvParam* функции потока.
- ▣ **Параметр *fdwCreate*** принимает одно из двух значений: 0 (исполнение потока начинается немедленно) или CREATE_SUSPENDED (поток приостанавливается).
- ▣ **Параметр *pdwThreadId*** – это адрес для возврата функцией *CreateThread ()* идентификатора нового потока.

Функция *CreateRemoteThread*

- Функция *CreateRemoteThread* () создает поток, который запускается в виртуальном адресном пространстве другого процесса с дескриптором *hProcess*.

```
HANDLE CreateRemoteThread (  
    HANDLE hProcess, // дескриптор процесса  
    LPSECURITY_ATTRIBUTES lpThreadAttributes, // дескриптор защиты  
    SIZE_T dwStackSize, // размер начального стека  
    LPTHREAD_START_ROUTINE lpStartAddress, // адрес функции потока  
    LPVOID lpParameter, // аргументы для вызова функции потока  
    DWORD dwCreationFlags, // параметры создания потока  
    LPDWORD lpThreadId // идентификатор потока  
);
```



Установка приоритета потока

```
BOOL SetThreadPriority(
```

```
    HANDLE hThread, // дескриптор потока
```

```
    int nPriority // уровень приоритета потока
```

```
);
```

- Функция *SetThreadPriority* () дает возможность установки базового уровня приоритета потока относительно класса приоритета его процесса.
- Например, устанавливая **THREAD_PRIORITY_HIGHEST** при вызове *SetThreadPriority* () для потока процесса **IDLE_PRIORITY_CLASS** базовый уровень приоритета потока устанавливается в значение 6.



Относительные приоритеты потоков

- `THREAD_PRIORITY_ABOVE_NORMAL`
- `THREAD_PRIORITY_BELOW_NORMAL`
- `THREAD_PRIORITY_HIGHEST`
- `THREAD_PRIORITY_IDLE`
- `THREAD_PRIORITY_LOWEST`
- `THREAD_PRIORITY_NORMAL`
- `THREAD_PRIORITY_TIME_CRITICAL`



Завершение потока

- Поток можно завершить четырьмя способами:
 - функция потока возвращает управление путем вызова оператора *return* (рекомендуемый способ);
 - поток самоуничтожается вызовом функции *ExitThread ()* (нежелательный способ);
 - один из потоков данного или стороннего процесса вызывает функцию *TerminateThread ()* (нежелательный способ);
 - завершается процесс, содержащий данный поток (тоже нежелательно).
 - Принудительное завершение потока с помощью функций *ExitThread ()* и *TerminateThread ()* нежелательно, т.к. процесс продолжает работать, но при этом весьма вероятна утечка памяти или других ресурсов, которые были запрошены закрытым потоком.
-
- ▶ ПОТОКОМ.

Возврат управления функцией потока

При этом:

- любые C++-объекты, созданные данным потоком, уничтожаются соответствующими деструкторами;
- система корректно освобождает память, которую занимал стек потока;
- система устанавливает код завершения данного потока (поддерживаемый объектом ядра "поток") – его и возвращает Ваша функция потока;
- счетчик пользователей данного объекта ядра "поток" уменьшается на 1.



Функция *ExitThread*

- Для принудительного завершения потока в рамках процесса-владельца необходимо вызвать функцию:

`VOID ExitThread (DWORD dwExitCode);`

- Через параметр *dwExitCode* Вы передаете код завершения потока.



Функция *TerminateThread*

- Вызов этой функции позволяет завершить практически любой поток:

BOOL TerminateThread
(HANDLE hThread, DWORD dwExitCode);

- В параметр *dwExitCode* помещается код завершения потока.
- После вызова этой функции поток с дескриптором *hThread* завершается, а счетчик пользователей, соответствующего ему экземпляра объекта ядра, уменьшится на 1.



Действия при завершении потока

- Освобождаются все описатели User-объектов, принадлежавших потоку.
- Код завершения потока меняется со `STILL_ACTIVE` на код, переданный в функцию *ExitThread* или *TerminateThread*.
- Объект ядра «поток» переводится в свободное состояние.
- Если данный поток является последним активным потоком в процессе, завершается и сам процесс.
- Счетчик пользователей объекта «поток» уменьшается на 1.

