Управление процессами, потоками и памятью в ОС Windows. Часть 1

Лекция

Ревизия: 0.2

Последняя модификация: Vlad Kovtun

Дата последней модификации: 03.04.2015 14:56:00

История изменений

12.09.2010 – Версия 0.1. Первичный документ. Ковтун В.Ю.

28.08.2014 – Версия 0.2. Замена рисунков. Ковтун В.Ю.

Содержание

Лекция 11. Управление процессами, потоками и памятью в ОС Windows

Вопросы

1. Процессы.
2. Потоки.
3. Память.

Процессы

Структуры данных

Каждый процесс в Windows **представлен блоком процесса** (EPROCESS), создаваемым исполнительной системой. Кроме многочисленных атрибутов, относящихся к процессу, в блоке EPROCESS содержатся указатели на некоторые структуры данных. Так, у каждого процесса есть один или более потоков, представляемых **блоками потоков** (ETHREAD) исполнительной системы. Блок EPROCESS и связанные с ним структуры данных, за исключением **блока переменных окружения процесса** (process environment block, PEB) — существуют в системном пространстве. PEB находится в адресном пространстве процесса, так как содержит данные, модифицируемые кодом пользовательского режима.

Для каждого процесса, выполняющего Windows-программу, процесс подсистемы Windows (Csrss) поддерживает в дополнение к блоку EPROCESS параллельную структуру данных. Кроме того, часть подсистемы Windows, работающая в режиме ядра (Win32k.sys), поддерживает структуру данных для каждого процесса, которая создается при первом вызове потоком любой функции USER или GDI, реализованной в режиме ядра.

Рис. 1. Структуры данных сопоставляемые с процессами и потоками

Сначала рассмотрим блок процесса. Ключевые поля EPROCESS показаны на Рис. 2.

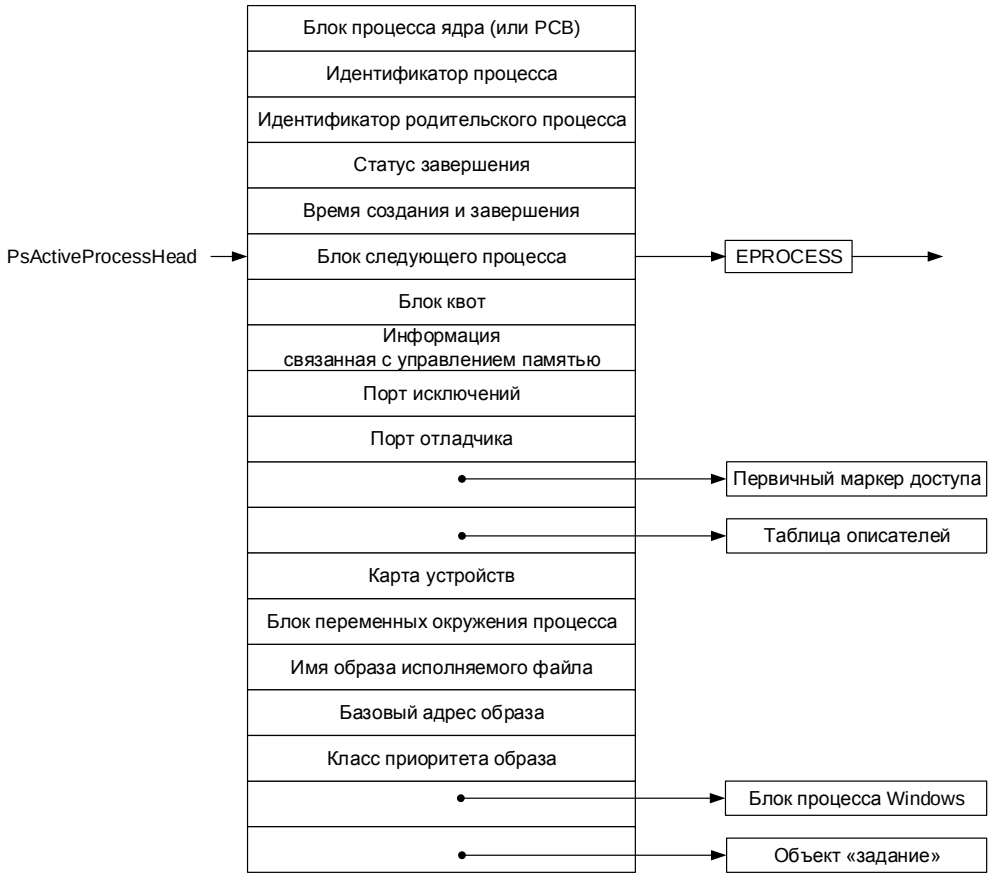

Рис. 2. Блок процесса (EPROCESS), создаваемый исполнительной системой

Таблица 1. Содержимое блока EPROCESS

Элемент	Описание
Блок процесса ядра (KPROCESS)	Общий заголовок объекта диспетчера, указатель на каталог страниц процесса, список блоков потоков ядра (KTHREAD), принадлежащих процессу, базовый приоритет по умолчанию выделяемый квант времени CPU, маска привязки к CPU, а также суммарное время работы потоков в режиме ядра и пользовательском режиме
Идентификация процесса	Уникальный идентификатор процесса, идентификатор процесса-создателя, имя выполняемого образа, имя WindowStation-объекта для данного процесса
Блок квот	Лимиты на пулы подкачиваемой и неподкачиваемой памяти, а также на использование страничного файла плюс текущее и пиковое использование пулов подкачиваемой и неподкачиваемой памяти процесса. Некоторые процессы могут разделять эту структуру: все системные процессы ссылаются на один общесистемный блок квот по умолчанию, а все процессы интерактивного сеанса используют один блок квот, определяемый Winlogon
Дескрипторы виртуальных адресов (virtual address descriptors)	Наборы структур данных, описывающих статус частей адресного пространства, которые существуют в процессе
Информация о рабочем наборе	Указатель на список рабочего набора (структура MNTWSL); текущий, пиковый, минимальный и максимальный размеры рабочего набора; время последнего усечения (last trim time); счетчик числа ошибок страниц; приоритет памяти; некоторые специфические флаги; хронология ошибок страниц

Информация о виртуальной памяти	Текущий и пиковый размер виртуальной памяти, использование страничного файла, элемент аппаратной таблицы страниц, указывающий на каталог страниц процесса
LPC-порт исключений	Канал межпроцессной связи, по которому диспетчер процессов посылает сообщение, если один из потоков этого процесса вызывает исключение
Отладочный LPC-порт	Канал межпроцессной связи, по которому диспетчер процессов посылает сообщение, если один из потоков этого процесса генерирует событие отладки
Маркер доступа (ACCESS TOKEN)	Объект исполнительной системы, описывающий профиль защиты данного процесса
Таблица описателей	Адрес таблицы описателей, принадлежащей процессу
Карта устройств	Адрес каталога объектов для разрешения ссылок на устройства по именам (поддерживается несколько пользователей)
Блок переменный окружения процесса (PEB)	Информация об образе исполняемого файла (базовый адрес, номера версий, список модулей), информация о куче процесса и TLS-памяти (указатели на кучи процесса начинаются с первого байта после PEB)
Блок процесса подсистемы Windows (W32PROCESS)	Дополнительная информация о процессе, необходимая той части подсистемы Windows, которая работает ь режиме ядра

Блок KPROCESS, входящий в блок EPROCESS, и PEB (process execution block), на который указывает EPROCESS, содержат дополнительные сведения об объекте «процесс». Блок KPROCESS, иногда называемый **блоком управления процессом** (process control block, PCB), показан на Рис. 3. Он содержит базовую информацию, нужную ядру Windows для планирования потоков.

PEB, размещаемый в адресном пространстве пользовательского процесса, содержит информацию, необходимую загрузчику образов, диспетчеру кучи и другим системным DLL-модулям Windows для доступа из пользовательского режима. (Блоки EPROCESS и KPROCESS доступны только из режима ядра.) Базовая структура PEB, показанная на Рис. 4.

Рис. 3. Блок управления процессом исполнительной системой

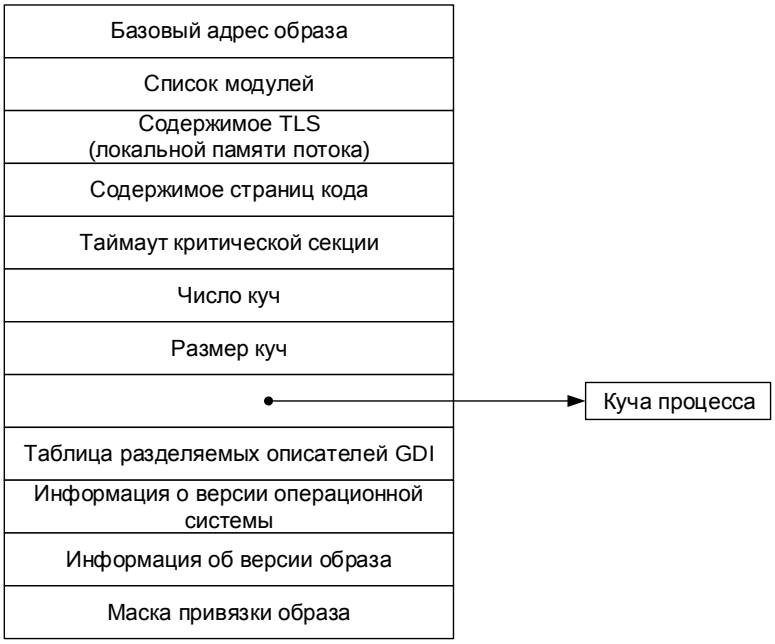

Рис. 4. Поля в блоке PEB

Переменные ядра

В Таблица 2 перечислено несколько важнейших глобальных переменных ядра, связанных с процессами. На эти переменные будем ссылаться по ходу изложения материала, в частности при описании этапов создания процесса.

Таблица 2. Переменные ядра, связанные с процессами

Переменная	Тип	Описание
PsActiveProcessHead	Заголовок очереди	Заголовок списка блоков процесса
PsIdleProcess	EPROCESS	Блок процесса Idle
PsInitialSystemProcess	Указатель на EPROCESS	Указатель на блок начального системного процесса (с идентификатором 2), который содержит системные потоки
PspCreateProcessNotifyRoutine	Массив указателей	Указатели на процедуры (максимум 8), вызываемые при создании и удалении процесса
PspCreateProcessNotifyRoutineCount	DWORD	Счетчик зарегистрированных процедур уведомления о создании процесса
PspLoadImageNotifyRoutine	Массив указателей	Указатели на процедуры, вызываемые при загрузке образа исполняемого файла
PspLoadImageNotifyRoutineCount	DWORD	Счетчик зарегистрированных процедур уведомления о загрузке образа
PspCidTable	Указатель на HANDLETABLE	Таблица описателей для клиентских идентификаторов процесса и потока

Счетчики производительности

Windows поддерживает **несколько счетчиков**, которые позволяют отслеживать процессы, выполняемые в системе; данные этих счетчиков можно считывать программно или просматривать с помощью оснастки Performance. В Таблица 3 перечислены счетчики производительности, имеющие отношение к процессам (кроме

счетчиков, связанных с управлением памятью и вводом-выводом, которые описываются далее).

Таблица 3. Счетчики производительности, связанные с процессами

Объект: счетчик	Описание
Process: % Privileged Time (Процесс: % работы в привилегированном режиме)	Процентная доля времени, в течение которого потоки данного процесса выполнялись в режиме ядра
Process: % Processor Time (Процесс: % загруженности CPU)	Процентная доля времени CPU, использованная потоками процесса за определенный период времени; вычисляется как сумма % Privileged Time и % User Time
Process: % User Time (Процесс: % работы в пользовательском режиме)	Процентная доля времени, в течение которого потоки данного процесса выполнялись в пользовательском режиме
Process: Elapsed Time (Процесс: Прошло времени)	Суммарное время (в секундах), прошедшее с момента создания процесса
Process: ID Process (Процесс: Идентификатор процесса)	Идентификатор процесса; полученное таким образом значение действительно лишь на время выполнения процесса, поскольку идентификаторы могут использоваться повторно
Process: Creating Process ID [Процесс: Код (ID) создавшего процесса]	Идентификатор родительского процесса; его значение не обновляется после завершения родительского процесса
Process: Thread Count (Процесс: Счетчик потоков)	Число потоков в процессе
Process: Handle Count (Процесс: Счетчик дескрипторов)	Число открытых процессом описателей

Сопутствующие функции

В Таблица 4 приведена информация по некоторым Windows-функциям, связанным с процессами [4].

Таблица 4. Функции, связанные с процессами

Функция	Описание
CreateProcess	Создает новый процесс и поток с использованием идентификации защиты вызывающего процесса
CreateProcessAsUser	Создает новый процесс и поток с указанным альтернативным маркером защиты
CreateProcessWithLogonW	Создает новый процесс и поток для выполнения под учетной записью, соответствующей указанным имени и паролю пользователя
CreateProccssWithTokenW	Создает новый процесс и поток с указанным альтернативным маркером защиты и поддерживает дополнительные возможности, например разрешает загрузку профиля пользователя
OpenProcess	Возвращает описатель указанного объекта «процесс»

ExitProcess	Завершает процесс с уведомлением всех подключенных DLL
TerminateProcess	Завершает процесс без уведомления подключенных DLL
FlushInstructionCache	Опустошает кэш команд указанного процесса
GetProcessTimes	Получает временные параметры процесса, описывающие, сколько времени процесс провел в режиме ядра и в пользовательском режиме
GetExitCodeProcess	Возвращает код завершения процесса, указывающий, как и почему завершился этот процесс
GetCommandLine	Возвращает указатель на командную строку, переданную текущему процессу
GetCurrentProcess	Возвращает псевдоописатель текущего процесса
GetCurrentProcessId	Возвращает идентификатор текущего процесса
GetProcessVersion	Возвращает старший и младший номера версии Windows, необходимой для запуска указанного процесса
GetStartupInfo	Возвращает содержимое структуры STARTUPINFO, заданное при вызове CreateProcess
GetEnvironmentStrings	Возвращает адрес блока переменных окружения
GetEnvironmentVariable	Возвращает значение указанной переменной окружения
GetProcessShutdownParameters и SetProcessShutdownParameters	Определяет приоритет завершения и число попыток для текущего процесса
GetGuiResources	Возвращает число открытых описателей USER и GDI

Что делает функция CreateProcess

Как процессы появляются на свет и как они завершаются, выполнив задачи, для которых они предназначались?

