Модуль 1. Основные принципы построения ОС

Лекции 3 ч.

Тема 1. История развития и принципы построения ОС

Тема 2. Классификация ОС

Тема 3. Архитектура ОС. Основные компоненты современных ОС.

1. Лекция: История развития и принципы построения ОС

Операционная система (ОС) – это программа, которая обеспечивает возможность рационального использования оборудования компьютера удобным для пользователя образом. Вводная лекция рассказывает о предмете, изучаемом в рамках настоящего курса. Вначале мы попытаемся ответить на вопрос, что такое ОС. Затем последует анализ эволюции ОС и рассказ о возникновении основных концепций и компонентов современных ОС. В заключение будет представлена классификация ОС с точки зрения особенностей архитектуры и использования ресурсов компьютера.

Структура вычислительной системы

Из чего состоит любая вычислительная система? Во-первых, из того, что в англоязычных странах принято называть словом hardware, или техническое обеспечение: процессор, память, монитор, дисковые устройства и т.д., объединенные магистральным соединением, которое называется шиной. Некоторые сведения об архитектуре компьютера имеются в приложении 1 к настоящей лекции.

Во-вторых, вычислительная система состоит из программного обеспечения. Все программное обеспечение принято делить на две части: прикладное и системное. К прикладному программному обеспечению, как правило, относятся разнообразные банковские и прочие бизнес-программы, игры, текстовые процессоры и т. п. Под системным программным обеспечением обычно понимают программы, способствующие функционированию и разработке прикладных программ. Надо сказать, что деление на прикладное и системное программное обеспечение является отчасти условным и зависит от того, кто осуществляет такое деление. Так, обычный пользователь, неискушенный в программировании, может считать Microsoft Word системной программой, а, с точки зрения программиста, это – приложение. Компилятор языка Си для обычного программиста – системная программа, а для системного – прикладная. Несмотря на эту нечеткую грань, данную ситуацию можно отобразить в виде последовательности слоев (см. рис. 1.1), выделив отдельно наиболее общую часть системного программного обеспечения – операционную систему:

Рис. 1.1. Слои программного обеспечения компьютерной системы

Что такое ОС

Операционная система как виртуальная машина

При разработке ОС широко применяется абстрагирование, которое является важным методом упрощения и позволяет сконцентрироваться на взаимодействии высокоуровневых компонентов системы, игнорируя детали их реализации. В этом смысле ОС представляет собой интерфейс между пользователем и компьютером.

Архитектура большинства компьютеров на уровне машинных команд очень неудобна для использования прикладными программами. Например, работа с диском предполагает знание внутреннего устройства его электронного компонента – контроллера для ввода команд вращения диска, поиска и форматирования дорожек, чтения и записи секторов и т. д. Ясно, что средний программист не в состоянии учитывать все особенности работы оборудования (в современной терминологии – заниматься разработкой драйверов устройств), а должен иметь простую высокоуровневую абстракцию, скажем представляя информационное пространство диска как набор файлов. Файл можно открывать для чтения или записи, использовать для получения или сброса информации, а потом закрывать. Это концептуально проще, чем заботиться о деталях перемещения головок дисков или организации работы мотора. Аналогичным образом, с помощью простых и ясных абстракций, скрываются от программиста все ненужные подробности организации прерываний, работы таймера, управления памятью и т. д. Более того, на современных вычислительных комплексах можно создать иллюзию неограниченного размера оперативной памяти и числа процессоров. Всем этим занимается операционная система. Таким образом, операционная система представляется пользователю виртуальной машиной, с которой проще иметь дело, чем непосредственно с оборудованием компьютера.

Операционная система как менеджер ресурсов

Операционная система предназначена для управления всеми частями весьма сложной архитектуры компьютера. Представим, к примеру, что произойдет, если несколько программ, работающих на одном компьютере, будут пытаться одновременно осуществлять вывод на принтер. Мы получили бы мешанину строчек и страниц, выведенных различными

программами. Операционная система предотвращает такого рода хаос за счет буферизации информации, предназначенной для печати, на диске и организации очереди на печать. Для многопользовательских компьютеров необходимость управления ресурсами и их защиты еще более очевидна. Следовательно, операционная система, как менеджер ресурсов, осуществляет упорядоченное и контролируемое распределение процессоров, памяти и других ресурсов между различными программами.

Операционная система как защитник пользователей и программ

Если вычислительная система допускает совместную работу нескольких пользователей, то возникает проблема организации их безопасной деятельности. Необходимо обеспечить сохранность информации на диске, чтобы никто не мог удалить или повредить чужие файлы. Нельзя разрешить программам одних пользователей произвольно вмешиваться в работу программ других пользователей. Нужно пресекать попытки несанкционированного использования вычислительной системы. Всю эту деятельность осуществляет операционная система как организатор безопасной работы пользователей и их программ. С такой точки зрения операционная система представляется системой безопасности государства, на которую возложены полицейские и контрразведывательные функции.

Операционная система как постоянно функционирующее ядро

Наконец, можно дать и такое определение: операционная система – это программа, постоянно работающая на компьютере и взаимодействующая со всеми прикладными программами. Казалось бы, это абсолютно правильное определение, но, как мы увидим дальше, во многих современных операционных системах постоянно работает на компьютере лишь часть операционной системы, которую принято называть ее ядром.

Как мы видим, существует много точек зрения на то, что такое операционная система. Невозможно дать ей адекватное строгое определение. Нам проще сказать не что есть операционная система, а для чего она нужна и что она делает. Для выяснения этого вопроса рассмотрим историю развития вычислительных систем.

Краткая история эволюции вычислительных систем

Мы будем рассматривать историю развития именно вычислительных, а не операционных систем, потому что hardware и программное обеспечение эволюционировали совместно, оказывая взаимное влияние друг на друга. Появление новых технических возможностей приводило к прорыву в области создания удобных, эффективных и безопасных программ, а свежие идеи в программной области стимулировали поиски новых технических решений. Именно эти критерии – удобство, эффективность и безопасность – играли роль факторов естественного отбора при эволюции вычислительных систем.

Первый период (1945–1955 гг.). Ламповые машины. Операционных систем нет

Мы начнем исследование развития компьютерных комплексов с появления электронных вычислительных систем (опуская историю механических и электромеханических устройств).

Первые шаги в области разработки электронных вычислительных машин были предприняты в конце Второй мировой войны. В середине 40-х были созданы первые ламповые вычислительные устройства и появился принцип программы, хранящейся в памяти машины (John Von Neumann, июнь 1945 г.). В то время одна и та же группа людей участвовала и в проектировании, и в эксплуатации, и в программировании вычислительной

машины. Это была скорее научно-исследовательская работа в области вычислительной техники, а не регулярное использование компьютеров в качестве инструмента решения каких-либо практических задач из других прикладных областей. Программирование осуществлялось исключительно на машинном языке. Об операционных системах не было и речи, все задачи организации вычислительного процесса решались вручную каждым программистом с пульта управления. За пультом мог находиться только один пользователь. Программа загружалась в память машины в лучшем случае с колоды перфокарт, а обычно с помощью панели переключателей.

Вычислительная система выполняла одновременно только одну операцию (ввод-вывод или собственно вычисления). Отладка программ велась с пульта управления с помощью изучения состояния памяти и регистров машины. В конце этого периода появляется первое системное программное обеспечение: в 1951–1952 гг. возникают прообразы первых компиляторов с символических языков (Fortran и др.), а в 1954 г. Nat Rochester разрабатывает Ассемблер для IBM-701.

Существенная часть времени уходила на подготовку запуска программы, а сами программы выполнялись строго последовательно. Такой режим работы называется последовательной обработкой данных. В целом первый период характеризуется крайне высокой стоимостью вычислительных систем, их малым количеством и низкой эффективностью использования.

Второй период (1955 г.–начало 60-х). Компьютеры на основе транзисторов. Пакетные операционные системы

С середины 50-х годов начался следующий период в эволюции вычислительной техники, связанный с появлением новой технической базы – полупроводниковых элементов. Применение транзисторов вместо часто перегоравших электронных ламп привело к повышению надежности компьютеров. Теперь машины могут непрерывно работать достаточно долго, чтобы на них можно было возложить выполнение практически важных задач. Снижается потребление вычислительными машинами электроэнергии, совершенствуются системы охлаждения. Размеры компьютеров уменьшились. Снизилась стоимость эксплуатации и обслуживания вычислительной техники. Началось использование ЭВМ коммерческими фирмами. Одновременно наблюдается бурное развитие алгоритмических языков (LISP, COBOL, ALGOL-60, PL-1 и т.д.). Появляются первые настоящие компиляторы, редакторы связей, библиотеки математических и служебных подпрограмм. Упрощается процесс программирования. Пропадает необходимость взваливать на одних и тех же людей весь процесс разработки и использования компьютеров. Именно в этот период происходит разделение персонала на программистов и операторов, специалистов по эксплуатации и разработчиков вычислительных машин.

Изменяется сам процесс прогона программ. Теперь пользователь приносит программу с входными данными в виде колоды перфокарт и указывает необходимые ресурсы. Такая колода получает название задания. Оператор загружает задание в память машины и запускает его на исполнение. Полученные выходные данные печатаются на принтере, и пользователь получает их обратно через некоторое (довольно продолжительное) время.

Смена запрошенных ресурсов вызывает приостановку выполнения программ, в результате процессор часто простаивает. Для повышения эффективности использования компьютера задания с похожими ресурсами начинают собирать вместе, создавая пакет заданий.

Появляются первые системы пакетной обработки, которые просто автоматизируют запуск одной программы из пакета за другой и тем самым увеличивают коэффициент загрузки процессора. При реализации систем пакетной обработки был разработан формализованный язык управления заданиями, с помощью которого программист сообщал системе и оператору, какую работу он хочет выполнить на вычислительной машине. Системы пакетной обработки стали прообразом современных операционных систем, они были первыми системными программами, предназначенными для управления вычислительным процессом.

Третий период (начало 60-х – 1980 г.). Компьютеры на основе интегральных микросхем.

Первые многозадачные ОС

Следующий важный период развития вычислительных машин относится к началу 60-х – 1980 г. В это время в технической базе произошел переход от отдельных полупроводниковых элементов типа транзисторов к интегральным микросхемам. Вычислительная техника становится более надежной и дешевой. Растет сложность и количество задач, решаемых компьютерами. Повышается производительность процессоров.

Повышению эффективности использования процессорного времени мешает низкая скорость работы механических устройств ввода-вывода (быстрый считыватель перфокарт мог обработать 1200 перфокарт в минуту, принтеры печатали до 600 строк в минуту). Вместо непосредственного чтения пакета заданий с перфокарт в память начинают использовать его предварительную запись, сначала на магнитную ленту, а затем и на диск. Когда в процессе выполнения задания требуется ввод данных, они читаются с диска. Точно так же выходная информация сначала копируется в системный буфер и записывается на ленту или диск, а печатается только после завершения задания. Вначале действительные операции ввода-вывода осуществлялись в режиме off-line, то есть с использованием других, более простых, отдельно стоящих компьютеров. В дальнейшем они начинают выполняться на том же компьютере, который производит вычисления, то есть в режиме on-line. Такой прием получает название spooling (сокращение от Simultaneous Peripheral Operation On Line) или подкачки-откачки данных. Введение техники подкачки-откачки в пакетные системы позволило совместить реальные операции ввода-вывода одного задания с выполнением другого задания, но потребовало разработки аппарата прерываний для извещения процессора об окончании этих операций.

Магнитные ленты были устройствами последовательного доступа, то есть информация считывалась с них в том порядке, в каком была записана. Появление магнитного диска, для которого не важен порядок чтения информации, то есть устройства прямого доступа, привело к дальнейшему развитию вычислительных систем. При обработке пакета заданий на магнитной ленте очередность запуска заданий определялась порядком их ввода. При обработке пакета заданий на магнитном диске появилась возможность выбора очередного выполняемого задания. Пакетные системы начинают заниматься планированием заданий: в зависимости от наличия запрошенных ресурсов, срочности вычислений и т.д. на счет выбирается то или иное задание.

Дальнейшее повышение эффективности использования процессора было достигнуто с помощью мультипрограммирования. Идея мультипрограммирования заключается в следующем: пока одна программа выполняет операцию ввода-вывода, процессор не простаивает, как это происходило при однопрограммном режиме, а выполняет другую программу. Когда операция ввода-вывода заканчивается, процессор возвращается к

выполнению первой программы. Эта идея напоминает поведение преподавателя и студентов на экзамене. Пока один студент (программа) обдумывает ответ на вопрос (операция ввода-вывода), преподаватель (процессор) выслушивает ответ другого студента (вычисления). Естественно, такая ситуация требует наличия в комнате нескольких студентов. Точно так же мультипрограммирование требует наличия в памяти нескольких программ одновременно. При этом каждая программа загружается в свой участок оперативной памяти, называемый разделом, и не должна влиять на выполнение другой программы. (Студенты сидят за отдельными столами и не подсказывают друг другу.)

Появление мультипрограммирования требует настоящей революции в строении вычислительной системы. Особую роль здесь играет аппаратная поддержка (многие аппаратные новшества появились еще на предыдущем этапе эволюции), наиболее существенные особенности которой перечислены ниже.
Реализация защитных механизмов. Программы не должны иметь самостоятельного доступа к распределению ресурсов, что приводит к появлению привилегированных и непривилегированных команд. Привилегированные команды, например команды ввода-вывода, могут исполняться только операционной системой. Говорят, что она работает в привилегированном режиме. Переход управления от прикладной программы к ОС сопровождается контролируемой сменой режима. Кроме того, это защита памяти, позволяющая изолировать конкурирующие пользовательские программы друг от друга, а ОС – от программ пользователей.

Наличие прерываний. Внешние прерывания оповещают ОС о том, что произошло асинхронное событие, например завершилась операция ввода-вывода. Внутренние прерывания (сейчас их принято называть исключительными ситуациями) возникают, когда выполнение программы привело к ситуации, требующей вмешательства ОС, например деление на ноль или попытка нарушения защиты.
Развитие параллелизма в архитектуре. Прямой доступ к памяти и организация каналов ввода-вывода позволили освободить центральный процессор от рутинных операций.

Не менее важна в организации мультипрограммирования роль операционной системы. Она отвечает за следующие операции.
Организация интерфейса между прикладной программой и ОС при помощи системных вызовов.
Организация очереди из заданий в памяти и выделение процессора одному из заданий потребовало планирования использования процессора.
Переключение с одного задания на другое требует сохранения содержимого регистров и структур данных, необходимых для выполнения задания, иначе говоря, контекста для обеспечения правильного продолжения вычислений.
Поскольку память является ограниченным ресурсом, нужны стратегии управления памятью, то есть требуется упорядочить процессы размещения, замещения и выборки информации из памяти.
Организация хранения информации на внешних носителях в виде файлов и обеспечение доступа к конкретному файлу только определенным категориям пользователей.
Поскольку программам может потребоваться произвести санкционированный обмен данными, необходимо их обеспечить средствами коммуникации.
Для корректного обмена данными необходимо разрешать конфликтные ситуации, возникающие при работе с различными ресурсами и предусмотреть координацию программами своих действий, т.е. снабдить систему средствами синхронизации.

Мультипрограммные системы обеспечили возможность более эффективного использования системных ресурсов (например, процессора, памяти, периферийных

устройств), но они еще долго оставались пакетными. Пользователь не мог непосредственно взаимодействовать с заданием и должен был предусмотреть с помощью управляющих карт все возможные ситуации. Отладка программ по-прежнему занимала много времени и требовала изучения многостраничных распечаток содержимого памяти и регистров или использования отладочной печати.

Появление электронно-лучевых дисплеев и переосмысление возможностей применения клавиатур поставили на очередь решение этой проблемы. Логическим расширением систем мультипрограммирования стали time-sharing системы, или системы разделения времени1). В них процессор переключается между задачами не только на время операций ввода-вывода, но и просто по прошествии определенного времени. Эти переключения происходят так часто, что пользователи могут взаимодействовать со своими программами во время их выполнения, то есть интерактивно. В результате появляется возможность одновременной работы нескольких пользователей на одной компьютерной системе. У каждого пользователя для этого должна быть хотя бы одна программа в памяти. Чтобы уменьшить ограничения на количество работающих пользователей, была внедрена идея неполного нахождения исполняемой программы в оперативной памяти. Основная часть программы находится на диске, и фрагмент, который необходимо в данный момент выполнять, может быть загружен в оперативную память, а ненужный – выкачан обратно на диск. Это реализуется с помощью механизма виртуальной памяти. Основным достоинством такого механизма является создание иллюзии неограниченной оперативной памяти ЭВМ.

В системах разделения времени пользователь получил возможность эффективно производить отладку программы в интерактивном режиме и записывать информацию на диск, не используя перфокарты, а непосредственно с клавиатуры. Появление on-line-файлов привело к необходимости разработки развитых файловых систем.

Параллельно внутренней эволюции вычислительных систем происходила и внешняя их эволюция. До начала этого периода вычислительные комплексы были, как правило, несовместимы. Каждый имел собственную операционную систему, свою систему команд и т. д. В результате программу, успешно работающую на одном типе машин, необходимо было полностью переписывать и заново отлаживать для выполнения на компьютерах другого типа. В начале третьего периода появилась идея создания семейств программно совместимых машин, работающих под управлением одной и той же операционной системы. Первым семейством программно совместимых компьютеров, построенных на интегральных микросхемах, стала серия машин IBM/360. Разработанное в начале 60-х годов, это семейство значительно превосходило машины второго поколения по критерию цена/производительность. За ним последовала линия компьютеров PDP, несовместимых с линией IBM, и лучшей моделью в ней стала PDP-11.

Сила "одной семьи" была одновременно и ее слабостью. Широкие возможности этой концепции (наличие всех моделей: от мини-компьютеров до гигантских машин; обилие разнообразной периферии; различное окружение; различные пользователи) порождали сложную и громоздкую операционную систему. Миллионы строчек Ассемблера, написанные тысячами программистов, содержали множество ошибок, что вызывало непрерывный поток публикаций о них и попыток исправления. Только в операционной системе OS/360 содержалось более 1000 известных ошибок. Тем не менее идея стандартизации операционных систем была широко внедрена в сознание пользователей и в дальнейшем получила активное развитие.

Четвертый период (с 1980 г. по настоящее время). Персональные компьютеры. Классические, сетевые и распределенные системы

Следующий период в эволюции вычислительных систем связан с появлением больших интегральных схем (БИС). В эти годы произошло резкое возрастание степени интеграции и снижение стоимости микросхем. Компьютер, не отличающийся по архитектуре от PDP-11, по цене и простоте эксплуатации стал доступен отдельному человеку, а не отделу предприятия или университета. Наступила эра персональных компьютеров. Первоначально персональные компьютеры предназначались для использования одним пользователем в однопрограммном режиме, что повлекло за собой деградацию архитектуры этих ЭВМ и их операционных систем (в частности, пропала необходимость защиты файлов и памяти, планирования заданий и т. п.).

Компьютеры стали использоваться не только специалистами, что потребовало разработки "дружественного" программного обеспечения.

Однако рост сложности и разнообразия задач, решаемых на персональных компьютерах, необходимость повышения надежности их работы привели к возрождению практически всех черт, характерных для архитектуры больших вычислительных систем.

В середине 80-х стали бурно развиваться сети компьютеров, в том числе персональных, работающих под управлением сетевых или распределенных операционных систем.

В сетевых операционных системах пользователи могут получить доступ к ресурсам другого сетевого компьютера, только они должны знать об их наличии и уметь это сделать. Каждая машина в сети работает под управлением своей локальной операционной системы, отличающейся от операционной системы автономного компьютера наличием дополнительных средств (программной поддержкой для сетевых интерфейсных устройств и доступа к удаленным ресурсам), но эти дополнения не меняют структуру операционной системы.

Распределенная система, напротив, внешне выглядит как обычная автономная система. Пользователь не знает и не должен знать, где его файлы хранятся – на локальной или удаленной машине – и где его программы выполняются. Он может вообще не знать, подключен ли его компьютер к сети. Внутреннее строение распределенной операционной системы имеет существенные отличия от автономных систем.

В дальнейшем автономные операционные системы мы будем называть классическими операционными системами.

Просмотрев этапы развития вычислительных систем, мы можем выделить шесть основных функций, которые выполняли классические операционные системы в процессе эволюции:
Планирование заданий и использования процессора.
Обеспечение программ средствами коммуникации и синхронизации.
Управление памятью.
Управление файловой системой.
Управление вводом-выводом.
Обеспечение безопасности

Каждая из приведенных функций обычно реализована в виде подсистемы, являющейся структурным компонентом ОС. В каждой операционной системе эти функции, конечно, реализовывались по-своему, в различном объеме. Они не были изначально придуманы как составные части операционных систем, а появились в процессе развития, по мере того как вычислительные системы становились все более удобными, эффективными и безопасными.

Эволюция вычислительных систем, созданных человеком, пошла по такому пути, но никто еще не доказал, что это единственно возможный путь их развития. Операционные системы существуют потому, что на данный момент их существование – это разумный способ использования вычислительных систем. Рассмотрение общих принципов и алгоритмов реализации их функций и составляет содержание большей части нашего курса, в котором будут последовательно описаны перечисленные подсистемы.

Основные понятия, концепции ОС

В процессе эволюции возникло несколько важных концепций, которые стали неотъемлемой частью теории и практики ОС. Рассматриваемые в данном разделе понятия будут встречаться и разъясняться на протяжении всего курса. Здесь дается их краткое описание.

Системные вызовы

В любой операционной системе поддерживается механизм, который позволяет пользовательским программам обращаться к услугам ядра ОС. В операционных системах наиболее известной советской вычислительной машины БЭСМ-6 соответствующие средства "общения" с ядром назывались экстракодами, в операционных системах IBM они назывались системными макрокомандами и т.д. В ОС Unix такие средства называют системными вызовами.

Системные вызовы (system calls) – это интерфейс между операционной системой и пользовательской программой. Они создают, удаляют и используют различные объекты, главные из которых – процессы и файлы. Пользовательская программа запрашивает сервис у операционной системы, осуществляя системный вызов. Имеются библиотеки процедур, которые загружают машинные регистры определенными параметрами и осуществляют прерывание процессора, после чего управление передается обработчику данного вызова, входящему в ядро операционной системы. Цель таких библиотек – сделать системный вызов похожим на обычный вызов подпрограммы.

Основное отличие состоит в том, что при системном вызове задача переходит в привилегированный режим или режим ядра (kernel mode). Поэтому системные вызовы иногда еще называют программными прерываниями, в отличие от аппаратных прерываний, которые чаще называют просто прерываниями.

В этом режиме работает код ядра операционной системы, причем исполняется он в адресном пространстве и в контексте вызвавшей его задачи. Таким образом, ядро операционной системы имеет полный доступ к памяти пользовательской программы, и при системном вызове достаточно передать адреса одной или нескольких областей памяти с параметрами вызова и адреса одной или нескольких областей памяти для результатов вызова.

В большинстве операционных систем системный вызов осуществляется командой программного прерывания (INT). Программное прерывание – это синхронное событие, которое может быть повторено при выполнении одного и того же программного кода.

Прерывания

Прерывание (hardware interrupt) – это событие, генерируемое внешним (по отношению к процессору) устройством. Посредством аппаратных прерываний аппаратура либо информирует центральный процессор о том, что произошло какое-либо событие, требующее немедленной реакции (например, пользователь нажал клавишу), либо сообщает о завершении асинхронной операции ввода-вывода (например, закончено чтение данных с

диска в основную память). Важный тип аппаратных прерываний – прерывания таймера, которые генерируются периодически через фиксированный промежуток времени. Прерывания таймера используются операционной системой при планировании процессов. Каждый тип аппаратных прерываний имеет собственный номер, однозначно определяющий источник прерывания. Аппаратное прерывание – это асинхронное событие, то есть оно возникает вне зависимости от того, какой код исполняется процессором в данный момент. Обработка аппаратного прерывания не должна учитывать, какой процесс является текущим.
Исключительные ситуации

Исключительная ситуация (exception) – событие, возникающее в результате попытки выполнения программой команды, которая по каким-то причинам не может быть выполнена до конца. Примерами таких команд могут быть попытки доступа к ресурсу при отсутствии достаточных привилегий или обращения к отсутствующей странице памяти. Исключительные ситуации, как и системные вызовы, являются синхронными событиями, возникающими в контексте текущей задачи. Исключительные ситуации можно разделить на исправимые и неисправимые. К исправимым относятся такие исключительные ситуации, как отсутствие нужной информации в оперативной памяти. После устранения причины исправимой исключительной ситуации программа может выполняться дальше. Возникновение в процессе работы операционной системы исправимых исключительных ситуаций считается нормальным явлением. Неисправимые исключительные ситуации чаще всего возникают в результате ошибок в программах (например, деление на ноль). Обычно в таких случаях операционная система реагирует завершением программы, вызвавшей исключительную ситуацию.

Файлы

Файлы предназначены для хранения информации на внешних носителях, то есть принято, что информация, записанная, например, на диске, должна находиться внутри файла. Обычно под файлом понимают именованную часть пространства на носителе информации.

Главная задача файловой системы (file system) – скрыть особенности ввода-вывода и дать программисту простую абстрактную модель файлов, независимых от устройств. Для чтения, создания, удаления, записи, открытия и закрытия файлов также имеется обширная категория системных вызовов (создание, удаление, открытие, закрытие, чтение и т.д.). Пользователям хорошо знакомы такие связанные с организацией файловой системы понятия, как каталог, текущий каталог, корневой каталог, путь. Для манипулирования этими объектами в операционной системе имеются системные вызовы. Файловая система ОС описана в лекциях 11–12.

Процессы, нити

Концепция процесса в ОС одна из наиболее фундаментальных. Процессы подробно рассмотрены в лекциях 2–7. Там же описаны нити, или легковесные процессы.

Архитектурные особенности ОС

До сих пор мы говорили о взгляде на операционные системы извне, о том, что делают операционные системы. Дальнейший наш курс будет посвящен тому, как они это делают. Но мы пока ничего не сказали о том, что они представляют собой изнутри, какие подходы существуют к их построению.

Монолитное ядро

По сути дела, операционная система – это обычная программа, поэтому было бы логично и организовать ее так же, как устроено большинство программ, то есть составить из процедур и функций. В этом случае компоненты операционной системы являются не самостоятельными модулями, а составными частями одной большой программы. Такая структура операционной системы называется монолитным ядром (monolithic kernel). Монолитное ядро представляет собой набор процедур, каждая из которых может вызвать каждую. Все процедуры работают в привилегированном режиме. Таким образом, монолитное ядро – это такая схема операционной системы, при которой все ее компоненты являются составными частями одной программы, используют общие структуры данных и взаимодействуют друг с другом путем непосредственного вызова процедур. Для монолитной операционной системы ядро совпадает со всей системой.

Во многих операционных системах с монолитным ядром сборка ядра, то есть его компиляция, осуществляется отдельно для каждого компьютера, на который устанавливается операционная система. При этом можно выбрать список оборудования и программных протоколов, поддержка которых будет включена в ядро. Так как ядро является единой программой, перекомпиляция – это единственный способ добавить в него новые компоненты или исключить неиспользуемые. Следует отметить, что присутствие в ядре лишних компонентов крайне нежелательно, так как ядро всегда полностью располагается в оперативной памяти. Кроме того, исключение ненужных компонентов повышает надежность операционной системы в целом.

Монолитное ядро – старейший способ организации операционных систем. Примером систем с монолитным ядром является большинство Unix-систем.

Даже в монолитных системах можно выделить некоторую структуру. Как в бетонной глыбе можно различить вкрапления щебенки, так и в монолитном ядре выделяются вкрапления сервисных процедур, соответствующих системным вызовам. Сервисные процедуры выполняются в привилегированном режиме, тогда как пользовательские программы – в непривилегированном. Для перехода с одного уровня привилегий на другой иногда может использоваться главная сервисная программа, определяющая, какой именно системный вызов был сделан, корректность входных данных для этого вызова и передающая управление соответствующей сервисной процедуре с переходом в привилегированный режим работы. Иногда выделяют также набор программных утилит, которые помогают выполнять сервисные процедуры.

Многоуровневые системы (Layered systems)

Продолжая структуризацию, можно разбить всю вычислительную систему на ряд более мелких уровней с хорошо определенными связями между ними, так чтобы объекты уровня N могли вызывать только объекты уровня N-1. Нижним уровнем в таких системах обычно является hardware, верхним уровнем – интерфейс пользователя. Чем ниже уровень, тем более привилегированные команды и действия может выполнять модуль, находящийся на этом уровне. Впервые такой подход был применен при создании системы THE (Technishe Hogeschool Eindhoven) Дейкстрой (Dijkstra) и его студентами в 1968 г. Эта система имела следующие уровни:

5	Интерфейс пользователя
4	Управление вводом-выводом
3	Драйвер устройства связи оператора и консоли
2	Управление памятью
1	Планирование задач и процессов
0	Hardware

Рис. 1.2. Слоеная система THE

Слоеные системы хорошо реализуются. При использовании операций нижнего слоя не нужно знать, как они реализованы, нужно лишь понимать, что они делают. Слоеные системы хорошо тестируются. Отладка начинается с нижнего слоя и проводится послойно. При возникновении ошибки мы можем быть уверены, что она находится в тестируемом слое. Слоеные системы хорошо модифицируются. При необходимости можно заменить лишь один слой, не трогая остальные. Но слоеные системы сложны для разработки: тяжело правильно определить порядок слоев и что к какому слою относится. Слоеные системы менее эффективны, чем монолитные. Так, например, для выполнения операций ввода-вывода программе пользователя придется последовательно проходить все слои от верхнего до нижнего.

Виртуальные машины

В начале лекции мы говорили о взгляде на операционную систему как на виртуальную машину, когда пользователю нет необходимости знать детали внутреннего устройства компьютера. Он работает с файлами, а не с магнитными головками и двигателем; он работает с огромной виртуальной, а не ограниченной реальной оперативной памятью; его мало волнует, единственный он на машине пользователь или нет. Рассмотрим несколько иной подход. Пусть операционная система реализует виртуальную машину для каждого пользователя, но не упрощая ему жизнь, а, наоборот, усложняя. Каждая такая виртуальная машина предстает перед пользователем как голое железо – копия всего hardware в вычислительной системе, включая процессор, привилегированные и непривилегированные команды, устройства ввода-вывода, прерывания и т.д. И он остается с этим железом один на один. При попытке обратиться к такому виртуальному железу на уровне привилегированных команд в действительности происходит системный вызов реальной операционной системы, которая и производит все необходимые действия. Такой подход позволяет каждому пользователю загрузить свою операционную систему на виртуальную машину и делать с ней все, что душа пожелает.

Программа пользователя	Программа пользователя	Программа пользователя
MS-DOS	Linux	Windows-NT
Виртуальное hardware	Виртуальное hardware	Виртуальное hardware
Реальная операционная система		
Реальное hardware		

Рис. 1.3. Вариант виртуальной машины

Первой реальной системой такого рода была система CP/CMS, или VM/370, как ее называют сейчас, для семейства машин IBM/370.

Недостатком таких операционных систем является снижение эффективности виртуальных машин по сравнению с реальным компьютером, и, как правило, они очень громоздки. Преимущество же заключается в использовании на одной вычислительной системе программ, написанных для разных операционных систем.

Микроядерная архитектура

Современная тенденция в разработке операционных систем состоит в перенесении значительной части системного кода на уровень пользователя и одновременной минимизации ядра. Речь идет о подходе к построению ядра, называемом микроядерной архитектурой (microkernel architecture) операционной системы, когда большинство ее составляющих являются самостоятельными программами. В этом случае взаимодействие между ними обеспечивает специальный модуль ядра, называемый микроядром. Микроядро работает в привилегированном режиме и обеспечивает взаимодействие между программами, планирование использования процессора, первичную обработку прерываний, операции ввода-вывода и базовое управление памятью.

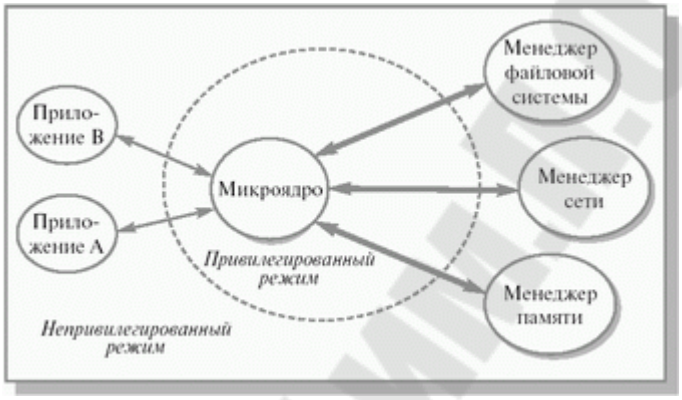

Рис. 1.4. Микроядерная архитектура операционной системы

Остальные компоненты системы взаимодействуют друг с другом путем передачи сообщений через микроядро.

Основное достоинство микроядерной архитектуры – высокая степень модульности ядра операционной системы. Это существенно упрощает добавление в него новых компонентов. В микроядерной операционной системе можно, не прерывая ее работы, загружать и выгружать новые драйверы, файловые системы и т. д. Существенно упрощается процесс отладки компонентов ядра, так как новая версия драйвера может загружаться без перезапуска всей операционной системы. Компоненты ядра операционной системы ничем принципиально не отличаются от пользовательских программ, поэтому для их отладки можно применять обычные средства. Микроядерная архитектура повышает надежность системы, поскольку ошибка на уровне непривилегированной программы менее опасна, чем отказ на уровне режима ядра.

В то же время микроядерная архитектура операционной системы вносит дополнительные накладные расходы, связанные с передачей сообщений, что существенно влияет на производительность. Для того чтобы микроядерная операционная система по скорости не уступала операционным системам на базе монолитного ядра, требуется очень аккуратно проектировать разбиение системы на компоненты, стараясь минимизировать взаимодействие между ними. Таким образом, основная сложность при создании микроядерных операционных систем – необходимость очень аккуратного проектирования.

Смешанные системы

Все рассмотренные подходы к построению операционных систем имеют свои достоинства и недостатки. В большинстве случаев современные операционные системы используют различные комбинации этих подходов. Так, например, ядро операционной системы Linux представляет собой монолитную систему с элементами микроядерной архитектуры. При

компиляции ядра можно разрешить динамическую загрузку и выгрузку очень многих компонентов ядра – так называемых модулей. В момент загрузки модуля его код загружается на уровне системы и связывается с остальной частью ядра. Внутри модуля могут использоваться любые экспортируемые ядром функции.

Другим примером смешанного подхода может служить возможность запуска операционной системы с монолитным ядром под управлением микроядра. Так устроены 4.4BSD и MkLinux, основанные на микроядре Mach. Микроядро обеспечивает управление виртуальной памятью и работу низкоуровневых драйверов. Все остальные функции, в том числе взаимодействие с прикладными программами, осуществляется монолитным ядром. Данный подход сформировался в результате попыток использовать преимущества микроядерной архитектуры, сохраняя по возможности хорошо отлаженный код монолитного ядра.

Наиболее тесно элементы микроядерной архитектуры и элементы монолитного ядра переплетены в ядре Windows NT. Хотя Windows NT часто называют микроядерной операционной системой, это не совсем так. Микроядро NT слишком велико (более 1 Мбайт), чтобы носить приставку "микро". Компоненты ядра Windows NT располагаются в вытесняемой памяти и взаимодействуют друг с другом путем передачи сообщений, как и положено в микроядерных операционных системах. В то же время все компоненты ядра работают в одном адресном пространстве и активно используют общие структуры данных, что свойственно операционным системам с монолитным ядром. По мнению специалистов Microsoft, причина проста: чисто микроядерный дизайн коммерчески невыгоден, поскольку неэффективен.

Таким образом, Windows NT можно с полным правом назвать гибридной операционной системой.

Классификация ОС

Существует несколько схем классификации операционных систем. Ниже приведена классификация по некоторым признакам с точки зрения пользователя.
Реализация многозадачности

По числу одновременно выполняемых задач операционные системы можно разделить на два класса:
многозадачные (Unix, OS/2, Windows);
однозадачные (например, MS-DOS).

Многозадачная ОС, решая проблемы распределения ресурсов и конкуренции, полностью реализует мультипрограммный режим в соответствии с требованиями раздела "Основные понятия, концепции ОС".

Многозадачный режим, который воплощает в себе идею разделения времени, называется вытесняющим (preemptive). Каждой программе выделяется квант процессорного времени, по истечении которого управление передается другой программе. Говорят, что первая программа будет вытеснена. В вытесняющем режиме работают пользовательские программы большинства коммерческих ОС.

В некоторых ОС (Windows 3.11, например) пользовательская программа может монополизировать процессор, то есть работать в невытесняющем режиме. Как правило, в большинстве систем не подлежит вытеснению код собственно ОС. Ответственные

программы, в частности задачи реального времени, также не вытесняются. Более подробно об этом рассказано в лекции, посвященной планированию работы процессора.

По приведенным примерам можно судить о приблизительности классификации. Так, в ОС MS-DOS можно организовать запуск дочерней задачи и наличие в памяти двух и более задач одновременно. Однако эта ОС традиционно считается однозадачной, главным образом из-за отсутствия защитных механизмов и коммуникационных возможностей.
Поддержка многопользовательского режима

По числу одновременно работающих пользователей ОС можно разделить на:
однопользовательские (MS-DOS, Windows 3.x);
многопользовательские (Windows NT, Unix).

Наиболее существенное отличие между этими ОС заключается в наличии у многопользовательских систем механизмов защиты персональных данных каждого пользователя.

Многопроцессорная обработка

Вплоть до недавнего времени вычислительные системы имели один центральный процессор. В результате требований к повышению производительности появились многопроцессорные системы, состоящие из двух и более процессоров общего назначения, осуществляющих параллельное выполнение команд. Поддержка мультипроцессирования является важным свойством ОС и приводит к усложнению всех алгоритмов управления ресурсами. Многопроцессорная обработка реализована в таких ОС, как Linux, Solaris, Windows NT, и ряде других.

Многопроцессорные ОС разделяют на симметричные и асимметричные. В симметричных ОС на каждом процессоре функционирует одно и то же ядро, и задача может быть выполнена на любом процессоре, то есть обработка полностью децентрализована. При этом каждому из процессоров доступна вся память.

В асимметричных ОС процессоры неравноправны. Обычно существует главный процессор (master) и подчиненные (slave), загрузку и характер работы которых определяет главный процессор.

Системы реального времени

В разряд многозадачных ОС, наряду с пакетными системами и системами разделения времени, включаются также системы реального времени, не упоминавшиеся до сих пор.

Они используются для управления различными техническими объектами или технологическими процессами. Такие системы характеризуются предельно допустимым временем реакции на внешнее событие, в течение которого должна быть выполнена программа, управляющая объектом. Система должна обрабатывать поступающие данные быстрее, чем они могут поступать, причем от нескольких источников одновременно.

Столь жесткие ограничения сказываются на архитектуре систем реального времени, например, в них может отсутствовать виртуальная память, поддержка которой дает непредсказуемые задержки в выполнении программ. (См. также разделы, связанные с планированием процессов и реализацией виртуальной памяти.)

Приведенная классификация ОС не является исчерпывающей. Более подробно особенности применения современных ОС рассмотрены в [Олифер, 2001].
Заключение

Мы рассмотрели различные взгляды на то, что такое операционная система; изучили историю развития операционных систем; выяснили, какие функции обычно выполняют операционные системы; наконец, разобрались в том, какие существуют подходы к построению операционных систем. Следующую лекцию мы посвятим выяснению понятия "процесс" и вопросам планирования процессов.

Приложение 1.

Некоторые сведения об архитектуре компьютера

Основными аппаратными компонентами компьютера являются: основная память, центральный процессор и периферийные устройства. Для обмена данными между собой эти компоненты соединены группой проводов, называемой магистралью (см. рис.1.5).

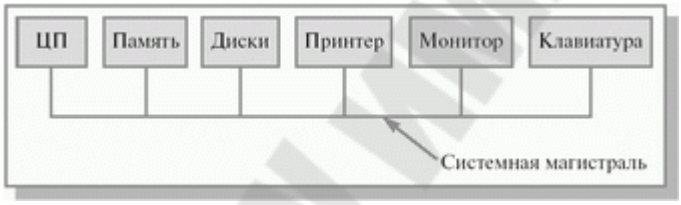

Рис. 1.5. Некоторые компоненты компьютера

Основная память используется для запоминания программ и данных в двоичном виде и организована в виде упорядоченного массива ячеек, каждая из которых имеет уникальный цифровой адрес. Как правило, размер ячейки составляет один байт. Типовые операции над основной памятью – считывание и запись содержимого ячейки с определенным адресом.

Выполнение различных операций с данными осуществляется изолированной частью компьютера, называемой центральным процессором (ЦП). ЦП также имеет ячейки для запоминания информации, называемые регистрами. Их разделяют на регистры общего назначения и специализированные регистры. В современных компьютерах емкость регистра обычно составляет 4–8 байт. Регистры общего назначения используются для временного хранения данных и результатов операций. Для обработки информации обычно организовывается передача данных из ячеек памяти в регистры общего назначения, выполнение операции центральным процессором и передача результатов операции в основную память.

Специализированные регистры используются для контроля работы процессора. Наиболее важными являются: программный счетчик, регистр команд и регистр содержащий информацию о состоянии программы.

Программы хранятся в виде последовательности машинных команд, которые должен выполнять центральный процессор. Каждая команда состоит из поля операции и полей операндов, то есть тех данных, над которыми выполняется данная операция. Весь набор машинных команд называется машинным языком.

Выполнение программы осуществляется следующим образом. Машинная команда, на которую указывает программный счетчик, считывается из памяти и копируется в регистр

команд. Здесь она декодируется, после чего исполняется. После выполнения команды программный счетчик указывает на следующую команду. Эти действия, называемые машинным циклом, затем повторяются.
Взаимодействие с периферийными устройствами

Периферийные устройства предназначены для ввода и вывода информации. Каждое устройство обычно имеет в своем составе специализированный компьютер, называемый контроллером или адаптером. Когда контроллер вставляется в разъем на материнской плате, он подключается к шине и получает уникальный номер (адрес). После этого контроллер осуществляет наблюдение за сигналами, идущими по шине, и отвечает на сигналы, адресованные ему.

Любая операция ввода-вывода предполагает диалог между ЦП и контроллером устройства. Когда процессору встречается команда, связанная с вводом-выводом, входящая в состав какой-либо программы, он выполняет ее, посылая сигналы контроллеру устройства. Это так называемый программируемый ввод-вывод.

В свою очередь, любые изменения с внешними устройствами имеют следствием передачу сигнала от устройства к ЦП. С точки зрения ЦП это является асинхронным событием и требует его реакции. Для того чтобы обнаружить такое событие, между машинными циклами процессор опрашивает специальный регистр, содержащий информацию о типе устройства, сгенерировавшего сигнал. Если сигнал имеет место, то ЦП выполняет специфичную для данного устройства программу, задача которой – отреагировать на это событие надлежащим образом (например, занести символ, введенный с клавиатуры, в специальный буфер). Такая программа называется программой обработки прерывания, а само событие прерыванием, поскольку оно нарушает плановую работу процессора. После завершения обработки прерывания процессор возвращается к выполнению программы. Эти действия компьютера называются вводом-выводом с использованием прерываний.

В современных компьютерах также имеется возможность непосредственного взаимодействия между контроллером и основной памятью, минуя ЦП, – так называемый механизм прямого доступа к памяти.

Модуль 2. Командный интерфейс ОС

Лекции 7 ч.
Тема 4. Командный интерфейс ОС Windows
Тема 5. Сервер сценариев Windows Script Host
Тема 6. Командный интерфейс ОС Unix(Linux)
Тема 7. Bash Shell ОС Unix(Linux)

Лекция. Командный интерфейс ОС Windows

1. Оболочка командной строки Windows. Интерпретатор Cmd.exe

В операционной системе Windows, как и в других операционных системах, интерактивные (набираемые с клавиатуры и сразу же выполняемые) команды выполняются с помощью так называемого командного интерпретатора, иначе называемого командным процессором или оболочкой командной строки (command shell). Командный интерпретатор или оболочка командной строки — это программа, которая, находясь в оперативной памяти, считывает набираемые вами команды и обрабатывает их.
Для запуска командного интерпретатора (открытия нового сеанса командной строки) можно выбрать пункт Выполнить… (Run) в меню Пуск (Start), ввести имя файла Cmd.exe и нажать кнопку OK

1.1 Синтаксис командной строки, перенаправление ввода – вывода

Файловая система имеет древовидную структуру и имена файлов задаются в формате *[диск:] [путь\]имя_файла*. Если путь начинается с символа «\», то маршрут вычисляется от корневого каталога – иначе от текущего.
Например, c:123.txt задает файл 123.txt в текущем каталоге, c:\123.txt – в корневом, а DOC\123.txt – в подкаталоге DOC текущего каталога.
Существуют особые обозначения для текущего каталога (точка «.») и трех его верхних уровней (две точки «..» - родительский, три «…» - второго уровня и, наконец, четыре «….» - третьего уровня).
Например, для текущего каталога C:\Windows\Media\Office97 путь к файлу autoexec.bat в корневом каталоге диска C: может быть записан в виде ….\autoexec.bat.
В именах файлов (но не дисков или каталогов) можно применять так называемые групповые символы или шаблоны: ? (вопросительный знак) и * (звездочка). Символ * в имени файла означает произвольное количество любых допустимых символов, символ ? — один произвольный символ или его отсутствие. Скажем, под шаблон `text??1.txt` подходят, например, имена text121.txt и text11.txt, под шаблон `text*.txt` — имена text.txt, textab12.txt, а под шаблон `text.*` — все файлы с именем text и произвольным расширением.
Например, `DIR /? > helpdir.txt` выведет справку по команде DIR в файл. Символ «>>» позволяет не создавать файл заново, а дописать в него. По аналогии символ «<» позволяет читать данные не с клавиатуры, а с файла. Например, `DATE < date.txt` ввод новой даты из файла.

1.2 Переменные окружения

При загрузке ОС Windows в оперативной памяти постоянно хранится набор т.н. *переменных окружения* (environment variables). Хотя в Windows есть более совершенный способ для хранения системных значений – реестр, многие программы по-прежнему используют переменные окружения. Наиболее важные переменные хранят системный путь

для поиска (PATH), каталог запуска Windows (WINDIR), место хранения временных файлов (TEMP) и многое другое.
Переменные устанавливаются с помощью команды
`SET [переменная=[строка]]`
Запуск SET без параметров приводит к выводу списка переменных среды. Для получения их значений (всегда строки) нужно имя соответствующей переменной заключить в символы «%», **например**: `%TEMP%`.

1.3 Внутренние и внешние команды. Структура команд

Некоторые команды распознаются и выполняются непосредственно самим командным интерпретатором — такие команды называются внутренними (например, `COPY` или `DIR`) Другие команды операционной системы представляют собой отдельные программы, расположенные по умолчанию в том же каталоге, что и Cmd.exe, которые Windows загружает и выполняет аналогично другим программам. Такие команды называются внешними (например, `MORE` или `XCOPY`).
Рассмотрим структуру самой командной строки и принцип работы с ней. Для того, чтобы выполнить команду, вы после приглашения командной строки (например, `C:\>`) вводите имя этой команды (регистр не важен), ее параметры и ключи (если они необходимы) и нажимаете клавишу <Enter>. Например:
C:\>COPY C:\myfile.txt A:\ /V
Имя команды здесь — `COPY`, параметры — `C:\myfile.txt` и `A:\`, а ключом является `/V`. Отметим, что в некоторых командах ключи могут начинаться не с символа /, а с символа – (минус), например, `-V`.
Многие команды Windows имеют большое количество дополнительных параметров и ключей, запомнить которые зачастую бывает трудно. Большинство команд снабжено встроенной справкой, в которой кратко описываются назначение и синтаксис данной команды. Получить доступ к такой справке можно путем ввода команды с ключом `/?`.

1.4 Условное выполнение и группировка команд

В командной строке Windows NT/2000/XP можно использовать специальные символы, которые позволяют вводить несколько команд одновременно и управлять работой команд в зависимости от результатов их выполнения. С помощью таких символов условной обработки можно содержание небольшого пакетного файла записать в одной строке и выполнить полученную составную команду.
Используя символ амперсанта `&`, можно разделить несколько утилит в одной командной строке, при этом они будут выполняться друг за другом. Например, если набрать команду `DIR & PAUSE & COPY /?` и нажать клавишу <Enter>, то вначале на экран будет выведено содержимое текущего каталога, а после нажатия любой клавиши — встроенная справка команды `COPY`.
Символ `^` позволяет использовать командные символы как текст, то есть при этом происходит игнорирование значения специальных символов. Например, если ввести в командной строке
ECHO Абв & COPY /?
и нажать клавишу <Enter>, то произойдет выполнение подряд двух команд: `ECHO Абв` и `COPY /?` (команда `ECHO` выводит на экран символы, указанные в командной строке после нее). Если же выполнить команду
ECHO Абв ^& COPY /?
то на экран будет выведено
Абв & COPY /?
В этом случае просто выполняется одна команда `ECHO` с соответствующими параметрами.

Условная обработка команд в Windows осуществляется с помощью символов && и || следующим образом. Двойной амперсант && запускает команду, стоящую за ним в командной строке, только в том случае, если команда, стоящая перед амперсантами была выполнена успешно. Например, если в корневом каталоге диска C: есть файл plan.txt, то выполнение строки `TYPE C:\plan.txt && DIR` приведет к выводу на экран этого файла и содержимого текущего каталога. Если же файл C:\plan.txt не существует, то команда DIR выполняться не будет.

Два символа || осуществляют в командной строке обратное действие, т.е. запускают команду, стоящую за этими символами, только в том случае, если команда, идущая перед ними, не была успешно выполнена. Таким образом, если в предыдущем примере файл C:\plan.txt будет отсутствовать, то в результате выполнения строки `TYPE C:\plan.txt || DIR` на экран выведется содержимое текущего каталога.

Отметим, что условная обработка действует только на ближайшую команду, то есть в строке

TYPE C:\plan.txt && DIR & COPY /?

команда `COPY /?` запустится в любом случае, независимо от результата выполнения команды `TYPE C:\plan.txt`.

Несколько утилит можно сгруппировать в командной строке с помощью скобок. Рассмотрим, например, две строки:

TYPE C:\plan.txt && DIR & COPY /?

TYPE C:\plan.txt && (DIR & COPY /?)

В первой из них символ условной обработки && действует только на команду DIR, во второй — одновременно на две команды: DIR и COPY.

Примеры команд для работы с файловой системой

Рассмотрим некоторые наиболее часто используемые команды для работы с файловой системой. Отметим сначала несколько особенностей определения путей к файлам в Windows.

1.5 Основные команды

Полный список команд можно вывести набрав HELP в командной строке.

Команда CD

Текущий каталог можно изменить с помощью команды

CD [диск:][путь\]

Путь к требуемому каталогу указывается с учетом приведенных выше замечаний. Например, команда `CD \` выполняет переход в корневой каталог текущего диска. Если запустить команду CD без параметров, то на экран будут выведены имена текущего диска и каталога.

Команда COPY

Одной из наиболее часто повторяющихся задач при работе на компьютере является копирование и перемещение файлов из одного места в другое. Для копирования одного или нескольких файлов используется команда COPY.

Синтаксис этой команды:

COPY [/A|/B] источник [/A|/B] [+ источник [/A|/B] [+ ...]]
[результат [/A|/B]] [/V][/Y|/–Y]

Краткое описание параметров и ключей команды COPY приведено в табл. 1.1.

Таблица 1.1. Параметры и ключи команды COPY

Параметр	Описание
источник	Имя копируемого файла или файлов
/A	Файл является текстовым файлом ASCII, то есть конец файла обозначается символом с кодом ASCII 26 (<Ctrl>+<Z>)

/B	Файл является двоичным. Этот ключ указывает на то, что интерпретатор команд должен при копировании считывать из источника число байт, заданное размером в каталоге копируемого файла
результат	Каталог для размещения результата копирования и/или имя создаваемого файла
/V	Проверка правильности копирования путем сравнения файлов после копирования
/Y	Отключение режима запроса подтверждения на замену файлов
/-Y	Включение режима запроса подтверждения на замену файлов

Приведем примеры использования команды COPY.

Копирование файла abc.txt из текущего каталога в каталог D:\PROGRAM под тем же именем:

```
COPY abc.txt D:\PROGRAM
```

Копирование файла abc.txt из текущего каталога в каталог D:\PROGRAM под новым именем def.txt:

```
COPY abc.txt D:\PROGRAM\def.txt
```

Копирование всех файлов с расширением txt с диска A: в каталог 'Мои документы' на диске C:

```
COPY A:\*.txt "C:\Мои документы"
```

Если не задать в команде целевой файл, то команда COPY создаст копию файла-источника с тем же именем, датой и временем создания, что и исходный файл, и поместит новую копию в текущий каталог на текущем диске. Например, для того, чтобы скопировать все файлы из корневого каталога диска A: в текущий каталог, достаточно выполнить такую краткую команду:

```
COPY A:\*.*
```

В качестве источника или результата при копировании можно указывать имена не только файлов, но и устройств компьютера. Например, для того, чтобы распечатать файл abc.txt на принтере, можно воспользоваться командой копирования этого файла на устройство PRN:

```
COPY abc.txt PRN
```

Другой интересный пример: создадим новый текстовый файл и запишем в него информацию, без использования текстового редактора. Для этого достаточно ввести команду `COPY CON my.txt`, которая будет копировать то, что вы набираете на клавиатуре, в файл my.txt (если этот файл существовал, то он перезапишется, иначе — создастся). Для завершения ввода необходимо ввести символ конца файла, то есть нажать клавиши <Ctrl>+<Z>.

Команда COPY может также объединять (склеивать) нескольких файлов в один. Для этого необходимо указать единственный результирующий файл и несколько исходных. Это достигается путем использования групповых знаков (? и *) или формата файл1 + файл2 + файл3. Например, для объединения файлов 1.txt и 2.txt в файл 3.txt можно задать следующую команду:

```
COPY 1.txt+2.txt 3.txt
```

Объединение всех файлов с расширением dat из текущего каталога в один файл all.dat может быть произведено так:

```
COPY /B *.dat all.dat
```

Ключ /B здесь используется для предотвращения усечения соединяемых файлов, так как при комбинировании файлов команда COPY по умолчанию считает файлами текстовыми.

Если имя целевого файла совпадает с именем одного из копируемых файлов (кроме первого), то исходное содержимое целевого файла теряется. Если имя целевого файла опущено, то в его качестве используется первый файл из списка. Например, команда `COPY 1.txt+2.txt` добавит к содержимому файла 1.txt содержимое файла 2.txt. Командой COPY можно воспользоваться и для присвоения какому-либо файлу текущей даты и времени без модификации его содержимого. Для этого нужно ввести команду типа

COPY /B 1.txt +,,

Здесь запятые указывают на пропуск параметра приемника, что и приводит к требуемому результату.

Команда COPY имеет и свои недостатки. Например, с ее помощью нельзя копировать скрытые и системные файлы, файлы нулевой длины, файлы из подкаталогов. Кроме того, если при копировании группы файлов COPY встретит файл, который в данный момент нельзя скопировать (например, он занят другим приложением), то процесс копирования полностью прервется, и остальные файлы не будут скопированы.

Команда XCOPY

Указанные при описании команды COPY проблемы можно решить с помощью команды XCOPY, которая предоставляет намного больше возможностей при копировании. Необходимо отметить, правда, что XCOPY может работать только с файлами и каталогами, но не с устройствами.

Синтаксис этой команды:

XCOPY источник [результат] [ключи]

Команда XCOPY имеет множество ключей, мы коснемся лишь некоторых из них. Ключ /D[:[дата]] позволяет копировать только файлы, измененные не ранее указанной даты. Если параметр дата не указан, то копирование будет производиться только если источник новее результата. Например, команда

XCOPY "C:\Мои документы*.*" "D:\BACKUP\Мои документы" /D

скопирует в каталог 'D:\BACKUP\Мои документы' только те файлы из каталога 'C:\Мои документы', которые были изменены со времени последнего подобного копирования или которых вообще не было в 'D:\BACKUP\Мои документы'.

Ключ /S позволяет копировать все непустые подкаталоги в каталоге-источнике. С помощью же ключа /E можно копировать вообще все подкаталоги, включая и пустые.

