Process 2007.

**Управление центральным процессором и объединение ресурсов.**

*Управление центральным процессором…*

*Процессы и потоки.*

Основные понятия:

* Задание – набор процессов, управляемых как единое целое, с общими квотами и лимитами
* **Процесс** – контейнер для ресурсов
* **Поток** – сущность планируемая ядром
* Волокно – облегченный поток, управляемый полностью в пространстве пользователя

*Процессы.*

Процесс – это совокупность системных ресурсов, задействованная для выполнения определенной работы.

Понятие "процесс" включает следующее:

* исполняемый код;
* собственное адресное пространство, которое представляет собой совокупность виртуальных адресов, которые может использовать процесс;
* ресурсы системы, такие как файлы, семафоры и т.п., которые назначены процессу операционной системой;
* хотя бы одну выполняемую нить.

*Потоки (thread)*

Поток (нить) – это непрерывная последовательность инструкций, выполняющих определенную функцию.

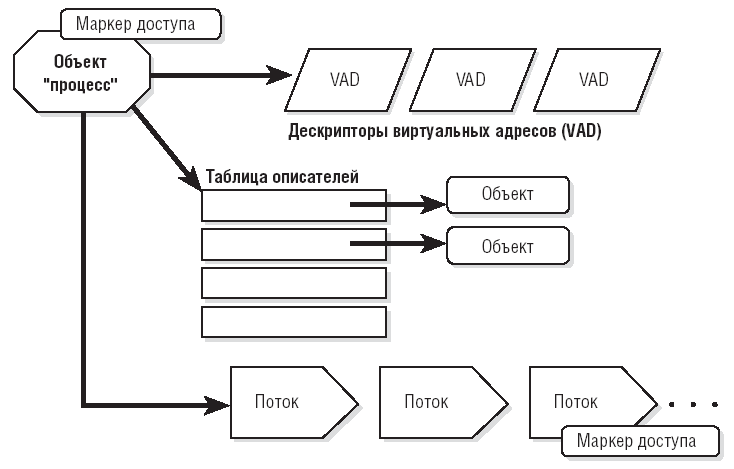
Для выполнения потока необходимы две вещи: системное время и адресное пространство.  Поток не имеет собственного адресного пространства и получает доступ к адресному пространству процесса-родителя.

*Процессы и нити в Windows NT-2000*

* Процессы Windows NT-2000 имеют многопоточную организацию.
* Каждый поток внутри одного процесса делит его ВАП, а также остальные ресурсы, принадлежащие процессу. Это означает, что все потоки в процессе могут записывать и считывать содержимое памяти любого из потоков данного процесса.
* Процессы и потоки в Windows NT-2000 реализованы в форме объектов, и доступ к ним осуществляется посредством службы объектов.
* Объекты-процессы и объекты-потоки имеют встроенные средства синхронизации.

*Атрибуты процесса в Windows NT-2000*

* Идентификатор процесса – уникальное значение, которое идентифицирует процесс в рамках операционной  системы.
* Закрытое *виртуальное адресное пространство –* диапазон адресов виртуальной памяти, которым может пользоваться процесс.
* Исполняемую программу – начальный код и данные, проецируемые на виртуальное адресное пространство процесса.
* Список открытых дескрипторов различных системных ресурсов – семафоров, файлов и других объектов, доступных всем потокам в данном процессе.
* Марке доступа – исполняемый объект, содержащий информацию о безопасности и идентифицирующий пользователя, группы безопасности и привилегии, сопоставленные с процессом.
* Базовый приоритет – основа для исполнительного приоритета нитей процесса.
* Процессорная совместимость – набор процессоров, на которых могут выполняться нити процесса.
* Предельные значения квот ресурсов (см. Job).
* Время исполнения – общее количество времени, в течение которого выполняются все нити процесса.
* Список потоков процесса (как минимум один поток).