Создание Windows-процесса осуществляется вызовом одной из таких функций, как `CreateProcess`, `CreateProcessAsUser`, `CreateProcessWtihTokenW` или `CreateProcessWithLogonW`, и проходит в несколько этапов с участием трех компонентов ОС: Kernel32.dll (библиотеки клиентской части Windows), исполнительной системы и процесса подсистемы окружения Windows (Csrss). Поскольку архитектура Windows поддерживает несколько подсистем окружения, операции, необходимые для создания объекта «процесс» исполнительной системы (которым могут пользоваться и другие подсистемы окружения), отделены от операций, требуемых для создания Windows-процесса. Поэтому часть действий Windows-функции `CreateProcess` специфична для семантики, привносимой подсистемой Windows.

В приведенном ниже списке перечислены основные этапы создания процесса Windows-функцией `CreateProcess`.

ПРИМЕЧАНИЕ. Многие этапы работы `CreateProcess` связаны с подготовкой виртуального адресного пространства процесса и поэтому требуют понимания массы структур и терминов, связанных с управлением памятью и описываемых в лекции далее.

1. Открывается файл образа (EXE), который будет выполняться в процессе.

2. Создается объект «процесс» исполнительной системы.

3. Создается первичный поток (стек, контекст и объект «поток» исполнительной системы).

4. Подсистема Windows уведомляется о создании нового процесса и потока.

5. Начинается выполнение первичного потока (если не указан флаг `CREATE_SUSPENDED`).

6. В контексте нового процесса и потока инициализируется адресное пространство (например, загружаются требуемые DLL) и начинается выполнение программы.

Общая схема создания процесса в Windows показана на Рис. 5. Прежде чем открыть исполняемый образ для выполнения, `CreateProcess` делает следующее.

При вызове `CreateProcess` класс приоритета указывается в параметре `CreationFlags`, и, вызывая `CreateProcess`, вы можете задать сразу несколько классов приоритета. Windows выбирает самый низкий из них.

Когда для нового процесса не указывается класс приоритета, по умолчанию принимается `Normal`, если только класс приоритета процесса-создателя не равен `Idle` или `Below Normal`. В последнем случае новый процесс получает тот же класс приоритета, что и у родительского процесса.

Если для нового процесса указан класс приоритета `Real-time`, а создатель не имеет привилегии `Increase Scheduling Priority`, устанавливается класс приоритета `High`. Иначе говоря, функция `CreateProcess` завершается успешно, даже если у того, кто ее вызвал, недостаточно привилегий для создания процессов с классом приоритета `Real-time`, просто класс приоритета нового процесса будет ниже `Real-time`.

Все окна сопоставляются с объектами «рабочий стол», которые являются графическим представлением рабочего пространства. Если при вызове `CreateProcess` не указан конкретный объект «рабочий стол», новый процесс сопоставляется с текущим объектом «рабочий стол» процесса-создателя.

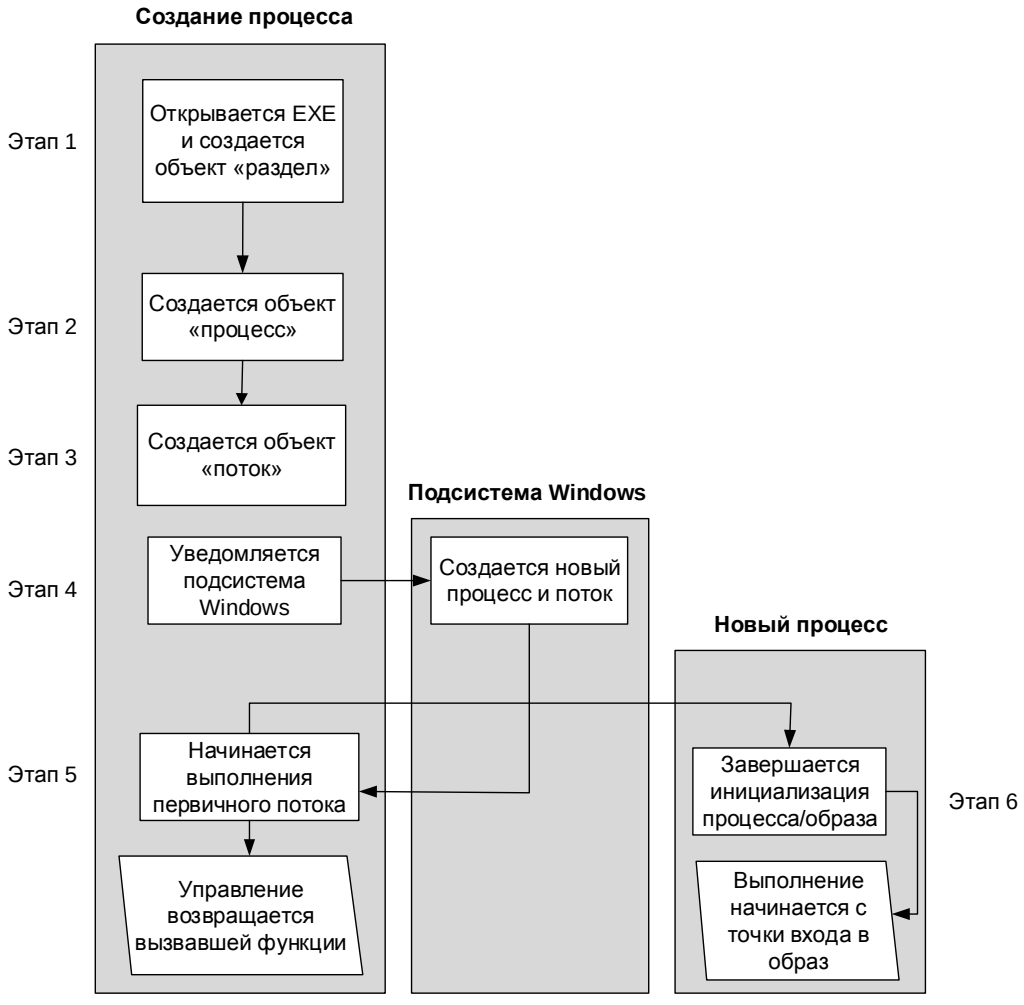

Рис. 5. Основные этапы создания процесса

Этап 1: открытие образа, подлежащего выполнению

Как показано на Рис. 6, на первом этапе `CreateProcess` должна найти нужный Windows-образ, который будет выполнять файл, указанный вызвавшим процессом, и создать объект «раздел» для его последующего проецирования на адресное пространство нового процесса. Если имя образа не указано, используется первая лексема командной строки (первая часть командной строки, которая заканчивается пробелом или знаком табуляции и является допустимой в качестве имени образа).

В Windows XP и Windows Server 2003 `CreateProcess` проверяет, не запрещает ли политика безопасности на данной машине запуск этого образа.

Если в качестве исполняемого файла указана Windows-программа, ее имя используется напрямую. А если исполняемый файл является не Windows-приложением, а программой MS-DOS, Win16 или POSIX, то `CreateProcess` ищет образ поддержки (support image) для запуска этой программы. Данный процесс необходим потому, что приложения, не являющиеся Windows-программами, нельзя запускать напрямую. Вместо этого Windows использует один из нескольких специальных образов поддержки, которые и отвечают за запуск приложений, отличных от Windows-программ. Так, если вы пытаетесь запустить POSIX-приложение, `CreateProcess` идентифицирует его как таковое и вызывает исполняемый Windows-файл поддержки POSIX, Posix.exe. А если запускается программа MS-DOS или Win16, стартует исполняемый Windows-файл поддержки Ntvdm.exe. Другими словами, нельзя напрямую создать процесс, не являющийся Windows-процессом. Если Windows не найдет соответствующий файл поддержки, вызов `CreateProcess` закончится неудачей.

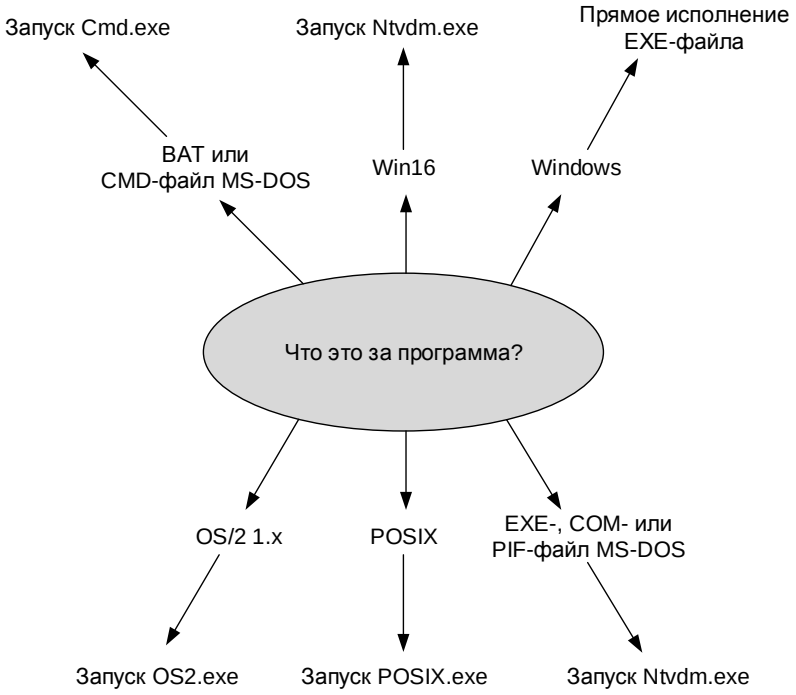

Рис. 6. Выбор активизируемого образа

Этап 2: создание объекта «процесс»

К началу второго этапа функция `CreateProcess` уже открыла допустимый исполняемый файл Windows и создала объект «раздел» для его проецирования на адресное пространство нового процесса. После этого она создает объект «процесс», чтобы запустить образ вызовом внутренней функции `NtCreateProcess`. Создание объекта «процесс» исполнительной системы включает следующие подэтапы:

- формируется блок EPROCESS;
- создается начальное адресное пространство процесса;
- инициализируется блок процесса ядра (KPROCESS);
- инициализируется адресное пространство процесса (в том числе список рабочего набора и дескрипторы виртуального адресного пространства), а также проецируется образ на это пространство;

- формируется блок PEB;
- завершается инициализация объекта «процесс» исполнительной системы.

ПРИМЕЧАНИЕ. Родительские процессы отсутствуют только при инициализации системы. Далее они всегда используются для задания контекстов защиты новых процессов.

Этап 2A: формирование блока EPROCESS

Этот подэтап включает девять операций.

1. Создается и инициализируется блок EPROCESS.

2. От родительского процесса наследуется маска привязки к CPUs.

3. Минимальный и максимальный размеры рабочего набора процесса устанавливаются равными значениям переменных `PsMinimumWorkingSet` и `PsMaximumWorkingSet`.

4. Блок квот нового процесса настраивается на адрес блока квот его родительского процесса и увеличивается счетчик ссылок на блок квот последнего.

5. Наследуется пространство имен устройств Windows (в том числе определение букв дисков, COM-портов и т. д.).

6. В поле `InheritedFromUniqueProcessId` нового объекта «процесс» сохраняется идентификатор родительского процесса.

7. Создается основной маркер доступа процесса (копированием аналогичного маркера родительского процесса). Новый процесс наследует профиль защиты своих родителей. Если используется функция `CreateProcessAsUser`, чтобы задать для нового процесса другой маркер доступа, он соответственно модифицируется.

8. Инициализируется таблица описателей, принадлежащая процессу. Если установлен флаг наследования описателей родительского процесса, наследуемые описатели из его таблицы копируются в новый процесс.

9. Статус завершения нового процесса устанавливается как `STATUS_PENDING`.

Этап 2B: создание начального адресного пространства процесса

Начальное адресное пространство процесса состоит из следующих страниц:

- каталога страниц (этих каталогов может быть больше одного в системах, где таблицы страниц имеют более двух уровней, например x86-ситемах в режиме PAE или в 64-разрядных системах);
- страницы гиперпространства;
- списка рабочего набора.

Для создания этих страниц выполняются операции, перечисленные ниже.

В соответствующих таблицах страниц формируются записи, позволяющие проецировать эти начальные страницы. Количество страниц вычитается из переменной ядра `MmTotalCommittedPages` и добавляется к переменной ядра `MmProcessCommit`.

Из `MmResidentAvailablePages` вычитается минимальный размер рабочего набора по умолчанию (`PsMinimumWorkingSet`).

На адресное пространство процесса проецируются страницы таблицы страниц для неподкачиваемой части системного пространства и системного кэша.

Этап 2C: создание блока процесса ядра

На этом подэтапе работы `CreateProcess` инициализируется блок KPROCESS, содержащий указатель на список потоков ядра. (Ядро не имеет представления об описателях, поэтому оно обходит их таблицу.) Блок процесса ядра также указывает на каталог таблицы страниц процесса (используемый для отслеживания виртуального адресного пространства процесса) и содержит суммарное время выполнения потоков процесса, базовый приоритет процесса по умолчанию (он начинается с Normal, или 8, если только его значение у родительского процесса не равно Idle или Below Normal; в последнем случае приоритет просто наследуется), привязку потоков к CPUs по умолчанию и начальный квант времени CPU, выделяемый процессу по умолчанию.