Если указан ключ /C, то копирование будет продолжаться даже в случае возникновения ошибок. Это бывает очень полезным при операциях копирования, производимых над группами файлов, например, при резервном копировании данных.

Ключ /I важен для случая, когда копируются несколько файлов, а файл назначения отсутствует. При задании этого ключа команда XCOPY считает, что файл назначения должен быть каталогом. Например, если задать ключ /I в команде копирования всех файлов с расширением txt из текущего каталога в несуществующий еще подкаталог TEXT,

XCOPY *.txt TEXT /I

то подкаталог TEXT будет создан без дополнительных запросов.

Ключи /Q, /F и /L отвечают за режим отображения при копировании. При задании ключа /Q имена файлов при копировании не отображаются, ключа /F — отображаются полные пути источника и результата. Ключ /L обозначает, что отображаются только файлы, которые должны быть скопированы (при этом само копирование не производится).

С помощью ключа /H можно копировать скрытые и системные файлы, а с помощью ключа /R — заменять файлы с атрибутом "Только для чтения". Например, для копирования всех файлов из корневого каталога диска C: (включая системные и скрытые) в каталог SYS на диске D:, нужно ввести следующую команду:

XCOPY C:*.* D:\SYS /H

Ключ /T позволяет применять XCOPY для копирования только структуры каталогов источника, без дублирования находящихся в этих каталогах файлов, причем пустые каталоги и подкаталоги не включаются. Для того, чтобы все же включить пустые каталоги и подкаталоги, нужно использовать комбинацию ключей /T /E.

Используя XCOPY можно при копировании обновлять только уже существующие файлы (новые файлы при этом не записываются). Для этого применяется ключ /U. Например, если в каталоге C:\2 находились файлы a.txt и b.txt, а в каталоге C:\1 — файлы a.txt, b.txt, c.txt и d.txt, то после выполнения команды

```
XCOPY C:\1 C:\2 /U
```

в каталоге C:\2 по-прежнему останутся лишь два файла a.txt и b.txt, содержимое которых будет заменено содержимым соответствующих файлов из каталога C:\1.Если с помощью `XCOPY` копировался файл с атрибутом "Только для чтения", то по умолчанию у файла-копии этот атрибут снимется. Для того, чтобы копировать не только данные, но и полностью атрибуты файла, необходимо использовать ключ `/K`.

Ключи `/Y` и `/-Y` определяют, нужно ли запрашивать подтверждение перед заменой файлов при копировании. `/Y` означает, что такой запрос нужен, `/-Y` — не нужен.

Команда DIR

Еще одной очень полезной командой является `DIR [диск:][путь][имя_файла] [ключи]`, которая используется для вывода информации о содержимом дисков и каталогов. Параметр `[диск:][путь]` задает диск и каталог, содержимое которого нужно вывести на экран. Параметр `[имя_файла]` задает файл или группу файлов, которые нужно включить в список. Например, команда

```
DIR C:\*.bat
```

выведет на экран все файлы с расширением bat в корневом каталоге диска C:. Если задать эту команду без параметров, то выводится метка диска и его серийный номер, имена (в коротком и длинном вариантах) файлов и подкаталогов, находящихся в текущем каталоге, а также дата и время их последней модификации. После этого выводится число файлов в каталоге, общий объем (в байтах), занимаемый файлами, и объем свободного пространства на диске. Например:

```
Том в устройстве C имеет метку PHYS1_PART2
 Серийный номер тома: 366D-6107
 Содержимое папки C:\aditor
.            <ПАПКА>     25.01.00  17:15 .
..           <ПАПКА>     25.01.00  17:15 ..
TEMPLT02 DAT         227  07.08.98   1:00 templt02.dat
UNINST1  000       1 093  02.03.99   8:36 UNINST1.000
HILITE   DAT       1 082  18.09.98  18:55 hilite.dat
TEMPLT01 DAT          48  07.08.98   1:00 templt01.dat
UNINST0  000      40 960  15.04.98   2:08 UNINST0.000
TTABLE   DAT         357  07.08.98   1:00 ttable.dat
ADITOR   EXE      461 312  01.12.99  23:13 aditor.exe
README   TXT        3 974  25.01.00  17:26 readme.txt
ADITOR   HLP       24 594  08.10.98  23:12 aditor.hlp
ТЕКСТО~1 TXT            0  11.03.01   9:02 Текстовый файл.txt
       11 файлов       533 647 байт
        2 папок     143 261 696 байт свободно
```

С помощью ключей команды `DIR` можно задать различные режимы расположения, фильтрации и сортировки. Например, при использовании ключа `/W` перечень файлов выводится в широком формате с максимально возможным числом имен файлов или каталогов на каждой строке. Например:

```
Том в устройстве C имеет метку PHYS1_PART2
 Серийный номер тома: 366D-6107
 Содержимое папки C:\aditor
[.]             [..]            TEMPLT02.DAT    UNINST1.000         HILITE.DAT
TEMPLT01.DAT    UNINST0.000         TTABLE.DAT      ADITOR.EXE
        README.TXT
ADITOR.HLP          ТЕКСТО~1.TXT
       11 файлов       533 647 байт
        2 папок     143 257 600 байт свободно
```

С помощью ключа /A[[:]атрибуты] можно вывести имена только тех каталогов и файлов, которые имеют заданные атрибуты (R — "Только чтение", A — "Архивный", S — "Системный", H — "Скрытый", префикс "–" имеет значение НЕ). Если ключ /A используется более чем с одним значением атрибута, будут выведены имена только тех файлов, у которых все атрибуты совпадают с заданными. Например, для вывода имен всех файлов в корневом каталоге диска C:, которые одновременно являются скрытыми и системными, можно задать команду

DIR C:\/A:HS

а для вывода всех файлов, кроме скрытых — команду

DIR C:\/A:-H

Отметим здесь, что атрибуту каталога соответствует буква D, то есть для того, чтобы, например, вывести список всех каталогов диска C:, нужно задать команду

DIR C: /A:D

Ключ /O[[:]сортировка] задает порядок сортировки содержимого каталога при выводе его командой DIR. Если этот ключ опущен, DIR печатает имена файлов и каталогов в том порядке, в котором они содержатся в каталоге. Если ключ /O задан, а параметр сортировка не указан, то DIR выводит имена в алфавитном порядке. В параметре сортировка можно использовать следующие значения: N — по имени (алфавитная), S — по размеру (начиная с меньших), E — по расширению (алфавитная), D — по дате (начиная с более старых), A — по дате загрузки (начиная с более старых), G — начать список с каталогов. Префикс "–" означает обратный порядок. Если задается более одного значения порядка сортировки, файлы сортируются по первому критерию, затем по второму и т.д.

Ключ /S означает вывод списка файлов из заданного каталога и его подкаталогов.

Ключ /B перечисляет только названия каталогов и имена файлов (в длинном формате) по одному на строку, включая расширение. При этом выводится только основная информация, без итоговой. Например:

templt02.dat
UNINST1.000
hilite.dat
templt01.dat
UNINST0.000
ttable.dat
aditor.exe
readme.txt
aditor.hlp
Текстовый файл.txt

Команды MKDIR и RMDIR

Для создания нового каталога и удаления уже существующего пустого каталога используются команды MKDIR [диск:]путь и RMDIR [диск:]путь [ключи] соответственно (или их короткие аналоги MD и RD). Например:

```
MKDIR "C:\Примеры"
RMDIR "C:\Примеры"
```

Команда MKDIR не может быть выполнена, если каталог или файл с заданным именем уже существует. Команда RMDIR не будет выполнена, если удаляемый каталог не пустой.

Команда DEL

Удалить один или несколько файлов можно с помощью команды

DEL [диск:][путь]имя_файла [ключи]

Для удаления сразу нескольких файлов используются групповые знаки ? и *. Ключ /S позволяет удалить указанные файлы из всех подкаталогов, ключ /F – принудительно удалить файлы, доступные только для чтения, ключ /A[[:]атрибуты] – отбирать файлы для удаления по атрибутам (аналогично ключу /A[[:]атрибуты] в команде DIR).

Команда REN

Переименовать файлы и каталоги можно с помощью команды RENAME (REN). Синтаксис этой команды имеет следующий вид:

REN [диск:][путь][каталог1|файл1] [каталог2|файл2]

Здесь параметр каталог1 | файл1 определяет название каталога/файла, которое нужно изменить, а каталог2 | файл2 задает новое название каталога/файла. В любом параметре команды REN можно использовать групповые символы ? и *. При этом представленные шаблонами символы в параметре файл2 будут идентичны соответствующим символам в параметре файл1. Например, чтобы изменить у всех файлов с расширением txt в текущей директории расширение на doc, нужно ввести такую команду:

REN *.txt *.doc

Если файл с именем файл2 уже существует, то команда REN прекратит выполнение, и произойдет вывод сообщения, что файл уже существует или занят. Кроме того, в команде REN нельзя указать другой диск или каталог для создания результирующих каталога и файла. Для этой цели нужно использовать команду MOVE, предназначенную для переименования и перемещения файлов и каталогов.

Команда MOVE

Синтаксис команды для перемещения одного или более файлов имеет вид:

MOVE [/Y|/–Y] [диск:][путь]имя_файла1[,...] результирующий_файл

Синтаксис команды для переименования папки имеет вид:

MOVE [/Y|/–Y] [диск:][путь]каталог1 каталог2

Здесь параметр результирующий_файл задает новое размещение файла и может включать имя диска, двоеточие, имя каталога, либо их сочетание. Если перемещается только один файл, допускается указать его новое имя. Это позволяет сразу переместить и переименовать файл. Например,

MOVE "C:\Мои документы\список.txt" D:\list.txt

Если указан ключ /–Y, то при создании каталогов и замене файлов будет выдаваться запрос на подтверждение. Ключ /Y отменяет выдачу такого запроса.

2. Язык интерпретатора Cmd.exe. Командные файлы

Язык оболочки командной строки (shell language) в Windows реализован в виде командных (или пакетных) файлов. Командный файл в Windows — это обычный текстовый файл с расширением bat или cmd, в котором записаны допустимые команды операционной системы (как внешние, так и внутренние), а также некоторые дополнительные инструкции и ключевые слова, придающие командным файлам некоторое сходство с алгоритмическими языками программирования.

2.1 Вывод сообщений и дублирование команд

По умолчанию команды пакетного файла перед исполнением выводятся на экран, что выглядит не очень эстетично. С помощью команды ECHO OFF можно отключить дублирование команд, идущих после нее (сама команда ECHO OFF при этом все же дублируется). Например,

```
REM Следующие две команды будут дублироваться на экране …
DIR C:\
ECHO OFF
REM А остальные уже не будут
DIR D:\
```

Для восстановления режима дублирования используется команда ECHO ON. Кроме этого, можно отключить дублирование любой отдельной строки в командном файле, написав в начале этой строки символ @, например:

```
ECHO ON
REM Команда DIR C:\ дублируется на экране
DIR C:\
REM А команда DIR D:\ — нет
@DIR D:\
```

Таким образом, если поставить в самое начало файла команду

```
@ECHO OFF
```

то это решит все проблемы с дублированием команд.

В пакетном файле можно выводить на экран строки с сообщениями. Делается это с помощью команды

```
ECHO сообщение
```

Например,

```
@ECHO OFF
ECHO Привет!
Команда ECHO. (точка должна следовать непосредственно за словом "ECHO")
выводит на экран пустую строку.
```

Часто бывает удобно для просмотра сообщений, выводимых из пакетного файла, предварительно полностью очистить экран командой `CLS`.

Используя механизм перенаправления ввода/вывода (символы > и >>), можно направить сообщения, выводимые командой `ECHO`, в определенный текстовый файл. Например:

```
@ECHO OFF
ECHO Привет! > hi.txt
ECHO Пока! >> hi.txt
С помощью такого метода можно, скажем,
 заполнять файлы-протоколы с отчетом о
  произведенных действиях. Например:
@ECHO OFF
REM Попытка копирования
XCOPY C:\PROGRAMS D:\PROGRAMS /s
REM Добавление сообщения в файл report.txt в случае
REM удачного завершения копирования
IF NOT ERRORLEVEL 1 ECHO Успешное копирование >> report.txt
```

2.2 Использование параметров командной строки

При запуске пакетных файлов в командной строке можно указывать произвольное число параметров, значения которых можно использовать внутри файла. Это позволяет, например, применять один и тот же командный файл для выполнения команд с различными параметрами.

Для доступа из командного файла к параметрам командной строки применяются символы `%0, %1, …, %9` или `%*`. При этом вместо `%0` подставляется имя выполняемого пакетного файла, вместо `%1, %2, …, %9` — значения первых девяти параметров командной строки соответственно, а вместо `%*` — все аргументы. Если в командной строке при вызове пакетного файла задано меньше девяти параметров, то "лишние" переменные из `%1 – %9` замещаются пустыми строками. Рассмотрим следующий пример. Пусть имеется командный файл copier.bat следующего содержания:

```
@ECHO OFF
CLS
ECHO Файл %0 копирует каталог %1 в %2
XCOPY %1 %2 /S
```

Если запустить его из командной строки с двумя параметрами, например

```
copier.bat C:\Programs D:\Backup
```

то на экран выведется сообщение

Файл copier.bat копирует каталог C:\Programs в D:\Backup

и произойдет копирование каталога C:\Programs со всеми его подкаталогами в D:\Backup.

При необходимости можно использовать более девяти параметров командной строки. Это достигается с помощью команды SHIFT, которая изменяет значения замещаемых параметров с %0 по %9, копируя каждый параметр в предыдущий, то есть значение %1 копируется в %0, значение %2 – в %1 и т.д. Замещаемому параметру %9 присваивается значение параметра, следующего в командной строке за старым значением %9. Если же такой параметр не задан, то новое значение %9 — пустая строка.

Рассмотрим пример. Пусть командный файл my.bat вызван из командной строки следующим образом:

my.bat p1 p2 p3

Тогда %0=my.bat, %1=p1, %2=p2, %3=p3, параметры %4 – %9 являются пустыми строками. После выполнения команды SHIFT значения замещаемых параметров изменятся следующим образом: %0=p1, %1=p2, %2=p3, параметры %3 – %9 – пустые строки.

При включении расширенной обработки команд SHIFT поддерживает ключ /n, задающий начало сдвига параметров с номера n, где n может быть числом от 0 до 9.

Например, в следующей команде:

SHIFT /2

параметр %2 заменяется на %3, %3 на %4 и т.д., а параметры %0 и %1 остаются без изменений.

Команда, обратная SHIFT (обратный сдвиг), отсутствует. После выполнения SHIFT уже нельзя восстановить параметр (%0), который был первым перед сдвигом. Если в командной строке задано больше десяти параметров, то команду SHIFT можно использовать несколько раз.

В командных файлах имеются некоторые возможности синтаксического анализа заменяемых параметров. Для параметра с номером n (%n) допустимы синтаксические конструкции (операторы), представленные в табл. 2.1.

Таблица 2.1. Операторы для заменяемых параметров

Операторы	Описание
%~Fn	Переменная %n расширяется до полного имени файла
%~Dn	Из переменной %n выделяется только имя диска
%~Pn	Из переменной %n выделяется только путь к файлу
%~Nn	Из переменной %n выделяется только имя файла
%~Xn	Из переменной %n выделяется расширение имени файла
%~Sn	Значение операторов N и X для переменной %n изменяется так, что они работают с кратким именем файла
%~$PATH:n	Проводится поиск по каталогам, заданным в переменной среды PATH, и переменная %n заменяется на полное имя первого найденного файла. Если переменная PATH не определена или в результате поиска не найден ни один файл, эта конструкция заменяется на пустую строку. Естественно, здесь переменную PATH можно заменить на любое другое допустимое значение

Данные синтаксические конструкции можно объединять друг с другом, например:

%~DPn — из переменной %n выделяется имя диска и путь,

%~NXn — из переменной %n выделяется имя файла и расширение.

Рассмотрим следующий пример. Пусть мы находимся в каталоге C:\TEXT и запускаем пакетный файл с параметром Рассказ.doc (%1=Рассказ.doc). Тогда применение операторов, описанных в табл. 2.1, к параметру %1 даст следующие результаты:

```
%~F1=C:\TEXT\Рассказ.doc
%~D1=C:
%~P1=\TEXT\
%~N1=Рассказ
%~X1=.doc
%DP1=C:\TEXT\
%NX1=Рассказ.doc
```

2.3 Работа с переменными среды

Внутри командных файлов можно работать с так называемыми переменными среды (или переменными окружения), каждая из которых хранится в оперативной памяти, имеет свое уникальное имя, а ее значением является строка. Стандартные переменные среды автоматически инициализируются в процессе загрузки операционной системы. Такими переменными являются, например, WINDIR, которая определяет расположение каталога Windows, TEMP, которая определяет путь к каталогу для хранения временных файлов Windows или PATH, в которой хранится системный путь (путь поиска), то есть список каталогов, в которых система должна искать выполняемые файлы или файлы совместного доступа (например, динамические библиотеки). Кроме того, в командных файлах с помощью команды `SET` можно объявлять собственные переменные среды.

Получение значения переменной

Для получения значения определенной переменной среды нужно имя этой переменной заключить в символы `%`. Например:

```
@ECHO OFF
CLS
REM Создание переменной MyVar
SET MyVar=Привет
REM Изменение переменной
SET MyVar=%MyVar%!
ECHO Значение переменной MyVar: %MyVar%
REM Удаление переменной MyVar
SET MyVar=
ECHO Значение переменной WinDir: %WinDir%
```

При запуске такого командного файла на экран выведется строка

```
Значение переменной MyVar: Привет!
Значение переменной WinDir: C:\WINDOWS
```

2.4 Преобразования переменных как строк

С переменными среды в командных файлах можно производить некоторые манипуляции. Во-первых, над ними можно производить операцию конкатенации (склеивания). Для этого нужно в команде `SET` просто написать рядом значения соединяемых переменных. Например,

```
  SET A=Раз
SET B=Два
SET C=%A%%B%
```

После выполнения в файле этих команд значением переменной `C` будет являться строка `'РазДва'`. Не следует для конкатенации использовать знак `+`, так как он будет воспринят просто в качестве символа. Например, после запуска файл следующего содержания

```
  SET A=Раз
SET B=Два
SET C=A+B
ECHO Переменная C=%C%
```

```
SET D=%A%+%B%
ECHO Переменная D=%D%
```
на экран выведутся две строки:
```
Переменная C=A+B
Переменная D=Раз+Два
```

2.5 Операции с переменными как с числами

При включенной расширенной обработке команд (этот режим в Windows XP используется по умолчанию) имеется возможность рассматривать значения переменных среды как числа и производить с ними арифметические вычисления. Для этого используется команда `SET` с ключом `/A`. Приведем пример пакетного файла add.bat, складывающего два числа, заданных в качестве параметров командной строки, и выводящего полученную сумму на экран:

```
@ECHO OFF
REM В переменной M будет храниться сумма
SET /A M=%1+%2
ECHO Сумма %1 и %2 равна %M%
REM Удалим переменную M
SET M=
```

2.6 Локальные изменения переменных

Все изменения, производимые с помощью команды `SET` над переменными среды в командном файле, сохраняются и после завершения работы этого файла, но действуют только внутри текущего командного окна. Также имеется возможность локализовать изменения переменных среды внутри пакетного файла, то есть автоматически восстанавливать значения всех переменных в том виде, в каком они были до начала запуска этого файла. Для этого используются две команды: `SETLOCAL` и `ENDLOCAL`. Команда `SETLOCAL` определяет начало области локальных установок переменных среды. Другими словами, изменения среды, внесенные после выполнения `SETLOCAL`, будут являться локальными относительно текущего пакетного файла. Каждая команда `SETLOCAL` должна иметь соответствующую команду `ENDLOCAL` для восстановления прежних значений переменных среды. Изменения среды, внесенные после выполнения команды `ENDLOCAL`, уже не являются локальными относительно текущего пакетного файла; их прежние значения не будут восстановлены по завершении выполнения этого файла.

2.7 Приостановка выполнения командных файлов

Для того, чтобы вручную прервать выполнение запущенного bat-файла, нужно нажать клавиши <Ctrl>+<C> или <Ctrl>+<Break>. Однако часто бывает необходимо программно приостановить выполнение командного файла в определенной строке с выдачей запроса на нажатие любой клавиши. Это делается с помощью команды PAUSE. Перед запуском этой команды полезно с помощью команды `ECHO` информировать пользователя о действиях, которые он должен произвести. Например:

```
ECHO Вставьте дискету в дисковод A: и нажмите любую клавишу
PAUSE
```

2.8 Операторы перехода

Командный файл может содержать метки и команды `GOTO` перехода к этим меткам. Любая строка, начинающаяся с двоеточия :, воспринимается при обработке командного файла как

метка. Имя метки задается набором символов, следующих за двоеточием до первого пробела или конца строки. Приведем пример.
Пусть имеется командный файл следующего содержания:

```
@ECHO OFF
COPY %1 %2
GOTO Label1
ECHO Эта строка никогда не выполнится
:Label1
REM Продолжение выполнения
DIR %2
После того, как в этом файле мы доходим до команды
GOTO Label1
его выполнение продолжается со строки
REM Продолжение выполнения
```

В команде перехода внутри файла `GOTO` можно задавать в качестве метки перехода строку `:EOF`, которая передает управление в конец текущего пакетного файла (это позволяет легко выйти из пакетного файла без определения каких-либо меток в самом его конце).
Также для перехода к метке внутри текущего командного файла кроме команды `GOTO` можно использовать и рассмотренную выше команду `CALL`:

```
CALL :метка аргументы
```

При вызове такой команды создается новый контекст текущего пакетного файла с заданными аргументами, и управление передается на инструкцию, расположенную сразу после метки. Для выхода из такого пакетного файла необходимо два раза достичь его конца. Первый выход возвращает управление на инструкцию, расположенную сразу после строки `CALL`, а второй выход завершает выполнение пакетного файла. Например, если запустить с параметром `"Копия-1"` командный файл следующего содержания:

```
@ECHO OFF
ECHO %1
CALL :2 Копия-2
:2
ECHO %1
```

то на экран выведутся три строки:

```
Копия-1
Копия-2
Копия-1
```

Таким образом, подобное использование команды `CALL` очень похоже на обычный вызов подпрограмм (процедур) в алгоритмических языках программирования.

2.9 Операторы условия

С помощью команды `IF ... ELSE` (ключевое слово `ELSE` может отсутствовать) в пакетных файлах можно выполнять обработку условий нескольких типов. При этом если заданное после `IF` условие принимает истинное значение, система выполняет следующую за условием команду (или несколько команд, заключенных в круглые скобки), в противном случае выполняется команда (или несколько команд в скобках), следующие за ключевым словом `ELSE`.

Проверка значения переменной

Первый тип условия используется обычно для проверки значения переменной. Для этого применяются два варианта синтаксиса команды `IF`:

```
IF [NOT] строка1==строка2 команда1 [ELSE команда2]
```

(квадратные скобки указывают на необязательность заключенных в них параметров) или

```
IF [/I] [NOT] строка1 оператор_сравнения строка2 команда
```

Рассмотрим сначала первый вариант. Условие `строка1==строка2` (здесь необходимо писать именно два знака равенства) считается истинным при точном совпадении обеих строк. Параметр `NOT` указывает на то, что заданная команда выполняется лишь в том случае, когда сравниваемые строки не совпадают.

Строки могут быть литеральными или представлять собой значения переменных (например, `%1` или `%TEMP%`). Кавычки для литеральных строк не требуются. Например,

```
IF %1==%2 ECHO Параметры совпадают!
IF %1==Петя ECHO Привет, Петя!
```

Отметим, что при сравнении строк, заданных переменными, следует проявлять определенную осторожность. Дело в том, что значение переменной может оказаться пустой строкой, и тогда может возникнуть ситуация, при которой выполнение командного файла аварийно завершится. Например, если вы не определили с помощью команды `SET` переменную `MyVar`, а в файле имеется условный оператор типа

```
IF %MyVar%==C:\ ECHO Ура!!!
```

то в процессе выполнения вместо `%MyVar%` подставится пустая строка и возникнет синтаксическая ошибка. Такая же ситуация может возникнуть, если одна из сравниваемых строк является значением параметра командной строки, так как этот параметр может быть не указан при запуске командного файла. Поэтому при сравнении строк нужно приписывать к ним в начале какой-нибудь символ, например:

```
IF -%MyVar%==-C:\ ECHO Ура!!!
```

С помощью команд `IF` и `SHIFT` можно в цикле обрабатывать все параметры командной строки файла, даже не зная заранее их количества. Например, следующий командный файл (назовем его primer.bat) выводит на экран имя запускаемого файла и все параметры командной строки:

```
@ECHO OFF
ECHO Выполняется файл: %0
ECHO.
ECHO Файл запущен со следующими параметрами...
REM Начало цикла
:BegLoop
IF -%1==- GOTO ExitLoop
ECHO %1
REM Сдвиг параметров
SHIFT
REM Переход на начало цикла
GOTO BegLoop
:ExitLoop
REM Выход из цикла
ECHO.
ECHO Все.
```

Если запустить primer.bat с четырьмя параметрами:

```
primer.bat А Б В Г
```

то в результате выполнения на экран выведется следующая информация:

```
Выполняется файл: primer.bat

Файл запущен со следующими параметрами:
А
Б
В
Г

Все.
```

Рассмотрим теперь оператор IF в следующем виде:
IF [/I] строка1 оператор_сравнения строка2 команда
Синтаксис и значение операторов_сравнения представлены в табл. 2.2.

Таблица 2.2. Операторы сравнения в IF

Оператор	Значение
EQL	Равно
NEQ	Не равно
LSS	Меньше
LEQ	Меньше или равно
GTR	Больше
GEQ	Больше или равно

Приведем пример использования операторов сравнения:

```
@ECHO OFF
CLS
IF -%1 EQL –Вася ECHO Привет, Вася!
IF -%1 NEQ –Вася ECHO Привет, но Вы не Вася!
```

Ключ /I, если он указан, задает сравнение текстовых строк без учета регистра. Ключ /I можно также использовать и в форме строка1==строка2 команды IF. Например, условие

```
IF /I DOS==dos …
```

будет истинным.

Проверка существования заданного файла

Второй способ использования команды IF — это проверка существования заданного файла. Синтаксис для этого случая имеет вид:

```
IF [NOT] EXIST файл команда1 [ELSE команда2]
```

Условие считается истинным, если указанный файл существует. Кавычки для имени файла не требуются. Приведем пример командного файла, в котором с помощью такого варианта команды IF проверяется наличие файла, указанного в качестве параметра командной строки.

```
@ECHO OFF
IF -%1==- GOTO NoFileSpecified
IF NOT EXIST %1 GOTO FileNotExist

REM Вывод сообщения о найденном файле
ECHO Файл '%1' успешно найден.
GOTO :EOF

:NoFileSpecified
REM Файл запущен без параметров
ECHO В командной строке не указано имя файла.
GOTO :EOF

:FileNotExist
REM Параметр командной строки задан, но файл не найден
ECHO Файл '%1' не найден.
```

Проверка наличия переменной среды

Аналогично файлам команда IF позволяет проверить наличие в системе определенной переменной среды:

```
IF DEFINED переменная команда1 [ELSE команда2]
```

Здесь условие DEFINED применяется подобно условию EXISTS наличия заданного файла, но принимает в качестве аргумента имя переменной среды и возвращает истинное значение, если эта переменная определена. Например:

```
@ECHO OFF
CLS
IF DEFINED MyVar GOTO :VarExists
ECHO Переменная MyVar не определена
GOTO :EOF
:VarExists
ECHO Переменная MyVar определена,
ECHO ее значение равно %MyVar%
```

Проверка кода завершения предыдущей команды

Еще один способ использования команды IF — это проверка кода завершения (кода выхода) предыдущей команды. Синтаксис для IF в этом случае имеет следующий вид:

```
IF [NOT] ERRORLEVEL число команда1 [ELSE команда2]
```

Здесь условие считается истинным, если последняя запущенная команда или программа завершилась с кодом возврата, равным либо превышающим указанное число.

Составим, например, командный файл, который бы копировал файл my.txt на диск C: без вывода на экран сообщений о копировании, а в случае возникновения какой-либо ошибки выдавал предупреждение:

```
@ECHO OFF
XCOPY my.txt C:\ > NUL
REM Проверка кода завершения копирования
IF ERRORLEVEL 1 GOTO ErrOccurred
ECHO Копирование выполнено без ошибок.
GOTO :EOF

:ErrOccurred
ECHO При выполнении команды XCOPY возникла ошибка!
```

В операторе IF ERRORLEVEL ... можно также применять операторы сравнения чисел, приведенные в табл. 2.2. Например:

```
IF ERRORLEVEL LEQ 1 GOTO Case1
```

2.10 Организация циклов

В командных файлах для организации циклов используются несколько разновидностей оператора FOR, которые обеспечивают следующие функции:

выполнение заданной команды для всех элементов указанного множества;
выполнение заданной команды для всех подходящих имен файлов;
выполнение заданной команды для всех подходящих имен каталогов;
выполнение заданной команды для определенного каталога, а также всех его подкаталогов;
получение последовательности чисел с заданными началом, концом и шагом приращения;
чтение и обработка строк из текстового файла;
обработка строк вывода определенной команды.

Цикл FOR ... IN ... DO ...

Самый простой вариант синтаксиса команды FOR для командных файлов имеет следующий вид:

```
FOR %%переменная IN (множество)
 DO команда [параметры]
```

Внимание

Перед названием переменной должны стоять именно два знака процента (%%), а не один, как это было при использовании команды FOR непосредственно из командной строки.

Сразу приведем пример. Если в командном файле заданы строки

```
@ECHO OFF
FOR %%i IN (Раз,Два,Три) DO ECHO %%i
```

то в результате его выполнения на экране будет напечатано следующее:

```
Раз
Два
Три
```

Параметр `множество` в команде `FOR` задает одну или более текстовых строк, разделенных запятыми, которые вы хотите обработать с помощью заданной команды. Скобки здесь обязательны. Параметр `команда [параметры]` задает команду, выполняемую для каждого элемента множества, при этом вложенность команд `FOR` на одной строке не допускается. Если в строке, входящей во множество, используется запятая, то значение этой строки нужно заключить в кавычки. Например, в результате выполнения файла с командами

```
@ECHO OFF
FOR %%i IN ("Раз,Два",Три) DO ECHO %%i
```

на экран будет выведено

```
Раз,Два
Три
```

Параметр `%%переменная` представляет подставляемую переменную (счетчик цикла), причем здесь могут использоваться только имена переменных, состоящие из одной буквы. При выполнении команда `FOR` заменяет подставляемую переменную текстом каждой строки в заданном множестве, пока команда, стоящая после ключевого слова `DO`, не обработает все такие строки.

Замечание.

Чтобы избежать путаницы с параметрами командного файла `%0 — %9`, для переменных следует использовать любые символы кроме `0 – 9`.

Параметр множество в команде `FOR` может также представлять одну или несколько групп файлов. Например, чтобы вывести в файл список всех файлов с расширениями txt и prn, находящихся в каталоге C:\TEXT, без использования команды `DIR`, можно использовать командный файл следующего содержания:

```
@ECHO OFF
FOR %%f IN (C:\TEXT\*.txt C:\TEXT\*.prn) DO ECHO %%f >> list.txt
```

При таком использовании команды `FOR` процесс обработки продолжается, пока не обработаются все файлы (или группы файлов), указанные во множестве.

Цикл FOR /D … IN … DO …

Следующий вариант команды `FOR` реализуется с помощью ключа /D:

```
FOR /D %переменная IN (набор) DO команда [параметры]
```

В случае, если набор содержит подстановочные знаки, то команда выполняется для всех подходящих имен каталогов, а не имен файлов. Скажем, выполнив следующий командный файл:

```
@ECHO OFF
CLS
FOR /D %%f IN (C:\*.*) DO ECHO %%f
```

мы получим список всех каталогов на диске C:, например:

```
C:\Arc
C:\CYR
C:\MSCAN
C:\NC
C:\Program Files
C:\TEMP
```

C:\TeX
C:\WINNT

Цикл FOR /R … IN … DO …

С помощью ключа `/R` можно задать рекурсию в команде: `FOR`:

```
FOR /R [[диск:]путь] %переменная IN (набор)
  DO команда [параметры]
```

В этом случае заданная команда выполняется для каталога `[диск:]путь`, а также для всех подкаталогов этого пути. Если после ключа `R` не указано имя каталога, то выполнение команды начинается с текущего каталога. Например, для распечатки всех файлов с расширением txt в текущем каталоге и всех его подкаталогах можно использовать следующий пакетный файл:

```
@ECHO OFF
CLS
FOR /R %%f IN (*.txt) DO PRINT %%f
```

Если вместо набора указана только точка `(.)`, то команда проверяет все подкаталоги текущего каталога. Например, если мы находимся в каталоге C:\TEXT с двумя подкаталогами BOOKS и ARTICLES, то в результате выполнения файла:

```
@ECHO OFF
CLS
FOR /R %%f IN (.) DO ECHO %%f
```

на экран выведутся три строки:

C:\TEXT\.
C:\TEXT\BOOKS\.
C:\TEXT\ARTICLES\.

Цикл FOR /L … IN … DO …

Ключ `/L` позволяет реализовать с помощью команды `FOR` арифметический цикл, в этом случае синтаксис имеет следующий: вид:

```
FOR /L %переменная IN (начало,шаг,конец) DO команда [параметры]
```

Здесь заданная после ключевого слова `IN` тройка (начало,шаг,конец) раскрывается в последовательность чисел с заданными началом, концом и шагом приращения. Так, набор (1,1,5) раскрывается в (1 2 3 4 5), а набор (5,-1,1) заменяется на (5 4 3 2 1). Например, в результате выполнения следующего командного файла:

```
@ECHO OFF
CLS
FOR /L %%f IN (1,1,5) DO ECHO %%f
```

переменная цикла `%%f` пробежит значения от 1 до 5, и на экране напечатаются пять чисел:

1
2
3
4
5

Числа, получаемые в результате выполнения цикла `FOR /L`, можно использовать в арифметических вычислениях. Рассмотрим командный файл my.bat следующего содержания:

```
@ECHO OFF
CLS
FOR /L %%f IN (1,1,5) DO CALL :2 %%f
GOTO :EOF
:2
SET /A M=10*%1
ECHO 10*%1=%M%
```

В третьей строке в цикле происходит вызов нового контекста файла my.bat с текущим значением переменной цикла `%%f` в качестве параметра командной строки, причем управление передается на метку `:2` (см. описание `CALL` в разделе "Изменения в командах перехода"). В шестой строке переменная цикла умножается на десять, и результат записывается в переменную `M`. Таким образом, в результате выполнения этого файла выведется следующая информация:

```
10*1=10
10*2=20
10*3=30
10*4=40
10*5=50
```

Цикл FOR /F ... IN ... DO ...

Самые мощные возможности (и одновременно самый запутанный синтаксис) имеет команда: `FOR` с ключом `/F`:

```
FOR /F ["ключи"] %переменная IN (набор)
 DO команда [параметры]
```

Здесь параметр набор содержит имена одного или нескольких файлов, которые по очереди открываются, читаются и обрабатываются. Обработка состоит в чтении файла, разбиении его на отдельные строки текста и выделении из каждой строки заданного числа подстрок. Затем найденная подстрока используется в качестве значения переменной при выполнении основного тела цикла (заданной команды).

По умолчанию ключ `/F` выделяет из каждой строки файла первое слово, очищенное от окружающих его пробелов. Пустые строки в файле пропускаются. Необязательный параметр `"ключи"` служит для переопределения заданных по умолчанию правил обработки строк. Ключи представляют собой заключенную в кавычки строку, содержащую приведенные в табл. 2.3 ключевые слова:

Таблица 2.3. Ключи в команде FOR /F

Ключ	Описание
`EOL=C`	Определение символа комментариев в начале строки (допускается задание только одного символа)
`SKIP=N`	Число пропускаемых при обработке строк в начале файла
`DELIMS=XXX`	Определение набора разделителей для замены заданных по умолчанию пробела и знака табуляции
`TOKENS=X,Y,M-N`	Определение номеров подстрок, выделяемых из каждой строки файла и передаваемых для выполнения в тело цикла

При использовании ключа `TOKENS=X,Y,M-N` создаются дополнительные переменные. Формат `M-N` представляет собой диапазон подстрок с номерами от `M` до `N`. Если последний символ в строке `TOKENS=` является звездочкой, то создается дополнительная переменная, значением которой будет весь текст, оставшийся в строке после обработки последней подстроки.

Разберем применение этой команды на примере пакетного файла parser.bat, который производит разбор файла myfile.txt:

```
@ECHO OFF
IF NOT EXIST myfile.txt GOTO :NoFile
FOR /F "EOL=; TOKENS=2,3* DELIMS=, " %%i IN
  (myfile.txt) DO @ECHO %%i %%j %%k
GOTO :EOF
:NoFile
ECHO Не найден файл myfile.txt!
```

Здесь во второй строке производится проверка наличия файла myfile.txt; в случае отсутствия этого файла выводится предупреждающее сообщение. Команда FOR в третьей строке обрабатывает файл myfile.txt следующим образом:
Пропускаются все строки, которые начинаются с символа точки с запятой (EOL=;).
Вторая и третья подстроки из каждой строки передаются в тело цикла, причем подстроки разделяются пробелами (по умолчанию) и/или запятыми (DELIMS=,).
В теле цикла переменная %%i используется для второй подстроки, %%j — для третьей, а %%k получает все оставшиеся подстроки после третьей.
В нашем примере переменная %%i явно описана в инструкции FOR, а переменные %%j и %%k описываются неявно с помощью ключа TOKENS=. Например, если в файле myfile.txt были записаны следующие три строки:

```
ААА БББББ ВВВВ,ГГГГГ ДДДД
ЕЕЕЕЕ,ЖЖЖЖ ЗЗЗЗ
;ККККК ЛЛЛЛЛ ММММММ
```

то в результате выполнения пакетного файла parser.bat на экран выведется следующее:

```
БББББ ВВВВ ГГГГГ ДДДД
ЖЖЖЖ ЗЗЗЗ
```

Замечание
Ключ TOKENS= позволяет извлечь из одной строки файла до 26 подстрок, поэтому запрещено использовать имена переменных, начинающиеся не с букв английского алфавита (a–z). Следует помнить, что имена переменных FOR являются глобальными, поэтому одновременно не может быть активно более 26 переменных.
Команда FOR /F также позволяет обработать отдельную строку. Для этого следует ввести нужную строку в кавычках вместо набора имен файлов в скобках. Строка будет обработана так, как будто она взята из файла. Например, файл следующего содержания:

```
@ECHO OFF
FOR /F "EOL=; TOKENS=2,3* DELIMS=, " %%i IN
 ("ААА БББББ ВВВВ,ГГГГГ ДДДД") DO @ECHO %%i %%j %%k
```

при своем выполнении напечатает

```
БББББ ВВВВ ГГГГГ ДДДД
```

Вместо явного задания строки для разбора можно пользоваться переменными среды, например:

```
@ECHO OFF
SET M=ААА БББББ ВВВВ,ГГГГГ ДДДД
FOR /F "EOL=; TOKENS=2,3* DELIMS=,
 " %%i IN ("%M%") DO @ECHO %%i %%j %%k
```

Наконец, команда FOR /F позволяет обработать строку вывода другой команды. Для этого следует вместо набора имен файлов в скобках ввести строку вызова команды в апострофах (не в кавычках!). Строка передается для выполнения интерпретатору команд cmd.exe, а вывод этой команды записывается в память и обрабатывается так, как будто строка вывода взята из файла. Например, следующий командный файл:

```
@ECHO OFF
CLS
ECHO Имена переменных среды:
ECHO.
FOR /F "DELIMS==" %%i IN ('SET') DO ECHO %%i
```

выведет перечень имен всех переменных среды, определенных в настоящее время в системе.
В цикле FOR допускается применение тех же синтаксических конструкций (операторов), что и для заменяемых параметров (табл. 2.4).

Таблица 2.4. Операторы для переменных команды FOR

Операторы	Описание
%~Fi	Переменная %i расширяется до полного имени файла
%~Di	Из переменной %i выделяется только имя диска
%~Pi	Из переменной %i выделяется только путь к файлу
%~Ni	Из переменной %i выделяется только имя файла
%~Xi	Из переменной %i выделяется расширение имени файла
%~Si	Значение операторов N и X для переменной %i изменяется так, что они работают с кратким именем файла

Замечание

Если планируется использовать расширения подстановки значений в команде FOR, то следует внимательно подбирать имена переменных, чтобы они не пересекались с обозначениями формата.

Например, если мы находимся в каталоге C:\Program Files\Far и запустим командный файл следующего содержания:

```
@ECHO OFF
CLS
FOR %%i IN (*.txt) DO ECHO %%~Fi
```

то на экран выведутся полные имена всех файлов с расширением txt:

```
C:\Program Files\Far\Contacts.txt
C:\Program Files\Far\FarFAQ.txt
C:\Program Files\Far\Far_Site.txt
C:\Program Files\Far\License.txt
C:\Program Files\Far\License.xUSSR.txt
C:\Program Files\Far\ReadMe.txt
C:\Program Files\Far\register.txt
C:\Program Files\Far\WhatsNew.txt
```

2.11 Циклы и связывание времени выполнения для переменных

Как и в рассмотренном выше примере с составными выражениями, при обработке переменных среды внутри цикла могут возникать труднообъяснимые ошибки, связанные с ранними связыванием переменных. Рассмотрим пример. Пусть имеется командный файл следующего содержания:

```
SET a=
FOR %%i IN (Раз,Два,Три) DO SET a=%a%%%i
ECHO a=%a%
```

В результате его выполнения на экран будет выведена строка "a=Три", то есть фактически команда

```
FOR %%i IN (Раз,Два,Три) DO SET a=%a%%%i
```

равносильна команде

```
FOR %%i IN (Раз,Два,Три) DO SET a=%%i
```

Для исправления ситуации нужно, как и в случае с составными выражениями, вместо знаков процента (%) использовать восклицательные знаки и предварительно включить режим связывания времени выполнения командой SETLOCAL ENABLEDELAYEDEXPANSION. Таким образом, наш пример следует переписать следующим образом:

```
SETLOCAL ENABLEDELAYEDEXPANSION
SET a=
FOR %%i IN (Раз,Два,Три) DO SET a=!a!%%i
```

ECHO a=%a%
В этом случае на экран будет выведена строка `"a=РазДваТри"`.

Лекция. Сервер сценариев Windows Script Host

С помощью Windows Script Host (WSH), предназначен для выполнения сценариев, написанные, в принципе, на любом языке (при условии, что для этого языка установлен соответствующий модуль (scripting engine), поддерживающим технологию ActiveX Scripting. В качестве стандартных языков поддерживаются Visual Basic Script Edition (VBScript) и JScript.
WSH предъявляет минимальные требования к объему оперативной памяти, и является очень удобным инструментом для автоматизации повседневных задач пользователей и администраторов операционной системы Windows. Используя сценарии WSH, можно непосредственно работать с файловой системой компьютера, а также управлять работой других приложений (серверов автоматизации). При этом возможности сценариев ограничены только средствами, которые предоставляют доступные серверы автоматизации.

1.1 Создание и запуск простейших сценариев WSH

Простейший WSH-сценарий, написанный на языке JScript или VBScript — это обычный текстовый файл с расширением js или vbs соответственно, создать его можно в любом текстовом редакторе, способном сохранять документы в формате "Только текст".
Размер сценария может изменяться от одной до тысяч строк, предельный размер ограничивается лишь максимальным размером файла в соответствующей файловой системе.
В качестве первого примера создадим JScript-сценарий, выводящий на экран диалоговое окно с надписью "Привет!". Для этого достаточно с помощью, например, стандартного Блокнота Windows (notepad.exe) создать файл First.js, содержащий всего одну строку:
WScript.Echo("Привет!");
Тот же самый сценарий на языке VBScript, естественно, отличается синтаксисом и выглядит следующим образом:
WScript.Echo "Привет!"

1.2 Запуск сценария из командной строки в консольном режиме

Можно выполнить сценарий из командной строки с помощью консольной версии WSH cscript.exe. Например, чтобы запустить сценарий, записанный в файле C:\Script\First.js, нужно загрузить командное окно и выполнить в нем команду
cscript C:\Script\First.js
Сценарий можно выполнить из командной строки с помощью (оконной) графической версии WSH wscript.exe. Для нашего примера в этом случае нужно выполнить команду
wscript C:\Script\First.js

1.3 Установка и изменение свойств сценариев

В случае необходимости для сценариев можно задавать различные параметры, влияющие на ход их выполнения. Для консольной (cscript.exe) и графической (wscript.exe) версий сервера сценариев эти параметры задаются по-разному.
Если сценарий запускается в консольном режиме, то его исполнение контролируется с помощью параметров командной строки для cscript.exe (см. табл. 1.1), которые включают или отключают различные опции WSH (все эти параметры начинаются с символов "//").

Таблица 1.1. Параметры командной строки для cscript.exe

Параметр	Описание
`//I`	Выключает пакетный режим (по умолчанию). При этом на экран будут выводиться все сообщения об ошибках в сценарии
`//B`	Включает пакетный режим. При этом на экран не будут выводиться никакие сообщения
`//T:nn`	Задает тайм-аут в секундах, т. е. сценарий будет выполняться nn секунд, после чего процесс прервется. По умолчанию время выполнения не ограничено
`//Logo`	Выводит (по умолчанию) перед выполнением сценария информацию о версии и разработчике WSH
`//Nologo`	Подавляет вывод информации о версии и разработчике WSH
`//H:CScipt` или `//H:Wscript`	Делает cscript.exe или wscript.exe приложением для запуска сценариев по умолчанию. Если эти параметры не указаны, то по умолчанию подразумевается wscript.exe
`//S`	Сохраняет установки командной строки для текущего пользователя
`//?`	Выводит встроенную подсказку для параметров командной строки
`//E:engine`	Выполняет сценарий с помощью модуля, заданного параметром engine
`//D`	Включает отладчик
`//X`	Выполняет программу в отладчике
`//Job:<JobID>`	Запускает задание с индексом `JobID` из многозадачного WS-файла (структура WS-файлов будет описана ниже)
`//U`	Позволяет использовать при перенаправлении ввода-вывода с консоли кодировку Unicode

Например, команда

```
cscript //Nologo C:\Script\First.js
```

запустит сценарий First.js без информации о версии WSH.

Если сценарий запускается в графическом режиме (с помощью wscript.exe), то свойства сценария можно устанавливать с помощью вкладки **Сценарий** (**Script**) диалогового окна, задающего свойства файла в Windows.

1.4 Собственная объектная модель WSH

`WScript`. Это главный объект WSH, который служит для создания других объектов или связи с ними, содержит сведения о сервере сценариев, а также позволяет вводить данные с клавиатуры и выводить информацию на экран или в окно Windows.

`WshArguments`. Обеспечивает доступ ко всем параметрам командной строки запущенного сценария или ярлыка Windows.

`WshNamed`. Обеспечивает доступ к именным параметрам командной строки запущенного сценария.

`WshUnnamed`. Обеспечивает доступ к безымянным параметрам командной строки запущенного сценария.

`WshShell`. Позволяет запускать независимые процессы, создавать ярлыки, работать с переменными среды, системным реестром и специальными папками Windows.

`WshSpecialFolders`. Обеспечивает доступ к специальным папкам Windows.

`WshShortcut`. Позволяет работать с ярлыками Windows.

`WshUrlShortcut`. Предназначен для работы с ярлыками сетевых ресурсов.

WshEnvironment. Предназначен для просмотра, изменения и удаления переменных среды.
WshNetwork. Используется при работе с локальной сетью: содержит сетевую информацию для локального компьютера, позволяет подключать сетевые диски и принтеры.
WshScriptExec. Позволяет запускать консольные приложения в качестве дочерних процессов, обеспечивает контроль состояния этих приложений и доступ к их стандартным входным и выходным потокам.
WshController. Позволяет запускать сценарии на удаленных машинах.
WshRemote. Позволяет управлять сценарием, запущенным на удаленной машине.
WshRemoteError. Используется для получения информации об ошибке, возникшей в результате выполнения сценария, запущенного на удаленной машине.
FileSystemObject обеспечивающий доступ к файловой системе компьютера.

1.5 Объект WScript

Свойства объекта WScript позволяют получить полный путь к использующемуся серверу сценариев (wscript.exe или cscript.exe), параметры командной строки, с которыми запущен сценарий, режим его работы (интерактивный или пакетный). Кроме этого, с помощью свойств объекта WScript можно выводить информацию в стандартный выходной поток и читать данные из стандартного входного потока. Также WScript предоставляет методы для работы внутри сценария с объектами автоматизации и вывода информации на экран (в текстовом режиме) или в окно Windows.
Отметим, что в сценарии WSH объект WScript можно использовать сразу, без какого-либо предварительного описания или создания, так как его экземпляр создается сервером сценариев автоматически. Для использования же всех остальных объектов нужно использовать либо метод CreateObject.
Свойства объекта WScript представлены в табл. 1.2.

Таблица 1.2. Свойства объекта WScript

Свойство	Описание
Application	Предоставляет интерфейс Idispatch для объекта WScript
Arguments	Содержит указатель на коллекцию WshArguments, содержащую параметры командной строки для исполняемого сценария
FullName	Содержит полный путь к исполняемому файлу сервера сценариев (в Windows XP обычно это C:\WINDOWS\SYSTEM32\CSCRIPT.EXE или C:\WINDOWS\SYSTEM32\WSCRIPT.EXE)
Name	Содержит название объекта Wscript (Windows Script Host)
Path	Содержит путь к каталогу, в котором находится cscript.exe или wscript.exe (в Windows XP обычно это C:\WINDOWS\SYSTEM32)
ScriptFullName	Содержит полный путь к запущенному сценарию
ScriptName	Содержит имя запущенного сценария
StdErr	Позволяет запущенному сценарию записывать сообщения в стандартный поток для ошибок
StdIn	Позволяет запущенному сценарию читать информацию из стандартного входного потока
StdOut	Позволяет запущенному сценарию записывать информацию в стандартный выходной поток
Version	Содержит версию WSH

Опишем более подробно некоторые свойства объекта WScript.

Свойство Arguments

В следующем примере с помощью цикла `For Each … Next` на экран выводятся все параметры командной строки, с которыми был запущен сценарий.

```
'**************************************************************
' Имя: Args.vbs
' Язык: VBScript
' Описание: Работа с аргументами запущенного сценария
'**************************************************************
Option Explicit

Dim i,objArgs,Arg
Set objArgs = WScript.Arguments  ' Создаем объект WshArguments
For Each Arg In objArgs
  WScript.Echo Arg  ' Формируем строки со значениями аргументов
Next

WScript.Echo s    ' Выводим сформированные строки
```

Свойства StdErr, StdIn, StdOut

Доступ к стандартным входным и выходным потокам с помощью свойств `StdIn`, `StdOut` и `StdErr` можно получить только в том случае, если сценарий запускался в консольном режиме с помощью cscript.exe. Если сценарий был запущен с помощью wscript.exe, то при попытке обратиться к этим свойствам возникнет ошибка `"Invalid Handle"`.

Таблица 1.3. Методы для работы с потоками

Метод	Описание
`Read(n)`	Считывает из потока `StdIn` заданное параметром n число символов и возвращает полученную строку
`ReadAll()`	Читает символы из потока `StdIn` до тех пор, пока не встретится символ конца файла ASCII 26 (<Ctrl>+<Z>), и возвращает полученную строку
`ReadLine()`	Возвращает строку, считанную из потока `StdIn`
`Skip(n)`	Пропускает при чтении из потока `StdIn` заданное параметром `n` число символов
`SkipLine()`	Пропускает целую строку при чтении из потока `StdIn`
`Write(string)`	Записывает в поток `StdOut` или `StdErr` строку `string` (без символа конца строки)
`WriteBlankLines(n)`	Записывает в поток `StdOut` или `StdErr` заданное параметром `n` число пустых строк
`WriteLine(string)`	Записывает в поток `StdOut` или `StdErr` строку `string` (вместе с символом конца строки)

Кроме этого, с помощью методов, работающих с входным потоком StdIn, можно организовывать диалог с пользователем, то есть создавать интерактивные сценарии.

```
'**************************************************************
'* Имя: Interact.vbs
'* Язык: VBScript
'* Описание: Ввод/вывод строк в консольном режиме
'**************************************************************
Dim  s
' Выводим строку на экран
WScript.StdOut.Write "Введите число: "
' Считываем строку
```

```
s = WScript.StdIn.ReadLine
' Выводим строку на экран
WScript.StdOut.WriteLine "Вы ввели число " & s
'*************  Конец ******************************************
```

1.6 Методы объекта WScript

Объект `WScript` имеет несколько методов:

Таблица 1.4. Методы объекта WScript

Метод	Описание
`CreateObject(strProgID [, strPrefix])`	Создает объект, заданный параметром `strProgID`
`ConnectObject(strObject, strPrefix)`	Устанавливает соединение с объектом `strObject`, позволяющее писать функции-обработчики его событий (имена этих функций должны начинаться с префикса `strPrefix`)
`DisconnectObject(obj)`	Отсоединяет объект `obj`, связь с которым была предварительно установлена в сценарии
`Echo([Arg1] [, Arg2] [,…])`	Выводит текстовую информацию на консоль или в диалоговое окно
`GetObject(strPathname [, strProgID], [strPrefix])`	Активизирует объект автоматизации, определяемый заданным файлом (параметр `strPathName`) или объект, заданный параметром `strProgID`
`Quit([intErrorCode])`	Прерывает выполнение сценария с заданным параметром `intErrorCode` кодом выхода. Если параметр `intErrorCode` не задан, то объект WScript установит код выхода равным нулю
`Sleep(intTime)`	Приостанавливает выполнения сценария (переводит его в неактивное состояние) на заданное параметром `intTime` число миллисекунд

Метод CreateObject

Строковый параметр `strProgID`, указываемый в методе `CreateObject`, называется программным идентификатором объекта (Programmic Identifier, ProgID).

Если указан необязательный параметр `strPrefix`, то после создания объекта в сценарии можно обрабатывать события, возникающие в этом объекте (естественно, если объект предоставляет интерфейсы для связи с этими событиями). Когда объект сообщает о возникновении определенного события, сервер сценариев вызывает функцию, имя которой состоит из префикса `strPrefix` и имени этого события. Например, если в качестве `strPrefix` указано `"MYOBJ_"`, а объект сообщает о возникновении события `"OnBegin,"` то будет запущена функция `"MYOBJ_OnBegin"`, которая должна быть описана в сценарии.

В следующем примере метод `CreateObject` используется для создания объекта `WshNetwork`:

```
set WshNetwork = WScript.CreateObject("WScript.Network")
```

Метод ConnectObject

Объект, соединение с которым осуществляется с помощью метода `ConnectObject`, должен предоставлять интерфейс к своим событиям.

В следующем примере в переменной `MyObject` создается абстрактный объект `"SomeObject"`, затем из сценария вызывается метод `SomeMetod` этого объекта. После этого устанавливается связь с переменной `MyObject` и задается префикс `"MyEvent"` для процедур обработки события этого объекта. Если в объекте возникнет событие с именем

"Event", то будет вызвана функция MyEvent_Event. Метод DisconnectObject объекта WScript производит отсоединение объекта MyObject.

```
var MyObject = WScript.CreateObject("SomeObject");
MyObject.SomeMethod();
WScript.ConnectObject(MyObject,"MyEvent");

function MyEvent_Event(strName){
  WScript.Echo(strName);
  }
WScript.DisconnectObject(MyObject);
```

Метод Echo

Параметры Arg1, Arg2, … метода Echo задают аргументы для вывода. Если сценарий был запущен с помощью wscript.exe, то метод Echo направляет вывод в диалоговое окно, если же для выполнения сценария применяется cscript.exe, то вывод будет направлен на экран (консоль).

```
'*****************************************************************
'* Имя: EchoExample.vbs
'* Язык: VBScript
'* Описание: Использование метода WScript.Echo
'*****************************************************************
WScript.Echo             'Выводим пустую строку
WScript.Echo 1,2,3       'Выводим числа
WScript.Echo "Привет!"   'Выводим строку
'*************  Конец ***************************************
```

Метод Sleep

В следующем примере сценарий переводится в неактивное состояние на 5 секунд:

```
'*****************************************************************
'* Имя: SleepExample.vbs
'* Язык: VBScript
'* Описание: Использование метода WScript.Sleep
'*****************************************************************
WScript.Echo "Сценарий запущен, отдыхаем…"
WScript.Sleep 5000
WScript.Echo "Выполнение сценария завершено"
'*************  Конец ***************************************
```

1.7 Объект WshShell

С помощью объекта WshShell можно запускать новый процесс, создавать ярлыки, работать с системным реестром, получать доступ к переменным среды и специальным папкам Windows. Создается этот объект следующим образом:

```
var WshShell=WScript.CreateObject("WScript.Shell");
```

Таблица 1.5. Свойства объекта WshShell

Свойство	Описание
CurrentDirectory	Здесь хранится полный путь к текущему каталогу (к каталогу, из которого был запущен сценарий)
Environment	Содержит объект WshEnvironment, который обеспечивает доступ к переменным среды операционной системы для Windows NT/2000/XP или к переменным среды текущего командного окна для Windows 9x
SpecialFolders	Содержит объект WshSpecialFolders для доступа к специальным папкам Windows (рабочий стол, меню Пуск (Start) и т. д.)

