Основные понятия. Архитектура ОС Windows

Лекция

Ревизия: 0.1

Последняя модификация: Vlad Kovtun

© NRJETIX 2000 - 2014

Дата последней модификации: 03.04.2015 14:55:00

История изменений

08.09.2010 – Версия 0.1. Первичный документ. Ковтун В.Ю.

03.04.2014 – Версия 0.2. Общая редакция. Ковтун В.Ю.

Содержание

История изменений 2

Содержание 3

Лекция 10. Основные понятия. Архитектура ОС Windows 4

- Вопросы 4
- Основные понятия 4
  - Процессы, потоки и задания 4
  - Виртуальная память 5
  - Режим ядра и пользовательский режим 7
  - Terminal Services и несколько сеансов 8
  - Объекты и описатели 8
  - Безопасность 9
  - Реестр 9
  - Unicode 10
- Изучение внутреннего устройства Windows 10
  - Модель операционной системы 10
  - Основана ли Windows на микроядре? 10
  - Обзор архитектуры 11
  - Переносимость 12
  - Симметричная многопроцессорная обработка 13
  - Масштабируемость 14
  - Различия между клиентскими и серверными версиями 15
  - Проверочный выпуск 15
  - Ключевые компоненты системы 15
  - Ntdll.dll 19
  - Ядро 21
  - Уровень абстрагирования от оборудования 22
  - Драйверы устройств 22
  - Системные процессы 24

Литература 24

Лекция 10. Основные понятия. Архитектура ОС Windows

Вопросы

1. Основные понятия.

2. Архитектура.

Основные понятия

Процессы, потоки и задания

Программа представляет собой статический набор команд, а **процесс** — это контейнер для набора ресурсов, используемых при выполнении экземпляра программы. На самом высоком уровне абстракции процесс в Windows включает следующее:

- **закрытое виртуальное адресное пространство** — диапазон адресов виртуальной памяти, которым может пользоваться процесс;
- **исполняемую программу** — исполняемый код и данные, проецируемые на виртуальное адресное пространство процесса;
- **список открытых описателей (handles) различных системных ресурсов** — семафоров, коммуникационных портов, файлов и других объектов, доступных всем потокам в данном процессе;
- **контекст защиты** (security context), называемый **маркером доступа** (access token) и идентифицирующий пользователя, группы безопасности и привилегии, сопоставленные с процессом;
- **уникальный идентификатор процесса** (во внутрисистемной терминологии называемый **идентификатором клиента**);
- **минимум один поток**.

Каждый процесс также указывает на свой родительский процесс (процесс-создатель). Однако, если родитель существует, эта информация не обновляется. Поэтому есть вероятность, что некий процесс указывает на уже несуществующего родителя. Это не создает никакой проблемы, поскольку никто не полагается на наличие такой информации. Следующий эксперимент иллюстрирует данный случай.

Для просмотра (и модификации) процессов и информации, связанной с ними, существует целый набор утилит, которые включаются:

- в саму Windows;
- в Windows Support Tools;
- Windows Debugging Tools;
- ресурсы Windows;
- Platform SDK;
- ряд утилит можно получить с сайта http://www.sysinternals.com.

Многие из этих утилит выводят перекрывающиеся подмножества информации о базовых процессах и потоках, иногда идентифицируемые по разным именам.

Поток (thread) — некая сущность внутри процесса, получающая время CPU для выполнения. Без потока программа процесса не может выполняться. Поток включает следующие наиболее важные элементы:

- содержимое набора регистров CPU, отражающих состояние CPU;
- два стека, один из которых используется потоком при выполнении в режиме ядра, а другой — в пользовательском режиме;
- закрытую область памяти, называемую локальной памятью потока (thread-local storage, TLS) и используемую подсистемами, библиотеками исполняющих систем (run-time libraries) и DLL;
- уникальный идентификатор потока (во внутрисистемной терминологии также называемый идентификатором клиента: идентификаторы процессов и потоков генерируются из одного пространства имен и никогда не перекрываются);

- иногда потоки обладают своим контекстом защиты, который обычно используется многопоточными серверными приложениями, подменяющими контекст защиты обслуживаемых клиентов.

Переменные регистры, стеки и локальные области памяти называются **контекстом потока**. Поскольку эта информация различна на каждой аппаратной платформе, на которой может работать Windows, соответствующая структура данных специфична для конкретной платформы. Windows-функция `GetThreadContext` предоставляет доступ к этой аппаратно-зависимой информации (называемой блоком `CONTEXT`).

Хотя у потоков свой контекст выполнения, каждый поток внутри одного процесса делит его виртуальное адресное пространство (а также остальные ресурсы, принадлежащие процессу). Это означает, что все потоки в процессе могут записывать и считывать содержимое памяти любого из потоков данного процесса. Однако потоки не могут случайно сослаться на адресное пространство другого процесса. Исключение возможно в ситуации, когда тот предоставляет часть своего адресного пространства, как **раздел общей памяти** (shared memory section), в Windows API называемый **объектом «проекция файла»** (file mapping object), или когда один из процессов имеет право на открытие другого процесса и использует функции доступа к памяти между процессами, например `ReadProcessMemory` и `WriteProcessMemory`.

Кроме закрытого адресного пространства и одного или нескольких потоков у каждого процесса имеются идентификация защиты и список открытых описателей таких объектов, как файлы и разделы общей памяти, или синхронизирующих объектов вроде мьютексов, событий и семафоров (Рис. 1).

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 (в том числе выполняемого на удаленной системе Windows).

Дескрипторы виртуальных адресов (virtual address descriptors, VAD) — это структуры данных, используемые диспетчером памяти для учета виртуальных адресов, задействованных процессом.

Ошибка! Объект не может быть создан из кодов полей редактирования.

Рис. 1. Процесс и его ресурсы

Windows предоставляет расширение для модели процессов — **задания (jobs)**. Они 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. В этом объекте также хранится информация обо всех процессах, которые были сопоставлены с заданием, но к настоящему времени уже завершены. В каких-то отношениях объект-задание компенсирует отсутствие иерархического дерева процессов в Windows, а в каких-то — даже превосходит по своим возможностям дерево процессов UNIX.

Виртуальная память

В Windows реализована система виртуальной памяти, основанная на **плоском (линейном) адресном пространстве**. Она создает каждому процессу иллюзию того, что у него есть собственное большое и закрытое адресное пространство. Виртуальная память дает логическое представление, не обязательно соответствующее структуре физической памяти. В период выполнения диспетчер памяти, используя аппаратную поддержку, транслирует, или **проецирует (maps)**, виртуальные адреса на физические, по которым реально хранятся данные. Управляя проецированием и защитой страниц памяти, ОС гарантирует, что ни один процесс не помешает другому и не сможет повредить данные самой ОС. На Рис. 2 показано, как три смежные страницы виртуальной памяти проецируются на три разрозненные страницы физической памяти.

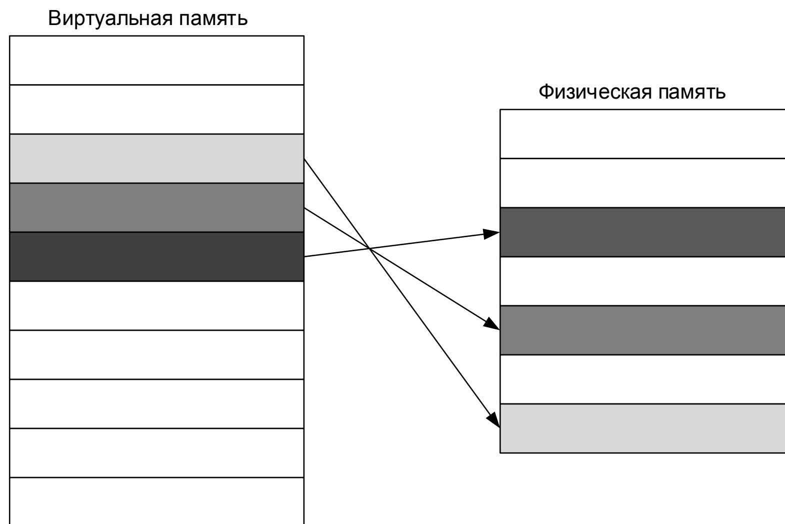

Рис. 2. Проецирование виртуальной памяти в физическую

Поскольку у большинства компьютеров объем физической памяти намного меньше общего объема виртуальной памяти, задействованной выполняемыми процессами, диспетчер памяти **перемещает**, или **подкачивает** страницы (pages), часть содержимого памяти на диск. Подкачка данных на диск освобождает физическую память для других процессов или самой ОС. Когда поток обращается к странице виртуальной памяти, сброшенной на диск, диспетчер виртуальной памяти загружает эту информацию с диска обратно в память. Для использования преимуществ подкачки в приложениях никакого дополнительного кода не требуется, так как диспетчер памяти опирается на аппаратную поддержку этого механизма.

Размер виртуального адресного пространства зависит от конкретной аппаратной платформы. На 32-разрядных x86-системах теоретический максимум для общего виртуального адресного пространства составляет 4 Гб. По умолчанию Windows выделяет нижнюю половину этого пространства (в диапазоне адресов от `0x00000000` до `0x7FFFFFFF`) процессам, а вторую половину (в диапазоне адресов от `0x80000000` до `0xFFFFFFFF`) использует в собственных целях. Windows 2000 Advanced Server, Windows 2000 Datacenter Server, Windows XP (SP2 и выше) и Windows Server 2003 поддерживают загрузочные параметры /3GB и /USERVA, которые указываются в файле `Boot.ini`, что позволяет процессам, выполняющим программы со специальным флагом в заголовке исполняемого образа, использовать до 3 Гб закрытого адресного пространства и оставляет ОС только 1 Гб. Этот вариант дает возможность приложению вроде сервера базы данных хранить в адресном пространстве своего процесса большие порции базы данных и тем самым уменьшить частоту проецирования отдельных представлений этой базы. Две структуры виртуальных адресных пространств, поддерживаемые 32-разрядной Windows, показаны на Рис. 3.

