Операционные системы

Управление центральным процессором и объединение ресурсов

Основы управления процессами

API Win32 для создания и завершения процессов

Создание процесса

- Вызов функции *CreateProcess.*
- Система создает объект ядра "процесс" с начальным значением счетчика числа его пользователей, равным 1. Этот объект – не сам процесс, а компактная структура данных, через которую ОС управляет процессом.
- 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. Как и в первом случае, объект ядра "поток" – это компактная структура данных, через которую система управляет потоком.
- 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, *CreateProcess* вернет TRUE.

- 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.

Создание процесса

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

Параметры CreateProcess

- Параметры ***pszApplicationName u pszCommandLine.*** Эти параметры определяют имя исполняемого файла, которым будет пользоваться новый процесс, и командную строку, передаваемую этому процессу.
- Параметры ***psaProcess, psaThread u bInheritHandles .*** Параметры *psaProcess* и *psaThread* позволяют определить нужные атрибуты защиты для объектов "процесс" и "поток" соответственно. В эти параметры можно занести NULL, и система закрепит за данными объектами дескрипторы защиты по умолчанию.
- Параметр ***fdwCreate*** определяет 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. Обычно вместо этого параметра передается NULL, в результате чего 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 PROCESS_INFORMATION, которую Вы должны предварительно создать; ее элементы инициализируются самой функцией *CreateProcess.*

Параметр *fdwCreate*

- Параметр *fdwCreate* определяет флаги, влияющие на то, как именно создается новый процесс Флаги комбинируются булевым оператором OR.
  - Флаг DEBUG_PROCESS даст возможность родительскому процессу проводить отладку дочернего, а также всех процессов, которые последним могут быть порождены.
  - Флаг DEBUG_ONLY_THIS_PROCESS аналогичен флагу DEBUG_PROCESS с тем исключением, что заставляет систему уведомлять родительский процесс о возникновении специфических событий только в одном дочернем процессе — его прямом потомке.
  - 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, изменять приоритет его первичного потока или включать этот процесс в задание (job) до того, как он получит шанс на выполнение. Внеся нужные изменения в дочерний процесс, родительский разрешает выполнение его кода вызовом функции *ResumeThread*.
  - Флаг DETACHED_PROCESS блокирует доступ процессу, инициированному консольной программой, к созданному родительским процессом консольному окну и сообщает системе, что вывод следует перенаправить в новое консольное окно.
  - Флаг CREATE_NEW_CONSOLE приводит к созданию нового консольного окна для нового процесса. Имейте в виду, что одновременная установка флагов CREATE_NEW_CONSOLE и DETACHED_PROCESS недопустима.
  - Флаг CREATE_NO_WINDOW не дает создавать никаких консольных окон для данного приложения и тем самым позволяет исполнять его без пользовательского интерфейса.
  - Флаг CREATE_BREAKAWAY_FROM_JOB позволяет процессу, включенному в задание, создать новый процесс, отделенный от этого задания

Параметр *fdwCreate*

- Параметр *fdwCreate* разрешает задать и класс приоритета процесса Однако это необязательно и даже, как правило, не рекомендуется, система присваивает новому процессу класс приоритета по умолчанию. Возможные классы приоритета перечислены в следующей таблице.
  - Idle (простаивающий) IDLE_PRIORITY_CLASS
  - Below normal (ниже обычного) BELOW_NORMAL_PRIORITY_CLASS
  - Normal (обычный) NORMAL_PRIORITY_CLASS
  - Above normal (выше обычного) ABOVE_NORMAL_PRIORITY_CLASS
  - High (высокий) HIGH_PRIORITY_CLASS
  - Realtime (реального времени) REALTIME_PRIORITY_CLASS
  - Классы приоритета влияют на распределение процессорного времени между процессами и их потоками.
  - NOTE
    Классы приоритета BELOW_NORMAL_PRIORITY_CLASS и ABOVE_NORMAL_ PRIORITY_CLASS введены лишь в Windows 2000; они не поддерживаются в Windows NT 4.0, Windows 95 или Windows 98.

Завершение процесса

- Существует 4 гипотетических варианта завершения процесса.
  - 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 умирают (большая редкость).
- Явный вызов *ExitProcess* и *TerminateProcess* – распространенная ошибка, которая мешает правильной очистке ресурсов.

Возврат управления входной функцией первичного потока

При этом:

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

Функция ExitProcess

Процесс завершается, когда один из его потоков вызывает *ExitProcess:*

VOID ExitProcess(UINT fuExitCode);

Эта функция завершает процесс и заносит в параметр *fuExitCode* код завершения процесса.