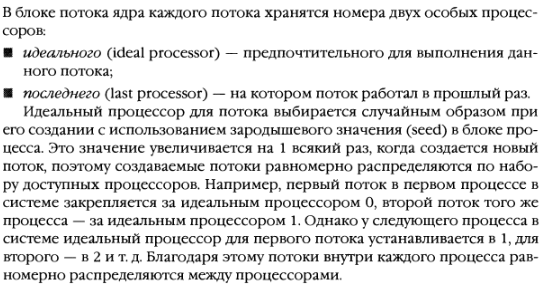
*Процесс и его ресурсы*

* Каждый процесс обладает контекстом защиты, который хранится в объекте — *маркере доступа*. Маркер доступа содержит идентификацию защиты и определяет полномочия данного процесса. По умолчанию у потока нет собственного маркера доступа, но он может получить его, и это позволит ему подменять контекст защиты другого процесса.
* Дескрипторы виртуальных адресов (virtual address descriptors, VAD) – это структуры данных, используемые диспетчером памяти для учета виртуальных адресов, задействованных процессом.

*Атрибуты потоков в Windows NT-2000*

* Идентификатор клиента – уникальное значение, которое идентифицирует поток при ее обращении к серверу.
* Контекст потока – информация, которая необходима ОС для того, чтобы продолжить выполнение прерванного потока. Контекст потока содержит текущее состояние регистров, стеков и индивидуальной области памяти.
* Два стека, один из которых используется потоком при выполнении в режиме ядра, а другой – в пользовательском режиме;
* Локальная память потока (thread local storage, TLS) – закрытая область памяти, используемую подсистемами, библиотеками исполняющих систем (runtime libraries) и DLL;
* Динамический приоритет – значение приоритета потока в данный момент.
* Базовый приоритет – нижний предел динамического приоритета потока.
* Процессорная совместимость потока – перечень типов процессоров, на которых может выполняться поток.
* Номер идеального процессора потока.
* Номер процессора, на котором поток выполнялся прошлый раз.
* Время выполнения потока – суммарное время выполнения потока в пользовательском режиме и в режиме ядра.
* Счетчик приостановок – текущее количество приостановок выполнения нити.

*Идеальный и последний процессор*

**

Функция SetThreadIdealProcessor

 *Задание (job) в Windows 2000*

* В Windows 2000 в модель процессов введено новое расширение – *задания* (jobs).
* Задания предназначены в основном для того, чтобы группами процессов можно было оперировать и управлять как единым целым.
* Объект-задание позволяет устанавливать определенные атрибуты и накладывать ограничения на процесс или процессы, сопоставленные с заданием.

*Квоты и лимиты для заданий (job)*

* квоты (базовые и расширенные базовые ограничения):
  + максимальное количество процессов;
  + суммарное время центрального процессора (для каждого процесса и для задания в целом);
  + максимальное количество используемой памяти (для каждого процесса и для задания в целом);
* базовые ограничения по пользовательскому интерфейсу;
* ограничения, связанные с защитой.

Волокна (fibers) в Windows 2000

* Введены в Windows 2000 для переноса существующих серверных приложений из UNIX.
* Реализованы на уровне кода пользовательского режима.
* Процессорное время между волокнами распределяется по пользовательскому алгоритму.