Последнее значение принимается равным `PspForegroundQuantum[0]`, первому элементу общесистемной таблицы величин квантов.

ПРИМЕЧАНИЕ. Начальный квант по умолчанию в клиентских и серверных версиях Windows неодинаков. Подробнее о квантах см. раздел «Планирование потоков» далее.

Этап 2D: инициализация адресного пространства процесса

Подготовка адресного пространства нового процесса довольно сложна, поэтому разберем ее отдельно по каждой операции. Для максимального усвоения материала этого раздела следует иметь представление о внутреннем устройстве диспетчера памяти Windows.

Диспетчер виртуальной памяти присваивает времени последнего усечения (last trim time) для процесса текущее время. Диспетчер рабочих наборов, выполняемый в контексте системного потока диспетчера настройки баланса (balance set manager), использует это значение, чтобы определить, когда нужно инициировать усечение рабочего набора.

Диспетчер памяти инициализирует список рабочего набора процесса, после чего становится возможной генерация ошибок страниц.

Раздел (созданный при открытии файла образа) проецируется на адресное пространство нового процесса, и базовый адрес раздела процесса приравнивается базовому адресу образа.

На адресное пространство процесса проецируется Ntdll.dll.

На адресное пространство процесса проецируются общесистемные таблицы NLS (national language support).

ПРИМЕЧАНИЕ. Процессы POSIX клонируют адресное пространство своих родителей, поэтому для них не нужны все вышеперечисленные операции создания нового адресного пространства. В случае приложений POSIX базовый адрес раздела нового процесса приравнивается тому же базовому адресу родительского процесса, а родительский PEB просто копируется.

Этап 2E: формирование блока PEB

`CreateProcess` выделяет страницу под PEB и инициализирует некоторые поля, описанные в Таблица 5.

Таблица 5. Начальные значения полей PEB

Поле	Начальное значение
ImagcBascAddress	Базовый адрес раздела
NumberOfProcessors	Значение переменной ядра `KeNumberProcessors`
NtGlobalFlag	Значение переменной ядра `NtGlobalFlag`
CriticalSectioriTimeout	Значение переменной ядра `MmCriticalSectionTimeout`
HeapSegmentReserve	Значение переменной ядра `MmHeapSegmentReserve`
HeapSegmentCommit	Значение переменной ядра `MmHeapSegmentCommit`
HeapDeCommitTotalFreeThreshold	Значение переменной ядра `MmHeapDeCommitTotalFreeThreshold`
HeapDcCommitFreeBlockThreshold	Значение переменной ядра `MmHcapDeCommitFreeBlockThreshold`
NumberOfHeaps	0

MaximumNumberOftieaps	(Размер страницы - размер PEB) / 4
ProcessHeaps	Первый байт после PEB
OSMajorVersion	Значение переменной ядра `NtMajorVersion`
OSMinorVersion	Значение переменной ядра `NtMinorVersion`
OSBuildNumber	Значение переменной ядра `NtBuildNumber & 0x3FFF`
OSPlatformId	2

Этап 2F: завершение инициализации объекта «процесс» исполнительной системы

Перед возвратом описателя нового процесса выполняется несколько завершающих операций.

1. Если общесистемный аудит процессов разрешен (через локальную политику безопасности или политику группы, вводимую контроллером домена), факт создания процесса отмечается в журнале безопасности.

2. Если родительский процесс входил в задание, новый процесс тоже включается в это задание.

3. Если в заголовке образа задан флаг `IMAGE_FILE_UP_SYSTEM_ONLY` (который указывает, что данную программу можно запускать только в однопроцессорной системе), для выполнения всех потоков процесса выбирается один CPU. Выбор осуществляется простым перебором доступных CPUs: при каждом запуске следующей программы такого типа выбирается следующий CPU. Благодаря этому подобные программы равномерно распределяются между CPUs.

4. Если в образе явно указана маска привязки к CPU (например, в поле конфигурационного заголовка), ее значение копируется в PEB и впоследствии устанавливается как маска привязки к CPUs по умолчанию.

5. `CreateProcess` помещает блок нового процесса в конец списка активных CPUs (`PsActiveProcessHead`).

6. Устанавливается время создания процесса, и вызвавшей функции (`CreateProcess` в Kernel32.dll) возвращается описатель нового процесса.

Этап 3: создание первичного потока, его стека и контекста

К началу третьего этапа объект «процесс» исполнительной системы полностью инициализирован. Однако у него еще нет ни одного потока, поэтому он не может ничего делать. Прежде чем создать поток, нужно создать стек и контекст, в котором он будет выполняться. Эта операция и является целью данного этапа. Размер стека первичного потока берется из образа—другого способа задать его размер нет.

Далее создается первичный поток вызовом `NtCreateThread`. Параметр потока — это адрес PEB (данный параметр нельзя задать при вызове `CreateProcess` — только при вызове `CreateThread`). Этот параметр используется кодом инициализации, выполняемым в контексте нового потока. Однако поток по-прежнему ничего не делает — он создается в приостановленном состоянии и возобновляется лишь по завершении инициализации процесса (см. этап 5). `NtCreateThread` вызывает `PspCreateThread` (функцию, которая используется и при создании системных потоков) и выполняет следующие операции:

1. Увеличивается счетчик потоков в объекте «процесс».

2. Создается и инициализируется блок потока исполнительной системы (ETHREAD).

3. Генерируется идентификатор нового потока.

4. В адресном пространстве пользовательского режима формируется TEB.

5. Стартовый адрес потока пользовательского режима сохраняется в блоке ETHREAD. В случае Windows-потоков это адрес системной стартовой функции потока в Kernel32.dll (`BaseProcessStart` для первого потока в процессе и `BaseThreadStart` для

дополнительных потоков). Стартовый адрес, указанный пользователем, также хранится в ETHREAD, но в другом его месте; это позволяет системной стартовой функции потока вызвать пользовательскую стартовую функцию.

6. Для подготовки блока KTHREAD вызывается `KelnitThread`. Начальный и текущий базовые приоритеты потока устанавливаются равными базовому приоритету процесса; привязка к процессорам и значение кванта также устанавливаются по соответствующим параметрам процесса. Кроме того, функция определяет идеальный процессор для первичного потока. Затем `KeInitThread` создает стек ядра для потока и инициализирует его аппаратно-зависимый контекст, включая фреймы ловушек и исключений. Контекст потока настраивается так, чтобы выполнение этого потока началось в режиме ядра в `KiThreadStartup`. Далее `KelnitThread` устанавливает состояние потока в Initialized (инициализирован) и возвращает управление `PspCreateThread`.

7. Вызываются общесистемные процедуры, зарегистрированные на уведомление о создании потока.

8. Маркер доступа потока настраивается как указатель на маркер доступа процесса. Затем вызывающая программа проверяется на предмет того, имеет ли она право создавать потоки. Эта проверка всегда заканчивается успешно, если поток создается в локальном процессе, но может дать отрицательный результат, если поток создается в другом процессе через функцию `CreateRemoteThread` и у создающего процесса нет привилегии отладки.

9. Наконец, поток готов к выполнению.

Этап 4: уведомление подсистемы Windows о новом процессе

Если заданы соответствующие правила, для нового процесса создастся маркер с ограниченными правами доступа. К этому моменту все необходимые объекты исполнительной системы созданы, и Kernel32.dll посылает подсистеме Windows сообщение, чтобы она подготовилась к выполнению нового процесса и потока. Сообщение включает следующую информацию:

- описатели процесса и потока;
- флаги создания;
- идентификатор родительского процесса;
- флаг, который указывает, относится ли данный процесс к Windows-приложениям (что позволяет Csrss определить, показывать ли курсор запуска).

Получив такое сообщение, подсистема Windows выполняет следующие операции.

1. `CreateProcess` дублирует описатели процесса и потока. На этом этапе счетчик числа пользователей процесса увеличивается с 1 (начального значения, установленного в момент создания процесса) до 2.

2. Если класс приоритета процесса не указан, `CreateProcess` устанавливает его в соответствии с алгоритмом, описанным ранее.

3. Создается блок процесса Csrss.

4. Порт исключений нового процесса настраивается как общий порт функций для подсистемы Windows, которая может таким образом получать сообщения при возникновении в процессе исключений.

5. Если в данный момент процесс отлаживается (т. е. подключен к процессу отладчика), в качестве общего порта функций выбирается отладочный порт. Такой вариант позволяет Windows пересылать события отладки в новом процессе (генерируемые при создании и удалении потоков, при исключениях и т.д) в виде сообщений подсистеме Windows, которая затем доставляет их процессу, выступающему в роли отладчика нового процесса.

6. Создается и инициализируется блок потока Csrss.

7. `CreateProcess` включает поток в список потоков процесса.

8. Увеличивается счетчик процессов в данном сеансе.

9. Уровень завершения процесса (process shutdown level) устанавливается как 0x280 (это значение по умолчанию; его описание ищите в документации MSDN Library по ключевому слову `SetProcessShutdownParameters`).

10. Блок нового процесса включается в список общесистемных Windows-процессов.

11. Создается и инициализируется структура данных (W32PROCESS), индивидуальная для каждого процесса и используемая той частью подсистемы Windows, которая работает в режиме ядра.

12. Выводится курсор запуска в виде стрелки с песочными часами. Тем самым Windows говорит пользователю: «Я запускаю какую-то программу, но ты все равно можешь пользоваться курсором.» Если в течение двух секунд процесс не делает GUI-вызова, курсор возвращается к стандартному виду. А если за это время процесс обратился к GUI, `CreateProcess` ждет открытия им окна в течение пяти секунд и после этого восстанавливает исходную форму курсора.

Этап 5: запуск первичного потока

К началу этого этапа окружение процесса уже определено, его потокам выделены ресурсы, у процесса есть поток, а подсистеме Windows известен факт существования нового процесса. Поэтому для завершения инициализации нового процесса (см. этап 6) возобновляется выполнение его первичного потока, если только не указан флаг `CREATE_SUSPENDED`.

Этап 6: инициализация в контексте нового процесса

Новый поток начинает свою жизнь с выполнения стартовой процедуры потока режима ядра, `KiThreadStartup`, которая понижает уровень IRQL потока с «DPC/dispatch» до «APC», а затем вызывает системную стартовую процедуру потока, `PspUserThreadStartup`. Параметром этой процедуры является пользовательский стартовый адрес потока.

В Windows XP и Windows Server 2003 `PspUserTbreadStartup` проверяет, разрешена ли в системе предварительная выборка для приложений (application prefetching), и, если да, вызывает **модуль логической предвыборки** (logical prefetcher) для **обработки файла команд предвыборки** (prefetch instruction file) (если таковой есть), а затем выполняет предвыборку страниц, на которые процесс ссылался в течение первых десяти секунд при последнем запуске. Наконец, `PspUserThreadStartup` ставит APC пользовательского режима в очередь для запуска процедуры инициализации загрузчика образов (`LdrlnitializeThunk` из Ntdll.dll). APC будет доставлен, когда поток попытается вернуться в пользовательский режим.

Когда `PspUserThreadStartup` возвращает управление `KiThreadStartup`, та возвращается из режима ядра, доставляет APC и обращается к `LdrlnitializeThunk`. Последняя инициализирует загрузчик, диспетчер кучи, таблицы NLS, **массив локальной памяти потока** (thread local storage, TLS) и структуры критической секции. После этого она загружает необходимые DLL и вызывает их точки входа с флагом `DLL_PROCESS_ATTACH`.

Наконец, когда процедура инициализации загрузчика возвращает управление диспетчеру APC пользовательского режима, начинается выполнение образа в пользовательском режиме. Диспетчер APC вызывает стартовую функцию потока, помещенную в пользовательский стек в момент доставки APC.

Внутреннее устройство потоков

Там, где явно не сказано обратное, считайте, что весь материал этого раздела в равной мере относится как к обычным потокам пользовательского режима, так и к системным потокам режима ядра.

Структуры данных

На уровне ОС поток представляется блоком потока, принадлежащим исполнительной системе (ETHREAD). Структура этого блока показана на Рис. 7

Блок ETHREAD и все структуры данных, на которые он ссылается, существуют в системном адресном пространстве, кроме **блока переменных окружения потока** (thread environment block, TEB) — он размещается в адресном пространстве процесса.

Помимо этого, процесс подсистемы Windows (Csrss) поддерживает параллельную структуру для каждого потока, созданного в Windows-процессе. Часть подсистемы Windows, работающая в режиме ядра (Win32k.sys), также поддерживает для каждого потока, вызывавшего USER- или GDI-функцию, структуру W32THREAD, на которую указывает блок ETHREAD.