Функция TerminateProcess

Вызов функции *TerminateProcess* тоже завершает процесс:

BOOL TerminateProcess (HANDLE hProcess, UINT fuExitCode);

Параметр *bProcess* идентифицирует описатель завершаемого процесса, а в параметре *fuExitCode* возвращается код завершения процесса.

TerminateProcess – функция асинхронная, т. е. она сообщает системе, что Вы хотите завершить процесс, но к тому времени, когда она вернет управление, процесс может быть еще не уничтожен. Так что, если Вам нужно точно знать момент завершения процесса, используйте WaitForSingleObject или аналогичную функцию, передав ей описатель этого 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, или незанятое (signaled), 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 выполнения процесс должен иметь право доступа
PROCESS_SET_INFORMATION

Дополнительная информация

- 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

Основы управления процессами

API Win32 для управления потоками

Создание потока

```
HANDLE CreateThread (
    PSECURITY_ATTRIBUTES psa,
    SIZE_T cbStack,
    PTHREAD_START_ROUTINE pfnStartAddr,
    PVOID pvParam,
    DWORD dwCreate,
    PDWORD pdwThreadID
);
```

Параметры создания потока

- ***Параметр psa*** является указателем на структуру SECURITY_ATTRIBUTES Если Вы хотите, чтобы объекту ядра "поток" были присвоены атрибуты защиты по умолчанию (что чаще всего и бывает), передайте в этом параметре NULL A чтобы дочерние процессы смогли наследовать описатель этого объекта, определите структуру SECURI TY_ATTRIBUTES и инициализируйте ее элемент *hInheritHandle* значением TRUE.
- ***Параметр cbStack*** определяет, какую часть адресного пространства поток сможет использовать под свой стек. Каждому потоку выделяется отдельный стек.
- **Параметр *pfnStartAddr*** определяет адрес функции потока, с которой должен будет начать работу создаваемый поток, а **параметр *pvParam*** идентичен параметру *pvParam* функции потока. *CreateTbread* лишь передает этот параметр по эстафете той функции, с которой начинается выполнение создаваемого потока. Таким образом, данный параметр позволяет передавать функции потока какое-либо инициализирующее значение. Оно может быть или просто числовым значением, или указателем на структуру данных с дополнительной информацией. Вполне допустимо и даже полезно создавать несколько потоков, у которых в качестве входной точки используется адрес одной и той же функции. Например, можно реализовать Web-сервер, который обрабатывает каждый клиентский запрос в от дельном потоке. При создании каждому потоку передается свое значение *pvParam.*
- ***Параметр fdwCreate*** определяет дополнительные флаги, управляющие созданием потока. Он принимает одно из двух значений. 0 (исполнение потока начинается немедленно) или CREATE_SUSPENDED. В последнем случае система создает поток, инициализирует его и приостанавливает до последующих указаний. Флаг CREATE_SUSPENDED позволяет программе изменить какие-либо свойства потока перед тем, как он начнет выполнять код.
- ***Параметр pdwThreadID***— это адрес переменной типа DWORD, в которой функция возвращает идентификатор, приписанный системой новому потоку. В Windows 2000 и Windows NT 4 в этом параметре можно передавать NULL (обычно так и делается). Тем самым Вы сообщаете функции, что Вас не интересует идентификатор потока в Windows 95/98 это приведет к ошибке, так как функция попытается записать идентификатор потока по нулевому адресу, что недопустимо. И поток не будет создан.

Функция CreateRemoteThread

Функция **CreateRemoteThread** создает поток, который запускается в виртуальном адресном пространстве другого процесса.

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

Установка приоритета

Поток создается с приоритетом потока **THREAD_PRIORITY_NORMAL**. Используйте функции GetThreadPriority и SetThreadPriority, чтобы получать и установить приоритетное значение 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.