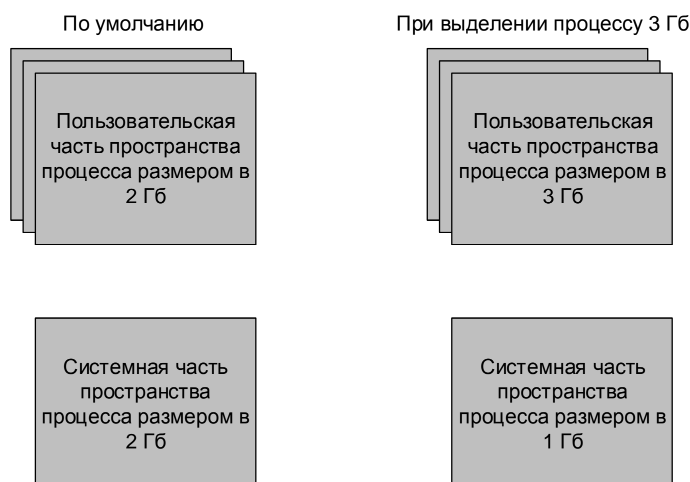

Рис. 3. Структуры адресных пространств для 32-разрядных Windows

Хотя 3 Gb лучше двух, этого все равно недостаточно для проецирования очень больших баз данных. В связи с этим в 32-разрядных Windows появился механизм Address Windowing Extension (AWE), который позволяет 32-разрядному приложению выделять до 64 Гб физической памяти, а затем проецировать **представления** (views), или **окна** (windows), на свое 2 GB виртуальное адресное пространство. Применение AWE усложняет управление проекциями виртуальной памяти на физическую, но снимает проблему прямого доступа к объему физической памяти, превышающему лимиты 32-разрядного адресного пространства процесса.

64-разрядная Windows предоставляет процессам гораздо большее адресное пространство: 7152 Gb на Itanium-системах и 8192 Gb на x64-системах. На Рис. 4 показана упрощенная схема структур 64-разрядных адресных пространств. Заметьте, что эти размеры отражают не архитектурные лимиты для данных платформ, а ограничения реализации в текущих версиях 64-разрядной Windows.

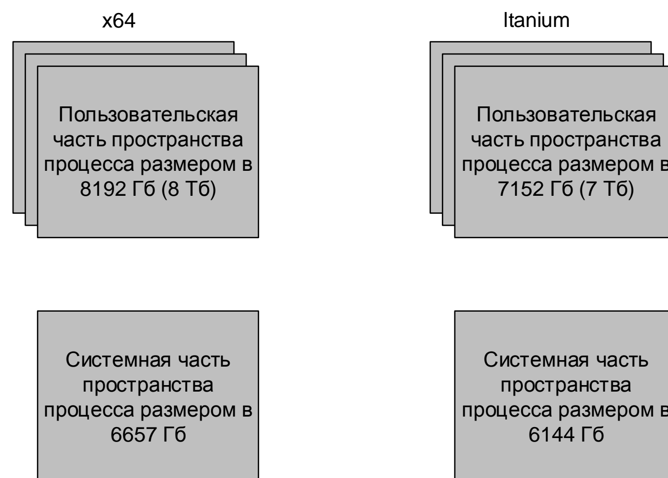

Рис. 4. Структура адресных пространств для 64-разрядных Windows

Режим ядра и пользовательский режим

Для предотвращения доступа приложений к критически важным данным ОС и устранения риска их модификации Windows использует два режима доступа к CPU (даже если он поддерживает более двух режимов):

- пользовательский (user mode),
- ядра (kernel mode).

Код приложений работает в **пользовательском режиме**, тогда как код ОС (например, системные сервисы и драйверы устройств) — в **режиме ядра**. В режиме ядра предоставляется доступ ко всей системной памяти и разрешается выполнять любые машинные команды CPU. Предоставляя ОС более высокий уровень привилегий, чем прикладным программам, CPU позволяет разработчикам ОС реализовать такие архитектуры, которые не дали возможности сбойным приложениям нарушать стабильность работы всей системы.

Хотя каждый Windows-процесс имеет свою (закрытую) память, код ОС и драйверы устройств, работающие в режиме ядра, делят единое виртуальное адресное пространство. Каждая страница в виртуальной памяти помечается тэгом, определяющим, в каком режиме должен работать CPU для чтения и/или записи данной страницы. Страницы в системном пространстве доступны лишь в режиме ядра, а все страницы в пользовательском адресном пространстве — в пользовательском режиме. Страницы только для чтения (например, содержащие лишь исполняемый код) ни в каком режиме для записи недоступны.

Windows не предусматривает никакой защиты системной памяти от компонентов, работающих в режиме ядра. Иначе говоря, код ОС и драйверов устройств в режиме ядра получает полный доступ к системной памяти и может обходить средства защиты Windows для обращения к любым объектам. Поскольку основная часть кода Windows выполняется в режиме ядра, крайне важно, чтобы компоненты, работающие в этом режиме, были тщательно продуманы и протестированы.

Это также подчеркивает, насколько надо быть осторожным при загрузке драйвера устройства от стороннего поставщика: перейдя в режим ядра, он получит полный доступ ко всем данным ОС. Такая уязвимость стала одной из причин, по которым в Windows введен механизм проверки цифровых подписей драйверов, предупреждающий пользователя о попытке установки неавторизованного (неподписанного) драйвера. Кроме того, **механизм Driver Verifier** (верификатор драйверов) помогает разработчикам драйверов устройств находить в них ошибки (вызывающие, например, утечку памяти или переполнения буферов).

Прикладные программы могут переключаться из пользовательского режима в режим ядра, обращаясь к системному сервису. Переключение из пользовательского режима в режим ядра осуществляется специальной командой CPU. ОС перехватывает эту команду, обнаруживает запрос системного сервиса, проверяет аргументы, которые поток передал системной функции, и выполняет внутреннюю подпрограмму. Перед возвратом управления пользовательскому потоку CPU переключается обратно в пользовательский режим. Благодаря этому ОС защищает себя и свои данные от возможной модификации пользовательскими процессами.

Terminal Services и несколько сеансов

Terminal Services (службы терминала) обеспечивают в Windows поддержку нескольких интерактивных сеансов пользователей на одной системе. С помощью Terminal Services удаленный пользователь может установить сеанс на другой машине, зарегистрироваться на ней и запускать приложения на сервере. Сервер предоставляет клиенту GUI, а клиент возвращает серверу пользовательский ввод.

Первый сеанс входа на физической консоли компьютера считается **консольным сеансом, или нулевым сеансом (session zero)**. Дополнительные сеансы можно создать с помощью программы соединения с удаленным рабочим столом (Msfsc.exe), в Windows XP — через механизм быстрого переключения пользователей.

Возможность создания удаленного сеанса поддерживается Windows 2000 Server, но не Windows 2000 Professional. Windows XP Professional позволяет одному удаленному пользователю подключаться к машине, однако если кто-то начинает процедуру входа в консоли, рабочая станция блокируется.

Windows 2000 Server и Windows Server 2003 поддерживают два одновременных удаленных сеанса. Windows 2000 Advanced Server, Datacenter Server и все издания Windows Server 2003 способны поддерживать более двух сеансов одновременно при условии правильного лицензирования и настройки системы в качестве сервера терминала.

Хотя Windows XP Home и Professional не поддерживают несколько удаленных подключений к рабочему столу, они все же поддерживают несколько сеансов, созданных локально через механизм быстрого переключения пользователей.

Для приложений, которым нужно знать, выполняются ли они в сеансе сервера терминала, предназначен набор Windows API-функций, позволяющих программно распознавать такую ситуацию и контролировать различные аспекты служб терминала.

Объекты и описатели

В ОС Windows **объект** — это единственный экземпляр периода выполнения (run-time instance) статически определенного типа объекта. Тип объекта состоит из общесистемного типа данных, функций, оперирующих экземплярами этого типа данных, и набора атрибутов.

Атрибут объекта (object attribute) — это поле данных в объекте, частично определяющее состояние этого объекта. Например, объект типа «процесс», имеет атрибуты, в число которых входят идентификатор процесса, базовый приоритет и указатель на объект маркера доступа. Методы объекта (средства для манипулирования объектами) обычно считывают или изменяют какие-либо атрибуты.

Самое главное различие между объектом и обычной структурой данных заключается в том, что внутренняя структура объекта скрыта. Чтобы получить данные из объекта или записать в него какую-то информацию, следует вызвать его сервис. Прямое чтение или изменение данных внутри объекта невозможно. Тем самым реализация объекта отделяется от кода, который просто использует его, а это позволяет менять реализацию объекта, не модифицируя остальной код.

Объекты очень удобны для поддержки четырех важных функций ОС:

- присвоения понятных имен системным ресурсам;
- разделения ресурсов и данных между процессами;
- защиты ресурсов от несанкционированного доступа;
- учета ссылок (благодаря этому система узнает, когда объект больше не используется, и автоматически уничтожает его).