Опишем теперь методы, имеющиеся у объекта WshShell.

Таблица 1.6. Методы объекта WshShell

Метод	Описание
`AppActivate(title)`	Активизирует заданное параметром `title` окно приложения. Строка `title` задает название окна (например, `"calc"` или `"notepad"`) или идентификатор процесса (`Process ID, PID`)
`CreateShortcut(strPathname)`	Создает объект `WshShortcut` для связи с ярлыком Windows (расширение lnk) или объект `WshUrlShortcut` для связи с сетевым ярлыком (расширение url). Параметр `strPathname` задает полный путь к создаваемому или изменяемому ярлыку
`Environment(strType)`	Возвращает объект `WshEnvironment`, содержащий переменные среды заданного вида
`Exec(strCommand)`	Создает новый дочерний процесс, который запускает консольное приложение, заданное параметром `strCommand`. В результате возвращается объект `WshScriptExec`, позволяющий контролировать ход выполнения запущенного приложения и обеспечивающий доступ к потокам `StdIn`, `StdOut` и `StdErr` этого приложения
`ExpandEnvironmentStrings(strString)`	Возвращает значение переменной среды текущего командного окна, заданной строкой `strString` (имя переменной должно быть окружено знаками `"%"`)
`LogEvent(intType, strMessage [,strTarget])`	Протоколирует события в журнале Windows NT/2000/XP или в файле WSH.log. Целочисленный параметр `intType` определяет тип сообщения, строка `strMessage` — текст сообщения. Параметр `strTarget` может задаваться только в Windows NT/2000/XP, он определяет название системы, в которой протоколируются события (по умолчанию это локальная система). Метод `LogEvent` возвращает `true`, если событие записано успешно и `false` в противном случае
`Popup(strText,[nSecToWait], [strTitle], [nType])`	Выводит на экран информационное окно с сообщением, заданным параметром `strText`. Параметр `nSecToWait` задает количество секунд, по истечении которых окно будет автоматически закрыто, параметр `strTitle` определяет заголовок окна, параметр `nType` указывает тип кнопок и значка для окна
`RegDelete(strName)`	Удаляет из системного реестра заданный параметр или раздел целиком
`RegRead(strName)`	Возвращает значение параметра реестра или значение по умолчанию для раздела реестра
`RegWrite(strName, anyValue [,strType])`	Записывает в реестр значение заданного параметра или значение по умолчанию для раздела
`Run(strCommand, [intWindowStyle], [bWaitOnReturn])`	Создает новый независимый процесс, который запускает приложение, заданное параметром `strCommand`
`SendKeys(string)`	Посылает одно или несколько нажатий клавиш в активное окно (эффект тот же, как если бы вы нажимали эти клавиши на клавиатуре)

`SpecialFolders(strSpecFolder)`	Возвращает строку, содержащую путь к специальной папке Windows, заданной параметром `strSpecFolder`

Метод CreateShortcut

Этот метод позволяет создать новый или открыть уже существующий ярлык для изменения его свойств. Приведем пример сценария, в котором создаются два ярлыка — на сам выполняемый сценарий (объект `oShellLink`) и на сетевой ресурс (`oUrlLink`).

```
'*************************************************************
'* Имя: MakeShortcuts.vbs
'* Язык: VBScript
'* Описание: Создание ярлыков из сценария
'*************************************************************
Dim WshShell,oShellLink,oUrlLink
' Создаем объект WshShell
Set WshShell=WScript.CreateObject("WScript.Shell")
' Создаем ярлык на файл
Set oShellLink=WshShell.CreateShortcut("Current Script.lnk")
' Устанавливаем путь к файлу
oShellLink.TargetPath=WScript.ScriptFullName
' Сохраняем ярлык
oShellLink.Save

' Создаем ярлык на сетевой ресурс
Set oUrlLink = WshShell.CreateShortcut("Microsoft Web Site.URL")
' Устанавливаем URL
oUrlLink.TargetPath = "http://www.microsoft.com"
' Сохраняем ярлык
oUrlLink.Save
'*************  Конец ***********************************
```

Метод Environment

Параметр `strType` задает вид переменных среды, которые будут записаны в коллекции `WshEnvironment`; возможными значениями этого параметра являются `"System"` (переменные среды операционной системы), `"User"` (переменные среды пользователя), `"Volatile"` (временные переменные) или `"Process"` (переменные среды текущего командного окна).

```
'*************************************************************
'* Имя: ShowEnvir.vbs
'* Язык: VBScript
'* Описание: Получение значений некоторых переменных среды
'*************************************************************
Dim WshShell,WshSysEnv
' Создаем объект WshShell
Set WshShell = WScript.CreateObject("WScript.Shell")
' Создание коллекции WshEnvironment
Set WshSysEnv = WshShell.Environment("SYSTEM")
WScript.Echo WshSysEnv("OS")
WScript.Echo WshShell.Environment.Item("NUMBER_OF_PROCESSORS")
'*************  Конец ***********************************
```

Метод Run

Параметр `intWindowStyle` устанавливает вид окна для запускаемого приложения.

Таблица 1.7. Типы окна (intWindowStyle)

Параметр	Константа Visual Basic	Описание
0	`vbHide`	Прячет текущее окно и активизирует другое окно (показывает его и передает ему фокус)
1	`vbNormalFocus`	Активизирует и отображает окно. Если окно было минимизировано или максимизировано, система восстановит его первоначальное положение и размер. Этот

		флаг должен указываться сценарием во время первого отображения окна
2	vbMinimizedFocus	Активизирует окно и отображает его в минимизированном (свернутом) виде
3	vbMaximizedFocus	Активизирует окно и отображает его в максимизированном (развернутом) виде
4	vbNormalNoFocus	Отображает окно в том виде, в котором оно находилось последний раз. Активное окно при этом остается активным
5		Активизирует окно и отображает его в текущем состоянии
6	vbMinimizedNoFocus	Минимизирует заданное окно и активизирует следующее (в Z-порядке) окно
7		Отображает окно в свернутом виде. Активное окно при этом остается активным
8		Отображает окно в его текущем состоянии. Активное окно при этом остается активным
9		Активизирует и отображает окно. Если окно было минимизировано или максимизировано, система восстановит его первоначальное положение и размер. Этот флаг должен указываться, если производится восстановление свернутого окна (его нельзя использовать в методе Run)
10		Устанавливает режим отображения, опирающийся на режим программы, которая запускает приложение

Необязательный параметр bWaitOnReturn является логической переменной, дающей указание ожидать завершения запущенного процесса. Если этот параметр не указан или установлен в false, то после запуска из сценария нового процесса управление сразу же возвращается обратно в сценарий (не дожидаясь завершения запущенного процесса). Если же bWaitOnReturn установлен в true, то сценарий возобновит работу только после завершения вызванного процесса. При этом если параметр bWaitOnReturn равен true, то метод Run возвращает код выхода вызванного приложения. Если же bWaitOnReturn равен false или не задан, то метод Run всегда возвращает ноль.

```
'*****************************************************************
'* Имя: RetCode.vbs
'* Язык: VBScript
'* Описание: Вывод кода выхода запущенного приложения
'*****************************************************************
' Создаем объект WshShell
Set WshShell = WScript.CreateObject("WScript.Shell")
' Запускаем Блокнот и ожидаем завершения его работы
Return = WshShell.Run("notepad " + WScript.ScriptFullName, 1, true)
' Печатаем код возврата
WScript.Echo "Код возврата:", Return
'*************  Конец **********************************************
```

Метод SendKeys

Каждая клавиша задается одним или несколькими символами. Например, Для того чтобы задать нажатие друг за другом букв А, Б и В нужно указать в качестве параметра для SendKeys строку "АБВ": string="АБВ".

Несколько символов имеют в методе SendKeys специальное значение: +, ^, %, ~, (,). Для того чтобы задать один из этих символов, их нужно заключить в фигурные скобки ({}). Например, для задания знака плюс используется {+}. Квадратные скобки ([]) хотя и не имеют в методе SendKeys специального смысла, их также нужно заключать в фигурные

скобки. Кроме этого, для задания самих фигурных скобок следует использовать следующие конструкции: {{} (левая скобка) и {}} (правая скобка).

Таблица 1.8. Коды специальных клавишей для SendKeys

Названия клавиши	Код	Названия клавиши	Код
<Backspace>	{BACKSPACE}, {BS} или {BKSP}	<→>	{RIGHT}
<Break>	{BREAK}	<F1>	{F1}
<Caps Lock>	{CAPSLOCK}	<F2>	{F2}
<Del> или <Delete>	{DELETE} или {DEL}	<F3>	{F3}
<End>	{END}	<F4>	{F4}
<Enter>	{ENTER} или ~	<F5>	{F5}
<Esc>	{ESC}	<F6>	{F6}
<Home>	{HELP}	<F7>	{F7}
<Ins> или <Insert>	{INSERT} или {INS}	<F8>	{F8}
<Num Lock>	{NUMLOCK}	<F9>	{F9}
<Page Down>	{PGDN}	<F10>	{F10}
<Page Up>	{PGUP}	<F11>	{F11}
<Print Screen>	{PRTSC}	<F12>	{F12}
<Scroll Lock>	{SCROLLLOCK}	<F13>	{F13}
<Tab>	{TAB}	<F14>	{F14}
<↑>	{UP}	<F15>	{F15}
<←>	{LEFT}	<F16>	{F16}
<↓>	{DOWN}		

Таблица 5.8. Коды клавиш <Shift>, <Ctrl> и <Alt>

Клавиша	Код
<Shift>	+
<Ctrl>	^
<Alt>	%

Для того чтобы задать комбинацию клавиш, которую нужно набирать, удерживая нажатыми клавиши <Shift>, <Ctrl> или <Alt>, нужно заключить коды этих клавиш в скобки. Например, если требуется сымитировать нажатие клавиш G и S при нажатой клавише <Shift>, следует использовать последовательность "+(GS)". Для того же, чтобы задать одновременное нажатие клавиш <Shift>+<G>, а затем <S> (уже без <Shift>), используется "+GS".

В методе SendKeys можно задать несколько нажатий подряд одной и той же клавиши. Для этого необходимо в фигурных скобках указать код нужной клавиши, а через пробел — число нажатий. Например, {LEFT 42} означает нажатие клавиши <?> 42 раза подряд; {h 10} означает нажатие клавиши h 10 раз подряд.

```
'*************************************************************
'* Имя: RunCalc.vbs
'* Язык: VBScript
'* Описание: Активизация приложения с помощью имени окна
'*************************************************************
Dim WshShell
' Создаем объект WshShell
Set WshShell=WScript.CreateObject("WScript.Shell")
```

```
' Запускаем Калькулятор
WshShell.Run "calc"
' Приостанавливаем сценарий на 0,1 секунды
WScript.Sleep 100
' Активизируем Калькулятор
WshShell.AppActivate "Calculator"
' Приостановка сценария на 0,1 секунды
WScript.Sleep(100)
' Посылаем нажатия клавиш в Калькулятор
WshShell.SendKeys "1{+}"
WScript.Sleep 500
WshShell.SendKeys "2"
WScript.Sleep 500
WshShell.SendKeys "~"
WScript.Sleep 2500
```

1.9 Объект WshArguments

Объект `WshArguments` содержит коллекцию всех параметров командной строки запущенного сценария или ярлыка Windows. Этот объект можно создать только с помощью свойства `Arguments` объектов `WScript` и `WshShortcut`.

С помощью объекта `WshArguments` можно также выделять и отдельно обрабатывать аргументы сценария, у которых имеются имена (например, `/Name:Andrey`) и безымянные аргументы. Ясно, что использование именных параметров более удобно, так как в этом случае нет необходимости запоминать, в каком порядке должны быть записаны параметры при запуске того или иного сценария.

Для доступа к именным и безымянным аргументам используются соответственно два специальных свойства объекта `WshArguments`: `Named` и `Unnamed`. Свойство `Named` содержит ссылку на коллекцию `WshNamed`, свойство `Unnamed` — на коллекцию `WshUnnamed`.

Таким образом, обрабатывать параметры командной строки запущенного сценария можно тремя способами:

- просматривать полный набор всех параметров (как именных, так и безымянных) с помощью коллекции `WshArguments`;
- выделить только те параметры, у которых есть имена (именные параметры) с помощью коллекции `WshNamed`;
- выделить только те параметры, у которых нет имен (безымянные параметры) с помощью коллекции `WshUnnamed`.

```
'**********************************************************************
' Имя: Args.vbs
' Язык: VBScript
' Описание: Работа с аргументами запущенного сценария
'**********************************************************************
Option Explicit

Dim i,Arg,objArgs,s,objNamedArgs,objUnnamedArgs  ' Объявляем переменные

Set objArgs = WScript.Arguments  ' Создаем объект WshArguments
' Определяем общее количество аргументов
s="Всего аргументов: " & objArgs.Count() & vbCrLf
For Each Arg In objArgs
  s=s & Arg & vbCrLf  ' Формируем строки со значениями аргументов
Next

Set objUnnamedArgs=objArgs.Unnamed  ' Создаем объект WshUnnamed
' Определяем количество безымянных аргументов
s=s & vbCrLf & "Безымянных аргументов: " & objUnnamedArgs.length & vbCrLf
For Each Arg In objUnnamedArgs
```

```
  ' Формируем строки со значениями безымянных аргументов
  s=s & Arg & vbCrLf
Next

Set objNamedArgs=objArgs.Named  ' Создаем объект WshNamed
' Определяем количество именных аргументов
s=s & vbCrLf & "Именных аргументов: " & objNamedArgs.Length & vbCrLf
' Проверяем, существует ли аргумент /Имя:
If objNamedArgs.Exists("Имя") Then
  s=s & objNamedArgs("Имя") & vbCrLf
End If
' Проверяем, существует ли аргумент /Comp:
If objNamedArgs.Exists("Comp") Then
  s=s & objNamedArgs("Comp") & vbCrLf
End If

WScript.Echo s    ' Выводим сформированные строки
'*************  Конец ********************************************
```

1.10 Объект WshEnvironment

Объект `WshEnvironment` позволяет получить доступ к коллекции, содержащей переменные среды заданного типа (переменные среды операционной системы, переменные среды пользователя или переменные среды текущего командного окна). Этот объект можно создать с помощью свойства `Environment` объекта `WshShell` или одноименного его метода:

```
Set WshShell=WScript.CreateObject("WScript.Shell")
Set WshSysEnv=WshShell.Environment
Set WshUserEnv=WshShell.Environment("User")
```

Объект `WshEnvironment` имеет свойство `Length`, в котором хранится число элементов в коллекции (количество переменных среды), и методы `Count` и `Item`. Для того чтобы получить значение определенной переменной среды, в качестве аргумента метода `Item` указывается имя этой переменной в двойных кавычках.

```
'*****************************************************************
' Имя: Environment.vbs
' Язык: VBScript
' Описание: Работа с переменными среды
'*****************************************************************
Dim WshShell, WshSysEnv
Set WshShell=WScript.CreateObject("WScript.Shell")
Set WshSysEnv=WshShell.Environment
WScript.Echo "Системный путь:",WshSysEnv.Item("PATH")
```

Можно также просто указать имя переменной в круглых скобках после имени объекта:

```
WScript.Echo "Системный путь:",WshSysEnv("PATH")
```

Кроме этого у объекта `WshEnvironment` имеется метод `Remove(strName)`, который удаляет заданную переменную среды.

1.11 Объект WshSpecialFolders

При установке Windows всегда автоматически создаются несколько специальных папок (например, папка для рабочего стола (Desktop) или папка для меню Пуск (Start)), путь к которым впоследствии может быть тем или иным способом изменен. Объект `WshSpecialFolders` обеспечивает доступ к коллекции, содержащей пути к специальным папкам Windows; задание путей к таким папкам может понадобиться, например, для создания непосредственно из сценария ярлыков на рабочем столе. В Windows XP поддерживаются следующие имена специальных папок:

1. Desktop;

2. Favorites;
3. Fonts;
4. MyDocuments;
5. NetHood;
6. PrintHood;
7. Programs;
8. Recent;
9. SendTo;
10. StartMenu;
11. Startup;
12. Templates;
13. AllUsersDesktop;
14. AllUsersStartMenu;
15. AllUsersPrograms;
16. AllUsersStartup.

Объект `WshSpecialFolders` создается с помощью свойства `SpecialFolders` объекта `WshShell`:

```
var WshShell=WScript.CreateObject("WScript.Shell"),
    WshSpecFold=WshShell.SpecialFolders;
```

Сценарий, формирующий список всех имеющихся в системе специальных папок.

```
'*****************************************************************
' Имя: SpecFold1.vbs
' Язык: VBScript
' Описание: Вывод названий всех специальных папок Windows
'*****************************************************************
Option Explicit

Dim WshShell, WshFldrs, SpecFldr, s  ' Объявляем переменные
' Создаем объект WshShell
Set WshShell = WScript.CreateObject("Wscript.Shell")
' Создаем объект WshSpecialFolders
Set WshFldrs = WshShell.SpecialFolders
s="Список всех специальных папок:" & vbCrLf & vbCrLf
' Перебираем все элементы коллекции WshFldrs
For Each SpecFldr In WshFldrs
  ' Формируем строки с путями к специальным папкам
  s=s & SpecFldr & vbCrLf
Next
WScript.Echo s
```

Объект `WshSpecialFolders` также позволяет получить путь к конкретно заданной специальной папке.

```
'*****************************************************************
' Имя: SpecFold2.vbs
' Язык: VBScript
' Описание: Вывод названий заданных специальных папок Windows
'*****************************************************************
Option Explicit

Dim WshShell, WshFldrs, s   ' Объявляем переменные
' Создаем объект WshShell
Set WshShell = WScript.CreateObject("Wscript.Shell")
' Создаем объект WshSpecialFolders
Set WshFldrs = WshShell.SpecialFolders
' Формируем строки с путями к конкретным специальным папкам
s="Некоторые специальные папки:" & vbCrLf & vbCrLf
s=s+"Desktop:"+WshFldrs("Desktop") & vbCrLf
s=s+"Favorites:"+WshFldrs("Favorites") & vbCrLf
s=s+"Programs:"+WshFldrs("Programs")
WScript.Echo s   ' Выводим сформированные строки на экран
```

```
'*************  Конец ********************************************/
```

1.12 Сценарии WSH для доступа к файловой системе. Объектная модель FileSystemObject

Для работы с файловой системой из сценариев WSH предназначены восемь объектов, главным из которых является `FileSystemObject`. С помощью методов объекта `FileSystemObject` можно выполнять следующие основные действия:

1. копировать или перемещать файлы и каталоги;
2. удалять файлы и каталоги;
3. создавать каталоги;
4. создавать или открывать текстовые файлы;
5. создавать объекты `Drive`, `Folder` и `File` для доступа к конкретному диску, каталогу или файлу соответственно.

С помощью свойств объектов `Drive`, `Folder` и `File` можно получить детальную информацию о тех элементах файловой системы, с которыми они ассоциированы. Объекты `Folder` и `File` также предоставляют методы для манипулирования файлами и каталогами (создание, удаление, копирование, перемещение); эти методы в основном копируют соответствующие методы объекта `FileSystemObject`.

Кроме этого имеются три объекта-коллекции: `Drives`, `Folders` и `Files`. Коллекция `Drives` содержит объекты `Drive` для всех имеющихся в системе дисков, `Folders` — объекты `Folder` для всех подкаталогов заданного каталога, `Files` — объекты `File` для всех файлов, находящихся внутри определенного каталога.

Наконец, из сценария можно читать информацию из текстовых файлов и записывать в них данные. Методы для этого предоставляет объект `TextStream`.

Таблица 1.9. Выполнение основных файловых операций

Операция	Используемые объекты, свойства и методы
Получение сведений об определенном диске (тип файловой системы, метка тома, общий объем и количество свободного места и т.д.)	Свойства объекта `Drive`. Сам объект `Drive` создается с помощью метода `GetDrive` объекта `FileSystemObject`
Получение сведений о заданном каталоге или файле (дата создания или последнего доступа, размер, атрибуты и т.д.)	Свойства объектов `Folder` и `File`. Сами эти объекты создаются с помощью методов `GetFolder` и `GetFile` объекта `FileSystemObject`
Проверка существования определенного диска, каталога или файла	Методы `DriveExists`, `FolderExists` и `FileExists` объекта `FileSystemObject`
Копирование файлов и каталогов	Методы `CopyFile` и `CopyFolder` объекта `FileSystemObject`, а также методы `File.Copy` и `Folder.Copy`
Перемещение файлов и каталогов	Методы `MoveFile` и `MoveFolder` объекта `FileSystemObject`, или методы `File.Move` и `Folder.Move`
Удаление файлов и каталогов	Методы `DeleteFile` и `DeleteFolder` объекта `FileSystemObject`, или методы `File.Delete` и `Folder.Delete`
Создание каталога	Методы `FileSystemObject.CreateFolder` или `Folders.Add`
Создание текстового файла	Методы `FileSystemObject.CreateTextFile` или

	`Folder.CreateTextFile`
Получение списка всех доступных дисков	Коллекция `Drives`, содержащаяся в свойстве `FileSystemObject.Drives`
Получение списка всех подкаталогов заданного каталога	Коллекция `Folders`, содержащаяся в свойстве `Folder.SubFolders`
Получение списка всех файлов заданного каталога	Коллекция `Files`, содержащаяся в свойстве `Folder.Files`
Открытие текстового файла для чтения, записи или добавления	Методы `FileSystemObject.CreateTextFile` или `File.OpenAsTextStream`
Чтение информации из заданного текстового файла или запись ее в него	Методы объекта `TextStream`

Получение сведений о диске

В сценарий DriveInfo.vbs, который выводит на экран некоторые свойства диска C.

```
'*****************************************************************
' Имя: DriveInfo.vbs
' Язык: VBScript
' Описание: Вывод на экран свойств диска C
'*****************************************************************
'Объявляем переменные
Dim FSO,D,TotalSize,FreeSpace,s
'Создаем объект FileSystemObject
Set FSO = WScript.CreateObject("Scripting.FileSystemObject")
'Создаем объект Drive для диска C
Set D = FSO.GetDrive("C:")
s = "Информация о диске C:" & VbCrLf
'Получаем серийный номер диска
s = s &"Серийный номер: " & D.SerialNumber & VbCrLf
'Получаем метку тома диска
s = s & "Метка тома: " & D.VolumeName & VbCrLf
'Вычисляем общий объем диска в килобайтах
TotalSize = D.TotalSize/1024
s = s & "Объем: " & TotalSize & " Kb" & VbCrLf
'Вычисляем объем свободного пространства диска в килобайтах
FreeSpace = D.FreeSpace/1024
s = s & "Свободно: " & FreeSpace & " Kb" & VbCrLf
'Выводим свойства диска на экран
WScript.Echo s
'*************  Конец ***********************************************
```

Получение сведений о каталоге

В сценарии FolderInfo.vbs на экран выводятся свойства каталога, из которого был запущен сценарий.

```
'*****************************************************************
' Имя: FolderInfo.vbs
' Язык: VBScript
' Описание: Вывод на экран даты создания текущего каталога
'*****************************************************************
Dim FSO,WshShell,FoldSize,s  'Объявляем переменные

'Создаем объект FileSystemObject
Set FSO = WScript.CreateObject("Scripting.FileSystemObject")
'Создаем объект WshShell
Set WshShell = WScript.CreateObject("WScript.Shell")

'Определяем каталог, из которого был запущен сценарий
'(текущий каталог)
Set Folder = FSO.GetFolder(WshShell.CurrentDirectory)
'Получаем имя текущего каталога
```

```
s = "Текущий каталог: " & Folder.Name & VbCrLf
'Получаем дату создания текущего каталога
s = s & "Дата создания: " & Folder.DateCreated & VbCrLf
'Вычисляем размер текущего каталога в килобайтах
FoldSize=Folder.Size/1024
s = s & "Объем: " & FoldSize & " Kb" & VbCrLf
'Выводим информацию на экран
WScript.Echo s
'*************  Конец ********************************************
```

Получение сведений о файле

В сценарий FileInfo.vbs, в котором на экран выводятся некоторые свойства файла C:\boot.ini.

```
'*********************************************************************
' Имя: FileInfo.vbs
' Язык: VBScript
' Описание: Вывод на экран некоторых свойств файла
'*********************************************************************
Dim FSO,F,s   'Объявляем переменные

'Создаем объект FileSystemObject
Set FSO = WScript.CreateObject("Scripting.FileSystemObject")
'Создаем объект File
Set F = FSO.GetFile("C:\boot.ini")

'Получаем имя файла
s = "Файл: " & F.Name & VbCrLf
'Получаем дату создания файла
s = s & "Дата создания: " & F.DateCreated & VbCrLf
'Получаем тип файла
s = s & "Тип: " & F.Type & VbCrLf
'Выводим информацию на экран
WScript.Echo s
'*************  Конец ********************************************
```

Проверка существования диска, каталога или файла

В сценарий IsExistsFile.vbs, в котором проверяется наличие на диске C файла boot.ini.

```
'*********************************************************************
' Имя: IsExistsFile.vbs
' Язык: VBScript
' Описание: Проверка существования файла
'*********************************************************************
Dim FSO,FileName 'Объявляем переменные

'Создаем объект FileSystemObject
Set FSO = WScript.CreateObject("Scripting.FileSystemObject")

FileName = "C:\boot.ini"
if  FSO.FileExists(FileName)  Then
  'Выводим информацию на экран
  WScript.Echo "Файл " & FileName & " существует"
else
  'Выводим информацию на экран
  WScript.Echo "Файл " & FileName & " не существует"
end if
'*************  Конец ********************************************
```

1.13 Получение списка всех имеющихся дисков

Каждому из дисков компьютера (включая подключенные сетевые диски и дисководы со сменными носителями) соответствует элемент коллекции `Drives` (объект `Drive`). Таким образом, для построения списка дисков компьютера нужно в цикле перебрать все элементы коллекции `Drives`.

В сценарий ListDrives.vbs, в котором на экран выводятся сведения обо всех доступных дисках.

```
'********************************************************************
' Имя: ListDrives.vbs
' Язык: VBScript
' Описание: Получение списка всех имеющихся дисков
'********************************************************************
'Объявляем переменные
Dim FSO,s,ss,Drives,D

'Создаем объект FileSystemObject
Set FSO = WScript.CreateObject("Scripting.FileSystemObject")
'Создаем коллекцию дисков, имеющихся в системе
Set Drives = FSO.Drives
s = ""

'Перебираем все диски в коллекции
For Each D In Drives
  'Получаем букву диска
  s = s & D.DriveLetter
  s = s & " - "
  if (D.DriveType = 3) then  'Проверяем, не является ли диск сетевым
    'Получаем имя сетевого ресурса
    ss = D.ShareName
  else
    'Диск является локальным
    if (D.IsReady) then  'Проверяем готовность диска
      'Если диск готов, то получаем метку тома для диска
      ss = D.VolumeName
    else
      ss = "Устройство не готово"
    end if
  s = s & ss & VbCrLf
  end if
Next

'Выводим полученные строки на экран
WScript.Echo s
'*************  Конец **********************************************
```

Получение списка всех подкаталогов заданного каталога

Для построения списка всех подкаталогов определенного каталога можно воспользоваться коллекцией `Folders`, которая хранится в свойстве `SubFolders` соответствующего объекта `Folder` и содержит объекты `Folder` для всех подкаталогов.

В сценарий ListSubFold.vbs, в котором на экран выводятся названия всех подкаталогов каталога C:\Program Files.

```
'********************************************************************
' Имя: ListSubFold.vbs
' Язык: VBScript
' Описание: Получение списка всех подкаталогов заданного каталога
'********************************************************************
'Объявляем переменные
Dim FSO,F,SFold,SubFolders,Folder,s

'Создаем объект FileSystemObject
Set FSO = WScript.CreateObject("Scripting.FileSystemObject")
'Путь к каталогу
SFold = "C:\Program Files"
s = "Каталог " & SFold & VbCrLf
s = s & "Подкаталоги:" & VbCrLf
'Создаем объект Folder для каталога C:\Program Files
Set F=FSO.GetFolder(SFold)

'Создаем коллекцию подкаталогов каталога C:\Program Files
Set SubFolders = F.SubFolders
```

```
'Цикл по всем подкаталогам
For Each Folder In SubFolders
  'Добавляем строку с именем подкаталога
   s = s & Folder & VbCrLf
Next

'Выводим полученные строки на экран
WScript.Echo s
'*************  Конец ******************************************/
```

Получение списка всех файлов заданного каталога

В свойстве `Files` объекта `Folder`, соответствующего определенному каталогу, хранится коллекция находящихся в этом каталоге файлов (объектов `File`). В сценарии ListFiles.vbs, выводящий на экран названия всех файлов, которые содержатся в специальной папке Мои документы.

```
'*********************************************************************
' Имя: ListFiles.vbs
' Язык: VBScript
' Описание: Получение списка всех файлов заданного каталога
'*********************************************************************
'Объявляем переменные
Dim FSO,F,File,Files,WshShell,PathList,s

'Создаем объект FileSystemObject
Set FSO = WScript.CreateObject("Scripting.FileSystemObject")
'Создаем объект WshShell
Set WshShell = WScript.CreateObject("Wscript.Shell")
'Создаем объект WshSpecialFolders
Set WshFldrs = WshShell.SpecialFolders
'Определяем путь к папке Мои документы
PathList = WshFldrs.item("MyDocuments") & "\"
'Создаем объект Folder для папки Мои документы
Set F = FSO.GetFolder(PathList)
'Создаем коллекцию файлов каталога Мои документы
Set Files = F.Files
s = "Файлы из каталога " & PathList & VbCrLf
'Цикл по всем файлам
For Each File In Files
  'Добавляем строку с именем файла
  s = s & File.Name & VbCrLf
Next

'Выводим полученные строки на экран
WScript.Echo s
'*************  Конец ******************************************
```

Создание каталога

Создать новый каталог на диске можно либо с помощью метода `CreateFolder` объекта `FileSystemObject`, либо с помощью метода `Add` коллекции `Folders`. Оба эти метода используются в сценарии MakeFolder.vbs для создания в каталоге C:\Мои документы подкаталогов Новая папка и Еще одна новая папка.

```
'*********************************************************************
' Имя: MakeFolder.vbs
' Язык: VBScript
' Описание: Создание нового каталога
'*********************************************************************
'Объявляем переменные
Dim FSO, F, SubFolders

'Создаем объект FileSystemObject
Set FSO = WScript.CreateObject("Scripting.FileSystemObject")
'Создаем каталог C:\Program Files\Новая папка
```

```
FSO.CreateFolder("C:\Program Files\Новая папка")
0'Создаем объект Folder для каталога C:\Program Files
Set F = FSO.GetFolder("C:\Program Files")
'Создаем коллекцию подкаталогов каталога C:\Program Files
Set SubFolders = F.SubFolders
'Создаем каталог C:\Program Files\Еще одна новая папка
SubFolders.Add "Еще одна новая папка"
'*************  Конец ******************************************
```

Создание текстового файла

Для создания текстового файла используется метод `CreateTextFile` объекта `FileSystemObject`, который имеет один обязательный текстовый параметр (путь к создаваемому файлу) и два необязательных логических параметра (`Overwrite` и `Unicode`).

Параметр `Overwrite` имеет значение в том случае, когда создаваемый файл уже существует. Если `Overwrite` равно `True`, то такой файл перепишется (старое содержимое будет утеряно), если же в качестве `Overwrite` указано `False`, то файл переписываться не будет. Если этот параметр вообще не указан, то существующий файл также не будет переписан.

```
'*********************************************************************
' Имя: CreateTempFile.vbs
' Язык: VBScript
' Описание: Создание временного файла со случайным именем
'*********************************************************************
Dim FSO,FileName,F,s  'Объявляем переменные
'Создаем объект FileSystemObject
Set FSO = WScript.CreateObject("Scripting.FileSystemObject")
'Генерируем случайное имя файла
FileName = FSO.GetTempName
'Создаем файл с именем FileName
Set F = FSO.CreateTextFile(FileName, true)
'Закрываем файл
F.Close
'Сообщаем о создании файла
WScript.Echo "Был создан файл " & FileName
'*************  Конец ******************************************
```

Операции чтения и записи строк для текстового файла

Последовательный (строка за строкой) доступ к текстовому файлу обеспечивает объект `TextStream`. Методы этого объекта позволяют читать информацию из файла и записывать ее в него. Создается объект `TextStream` при открытии или создании текстового файла с помощью следующих методов:

1. CreateTextFile объектов FileSystemObject и Folder;
2. OpenTextFile объекта FileSystemObject;
3. OpenAsTextStream объекта File.

Таблица 1.10. Параметр Iomode

Константа	Значение	Описание
`ForReading`	1	Файл открывается только для чтения, записывать информацию в него нельзя
`ForWriting`	2	Файл открывается для записи. Если файл с таким именем уже существовал, то при новой записи его содержимое теряется
`ForAppending`	8	Файл открывается для добавления. Если файл уже существовал, то информация будет дописываться в конец этого файла

Таблица 1.11. Параметр Format

Константа	Значение	Описание
TristateUseDefault	-2	Файл открывается в формате, используемом системой по умолчанию
TristateTrue	-1	Файл открывается в формате Unicode
TristateFalse	0	Файл открывается в формате ASCII

```
'*****************************************************************
' Имя: TextFile.vbs
' Язык: VBScript
' Описание: Запись строк в текстовый файл и чтение из него
'*****************************************************************
Dim FSO,F,TextStream,s  'Объявляем переменные
' Инициализируем константы
Const ForReading = 1, ForWriting = 2, TristateUseDefault = -2

' Создаем объект FileSystemObject
Set FSO=WScript.CreateObject("Scripting.FileSystemObject")
' Создаем в текущем каталоге файл test1.txt
FSO.CreateTextFile "test1.txt"
' Создаем объект File для файла test1.txt
set F=FSO.GetFile("test1.txt")
' Создаем объект TextStream (файл открывается для записи)
Set TextStream=F.OpenAsTextStream(ForWriting, TristateUseDefault)
' Записываем в файл строку
TextStream.WriteLine "Это первая строка"
' Закрываем файл
TextStream.Close
' Открываем файл для чтения
Set TextStream=F.OpenAsTextStream(ForReading, TristateUseDefault)
' Считываем строку из файла
s=TextStream.ReadLine
' Закрываем файл
TextStream.Close
' Отображаем строку на экране
WScript.Echo "Первая строка из файла test1.txt:" & vbCrLf & vbCrLf & s
'*************  Конец **************************************
```

Копирование и перемещение файлов и каталогов

Для копирования файлов/каталогов можно применять метод CopyFile/CopyFolder объекта FileSystemObject или метод Copy соответствующего этому файлу/каталогу объекта File/Folder. Перемещаются файлы/каталоги с помощью методов MoveFile/MoveFolder объекта FileSystemObject или метода Move соответствующего этому файлу/каталогу объекта File/Folder.

В сценарий CopyFile.vbs, иллюстрирующий использование метода Copy. В этом сценариях на диске C создается файл TestFile.txt, который затем копируется на рабочий стол.

```
'*****************************************************************
' Имя: CopyFile.vbs
' Язык: VBScript
' Описание: Создание и копирование файла
'*****************************************************************
'Объявляем переменные
Dim FSO,F,WshShell,WshFldrs,PathCopy

'Создаем объект FileSystemObject
Set FSO = WScript.CreateObject("Scripting.FileSystemObject")
'Создаем файл
Set F = FSO.CreateTextFile("C:\TestFile.txt", true)
'Записываем в файл строку
F.WriteLine "Тестовый файл"
'Закрываем файл
```

```
F.Close

'Создаем объект WshShell
Set WshShell = WScript.CreateObject("Wscript.Shell")
'Создаем объект WshSpecialFolders
Set WshFldrs = WshShell.SpecialFolders
'Определяем путь к рабочему столу
PathCopy = WshFldrs.item("Desktop")+"\"
'Создаем объект File для файла C:\TestFile.txt
Set F = FSO.GetFile("C:\TestFile.txt")
'Копируем файл на рабочий стол
F.Copy PathCopy
'*************  Конец ********************************************
```

Удаление файлов и каталогов

Для удаления файлов/каталогов можно применять метод `DeleteFile/DeleteFolder` объекта `FileSystemObject` или метод `Delete` соответствующего этому файлу/каталогу объекта `File/Folder`. Отметим, что при удалении каталога неважно, является ли он пустым или нет — удаление будет произведено в любом случае. Если же заданный для удаления файл/каталог не будет найден, то возникнет ошибка.

В сценарий DeleteFile.vbs, в котором производится удаление предварительно созданного файла C:\TestFile.txt.

```
'*********************************************************************
' Имя: DeleteFile.vbs
' Язык: VBScript
' Описание: Создание и удаление файла
'*********************************************************************
'Объявляем переменные
Dim FSO,F,FileName

'Создаем объект FileSystemObject
Set FSO = WScript.CreateObject("Scripting.FileSystemObject")
'Задаем имя файла
FileName="C:\TestFile.txt"
'Создаем файл
Set F = FSO.CreateTextFile(FileName, true)
'Записываем в файл строку
F.WriteLine "Тестовый файл"
'Закрываем файл
F.Close
WScript.Echo "Файл создан"
FSO.DeleteFile FileName
WScript.Echo "Файл удален"
'*************  Конец ********************************************
```

Пример

Создать сценарий реализующий в консольном режиме диалог с пользователем в виде меню реализующий: создание файла со списком файлов с расширением .ini", создание ярлыка на заданный сетевой ресурс.

```
'* Имя: Interact.vbs
'* Язык: VBScript
'* Описание: пример лаболаторной работы
'*********************************************************************
Dim  s
' Выводим строку на экран
do
 WScript.StdOut.WriteLine "МЕНЮ:"
 WScript.StdOut.WriteLine "--------------------------"
 WScript.StdOut.WriteLine "1. Информация о авторе"
 WScript.StdOut.WriteLine "2. Создание файла со списком файлов с расширением
.ini"
 WScript.StdOut.WriteLine "3. Создание ярлыка на заданный сетевой ресурс"
 WScript.StdOut.WriteLine "4. Выход"
 WScript.StdOut.Write "Выберите пункт меню:"
 ' Считываем строку
 s = WScript.StdIn.ReadLine
 ' Создаем объект WshShell
 Set WshShell = WScript.CreateObject("WScript.Shell")

 if (s="1") Then
      WScript.StdOut.WriteLine "ФИО, группа"

 elseif(s="2") Then
      WScript.StdOut.Write "Введите имя файла:"
  f = WScript.StdIn.ReadLine

      ' Запускаем командный файл и ожидаем окончания ее работы
  Code=WshShell.Run("%COMSPEC% /c 02.cmd >"+f ,0,true)

      elseif(s="3") Then
      WScript.StdOut.Write "Введите имя ярлыка:"
  f = WScript.StdIn.ReadLine
  ' Создаем ярлык на сетевой ресурс
  Set oUrlLink = WshShell.CreateShortcut(f+".URL")
      WScript.StdOut.Write "Введите имя ресурса:"
  f = WScript.StdIn.ReadLine
  ' Устанавливаем URL
  oUrlLink.TargetPath = f
  ' Сохраняем ярлык
  oUrlLink.Save
End if

loop until (s="4")

'*************  Конец ***********************************************
```

Командный файл 02.cmd

```
@ECHO OFF
CLS
FOR /R c:\ %%f IN (*.ini) DO echo %%f
```

Лекция. Командный интерфейс ОС Unix(Linux)

1.1 Организация файловой системы

Файловая система ОС Линукс (как и прочих unix-подобных систем) устроена так, что все ресурсы представлены единообразно, в виде файлов. Такой подход позволяет обеспечить универсальный интерфейс доступа к любым ресурсам: от физических устройств, до процессов, выполняющихся в системе. С точки зрения пользователя файловая система представляет логическую структуру каталогов и файлов. С другой стороны, невидимой пользователю, внутреннее устройство файловой системы реализует физические операции чтения/записи файлов на различные носители, алгоритмы доступа и многое другое.

Типы файлов

Для обеспечения единообразного доступа к файлам их прежде всего необходимо классифицировать. В Линукс это сделано следующим образом:

1. обычные (regular) файлы - текстовые, исполняемые, графические и пр. файлы, создаваемые пользователями и прикладными программами;
2. каталоги (directories) - именованные группы файлов и вложенных каталогов (т.е. содержимое каталога - суть файлы и другие каталоги);
3. файлы устройств (devices) - соответствуют присутствующим в системе реальным (жесткие диски, принтеры, мыши, ЦП и т.д.) устройствам и т.н. псевдоустройствам (например, /dev/null). Файлы устройств представляют символьные (последовательного доступа) и блочные (произвольного доступа) устройства. К первыми относятся, например, параллельные и последовательные порты, ко вторым - жесткие диски;
4. специальные файлы - сокеты (sockets) и именованные каналы (named pipes), которые предназначены для обмена информацией между процессами;
5. символьные ссылки (symlinks) - именованные указатели на физические файлы (аналог ярлыков ОС Windows), содержащие только путь к некоторому файлу. Символьные ссылки могут указывать на файлы, хранящиеся как локальных, так и в сетевых каталогах.

Символьные ссылки (или "мягкие") не нужно путать с "жесткими", которые указывают на inode файла. Inode (идентификатор узла) - это уникальный числовой идентификатор узла (файла или каталога) файловой системы, по которому и осуществляется доступ к нему. Имя же файла (включая полный путь) ориентировано на пользовательское восприятие. Для оператора проще оперировать с осмысленными именами файлов (например: report.txt, myfoto.jpg и т.п.), чем с числовыми значениями. Прочие отличия "жестких" и "мягких" ссылок вам предстоит выяснить в ходе выполнения этой лабораторной работы.

Каталоги Линукс

Все файлы упорядочены по каталогам. Структура и назначение каждого из каталогов, созданных на этапе установке предопределены, хотя и могут быть (что крайне не рекомендуется) изменены суперпользователем.

Файловая система имеет иерархическую структуру и начинается от корневого каталога (/). Его подкаталогами являются:

1. /bin - исполняемые файлы общего назначения;
2. /boot - содержит образ загружаемого ядра;

3. /dev - файлы устройств;
4. /etc - конфигурационные файлы общего пользования;
5. /home - домашние каталоги пользователей, включая программы и файлы личных предпочтений;
6. /lib - общесистемные библиотеки;
7. /mnt - каталог монтирования внешних файловых систем;
8. /proc - виртуальная файловая система для чтения информации о процессах;
9. /root - домашний каталог суперпользователя;
10. /sbin - программы системного администрирования;
11. /tmp - каталог для хранения временной информации;
12. /usr - каталог пользовательских прикладных программ со всеми их исполнимыми и конфигурационными файлами. Например, в подкаталог /usr/local инсталлируются программы, не входящие в дистрибутив Линукс, или собираемые из исходных текстов.
13. /var - каталог для хранения часто изменяющихся файлов. Например, спулера печати, различных лог-файлов, почтовых сообщений и т.п.
14. /lost+found - каталог для нарушенных фрагментов файлов, обнаруженных в результате проверки файловой системы после сбоя.

Такая стуктура типична для большинства дистрибутивов Линукс, но могут иметься и дополнительные каталоги.

Именование файлов и каталогов

Файловая система Линукс поддерживает "длинные" имена, содержащие символы латиницы, национальных алфавитов, знаки пунктуации и спецсимволы. Абсолютно запрещенными к использованию в имени являются прямой и обратный слэши (/ и \). Максимальное количество символов в имени - 255. Понятие "расширения файла" в unix-системах отсутсвует как таковое, поэтому в имени может быть несколько частей, разделенных точками. Все имена - регистрозависимые.

Приведенные выше правила справедливы и для каталогов.

Файлы и каталоги, названия которых начинаются с точки (т.н. dot-файлы), являются аналогами "скрытых" файлов MS-DOS. Т.е. в общем случае они не отображаются при просмотре содержимого файловой системы.

Для быстрого доступа к файлам в оболочке имеются несколько переменных окружения, хранящих соответвующие пути. Это, например, переменная $HOME, в которой содержится пути к домашнему каталогу текущего пользователя. Т.е. действия команд

```
[usr1@localhost var]$ cd /home/usr1
```

и

```
[usr1@localhost var]$ cd $HOME
```

приведут к одному результату - переходу в домашний каталог пользователя usr1. Более того, в оболочке определен псевдоним для домашнего каталога - символ ~ (тильда) можно использовать аналогично $HOME. Например:

```
[usr1@localhost var]$ cd ~
```

[usr1@localhost ~]$ pwd

/home/usr1

[usr1@localhost var]$

1.2 Регистрация пользователя в системе

Для входа пользователя с терминала в многопользовательскую операционную систему LINUX необходимо зарегистрироваться в качестве пользователя. Для этого нужно после сообщения Login: ввести системное имя пользователя, например, "student". Если имя задано верно, выводится запрос на ввод пароля:

Password:

Наберите пароль "student" и нажмите клавишу Enter.

Если имя или пароль указаны неверно, сообщение login повторяется. Значение пароля проверяется в системном файле password, где приводятся и другие сведения о пользователях. После правильного ответа появляется приветствие LINUX и приглашение:

student@linux:>

Вы получили доступ к ресурсам ОС LINUX.

Выход из системы:

exit - окончание сеанса пользователя.

После успешного ввода имени пользователя и пароля система выводит приглашение к вводу команды.

- для суперпользователя root

$ - для всех остальных пользователей

Часто при первом входе в систему пользователя требуется поменять пароль, назначенный пользователю администратором - используйте команду passwd.

$ passwd

2. Командный интерфейс

Формат команд в ОС LINUX следующий:

имя команды [аргументы] [параметры] [метасимволы]

Имя команды может содержать любое допустимое имя файла; аргументы - одна или несколько букв со знаком минус (-); параметры - передаваемые значения для обработки; метасимволы интерпретируются как специальные операции. В квадратных скобках указываются необязательные части команд.

2.1. Группирование команд

Группы команд или сложные команды могут формироваться с помощью специальных символов (метасимволов):

1. & - процесс выполняется в фоновом режиме, не дожидаясь окончания предыдущих процессов;

2. ? - шаблон, распространяется только на один символ;

3. - шаблон, распространяется на все оставшиеся символы;

4. | - программный канал - стандартный вывод одного процесса является стандартным вводом другого;

5. - переадресация вывода в файл;

6. < - переадресация ввода из файла;

7. ; - если в списке команд команды отделяются друг от друга точкой с запятой, то они выполняются друг за другом;

8. && - эта конструкция между командами означает, что последующая команда выпол8няется только при нормальном завершении предыдущей команды (код возврата 0);

9. || - последующая команда выполняется только, если не выполнилась предыдущая команда (код возврата 1);

10. () - группирование команд в скобки;

11. { } - группирование команд с объединенным выводом;

12. [] - указание диапазона или явное перечисление (без запятых);

13. >> - добавление содержимого файла в конец другого файла.

Примеры.

who | wc - подсчет количества работающих пользователей командой wc (word count - счет слов);

cat text.1 > text.2 - содержимое файла text.1 пересылается в файл text.2;

mail student < file.txt - электронная почта передает файл file.txt всем пользователям, перечисленным в командной строке;

cat text.1,text.2 - просматриваются файлы text.1 и text.2;

cat text.1 >> text.2 - добавление файла text.1 в конец файла text.2;

cc primer.c & - трансляция СИ - программы в фоновом режиме.

cc -o primer.o primer.c - трансляция СИ-программы с образованием файла выполняемой программы с именем primer.o;

rm text.* - удаление всех файлов с именем text;

{cat text.1; cat text.2} | lpr - просмотр файлов text.1 и text.2 и вывод их на печать;

2.2 Основные команды ОС

man <название команды> - вызов электронного справочника об указанной команде. Выход из справочника - нажатие клавиши Q.

Команда man man сообщает информацию о том, как пользоваться справочником.

echo выдача на стандартный вывод строки символов, которая задана ей в качестве аргумента. Синтаксис команды echo:

echo [-n] [arg1] [arg2] [arg3]...

Команда помещает в стандартный вывод свои аргументы, разделенные пробелами и завершаемые символом перевода строки. При наличии флага -n символ перевода строки исключается. Передаваемая строка может быть перенаправлена в файл с использованием оператора перенаправления вывода > . Например:

$echo "Hello, world!" > myfile

who [am i] - получение информации о работающих пользователях.

В квадратных скобках указываются аргументы команды, которые можно опустить. Ответ представляется в виде таблицы, которая содержит следующую информацию:

1. идентификатор пользователя;

2. идентификатор терминала;

3. дата подключения;

4. время подключения.

date - вывод на экран текущей даты и текущего времени.

cal [[месяц]год] - календарь; если календарь не помещается на одном экране, то используется команда cal год | more и клавишей пробела производится постраничный вывод информации.

top – показывает список работающих в данный момент процессов и информацию о них, включая использование ими памяти и процессора. Список интерактивно формируется в реальном времени. Чтобы выйти из программы top, нажмите клавишу [q].

ps [Опции] [number] - команда для вывода информации о процессах:

Опции

1. -a все терминальные процессы

2. -e все процессы.

3. -gсписок выбирать процессы по списку лидеров групп.

4. -pсписок выбирать процессы по списку идентификаторов процессов.

5. -tсписок выбирать процессы по списку терминалов.

6. -uсписок выбирать процессы по списку идентификаторов пользователей.

7. f генерировать полный листинг.

8. -l генерировать листинг в длинном формате.

9. number - номер процесса.

Команда ps без параметров выводит информацию только об активных процессах, запущенных с данного терминала, в том числе и фоновых. На экран выводится подробная информация обо всех активных процессах в следующей форме:

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
1	S	200	210	7	0	2	20	80	30	703a	03	0:07	cc
1	R	12	419	7	11	5	20	56	20	03	0:12	ps	

1. F - флаг процесса (1 - в оперативной памяти, 2 - системный процесс, 4 - заблокирован в ОЗУ, 20 - находится под управлением другого процесса, 10 - подвергнут свопингу);

2. S - состояние процесса (O - выполняется процессором , S - задержан, R - готов к выполнению, I - создается);

3. UID - идентификатор пользователя;

4. PID - идентификатор процесса;

5. PPID - номер родительского процесса;

6. C - степень загруженности процессора;

7. PRI - приоритет процесса, вычисляется по значению переменной

8. NICE и чем больше число, тем меньше его приоритет;

9. NI - значение переменной NICE для вычисления динамического

10. приоритета, принимает величины от 0 до 39;

11. ADDR - адрес процесса в памяти;

12. SZ - объем ОЗУ, занимаемый процессом;

13. WCHAN - имя события, до которого процесс задержан, для ак-

14. тивного процесса - пробел;

15. TTY - номер управляющего терминала для процесса;

16. TIME - время выполнения процесса;

17. CMD - команда, которая породила процесс.

nice [-приращение приоритета] команда[аргументы] - команда изменения приоритета. Каждое запущенное задание (процесс) имеет номер приоритета в диапазоне от 0 до 39, на основе которого ядро вычисляет фактический приоритет, используемый для планирования процесса. Значение 0 представляет наивысший приоритет, а 39 - самый низший. Увеличение номера приоритета приводит к понижению приоритета, присвоенного процессу. Команда

nice -10 ls -l

увеличивает номер приоритета, присвоенный процессу ls -l на 10.

renice 5 1836 - команда устанавливает значение номера приоритета процесса с идентификатором 1836 равным 5. Увеличить приоритет процесса может только администратор системы.

kill [-sig] <идентификатор процесса> - прекращение процесса до его программного завершения. sig - номер сигнала. Sig = -15 означает программное (нормальное) завершение процесса, номер сигнала = -9 - уничтожение процесса. По умолчанию sig= -9. Вывести себя из системы можно командой kill -9 0. Пользователь с низким приоритетом может прервать процессы, связанные только с его терминалом.

free – Показывает общее количество свободной и используемой физической памяти и памяти отведенной для свопирования в системе, так же и совместно используемую память и буфера используемые ядром. Синтаксис :

free [-b | -k | -m] [-o] [-s delay] [-t] [-V]

Опции :

1. -b показывает количество памяти в байтах; опция -k (по умолчанию) показывает количество памяти в килобайтах;

2. Опция -m показывает количество памяти в мегабайтах.

3. -t показывает строки содержащие полное количество памяти.

4. -o запрещает показывать строки относящиеся к "массиву буфера" . Если не определено отнять/добавить память буферов из/в используемую/свободную память (соответственно!).

5. -s разрешает безостановочно выводить информацию с промежутком в delay секунд.

6. -V показывает информацию о версии программы.

tty – информация о терминалах пользователя.

2.3 Команды работы с файловой системой

Для управления файловой системой имеются различные команды, реализующие операции по созданию, чтению, копированию, переименованию/перемещению, изменению и удалению файлов и каталогов. Как правило, это специализированные команды, хорошо выполняющие свою задачу, однако некоторые функции могут частично дублироваться другими командами, что только добавляет гибкости управлению файлами.

Основными командами для выполнения файловых операций являются: pwd, ls, cp, mv, dir, rm, cd, rmdir, mvdir, mkdir, ln. Информацию о их назначении и параметрах доступна в справке.

pwd – Выдача имени текущего каталога

cd – Смена текущего каталога Синтаксис команды:

cd [каталог]

Команда cd применяется для того, чтобы сделать заданный каталог текущим. Если каталог не указан, используется значение переменной окружения $HOME (обычно это каталог, в который Вы попадаете сразу после входа в систему). Если каталог задан полным маршрутным именем, он становится текущим. По отношению к новому каталогу нужно иметь право на выполнение, которое в данном случае трактуется как разрешение на поиск. Текущий каталог - это каталог, в котором в данный момент находится пользователь. При наличии прав доступа, пользователь может перейти после входа в систему в другой каталог. Текущий каталог обозначается точкой (.); родительский каталог, которому принадлежит текущий, обозначается двумя точками (..).

cat <имя файла> - вывод содержимого файла на экран.

Команда cat > text.1 – создает новый файл с именем text.1, который можно заполнить символьными строками, вводя их с клавиатуры. Нажатие клавиши Enter создает новую строку. Завершение ввода – нажатие Ctrl - d.

Команда cat text.1 > text.2 пересылает содержимое файла text.1 в файл text.2.

Слияние файлов осуществляется командой cat text.1 text.2 > text.3.

ls [-опции] [имя] - вывод содержимого каталога на экран. Если аргумент не указан, выдается содержимое текущего каталога. Аргументы команды:

1. l –- список включает всю информацию о файлах;

2. t – сортировка по времени модификации файлов;

3. a – в список включаются все файлы, в том числе и те, которые начинаются с точки;

4. s – размеры файлов указываются в блоках;

5. d – вывести имя самого каталога, но не содержимое;

6. r – сортировка строк вывода;

7. i – указать идентификационный номер каждого файла;

8. v – сортировка файлов по времени последнего доступа;

9. q – непечатаемые символы заменить на знак ?;

10. c – использовать время создания файла при сортировке;

11. g – то же что -l, но с указанием имени группы пользователей;

12. f – вывод содержимого всех указанных каталогов, отменяет флаги -l, -t, -s, -r и активизирует флаг -a;

13. C – вывод элементов каталога в несколько столбцов;

14. F – добавление к имени каталога символа / и символа * к имени файла, для которых разрешено выполнение;

15. R – рекурсивный вывод содержимого подкаталогов заданного каталога.

ls -l file.1 - чтение атрибутов файла;

mkdir – Создание каталога. Синтаксис:

mkdir [-m режим_доступа] [-p] каталог ...