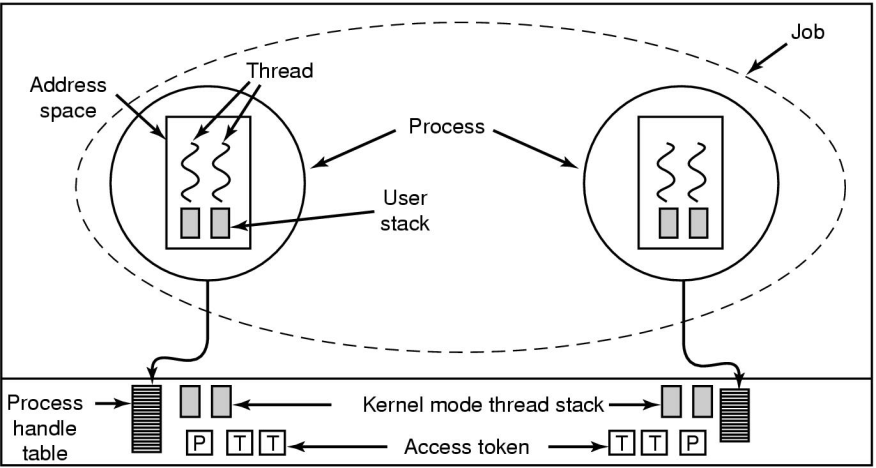
Microsoft добавила в Windows поддержку волокон (fibers), чтобы упростить портирование (перенос) существующих серверных приложений из UNIX в Windows C точки зрения терминологии, принятой в Windows, такие серверные приложения следует считать однопоточными, но способными обслуживать множество клиентов. Иначе говоря, разработчики UNIX-приложений создали свою библиотеку для организации многопоточности и с ее помощью эмулируют истинные потоки. Она создает набор стеков, сохраняет определенные регистры процессора и переключает контексты при обслуживании клиентских запросов.

Разумеется, чтобы добиться большей производительности от таких UNIX-приложений, их следует перепроектировать, заменив библиотеку, эмулирующую потоки, на настоящие потоки, используемые в Windows. Но переработка может занять несколько месяцев, и поэтому компании сначала просто переносят существующий UNIX-код в Windows — это позволяет быстро предложить новый продукт на рынке Windows приложений.

Волокны реализованы на уровне кода пользовательского режима, ядро ничего не знает о них, и процессорное время распределяется между волокнами по алгоритму, определяемому Вами. А раз так, то о вытеснении волокон говорить не приходится – по крайней мере, когда дело касается ядра.

В потоке может быть одно или несколько волокон. Для ядра поток – все то, что можно вытеснить и что выполняет код. Единовременно поток будет выполнять код лишь одного волокна.

*Взаимосвязь между заданиями, процессами и потоками*

**

Управление центральным процессором…

Планирование загрузки процессорного времени.

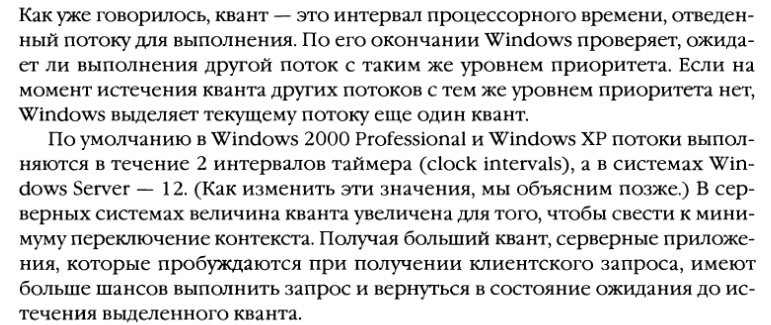
*Системный планировщик в Windows NT-2000-2003*

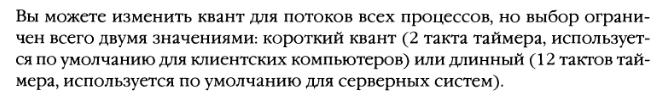
* Вытесняющая многопоточность.
* Квантование времени.
* Приоритетный режим обслуживания:
  + абсолютные приоритеты;
  + динамические приоритеты.

*Планирование загрузки процессорного времени*

* В Windows 2000-2003 реализована вытесняющая многозадачность, при которой операционная система не ждет, когда нить сама захочет освободить процессор, а принудительно снимает ее с выполнения после того, как та израсходовала отведенное ей время (квант), или если в очереди готовых появилась нить с более высоким приоритетом.
* При такой организации разделения процессора ни одна нить не займет процессор на очень долгое время.