`BOOL GetExitCodeThread`

▶ `(HANDLE hThread, PDWORD pdwExitCode);`

Действия при принудительном завершении процесса

- Функции *ExitProcess ()* и *TerminateProcess ()* принудительно завершают потоки, принадлежащие завершаемому процессу.
- При этом гарантируется высвобождение любых выделенных процессу ресурсов, в том числе стеков потоков.
- Однако эти две функции уничтожают потоки принудительно так, будто для каждого из них вызывается функция *TerminateThread ()*. А это означает, что очистка проводится некорректно, деструкторы C++-объектов не вызываются.



Управление динамическими приоритетами потока

```
BOOL SetThreadPriorityBoost(  
    HANDLE hThread,        // дескриптор потока  
    BOOL DisablePriorityBoost // состояние  
    //форсирования приоритета  
);  
  
BOOL GetThreadPriorityBoost(  
    HANDLE hThread,        // дескриптор потока  
    PBOOL pDisablePriorityBoost // состояние форсажа  
    //приоритета  
);
```

- Для успешного выполнения функций поток должен иметь привилегию **THREAD_SET_INFORMATION**.



Управление потоками – функция *ResumeThread*

- Если поток создан с флагом `CREATE_SUSPENDED`, то после своего создания он остается в состоянии ожидания.
- Вы можете настроить некоторые его свойства и подготовить данные для обработки. Закончив настройку, Вы должны разрешить выполнение потока (в рамках его процесса).
- Для этого вызовите *ResumeThread* () и передайте в качестве параметра дескриптор потока, возвращенный функцией *CreateThread* () .

DWORD ResumeThread (HANDLE hThread);



Управление потоками – функция *SuspendThread*

- Выполнение потока можно приостановить не только при его создании с флагом `CREATE_SUSPENDED`, но и вызовом *SuspendThread* ().

`DWORD SuspendThread (HANDLE hThread);`

- Выполнение отдельного потока можно приостанавливать несколько раз. Количество приостановок сохраняется в атрибутах объекта типа «поток» с дескриптором `hThread`.
- Если поток был приостановлен N раз, то и возобновлен он должен быть тоже N раз – только в это случае поток сможет покинуть состояние ожидания.



Засыпание и переключение потоков с помощью *Sleep*

- Вызывая функцию *Sleep ()*, поток добровольно отказывается от остатка выделенного ему кванта времени:

VOID Sleep (DWORD dwMilliseconds);

- Если вызвать *Sleep ()* и передать в *dwMilliseconds* значение большее нуля, то поток перейдет в состояние ожидания на период, примерно равный *dwMilliseconds* миллисекунд.
- Если вызвать *Sleep ()* и передать в *dwMilliseconds* значение INFINITE, то поток перейдет в состоянии ожидания «навечно».
- Если вызвать *Sleep ()* и передать в *dwMilliseconds* нулевое значение, то в этом случае поток откажется от остатка своего кванта времени и предложит операционной системе поставить на выполнение другой готовый поток с тем же приоритетом, что и у него самого. Если такого потока нет, то поток продолжит работу.



Вопрос

- Через какой промежуток времени поток вернется в состояние *Running* после вызова функции *Sleep (100)* ?
- В какое другое состояние можно перевести поток, вызвавший функцию *Sleep (INFINITE)* ?



Засыпание и переключение потоков

- Функция позволяет подключить к процессору другой поток (если он есть):

BOOL SwitchToThread ();

- Вызов *SwitchToThread ()* аналогичен вызову *Sleep ()* с передачей в *dwMilliseconds* нулевого значения. Разница лишь в том, что *SwitchToThread ()* дает возможность выполнять потоки с более низким приоритетом, которым не хватает процессорного времени, а *Sleep ()* действует без оглядки на «голодающие» потоки.



Определение периодов выполнения потока

```
BOOL GetThreadTimes(  
    HANDLE hThread,  
    PFILETIME pftCreationTime,  
    PFILETIME pftExitTime,  
    PFILETIME pftKernelTime,  
    PFILETIME pftUserTime  
);
```

- С помощью этой функции можно определить время создания и время завершения потока (отсчитываются с полуночи 1 января 1601 года по Гринвичу), а также время работы потока в режиме ядра и в пользовательском режиме.
 - Время измеряется интервалами по 100 наносекунд, поэтому функция *GetThreadTimes ()* не подходит для высокоточного измерения.
-