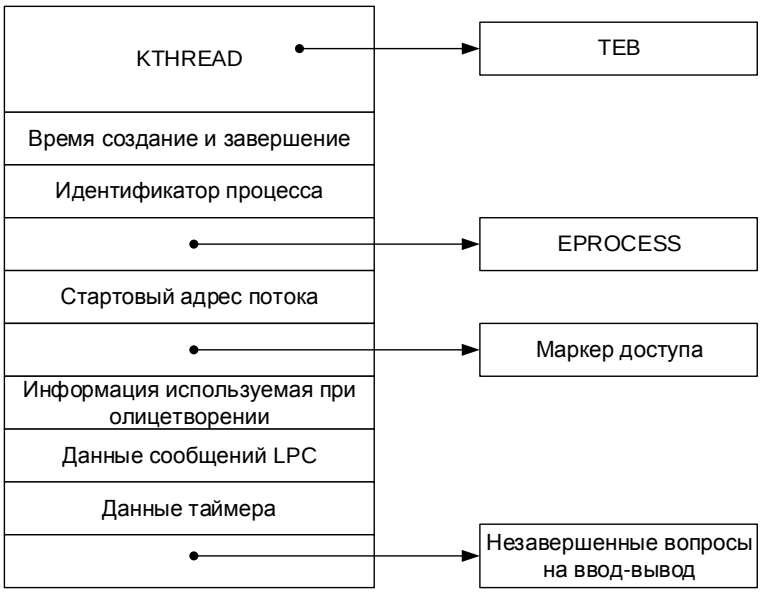

Рис. 7. Блок потока (ETHREAD) исполнительной системы

Поля блока потока ETHREAD, показанные на Рис. 7, в большинстве своем не требуют дополнительных пояснений. Первое поле - блок потока ядра (KTHREAD). За ним следуют идентификационные данные потока и процесса (включая указатель на процесс — владелец данного потока, что обеспечивает доступ к информации о его окружении), затем информация о защите в виде указателя на маркер доступа и сведения, необходимые для олицетворения (подмены одного процесса другим), а также поля, связанные с сообщениями LPC и незавершенными запросами на ввод-вывод.

Таблица 6. Ключевые поля блока потока ETHREAD

Элемент	Описание
KTHREAD	Таблица 7
Временные показатели потока	Время создания и завершения потока
Идентификационные данные процесса	Идентификатор процесса и указатель на блок EPROCESS процесса, которому принадлежит данный поток
Стартовый адрес	Адрес стартовой процедуры потока
Информация об олицетворении	Маркер доступа и уровень олицетворения (если поток олицетворяет, или подменяет, клиент)
Информация LPC	Идентификатор и адрес сообщения, ожидаемого потоком
Информация, связанная с вводом-выводом	Список необработанных пакетов запросов на ввод-вывод (I/O request packets, IRP)

Рассмотрим две ключевые структуры потока, упомянутым выше, — к блокам KTHREAD и TEB. Первый содержит информацию, нужную ядру Windows для планирования потоков и их синхронизации с другими потоками. Схема блока KTHREAD показана на Рис. 8.

Рис. 8. Схема блока потока

Ключевые поля блока KTHREAD кратко рассмотрены в Рис. 7.

Таблица 7. Ключевые поля блока KTHREAD

Элемент	Описание
Заголовок диспетчера	Поскольку поток — это объект, на котором можно ждать, он запускается со стандартным заголовком объекта диспетчера ядра
Время выполнения	Суммарное время выполнения (в режиме ядра и в пользовательском режиме)
Указатель на информацию стека ядра	Базовый и верхний адреса стека ядра
Указатель на таблицу системных сервисов	В начале выполнения каждого потока указатель в данном поле ссылается на главную таблицу системных сервисов `KeServiceDescriptorTable`, когда поток впервые вызывает GUI-сервис Windows, текущей таблицей системных сервисов становится та, которая включает сервисы GDT и USER в Win32k.sys
Информация, связанная с планированием	Базовый и текущий приоритеты, значение кванта, маска привязки к CPUs, идеальный CPU, статус планирования, счетчики числа замораживаний (freeze count) и приостановок (suspend count)
Блоки ожидания	В блоке потока содержится четыре встроенных блока ожидания, поэтому их не нужно создавать и инициализировать при каждом переходе потока в состояние ожидания какого-либо объекта (один блок ожидания принадлежит таймерам)
Информация об ожидании	Список объектов, ожидаемых потоком, причина ожидания и время перехода потока в состояние ожидания
Список мутантов	Список принадлежащих потоку объектов «мутант»

Очереди APC	Список незавершенных APC режима ядра и пользовательского режима, а также флаг оповещения (alertable flag)
Блок таймера	Встроенный блок таймера (а также соответствующий блок ожидания)
Список очередей	Указатель на сопоставленный с потоком объект очереди
Указатель на TEB	Идентификатор потока, данные TEB, указатель на PEB, информация, связанная с GDI и OpenGL

В отличие от других структур данных, описываемых в этом разделе, только блок TEB, показанный на Рис. 9, присутствует в адресном пространстве процесса, а не системы. В TEB хранится контекстная информация загрузчика образов и различных Windows DLL. Поскольку эти компоненты выполняются в пользовательском режиме, им нужны структуры данных, доступные для записи из этого режима. Вот почему TEB размещается в адресном пространстве процесса, а не системы, где он был бы доступен для записи только из режима ядра. Адрес TEB можно найти с помощью команды thread отладчика ядра.

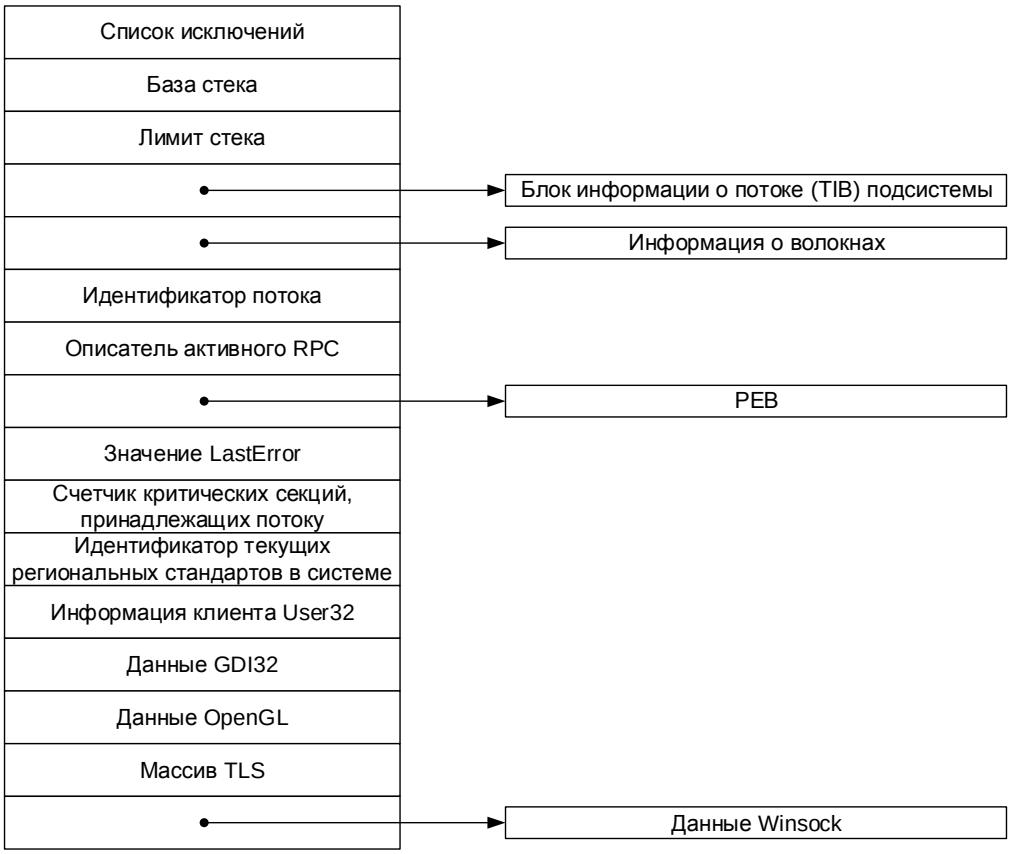

Рис. 9. Поля блока переменных окружения потоков

Переменные ядра

Как и в случае процессов, ряд переменных ядра Windows контролирует выполнение потоков. Список таких переменных, связанных с потоками, приводится в Таблица 8.

Таблица 8. Переменные ядра, относящиеся к потокам

Переменная	Тип	Описание
PspCreateThread NotifyRoutine	Массив указателей	Массив указателей на процедуры (максимум 8) вызываемых при создании и удалении потока
PspCreateThread	DWORD	Счетчик зарегистрированных процедур

NotifyRoutineCount		уведомления потока
PspCreateProcessNotifyRoutin	Массив указателей	Массив указателей на процедуры (максимум 8) вызываемых при создании и удалении процесса

Счетчики производительности

Большая часть важной информации в структурах данных потоков экспортируется в виде счетчиков производительности, перечисленных в Таблица 9. Даже используя только оснастку Performance, вы можете получить довольно много сведений о внутреннем устройстве потоков.

Таблица 9. Счетчики производительности для потоков

Объект: счетчик	Описание
Process: Priority Base (Процесс: Базовый приоритет)	Возвращает текущий базовый приоритет процесса; это начальный приоритет для потоков, создаваемых в данном процессе
Thread: % Privileged Time (Поток: % работы в привилегированном режиме)	Процентная доля времени, в течение которого поток выполнялся в режиме ядра
Thread: % Processor Time (Поток: % загруженности CPU)	Процентная доля времени CPU, использованная потоком за определенный период времени; вычисляется как сумма % Privileged Time и % User Time
Thread: % User Time (Поток: % работы в пользовательском режиме)	Процентная доля времени, в течение которого поток выполнялся в пользовательском режиме
Thread: Context Switches/Sec (Поток: Контекстных переключений/сек)	Число переключений контекстов в секунду в системе
Thread: Elapsed Time (Поток: Прошло времени)	Время CPU (в секундах), использованное потоком
Thread: ID Process (Поток: Идентификатор процесса)	Идентификатор процесса — владельца потока; действителен лишь на время существования процесса, так как идентификаторы процессов подлежат повторному использованию
Thread: ID Thread (Поток: Идентификатор потока)	Идентификатор потока; действителен лишь на время существования потока, так как идентификаторы потоков подлежат повторному использованию
Thread: Priority Base (Поток: Базовый приоритет)	Текущий базовый приоритет потока; его значение может отличаться от начального базового приоритета
Thread: Priority Current (Поток: Текущий приоритет)	Текущий динамический приоритет потока
Thread: Start Address (Поток: Начальный адрес)	Стартовый виртуальный адрес потока (у большинства потоков этот адрес одинаков)
Thread: Thread state (Поток: Состояние потока)	Текущее состояние потока — значение в интервале от 0 до 7
Thread: Thread Wait Reason (Поток: Причина состояния ожидания для потока)	Причина перехода потока в состояние ожидания — значение в интервале от 0 до 19

Сопутствующие функции

В Таблица 10 перечислены Windows-функции, позволяющие создавать потоки и манипулировать ими. Здесь не показаны функции, связанные с планированием и управлением приоритетами потоков.

Таблица 10. Функции Windows, относящиеся к потокам

Функция	Описание
CreateThread	Создает новый поток
CreateRemoteThread	Создает поток в другом процессе
ExitThread	Нормально завершает поток
TerminateThread	Аварийно завершает поток
GetExitCodeThread	Получает код завершения другого потока
GetThreadTimes	Возвращает временные характеристики другого потока
GetCurrentThread	Возвращает псевдоописатель текущего потока
GetCurrentThreadId	Возвращает идентификатор текущего потока
GetThreadId	Возвращает идентификатор указанного потока
GetThreadContext	Возвращает или изменяет регистры CPU SetThreadContext для данного потока
GetThreadSelectorEntry	Возвращает элемент таблицы дескрипторов другого потока (только для систем типа x86)

Рождение потока

Жизненный цикл потока начинается при его создании программой. Запрос на его создание в конечном счете поступает исполнительной системе Windows, где диспетчер процессов выделяет память для объекта «поток» и вызывает ядро для инициализации блока потока ядра. Ниже перечислены основные этапы создания потока Windows-функцией `CreateThread` (которая находится в Kernel32.dll).

1. `CreateThread` создает стек пользовательского режима в адресном пространстве процесса.

2. `CreateThread` инициализирует аппаратный контекст потока, специфичный для конкретной архитектуры CPU.

3. Для создания объекта «поток» исполнительной системы вызывается `NtCreateThread`. Он создается в приостановленном состоянии.

4. `CreateThread` уведомляет подсистему Windows о создании нового потока, и та выполняет некоторые подготовительные операции.

5. Вызвавшему коду возвращаются описатель и идентификатор потока (сгенерированный на этапе 3).

6. Выполнение потока возобновляется, и ему может быть выделено время CPU, если только он не был создан с флагом `CREATE_SUSPENDED`. Перед вызовом по пользовательскому стартовому адресу поток выполняет операции, описанные в разделе «Этап 3: создание первичного потока, его стека и контекста» ранее в этой главе.

Планирование потоков

Обзор планирования в Windows

В Windows реализована подсистема **вытесняющего планирования на основе уровней приоритета**, в которой всегда выполняется поток с наибольшим

приоритетом, готовый к выполнению. Однако выбор потока для выполнения может быть ограничен набором CPUs, на которых он может работать. Это явление называется **привязкой к CPUs** (processor affinity). По умолчанию поток выполняется на любом доступном CPU, но можно изменить привязку к CPUs через Windows-функции планирования, или заданием маски привязки в заголовке образа.

Выполняется поток, пока не наступит очередь другого потока с тем же приоритетом (или более высоким, что возможно в многопроцессорной системе). Длительность квантов зависит от трех факторов: конфигурационных параметров системы (длинные или короткие кванты), статуса процесса (активный или фоновый) и использования объекта «задание» для изменения длительности квантов. Однако поток может не полностью использовать свой квант. Поскольку в Windows реализован вытесняющий планировщик, то происходит вот что. Как только другой поток с более высоким приоритетом готов к выполнению, текущий поток вытесняется, даже если его квант еще не истек. Фактически поток может быть выбран следующим для выполнения и вытеснен, не успев воспользоваться своим квантом!

Код Windows, отвечающий за планирование, реализован в ядре. Поскольку этот код рассредоточен по ядру, единого модуля или процедуры с названием «планировщик» нет. Совокупность процедур, выполняющих эти обязанности, называется **диспетчерам ядра** (kernel's dispatcher). Диспетчеризация потоков может быть вызвана любым из следующих событий.