Квантование времени





*Приоритетный режим обслуживания*

* Разработчик ПО может использовать приоритеты от 1 до 31.
* Нулевой приоритет зарезервирован для потока обнуления страниц.
* Поток наследует приоритет процесса, породившего его.
* ОС Windows NT 4.0 предоставляет 4 класса приоритетов: Realtime, High, Normal и Idle.
* ОС Windows 2000: еще 2 дополнительных класса приоритетов – Below Normal и Above Normal.
* Относительный приоритет потока: idle, lowest, below normal, normal (обычный), above normal, highest и time-critical.

Системный планировщик Windows 2000 является многозадачным планировщиком с приоритетным режимом обслуживания. Это означает, что он делит процессорное время между активными потоками системы. Планировщик предоставляет каждому потоку некоторый квант времени работы процессора. Если поток по каким-то причинам преждевременно перестает быть активным (например, когда останавливается в ожидании завершения операции ввода/вывода), ядро Windows 2000 вызывает планировщик для поиска следующего потока, которому можно передать управление процессором. Если поток выбирает весь отведенный ему квант времени, Windows 2000 вызывает планировщик с тем, чтобы другие потоки также могли выполнять свою работу. Планировщик Windows 2000 (который в Microsoft называют диспетчером) с помощью системы приоритетов устанавливает, какому именно из активных потоков в каждый момент времени следует передать управление одним из процессоров системы. Планировщик всякий раз выбирает поток с наивысшим приоритетом; если сразу несколько потоков имеют одинаковый приоритет, планировщик предоставляет квант процессорного времени каждому из них по очереди. Переключение процессора с выполнения команд одного потока на выполнение команд другого называют переключением контекста.

Приоритет процесса, породившего поток, определяет и приоритет самого потока. Приоритеты имеют значения от 1 до 31. Разработчики приложений или системные администраторы могут манипулировать уровнем приоритета процесса, приписывая процесс к одному из нескольких возможных классов. Каждый класс содержит значения приоритетов определенного диапазона, и операционная система Windows 2000 интерпретирует установку приоритета потока – средствами программного интерфейса или же с помощью специальных административных утилит — как изменение приоритета процесса. Операционная система NT 4.0 предоставляет четыре класса приоритетов: Realtime, High, Normal и Idle. Класс Realtime содержит приоритеты высшего уровня. И разработчики, и администраторы пользуются им сравнительно редко, поскольку потоки с приоритетом из этого диапазона соперничают с потоками ядра ОС. Таким образом, большинству приложений доступны лишь три оставшихся класса. Часто разработчикам и администраторам бывает необходима большая дискретность в назначении уровня приоритета индивидуальных потоков или приложения в целом, в зависимости от задачи, выполняемой конкретным потоком или приложением. В ответ на подобные пожелания разработчики Windows 2000 ввели два новых класса приоритетов: Below Normal и Above Normal.

Кроме класса приоритета потока существует также относительный приоритет потока.

