Конспект лекций по дисциплине

«ОПЕРАЦИОННЫЕ СИСТЕМЫ»

Литература

1. Таненбаум Э., Вудхал А. Операционные системы. Разработка и реализация. 3-е изд. – СПб.: Питер, 2007. – 704 с.
2. Гордеев А.В. Операционные системы. – СПб.: Питер, 2007. – 416 с.
3. Гордеев А.В., Молчанов А.Ю. Системное программное обеспечение. – СПб.: Питер, 2002. – 736 с.
4. Харт Д.М. Системное программирование в среде Windows.: пер. с англ. – М.: Издательский дом «Вильямс», 2005. – 529 с.
5. Гагарина Л.Г., Кокорева Е.В., Виснадул Б.Д. Технология разработки программного обеспечения. – М.: ИД «ФОРУМ»: ИНФРА-М, 2008. – 400 с.
6. Иртегов Д.В. Введение в операционные системы. – СПб.: БХВ-Петербург, 2002. – 624 с.
7. Олифер В.Г., Олифер Н.А. Сетевые операционные системы. – СПб.: Питер, 2001. – 544 с.
8. Партыка Т.Л., Попов И.И. Операционные системы, среды и оболочки. М.: ФОРУМ: ИНФРА-М, 2007. – 528 с.
9. Реймонд С. Искусство программирования для UNIX. – М.: Издательский дом «Вильямс», 2005. – 544 с.

Лекция № 1. ОСНОВНЫЕ ПОНЯТИЯ

1.1. Системное и прикладное программное обеспечение

Все программное обеспечение (ПО) делится на ***системное*** и ***прикладное***. Системными принято называть такие программы, которые используются всеми остальными программами, без них невозможно создание и выполнение прикладных программ. Прикладные программы выполняют научные, технические и иные задачи, непосредственно не связанные с управлением компьютером.

В составе системного ПО можно выделить две составляющие: базовое и сервисное.

Базовое ПО – это минимальный набор программных средств, обеспечивающих работу компьютера. В него входят: ***операционная система*** и ***операционные оболочки***.

Сервисное ПО – программы и программные комплексы, которые расширяют возможности базового ПО и организуют более удобную среду работы пользователя. К сервисному ПО также относятся средства разработки (редакторы) и создания программ (компиляторы и интерпретаторы).

<table>
<tr><td>Веб-браузер</td><td>Базы данных</td><td>Игры</td><td>Прикладное ПО</td></tr>
<tr><td>Компиляторы</td><td>Редакторы</td><td>Интерпретаторы</td><td rowspan="2">Системное ПО</td></tr>
<tr><td colspan="3">Операционная система</td></tr>
<tr><td colspan="3">Машинный язык</td><td rowspan="3">Аппаратура</td></tr>
<tr><td colspan="3">Микроархитектура</td></tr>
<tr><td colspan="3">Физические устройства</td></tr>
</table>

Рис. 1.1.

Место операционной системы в общей структуре компьютера показано на рис. 1.1. Самый нижний слой модели – это физические устройства, которые входят в состав компьютера: интегральные микросхемы, платы, источники питания, дисплей, клавиатура и т.д. Отдельные устройства

объединяются в функциональные блоки и образуют ***микроархитектуру*** компьютера. На микроархитектурном уровне находятся внутренние регистры ЦПУ (центрального процессорного устройства) и тракт данных, включающий арифметико-логическое устройство.

Тракт данных предназначен для выполнения набора команд. Аппаратное обеспечение и команды, доступные программисту на языке ***ассемблера***, образуют ***архитектуру набора команд***. Зачастую данный уровень называют также ***машинным языком***.

Ассемблер подробно описывает все действия, которые необходимо выполнить, чтобы управлять многочисленными устройствами компьютера. Основное предназначение операционной системы заключается в том, чтобы скрыть все эти сложности и предоставить программисту более удобную систему команд. С точки зрения пользователя операционная система выполняет функцию ***виртуальной машины***, для которой проще программировать и с которой легче работать, чем непосредственно с аппаратным обеспечением компьютера.

Под операционной системой обычно понимается то программное обеспечение, которое запускается в ***режиме ядра*** или, как его еще называют, ***режиме супервизора***. Операционная система защищена от вмешательства пользователя с помощью аппаратных средств.

Компиляторы и редакторы запускаются в ***пользовательском режиме***. Если пользователю не нравится какой-либо компилятор, он при желании может написать собственный, но ему не удастся написать собственный обработчик прерываний от системных часов, являющийся частью операционной системы и обычно защищенный аппаратно от попыток его модифицировать.

Поверх системных программ выполняются прикладные программы. Обычно они покупаются пользователем (или пишутся им) для решения собственных проблем – обработке текста, электронных таблиц, технических расчетов или хранения информации в базе данных.

1.2. Операционные системы

Операционная система (ОС) представляет собой комплекс управляющих и обрабатывающих программ, которые, с одной стороны, выступают как интерфейс между аппаратурой компьютера и пользователем с его задачами, а с другой стороны, предназначены для наиболее эффективного расходования ресурсов вычислительной системы и организации надежных вычислений.

Операционная система изолирует аппаратное обеспечение компьютера от прикладных программ пользователей. И пользователь, и его программы взаимодействуют с компьютером через интерфейсы операционной системы.

Примеры операционных систем: UNIX, OS/2, Windows, Linux, QNX, MacOS, BeOS.

Пренебрегая детализацией, можно сказать, что операционные системы выполняют две основные функции. Во-первых, ***расширяют реальные физические возможности компьютера.*** Например, путем виртуального увеличения объема его оперативной памяти или обеспечения многозадачного режима работы на одном процессоре. Во-вторых, ***управляют ресурсами компьютера***, в частности, памятью и устройствами ввода-вывода.

Более детальный перечень функций приведен ниже:

1. Прием от пользователя (или от оператора системы) заданий, или команд, сформулированных на соответствующем языке, и их обработка.
2. Загрузка в оперативную память подлежащих исполнению программ.
3. Распределение памяти.
4. Запуск программы (передача ей управления, в результате чего процессор исполняет программу).
5. Идентификация всех программ и данных.
6. Прием и исполнение различных запросов от выполняющихся приложений. ОС умеет выполнять большое количество системных

функций (сервисов), которые могут быть запрошены из выполняющейся программы. Обращение к этим сервисам осуществляется по определенным правилам, которые определяют ***интерфейс прикладного программирования*** (Application Program Interface, API) этой операционной системы.

7. Обслуживание всех операций ввода-вывода.
8. Обеспечение работы систем управления файлами (СУФ) и/или систем управления базами данных (СУБД), что позволяет резко увеличить эффективность всего программного обеспечения.
9. Обеспечение режима мультипрограммирования, то есть организации параллельного выполнения двух или более программ на одном процессоре, которая создает видимость их одновременного исполнения.
10. Планирование и диспетчеризация задач в соответствии с заданными стратегией и дисциплинами обслуживания.
11. Организация механизмов обмена сообщениями и данными между выполняющимися программами.
12. Для сетевых операционных систем характерной является функция обеспечения взаимодействия связанных между собой компьютеров.
13. Защита одной программы от влияния другой, обеспечение сохранности данных, защита самой операционной системы от исполняющихся на компьютере приложений.
14. Аутентификация и авторизация пользователей (для большинства диалоговых операционных систем). Под ***аутентификацией*** понимается процедура проверки имени пользователя и его пароля на соответствие тем значениям, которые хранятся в его учетной записи. Если входное имя (login) пользователя и его пароль совпадают, то, скорее всего, это и будет тот самый пользователь. Термин ***авторизация*** означает, что в соответствии с учетной записью пользователя, который прошел аутентификацию, ему (и

всем запросам, которые будут идти к операционной системе от его имени) назначаются определенные права (привилегии), определяющие, что он может, а чего не может делать на компьютере.

15. Удовлетворение жестким ограничениям на время ответа в режиме реального времени (характерно для операционных систем реального времени).
16. Обеспечение работы ***систем программирования***, с помощью которых пользователи готовят свои программы.
17. Предоставление услуг на случай частичного сбоя системы.

Операционная система состоит из множества программных модулей. Главный модуль операционной системы называется с***упервизором*** (supervisor). В сложных операционных системах он может состоять из нескольких модулей, например супервизора ввода-вывода, супервизора прерываний, супервизора программ, диспетчера задач и т.д.

В литературе также часто используется термин ***ядро*** (kernel) операционной системы, который понимается как синоним супервизора.

При необходимости использовать какой-нибудь ресурс (оперативную память, устройство ввода-вывода, массив данных и т.п.) вычислительный процесс путем обращения к супервизору операционной системы посредством специальных вызовов сообщает о своем требовании. При этом указывается вид ресурса и, если надо, его объем. Например, при запросе оперативной памяти указывается количество адресуемых ячеек, необходимое для дальнейшей работы.

Команда обращения к операционной системе передает ей управление, переводя процессор в привилегированный режим работы. Большинство компьютеров имеют два (и более) режима работы: ***привилегированный*** (режим супервизора) и ***пользовательский***. Ресурс может быть выделен

вычислительному процессу, обратившемуся к операционной системе с соответствующим запросом, если:

1. ресурс свободен и в системе нет запросов от задач более высокого приоритета к этому же ресурсу;
2. текущий запрос и ранее выданные запросы допускают совместное использование ресурсов;
3. ресурс используется задачей низшего приоритета и может быть временно отобран (разделяемый ресурс).

Получив запрос, операционная система либо удовлетворяет его и возвращает управление задаче, выдавшей данный запрос, либо, если ресурс занят, ставит задачу в очередь к ресурсу, переводя ее в состояние ожидания (блокируя). Очередь к ресурсу может быть организована несколькими способами, но чаще всего она реализуется с помощью списковой структуры.

После окончания работы с ресурсом задача опять с помощью специального вызова супервизора (посредством соответствующей команды) сообщает операционной системе от отказа от ресурса, либо операционная система забирает ресурс сама, если управление возвращается супервизору после выполнения какой-либо системной функции.

Супервизор, получив управление по этому обращению, освобождает ресурс и проверяет, имеется ли очередь к освободившемуся ресурсу. Если очередь есть, то он выводит из состояния ожидания задачу, ждущую ресурс, и переводит ее в состояние готовности к выполнению, после чего либо передает управление ей, либо возвращает управление задаче, только что освободившей ресурс.

1.3. Операционные среды

Прикладная программа, созданная для работы в некоторой операционной системе, не будет работать в другой операционной системе, поскольку API у этих операционных систем различаются. Стремясь преодолеть это ограничение, разработчики операционных систем стали

создавать так называемые ***операционные среды***. Операционная система (в общем случае) может поддерживать несколько операционных сред, связанных с другими операционными системами.

Та программная среда, которая непосредственно образуется кодом операционной системы, называется ***основной, естественной***, или ***нативной*** (native – по английски «туземец»). Помимо основной операционной среды в операционной системе могут быть организованы (путем эмуляции иной операционной среды) дополнительные программные среды.

Эмуляция (англ. emulation) – воспроизведение программными или аппаратными средствами (либо их комбинацией) работы других программ или устройств. В отличие от ***симуляции*** (simulation), которая лишь воспроизводит поведение программы, при эмуляции ставится цель точного моделирования состояния имитируемой системы, для выполнения оригинального машинного кода.

Обычно эмуляцию используют для осуществления следующих целей.

- Создание нового микропроцессора. В этом случае при помощи эмулятора на другом микропроцессоре выполняются команды этого еще не существующего процессора.
- Необходимость выполнения программного обеспечения, написанного для другого устройства или операционной системы.
- Тестирование программ написанных для различных систем.

При использовании языков высокого уровня, иногда в целях сохранения быстродействия исполняемой программы, вместо эмуляции делают ***портирование*** программ в новую среду. В этом случае производится переписывание заново аппаратно-зависимых участков кода.

Если в операционной системе организована работа с различными операционными средами, то в такой системе можно выполнять программы, созданные не только для данной, но и для других операционных систем. Например, можно создать программу для работы в среде DOS. Если такая программа все функции, связанные с операциями ввода-вывода и запросами

памяти, выполняет не сама, а за счет обращения к системным функциям DOS, то она будет (в абсолютном большинстве случаев) успешно выполняться и в MS DOS, и в OS/2, и в Windows 2000, и даже в Linux.

Операционная система Windows XP позволяет выполнять помимо основных приложений, созданных с использованием Win32API, 16-разрядные приложения для Windows 3.x, 16-разрядные DOS-приложения, 16-разрядные приложения для первой версии OS/2.

Эмуляцию Windows в UNIX можно осуществить с помощью программы WineHQ. Информация по этому вопросу и по загрузке пакета с открытым исходным кодом Wine, позволяющего эмулировать Windows API поверх UNIX, содержится на сайте http://www.winehq.com.

Необходимо заметить, что WineHQ не является обычным эмулятором. Об этом говорит аббревиатура этой программы: «Wine Is Not an Emulator». Вместо действия в качестве полного эмулятора Wine создает «слой совместимости», обеспечивая альтернативное подсоединение динамически связываемых библиотек (Dynamic Link Library), которые вызывают программы Windows, и выполнение процессов, совместимых с ядром Windows NT.

Операционная среда – это то системное программное окружение, в котором могут выполняться программы, созданные по правилам работы этой среды. Операционная среда может быть либо нативной (естественной), либо может быть организована в чужой операционной системе путем эмуляции.

1.4. Операционные оболочки

Как правило, все операционные системы имеют интерфейс командной строки. Хотя системному администратору без него не обойтись, пользоваться им не всегда удобно, поскольку необходимо держать в голове множество команд, принятых в данной операционной системе.

Для преодоления этого недостатка было создано множество программных «оболочек» – shell (по английски – «раковина»).

К ним относятся Norton Commander – программа, созданная как надстройка над DOS, FAR Manager – текстовая оболочка для Windows 95/98/NT/2000/XP, Midnight Commander – программная оболочка системы Linux и т.п. Программные оболочки предлагают пользователю меню, из которого он может выбрать желаемое действие.

В последнее время операционные оболочки активно вытесняются графическими интерфейсами (Graphical User Interface – GUI), например X-Window с различными менеджерами окон – KDE, Gnome и т.п., которые приобретают все большую популярность у пользователей.

По-видимому, операционные оболочки можно рассматривать как нечто промежуточное между интерфейсом командной строки и графическими интерфейсами.

Лекция № 2. КЛАССИФИКАЦИЯ ОПЕРАЦИОННЫХ СИСТЕМ

2.1. Введение

В зависимости от количества одновременно выполняемых задач операционные системы делятся на ***однозадачные*** и ***многозадачные***.

В зависимости от размера ядра – на ***микроядерные*** и ***макроядерные***.

В зависимости от количества выполняемых функций – на ***специализированные*** и ***системы общего назначения***.

Операционные системы компьютеров, оснащенных несколькими процессорами, делятся на две категории: с ***асимметричной*** либо ***симметричной*** обработками.

2.2. Однозадачные и многозадачные ОС

В первых вычислительных системах любая программа могла выполняться только после полного завершения предыдущей. Поскольку эти первые вычислительные системы были построены в соответствии с принципами, изложенными в известной работе Яноша Джона фон Неймана, все подсистемы и устройства компьютера управлялись исключительно центральным процессором. Центральный процессор осуществлял и выполнение вычислений, и управление операциями ввода-вывода данных. Соответственно, пока осуществлялся обмен данными между оперативной памятью и внешними устройствами, процессор не мог выполнять вычисления. Введение в состав вычислительной машины специальных контроллеров позволило совместить во времени (распараллелить) операции вывода полученных данных и последующие вычисления на центральном процессоре. Однако все равно процессор продолжал часто и долго простаивать, дожидаясь завершения очередной операции ввода-вывода. Поэтому было предложено организовать так называемый ***мультипрограммный***, или ***мультизадачный***, режим работы вычислительной системы.

Смысл мультипрограммного режима работы заключается в том, что пока одна программа (один вычислительный процесс) ожидает завершения

очередной операции ввода-вывода, другая программа может быть поставлена на решение. Это позволяет более полно использовать имеющиеся ресурсы и уменьшить общее время, необходимое для решения некоторого множества задач.

Однако, при этом время выполнения каждого процесса в общем случае больше, чем, если бы мы выполняли каждый из них как единственный. При мультипрограммировании повышается пропускная способность системы, но отдельный процесс никогда не может быть выполнен быстрее, чем, если бы он выполнялся в однопрограммном режиме. Всякое разделение ресурсов замедляет работу одного из участников за счет дополнительных затрат времени на ожидание освобождения ресурса.

Совмещение диалогового режима работы с компьютером и режима мультипрограммирования привело к появлению ***многопользовательских*** систем. Организовать параллельное выполнение нескольких задач можно разными способами. Если это осуществляется таким образом, что на каждую задачу поочередно выделяется некий квант времени, после чего процессор передается другой задаче, готовой к продолжению вычислений, то такой режим принято называть режимом ***разделения времени*** (time sharing).

2.3. Микроядерные и макроядерные ОС

В микроядерных системах ядро (главный модуль системы) имеет размер порядка десятков килобайтов. Остальные модули, образующие набор сервисных приложений, вызываются по мере необходимости. Этот подход вполне соответствует принципам структурного программирования.

Все первые ОС вынужденно являлись микроядерными, поскольку объем оперативной памяти компьютеров поначалу был небольшим. Например, первая версия операционной системы UNIX занимала всего около 12 Кбайт. Однако по мере увеличения объема оперативной памяти ядра операционных систем постепенно начали разрастаться.

Наиболее ярким представителем микроядерных операционных систем является ОС реального времени QNX. Разные версии этой операционной системы имеют объемы – от 8 до 46 Кбайт.

В 90-е годы XX века было весьма распространенным убеждение, что большинство операционных систем следующих поколений будут строиться как микроядерные. Однако практика показывает, что это не совсем так. Разработчики желают иметь компактное микроядро, но при этом включить в него как можно больше функций, исполняемых непосредственно этим программным модулем. Это связано с тем, что выполнение затребованной функции другим модулем, вызываемым из микроядра, приводит и к дополнительным задержкам, и к дополнительным сложностям.

В макроядерных системах ядро получается монолитным, неделимым. Современные ОС общего назначения, такие как Windows, UNIX и Linux относятся к макроядерным системам. Ядра этих систем, представленные в виде программ, написанных на языке высокого уровня, содержат многие миллионы строк кода.

2.4. Специализированные операционные системы

К этому классу относятся системы реального времени; системы мобильных вычислительных устройств; специализированные сетевые системы; а также системы, предназначенные для обучения студентов. В момент возникновения персональных компьютеров их операционные системы (ДОС) также относились к этому классу.

ДОС (Дисковые Операционные Системы). Как правило, это просто некий резидентный набор подпрограмм, не более того. Он загружает пользовательскую программу в память и передает ей управление, после чего программа делает с системой все, что ей заблагорассудится. Считается желательным, чтобы после завершения программы машина оставалась в таком состоянии, чтобы ДОС могла продолжить работу (принципиально же, ДОС ничем не может помешать программе привести систему в нерабочее

состояние). Дисковая операционная система *MS DOS* для *IBM PC* является примером систем подобного класса. Она, правда, умеет загружать несколько программ, но не предоставляет средств для одновременного исполнения этих программ. Существование систем такого класса обусловлено их простотой и тем, что они потребляют мало ресурсов. Еще одна причина, по которой такие системы могут использоваться даже на довольно мощных машинах – требование программной совместимости с ранними моделями того же семейства компьютеров.