По команде mkdir создается дин или несколько каталогов с режимом доступа 0777 [возможно измененном с учетом umask и опции -m]. Стандартные файлы (. - для самого каталога и .. - для вышележащего) создаются автоматически; их нельзя создать по имени. Для создания каталога необходимо располагать правом записи в вышележащий каталог. Идентификаторы владельца и группы новых каталогов устанавливаются соответственно равными реальным идентификаторам владельца и группы процесса. Командой mkdir обрабатываются две опции:

? -m режим_доступа - (явное задание режима_доступа для создаваемых каталогов [см. chmod]).

? -p (при указании этой опции перед созданием нового каталога предварительно создаются все несуществующие вышележащие каталоги).

cp – Копирование файлов. Синтаксис :

cp файл1 [файл2 ...] целевой_файл

Команда cp копирует файл1 в целевой_файл. Файл1 не должен совпадать с целевым_файлом (будьте внимательны при использовании метасимволов shell'a). Если целевой_файл является каталогом, то файл1, файл2, ..., копируются в него под своими именами. Только в этом случае можно указывать несколько исходных файлов.

Если целевой_файл существует и не является каталогом, его старое содержимое теряется. Режим, владелец и группа целевого_файла при этом не меняются.

Если целевой_файл не существует или является каталогом, новые файлы создаются с теми же режимами, что и исходные (кроме бита навязчивости, если Вы не суперпользователь). Время последней модификации целевого_файла (и последнего доступа, если он не существовал), а также время последнего доступа к исходным файлам устанавливается равным времени, когда выполняется копирование.

Если целевой_файл был ссылкой на другой файл, все ссылки сохраняются, а содержимое файла изменяется.

Пример:

cp file.1 file.2 - копирование файла с переименованием;

mv – Перемещение (переименование) файлов. Синтаксис команды:

mv [-f] файл1 [файл2 ...] целевой_файл

Команда mv перемещает (переименовывает) файл1 в целевой_файл. Файл1 не должен совпадать с целевым_файлом (будьте внимательны при использовании метасимволов shell'a).

Если целевой_файл является каталогом, то файл1, файл2, ..., перемещаются в него под своими именами. Только в этом случае можно указывать несколько исходных файлов.

Если целевой_файл существует и не является каталогом, его старое содержимое теряется. Если при этом обнаруживается, что в целевой_файл не разрешена запись, то выводится режим этого файла [см. chmod] и запрашивается строка со стандартного ввода. Если эта строка начинается с символа y, то требуемые действия все же выполняются, при условии, что у пользователя достаточно прав для удаления целевого_файла. Если была указана опция –f или стандартный ввод назначен не на терминал, то требуемые действия выполняются без всяких запросов. Вместе с содержимым целевой_файл наследует режим файла1.

Если файл1 является каталогом, то он переименовывается в целевой_файл, только если у этих двух каталогов общий надкаталог; при этом все файлы, находившиеся в файле1, перемещаются под своими именами в целевой_файл. Если файл1 является файлом, а целевой_файл - ссылкой, причем не единственной, на другой файл, то все остальные ссылки сохраняются, а целевой_файл становится новым независимым файлом.

Пример:

mv file.1 file.2 - переименование файла file.1 в file.2;

mv file.1 file.2 file.3 directory - перемещение файлов file.1, file.2, file.3 в указанную директорию;

rm – удаление файлов. Синтаксис команды:

rm [-f] [-i] файл ...

rm -r [-f] [-i] каталог ... [файл ...]

Команда rm служит для удаления указанных имен файлов из каталога. Если заданное имя было последней ссылкой на файл, то файл уничтожается. Для удаления пользователь должен обладать правом записи в каталог; иметь право на чтение или запись файла не обязательно. Следует заметить, что при удалении файла в Linux, он удаляется навсегда. Здесь нет возможностей вроде "мусорной корзины" в windows 95/98/NT или команды undelete в DOS. Так что, если файл удален, то он удален!

Если нет права на запись в файл и стандартный ввод назначен на терминал, то выдается (в восьмеричном виде) режим доступа к файлу и запрашивается подтверждение; если оно начинается с буквы y, то файл удаляется, иначе - нет. Если стандартный ввод назначен не на терминал, команда rm ведет себя так же, как при наличии опции -f. Допускаются следующие три опции:

1. -f Команда не выдает сообщений, когда удаляемый файл не существует, не запрашивает подтверждения при удалении файлов, на запись в которые нет прав. Если нет права и на запись в каталог, файлы не удаляются. Сообщение об ошибке выдается лишь при попытке удалить каталог, на запись в который нет прав (см. опцию -r).

2. -r Происходит рекурсивное удаление всех каталогов и подкаталогов, перечисленных в списке аргументов. Сначала каталоги опустошаются, затем удаляются. Подтверждение при удалении файлов, на запись в которые нет прав, не запрашивается, если задана опция -f или стандартный ввод не назначен на терминал и не задана опция -i. При удалении непустых каталогов команда rm -r предпочтительнее команды rmdir, так как последняя способна удалить только пустой каталог. Но команда rm -r может доставить немало острых впечатлений при ошибочном указании каталога!

3. -i Перед удалением каждого файла запрашивается подтверждение. Опция -i устраняет действие опции -f; она действует даже тогда, когда стандартный ввод не назначен на терминал.

ПРИМЕРЫ

rm file.1 file.2 file.3 - удаление файлов file.1, file.2, file.3;.

Опция -i часто используется совместно с -r. По команде:

rm -ir dirname

запрашивается подтверждение: directory dirname: ?

При положительном ответе запрашиваются подтверждения на удаление всех содержащихся в каталоге файлов (для подкаталогов выполняются те же действия), а затем подтверждение на удаление самого каталога.

rmdir – Удаление каталогов. Синтаксис команды:

rmdir [-p] [-s] каталог ...

Команда rmdir удаляет указанные каталоги, которые должны быть пустыми. Для удаления каталога вместе с содержимым следует воспользоваться командой rm с опцией -r. Текущий каталог [см. pwd] не должен принадлежать поддереву иерархии файлов с корнем – удаляемым каталогом. Для удаления каталогов нужно иметь те же права доступа, что и в случае удаления обычных файлов [см. rm]. Командой rmdir обрабатываются следующие опции:

1. -p Позволяет удалить каталог и вышележащие каталоги, оказавшиеся пустыми. На стандартный вывод выдается сообщение об удалении всех указанных в маршруте каталогов или о сохранении части из них по каким-либо причинам.

2. -s Подавление сообщения, выдаваемого при действии опции -p.

grep – поиск файлов с указанием или без указания контекста (шаблона поиска). Синтаксис:

grep [-vcilns] [шаблон поиска] <имя файла>

Значение ключей:

1. - v – выводятся строки, не содержащие шаблон поиска;

2. - c – выводится только число строк, содержащих или не содер-

3. жащих шаблон;

4. - i – при поиске не различаются прописные и строчные буквы;

5. - l – выводятся только имена файлов, содержащие указанный

6. шаблон;

7. - n – перенумеровать выводимые строки;

8. - s – формируется только код завершения.

ln (link) – создание ссылок. Один файл можно сделать принадлежащим нескольким каталогам. Для этого используется команда:

ln <имя файла 1> <имя файла 2>

Имя 1-го файла - это полное составное имя файла, с которым устанавливается связь; имя 2-го файла - это полное имя файла в новом каталоге, где будет использоваться эта связь.

Новое имя может не отличаться от старого. Каждый файл может иметь несколько связей, т.е. он может использоваться в разных каталогах под разными именами.

Команда ln с аргументом -s создает символическую связь:

ln -s <имя файла 1> <имя файла 2>

Здесь имя 2-го файла является именем символической связи. Символическая связь является особым видом файла, в котором хранится имя файла, на который символическая связь ссылается. LINUX работает с символической связью не так, как с обычным файлом - например, при выводе на экран содержимого символической связи появятся данные файла, на который эта символическая связь ссылается.

2.4 Режимы доступа к файлам

В LINUX различаются 3 уровня доступа к файлам и каталогам:

1) доступ владельца файла;

2) доступ группы пользователей, к которой принадлежит владелец файла;

3) остальные пользователи.

Для каждого уровня существуют свои байты атрибутов, значение которых расшифровывается следующим образом:

r – разрешение на чтение;

w – разрешение на запись;

x – разрешение на выполнение;

- – отсутствие разрешения.

Первый символ байта атрибутов определяет тип файла и может интерпретироваться со следующими значениями:

- – обычный файл;

d – каталог;

l – символическая связь;

в – блок-ориентированный специальный файл, который соответствует таким периферийным устройствам, как накопители на магнитных дисках;

с – байт-ориентированный специальный файл, который может соответствовать таким периферийным устройствам как принтер, терминал.

В домашнем каталоге пользователь имеет полный доступ к файлам (READ, WRITE, EXECUTE; r, w, x). Атрибуты файла можно просмотреть командой ls -l и они представляются в следующем формате:

d rwx rwx rwx

| | |Доступ для остальных пользователей

| | |Доступ к файлу для членов группы

| |Доступ к файлу владельца

|Тип файла (директория)

Пример. Командой ls -l получим листинг содержимого текущей директории student:

- rwx --- --- 2 student 100 Mar 10 10:30 file_1

- rwx --- r-- 1 adm 200 May 20 11:15 file_2

- rwx --- r-- 1 student 100 May 20 12:50 file_3

После байтов атрибутов на экран выводится следующая информация о файле:

- число связей файла;

- имя владельца файла;

- размер файла в байтах;

- дата создания файла (или модификации);

- время;

- имя файла.

Атрибуты файла и доступ к нему, можно изменить командой:

chmod <коды защиты> <имя файла>

Коды защиты могут быть заданы в числовом или символьном виде. Для символьного кода используются:

1. знак плюс (+) - добавить права доступа;

2. знак минус (-) - отменить права доступа;

3. r,w,x - доступ на чтение, запись, выполнение;

4. u,g,o - владельца, группы, остальных.

Коды защиты в числовом виде могут быть заданы в восьмеричной форме. Для контроля установленного доступа к своему файлу после каждого изменения кода защиты нужно проверять свои действия с помощью команды ls -l.

Примеры:

chmod g+rw,o+r file.1 - установка атрибутов чтения и записи для группы и чтения для всех остальных пользователей;

ls -l file.1 - чтение атрибутов файла;

chmod o-w file.1 - отмена атрибута записи у остальных пользователей;

2.5 Создание командных файлов

Командный файл в Unix представляет собой обычный текстовый файл, содержащий набор команд Unix и команд Shell. Для того чтобы командный интерпретатор воспринимал этот текстовый файл, как командный необходимо установить атрибут на исполнение.

Установку атрибута на исполнение можно осуществить командой chmod или через mc по клавише F9 выйти в меню и выбрать вкладку File, далее выбрать изменение атрибутов файла.

Например.

$ echo " ps -af " > commandfile

$ chmod +x commandfile

$ ./commandfile
$ echo " ps -af " > commandfile
$ chmod +x commandfile
$ ./commandfile

В представленном примере команда echo " ps -af " > commandfile создаст файл с одной строкой " ps -af ", команда chmod +x commandfile установит атрибут на исполнение для этого файла, команда ./commandfile осуществит запуск этого файла.

Лекция. Shell – ПРОГРАММИРОВАНИЕ

1. ОПЕРАТОРЫ – КОМАНДЫ

Язык shell по своим возможностям приближается к высокоуровневым алгоритмическим языкам программирования. Операторы языка shell позволяют создавать собственные программы, не требует компиляции, построения объектного файла и последующей компоновки, так как shell, обрабатывающий их, является транслятором интерпретирующего, а не компилирующего типа.

Текст процедуры набирается как обычный текстовый файл. Проверенный и отлаженный shell-файл может быть вызван на исполнение, например, следующим способом:

$ chmod u+x shfil
$ shfil
$

Такая форма предполагает, что файл процедуры новый и его надо сначала сделать выполняемым. Можно использовать также и следующий способ:

$ sh -c "shfil" или $ sh shfil

В этих случаях по команде sh вызывается вторичный интерпретатор shell, и в качестве аргумента ему передается командная строка, содержащая имя файла процедуры shfil, находящегося в текущем каталоге. Однако, этот способ накладывает ограничения на исполнение некоторых команд ОС управления процессами.

Процедуре при ее запуске могут быть переданы аргументы. В общем случае командная строка вызова процедуры имеет следующий вид:

$ имя_процедуры $1 $2 ...$9

Каждому из девяти первых аргументов командной строки в тексте процедуры соответствует один из позиционных параметров: $1, $2, ..., $9 соответственно. Параметр $0 соответствует имени самой процедуры, т.е. первому полю командной строки. К каждому из 10 первых аргументов можно обратиться из процедуры, указав номер его позиции. Количество аргументов присваивается другой переменной: $#(диез). Наконец, имя процедуры - это $0; переменная $0 не учитывается при подсчете $#.

Сам интерпретатор shell автоматически присваивает значения следующим переменным (параметрам):

? значение, возвращенное последней командой;
$ номер процесса;
! номер фонового процесса;
число позиционных параметров, передаваемых в shell;
* перечень параметров, как одна строка;
@ перечень параметров, как совокупность слов;
флаги, передаваемые в shell.

При обращении к этим переменным (т.е при использовании их в командном файле - shell-программе) следует впереди ставить "$".

Пример.

Вызов фала – specific par1 par2 par3 имеющего вид

echo $0 - имя расчета
echo $? - код завершения
echo $$ - идентификатор последнего процесса
echo $! - идентификатор последнего фонового процесса
echo
echo $ - значения параметров, как строки*
echo $@ - значения параметров, как слов
echo
set -au
echo $- - режимы работы интерпретатора

Выдаст на экран

specific - имя расчета
0 - код завершения
499 - идентификатор последнего процесса
98 - идентификатор последнего фонового процесса
par1 par2 par3 - значения параметров, как строки
par1 par2 par3 - значения параметров, как слов
au - режимы работы интерпретатора

Некоторые вспомогательные операторы:
echo - вывод сообщений из текста процедуры на экран.
$ echo "начало строки
> продолжение строки"
- для обозначения строки комментария в процедуре. (Строка не будет обрабатываться shell-ом).
banner - вывод сообщения на экран заглавными буквами (например для идентификации следующих за ним сообщений).
$banner 'hello ira'
HELLO IRA
$

Простейший пример.
Здесь оператор echo выполняется в командном режиме.
$shfil p1 pp2 petr
$echo $3
petr
$

Пример
Передача большого числа параметров
echo "$0: Много параметров"
echo " Общее число параметров = $#
Исходное состояние: $1 $5 $9 "
shift
echo "1 сдвиг: первый=$1 пятый=$5 девятый=$9"
shift 2
echo "1 + 2 = 3 сдвига: первый=$1 пятый=$5 девятый=$9"
perem=`expr $1 + $2 + $3`
echo $perem

Значения параметрам, передаваемым процедуре, можно присваивать и в процессе работы процедуры с помощью оператора set - присвоить значения позиционным параметрам;

$set a1 ab2. abc
$echo $1 $2
a1 ab2 - в этом примере параметры указываются в явном виде.
$

Запуск set без параметров выводит список установленных системных переменных:
HOME=/home/sae
PATH=/usr/local/bin:/usr/bin:/bin:.:/usr/bin/X11:
IFS=
LOGNAME=sae
MAIL=/var/spool/mail/sae
PWD=/home/sae/STUDY/SHELL
PS1=${PWD}:" "
PS2=>
SHELL=/bin/bash
TERM=linux
TERMCAP=console|con80x25|dumb|linux:li#25:co#80::

UID=501

Количество позиционных параметров может быть увеличено до необходимого значения путем "сдвига" их в командной строке влево на одну позицию с помощью команды shift без аргументов:

shift - сдвинуть позиционные параметры влево на одну позицию

После выполнения shift прежнее значение параметра $1 теряется, значение $1 приобретает значение $2, значение $2 - значение $3 и т.д..Продолжение предыдущего примера:

```
$shift
$echo $1 $2
ab2 abc
$
```

В UNIX при написании операторов важное значение отводится кавычкам (апострофам):

'...' - для блокирования специальных символов, которые могут быть интерпретированы как управляющие;

"..." - блокирование наиболее полного набора управляющих символов или указания того, что здесь будет обрабатываться не сам аргумент, а его значение;

`...` - (обратные кавычки или знак ударения) для указания того, что они обрамляют команду и здесь будет обрабатываться результат работы этой команды (подстановка результатов работы указанной команды).

Пример 1.

```
$ date
Apr 3 14:27:07 2005
$ set `date`
$ echo $3
14:30:25
$
```

Пример 2.

```
$echo `ls`

fil.1

fil.2

 ...

$echo "`ls`"

# одинарные кавычки блокируют действие обратных кавычек
```

т.е. они распечатываются как обычные символы

`ls`

$

Для ввода строки текста со стандартного устройства ввода используется оператор:

read имя1 [имя2 имя3 .] - чтение строки слов со стандартного ввода

Команда вводит строку, состоящую из нескольких полей (слов), со стандартного ввода, заводит переменную для каждого поля и присваивает первой переменной имя1, второй переменной - имя2, и т.д. Если имен больше, чем полей в строке, то оставшиеся переменные будут инициализированы пустым значением. Если полей больше, чем имен переменных, то последней переменной будет присвоена подстрока введенной строки, содержащая все оставшиеся поля, включая разделители между ними. В частности, если имя указано только одно, то соответствующей ему переменной присваивается значение всей строки целиком.

Пример:

#Текст процедуры:

echo "Введите значения текущих: гг мм ччвв"

read 1v 2v 3v

echo "год 1v"

echo "месяц 2v"

echo "сегодня 3v"

здесь кавычки используются для блокирования пробелов

#Результат выполнения процедуры:

Введите значения текущих: гг мм ччвв

2005 Март 21 9:30 <Enter>

год 2005

месяц Март

сегодня 21 9:30

2 УПРАВЛЕНИЕ ЛОКАЛЬНЫМИ ПЕРЕМЕННЫМИ

В отличии от рассмотренных в начале курса системных переменных среды, переменные языка shell называются локальными переменными и используются в теле процедур для

решения обычных задач. Локальные переменные связаны только с породившим их процессом. Локальные переменные могут иметь имя, состоящее из одного или нескольких символов. Присваивание значений переменным осуществляется с помощью известного оператора:

"=" - присвоить (установить) значение переменной.

При этом если переменная существовала, то новое значение замещает старое. Если переменная не существовала, то она строится автоматически shell. Переменные хранятся в области ОП - области локальных данных.

$count=3

$color=red belt

$fildir=lev/d1/d12

$

Еще пример:

текст процедуры

b=”1 + 2”

echo c=$b

#в результате выполнения процедуры выводится текст,

включающий текст переменной b

c=1+2

3. ПОДСТАНОВКА ЗНАЧЕНИЙ ПЕРЕМЕННЫХ

Процедура подстановки выполняется shell-ом каждый раз когда надо обработать значение переменной если используется следующая конструкция: первый вид подстановки
$имя_переменной -. на место этой конструкции будет подставлено значение переменной.

Примеры подстановки длинных маршрутных имен:
$echo $HOME
/home/lev
$filname=$HOME/f1
$more $filname
<текст файла f1 из каталога lev>
$

Использование полного маршрутного имени в качестве значения переменной позволяет пользователю независимо от местонахождения в файловой системе получать доступ к требуемому файлу или каталогу.

Если в строке несколько присваиваний, то последовательность их выполнения - справа налево.

Пример 1.

```
$ z=$y y=123
$ echo $z $y
123 123
$ y=abc z=$y
$ echo “z”
123
$ echo “y”
abc
$
```

Пример 2.

```
$ var=/user/lab/ivanov
$ cd $var
$ pwd
/udd/lab/ivanov
$
```

Пример 3.

```
$ namdir='ls'
$ $namdir
fil1
fil2
fil3
...
$
```

В последнем примере переменной namdir присвоено значение, которое затем используется в качестве командной строки запускаемой команды. Это команда ls.

Второй вид подстановки – подстановка результатов работы команды вместо самой команды.

Пример 4.

В данном случае команда ls непосредственно выполняется уже в первой строке, и переменной filnam присваивается результат ее работы.

```
$ filnam=`ls`
$ echo $filnam
fil1
fil2
fil3
...
$
```

Пример 5.

```
$ A=1 B=2
```

```
$ dat="$A + $B"
$ echo $dat
1 + 2
$
```

С переменными можно выполнять арифметические действия как и с обычными числами с использованием специального оператора:

expr - вычисление выражений.

Для арифметических операций, выполнимых командой expr, используются операторы:
+ сложение; - вычитание; * умножение (обратная прямая скобка \ используется для отмены действия управляющих символов, здесь *); / деление нацело; % остаток от деления.

Для логических операций арифметического сравнения чисел командой expr используются следующие обозначения: = равно; != не равно; \< меньше; \<= меньше или равно; \> больше; \>= больше или равно.

Все операнды и операторы являются самостоятельными аргументами команды expr и поэтому должны отделяться друг от друга и от имени команды expr пробелами.

Пример 6.

```
# Текст процедуры:
a=2
a=`expr $a + 7`
b=`expr $a / 3`
c=`expr $a - 1 + $b`
d=`expr $c % 5`
e=`expr $d - $b`
echo $a $b $c $d $e

#Результат работы процедуры:
9 3 11 1 -2
```

Команда expr выводит результат вычисления на экран. Поэтому, если он не присвоен никакой переменной, то не может быть использован в программе.

При решении логических задач, связанных с обработкой символьных строк (текстов) команда expr может быть использована, например, как средство для подсчета символов в строках или для вычленения из строки цепочки символов. Операция обработки строк символов задается кодом операции ":" и шаблонами. В частности:

'.*' - шаблон для подсчета числа символов в строке,
'...\(.*\)....' - шаблон для выделения подстроки удалением символов строки, соответствующих точкам в шаблоне .

Пример 7.

```
$ m=aaaaaa
$ expr $m : '.*'
6
$
```

Пример 8.

```
$ n=abcdefgh
```

```
$ expr $n : '...\(.*\)..'
def
$
```

Выводимая информация - количество символов или подстрока - может быть присвоена некоторой переменной и использована в дальнейших вычислениях.

Третий вид подстановки - применяется для подстановки команд или целых shell-процедур. Используется для замены кода команды или текста процедуры на результат их выполнения в той же командной строке:

$(командная_строка) - подстановка осуществляется также перед запуском на исполнение командной строки.

В данном случае в качестве подставляемой команды может быть использована также любая имеющая смысл sh-процедура. Shell просматривает командную строку и выполняет все команды между открывающей и закрывающей скобками. Рассмотрим примеры присвоения значений и подстановки значений локальных переменных.

Пример 9.

```
$ A='string n'
$ count=$(expr $A : '.*')
$ echo $count
8
$
#Продолжение примера:
$ B=$(expr $A : '..\(.*\))
$ echo $B
ring
$
```

Рассмотрим пример на разработку простейшей линейной процедуры обработки переменных средствами языка shell.

ЗАДАНИЕ:

Создать файл, содержащий процедуру сложения двух чисел. Числа передаются в виде параметров при обращении к процедуре. Выполнить процедуру.

```
$ cat>comf
SUM=$(expr $1 + $2)
echo "$1 + $2 = $SUM"
<Ctrl*D>
$ sh comf 3 5
3 + 5 = 8
$
```

На экране можно просмотреть все заведенные локальные переменные с помощью известной команды:

```
$ set
$
```

Удаление переменных:

```
$unset перем1 [перем2 .....]
$
```

4. ЭКСПОРТИРОВАНИЕ ЛОКАЛЬНЫХ ПЕРЕМЕННЫХ В СРЕДУ shell

При выполнении процедуры ей можно передавать как позиционные параметры так и ключевые - локальные переменные порожденного процесса. Локальные переменные помещаются в область локальных переменных, связанную с конкретным текущим процессом, породившим переменную. Они доступны только этому процессу и недоступны порожденным процессам-потомкам (например – sh-процедурам). Переменные другим процессам можно передавать неявно через среду. Для этого локальная переменная должна быть экспортирована (включена) в среду, в которой исполняется процедура, использующая эту переменную. Среда пользователя, т.е. глобальные переменные, доступна всем процессам.

Три формата команды экспортирования:
$export список имен локальных переменных
$export имя_лок_переменной=значение
$export (без параметров) - выводит перечень всех экспортированных локальных и переменных среды (аналог команды env).

Рассмотрим некоторый фрагмент протокола работы с системой.
$color = red переменная определена, но не экспортирована
$export count = 1 переменная определена и экспортирована, т.е. потенциально доступна всем порождаемым процессам
$export
PATH = …..
HOME = …..
color = red
count = 1
$cat proc1 создание порожденного процесса процедуры
echo $color
echo $count
exit завершение процесса
<ctrl.D>
$proc1 выполнение процедуры на экран выводится значение только одной, экспортированной переменной; вторая переменная – не определена
1
$cat proc2 еще одна процедура выводятся значения обеих переменных, т.к. они определены в самой процедуре
color = black
count = 2
echo $color
echo $count
exit
$proc2
black
1
.
$echo $color
red
$echo $count
1
$

На экран выводятся первоначальные значения переменных родительского процесса – shell. Новые (измененные) значения локальных переменных существуют только на

время существования породившего их порожденного процесса. Чтобы изменить значене переменной родительского процесса ее надо экспортировать. Но после завершения порожденного среда родительского восстанавливается.

5. ПРОВЕРКА УСЛОВИЙ И ВЕТВЛЕНИЕ ВЫЧИСЛИТЕЛЬНЫХ ПРОЦЕССОВ

Все команды UNIX вырабатывают код завершения (возврата), обычно для того, чтобы в дальнейшем можно было выполнить диагностику посредством проверки значения кода завершения и определить: нормально завершилось выполнение команды (=0 или true) или не нормально (# 0 или false). Например, если (=1), то ошибка синтаксическая.

Код завершения после выполнения каждой команды помещается автоматически в некоторую специальную.

Системную переменную и ее значение можно вывести на экран: echo $?

Пример:

$true
$echo $?
0
$ls
$echo $?
0
$false
$echo $?
1
$cp
<сообщение о некорректности заданной команды – нет параметров>
$echo $?
1
$echo $?
0
$

Код завершения используется для программирования условных переходов в sh-процедурах. Проверка истинности условий для последующего ветвления вычислительного процесса процедур может быть выполнена с помощью команды:

test <проверяемое отношение/условие>

Вместо мнемоники команды может использоваться конструкция с квадратными скобками:
[проверяемое отношение/условие] синоним команды test.

Аргументами этой команды могут быть имена файлов, числовые или нечисловые строки (цепочки символов). Командой вырабатывается код завершения (код возврата), соответствующий закодированному в команде test условию. Код завершения проверяется следующей командой. Если закодированное параметрами условие выполняется, то вырабатывается логический результат (значение некоторой системной переменной) - true, если нет - false.

Код возврата может обрабатываться как следующей за test командой, так и специальной конструкцией языка: if-then-else-fi.

1. Проверка файлов:
test -ключ имя_файла
Ключи:
-r файл существует и доступен для чтения;

-w файл существует и доступен для записи;
-x файл существует и доступен для исполнения;
-f файл существует и имеет тип "-", т.е. обычный файл;
-s файл существует, имеет тип "-" и не пуст;
-d файл существует и имеет тип "d", т.е. файл - каталог.

2. Сравнение числовых значений:

test число1 –к число2

Числа могут быть как просто числовыми строками, так и переменными, которым эти строки присвоены в качестве значений. Ключи для анализа числовых значений:
-eq равно; -ne не равно;
-lt меньше; -le меньше или равно;
-gt больше; -ge больше или равно.
Пример:

```
$x=5
$[$x -lt 7]
$echo $?
0
$[$x -gt 7]
$echo $?
1
$
```

3. Сравнение строк:

test [-n] 'строка' - строка не пуста (n – число проверяемых строк)
test -z 'строка' - строка пуста
test 'строка1' = 'строка2' - строки равны
test 'строка1' != 'строка2' - строки не равны

Необходимо заметить, что все аргументы команды test - строки, имена, числа, ключи и знаки операций являются самостоятельными аргументами и должны разделяться пробелами.

Пример.

```
$x = abc
$["$x" = "abc"]
$echo $?
0
$["$x" != "abc"]
$echo $?
1
$
```

ЗАМЕЧАНИЕ: выражение вида "$переменная" лучше заключать в двойные кавычки, что предотвращает в некоторых ситуациях возможную неподходящую замену переменных shell-ом.

Особенности сравнения чисел и строк. Shell трактует все аргументы как числа в случае, если осуществляется сравнение чисел, и все аргументы как строки, если осуществляется сравнение строк. Пример:

```
$X = 03
```

```
$Y =3
$[$X -eq $Y] - сравниваются значения чисел
$echo $?
0
$["$X" = "$Y"] - числа сравниваются как строки символов
$echo $?
1
$
```

Ветвление вычислительного процесса в shell-процедурах осуществляется семантической конструкцией:

```
if список_команд1
then список_команд2
[else список_команд3
fi
```

Список_команд - это или одна команда или несколько команд, или фрагмент shell-процедуры. Если команды записаны на одной строке, то они разделяются точкой с запятой. Для задания пустого списка команд следует использовать специальный оператор: : (двоеточие) -пустой оператор.

Список_команд1 передает оператору if код завершения последней выполненной в нем команды. Если он равен 0, то выполняется список_команд2.Таким образом, код возврата 0 эквивалентен логическому значению "истина". В противном случае он эквивалентен логическому значению "ложь" и выполняется либо список_команд3 после конструкции else, либо - завершение конструкции if словом fi.

В качестве списка_команд1 могут использоваться списки любых команд. Однако, чаще других используется команда test. В операторе if так же допускается две формы записи этой команды:

```
if test аргументы
if [ аргументы ]
```

Каждый оператор if произвольного уровня вложенности обязательно должен завершаться словом fi.

ЗАДАНИЕ:

Создать и выполнить файл с процедурой, сравнивающей передаваемый ей параметр с некоторым набором символов (паролем).

```
$ cat>com
if test 'param' = "$1" - сравниваются строки символов
then echo Y
else echo N
fi
<Ctrl*D>
$ chmod u+x com
$ com param
Y
$ com parm
N
$
```

ЗАДАНИЕ.

Организовать ветвление вычислительного процесса в процедуре в зависимости от значения переменной X (<10, >10, = 10).

```
if
 [$X -lt 10]
then
 echo X is less 10
else
 if
 [$X -gt 10]
 then
 echo X is greatr 10
 else
 echo X is equal to 10
 fi
fi
```

Для улучшения восприятия программ и облегчения отладки целесообразно придерживаться структурированного стиля написания программы. Каждому if должен соответствовать свой fi.

6. ПОСТРОЕНИЕ ЦИКЛОВ

Циклы обеспечивают многократное выполнение отдельных участков процедуры до достижения заданных условий.

Цикл типа while (пока true):

```
while список_команд1
do список_команд2
done
```

Список_команд1 возвращает код возврата последней выполненной команды. Если условие истинно, выполняется список_- команд2, затем снова список_команд1 с целью проверки условия, а если ложно, выполнение цикла завершается. Таким образом, циклический список_команд2 выполняется до тех пор, пока условие истинно.

Пример 1.

Проверка наличия параметров при обращении к данной процедуре. Вывод на экран сообщений о наличии параметров и тексты параметров.

Текст процедуры, которой присвоено имя P1:

```
if $# –eq 0
 then echo “No param”
 else echo “Param:’;
 while test ‘$1’
 do
echo “$1”
shift
done
fi
```

Результат работы процедуры:

```
$P1
No param
$P1 abc df egh
Param:
abc
df
egh
$
```

Пример 2.

Ввод строки из нескольких слов. Подсчет и вывод числа символов в каждом слове. Текст процедуры, которой присвоено имя P2:

```
echo “Input string:”
read A
set $A
while [ “$1” ]
do
 echo “$1 = `expr ”$1” : ‘.*’`”
 shift
done
```

Результат работы процедуры:

```
$P2
Input string:
df ghghhhh aqw
df = 2
ghghhhh = 7
aqw = 3
$
```

Пример 3.

Вывести на экран слово строки (поле), номер которого (переменная N) указан в параметре при обращении к процедуре, которой присвоено имя P3. Процедура запрашивает ввод строки с клавиатуры. Номер слова вводится как аргумент процедуры. Текст процедуры P3:

```
i=1 - счетчик номеров слов в строке, формируется при каждом выполнении цикла
N=$1 -значение первого параметра
echo "Введи строку: "
read a
set $a
while test $i -lt $N
 do
i=`expr $i + 1` - формирование номера следующего слова
shift
 done
echo "$N поле строки: \"$1\""
Пример работы процедуры P3:
$P3 2
```

```
Введи строку: aa bb cc dd
2 поле строки: "bb"
$
```
Цикл типа until (пока false):

```
until список_команд1
do список_команд2
done
```

Логическое условие, определяемое по коду возврата списка_команд1, инвертируется, т.е. цикл выполняется до техпор, пока условие ложно.

Пример процедуры с именем P4, выполняющей заданное число циклов.

```
$cat>P4
X = 1 - счетчик числа циклов
until test $X -gt 10 - задано число циклов = 10
do
 echo X is $X
 X = `expr $X + 1`
done
<Ctrl*D>
$sh P4
 X is 1
 X is 2
.....
 X is 10
$
```

Цикл типа for:

```
for имя_ переменной [in список_значений]
do список_команд
done
```

Переменная с указанным в команде именем заводится автоматически. Переменной присваивается значение очередного слова из списка_значений и для этого значения выполняется список_команд. Количество итераций равно количеству значений в списке, разделенных пробелами (т.е. циклы выполняются пока список не будет исчерпан).

Пример текста процедуры, печатающей в столбец список имен файлов текущего каталога.

```
list =`ls`
for val in $list
do
 echo "$val"
done
echo end
```

Пример

Процедура, должна скопировать все обычные файлы из текущего каталога в каталог, который задается в качестве аргумента при обращении к данной процедуре по имени comfil. Процедура проверяет так же наличие каталога-адресата и сообщает количество скопированных файлов.

```
m=0 - переменная для счетчика скопированных файлов
if [ -d $HOME/$1 ]
 then echo "Каталог $1 существует"
```

```
 else
mkdir $HOME/$1 .
 echo "Каталог $1 создан"
fi
 for file in *
 do
 if [ -f "$file" ]
 then cp "$file" $HOME/$1;
m=`expr $m + 1`
fi
 done
echo "Число скопированных файлов: $m"
```

Выполнение процедуры:

```
$sh comfil dir1
```

Число скопированных файлов:….

```
$
```

Здесь символ * - имеет смысл <список_имен_файлов_текущего_ каталога>

Пример

Отобразить сведения о файлах текущего каталога

```
for nam in `ls`
do
 echo $ nam
done
```

Пример

Процедура PROC, выводит на экран имена файлов из текущего каталога, число символов в имени которых не превышает заданного параметром числа.

```
if [ “$1” = “” ]
then
exit
fi
for nam in *
do
 size = `expr $nam : ‘.*’`
 if [ “$size” –le “$1” ]
 then echo “Длина имени $nam $size символа”
 fi
done
```

Вывод содержимого текущего каталога для проверки работы процедуры:

```
$ ls –l
total 4
drwxrwxrwx 2 lev lev 1024 Feb 7 18:18 dir1
-rw-rw-r-- 1 lev lev 755 Feb 7 18:24 out
-rwxr-xr-x 1 lev lev 115 Feb 7 17:55 f1
-rwxr-xr-x 1 lev lev 96 Feb 7 18:00 f2

$
```

Результаты работы процедуры:

```
$ PROC 2
 Длина имени f1 2 символа
 Длина имени f2 2 символа
```

```
$PROC 3
 Длина имени out 3 символа
 Длина имени f1 2 символа
 Длина имени f2 2 символа
$
```

Пример.

Процедура с именем PR выводит на экран из указанного параметром подкаталога имена файлов с указанием их типа.

```
cd $1
for fil in *
do
 If [ -d $fil]
 then echo “$fil – catalog”
 else echo “$fil – file”
 fi
done
```

Вывод содержимого подкаталога для проверки работы процедуры:

```
$ ls –l pdir
total 4
drwxrwxrwx 2 lev lev 1024 Feb 7 18:18 dir1
-rw-rw-r-- 1 lev lev 755 Feb 7 18:24 out
-rwxr-xr-x 1 lev lev 115 Feb 7 17:55 f1
-rwxr-xr-x 1 lev lev 96 Feb 7 18:00 f2
```

Результаты работы процедуры:

```
$ PR pdir
dir1 – catalog
out – file
f1 – file
f2 – file
$
```

Ветвление по многим направлениям case. Команда case обеспечивает ветвление по многим направлениям в зависимости от значений аргументов команды. Формат:

```
case <string> in
s1) <list1>;;
s2) <list2>;;
sn) <listn>;;
*) <list>
esac
```

3десь list1, list2 ... listn - список команд. Производится сравнение шаблона string с шаблонами s1, s2 ... sk ... sn. При совпадении выполняется список команд, стоящий между текущим шаблоном sk и соответствующими знаками ;;.

Пример:

```
echo -n 'Please, write down your age'
read age
case $age in
 test $age -le 20) echo 'you are so young' ;;
 test $age -le 40) echo 'you are still young' ;;
 test $age -le 70) echo 'you are too young' ;;
 *)echo 'Please, write down once more'
esac
```

В конце текста помещена звездочка * на случай неправильного ввода числа.

Некоторые дополнительные команды, которые могут быть использованы в процедурах:

sleep t - приостанавливает выполнение процесса на t секунд

Пример.

Бесконечная (циклическая) процедура выводит каждые пять секунд сообщение в указанный файл fil.

```
while true
do
 echo “текст_сообщения”>fil
 sleep 5
done
```

Примечание: вместо файла (или экрана) может быть использовано фиктивное устройство /dev/null (например для отладки процедуры).

Примечание: в процедуре реализуется бесконечный цикл. Для ограничения числа циклов надо предусмотреть счетчик циклов (см. выше) или пркратить выполнение процесса процедуры с помощью команды управления процессами - $kill (см. ниже).

exit [n] - прекращение выполнения процедуры с кодом завершения [n] или с кодом завершения последней выполненной команды.

В качестве “n” можно использовать любое число, например, для идентификации выхода из сложной процедуры, имеющей несколько причин завершения выполнения.

Модуль 3. Многозадачность и управление процессами в ОС

Лекции 8ч.
Тема 8. Понятие многозадачности в ОС
Тема 9. Планирование процессов и потоков
Тема 10. Синхронизация процессов.
Тема 11. Тупики.

Лекция: Процессы

Рассмотрим следующий пример. Два студента запускают программу извлечения квадратного корня. Один хочет вычислить квадратный корень из 4, а второй – из 1. С точки зрения студентов, запущена одна и та же программа; с точки зрения компьютерной системы, ей приходится заниматься двумя различными вычислительными процессами, так как разные исходные данные приводят к разному набору вычислений. Следовательно, на уровне происходящего внутри вычислительной системы мы не можем использовать термин "программа" в пользовательском смысле слова.

Рассматривая системы пакетной обработки, мы ввели понятие "задание" как совокупность программы, набора команд языка управления заданиями, необходимых для ее выполнения, и входных данных. С точки зрения студентов, они, подставив разные исходные данные, сформировали два различных задания. Может быть, термин "задание" подойдет нам для описания внутреннего функционирования компьютерных систем? Чтобы выяснить это, давайте рассмотрим другой пример. Пусть оба студента пытаются извлечь корень квадратный из 1, то есть пусть они сформировали идентичные задания, но загрузили их в вычислительную систему со сдвигом по времени. В то время как одно из выполняемых заданий приступило к печати полученного значения и ждет окончания операции ввода-вывода, второе только начинает исполняться. Можно ли говорить об идентичности заданий внутри вычислительной системы в данный момент? Нет, так как состояние процесса их выполнения различно. Следовательно, и слово "задание" в пользовательском смысле не может применяться для описания происходящего в вычислительной системе.

Это происходит потому, что термины "программа" и "задание" предназначены для описания статических, неактивных объектов. Программа же в процессе исполнения является динамическим, активным объектом. По ходу ее работы компьютер обрабатывает различные команды и преобразует значения переменных. Для выполнения программы операционная система должна выделить определенное количество оперативной памяти, закрепить за ней определенные устройства ввода-вывода или файлы (откуда должны поступать входные данные и куда нужно доставить полученные результаты), то есть зарезервировать определенные ресурсы из общего числа ресурсов всей вычислительной системы. Их количество и конфигурация с течением времени могут изменяться. Для описания таких активных объектов внутри компьютерной системы вместо терминов "программа" и "задание" мы будем использовать новый термин – "процесс".

Понятие процесса характеризует некоторую совокупность набора исполняющихся команд, ассоциированных с ним ресурсов (выделенная для исполнения память или адресное пространство, стеки, используемые файлы и устройства ввода-вывода и т. д.) и текущего момента его выполнения (значения регистров, программного счетчика, состояние стека и значения переменных), находящуюся под управлением операционной системы. Не существует взаимно-однозначного соответствия между процессами и программами, обрабатываемыми вычислительными системами. Как будет показано далее,

в некоторых операционных системах для работы определенных программ может организовываться более одного процесса или один и тот же процесс может исполнять последовательно несколько различных программ. Более того, даже в случае обработки только одной программы в рамках одного процесса нельзя считать, что процесс представляет собой просто динамическое описание кода исполняемого файла, данных и выделенных для них ресурсов. Процесс находится под управлением операционной системы, поэтому в нем может выполняться часть кода ее ядра (не находящегося в исполняемом файле!), как в случаях, специально запланированных авторами программы (например, при использовании системных вызовов), так и в непредусмотренных ситуациях (например, при обработке внешних прерываний).

Состояния процесса

При использовании такой абстракции все, что выполняется в вычислительных системах (не только программы пользователей, но и, возможно, определенные части операционных систем), организовано как набор процессов. Понятно, что реально на однопроцессорной компьютерной системе в каждый момент времени может исполняться только один процесс. Для мультипрограммных вычислительных систем псевдопараллельная обработка нескольких процессов достигается с помощью переключения процессора с одного процесса на другой. Пока один процесс выполняется, остальные ждут своей очереди.

Как видим, каждый процесс может находиться как минимум в двух состояниях: процесс исполняется и процесс не исполняется. Диаграмма состояний процесса в такой модели изображена на рис. 2.1.

Рис. 2.1. Простейшая диаграмма состояний процесса

Процесс, находящийся в состоянии процесс исполняется, через некоторое время может быть завершен операционной системой или приостановлен и снова переведен в состояние процесс не исполняется. Приостановка процесса происходит по двум причинам: для его дальнейшей работы потребовалось какое-либо событие (например, завершение операции ввода-вывода) или истек временной интервал, отведенный операционной системой для работы данного процесса. После этого операционная система по определенному алгоритму выбирает для исполнения один из процессов, находящихся в состоянии процесс не исполняется, и переводит его в состояние процесс исполняется. Новый процесс, появляющийся в системе, первоначально помещается в состояние процесс не исполняется.

Это очень грубая модель, она не учитывает, в частности, то, что процесс, выбранный для исполнения, может все еще ждать события, из-за которого он был приостановлен, и реально к выполнению не готов. Для того чтобы избежать такой ситуации, разобьем состояние процесс не исполняется на два новых состояния: готовность и ожидание (см. рис. 2.2).

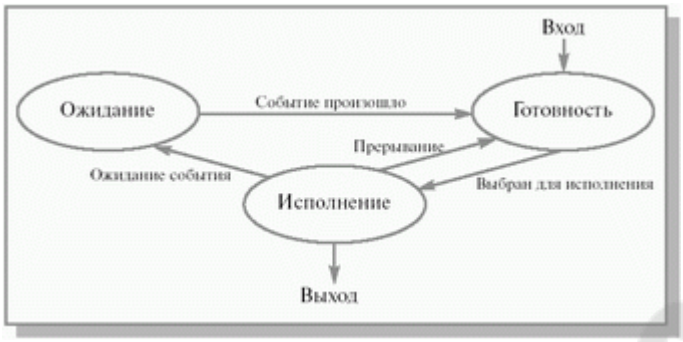

Рис. 2.2. Более подробная диаграмма состояний процесса

Всякий новый процесс, появляющийся в системе, попадает в состояние готовность. Операционная система, пользуясь каким-либо алгоритмом планирования, выбирает один из готовых процессов и переводит его в состояние исполнение. В состоянии исполнение происходит непосредственное выполнение программного кода процесса. Выйти из этого состояния процесс может по трем причинам:

операционная система прекращает его деятельность;

он не может продолжать свою работу, пока не произойдет некоторое событие, и операционная система переводит его в состояние ожидание;

в результате возникновения прерывания в вычислительной системе (например, прерывания от таймера по истечении предусмотренного времени выполнения) его возвращают в состояние готовность.

Из состояния ожидание процесс попадает в состояние готовность после того, как ожидаемое событие произошло, и он снова может быть выбран для исполнения.

Наша новая модель хорошо описывает поведение процессов во время их существования, но она не акцентирует внимания на появлении процесса в системе и его исчезновении. Для полноты картины нам необходимо ввести еще два состояния процессов: рождение и закончил исполнение (см. рис. 2.3).

Рис. 2.3. Диаграмма состояний процесса, принятая в курсе

Теперь для появления в вычислительной системе процесс должен пройти через состояние рождение. При рождении процесс получает в свое распоряжение адресное пространство, в которое загружается программный код процесса; ему выделяются стек и системные ресурсы; устанавливается начальное значение программного счетчика этого процесса и т. д. Родившийся процесс переводится в состояние готовность. При завершении своей деятельности процесс из состояния исполнение попадает в состояние закончил исполнение.

В конкретных операционных системах состояния процесса могут быть еще более детализированы, могут появиться некоторые новые варианты переходов из одного состояния в другое. Так, например, модель состояний процессов для операционной системы Windows NT содержит 7 различных состояний, а для операционной системы Unix – 9. Тем не менее так или иначе, все операционные системы подчиняются изложенной выше модели.

Операции над процессами и связанные с ними понятия

Набор операций

Процесс не может перейти из одного состояния в другое самостоятельно. Изменением состояния процессов занимается операционная система, совершая операции над ними. Количество таких операций в нашей модели пока совпадает с количеством стрелок на диаграмме состояний. Удобно объединить их в три пары:

- создание процесса – завершение процесса;
- приостановка процесса (перевод из состояния исполнение в состояние готовность) – запуск процесса (перевод из состояния готовность в состояние исполнение);
- блокирование процесса (перевод из состояния исполнение в состояние ожидание) – разблокирование процесса (перевод из состояния ожидание в состояние готовность).

В дальнейшем, когда мы будем говорить об алгоритмах планирования, в нашей модели появится еще одна операция, не имеющая парной: изменение приоритета процесса.

Операции создания и завершения процесса являются одноразовыми, так как применяются к процессу не более одного раза (некоторые системные процессы при работе вычислительной системы не завершаются никогда). Все остальные операции, связанные с изменением состояния процессов, будь то запуск или блокировка, как правило, являются многоразовыми. Рассмотрим подробнее, как операционная система выполняет операции над процессами.

Process Control Block и контекст процесса

Для того чтобы операционная система могла выполнять операции над процессами, каждый процесс представляется в ней некоторой структурой данных. Эта структура содержит информацию, специфическую для данного процесса:
состояние, в котором находится процесс;
программный счетчик процесса или, другими словами, адрес команды, которая должна быть выполнена для него следующей;
содержимое регистров процессора;
данные, необходимые для планирования использования процессора и управления памятью (приоритет процесса, размер и расположение адресного пространства и т. д.);
учетные данные (идентификационный номер процесса, какой пользователь инициировал его работу, общее время использования процессора данным процессом и т. д.);
сведения об устройствах ввода-вывода, связанных с процессом (например, какие устройства закреплены за процессом, таблицу открытых файлов).

Ее состав и строение зависят, конечно, от конкретной операционной системы. Во многих операционных системах информация, характеризующая процесс, хранится не в одной, а в

нескольких связанных структурах данных. Эти структуры могут иметь различные наименования, содержать дополнительную информацию или, наоборот, лишь часть описанной информации. Для нас это не имеет значения. Для нас важно лишь то, что для любого процесса, находящегося в вычислительной системе, вся информация, необходимая для совершения операций над ним, доступна операционной системе. Для простоты изложения будем считать, что она хранится в одной структуре данных. Мы будем называть ее PCB (Process Control Block) или блоком управления процессом. Блок управления процессом является моделью процесса для операционной системы. Любая операция, производимая операционной системой над процессом, вызывает определенные изменения в PCB. В рамках принятой модели состояний процессов содержимое PCB между операциями остается постоянным.

Информацию, для хранения которой предназначен блок управления процессом, удобно для дальнейшего изложения разделить на две части. Содержимое всех регистров процессора (включая значение программного счетчика) будем называть регистровым контекстом процесса, а все остальное – системным контекстом процесса. Знания регистрового и системного контекстов процесса достаточно для того, чтобы управлять его работой в операционной системе, совершая над ним операции. Однако этого недостаточно для того, чтобы полностью охарактеризовать процесс. Операционную систему не интересует, какими именно вычислениями занимается процесс, т. е. какой код и какие данные находятся в его адресном пространстве. С точки зрения пользователя, наоборот, наибольший интерес представляет содержимое адресного пространства процесса, возможно, наряду с регистровым контекстом определяющее последовательность преобразования данных и полученные результаты. Код и данные, находящиеся в адресном пространстве процесса, будем называть его пользовательским контекстом. Совокупность регистрового, системного и пользовательского контекстов процесса для краткости принято называть просто контекстом процесса. В любой момент времени процесс полностью характеризуется своим контекстом.

Одноразовые операции

Сложный жизненный путь процесса в компьютере начинается с его рождения. Любая операционная система, поддерживающая концепцию процессов, должна обладать средствами для их создания. В очень простых системах (например, в системах, спроектированных для работы только одного конкретного приложения) все процессы могут быть порождены на этапе старта системы. Более сложные операционные системы создают процессы динамически, по мере необходимости. Инициатором рождения нового процесса после старта операционной системы может выступить либо процесс пользователя, совершивший специальный системный вызов, либо сама операционная система, то есть, в конечном итоге, тоже некоторый процесс. Процесс, инициировавший создание нового процесса, принято называть процессом-родителем (parent process), а вновь созданный процесс – процессом-ребенком (child process). Процессы-дети могут в свою очередь порождать новых детей и т. д., образуя, в общем случае, внутри системы набор генеалогических деревьев процессов – генеалогический лес. Пример генеалогического леса изображен на рисунке 2.4. Следует отметить, что все пользовательские процессы вместе с некоторыми процессами операционной системы принадлежат одному и тому же дереву леса. В ряде вычислительных систем лес вообще вырождается в одно такое дерево.

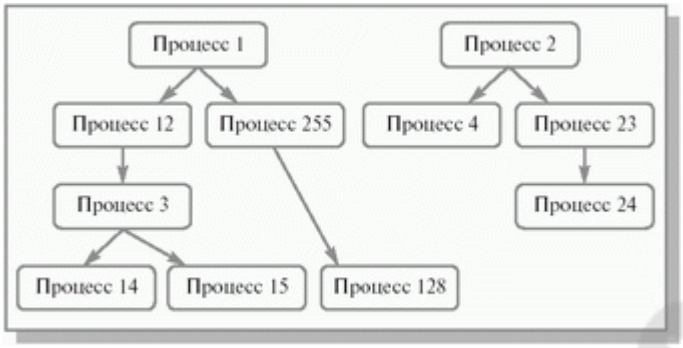

Рис. 2.4. Упрощенный генеалогический лес процессов. Стрелочка означает отношение родитель–ребенок

При рождении процесса система заводит новый PCB с состоянием процесса рождение и начинает его заполнять. Новый процесс получает собственный уникальный идентификационный номер. Поскольку для хранения идентификационного номера процесса в операционной системе отводится ограниченное количество битов, для соблюдения уникальности номеров количество одновременно присутствующих в ней процессов должно быть ограничено. После завершения какого-либо процесса его освободившийся идентификационный номер может быть повторно использован для другого процесса.

Обычно для выполнения своих функций процесс-ребенок требует определенных ресурсов: памяти, файлов, устройств ввода-вывода и т. д. Существует два подхода к их выделению. Новый процесс может получить в свое распоряжение некоторую часть родительских ресурсов, возможно разделяя с процессом-родителем и другими процессами-детьми права на них, или может получить свои ресурсы непосредственно от операционной системы. Информация о выделенных ресурсах заносится в PCB.

После наделения процесса-ребенка ресурсами необходимо занести в его адресное пространство программный код, значения данных, установить программный счетчик. Здесь также возможны два решения. В первом случае процесс-ребенок становится дубликатом процесса-родителя по регистровому и пользовательскому контекстам, при этом должен существовать способ определения, кто для кого из процессов-двойников является родителем. Во втором случае процесс-ребенок загружается новой программой из какого-либо файла. Операционная система Unix разрешает порождение процесса только первым способом; для запуска новой программы необходимо сначала создать копию процесса-родителя, а затем процесс-ребенок должен заменить свой пользовательский контекст с помощью специального системного вызова. Операционная система VAX/VMS допускает только второе решение. В Windows NT возможны оба варианта (в различных API).

Порождение нового процесса как дубликата процесса-родителя приводит к возможности существования программ (т. е. исполняемых файлов), для работы которых организуется более одного процесса. Возможность замены пользовательского контекста процесса по ходу его работы (т. е. загрузки для исполнения новой программы) приводит к тому, что в рамках одного и того же процесса может последовательно выполняться несколько различных программ.

После того как процесс наделен содержанием, в PCB дописывается оставшаяся информация, и состояние нового процесса изменяется на готовность. Осталось сказать несколько слов о том, как ведут себя процессы-родители после рождения процессов-детей. Процесс-родитель может продолжать свое выполнение одновременно с

выполнением процесса-ребенка, а может ожидать завершения работы некоторых или всех своих "детей".

Мы не будем подробно останавливаться на причинах, которые могут привести к завершению жизненного цикла процесса. После того как процесс завершил свою работу, операционная система переводит его в состояние закончил исполнение и освобождает все ассоциированные с ним ресурсы, делая соответствующие записи в блоке управления процессом. При этом сам PCB не уничтожается, а остается в системе еще некоторое время. Это связано с тем, что процесс-родитель после завершения процесса-ребенка может запросить операционную систему о причине "смерти" порожденного им процесса и/или статистическую информацию о его работе. Подобная информация сохраняется в PCB отработавшего процесса до запроса процесса-родителя или до конца его деятельности, после чего все следы завершившегося процесса окончательно исчезают из системы. В операционной системе Unix процессы, находящиеся в состоянии закончил исполнение, принято называть процессами-зомби.

Следует заметить, что в ряде операционных систем (например, в VAX/VMS) гибель процесса-родителя приводит к завершению работы всех его "детей". В других операционных системах (например, в Unix) процессы-дети продолжают свое существование и после окончания работы процесса-родителя. При этом возникает необходимость изменения информации в PCB процессов-детей о породившем их процессе для того, чтобы генеалогический лес процессов оставался целостным. Рассмотрим следующий пример. Пусть процесс с номером 2515 был порожден процессом с номером 2001 и после завершения его работы остается в вычислительной системе неограниченно долго. Тогда не исключено, что номер 2001 будет использован операционной системой повторно для совсем другого процесса. Если не изменить информацию о процессе-родителе для процесса 2515, то генеалогический лес процессов окажется некорректным – процесс 2515 будет считать своим родителем новый процесс 2001, а процесс 2001 будет открещиваться от нежданного потомка. Как правило, "осиротевшие" процессы "усыновляются" одним из системных процессов, который порождается при старте операционной системы и функционирует все время, пока она работает.

Многоразовые операции

Одноразовые операции приводят к изменению количества процессов, находящихся под управлением операционной системы, и всегда связаны с выделением или освобождением определенных ресурсов. Многоразовые операции, напротив, не приводят к изменению количества процессов в операционной системе и не обязаны быть связанными с выделением или освобождением ресурсов.

В этом разделе мы кратко опишем действия, которые производит операционная система при выполнении многоразовых операций над процессами. Более подробно эти действия будут рассмотрены далее в соответствующих лекциях.

Запуск процесса. Из числа процессов, находящихся в состоянии готовность, операционная система выбирает один процесс для последующего исполнения. Критерии и алгоритмы такого выбора будут подробно рассмотрены в лекции 3 – "Планирование процессов". Для избранного процесса операционная система обеспечивает наличие в оперативной памяти информации, необходимой для его дальнейшего выполнения. То, как она это делает, будет в деталях описано в лекциях 8-10. Далее состояние процесса изменяется на исполнение, восстанавливаются значения регистров для данного процесса и управление передается команде, на которую указывает счетчик команд процесса. Все данные, необходимые для

восстановления контекста, извлекаются из PCB процесса, над которым совершается операция.