Не все структуры данных в Windows являются объектами. В объекты помещаются лишь те данные, которые нужно разделять, защищать, именовать или делать доступными программам пользовательского режима (через системные сервисы).

Безопасность

Windows с самого начала разрабатывалась как защищенная система, удовлетворяющая требованиям различных правительственных и промышленных стандартов безопасности, например спецификации **Common Criteria for Information Technology Security Evaluation (CCITSE)**. Подтверждение правительством рейтинга безопасности ОС позволяет ей конкурировать в сферах, требующих повышенной защиты. Разумеется, многим из этих требований должна удовлетворять любая многопользовательская система.

Базовые возможности защиты в Windows таковы:

- избирательная защита любых разделяемых системных объектов (файлов, каталогов, процессов, потоков и т.д.),
- аудит безопасности (для учета пользователей и инициируемых ими операций),
- аутентификация паролей при входе и предотвращение доступа одного из пользователей к неинициализированным ресурсам, освобожденным другим пользователем.

Windows поддерживает два вида контроля доступа к объектам.

Управление избирательным доступом (discretionary access control) — является механизмом, который как раз и связывается большинством пользователей с защитой. Это метод, при котором владельцы объектов (например, файлов или принтеров) разрешают или запрещают доступ к ним для других пользователей. При входе пользователь получает набор **удостоверений защиты (security credentials)**, или **контекст защиты (security context)**. Когда он пытается обратиться к объекту, его контекст защиты сверяется со **списком управления доступом (access control list, ACL)** для данного объекта, чтобы определить, имеет ли он разрешение на выполнение запрошенной операции.

Управление привилегированным доступом (privileged access control) — необходим в тех случаях, когда управления избирательным доступом недостаточно. Данный метод гарантирует, что пользователь сможет обратиться к защищенным объектам, даже если их владелец недоступен. Например, если какой-то сотрудник увольняется из компании, администратору нужно получить доступ к файлам, которые могли быть доступны только бывшему сотруднику. В таких случаях Windows позволяет администратору стать владельцем этих файлов и при необходимости управлять правами доступа к ним.

Защита пронизывает весь интерфейс Windows API. Подсистема Windows реализует защиту на основе объектов точно так же, как и сама ОС. При первой попытке доступа приложения к общему (разделяемому) объекту, подсистема Windows проверяет, имеет ли это приложение соответствующие права. Если проверка завершается успешно, подсистема Windows разрешает приложению доступ.

Подсистема Windows реализует защиту для общих объектов, часть из которых построена на основе родных объектов Windows. К Windows-объектам относятся объекты рабочего стола, меню, окна, файлы, процессы, потоки и ряд синхронизирующих объектов.

Реестр

Реестр - системная база данных с информацией, необходимой для загрузки и конфигурирования системы; в ней содержатся общесистемные параметры, контролирующие работу Windows, база данных защиты и конфигурационные настройки, индивидуальные для каждого пользователя.

Кроме того, реестр — это окно, через которое можно заглянуть в переменные системные данные, чтобы, например, выяснить текущее состояние аппаратной части системы (какие драйверы устройств загружены, какие ресурсы они используют и т.д.) или значения счетчиков производительности Windows. Счетчики производительности, которые на самом деле в реестре не хранятся, доступны через функции реестра.

Хотя у многих пользователей и администраторов Windows никогда не возникает необходимости работать непосредственно с реестром, он все же является источником полезной информации о внутренних структурах данных Windows, так как содержит множество параметров, влияющих на быстродействие и поведение системы.

Unicode

Windows отличается от большинства других ОС тем, что в качестве внутреннего формата для хранения и обработки текстовых строк использует Unicode. **Unicode** — это стандартная кодировка, которая поддерживает многие известные в мире наборы символов и в которой каждый символ представляется 16-битным (двухбайтовым) кодом. (Подробнее о Unicode см. www.unicode.org и MSDN Library).

Поскольку многие приложения имеют дело с 8-битными (однобайтовыми) ANSI-символами, Windows-функции, принимающие строковые параметры, существуют в двух версиях: для Unicode и для ANSI. В Windows 95, Windows 98 и Windows ME реализована лишь часть Unicode - версий Windows-функций, поэтому приложения, рассчитанные на выполнение как в одной из этих операционных систем, так и в NT-подобных Windows, обычно используют ANSI-версии функций. Если вы вызываете ANSI-версию Windows-функции, входные строковые параметры перед обработкой системой преобразуются в Unicode, а выходные — из Unicode в ANSI (перед возвратом приложению). Таким образом, при использовании в Windows устаревшего сервиса или фрагмента кода, написанного в расчете на ANSI-строки, эта ОС будет вынуждена преобразовывать ANSI-символы в Unicode. Однако Windows никогда не преобразует данные внутри файлов — решения о том, в какой кодировке хранить текстовую информацию в файлах, принимают лишь сами приложения.

Начиная с Windows 2000, все языковые выпуски содержат одинаковые Windows-функции. Единая для всех стран двоичная кодовая база Windows способна поддерживать множество языков за счет простого добавления нужных компонентов языковой поддержки. Используя эти Windows-функции, разработчики могут создавать универсальные приложения, способные работать со множеством языков.

Изучение внутреннего устройства Windows

Модель операционной системы

В большинстве многопользовательских ОС приложения отделены от собственно ОС: код ее ядра выполняется в **привилегированном режиме CPU** (называемом **режимом ядра**), который обеспечивает доступ к системным данным и оборудованию. Код приложений выполняется в **непривилегированном режиме CPU** (называемом **пользовательским**) с неполным набором интерфейсов, ограниченным доступом к системным данным и без прямого доступа к оборудованию. Когда программа пользовательского режима вызывает системный сервис, CPU перехватывает вызов и переключает вызывающий поток в режим ядра. По окончании работы системного сервиса ОС переключает контекст потока обратно в пользовательский режим и продолжает его выполнение.

Windows, как и большинство UNIX-систем, является **монолитной ОС** — в том смысле, что большая часть ее кода и драйверов использует одно и то же пространство защищенной памяти режима ядра. Это значит, что любой компонент ОС или драйвер устройства потенциально способен повредить данные, используемые другими компонентами ОС.

Основана ли Windows на микроядре?

Хотя некоторые объявляют ее таковой, Windows не является ОС на основе микроядра в классическом понимании этого термина. В подобных системах основные компоненты ОС (диспетчеры памяти, процессов, ввода-вывода) выполняются как отдельные процессы в собственных адресных пространствах и представляют собой надстройки над примитивными сервисами микроядра. Пример современной системы с архитектурой на основе микроядра - ОС Mach, разработанная в Carnegie Mellon University. Она реализует

крошечное ядро, которое включает сервисы планирования потоков, передачи сообщений, виртуальной памяти и драйверов устройств. Все остальное, в том числе разнообразные API, файловые системы и поддержка сетей, работает в пользовательском режиме. Однако в коммерческих реализациях на основе микроядра Mach код файловой системы, поддержки сетей и управления памятью выполняется в режиме ядра. Причина проста: системы, построенные строго по принципу микроядра, непрактичны с коммерческой точки зрения из-за слишком низкой эффективности.

Означает ли тот факт, что большая часть Windows работает в режиме ядра, ее меньшую надежность в сравнении с ОС на основе микроядра? Вовсе нет. Рассмотрим следующий сценарий. Допустим, в коде ФС имеется ошибка, которая время от времени приводит к краху системы. Ошибка в коде режима ядра (например, в диспетчере памяти или ФС) скорее всего вызовет полный крах традиционной ОС. В истинной ОС на основе микроядра подобные компоненты выполняются в пользовательском режиме, поэтому теоретически ошибка приведет лишь к завершению процесса соответствующего компонента. Но на практике такая ошибка все равно вызовет крах системы, так как восстановление после сбоя столь критически важного процесса невозможно.

ПРИМЕЧАНИЕ. Все эти компоненты ОС, конечно, полностью защищены от сбойных приложений, поскольку такие программы не имеют прямого доступа к коду и данным привилегированной части ОС (хотя и способны вызывать сервисы ядра). Эта защита — одна из причин, по которым Windows заслужила репутацию отказоустойчивой и стабильной ОС в качестве сервера приложений и платформы рабочих станций, обеспечивающей быстродействие основных системных сервисов вроде поддержки виртуальной памяти, файлового ввода-вывода, работы с сетями и доступа к общим файлам и принтерам.

Компоненты Windows режима ядра также построены на принципах объектно-ориентированного программирования (ООП). Так, для получения информации о каком-либо компоненте они, как правило, не обращаются к его структурам данных. Вместо этого для передачи параметров, доступа к структурам данных и их изменения используются формальные интерфейсы.

Обзор архитектуры

Теперь обратимся к ключевым компонентам системы, составляющим ее архитектуру. Упрощенная версия этой архитектуры показана на Рис. 5 (упрощенная схема не отражает всех деталей архитектуры).