Системы реального времени – системы, предназначенные для облегчения разработки так называемых приложений реального времени. Это программы, управляющие некомпьютерным по своей природе оборудованием, часто с очень жесткими ограничениями по времени реакции. Примером такого приложения может быть программа бортового компьютера крылатой ракеты, системы управления ускорителем элементарных частиц или промышленным оборудованием. Такие системы обязаны поддерживать многопроцессность, ***гарантированное*** время реакции на внешнее событие, простой доступ к таймеру и внешним устройствам. Примером такой системы может служить ОС *QNX*. Любопытно, что *multimedia* при качественной реализации предъявляет к системе те же требования, что и промышленные задачи реального времени. В *multimedia* основной проблемой является синхронизация изображения на экране со звуком. Именно в таком порядке. Звук обычно генерируется внешним аппаратным устройством с собственным таймером, и изображение синхронизируется с ним же. Человек способен заметить довольно малые временные неоднородности в звуковом потоке. Напротив, пропуск кадров в визуальном потоке не так заметен, а расхождение звука и изображения заметно уже при задержках около 30 мс. Поэтому системы качественного *multimedia* должны обеспечивать синхронизацию с такой же или более высокой точностью, что мало отличается от систем мягкого реального времени.

Системы для обучения студентов. Во времена молодости UNIX (версия 6) ее исходные коды были широко доступны по лицензии AT&T и активно изучались. Джон Лайонс (John Lions) из университета Нового Южного Уэльса в Австралии даже написал небольшую брошюру, шаг за шагом описывающую работу UNIX. С разрешения AT&T эта брошюра использовалась во многих университетских курсах по операционным системам. С выходом версии 7 система UNIX превратилась в дорогостоящий коммерческий продукт. Лицензия, под которой она распространялась, запрещала преподавание исходного кода на учебных курсах, чтобы не подвергать риску его статус коммерческого секрета. Поэтому многие университеты просто прекратили изучение UNIX, довольствуясь одной теорией. Чтобы исправить ситуацию Эндрив Таненбаум в 1987 году написал собственную операционную систему MINIX (mini-UNIX), предназначенную для обучения студентов, которая с точки зрения пользователя совместима с UNIX, но внутри совершенно самостоятельна. Ядро этой системы имело всего 4000 строк кода, в то время как в UNIX, или в Windows – это миллионы строк кода. Система настолько мала, что даже начинающий мог понять, как она работает. Одним из пользователей MINIX был финский студент по имени Линус Торвальдсен. Он установил ее на свой компьютер и тщательно изучил исходный код. Опыты по усовершенствованию этой системы привели к созданию операционной системы LINUX в 1991 году.

Сетевые системы. Этот термин употребляют в двух различных смыслах:

1. Системы, предназначенные только для предоставления сетевых услуг, аналогично тому, как ДОС предназначена для предоставления средств работы с диском. Под такое понимание подходят узкоспециализированные системы, такие как Novell Netware или, например, программное обеспечение маршрутизаторов Cisco.

2. Системы, способные предоставят сетевые услуги. Под такое определение подходят практически все современные ОС общего назначения.

2.5. Операционные системы общего назначения

К этому классу относятся системы, берущие на себя выполнение всех ранее перечисленных функций (за исключением, может быть, поддержки режима реального времени) и рассчитанные на интерактивную работу одного или нескольких пользователей в режиме разделения времени, при не очень жестких требованиях на время реакции системы на внешние события. Как правило, в таких системах уделяется большое внимание защите самой системы, программного обеспечения и пользовательских данных от ошибочных и злонамеренных действий программ и пользователей. Обычно такие системы используют встроенные в архитектуру процессора средства защиты и виртуализации памяти. К этому классу относятся такие широко распространенные системы, как различные семейства *Unix*, *OS/2,* семейство *Windows NT*, хотя некоторые из них не обеспечивают одновременной работы нескольких пользователей и защиты пользователей и программ друг от друга.

ОС общего назначения могут работать на компьютерах с одним или несколькими процессорами. ОС с мультипроцессорной обработкой делятся на две категории – с асимметричной либо симметричной обработкой.

2.6. Системы с асимметричной процессорной обработкой

Операционные системы с ***асимметричной мультипроцессорной обработкой*** (asymmetric multiprocessing, ASMP) обычно выбирают для исполнения собственного кода один и тот же процессор, в то время как другие процессоры выполняют только пользовательские задачи. Так как код ОС исполняется на одном процессоре, то ОС ASMP довольно просто создать, усовершенствовав существующую однопроцессорную ОС. Особенно хорошо ОС ASMP подходят для работы на асимметричном оборудовании, например,

процессоре, к которому подключен сопроцессор, или на двух процессорах, совместно использующих не всю доступную память. Однако такую ОС трудно сделать переносимой. Аппаратура разных производителей (и даже разные версии аппаратуры одного производителя) имеет тенденцию различаться по типу и степени асимметрии. Либо производители оборудования должны ориентироваться на одну ОС, либо ОС придется постоянно переписывать для каждой аппаратной платформы.

2.7. Системы с симметричной процессорной обработкой

Системы с ***симметричной мультипроцессорной обработкой*** (symmetric multiprocessing, SMP) позволяют коду ОС выполняться на любом свободном процессоре или на всех процессорах одновременно, причем каждому из процессоров доступна вся память. Такой подход полнее реализует возможности нескольких процессоров, так как дает возможность полнее использовать процессорное время, независимо от того, какие приложения исполняются. Исполнение ОС только на одном процессоре может сильно загружать его, в то время как остальные простаивают, что уменьшит производительность системы; при увеличении числа процессоров в системе возрастает вероятность того, что узким местом станут именно действия, выполняемые ОС. Помимо равномерного распределения системной загрузки, системы SMP сокращают время простоя из-за неисправностей, так как при сбое одного процессора код ОС может исполняться на других. Наконец, поскольку симметричная аппаратура реализуется разными производителями сходным образом, имеется возможность создания переносимой ОС SMP.

Лекция № 3. ПРОЦЕССЫ

Определение процесса

Процесс – это выполняемая программа вместе с текущими значениями счетчика команд, регистров и переменных.

Концепция процесса является основополагающей для любой операционной системы. Она предполагает два аспекта: во-первых, процесс является носителем данных и, во-вторых, он собственно и выполняет операции, связанные с обработкой этих данных.

В качестве примеров процессов можно назвать прикладные программы пользователей, утилиты и другие системные обрабатываемые программы. Процессом может быть редактирование какого-либо текста, трансляция исходной программы, ее компоновка, исполнение. Причем, трансляция какой-нибудь исходной программы является одним процессом, а трансляция следующей исходной программы – другим процессом, поскольку транслятор как объединение программных модулей здесь выступает как одна и та же программа, но данные, которые он обрабатывает, являются разными.

При исполнении программ на процессоре чаще всего различают следующие характерные отдельные состояния:

- ***порождение*** – подготовка для первого исполнения на процессоре;
- ***активное состояние***, или состояние “Счет” – программа исполняется на процессоре;
- ***ожидание*** – программа не исполняется на процессоре по причине занятости какого-либо требуемого ресурса;
- ***готовность*** – программа не исполняется, но для исполнения предоставлены все необходимые в текущий момент ресурсы, кроме центрального процессора;
- ***окончание*** – нормальное или аварийное окончание исполнения программы, после которого процессор и другие ресурсы ей не предоставляются.

Процесс находится в каждом из своих допустимых состояний в течение некоторого времени, после чего переходит в какое-то другое допустимое состояние. Состав допустимых состояний, а также допустимые переходы из состояния в состояние обычно задают в форме графа существования процесса, пример которого изображен на рис. 3.1:

Рис. 3.1. Граф существования процесса

Для ОС процесс в такой трактовке рассматривается как объект, в отношении которого требуется обеспечить реализацию каждого из допустимых состояний, а также допустимые переходы из состояния в состояние в ответ на события, которые могут явиться причиной таких переходов. Эти события могут инициироваться и самими процессами, которые способны затребовать процессор или какой-либо другой ресурс, необходимый для исполнения программы.

Классификация процессов

Процессы определяются рядом временных характеристик. В некоторый момент времени процесс может быть порожден (образован), а через некоторое время закончен. Интервал между этими моментами называют ***интервалом существования процесса***.

В момент порождения последовательность и длительность пребывания процесса в каждом из своих состояний (***трасса процесса***) в общем случае непредсказуемы. Следовательно, непредсказуема и длительность интервала

существования. Однако отдельные виды процессов требуют такого планирования, чтобы гарантировать окончание процесса до наступления некоторого конкретного момента времени. Процессы такого класса называют ***процессами реального времени***. В другой класс входят процессы, время существования которых должно быть не более интервала времени допустимой реакции компьютера на запросы пользователя. Процессы такого класса называют ***интерактивными***. Процессы, не вошедшие в эти классы, называют ***пакетными***.

В любой ОС по требованию существующего или существовавшего процесса проводится работа по порождению процессов. Процесс, задающий данное требование, называют ***порождающим***, а создаваемый по требованию – ***порожденным***. Если порожденный процесс на интервале своего существования в свою очередь выдает требование на порождение второго процесса, то он одновременно становится и порождающим.

Если интервалы двух процессов не пересекаются во времени, то такие два процесса называют ***последовательными*** друга относительно друга. Если на рассматриваемом интервале времени существуют одновременно два процесса, то они на этом интервале являются ***параллельными*** друг относительно друга. Если на рассматриваемом интервале найдется хотя бы одна точка, в которой существует один процесс, но не существует другой, и хотя бы одна точка, в которой оба процесса существуют одновременно, то такие два процесса называют ***комбинированными***.

Программные процессы принято делить на ***системные*** и ***пользовательские***. При развитии системного процесса исполняется программный код из состава операционной системы. При развитии пользовательского процесса исполняется пользовательская (прикладная) программа.

Процессы независимо от их вида могут быть ***взаимосвязанными*** или ***изолированными*** друг от друга. Два процесса являются взаимосвязанными, если между ними поддерживаются с помощью системы управления

процессами какого-либо рода связи: функциональные, пространственно-временные, управляющие, информационные и т.д. В противном случае они являются изолированными (точнее – процессами со слабыми связями, так как при отсутствии явных связей они могут быть связаны косвенно, и определенным образом влиять на развитие друг друга). Когда необходимо подчеркнуть связь между взаимосвязанными процессами по ресурсам, их называют ***конкурирующими***.

Управление взаимосвязанными процессами в составе ОС основано на контроле и удовлетворении определенных ограничений, которые накладываются на порядок выполнения таких процессов. Данные ограничения определяют ***виды отношений***, которые допустимы между процессами, и составляют в совокупности синхронизирующие правила.

Отношение предшествования. Для двух процессов это отношение означает, что первый процесс должен переходить в активное состояние всегда раньше второго.

Отношение приоритетности. Процесс с приоритетом *P* может быть переведен в активное состояние только при соблюдении двух условий: в состоянии готовности к рассматриваемому процессору нет процессов с большим приоритетом; процессор либо свободен, либо используется процессом с меньшим, чем *P*, приоритетом.

Отношение взаимного исключения. Здесь два процесса используют общий ресурс. При этом совокупность действий над этим ресурсом в составе одного процесса называют критической областью. Критическая область одного процесса не должна выполняться одновременно с критической областью над этим же ресурсом в составе другого процесса.

Программные потоки

В традиционных операционных системах у каждого процесса есть адресное пространство и один поток управления. Тем не менее, нередко возникают ситуации, в которых желательно иметь несколько потоков

управления, квазипараллельно выполняющихся в одном адресном пространстве так, как будто они представляют собой отдельные процессы (за исключением общности адресного пространства). Такие потоки управления называются ***программными потоками***, или ***легковесными процессами***.

На рис. 3.2.а представлены три обычных процесса. Каждый из них имеет собственное адресное пространство и единственный поток управления. На рис. 3.2.б представлен один процесс – с тремя потоками управления. В обоих случаях присутствуют три программных потока, но в первом случае каждый из них имеет собственное адресное пространство, а во втором – программные потоки имеют общую память.

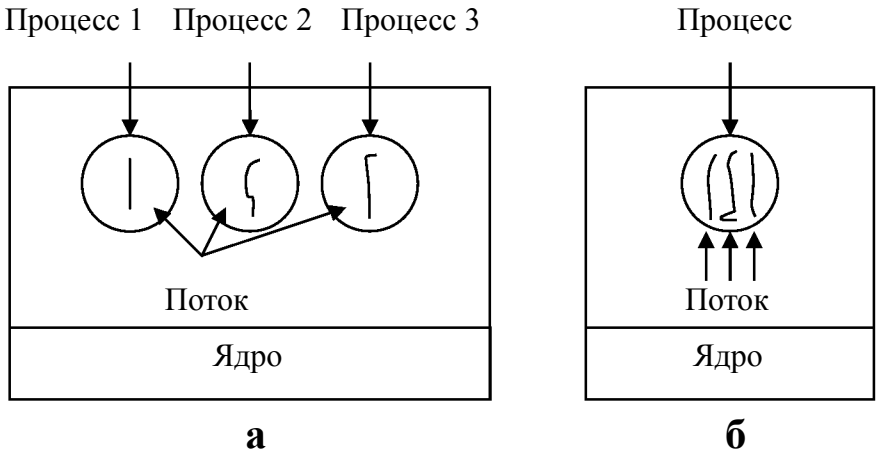

Рис. 3.2. Модель программных потоков: **а** – три процесса с собственным программным потоком каждый; **б** – один процесс с тремя потоками

В качестве примера приложения, рассчитанного на многопоточность, можно привести веб-браузер. Веб-страницы часто содержат множество картинок небольшого размера. Для загрузки каждой из них браузер должен установить отдельное соединение с сайтом. Установка и разрыв соединения отнимают много времени. При поддержке браузером многопоточности можно загружать несколько картинок одновременно, что значительно ускоряет загрузку страницы. Для небольших изображений установка соединений занимает гораздо больше времени, чем передача данных.

Лекция № 4. УПРАВЛЕНИЕ РЕСУРСАМИ

4.1. Понятие ресурса

Концепция процесса преследует цель выработки механизма распределения и управления ресурсами. При разработке первых систем ресурсами считались:

- ***процессорное время***,
- ***память***,
- ***каналы ввода-вывода***,
- ***периферийные устройства***.

Однако очень скоро понятие ресурса стало гораздо более универсальным и общим. Различного рода программные и информационные ресурсы также могут быть определены для системы как объекты, которые могут распределяться, и доступ к которым необходимо соответствующим образом контролировать.

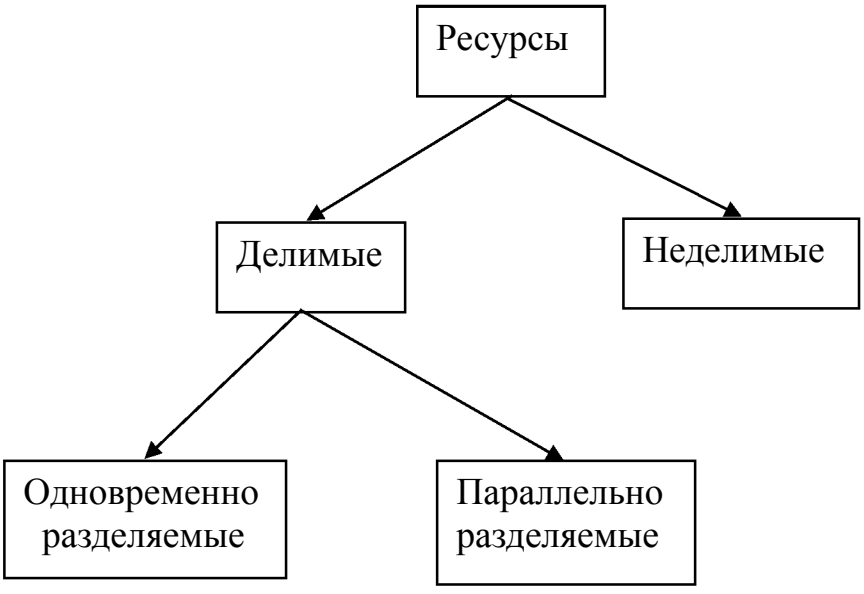

Рис. 4.1. Классификация ресурсов

В настоящее время понятие ресурса превратилось в абстрактную структуру с целым рядом атрибутов, характеризующих способы доступа к

этой структуре и ее физическое представление в системе. Ресурсами стали называть и такие объекты, как сообщения и синхросигналы, которыми обмениваются задачи.

Ресурсы могут быть ***разделяемыми***, когда несколько процессов используют их ***одновременно*** (в один и тот же момент времени) или ***параллельно*** (попеременно в течение некоторого интервала времени), а могут быть и ***неделимыми*** (рис. 4.1).

4.2. Способы управления памятью

Часть операционной системы, отвечающая за управление памятью, называется ***менеджером памяти***. В самой простой однозадачной системе в каждый конкретный момент времени работает только одна программа, при этом память разделяется между программой и операционной системой. Как только пользователь набирает команду, операционная система копирует запрашиваемую программу с диска в память и выполняет ее, а после окончания процесса выводит на экран приглашение и ждет новой команды. Получив инструкции, она загружает другую программу в память, записывая ее поверх предыдущей.

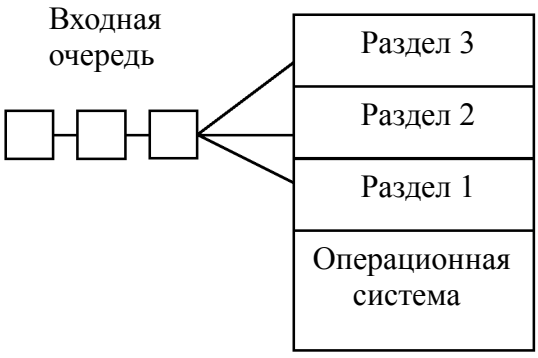

Рис. 4.2. Фиксированные разделы памяти с одной входной очередью

Многозадачность вносит проблему переадресации. Во время компоновки программы компоновщик должен знать, с какого адреса будет

начинаться программа в памяти. На рис. 4.2 показаны разделы памяти, используемые различными процессами, стоящими на очередь для обработки в пакетном режиме. Допустим, что в данном примере операционная система занимает адресное пространство 0-100К в памяти компьютера.

Предположим, что первая команда программы представляет собой вызов процедуры с абсолютным адресом 100 внутри двоичного файла, создаваемого компоновщиком. Если эта программа загрузится в раздел 1 (по адресу 100К), команда обратится к абсолютному адресу 100, принадлежащему операционной системе. А нужно вызвать процедуру по адресу 100К+100. Если же программа загрузится в раздел 2, команду нужно переадресовать по адресу 200К+100 и т.д. Эта проблема известна как проблема ***переадресации***.

Одним из возможных решений является модификация команд во время загрузки программы в память. В программе, загружаемой в раздел 1, к каждому адресу прибавляется значение 100К, в программе, которая попадет в раздел 2, к адресам добавляется значение 200К и т.д. Чтобы выполнить подобную переадресацию во время загрузки, компоновщик должен включить в двоичную программу список или битовую карту с информацией о том, какие слова в программе являются адресами (их нужно перераспределить), а какие – кодами машинных команд, которые не нужно изменять. Так работает операционная система OS/MFT.