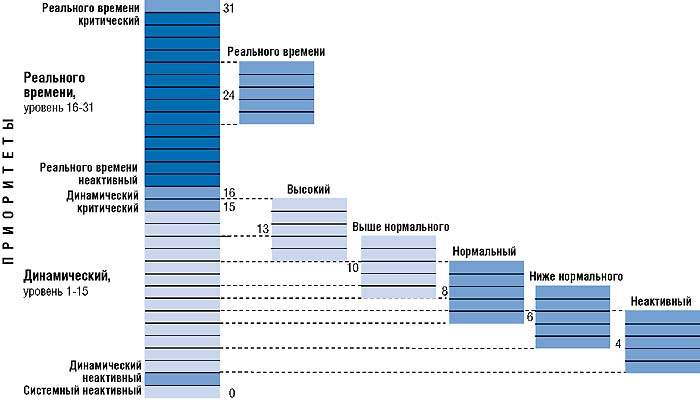
*Классы приоритета процессов*

|  |  |
| --- | --- |
| Класс приоритета | Описание |
| Real-lime (24) | Потоки в этом процессе обязаны немедленно реагировать на события, обеспечивая выполнение критических по времени задач. Такие потоки вытесняют даже компоненты ОС. Процессы этого класса нельзя запустить, если пользователь не имеет привилегии Increase Scheduling Priority (администратор и пользователь с расширенными полномочиями). |
| High (13) | Потоки в этом процессе тоже должны немедленно реагировать на события, обеспечивая выполнение критических по времени задач. Этот класс присвоен, например, Task Manager. |
| *Above normal* (10) | *Класс приоритета, промежуточный между normal и high.* |
| Normal (8) | Потоки в этом процессе не предъявляют особых требований к выделению им процессорного времени (*99%* приложений). |
| *Below normal* (6) | *Класс приоритета, промежуточный между normal и idle.* |
| Idle (4) | Потоки в этом процессе выполняются, когда система не занята другой работой. Этот класс приоритета обычно используется для утилит, работающих в фоновом режиме, экранных заставок и пр. |

*Относительные приоритеты потоков*

|  |  |
| --- | --- |
| 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 обычною 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 |

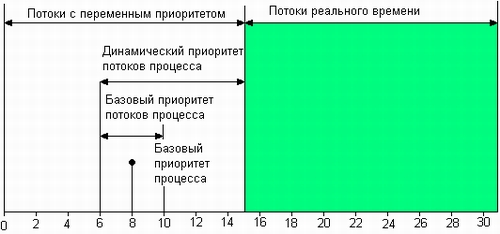
*Иллюстрация по приоритетам для Windows 2000*



Базовый и динамический приоритеты потока

* Уровень приоритета, получаемый комбинацией относительного приоритета потока и класса приоритета процесса, которому принадлежит данный поток, называют базовым уровнем приоритета потока. Иногда система изменяет уровень приоритета потока.
* Использование динамических приоритетов, изменяющихся во времени, позволяет реализовать адаптивное планирование, при котором не дискриминируются интерактивные задачи, часто выполняющие операции ввода-вывода и недоиспользующие выделенные им кванты.
* Система повышает приоритет только тех потоков, базовый уровень которых находится в пределах 1-15. Именно поэтому данный диапазон называется "областью динамического приоритета" (dynamic priority range). Система не допускает динамического повышения приоритета потока до уровней реального времени (более 15). Поскольку потоки с такими уровнями обслуживают системные функции, это ограничение не дает приложению нарушить работу операционной системы. И, кстати, система никогда не меняет приоритет потоков с уровнями реального времени (от 16 до 31).

*Динамические приоритеты*

В ходе выполнения нити ее приоритет (1-15) может меняться – механизм адаптивного планирования.

Принципы адаптивного планирования

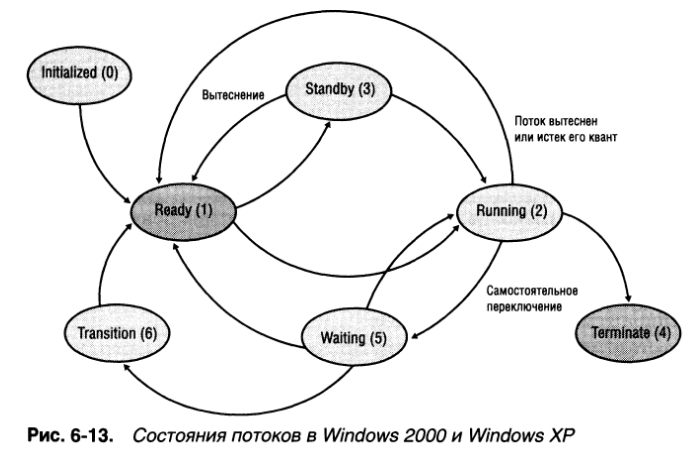
* Если поток полностью исчерпал свой квант, то его приоритет понижается на некоторую величину.
* Приоритет потоков, которые перешли в состояние ожидания, не использовав полностью выделенный им квант, повышается.
* Приоритет не изменяется, если поток вытеснен более приоритетным потоком.
* Повышение приоритета для “голодающих” потоков.