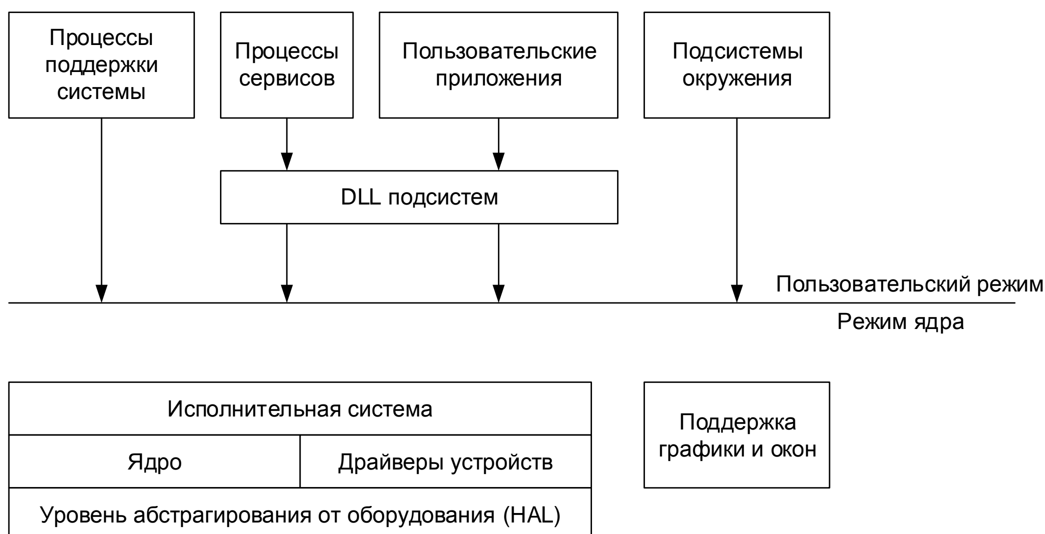

Рис. 5. Упрощенная архитектура Windows

Прямоугольники над разделительной линией соответствуют процессам пользовательского режима, а компоненты под ней — сервисам режима ядра. Известно, что потоки пользовательского режима выполняются в защищенных адресных пространствах процессов (хотя при выполнении в режиме ядра они получают доступ к системному пространству). Таким образом, процессы поддержки системы, сервисов, приложений и подсистем окружения имеют свое адресное пространство.

Существует четыре типа пользовательских процессов:

- **фиксированные процессы поддержки системы** (system support processes) — например, процесс обработки входа в систему и диспетчер сеансов, не являющиеся сервисами Windows (т. е. не запускаемые диспетчером управления сервисами);
- **процессы сервисов** (service processes) — носители Windows-сервисов вроде Task Scheduler и Spooler. Многие серверные приложения Windows, например Microsoft SQL Server и Microsoft Exchange Server, тоже включают компоненты, выполняемые как сервисы;
- **пользовательские приложения** (user applications) — бывают шести типов:
  - 32-разрядной Windows,
  - 64-разрядной Windows,
  - 16-разрядной Windows 3.1,
  - 16-разрядной MS-DOS,
  - 32-разрядной POSIX,
  - 32-разрядной OS/2;
- **подсистемы окружения** (environment subsystems) — реализованы как часть поддержки среды ОС, предоставляемой пользователям и программистам. Изначально Windows NT поставлялась с тремя подсистемами окружения: Windows, POSIX и OS/2. Последняя была изъята в Windows 2000. Что касается Windows XP, то в ней исходно поставляется только подсистема Windows, улучшенная подсистема POSIX доступна как часть бесплатного продукта Services for UNIX.

В Windows пользовательские приложения не могут вызывать родные сервисы ОС напрямую, вместо этого они работают с одной или несколькими DLL подсистем. Их назначение заключается в трансляции документированных функций в соответствующие внутренние (и обычно недокументированные) вызовы системных сервисов Windows.

Windows включает следующие компоненты режима ядра:

- **Исполнительная система** (executive) Windows, содержащая базовые сервисы ОС, которые обеспечивают управление памятью, процессами и потоками, защиту, ввод-вывод и взаимодействие между процессами.
- **Ядро** (kernel) Windows, содержащее низкоуровневые функции ОС, которые поддерживают, например, планирование потоков, диспетчеризацию прерываний и исключений, а также синхронизацию при использовании нескольких CPU. Оно также предоставляет набор процедур и базовых объектов, применяемых исполнительной системой для реализации структур более высокого уровня.
- **Драйверы устройств** (device drivers), в состав которых входят драйверы аппаратных устройств, транслирующие пользовательские вызовы функции ввода-вывода в запросы, специфичные для конкретного устройства, а также сетевые драйверы и драйверы файловых систем.
- **Уровень абстрагирования от оборудования** (hardware abstraction layer, HAL), изолирующий ядро, драйверы и исполнительную систему Windows от специфики оборудования на данной аппаратной платформе (например, от различий между материнскими платами).
- **Подсистема поддержки окон и графики** (windowing and graphics system), реализующая функции графического пользовательского интерфейса (GUI), более известные как Windows-функции модулей USER и GDI. Эти функции обеспечивают поддержку окон, элементов управления пользовательского интерфейса и отрисовку графики.

Переносимость

Windows рассчитана на разные аппаратные платформы, включая как CISC-системы Intel, так и RISC-системы.

Переносимость Windows между системами с различной аппаратной архитектурой и платформами достигается главным образом двумя способами.

- **Windows имеет многоуровневую структуру**. Специфичные для архитектуры CPU или платформы низкоуровневые части системы вынесены в отдельные

модули. Благодаря этому высокоуровневая часть системы не зависит от специфики архитектур и аппаратных платформ. Ключевые компоненты, обеспечивающие переносимость ОС, — ядро (содержится в файле Ntoskrnl.exe) и уровень абстрагирования от оборудования (HAL) (содержится в файле Hal.dll). Функции, специфичные для конкретной архитектуры (переключение контекста потоков, диспетчеризация ловушек и др.), реализованы в ядре. Функции, которые могут отличаться на компьютерах с одинаковой архитектурой (например, в системах с разными материнскими платами), реализованы в HAL. Еще один компонент, содержащий большую долю кода, специфичного для конкретной архитектуры — диспетчер памяти (memory manager), но если рассматривать систему в целом, такого кода все равно немного.

- **Подавляющее большинство компонентов Windows написано на C и лишь часть из них — на C++**. Язык ассемблера применяли только при создании частей системы, напрямую взаимодействующих с системным оборудованием (например, при написании обработчика ловушек прерываний) или требующих исключительного быстродействия (скажем, при переключении контекста). Ассемблерный код имеется не только в ядре и HAL, но и в составе некоторых других частей ОС: процедур, реализующих взаимоблокировку, механизма вызова локальных процедур (LPC), части подсистемы Windows, выполняемой в режиме ядра, и даже в некоторых библиотеках пользовательского режима (например, в коде запуска процессов в Ntdll.dll — системной библиотеке).

Симметричная многопроцессорная обработка

Многозадачность (multitasking) — механизм ОС, позволяющий использовать один CPU для выполнения нескольких потоков. Однако истинно одновременное выполнение, например, двух потоков возможно, только если на компьютере установлено два CPU. При многозадачности система лишь создает видимость одновременного выполнения множества потоков, тогда как multi-CPU система действительно выполняет сразу несколько потоков — по одному на каждом CPU.

Windows является ОС, поддерживающей **симметричную многопроцессорную обработку** (symmetric multiprocessing, SMP). В этой модели нет главного CPU; ОС, как и пользовательские потоки, может выполняться на любом CPU. Кроме того, все CPU используют одну и туже память. При **асимметричной многопроцессорной обработке** (asymmetric multiprocessing, ASMP) система, напротив, выбирает один из CPU для выполнения кода ядра ОС, а другие CPU выполняют только пользовательский код. Различия между этими двумя моделями показаны на Рис. 6.

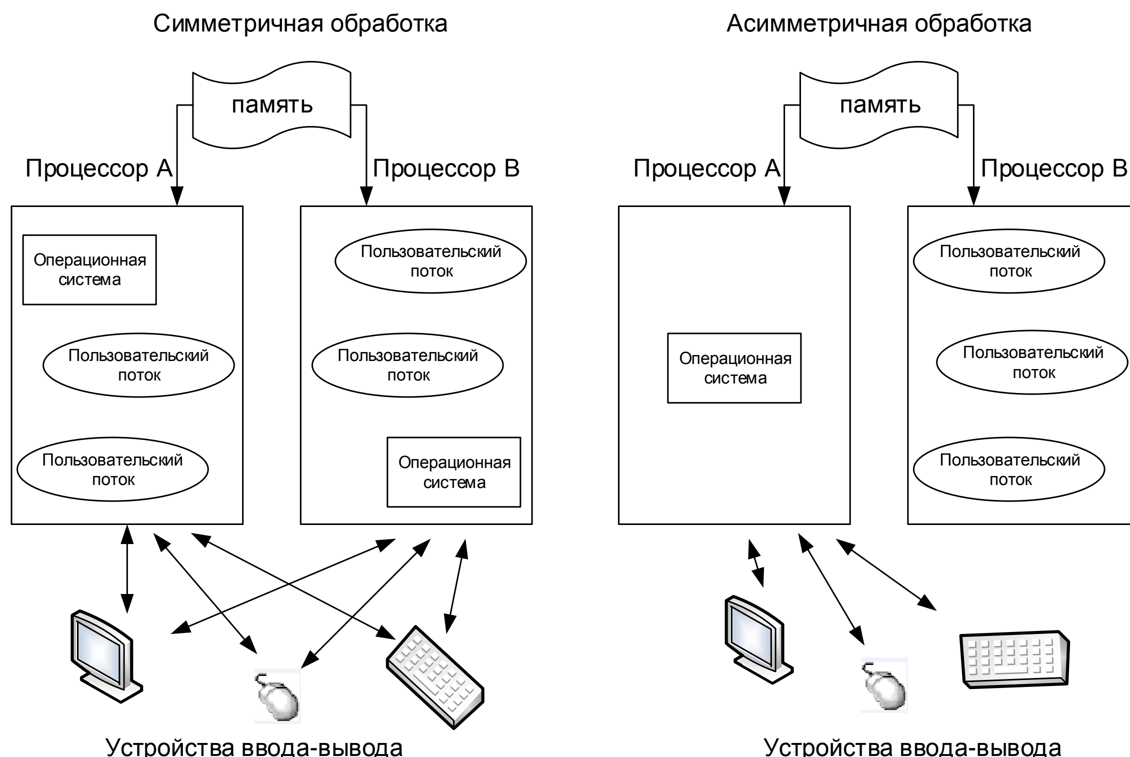

Рис. 6. Симметричная и ассиметричная процессорная обработка

Windows XP и Windows Server 2003 поддерживают два новых типа multi-CPU систем: **логические процессоры** (hyperthreading) и **NUMA** (Non-Uniform Memory Architecture).