Приостановка процесса. Работа процесса, находящегося в состоянии исполнение, приостанавливается в результате какого-либо прерывания. Процессор автоматически сохраняет счетчик команд и, возможно, один или несколько регистров в стеке исполняемого процесса, а затем передает управление по специальному адресу обработки данного прерывания. На этом деятельность hardware по обработке прерывания завершается. По указанному адресу обычно располагается одна из частей операционной системы. Она сохраняет динамическую часть системного и регистрового контекстов процесса в его PCB, переводит процесс в состояние готовность и приступает к обработке прерывания, то есть к выполнению определенных действий, связанных с возникшим прерыванием.

Блокирование процесса. Процесс блокируется, когда он не может продолжать работу, не дождавшись возникновения какого-либо события в вычислительной системе. Для этого он обращается к операционной системе с помощью определенного системного вызова. Операционная система обрабатывает системный вызов (инициализирует операцию ввода-вывода, добавляет процесс в очередь процессов, дожидающихся освобождения устройства или возникновения события, и т. д.) и, при необходимости сохранив нужную часть контекста процесса в его PCB, переводит процесс из состояния исполнение в состояние ожидание.

Разблокирование процесса. После возникновения в системе какого-либо события операционной системе нужно точно определить, какое именно событие произошло. Затем операционная система проверяет, находился ли некоторый процесс в состоянии ожидание для данного события, и если находился, переводит его в состояние готовность, выполняя необходимые действия, связанные с наступлением события (инициализация операции ввода-вывода для очередного ожидающего процесса и т. п.).

Переключение контекста

До сих пор мы рассматривали операции над процессами изолированно, независимо друг от друга. В действительности же деятельность мультипрограммной операционной системы состоит из цепочек операций, выполняемых над различными процессами, и сопровождается переключением процессора с одного процесса на другой.

Давайте для примера упрощенно рассмотрим, как в реальности может протекать операция разблокирования процесса, ожидающего ввода-вывода (см. рис. 2.5). При исполнении процессором некоторого процесса (на рисунке – процесс 1) возникает прерывание от устройства ввода-вывода, сигнализирующее об окончании операций на устройстве. Над выполняющимся процессом производится операция приостановки. Далее операционная система разблокирует процесс, инициировавший запрос на ввод-вывод (на рисунке – процесс 2) и осуществляет запуск приостановленного или нового процесса, выбранного при выполнении планирования (на рисунке был выбран разблокированный процесс). Как мы видим, в результате обработки информации об окончании операции ввода-вывода возможна смена процесса, находящегося в состоянии исполнение.

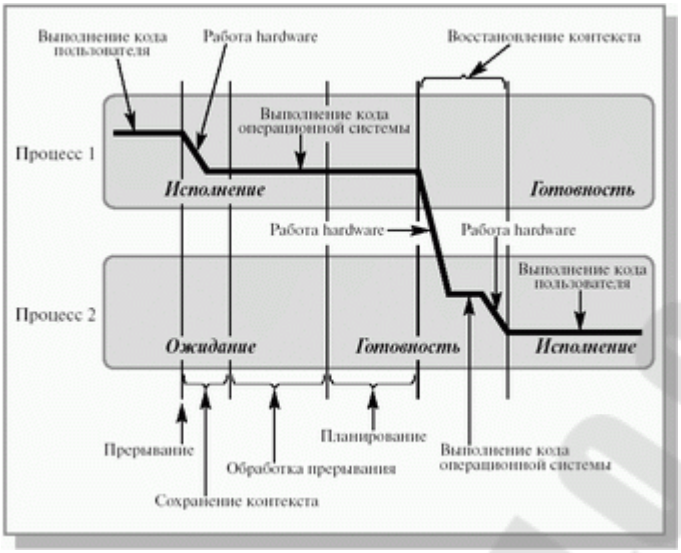

Рис. 2.5. Выполнение операции разблокирования процесса. Использование термина "код пользователя" не ограничивает общности рисунка только пользовательскими процессами

Для корректного переключения процессора с одного процесса на другой необходимо сохранить контекст исполнявшегося процесса и восстановить контекст процесса, на который будет переключен процессор. Такая процедура сохранения/восстановления работоспособности процессов называется переключением контекста. Время, затраченное на переключение контекста, не используется вычислительной системой для совершения полезной работы и представляет собой накладные расходы, снижающие производительность системы. Оно меняется от машины к машине и обычно колеблется в диапазоне от 1 до 1000 микросекунд. Существенно сократить накладные расходы в современных операционных системах позволяет расширенная модель процессов, включающая в себя понятие threads of execution (нити исполнения или просто нити). Подробнее о нитях исполнения мы будем говорить в лекции 4 – "Кооперация процессов и основные аспекты ее логической организации".

Заключение

Понятие процесса характеризует некоторую совокупность набора исполняющихся команд, ассоциированных с ним ресурсов и текущего момента его выполнения, находящуюся под управлением операционной системы. В любой момент процесс полностью описывается своим контекстом, состоящим из регистровой, системной и пользовательской частей. В операционной системе процессы представляются определенной структурой данных – PCB, отражающей содержание регистрового и системного контекстов. Процессы могут находиться в пяти основных состояниях: рождение, готовность, исполнение, ожидание, закончил исполнение. Из состояния в состояние процесс переводится операционной системой в результате выполнения над ним операций. Операционная система может выполнять над процессами следующие операции: создание процесса, завершение процесса, приостановка процесса, запуск процесса, блокирование процесса, разблокирование процесса, изменение приоритета процесса. Между операциями содержимое PCB не изменяется. Деятельность мультипрограммной операционной системы состоит из цепочек перечисленных операций, выполняемых над различными процессами, и сопровождается процедурами сохранения/восстановления работоспособности процессов, т. е. переключением контекста. Переключение контекста не имеет отношения к полезной работе, выполняемой процессами, и время, затраченное на него, сокращает полезное время работы процессора.

Лекция: Планирование процессов

Уровни планирования - планировании заданий и планировании использования процессора. Планирование заданий появилось в пакетных системах после того, как для хранения сформированных пакетов заданий начали использоваться магнитные диски. Магнитные диски, являясь устройствами прямого доступа, позволяют загружать задания в компьютер в произвольном порядке, а не только в том, в котором они были записаны на диск. Изменяя порядок загрузки заданий в вычислительную систему, можно повысить эффективность ее использования. Процедуру выбора очередного задания для загрузки в машину, т. е. для порождения соответствующего процесса, мы и назвали планированием заданий. Планирование использования процессора впервые возникает в мультипрограммных вычислительных системах, где в состоянии готовность могут одновременно находиться несколько процессов. Именно для процедуры выбора из них одного процесса, который получит процессор в свое распоряжение, т. е. будет переведен в состояние исполнение.

Планирование заданий используется в качестве долгосрочного планирования процессов. Оно отвечает за порождение новых процессов в системе, определяя ее степень мультипрограммирования, т. е. количество процессов, одновременно находящихся в ней. Если степень мультипрограммирования системы поддерживается постоянной, т. е. среднее количество процессов в компьютере не меняется, то новые процессы могут появляться только после завершения ранее загруженных. Поэтому долгосрочное планирование осуществляется достаточно редко, между появлением новых процессов могут проходить минуты и даже десятки минут. Решение о выборе для запуска того или иного процесса оказывает влияние на функционирование вычислительной системы на протяжении достаточно длительного времени. Отсюда и название этого уровня планирования – долгосрочное. В некоторых операционных системах долгосрочное планирование сведено к минимуму или отсутствует вовсе. Так, например, во многих интерактивных системах разделения времени порождение процесса происходит сразу после появления соответствующего запроса. Поддержание разумной степени мультипрограммирования осуществляется за счет ограничения количества пользователей, которые могут работать в системе, и особенностей человеческой психологии. Если между нажатием на клавишу и появлением символа на экране проходит 20–30 секунд, то многие пользователи предпочтут прекратить работу и продолжить ее, когда система будет менее загружена.

Планирование использования процессора применяется в качестве краткосрочного планирования процессов. Оно проводится, к примеру, при обращении исполняющегося процесса к устройствам ввода-вывода или просто по завершении определенного интервала времени. Поэтому краткосрочное планирование осуществляется, как правило, не реже одного раза в 100 миллисекунд. Выбор нового процесса для исполнения оказывает влияние на функционирование системы до наступления очередного аналогичного события, т. е. в течение короткого промежутка времени, чем и обусловлено название этого уровня планирования – краткосрочное.

В некоторых вычислительных системах бывает выгодно для повышения производительности временно удалить какой-либо частично выполнившийся процесс из оперативной памяти на диск, а позже вернуть его обратно для дальнейшего выполнения. Такая процедура в англоязычной литературе получила название swapping, что можно перевести на русский язык как "перекачка", хотя в специальной литературе оно

употребляется без перевода – свопинг. Когда и какой из процессов нужно перекачать на диск и вернуть обратно, решается дополнительным промежуточным уровнем планирования процессов – среднесрочным.

Критерии планирования и требования к алгоритмам

Для каждого уровня планирования процессов можно предложить много различных алгоритмов. Выбор конкретного алгоритма определяется классом задач, решаемых вычислительной системой, и целями, которых мы хотим достичь, используя планирование. К числу таких целей можно отнести следующие:

- Справедливость – гарантировать каждому заданию или процессу определенную часть времени использования процессора в компьютерной системе, стараясь не допустить возникновения ситуации, когда процесс одного пользователя постоянно занимает процессор, в то время как процесс другого пользователя фактически не начинал выполняться.
- Эффективность – постараться занять процессор на все 100% рабочего времени, не позволяя ему простаивать в ожидании процессов, готовых к исполнению. В реальных вычислительных системах загрузка процессора колеблется от 40 до 90%.
- Сокращение полного времени выполнения (turnaround time) – обеспечить минимальное время между стартом процесса или постановкой задания в очередь для загрузки и его завершением.
- Сокращение времени ожидания (waiting time) – сократить время, которое проводят процессы в состоянии готовность и задания в очереди для загрузки.
- Сокращение времени отклика (response time) – минимизировать время, которое требуется процессу в интерактивных системах для ответа на запрос пользователя.

Независимо от поставленных целей планирования желательно также, чтобы алгоритмы обладали следующими свойствами:

- Были предсказуемыми. Одно и то же задание должно выполняться приблизительно за одно и то же время. Применение алгоритма планирования не должно приводить, к примеру, к извлечению квадратного корня из 4 за сотые доли секунды при одном запуске и за несколько суток – при втором запуске.
- Были связаны с минимальными накладными расходами. Если на каждые 100 миллисекунд, выделенные процессу для использования процессора, будет приходиться 200 миллисекунд на определение того, какой именно процесс получит процессор в свое распоряжение, и на переключение контекста, то такой алгоритм, очевидно, применять не стоит.
- Равномерно загружали ресурсы вычислительной системы, отдавая предпочтение тем процессам, которые будут занимать малоиспользуемые ресурсы.
- Обладали масштабируемостью, т. е. не сразу теряли работоспособность при увеличении нагрузки. Например, рост количества процессов в системе в два раза не должен приводить к увеличению полного времени выполнения процессов на порядок.

Многие из приведенных выше целей и свойств являются противоречивыми. Улучшая работу алгоритма с точки зрения одного критерия, мы ухудшаем ее с точки зрения другого. Приспосабливая алгоритм под один класс задач, мы тем самым дискриминируем задачи другого класса.

Параметры планирования

Для осуществления поставленных целей разумные алгоритмы планирования должны опираться на какие-либо характеристики процессов в системе, заданий в очереди на загрузку, состояния самой вычислительной системы, иными словами, на параметры планирования. В этом разделе мы опишем ряд таких параметров, не претендуя на полноту изложения.

Все параметры планирования можно разбить на две большие группы: статические параметры и динамические параметры. Статические параметры не изменяются в ходе функционирования вычислительной системы, динамические же, напротив, подвержены постоянным изменениям.

К статическим параметрам вычислительной системы можно отнести предельные значения ее ресурсов (размер оперативной памяти, максимальное количество памяти на диске для осуществления свопинга, количество подключенных устройств ввода-вывода и т. п.). Динамические параметры системы описывают количество свободных ресурсов на данный момент.

К статическим параметрам процессов относятся характеристики, как правило, присущие заданиям уже на этапе загрузки.

- Каким пользователем запущен процесс или сформировано задание.
- Насколько важной является поставленная задача, т. е. каков приоритет ее выполнения.
- Сколько процессорного времени запрошено пользователем для решения задачи.
- Каково соотношение процессорного времени и времени, необходимого для осуществления операций ввода-вывода.
- Какие ресурсы вычислительной системы (оперативная память, устройства ввода-вывода, специальные библиотеки и системные программы и т. д.) и в каком количестве необходимы заданию.

Алгоритмы долгосрочного планирования используют в своей работе статические и динамические параметры вычислительной системы и статические параметры процессов (динамические параметры процессов на этапе загрузки заданий еще не известны). Алгоритмы краткосрочного и среднесрочного планирования дополнительно учитывают и динамические характеристики процессов. Для среднесрочного планирования в качестве таких характеристик может использоваться следующая информация:

- сколько времени прошло с момента выгрузки процесса на диск или его загрузки в оперативную память;
- сколько оперативной памяти занимает процесс;
- сколько процессорного времени уже предоставлено процессу.

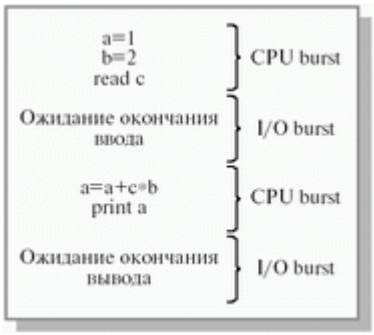

Рис. 3.1. Фрагмент деятельности процесса с выделением промежутков непрерывного использования процессора и ожидания ввода-вывода

Для краткосрочного планирования нам понадобится ввести еще два динамических параметра. Деятельность любого процесса можно представить как последовательность циклов использования процессора и ожидания завершения операций ввода-вывода. Промежуток времени непрерывного использования процессора носит название CPU burst, а промежуток времени непрерывного ожидания ввода-вывода – I/O burst. На рисунке 3.1. показан фрагмент деятельности некоторого процесса на псевдоязыке программирования с выделением указанных промежутков. Для краткости мы будем использовать термины CPU burst и I/O burst без перевода. Значения продолжительности последних и очередных CPU burst и I/O burst являются важными динамическими параметрами процесса.

Вытесняющее и невытесняющее планирование

Процесс планирования осуществляется частью операционной системы, называемой планировщиком. Планировщик может принимать решения о выборе для исполнения нового процесса из числа находящихся в состоянии готовность в следующих четырех случаях:

1. Когда процесс переводится из состояния исполнение в состояние закончил исполнение.
2. Когда процесс переводится из состояния исполнение в состояние ожидание.
3. Когда процесс переводится из состояния исполнение в состояние готовность (например, после прерывания от таймера).
4. Когда процесс переводится из состояния ожидание в состояние готовность (завершилась операция ввода-вывода или произошло другое событие). Подробно процедура такого перевода рассматривалась в лекции 2 (раздел "Переключение контекста"), где мы показали, почему при этом возникает возможность смены процесса, находящегося в состоянии исполнение.

В случаях 1 и 2 процесс, находившийся в состоянии исполнение, не может дальше исполняться, и операционная система вынуждена осуществлять планирование выбирая новый процесс для выполнения. В случаях 3 и 4 планирование может как проводиться, так и не проводиться, планировщик не вынужден обязательно принимать решение о выборе процесса для выполнения, процесс, находившийся в состоянии исполнение может просто продолжить свою работу. Если в операционной системе планирование осуществляется только в вынужденных ситуациях, говорят, что имеет место невытесняющее (nonpreemptive) планирование. Если планировщик принимает и вынужденные, и невынужденные решения, говорят о вытесняющем (preemptive) планировании. Термин "вытесняющее планирование" возник потому, что исполняющийся процесс помимо своей воли может быть вытеснен из состояния исполнение другим процессом.

Невытесняющее планирование используется, например, в MS Windows 3.1 и ОС Apple Macintosh. При таком режиме планирования процесс занимает столько процессорного времени, сколько ему необходимо. При этом переключение процессов возникает только при желании самого исполняющегося процесса передать управление (для ожидания завершения операции ввода-вывода или по окончании работы). Этот метод планирования относительно просто реализуем и достаточно эффективен, так как позволяет выделить большую часть процессорного времени для работы самих процессов и до минимума сократить затраты на переключение контекста. Однако при невытесняющем планировании возникает проблема возможности полного захвата процессора одним процессом, который вследствие каких-либо причин (например, из-за ошибки в программе) зацикливается и не может передать управление другому процессу. В такой ситуации спасает только перезагрузка всей вычислительной системы.

Вытесняющее планирование обычно используется в системах разделения времени. В этом режиме планирования процесс может быть приостановлен в любой момент исполнения. Операционная система устанавливает специальный таймер для генерации сигнала прерывания по истечении некоторого интервала времени – кванта. После прерывания процессор передается в распоряжение следующего процесса. Временные прерывания помогают гарантировать приемлемое время отклика процессов для пользователей, работающих в диалоговом режиме, и предотвращают "зависание" компьютерной системы из-за зацикливания какой-либо программы.

Алгоритмы планирования

Существует достаточно большой набор разнообразных алгоритмов планирования, которые предназначены для достижения различных целей и эффективны для разных классов задач. Многие из них могут использоваться на нескольких уровнях планирования. В этом разделе мы рассмотрим некоторые наиболее употребительные алгоритмы применительно к процессу кратковременного планирования.

First-Come, First-Served (FCFS)

Простейшим алгоритмом планирования является алгоритм, который принято обозначать аббревиатурой FCFS по первым буквам его английского названия – First-Come, First-Served (первым пришел, первым обслужен). Представим себе, что процессы, находящиеся в состоянии готовность, выстроены в очередь. Когда процесс переходит в состояние готовность, он, а точнее, ссылка на его PCB помещается в конец этой очереди. Выбор нового процесса для исполнения осуществляется из начала очереди с удалением оттуда ссылки на его PCB. Очередь подобного типа имеет в программировании специальное наименование – FIFO1), сокращение от First In, First Out (первым вошел, первым вышел).

Такой алгоритм выбора процесса осуществляет невытесняющее планирование. Процесс, получивший в свое распоряжение процессор, занимает его до истечения текущего CPU burst. После этого для выполнения выбирается новый процесс из начала очереди.

Таблица 3.1.

Процесс	p0	p1	p2
Продолжительность очередного CPU burst	13	4	1

Преимуществом алгоритма FCFS является легкость его реализации, но в то же время он имеет и много недостатков. Рассмотрим следующий пример. Пусть в состоянии готовность находятся три процесса p0, p1 и p2, для которых известны времена их очередных CPU burst. Эти времена приведены в таблице 3.1. в некоторых условных единицах. Для простоты будем полагать, что вся деятельность процессов ограничивается использованием только одного промежутка CPU burst, что процессы не совершают операций ввода-вывода и что время переключения контекста так мало, что им можно пренебречь.

Если процессы расположены в очереди процессов, готовых к исполнению, в порядке p0, p1, p2, то картина их выполнения выглядит так, как показано на рисунке 3.2. Первым для выполнения выбирается процесс p0, который получает процессор на все время своего CPU burst, т. е. на 13 единиц времени. После его окончания в состояние исполнение переводится процесс p1, он занимает процессор на 4 единицы времени. И, наконец, возможность работать получает процесс p2. Время ожидания для процесса p0 составляет 0 единиц времени, для процесса p1 – 13 единиц, для процесса p2 – 13 + 4 = 17 единиц.

Таким образом, среднее время ожидания в этом случае – (0 + 13 + 17)/3 = 10 единиц времени. Полное время выполнения для процесса p0 составляет 13 единиц времени, для процесса p1 – 13 + 4 = 17 единиц, для процесса p2 – 13 + 4 + 1 = 18 единиц. Среднее полное время выполнения оказывается равным (13 + 17 + 18)/3 = 16 единицам времени.

Рис. 3.2. Выполнение процессов при порядке p0,p1,p2

Если те же самые процессы расположены в порядке p2, p1, p0, то картина их выполнения будет соответствовать рисунку 3.3. Время ожидания для процесса p0 равняется 5 единицам времени, для процесса p1 – 1 единице, для процесса p2 – 0 единиц. Среднее время ожидания составит (5 + 1 + 0)/3 = 2 единицы времени. Это в 5 (!) раз меньше, чем в предыдущем случае. Полное время выполнения для процесса p0 получается равным 18 единицам времени, для процесса p1 – 5 единицам, для процесса p2 – 1 единице. Среднее полное время выполнения составляет (18 + 5 + 1)/3 = 8 единиц времени, что почти в 2 раза меньше, чем при первой расстановке процессов.

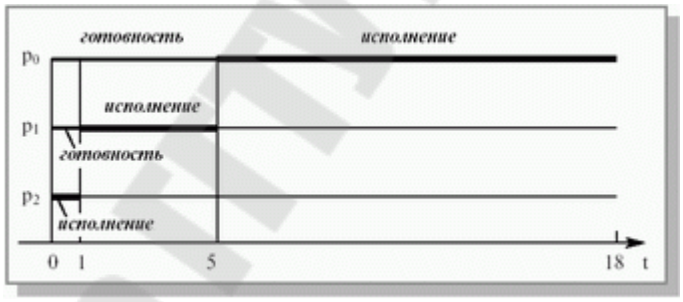

Рис. 3.3. Выполнение процессов при порядке p2, p1, p0

Как мы видим, среднее время ожидания и среднее полное время выполнения для этого алгоритма существенно зависят от порядка расположения процессов в очереди. Если у нас есть процесс с длительным CPU burst, то короткие процессы, перешедшие в состояние готовность после длительного процесса, будут очень долго ждать начала выполнения. Поэтому алгоритм FCFS практически неприменим для систем разделения времени – слишком большим получается среднее время отклика в интерактивных процессах.

Round Robin (RR)

Модификацией алгоритма FCFS является алгоритм, получивший название Round Robin (Round Robin – это вид детской карусели в США) или сокращенно RR. По сути дела, это тот же самый алгоритм, только реализованный в режиме вытесняющего планирования. Можно представить себе все множество готовых процессов организованным циклически – процессы сидят на карусели. Карусель вращается так, что каждый процесс находится около процессора небольшой фиксированный квант времени, обычно 10 – 100 миллисекунд (см. рис. 3.4.). Пока процесс находится рядом с процессором, он получает процессор в свое распоряжение и может исполняться.

Рис. 3.4. Процессы на карусели

Реализуется такой алгоритм так же, как и предыдущий, с помощью организации процессов, находящихся в состоянии готовность, в очередь FIFO. Планировщик выбирает для очередного исполнения процесс, расположенный в начале очереди, и устанавливает таймер для генерации прерывания по истечении определенного кванта времени. При выполнении процесса возможны два варианта.

- Время непрерывного использования процессора, необходимое процессу (остаток текущего CPU burst), меньше или равно продолжительности кванта времени. Тогда процесс по своей воле освобождает процессор до истечения кванта времени, на исполнение поступает новый процесс из начала очереди, и таймер начинает отсчет кванта заново.
- Продолжительность остатка текущего CPU burst процесса больше, чем квант времени. Тогда по истечении этого кванта процесс прерывается таймером и помещается в конец очереди процессов, готовых к исполнению, а процессор выделяется для использования процессу, находящемуся в ее начале.

Рассмотрим предыдущий пример с порядком процессов p0, p1, p2 и величиной кванта времени равной 4. Выполнение этих процессов иллюстрируется таблицей 3.2. Обозначение "И" используется в ней для процесса, находящегося в состоянии исполнение, обозначение "Г" – для процессов в состоянии готовность, пустые ячейки соответствуют завершившимся процессам. Состояния процессов показаны на протяжении соответствующей единицы времени, т. е. колонка с номером 1 соответствует промежутку времени от 0 до 1.

Таблица 3.2.

Время	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
p0	И	И	И	И	Г	Г	Г	Г	Г	И	И	И	И	И	И	И	И	И
p1	Г	Г	Г	Г	И	И	И	И										
p2	Г	Г	Г	Г	Г	Г	Г	Г	И									

Первым для исполнения выбирается процесс p0. Продолжительность его CPU burst больше, чем величина кванта времени, и поэтому процесс исполняется до истечения кванта, т. е. в течение 4 единиц времени. После этого он помещается в конец очереди готовых к исполнению процессов, которая принимает вид p1, p2, p0. Следующим начинает выполняться процесс p1. Время его исполнения совпадает с величиной выделенного кванта, поэтому процесс работает до своего завершения. Теперь очередь процессов в состоянии готовность состоит из двух процессов, p2 и p0. Процессор выделяется процессу p2. Он завершается до истечения отпущенного ему процессорного времени, и очередные кванты отмеряются процессу p0 – единственному не закончившему к этому моменту свою работу. Время ожидания для процесса p0 (количество символов "Г" в соответствующей строке) составляет 5 единиц времени, для процесса p1 – 4 единицы

времени, для процесса p2 – 8 единиц времени. Таким образом, среднее время ожидания для этого алгоритма получается равным (5 + 4 + 8)/3 = 5,6(6) единицы времени. Полное время выполнения для процесса p0 (количество непустых столбцов в соответствующей строке) составляет 18 единиц времени, для процесса p1 – 8 единиц, для процесса p2 – 9 единиц. Среднее полное время выполнения оказывается равным (18 + 8 + 9)/3 = 11,6(6) единицы времени.

Легко увидеть, что среднее время ожидания и среднее полное время выполнения для обратного порядка процессов не отличаются от соответствующих времен для алгоритма FCFS и составляют 2 и 6 единиц времени соответственно.

На производительность алгоритма RR сильно влияет величина кванта времени. Рассмотрим тот же самый пример с порядком процессов p0, p1, p2 для величины кванта времени, равной 1 (см. табл. 3.3.). Время ожидания для процесса p0 составит 5 единиц времени, для процесса p1 – тоже 5 единиц, для процесса p2 – 2 единицы. В этом случае среднее время ожидания получается равным (5 + 5 + 2)/3 = 4 единицам времени. Среднее полное время исполнения составит (18 + 9 + 3)/3 = 10 единиц времени.

Таблица 3.3.

Время	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
p0	И	Г	Г	И	Г	И	Г	И	Г	И	И	И	И	И	И	И	И	И
p1	Г	И	Г	Г	И	Г	И	Г	И									
p2	Г	Г	И															

При очень больших величинах кванта времени, когда каждый процесс успевает завершить свой CPU burst до возникновения прерывания по времени, алгоритм RR вырождается в алгоритм FCFS. При очень малых величинах создается иллюзия того, что каждый из n процессов работает на собственном виртуальном процессоре с производительностью ~ 1/n от производительности реального процессора. Правда, это справедливо лишь при теоретическом анализе при условии пренебрежения временами переключения контекста процессов. В реальных условиях при слишком малой величине кванта времени и, соответственно, слишком частом переключении контекста накладные расходы на переключение резко снижают производительность системы.

Shortest-Job-First (SJF)

При рассмотрении алгоритмов FCFS и RR мы видели, насколько существенным для них является порядок расположения процессов в очереди процессов, готовых к исполнению. Если короткие задачи расположены в очереди ближе к ее началу, то общая производительность этих алгоритмов значительно возрастает. Если бы мы знали время следующих CPU burst для процессов, находящихся в состоянии готовность, то могли бы выбрать для исполнения не процесс из начала очереди, а процесс с минимальной длительностью CPU burst. Если же таких процессов два или больше, то для выбора одного из них можно использовать уже известный нам алгоритм FCFS. Квантование времени при этом не применяется. Описанный алгоритм получил название "кратчайшая работа первой" или Shortest Job First (SJF).

SJF-алгоритм краткосрочного планирования может быть как вытесняющим, так и невытесняющим. При невытесняющем SJF-планировании процессор предоставляется избранному процессу на все необходимое ему время, независимо от событий, происходящих в вычислительной системе. При вытесняющем SJF-планировании учитывается появление новых процессов в очереди готовых к исполнению (из числа вновь родившихся или разблокированных) во время работы выбранного процесса. Если CPU

burst нового процесса меньше, чем остаток CPU burst у исполняющегося, то исполняющийся процесс вытесняется новым.

Рассмотрим пример работы невытесняющего алгоритма SJF. Пусть в состоянии готовность находятся четыре процесса, p0, p1, p2 и p3, для которых известны времена их очередных CPU burst. Эти времена приведены в таблице 3.4. Как и прежде, будем полагать, что вся деятельность процессов ограничивается использованием только одного промежутка CPU burst, что процессы не совершают операций ввода-вывода и что временем переключения контекста можно пренебречь.

Таблица 3.4.

Процесс	p0	p1	p2	p3
Продолжительность очередного CPU burst	5	3	7	1

При использовании невытесняющего алгоритма SJF первым для исполнения будет выбран процесс p3, имеющий наименьшее значение продолжительности очередного CPU burst. После его завершения для исполнения выбирается процесс p1, затем p0 и, наконец, p2. Эта картина отражена в таблице 3.5.

Таблица 3.5.

Время	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
p0	Г	Г	Г	Г	И	И	И	И	И							
p1	Г	И	И	И												
p2	Г	Г	Г	Г	Г	Г	Г	Г	Г	И	И	И	И	И	И	И
p3	И															

Как мы видим, среднее время ожидания для алгоритма SJF составляет (4 + 1 + 9 + 0)/4 = 3,5 единицы времени. Легко посчитать, что для алгоритма FCFS при порядке процессов p0, p1, p2, p3 эта величина будет равняться (0 + 5 + 8 + 15)/4 = 7 единицам времени, т. е. будет в два раза больше, чем для алгоритма SJF. Можно показать, что для заданного набора процессов (если в очереди не появляются новые процессы) алгоритм SJF является оптимальным с точки зрения минимизации среднего времени ожидания среди класса невытесняющих алгоритмов.

Для рассмотрения примера вытесняющего SJF планирования мы возьмем ряд процессов p0, p1, p2 и p3 с различными временами CPU burst и различными моментами их появления в очереди процессов, готовых к исполнению (см. табл. 3.6.).

Таблица 3.6.

Процесс	Время появления в очереди очередного CPU burst	Продолжительность
p0	0	6
p1	2	2
p2	6	7
p3	0	5

В начальный момент времени в состоянии готовность находятся только два процесса, p0 и p3. Меньшее время очередного CPU burst оказывается у процесса p3, поэтому он и выбирается для исполнения (см. таблицу 3.7.). По прошествии 2 единиц времени в систему поступает процесс p1. Время его CPU burst меньше, чем остаток CPU burst у процесса p3, который вытесняется из состояния исполнение и переводится в состояние готовность. По прошествии еще 2 единиц времени процесс p1 завершается, и для исполнения вновь выбирается процесс p3. В момент времени t = 6 в очереди процессов,

готовых к исполнению, появляется процесс p2, но поскольку ему для работы нужно 7 единиц времени, а процессу p3 осталось трудиться всего 1 единицу времени, то процесс p3 остается в состоянии исполнение. После его завершения в момент времени t = 7 в очереди находятся процессы p0 и p2, из которых выбирается процесс p0. Наконец, последним получит возможность выполняться процесс p2.

Таблица 3.7.

Время	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
p0	Г	Г	Г	Г	Г	Г	Г	И	И	И	И	И	И							
p1			И	И																
p2							Г	Г	Г	Г	Г	Г	Г	И	И	И	И	И	И	И
p3	И	И	Г	Г	И	И	И													

Основную сложность при реализации алгоритма SJF представляет невозможность точного знания продолжительности очередного CPU burst для исполняющихся процессов. В пакетных системах количество процессорного времени, необходимое заданию для выполнения, указывает пользователь при формировании задания. Мы можем брать эту величину для осуществления долгосрочного SJF-планирования. Если пользователь укажет больше времени, чем ему нужно, он будет ждать результата дольше, чем мог бы, так как задание будет загружено в систему позже. Если же он укажет меньшее количество времени, задача может не досчитаться до конца. Таким образом, в пакетных системах решение задачи оценки времени использования процессора перекладывается на плечи пользователя. При краткосрочном планировании мы можем делать только прогноз длительности следующего CPU burst, исходя из предыстории работы процесса. Пусть $\tau(n)$ – величина n-го CPU burst, $T(n + 1)$ – предсказываемое значение для n + 1-го CPU burst, – некоторая величина в диапазоне от 0 до 1.

Определим рекуррентное соотношение $T(n+1)= \tau(n)+(1-)T(n)$

T(0) положим произвольной константой. Первое слагаемое учитывает последнее поведение процесса, тогда как второе слагаемое учитывает его предысторию. При = 0 мы перестаем следить за последним поведением процесса, фактически полагая

$T(n)= T(n+1)=...=T(0)$

т. е. оценивая все CPU burst одинаково, исходя из некоторого начального предположения.

Положив = 1, мы забываем о предыстории процесса. В этом случае мы полагаем, что время очередного CPU burst будет совпадать со временем последнего CPU burst: $T(n+1)= \tau(n)$

Обычно выбирают = 1/2 для равноценного учета последнего поведения и предыстории. Надо отметить, что такой выбор удобен и для быстрой организации вычисления оценки $T(n + 1)$. Для подсчета новой оценки нужно взять старую оценку, сложить с измеренным временем CPU burst и полученную сумму разделить на 2, например, сдвинув ее на 1 бит вправо. Полученные оценки $T(n + 1)$ применяются как продолжительности очередных промежутков времени непрерывного использования процессора для краткосрочного SJF-планирования.

Гарантированное планирование

При интерактивной работе N пользователей в вычислительной системе можно применить алгоритм планирования, который гарантирует, что каждый из пользователей будет иметь в своем распоряжении ~1/N часть процессорного времени. Пронумеруем всех пользователей от 1 до N. Для каждого пользователя с номером i введем две величины: Ti – время нахождения пользователя в системе или, другими словами, длительность сеанса его общения с машиной и τi – суммарное процессорное время уже выделенное всем его процессам в течение сеанса. Справедливым для пользователя было бы получение Ti/N процессорного времени. Если

τi<<Ti/N

то i-й пользователь несправедливо обделен процессорным временем. Если же

τi>>Ti/N

то система явно благоволит к пользователю с номером i. Вычислим для процессов каждого пользователя значение коэффициента справедливости

τiN/Ti

и будем предоставлять очередной квант времени готовому процессу с наименьшей величиной этого отношения. Предложенный алгоритм называют алгоритмом гарантированного планирования. К недостаткам этого алгоритма можно отнести невозможность предугадать поведение пользователей. Если некоторый пользователь отправится на пару часов пообедать и поспать, не прерывая сеанса работы, то по возвращении его процессы будут получать неоправданно много процессорного времени.

Приоритетное планирование

Алгоритмы SJF и гарантированного планирования представляют собой частные случаи приоритетного планирования. При приоритетном планировании каждому процессу присваивается определенное числовое значение – приоритет, в соответствии с которым ему выделяется процессор. Процессы с одинаковыми приоритетами планируются в порядке FCFS. Для алгоритма SJF в качестве такого приоритета выступает оценка продолжительности следующего CPU burst. Чем меньше значение этой оценки, тем более высокий приоритет имеет процесс. Для алгоритма гарантированного планирования приоритетом служит вычисленный коэффициент справедливости. Чем он меньше, тем больше у процесса приоритет.

Алгоритмы назначения приоритетов процессов могут опираться как на внутренние параметры, связанные с происходящим внутри вычислительной системы, так и на внешние по отношению к ней. К внутренним параметрам относятся различные количественные и качественные характеристики процесса такие как: ограничения по времени использования процессора, требования к размеру памяти, число открытых файлов и используемых устройств ввода-вывода, отношение средних продолжительностей I/O burst к CPU burst и т. д. Алгоритмы SJF и гарантированного планирования используют внутренние параметры. В качестве внешних параметров могут выступать важность процесса для достижения каких-либо целей, стоимость оплаченного процессорного времени и другие политические факторы. Высокий внешний приоритет может быть присвоен задаче лектора или того, кто заплатил $100 за работу в течение одного часа.

Планирование с использованием приоритетов может быть как вытесняющим, так и невытесняющим. При вытесняющем планировании процесс с более высоким приоритетом, появившийся в очереди готовых процессов, вытесняет исполняющийся процесс с более низким приоритетом. В случае невытесняющего планирования он просто становится в начало очереди готовых процессов. Давайте рассмотрим примеры использования различных режимов приоритетного планирования.

Пусть в очередь процессов, находящихся в состоянии готовность, поступают те же процессы, что и в примере для вытесняющего алгоритма SJF, только им дополнительно еще присвоены приоритеты (см. табл. 3.8.). В вычислительных системах не существует определенного соглашения, какое значение приоритета – 1 или 4 считать более приоритетным. Во избежание путаницы, во всех наших примерах мы будем предполагать, что большее значение соответствует меньшему приоритету, т. е. наиболее приоритетным в нашем примере является процесс p3, а наименее приоритетным – процесс p0.

Таблица 3.8.

Процесс	Время появления в очереди	Продолжительность очередного CPU burst	Приоритет
p0	0	6	4
p1	2	2	3
p2	6	7	2
p3	0	5	1

Как будут вести себя процессы при использовании невытесняющего приоритетного планирования? Первым для выполнения в момент времени t = 0 выбирается процесс p3, как обладающий наивысшим приоритетом. После его завершения в момент времени t = 5 в очереди процессов, готовых к исполнению, окажутся два процесса p0 и p1. Больший приоритет из них у процесса p1, он и начнет выполняться (см. табл. 3.9.). Затем в момент времени t = 8 для исполнения будет избран процесс p2, и лишь потом – процесс p0.

Таблица 3.9.

Время	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
p0	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	И	И	И	И	И	И
p1			Г	Г	Г	И	И													
p2							Г	И	И	И	И	И	И	И						
p3	И	И	И	И	И															

Иным будет предоставление процессора процессам в случае вытесняющего приоритетного планирования (см. табл. 3.10.). Первым, как и в предыдущем случае, начнет исполняться процесс p3, а по его окончании – процесс p1. Однако в момент времени t = 6 он будет вытеснен процессом p2 и продолжит свое выполнение только в момент времени t = 13. Последним, как и раньше, будет исполняться процесс p0.

Таблица 3.10.

Время	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
p0	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	И	И	И	И	И	И
p1			Г	Г	Г	И	Г	Г	Г	Г	Г	Г	Г	И						
p2							И	И	И	И	И	И	И							
p3	И	И	И	И	И															

В рассмотренном выше примере приоритеты процессов с течением времени не изменялись. Такие приоритеты принято называть статическими. Механизмы статической приоритетности легко реализовать, и они сопряжены с относительно небольшими издержками на выбор наиболее приоритетного процесса. Однако статические приоритеты не реагируют на изменения ситуации в вычислительной системе, которые могут сделать желательной корректировку порядка исполнения процессов. Более гибкими являются динамические приоритеты процессов, изменяющие свои значения по ходу исполнения процессов. Начальное значение динамического приоритета, присвоенное процессу, действует в течение лишь короткого периода времени, после чего ему назначается новое, более подходящее значение. Изменение динамического приоритета процесса является единственной операцией над процессами, которую мы до сих пор не рассмотрели. Как правило, изменение приоритета процессов проводится согласованно с совершением каких-либо других операций: при рождении нового процесса, при разблокировке или блокировании процесса, по истечении определенного кванта времени или по завершении процесса. Примерами алгоритмов с динамическими приоритетами являются алгоритм SJF и алгоритм гарантированного планирования. Схемы с динамической приоритетностью гораздо сложнее в реализации и связаны с большими издержками по сравнению со статическими схемами. Однако их использование предполагает, что эти издержки оправдываются улучшением работы системы.

Главная проблема приоритетного планирования заключается в том, что при ненадлежащем выборе механизма назначения и изменения приоритетов низкоприоритетные процессы могут не запускаться неопределенно долгое время. Обычно случается одно из двух. Или они все же дожидаются своей очереди на исполнение (в девять часов утра в воскресенье, когда все приличные программисты ложатся спать). Или вычислительную систему приходится выключать, и они теряются (при остановке IBM 7094 в Массачусетском технологическом институте в 1973 году были найдены процессы, запущенные в 1967 году и ни разу с тех пор не исполнявшиеся). Решение этой проблемы может быть достигнуто с помощью увеличения со временем значения приоритета процесса, находящегося в состоянии готовность. Пусть изначально процессам присваиваются приоритеты от 128 до 255. Каждый раз по истечении определенного промежутка времени значения приоритетов готовых процессов уменьшаются на 1. Процессу, побывавшему в состоянии исполнение, присваивается первоначальное значение приоритета. Даже такая грубая схема гарантирует, что любому процессу в разумные сроки будет предоставлено право на исполнение.

Многоуровневые очереди (Multilevel Queue)

Для систем, в которых процессы могут быть легко рассортированы по разным группам, был разработан другой класс алгоритмов планирования. Для каждой группы процессов создается своя очередь процессов, находящихся в состоянии готовность (см. рис. 3.5).

Рис. 3.5. Несколько очередей планирования

Этим очередям приписываются фиксированные приоритеты. Например, приоритет очереди системных процессов устанавливается выше, чем приоритет очередей пользовательских процессов. А приоритет очереди процессов, запущенных студентами, ниже, чем для очереди процессов, запущенных преподавателями. Это значит, что ни один пользовательский процесс не будет выбран для исполнения, пока есть хоть один готовый системный процесс, и ни один студенческий процесс не получит в свое распоряжение процессор, если есть процессы преподавателей, готовые к исполнению. Внутри этих очередей для планирования могут применяться самые разные алгоритмы. Так, например, для больших счетных процессов, не требующих взаимодействия с пользователем (фоновых процессов), может использоваться алгоритм FCFS, а для интерактивных процессов – алгоритм RR. Подобный подход, получивший название многоуровневых очередей, повышает гибкость планирования: для процессов с различными характеристиками применяется наиболее подходящий им алгоритм.

Многоуровневые очереди с обратной связью (Multilevel Feedback Queue)

Дальнейшим развитием алгоритма многоуровневых очередей является добавление к нему механизма обратной связи. Здесь процесс не постоянно приписан к определенной очереди, а может мигрировать из одной очереди в другую в зависимости от своего поведения.
Для простоты рассмотрим ситуацию, когда процессы в состоянии готовность организованы в 4 очереди, как на рисунке 3.6.

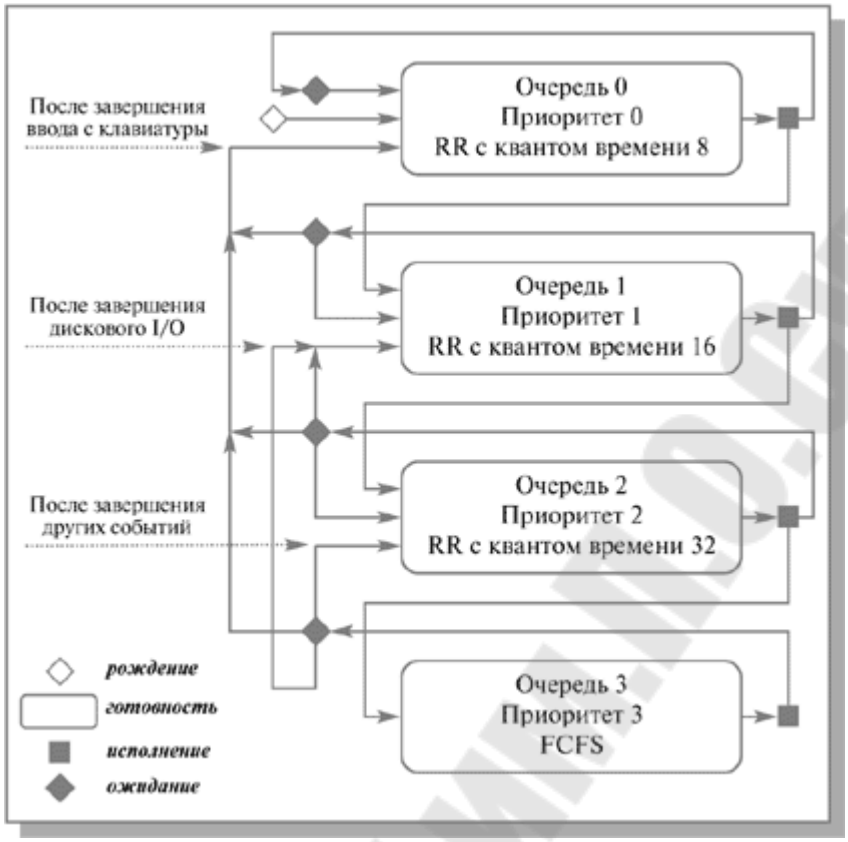

Рис. 3.6. Схема миграции процессов в многоуровневых очередях планирования с обратной связью. Вытеснение процессов более приоритетными процессами и завершение процессов на схеме не показано

Планирование процессов между очередями осуществляется на основе вытесняющего приоритетного механизма. Чем выше на рисунке располагается очередь, тем выше ее приоритет. Процессы в очереди 1 не могут исполняться, если в очереди 0 есть хотя бы один процесс. Процессы в очереди 2 не будут выбраны для выполнения, пока есть хоть один процесс в очередях 0 и 1. И наконец, процесс в очереди 3 может получить процессор в свое распоряжение только тогда, когда очереди 0, 1 и 2 пусты. Если при работе процесса появляется другой процесс в какой-либо более приоритетной очереди, исполняющийся процесс вытесняется новым. Планирование процессов внутри очередей 0–2 осуществляется с использованием алгоритма RR, планирование процессов в очереди 3 основывается на алгоритме FCFS.

Родившийся процесс поступает в очередь 0. При выборе на исполнение он получает в свое распоряжение квант времени размером 8 единиц. Если продолжительность его CPU burst меньше этого кванта времени, процесс остается в очереди 0. В противном случае он переходит в очередь 1. Для процессов из очереди 1 квант времени имеет величину 16. Если процесс не укладывается в это время, он переходит в очередь 2. Если укладывается – остается в очереди 1. В очереди 2 величина кванта времени составляет 32 единицы. Если для непрерывной работы процесса и этого мало, процесс поступает в очередь 3, для которой квантование времени не применяется и, при отсутствии готовых процессов в других очередях, может исполняться до окончания своего CPU burst. Чем больше значение продолжительности CPU burst, тем в менее приоритетную очередь попадает процесс, но тем на большее процессорное время он может рассчитывать. Таким образом, через некоторое время все процессы, требующие малого времени работы процессора, окажутся размещенными в высокоприоритетных очередях, а все процессы, требующие большого счета и с низкими запросами к времени отклика, – в низкоприоритетных.

Миграция процессов в обратном направлении может осуществляться по различным принципам. Например, после завершения ожидания ввода с клавиатуры процессы из

очередей 1, 2 и 3 могут помещаться в очередь 0, после завершения дисковых операций ввода-вывода процессы из очередей 2 и 3 могут помещаться в очередь 1, а после завершения ожидания всех других событий – из очереди 3 в очередь 2. Перемещение процессов из очередей с низкими приоритетами в очереди с высокими приоритетами позволяет более полно учитывать изменение поведения процессов с течением времени.

Многоуровневые очереди с обратной связью представляют собой наиболее общий подход к планированию процессов из числа подходов, рассмотренных нами. Они наиболее трудны в реализации, но в то же время обладают наибольшей гибкостью. Понятно, что существует много других разновидностей такого способа планирования, помимо варианта, приведенного выше. Для полного описания их конкретного воплощения необходимо указать:

- Количество очередей для процессов, находящихся в состоянии готовность.
- Алгоритм планирования, действующий между очередями.
- Алгоритмы планирования, действующие внутри очередей.
- Правила помещения родившегося процесса в одну из очередей.
- Правила перевода процессов из одной очереди в другую.

Изменяя какой-либо из перечисленных пунктов, мы можем существенно менять поведение вычислительной системы.

Простейшим алгоритмом планирования является невытесняющий алгоритм FCFS, который, однако, может существенно задерживать короткие процессы, не вовремя перешедшие в состояние готовность. В системах разделения времени широкое распространение получила вытесняющая версия этого алгоритма – RR.

Среди всех невытесняющих алгоритмов оптимальным с точки зрения среднего времени ожидания процессов является алгоритм SJF. Существует и вытесняющий вариант этого алгоритма. В интерактивных системах часто используется алгоритм гарантированного планирования, обеспечивающий пользователям равные части процессорного времени.

Алгоритм SJF и алгоритм гарантированного планирования являются частными случаями планирования с использованием приоритетов. В более общих методах приоритетного планирования применяются многоуровневые очереди процессов, готовых к исполнению, и многоуровневые очереди с обратной связью. Будучи наиболее сложными в реализации, эти способы планирования обеспечивают гибкое поведение вычислительных систем и их адаптивность к решению задач разных классов.

Лекция: Логическая организация взаимодействия процессов

Взаимодействие процессов в вычислительной системе напоминает жизнь в коммунальной квартире. Постоянное ожидание в очереди к местам общего пользования (процессору) и ежедневная борьба за ресурсы (кто опять занял все конфорки на плите?). Для нормального функционирования процессов операционная система старается максимально обособить их друг от друга. Каждый процесс имеет собственное адресное пространство (каждая семья должна жить в отдельной комнате), нарушение которого, как правило, приводит к аварийной остановке процесса (вызов милиции). Каждому процессу по возможности предоставляются свои дополнительные ресурсы (каждая семья предпочитает иметь собственный холодильник). Тем не менее для решения некоторых задач (приготовление праздничного стола на всю квартиру) процессы могут объединять свои усилия. В настоящей лекции описываются причины взаимодействия процессов, способы их

взаимодействия и возникающие при этом проблемы (попробуйте отремонтировать общую квартиру так, чтобы жильцы не перессорились друг с другом).

Взаимодействующие процессы

Для достижения поставленной цели различные процессы (возможно, даже принадлежащие разным пользователям) могут исполняться псевдопараллельно на одной вычислительной системе или параллельно на разных вычислительных системах, взаимодействуя между собой.

Для чего процессам нужно заниматься совместной деятельностью? Какие существуют причины для их кооперации?

- Повышение скорости работы. Пока один процесс ожидает наступления некоторого события (например, окончания операции ввода-вывода), другие могут заниматься полезной работой, направленной на решение общей задачи. В многопроцессорных вычислительных системах программа разбивается на отдельные кусочки, каждый из которых будет исполняться на своем процессоре.
- Совместное использование данных. Различные процессы могут, к примеру, работать с одной и той же динамической базой данных или с разделяемым файлом, совместно изменяя их содержимое.
- Модульная конструкция какой-либо системы. Типичным примером может служить микроядерный способ построения операционной системы, когда различные ее части представляют собой отдельные процессы, взаимодействующие путем передачи сообщений через микроядро.
- Наконец, это может быть необходимо просто для удобства работы пользователя, желающего, например, редактировать и отлаживать программу одновременно. В этой ситуации процессы редактора и отладчика должны уметь взаимодействовать друг с другом.

Процессы не могут взаимодействовать, не общаясь, то есть не обмениваясь информацией. "Общение" процессов обычно приводит к изменению их поведения в зависимости от полученной информации. Если деятельность процессов остается неизменной при любой принятой ими информации, то это означает, что они на самом деле в "общении" не нуждаются. Процессы, которые влияют на поведение друг друга путем обмена информацией, принято называть кооперативными или взаимодействующими процессами, в отличие от независимых процессов, не оказывающих друг на друга никакого воздействия.

Различные процессы в вычислительной системе изначально представляют собой обособленные сущности. Работа одного процесса не должна приводить к нарушению работы другого процесса. Для этого, в частности, разделены их адресные пространства и системные ресурсы, и для обеспечения корректного взаимодействия процессов требуются специальные средства и действия операционной системы. Нельзя просто поместить значение, вычисленное в одном процессе, в область памяти, соответствующую переменной в другом процессе, не предприняв каких-либо дополнительных усилий. Давайте рассмотрим основные аспекты организации совместной работы процессов.

Категории средств обмена информацией

Процессы могут взаимодействовать друг с другом, только обмениваясь информацией. По объему передаваемой информации и степени возможного воздействия на поведение другого процесса все средства такого обмена можно разделить на три категории.

1. Сигнальные. Передается минимальное количество информации – один бит, "да" или "нет". Используются, как правило, для извещения процесса о наступлении какого-либо события. Степень воздействия на поведение процесса, получившего информацию, минимальна. Все зависит от того, знает ли он, что означает полученный сигнал, надо ли на него реагировать и каким образом. Неправильная реакция на сигнал или его игнорирование могут привести к трагическим последствиям.
2. Канальные. "Общение" процессов происходит через линии связи, предоставленные операционной системой, и напоминает общение людей по телефону, с помощью записок, писем или объявлений. Объем передаваемой информации в единицу времени ограничен пропускной способностью линий связи. С увеличением количества информации возрастает и возможность влияния на поведение другого процесса.
3. Разделяемая память. Два или более процессов могут совместно использовать некоторую область адресного пространства. Созданием разделяемой памяти занимается операционная система (если, конечно, ее об этом попросят). "Общение" процессов напоминает совместное проживание студентов в одной комнате общежития. Возможность обмена информацией максимальна, как, впрочем, и влияние на поведение другого процесса, но требует повышенной осторожности (если вы переложили на другое место вещи вашего соседа по комнате, а часть из них еще и выбросили). Использование разделяемой памяти для передачи/получения информации осуществляется с помощью средств обычных языков программирования, в то время как сигнальным и канальным средствам коммуникации для этого необходимы специальные системные вызовы. Разделяемая память представляет собой наиболее быстрый способ взаимодействия процессов в одной вычислительной системе.

Логическая организация механизма передачи информации

При рассмотрении любого из средств коммуникации нас будет интересовать не их физическая реализация (общая шина данных, прерывания, аппаратно разделяемая память и т. д.), а логическая, определяющая в конечном счете механизм их использования. Некоторые важные аспекты логической реализации являются общими для всех категорий средств связи, некоторые относятся к отдельным категориям. Давайте кратко охарактеризуем основные вопросы, требующие разъяснения при изучении того или иного способа обмена информацией.

Как устанавливается связь?

Могу ли я использовать средство связи непосредственно для обмена информацией сразу после создания процесса или первоначально необходимо предпринять определенные действия для инициализации обмена? Например, для использования общей памяти различными процессами потребуется специальное обращение к операционной системе, которая выделит необходимую область адресного пространства. Но для передачи сигнала от одного процесса к другому никакая инициализация не нужна. В то же время передача информации по линиям связи может потребовать первоначального резервирования такой линии для процессов, желающих обменяться информацией.

К этому же вопросу тесно примыкает вопрос о способе адресации при использовании средства связи. Если я передаю некоторую информацию, я должен указать, куда я ее передаю. Если я желаю получить некоторую информацию, то мне нужно знать, откуда я могу ее получить.

Различают два способа адресации: прямую и непрямую. В случае прямой адресации взаимодействующие процессы непосредственно общаются друг с другом, при каждой операции обмена данными явно указывая имя или номер процесса, которому информация предназначена или от которого она должна быть получена. Если и процесс, от которого данные исходят, и процесс, принимающий данные, указывают имена своих партнеров по взаимодействию, то такая схема адресации называется симметричной прямой адресацией. Ни один другой процесс не может вмешаться в процедуру симметричного прямого общения двух процессов, перехватить посланные или подменить ожидаемые данные. Если только один из взаимодействующих процессов, например передающий, указывает имя своего партнера по кооперации, а второй процесс в качестве возможного партнера рассматривает любой процесс в системе, например ожидает получения информации от произвольного источника, то такая схема адресации называется асимметричной прямой адресацией.

При непрямой адресации данные помещаются передающим процессом в некоторый промежуточный объект для хранения данных, имеющий свой адрес, откуда они могут быть затем изъяты каким-либо другим процессом. Примером такого объекта может служить обычная доска объявлений или рекламная газета. При этом передающий процесс не знает, как именно идентифицируется процесс, который получит информацию, а принимающий процесс не имеет представления об идентификаторе процесса, от которого он должен ее получить.

При использовании прямой адресации связь между процессами в классической операционной системе устанавливается автоматически, без дополнительных инициализирующих действий. Единственное, что нужно для использования средства связи, – это знать, как идентифицируются процессы, участвующие в обмене данными.

При использовании непрямой адресации инициализация средства связи может и не требоваться. Информация, которой должен обладать процесс для взаимодействия с другими процессами, – это некий идентификатор промежуточного объекта для хранения данных, если он, конечно, не является единственным и неповторимым в вычислительной системе для всех процессов.