Голодающие” потоки

**Пример:** Представьте, что поток с приоритетом 4 готов к выполнению, но не может получить доступ к процессору из-за того, что его постоянно занимают потоки с приоритетом 8.

Это типичный случай "голодания" потока с более низким приоритетом. Обнаружив такой поток, не выполняемый на протяжении уже трех или четырех секунд, система поднимает его приоритет до 15 и выделяет ему двойную порцию времени. По его истечении потоку немедленно возвращается его базовый приоритет.

Граф состояний потоков для Windows 2000



Алгоритм планировщика

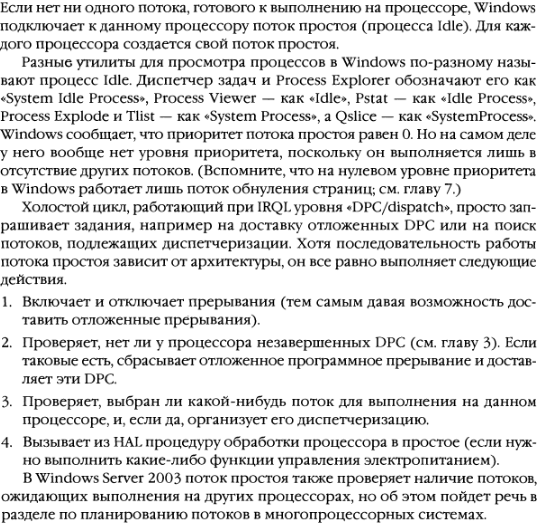
Поток освобождает процессор, если:

* исчерпан квант;
* в очереди готовых появляется более приоритетный поток;
* поток добровольно переходит в очередь готовых (Sleep (0), SwitchToThread);
* поток добровольно блокируется, уходя в состояние ожидания (Sleep (n), WaitForSingleObject, …);
* поток добровольно исключает текущий процессор из маски совместимости (привязки) (для МПС);
* поток завершается.

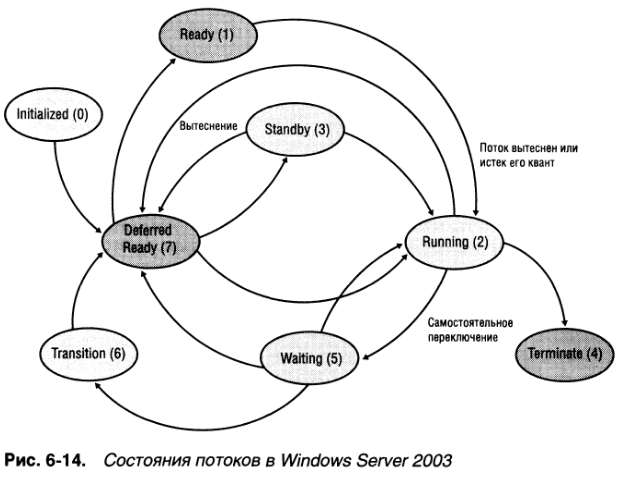
Состояния потоков

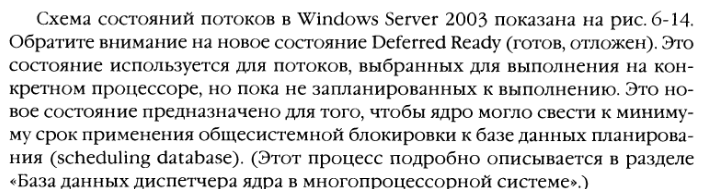
* **Готовность.** При поиске потока на выполнение диспетчер просматривает только потоки, находящиеся в состоянии готовности, у которых есть все для выполнения, но не хватает только процессора.
* **Первоочередная готовность (standby).** Для каждого процессора системы выбирается один поток, который будет выполняться следующим. Когда условия позволяют, происходит переключение на контекст этого потока.
* **Выполнение.** Как только происходит переключение контекстов, поток переходит в состояние выполнения и находится в нем до тех пор, пока не выполнится одно из вышеописанных условий освобождения процессора.
* **Ожидание.** Поток может входить в состояние ожидания несколькими способами: поток по своей инициативе ожидает некоторый объект для того, чтобы синхронизировать свое выполнение; операционная система (например, подсистема ввода-вывода) может ожидать в интересах потока; подсистема окружения может непосредственно заставить поток приостановить себя. Когда ожидание потока подойдет к концу, она возвращается в состояние готовности.
* **Переходное состояние.** Поток входит в переходное состояние, если он готов к выполнению, но ресурсы, которые ей нужны, заняты. Например, страница, содержащая стек потока, может быть выгружена из оперативной памяти на диск. При освобождении ресурсов поток переходит в состояние готовности.
* **Завершение.** Когда выполнение потока закончилось, он входит в состояние завершения.