Логические процессоры — это технологии, созданная Intel; благодаря ей на одном физическом CPU может быть несколько логических. Каждый логический CPU имеет свое состояние, но **исполняющее ядро** (execution engine) и **набортный кэш** (onboard cache) являются общими. Это позволяет одному из логических CPU продолжать работу, пока другой логический CPU занят (например, обработкой прерывания, которая не дает потокам выполняться на этом логическом CPU). Алгоритмы планирования в Windows XP были оптимизированы под компьютеры с такими CPU.

В **NUMA-системах** CPU группируются в блоки, называемые узлами (nodes). В каждом узле имеются свои CPU и память, и он соединяется с остальными узлами специальной шиной. Windows в NUMA-системе по-прежнему работает как SMP-система, в которой все CPU имеют доступ ко всей памяти, — просто доступ к памяти, локальной для узла, осуществляется быстрее, чем к памяти в других узлах. Система стремится повысить производительность, выделяя потокам время на CPU, которые находятся в том же узле, что и используемая память. Она также пытается выделять память в пределах узла, но при необходимости выделяет память и из других узлов.

Хотя Windows изначально разрабатывалась для поддержки до 32 CPU, многопроцессорной модели не свойственны никакие внутренние особенности, которые ограничивали бы число используемых CPU до 32. Просто это число легко представить битовой маской с помощью машинного 32-разрядного типа данных. И действительно, 64 разрядные версии Windows поддерживают до 64 CPU, потому что размер слова на 64-разрядных CPU равен 64 битам.

Реальное число поддерживаемых CPU зависит от конкретного выпуска Windows. Это число хранится в параметре реестра `HKLM\SYSTEM\CurrentControlSet\Control\Session\Manager\Licensed-Processors`.

Для большей производительности ядро и HAL, имеют одно- и multi-CPU версии. В случае Windows 2000 это относится к шести ключевым системным файлам, а в 32-разрядных Windows XP и Windows Server 2003 — только к трем. В 64-разрядных системах Windows ядра PAE нет, поэтому single- и multi-CPU системы отличаются лишь ядром и HAL.

Соответствующие файлы выбираются и копируются в локальный каталог `\Windows\System32` на этапе установки. Чтобы определить, какие файлы были скопированы, см. файл `\Windows\Repair\Setup.log`, где перечисляются все файлы, копировавшиеся на локальный системный диск, и каталоги на дистрибутивном носителе, откуда они были взяты.

Масштабируемость

Масштабируемость (scalability) — одни из ключевых целей multi-CPU систем. Для корректного выполнения в SMP-системах ОС должна строго соответствовать определенным требованиям. Решить проблемы конкуренции за ресурсы и другие вопросы в multi-CPU системах сложнее, чем в single-CPU, и это нужно учитывать при разработке системы. Некоторые особенности Windows оказались решающими для ее успеха как multi-CPU ОС:

- способность выполнять код ОС на любом доступном CPU и на нескольких CPU одновременно;
- несколько потоков одного процесса можно параллельно выполнять на разных CPU;
- тонкая синхронизация внутри ядра (спин-блокировки, спин-блокировки с очередями и др.), драйверов устройств и серверных процессов позволяет выполнять больше компонентов на нескольких CPU одновременно;
- механизмы вроде портов завершения ввода-вывода, облегчающие эффективную реализацию многопоточных серверных процессов, хорошо масштабируемых в multi-CPU системах. Масштабируемость ядра Windows со временем улучшалась. Например, в Windows Server 2003 имеются очереди планирования, индивидуальные для каждого CPU, что дает возможность планировать потоки параллельно на нескольких машинах.

Различия между клиентскими и серверными версиями

Windows поставляется в клиентских и серверных версиях. В Windows 2000 клиентская версия называется Windows 2000 Professional. Существует также три серверных версии Windows 2000: Windows 2000 Server, Advanced Server и Datacenter Server.

У Windows XP шесть клиентских версий: Windows XP Home Edition, Windows XP Professional, Windows XP Starter Edition, Windows XP Tablet PC Edition, Windows XP Media Center Edition и Windows XP Embedded.

Windows Server 2003 выпускается в шести разновидностях: Windows Server 2003 Web Edition, Standard Edition, Small Business Server, Storage Server, Enterprise Edition и Datacenter Edition.

Эти версии различаются по следующим параметрам:

- числу поддерживаемых CPU;
- объему поддерживаемой физической памяти;
- возможному количеству одновременных сетевых соединений (например, в клиентской версии допускается максимум 10 одновременных соединений со службой доступа к общим файлам и принтерам);
- наличием в выпусках Server сервисов, не входящих в Professional (например, служб каталогов, поддержкой кластеризации и многопользовательской службы терминала).

Проверочный выпуск

Специальная отладочная версия Windows 2000 Professional, Windows XP Professional или Windows Server 2003 называется проверочным выпуском (checked build). Она доступна только подписчикам MSDN уровня Professional (или выше). Проверочный выпуск представляет собой перекомпилированный исходный код Window для которого флаг «DBG» (заставляющий включать код отладки и трассировки этапа компиляции) был установлен как TRUE. Кроме того, чтобы облегчить восприятие машинного кода, отключается обработка двоичных файлов, при которой структура кода оптимизируется для большего быстродействия (см. раздел «Performance-Optimized Code» в справочном файле Debugging Tools).

Проверочный выпуск предназначен главным образом разработчикам драйверов устройств, поскольку эта версия выполняет более строгую проверку ошибок при вызове функций режима ядра драйверами устройств или другим системным кодом. Например, если драйвер (или какой-то иной код режима ядра) вызывает системную функцию, контролирующую передаваемые параметры, то при обнаружении этой проблемы система останавливается, предотвращая повреждение структур данных и возможный крах.

Ключевые компоненты системы

Рассмотрим роль каждого ключевого компонента системы. На Рис. 7 отражена более подробная схема системной архитектуры Windows. Заметьте, что на ней все равно не показаны некоторые компоненты (в частности, компоненты сетевой поддержки).