Информационная валентность процессов и средств связи

Следующий важный вопрос – это вопрос об информационной валентности связи. Слово "валентность" здесь использовано по аналогии с химией. Сколько процессов может быть одновременно ассоциировано с конкретным средством связи? Сколько таких средств связи может быть задействовано между двумя процессами?

Понятно, что при прямой адресации только одно фиксированное средство связи может быть задействовано для обмена данными между двумя процессами, и только эти два процесса могут быть ассоциированы с ним. При непрямой адресации может существовать более двух процессов, использующих один и тот же объект для данных, и более одного объекта может быть использовано двумя процессами.

К этой же группе вопросов следует отнести и вопрос о направленности связи. Является ли связь однонаправленной или двунаправленной? Под однонаправленной связью мы будем понимать связь, при которой каждый процесс, ассоциированный с ней, может использовать средство связи либо только для приема информации, либо только для ее передачи. При двунаправленной связи каждый процесс, участвующий в общении, может

использовать связь и для приема, и для передачи данных. В коммуникационных системах принято называть однонаправленную связь симплексной, двунаправленную связь с поочередной передачей информации в разных направлениях – полудуплексной, а двунаправленную связь с возможностью одновременной передачи информации в разных направлениях – дуплексной. Прямая и непрямая адресация не имеет непосредственного отношения к направленности связи.

Особенности передачи информации с помощью линий связи

Как уже говорилось выше, передача информации между процессами посредством линий связи является достаточно безопасной по сравнению с использованием разделяемой памяти и более информативной по сравнению с сигнальными средствами коммуникации. Кроме того, разделяемая память не может быть использована для связи процессов, функционирующих на различных вычислительных системах. Возможно, именно поэтому каналы связи из средств коммуникации процессов получили наибольшее распространение. Коснемся некоторых вопросов, связанных с логической реализацией канальных средств коммуникации.

Буферизация

Может ли линия связи сохранять информацию, переданную одним процессом, до ее получения другим процессом или помещения в промежуточный объект? Каков объем этой информации? Иными словами, речь идет о том, обладает ли канал связи буфером и каков объем этого буфера. Здесь можно выделить три принципиальных варианта.

1. Буфер нулевой емкости или отсутствует. Никакая информация не может сохраняться на линии связи. В этом случае процесс, посылающий информацию, должен ожидать, пока процесс, принимающий информацию, не соблаговолит ее получить, прежде чем заниматься своими дальнейшими делами (в реальности этот случай никогда не реализуется).
2. Буфер ограниченной емкости. Размер буфера равен n, то есть линия связи не может хранить до момента получения более чем n единиц информации. Если в момент передачи данных в буфере хватает места, то передающий процесс не должен ничего ожидать. Информация просто копируется в буфер. Если же в момент передачи данных буфер заполнен или места недостаточно, то необходимо задержать работу процесса отправителя до появления в буфере свободного пространства.
3. Буфер неограниченной емкости. Теоретически это возможно, но практически вряд ли реализуемо. Процесс, посылающий информацию, никогда не ждет окончания ее передачи и приема другим процессом.

При использовании канального средства связи с непрямой адресацией под емкостью буфера обычно понимается количество информации, которое может быть помещено в промежуточный объект для хранения данных.

Поток ввода/вывода и сообщения

Существует две модели передачи данных по каналам связи – поток ввода-вывода и сообщения. При передаче данных с помощью потоковой модели операции передачи/приема информации вообще не интересуются содержимым данных. Процесс, прочитавший 100 байт из линии связи, не знает и не может знать, были ли они переданы одновременно, т. е. одним куском или порциями по 20 байт, пришли они от одного процесса или от разных. Данные представляют собой простой поток байтов, без какой-

либо их интерпретации со стороны системы. Примерами потоковых каналов связи могут служить pipe и FIFO, описанные ниже.

Одним из наиболее простых способов передачи информации между процессами по линиям связи является передача данных через pipe (канал, трубу или, как его еще называют в литературе, конвейер). Представим себе, что у нас есть некоторая труба в вычислительной системе, в один из концов которой процессы могут "сливать" информацию, а из другого конца принимать полученный поток. Такой способ реализует потоковую модель ввода/вывода. Информацией о расположении трубы в операционной системе обладает только процесс, создавший ее. Этой информацией он может поделиться исключительно со своими наследниками – процессами-детьми и их потомками. Поэтому использовать pipe для связи между собой могут только родственные процессы, имеющие общего предка, создавшего данный канал связи.

Если разрешить процессу, создавшему трубу, сообщать о ее местонахождении в системе другим процессам, сделав вход и выход трубы каким-либо образом видимыми для всех остальных, например, зарегистрировав ее в операционной системе под определенным именем, мы получим объект, который принято называть FIFO или именованный pipe. Именованный pipe может использоваться для организации связи между любыми процессами в системе.

В модели сообщений процессы налагают на передаваемые данные некоторую структуру. Весь поток информации они разделяют на отдельные сообщения, вводя между данными, по крайней мере, границы сообщений. Примером границ сообщений являются точки между предложениями в сплошном тексте или границы абзаца. Кроме того, к передаваемой информации могут быть присоединены указания на то, кем конкретное сообщение было послано и для кого оно предназначено. Примером указания отправителя могут служить подписи под эпиграфами в книге. Все сообщения могут иметь одинаковый фиксированный размер или могут быть переменной длины. В вычислительных системах используются разнообразные средства связи для передачи сообщений: очереди сообщений, sockets (гнезда) и т. д.

И потоковые линии связи, и каналы сообщений всегда имеют буфер конечной длины. Когда мы будем говорить о емкости буфера для потоков данных, мы будем измерять ее в байтах. Когда мы будем говорить о емкости буфера для сообщений, мы будем измерять ее в сообщениях.

Надежность средств связи

Одним из существенных вопросов при рассмотрении всех категорий средств связи является вопрос об их надежности.

Мы будем называть способ коммуникации надежным, если при обмене данными выполняются четыре условия.

1. Не происходит потери информации.
2. Не происходит повреждения информации.
3. Не появляется лишней информации.
4. Не нарушается порядок данных в процессе обмена.

Очевидно, что передача данных через разделяемую память является надежным способом связи. То, что мы сохранили в разделяемой памяти, будет считано другими процессами в

первозданном виде, если, конечно, не произойдет сбоя в питании компьютера. Для других средств коммуникации, как видно из приведенных выше примеров, это не всегда верно.

Каким образом в вычислительных системах пытаются бороться с ненадежностью коммуникаций? Давайте рассмотрим возможные варианты на примере обмена данными через линию связи с помощью сообщений. Для обнаружения повреждения информации будем снабжать каждое передаваемое сообщение некоторой контрольной суммой, вычисленной по посланной информации. При приеме сообщения контрольную сумму будем вычислять заново и проверять ее соответствие пришедшему значению. Если данные не повреждены (контрольные суммы совпадают), то подтвердим правильность их получения. Если данные повреждены (контрольные суммы не совпадают), то сделаем вид, что сообщение к нам не поступило. Вместо контрольной суммы можно использовать специальное кодирование передаваемых данных с помощью кодов, исправляющих ошибки. Такое кодирование позволяет при числе искажений информации, не превышающем некоторого значения, восстановить первоначальные неискаженные данные. Если по прошествии некоторого интервала времени подтверждение правильности полученной информации не придет на передающий конец линии связи, будем считать информацию утерянной и пошлем ее повторно. Для того чтобы избежать двойного получения одной и той же информации, на приемном конце линии связи должен осуществляться соответствующий контроль. Для гарантии правильного порядка получения сообщений будем их нумеровать. При приеме сообщения с номером, не соответствующим ожидаемому, поступаем с ним как с утерянным и ждем сообщения с правильным номером.

Подобные действия могут быть возложены:

- на операционную систему;
- на процессы, обменивающиеся данными;
- совместно на систему и процессы, разделяя их ответственность. Операционная система может обнаруживать ошибки при передаче данных и извещать об этом взаимодействующие процессы для принятия ими решения о дальнейшем поведении.

Как завершается связь?

Наконец, важным вопросом при изучении средств обмена данными является вопрос прекращения обмена. Здесь нужно выделить два аспекта: требуются ли от процесса какие-либо специальные действия по прекращению использования средства коммуникации и влияет ли такое прекращение на поведение других процессов. Для способов связи, которые не подразумевали никаких инициализирующих действий, обычно ничего специального для окончания взаимодействия предпринимать не надо. Если же установление связи требовало некоторой инициализации, то, как правило, при ее завершении бывает необходимо выполнить ряд операций, например сообщить операционной системе об освобождении выделенного связного ресурса.

Если кооперативные процессы прекращают взаимодействие согласованно, то такое прекращение не влияет на их дальнейшее поведение. Иная картина наблюдается при несогласованном окончании связи одним из процессов. Если какой-либо из взаимодействующих процессов, не завершивших общение, находится в этот момент в состоянии ожидания получения данных либо попадает в такое состояние позже, то операционная система обязана предпринять некоторые действия для того, чтобы исключить вечное блокирование этого процесса. Обычно это либо прекращение работы

ожидающего процесса, либо его извещение о том, что связи больше нет (например, с помощью передачи заранее определенного сигнала).

Нити исполнения

Рассмотренные выше аспекты логической реализации относятся к средствам связи, ориентированным на организацию взаимодействия различных процессов. Однако усилия, направленные на ускорение решения задач в рамках классических операционных систем, привели к появлению совершенно иных механизмов, к изменению самого понятия "процесс".

В свое время внедрение идеи мультипрограммирования позволило повысить пропускную способность компьютерных систем, т. е. уменьшить среднее время ожидания результатов работы процессов. Но любой отдельно взятый процесс в мультипрограммной системе никогда не может быть выполнен быстрее, чем при работе в однопрограммном режиме на том же вычислительном комплексе. Тем не менее, если алгоритм решения задачи обладает определенным внутренним параллелизмом, мы могли бы ускорить его работу, организовав взаимодействие нескольких процессов. Рассмотрим следующий пример. Пусть у нас есть следующая программа на псевдоязыке программирования:

```
Ввести массив a
Ввести массив b
Ввести массив c
a = a + b
c = a + c
Вывести массив c
```

При выполнении такой программы в рамках одного процесса этот процесс четырежды будет блокироваться, ожидая окончания операций ввода-вывода. Но наш алгоритм обладает внутренним параллелизмом. Вычисление суммы массивов a + b можно было бы выполнять параллельно с ожиданием окончания операции ввода массива c.

```
Ввести массив a
Ожидание окончания операции ввода
Ввести массив b
Ожидание окончания операции ввода
Ввести массив c
Ожидание окончания операции ввода a = a + b
c = a + c
Вывести массив c
Ожидание окончания операции вывода
```

Такое совмещение операций по времени можно было бы реализовать, используя два взаимодействующих процесса. Для простоты будем полагать, что средством коммуникации между ними служит разделяемая память. Тогда наши процессы могут выглядеть следующим образом.

Процесс 1	Процесс 2
1. Ввести массив a	1. Ожидание ввода массивов a и b
2. Ожидание окончания операции ввода	2. a = a + b
3. Ввести массив b	
4. Ожидание окончания операции ввода	
5. Ввести массив c	
6. Ожидание окончания операции ввода	
7. c = a + c	

8. Вывести массив c
9. Ожидание окончания операции вывода

Казалось бы, мы предложили конкретный способ ускорения решения задачи. Однако в действительности дело обстоит не так просто. Второй процесс должен быть создан, оба процесса должны сообщить операционной системе, что им необходима память, которую они могли бы разделить с другим процессом, и, наконец, нельзя забывать о переключении контекста. Поэтому реальное поведение процессов будет выглядеть примерно так.

Процесс 1	Процесс 2
1. Создать процесс 2	
Переключение контекста	
	1. Выделение общей памяти
	2. Ожидание ввода массивов a и b
Переключение контекста	
2. Выделение общей памяти	
3. Ввести массив a	
4. Ожидание окончания операции ввода	
5. Ввести массив b	
6. Ожидание окончания операции ввода	
7. Ввести массив c	
8. Ожидание окончания операции ввода	
Переключение контекста	
	3. a = a + b
Переключение контекста	
9. c = a + c	
10. Вывести массив c	
11. Ожидание окончания операции вывода	

Очевидно, что мы можем не только не выиграть во времени при решении задачи, но даже и проиграть, так как временные потери на создание процесса, выделение общей памяти и переключение контекста могут превысить выигрыш, полученный за счет совмещения операций. Для того чтобы реализовать нашу идею, введем новую абстракцию внутри понятия "процесс" – нить исполнения или просто нить (в англоязычной литературе используется термин thread). Нити процесса разделяют его программный код, глобальные переменные и системные ресурсы, но каждая нить имеет собственный программный счетчик, свое содержимое регистров и свой стек. Теперь процесс представляется как совокупность взаимодействующих нитей и выделенных ему ресурсов. Процесс, содержащий всего одну нить исполнения, идентичен процессу в том смысле, который мы употребляли ранее. Для таких процессов мы в дальнейшем будем использовать термин "традиционный процесс". Иногда нити называют облегченными процессами или мини-процессами, так как во многих отношениях они подобны традиционным процессам. Нити, как и процессы, могут порождать нити-потомки, правда, только внутри своего процесса, и переходить из одного состояния в другое. Состояния нитей аналогичны состояниям традиционных процессов. Из состояния рождение процесс приходит содержащим всего одну нить исполнения. Другие нити процесса будут являться потомками этой нити-прародительницы. Мы можем считать, что процесс находится в состоянии готовность, если хотя бы одна из его нитей находится в состоянии готовность и ни одна из нитей не находится в состоянии исполнение. Мы можем считать, что процесс находится в состоянии исполнение, если одна из его нитей находится в состоянии исполнение. Процесс будет находиться в состоянии ожидание, если все его нити находятся в состоянии ожидание. Наконец, процесс находится в состоянии закончил исполнение, если все его

нити находятся в состоянии закончила исполнение. Пока одна нить процесса заблокирована, другая нить того же процесса может выполняться. Нити разделяют процессор так же, как это делали традиционные процессы, в соответствии с рассмотренными алгоритмами планирования.

Поскольку нити одного процесса разделяют существенно больше ресурсов, чем различные процессы, то операции создания новой нити и переключения контекста между нитями одного процесса занимают значительно меньше времени, чем аналогичные операции для процессов в целом. Предложенная нами схема совмещения работы в терминах нитей одного процесса получает право на существование. В нитях нет необходимости осуществлять выделение общей памяти.

Различают операционные системы, поддерживающие нити на уровне ядра и на уровне библиотек. Все сказанное выше справедливо для операционных систем, поддерживающих нити на уровне ядра. В них планирование использования процессора происходит в терминах нитей, а управление памятью и другими системными ресурсами остается в терминах процессов. В операционных системах, поддерживающих нити на уровне библиотек пользователей, и планирование процессора, и управление системными ресурсами осуществляются в терминах процессов. Распределение использования процессора по нитям в рамках выделенного процессу временного интервала осуществляется средствами библиотеки. В подобных системах блокирование одной нити приводит к блокированию всего процесса, ибо ядро операционной системы не имеет представления о существовании нитей. По сути дела, в таких вычислительных системах просто имитируется наличие нитей исполнения.

Лекция: Синхронизация процессов. Тупики.

Interleaving, race condition и взаимоисключения

Давайте временно отвлечемся от операционных систем, процессов и нитей исполнения и поговорим о некоторых "активностях". Под активностями мы будем понимать последовательное выполнение ряда действий, направленных на достижение определенной цели. Активности могут иметь место в программном и техническом обеспечении, в обычной деятельности людей и животных. Мы будем разбивать активности на некоторые неделимые, или атомарные, операции. Например, активность "приготовление бутерброда" можно разбить на следующие атомарные операции:

1. Отрезать ломтик хлеба.
2. Отрезать ломтик колбасы.
3. Намазать ломтик хлеба маслом.
4. Положить ломтик колбасы на подготовленный ломтик хлеба.

Неделимые операции могут иметь внутренние невидимые действия (взять батон хлеба в левую руку, взять нож в правую руку, произвести отрезание). Мы же называем их неделимыми потому, что считаем выполняемыми за раз, без прерывания деятельности.

Пусть имеется две активности

```
P:   a b c
Q:   d e f
```

где a, b, c, d, e, f – атомарные операции. При последовательном выполнении активностей мы получаем такую последовательность атомарных действий:

```
PQ: a b c d e f
```

Что произойдет при исполнении этих активностей псевдопараллельно, в режиме разделения времени? Активности могут расслоиться на неделимые операции с различным чередованием, то есть может произойти то, что на английском языке принято называть словом interleaving. Возможные варианты чередования:

```
a b c d e f
a b d c e f
a b d e c f
a b d e f c
a d b c e f
......
d e f a b c
```

Атомарные операции активностей могут чередоваться всевозможными различными способами с сохранением порядка расположения внутри активностей. Так как псевдопараллельное выполнение двух активностей приводит к чередованию их неделимых операций, результат псевдопараллельного выполнения может отличаться от результата последовательного выполнения. Рассмотрим пример. Пусть у нас имеется две активности P и Q, состоящие из двух атомарных операций каждая:

```
P:   x=2              Q:   x=3
     y=x-1                  y=x+1
```

Что мы получим в результате их псевдопараллельного выполнения, если переменные x и y являются для активностей общими? Очевидно, что возможны четыре разных набора значений для пары (x, y): (3, 4), (2, 1), (2, 3) и (3, 2). . Мы будем говорить, что набор активностей (например, программ) детерминирован, если всякий раз при псевдопараллельном исполнении для одного и того же набора входных данных он дает одинаковые выходные данные. В противном случае он недетерминирован. Выше приведен пример недетерминированного набора программ. Понятно, что детерминированный набор активностей можно безбоязненно выполнять в режиме разделения времени. Для недетерминированного набора такое исполнение нежелательно.

Можно ли до получения результатов определить, является ли набор активностей детерминированным или нет? Для этого существуют достаточные условия Бернстайна. Изложим их применительно к программам с разделяемыми переменными.

Введем наборы входных и выходных переменных программы. Для каждой атомарной операции наборы входных и выходных переменных – это наборы переменных, которые атомарная операция считывает и записывает. Набор входных переменных программы R(P) (R от слова read) суть объединение наборов входных переменных для всех ее неделимых действий. Аналогично, набор выходных переменных программы W(P) (W от слова write) суть объединение наборов выходных переменных для всех ее неделимых действий. Например, для программы

```
P:   x=u+v
     y=x*w
```

получаем R(P) = {u, v, x, w}, W(P) = {x, y}. Заметим, что переменная x присутствует как в R(P), так и в W(P).

Теперь сформулируем условия Бернстайна.

Если для двух данных активностей P и Q:

1. пересечение W(P) и W(Q) пусто;
2. пересечение W(P) с R(Q) пусто;
3. пересечение R(P) и W(Q) пусто,

тогда выполнение P и Q детерминировано.

Если эти условия не соблюдены, возможно, параллельное выполнение P и Q детерминировано, а может быть, и нет.

Случай двух активностей естественным образом обобщается на их большее количество.

Условия Бернстайна информативны, но слишком жестки. По сути дела, они требуют практически невзаимодействующих процессов. А нам хотелось бы, чтобы детерминированный набор образовывали активности, совместно использующие информацию и обменивающиеся ею. Для этого нам необходимо ограничить число возможных чередований атомарных операций, исключив некоторые чередования с помощью механизмов синхронизации выполнения программ, обеспечив тем самым упорядоченный доступ программ к некоторым данным.

Про недетерминированный набор программ (и активностей вообще) говорят, что он имеет race condition (состояние гонки, состояние состязания). В приведенном выше примере процессы состязаются за вычисление значений переменных x и y.

Задачу упорядоченного доступа к разделяемым данным (устранение race condition) в том случае, когда нам не важна его очередность, можно решить, если обеспечить каждому процессу эксклюзивное право доступа к этим данным. Каждый процесс, обращающийся к разделяемым ресурсам, исключает для всех других процессов возможность одновременного общения с этими ресурсами, если это может привести к недетерминированному поведению набора процессов. Такой прием называется взаимоисключением (mutual exclusion). Если очередность доступа к разделяемым ресурсам важна для получения правильных результатов, то одними взаимоисключениями уже не обойтись, нужна взаимосинхронизация поведения программ.

Критическая секция

Важным понятием при изучении способов синхронизации процессов является понятие критической секции (critical section) программы. Критическая секция – это часть программы, исполнение которой может привести к возникновению race condition для определенного набора программ. Чтобы исключить эффект гонок по отношению к некоторому ресурсу, необходимо организовать работу так, чтобы в каждый момент времени только один процесс мог находиться в своей критической секции, связанной с этим ресурсом. Иными словами, необходимо обеспечить реализацию взаимоисключения для критических секций программ. Реализация взаимоисключения для критических секций программ с практической точки зрения означает, что по отношению к другим процессам, участвующим во взаимодействии, критическая секция начинает выполняться как атомарная операция. Давайте рассмотрим следующий пример, в котором псевдопараллельные взаимодействующие процессы представлены действиями различных студентов.
Здесь критический участок для каждого процесса – от операции "Обнаруживает, что хлеба нет" до операции "Возвращается в комнату" включительно. В результате отсутствия взаимоисключения мы из ситуации "Нет хлеба" попадаем в ситуацию "Слишком много

хлеба". Если бы этот критический участок выполнялся как атомарная операция – "Достает два батона хлеба", то проблема образования излишков была бы снята.

Таблица 5.1

Время	Студент 1	Студент 2	Студент 3
17-05	Приходит в комнату		
17-07	Обнаруживает,что хлеба нет		
17-09	Уходит в магазин		
17-11		Приходит в комнату	
17-13		Обнаруживает, что хлеба нет	
17-15		Уходит в магазин	
17-17			Приходит в комнату
17-19			Обнаруживает,что хлеба н
17-21			Уходит в магазин
17-23	Приходит в магазин		
17-25	Покупает 2 батона на всех		
17-27	Уходит из магазина		
17-29		Приходит в магазин	
17-31		Покупает 2 батона на всех	
17-33		Уходит из магазина	
17-35			Приходит в магазин
17-37			Покупает 2 батона на всех
17-39			Уходит из магазина
17-41	Возвращается в комнату		
17-43			
17-45			
17-47		Возвращается в комнату	
17-49			
17-51			
17-53			Возвращается в комнату

Сделать процесс добывания хлеба атомарной операцией можно было бы следующим образом: перед началом этого процесса закрыть дверь изнутри на засов и уходить добывать хлеб через окно, а по окончании процесса вернуться в комнату через окно и отодвинуть засов. Тогда пока один студент добывает хлеб, все остальные находятся в состоянии ожидания под дверью. Итак, для решения задачи необходимо, чтобы в том случае, когда процесс находится в своем критическом участке, другие процессы не могли войти в свои критические участки. Мы видим, что критический участок должен сопровождаться прологом (entry section) – "закрыть дверь изнутри на засов" – и эпилогом (exit section) – "отодвинуть засов", которые не имеют отношения к активности одиночного процесса. Во время выполнения пролога процесс должен, в частности, получить разрешение на вход в критический участок, а во время выполнения эпилога – сообщить другим процессам, что он покинул критическую секцию.

В общем случае структура процесса, участвующего во взаимодействии, может быть представлена следующим образом:

```
while (some condition) {
   entry section
      critical section
   exit section
      remainder section
}
```

Здесь под remainder section понимаются все атомарные операции, не входящие в критическую секцию.

Программные алгоритмы организации взаимодействия процессов

Требования, предъявляемые к алгоритмам:

1. задача должна быть решена чисто программным способом на обычной машине, не имеющей специальных команд взаимоисключения. При этом предполагается, что основные инструкции языка программирования (такие примитивные инструкции, как load, store, test) являются атомарными операциями.
2. Не должно существовать никаких предположений об относительных скоростях выполняющихся процессов или числе процессоров, на которых они исполняются.
3. Если процесс Pi исполняется в своем критическом участке, то не существует никаких других процессов, которые исполняются в соответствующих критических секциях. Это условие получило название условия взаимоисключения (mutual exclusion).
4. Процессы, которые находятся вне своих критических участков и не собираются входить в них, не могут препятствовать другим процессам входить в их собственные критические участки. Если нет процессов в критических секциях и имеются процессы, желающие войти в них, то только те процессы, которые не исполняются в remainder section, должны принимать решение о том, какой процесс войдет в свою критическую секцию. Такое решение не должно приниматься бесконечно долго. Это условие получило название условия прогресса (progress).
5. Не должно возникать неограниченно долгого ожидания для входа одного из процессов в свой критический участок. От того момента, когда процесс запросил разрешение на вход в критическую секцию, и до того момента, когда он это разрешение получил, другие процессы могут пройти через свои критические участки лишь ограниченное число раз. Это условие получило название условия ограниченного ожидания (bound waiting).

Надо заметить, что описание соответствующего алгоритма в нашем случае означает описание способа организации пролога и эпилога для критической секции.

Запрет прерываний

Наиболее простым решением поставленной задачи является следующая организация пролога и эпилога:

```
while (some condition) {
   запретить все прерывания
      critical section
   разрешить все прерывания
      remainder section
}
```

Поскольку выход процесса из состояния исполнение без его завершения осуществляется по прерыванию, внутри критической секции никто не может вмешаться в его работу. Однако такое решение может иметь далеко идущие последствия, поскольку позволяет процессу пользователя разрешать и запрещать прерывания во всей вычислительной системе. Допустим, что пользователь случайно или по злому умыслу запретил прерывания в системе и зациклил или завершил свой процесс. Без перезагрузки системы в такой ситуации не обойтись.

Тем не менее запрет и разрешение прерываний часто применяются как пролог и эпилог к критическим секциям внутри самой операционной системы, например при обновлении содержимого PCB.

Переменная-замок

В качестве следующей попытки решения задачи для пользовательских процессов рассмотрим другое предложение. Возьмем некоторую переменную, доступную всем процессам, с начальным значением равным 0. Процесс может войти в критическую секцию только тогда, когда значение этой переменной-замка равно 0, одновременно изменяя ее значение на 1 – закрывая замок. При выходе из критической секции процесс сбрасывает ее значение в 0 – замок открывается (как в случае с покупкой хлеба студентами в разделе "Критическая секция").

```
shared int lock = 0;
/* shared означает, что */
/* переменная является разделяемой */

while (some condition) {
   while(lock); lock = 1;
      critical section
   lock = 0;
      remainder section
}
```

К сожалению, при внимательном рассмотрении мы видим, что такое решение не удовлетворяет условию взаимоисключения, так как действие while(lock); lock = 1; не является атомарным. Допустим, процесс P0 протестировал значение переменной lock и принял решение двигаться дальше. В этот момент, еще до присвоения переменной lock значения 1, планировщик передал управление процессу P1. Он тоже изучает содержимое переменной lock и тоже принимает решение войти в критический участок. Мы получаем два процесса, одновременно выполняющих свои критические секции.

Строгое чередование

Попробуем решить задачу сначала для двух процессов. Очередной подход будет также использовать общую для них обоих переменную с начальным значением 0. Только теперь она будет играть не роль замка для критического участка, а явно указывать, кто может следующим войти в него. Для i-го процесса это выглядит так:

```
shared int turn = 0;

while (some condition) {
   while(turn != i);
      critical section
   turn = 1-i;
      remainder section
}
```

Очевидно, что взаимоисключение гарантируется, процессы входят в критическую секцию строго по очереди: P0, P1, P0, P1, P0, ... Но наш алгоритм не удовлетворяет условию прогресса. Например, если значение turn равно 1, и процесс P0 готов войти в критический участок, он не может сделать этого, даже если процесс P1 находится в remainder section.

Флаги готовности

Недостаток предыдущего алгоритма заключается в том, что процессы ничего не знают о состоянии друг друга в текущий момент времени. Давайте попробуем исправить эту ситуацию. Пусть два наших процесса имеют разделяемый массив флагов готовности входа процессов в критический участок

```
shared int ready[2] = {0, 0};
```

Когда i-й процесс готов войти в критическую секцию, он присваивает элементу массива ready[i] значение равное 1. После выхода из критической секции он, естественно, сбрасывает это значение в 0. Процесс не входит в критическую секцию, если другой процесс уже готов к входу в критическую секцию или находится в ней.

```
while (some condition) {
   ready[i] = 1;
   while(ready[1-i]);
      critical section
   ready[i] = 0;
      remainder section
}
```

Полученный алгоритм обеспечивает взаимоисключение, позволяет процессу, готовому к входу в критический участок, войти в него сразу после завершения эпилога в другом процессе, но все равно нарушает условие прогресса. Пусть процессы практически одновременно подошли к выполнению пролога. После выполнения присваивания ready[0]=1 планировщик передал процессор от процесса 0 процессу 1, который также выполнил присваивание ready[1]=1. После этого оба процесса бесконечно долго ждут друг друга на входе в критическую секцию. Возникает ситуация, которую принято называть тупиковой (deadlock).

Алгоритм Петерсона

Первое решение проблемы, удовлетворяющее всем требованиям и использующее идеи ранее рассмотренных алгоритмов, было предложено датским математиком Деккером (Dekker). В 1981 году Петерсон (Peterson) предложил более изящное решение. Пусть оба процесса имеют доступ к массиву флагов готовности и к переменной очередности.

```
shared int ready[2] = {0, 0};
shared int turn;
while (some condition) {
   ready[i] = 1;
   turn =1-i;
   while(ready[1-i] && turn == 1-i);
      critical section
   ready[i] = 0;
      remainder section
}
```

При исполнении пролога критической секции процесс Pi заявляет о своей готовности выполнить критический участок и одновременно предлагает другому процессу приступить к его выполнению. Если оба процесса подошли к прологу практически одновременно, то они оба объявят о своей готовности и предложат выполняться друг другу. При этом одно из предложений всегда следует после другого. Тем самым работу в критическом участке продолжит процесс, которому было сделано последнее предложение.

Давайте докажем, что все пять наших требований к алгоритму действительно удовлетворяются.

Удовлетворение требований 1 и 2 очевидно.

Докажем выполнение условия взаимоисключения методом от противного. Пусть оба процесса одновременно оказались внутри своих критических секций. Заметим, что процесс Pi может войти в критическую секцию, только если ready[1-i] == 0 или turn == i. Заметим также, что если оба процесса выполняют свои критические секции одновременно, то значения флагов готовности для обоих процессов совпадают и равны 1. Могли ли оба процесса войти в критические секции из состояния, когда они оба одновременно находились в процессе выполнения цикла while? Нет, так как в этом случае переменная turn должна была бы одновременно иметь значения 0 и 1 (когда оба процесса выполняют цикл, значения переменных измениться не могут). Пусть процесс P0 первым вошел в критический участок, тогда процесс P1 должен был выполнить перед вхождением в цикл while по крайней мере один предваряющий оператор (turn = 0;). Однако после этого он не может выйти из цикла до окончания критического участка процесса P0, так как при входе в цикл ready[0] == 1 и turn == 0, и эти значения не могут измениться до тех пор, пока процесс P0 не покинет свой критический участок. Мы пришли к противоречию. Следовательно, имеет место взаимоисключение.

Докажем выполнение условия прогресса. Возьмем, без ограничения общности, процесс P0. Заметим, что он не может войти в свою критическую секцию только при совместном выполнении условий ready[1] == 1 и turn == 1. Если процесс P1 не готов к выполнению критического участка, то ready[1] == 0, и процесс P0 может осуществить вход. Если процесс P1 готов к выполнению критического участка, то ready[1] == 1 и переменная turn имеет значение 0 либо 1, позволяя процессу P0 либо процессу P1 начать выполнение критической секции. Если процесс P1 завершил выполнение критического участка, то он сбросит свой флаг готовности ready[1] == 0, разрешая процессу P0 приступить к выполнению критической работы. Таким образом, условие прогресса выполняется.

Отсюда же вытекает выполнение условия ограниченного ожидания. Так как в процессе ожидания разрешения на вход процесс P0 не изменяет значения переменных, он сможет начать исполнение своего критического участка после не более чем одного прохода по критической секции процесса P1.

Алгоритм булочной (Bakery algorithm)

Алгоритм Петерсона дает нам решение задачи корректной организации взаимодействия двух процессов. Давайте рассмотрим теперь соответствующий алгоритм для n взаимодействующих процессов, который получил название алгоритм булочной, хотя применительно к нашим условиям его следовало бы скорее назвать алгоритм регистратуры в поликлинике. Основная его идея выглядит так. Каждый вновь прибывающий клиент (он же процесс) получает талончик на обслуживание с номером. Клиент с наименьшим номером на талончике обслуживается следующим. К сожалению, из-за неатомарности операции вычисления следующего номера алгоритм булочной не гарантирует, что у всех процессов будут талончики с разными номерами. В случае равенства номеров на талончиках у двух или более клиентов первым обслуживается клиент с меньшим значением имени (имена можно сравнивать в лексикографическом порядке). Разделяемые структуры данных для алгоритма – это два массива

```
shared enum {false, true} choosing[n];
shared int number[n];
```

Изначально элементы этих массивов инициируются значениями false и 0 соответственно. Введем следующие обозначения

```
(a,b) < (c,d), если a < c
или если a == c и b < d
max(a0, a1, ...., an) – это число k такое, что
k >= ai для всех i = 0, ...,n
```

Структура процесса Pi для алгоритма булочной приведена ниже

```
while (some condition) {
   choosing[i] = true;
   number[i] = max(number[0], ...,
                   number[n-1]) + 1;
   choosing[i] = false;
   for(j = 0; j < n; j++){
      while(choosing[j]);
      while(number[j] != 0 && (number[j],j) <
         (number[i],i));
   }
      critical section
   number[i] = 0;
      remainder section
}
```

Доказательство того, что этот алгоритм удовлетворяет условиям 1 – 5, выполните самостоятельно в качестве упражнения.

Аппаратная поддержка взаимоисключений

Наличие аппаратной поддержки взаимоисключений позволяет упростить алгоритмы и повысить их эффективность точно так же, как это происходит и в других областях программирования. Мы уже обращались к общепринятому hardware для решения задачи реализации взаимоисключений, когда говорили об использовании механизма запрета/разрешения прерываний.

Многие вычислительные системы помимо этого имеют специальные команды процессора, которые позволяют проверить и изменить значение машинного слова или поменять местами значения двух машинных слов в памяти, выполняя эти действия как атомарные операции. Давайте обсудим, как концепции таких команд могут использоваться для реализации взаимоисключений.

Команда Test-and-Set (проверить и присвоить 1)

О выполнении команды Test-and-Set, осуществляющей проверку значения логической переменной с одновременной установкой ее значения в 1, можно думать как о выполнении функции

```
int Test_and_Set (int *target){
   int tmp = *target;
   *target = 1;
   return tmp;
}
```

С использованием этой атомарной команды мы можем модифицировать наш алгоритм для переменной-замка, так чтобы он обеспечивал взаимоисключения

```
shared int lock = 0;

while (some condition) {
   while(Test_and_Set(&lock));
      critical section
   lock = 0;
      remainder section
}
```

К сожалению, даже в таком виде полученный алгоритм не удовлетворяет условию ограниченного ожидания для алгоритмов. Подумайте, как его следует изменить для соблюдения всех условий.

Команда Swap (обменять значения)

Выполнение команды Swap, обменивающей два значения, находящихся в памяти, можно проиллюстрировать следующей функцией:

```
void Swap (int *a, int *b){
   int tmp = *a;
   *a = *b;
   *b = tmp;
}
```

Применяя атомарную команду Swap, мы можем реализовать предыдущий алгоритм, введя дополнительную логическую переменную key, локальную для каждого процесса:

```
shared int lock = 0;
int key;

while (some condition) {
   key = 1;
   do Swap(&lock,&key);
   while (key);
      critical section
   lock = 0;
      remainder section
}
```

Механизмы синхронизации потоков

Существуют серьезные недостатки у алгоритмов, построенных средствами обычных языков программирования. Допустим, что в вычислительной системе находятся два взаимодействующих процесса: один из них – H – с высоким приоритетом, другой – L – с низким приоритетом. Пусть планировщик устроен так, что процесс с высоким приоритетом вытесняет низкоприоритетный процесс всякий раз, когда он готов к исполнению, и занимает процессор на все время своего CPU burst (если не появится процесс с еще большим приоритетом). Тогда в случае, если процесс L находится в своей критической секции, а процесс H, получив процессор, подошел ко входу в критическую область, мы получаем тупиковую ситуацию. Процесс H не может войти в критическую область, находясь в цикле, а процесс L не получает управления, чтобы покинуть критический участок.

Для того чтобы не допустить возникновения подобных проблем, были разработаны различные механизмы синхронизации более высокого уровня.

Семафоры

Одним из первых механизмов, предложенных для синхронизации поведения процессов, стали семафоры, концепцию которых описал Дейкстра (Dijkstra) в 1965 году.

Концепция семафоров

Семафор представляет собой целую переменную, принимающую неотрицательные значения, доступ любого процесса к которой, за исключением момента ее инициализации, может осуществляться только через две атомарные операции: P (от датского слова proberen – проверять) и V (от verhogen – увеличивать). Классическое определение этих операций выглядит следующим образом:

```
P(S):   пока S == 0 процесс блокируется;
        S = S - 1;
V(S):   S = S + 1;
```

Эта запись означает следующее: при выполнении операции P над семафором S сначала проверяется его значение. Если оно больше 0, то из S вычитается 1. Если оно меньше или равно 0, то процесс блокируется до тех пор, пока S не станет больше 0, после чего из S вычитается 1. При выполнении операции V над семафором S к его значению просто прибавляется 1. В момент создания семафор может быть инициализирован любым неотрицательным значением.

Подобные переменные-семафоры могут с успехом применяться для решения различных задач организации взаимодействия процессов. В ряде языков программирования они были непосредственно введены в синтаксис языка (например, в ALGOL-68), в других случаях реализуются с помощью специальных системных вызовов. Соответствующая целая переменная располагается внутри адресного пространства ядра операционной системы. Операционная система обеспечивает атомарность операций P и V, используя, например, метод запрета прерываний на время выполнения соответствующих системных вызовов. Если при выполнении операции P заблокированными оказались несколько процессов, то порядок их разблокирования может быть произвольным, например, FIFO.

Решение проблемы producer-consumer с помощью семафоров

Одной из типовых задач, требующих организации взаимодействия процессов, является задача producer-consumer (производитель-потребитель). Пусть два процесса обмениваются информацией через буфер ограниченного размера. Производитель закладывает информацию в буфер, а потребитель извлекает ее оттуда. На этом уровне деятельность потребителя и производителя можно описать следующим образом.

```
Producer:  while(1) {
              produce_item;
              put_item;
           }
Consumer:  while(1) {
              get_item;
              consume_item;
           }
```

Если буфер заполнен, то производитель должен ждать, пока в нем появится место, чтобы положить туда новую порцию информации. Если буфер пуст, то потребитель должен дожидаться нового сообщения. Как можно реализовать эти условия с помощью семафоров? Возьмем три семафора: empty, full и mutex. Семафор full будем использовать для гарантии того, что потребитель будет ждать, пока в буфере появится информация. Семафор empty будем использовать для организации ожидания производителя при заполненном буфере, а семафор mutex – для организации взаимоисключения на критических участках, которыми являются действия put_item и get_item (операции "положить информацию" и "взять информацию" не могут пересекаться, так как в этом случае возникнет опасность искажения информации). Тогда решение задачи на С-подобном языке выглядит так:

```
 Semaphore mutex = 1;
   Semaphore empty = N; /* где N – емкость буфера*/
   Semaphore full = 0;
```

Producer:

```
   while(1) {
      produce_item;
      P(empty);
      P(mutex);
      put_item;
      V(mutex);
      V(full);
   }
```

Consumer:

```
   while(1) {
      P(full);
      P(mutex);
      get_item;
      V(mutex);
      V(empty);
      consume_item;
   }
```

Легко убедиться, что это действительно корректное решение поставленной задачи. Попутно заметим, что семафоры использовались здесь для достижения двух целей: организации взаимоисключения на критическом участке и взаимосинхронизации скорости работы процессов.

Мониторы

Хотя решение задачи producer-consumer с помощью семафоров выглядит достаточно изящно, программирование с их использованием требует повышенной осторожности и внимания, чем отчасти напоминает программирование на языке Ассемблера. Допустим, что в рассмотренном примере мы случайно поменяли местами операции P, сначала выполнив операцию для семафора mutex, а уже затем для семафоров full и empty. Допустим теперь, что потребитель, войдя в свой критический участок (mutex сброшен), обнаруживает, что буфер пуст. Он блокируется и начинает ждать появления сообщений. Но производитель не может войти в критический участок для передачи информации, так как тот заблокирован потребителем. Получаем тупиковую ситуацию.

В сложных программах произвести анализ правильности использования семафоров с карандашом в руках становится очень непросто. В то же время обычные способы отладки программ зачастую не дают результата, поскольку возникновение ошибок зависит от

interleaving атомарных операций, и ошибки могут быть трудновоспроизводимы. Для того чтобы облегчить работу программистов, в 1974 году Хором (Hoare) был предложен механизм еще более высокого уровня, чем семафоры, получивший название мониторов. Мы с вами рассмотрим конструкцию, несколько отличающуюся от оригинальной.

Мониторы представляют собой тип данных, который может быть с успехом внедрен в объектно-ориентированные языки программирования. Монитор обладает собственными переменными, определяющими его состояние. Значения этих переменных извне могут быть изменены только с помощью вызова функций-методов, принадлежащих монитору. В свою очередь, эти функции-методы могут использовать в работе только данные, находящиеся внутри монитора, и свои параметры. На абстрактном уровне можно описать структуру монитора следующим образом:

```
monitor monitor_name {
   описание внутренних переменных ;

   void m1(...){...
   }
   void m2(...){...
   }
   ...
   void mn(...){...
   }

   {
      блок инициализации
      внутренних переменных;
   }
}
```

Здесь функции m1,..., mn представляют собой функции-методы монитора, а блок инициализации внутренних переменных содержит операции, которые выполняются один и только один раз: при создании монитора или при самом первом вызове какой-либо функции-метода до ее исполнения.

Важной особенностью мониторов является то, что в любой момент времени только один процесс может быть активен, т. е. находиться в состоянии готовность или исполнение, внутри данного монитора. Поскольку мониторы представляют собой особые конструкции языка программирования, компилятор может отличить вызов функции, принадлежащей монитору, от вызовов других функций и обработать его специальным образом, добавив к нему пролог и эпилог, реализующий взаимоисключение. Так как обязанность конструирования механизма взаимоисключений возложена на компилятор, а не на программиста, работа программиста при использовании мониторов существенно упрощается, а вероятность возникновения ошибок становится меньше.

Однако одних только взаимоисключений недостаточно для того, чтобы в полном объеме реализовать решение задач, возникающих при взаимодействии процессов. Нам нужны еще и средства организации очередности процессов, подобно семафорам full и empty в предыдущем примере. Для этого в мониторах было введено понятие условных переменных (condition variables)1), над которыми можно совершать две операции wait и signal, отчасти похожие на операции P и V над семафорами.

Если функция монитора не может выполняться дальше, пока не наступит некоторое событие, она выполняет операцию wait над какой-либо условной переменной. При этом

процесс, выполнивший операцию wait, блокируется, становится неактивным, и другой процесс получает возможность войти в монитор.

Когда ожидаемое событие происходит, другой процесс внутри функции-метода совершает операцию signal над той же самой условной переменной. Это приводит к пробуждению ранее заблокированного процесса, и он становится активным. Если несколько процессов дожидались операции signal для этой переменной, то активным становится только один из них. Что можно предпринять для того, чтобы у нас не оказалось двух процессов, разбудившего и пробужденного, одновременно активных внутри монитора? Хор предложил, чтобы пробужденный процесс подавлял исполнение разбудившего процесса, пока он сам не покинет монитор. Несколько позже Хансен (Hansen) предложил другой механизм: разбудивший процесс покидает монитор немедленно после исполнения операции signal. Мы будем придерживаться подхода Хансена.

Необходимо отметить, что условные переменные, в отличие от семафоров Дейкстры, не умеют запоминать предысторию. Это означает, что операция signal всегда должна выполняться после операции wait. Если операция signal выполняется над условной переменной, с которой не связано ни одного заблокированного процесса, то информация о произошедшем событии будет утеряна. Следовательно, выполнение операции wait всегда будет приводить к блокированию процесса.

Давайте применим концепцию мониторов к решению задачи производитель-потребитель.

```
monitor ProducerConsumer {
      condition full, empty;
      int count;
      void put() {
         if(count == N) full.wait;
         put_item;
         count += 1;
         if(count == 1) empty.signal;
      }
      void get() {
         if (count == 0) empty.wait;
         get_item();
         count -= 1;
         if(count == N-1) full.signal;
      }
      {
         count = 0;
      }
   }
```

Producer:

```
   while(1) {
      produce_item;
      ProducerConsumer.put();
   }
```

Consumer:

```
   while(1) {
      ProducerConsumer.get();
      consume_item;
   }
```

Легко убедиться, что приведенный пример действительно решает поставленную задачу.

Реализация мониторов требует разработки специальных языков программирования и компиляторов для них. Мониторы встречаются в таких языках, как параллельный Евклид, параллельный Паскаль, Java и т. д. Эмуляция мониторов с помощью системных вызовов для обычных широко используемых языков программирования не так проста, как эмуляция семафоров. Поэтому можно пользоваться еще одним механизмом со скрытыми взаимоисключениями, механизмом, о котором мы уже упоминали, – передачей сообщений.

Сообщения

Для прямой и непрямой адресации достаточно двух примитивов, чтобы описать передачу сообщений по линии связи – send и receive. В случае прямой адресации мы будем обозначать их так:

```
send(P, message) – послать сообщение message процессу P;
receive(Q, message) – получить сообщение message от процесса Q.
```

В случае непрямой адресации мы будем обозначать их так:

```
send(A, message) – послать сообщение message в почтовый ящик A;
receive(A, message) – получить сообщение message из почтового
ящика A.
```

Примитивы send и receive уже имеют скрытый от наших глаз механизм взаимоисключения. Более того, в большинстве систем они уже имеют и скрытый механизм блокировки при чтении из пустого буфера и при записи в полностью заполненный буфер. Реализация решения задачи producer-consumer для таких примитивов становится неприлично тривиальной. Надо отметить, что, несмотря на простоту использования, передача сообщений в пределах одного компьютера происходит существенно медленнее, чем работа с семафорами и мониторами.

Эквивалентность семафоров, мониторов и сообщений

Мы рассмотрели три высокоуровневых механизма, использующихся для организации взаимодействия процессов. Можно показать, что в рамках одной вычислительной системы, когда процессы имеют возможность использовать разделяемую память, все они эквивалентны. Это означает, что любые два из предложенных механизмов могут быть реализованы на базе третьего, оставшегося механизма.

Реализация мониторов и передачи сообщений с помощью семафоров

Рассмотрим сначала, как реализовать мониторы с помощью семафоров. Для этого нам нужно уметь реализовывать взаимоисключения при входе в монитор и условные переменные. Возьмем семафор mutex с начальным значением 1 для реализации взаимоисключения при входе в монитор и по одному семафору ci для каждой условной переменной. Кроме того, для каждой условной переменной заведем счетчик fi для индикации наличия ожидающих процессов. Когда процесс входит в монитор, компилятор будет генерировать вызов функции monitor_enter, которая выполняет операцию P над семафором mutex для данного монитора. При нормальном выходе из монитора (то есть при выходе без вызова операции signal для условной переменной) компилятор будет

генерировать вызов функции monitor_exit, которая выполняет операцию V над этим семафором.

Для выполнения операции wait над условной переменной компилятор будет генерировать вызов функции wait, которая выполняет операцию V для семафора mutex, разрешая другим процессам входить в монитор, и выполняет операцию P над соответствующим семафором ci, блокируя вызвавший процесс. Для выполнения операции signal над условной переменной компилятор будет генерировать вызов функции signal_exit, которая выполняет операцию V над ассоциированным семафором ci (если есть процессы, ожидающие соответствующего события), и выход из монитора, минуя функцию monitor_exit.

```
Semaphore mutex = 1;

void monitor_enter(){
   P(mutex);
}

void monitor_exit(){
   V(mutex);
}

Semaphore ci = 0;
int fi = 0;

void wait(i){
   fi=fi + 1;
   V(mutex);
   P(ci);
   fi=fi – 1;
}

void signal_exit(i){
   if (fi)V(ci);
   else V(mutex);
}
```

Заметим, что при выполнении функции signal_exit, если кто-либо ожидал этого события, процесс покидает монитор без увеличения значения семафора mutex, не разрешая тем самым всем процессам, кроме разбуженного, войти в монитор. Это увеличение совершит разбуженный процесс, когда покинет монитор обычным способом или когда выполнит новую операцию wait над какой-либо условной переменной.

Рассмотрим теперь, как реализовать передачу сообщений, используя семафоры. Для простоты опишем реализацию только одной очереди сообщений. Выделим в разделяемой памяти достаточно большую область под хранение сообщений, там же будем записывать, сколько пустых и заполненных ячеек находится в буфере, хранить ссылки на списки процессов, ожидающих чтения и памяти. Взаимоисключение при работе с разделяемой памятью будем обеспечивать семафором mutex. Также заведем по одному семафору ci на взаимодействующий процесс, для того чтобы обеспечивать блокирование процесса при попытке чтения из пустого буфера или при попытке записи в переполненный буфер. Посмотрим, как такой механизм будет работать. Начнем с процесса, желающего получить сообщение.

Процесс-получатель с номером i прежде всего выполняет операцию P(mutex), получая в монопольное владение разделяемую память. После чего он проверяет, есть ли в буфере сообщения. Если нет, то он заносит себя в список процессов, ожидающих сообщения, выполняет V(mutex) и P(ci). Если сообщение в буфере есть, то он читает его, изменяет счетчики буфера и проверяет, есть ли процессы в списке процессов, жаждущих записи. Если таких процессов нет, то выполняется V(mutex), и процесс-получатель выходит из критической области. Если такой процесс есть (с номером j), то он удаляется из этого списка, выполняется V для его семафора cj, и мы выходим из критического района. Проснувшийся процесс начинает выполняться в критическом районе, так как mutex у нас имеет значение 0 и никто более не может попасть в критический район. При выходе из критического района именно разбуженный процесс произведет вызов V(mutex).

Как строится работа процесса-отправителя с номером i? Процесс, посылающий сообщение, тоже ждет, пока он не сможет иметь монополию на использование разделяемой памяти, выполнив операцию P(mutex). Далее он проверяет, есть ли место в буфере, и если да, то помещает сообщение в буфер, изменяет счетчики и смотрит, есть ли процессы, ожидающие сообщения. Если нет, выполняет V(mutex) и выходит из критической области, если есть, "будит" один из них (с номером j), вызывая V(cj), с одновременным удалением этого процесса из списка процессов, ожидающих сообщений, и выходит из критического региона без вызова V(mutex), предоставляя тем самым возможность разбуженному процессу прочитать сообщение. Если места в буфере нет, то процесс-отправитель заносит себя в очередь процессов, ожидающих возможности записи, и вызывает V(mutex) и P(ci).

Реализация семафоров и передачи сообщений с помощью мониторов

Нам достаточно показать, что с помощью мониторов можно реализовать семафоры, так как получать из семафоров сообщения мы уже умеем.

Самый простой способ такой реализации выглядит следующим образом. Заведем внутри монитора переменную-счетчик, связанный с эмулируемым семафором список блокируемых процессов и по одной условной переменной на каждый процесс. При выполнении операции P над семафором вызывающий процесс проверяет значение счетчика. Если оно больше нуля, уменьшает его на 1 и выходит из монитора. Если оно равно 0, процесс добавляет себя в очередь процессов, ожидающих события, и выполняет операцию wait над своей условной переменной. При выполнении операции V над семафором процесс увеличивает значение счетчика, проверяет, есть ли процессы, ожидающие этого события, и если есть, удаляет один из них из списка и выполняет операцию signal для условной переменной, соответствующей процессу.

Реализация семафоров и мониторов с помощью очередей сообщений

Покажем, наконец, как реализовать семафоры с помощью очередей сообщений. Для этого воспользуемся более хитрой конструкцией, введя новый синхронизирующий процесс. Этот процесс имеет счетчик и очередь для процессов, ожидающих включения семафора. Для того чтобы выполнить операции P и V, процессы посылают синхронизирующему процессу сообщения, в которых указывают свои потребности, после чего ожидают получения подтверждения от синхронизирующего процесса.

После получения сообщения синхронизирующий процесс проверяет значение счетчика, чтобы выяснить, можно ли совершить требуемую операцию. Операция V всегда может быть выполнена, в то время как операция P может потребовать блокирования процесса.

Если операция может быть совершена, то она выполняется, и синхронизирующий процесс посылает подтверждающее сообщение. Если процесс должен быть блокирован, то его идентификатор заносится в очередь блокированных процессов, и подтверждение не посылается. Позднее, когда какой-либо из других процессов выполнит операцию V, один из блокированных процессов удаляется из очереди ожидания и получает соответствующее подтверждение.

Тупики

Предположим, что несколько процессов конкурируют за обладание конечным числом ресурсов. Если запрашиваемый процессом ресурс недоступен, ОС переводит данный процесс в состояние ожидания. В случае когда требуемый ресурс удерживается другим ожидающим процессом, первый процесс не сможет сменить свое состояние. Такая ситуация называется тупиком (deadlock). Говорят, что в мультипрограммной системе процесс находится в состоянии тупика, если он ожидает события, которое никогда не произойдет. Системная тупиковая ситуация, или "зависание системы", является следствием того, что один или более процессов находятся в состоянии тупика. Иногда подобные ситуации называют взаимоблокировками. В общем случае проблема тупиков эффективного решения не имеет.

Рассмотрим пример. Предположим, что два процесса осуществляют вывод с ленты на принтер. Один из них успел монополизировать ленту и претендует на принтер, а другой наоборот. После этого оба процесса оказываются заблокированными в ожидании второго ресурса (см. рис. 7.1).

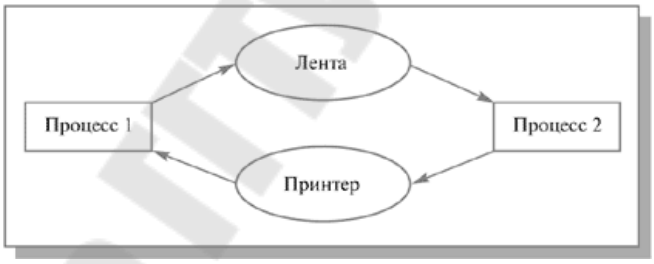

Рис. 7.1. Пример тупиковой ситуации

Определение. Множество процессов находится в тупиковой ситуации, если каждый процесс из множества ожидает события, которое может вызвать только другой процесс данного множества. Так как все процессы чего-то ожидают, то ни один из них не сможет инициировать событие, которое разбудило бы другого члена множества и, следовательно, все процессы будут спать вместе.

Выше приведен пример взаимоблокировки, возникающей при работе с так называемыми выделенными устройствами. Тупики, однако, могут иметь место и в других ситуациях. Например, в системах управления базами данных записи могут быть локализованы процессами, чтобы избежать состояния гонок (см. лекцию 5 "Алгоритмы синхронизации"). В этом случае может получиться так, что один из процессов заблокировал записи, необходимые другому процессу, и наоборот. Таким образом, тупики могут иметь место как на аппаратных, так и на программных ресурсах.

Тупики также могут быть вызваны ошибками программирования. Например, процесс может напрасно ждать открытия семафора, потому что в некорректно написанном приложении эту операцию забыли предусмотреть. Другой причиной бесконечного ожидания может быть дискриминационная политика по отношению к некоторым

процессам. Однако чаще всего событие, которого ждет процесс в тупиковой ситуации, – освобождение ресурса, поэтому в дальнейшем будут рассмотрены методы борьбы с тупиками ресурсного типа.

Ресурсами могут быть как устройства, так и данные. Некоторые ресурсы допускают разделение между процессами, то есть являются разделяемыми ресурсами. Например, память, процессор, диски коллективно используются процессами. Другие не допускают разделения, то есть являются выделенными, например лентопротяжное устройство. К взаимоблокировке может привести использование как выделенных, так и разделяемых ресурсов. Например, чтение с разделяемого диска может одновременно осуществляться несколькими процессами, тогда как запись предполагает исключительный доступ к данным на диске. Можно считать, что часть диска, куда происходит запись, выделена конкретному процессу. Поэтому в дальнейшем мы будем исходить из предположения, что тупики связаны с выделенными ресурсами , то есть тупики возникают, когда процессу предоставляется эксклюзивный доступ к устройствам, файлам и другим ресурсам.