Поток простоя



Граф состояний потоков для Windows 2003

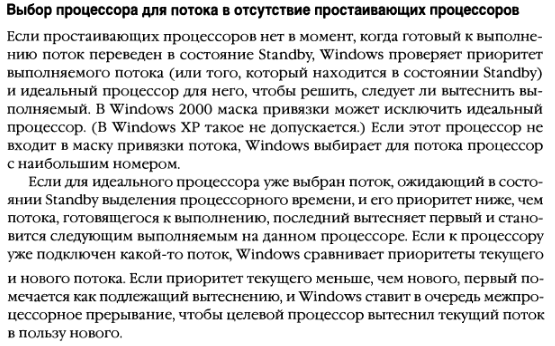


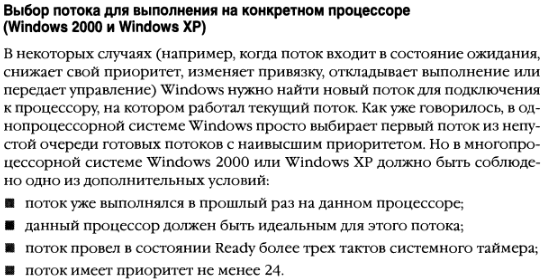


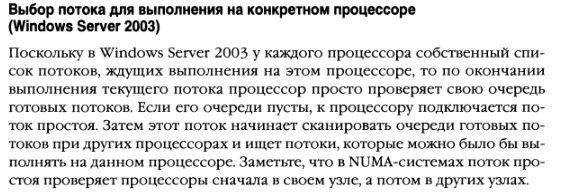
Управление потоками в МПС











**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 С/С++, 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.

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 и pszCommandLine.*** Эти параметры определяют имя исполняемого файла, которым будет пользоваться новый процесс, и командную строку, передаваемую этому процессу.
* Параметры ***psaProcess, psaThread и blnheritHandles .*** 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 ***psiStartlnfo*** указывает на структуру STARTUPINFO. Элементы структуры STARTUPINFO используются Windows-функциями при создании нового процесса.
* Параметр ***ppiProclnfo*** указывает на структуру PROCESS\_INFORMATION, которую Вы должны предварительно создать; ее элементы инициализируются самой функцией *CreateProcess.*

DEBUG\_PROCESS DEBUG\_ONLY\_THIS\_PROCESS

CREATE\_SUSPENDED

DETACHED\_PROCESS CREATE\_NEW\_CONSOLE CREATE\_NO\_WINDOW

CREATE\_BREAKAWAY\_FROM\_JOB

IDLE\_PRIORITY\_CLASS

BELOW\_NORMAL\_PRIORITY\_CLASS

NORMAL\_PRIORITY\_CLASS

ABOVE\_NORMAL\_PRIORITY\_CLASS

HIGH\_PRIORITY\_CLASS

REALTIME\_PRIORITY\_CLASS

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* 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 С++-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. e. 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**

**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 *hlnheritHandle* 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 *рvРаrаm* функции потока. *CreateTbread* лишьпередает 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 *рvParam.*
* ***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 pdwThreadlD****—* 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**](http://vsokovikov.narod.ru/Msdn_api/Global_SDK/Reference_API/Functions/Fn_G/fn_getthreadpriority.htm) и [**SetThreadPriority**](http://vsokovikov.narod.ru/Msdn_api/Global_SDK/Reference_API/Functions/Fn_S/fn_setthreadpriority.htm), 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 С++-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 dwExitCоde);*

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 С++-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.