Приоритеты потоков

Приоритет	Назначение
THREAD_PRIORITY_ABOVE_NORMAL	Приоритет на 1 пункт выше класса приоритета.
THREAD_PRIORITY_BELOW_NORMAL	Приоритет на 1 пункт ниже класса приоритета.
THREAD_PRIORITY_HIGHEST	Приоритет на 2 пункта выше класса приоритета.
THREAD_PRIORITY_IDLE	Базовый приоритет 1 для процессов **IDLE_PRIORITY_CLASS**, **BELOW_NORMAL_PRIORITY_CLASS**, **NORMAL_PRIORITY_CLASS**, **ABOVE_NORMAL_PRIORITY_CLASS** или **HIGH_PRIORITY_CLASS** и уровень базового приоритета 16 для процессов **REALTIME_PRIORITY_CLASS**.
THREAD_PRIORITY_LOWEST	Приоритет на 2 пункта ниже класса приоритета.
THREAD_PRIORITY_NORMAL	Нормальный приоритет класса приоритета.
THREAD_PRIORITY_TIME_CRITICAL	Базовый приоритет 15 для процессов **IDLE_PRIORITY_CLASS**, **BELOW_NORMAL_PRIORITY_CLASS**, **NORMAL_PRIORITY_CLASS**, **ABOVE_NORMAL_PRIORITY_CLASS** или **HIGH_PRIORITY_CLASS** и уровень базового приоритета 31 для процессов **REALTIME_PRIORITY_CLASS**.

Завершение потока

- Поток можно завершить четырьмя способами:
  - функция потока возвращает управление (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).
- Явный вызов *ExitThread* и *TerminateThread* нежелателен, т.к. процесс продолжает работать, но при этом весьма вероятна утечка памяти или других ресурсов.

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

Поток можно завершить принудительно, вызвав:

VOID ExitThread(DWORD dwExitCode);

В параметр *dwExitCode* Вы помещаете значение, которое система рассматривает как код завершения потока.

Функция TerminateThread

Вызов этой функции также завершает поток:

```
BOOL TerminateThread( HANDLE hThread,
                      DWORD dwExitCode);
```

В параметр *dwExitCode* помещается код завершения потока. После того как поток будет уничтожен, счетчик пользователей его объекта ядра "поток” уменьшится.

Если завершается процесс

- Функции *ExitProcess* и *TerminateProcess* принудительно завершают потоки, принадлежащие завершаемому процессу.
- Эти функции прекращают выполнение всех потоков, принадлежавших завершенному 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, данные на диск не сбрасываются и т. д.

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 ядра "поток" уменьшается на 1.

BOOL GetExitCodeThread(HANDLE hThread, PDWORD pdwExitCode);

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
);
```

Поток должен иметь право доступа **THREAD_SET_INFORMATION**.

Управление потоками

Флаг CREATE_SUSPENDED

Если поток создан с флагом CREATE_SUSPENDED, то после своего создания он остается в приостановленном состоянии. Вы можете настроить некоторые его свойства (например, приоритет, о котором мы поговорим позже). Закончив настройку, Вы должны разрешить выполнение потока. Для этого вызовите *ResumeThread* и пере дайте описатель потока, возвращенный функцией *CreateThread*.

DWORD ResumeThread(HANDLE hThread);

Выполнение потока можно приостановить не только при его создании с флагом CREATE_SUSPENDED, но и вызовом *SuspendThread.* Выполнение отдельного потока можно приостанавливать несколько раз. Если поток приостановлен 3 раза, то и возобновлен он должен быть тоже 3 раза — лишь тогда система выделит ему процессорное время.

DWORD SuspendThread(HANDLE hThread);

Засыпание и переключение потоков

VOID Sleep (DWORD dwMilliseconds);

- Эта функция приостанавливает поток па *dwMilliseconds* миллисекунд. Отметим несколько важных моментов, связанных с функцией *Sleep.*
  - Вызывая *Sleep,* поток добровольно отказывается от остатка выделенного ему кванта времени
  - Система прекращает выделять потоку процессорное время на период, *пример но* равный заданному, Все верно: если Вы укажете остановить поток на 100 мс, приблизительно на столько он и "заснет", хотя не исключено, что его сон про длится на несколько секунд или даже минут больше. Вспомните, Windows не является системой реального времени. Ваш поток может возобновиться в заданный момент, но это зависит от того, какая ситуация сложится в системе к тому времени.
  - Вы можете вызвать *Sleep* и передать в *dwMilliseconds* значение INFINITE, вообще запретив планировать поток. Но это не очень практично — куда лучше корректно завершить поток, освободив его стек и объект ядра.
  - Вы можете вызвать *Sleep* и передать в *dwMilliseconds* нулевое значение. Тогда Вы откажетесь от остатка своего кванта времени и заставите систему подключить к процессору другой поток. Однако система может снова запустить Ваш поток, если других планируемых потоков с тем же приоритетом нет.

Засыпание и переключение потоков

BOOL SwitchToThread();

- Функция *SwitchToThread* позволяет подключить к процессору другой поток (если он есть).
- Вызов *SwitchToThread* аналогичен вызову *Sleep c* 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, необходимое для выполнения сложного алгоритма.
- *GetThreadTimes* не годится для высокоточного измерения временных интервалов.