В случае пакетных систем память с фиксированными разделами действует просто и эффективно. Но совершенно другая ситуация имеет место в системах разделения времени и персональных компьютерах, ориентированных на работу с графикой. Оперативной памяти иногда оказывается недостаточно для того, чтобы вместить все текущие активный процессы, и тогда избыток процессов приходится хранить на диске, а для обработки динамически переносить в память.

Существует два основных подхода к управлению памятью, зависящих от доступного аппаратного обеспечения. Самая простая стратегия,

называемая ***подкачкой*** (swapping), заключается в том, что каждый процесс полностью копируется в память, работает некоторое время и затем полностью же возвращается на диск. Другая стратегия, носящая название ***виртуальной памяти***, позволяет программам работать даже тогда, когда они только частично находятся в оперативной памяти.

В операционной системе Windows существует файл подкачки, в котором временно сохраняются процессы, для которых не хватает места в оперативной памяти компьютера. Имеется возможность настроить режим работы этого файла. Для этого необходимо открыть «Панель управления > Производительность и обслуживание > Система > Дополнительно > Быстродействие (параметры) > Дополнительно > Виртуальная память > Изменить». Если на компьютере с объемом оперативной памяти 512 Мбайт установлена операционная система Windows XP, то, в принципе, можно обойтись без файла подкачки вообще – это только повысит быстродействие системы. Если объем оперативной памяти меньше этой величины, рекомендуется установить одинаковый исходный и максимальный размер файла подкачки. Тем самым, можно снизить фрагментацию диска и избавиться от распространенной проблемы, когда из-за разросшегося файла подкачки на диске не хватает места. Теперь он будет занимать фиксированный размер, и распределить место на диске будет гораздо проще. Вполне приемлемая цифра – размер файла подкачки в 2-4 раза больше объема физической оперативной памяти компьютера.

4.3. Виртуальная память

Еще на заре компьютерной техники люди столкнулись с проблемой размещения программ, оказавшихся слишком большими и поэтому не помещавшиеся в доступной физической памяти. Обычно принималось решение о разделении программы на части, называемые ***оверлеями*** (overlays). Нулевой оверлей запускался первым. По завершению своего выполнения он вызывал следующий оверлей. Некоторые оверлейные

системы были очень сложными, позволяя одновременно находиться в памяти нескольким оверлеям. Оверлеи хранились на диске и по мере необходимости динамически перемещались между памятью и диском средствами операционной системы.

Хотя фактическая работа по загрузке оверлеев с диска и выгрузке на диск выполнялась системой, делить программы на части должен был программист. Однако такая ситуация длилась недолго, так как вскоре удалось поручить всю работу компьютеру.

Разработанный подход стал известен как ***виртуальная память***. Основная идея этого подхода состоит в том, что хотя общий размер программы, данных и стека может превышать объем доступной физической памяти, операционная система хранит части программы, используемые в настоящий момент, в оперативной памяти, остальные – на диске.

Например, программа размером 512 Мбайт сможет работать на машине с объемом памяти 256 Мбайт, если тщательно продумать, какие 256 Мбайт должны находиться в памяти в каждый момент времени. При этом по мере необходимости части программы, находящиеся на диске, будут меняться местами с частями в памяти.

Виртуальная память вполне работоспособна и в многозадачной системе при наличии множества одновременно загружаемых в память программ. Когда программа ждет перемещения в память очередной ее части, она находится в состоянии ожидания ввода-вывода и не работает, поэтому центральный процессор может быть отдан другому процессу тем же самым способом, как в любой другой многозадачной системе.

Адреса виртуальной памяти формируются программным путем с использованием для этой цели специальных регистров (базовых или сегментных). Программно формируемые адреса, называемые ***виртуальными***, образуют ***виртуальное адресное пространство***. На компьютерах без виртуальной памяти виртуальные адреса подаются непосредственно на шину памяти и при чтении или записи читается или записывается слово в

физической памяти с тем же самым адресом. При применении виртуальной памяти виртуальные адреса не передаются напрямую шиной памяти. Вместо этого они направляются в ***блок управления памятью*** (Memory Management Unit, MMU), который отображает виртуальные адреса на физические адреса (рис. 4.3).

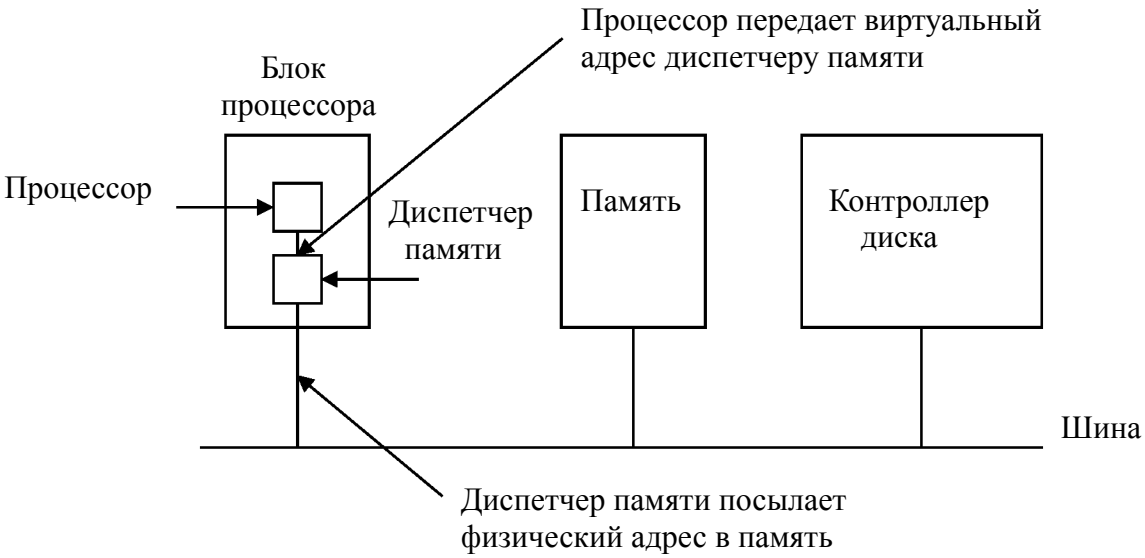

Рис. 4.3. Расположение и функции блока управления памятью (MMU)

Большинство систем виртуальной памяти опираются на прием, называемый ***замещение страниц*** (paging). Пространство виртуальных адресов разделено на единичные блоки, называемые страницами. Соответствующие блоки в физической памяти называются страничными блоками (page frame). Размер страниц и их блоков всегда одинаков. Используются размеры от 512 байт до 64 Кбайт. Передача данных между ОЗУ и диском всегда происходит постранично.

4.4. Управление устройствами ввода-вывода

Одна из важнейших функций операционной системы состоит в управлении всеми устройствами ввода-вывода компьютера. Операционная система должна давать этим устройствам команды, перехватывать прерывания и обрабатывать ошибки. Она должна также предоставить

простой и удобный интерфейс между устройствами и остальной частью системы. Интерфейс, насколько это возможно, должен быть одинаковым для всех устройств.

Устройства ввода-вывода можно грубо разделить на две категории: ***блочные*** и ***символьные***. Блочными называются устройства, хранящие информацию в виде адресуемых блоков фиксированного размера. Обычно размеры блоков варьируются от 512 до 32 768 байт. Каждый блок может быть прочитан независимо от остальных блоков. Наиболее распространенными блочными устройствами являются диски.

Символьное устройство принимает или предоставляет поток символов без какой-либо блочной структуры. Символьное устройство не является адресуемым и не выполняет операцию поиска. Принтеры, сетевые интерфейсные адаптеры, мыши и большинство других устройств, не похожих на диски, можно рассматривать как символьные устройства.

Устройства ввода-вывода, как правило, состоят из механических и электронных компонентов. Электронный компонент называется ***контроллером устройства***, или ***адаптером***. В персональных компьютерах он обычно имеет вид печатной платы, вставляемый в слот расширения. Механический компонент – это само устройство. Плата контроллера снабжается разъемом, к которому подключается кабель, ведущий к самому устройству.

Операционная система практически всегда имеет дело с контроллером, а не с самим устройством. У большинства небольших компьютеров взаимодействие с устройствами организуется по модели единой шины. У больших машин, мэйнфреймов, применяется другая модель с несколькими шинами, которые обслуживаются специализированными компьютерами ввода-вывода, называемыми ***каналами ввода-вывода***. Такая организация позволяет снизить нагрузку на основной процессор.

У каждого контроллера есть несколько регистров, с помощью которых к ним может обращаться центральный процессор. Записывая в эти регистры

определенные значения, операционная система посылает устройству команды передачи и приема данных, включения, отключения и др. Считывание регистров устройства позволяет определить его состояние, готовность принять команду и т. д.

В дополнение к регистрам управления многие устройства имеют буфер данных, доступный для чтения и записи со стороны операционной системы. Например, отображение пикселей на экране в большинстве компьютеров осуществляется с помощью видеопамяти, которая представляет собой такой буфер.

Процессор взаимодействует с регистрами управления и буферами данных устройств двумя способами. Первый предполагает назначение каждому регистру номера ***порта ввода-вывода*** – 8-ми или 16-ти разрядного числа. В других компьютерах регистры ввода-вывода являются частью обычного адресного пространства памяти. Такая организация называется ***вводом-выводом с отображением на память***. Она была впервые применена в мини-компьютере PDP-11. Каждому регистру управления назначается уникальный адрес памяти, с которым обычная память не связана. В компьютерах с процессором Pentium диапазон адресов от 640 Кбайт до 1 Мбайт зарезервирован под буферы данных устройств, а область портов ввода-вывода занимает первые 64 Кбайт.

Процессор, желающий считать слово из регистра, выставляет его адрес на адресные линии шины, а затем выдает сигнал чтения по линии управления. Для того чтобы отличить обращение к пространству ввода-вывода от обращения к памяти, требуется вторая сигнальная линия.

Для управления каждым устройством ввода-вывода, подключенным к компьютеру, требуется специальная программа. Эта программа, называемая ***драйвером устройства***, часто пишется производителем устройства и распространяется на компакт-дисках вместе с самим устройством. Поскольку для каждой операционной системы требуются специальные драйверы,

производители обычно поставляют драйверы для нескольких наиболее популярных операционных систем.

4.5. Прерывания

Как правило, регистры контроллеров содержат один или несколько битов состояния. Их можно проверить и определить, завершена ли операция вывода и имеются ли новые данные в устройстве ввода. В дополнение к битам состояния, многие контроллеры часто используют прерывания, которые позволяют сообщить процессору, что регистры готовы для записи или чтения.

Прерывание – это принудительная передача управления от выполняемой программы к операционной системе (а через нее – к соответствующей программе обработки прерываний), происходящая при возникновении определенного события. Механизм прерываний реализуется аппаратно-программными средствами. Прерывание непременно влечет за собой изменение порядка выполнения команд процессором.

Прерывания осуществляются с помощью контроллера прерываний. Количество входов этого контроллера ограничено. Например, у персональных компьютеров Pentium только 15 линий прерывания доступны для устройств ввода-вывода. Некоторые из контроллеров устаревших компьютеров встроены в материнскую плату, как, например, контроллер клавиатуры на IBM PC. У тех контроллеров, что вставляются в разъем на объединительной плате, установить соответствие между IRQ сигналом и устройством иногда можно при помощи перемычек или переключателей. Если пользователь приобретал новую карту, он был вынужден вручную устанавливать линию прерывания, чтобы избежать ее конфликта с существующими устройствами. Большинство пользователей совершало в этом ошибки, что, в конечном счете, привело к появлению механизма ***автоконфигурирования*** (Plug and Play), благодаря которому BIOS

самостоятельно назначает устройствам корректные линии прерывания на этапе загрузки системы.

Главные функции механизма прерываний следующие:

1. распознавание прерываний;
2. передача управления соответствующему обработчику прерываний;
3. корректное возвращение к прерванной программе.

Прерывания бывают внешние (асинхронные) и внутренние (синхронные).

Внешние прерывания – это:

1. прерывания от таймера;
2. прерывания от внешних устройств (прерывания по вводу-выводу);
3. прерывания по нарушению питания;
4. прерывания с пульта оператора вычислительной системы;
5. прерывания от другого процессора или другой вычислительной системы.

Внутренние прерывания вызываются событиями, которые связаны с работой процессора и являются синхронными с его операциями. Примерами являются следующие запросы на прерывания:

1. при нарушении адресации (в адресной части выполняемой команды указан запрещенный или несуществующий адрес, обращение к отсутствующему сегменту или странице при организации механизмов виртуальной памяти);
2. при наличии в поле кода операции незадействованной двоичной комбинации;
3. при делении на ноль;
4. вследствие переполнения или исчезновения порядка;
5. от средств контроля (например, вследствие обнаружения ошибки четности, ошибок в работе различных устройств).

Могут еще существовать прерывания в связи с попыткой выполнить команду, которая сейчас запрещена. Во многих компьютерах часть команд должна выполняться только кодом самой операционной системы, но не

прикладными программами. Это делается с целью повышения защищенности выполняемых на компьютере вычислений. Соответственно в аппаратуре предусмотрены различные режимы работы, и пользовательские программы выполняются в режиме, в котором некоторое подмножество команд, называемых привилегированными, не исполняется. К привилегированным командам помимо команд ввода-вывода относятся и команды переключения режима работы центрального процессора, и команды инициализации некоторых системных регистров процессора. При попытке использовать команду, запрещенную в данном режиме, происходит внутреннее прерывание, и управление передается самой операционной системе.

Наконец, существуют собственно ***программные прерывания***. Эти прерывания происходят по соответствующей команде прерываний, то есть по этой команде процессор осуществляет практически те же действия, что и при обычных внутренних прерываниях. Этот механизм был специально введен для того, чтобы переключение на системные программные модули происходило не просто как переход на подпрограмму, а точно таким же образом, как и обычное прерывание.

Наличие сигнала прерывания не обязательно должно вызывать прерывание исполняющей программы. Процессор может обладать средствами защиты от прерываний: ***отключение*** системы прерываний, ***маскирование*** (запрет) отдельных сигналов прерывания.

Чтобы обработать сигналы прерывания в разумном порядке, им (как уже отмечалось) присваиваются приоритеты. Сигнал с более высоким приоритетом обрабатывается в первую очередь, обработка остальных сигналов прерывания откладывается.

Лекция № 5. ФАЙЛОВЫЕ СИСТЕМЫ

Введение

Часть операционной системы, работающая с файлами, называется ***файловой системой***. Она предоставляет пользователю следующие возможности.

- создание, удаление, переименование (и другие операции) именованных наборов данных (файлов) из своих программ или посредством специальных управляющих программ, реализующих функции интерфейса пользователя с его данными и активно использующих систему управления файлами;
- работа с недисковыми периферийными устройствами как файлами;
- обмен данными между файлами, между устройствами, между файлом и устройством (и наоборот);
- работа с файлами путем обращений к программным модулям системы управления файлами (часть API ориентирована именно на работу с файлами);
- защита файлов от несанкционированного доступа.

Как правило, все современные операционные системы имеют соответствующие файловые системы. А некоторые операционные системы имеют возможность работать с несколькими файловыми системами. В этом случае говорят о ***монтируемых файловых системах***.

Операционные системы персональных компьютеров MS DOC и Windows имели файловую систему FAT (File Allocation Table – таблица размещения файлов). Разработка новой операционной системы Windows NT привела к появлению новой файловой системы, названной NTFS (New Technology File System – файловая система новой технологии).

Файловая система FAT

Эта файловая система получила свое название благодаря простой таблице, в которой указываются:

- непосредственно адресуемые участки логического диска, отведенные для размещения в них файлов или их фрагментов;
- свободные области дискового пространства;
- дефектные области диска (эти области содержат дефектные участки и не гарантируют чтение и запись данных без ошибок).

В файловой системе FAT дисковое пространство любого логического диска делится на две части: ***системную область*** и ***область данных***.

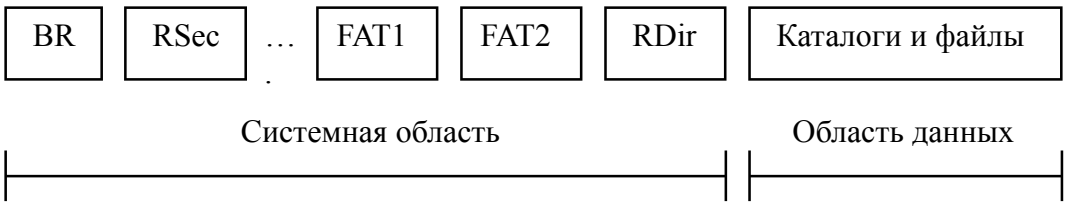

Рис. 5.1. Структура логического диска в FAT

Системная область логического диска создается и инициализируется при форматировании, а в последующем обновляется при работе с файловой структурой. Область данных логического диска содержит обычные файлы и файлы-каталоги, эти объекты образуют иерархию, починенную корневому каталогу. Элемент каталога описывает файловый объект, который может быть либо обычным файлом, либо файлом-каталогом. Область данных, в отличие от системной области, доступна через пользовательский интерфейс операционной системы. Системная область состоит из следующих компонентов (расположенных в логическом адресном пространстве друг за другом):

- загрузочной записи (Boot Record, BR);
- зарезервированных секторов (Reserved Sectors, ResSec);

- таблицы размещения файлов (File Allocation Table, FAT);
- корневого каталога (Root Directory, RDir).

Таблица размещения файлов является очень важной информационной структурой. Можно сказать, что она представляет собой адресную карту области данных, в которой описывается и состояние каждого участка области данных, и принадлежность его к тому или иному файловому объекту.

Всю область данных разбивают на так называемые ***кластеры***. Кластер представляет собой один или несколько смежных секторов в логическом дисковом адресном пространстве (точнее – только в области данных). Кластер – это минимальная адресуемая единица дисковой памяти, выделяемая файлу (или некорневому каталогу). Кластеры введены для того, чтобы уменьшить количество адресуемых единиц в области данных логического диска.

Каждый файл занимает целое число кластеров. Последний кластер при этом может быть задействован не полностью, что при большом размере кластера может приводить к заметной потере дискового пространства. На дискетах кластер занимает один или два сектора, а на жестких дисках его размер его размер зависит от объема раздела. В таблице FAT кластеры, принадлежащие одному файлу (или файлу-каталогу), связываются в цепочки. Для указания номера кластера в файловой системе FAT16 используется 16-ти разрядное слово, следовательно, можно иметь до $2^{16} = 65\,536$ кластеров (с номерами от 0 до 65 535).

В Windows NT/2000/XP разделы файловой системы FAT могут иметь размер до 4097 Мбайт. В этом случае кластер будет объединять уже 128 секторов.

Логическое разбиение области данных на кластеры как совокупности секторов взамен использования одиночных секторов имеет следующий смысл:

- прежде всего, уменьшается размер самой таблицы FAT;

- уменьшается возможная фрагментация файлов;
- ускоряется доступ к файлу, так как в несколько раз сокращается длина цепочек фрагментов дискового пространства, выделенных для него.

Однако слишком большой размер кластера ведет к неэффективному использованию области данных, особенно в случае большого количества маленьких файлов.