BOOL SwitchToThread();

* Функция *SwitchToThread* позволяет подключить к процессору другой поток (если он есть).
* Вызов *SwitchToThread* аналогичен вызову *Sleep с* передачей в *dwMilliseconds* нулевого значения. Разница лишь в том, что *SwitchToThread* даетвозможность 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.

*GetThreadTimes* возвращает четыре временных параметра:

* Время coздания (creation time) – Абсолютная величина, выраженная в интервалах по 100 нс. Отсчитывается с полуночи 1 января 1601 года по Гринвичу до момента создания потока.
* Время завершении (exit time) – Абсолютная величина, выраженная в интервалах по 100 нс Отсчитывается с полуночи 1 января 1601 года по Гринвичу до момента завершения потока. Если поток все еще выполняется, этот показатель имеет неопределенное значение.
* Время выполнения ядра (kernel time) – Относительная величина, выраженная в интервалах по 100 нс. Сообщает время, затраченное этим потоком на выполнение кода операционной системы.
* Время выполнения User (User time) – Относительная величина, выраженная в интервалах по 100 нc. Сообщает время, затраченное потоком на выполнение кода приложения.

С помощью этой функции можно определить время, необходимое для выполнения сложного алгоритма.

*GetThreadTimes* не годится для высокоточного измерения временных интервалов.

**Управление центральным процессором…**

Взаимная блокировка и тупики

Исходная постановка :

* Есть несколько процессов
* Процессы нуждаются в доступе к нескольким ресурсам

Результат:

Возможна взаимная блокировка группы процессов, когда каждый процесс из группы ожидает события, которое может вызвать только другой процесс из той же группы.

Deadlock означает "мертвая блокировка“ или тупик.

Ресурс – это все, что может использоваться только одним процессом в любой момент времени (аппаратные или программные).

Причины возникновения тупиков

Коффман и другие исследователи доказали, что для возникновения ситуации взаимной блокировки должны выполнять все 4 условия:

* Взаимное исключение
  + Только один процесс может использовать ресурс в один момент времени
* “Удержание и ожидание”
  + Процессы, в данный момент удерживающие полученные ранее ресурсы, могут запрашивать новые ресурсы
* Отсутствие принудительной выгрузки ресурса
  + Ресурс не может быть отнят у процесса
* Цикличное ожидание
  + Существует цепочка процессов, в которой каждый процесс удерживает, по крайней мере, один ресурс необходимый для следующего процесса в цепочке

Пример 1. Блокировка по ресурсу

* Для размещения пользовательских данных доступно 200Kб, и двумя процессами выполняются следующие запросы памяти:



* В результате возникает тупик, т.к. ни один из процессов не может успешно выполнить второй запрос.

Блокировка возникает, когда процесс P1 удерживает ресурс, который необходим процессу P2, а процесс P2 удерживает ресурс, который необходим P1.

Пример 2. Круговое ожидание

ожидание "по кругу", круговое ожидание

Предотвращение тупиков

* Взаимное исключение
  + Не может быть предотвращено (на уровне ОС)
* “Удержание и ожидание”
  + Все процессы запрашивает ресурсы в один момент времени
  + Все запросы процесса на ресурсы должны удовлетворяться одновременно
  + Процесс может быть задержан в течение долгого времени в ожидании выполнения всех его запросов
  + Если процесс долгое время не использует ресурс, то ресурс может быть использован другими процессами
* Отсутствие принудительной выгрузки ресурса
  + если процессу отказывают в дальнейшем запросе, процесс должен освободить уже занятые ресурсы.
  + если процесс не может получить ресурс, процессу, вероятно, придется освободить его ресурсы. Должна быть возможность восстановления к текущему состоянию.
* Цикличное ожидание