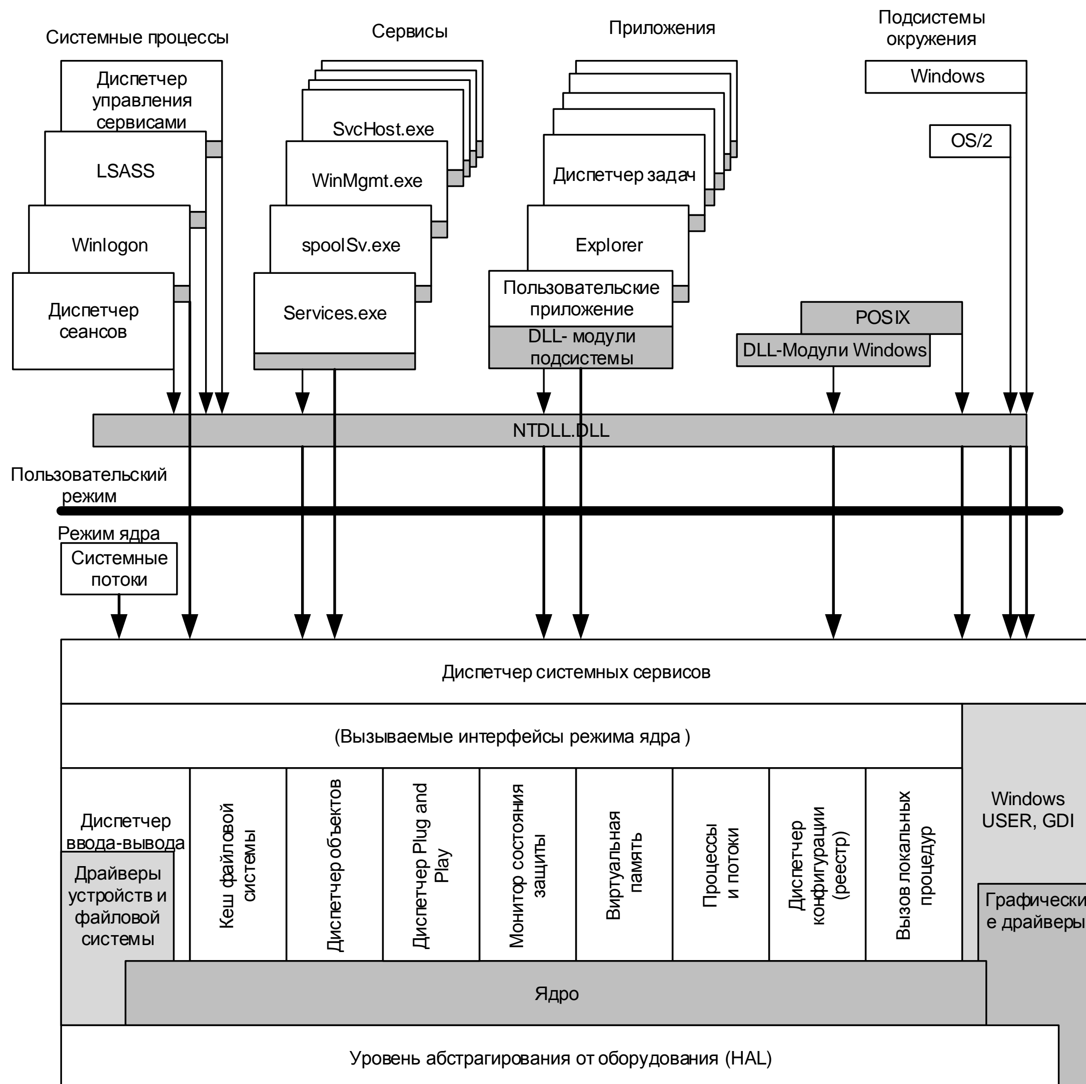

Рис. 7. Архитектура Windows

Подсистемы окружения и их DLL

Как показано на Рис. 7, в Windows имеется три подсистемы окружения: OS/2, POSIX и Windows. Как уже говорили, подсистема OS/2 была удалена в Windows 2000, Начиная с Windows XP, базовая подсистема POSIX не поставляется с Windows, но ее гораздо более совершенную версию можно получить бесплатно как часть продукта Services for UNIX.

Подсистема Windows отличается от остальных двух тем, что без нее Windows работать не может (эта подсистема обрабатывает все, что связано с клавиатурой, мышью и экраном, и нужна даже на серверах в отсутствие интерактивных пользователей). Фактически остальные две подсистемы запускаются только по требованию, тогда как подсистема Windows работает всегда, стартовая информация подсистемы хранится в разделе реестра:

`HKLM\ SYSTEM\CurrentControlSet\Control\Session Manager\SubSystems`.

Подсистемы окружения предоставляют прикладным программам некое подмножество базовых сервисов исполнительной системы Windows. Каждая подсистема обеспечивает доступ к разным подмножествам встроенных сервисов Windows. Это значит, что приложения, созданные для одной подсистемы, могут выполнять операции, невозможные в другой подсистеме. Так, Windows-приложения не могут использовать POSIX-функцию fork.

Каждый исполняемый образ (EXE) принадлежит одной — и только одной — подсистеме. При запуске образа код, отвечающий за создание процесса, получает тип подсистемы,

указанный в заголовке образа, и уведомляет соответствующую подсистему о новом процессе. Тип указывается специфика тором /SUBSYSTEM в команде link в Microsoft Visual C++, его можно просмотреть с помощью утилиты Exetype, входящей в состав ресурсов Windows.

ПРИМЕЧАНИЕ. Процесс подсистемы Windows назван Csrss.exe потому, что в Windows NT все подсистемы изначально предполагалось выполнять, как потоки внутри единственного общесистемного процесса. Когда подсистемы POSIX и OS/2 были выделены в собственные процессы, имя файла процесса подсистемы Windows осталось прежним.

Смешивать вызовы функций разных подсистем нельзя. Иными словами, приложения POSIX могут вызывать только сервисы, экспортируемые подсистемой POSIX, а приложения Windows — лишь сервисы, экспортируемые подсистемой Windows. Это ограничение послужило одной из причин, по которой исходная подсистема POSIX, реализующая весьма ограниченный набор функций (только POSIX 1UU3.1), не стала полезной средой для переноса в нее UNIX-приложений.

Как говорилось, пользовательские приложения не могут вызывать системные сервисы Windows напрямую. Вместо этого они обращаются к DLL подсистем. Эти DLL предоставляют документированный интерфейс между программами и вызываемой ими подсистемой. Так, DLL подсистемы Windows (Kernel32.dll, Advapi32.dll, User32.dll и Gdi32.dll) реализуют функции Windows API. DLL подсистемы POSIX (Psxdll.dll) реализует POSIX API.

При вызове приложением одной из функций DLL подсистемы возможно одно из трех:

- **Функция полностью реализована в пользовательском режиме внутри DLL подсистемы.** Иначе говоря, никаких сообщений процессу подсистемы окружения не посылается, и вызова сервисов исполнительной системы Windows не происходит. После выполнения функции в пользовательском режиме результат возвращается вызвавшей ее программе. Примерами таких функций могут служить `GetCurrentProcess` (всегда возвращает -1, значение, определенное для ссылки на текущий процесс во всех функциях, связанных с процессами) и `GetCurrentProcessId` (идентификатор процесса не меняется в течение его срока жизни, поэтому его можно получить из кэша, что позволяет избежать переключения в режим ядра).
- **Функция требует одного или более вызовов исполнительной системы Windows.** Например, Windows-функции `ReadFile` и `WriteFile` обращаются к внутренним недокументированным сервисам ввода-вывода — соответственно к `NtReadFile` и `NtWriteFile`.
- **Функция требует выполнения каких-либо операций в процессе подсистемы окружения** (такие процессы, работающие в пользовательском режиме, отвечают за обслуживание клиентских приложений, выполняемых под их контролем). В этом случае подсистеме окружения выдается клиент-серверный запрос через сообщение с требованием выполнить какую-либо операцию, и DLL подсистемы, прежде чем вернуть управление вызвавшей программе, ждет соответствующего ответа.

Некоторые функции вроде `CreateProcess` и `CreateThread` могут требовать выполнения как второго, так и третьего пункта.

Подсистема Windows

Эта подсистема состоит из следующих основных элементов.

Процесса подсистемы окружения (Csrss.exe), предоставляющего:

- поддержку консольных (текстовых) окон;
- поддержку создания и удаления процессов и потоков;
- частичную поддержку процессов 16-разрядной виртуальной DOS-машины (VDM);
- множество других функций, например `GetTempFile`, `DefineDosDevice`, `ExitWindowsEx`, а также несколько функций поддержки естественных языков.

Драйвера режима ядра (Wln32k.sys), включающего:

- **Диспетчер окон**, который управляет отрисовкой и выводом окон на экран, принимает ввод с клавиатуры, мыши и других устройств, а также передает пользовательские сообщения приложениям;
- **Graphics Device Interface** (GDI), который представляет собой библиотеку функций для устройств графического вывода. В GDI входят функции для манипуляций с графикой и отрисовки линий, текста и фигур.
- **DLL-модулей подсистем** (Kernel32.dll, Advapi32.dll, User32.dll и Gdi32.dll), транслирующих вызовы документированных функций Windows API в вызовы соответствующих (и в большинстве своем недокументированных) сервисов режима ядра из Ntoskrnl.exe и Win32k.sys.
- **Драйверов графических устройств**, представляющих собой специфичные для конкретного оборудования драйверы графического дисплея, принтера и минипорт-драйверы видеоплат.

Для формирования элементов управления пользовательского интерфейса на экране, например окон и кнопок, приложения могут вызывать стандартные функции USER Диспетчер окон передает эти вызовы GDT, а тот — драйверам графических устройств, где они форматируются для дисплея. Драйвер дисплея работает в паре с соответствующим минипорт-драйвером видеокарты, обеспечивая полную поддержку видео.

GDI предоставляет набор стандартных функций двухмерной графики, которые позволяют приложениям, не имеющим представления о графических устройствах, обращаться к ним. GDI-функции играют роль посредника между приложениями и драйверами дисплея и принтера. GDI интерпретирует запросы приложений на вывод графики и посылает соответствующие запросы драйверам. Он также предоставляет приложениям стандартный унифицированный интерфейс для использования самых разнообразных устройств графического вывода. Этот интерфейс обеспечивает независи-мость кода приложений от конкретного оборудования и его драйверов. GDI выдает свои запросы с учетом возможностей конкретного устройства, час-то разделяя запрос на несколько частей для обработки. Так, некоторые устройства сами умеют формировать эллипсы, а другие требуют от GDI интерпретировать эллипсы как набор пикселов с определенными координатами. Подробнее об архитектуре подсистемы вывода графики и драйвере дисплея см. раздел «Design Guide» в книге «Graphics Drivers» из Windows DDK.

Например, каждый клиентский поток обслуживается парным серверным потоком в процессе подсистемы Windows, ожидающем запросов от клиентского потока. Для передачи сообщений между потоками используется специальный механизм взаимодействия между процессами, так называемый **быстрый LPC** (fast LPC). В отличие от обычного переключения контекста потоков передача данных между парными потоками через быстрый LPC не вызывает в ядре события перепланирования, что позволяет серверному потоку выполняться в течение оставшегося кванта времени клиентского потока (вне очереди, определенной планировщиком). Более того, для быстрой передачи больших структур данных, например битовых карт, используются разделяемые буферы памяти, и клиенты получают прямой доступ (только для чтения) к ключевым структурам данных сервера, а это сводит к минимуму необходимость в частом переключении контекста между клиентами и сервером Windows.

GDI-операции выполняются в пакетном режиме. При этом серия графических объектов, запрошенных Windows-приложениями, не обрабатывается сервером и не прорисовывается на устройстве вывода до тех пор, пока не будет заполнена вся очередь GDI. Размер очереди можно установить через Windows-функцию `GdiSetBatchLimit`. В любой момент все объекты из очереди можно сбросить вызовом функции `GdiFlush`. С другой стороны, неизменяемые свойства и структуры данных GDI после получения от процессов подсистемы Windows кэшируются клиентскими процессами для ускорения последующего доступа к ним.