При выделении нового кластера для записи файла берется первый свободный кластер. Поскольку файлы на диске изменяются, то это приводит к ***фрагментации*** файлов, при которой данные одного файла могут размещаться не в смежных кластерах, а порой в очень удаленных друг от друга, образуя сложные цепочки. Это приводит к существенному замедлению при работе с файлами.

В связи с тем, что таблица FAT используется при доступе к диску очень интенсивно, она обычно загружается в оперативную память (в буферы ввода-вывода или в кэш) и остается там настолько долго, насколько это возможно. Если таблица большая, а файловый кэш, напротив, относительно небольшой, в памяти размещаются только фрагменты этой таблицы, к которым обращались последнее время.

В связи с чрезвычайной важностью таблицы FAT она обычно хранится в двух идентичных экземплярах, второй из которых непосредственно следует за первым. Обновляются копии FAT одновременно, используется же только первый экземпляр. Если он по каким-либо причинам окажется разрушенным, то произойдет обращение ко второму экземпляру.

Файловая система NTFS

При проектировании NTFS особое внимание было уделено надежности, механизмам ограничения доступа к файлам и каталогам, расширенной функциональности, поддержке дисков большого объема и пр. Одним из основных понятий, используемых при работе с NTFS, является понятие

тома (volume). Том означает логическое дисковое пространство, которое может быть воспринято как логический диск, то есть том может иметь буквенный идентификатор диска.

Что касается надежности, система NTFS обладает определенными средствами самовосстановления. Она поддерживает различные механизмы проверки целостности системы, включая ведение журналов транзакций, позволяющих воспроизвести файловые операции записи по специальному системному журналу.

Поскольку NTFS разрабатывалась как файловая система для серверов, для которых очень важно обеспечить бесперебойную работу без перезагрузок, в ней для повышения надежности был введен механизм аварийной замены дефектных секторов резервными.

Файловая система NTFS поддерживает объектную модель безопасности операционной системы Windows NT и рассматривает все тома, каталоги и файлы как самостоятельные объекты. Система NTFS обеспечивает безопасность на уровне файлов и каталогов. Это означает, что разрешения доступа к томам, каталогам и файлам могут зависеть от учетной записи пользователя и тех групп, к которым он принадлежит.

Система NTFS создавалась с расчетом на работу с большими дисками. Она достаточно хорошо проявляет себя при работе с томами объемом 100 Гбайт и выше. Чем больше объем диска и чем больше на нем файлов, тем больший выигрыш мы получаем, используя NTFS вместо FAT16 или FAT32. Максимально возможные размеры тома (и размеры файла) составляют 16 Эбайт (один экзабайт равен 2^{64} байт, или приблизительно 16 000 млрд. гигабайт), в то время как при работе под Windows NT/2000/XP диск с FAT16 не может иметь размер более 4 Гбайт, а с FAT32 – 32 Гбайт.

Количество файлов в корневом и некорневом каталогах (в NTFS они называются «folders» – «папками») не ограничено. Поскольку в основу структуры папок заложена эффективная структура данных, называемая «двоичным деревом», время поиска файлов в NTFS не связано линейной

зависимостью с их количеством (в отличие от систем на базе FAT). Наконец, помимо немыслимых размеров томов и файлов, система NTFS также обладает встроенными средствами сжатия, что позволяет экономить дисковое пространство и размещать в нем больше файлов. Сжатие можно применять, как к отдельным файлам, так и целым папкам и даже томам.

Как и многие другие файловые системы, NTFS делит все полезное дисковое пространство тома на кластеры – блоки данных, адресуемые как единицы данных. Файловая система NTFS поддерживает размеры кластеров от 512 байт до 64 Кбайт; неким стандартом считается кластер размером 2 или 4 Кбайт. При увеличении размера кластера свыше 4 Кбайт становится невозможным сжимать файлы и папки.

Все дисковое пространство в NTFS делится на две неравные части. Первые 12 % диска отводятся под так называемую зону MFT (Master File Table – главная таблица файлов). Эта зона предназначена для таблицы MFT (с учетом ее будущего роста), представляющей собой специальный файл со служебной информацией, позволяющей определять местонахождение всех остальных файлов. Запись каких-либо данных в зону MFT невозможна – она всегда остается пустой, чтобы при росте MFT по возможности не было фрагментации. Остальные 88 % тома представляют собой обычное пространство для хранения файлов.

Таблица MFT поделена на записи фиксированного размера в 1 Кбайт, и каждая запись соответствует какому-либо файлу. Первые 16 файлов носят служебный характер и недоступны через интерфейс операционной системы – они называются метафайлами, причем самый первый ***метафайл*** – это сам MFT. Часть диска с метафайлами – единственная часть диска, имеющая строго фиксированное положение. Копия же 16 записей таблицы MFT хранится в середине тома – для надежности, поскольку они очень важны.

Лекция № 6. АРХИТЕКТУРА ОПЕРАЦИОННЫХ СИСТЕМ

6.1. Принципы построения операционных систем

Наиболее важными принципами, закладываемыми в основу построения операционных систем, являются следующие: принцип модульности, принцип виртуализации, принцип мобильности (переносимости), принцип совместимости, принцип открытости, принцип генерации операционной системы из программных компонентов. Необходимо отметить, что не все из перечисленных принципов реализованы в существующих операционных системах.

- ***Принцип модульности***. Операционная система строится из множества программных модулей. Под ***модулем*** понимают функционально законченный элемент системы, выполненный в соответствии с принятыми межмодульными интерфейсами. По своему определению модуль предполагает легкий способ его замены другим при необходимости. Принцип модульности отражает технологические и эксплуатационные свойства системы. Наибольший эффект его использования достижим в случае, когда принцип распространен одновременно на операционную систему, прикладные программы и аппаратуру. Принцип модульности является одним из основных в UNIX-системах.
- ***Принцип виртуализации***. Любая операционная система, являясь средством распределения ресурсов и организуя по определенным правилам управление процессами, скрывает от пользователя и его приложений реальные аппаратные и иные ресурсы, заменяя их некоторой абстракцией. Операционная система существенно изменяет наши представления о компьютере. Она виртуализирует его, добавляя ему функциональности, удобства управления, предоставляя средства организации параллельных вычислений и т.д. Именно благодаря операционной системе мы воспринимаем компьютер совершенно

иначе, чем без нее. Одним из аспектов принципа виртуализации является независимость программ от внешних устройств. Связь программы с этими устройствами производится не в процессе ее создания, а в период планирования исполнения. В результате перекомпиляция программы при работе с новым устройством не требуется.

- ***Принцип мобильности***. Мобильность означает возможность легкого переноса операционной системы на другую аппаратную платформу. Мобильная операционная система обычно разрабатывается с помощью специального языка высокого уровня, предназначенного для создания системного программного обеспечения. Одним из таких языков является язык С, который был специально создан для того, чтобы написать на нем очередную версию операционной системы UNIX. В последние годы язык С++ также стал использоваться для этих целей, поскольку идеи объектно-ориентированного программирования оказались плодотворными не только для прикладного, но и для системного программирования.
- ***Принцип совместимости***. Соблюдение этого принципа гарантирует способность операционной системы выполнять программы, написанные для других систем или для более ранних версий данной операционной системы, а также для другой аппаратной платформы.
- ***Принцип открытости***. Этот принцип иногда трактуют как принцип расширяемости системы. Открытая операционная система доступна для анализа как пользователям, так и системным специалистам. Прекрасные возможности для расширения ОС предоставляет подход к структурированию операционной системы по типу клиент-сервер с использованием микроядерной технологии. В соответствии с этим подходом операционная система строится как совокупность привилегированной управляющей программы и набора непривилегированных служб – «серверов». Основная часть

операционной системы может оставаться неизменной, в то время как добавляются новые службы или изменяются старые. К открытым ОС прежде всего следует отнести UNIX-системы и Linux.

- ***Принцип генерируемости.*** Согласно этому принципу исходное представление ядра системы должно обеспечивать возможность настройки, исходя из конкретной конфигурации вычислительного центра и круга решаемых задач. Под генерацией ОС понимается ее сборка из отдельных программных модулей. Процесс генерации осуществляется с помощью специальной программы-генератора. В наши дни при использовании персональных компьютеров с принципом генерируемости можно столкнуться разве что при работе с Linux. В этой системе имеется возможность не только использовать какое-либо готовое ядро, но и самому сгенерировать (скомпилировать) такое ядро, которое будет оптимальным для данного конкретного персонального компьютера и решаемых на нем задач. В остальных ОС конфигурирование системы под соответствующий состав оборудования осуществляется на этапе установки, причем в большинстве случаев не представляется возможным серьезно вмешаться в этот процесс.

6.2. Интерфейсы операционных систем

Операционная система всегда выступает как интерфейс между аппаратурой компьютера и пользователем с его задачами. Автор монографии «Системное программирование в среде Win32» Джонсон М. Харт на восемнадцатой странице пишет: «В соответствии с принятой в данной книге точке зрения Windows – это всего лишь API операционной системы, предоставляющий набор вполне понятных средств».

Интерфейсы системного и прикладного программирования (API) предназначены для выполнения следующих задач:

1. ***Управление процессами, которое включает в себя следующий набор основных функций***:
    1.1. запуск, приостанов и снятие задачи с выполнения;
    1.2. задание или изменение приоритета задачи;
    1.3. взаимодействие задач между собой;
    1.4. вызов удаленных процедур (Remote Procedure Call).
2. ***Управление памятью***:
    2.1. запрос на выделение блока памяти;
    2.2. освобождение памяти;
    2.3. изменение параметров блока памяти (например, память может быть заблокирована процессом либо предоставлена в общий доступ);
    2.4. отображение файлов на память (имеется не во всех системах).
3. ***Управление вводом-выводом***:
    3.1. запрос на управление виртуальными устройствами (напомним, что управление вводом-выводом является привилегированной функцией самой операционной системы, и никакая из пользовательских задач не должна иметь возможности непосредственно управлять устройствами);
    3.2. файловые операции (запросы к системе управления файлами на создание, изменение и удаление данных, организованных в файлы).

В последние годы большую популярность получили графические интерфейсы (Graphical User Interface, GUI), в которых задействованы манипуляторы типа «мышь». Указание курсором на объект и щелчок на кнопке мыши приводит к каким-либо действиям. Можно сказать, что такая интерфейсная подсистема транслирует «команды» пользователя в обращения к операционной системе.

Управление GUI является частным случаем задачи управления вводом-выводом и не относится к функциям ядра операционной системы, хотя в ряде

случаев разработчики ОС относят функции GUI к основному системному интерфейсу API.

Общий термин API (Application Program Interface – интерфейс прикладного программирования) можно разделить на следующие направления:

- API как интерфейс высокого уровня, принадлежащий к библиотекам RTL (Run Time Library);
- API прикладных и системных программ, входящих в поставку операционной системы;
- Прочие интерфейсы API.

При реализации функций API на уровне системы программирования эти функции предоставляются пользователю в виде библиотеки функций соответствующего языка программирования. Обычно речь идет о библиотеке времени исполнения (RTL). Очевидно, что эффективность вызова функций API в таком варианте будет несколько ниже, чем при непосредственном обращении к функциям операционной системы. Так происходит, поскольку для выполнения многих функций API библиотека RTL должна все равно выполнять обращения к функциям операционной системы.

Однако переносимость исходного кода программы в таком варианте оказывается самой высокой, поскольку синтаксис и семантика всех функций зависят от языка программирования и не зависят от архитектуры вычислительной системы.

При реализации функций API на уровне модулей операционной системы результирующая программа обращается непосредственно к операционной системе. Поэтому достигается наибольшая эффективность выполнения функций API по сравнению со всеми другими вариантами реализации API. Недостатком такой схемы является практически полное отсутствие переносимости не только кода результирующей программы, но и кода исходной программы.

При реализации функций API с помощью внешних библиотек эти функции предоставляются пользователю в виде библиотеки процедур и функций, созданной сторонним разработчиком. Система программирования ответственна только за то, чтобы подключить объектный код библиотеки к результирующей программе. С точки зрения эффективности выполнения этот метод реализации API имеет самые низкие результаты, поскольку внешняя библиотека обращается как к функциям операционной системы, так и к функциям RTL языка программирования.

Если говорить о переносимости исходного кода, то здесь требование только одно – используемая внешняя библиотека должна быть доступна в любой из архитектур вычислительных систем, на которые ориентирована прикладная программа. Тогда удается достигнуть переносимости. Это возможно, если внешняя библиотека удовлетворяет какому-либо принятому стандарту, а система программирования поддерживает этот стандарт.

Например, библиотеки, удовлетворяющие стандарту POSIX, доступны в большинстве систем программирования для языка C. И если прикладная программа использует только библиотеки этого стандарта, то ее исходный код будет переносимым. Еще одним примером является широко известная библиотека графического интерфейса XLib, поддерживающая стандарт графической среды X-Windows.

Стандарт POSIX (Portable Operating System Interface for Computer Environments – независимый от платформы системный интерфейс для компьютерного окружения) – это стандарт IEEE (Institute of Electrical and Electronics Engineers – институт инженеров по электротехнике и радиоэлектронике), описывающий системные интерфейсы для открытых операционных систем, в том числе оболочки, утилиты и инструментарии. Стандарт базируется на UNIX-системах, но допускает реализацию и в других операционных системах.

Лекция № 7. СРАВНЕНИЕ ОПЕРАЦИОННЫХ СИСТЕМ

7.1. Схема исторических связей между операционными системами

На рис. 7.1. отражены генетические связи между наиболее известными операционными системами.

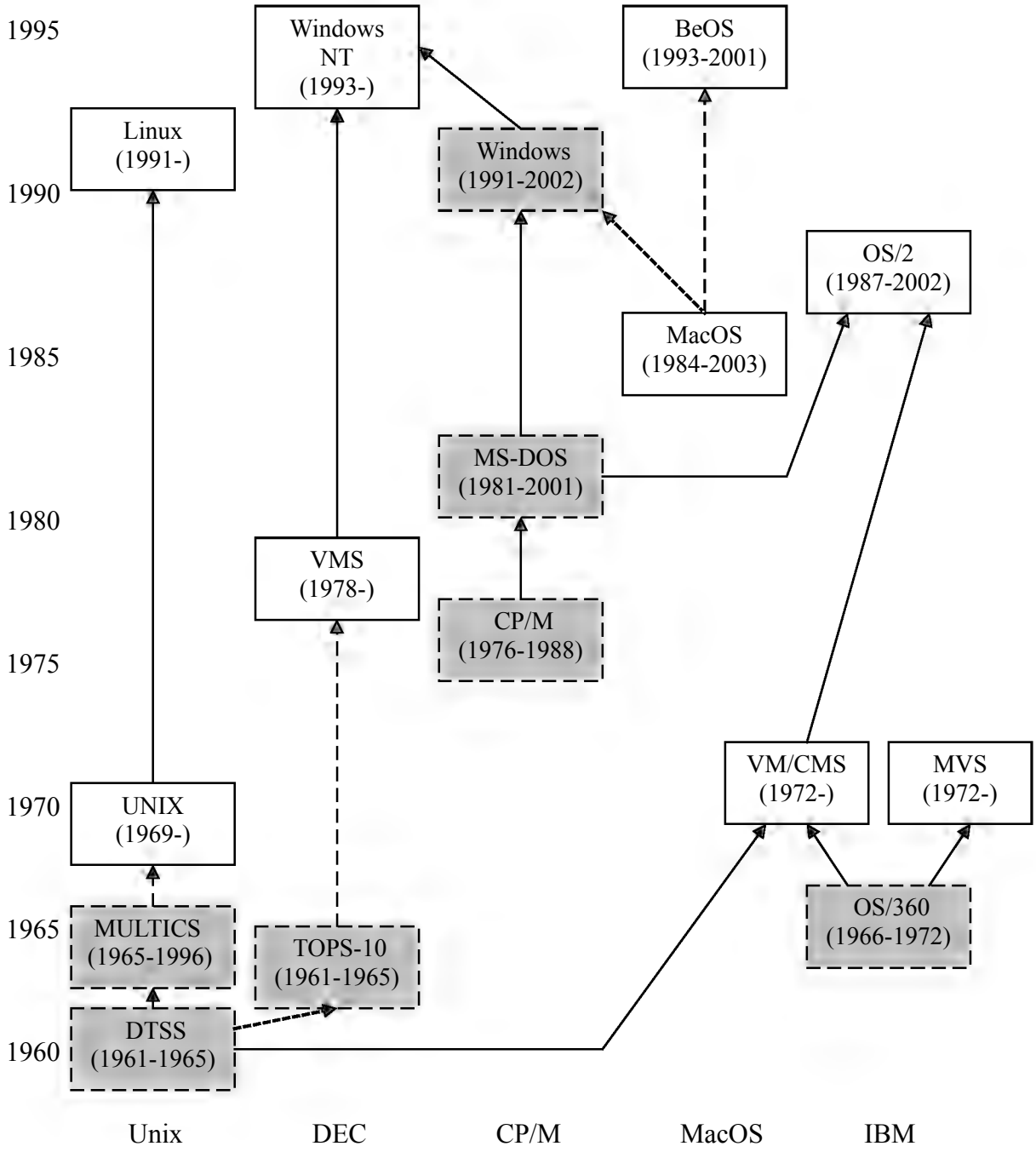

Рис. 7.1. Схема исторических связей между операционными системами

Отмеченные серым цветом операционные системы не являются системами разделения времени. Системы, названия которых обрамлены сплошными линиями, до сих пор существуют. Дата «рождения» представляет собой дату первой поставки, дата «смерти» – это дата, когда поставщик прекратил поставку системы.

Сплошные стрелки указывают на генетическую связь или очень сильное влияние дизайна (т.е. более позднюю систему с API-интерфейсом, умышленно переработанным путем обратного проектирования для соответствия более ранней системе). Штриховые линии указывают на значительное влияние конструкции, а пунктирные – наоборот, на слабое влияние конструкции.

Блок «UNIX» включает в себя все частные UNIX-системы, включая AT&T и ранние версии Berkeley Unix. Блок «Linux» включает с себя UNIX-системы с открытыми исходными кодами, каждая из которых была основана в 1991 году. Они имеют генетическую наследственность от раннего UNIX -кода, который был освобожден от частного контроля компании AT&T соглашением по судебному процессу 1993 года.

На рис. 7.1, естественно, представлены не все операционные системы. В частности здесь нет систем реального времени. Были выбраны системы разделения времени, которые либо в настоящее время, либо в прошлом составляли конкуренцию UNIX. Достойных конкурентов не много. Большинство подобных систем (MULTICS, ITS, DTSS, TOPS-10, TOPS-20, MTS, GCOS, MPE и десятки других) исчезли так давно, что почти стерлись из коллективной памяти компьютерного сообщества. Отмирают системы VMS и OS/2, MacOS поглощается UNIX-производными. Операционные системы MVS и VM/CMS были ограничены одной частной линейкой мэйнфреймов. Только Microsoft Windows остается жизнеспособным конкурентом, независимым от традиций UNIX.

7.2. Семейство операционных систем UNIX

UNIX является исключительно удачным примером реализации простой мультипрограммной и многопользовательской операционной системы. В свое время она проектировалась как инструментальная среда для разработки программного обеспечения. Своей уникальностью система UNIX обязана во многом тому обстоятельству, что была, по сути, создана всего двумя разработчиками, которые делали ее исключительно для себя и первое время использовали на мини-ЭВМ с очень скромными вычислительными ресурсами PDP-7.