Традиционная последовательность событий при работе с ресурсом состоит из запроса, использования и освобождения ресурса. Тип запроса зависит от природы ресурса и от ОС. Запрос может быть явным, например специальный вызов request, или неявным – open для открытия файла. Обычно, если ресурс занят и запрос отклонен, запрашивающий процесс переходит в состояние ожидания.

Далее в данной лекции будут рассматриваться вопросы обнаружения, предотвращения, обхода тупиков и восстановления после тупиков. Как правило, борьба с тупиками – очень дорогостоящее мероприятие. Тем не менее для ряда систем, например для систем реального времени, иного выхода нет.

Условия возникновения тупиков

Условия возникновения тупиков были сформулированы Коффманом, Элфиком и Шошани в 1970 г.

1. Условие взаимоисключения (Mutual exclusion). Одновременно использовать ресурс может только один процесс.
2. Условие ожидания ресурсов (Hold and wait). Процессы удерживают ресурсы, уже выделенные им, и могут запрашивать другие ресурсы.
3. Условие неперераспределяемости (No preemtion). Ресурс, выделенный ранее, не может быть принудительно забран у процесса. Освобождены они могут быть только процессом, который их удерживает.
4. Условие кругового ожидания (Circular wait). Существует кольцевая цепь процессов, в которой каждый процесс ждет доступа к ресурсу, удерживаемому другим процессом цепи.

Для образования тупика необходимым и достаточным является выполнение всех четырех условий.

Обычно тупик моделируется циклом в графе, состоящем из узлов двух видов: прямоугольников – процессов и эллипсов – ресурсов, наподобие того, что изображен на рис. 7.1. Стрелки, направленные от ресурса к процессу, показывают, что ресурс выделен данному процессу. Стрелки, направленные от процесса к ресурсу, означают, что процесс запрашивает данный ресурс.

Основные направления борьбы с тупиками

Проблема тупиков инициировала много интересных исследований в области информатики. Очевидно, что условие циклического ожидания отличается от остальных. Первые три условия формируют правила, существующие в системе, тогда как четвертое условие описывает ситуацию, которая может сложиться при определенной неблагоприятной последовательности событий. Поэтому методы предотвращения взаимоблокировок ориентированы главным образом на нарушение первых трех условий путем введения ряда ограничений на поведение процессов и способы распределения ресурсов. Методы обнаружения и устранения менее консервативны и сводятся к поиску и разрыву цикла ожидания ресурсов.

Итак, основные направления борьбы с тупиками:

- игнорирование проблемы в целом
- предотвращение тупиков
- обнаружение тупиков
- восстановление после тупиков

Игнорирование проблемы тупиков

Простейший подход – не замечать проблему тупиков. Для того чтобы принять такое решение, необходимо оценить вероятность возникновения взаимоблокировки и сравнить ее с вероятностью ущерба от других отказов аппаратного и программного обеспечения. Проектировщики обычно не желают жертвовать производительностью системы или удобством пользователей для внедрения сложных и дорогостоящих средств борьбы с тупиками.

Любая ОС, имеющая в ядре ряд массивов фиксированной размерности, потенциально страдает от тупиков, даже если они не обнаружены. Таблица открытых файлов, таблица процессов, фактически каждая таблица являются ограниченными ресурсами. Заполнение всех записей таблицы процессов может привести к тому, что очередной запрос на создание процесса может быть отклонен. При неблагоприятном стечении обстоятельств несколько процессов могут выдать такой запрос одновременно и оказаться в тупике. Следует ли отказываться от вызова CreateProcess, чтобы решить эту проблему? Подход большинства популярных ОС (Unix, Windows и др.) состоит в том, чтобы игнорировать данную проблему в предположении, что маловероятный случайный тупик предпочтительнее, чем нелепые правила, заставляющие пользователей ограничивать число процессов, открытых файлов и т. п. Сталкиваясь с нежелательным выбором между строгостью и удобством, трудно найти решение, которое устраивало бы всех.

Способы предотвращения тупиков

Цель предотвращения тупиков – обеспечить условия, исключающие возможность возникновения тупиковых ситуаций. Большинство методов связано с предотвращением одного из условий возникновения взаимоблокировки.

Система, предоставляя ресурс в распоряжение процесса, должна принять решение, безопасно это или нет. Возникает вопрос: есть ли такой алгоритм, который помогает всегда избегать тупиков и делать правильный выбор. Ответ – да, мы можем избегать тупиков, но только если определенная информация известна заранее.

Способы предотвращения тупиков путем тщательного распределения ресурсов. Алгоритм банкира

Можно избежать взаимоблокировки, если распределять ресурсы, придерживаясь определенных правил. Среди такого рода алгоритмов наиболее известен алгоритм банкира, предложенный Дейкстрой, который базируется на так называемых безопасных или надежных состояниях (safe state). Безопасное состояние – это такое состояние, для которого имеется по крайней мере одна последовательность событий, которая не приведет к взаимоблокировке. Модель алгоритма основана на действиях банкира, который, имея в наличии капитал, выдает кредиты.

Суть алгоритма состоит в следующем.

1. Предположим, что у системы в наличии n устройств, например лент.
2. ОС принимает запрос от пользовательского процесса, если его максимальная потребность не превышает n.
3. Пользователь гарантирует, что если ОС в состоянии удовлетворить его запрос, то все устройства будут возвращены системе в течение конечного времени.
4. Текущее состояние системы называется надежным, если ОС может обеспечить всем процессам их выполнение в течение конечного времени.
5. В соответствии с алгоритмом банкира выделение устройств возможно, только если состояние системы остается надежным.

Рассмотрим пример надежного состояния для системы с 3 пользователями и 11 устройствами, где 9 устройств задействовано, а 2 имеется в резерве. Пусть текущая ситуация такова:

Пользователи	Максимальная потребность в ресурсах	Выделенное пользователям количество ресурсов
Первый	9	6
Второй	10	2
Третий	3	1

Рис. 7.2. Пример надежного состояния для системы с 3 пользователями и 11 устройствами.

Данное состояние надежно. Последующие действия системы могут быть таковы. Вначале удовлетворить запросы третьего пользователя, затем дождаться, когда он закончит работу и освободит свои три устройства. Затем можно обслужить первого и второго пользователей. То есть система удовлетворяет только те запросы, которые оставляют ее в надежном состоянии, и отклоняет остальные.

Термин ненадежное состояние не предполагает, что обязательно возникнут тупики. Он лишь говорит о том, что в случае неблагоприятной последовательности событий система может зайти в тупик.

Данный алгоритм обладает тем достоинством, что при его использовании нет необходимости в перераспределении ресурсов и откате процессов назад. Однако использование этого метода требует выполнения ряда условий.

- Число пользователей и число ресурсов фиксировано.
- Число работающих пользователей должно оставаться постоянным.
- Алгоритм требует, чтобы клиенты гарантированно возвращали ресурсы.
- Должны быть заранее указаны максимальные требования процессов к ресурсам. Чаще всего данная информация отсутствует.

Наличие таких жестких и зачастую неприемлемых требований может склонить разработчиков к выбору других решений проблемы взаимоблокировки.

Предотвращение тупиков за счет нарушения условий возникновения тупиков

В отсутствие информации о будущих запросах единственный способ избежать взаимоблокировки – добиться невыполнения хотя бы одного из условий раздела "Условия возникновения тупиков".

Нарушение условия взаимоисключения

В общем случае избежать взаимоисключений невозможно. Доступ к некоторым ресурсам должен быть исключительным. Тем не менее некоторые устройства удается обобществить. В качестве примера рассмотрим принтер. Известно, что пытаться осуществлять вывод на принтер могут несколько процессов. Во избежание хаоса организуют промежуточное формирование всех выходных данных процесса на диске, то есть разделяемом устройстве. Лишь один системный процесс, называемый сервисом или демоном принтера, отвечающий за вывод документов на печать по мере освобождения принтера, реально с ним взаимодействует. Эта схема называется спулингом (spooling). Таким образом, принтер становится разделяемым устройством, и тупик для него устранен.

К сожалению, не для всех устройств и не для всех данных можно организовать спулинг. Неприятным побочным следствием такой модели может быть потенциальная тупиковая ситуация из-за конкуренции за дисковое пространство для буфера спулинга. Тем не менее в той или иной форме эта идея применяется часто.

Нарушение условия ожидания дополнительных ресурсов

Условия ожидания ресурсов можно избежать, потребовав выполнения стратегии двухфазного захвата.

- В первой фазе процесс должен запрашивать все необходимые ему ресурсы сразу. До тех пор пока они не предоставлены, процесс не может продолжать выполнение.
- Если в первой фазе некоторые ресурсы, которые были нужны данному процессу, уже заняты другими процессами, он освобождает все ресурсы, которые были ему выделены, и пытается повторить первую фазу.

В известном смысле этот подход напоминает требование захвата всех ресурсов заранее. Естественно, что только специально организованные программы могут быть приостановлены в течение первой фазы и рестартованы впоследствии.

Таким образом, один из способов – заставить все процессы затребовать нужные им ресурсы перед выполнением ("все или ничего"). Если система в состоянии выделить процессу все необходимое, он может работать до завершения. Если хотя бы один из ресурсов занят, процесс будет ждать.

Данное решение применяется в пакетных мэйнфреймах (mainframe), которые требуют от пользователей перечислить все необходимые его программе ресурсы. Другим примером может служить механизм двухфазной локализации записей в СУБД. Однако в целом подобный подход не слишком привлекателен и приводит к неэффективному использованию компьютера. Как уже отмечалось, перечень будущих запросов к ресурсам редко удается спрогнозировать. Если такая информация есть, то можно воспользоваться

алгоритмом банкира. Заметим также, что описываемый подход противоречит парадигме модульности в программировании, поскольку приложение должно знать о предполагаемых запросах к ресурсам во всех модулях.

Нарушение принципа отсутствия перераспределения

Если бы можно было отбирать ресурсы у удерживающих их процессов до завершения этих процессов, то удалось бы добиться невыполнения третьего условия возникновения тупиков. Перечислим минусы данного подхода.

Во-первых, отбирать у процессов можно только те ресурсы, состояние которых легко сохранить, а позже восстановить, например состояние процессора. Во-вторых, если процесс в течение некоторого времени использует определенные ресурсы, а затем освобождает эти ресурсы, он может потерять результаты работы, проделанной до настоящего момента. Наконец, следствием данной схемы может быть дискриминация отдельных процессов, у которых постоянно отбирают ресурсы. Весь вопрос в цене подобного решения, которая может быть слишком высокой, если необходимость отбирать ресурсы возникает часто.

Нарушение условия кругового ожидания

Трудно предложить разумную стратегию, чтобы избежать последнего условия из раздела "Условия возникновения тупиков" – циклического ожидания.

Один из способов – упорядочить ресурсы. Например, можно присвоить всем ресурсам уникальные номера и потребовать, чтобы процессы запрашивали ресурсы в порядке их возрастания. Тогда круговое ожидание возникнуть не может. После последнего запроса и освобождения всех ресурсов можно разрешить процессу опять осуществить первый запрос. Очевидно, что практически невозможно найти порядок, который удовлетворит всех.

Один из немногих примеров упорядочивания ресурсов – создание иерархии спин-блокировок в Windows 2000. Спин-блокировка – простейший способ синхронизации (вопросы синхронизации процессов рассмотрены в соответствующей лекции). Спин-блокировка может быть захвачена и освобождена процессом. Классическая тупиковая ситуация возникает, когда процесс P1 захватывает спин-блокировку S1 и претендует на спин-блокировку S2, а процесс P2, захватывает спин-блокировку S2 и хочет дополнительно захватить спин-блокировку S1. Чтобы этого избежать, все спин-блокировки помещаются в упорядоченный список. Захват может осуществляться только в порядке, указанном в списке.

Другой способ атаки условия кругового ожидания – действовать в соответствии с правилом, согласно которому каждый процесс может иметь только один ресурс в каждый момент времени. Если нужен второй ресурс – освободи первый. Очевидно, что для многих процессов это неприемлемо.

Таким образом, технология предотвращения циклического ожидания, как правило, неэффективна и может без необходимости закрывать доступ к ресурсам.

Обнаружение тупиков

Обнаружение взаимоблокировки сводится к фиксации тупиковой ситуации и выявлению вовлеченных в нее процессов. Для этого производится проверка наличия циклического ожидания в случаях, когда выполнены первые три условия возникновения тупика. Методы обнаружения активно используют графы распределения ресурсов.

Рассмотрим модельную ситуацию.

- Процесс P1 ожидает ресурс R1.
- Процесс P2 удерживает ресурс R2 и ожидает ресурс R1.
- Процесс P3 удерживает ресурс R1 и ожидает ресурс R3.
- Процесс P4 ожидает ресурс R2.
- Процесс P5 удерживает ресурс R3 и ожидает ресурс R2.

Вопрос состоит в том, является ли данная ситуация тупиковой, и если да, то какие процессы в ней участвуют. Для ответа на этот вопрос можно сконструировать граф ресурсов, как показано на рис. 7.3. Из рисунка видно, что имеется цикл, моделирующий условие кругового ожидания, и что процессы P2,P3,P5, а может быть, и другие находятся в тупиковой ситуации.

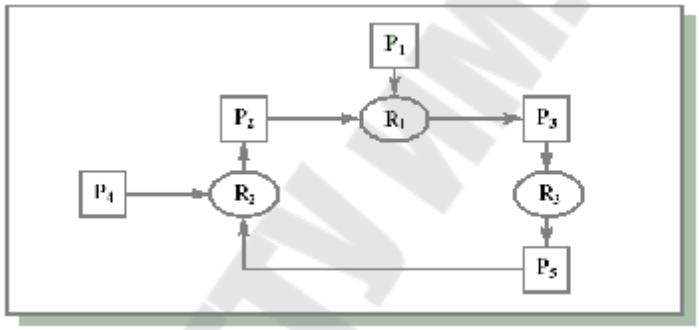

Рис. 7.3. Граф ресурсов

Визуально легко обнаружить наличие тупика, но нужны также формальные алгоритмы, реализуемые на компьютере. Один из таких алгоритмов описан в [Таненбаум, 2002], там же можно найти ссылки на другие алгоритмы. Существуют и другие способы обнаружения тупиков, применимые также в ситуациях, когда имеется несколько ресурсов каждого типа. Так в [Дейтел, 1987] описан способ, называемый редукцией графа распределения ресурсов, а в [Таненбаум, 2002] – матричный алгоритм.

Восстановление после тупиков

Обнаружив тупик, можно вывести из него систему, нарушив одно из условий существования тупика. При этом, возможно, несколько процессов частично или полностью потеряют результаты проделанной работы.

Сложность восстановления обусловлена рядом факторов.

- В большинстве систем нет достаточно эффективных средств, чтобы приостановить процесс, вывести его из системы и возобновить впоследствии с того места, где он был остановлен.
- Если даже такие средства есть, то их использование требует затрат и внимания оператора.
- Восстановление после тупика может потребовать значительных усилий.

Самый простой и наиболее распространенный способ устранить тупик – завершить выполнение одного или более процессов, чтобы впоследствии использовать его ресурсы.

Тогда в случае удачи остальные процессы смогут выполняться. Если это не помогает, можно ликвидировать еще несколько процессов. После каждой ликвидации должен запускаться алгоритм обнаружения тупика.

По возможности лучше ликвидировать тот процесс, который может быть без ущерба возвращен к началу (такие процессы называются идемпотентными). Примером такого процесса может служить компиляция. С другой стороны, процесс, который изменяет содержимое базы данных, не всегда может быть корректно запущен повторно.

В некоторых случаях можно временно забрать ресурс у текущего владельца и передать его другому процессу. Возможность забрать ресурс у процесса, дать его другому процессу и затем без ущерба вернуть назад сильно зависит от природы ресурса. Подобное восстановление часто затруднительно, если не невозможно.

В ряде систем реализованы средства отката и перезапуска или рестарта с контрольной точки (сохранение состояния системы в какой-то момент времени). Если проектировщики системы знают, что тупик вероятен, они могут периодически организовывать для процессов контрольные точки. Иногда это приходится делать разработчикам прикладных программ.

Когда тупик обнаружен, видно, какие ресурсы вовлечены в цикл кругового ожидания. Чтобы осуществить восстановление, процесс, который владеет таким ресурсом, должен быть отброшен к моменту времени, предшествующему его запросу на этот ресурс.

Лекция 6. Управление процессами и потоками в ОС Windows.

Классификация средств реализации многозадачности

Используемые в операционной системе средства реализации многозадачности можно разделить на 5 основных групп:

1) Взаимодействие с устройствами. Существуют как специализированные средства взаимодействия, специфичные для конкретного вида устройств (например, для графических устройств), так и относительно универсальные средства, применимые к устройствам разных типов. Наиболее характерным примером являются средства асинхронного ввода-вывода.
2) Средства управления потоками. Когда процесс запускается, операционная система в нем самостоятельно создает первичный поток, начинающий исполнение кода этого процесса. Создание всех остальных потоков процесса требует специальных действий. Операционная система должна предоставлять средства для создания процессов и потоков, их завершения, приостановки или возобновления, изменения их характеристик (например, приоритетов, прав доступа и т.д.). В Windows существует очень интересный гибрид средств управления потоками и асинхронного ввода-вывода под названием "порт завершения ввода-вывода". Несмотря на такое название, это, по большей части, именно специализированный механизм управления потоками.
3) Взаимодействие потоков в рамках одного процесса. Потоки, работающие в рамках одного процесса, имеют возможность взаимодействовать друг с другом, используя общее адресное пространство процесса. Это взаимодействие может, с одной стороны, приводить к конфликтам одновременного доступа (нужны средства разрешения конфликтов) и, с другой стороны, требовать средств изоляции некоторых данных одного потока от данных другого (нужны механизмы организации памяти, локальной для потока).

4) Взаимодействие между процессами одного компьютера. Чуть более сложный случай – необходимость взаимодействия различных процессов в рамках одной вычислительной системы. Особенность этого случая связана с тем, что разные процессы работают в изолированных друг от друга адресных пространствах, однако при этом существует возможность осуществить обмен данными через общую, разделяемую несколькими процессами, память. Диспетчер памяти операционной системы должен предусмотреть средства организации разделяемой между процессами памяти. Кроме того, многие операционные системы предоставляют дополнительные средства межпроцессного взаимодействия; многие из них являются надстройками над средствами работы с разделяемой памятью.
5) Взаимодействие между процессами разных компьютеров. В общем случае память разных компьютеров можно рассматривать как изолированную друг от друга. В этом случае для взаимодействия разных процессов потребуется пересылка сообщений, содержащих данные и некоторую управляющую информацию, между разными компьютерами (узлами сети). Операционная система должна предоставить некоторый базовый набор функций по пересылке данных по коммуникационной сети и, зачастую, богатый набор различных средств, работающих "поверх" этого базового уровня.

Реализация процессов и потоков.

В ОС Windows кванты выделяются не программам или процессам, а порожденным ими потокам. Как минимум, каждый процесс имеет хотя бы один (главный) поток. Поэтому основным объектом многозадачности в Windows выступает поток.

В современных вычислительных системах выделяют два основных типа потоков:

- симметричные потоки (symmetric threads) — это потоки, которые имеют одинаковое предназначение, исполняют один и тот же код и могут разделять одни и те же ресурсы. Более того, приложения, рассчитанные на серьезную нагрузку, могут поддерживать пул (pool) однотипных потоков. Поскольку создание потока требует определенного времени, для ускорения работы желательно заранее иметь нужное число готовых потоков и активизировать их по мере подключения очередного клиента.
- асимметричные потоки (asymmetric threads) — это потоки, решающие различные задачи и, как правило, не разделяющие совместные ресурсы. Необходимость в асимметричных потоках возникает: когда в программе необходимы длительные вычисления, при этом необходимо сохранить нормальную реакцию на ввод; когда нужно обрабатывать асинхронный ввод/вывод с использованием различных устройств (COM-порта, звуковой карты, принтера и т. п.); когда вы хотите создать несколько окон и одновременно обрабатывать ввод в них.

В Windows определен список событий, которые приводят к перепланированию потоков:

- создание и завершение потока;
- выделенный потоку квант исчерпан;
- поток вышел из состояния ожидания;
- поток перешел в состояние ожидания;
- изменен приоритет потока;
- изменена привязка к процессору.

Переход из состояния "готовность" в состояние "выполнение" сделан в два этапа - выбранный к выполнению поток подготавливается к выполнению и переводится в состояние "выбран"; эта подготовка может осуществляться до наступления момента

перепланирования, и в нужный момент достаточно просто переключить контекст выполняющегося потока на выбранный.

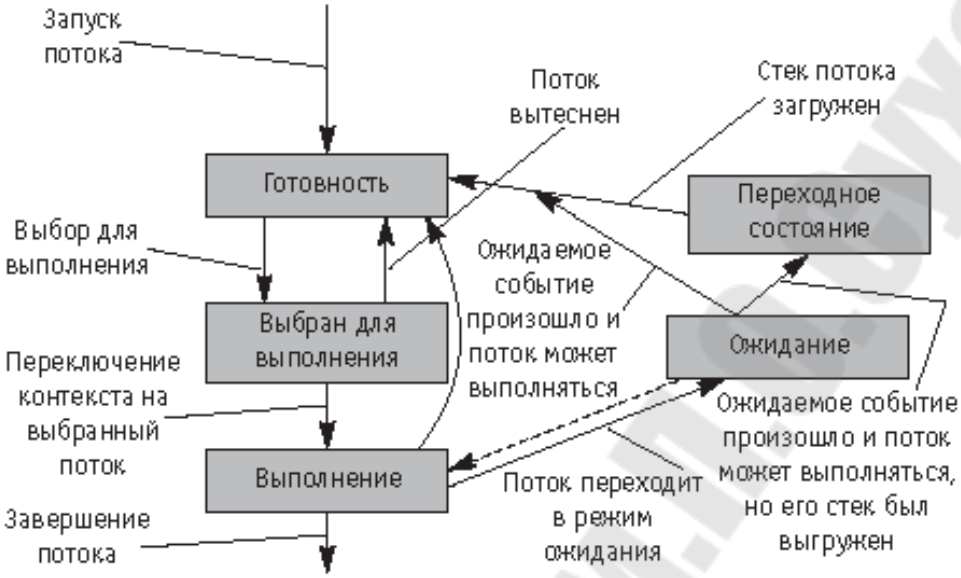

Рис.1. Граф состояний потоков в ОС Windows

Также в два этапа может происходить переход из состояния "ожидание" в "готовность": если ожидание было долгим, то стек потока может быть выгружен из оперативной памяти. В этом случае поток переводится в промежуточное состояние до завершения загрузки стека – в списке готовых к выполнению потоков находятся только те, которые можно начать выполнять без лишнего ожидания.

Операционная система имеет различные очереди готовых к выполнению потоков — для каждого уровня приоритета свой. В момент распределения нового кванта времени она просматривает очереди — от высшего приоритета к низшему. Готовый к выполнению поток, стоящий первым в очереди, получает квант времени и перемещается в хвост очереди. Поток будет исполняться всю продолжительность кванта, если не произойдет одно из двух событий:

- выполняющийся поток остановился для ожидания;
- появился готовый к выполнению поток с более высоким приоритетом.

Замечание: *Нормальная практика для асимметричных потоков — это назначение потоку, обрабатывающему ввод, более высокого приоритета, а всем остальным — более низкого или даже приоритета idle, если этот поток должен выполняться только во время простоя системы*.

Механизм выбора потока для выполнения усложняется в случае SMP (Shared Memory Processor или Symmetric MultiProcessor) систем, когда помимо приоритета готового к исполнению потока учитывается, на каком процессоре ранее выполнялся код данного потока.

В Windows выделяют понятие "идеального" процессора – им назначается процессор, на котором запускается приложение в первый раз. В дальнейшем система старается выполнять код потока именно на этом процессоре – для SMP систем это решение улучшает использование кэш-памяти, а для NUMA систем позволяет, по большей части, ограничиться использованием оперативной памяти, локальной для данного процессора. Заметим, что диспетчер памяти Windows при выделении памяти для запускаемого

процесса старается учитывать доступность памяти для назначенного процессора в случае NUMA системы.

В многопроцессорной системе используется либо первый простаивающий процессор, либо, при необходимости вытеснения уже работающего потока, проверяются идеальный процессор, последний использовавшийся и процессор с наибольшим номером. Если на одном из них работает поток с меньшим приоритетом, то последний вытесняется и заменяется новым потоком; в противном случае выполнение потока откладывается (даже если в системе есть процессоры, занятые потоками с меньшим приоритетом).

Современные реализации Windows в рамках единого дерева кодов могут быть использованы для различных классов задач – от рабочих станций, обслуживающих преимущественно интерфейс пользователя, до серверных установок на многопроцессорных машинах. Чтобы можно было эффективно использовать одну ОС в столь разных классах систем, планировщик Windows динамически изменяет длительность квантов и приоритеты, назначаемые потокам. Администратор системы может в некоторой степени изменить поведение системы при назначении длительности квантов и приоритетов потоков.

Управление квантованием в ОС Windows

Квантование потоков осуществляется по тикам системного таймера, продолжительность одного тика составляет обычно 10 или 15 мс, больший по продолжительности тик назначают многопроцессорным машинам. Каждый тик системного таймера соответствует 3 условным единицам; величина кванта может варьироваться от 2 до 12 тиков (от 6 до 36 единиц).

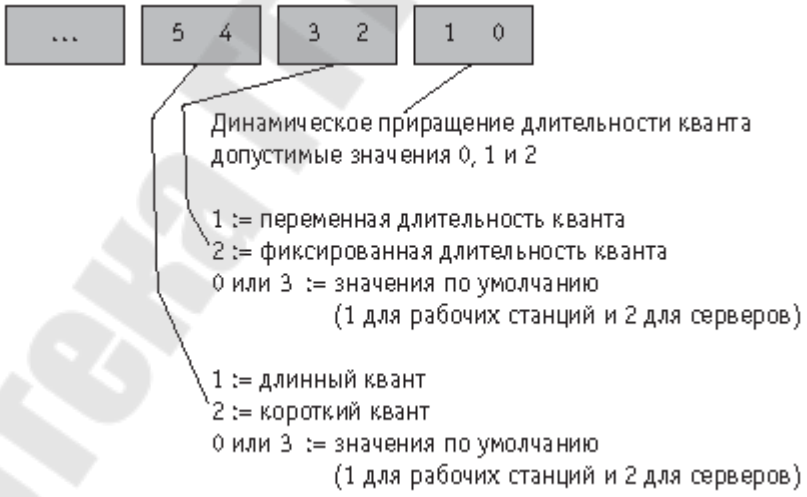

Рис.2. Управление квантованием в ОС Windows

Параметр реестра HKLM\SYSTEM\CurrentControlSet\Control\PriorityControl\Win32PrioritySeparation предназначен для управления квантованием. На рис. 2 дан формат этого параметра для Windows 2000-2003, а в таблице 1 приводятся длительности квантов в условных единицах для разных значений полей параметра Win32PrioritySeparation.

Таблица 1.

	Короткий квант			**Длинный квант**		
Значение младших 2-х бит параметра Win32PrioritySeparation	**0**	**1**	**2**	**0**	**1**	**2**

Переменная длительность	6	12	18	12	24	36
Фиксированная длительность	18	18	18	36	36	36

Этот параметр может быть изменен с помощью панели управления, однако, лишь в очень ограниченных рамках: "System Properties->Advanced->Performance:Settings->Advanced->Adjust for best performance of:" позволяет выбрать только:

- "applications": короткие кванты переменной длины, значение 0x26 т.е. 10 01 10 (короткие кванты переменной длительности, 18 ед.).
- "background services": длинные кванты фиксированной длины, значение 0x18 т.е. 01 10 00 (длинные кванты фиксированной длительности, 36 ед.). Более тонкая настройка возможна с помощью редактора реестра.

Управление длительностью кванта связано с активностью процесса, которая определяется наличием интерфейса пользователя (GUI или консоль) и его активностью. Если процесс находится в фоновом режиме, то длительность назначенного ему кванта соответствует "нулевым" колонкам таблицы 1 (т.е. длительности 6 или 24 - для переменной длины кванта или 18 и 36 - для фиксированной). Когда процесс становится активным, то ему назначается продолжительность квантов, исходя из значения двух младших бит параметра Win32PrioritySeparation в соответствии с приведенной таблицей.

Еще один случай увеличения длительности кванта – процесс долгое время не получал процессорного времени (это может случиться, если все время есть активные процессы более высокого приоритета). В этой ситуации система раз в 3-4 секунды (в зависимости от продолжительности тика) назначает процессу повышенный приоритет и квант удвоенной длительности. По истечении этого кванта приоритет возвращается к прежнему значению и восстанавливается рекомендуемая длительность кванта.

Управление приоритетами

В Windows выделяется 32 уровня приоритетов:

- Приоритет 0 - соответствует приоритету потока обнуления страниц.
- Приоритеты с 1 по 15 - соответствуют динамическим уровням приоритетов. Большинство потоков работают именно в этом диапазоне приоритетов, и Windows может корректировать в некоторых случаях приоритеты потоков из этого диапазона.
- Приоритеты с 16 по 31 - соответствуют приоритетам "реального времени". Этот уровень достаточно высок для того, чтобы поток, работающий с таким приоритетом, мог реально помешать нормальной работе других потоков в системе – например, помешать обрабатывать сообщения от клавиатуры и мыши. Windows самостоятельно не корректирует приоритеты этого диапазона.

Для некоторого упрощения управления приоритетами в Windows выделяют "классы приоритета" (priority class), которые задают базовый уровень приоритета, и "относительные приоритеты" потоков, которые корректируют указанный базовый уровень. Операционная система предоставляет набор функций для управления классами и относительными приоритетами потоков. Планировщик операционной системы также может корректировать уровень приоритета (из диапазона 1-15), однако базовый уровень (т.е. класс) не может быть изменен. Такая коррекция приоритета выполняется в случае:

- завершения операции ввода-вывода – в зависимости от устройства, приоритет повышается на 1 - 8 уровней;
- по окончании ожидания события или семафора – на один уровень;
- при пробуждении GUI потоков – на 2 уровня;

– по окончании ожидания потоком активного процесса (определяется по активности интерфейса) – на величину, указанную младшими 2 битами параметра Win32PrioritySeparation.

В случае коррекции приоритета по одной из перечисленных причин, повышенный приоритет начинает постепенно снижаться до начального уровня потока - с каждым тиком таймера на один уровень. Еще один случай повышения приоритета (вместе с увеличением длительности кванта) - процесс долгое время не получал процессорного времени. В этой ситуации система раз в 3-4 секунды назначает процессу приоритет, равный 15, и квант удвоенной длительности. По истечении этого кванта приоритет возвращается к прежнему значению и восстанавливается рекомендуемая длительность кванта.

Разработчик приложения может изменять класс приоритета, назначенный процессу, и относительный приоритет потоков в процессе в соответствии с приведенными таблицами 2 и 3. Планировщик при выборе потока для исполнения учитывает итоговый уровень приоритета.

Таблица 2

Классы приоритетов

Класс	**Флаг в функции CreateProcess**	**Базовый уровень приоритета**
Idle	IDLE_PRIORITY_CLASS	4
Below normal	BELOW_NORMAL_PRIORITY_CLASS	6
Normal	NORMAL_PRIORITY_CLASS	8
Above normal	ABOVE_NORMAL_PRIORITY_CLASS	10
High	HIGH_PRIORITY_CLASS	13
Realtime	REALTIME_PRIORITY_CLASS	24

Таблица 3

Относительные приоритеты потоков

Относительный приоритет	**Флаг в функции SetThreadPriority**	**Описание**
Idle	THREAD_PRIORITY_IDLE	Для процессов класса Realtime приоритет потока равен 16, для процессов остальных классов
Lowest	THREAD_PRIORITY_LOWEST	Приоритет потока меньше базового приоритета на 2
Below normal	THREAD_PRIORITY_BELOW_NORMAL	Приоритет потока меньше базового приоритета на 1
Normal	THREAD_PRIORITY_NORMAL	Приоритет потока равен базовому приоритету
Above normal	THREAD_PRIORITY_ABOVE_NORMAL	Приоритет потока больше базового приоритета на 1
Highest	THREAD_PRIORITY_HIGHEST	Приоритет потока больше базового приоритета на 2
Time critical	THREAD_PRIORITY_TIME_CRITICAL	Для процессов класса Realtime приоритет потока равен 31, для процессов остальных классов

Виды приложений с параллельным выполнением операций.

Асинхронный ввод-вывод

Обычные операции ввода-вывода происходят в синхронном режиме. Например, в приведенном ниже примере все операции выполняются строго последовательно: файл открывается, вызывается операция чтения данных, и только после того как все данные прочитаны, продолжается выполнение задачи:

```
#include <stdio.h>
int sequential_io( char *filename, char *buffer, int size )
{
  FILE *fp;
  int  done;
  fp = fopen( filename, "rb" );
  if ( fp ) {
    done = fread( fp, 1, size, buffer );
    /* код не будет выполняться, пока чтение не завершится*/
    fclose( fp );
  } else {
    done = 0;
  }
  buffer[done] = '\0';
  return done;
}
```

Такой подход прост в реализации, как с точки зрения операционной системы, так и с точки зрения пользователя. Однако во время выполнения операций ввода-вывода сама программа не выполняется - она ожидает завершения ввода-вывода. Во многих случаях такие операции, во-первых, занимают достаточно много времени и, во-вторых, выполняются специализированным оборудованием без участия центрального процессора, который в это время находится в состоянии ожидания и не выполняет никакой полезной работы (по крайней мере, с точки зрения данной программы, так как в общем случае начало ожидания приводит к перепланировке потоков).

Эффективность использования процессора можно было бы повысить, если бы существовала возможность выполнять код программы во время выполнения операций ввода-вывода. Конечно, это не всегда возможно или целесообразно. Например, если для продолжения работы программы необходимы данные, которые еще не получены, то нам все равно надо ожидать завершения ввода-вывода. Более того, структура приложения должна быть разработана с учетом специфики использования асинхронных операций ввода-вывода. В Windows для реализации асинхронного ввода-вывода предусмотрены функции типа ReadFile, WriteFile, ReadFileEx, WriteFileEx и др. и специальная структура OVERLAPPED, которая используется для взаимодействия с асинхронной операцией.

Асинхронные операции применяются следующим образом:

- перед началом операции заполняется структура OVERLAPPED;
- вызывается нужная функция для выполнения ввода-вывода, которая ставит операцию в очередь и немедленно возвращает управление вызвавшей программе.
- программа продолжает свою работу независимо от хода выполнения операции ввода-вывода.

При необходимости можно выяснить состояние асинхронной операции, дождаться ее завершения или отменить ее, не дожидаясь завершения. Для этого предназначен специальный набор функций, например, CancelIo, GetOverlappedResult, HasOverlappedIoCompleted и некоторые другие. Существует несколько вариантов использования асинхронного ввода-вывода; рассмотрим их на небольшом примере.

Для начала надо описать необходимые переменные и открыть файл с разрешением асинхронных операций (FILE_FLAG_OVERLAPPED):

```
OVERLAPPED  ov;
DWORD       dwWritten;
BYTE        buffer[ 5000000 ];

HANDLE fh = CreateFile(
  "file.dat", FILE_READ_DATA|FILE_WRITE_DATA,
  FILE_SHARE_READ, (LPSECURITY_ATTRIBUTES)NULL, OPEN_ALWAYS,
  FILE_ATTRIBUTE_NORMAL|FILE_FLAG_OVERLAPPED, NULL
);
if ( fh == INVALID_HANDLE_VALUE ) {
  /* возникла ошибка */
}

ZeroMemory( &ov, sizeof(OVERLAPPED) );
FillMemory( buffer, sizeof(buffer), 123 );
```

Выполнение асинхронных операций с опросом состояния; этот способ может обеспечить самую быструю реакцию на завершение операции ввода-вывода, но ценой более высокой загрузки процессора:

```
ov.Offset = 12345;
if (
  WriteFile( fh, buffer, sizeof(buffer), &dwWritten, &ov ) ||
  GetLastError() == ERROR_IO_PENDING
) {
  /* пока операция ввода-вывода выполняется,
    выполняем некоторые операции.
    дожидаемся завершения операции ввода-вывода */
  while (!GetOverlappedResult(fh, &ov, &dwWritten, FALSE)){}
} else {
  /* возникла ошибка */
}
```

Функция GetOverlappedResult в данном случае проверяет состояние операции ввода-вывода и возвращает признак ее завершения. Опрос состояния операции в цикле позволяет наиболее быстро отреагировать на завершение операции (особенно на многопроцессорных машинах), однако ценой увеличения загрузки процессора, что может снизить общую производительность системы.

Функция WriteFile возвращает значение TRUE, если операция записи завершена синхронно, а в случае ошибки или начатой асинхронной операции она возвращает FALSE, поэтому требуется анализ кода возникшей "ошибки", которая может и не являться ошибкой.
Выполнение асинхронных операций с ожиданием на объектах ядра:

```
ov.Offset = 12345;
ov.hEvent    =    CreateEvent((LPSECURITY_ATTRIBUTES)NULL,    TRUE,
FALSE, 0);

if (
  WriteFile( fh, buffer, sizeof(buffer), &dwWritten, &ov ) ||
  GetLastError() == ERROR_IO_PENDING
) {
  /* пока операция ввода-вывода выполняется, выполняем некоторые
          операции...
          дожидаемся завершения операции ввода-вывода */
  GetOverlappedResult( fh, &ov, &dwWritten, TRUE );
} else {
  /* возникла ошибка */
}
```

Функция GetOverlappedResult в данном случае проверяет состояние операции и, если она еще не завершена, вызывает функцию WaitForSingleObject для ожидания завершения операции ввода-вывода. Объект "событие" можно не создавать – в этом случае функция будет ожидать освобождения объекта "файл"; однако в случае нескольких, накладывающихся друг на друга, асинхронных операций будет непонятно, какая именно операция завершилась, и использование специфичных для каждой операции событий снимает эту проблему.

Ожидание на объектах ядра является наиболее экономным, но реакция приложения на завершение ввода-вывода связана с работой планировщика, поэтому для достижения малых задержек иногда надо дополнительно повышать приоритеты потоков, переходящих в режим ожидания завершения ввода-вывода.

Выполнение асинхронных операций с использованием функций завершения операций ввода-вывода.

Эта функция будет автоматически вызвана после завершения ввода-вывода и может выполнить некоторые специальные действия. Процедура завершения обязательно вызывается в контексте того потока, который вызвал операцию ввода-вывода, а для этого необходимо, чтобы поток был приостановлен, так как операционная система не должна прерывать работу активного потока. Следует перевести поток в состояние ожидания оповещения (alertable waiting) – при этом он не выполняется и может быть прерван для обработки процедуры завершения:

```
ov.Offset = 12345;
if ( WriteFileEx( fh, buffer, sizeof(buffer), &ov, io_done ) ) {
  /* пока операция ввода-вывода выполняется, выполняем некоторые
        операции и переходим в режим ожидания оповещения,
        например, так: */
  if ( SleepEx( INFINITE, TRUE ) != WAIT_IO_COMPLETION ) {
    /* ввод-вывод пока не завершен, возможно, ошибка */
  }
} else {
    /* возникла ошибка */
}
/*   нам еще надо предоставить собственную процедуру
```

```
        завершения ввода-вывода. В простейшем варианте
        она может ничего не делать: */
VOID CALLBACK io_done(
  DWORD dwErr, DWORD dwWritten, LPOVERLAPPED lpOv
) {
  ...
}
```

Этот подход наиболее трудоемок и наименее распространен; на практике самым эффективным является механизм выполнения асинхронных операций с ожиданием на объектах ядра. Однако механизм вызова функций завершения (расширенный возможностью автоматического выбора потока, осуществляющего обработку функции завершения) послужил основой для реализации одного из очень эффективных механизмов взаимодействия потоков – порта завершения ввода-вывода. При использовании асинхронного ввода-вывода необходимо очень внимательно следить за выделением и освобождением ресурсов – особенно памяти, занятой структурами OVERLAPPED, и буферами, участвующими в операциях ввода-вывода. Асинхронный ввод-вывод является примером мультипроцессирования с использованием функционально различных устройств.

Асинхронные вызовы процедур

Для реализации асинхронного ввода-вывода в операционной системе предусмотрен специальный механизм, основанный на так называемых асинхронных вызовах процедур (Asynchronous Procedure Call, APC). Это один из базовых механизмов, необходимый для нормального функционирования операционной системы. Практика показала, что такой механизм был бы эффективен и для реализации самих приложений. Более того, для реализации асинхронного ввода-вывода с поддержкой функции завершения система уже обязана была предоставить этот механизм. Для реализации этого механизма операционная система ведет списки процедур, которые она должна вызывать в контексте данного потока, с тем ограничением, что прерывать работу занятого потока в произвольный момент времени система не должна. Поэтому для обслуживания накопившихся в очереди процедур необходимо перевести поток в специальное состояние ожидания оповещения (alertable waiting) – для этого Win32 API предусматривает специальный набор функций: например, SleepEx, WaitForSingleObjectEx и др.

```
VOID CALLBACK ApcProc( ULONG_PTR dwData )
{
  /* ... */
}
int main( void )
{
  QueueUserAPC( ApcProc, GetCurrentThread(), 0 );
  /* ... */
  SleepEx( 1000, TRUE );
  return 0;
}
```

Обычно APC используются самой системой для реализации асинхронного ввода-вывода и некоторых других операций, но разработчикам также предоставлена функция QueueUserAPC, с помощью которой можно ставить в APC очередь запросы для вызова собственных функций.

Процессы, потоки и объекты ядра

В мультипрограммной среде может одновременно выполняться значительное количество процессов, представляющих разных пользователей, и ядро операционной системы вынуждено проверять полномочия каждого процесса при их попытках доступа к защищаемым объектам. Для того чтобы ядро операционной системы могло контролировать доступ к тем или иным объектам, сами объекты должны управляться ядром системы. Это приводит к понятию объектов ядра (kernel objects), которые создаются по запросу процессов ядром системы, управляются ядром, и доступ к которым также контролируется ядром системы.

Объекты ядра

Объекты ядра представлены в качестве некоторых структур и типов данных, управляемых ядром операционной системы и размещенных в памяти, принадлежащей ядру. Пользовательские процессы, как правило, не имеют возможности доступа к этим объектам напрямую. Для того чтобы пользовательский процесс мог оперировать такими объектами, введено понятие описатель (handle) объекта ядра. Так, например, объектами ядра являются файлы, процессы, потоки, события, почтовые ящики и многое другое.

Все созданные описатели объектов ядра должны удаляться с помощью функции

```
BOOL CloseHandle( HANDLE hKernelObject )
```

которая уменьшает счетчик использования объекта и уничтожает его, когда на объект никто больше не ссылается.

Доступ к защищаемым объектам в Windows задается так называемыми дескрипторами безопасности (Security Descriptor). Дескриптор содержит информацию о владельце объекта и первичной группе пользователей и два списка управления доступом (ACL, Access Control List): один список задает разрешения доступа, другой - необходимость аудита при доступе к объекту. Список содержит записи, указывающие права выполнения действий, и запреты, назначенные конкретным пользователям и группам. При доступе к защищаемым объектам для начала проверяются запреты - если для данного пользователя и группы имеется запрет доступа, то дальнейшая проверка не выполняется и попытка доступа отклоняется. Если запретов нет, то проверяются права доступа - при отсутствии разрешений доступ отклоняется. Запрет обладает более высоким "приоритетом", чем наличие разрешений - это позволяет разрешить доступ, к примеру, целой группе пользователей и выборочно запретить некоторым ее членам. Объект, осуществляющий доступ (выполняющийся поток), обладает так называемым маркером доступа (access token). Маркер идентифицирует пользователя, от имени которого предпринимается попытка доступа, а также его привилегии и умолчания (например, стандартный ACL объектов, создаваемых этим пользователем). В Windows маркерами доступа обладают как потоки, так и процессы. С процессом связан так называемый первичный маркер доступа, который используется при создании потоков, а вот в дальнейшем поток может работать от имени какого-либо иного пользователя, используя собственный маркер воплощения (impersonation).

Процессы и потоки в Windows являются с одной стороны "представителями" пользователя, выступающими от его имени, а с другой стороны – защищаемыми объектами, при доступе к которым выполняется проверка прав, то есть они обладают одновременно и маркерами доступа, и дескрипторами безопасности.

Описатели объектов ядра в Windows позволяют разным процессам и потокам взаимодействовать с объектами с учетом их контекстов безопасности, располагаемых прав и требуемого режима доступа. Примерами защищаемых объектов являются процессы, потоки, файлы, большинство синхронизирующих примитивов (события, семафоры и т.д.), проекции файлов в память и многое другое. Для создания большинства объектов ядра используются функции, начинающиеся на слово "Create" и возвращающие описатель созданного объекта, например функции CreateFile, CreateProcess, CreateEvent и т.д. Многие объекты при их создании могут получить собственное имя или остаться неименованными.

Любой процесс или поток может ссылаться на объекты ядра, созданные другим процессом или потоком. Для этого предусмотрено три механизма:

1) Объекты могут быть унаследованы дочерним процессом при его создании. В этом случае объекты ядра должны быть "наследуемыми", и родительский процесс должен принять меры к тому, чтобы потомок мог узнать их описатели. Возможность передавать описатель потомкам по наследованию явно указывается в большинстве функций, так или иначе создающих объекты ядра (обычно такие функции содержат аргумент "BOOL bInheritHandle", который указывает возможность наследования).
2) Объект может иметь собственное уникальное имя - тогда можно получить описатель этого объекта по его имени. Для разных типов объектов Win32 API предоставляет набор функций, начинающийся на Open... например, OpenMutex, OpenEvent и т.д.
3) Процесс-владелец объекта может передать его описатель любому другому процессу. Для этого процесс-владелец объекта должен получить специальный описатель объекта для "экспорта" в указанный процесс. В Win32 API для этого предназначена функция DuplicateHandle, создающая для объекта, заданного описателем в контексте данного процесса, новый описатель, корректный в контексте нового процесса:

```
BOOL DuplicateHandle(
  HANDLE hFromProcess, HANDLE hSourceHandle,
  HANDLE hToProcess, LPHANDLE lpResultHandle,
  DWORD dwDesiredAccess, BOOL bInheritHandle, DWORD
dwOptions
);
```

Существенно проследить, чтобы все создаваемые описатели закрывались вызовом функции CloseHandle, включая описатели, созданные функцией DuplicateHandle. Хорошая практика при разработке приложений - проводить мониторинг выделяемых описателей и количества объектов в процессе (например, с помощью таких стандартных средств как менеджер задач или оснастка "производительность" панели управления).

Описатели процесса и потока

Для взаимодействия потоков и процессов между собой необходимы средства, обеспечивающие идентификацию соответствующих объектов. В Windows для идентификации процессов и потоков используют их описатели (HANDLE) и идентификаторы (DWORD). Описатели идентифицируют в данном случае объект ядра, представляющий процесс или поток, при доступе к которому, как ко всякому объекту ядра, учитывается контекст защиты, проверяются права доступа и т.д.

Идентификаторы процесса и потока, назначаемые при их создании, исполняют роль уникальных имен. Описатели и идентификаторы процессов и потоков можно получить при создании соответствующих объектов. Кроме того, можно узнать идентификаторы текущего процесса и потока (GetCurrentThreadId, GetCurrentProcessId), или по описателю узнать соответствующий идентификатор (GetProcessId и GetThreadId). Функции OpenProcess и OpenThread позволяют получить описатели этих объектов по их идентификатору.

Функции GetCurrentProcess и GetCurrentThread возвращают описатели текущего процесса и потока, однако возвращаемое ими значение не является настоящим описателем, а представлено некоторой константой, получившей название "псевдоописатель". Эта константа, использованная вместо описателя потока или процесса, рассматривается как описатель процесса/потока, сделавшего вызов системной функции. Псевдоописателями можно свободно пользоваться в рамках процесса (потока), в котором они получены, а при попытке передать их другому процессу или потоку они будут рассматриваться как описатели того процесса (потока), в контексте которого используются.

В тех случаях, когда необходимо дать другому процессу или потоку доступ к данным описателям, нужно с помощью DuplicateHandle сделать с них "копии", которые будут являться настоящими описателями в контексте процесса-получателя. Так, например, с помощью этой функции можно превратить псевдоописатель процесса в настоящий описатель, действующий только в текущем процессе:

```
HANDLE hrealThread;
DuplicateHandle(
  GetCurrentProcess(), GetCurrentProcess(),
  GetCurrentProcess(), &hrealThread,
  DUPLICATE_SAME_ACCESS, FALSE, 0
);
```

У процессов и потоков есть интересная особенность – объекты ядра, представляющие процесс и поток, сразу после создания имеют счетчик использования не менее двух: во-первых, это описатель, возвращенный функцией, и, во-вторых, объект используется работающим потоком. В итоге завершение потока и завершение последнего потока в процессе не приводят к удалению соответствующих объектов - они будут сохраняться все время, пока существуют их описатели. Это сделано для того, чтобы уже после завершения работы потока или процесса можно было получить от него какую-либо информацию, чаще всего – код завершения (функции GetExitCodeThread и GetExitCodeProcess);

Объекты ядра "процесс" и "поток" поддерживают также интерфейс синхронизируемых объектов, так что их можно использовать для синхронизации работы: поток считается занятым до завершения, а процесс занят до тех пор, пока в нем есть хоть один работающий поток. Если ни синхронизация с этими объектами, ни получение кодов завершения не требуются разработчику, надо сразу после создания соответствующего объекта закрывать его описатель.

В Windows для задания приоритета работающего потока используют понятия классов приоритетов и относительных приоритетов в классе. При этом класс приоритета связывается с процессом, а относительный приоритет - с потоком, исполняющимся в данном процессе.

Соответственно Win32 API предоставляет функции для изменения класса приоритета для процесса (GetPriorityClass, SetPriorityClass) и для изменения относительного приоритета потока (GetThreadPriority и SetThreadPriority).

Планировщик операционной системы может динамически корректировать приоритет потока, кратковременно повышая уровень. Разработчикам предоставлена возможность отказаться от этой возможности или, наоборот, задействовать ее (функции GetProcessPriorityBoost, SetProcessPriorityBoost, GetThreadPriorityBoost и SetThreadPriorityBoost).

Основы использования потоков

Для реализации мультипрограммирования или мультипроцессирования на однотипных устройствах можно применять процессы, потоки и волокна. Разница между ними связана с возможностью обмена данными: адресное пространство процессов изолировано друг от друга, поэтому взаимодействие затруднено; потоки находятся в общем адресном пространстве процесса и могут легко взаимодействовать друг с другом; волокна отчасти аналогичны потокам, но планирование волокон выполняется непосредственно приложением, а не операционной системой.

Потоко-безопасные и небезопасные функции

При реализации многопоточного приложения следует учитывать возможные побочные эффекты. Наличие таких эффектов обусловлено реализацией библиотеки времени исполнения: она содержит много функций (в том числе внутренних, обеспечивающих семантику языка программирования), являющихся потоко-небезопасными. Примеры таких функций - стандартная процедура strtok, операторы new и delete или функции malloc, calloc, free и так далее. Фактически любая стандартная функция, оперирующая статическими переменными, объектами или данными, может являться потоко-небезопасной в силу того, что два потока могут получить одновременный конкурирующий доступ к этим данным и в итоге разрушить их. Существует несколько подходов к решению этой проблемы:

- Можно предоставить потоко-безопасные аналоги (например, strtok_r, являющийся в Linux потоко-безопасным аналогом функции strtok).
- Можно переписать код всех потоко-небезопасных функций так, чтобы они вместо глобальных объектов использовали локальную для потока память или синхронизировали доступ к общим данным.

В Windows принят второй подход, однако, с некоторой оговоркой. Потоко-безопасные версии функций более ресурсо- и время- емкие, чем обычные. В итоге используется два вида библиотек: один для однопоточных приложений, другой для многопоточных. Следует отметить, что выбор той или иной библиотеки определяется, как правило, свойствами проекта (параметрами компилятора), а вовсе не кодом приложения. Поэтому при разработке многопотокового приложения важно проследить, чтобы при компиляции использовалась правильная версия библиотеки, во избежание возникновения трудно диагностируемых ошибок, проявляющихся в самых разных и совершенно "невинных" на первый взгляд местах кода. В случае Visual Studio однопоточные версии библиотек выбираются ключами /ML или /MLd компилятора, а многопоточные ключами /MT, /MD, /MTd или /MDd (свойства проекта Configuration Properties|C/C++|Code Generation|Runtime Library).

Работа с потоками

Наличие специальных потоко-безопасных версий библиотек требует использования специальных функций для создания и завершения потоков, принадлежащих не системному API, а библиотеке времени исполнения. Так, вместо функций Win32 API CreateThread, ExitThread необходимо использовать библиотечные функции _beginthread, _endthread или _beginthreadex, _endthreadex. Это требование связано с тем, что при создании нового потока необходимо, помимо выполнения определенных действий по созданию потока со стороны операционной системы, инициализировать специфичные структуры данных, обслуживающих потоко-безопасные версии функций библиотеки времени исполнения:

```
unsigned _ _stdcall ThreadProc( void *param )
{
  /* вновь созданный поток будет выполнять эту функцию */
  Sleep( 1000 );
  delete[] (int*)param;
  return 0;
  /* завершение функции = завершение потока */
}

int main( void )
{
  HANDLE   hThread;
  unsigned  dwThread;
  /* создаем новый поток */
  hThread = (HANDLE)_beginthreadex (
    NULL, 0, ThreadProc, new int [128], 0, &dwThread
  );
  /* код в этом месте может выполняться
     одновременно с кодом функции потока ThreadProc,
     планирование потоков осуществляется системой
  */
  /* дождаться завершения созданного потока */
  WaitForSingleObject( hThread, INFINITE );
  CloseHandle( hThread );
  return 0;
}
```

В данном примере можно было бы создавать поток не вызовом функции _beginthreadex (или _beginthread), а вызовом функции API CreateThread. Но при незначительном усложнении примера, скажем, создании не одного, а двух потоков, уже было бы возможно возникновение ошибки при одновременном обращении к операторам new или delete в разных потоках (причем именно "возможно", так как ничтожные временные задержки могут изменить поведение потоков - это крайне осложняет выявление таких ошибок). Применение функций библиотеки времени исполнения для создания потоков решает эту проблему.

Windows содержит достаточно богатый набор функций для управления потоками, включающий функции создания и завершения потоков (функции API CreateThread, ExitThread, TerminateThread и их "обертки" в библиотеке времени исполнения _beginthread, _endthread, _beginthreadex и _endthreadex).

Функция Sleep(DWORD dwMilliseconds) может переводить поток в "спячку" на заданное время. Продолжительность задается с точностью до кванта работы планировщика, то есть не лучше, чем 10-15 мс, несмотря на то, что при вызове функции задать можно до 1 мс. Измерение времени реальной паузы, заданной, например, вызовом Sleep(1), позволяет получить косвенную информацию о работе планировщика.

В Windows существует интересная особенность, связанная с работой планировщика и измерением интервалов времени. Система предоставляет три способа измерения интервалов:

- таймер низкого разрешения, основанный на квантах планировщика (GetTickCount);
- "мультимедийный", с разрешением до 1 мс (timeGetTime, timeBeginPeriod и пр.);
- высокоточный, использующий счетчик тактов процессора и с разрешением ощутимо лучше микросекунды на современных процессорах (QueryPerformanceCounter, QueryPerformanceFrequency).

Обычно мультимедийный таймер работает с разрешением от 1-5 мс и хуже (зависит от аппаратуры), однако функция timeBeginPeriod позволяет изменить разрешение вплоть до 1 мс. Если стандартное разрешение мультимедийного таймера на данном компьютере хуже 5-10 мс, то у функции timeBeginPeriod есть побочный эффект - улучшение разрешения повлияет на работу планировщика во всей системе, а не только в процессе, вызвавшем эту функцию. В результате, если один процесс повысит разрешение мультимедийного таймера, то функция Sleep также получит возможность задавать интервалы вплоть до 1 мс и эффект наблюдается даже в других процессах. Если мультимедийный таймер на данной аппаратуре стандартно работает с разрешением порядка 1 мс, то такого влияния на планировщик не наблюдается.

Есть частный случай применения функции Sleep - при задании интервала 0 вызов функции просто приводит к срабатыванию планировщика и, при наличии других готовых потоков, к их активации. Аналогичного эффекта можно добиться, применяя функцию SwitchToThread, вызывающую перепланирование потоков.