- **Поток готов к выполнению** — например, он только что создан или вышел из состояния ожидания.
- **Поток выходит из состояния Running** (выполняется), так как его квант истек или поток завершается либо переходит в состояние ожидания.
- **Приоритет потока изменяется** в результате вызова системного сервиса или самой Windows.
- **Изменяется привязка к CPUs**, из-за чего поток больше не может работать на CPU, на котором он выполнялся.

В любом случае Windows должна определить, какой поток выполнять следующим. Выбрав новый поток, Windows **переключает контекст**. Эта операция заключается в сохранении параметров состояния машины, связанных с выполняемым потоком, и загрузке аналогичных параметров для другого потока, после чего начинается выполнение нового потока.

Как уже говорилось, планирование в Windows осуществляется на уровне потоков. Этот подход станет понятен, если вы вспомните, что сами процессы не выполняются, а лишь предоставляют ресурсы и контекст для выполнения потоков. Поскольку решения, принимаемые в ходе планирования, касаются исключительно потоков, система не обращает внимания на то, какому процессу принадлежит тот или иной поток. Так, если у процесса A есть 10, у процесса B — 2 готовых к выполнению потока, и все 12 имеют одинаковый приоритет, каждый из потоков теоретически получит 1/12 времени CPU, потому что Windows не станет поровну делить время CPU между двумя процессами.

Чтобы понять алгоритмы планирования потоков, вы должны сначала разобраться в уровнях приоритета, используемых Windows.

Уровни приоритета

Как показано на Рис. 10, в Windows предусмотрено 32 уровня приоритета — от 0 до 31. Эти значения группируются так:

- шестнадцать уровней **реального времени** (16-31);
- пятнадцать **варьируемых (динамических)** уровней (1—15);
- один **системный уровень** (0), зарезервированный для потока обнуления страниц (zero page thread).

Рис. 10. Уровни приоритета потоков

Уровни приоритета потока назначаются с учетом двух разных точек зрений Windows API и ядра Windows. Windows API сначала упорядочивает процессы по классам приоритета, назначенным при их создании [Real-time (реального времени), High (высокий), Above Normal (выше обычного), Normal (обычный), Below Normal (ниже обычного) и Idle (простаивающий)], а затем — по относительному приоритету индивидуальных потоков в рамках этих процессов [Time-critical (критичный по времени), Highest (наивысший), Above-normal (выше обычного), Normal (обычный), Below-normal (ниже обычного), Lowest (наименьший) и Idle (простаивающий)].

Базовый приоритет каждого потока в Windows API устанавливается, исходя из класса приоритета его процесса и относительного приоритета самого потока. Связь между приоритетами Windows API и внутренними приоритетами ядра Windows (в числовой форме) показана на Рис. 11.

Если у процесса только одно значение приоритета (базовое), то у каждого потока их два: текущее и базовое. Решения, связанные с планированием, принимаются на основе текущего приоритета. В определенных обстоятельствах система может на короткое время повышать приоритеты потоков в динамическом диапазоне (1-15). Windows никогда не изменяет приоритеты потоков в диапазоне реального времени (16-31), поэтому у таких потоков базовый приоритет идентичен текущему.

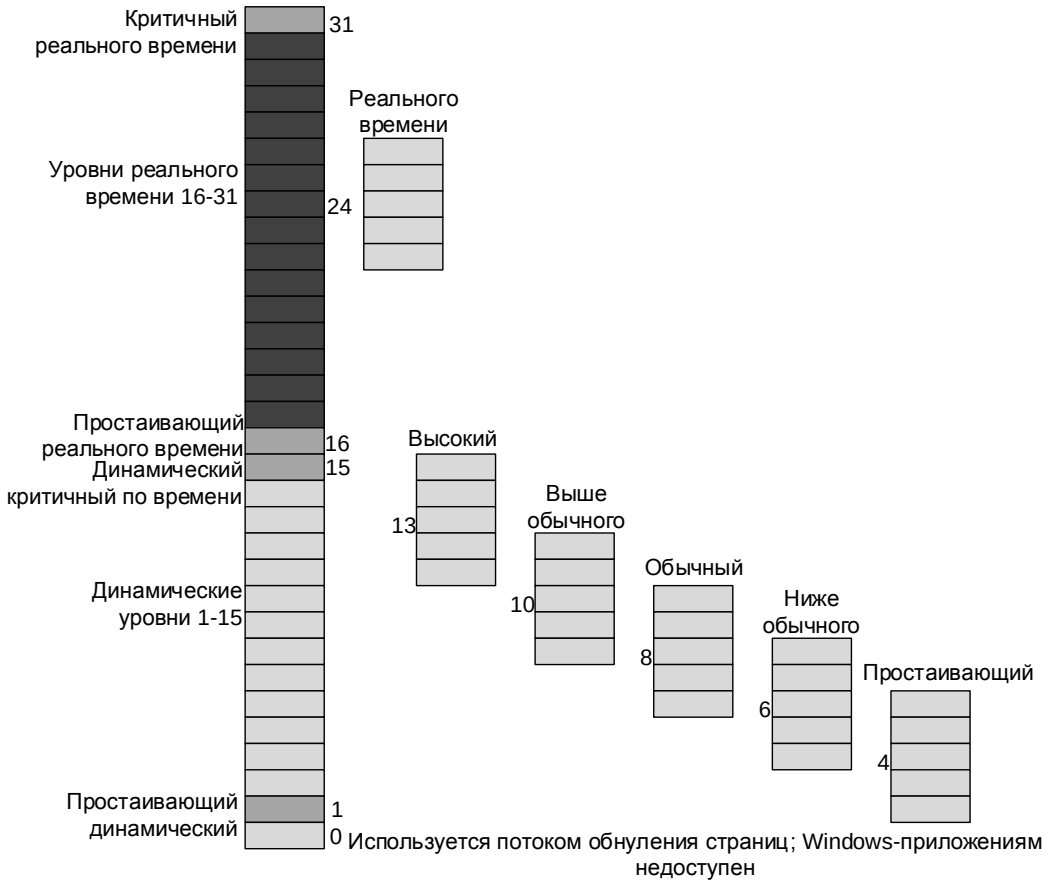

Рис. 11. Взаимосвязь приоритетов в ядре и Windows API

Начальный базовый приоритет потока наследуется от базового приоритета процесса, а тот наследует его от родительского процесса. Это поведение можно изменить при вызове Windows-функции `CreateProcess` или команды `START`. Приоритет процесса можно изменить и после его создания, используя функцию `SetPriorityClass` или различные утилиты, предоставляющие доступ к этой функции через UI, например диспетчер задач и Process Explorer. В частности, можете понизить приоритет процесса, интенсивно использующего время CPU, чтобы он не мешал обычным операциям в системе. Смена приоритета процесса влечет за собой смену приоритетов всех его потоков, но их относительные приоритеты остаются прежними. Но изменение приоритетов индивидуальных потоков внутри процесса обычно не имеет смысла, потому что вы не знаете, чем именно занимается каждый из них (если только сами не пишете программу или не располагаете исходным кодом); так что изменение относительных приоритетов потоков может привести к неадекватному поведению этого приложения.

Обычно базовый приоритет процесса (а значит, и базовый приоритет первичного потока) по умолчанию равен значению из середины диапазонов приоритетов процессов (24, 13, 10, 8, 6 или 4). Однако базовый приоритет некоторых системных процессов (например, диспетчера сеансов, контроллера сервисов и сервера локальной аутентификации) несколько превышает значение по умолчанию для класса Normal (8). Более высокий базовый приоритет по умолчанию обеспечивает запуск потоков этих процессов с приоритетом выше 8. Чтобы изменить свой начальный базовый приоритет, такие системные процессы используют внутреннюю функцию `NtSetInformationProcess`.

Функции Windows API, связанные с планированием

Эти функции перечислены в Таблица 11 (более подробную информацию см. в справочной документации Windows API).

Таблица 11. API-функции планирования и их назначение

API-функция	Описание
SuspenThread/ ResumeThread	Приостанавливает/возобновляет поток
GetPriorityClass/ SetPriorityClass	Возвращает/устанавливает класс приоритета процесса (базовый приоритет)
GetThreadPriority/ SetThreadPriority	Возвращает/устанавливает приоритет потока (относительный базовому приоритету его процесса)
GetProcessAffinityMask/ SetProcessAffinityMask	Возвращает/устанавливает маску привязки процесса к CPU
SetThreadAffinityMask	Устанавливает маску привязки потока (которая должна быть подмножеством маски привязки процесса) к конкретному набору CPU, ограничивая доступные для выполнения этого потока CPUs
SetInformationJobObject	Задает атрибуты для задания; некоторые из них влияют на планирование, изменяя, например, привязку к CPU и приоритет (описание объекта «задание» см. в разделе «Объекты-задания» далее в этой главе)
GetLogicalProcessorInformation	Возвращает детальные сведения о конфигурации CPU [для систем с поддержкой логических CPUs (hyperthreaded systems) и NUMA]
GetThreadPriorityBoost/ SetThreadPriorityBoost	Возвращает детальную информацию о динамическом повышении приоритета потока или разрешает такое повышение приоритета (только для потоков с приоритетами из динамического диапазона)
SetThreadIdealProcessor	Задает предпочтительный CPU для данного потока, но не

	ограничивает им набор CPUs, доступных этому потоку
GetProcessPriorityBoost SetProcessPriorityBoost	Возвращает/устанавливает статус динамического повышения приоритета данного процесса по умолчанию (используется для установки этого статуса при создании потока)
SwitchToThread	Переключает CPU на выполнение другого потока (с приоритетом от 1 и выше), готового к выполнению на текущем CPU
Sleep	Переводит текущий поток в состояние ожидания на указанное время (в мс); нулевое значение отбирает у потока остаток его кванта
SleepEx	Переводит текущий поток в состояние ожидания до тех пор, пока не закончится операция ввода-вывода, пока не истечет указанный временной интервал или пока в очереди потока не появится APC

Приоритеты реального времени

Можно повысить или понизить приоритет потока любого приложения в динамическом диапазоне; однако, чтобы задать значение из диапазона реального времени, должна быть привилегия Increase Scheduling Priority. Учтите, что многие важные системные потоки режима ядра выполняются в диапазоне приоритетов реального времени. **Поэтому, если потоки слишком долго выполняются с приоритетом этого диапазона, они могут блокировать критичные системные функции** (например, в диспетчере памяти, диспетчере кэша или драйверах устройств).

ПРИМЕЧАНИЕ. Как показано на следующей иллюстрации, где изображены уровни запросов прерываний (Interrupt Request Levels, IRQL) на платформе x86, в Windows имеется набор приоритетов, называемых приоритетами реального времени, но они не являются таковыми в общепринятом смысле этого термина, так как Windows не относится к ОС реального времени.

Состояния потоков

Прежде чем перейти к алгоритмам планирования потоков, следует разобраться, в каких состояниях могут находиться потоки в процессе выполнения в Windows 2000 и Windows XP. Соответствующая схема дана на Рис. 12 [числовые значения отражают показатели счетчика производительности Thread: thread state (Поток: Состояние потока)].

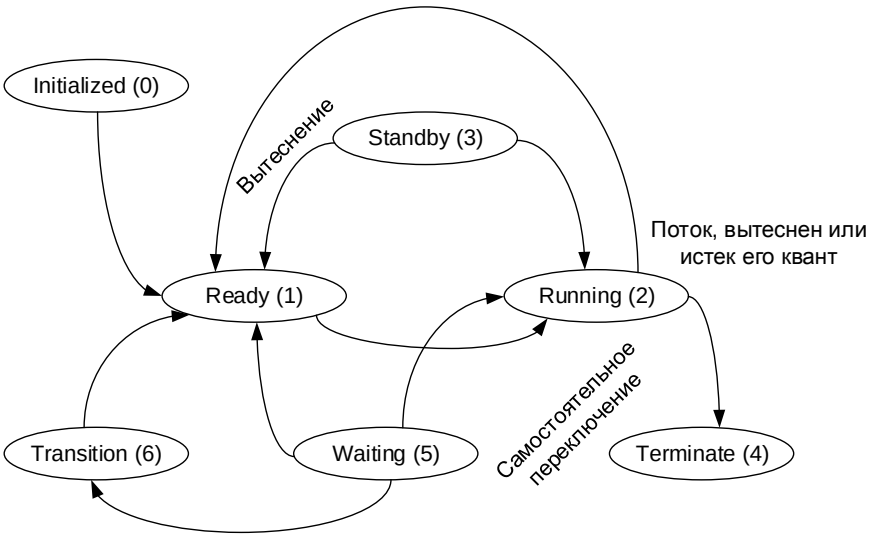

Рис. 12. Состояния потоков в Windows 2000 и Windows XP

Ready (готов) Поток в состоянии готовности ожидает выполнения. Выбирая следующий поток для выполнения, диспетчер принимает во внимание только пул потоков, готовых к выполнению.

Standby (простаивает) Поток в этом состоянии уже выбран следующим для выполнения на конкретном CPU. В подходящий момент диспетчер переключает контекст на этот поток. В состоянии Standby может находиться только один поток для каждого CPU в системе. Заметьте, что поток может быть вытеснен даже в этом состоянии (если, например, до начала выполнения потока, который пока находится в состоянии Standby, к выполнению будет готов поток с более высоким приоритетом).

Running (выполняется) Поток переходит в это состояние и начинает выполняться сразу после того, как диспетчер переключает на него контекст. Выполнение потока прекращается, как только он завершается, вытесняется потоком с более высоким приоритетом, переключает контекст на другой поток, самостоятельно переходит в состояние ожидания или истекает выделенный ему квант времени CPU (и другой поток с тем же приоритетом готов к выполнению).

Waiting (ожидает) Поток входит в состояние Waiting несколькими способами. Он может самостоятельно начать ожидание на синхронизирующем объекте или его вынуждает к этому подсистема окружения. По окончании ожидания поток — в зависимости от приоритета — либо немедленно начинает выполняться, либо переходит в состояние Ready.