Создателями UNIX являются Кен Томпсон и Денис Ритчи. В своей операционной системе они учли опыт работы над проектом сложной мультизадачной операционной системы с разделением времени MULTICS (MULTiplexed Information and Computing Service). Название новой системы UNIX произошло от аббревиатуры UNICS (Uniplexed Information and Computing Service). Позднее Ритчи описывал эту аббревиатуру как «слегка изменнический каламбур на слово MULTICS».

Первая версия этой системы занимала всего около 12 Кбайт и могла работать на компьютерах с очень небольшим объемом оперативной памяти.

Первой реальной задачей UNIX в 1971 году была поддержка того, что в наши дни назвали бы текстовым процессором, для патентного департамента Bell Laboratories (Кен Томпсон был сотрудником Bell Laboratories). Этот проект оправдывал приобретение гораздо более мощного мини-компьютера PDP-11. Руководство оставалось в счастливом неведении о том, что система обработки текста, которую создавали Томпсон и его коллеги, была инкубатором для операционной системы. Операционные системы не входили в планы Bell Laboratories – AT&T присоединилась к консорциуму MULTICS именно для того, чтобы избежать разработки собственной операционной системы.

Первоначально операционная система UNIX была написана на ассемблере, а ее приложения – на «смеси» ассемблера и интерпретируемого

языка, который назывался «B». Его преимущество заключалось в том, что он был достаточно мал для работы на компьютерах PDP-7. Однако язык «B» был недостаточно мощным для системного программирования, поэтому Денис Ритчи добавил в него типы данных и структуры, в результате чего появился язык «C». Это произошло в начале 1971 года. А в 1973 году Томпсон и Ритчи переписали UNIX на новом языке. Это был весьма смелый шаг. В то время для того, чтобы получить максимальную производительность аппаратного обеспечения, системное программирование осуществлялось на ассемблере, и сама концепция переносимой операционной системы представлялась весьма сомнительной. Только в 1979 году Ритчи смог написать: «Кажется бесспорным, что в основном успех UNIX обусловлен читаемостью, редактируемостью и переносимостью ее программного обеспечения, причем, такие позитивные характеристики, в свою очередь, являются следствием ее написания на языках высокого уровня».

Первой целью при разработке системы UNIX было стремление сохранить простоту и обойтись минимальным количеством функций. Второй целью была *общность* (начало названия системы «Uni-» можно перевести как «общий» или «единый»). Одни и те же методы и механизмы должны были использоваться во многих случаях:

- обращения к файлам, устройствам ввода-вывода и буферам межпроцессорных сообщений выполняются с помощью одних и тех же примитивов;
- одни и те же механизмы именования, присвоения альтернативных имен и защиты от несанкционированного доступа применяются и к файлам с данными, и к каталогам, и к устройствам;
- одни и те же механизмы работают в отношении программно и аппаратно инициируемых прерываний.

Третья цель заключалась в том, чтобы сложные задачи можно было решать, комбинируя существующие небольшие программы, а не разрабатывая их заново.

Наконец, четвертая цель состояла в создании мультитерминальной операционной системы с эффективными механизмами разделения не только процессорного времени, но и всех остальных ресурсов. В мультитерминальной ОС на одно из первых мест по значимости выходят вопросы защиты одних вычислительных процессов от вмешательства других вычислительных процессов.

В операционной системе UNIX имеется несколько унифицирующих идей или метафор, которые формируют ее API-интерфейсы и определяемый ими стиль разработки. Наиболее важными из них является модель, согласно которой «каждый объект является файлом», и метафора каналов (pipes). Канал представляет собой способ соединения вывода одной программы с вводом другой.

Специалисты, относящиеся к UNIX-сообществу, утверждают, что философия этой операционной системы сводится к одному железному правилу, священному «принципу KISS»: «Keep It Simple, Stupid!». По-русски это можно перевести так: «Будь проще, тупица!»

Хотя система UNIX появилась еще в 1969 году, она существует и поныне. Какие недостатки конкурентов оставили их в проигрыше?

Наиболее очевидной общей проблемой является неспособность к переносу на другие платформы. Большинство конкурентов UNIX, появившихся до 1980 года, были привязаны к какой-либо одной аппаратной платформе и исчезли вместе с ней. Единственной причиной того, что VMS просуществовала достаточно долго для того, чтобы рассматриваться здесь в качестве учебного примера, является то, что она была успешно перенесена с ее первоначального аппаратного обеспечения VAX на процессор Alpha (а в 2003 году – с Alpha на Itanium). MacOS была успешно перенесена с микросхем Motorola 68000 на Power PC в конце 80-х годов прошлого века.

Операционная система Microsoft Windows избежала данной проблемы, оказавшись в нужном месте, когда стремление превратить программное обеспечение в продукт массового потребления привело к заполнению рынка универсальными компьютерами монокультуры PC.

С 1980 года все более проявляется другой недостаток, характерный для различных операционных систем: неспособность обеспечить изящную поддержку сети. В мире распространяющихся сетей даже операционная система, спроектированная для однопользовательской работы, нуждается в многопользовательских средствах (множество групп привилегий), поскольку без таких средств любая сетевая транзакция, которая способна обманным путем вынудить пользователя запустить злонамеренный код, может разрушить всю систему.

Windows, имея серьезные недостатки в данных областях, избегает упадка только благодаря тому, что она получила монопольное положение еще до того, как сетевое взаимодействие стало действительно важным, а также благодаря множеству пользователей, которые в силу определенных условий вынуждены принимать как должное аварийные отказы и бреши в системе безопасности. Данную ситуацию нельзя назвать стабильной. Сторонники Linux успешно ее использовали в 2003 году для того, чтобы проникнуть на рынок серверных операционных систем.

7.3. Операционная система VMS

VMS – частная операционная система, первоначально разработанная для миникомпьютера VAX корпорации «Digital Equipment Corporation» (DEC). Впервые она была выпущена в 1978 году и была важной действующей операционной системой в 80-х и начале 90-х годов. Сопровождение данной системы продолжалось, когда DEC была приобретена компанией Compaq, а последняя – корпорацией Hewlett-Packard. На данный момент операционная система VMS продолжает продаваться и поддерживаться. Она имеет полную вытесняющую многозадачность (при

которой процессор принудительно может быть отобран у текущей задачи), однако создание дочерних процессов в ней весьма дорогостоящее. Файловая система в VMS имеет детально разработанное понятие типов записи (хотя в ней нет атрибутов). Поэтому в VMS проявляется тенденция к увеличению размеров программ и созданию тяжеловесных монолитов.

VMS характеризуется длинными четкими системными командами, подобными инструкциям языка COBOL. Имеется весьма полная интерактивная справочная система (не по API интерфейсам, а по запускаемым программам и синтаксису командной строки). Фактически CLI-интерфейс (Command Line Interface – интерфейс командной строки) VMS и ее справочная система являются организационной метафорой VMS. Хотя система Windows модифицирована для VMS, CLI-интерфейс продолжает оказывать наиболее важное стилистическое влияние на проектирование программ. В связи с этим определяется ряд следующих факторов и последствий.

- Чем длиннее команда, которую необходимо ввести, тем меньше пользователь хочет это делать.
- Люди хотят вводить с клавиатуры меньше команд, поэтому используется малое количество обслуживающих программ, зато с большим количеством функций (и потому слишком крупных).
- Количество и тип принимаемых программой параметров определяются синтаксическими ограничениями, налагаемыми справочной системой.
- Справка в VMS весьма полная, но поиск и поисковые средства в ней отсутствуют. Это затрудняет получение сведений, поддерживает специализацию и препятствует любительскому программированию.

VMS была разработана для многопользовательской работы, она имеет заслуживающую доверия систему внутренних границ, разделяющих процессы друг от друга с помощью аппаратного блока MMU. Взломы системы безопасности VMS бывают редко.

Первоначально VMS-инструменты были дорогими, а интерфейсы сложными. Огромное количество программной документации для VMS доступны только в бумажной форме, поэтому поиск каких-либо сведений является трудоемкой операцией. Только после того как поставщик VMS почти забросим данную систему, вокруг нее развилось любительское программирование, но данная культура не является особенно стойкой.

Подобно UNIX, операционная система VMS предшествовала разграничению клиент/сервер. Она была успешной в свое время в качестве общецелевой системы разделения времени. Целевую аудиторию главным образом представляли технические пользователи и программные предприятия, допускающие умеренную сложность.

Лекция № 8. ОПЕРАЦИОННЫЕ СИСТЕМЫ MacOS и BeOS

8.1. Операционная система MacOS

Операционная система Macintosh была разработана в компании Apple в начале 80-х годов прошлого века. Ее создателей вдохновила передовая работа по разработке GUI-интерфейсов, осуществленная ранее в Исследовательском центре Palo Alto (Palo Alto Research Center) компании Xerox. Она увидела свет вместе с компьютером Macintosh в 1984 году. С тех пор MacOS подверглась двум значительным преобразованиям конструкции, а в настоящее время претерпевает третье. Первое преобразование было связано с переходом от поддержки только одного приложения в тот или иной момент времени к невытесняющей многозадачности (Multi Finder). Вторым преобразованием был переход с процессоров серии 68000 на процессоры PowerPC, что позволило сохранить обратную бинарную совместимость с приложениями 68К, а также добавило для PowerPC-приложений усовершенствованную систему управления общими библиотеками, заменяющими исходную систему прерываний совместно используемых программ на основе инструкций процессора 68К. Третьим преобразованием было объединение в системе MacOS X конструкторских идей MacOS с UNIX-производной инфраструктурой. Далее мы будем рассматривать версии данной системы, предшествующие MacOS X.

В MacOS прослеживается очень сильное влияние унифицирующей идеи, которая весьма отличается от идеи UNIX: нормы проектирования интерфейсов компьютеров Macintosh (Mac Interface Guidelines). Они подробнейшим образом определяют внешний вид графического интерфейса приложений и режимы его работы. Согласованность норм значительно влияет на культуру пользователей компьютеров Macintosh. Нередко просто перенесенные программы из DOS или UNIX, не соблюдающие определенные нормы, немедленно отвергаются сообществом Mac-пользователей и терпят неудачи на рынке.

Одна из ключевых идей относительно норм заключается в том, что рабочие компоненты должны находиться там, куда их перенес пользователь. Документы, каталоги и другие объекты постоянно сохраняются на рабочем столе, не смешиваясь с системной информацией, а содержание рабочего стола сохраняется при перезагрузках.

Унифицирующая идея Macintosh проявляется настолько сильно, что большинство других вариантов конструкции либо находятся под ее влиянием, либо незаметны. Во всех программах предусмотрены графические интерфейсы. Интерфейса командной строки не существует вообще. Средства сценариев есть в наличии, однако они используются значительно реже, чем в UNIX; многие Mac-программисты никогда их не изучают. Присущая MacOS метафора неотделимости GUI-интерфейса (организованная вокруг единственного главного событийного цикла) обусловила наличие слабого планировщика задач без приоритетности обслуживания. Слабый планировщик и работа всех MultiFinder-приложений в одном большом адресном пространстве означают, что использовать отдельные процессы или даже параллельные процессы вместо поочередного опроса непрактично.

Вместе с тем приложения MacOS не являются неизменно огромными монолитами. Системный код поддержки графического интерфейса, который частично реализован в ПЗУ, поставляемом с аппаратным обеспечением, а частично в совместно используемых библиотеках, обменивается данными с MacOS-программами посредством интерфейса событий, который с момента возникновения является весьма стабильным. Таким образом, конструкция данной операционной системы поддерживает относительно четкое обособление ядра приложений от GUI-интерфейса.

В MacOS также имеется мощная поддержка для изоляции метаданных приложений, таких как структуры меню, от кода ядра. Файлы данной операционной системы имеют как «ветвь данных» (data folk) (блок байтов в UNIX-стиле, который содержит документ или программный код), так и «ветвь ресурсов» (resource folk) (набор определяемых пользователем

атрибутов файла). Mac-приложения часто проектируются так для того, чтобы (например) используемые в приложениях изображения и звук хранились в ветви ресурса и чтобы их можно было бы модифицировать отдельно от кода приложения.

Система внутренних границ MacOS является очень непрочной. Имеется жесткое предположение о том, что есть только один пользователь, поэтому групп привилегий не существует. Многозадачность определяется как невытесняющая. Все MultiFinder-приложения запускаются в одном адресном пространстве, поэтому некорректный код какого-либо приложения способен разрушить любые данные, находящиеся за пределами низкоуровневого ядра операционной системы. Взломать систему безопасности MacOS-машин весьма просто. Данная операционная система избавлена от вирусных эпидемий главным образом потому, что очень немногие заинтересованы в ее взломе.

Mac-программисты стремятся разрабатывать приложения в противоположном относительно UNIX-программирования направлении, т.е. это направление от интерфейса к ядру, а не наоборот. Такой подход поощряется всей конструкцией MacOS.

Macintosh задумывалась как клиентская операционная система для нетехнический конечных пользователей, что предполагает очень низкую толерантность относительно сложности интерфейса. Разработчики в Mac-культуре достигли значительных успехов в проектировании простых интерфейсов.

Классическая система MacOS устаревает. Большинство имеющихся в ней средств импортируются в MacOS X, которая объединяет их с UNIX-инфраструктурой, вышедшей из традиций университета в Беркли. MacOS X состоит из двух частных уровней (перенесенные приложения OpenStep и классические GUI-интерфейсы Macintosh) поверх UNIX-основы с открытым исходным кодом (проект Darwin).

В то же время лидирующие UNIX-системы, например Linux, начинают заимствовать у MacOS такие идеи, как использование атрибутов файлов (обобщение ветви ресурса).

8.2. Операционная система BeOS

Компания Be Inc. была основана в 1989 году как производитель аппаратного обеспечения в виде передовых многопроцессорных машин на основе микросхем PowerPC. Операционная система BeOS бы попыткой компании Be повысить стоимость аппаратного обеспечения путем создания новой сетевой модели операционной системы, учитывающей уроки UNIX и MacOS, но не являющейся подобием одной из них. В результате появилась изящная, ясная и интересная конструкция с превосходной производительностью в предопределенной для нее роли мультимедийной платформы.

Унифицирующими идеями BeOS были «заполняющая параллельная обработка» (pervasive threading), мультимедийные потоки и файловая система, выполненная в виде базы данных. BeOS была спроектирована для минимизации задержек в ядре, что делало ее применимым для обработки в реальном времени больших объемов таких данных, как аудио- и видео-потоки. «Параллельные процессы» (threads) BeOS по существу были легковесными процессами в терминологии UNIX, так как они поддерживали локальную память процесса и, следовательно, не обязательно совместно использовали все адресное пространство. Межпроцессорный обмен данными посредством совместно используемой памяти был быстрым и эффективным.

BeOS придерживалась модели UNIX в том, что не имела файловой структуры выше байтового уровня. Подобно MacOS, операционная система BeOS поддерживала и использовала атрибуты файлов. По сути, файловая система BeOS была базой данных с возможностью индексации по любому атрибуту.

Одним из элементов, заимствованным BeOS у UNIX, была логичная конструкция внутренних границ. В описываемой системе полностью использовался блок MMU, и работающие процессы были надежно изолированы друг от друга. Несмотря на то, что BeOS представлялась как однопользовательская операционная система (без необходимости регистрации в ней), в ее файловой системе и во всем внутреннем устройстве поддерживались UNIX-подобные группы привилегий. Они использовались для защиты важнейших системных файлов от воздействия ненадежного кода. В терминологии UNIX пользователь во время загрузки регистрировался в системе как анонимный гость, а другим единственным «пользователем» был root. Полная многопользовательская функциональность потребовала бы небольших изменений в верхних уровнях системы и по сути была представлена утилитой BeLogin.

Предпочтительным стилем пользовательского интерфейса был GUI, который соответствовал опыту MacOS в области дизайна интерфейсов. Вместе с тем также полностью поддерживались CLI-интерфейс и сценарии. Оболочкой командной строки была перенесенная с UNIX программа *bash(1)*, доминирующая UNIX-оболочка с открытым исходным кодом, работающая посредством библиотеки совместимости POSIX. Благодаря такой конструкции, преобразование программного обеспечения UNIX CLI было очень простым. В системе присутствовала инфраструктура для поддержки всего разнообразия сценариев, фильтров и служебных доменов, сопутствующих UNIX-модели.

BeOS была предназначена для работы в качестве клиентской операционной системы, специализирующейся на обработке мультимедийных данных (особенно на обработке звука и видео) почти в реальном времени. Целевая аудитория данной системы включала в себя технических и деловых конечных пользователей, предполагающих умеренную толерантность к сложности интерфейса.

Препятствия на пути к разработке BeOS-приложений были небольшими. Несмотря на то, что операционная система была частной, средства разработки были недорогими, и доступ к полной документации не представлял сложности. Проект BeOS начался как часть усилий по освобождению от аппаратного обеспечения Intel с RISC-технологией, и после взрывного роста Internet продолжался исключительно как программный проект. Его стратеги изучили опыт периода формирования Linux в начале 90-х годов и были полностью осведомлены о значении крупной базы любительской разработки. Фактически они преуспели в привлечении верных последователей; в 2003 году не менее пяти отдельных проектов пытались возродить операционную систему BeOS в открытом исходном коде.

К сожалению, в отличие от технического дизайна, окружавшая BeOS бизнес-стратегия была не столь мудрой. Программное обеспечение BeOS первоначально было привязано к специализированному аппаратному обеспечению и продавалось только с неопределенными указаниями о целевых приложениях. Позднее (в 1998 году) операционная система BeOS была перенесена на общее аппаратное обеспечение PC, а мультимедийным приложениям было уделено более пристальное внимание, но система так и не привлекла критическую массу пользователей. Наконец в 2001 году BeOS уступила комбинации конкурентного маневрирования Microsoft (судебный процесс продолжался в 2003 году) и конкуренции со стороны вариантов операционной системы Linux, адаптированных для обработки мультимедиа.

Лекция № 9. ОПЕРАЦИОННЫЕ СИСТЕМЫ ФИРМЫ IBM: OS/2, MVS и VM/CMS

9.1. Операционная система OS/2

Операционная система OS/2 зародилась как опытный проект компании IBM, называвшийся ADOS (Advanced DOS), один из трех претендентов на роль DOS 4. В то время компании IBM и Microsoft формально сотрудничали при разработке операционной системы следующего поколения для компьютеров PC. OS/2 версии 1.0 впервые вышла в 1987 году для компьютеров с процессорами 286-й серии, но не имела успеха. Версия 2.0 для процессоров 386 появилась в 1992 году, но к этому времени альянс IBM/ Microsoft уже распался. Microsoft с системой Windows 3.0 двинулась в другом (более выгодном) направлении. OS/2 привлекала преданное меньшинство последователей, но так и не смогла привлечь критическую массу разработчиков и пользователей. На рынке настольных систем она оставалась третьей после Macintosh до тех пор, пока не была отнесена к Java-инициативе IBM 1996 года. Последней была версия 4.0 в 1996 году. Ранние версии нашли применение во встроенных системах и продолжают работать во многих машинах автоматических справочных служб во всем мире.