Однако, несмотря на такую оптимизацию, общая производительность системы по-прежнему не соответствовала требованиям приложений, интенсивно работающих с графикой. Очевидным решением проблемы стал перевод подсистемы поддержки окон и графики в режим ядра, что позволило избежать потребности в дополнительных потоках и связанных с ними переключениями контекста. Кроме того, как только приложения вызывают диспетчер окон и GDI, эти подсистемы теперь получают прямой доступ к компонентам исполнительной системы Windows без перехода из пользовательского режима в режим ядра и обратно. Прямой доступ особенно важен в случае вызова GDI

через видеодрайверы, когда взаимодействие с видеооборудованием требует высокой пропускной способности.

Так что же остается в той части процесса подсистемы Windows, которая работает в пользовательском режиме? Поскольку консольные программы не перерисовывают окна, все операции по отрисовке и обновлению консольных и текстовых окон проводятся именно этой частью Windows. Увидеть ее деятельность несложно: просто откройте окно командной строки и перетащите поверх него другое окно, вы увидите, что процесс подсистемы Windows начинает расходовать время CPU, перерисовывая консольное окно. Кроме поддержки консольных окон, только небольшая часть Windows-функций посылает сообщения процессу подсистемы Windows. К ним относятся функции, отвечающие за создание и завершение процессов и потоков, назначение букв сетевым дискам, создание временных файлов. Как правило, Windows-приложение нечасто переключает (если вообще переключает) контекст в процесс подсистемы Windows.

Подсистема POSIX

POSIX, название которой представляет собой аббревиатуру от «portable operating system interface based on UNIX» (переносимый интерфейс ОС на основе UNIX), — это совокупность международных стандартов на интерфейсы ОС типа UNIX. Стандарты POSIX стимулировали производителей поддерживать совместимость реализуемых ими UNIX-подобных интерфейсов, тем самым позволяя программистам легко переносить свои приложения между системам.

Подсистема OS/2

Подсистема окружения OS/2, как и подсистема POSIX, обладает довольно ограниченной функциональностью и поддерживает лишь 16-разрядные приложения OS/2 версии 1.2 с символьным или графическим вводом-выводом. Кроме того, Windows запрещает прикладным программам прямой доступ к оборудованию и поэтому не поддерживает приложения OS/2, использующие расширенный ввод-вывод видео или включающие сегменты привилегированного ввода-вывода, которые пытаются выполнять инструкции IN/ OUT (для доступа к некоторым аппаратным устройствам). Приложения, выдающие машинные команды CLI/STI, могут работать в Windows, но на время выполнения команды STI все другие приложения OS/2 в системе и потоки процессов OS/2, выдающих команды CLI, приостанавливаются.

Ntdll.dll

Ntdll.dll — специальная библиотека системной поддержки, нужная в основном при использовании DLL подсистем. Она содержит функции двух типов:

- интерфейсы диспетчера системных сервисов (system service dispatch stubs) к сервисам исполнительной системы Windows;
- внутренние функции поддержки, используемые подсистемами DLL, подсистем и другими компонентами ОС.

Первая группа функций предоставляет интерфейс к сервисам исполнительной системы Windows, которые можно вызывать из пользовательского режима. Таких функций более 200, например `NtCreateFile`, `NtSetEvent` и т. д. Как уже говорилось, большинство из них доступно через Windows API (однако некоторые из них предназначены только для применения внутри самой ОС).

Для каждой из этих функций в Ntdll существует точка входа с тем же именем. Код внутри функции содержит специфичную для конкретной аппаратной архитектуры команду перехода в режим ядра для вызова диспетчера системных сервисов, который после проверки некоторых параметров вызывает уже настоящий сервис режима ядра из Ntoskrnl.exe.

Исполнительная система

Исполнительная система (executive) находится на верхнем уровне Ntoskrnl.exe (ядро располагается на более низком уровне). В ее состав входят функции следующего типа:

- Экспортируемые функции, доступные для вызова из пользовательского режима. Эти функции называются **системными сервисами** и экспортируются через Ntdll. Большинство сервисов доступно через Windows API или API других подсистем окружения. Однако некоторые из них недоступны через

документированные функции (примером могут служить LPC, функции запросов вроде `NtQuerylnformationProcess`, специализированные функции типа `NtCreatePagingFile` и т.д.).

- Функции драйверов устройств, вызываемые через функцию `DeviceloControl`. Последняя является универсальным интерфейсом от пользовательского режима к режиму ядра для вызова функций в драйверах устройств, не связанных с чтением или записью.
- Экспортируемые функции, доступные для вызова только из режима ядра и документированные в Windows DDK или Windows Installable File System (IFS) Kit (см. www.microsoft.com/whdc/ddk/ifskit.default.mspx).
- Экспортируемые функции, доступные для вызова только из режима ядра, но не описанные в Windows DDK или IFS Kit (например, функции, которые используются видеодрайвером, работающим на этапе загрузки, и чьи имена начинаются с **Inbv**).
- Функции, определенные как глобальные, но не экспортируемые символы. Включают внутренние функции поддержки, вызываемые в Ntoskrnl; их имена начинаются с **lop** (функции поддержки диспетчера ввода-вывода) или с **Mi** (функции поддержки управления памятью).
- Внутренние функции в каком-либо модуле, не определенные как глобальные символы.

Исполнительная система состоит из следующих основных компонентов.

- **Диспетчер конфигурации**, отвечающий за реализацию и управление системным реестром.
- **Диспетчер процессов и потоков**, создающий и завершающий процессы и потоки. Низкоуровневая поддержка процессов и потоков реализована в ядре Windows, а исполнительная система дополняет эти низкоуровневые объекты своей семантикой и функциями.
- **Монитор состояния защиты**, реализующий политики безопасности на локальном компьютере. Он охраняет ресурсы ОС, осуществляя аудит и контролируя доступ к объектам в период выполнения.
- **Диспетчер ввода-вывода**, реализующий аппаратно-независимый ввод-вывод и отвечающий за пересылку ввода-вывода нужным драйверам устройств для дальнейшей обработки.
- **Диспетчер Plug and Play**, определяющий, какие драйверы нужны для поддержки конкретного устройства, и загружающий их. Требования каждого устройства в аппаратных ресурсах определяются в процессе перечисления. В зависимости от требований каждого устройства диспетчер PnP распределяет такие ресурсы, как порты ввода-вывода, IRQ, каналы DMA и области памяти. Он также отвечает за посылку соответствующих уведомлений об изменениях в аппаратном обеспечении системы (при добавлении или удалении устройств).
- **Диспетчер электропитания**, который координирует события, связанные с электропитанием, и генерирует уведомления системы управления электропитанием, посылаемые драйверам. Когда система не занята, диспетчер можно настроить на остановку CPU для снижения энергопотребления. Изменение энергопотребления отдельных устройств возлагается на их драйверы, но координируется диспетчером электропитания.
- **Подпрограммы WDM Windows Management Instrumentation**, позволяющие драйверам публиковать информацию о своих рабочих характеристиках и конфигурации, а также получать команды от службы WMI пользовательского режима. Потребители информации WMI могут находиться как на локальной машине, так и на любом компьютере в сети.
- **Диспетчер кэша**, повышающий производительность файлового ввода-вывода за счет сохранения в основной памяти дисковых данных, к которым недавно было обращение (это также уменьшает число обращений к диску для записи, поскольку модифицированные данные предварительно накапливаются в памяти в течение определенного периода). Как видите, диспетчер кэша выполняет эту задачу, используя поддержку проецируемых файлов со стороны диспетчера памяти.

- **Диспетчер памяти**, реализующий виртуальную память — схему управления памятью, позволяющую выделять каждому процессу большое закрытое адресное пространство, объем которого может превышать доступную физическую память. Диспетчер памяти также обеспечивает низкоуровневую поддержку диспетчера кэша.
- **Средство логической предвыборки**, ускоряющее запуск системы и процессов за счет оптимизации загрузки данных, к которым происходит обращение при запуске системы или процессов.

Кроме того, в состав исполнительной системы входят четыре основные группы функций поддержки, используемые вышеперечисленными компонентами. Примерно треть из них описана в DDK, поскольку драйверы тоже используют их.

- **Диспетчер объектов** — создает, управляет и удаляет объекты и абстрактные типы данных исполнительной системы, используемые для представления таких ресурсов операционной системы, как процессы, потоки и различные синхронизирующие объекты.
- **Механизм LPC** — передает сообщения между клиентским и серверным процессами на одном компьютере. LPC является гибкой, оптимизированной версией RPC (remote procedure call), стандартного механизма взаимодействия между клиентскими и серверными процессами через сеть.
- **Большой набор стандартных библиотечных функций для обработки строк**, арифметических операций, преобразования типов данных и обработки структур безопасности.
- **Подпрограммы поддержки исполнительной системы**, например, для выделения системной памяти (пулов подкачиваемых и не подкачиваемых страниц), доступа к памяти со взаимоблокировкой, а также два специальных типа синхронизирующих объектов - ресурс (resource) и быстродействующий мьютекс (fast mutex).

Ядро

Ядро состоит из набора функций в Ntoskrnl.exe, предоставляющих фундаментальные механизмы, которые используются компонентами исполнительной системы и низкоуровневыми аппаратно-зависимыми средствами поддержки (диспетчеризации прерываний и исключений), различными в каждой архитектуре CPU. Код ядра написан в основном на С, а ассемблер использовали лишь для решения специфических задач, трудно реализуемых на С.