- **Transition (переходное состояние)** Поток переходит в это состояние, если он готов к выполнению, но его стек ядра выгружен из памяти. Как только этот стек загружается в память, поток переходит в состояние Ready.
- **Terminated (завершен)** Заканчивая выполнение, поток переходит в состояние Terminated. После этого блок потока исполнительной системы (структура данных в пуле неподкачиваемой памяти, описывающая данный поток) может быть удален, а может быть и не удален — это уже определяется диспетчером объектов.
- **Initialized (инициализирован)** В это состояние поток входит в процессе своего создания.

Схема состояний потоков в Windows Server 2003 показана на Рис. 13. Обратите внимание на новое состояние Deferred Ready (готов, отложен). Это состояние используется для потоков, выбранных для выполнения на конкретном CPU, но пока не запланированных к выполнению. Это новое состояние предназначено для того, чтобы ядро могло свести к минимуму срок применения общесистемной блокировки к базе данных планирования (scheduling database).

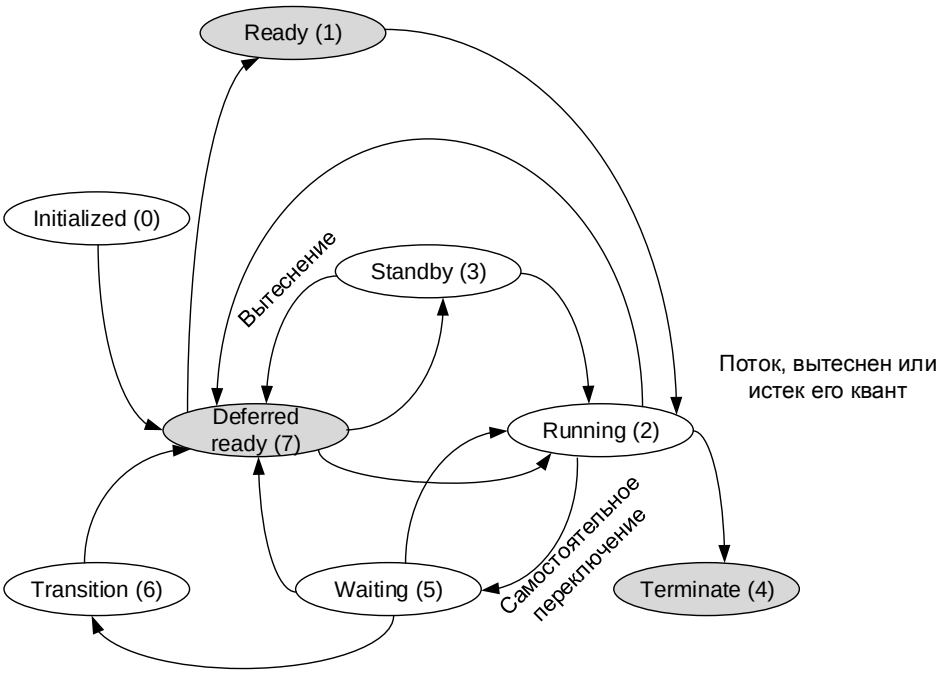

Рис. 13. Состояния потоков в Windows Server 2003

База данных диспетчера ядра

Для принятия решений при планировании потоков ядро поддерживает набор структур данных, в совокупности известных как **база данных диспетчера ядра** (dispatcher database) (Рис. 14). Эта БД позволяет отслеживать потоки, ждущие выполнения, и потоки, выполняемые на тех или иных CPUs.

ПРИМЕЧАНИЕ. БД диспетчера ядра в однопроцессорной системе имеет ту же структуру, что и в многопроцессорных системах Windows 2000 и Windows XP, но отличается от структуры такой базы данных в системах Windows Server 2003.

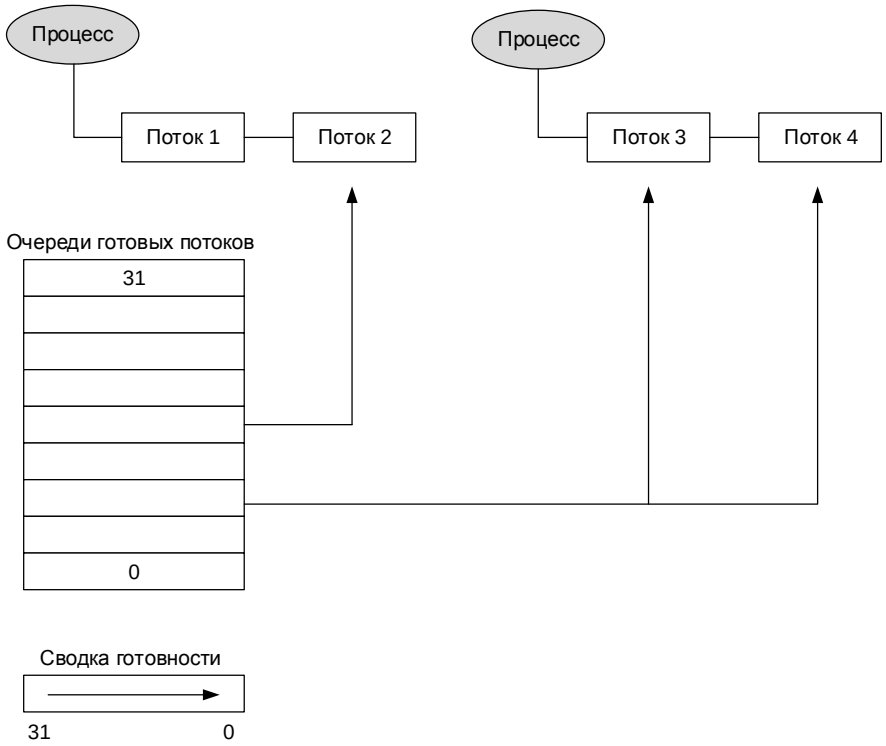

Рис. 14. База данных диспетчера ядра (однопроцессорная и многопроцессорная в Windows 2000/XP)

Очереди готовых потоков (ready queues) диспетчера ядра включают потоки в состоянии Ready, ожидающие выделения им времени CPU. Для каждого из 32 уровней приоритета существует по одной очереди. Для ускорения выбора потока, подлежащего выполнению или вытеснению, Windows поддерживает 32-битную маску, называемую **сводкой готовности** (ready summary) `KiReadySummary`. Каждый установленный в ней бит указывает на присутствие одного или более потоков в очереди готовых потоков для данного уровня приоритета (бит 0 соответствует приоритету 0, бит 1 — приоритету 1 и т.д.).

В однопроцессорных системах база данных диспетчера ядра синхронизируется повышением IRQL до уровня «DPC/dispatch» и SYNCH_LEVEL (оба определены как уровень 2). Такое повышение IRQL не дает другим потокам прервать диспетчеризацию потоков, так как потоки обычно выполняются при IRQL 0 или 1. В многопроцессорных системах одного повышения IRQL мало, потому что каждый CPU может одновременно увеличить IRQL до одного уровня и попытаться обратиться к базе данных диспетчера ядра.

Квант

Квант — это интервал времени CPU, отведенный потоку для выполнения. По его окончании Windows проверяет, ожидает ли выполнения другой поток с таким же уровнем приоритета. Если на момент истечения кванта других потоков с тем же уровнем приоритета нет, Windows выделяет текущему потоку еще один квант.

По умолчанию в Windows 2000 Professional и Windows XP потоки выполняются в течение 2 **интервалов таймера** (clock intervals), а в системах Windows Server - 12. **В серверных системах величина кванта увеличена для того, чтобы свести к минимуму переключение контекста.** Получая больший квант, серверные

приложения, которые пробуждаются при получении клиентского запроса, имеют больше шансов выполнить запрос и вернуться в состояние ожидания до истечения выделенного кванта.

Длительность интервала таймера зависит от аппаратной платформы и определяется HAL, а не ядром. Например, этот интервал на большинстве однопроцессорных x86-систем составляет 10 мс, а на большинстве многопроцессорных x86-систем — около 15 мс.

Учет квантов времени

Величина кванта для каждого процесса хранится в блоке процесса ядра. Это значение используется, когда потоку предоставляется новый квант. Когда поток выполняется, его квант уменьшается по истечении каждого интервала таймера, и в конечном счете срабатывает алгоритм обработки завершения кванта. Если имеется другой поток с тем же приоритетом, ожидающий выполнения, происходит переключение контекста на следующий поток в очереди готовых потоков. Заметьте: когда системный таймер прерывает DPC или процедуру обработки другого прерывания, квант выполнившегося потока все равно уменьшается, даже если этот поток не успел отработать полный интервал таймера. Если бы это было не так и если бы аппаратное прерывание или DPC появилось непосредственно перед прерыванием таймера, квант потока мог бы вообще никогда не уменьшиться.

Внутренне величина кванта хранится как число тактов таймера, умноженное на 3. То есть в Windows 2000 и Windows XP потоки по умолчанию получают квант величиной 6 (2 • 3), в Windows Server — 36 (12 • 3). Всякий раз, когда возникает прерывание таймера, процедура его обработки вычитает из кванта потока постоянную величину (3).

Это сделано для того, чтобы можно было уменьшать значение кванта по завершении ожидания. Когда поток с текущим приоритетом ниже 16 и базовым приоритетом ниже 14 запускает функцию ожидания (`WaitForSingleObject` или `WaitForMultipleObjects`) и его запрос на доступ удовлетворяется немедленно (например, он не переходит в состояние ожидания), его квант уменьшается на одну единицу. Благодаря этому кванты ожидающих потоков в конечном счете заканчиваются.

Управление величиной кванта

Можно изменить квант для потоков всех процессов, но выбор ограничен всего двумя значениями: короткий квант (2 такта таймера, используется по умолчанию для клиентских компьютеров) или длинный (12 тактов таймера, используется по умолчанию для серверных систем).

ПРИМЕЧАНИЕ. Используя объект «задание» в системе с длинными квантами, можете указать другие величины квантов для процессов в задании.

Для изменения величины кванта в Windows XP или Windows Server 2003 щелкните правой кнопкой мыши My Computer (Мой компьютер), выберите Properties (Свойства), перейдите на вкладку Advanced (Дополнительно), щелкните кнопку Settings (Параметры) в разделе Performance (Быстродействие), а зятем перейдите на вкладку Advanced (Дополнительно). Соответствующие диалоговые окна в Windows XP и Windows Server 2003 немного различаются. Они показаны на рис. 15.

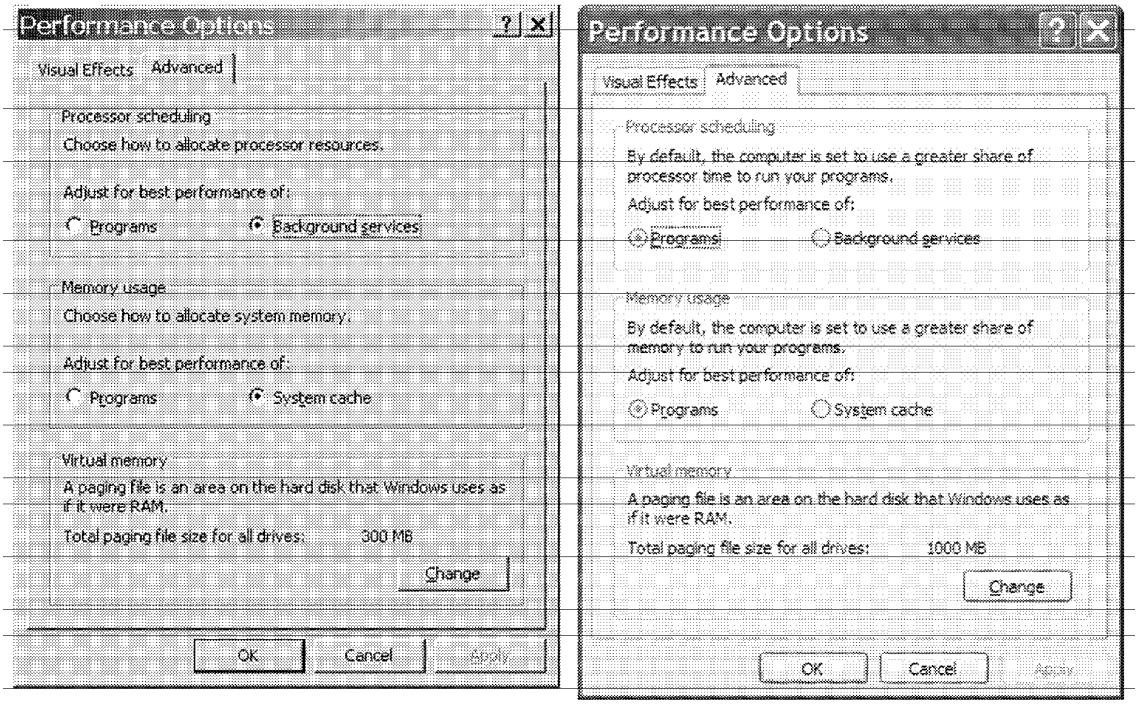

Рис. 15. Задание кванта в Windows XP/Windows Server 2003

Параметр **Background Services** (Фоновых служб) подразумевает применение длинных квантов фиксированного размера, что предлагается по умолчанию в системах Windows Server. Единственная причина, по которой имело бы смысл выбрать этот параметр на рабочей станции, — ее использование в качестве серверной системы.

Еще одно различие между параметрами Programs и Background Services заключается в том, какой эффект они оказывают на кванты потоков в активном процессе.