Подобно UNIX, OS/2 была создана с поддержкой вытесняющей многозадачности и не работала бы без блока MMU (ранние версии имитировали MMU с помощью сегментации памяти в 286-х процессорах). В отличие от UNIX, OS/2 никогда не создавалась для работы в качестве многопользовательской системы. Создание дочерних процессов было относительно недорогим, но межпроцессное взаимодействие было сложным и ненадежным. Поддержка сети первоначально сводилась к LAN-протоколам, однако в более поздних версиях был добавлен набор протоколов TCP/IP. В OS/2 не было программ, аналогичных системным службам UNIX, поэтому данная система никогда не обеспечивала многофункциональную поддержку сети.

В данной операционной системе были как CLI-, так и GUI-интерфейс. Большинство положительных отзывов, касающихся OS/2, относились к ее рабочему столу Workplace Shell (WPS). Часть этой технологии была лицензирована у разработчиков AmigaOS Workbench, революционного графического интерфейса настольных систем, который имел верных почитателей в Европе. Это одна из областей дизайна, где OS/2 приобрела такой потенциал, которого UNIX, вероятно, еще не достигла. Оболочка WPS представляла собой четкую, мощную, объектно-ориентированную конструкцию с ясным режимом работы и хорошей расширяемостью. По прошествии нескольких лет она станет исходной моделью для Linux-проекта GNOME.

Конструкция WPS с иерархией классов была одной из унифицирующих идей операционной системы OS/2. Другой идеей была мультипроцессорная обработка. OS/2-программисты использовали организацию параллельной обработки в большой степени как частичную замену IPC (Inter-Process Communication – межпроцессорное взаимодействие) между равноправными процессами. Традиции создания взаимодействующих инструментов не развивались.

OS/2 имела внутренние границы. Работающие процессы были защищены друг от друга, пространство ядра было защищено от пользовательского пространства, но пользовательских групп привилегий не было. Это означало, что файловая система не была защищена от злонамеренного кода.

OS/2 имела текстовый, а не двоичный режим. Она поддерживала атрибуты файлов, которые использовались для сохранения постоянство рабочего стола подобно Macintosh. Системные базы данных хранились главным образом в двоичных форматах.

Пользовательские интерфейсы часто были более эргономичными, чем в Windows, хотя и не достигали стандартов Macintosh. Подобно UNIX и Windows, пользовательский интерфейс OS/2 был организован вокруг

множества независимых групп окон для различных задач, вместо захвата рабочего стола действующим приложением.

Целевой аудиторией OS/2 были предприятия и нетехнические пользователи, в связи с чем предполагалась низкая толерантность относительно сложности интерфейса. Данная система использовалась как в качестве клиентской, так и в качестве файлового сервера и сервера печати.

OS/2 представляет интерес как учебный пример того, как далеко может продвинуться конструкция многозадачной, но однопользовательской операционной системы.

9.2. Операционная система MVS

MVS (Multiple Virtual Storage – многопользовательское виртуальное хранение) – ведущая операционная система IBM для мэйн-фреймов корпорации. Ее происхождение связывают с OS/360, операционной системой IBM, появившейся в середине 60-х годов прошлого века.

Из всех рассмотренных здесь операционных систем только MVS может считаться более старшей, чем UNIX. Данная система также менее остальных подверглась влиянию идей и технологии UNIX и представляет надежнейшую конструкцию, противоположную последней. Унифицирующей идеей MVS является то, что вся работа формируется в виде пакета. Система разработана для наиболее эффективного использования машины для пакетной обработки больших объемов данных с минимальной необходимостью взаимодействия с пользователями.

Собственные MVS-терминалы функционируют только в режиме блокировки. Пользователю предоставлен экран, который он заполняет, модифицируя локальную память терминала. Прерывание не происходит до тех пор, пока пользователь не нажмет клавишу отправки. Взаимодействие с помощью командной строки, непосредственный ввод данных с клавиатуры, невозможен.

Другое следствие архитектуры, ориентированной на пакетную обработку, заключается в том, что создание подпроцессов является медленной операцией. В данной системе большая пропускная способность достигается ценой дорогостоящей установки (и связанной с этим задержки). Подобный подход хорошо соответствует пакетным операциям, но плохо сказывается на интерактивном отклике.

Операционная система MVS использует аппаратный блок MMU. Процессы выполняются в отдельных адресных пространствах. Межпроцессорный обмен данными осуществляется через совместно используемую память.

Безопасность файловой системы была поздним дополнением к первоначальной конструкции. Однако когда выяснилось, что безопасность необходима, IBM добавила соответствующие функции оригинальным образом: разработчики определили общий API-интерфейс функций безопасности, а затем все запросы на доступ к файлам перед обработкой направили через данный интерфейс. В результате существует, по крайней мере, три конкурирующих пакета обеспечения безопасности с различной философией дизайна, и все они весьма хороши, учитывая то, что известных взломов между 1980 годами и настоящим временем не было.

Сетевые средства также были добавлены с опозданием. Любительское программирование в MVS почти отсутствует, существуя только внутри сообщества крупных предприятий, использующих данную операционную систему. Поскольку стоимость среды резко снижается, уже определилась небольшая, но растущая группа пользователей последней свободно распространяемой версии MVS (3.8, датированной 1979 годом). Эта система, как и все средства разработки, а также эмулятор для их запуска, доступны по цене компакт-диска.

9.3. Операционная система VM/CMS

VM/CMS – другой пример операционной системы для мэйн-фреймов. Ее вполне можно назвать «родственницей» системы UNIX: их общим предком является система CTSS, созданная в Массачусетском технологическом институте приблизительно в 1963 году и работавшая на мэнфрейме IBM 7094. Группа, разработавшая CTSS, позднее приступила к написанию MULTICS, прямого предка UNIX. IBM учредила в Кембридже группу по созданию системы разделения времени для IBM 360/40, модифицированного компьютера 360-й серии со страничным (впервые для систем IBM) диспетчером MMU.

Операционные системы VM/CMS и UNIX являлись видоизмененными «отражениями» друг друга. Унифицирующая идея системы, обеспеченная компонентом VM, воплощена в виртуальных машинах, которые выглядят как физические машины. Они поддерживают вытесняющую многозадачность и работают либо с однопользовательской операционной системой CMS, либо с полностью многозадачной системой (обычно MVS, Linux или другой экземпляр самой VM). Виртуальные машины соответствуют UNIX-процессам, демонам и эмуляторам, а обмен данными между ними осуществляется путем соединения виртуального карточного перфоратора одной машины с виртуальным считывателем перфокарт другой машины. В дополнение к этому, внутри системы обеспечивается многоуровневая инструментальная среда, которая называется CMS Pipelines (конвейеры CMS), непосредственно смоделированная с каналов UNIX, но архитектурно расширенная для поддержки множества вводов и выводов.

В ситуации, когда обмен данными между виртуальными машинами явно не установлен, они полностью изолированы друг от друга. Данная операционная система характеризуется тем же высоким уровнем надежности, расширяемости и безопасности, что и MVS, а также имеет гораздо большую гибкость и проще в использовании.

Стиль пользовательского интерфейса в CMS является интерактивным и диалоговым, весьма отличающимся от MVS, но похожим на пользовательские интерфейсы VMS и UNIX. Интенсивно используется полноэкранный редактор XEDIT.

Операционная система VM/CMS предшествовала разделению клиент/сервер и в настоящее время используется почти полностью как серверная операционная система с эмуляцией IBM-терминалов. До того как Windows стала полностью доминировать на рынке настольных систем, VM/CMS предоставляла службы обработки текстов и электронную почту как внутри IBM, так и между участками пользователей мэйнфреймов. Многие VM-системы были инсталлированы исключительно для запуска таких приложений благодаря доступной расширяемости VM (десятки тысяч пользователей).

Язык сценариев REXX поддерживает программирование в стиле, не отличающемся от shell, awk, Perl или Python. Следовательно, любительское программирование (особенно системными администраторами) является весьма важным в системе VM/CMS.

Прослеживаются поразительные параллели между историей VM/CMS внутри корпорации IBM и UNIX внутри Digital Equipment Corporation (которая создавала компьютеры, где впервые работала UNIX). Компании IBM потребовались годы, чтобы осознать стратегическую важность ее неофициальной системы разделения времени. В течение этого времени появилось сообщество программистов VM/CMS, поведение которого было весьма сходно с поведением раннего UNIX-сообщества. Они обменивались идеями, открытиями в исследовании систем, а главное – обменивались исходным кодом для утилит. Независимо от того, как часто IBM пыталась объявлять о смерти системы VM/CMS, сообщество настаивало на поддержании ее в рабочем состоянии.

Однако системе VM/CMS не хватает какого-либо реального аналога для языка C. Эта операционная системе написана на ассемблере, и остается

такой и поныне. Единственным эквивалентом C были различные сокращенные версии языка PL/I, который использовался в IBM для системного программирования, однако клиентам компании не предоставлялся.

С 2000 года IBM очень активно продвигает операционную систему VM/CMS на мэйнфреймах как способ одновременной поддержки тысяч виртуальных Linux-машин.

Лекция № 10. ОПЕРАЦИОННЫЕ СИСТЕМЫ QNX и Linux

10.1. Операционная система реального времени QNX

Сетевая операционная система реального времени QNX является мощной операционной системой, разработанной для процессоров с архитектурой i32. Она позволяет проектировать сложные программные комплексы, работающие в реальном времени как на отдельном компьютере, так и в локальной вычислительной сети. Встроенные средства QNX обеспечивают поддержку многозадачного режима на одном компьютере и взаимодействие параллельно выполняемых задач на разных компьютерах, работающих в среде локальной вычислительной сети.

Основным языком программирования в системе является C. Основная операционная среда соответствует стандарту POSIX. Это позволяет с небольшими доработками переносить ранее разработанное программное обеспечение в QNX.

Операционная система QNX обладает свойствами предсказуемости и масштабируемости.

Предсказуемость означает применимость системы к задачам жесткого реального времени. QNX является операционной системой, которая дает полную гарантию того, что процесс с наивысшим приоритетом начнет выполняться практически немедленно, и критически важное событие (например, сигнал тревоги) никогда не будет потеряно.

Масштабируемость – свойство, выражающееся в возможности исполнения программы на различных ресурсах (объём памяти, число и производительность процессоров) с пропорциональным изменению ресурсов значением показателей эффективности. Именно способность работать на ограниченных аппаратных ресурсах позволяет использовать QNX во встроенных системах.

С точки зрения пользовательского интерфейса QNX очень похожа на UNIX, поскольку выполняет требования стандарта POSIX. Однако QNX – это

не версия UNIX, хотя многие так считают. Операционная система QNX была разработана «с нуля» канадской фирмой QNX Software Systems Limited в 1989 году по заказу Министерства обороны США, причем на совершенно иных архитектурных принципах, нежели использовались при создании операционной системы UNIX.

QNX была первой коммерческой операционной системой, построенной на принципах микроядра и обмена сообщениями. Система реализована в виде совокупности независимых (но взаимодействующих путем обмена сообщениями) процессов различного уровня (менеджеры и драйверы), каждый из которых реализует определенный вид услуг.

Микроядро операционной системы QNX имеет объем всего в несколько десятков килобайтов (в одной из версий – 10 Кбайт, в другой – менее 32 Кбайт, хотя есть вариант и на 46 Кбайт). В этом объеме помещаются:

- Механизм передачи сообщений между процессами IPC (Inter Process Communication).
- Редиректор (redirector) прерываний.
- Блок планирования выполнения задач (диспетчер задач).
- Сетевой интерфейс для перенаправления сообщений (менеджер Net).

10.2. Операционная система Linux

Операционная система Linux, созданная Линусом Торвальдсом в 1991 году лидирует среди UNIX-систем новой школы с открытым исходным кодом, появившихся в 1990 году (в их число также входит FreeBSD, NetBSD, OpenBSD и Darwin), и представляет направление конструирования, принятое данной группой в целом.

Linux не включает в себя код из дерева исходных кодов первоначальной UNIX, но данная система была сконструирована на основе UNIX-стандартов и работает подобно UNIX.

Многие разработчики и активисты Linux-сообщества стремятся к тому, чтобы данная операционная система заняла прочные позиции на рабочих столах конечных пользователей. Это делает целевую аудиторию Linux несколько шире, чем в случае какой-либо из UNIX-систем старой школы, которые в основном были нацелены на рынки серверов и рабочих станций технических пользователей.

Наиболее очевидным новшеством является смена предпочтительных стилей интерфейса. Большинство UNIX-программистов долго оставались прочно привязанными к командной строке. Разработчики Linux перешли к созданию GUI-интерфейсов.

Желание склонить на свою сторону конечных пользователей также заставило Linux-разработчиков гораздо больше интересоваться проблемами простоты инсталляции и распространения программного обеспечения, чем это было принято при разработке частных UNIX-систем.

Linux-сообщество стремится превратить свое программное обеспечение в некий универсальный канал связи между различными средами. Поэтому Linux предоставляет поддержку чтения и (нередко) записи форматов файловых систем и методов сетевого взаимодействия, характерных для других операционных систем. Linux также поддерживает возможность выбора операционной системы при начальной загрузке на одном и том же аппаратном обеспечении (мультизагрузку), а также программную эмуляцию данных систем внутри самой себя. Долгосрочной целью является поглощение этих систем.

Linux-разработчики перенимают конструкторские идеи из операционных систем, не относящихся к семейству UNIX. Примером могут служить Linux-приложения, использующие для конфигурации INI-файлы формата Windows. Внедрение в ядро 2.5 Linux расширенных атрибутов файлов, которые среди прочего можно использовать для эмуляции семантики ветви ресурса в Macintosh, – еще один яркий пример этого.

Остальные частные UNIX-системы (такие как Solaris, HP-UX, AIX и другие) разрабатываются как большие продукты для суперкомпьютеров. Их рыночная ниша поддерживает конструкции, оптимизированные под максимальную мощность на высококлассном, инновационном аппаратном обеспечении. Поскольку частично Linux связана со средой энтузиастов PC, особое значение в данной системе уделяется выполнению большего количества задач при меньших затратах.

Лекция № 11. ОПЕРАЦИОННЫЕ СИСТЕМЫ ФИРМЫ MICROSOFT

11.1. Операционная система DOS

DOS (дисковая операционная система) – одна из первых операционных систем, предназначенных для работы персональных компьютеров. Она состоит из следующих частей.

- ***Базовая система ввода-вывода (BIOS)***. Находится в ПЗУ компьютера. Ее назначение состоит в выполнении наиболее простых и универсальных услуг операционной системы, связанных с осуществлением ввода-вывода. BIOS содержит также тест функционирования компьютера, проверяющий работу памяти и устройств компьютера при включении электропитания. Кроме того, BIOS содержит также программу вызова загрузчика операционной системы.
- ***Загрузчик операционной системы***. Это очень короткая программа, находящаяся в первом секторе каждой дискеты с операционной системой DOS. Функция этой программы заключается в считывании в память еще двух модулей операционной системы.
- ***Системные файлы IO.SYS и MSDOS.SYS***. Эти файлы загружаются в оперативную память компьютера загрузчиком операционной системы и остаются там постоянно. Файл IO.SYS представляет собой дополнение к BIOS. Файл MSDOS.SYS реализует основные высокоуровневые услуги DOS.
- ***Командный процессор DOS***. Командный процессор обрабатывает команды, вводимые пользователем. Он находится в файле COMMAND.COM на диске, с которого загружается операционная система. Некоторые команды пользователя, например Type, Dir или Copy командный процессор выполняет сам. Такие команды называются внутренними. Для выполнения остальных (внешних) команд пользователя командный процессор ищет на дисках

программу с соответствующим именем и, если находит ее, то загружает в память и передает ей управление. По окончании работы программы командный процессор удаляет программу из памяти и выводит сообщение о готовности к выполнению команд (приглашение DOS).

- ***Внешние команды DOS***. Это программы, поставляемые вместе с операционной системой в виде отдельных файлов. Эти программы выполняют действия обслуживающего характера, например, форматирование дискет, проверку дисков и т.д.
- ***Драйверы устройств***. Это специальные программы, которые дополняют систему ввода-вывода DOS и обеспечивают обслуживание новых или нестандартное использование имеющихся устройств. Например, с помощью драйверов возможна работа с «электронным диском», часть операционной памяти компьютера, с которой можно работать так же, как с диском. Драйверы загружаются в операционную память при загрузке операционной системы, их имена указываются в специальном файле CONFIG.SYS. Такая схема облегчает добавление новых устройств и позволяет делать это, не затрагивая системные файлы DOS.

11.2. Windows NT

Windows NT (***New Technology***) – операционная система корпорации Microsoft для использования на мощных персональных компьютерах и серверах. Windows NT 4 (1996) (и все последующие модификации: Windows 2000 (2000), Windows XP (2002), Windows Server 2003 (2003), Windows Vista (2007)) – это нечто совсем иное, чем DOS. Хотя в этих операционных системах можно открыть окно сеанса DOS, они вовсе не являются оболочкой в традиционном понимании этого слова. Здесь речь может идти, скорее, об эмуляции DOS (для того чтобы все желающие могли поработать с привычным интерфейсом командной строки). В сеансе DOS Windows NT

многие DOS-программы работать не будут. И символьного режима экрана, который в Windows *9x* предшествует загрузке графической оболочки, вы здесь не увидите.

Для хранения параметров и загрузки драйверов Windows NT, как и Windows *9x,* использует системный реестр. Файлов Config. sys, Autoexec. bat и .ini здесь нет вообще. Более того, модернизировать Windows *9x* до Windows NT невозможно. При установке Windows NT все приложения придется устанавливать и настраивать заново. Windows NT может использовать файловую систему FAT, и поэтому вы можете загружать компьютер с DOS-диска и иметь полный доступ ко всем файлам. Однако некоторые из самых прогрессивных возможностей Windows NT обеспечиваются ее собственной файловой системой NTFS (NT File System). NTFS позволяет создавать на диске разделы объемом до 2 Тбайт (как и FAT 32), но, кроме этого, в нее встроены функции сжатия файлов, безопасности и аудита, необходимые при работе в сетевой среде. Установка операционной системы Windows NT начинается на диске FAT, но по желанию пользователя в конце установки данные на диске могут быть конвертированы в формат NTFS. Можно сделать это и позже, воспользовавшись утилитой Convert. exe, поставляемой вместе с операционной системой. Преобразованный к системе NTFS раздел диска становится недоступным для других операционных систем. Чтобы вернуться в DOS, Windows 3.1 или Windows *9x,* нужно удалить раздел NTFS, а вместо него создать раздел FAT. Windows 2000 можно устанавливать на диск с файловой системой FAT 32 и NTFS.

11.3. Windows API

Интерфейс прикладного программирования API (Application Program Interface) **–** это набор функций, принадлежащих ядру (или надстройкам) операционной системы, который используется прикладными и системными программами как в составе операционной системы, так и в составе системы программирования.

Все API можно разделить на три класса:

- API как интерфейс высокого уровня, принадлежащий к библиотекам RTL (Run Time Library – библиотека времени выполнения);
- API прикладных и системных программ, входящих в поставку операционной системы;
- Прочие интерфейсы API.

Библиотека RTL включает в себя стандартные подпрограммы, которые система программирования поставляет на этапе компиляции. В общем случае это не только модули системы программирования, но и модули операционной системы.

В Windows API имеется множество как самых незаметных, так и значительных отличий от других API, таких как POSIX API, с которым знакомы программисты, работающие в UNIX и Linux. Многие системные ресурсы Windows представляются в виде ***объектов ядра*** (kernel objects), для идентификации и обращения к которым используются ***дескрипторы*** (handles). По смыслу эти дескрипторы аналогичны дескрипторам (descriptors) файлов и идентификаторам (ID) процессов в UNIX.