Как и функции поддержки исполнительной системы, часть функций ядра описана в DDK, поскольку они необходимы для реализации драйверов устройств.

Объекты ядра

Ядро состоит из низкоуровневых, четко определенных и хорошо предсказуемых примитивов и механизмов ОС, позволяющих компонентам исполнительной системы более высокого уровня выполнять свои функции. Ядро отделено от остальной части исполнительной системы; оно реализует системные механизмы и не участвует в принятии решений, связанных с системной политикой.

Вне ядра исполнительная система представляет потоки и другие разделяемые ресурсы в виде объектов. Управление этими объектами требует определенных издержек, так как нужны описатели, позволяющие манипулировать объектами, средства защиты и квоты ресурсов, резервируемых при их создании. В ядре можно избежать таких издержек, поскольку оно реализует набор более простых объектов, называемых **объектами ядра** (kernel objects). Эти объекты позволяют ядру контролировать обработку данных CPU и поддерживают объекты исполнительной системы.

Одна из групп объектов ядра, называемых **управляющими** (control objects), определяет семантику управления различными функциями ОС. В эту группу входят объекты АРС, DPC (deferred procedure call) и несколько объектов, используемых диспетчером ввода-вывода (например, объект прерывания).

Другая группа объектов - **объекты диспетчера** (dispatcher objects) реализует средства синхронизации, позволяющие изменять планирование потоков. В группу таких объектов входят:

- поток ядра (kernel thread),
- мьютекс (mutex),
- событие (event),
- семафор (semaphore),
- таймер (timer), ожидаемый таймер (waitable timer)
- и некоторые другие.

С помощью функций ядра исполнительная система создает объекты ядра, манипулирует ими и конструирует более сложные объекты, предоставляемые в пользовательском режиме.

Поддержка оборудования

Другая важная задача ядра — **абстрагирование** или **изоляция исполнительной системы** и **драйверов устройств** от различий между аппаратными архитектурами, поддерживаемыми Windows (т.е. различий в обработке прерываний, диспетчеризации исключений и синхронизации между несколькими CPU).

Архитектура ядра нацелена на максимальное обобщение кода — даже в случае аппаратно-зависимых функций. Ядро поддерживает набор семантически идентичных и переносимых между архитектурами интерфейсов. Большая часть кода, реализующего переносимые интерфейсы, также идентична для разных архитектур.

Но одна часть этих интерфейсов по-разному реализуется на разных архитектурах, а другая включает код, специфичный для конкретной архитектуры. Архитектурно-независимые интерфейсы могут быть вызваны на любой машине, причем семантика интерфейса будет одинаковой — независимо от специфического кода для той или иной архитектуры. Некоторые интерфейсы ядра на самом деле реализуются в HAL, поскольку их реализация может отличаться даже в пределах семейства CPU с одинаковой архитектурой.

В ядре также содержится небольшая порция кода с x86-специфичными интерфейсами, необходимыми для поддержки старых программ MS-DOS. Эти интерфейсы не являются переносимыми в том смысле, что их нельзя вызывать на машине с другой архитектурой, где они попросту отсутствуют. Этот x86-специфичный код, например, поддерживает манипуляции с **GDT** (global descriptor tables) и **LDT** (local descriptor tables) — аппаратными средствами x86.

Уровень абстрагирования от оборудования

Как отмечалось, одной из важнейших особенностей архитектуры Windows является переносимость между различными аппаратными платформами. Ключевой компонент, обеспечивающий такую переносимость, — **уровень абстрагирования от оборудования** (hardware abstraction layer, HAL). **HAL** — это загружаемый модуль режима ядра (Hal.dll), предоставляющий низкоуровневый интерфейс с аппаратной платформой, на которой выполняется Windows. Он скрывает от ОС специфику конкретной аппаратной платформы, в том числе ее интерфейсов ввода-вывода, контроллеров прерываний и механизмов взаимодействия между CPU, т. е. все функции, зависимые от архитектуры и от конкретной машины.

Когда внутренним компонентам Windows и драйверам устройств нужна платформенно-зависимая информация, они обращаются не к самому оборудованию, а к подпрограммам HAL, что и обеспечивает переносимость этой ОС. По этой причине подпрограммы HAL документированы в Windows DDK, где можно найти более подробные сведения о HAL и о его использовании драйверами.

Драйверы устройств

Драйверы устройств являются загружаемыми модулями режима ядра (как правило, это файлы с расширением .sys); они образуют интерфейс между диспетчером ввода-вывода и соответствующим оборудованием. Эти драйверы выполняются в режиме ядра в одном из трех контекстов:

- в контексте пользовательского потока, инициировавшего функцию ввода-вывода;
- в контексте системного потока режима ядра;
- как результат прерывания (а значит, не в контексте какого-либо процесса или потока, который был текущим на момент прерывания).

Драйверы, как правило, пишутся на С (иногда на С++), поэтому при правильном использовании процедур HAL они являются переносимыми между поддерживаемыми Windows архитектурами на уровне исходного кода, а на уровне двоичных файлов — внутри семейства с одинаковой архитектурой. Существует несколько типов драйверов устройств.

- **Драйверы аппаратных устройств**, которые управляют (через HAL) оборудованием, записывают на них выводимые данные и получают вводимые данные от физического устройства или из сети. Есть множество типов таких драйверов — драйверы шин, интерфейсов, устройств массовой памяти и т.д.
- **Драйверы файловой системы** — драйверы Windows, принимающие запросы на файловый ввод-вывод и транслирующие их в запросы ввода-вывода для конкретного устройства.
- **Драйверы фильтра файловой системы**, которые обеспечивают зеркалирование и шифрование дисков, перехват ввода-вывода и некоторую дополнительную обработку информации перед передачей ее на следующий уровень.
- **Сетевые редиректоры и серверы**, являющиеся драйверами файловых систем, которые передают запросы файловой системы на ввод-вывод другим компьютерам в сети и принимают от них аналогичные запросы.
- **Драйверы протоколов**, реализующие сетевые протоколы вроде TCP/IP, NetBIOS и IPX/SPX.
- **Драйверы потоковых фильтров ядра**, действующие по цепочке для обработки потоковых данных, например при записи и воспроизведении аудио- и видеоинформации.

Поскольку установка драйвера устройства — единственный способ добавления в систему стороннего кода режима ядра, некоторые программисты пишут драйверы просто для того, чтобы получить доступ к внутренним функциям или структурам данных ОС, недоступным из пользовательского режима.

Усовершенствования в модели драйверов Windows

С точки зрения WDM, существует три типа драйверов.

- **Драйвер шины** (bus driver), обслуживающий контроллер шины, адаптер, мост или любые другие устройства, имеющие дочерние устройства. Драйверы шин нужны для работы системы и в общем случае поставляются Microsoft. Для каждого типа шины (PCI, PCMCIA и USB) в системе имеется свой драйвер. Сторонние разработчики создают драйверы для поддержки новых шин вроде VMEbus, Multibus и Futurebus.
- **Функциональный драйвер** (function driver) — основной драйвер устройства, предоставляющий его функциональный интерфейс. Обязателен, кроме тех случаев, когда устройство используется без драйверов (т.е. ввод-вывод осуществляется драйвером шины или драйверами фильтров шины, как в случае SCSI PassThru). Функциональный драйвер по определению обладает наиболее полной информацией о своем устройстве. Обычно только этот драйвер имеет доступ к специфическим регистрам устройства.
- **Драйвер фильтра** (filter driver) поддерживает дополнительную функциональность устройства (или существующего драйвера) или изменяющий запросы на ввод-вывод и ответы на них от других драйверов (это часто используется для коррекции устройств, предоставляющих неверную информацию о своих требованиях к аппаратным ресурсам). Такие драйверы не обязательны, и их может быть несколько. Они могут работать как на более высоком уровне, чем функциональный драйвер или драйвер шины, так и на более низком. Обычно эти драйверы предоставляются OEM-производителями или независимыми поставщиками оборудования (IHV).

В среде WDM один драйвер не может контролировать все аспекты устройства: драйвер шины информирует диспетчер PnP об устройствах, подключенных к шине, в то время как функциональный драйвер управляет устройством.

В большинстве случаев драйвер фильтра более низкого уровня модифицирует поведение устройства. Драйвер фильтра более высокого уровня обычно придает

устройству дополнительную функциональность. Так, высокоуровневый драйвер фильтра для клавиатуры может обеспечивать дополнительную защиту.

Системные процессы

В каждой системе Windows выполняются перечисленные ниже процессы. (Два из них, Idle и System, не являются процессами в строгом смысле этого слова, поскольку они не выполняют какой-либо код пользовательского режима).

- Процесс Idle (включает по одному потоку на CPU для учета времени простоя CPU).
- Процесс System (содержит большинство системных потоков режима ядра).
- Диспетчер сеансов (Smss.exe).
- Подсистема Windows (Csrss.exe).
- Процесс входа в систему (Winlogon.exe).
- Диспетчер управления сервисами (Services.exe) и создаваемые им дочерние процессы сервисов (например, универсальный процесс для хостинга сервисов, Svchost.exe).
- Серверный процесс локальной аутентификации (Lsass.exe).

Чтобы понять взаимоотношения этих процессов, полезно просмотреть «дерево» процессов, отражающее связи между родительскими и дочерними процессами. Увидев, кем создается тот или иной процесс, вам будет легче понять, откуда берется каждый процесс. Более детально о каждом процесс можно узнать в книге [3].

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