Динамическое увеличение кванта

До Windows NT 4.0, когда на рабочей станции или в клиентской системе какое-то окно становилось активным, приоритет всех потоков активного процесса (которому принадлежит поток, владеющий окном в фокусе ввода) динамически повышался на 2. Повышенный приоритет действовал до тех пор, пока любому потоку процесса принадлежало активное окно. Проблема с этим подходом была в том, что, если вы запустили длительный процесс, интенсивно использующий CPU (например, начали пересчет электронной таблицы), и переключились на другой процесс, требующий больших вычислительных ресурсов (скажем, на одну из программ CAD, графический редактор или какую-нибудь игру), то первый процесс, ставший теперь фоновым, получит лишь очень малую часть времени CPU (или вообще не получит его). А все дело в том, что приоритет потоков активного процесса повышается на 2 (здесь предполагается, чти базовый приоритет потоков обоих процессов был одинаковым).

Это поведение по умолчанию изменилось с появлением Windows NT 4.0 Workstation — кванты потоков активного процесса стали увеличиваться в 3 раза. Таким образом, по умолчанию на рабочих станциях их квант достигал 6 тактов таймера, а у потоков остальных процессов — 2 тактов. Благодаря этому, когда процесс, интенсивно использующий ресурсы CPU, оказывается фоновым, новый активный процесс получает пропорционально большее время CPU (и вновь предполагается, что приоритеты потоков одинаковы как в активном, так и в фоновом процессе).

Заметьте, что это изменение квантов относится лишь к процессам с приоритетом выше Idle в системах с установленным параметром Programs (или Applications в Windows 2000) в диалоговом окне Performance Options (Параметры быстродействия), как пояснялось в предыдущем разделе. Кванты потоков активного процесса в системах с установленным параметром Background Services (настройка по умолчанию в системах Windows Server) не изменяются.

Параметр реестра для настройки кванта

Пользовательский интерфейс, позволяющий изменить относительную величину кванта, модифицирует в реестре параметр `HKLM\SYSTEM\CurrentCont-`

rolSet\Control\PriorityControl\Win32PrioritySeparation. Этот же параметр определяет, можно ли динамически увеличивать (и, если да, то насколько) кванты потоков, выполняемых в активном процессе. Данный параметр содержит 3 двухбитных поля.

- **Короткие или длинные**. Значение 1 указывает на длинные кванты, а 2 — на короткие. Если это поле равно 0 или 3, используются кванты по умолчанию (короткие в Windows 2000 Professional или Windows XP и длинные в системах Windows Server).
- **Переменные или фиксированные**. Если задано значение 1, кванты потоков активного процесса могут варьироваться, а если задано значение 2 — нет. Если это поле равно 0 или 3, используется настройка по умолчанию (переменные в Windows 2000 Professional или Windows XP и фиксированные и системах Windows Server).

Динамическое приращение кванта потока активного процесса. Это поле (хранящееся в переменной ядра PsPrioritySeparation) может быть равно 0, 1 или 2 (значение 3 недопустимо и интерпретируется как 2) и представляет собой индекс в трехэлементном байтовом массиве (PspForegroundQuanturri), используемом для расчета квантов потоков активного процесса. Кванты потоков фоновых процессов определяются первым элементом этого массива.

Заметьте, что при использовании диалогового окна Performance Options (Параметры быстродействия) доступны лишь две комбинации: короткие кванты с утроением в активном процессе или длинные без изменения в таком процессе. Но прямое редактирование параметра Win32PrioritySeparation в реестре позволяет выбирать и другие комбинации.

Сценарии планирования

Известно, что вопрос «Какому потоку отдать CPU время?» Windows 2000 решает, исходя из приоритетов. Но как этот подход работает на практике? Следующие разделы иллюстрируют, как вытесняющая многозадачность, управляемая на основе приоритетов, действует на уровне потоков.

Самостоятельное переключение

Во-первых, поток может самостоятельно освободить CPU, перейдя в состояние ожидания на каком-либо объекте (например, событии, мьютексе, семафоре, порте завершения ввода-вывода, процессе, потоке, оконном сообщении и др.) путем вызова одной из многочисленных Windows-функций ожидания (скажем, WaitForSingleObject или WaitForMultipleObjects).

На Рис. 16 показано, как поток входит в состояние ожидания и как Windows выбирает новый поток для выполнения.

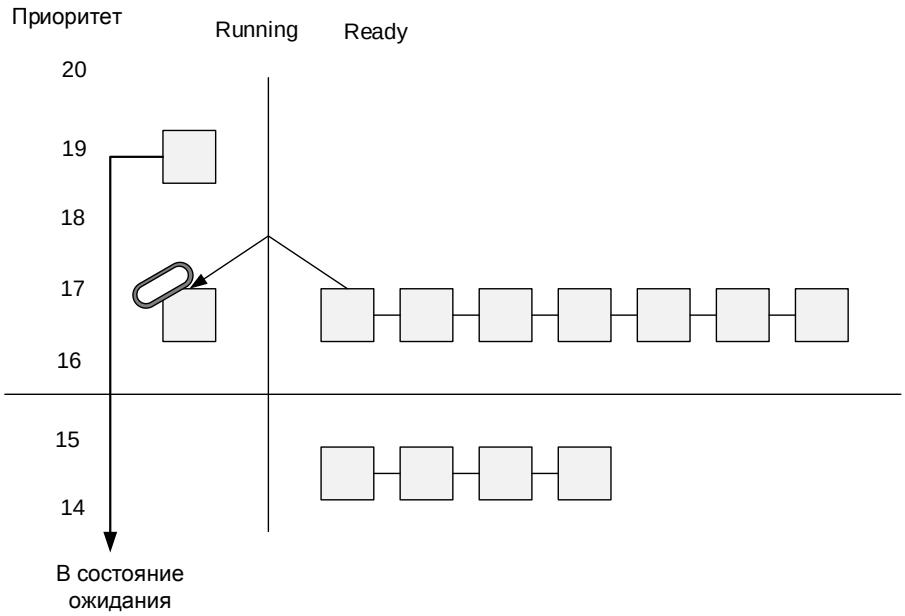

Рис. 16. Самостоятельное переключение

На Рис. 16 поток (верхний блок) самостоятельно освобождает процессор, в результате чего к процессору подключается другой поток из очереди (отмеченный кольцом в колонке Running). Когда поток входит в состояние ожидания, квант не сбрасывается. Как уже говорилось, после успешного завершения ожидания квант потока уменьшается на одну единицу, что эквивалентно трети интервала таймера (исключение составляют потоки с приоритетом от 14 и выше, у которых после ожидания квант сбрасывается).

Вытеснение

В этом сценарии поток с более низким приоритетом вытесняется готовым к выполнению потоком с более высоким приоритетом. Такая ситуация может быть следствием двух обстоятельств:

- завершилось ожидание потока с более высоким приоритетом (т.е. произошло событие, которого он ждал);
- приоритет потока увеличился или уменьшился.

В любом из этих случаев Windows решает, продолжить выполнение текущего потока или вытеснить его потоком с более высоким приоритетом.

ПРИМЕЧАНИЕ. Потоки пользовательского режима могут вытеснять потоки режима ядра. То есть режим выполнения потока значения не имеет — определяющим фактором является его приоритет.

Когда поток вытесняется, он помещается в начало очереди готовых потоков соответствующего уровня приоритета. Эту ситуацию иллюстрирует Рис. 17.

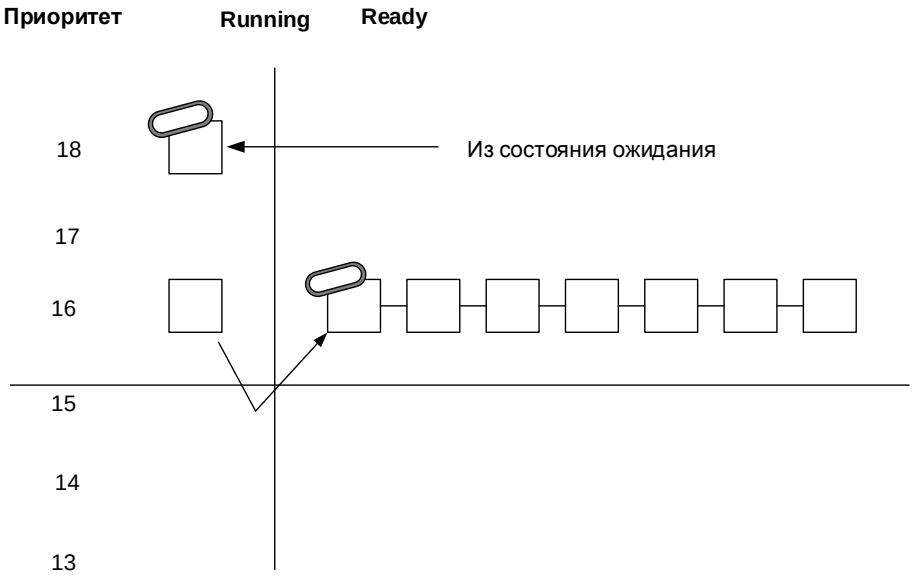

Рис. 17. Планирование потоков с вытеснением

На Рис. 17 поток с приоритетом 18 выходит из состояния ожидания и вновь захватывает процессор, вытесняя выполняемый в этот момент поток (с приоритетом 16) в очередь готовых потоков. Заметьте, что вытесненный поток помещается не в конец, а в начало очереди. После завершения вытеснившего потока вытесненный сможет отработать остаток своего кванта.

Завершение кванта

Когда поток израсходует свой квант времени CPU, Windows должна решить, следует ли понизить его приоритет и подключить к CPU другой поток.

Снизив приоритет потока, Windows ищет более подходящий для выполнения поток (таким потоком, например, будет любой из очереди готовых потоков с приоритетом выше нового приоритета текущего потока). Если Windows оставляет приоритет потока прежним и в очереди готовых потоков есть другие потоки с тем же приоритетом, она выбирает из очереди следующий поток с тем же приоритетом, а выполнявшийся до этого поток перемещает в хвост очереди (задавая ему новую величину кванта и переводя его из состояния Running в состояние Ready). Этот случай иллюстрирует

Рис. 18. Если ни один поток с тем же приоритетом не готов к выполнению, текущему потоку выделяется еще один квант времени CPU.

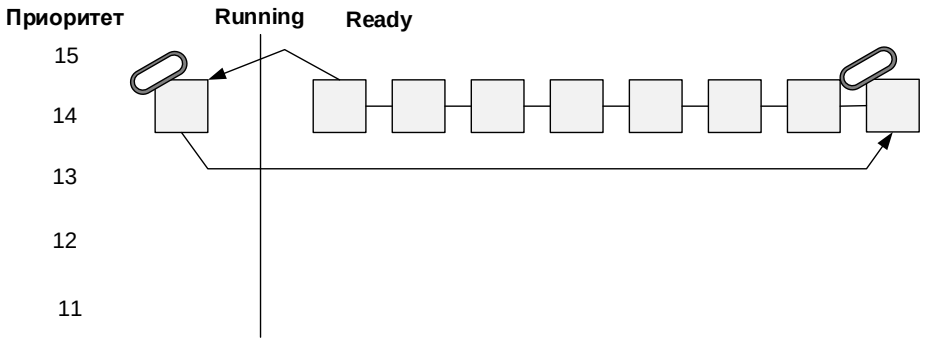

Рис. 18. Планирование потоков в момент завершения кванта текущего потока

Завершение потока

Завершаясь (после возврата из основной процедуры и вызова `ExitThread` или из-за уничтожения вызовом `TerminateThread`), поток переходит в состояние `Terminated`. Если в этот момент ни один описатель его объекта «поток» не открыт, поток удаляется из списка потоков процесса, и соответствующие структуры данных освобождаются.

Переключение контекста

Контекст потока и процедура его переключения зависят от архитектуры процессора. В типичном случае переключение контекста требует сохранения и восстановления следующих данных:

- указателя команд;
- указателей на стек ядра и пользовательский стек;
- указателя на адресное пространство, в котором выполняется поток (каталог таблиц страниц процесса).

Ядро сохраняет эту информацию, заталкивая ее в текущий стек ядра, обновляя указатель стека и сохраняя его в блоке `KTHREAD` потока. Далее указатель стека ядра устанавливается на стек ядра нового потока и загружается контекст этого потока. Если новый поток принадлежит другому процессу, в специальный регистр процессора загружается адрес его каталога таблиц страниц, в результате чего адресное пространство этого процесса становится доступным. При наличии отложенной APC ядра запрашивается прерывание IRQL уровня 1. В ином случае управление передается загруженному для нового потока указателю команд, и выполнение этого потока возобновляется.

Поток простоя

Если нет ни одного потока, готового к выполнению на CPU, Windows подключает к данному CPU поток простоя (процесса Idle). Для каждого CPU создается свой поток простоя.

Разные утилиты для просмотра процессов в Windows по-разному называют процесс Idle. Диспетчер задач и Process Explorer обозначают его как «System Idle Process». Windows сообщает, что приоритет потока простоя равен 0. Но на самом деле у него вообще нет уровня приоритета, поскольку он выполняется лишь в отсутствие других потоков.

Холостой цикл, работающий при IRQL уровня «DPC/dispatch», просто запрашивает задания, например на доставку отложенных DPC или на поиск потоков, подлежащих диспетчеризации. Хотя последовательность работы потока простоя зависит от архитектуры, он все равно выполняет следующие действия.

1. Включает и отключает прерывания (тем самым давая возможность доставить отложенные прерывания).

2. Проверяет, нет ли у CPU незавершенных DPC. Если таковые есть, сбрасывает отложенное программное прерывание и доставляет эти DPC.

3. Проверяет, выбран ли какой-нибудь поток для выполнения на данном CPU, и, если да, организует его диспетчеризацию.

4. Вызывает из HAL процедуру обработки CPU в простое (если нужно выполнить какие-либо функции управления электропитанием).