Существует точка зрения, что Windows – это всего лишь API операционной системы, предоставляющий набор средств для решения пользовательских задач.

Любые манипуляции с объектами ядра осуществляются только с использованием Windows API. "Лазеек" для обхода этого правила нет. Подобная организация работы согласуется с принципами абстрагирования данных, используемыми в объектно-ориентированном программировании, хотя сама система Windows объектно-ориентированной не является.

К объектам относятся файлы, процессы, потоки, каналы межпроцессного взаимодействия, объекты отображения файлов, события и многое другое. Объекты имеют атрибуты защиты.

Windows – богатый возможностями и гибкий интерфейс. Во-первых, одни и те же или аналогичные задачи могут решаться с помощью сразу

нескольких функций; так, имеются вспомогательные функции (convenience functions), полученные объединением часто встречающихся последовательностей функциональных вызовов в одну функцию (к числу подобных функций принадлежит и функция **CopyFile**, используемая в одном из примеров далее). Во-вторых, функции часто имеют многочисленные параметры и флаги, многие из которых обычно игнорируются.

Windows предлагает многочисленные механизмы синхронизации и взаимодействия, обеспечивающие удовлетворение самых разнообразных запросов.

Базовой единицей выполнения в Windows является поток (**thread**). В одном процессе (**process**) могут выполняться один или несколько потоков.

Для функций Windows используются длинные описательные имена. Приведенные ниже в качестве примера имена функций иллюстрируют не только соглашения об использовании имен, но и многоликость функций Windows:

WaitForSingleObject

WaitForSingleObjectEx

WaitForMultipleObjects

WaitNamedPipe

Существует также несколько соглашений, регулирующих порядок использования имен типов:

Имена предопределенных типов данных, необходимых API, также являются описательными, и в них должны использоваться прописные буквы. К числу наиболее распространенных относятся следующие типы данных:

BOOL (определен как 32-битовый объект, предназначенный для хранения одного логического значения),

HANDLE,

DWORD (вездесущее 32-битовое целое без знака),

LPTSTR (указатель на строку, состоящую из 8- или 16-битовых символов) **LPSECURITY_ATTRIBUTES**.

С другими многочисленными типами данных будем знакомиться по мере изложения материала.

В именах предопределенных типов указателей операция * не используется, и они отражают дополнительные отличия между указателями различного типа, как, например, в случае типов **LPTSTR** (определен как **TCHAR** *) и **LPCTSTR** (определен как **const TCHAR** *). Тип **TCHAR** может обозначать как обычный символьный тип **char**, так и двухбайтовый тип **wchar_t**.

В отношении использования имен переменных, – по крайней мере, в прототипах функций, – также имеются определенные соглашения. Так, имя **lpszFileName** соответствует "длинному указателю на строку, завершающуюся нулевым символом", которая содержит имя файла. Этот пример иллюстрирует применение так называемой "венгерской нотации", которой мы, как правило, не стремимся придерживаться. Точно так же, **dwAccess** – двойное слово (32 бита), содержащее флаги прав доступа к файлу, где "**dw**" означает "**double word**" – "двойное слово".

Наконец, несмотря на то что оригинальный API Win32 с самого начала разрабатывался как совершенно независимый интерфейс, он проектировался с учетом обеспечения обратной совместимости с API Winl6, входившим в состав Windows 3.1. Это привело к некоторым досадным с точки зрения программиста последствиям:

- В названиях типов встречаются элементы анахронизма, как, например, в случае типов **LPTSTR** и **LPDWORD**, ссылающихся на "длинный указатель", который является простым 32- или 64-битовым указателем. Необходимость в указателях какого-либо иного типа отсутствует. Иногда составляющая "длинный"

опускается, и тогда, например, типы **LPVOID** и **PVOID** являются эквивалентными.

- В имена некоторых символических констант, например **WIN32_FIND_DATA**, входит компонент "WIN32", хотя те же константы используются и в Win64.
- Несмотря на то что упомянутая проблема обратной совместимости в настоящее время потеряла свою актуальность, она оставила после себя множество 16-разрядных функций, ни одна из которых, как правило, не используется, хотя и могло бы показаться, что эти функции играют весьма важную роль. В качестве примера можно привести функцию **OpenFile**, которая, судя по ее названию, нужна для открытия файлов, тогда как в действительности для открытия существующих файлов всегда следует пользоваться только функцией **CreateFile**.

Лекция № 12. ВВЕДЕНИЕ В СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

12.1. Основные понятия

Системная программа – программа, предназначенная для поддержания работоспособности прикладных программ пользователя или для повышения эффективности их использования.

Прикладная программа – программа, предназначенная для решения задач в определенной области науки, техники, культуры, или для личных нужд пользователя.

Системное программирование – это процесс разработки системных программ.

В последнее время появилось понятие промежуточного программного обеспечения (ПО), включающего в себя элементы как системного, так и прикладного программирования.

Промежуточное ПО (middleware) – это совокупность программ, осуществляющих управление вторичными (конструируемыми самим ПО) ресурсами, ориентированными на решение широкого класса задач универсального значения. К такому ПО относятся менеджеры транзакций, серверы БД, серверы коммуникаций и другие программные серверы.

С точки зрения инструментальных средств разработки промежуточное ПО ближе к прикладному, так как не работает на прямую с первичными ресурсами, а использует для этого сервисы, предоставляемые системным ПО. С точки зрения алгоритмов и технологий разработки промежуточное ПО ближе к системному, так как всегда является сложным программным изделием многократного и многоцелевого использования и в нем применяются те же или сходные алгоритмы, что и в системном ПО.

Современная тенденция развития ПО состоит в снижении объема как системного, так и прикладного программирования. Основная часть работы программистов выполняется в промежуточном ПО.

Снижение объема системного программирования определено современными концепциями ОС, объектно-ориентированной архитектурой и архитектурой микроядра, в соответствии с которыми большая часть функций системы выносится в утилиты, которые можно отнести и к промежуточному ПО. Снижение объема прикладного программирования обусловлено тем, что современные продукты промежуточного ПО предлагают все больший набор инструментальных средств и шаблонов для решения задач своего класса.

Значительная часть системного и практически все прикладное ПО пишется на языках высокого уровня, что обеспечивает сокращение расходов на их разработку, модификацию и переносимость.

Системное ПО подразделяется на системные управляющие программы и системные обслуживающие программы.

Управляющая программа – системная программа, реализующая набор функций управления, который включает в себя управление ресурсами и взаимодействие с внешней средой, восстановление работы системы после проявления неисправностей в технических средствах.

Программа обслуживания (утилита) – программа, предназначенная для оказания услуг общего характера пользователям и обслуживающему персоналу.

Управляющая программа совместно с набором необходимых для эксплуатации системы утилит составляют операционную систему.

Кроме входящих в состав операционной системы утилит могут существовать и другие утилиты (того же или стороннего производителя), выполняющие дополнительное (опционное) обслуживание. Как правило, это утилиты, обеспечивающие разработку программного обеспечения для операционной системы.

Система программирования – система, образуемая языком программирования, компилятором или интерпретатором программ, представленных на этом языке, соответствующей документацией, а также

вспомогательными средствами для подготовки программ к форме, пригодной для выполнения.

12.2. Использование командной строки

В операционной системе Windows NT/XP/Vista параллельно с красочными графическими интерфейсами продолжает сохраняться интерфейс командной строки. Хотя пользователями он используется достаточно редко, системному администратору без него не обойтись. Чтобы вызвать командную строку, нужно нажать кнопку «Пуск», а затем выбрать опцию «Выполнить». В раскрывшемся окне следует набрать **cmd** (командный процессор Windows) и щелкнуть на «Ok». Появится окно черного цвета с белыми буквами – интерфейс командной строки. На экране может быть написано следующее.

```
C:\ Documents and Settings\ UserName >
```

Это означает, что открыта папка «**UserName**», которая в свою очередь находится в папке «**Documents and Settings**», расположенной на диске «**C**». Знак «**>**» является приглашением к вводу команды. Если вы наберете команду «**dir**», а затем введете ее с помощью клавиши «**Enter**», то на экране появится список папок и файлов, находящихся в папке «**UserName**». Если хотите перейти в другой каталог, то нужно набрать команду «**cd**», а за ней указать путь, куда вы желаете перейти. Если вы решили ознакомиться со списком других внутренних команд процессора, то следует ввести команду «**help**».

Большинство операционных систем, в том числе DOS и UNIX, позволяют передать программе, написанной на языке C++, при запуске один или несколько параметров. Они называются ***параметрами командной строки*** и разделяются при записи пробелами. Непосредственно в функцию **main()** эти параметры не передаются. Вместо них функция **main()** получает два других параметра. Один из них – это количество аргументов

командной строки (целое число). По традиции он обозначается как **argc** (argument count – количество аргументов). Второй параметр – это массив указателей на символьные строки. Его обычно называют **argv** (argument vector – вектор аргумента). Имя запускаемой программы является первым аргументом, поэтому каждая программа имеет, по крайней мере, один аргумент.

Общепринятым подходом является проверка аргумента **argc**, гарантирующая соответствие количества переданных и полученных аргументов. В листинге 12.1 показан пример использования аргументов командной строки.

Листинг 12.1. Код программы «TestProgram»

```
#include <iostream.h>

int main(int argc, char *argv[])

{

    cout<<"Received "<<argc<<" arguments...\n";

    for (int i=0; i<argc; i++)

        cout<<"argument "<<i<<": "<<argv[i]<<endl;

    return 0;

}
```

Этот код нужно запустить из командной строки. Предположим, что для файла исполняемой программы мы выбрали название: **TestProgram.exe**. Тогда требуется войти в ту папку, в которой размещается этот файл, и набрать после знака приглашения, например, следующее:

```
TestProgram I am system programmer !
```

Получим следующий результат:

```
Received 6 arguments…
argument 0: TestProgram
argument 1: I
argument 2: am
argument 3: system
argument 4: programmer
argument 5: !
```

Как можно видеть, элемент **`argv[0]`** – это имя программы, а первый аргумент командной строки – **`argv[1]`**.

Далее в лекции приведены примеры коротких программ, реализующих простое последовательное копирование содержимого файла тремя различными способами:

1. С использованием библиотеки C.
2. С использованием Windows API.
3. С использованием вспомогательной функции Windows – **`CopyFile`**.

Последовательная обработка файлов является простейшей, наиболее распространенной и самой важной из возможностей, обеспечиваемых любой операционной системой, и почти в каждой большой программе хотя бы несколько файлов обязательно подвергаются этому виду обработки. Поэтому простая программа обработки файлов предоставляет прекрасную возможность ознакомиться с Windows и принятыми в ней соглашениями.

В приведенных программах организована лишь простейшая проверка ошибок, которые могут возникать на стадии выполнения, а существующие файлы просто перезаписываются.

12.3. Копирование файла с использованием стандартной библиотеки языка C

Как видно из текста программы 12.2, стандартная библиотека C поддерживает объекты потоков ввода/вывода **FILE**, которые напоминают, несмотря на меньшую общность, объекты Windows **HANDLE**, представленные в программе 12.3.

Листинг 12.2. Копирование файлов с использованием библиотеки C

```
/* Программа копирования файлов cpC.
   Реализация, использующая библиотеку C. */
/* cpC file1 file2: Копировать файл1 в файл2. */
# include <iostream.h>
# include <stdio.h>
# include <errno.h>
# define BUF_SIZE 256
int main (int argc, char *argv [])
{
FILE *in_file, *out_file;
char rec [BUF_SIZE];
size_t bytes_in, bytes_out;
if (argc != 3) {
     cout<< "Use: cpC file1 file2\n";
     return 1;
}
in_file = fopen (argv [1], "rb");
if (in_file == NULL) {
     perror (argv [1]);
     return 2;
}
out_file = fopen (argv [2], "wb");
if (out_file == NULL) {
     perror (argv [2]);
```

```
    return 3;
}
/* Обработать входной файл по одной записи за один раз.*/
while((bytes_in = fread (rec, 1, BUF_SIZE, in_file))>0) {
    bytes_out = fwrite (rec, 1, bytes_in, out_file);
    if (bytes_out != bytes_in) {
        perror ("Неустранимая ошибка записи.");
        return 4;
    }
}
fclose (in_file);
fclose (out_file);
return 0;
}
```

Предположим, что с помощью приведенной программы мы хотим содержание файла **my.doc** скопировать в файл **my2.doc**. Тогда в командной строке нужно набрать следующее

```
cpC my.doc my2.doc
```

Расширения имен файлов нужно указывать обязательно. Исполняемый файл должен иметь название **cpC.exe.**

Этот простой пример может служить наглядной иллюстрацией ряда общепринятых допущений и соглашений программирования, которые не всегда применяются в Windows.

1. Объекты открытых файлов идентифицируются указателями на структуры **FILE** (в UNIX используются целочисленные дескрипторы файлов). Указателю **NULL** соответствует несуществующий объект. По сути, указатели являются разновидностью дескрипторов объектов открытых файлов.
2. В вызове функции **fopen** указывается, каким образом должен обрабатываться файл – как текстовый или как двоичный. В

текстовых файлах содержатся специфические для каждой системы последовательности символов, используемых, например, для обозначения конца строки. Во многих системах, включая Windows, в процессе выполнения операций ввода/вывода каждая из таких последовательностей автоматически преобразуется в нулевой символ, который интерпретируется в языке C как метка конца строки, и наоборот. В нашем примере оба файла открываются как двоичные.

3. Диагностика ошибок реализуется с помощью функции **`perror`**, которая, в свою очередь, получает информацию относительно природы сбоя, возникающего при вызове функции **`fopen`**, из глобальной переменной **`errno`**. Вместо этого можно было бы воспользоваться функцией **`ferror`**, возвращающей код ошибки, ассоциированный не с системой, а с объектом **`FILE`**.
4. Функции **`fread`** и **`fwrite`** возвращают количество обработанных байтов, непосредственно, а не через аргумент, что оказывает существенное влияние на логику организации программы. Неотрицательное возвращаемое значение говорит об успешном выполнении операции чтения, тогда как нулевое – о попытке чтения метки конца файла.
5. Функция **`fclose`** может применяться лишь к объектам типа **`FILE`** (аналогичное утверждение справедливо и в отношении дескрипторов файлов UNIX).
6. Операции ввода/вывода осуществляются в синхронном режиме, то есть прежде чем программа сможет выполняться дальше, она должна дождаться завершения операции ввода/вывода.

Преимуществом реализации, использующей библиотеку C, является ее переносимость на платформы UNIX, Windows, а также другие системы, которые поддерживают стандарт ANSI C. Кроме того, в том, что касается

производительности, вариант, использующий функции ввода/вывода библиотеки C, ничуть не уступает другим вариантам реализации. Тем не менее, в этом случае программы вынуждены ограничиваться синхронными операциями ввода/вывода, хотя влияние этого ограничения будет несколько ослаблено использованием потоков Windows.

Как и их эквиваленты в UNIX, программы, основанные на функциях для работы с файлами, входящих в библиотеку C, способны выполнять операции произвольного доступа к файлам (с использованием функции **`fseek`** или, в случае текстовых файлов, функций **`fsetpos`** и **`fgetpos`**), но это является уже потолком сложности для функций ввода/вывода стандартной библиотеки C, выше которого они подняться не могут. Вместе с тем, Visual C++ предоставляет нестандартные расширения, способные, например, поддерживать блокирование файлов. Наконец, библиотека C не позволяет управлять средствами защиты файлов.

Резюмируя, можно сделать вывод, что если простой синхронный файловый или консольный ввод/вывод – это все, что вам надо, то для написания переносимых программ, которые будут выполняться под управлением Windows, следует использовать библиотеку C.

12.4. Копирование файла с использованием Windows

В программе 12.3 решается та же задача копирования файлов, но делается это с помощью Windows API.

Листинг 12.3. Копирование файлов с использованием Windows API, первая реализация

```
/* Программа копирования файлов cpW.
   Реализация, использующая Windows.*/
/* cpW file1 file2: Копировать файл1 в файл2.*/
#include <windows.h>
```

```
#include <stdio.h>
#include <iostream.h>
#define BUF_SIZE 256
int main(int argc, LPTSTR argv[]) {
    HANDLE hIn, hOut;
    DWORD nIn, nOut;
    CHAR Buffer [BUF_SIZE];
    if(argc != 3) {
        cout<<"Use: cpW file1 file2\n";
        return  1;
    }
    hIn = CreateFile(argv[1], GENERIC_READ, 0, NULL,
OPEN_EXISTING, 0, NULL);
    if (hIn==INVALID_HANDLE_VALUE) {
        cout<<"Невозможно открыть входной файл.
Ошибка: %x\n", GetLastError ();
        return  2;
    }
    hOut = CreateFile (argv[2], GENERIC_WRITE, 0,
NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
    if(hOut==INVALID_HANDLE_VALUE) {
        cout<<"Невозможно открыть выходной файл.
Ошибка: %x\n", GetLastError ();
        return 3;
    }
    while(ReadFile(hIn, Buffer, BUF_SIZE, &nIn,
NULL)&&nIn > 0) {
        WriteFile (hOut, Buffer, nIn, &nOut, NULL);
        if (nIn != nOut) {
            cout<<"Неустранимая ошибка записи: %x\n",
GetLastError ();
```

```
            return 4;
        }
    }
    CloseHandle (hIn);
    CloseHandle (hOut);
    return 0;
}
```

Этот пример иллюстрирует некоторые особенности программирования в среде Windows.

1. В программу всегда включается файл <windows.h>, в котором содержатся все необходимые определения функций и типов данных Windows.
2. Все объекты Windows идентифицируются переменными типа **`Handle`**, причем для большинства объектов можно использовать одну и ту же общую функцию **`CloseHandle`**.
3. Рекомендуется закрывать все ранее открытые дескрипторы, если в необходимость в них отпала, чтобы освободить ресурсы. В то же время, при завершении процессов относящиеся к ним дескрипторы автоматически закрываются ОС, и если не остается ни одного дескриптора, ссылающегося на какой-либо объект, то ОС уничтожает этот объект и освобождает соответствующие ресурсы. Как правило, файлы подобным способом не уничтожаются.
4. Windows определяет многочисленные символические константы и флаги. Обычно они имеют длинные имена, нередко поясняющие назначение данного объекта. В качестве типичного примера можно привести имена **`INVALID_HANDLE_VALUE`** и **`GENERIC_READ`**.

5. Функции **ReadFile** и **WriteFile** возвращают булевские значения, а не количества обработанных байтов, для передачи которых используются аргументы функций. Это определенным образом изменяет логику организации работы циклов. Нулевое значение счетчика байтов указывает на попытку чтения метки конца файла и не считается ошибкой.
6. Функция **GetLastError** позволяет получать в любой точке программы коды системных ошибок, представляемые значениями типа **DWORD**. В листинге 12.3 показано, как организовать вывод генерируемых Windows текстовых сообщений об ошибках.
7. Windows NT обладает более мощной системой защиты файлов, чем предшествующие ей версии. В данном примере защита выходного файла не обеспечивается.
8. Такие функции, как **CreateFile**, обладают богатым набором дополнительных параметров, но в данном примере использованы значения по умолчанию.

12.5. Копирование файла с использованием вспомогательной функции Windows

Для повышения удобства работы в Windows предусмотрено множество вспомогательных функций (convenience functions), которые, объединяя в себе несколько других функций, обеспечивают выполнение часто встречающихся задач программирования. В некоторых случаях использование этих функций может приводить к повышению производительности. Например, благодаря применению функции **CopyFile** значительно упрощается программа копирования файлов (листинг 12.4). Помимо всего прочего, это избавляет нас от необходимости заботиться о буфере, размер которого в двух предыдущих программах произвольно устанавливался равным 256.

Листинг 12.4. Копирование файлов с использованием вспомогательной функции Windows

```
/* Программа копирования файлов cpCF.
   Реализация, в которой для повышения удобства
   использования и производительности программы
   используется функция Windows CopyFile. */
/* cpCF файл1 файл2: Копировать файл1 в файл2. */
#include <windows.h>
#include <stdio.h>
#include <iostream.h>
int main (int argc, LPTSTR argv[]) {
 if (argc != 3) {
  cout<<"Use: cpCF file1 file2\n";
  return 1;
 }
 if(!CopyFile (argv[1], argv[2], FALSE))  {
  cout<<"Ошибка при выполнении функции CopyFile:
  %x\n", GetLastError ();
  return 2;
  }
  return  0;
}
```

12.6. О целесообразности использования стандартной библиотеки C

В каких случаях при обработке файлов можно обойтись библиотекой C, а в каких необходимо использовать системные вызовы Windows? Тот же вопрос можно задать и в отношении использования потоков (streams) ввода/вывода C++ или системы ввода/вывода, которая предоставляется

платформой .NET. Простых ответов на эти вопросы не существует, но если во главу угла поставить переносимость программ на платформы, отличные от Windows, то в тех случаях, когда приложению требуется только обработка файлов, а не, например, управление процессами или другие специфические возможности Windows, предпочтение следует отдавать библиотеке C и потокам ввода/вывода C++.

К числу возможностей Windows, не поддерживаемых библиотекой C, относятся блокирование и отображение файлов (необходимое для разделения общих областей памяти), асинхронный ввод/вывод, произвольный доступ к файлам чрезвычайно крупных размеров (4 Гбайт и выше) и взаимодействие между процессами.

Лекция № 13. ОПЕРАЦИИ ОТКРЫТИЯ, ЧТЕНИЯ, ЗАПИСИ И ЗАКРЫТИЯ ФАЙЛОВ

13.1. Создание и открытие файла

Первой функцией Windows, которую мы подробно опишем, является функция **CreateFile**, используемая как для создания новых, так и для открытия существующих файлов. Для этой функции, как и для всех остальных, сначала приводится прототип, а затем обсуждаются соответствующие параметры и порядок работы с ней.

Простейшее использование функции **CreateFile** иллюстрирует приведенный в предыдущей лекции пример ознакомительной Windows-программы (программа 12.3), содержащей два вызова функций, в которых для параметров **dwShareMode**, **lpSecurityAttributes** и **hTemplateFile** были использованы значения по умолчанию. Параметр **dwAccess** может принимать значения **GENERIC_READ** и **GENERIC_WRITE**.

```
HANDLE CreateFile (
     LPCTSTR lpName,
     DWORD dwAccess,
     DWORD dwShareMode,
     LPSECURITY_ATTRIBUTES lpSecurityAttributes,
     DWORD dwCreate,
     DWORD dwAttrsAndFlags,
     HANDLE hTemplateFile)
```

Возвращаемое значение: в случае успешного выполнения – дескриптор открытого файла (типа **HANDLE**), иначе – **INVALID_HANDLE_VALUE**.

Имена параметров иллюстрируют некоторые соглашения Windows. Префикс **dw** используется в именах параметров типа **DWORD** (32-битовые целые без знака), в которых могут храниться флаги или числовые значения, например счетчики, тогда как префикс **lpsz** (длинный указатель на строку, завершающуюся нулем), или в упрощенной форме – **lp**, используется для строк, содержащих пути доступа, либо иных строковых значений, хотя документация Microsoft в этом отношении не всегда последовательна. В некоторых случаях для правильного определения типа данных вам придется обратиться к здравому смыслу или внимательно прочесть документацию.

lpName – указатель на строку с завершающим нулевым символом, содержащую имя файла, канала или любого другого именованного объекта, который необходимо открыть или создать. Допустимое количество символов при указании путей доступа обычно ограничивается значением **MAX_PATH (260)**, однако в Windows NT это ограничение можно обойти, поместив перед именем префикс **\\?**, что обеспечивает возможность использования очень длинных имен (с числом символов вплоть до 32 К). Сам префикс в имя не входит. О типе данных **LPCTSTR** говорится в одном из последующих разделов, а пока достаточно знать, что он относится к строковым данным.

dwAccess – определяет тип доступа к файлу – чтение или запись, что соответственно указывается флагами **GENERIC_READ** и **GENERIC_WRITE**. Ввиду отсутствия флаговых значений **READ** и **WRITE** использование префикса **GENERIC_** может показаться излишним, однако он необходим для совместимости с именами макросов, определенных в заголовочном файле Windows **WINNT.H**. Мы еще неоднократно столкнемся с именами, которые кажутся длиннее, чем необходимо.

Указанные значения можно объединять операцией поразрядного "или", и тогда для получения доступа к файлу как по чтению, так и по записи, следует воспользоваться таким выражением:

```
GENERIC_READ | GENERIC_WRITE
```

dwShareMode – может объединять с помощью операции поразрядного "или" следующие значения:

- **0** – запрещает разделение (совместное использование) файла. Более того, открытие второго дескриптора для данного файла запрещено даже в рамках одного и того же вызывающего процесса.
- **FILE_SHARE_READ** – другим процессам, включая и тот, который осуществил данный вызов функции, разрешается открывать этот файл для параллельного доступа по чтению.
- **FILE_SHARE_WRITE** – разрешает параллельную запись в файл.

Используя блокирование файла или иные механизмы, программист должен самостоятельно позаботиться об обработке ситуаций, в которых осуществляются одновременно несколько попыток записи в одно и то же место в файле.

lpSecurityAttributes – указывает на структуру **SECURITY_ATTRIBUTES**. При вызовах функции **CreateFile** и всех остальных функций достаточно использовать значение **NULL**.

dwCreate – конкретизирует запрашиваемую операцию: создать новый файл, перезаписать существующий файл и тому подобное. Может принимать одно из приведенных ниже значений, которые могут объединяться при помощи операции поразрядного "или" языка С.

- **CREATE_NEW** – создать новый файл; если указанный файл уже существует, выполнение функции завершается неудачей.
- **CREATE_ALWAYS** – создать новый файл; если указанный файл уже существует, функция перезапишет его.

- **OPEN_E_XI STING** – открыть файл; если указанный файл не существует, выполнение функции завершается неудачей.
- **OPEN_ALWAYS** – открыть файл; если указанный файл не существует, функция создаст его.
- **TRUNCATE_EXISTING** – открыть файл; размер файла будет установлен равным нулю. Уровень доступа к файлу, установленный параметром **dwAccess**, должен быть не ниже **GENERIC_WRITE**. Если указанный файл существует, его содержимое будет уничтожено. В отличие от случая **CREATENEW** выполнение функции будет успешным даже в тех случаях, когда указанный файл не существует.

dwAttrsAndFlags – позволяет указать атрибуты файла и флаги. Всего имеется 16 флагов и атрибутов. Атрибуты являются характеристиками файла, а не открытого дескриптора, и игнорируются, если открывается существующий файл. Некоторые из наиболее важных флаговых значений приводятся ниже.

- **FILE_ATTRIBUTE_NORMAL** – этот атрибут можно использовать лишь при условии, что одновременно с ним не устанавливаются никакие другие атрибуты (тогда как для всех остальных флагов одновременная установка допускается).
- **FILE_ATTRIBUTE_READONLY** – этот атрибут запрещает приложениям осуществлять запись в данный файл или удалять его.
- **FILE_FLAG_DELETE_ON_CLOSE** – этот флаг полезно применять в случае временных файлов. Файл будет удален сразу же после закрытия последнего из его открытых дескрипторов.
- **FILE_FLAG_OVERLAPPED** – этот флаг играет важную роль при выполнении операций асинхронного ввода/вывода.

Кроме того, существует несколько дополнительных флагов, позволяющих уточнить способ обработки файла и облегчить реализации

Windows, оптимизацию производительности и обеспечение целостности файлов.

- **FILE_FLAG_WRITE_THROUGH** – устанавливает режим сквозной записи промежуточных данных непосредственно в файл на диске, минуя кэш.
- **FILE_FLAG_NO_BUFFERING** – устанавливает режим отсутствия промежуточной буферизации или кэширования, при котором обмен данными происходит непосредственно с буферами данных программы, указанными при вызове функций ReadFile или WriteFile (описаны далее). Соответственно требуется, чтобы начала программных буферов совпадали с границами секторов, а их размеры были кратными размеру сектора тома. Чтобы определить размер сектора при указании этого флага, можно воспользоваться функцией **GetDiskFreeSpace**.
- **FILE_FLAG_RANDOM_ACCESS** – предполагается открытие файла для произвольного доступа; Windows будет пытаться оптимизировать кэширование файла применительно к этому виду доступа.
- **FILE_FLAG_SEQUENTIAL_SCAN** – предполагается открытие файла для последовательного доступа; Windows будет пытаться оптимизировать кэширование файла применительно к этому виду доступа. Оба последних режима реализуются системой лишь по мере возможностей.

hTemplateFile – дескриптор с правами доступа **GENERIC_READ** к шаблону файла, предоставляющему расширенные атрибуты, которые будут применены к создаваемому файлу вместо атрибутов, указанных в параметре **dwAttrsAndFlags**. Обычно значение этого параметра устанавливается равным **NULL**. При открытии существующего файла параметр **hTemplateFile** игнорируется. Этот параметр используется в тех случаях, когда требуется, чтобы атрибуты вновь создаваемого файла совпадали с атрибутами уже существующего файла.

Оба вызова функции **CreateFile** в программе 12.3 максимально упрощены за счет использования для параметров значений по умолчанию, и, тем не менее, они вполне справляются со своими задачами. В обоих случаях было бы целесообразно использовать флаг **FILE_FLAG_SEQUENTIAL_SCAN**.

Для данного файла могут быть одновременно открыты несколько дескрипторов, если только это разрешается атрибутами совместного доступа и защиты файла. Открытые дескрипторы могут принадлежать одному и тому же или различным процессам.

В Windows Server 2003 предоставляется функция **ReOpenFile**, которая возвращает новый дескриптор с иными флагами, правами доступа и прочим, нежели те, которые были указаны при первоначальном открытии файла, если только это не приводит к возникновению конфликта между новыми и прежними правами доступа.

13.2. Закрытие файла

Для закрытия объектов любого типа, объявления недействительными их дескрипторов и освобождения системных ресурсов почти во всех случаях используется одна и та же универсальная функция. Исключения из этого правила будут оговариваться отдельно. Закрытие дескриптора сопровождается уменьшением на единицу счетчика ссылок на объект, что делает возможным удаление таких не хранимых постоянно (nonpersistent) объектов, как временные файлы или события. При выходе из программы система автоматически закрывает все открытые дескрипторы, однако лучше все же, чтобы программа самостоятельно закрывала свои дескрипторы перед тем, как завершить работу.

Попытки закрытия недействительных дескрипторов или повторного закрытия одного и того же дескриптора приводят к исключениям. Не только излишне, но и не следует закрывать дескрипторы стандартных устройств.

Рассмотрим следующий пример.

```
BOOL CloseHandle (HANDLE hObject)
```

Возвращаемое значение: в случае успешного выполнения функции – **TRUE**, иначе – **FALSE**.

Функции UNIX, сопоставимые с рассмотренными выше, отличаются от них в нескольких отношениях. Функция (системный вызов) UNIX **open** возвращает целочисленный дескриптор (descriptor) файла, а не дескриптор типа **HANDLE**, причем для указания всех параметров доступа, разделения и создания файлов, а также атрибутов и флагов используется единственный целочисленный параметр **of lag**. Возможные варианты выбора, доступные в обеих системах, перекрываются, однако набор опций, предлагаемый Windows, отличается большим разнообразием.

В UNIX отсутствует параметр, эквивалентный параметру **dwShareMode**. Файлы UNIX всегда являются разделяемыми. В обеих системах при создании файла используется информация, касающаяся его защиты. В UNIX для задания хорошо известных разрешений на доступ к файлу для владельца, членов группы и прочих пользователей используется аргумент **mode**. Функция **close**, хотя ее и можно сопоставить с функцией **CloseHandle**, отличается от последней меньшей универсальностью. Функции библиотеки С, описанные в заголовочном файле **<stdio.h>**, используют объекты **FILE**, которые можно поставить в соответствие дескрипторам (дисковые файлы, терминалы, ленточные устройства и тому подобные), связанным с потоками. Параметр **mode** функции **fopen** позволяет указать, должны ли содержащиеся в файле данные обрабатываться как двоичные или как текстовые. Имеются также опции открытия файла в режиме "только чтение", обновления файла, присоединения к другому файлу и так далее. Функция **freopen** обеспечивает возможность повторного использования объектов **FILE** без их предварительного закрытия. Средства для задания параметров защиты стандартной библиотекой С не предоставляются. Для закрытия объектов типа **FILE** предназначена функция **fclose**. Имена большинства функций

стандартной библиотеки C, предназначенных для работы с объектами **FILE**, снабжены префиксом "f".

13.3. Чтение файла

Чтение файла выполняется с помощью следующей функции:

```
BOOL ReadFile (
    HANDLE hFile,
    LPVOID lpBuffer,
    DWORD nNumberOfBytesToRead,
    LPDWORD lpNumberOfBytesRead,
    LPOVERLAPPED lpOverlapped)
```

Возвращаемое значение: в случае успешного выполнения (которое считается таковым, даже если не был считан ни один байт из-за попытки чтения с выходом за пределы файла) – **TRUE**, иначе – **FALSE**.

Пока мы будем предполагать, что дескрипторы файлов создаются без указания флага перекрывающегося ввода/вывода **FILE_FLAG_OVERLAPPED** в параметре **dwAttrsAndFlags**. В этом случае функция **ReadFile** начинает чтение с текущей позиции указателя файла, и указатель файла сдвигается на число считанных байтов.

Если значения дескриптора файла или иных параметров, используемых при вызове функции, оказались недействительными, возникает ошибка, и функция возвращает значение **FALSE**. Попытка выполнения чтения в ситуациях, когда указатель файла позиционирован в конце файла, не приводит к ошибке; вместо этого количество считанных байтов **(*lpNumberOfBytesRead)** устанавливается равным **0**.

Параметры

Описательные имена переменных и естественный порядок расположения параметров во многом говорят сами за себя. Тем не менее, ниже приводятся некоторые краткие пояснения.

hFile – дескриптор считываемого файла, который должен быть создан с правами доступа **GENERIC_READ**. **lpBuffer** является указателем на буфер в памяти, куда помещаются считываемые данные. **nNumberOfBytesToRead** – количество байт, которые должны быть считаны из файла.

lpNumberOfBytesRead – указатель на переменную, предназначенную для хранения числа байт, которые были фактически считаны в результате вызова функции **ReadFile**. Этот параметр может принимать нулевое значение, если перед выполнением чтения указатель файла был позиционирован в конце файла или если во время чтения возникли ошибки, а также после чтения из именованного канала, работающего в режиме обмена сообщениями, если переданное сообщение имеет нулевую длину.

lpOverlapped – указатель на структуру **OVERLAPPED**. На данном этапе просто устанавливайте значение этого параметра равным **NULL**.

13.4. Запись в файл

Запись в файл выполняется с помощью следующей функции:

```
BOOL WritedFile (
    HANDLE hFile,
    LPCVOID lpBuffer,
    DWORD nNumberOfBytesToWrite,
    LPDWORD lpNumberOfBytesWritten,
    LPOVERLAPPED lpOverlapped)
```

Возвращаемое значение: в случае успешного выполнения – **TRUE**, иначе – **FALSE**.

Все параметры этой функции вам уже знакомы. Заметьте, что успешное выполнение записи еще не говорит о том, что данные действительно оказались записанными на диск, если только при создании файла с помощью функции **`CreateFile`** не был использован флаг **`FILE_FLAG_WRITE_THROUGH`**. Если во время вызова функции указатель файла был позиционирован в конце файла, Windows увеличит длину существующего файла.

Функции **`ReadFileGather`** и **`WriteFileGather`** позволяют выполнять операции чтения и записи с использованием набора буферов различного размера.

Лекция № 14. ОРГАНИЗАЦИЯ ПРОГРАММНЫХ ПОТОКОВ

Рассмотрим простейший пример, в котором необходимо подсчитать количество пробелов в текстовых файлах, имена которых должны указываться в командной строке.

Поскольку нас интересует работа с параллельными задачами, пусть при выполнении программы для каждого из перечисленных в командной строке файлов создается свой процесс (или поток выполнения), который параллельно с другими процессами (потоками) производит работу по подсчету пробелов в «своем» файле. Результатом работы программы будет являться список файлов с подсчитанным количеством пробелов для каждого.

Приведенная ниже реализация программы решения данной задачи не является единственно возможной. В данном случае рассматривается наиболее характерный вариант.

Основная программа запускает потоки и ждет окончания их выполнения. Мы имеем всего один вычислительный процесс, но используем мультизадачные возможности операционной системы Windows.

Листинг 14.1. Подсчет количества пробелов в текстовых файлах

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

//Название: processFile
//Описание: исполняемый код потока
//Входные параметры: lpFileName – имя файла
```

```
//Выходные параметры: нет
DWORD processFile(LPVOID lpFileName) {
  HANDLE handle;  //дескриптор файла
  DWORD numRead, total = 0;
  char buf;

  //Запрос к ОС на открытие файла (только для чтения)
  handle = CreateFile((LPCTSTR)lpFileName, GENERIC_READ,
     FILE_SHARE_READ, NULL, OPEN_EXISTING,
     FILE_ATTRIBUTE_NORMAL, NULL);
//Цикл чтения до конца файла
  do {
     //Чтение одного символа из файла
     ReadFile(handle,(LPVOID) &buf, 1, &numRead, NULL);
     if (buf == 0x20) total++;
  } while (numRead>0);

  fprintf(stderr,"( ThreadId: %Lu), File %s, spaces =
  %d\n",
     GetCurrentThreadId(), lpFileName, total);

  //Закрытие файла
  CloseHandle( handle );
  return(0);
}

//Название: main
//Описание: главная программа
//Входные параметры: список имен файлов для обработки
//Выходные параметры: нет
int main( int argc, char *argv[]) {
```

```
  int i;
  DWORD pid;
  HANDLE hThrd[255];    //массив ссылок на потоки
  //для всех файлов, перечисленных в командной строке
  for (i=0; i<(argc-1); i++) {
     //запуск потока – обработка одного файла
     hThrd[i]=CreateThread( NULL, 0x4000,
          (LPTHREAD_START_ROUTINE) processFile,
          (LPVOID) argv[i+1], 0, &pid);
     fprintf( stdout, "processFile started (HND=%d)\n",
  hThrd[i]);
  }

//ожидание окончания выполнения всех запущенных потоков
  WaitForMultipleObjects( argc-1, hThrd, true,
  INFINITE);
  return(0);
}
```