В Windows Server 2003 поток простоя также проверяет наличие потоков, ожидающих выполнения на других CPU, но об этом пойдет речь в разделе по планированию потоков в многопроцессорных системах.

Динамическое повышение приоритета

Windows может динамически повышать значение текущего приоритета потока в одном из пяти случаев:

- после завершения операций ввода-вывода;
- при окончании ожидания на событии или семафоре исполнительной системы;
- по окончании операции ожидания потоками активного процесса;
- при пробуждении GUI-потоков из-за операций с окнами;
- если поток, готовый к выполнению, задерживается из-за нехватки процессорного времени.

Динамическое повышение приоритета предназначено для оптимизации общей пропускной способности и отзывчивости системы, а также для устранения потенциально «нечестных» сценариев планирования. Однако, как и любой другой алгоритм планирования, динамическое повышение приоритета — не панацея, и от него выигрывают не все приложения.

ПРИМЕЧАНИЕ. Windows никогда не увеличивает приоритет потоков в диапазоне реального времени (16-31). Поэтому планирование таких потоков по отношению к другим всегда предсказуемо. Windows считает: тот, кто использует приоритеты реального времени, знает, что делает.

Многопроцессорные системы

В однопроцессорной системе алгоритм планирования относительно прост: всегда выполняется поток с наивысшим приоритетом, готовый к выполнению. В многопроцессорной системе планирование усложняется, так как Windows пытается подключать поток к наиболее оптимальному для него CPU, учитывая предпочтительный и предыдущий CPU для этого потока, а также конфигурацию многопроцессорной системы. Поэтому, хотя Windows пытается подключать готовые к выполнению потоки с наивысшим приоритетом ко всем доступным CPUs, она гарантирует лишь то, что на одном из процессоров будет работать (единственный) поток с наивысшим приоритетом.

Прежде чем описывать специфические алгоритмы, позволяющие выбирать, какие потоки, когда и на каком процессоре будут выполняться, давайте рассмотрим дополнительную информацию, используемую Windows для отслеживания состояния потоков и CPUs как в обычных многопроцессорных системах, так и в двух новых типах таких систем, поддерживаемых Windows, — в системах с физическими CPUs, **поддерживающими логические** (hyperthreaded systems), и NUMA.

База данных диспетчера ядра в многопроцессорной системе

В такой БД хранится информация, поддерживаемая ядром и необходимая для планирования потоков. В многопроцессорных системах Windows 2000 и Windows XP очереди готовых потоков и сводка готовых потоков имеют ту же структуру, что и в однопроцессорных системах. Кроме того, Windows поддерживает две битовые маски для отслеживания состояния процессоров в системе. Вот что представляют собой эти маски.

- **Маска активных процессоров** (`KeActiveProcessors`), в которой устанавливаются биты для каждого используемого в системе CPU. (Их может быть меньше числа установленных CPUs, если лицензионные ограничения данной версии Windows не позволяют задействовать все физические CPUs.)
- **Сводка простоя** (idle summary) (`KildleSummary`), в которой каждый установленный бит представляет простаивающий CPU.

Если в single-CPU системе диспетчерская база данных блокируется повышением IRQT (в Windows 2000 и Windows XP до уровня «DPC/dispatch», а в Windows Server 2003 до уровней «DPC/dispatch» и «Synch»), то в multi-CPU системе требуется большее, потому что каждый CPU одновременно может повысить IRQL и попытаться манипулировать этой базой данных. В Windows 2000 и Windows XP для синхронизации доступа к информации о диспетчеризации потока применяется две спин-блокировки режима ядра: спин-блокировка диспетчера ядра (dispatcher spinlock) (`KiDispatcherLock`) и спин-блокировка обмена контекста (context swap spinlock) (`KiContextSwapLock`). Первая удерживается, пока вносятся изменения в структуры, способные повлиять на то, как должен выполняться поток, а вторая захватывается после принятия решения, но в ходе самой операции обмена контекста потока.

Для большей масштабируемости и улучшения поддержки параллельной диспетчеризации потоков в multi-CPU системах Windows Server 2003 очереди готовых потоков диспетчера создаются для каждого CPU, как показано на Рис. 19. Благодаря этому в Windows Server 2003 каждый CPU может проверять свои очереди готовых потоков, не блокируя аналогичные общесистемные очереди.

Очереди готовых потоков и сводки готовности, индивидуальные для каждого CPU, являются частью структуры PRCB (processor control block). Поскольку в multi-CPU системе одному из CPUs может понадобиться изменить структуры данных, связанные с планированием, для другого CPU (например, вставить поток, предпочитающий работать на определенном CPU), доступ к этим структурам синхронизируется с применением **новой спин-блокировки с очередями**, индивидуальной для каждой PRCB; она захватывается при IRQL SYNCHLEVEL. Таким образом, поток может быть выбран при блокировке PRCB лишь какого-то одного CPU, в отличие от Windows 2000 и Windows XP, где с этой целью нужно захватить общесистемную спин-блокировку диспетчера ядра.

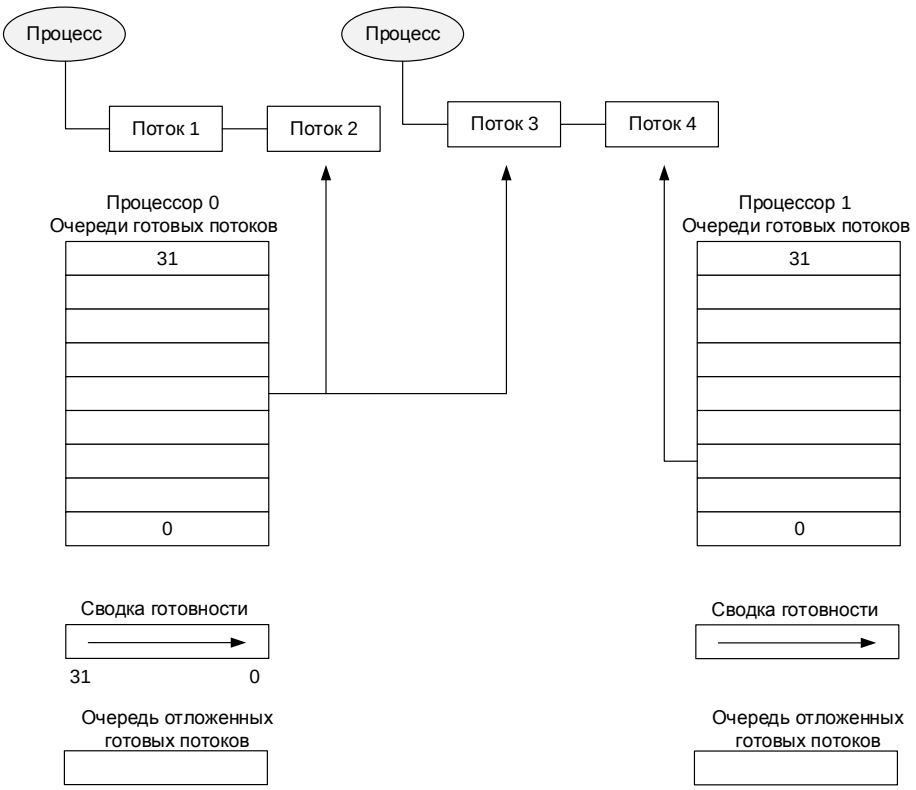

Рис. 19. База данных диспетчера ядра в multi-CPU системе Windows Server 2003

Для каждого CPU создается и список потоков в **готовом, но отложенном состоянии** (deferred ready state). Это потоки, готовые к выполнению, но операция, уведомляющая в результате об их готовности, отложена до более подходящего времени. Поскольку каждый CPU манипулирует только своим списком отложенных готовых потоков, доступ к этому списку не синхронизируется по спин-блокировке PRCB. Список отложенных готовых потоков обрабатывается до выхода из диспетчера потоков, до переключения контекста и после обработки DPC. Потоки в этом списке либо немедленно диспетчеризуются, либо перемещаются в одну из индивидуальных для каждого CPU очередей готовых потоков (в зависимости от приоритета).

Наконец, в Windows Server 2003 улучшена синхронизация переключения контекста потоков, так как теперь оно синхронизируется с применением спин-блокировки, индивидуальной для каждого потока, а в Windows 2000 и Windows XP переключение контекста синхронизировалось захватом общесистемной спин-блокировки обмена контекста.

Системы с поддержкой Hyperthreading

Windows XP и Windows Server 2003 поддерживают multi-CPU системы, использующие технологию **Hyperthreading** (аппаратная реализация логических CPU на одном физическом).

1. Логические CPUs не подпадают под лицензионные ограничения на число физических CPU. Так, Windows XP Home Edition, которая по условиям лицензии может использовать только один CPU, задействует оба логических CPU в single-CPU системе с поддержкой Hyperthreading.

2. Если все логические CPUs какого-либо физического CPU простаивают, для выполнения потока выбирается один из логических CPUs этого физического CPU, а не того, у которого один из логических CPUs уже выполняет другой поток.

Системы NUMA

Другой тип multi-CPU систем, поддерживаемый Windows XP и Windows Server 2003, — архитектуры памяти с неоднородным доступом (nonuniform memory access, NUMA). В **NUMA-системе** CPUs группируются в узлы. В каждом узле имеются свои CPUs и память, и он подключается к системе соединительной шиной с когерентным кэшем (cache-coherent interconnect bus). Доступ к памяти в таких системах называется неоднородным потому, что у каждого узла есть локальная высокоскоростная память. Хотя любой CPU в любом узле может обращаться ко всей памяти, доступ к локальной для узла памяти происходит гораздо быстрее.

Приложения, которым нужно выжать максимум производительности из NUMA-систем, могут устанавливать маски привязки процесса к CPUs в определенном узле.

Привязка к CPU

У каждого потока есть **маска привязки к CPU** (affinity mask), указывающая, на каких CPUs можно выполнять данный поток. Потоки наследуют маску привязки процесса. По умолчанию начальная маска для всех CPU (а значит, и для всех потоков) включает весь набор активных CPUs в системе, т. е. любой поток может выполняться на любом CPU.

Однако для повышения пропускной способности и/или оптимизации рабочих нагрузок на определенный набор CPUs приложения могут изменять маску привязки потока к CPU. Это можно сделать на нескольких уровнях.

- Вызовом функции `SetThreadAffinityMask`, чтобы задать маску привязки к CPUs для индивидуального потока;
- Вызовом функции `SetProcessAffinityMask`, чтобы задать маску привязки к CPUs для всех потоков в процессе. Диспетчер задач и Process Explorer предоставляют GUI-интерфейс к этой функции: щелкните процесс правой кнопкой мыши и выберите Set Affinity (Задать соответствие).
- Включением процесса в задание, в котором действует глобальная для задания маска привязки к CPUs, установленная через функцию `SetlnformationJobObject`
- Определением маски привязки к CPU в заголовке образа с помощью, например, утилиты Imagecfg из Windows 2000 Server Resource Kit Supplement.

Идеальный и последний CPU

В блоке потока ядра каждого потока хранятся номера двух особых CPUs:

- **идеального** (ideal processor) — предпочтительного для выполнения данного потока;
- **последнего** (last processor) — на котором поток работал в прошлый раз.

Идеальный CPU для потока выбирается случайным образом при его создании с использованием **зародышевого значения** (seed) в блоке процесса. Это значение увеличивается на 1 всякий раз, когда создается новый поток, поэтому создаваемые

потоки равномерно распределяются по набору доступных CPUs. Например, первый поток в первом процессе в системе закрепляется за идеальным CPU 0, второй поток того же процесса — за идеальным CPU 1. Однако у следующего процесса в системе идеальный CPU для первого потока устанавливается в 1, для второго в 2 и т. д. Благодаря этому потоки внутри каждого процесса равномерно распределяются между CPUs.

Заметьте: здесь предполагается, что потоки внутри процесса выполняют равные объемы работы. Но в многопоточном процессе это обычно не так; в нем есть, как правило, один или более **«служебных» потоков** (housekeeping threads) и несколько **рабочих**. Поэтому, если в многопоточном приложении нужно задействовать все преимущества multi-CPU платформы, целесообразно указывать номера идеальных CPUs для потоков вызовом функции `SetThreadIdealProcessor`.

В системах с Hyperthreading **следующим идеальным CPU** является первый логический CPU на следующем физическом. Например, в double-CPU системе с Hyperthreading логических CPUs — 4; если для первого потока идеальным CPU назначен логический CPU 0, то для второго потока имело бы смысл назначить таковым логический CPU 2, для третьего — логический CPU 1, для четвертого — логический CPU 3 и т. д. Тогда потоки равномерно распределялись бы по физическим CPUs.

В NUMA-системах идеальный узел для процесса выбирается при его (процесса) создании. Первому процессу назначается узел 0, второму— 1 и т. д. Затем идеальные CPUs для потоков процесса выбираются из идеального узла. Идеальным CPU для первого потока в процессе назначается первый CPU в узле. По мере создания дополнительных потоков в процессе за ними закрепляется тот же идеальный узел; следующий CPU в этом узле становится идеальным для следующего потока и т. д.

Домашнее задание

Самостоятельно рассмотреть:

- Объекты задания (jobs)
- Алгоритмы планирования потоков в multi-CPU системах

Литература

1. Э. Таненбаум. Современные операционные системы. 2-ое изд. –СПб.: Питер, 2002. – 1040 с.

2. Э. Таненбаум, А. Вудхалл. Операционные системы: разработка и реализация. Классика CS. –СПб.: Питер, 2006. –576 с.

3. М. Руссинович, Д. Соломон. Внутреннее устройство Microsoft Windows: Windows Server 2003, Windows XP, Windows 2000. Мастер-класс. / Пер. с англ. -4-е изд. –М.: Издательско-торговый дом «Русская редакция»; СПб.: Питер; 2005. -992 с.

4. Microsoft Development Network. URL: http://msdn.com