Поток может быть создан в приостановленном (suspended) состоянии с помощью задания специального флага CREATE_SUSPENDED при вызове функций _beginthreadex или CreateThread, а также переведен в это состояние (функция SuspendThread) или, наоборот, пробужден с помощью функции ResumeThread.

Работа с волокнами

Работа с волокнами в приложении в чем-то сложнее, в чем-то проще. Сложнее, потому что необходимо реализовать собственный планировщик волокон. Сложность разработки планировщика резко возрастает при необходимости синхронизации волокон - стандартные средства синхронизации Windows переводят в режим ожидания поток целиком, даже если он должен планировать множество волокон. Проще, потому что все волокна могут разделять один поток - в этом случае легко избежать проблем конкурирующего доступа к данным и можно применять любую библиотеку времени исполнения, в том числе потоко-небезопасную.

При работе с волокнами используется функция ConvertThreadToFiber для предварительного создания необходимых операционной системе структур данных. Функция ConvertFiberToThread выполняет обратную задачу и уничтожает выделенные данные. После того как необходимые структуры созданы (поток "превращен" в волокно),

появляется возможность создавать новые волокна (CreateFiber), удалять существующие (DeleteFiber) и планировать их исполнение (SwitchToFiber).

Приведем пример применения двух рабочих волокон, выполняющих целевую функцию, и одного управляющего, удаляющего рабочие волокна по их завершении.

Функция main превращает текущий поток в волокно (инициализация внутренних структур данных для работы с волокнами), затем создает рабочие волокна и организует цикл, в котором ожидает их завершения и удаляет. Цикл завершается тогда, когда все рабочие волокна удалены, после чего функция main принимает меры к корректному завершению работы с волокнами. Собственно целевая функция FiberProc эпизодически вызывает функцию SwitchToFiber для переключения выполняемого волокна. В данном примере для определения нового волокна, подлежащего исполнению, реализован простейший планировщик (функция schedule, инкапсулирующая вызов функции SwitchToFiber).

```
#define _WIN32_WINNT 0x0400
#include <windows.h>

#define FIBERS 2

static LPVOID  fiberEnd;
static LPVOID  fiberCtl;
static LPVOID  fiber[ FIBERS ];

static void shedule( BOOL fDontEnd )
{
  int     n, current;
  if ( !fDontEnd ) {  /* волокно надо завершить */
    fiberEnd = GetCurrentFiber();
    SwitchToFiber(fiberCtl );
  }
  /* выбираем следующее волокно для выполнения */
  for ( n = 0; n < FIBERS; n++ ) {
    if ( fiber[n] && fiber[n] != GetCurrentFiber() ) break;
  }
  if ( n >= FIBERS ) return;  /* нет других готовых волокон*/
  SwitchToFiber( fiber[n] );
}

VOID CALLBACK FiberProc( PVOID lpParameter )
{  /* волокно будет выполнять код этой функции */
  int  i;
  for ( i = 0; i < 100; i++ ) {
    Sleep( 1000 );
    shedule( TRUE );  /* выполнение продолжается */
  }
  shedule( FALSE ); /* волокно завершается */
}

int main( void )
{
  int  i;
  fiberCtl = ConvertThreadToFiber( NULL );
```

```
  fiberEnd = NULL;
  for ( i = 0; i < FIBERS; i++ ) {
    fiber[i] = CreateFiber( 10000, FiberProc, NULL );
  }
  for ( i = 0; i < FIBERS;) {
    SwitchToFiber( fiber[i] );
    if ( fiberEnd ) {
      DeleteFiber(fiberEnd );
      for ( i = 0; i < FIBERS; i++ ) {
        if ( fiber[i] == fiberEnd ) fiber[i] = NULL;
      }
      fiberEnd = NULL;
    }
    for ( i = 0; i < FIBERS; i++ ) if ( fiber[i] ) break;
  }
  ConvertFiberToThread();
  return 0;
}
```

Следует еще раз отметить одну важную особенность волокон - они работают в рамках одного потока и не позволяют задействовать возможности мультипроцессирования. Если нужно обеспечить параллельное исполнение кода на разных процессорах, то надо применять потоки либо даже отдельные процессы. В частных случаях возможно создание гибридных вариантов - когда несколько потоков выполняют несколько волокон; при этом число волокон может существенно превышать число потоков. Однако и в этом случае целесообразность применения волокон должна быть тщательно изучена: очень часто эффективнее не выполнять одновременно много мелких заданий, а выполнять их поочередно - тогда уменьшатся затраты на переключения и, возможно, возрастет утилизация кэша. Волокна представляют, по большей части, теоретический интерес, как возможность реализовать планировщик пользовательского режима, помимо существующего стандартного планировщика режима ядра.

Пулы потоков, порт завершения ввода-вывода

Одна из типовых задач – разработка серверов, обслуживающих асинхронно поступающие запросы. Реализация однопоточного сервера для такой задачи нецелесообразна, во-первых, потому что запросы могут приходить в то время, пока сервер занят выполнением предыдущего, а, во-вторых, потому что такой сервер не сможет эффективно задействовать многопроцессорную систему. Можно, конечно, запускать несколько экземпляров однопоточного сервера - но в этом случае потребуется разработка специального диспетчера, поддерживающего очередь запросов и их распределение по списку доступных экземпляров сервера. Альтернативным решением является разработка многопоточного сервера, создающего по специальному рабочему потоку для обработки каждого запроса. Этот вариант также имеет свои недостатки: создание и удаление потоков требует затрат времени, которые будут иметь место в обработке каждого запроса; сверх того, создание большого числа одновременно выполняющихся потоков приведет к общему снижению производительности (и значительному увеличению времени обработки каждого конкретного запроса).

Эти соображения приводят к решению, получившему название пула потоков (thread pool). Для реализации пула потоков необходимо создание некоторого количества потоков, занятых обслуживанием запросов, и диспетчера с очередью запросов. При наличии необработанных запросов диспетчер находит свободный поток и передает запрос этому

потоку; если свободных потоков нет, то диспетчер ожидает освобождения какого-либо из занятых потоков. Такой подход обеспечивает, с одной стороны, малые затраты на управление потоками, с другой - достаточно высокую загрузку процессоров и хорошую масштабируемость приложения.

Обычно имеет смысл ограничивать общее число рабочих потоков либо числом доступных процессоров, либо кратным этому числом. Если потоки занимаются исключительно вычислительной работой, то их число не должно превышать число процессоров. Если потоки, сверх того, проводят некоторое время в состоянии ожидания (например, при выполнении операций ввода-вывода), то число потоков следует увеличить - решение следует принимать, исходя из доли времени, которое поток проводит в состоянии простоя, и из полного времени обработки запроса.

Достаточно типичная рекомендация: ограничивать число потоков удвоенным числом процессоров. В случае вычислительных потоков накладные потери будут достаточно малы; в случае потоков, занятых вводом-выводом, утилизация процессоров будет близка к полной. Предполагается, что потоки, не занятые вводом-выводом и при этом проводящие много времени в состоянии ожидания, встречаются весьма редко. Реализация пула потоков является, на самом деле, нетривиальной задачей – необходимо поддерживать очередь запросов и учитывать состояния потоков из пула (поток простаивает; поток выполняется; поток выполняется, но находится в состоянии ожидания). Также следует учитывать возможность выгрузки части данных в файл подкачки (например, выгрузка стека давно не используемого потока) - иногда быстрее подождать завершения работающего потока, чем активировать простаивающий. Для учета всех этих соображений необходимо реализовать поддержку пулов потоков ядром операционной системы, так как на уровне приложения некоторые нужные сведения просто недоступны. В Windows такая поддержка реализована в виде порта завершения ввода-вывода. Этот объект ядра берет на себя функциональность, необходимую для организации очереди запросов (используя для этого очередь APC) и списков рабочих потоков, обеспечивая оптимальное управление пулом.

С точки зрения разработчика приложения необходимо:

1) создать порт завершения ввода-вывода;
2) создать пул потоков, ожидающий поступления запросов от этого порта;
3) обеспечить передачу запросов порту.

Порт завершения создается с помощью функции

```
HANDLE CreateIoCompletionPort(
  HANDLE FileHandle, HANDLE ExistingCompletionPort,
  ULONG_PTR CompletionKey, DWORD NumberOfConcurrentThreads
);
```

Эта функция выполняет две разных операции - во-первых, она создает новый порт завершения, и, во-вторых, она ассоциирует порт с завершением операций ввода-вывода с заданным файлом. Обе эти операции могут быть выполнены одновременно одним вызовом функции, а могут быть исполнены раздельно. Более того, вторая операция - ассоциирование порта завершения ввода-вывода с реальным файлом - может вообще не выполняться. Две типичных формы применения функции CreateIoCompletionPort:

Создание нового порта завершения ввода-вывода:

```
#define CONCURRENTS  4

HANDLE   hCP;
hCP = CreateIoCompletionPort(
  INVALID_HANDLE_VALUE, NULL, NULL, CONCURRENTS
);
```

При простом создании порта завершения ввода-вывода достаточно указать только максимальное число одновременно работающих потоков (здесь CONCURRENTS, целесообразно ограничивать числом доступных процессоров). Далее, когда будет создаваться пул потоков, в нем можно будет создать и большее число потоков, чем указано при создании порта - система будет отслеживать, чтобы одновременно исполнялся код не более чем указанного числа потоков. При этом поток, перешедший в состояние ожидания, не считается исполняющимся, так что в случае потоков, проводящих часть времени в режиме ожидания, имеет смысл создавать пул потоков большего размера, чем указано при вызове функции CreateIoCompletionPort.
Ассоциирование порта с файлом:

```
#define SOME_NUMBER  123
CreateIoCompletionPort( hFile, hCP, SOME_NUMBER, 0 );
```

В этом варианте функция CreateIoCompletionPort не создает нового порта, а возвращает переданный ей описатель уже существующего. Существующий порт завершения ввода-вывода можно связать с несколькими различными файлами одновременно; при этом процедура, обслуживающая завершение ввода-вывода, сможет различать, операция с каким именно файлом поставила в очередь данный запрос, с помощью параметра CompletionKey (здесь SOME_NUMBER), назначаемого разработчиком. Созданный порт можно не ассоциировать ни с одним файлом - тогда с помощью функции PostQueuedCompletionStatus надо будет помещать в очередь порта запросы, имитирующие завершение ввода-вывода.

Рассмотрим небольшой пример (проверка ошибок для упрощения пропущена):

```
#include <process.h>
#define _WIN32_WINNT 0x0500
#include <windows.h>

#define MAXQUERIES   15
#define CONCURENTS   3
#define POOLSIZE         5

unsigned _ _stdcall PoolProc( void *arg );

int main( void )
{
  int     i;
  HANDLE  hcport, hthread[ POOLSIZE ];
  DWORD   temp;
  /* создаем порт завершения ввода-вывода */
  hcport = CreateIoCompletionPort(
    INVALID_HANDLE_VALUE, NULL, NULL, CONCURENTS
  );
```

После создания порта надо создать пул потоков. Число потоков в пуле обычно превышает число одновременно работающих потоков, задаваемое при создании порта:

```
/* создаем пул потоков */
for ( i = 0; i < POOLSIZE; i++ ) {
  hthread[i] = (HANDLE)_beginthreadex(
    NULL,0,PoolProc,(void*)hcport,0,(unsigned*)&temp
  );
}
```

Если созданный порт ассоциирован с одним или несколькими файлами, то после завершения асинхронных операций ввода-вывода в очереди порта будут размещаться асинхронные запросы, которые система будет направлять для обработки потокам из пула. Однако порт завершения ввода-вывода можно и не связывать с файлами - тогда для размещения запроса можно воспользоваться функцией PostQueuedCompletionStatus, которая размещает запросы в очереди без выполнения реальных операций ввода-вывода.

```
/* посылаем несколько запросов в порт */
for ( i = 0; i < MAXQUERIES; i++ ) {
  PostQueuedCompletionStatus( hcport, 1, i, NULL );
  Sleep( 60 );
}
```

Функция помещает в очередь запросов информацию о "как будто" выполненной операции ввода-вывода, полностью повторяя аргументы - такие как размер переданного блока данных, ключ завершения и указатель на структуру OVERLAPPED, содержащую сведения об операции. Мы можем передавать вместо этих значений произвольные данные. В данном примере, скажем, принято, что значение ключа завершения -1 совместно с длиной переданного блока 0 означает необходимость завершить поток:

```
/* для завершения работы посылаем специальные запросы */
for ( i = 0; i < POOLSIZE; i++ ) {
  PostQueuedCompletionStatus(hcport,0, (ULONG_PTR)-1,NULL);
}
/* дожидаемся завершения всех потоков пула
     и закрываем описатели */
WaitForMultipleObjects( POOLSIZE, hthread, TRUE, INFINITE );
for ( i = 0; i < POOLSIZE; i++ ) CloseHandle( hthread[i] );
CloseHandle( hcport );
return 0;
}
```

В общем виде поток в пуле реализует цикл с выбором запросов из очереди с помощью функции GetQueuedCompletionStatus. Следует внимательно ознакомиться с описанием этой функции, чтобы грамотно обрабатывать возможные ошибочные ситуации и предусмотреть при необходимости завершение работы потока.

В данном примере поток в течении 0.3 секунды просто ждет, то есть не исполняется, и порт завершения может передать запросы всем потокам пула, хотя их количество превышает максимальное число одновременно работающих потоков, указанное при создании порта:

```
unsigned _ _stdcall PoolProc( void *arg )
```

```
{
  DWORD        size;
  ULONG_PTR    key;
  LPOVERLAPPED      lpov;
  while (
    GetQueuedCompletionStatus(
      (HANDLE)arg, &size, &key, &lpov, INFINITE
  )) {
    /* проверяем условия завершения цикла */
    if ( !size && key == (ULONG_PTR)-1 ) break;
    Sleep( 300 );
  }
  return 0L;
}
```

Рассмотренный механизм управления пулом потоков весьма эффективен, однако требует некоторого объема ручной работы по созданию порта, по созданию пула потоков и по управлению этими потоками.

В современных реализациях Windows предусмотрена возможность автоматического создания и управления пулом потоков с помощью функции

```
BOOL QueueUserWorkItem(
  LPTHREAD_START_ROUTINE QueryFunction,
  PVOID pContext, ULONG Flags
);
```

Эта функция при необходимости создает пул потоков (число потоков в пуле определяется числом процессоров), создает порт завершения ввода-вывода и размещает в очереди порта запрос. Если нужный порт и пул потоков уже созданы, то она просто размещает новый запрос в очереди порта. При обработке запроса будет вызвана указанная параметром QueryFunction процедура с аргументом pContext:

```
DWORD WINAPI QueryFunction( PVOID pContext )
{
  ...
 return 0L;
}
```

Таким образом, управление пулом потоков сильно упрощается, хотя при этом теряется возможность связывания порта завершения ввода-вывода с конкретными файлами и все запросы должны размещаться в очереди явным вызовом функции QueueUserWorkItem.

Есть и еще одна особенность у такого способа управления пулом - явного механизма задания числа потоков в пуле не предусмотрено. Однако у разработчика есть возможность управлять этим процессом с помощью последнего параметра функции, содержащего специфичные флаги. Так, с помощью флага WT_EXECUTEDEFAULT запрос будет направлен обычному потоку из пула, флаг WT_EXECUTEINIOTHREAD заставит систему обрабатывать запрос в потоке, который находится в состоянии ожидания оповещения (то есть, надо предусмотреть явные вызовы функции типа SleepEx или WaitForMultipleObjectsEx и т.д.). Флаг WT_EXECUTELONGFUNCTION предназначен для случаев, когда обработка запроса может привести к продолжительному ожиданию - тогда система может увеличить число потоков в пуле:

```
#include <process.h>
#define _WIN32_WINNT 0x0500
#include <windows.h>

#define MAXQUERIES  15
#define POOLSIZE    3
static LONG             cnt;
static HANDLE       hEvent;

DWORD WINAPI QProc( LPVOID lpData )
{
  int  r = InterlockedIncrement( &cnt );
  Sleep( 300 );
  if ( r >= MAXQUERIES ) SetEvent( hEvent );
  return 0L;
}

int main( void )
{
  int  i;
  hEvent = CreateEvent( NULL, TRUE, FALSE, 0 );
  /* в пуле будет не менее POOLSIZE потоков */
  for ( i = 0; i < POOLSIZE; i++ ) {
    QueueUserWorkItem( QProc, NULL, WT_EXECUTELONGFUNCTION );
    Sleep( 60 );
  }
  /* остальные запросы будут распределяться между потоками
          пула,даже если их больше, чем число процессоров */
  for ( ; i < MAXQUERIES; i++ ) {
    QueueUserWorkItem( QProc, NULL, WT_EXECUTEDEFAULT );
    Sleep( 60 );
  }
  /* со временем система может уменьшить число потоков пула */
  /* дожидаемся обработки последнего запроса */
  WaitForSingleObject( hEvent, INFINITE );
  CloseHandle( hEvent );
  return 0;
}
```

Последний пример качественно проще, чем пример с явным созданием порта завершения ввода-вывода, хотя часть возможностей порта завершения при этом не может быть использована.

Память, локальная для потоков и волокон

При разработке многопоточных приложений возникает необходимость обеспечивать не только параллельное исполнение кода потоков, но также их взаимодействие - обмен данными, доступ к общим, разделяемым всеми потоками данным и изоляцию некоторых данных одного потока от других. Поскольку все потоки разделяют общее адресное пространство процесса, то все они имеют общий и равноправный доступ ко всем данным, хранимым в адресном пространстве. Поэтому для потоков, как правило, не существует проблем с передачей данных друг другу - нужна лишь организация корректного

взаимного доступа и изоляция собственных данных от данных других потоков. Очень часто для изоляции данных достаточно их размещать в стеке - тогда другие потоки смогут получить к ним доступ либо по явно переданным указателям, либо путем сканирования памяти в поисках стеков других потоков и нужных данных в этих стеках, что относится уже к достаточно трудоемким хакерским технологиям. Однако таким образом трудно организовать постоянное хранение данных, и необходимо постоянно явным образом передавать эти данные (или указатель на них) во все вызываемые процедуры; это достаточно неудобно и не всегда возможно.

Для решения подобных задач в Windows предусмотрен механизм управления данными, локальными для потока (TLS память, Thread Local Storage). Система предоставляет небольшой специальный блок данных, ассоциированный с каждым потоком. В таком блоке возможно в общем случае хранение произвольных данных, однако, так как размеры этого блока крайне малы, то обычно там размещаются указатели на данные большего объема, выделяемые в приложении для каждого потока; в связи с этим ассоциированную с потоком память можно рассматривать как массив двойных слов или массив указателей. ОС Windows предоставляет четыре функции, необходимые для работы с локальной для потока памятью.

Функция DWORD TlsAlloc(void) выделяет в ассоциированной с потоком памяти двойное слово, индекс которого возвращается вызвавшей процедуре. Если ассоциированный массив полностью использован, возвращаемое значение будет равно TLS_OUT_OF_INDEXES, что сообщает об ошибке выделения ячейки.

Функция TlsFree освобождает выделенную ячейку.

Если поток выделил некоторую ячейку в ассоциированном массиве, то все потоки данного процесса могут обращаться к ячейке с этим индексом - они получат доступ к ячейкам своих собственных ассоциированных массивов и не смогут узнать или изменить значения, сохраненные в этих ячейках другими потоками. Для доступа к данным зарезервированной ячейки используется функция TlsGetValue, возвращающая значение данной ячейки (в виде указателя, т.к. предполагается, что в ячейках хранятся указатели на некоторые структуры данных) и функция TlsSetValue, изменяющая значение в соответствующей ячейке:

```
#include <process.h>
#include <windows.h>

#define THREADS 18
static DWORD dwTlsData;

void ProcA( int x )
{
  int  i;
  int  *iptr = (int*)TlsGetValue( dwTlsData );
  for ( i = 0; i < 100; i++ ) iptr[i] = x;
}

int ProcB( void )
{
  int i, x;
  int *iptr = (int*)TlsGetValue( dwTlsData );
  for ( i = x = 0; i < 100; i++ ) x += iptr[i];
  return x;
```

```
}

unsigned __stdcall ThreadProc( void *param )
{
  TlsSetValue( dwTlsData, (LPVOID)new int array[100] );
  /* выделенные потоком данные размещены в общей куче,
     используемой всеми потоками, однако указатель на
     эти данные известен только потоку-создателю, так
     как сохраняется в локальной для потока области */
  ProcA( (int)param );
  Sleep( 0 );
  if ( ProcB() != 100*(int)param ) { /* ОШИБКА!!! */ }
  delete[] (int*)TlsGetValue( dwTlsData );
  return 0;
}
int main( void )
{
  HANDLE    hThread[ THREADS ];
  unsigned  dwThread;
  dwTlsData = TlsAlloc();
  /* создаем новые потоки */
  for ( i = 0; i < THREADS; i++ )
    hThread[i]=(HANDLE)_beginthreadex(
      NULL, 0, ThreadProc, (void*)i, 0, &dwThread
    );
  /* дождаться завершения созданных потоков */
  WaitForMultipleObjects( THREADS, hThread, TRUE, INFINITE );
  for ( i = 0; i < THREADS; i++ ) CloseHandle( hThread[i] );
  TlsFree( dwTlsData );
  return 0;
}
```

В приведенном примере в функции main выделяется ячейка в ассоциированном списке, индекс которой сохраняется в глобальной переменной dwTlsData, после чего потоки могут сохранять в этой ячейке свои данные.

В Visual Studio работа с локальной для потока памятью может быть упрощена при использовании _declspec(thread) при описании переменных. В этом случае компилятор будет размещать эти переменные в специальном сегменте данных (_TLS), который будет создаваться библиотекой времени исполнения и ссылки на который будут разрешаться с использованием ассоциированной с потоком памяти. Этот способ во многих случаях предпочтительнее явного управления локальной для потока памятью, так как независимо от числа модулей, использующих такой сегмент, будет задействован только один указатель в ассоциированной памяти (построитель объединит в один большой сегмент все _TLS сегменты модулей).

```
#include <process.h>
#include <windows.h>

#define THREADS 18
_ _declspec(thread) static int iptr[ 100 ];

void ProcA( int x )
```

```
{
  int  i;

  for ( i = 0; i < 100; i++ ) iptr[i] = x;
}
int ProcB( void )
{
  int  i, x;
  for ( i = x = 0; i < 100; i++ ) x += iptr[i];
  return x;
}

unsigned __stdcall ThreadProc( void *param )
{
  ProcA( (int)param );
  Sleep( 0 );
  if ( ProcB() != 100*(int)param ) { /* ОШИБКА!!! */ }
  return 0;
}

int main( void )
{
  HANDLE       hThread[THREADS];
  unsigned     dwThread;
  int          i;
  /* создаем новые потоки */
  for ( i = 0; i < THREADS; i++ )
    hThread[i] = (HANDLE)_beginthreadex(
      NULL, 0, ThreadProc, (void*)i, 0, &dwThread
    );
  /* дождаться завершения созданных потоков */
  WaitForMultipleObjects( THREADS, hThread, TRUE, INFINITE );
  for ( i = 0; i < THREADS; i++ ) CloseHandle( hThread[i] );
  TlsFree( dwTlsData );
  return 0;
}
```

Следует внимательно следить за выделением и освобождением данных, указатели на которые сохраняются в TLS памяти (как в случае явного управления, так и при использовании _ _declspec(thread)). Могут возникнуть две потенциально ошибочных ситуации:

- TLS память резервируется в то время, когда уже существуют потоки. Это возможно при явном управлении TLS памятью, и для существующих потоков будут зарезервированы ячейки, но придется предусмотреть специальные меры для их корректной инициализации или для исключения их использования до этого.
- Все случаи завершения потока. Если TLS память содержит какие-либо указатели, то сама TLS память будет освобождена, а вот те данные, указатели на которые хранились в TLS памяти, - нет. Необходимо специально отслеживать все возможные случаи завершения потоков, включая завершение по ошибке, и принимать меры для освобождения выделенной памяти. При использовании _ _declspec(thread) эта ситуация встречается реже, так как позволяет хранить в _TLS сегментах данные любого фиксированного размера.

Следует отметить еще один нюанс, связанный с использованием TLS памяти, волокон и оптимизации. В частных случаях волокна могут исполняться разными потоками - при этом одно и то же волокно должно иметь доступ к TLS памяти именно того потока, в котором оно в данный момент исполняется. А если компилятор генерирует оптимизированный код, то он может разместить указатель на данные TLS памяти в каком-либо регистре или временной переменной, что при переключении волокна на другой поток приведет к ошибке - будет использована TLS память предыдущего потока. Чтобы избежать такой ситуации, компилятору можно указать специальный ключ /GT, отключающий некоторые виды оптимизации при работе с TLS памятью. Это может потребоваться в крайне редких случаях - когда приложение использует несколько волокон, исполняемых в нескольких потоках, и при этом волокна должны использовать TLS память потоков.

Аналогично TLS памяти, Windows поддерживает память, локальную для волокон, - так называемую FLS память, или Fiber Local Storage. При этом FLS память не зависит от того, какой именно поток выполняет данную нить. Для работы с FLS памятью Windows предоставляет набор функций, аналогичный Tls-функциям, отличие заключается только в функции выделения ячейки FLS памяти:

```
DWORD FlsAlloc( PFLS_CALLBACK_FUNCTION lpCallback );

VOID WINAPI FlsCallback( PVOID lpFlsData )
{
  ...
}
```

Функция отличается от ее аналога TlsAlloc указателем на специальную необязательную процедуру FlsCallback, предоставляемую разработчиком. Эта процедура будет вызвана автоматически при освобождении ячейки FLS памяти (как при завершении волокна, так и при завершении потока или возникновении ошибки), и разработчик может легко предоставить средства для освобождения памяти, указатели на которую были сохранены в ячейках FLS памяти.

```
DWORD dwFlsID;
VOID WINAPI FlsCallback( PVOID lpFlsData )
{
  /* при завершении волокна или потока память будет освобождена
*/
  delete[] (int*)lpFlsData;
}
void initialize( void )
{
  dwFlsID = FlsAlloc( FlsCallback );
  ...
}
void fiberstart( void )
{
  FlsSetValue( dwFlsID, new int [ 100 ] );
  /* здесь мы можем не следить за освобождением выделенной
памяти */
}
```

Остальные функции для работы с FLS аналогичны Tls-функциям как по описаниям, так и по применению.

Привязка к процессору и системы с неоднородным доступом к памяти

ОС Windows предоставляет небольшой набор функций, предназначенных для поддержки систем с неоднородным доступом к памяти (NUMA). К таким функциям относятся средства, обеспечивающие выполнение потоков на конкретных процессорах, и функции, позволяющие получить информацию о структуре NUMA машины. В некоторых случаях привязка потоков к процессорам может преследовать и иные цели, чем поддержка NUMA архитектуры. Так, например, привязка потока к процессору может улучшить использование кэша; на некоторых SMP машинах могут возникать проблемы с использованием таймеров высокого разрешения (опирающихся на счетчики процессоров) и т.д.

Привязка потоков к процессору задается с помощью специального битового вектора (affinity mask), сохраняемого в целочисленной переменной. Каждый бит этого вектора указывает на возможность исполнения потока на процессоре, номер которого совпадает с номером бита. Таким образом, заданием маски сродства можно ограничить множество процессоров, на которых будет выполняться данный поток. В Windows такие маски назначаются процессу (функции GetProcessAffinityMask и SetProcessAffinityMask) и потоку (функция SetThreadAffinityMask). Маска, назначаемая потоку, должна быть подмножеством маски процесса. Помимо ограничения множества процессоров, на которых может исполняться поток, может быть целесообразно назначить потоку самый "удобный" для него процессор (по умолчанию - тот, на котором поток был запущен первый раз). Для этого предназначена функция SetThreadIdealProcessor.

При использовании NUMA систем следует учитывать, что распределение доступных процессоров по узлам NUMA системы не обязательно последовательное - узлы со смежными номерами могут быть с аппаратной точки зрения весьма удалены друг от друга. Функция GetNumaHighestNodeNumber позволяет определить число NUMA узлов, после чего с помощью обращений к функциям GetNumaProcessorNode, GetNumaNodeProcessorMask и GetNumaAvailableMemoryNode можно определить размещение узлов NUMA системы на процессорах и доступную каждому узлу память.

Лекция. Взаимодействие процессов и потоков в ОС Windows

Синхронизация потоков

В Windows реализовано несколько разных способов синхронизации потоков:

- Атомарные операции. В достаточно частых случаях необходимо обеспечить конкурентный доступ к какой-либо целочисленной переменной, являющейся счетчиком. Тогда бывает достаточно просто обеспечить атомарность выполнения операций увеличения, уменьшения или изменения значения переменной.
- Критические секции. В более сложном случае необходимо гарантировать, что какой-либо фрагмент кода будет выполняться в монопольном режиме. Критические секции могут быть использованы только в рамках одного процесса, однако при этом критическая секция может работать чуть быстрее, чем синхронизация с использованием объектов ядра.

- Синхронизация с использованием объектов ядра. Объекты ядра должны поддерживать специальный интерфейс синхронизируемых объектов, чтобы они могли быть использованы для взаимной синхронизации потоков или процессов. Этот интерфейс поддерживают многие объекты ядра, например, файлы, процессы, потоки, консоли, задания и пр. Кроме того, Windows предоставляет специальный набор объектов, предназначенный именно для взаимной синхронизации потоков. Объекты ядра могут быть использованы как для синхронизации потоков в рамках одного процесса, так и для синхронизации потоков в разных процессах.
- Ожидающие таймеры. В некоторых случаях необходимо обеспечить выполнение каких-либо операций либо в заданное время, либо с определенной периодичностью. Для решения этих задач в Windows предусмотрены ожидающие таймеры, которые могут использоваться или средствами синхронизации (ожидающий таймер является объектом ядра, поддерживающим интерфейс синхронизируемых объектов), или для вызова асинхронных процедур.

Атомарные операции

Атомарные операции обычно имеют более-менее близкое соответствие с командами процессора. Так, например, они могут сводиться к операциям с блокировкой шины (префикс lock) и специальным командам (типа cmpxchg) процессора. ОС Windows предоставляет функции для увеличения (InterlockedIncrement, InterlockedIncrement64) или уменьшения (InterlockedDecrement, InterlockedDecrement64) значения целочисленных переменных и изменения их значений (InterlockedExchange, InterlockedExchange64, InterlockedExchangeAdd, InterlockedExchangePointer), в том числе со сравнением (InterlockedCompareExchange, InterlockedCompareExchangePointer).

```
unsigned _ _stdcall ThreadProc( void *param )
{
  int  i;
  for ( i = 0; i < ASIZE; i++ ) InterlockedIncrement(array+i);
  return 0;
}
```

Еще несколько функций предназначены для работы с односвязными LIFO списками. Функция InitializeSListHead подготавливает начальный указатель на LIFO список, функция InterlockedPushEntrySList добавляет новую запись в список, функция InterlockedPopEntrySList извлекает из списка последнюю добавленную запись и функция InterlockedFlushSList очищает список. Операции изменения указателей в списке являются атомарными.

Критические секции

Термин "критическая секция" обозначает некоторый фрагмент кода, который должен выполняться в исключительном режиме - никакие другие потоки и процессы не должны выполнять этот же фрагмент в то же время. В некоторых операционных системах для однопроцессорных машин вход в критическую секцию просто блокирует работу планировщика до выхода из секции. Этот подход не всегда эффективен: нахождение потока в критической секции не должно влиять на возможность исполнения кода, не принадлежащего именно данной критической секции, другими потоками, особенно на многопроцессорных машинах. В Windows предусмотрен специальный тип данных, называемый CRITICAL_SECTION и предназначенный для реализации критических секций. В приложении может существовать произвольное количество данных этого типа, реализующих различные критические секции; равно как несколько секций кода могут использовать один общий объект CRITICAL_SECTION.

Существуют несколько основных функции для работы с критическими секциями:

- перед использованием критическая секция должна быть инициализирована с помощью функции InitializeCriticalSection. Объект CRITICAL_SECTION, принадлежащий пользовательскому процессу, может использовать в своей реализации объекты ядра для ожидания;
- для окончательного освобождения ресурсов, после использования критической секции она должна быть удалена с помощью функции DeleteCriticalSection;
- функция EnterCriticalSection, которая соответствует входу в критическую секцию, при необходимости с ожиданием, не ограниченным по времени;
- TryEnterCriticalSection, которая позволяет при необходимости не входить в секцию, если она занята;
- функция LeaveCriticalSection, которая соответствует выходу из этой секции, возможно с пробуждением потоков, ожидающих ее освобождения. При этом система исключает вход в критическую секцию всех остальных потоков процесса, в то время как поток, уже вошедший в данную секцию, может входить в нее рекурсивно - надо лишь, чтобы число выходов из секции соответствовало числу входов:

```
#include <stdio.h>
#include <process.h>
#define _WIN32_WINNT 0x0403
#include <windows.h>

#define THREADS          10
#define ASIZE            10000000
static  LONG                array[ASIZE];
static  CRITICAL_SECTION      CS;

unsigned _ _stdcall ThreadProc( void *param )
{
  int  i;
  for ( i = 0; i < ASIZE; i++ ) {
    EnterCriticalSection( &CS );
    array[i]++;
    LeaveCriticalSection( &CS );
  }
  return 0;
```

```
}

int main( void )
{
  HANDLE       hThread[THREADS];
  unsigned     dwThread;
  int          i, errs;
  InitializeCriticalSectionAndSpinCount( &CS, 100 );
  for ( i = 0; i < THREADS; i++ )
    hThread[i] = (HANDLE)_beginthreadex(
      NULL, 0, ThreadProc, (void*)i, 0, &dwThread );
  WaitForMultipleObjects( THREADS, hThread, TRUE, INFINITE );
  for ( i = 0; i < THREADS; i++ ) CloseHandle( hThread[i] );
  for ( errs=i=0; i<ASIZE; i++ )
    if ( array[i] != THREADS ) errs++;
  if ( errs ) printf("Detected %d errors!\n", errs );
  DeleteCriticalSection( &CS );
  return 0;
}
```

Кроме того, эффективность критических секций на многопроцессорных машинах может быть повышена, если перед началом ожидания занятой секции, вместо перехода к ожиданию в режиме ядра, выполнить предварительный кратковременный цикл с опросом состояния секции. Если секция занимается другим потоком на небольшое время, то такой цикл позволяет дождаться ее освобождения, не переходя в режим ядра. Для этого предназначены функции: InitializeCriticalSectionAndSpinCount и SetCriticalSectionSpinCount. Они позволяют задавать число опросов состояния занятой критической секции перед переходом в режим ядра для ожидания. На однопроцессорных машинах опросы не выполняются и функция InitializeCriticalSectionAndSpinCount ничем не отличается от обычной функции InitializeCriticalSection.

Синхронизация с использованием объектов ядра

В Windows для синхронизации используются самые разные объекты, применение которых существенно различается. Однако при рассмотрении синхронизации особое положение имеет момент перехода ожидаемого объекта в свободное состояние - с точки зрения ожидающего потока совершенно неважно, какие события привели к этому и какой именно объект стал свободным. Поэтому при большом разнообразии объектов, пригодных для синхронизации, существует всего несколько основных функций, осуществляющих ожидание объекта ядра:

```
DWORD WaitForSingleObject( HANDLE hHandle, DWORD dwMsecs );

DWORD  WaitForMultipleObjects(  DWORD  nCount,  const  HANDLE*
lpHandles,
  BOOL bWaitAll, DWORD dwMsecs);
```

С точки зрения операционной системы объекты ядра, поддерживающие интерфейс синхронизируемых объектов, могут находиться в одном из двух состояний: свободном (signaled) и занятом (nonsignaled). Функции проверяют состояние ожидаемого объекта или ожидаемых объектов и продолжают выполнение, только если объекты свободны. В зависимости от типа ожидаемого объекта, система может предпринять специальные

действия (например, как только поток дожидается освобождения объекта исключительного владения, он сразу должен захватить его).

Функция WaitForSingleObject осуществляет ожидание одного объекта, а функция WaitForMultipleObjects может ожидать как освобождения любого из указанных объектов (bWaitAll = FALSE), так и всех сразу (bWaitAll = TRUE). Ожидание завершается либо по освобождении объекта(ов), либо по истечении указанного интервала времени (dwMsecs) в миллисекундах (бесконечное при dwMsecs = INFINITE). Код возврата функции позволяет определить причину - таймаут, освобождение конкретного объекта либо ошибка.

Если функция WaitForMultipleObjects (или ее клон) ожидает сразу все объекты группы, то до освобождения всех ожидаемых объектов одновременно никаких мер по занятию ранее освободившихся объектов функция не предпринимает.

В тех случаях, когда поток переходит в состояние ожидания, его исполнение блокируется до конца ожидания. Для реализации APC (и функций завершения ввода-вывода) необходимо было предусмотреть в операционной системе возможность приостановки потока с вызовом асинхронных процедур. Это связано с тем, что система должна гарантировать выполнение функций в контексте конкретного потока для соблюдения норм безопасности. Ожидание оповещения - это такое состояние ожидания, которое может быть завершено либо по достижении таймаута, либо при освобождении указанного объекта, либо после обработки APC. При этом в контексте потока, находящегося в состоянии ожидания оповещения, обрабатывается APC вызов и только затем завершается состояние ожидания.

Для перехода в ожидание оповещения предусмотрены функции SleepEx, WaitForMultipleObjectsEx, WaitForSingleObjectEx и SignalObjectAndWait.

Еще несколько функций предназначены для разработки GUI-приложений Win32: MsgWaitForMultipleObjects и MsgWaitForMultipleObjectsEx выполняют ожидание указанных объектов и, кроме того, могут контролировать состояние очереди потока, предназначенной для обработки оконных сообщений. Функция WaitForInputIdle ожидает, пока GUI-приложение не закончит свою инициализацию и не перейдет в ожидание в цикле обработки сообщений.

Несколько типов объектов в Win32 предназначены только для взаимной синхронизации потоков: события (event), семафоры (semaphore), объекты исключительного владения (мьютексы, mutex, mutual exclusion) и ожидающие таймеры (waitable timer). Для изменения их состояния предусмотрены специальные функции, так что потоки могут явным образом управлять состоянием таких объектов, обеспечивая взаимную синхронизацию. Эти объекты являются базовыми примитивами, на которых часто строятся более сложные синхронизирующие объекты.

При проектировании составных объектов иногда возникает задача изменения состояния одного из примитивных объектов при переходе к ожиданию другого. Если такие операции выполнять поочередно, то между вызовами функции, изменяющей состояние объекта, и функции ожидания возможно срабатывание планировщика и переключение потоков; то есть время, проходящее между изменением состояния одного объекта и началом ожидания другого, оказывается непредсказуемым. Чтобы избежать такой ситуации, предусмотрена функция SignalObjectAndWait, переводящая один объект в свободное состояние и ожидающая другой.

Стандартные синхронизирующие объекты могут быть именованными (функции, создающие эти объекты, имеют параметр, указывающий их имя), а могут быть "безымянными", если вместо имени указать NULL. Попытка создания именованных объектов с совпадающими именами приведет или к получению нового описателя существующего объекта (если типы существующего объекта и вновь создаваемого одинаковы), или к ошибке (если типы разные). Именованные объекты обычно используют для межпроцессного взаимодействия, а безымянные - для взаимодействия потоков в одном процессе.

События

Обычно событие - некоторый объект, который может находиться в одном из двух состояний: занятом или свободном. Переключение состояний осуществляется явным вызовом соответствующих функций; при этом любой процесс/поток, имеющий необходимые права доступа к объекту "событие", может изменить его состояние.

В Windows различают события с ручным и с автоматическим сбросом (тип и начальное состояние задаются при создании события функцией CreateEvent). События с ручным сбросом ведут себя обычным образом: функция SetEvent переводит событие в свободное (сигнальное) состояние, а функция ResetEvent - в занятое (не сигнальное). События с автоматическим сбросом переводятся в занятое состояние либо явным вызовом функции ResetEvent, либо ожидающей функцией WaitFor....

```
#define DEFAULT_SECURITY (LPSECURITY_ATTRIBUTES)NULL

VOID CALLBACK ApcProc( ULONG_PTR dwData )
{
  SetEvent( (HANDLE)dwData );
}
int main( void )
{
  HANDLE  hEvent;
  hEvent = CreateEvent( DEFAULT_SECURITY, TRUE, FALSE, NULL );
  QueueUserAPC(ApcProc, GetCurrentThread(), (ULONG_PTR)hEvent);
  WaitForSingleObject(hEvent,100);
    /* код завершения WAIT_TIMEOUT */
  SleepEx( 100, TRUE );
    /* APC процедура освобождает событие */
  WaitForSingleObject(hEvent,100);
    /* код завершения WAIT_OBJECT_0 */
  WaitForSingleObject(hEvent,100);
    /* код завершения WAIT_OBJECT_0 */
  CloseHandle( hEvent );
  return 0;
}
```

В примере создается событие с ручным сбросом в занятом состоянии. При первом вызове функции WaitForSingleObject событие все еще занято, поэтому выход из функции осуществляется по таймауту. При втором вызове уже успевает сработать APC и событие свободно - поэтому функция ожидания завершится с успехом. К третьему вызову состояние события не меняется, поэтому результат аналогичный. Если в приведенном примере изменить событие с ручным сбросом на событие с автоматическим сбросом (второй параметр функции CreateEvent должен быть FALSE), то при втором обращении к

WaitForSingleObject функция завершится также с кодом WAIT_OBJECT_0, но при этом событие автоматически будет переведено в занятое состояние. В результате третий вызов функции WaitForSingleObject завершится по таймауту.

Поведение событий с ручным и автоматическим сбросом особенно различаются в случае, если несколько потоков ждут одного события: для события с ручным сбросом, при установке его в свободное состояние, выполнение могут продолжить все ожидающие потоки; а для события с автоматическим сбросом - только один, так как событие будет сразу переведено в занятое состояние. В некоторых случаях бывает надо перевести событие в свободное состояние, чтобы ожидающие на данный момент времени потоки могли продолжить свое выполнение, после чего снова вернуть в занятое. Вместо пары вызовов SetEvent...ResetEvent, во время выполнения которых планировщик вполне может переключиться на другие потоки (в итоге время нахождения события в свободном состоянии непредсказуемо), целесообразно использовать функцию PulseEvent, которая как бы выполняет сброс и установку события, но в рамках одной операции.

Семафоры

Семафор представляет собой счетчик, который считается свободным, если значение счетчика больше нуля, и занятым при нулевом значении. При создании семафора задаются его максимально допустимое и начальное состояния. Ожидающие функции WaitFor... уменьшают значение свободного семафора на 1, если счетчик ненулевой, или переходят в режим ожидания до тех пор пока кто-либо не увеличит значение семафора. Увеличение счетчика осуществляется функцией ReleaseSemaphore:

```
#define DEFAULT_SECURITY (LPSECURITY_ATTRIBUTES)NULL

int main( void )
{
  HANDLE   hSem;
  LONG     lPrev;
  hSem = CreateSemaphore( DEFAULT_SECURITY, 0, 5, NULL );
  WaitForSingleObject( hSem, 100 );
    /* код завершения WAIT_TIMEOUT */
  ReleaseSemaphore( (HANDLE)dwData, 1, &lPrev );
  WaitForSingleObject( hSem, 100 );
    /* код завершения WAIT_OBJECT_0 */
  WaitForSingleObject( hSem, 100 );
    /* код завершения WAIT_TIMEOUT */
  ReleaseSemaphore( (HANDLE)dwData, 2, &lPrev );
  WaitForSingleObject( hSem, 100 );
    /* код завершения WAIT_OBJECT_0 */
  WaitForSingleObject( hSem, 100 );
    /* код завершения WAIT_OBJECT_0 */
  WaitForSingleObject( hSem, 100 );
    /* код завершения WAIT_TIMEOUT */
  CloseHandle( hSem );
  return 0;
}
```

В примере создается семафор в первоначально занятом состоянии, поэтому первое ожидание завершается по таймауту. После этого его счетчик увеличивается на 1, и следующее ожидание завершается успехом, при этом счетчик семафора уменьшается и он

снова становится занятым. В результате третье ожидание завершается по таймауту. После этого счетчик увеличивается на 2, и два последующих ожидания завершаются успехом, после чего счетчик оказывается снова нулевым и третье ожидание завершается по таймауту.

К сожалению, средств для проверки текущего значения счетчика без изменения состояния семафора нет: функция ReleaseSemaphore позволяет узнать предыдущее значение, но при этом обязательно увеличит его значение хотя бы на 1 (попытка увеличить на 0 или на отрицательную величину рассматривается как ошибка), а ожидающая функция обязательно уменьшит счетчик, если семафор был свободен. Поэтому для определения значения счетчика надо использовать что-то вроде приведенного ниже примера:

```
LONG  lCounter;
lCounter = 0;
if ( WaitForSingleObject( hSem, 0 ) == WAIT_OBJECT_0 )
  ReleaseSemaphore( hSem, 1, &lCounter );
/* теперь переменная lCounter содержит значение счетчика */
```

Семафоры предназначены для ограничения числа потоков, имеющих одновременный доступ к какому-либо ресурсу.

Мьютексы

Объекты исключительного владения могут быть использованы в одно время не более чем одним потоком. В этом отношении мьютексы подобны критическим секциям, с той оговоркой, что работа с ними выполняется в режиме ядра (при использовании критических секций переход в режим ядра необязателен) и что мьютексы могут быть использованы для межпроцессного взаимодействия, тогда как критические секции реализованы для применения внутри процесса. Для захвата мьютекса используется ожидающая функция WaitFor..., а для освобождения - функция ReleaseMutex. При создании мьютекса функцией CreateMutex можно указать, чтобы он создавался сразу в занятом состоянии:

```
#include <process.h>
#include <windows.h>
#define DEFAULT_SECURITY  (LPSECURITY_ATTRIBUTES)NULL

unsigned __stdcall TProc( void *pdata )
{
  WaitForSingleObject( (HANDLE)pdata, 2000 );
  WaitForSingleObject( (HANDLE)pdata, 2000 );
  Sleep( 1000 );
  ReleaseMutex( (HANDLE)pdata );
  ReleaseMutex( (HANDLE)pdata );
  return 0;
}
int main( void )
{
  unsigned  id;
  HANDLE    hMutex, hThread;
  hMutex = CreateMutex( DEFAULT_SECURITY, TRUE, NULL );
  hThread = (HANDLE)_beginthreadex(
    (void*)0, 0, TProc, (void*)hMutex, 0, &id
  );
  Sleep( 1000 );
  ReleaseMutex( hMutex );
  WaitForSingleObject( hThread, INFINITE );
  CloseHandle( hThread );
  CloseHandle( hMutex );
  return 0;
}
```

ОС Windows предоставляет достаточно удобный набор объектов, пригодных для синхронизации. Однако, в большинстве случаев эти объекты эффективны с точки зрения операционной системы, представляя самые базовые примитивы. В реальных задачах часто возникают ситуации, когда необходимо создавать на основе этих примитивов более сложные составные синхронизирующие объекты. Достаточно распространенными примерами таких задач являются задачи с барьерной синхронизацией или задачи с множественным доступом к ресурсу по чтению и исключительным доступом по записи. При реализации барьерной синхронизации надо обеспечить не только возможность проконтролировать достижение барьера всеми потоками, но также снова "поставить барьер" сразу после того, как потоки начнут свое выполнение (иначе какой-либо поток может быстро справиться со своей работой и снова придет к барьеру, пока тот еще открыт). При этом потоки, прошедшие барьер, могут начинать свое выполнение со

значительной задержкой. Синхронизирующие объекты, обслуживающие ресурс с множественным доступом по чтению и исключительным по записи, должны отслеживать число потоков, осуществляющих чтение данного ресурса, и потоки, требующие изменения ресурса. Реализация такого объекта должна предусматривать работу в условиях высокой нагрузки - когда несколько потоков могут одновременно считывать ресурс и при этом не возникает пауз, когда ресурс является свободным; при этом к тому же ресурсу должны обращаться изменяющие его потоки, требуя при этом исключительного доступа.

Стандартных типов объектов, решающих такие задачи, в Windows нет.

Ожидающие таймеры

Эти объекты предназначены для выполнения операций через заданные промежутки времени или в заданное время. Таймеры бывают периодическими или однократными, также их разделяют на таймеры с ручным сбросом и синхронизирующие:

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

int main()
{
  HANDLE            hTimer = NULL;
  LARGE_INTEGER     liDueTime;
  hTimer = CreateWaitableTimer(NULL, TRUE, "WaitableTimer");
  /* задать срабатывание через 5 секунд */
  liDueTime.QuadPart=-50000000;
  SetWaitableTimer( hTimer, &liDueTime, 0, NULL, NULL, 0 );
  WaitForSingleObject( hTimer, INFINITE );
  return 0;
}
```

Таймеры могут служить в качестве синхронизирующих объектов, как в данном примере, а могут вызывать указанную разработчиком функцию, если поток в нужное время находится в ожидании оповещения. Следует подчеркнуть, что ожидающие таймеры обладают ограниченной точностью работы. При необходимости точно планировать время выполнения (например, в случае обработки потоков мультимедиа данных) надо использовать специальный таймер, предназначенный для работы с мультимедиа (см. функции timeGetSystemTime, timeBeginPeriod и др.).

Процессы

Процессы в Windows определяют адресное пространство, которое будет использоваться всеми потоками, работающими в этом процессе. В отличие от UNIX-подобных систем системного вызова типа fork в Windows не предусмотрено – новый процесс создается заново, с выделением для него адресного пространства, проецирования на него компонент системы, образа исполняемого файла, необходимых динамических библиотек и других компонент. Эта операция требует чуть больше ресурсов, чем в UNIX-подобных системах, однако выполняется достаточно быстро - диспетчер памяти позволяет просто проецировать на адресное пространство компоненты из других процессов с режимом "копирование при записи".

Основные механизмы взаимодействия процессов могут быть разделены на несколько групп:

1) Использование объектов ядра для взаимной синхронизации. Рассмотрено при обсуждении взаимной синхронизации потоков. При использовании именованных объектов или передаче описателей объектов ядра другим процессам рассмотренные средства могут использоваться для межпроцессной синхронизации.
2) Проецирование файлов в адресное пространство процесса (File Mapping). Один из базовых механизмов.
3) Использование файловых объектов. Каналы (Pipes), почтовые ящики (Mailslots) и сокеты (Sockets). Еще один базовый механизм; чаще применяется для организации межузлового взаимодействия, за исключением анонимных каналов (unnamed pipes, anonymous pipes), которые используются для межпроцессного взаимодействия в рамках одного узла.
4) Механизмы, ориентированные на обмен оконными сообщениями (буфер обмена, DDE, сообщение WM_COPYDATA и др.). В своей основе используют механизм проецирования файлов для передачи данных между адресными пространствами процессов. В данном курсе эти механизмы не затрагиваются.
5) Вызов удаленных процедур (Remote Procedure Call, RPC). Является надстройкой, использующей проецирование для реальной передачи данных. Позволяет описать процедуры, реализованные в других процессах, и обращаться к ним как к обычным процедурам, локальным для данного процесса. RPC инкапсулирует вопросы нахождения реальной процедуры, выполняющей необходимую работу, передачу данных в эту процедуру и получение от нее ответа. RPC позволяет организовать не только межпроцессное взаимодействие, но также межузловое с передачей данных по сети. В данном курсе не рассматривается.
6) COM (DCOM). Является еще более высокоуровневой абстракцией, в данном курсе также не рассматривается.

Создание процессов

Для создания процессов используются функции CreateProcess, CreateProcessAsUser, CreateProcessWithLogonW и CreateProcessWithTokenW. Функция CreateProcess создает новый процесс, который будет исполняться от имени текущего пользователя потока, вызвавшего эту функцию. Функция CreateProcessAsUser позволяет запустить процесс от имени другого пользователя, который идентифицируется его маркером безопасности (security token); однако вызвавший эту функцию поток должен принять меры к правильному использованию реестра, так как профиль нового пользователя не будет загружен. Функции CreateProcessWithTokenW и CreateProcessWithLogonW позволяют при необходимости загрузить профиль пользователя и, кроме того, функция CreateProcessWithLogonW сама получает маркер пользователя по известному учетному имени, домену и паролю.

```
BOOL CreateProcess(
  LPCTSTR lpApplicationName,     // имя исполняемого файла
  LPTSTR lpCommandLine,          // командная строка
  LPSECURITY_ATTRIBUTES  lpProcessAttributes,  // атрибуты доступа к процессу
  LPSECURITY_ATTRIBUTES lpThreadAttributes,  // атрибуты доступа к потоку
  BOOL bInheritHandles,   // флаг наследования дескрипторов
  DWORD dwCreationFlags,  // флаги создания и флаги класса приоритета
  LPVOID lpEnvironment,   // указатель на параметры настройки окружения
  LPCTSTR lpCurrentDirectory,  // путь к текущему каталогу
  LPSTARTUPINFO lpStartupInfo,  // указатель на стуктуру отображения нового процесса
```

```
  LPPROCESS_INFORMATION lpProcessInformation // указатель на структуру с информацией о созданном процессе
);

typedef struct _PROCESS_INFORMATION {
HANDLE hProcess;    // дескриптор нового процесса
HANDLE hThread;     // дескриптор первичного потока нового процесса
DWORD dwProcessId;  // идентификатор нового процесса
DWORD dwThreadId;   // идентификатор первичного потока нового процесса
} PROCESS_INFORMATION;

#include <windows.h>
#define DEFAULT_SECURITY  (LPSECURITY_ATTRIBUTES)NULL

int main( void )
{
  STARTUPINFO         si;
  PROCESS_INFORMATION  pi;
  memset( &si, 0, sizeof(si) );
  memset( &pi, 0, sizeof(pi) );
  si.cb = sizeof(si);
  CreateProcess(
    NULL, "cmd.exe", DEFAULT_SECURITY, DEFAULT_SECURITY,
    FALSE, NORMAL_PRIORITY_CLASS, NULL, NULL, &si, &pi
  );
  CloseHandle( pi.hThread );
  WaitForSingleObject( pi.hProcess, INFINITE );
  CloseHandle( pi.hProcess );
  return 0;
}
```

При создании процесса ему можно передать описатели каналов (CreatePipe), предназначенные для перенаправления stdin, stdout и stderr. Описатели каналов должны быть наследуемыми.

Для завершения процесса рекомендуется применять функцию ExitProcess, которая завершит процесс, сделавший этот вызов. В крайних случаях можно использовать функцию TerminateProcess, которая может завершить процесс, заданный его описателем. Этой функцией пользоваться не рекомендуется, так как при таком завершении разделяемые библиотеки будут удалены из адресного пространства уничтожаемого процесса без предварительных уведомлений - это может привести в некоторых случаях к утечке ресурсов.

Завершение процессов

Процесс можно завершить четырьмя способами:

- Входная функция первичного потока возвращает управление (самый предпочтительный способ)
- Один из потоков процесса вызывает функцию ExitProcess
- Поток другого процесса вызывает функцию TerminateProcess (нежелательный способ)
- Все потоки процесса корректно завершаются (редкий случай)

```
VOID ExitProcess( UINT uExitCode);
BOOL TerminateProcess(HANDLE hProcess, UINT uExitCode);
```

Создание потоков

Каждый поток начинает выполнение с некоторой входной функции. В первичном потоке используется одна из функций: WinMain() или main(). Для создания вторичного потока необходимо определить его входную функцию:

```
DWORD WINAPI ThreadFunc ( PVoid pvParam)
{
   DWORD dwResult = 0;
   …
   Return dwResult
}
```

Для создания потока используется функция:

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES lpThreadAttributes, // атрибуты доступа
  SIZE_T dwStackSize,                       // размер стека
  LPTHREAD_START_ROUTINE lpStartAddress,    // адрес функции потока
  LPVOID lpParameter,                       // параметр функции потока
  DWORD dwCreationFlags,                    // флаги потока
  LPDWORD lpThreadId                        // идентификатор потока
);
```

При необходимости можно создать поток в виртуальном адресном пространстве другого процесса. Для этого используется следующая функция:

```
HANDLE CreateRemoteThread(
  HANDLE hProcess,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  SIZE_T dwStackSize,
  LPTHREAD_START_ROUTINE lpStartAddress,
  LPVOID lpParameter,
  DWORD dwCreationFlags,
  LPDWORD lpThreadId
);
```

Пример:

```
#include <windows.h>
#include <strsafe.h>

#define MAX_THREADS 3
#define BUF_SIZE 255

typedef struct _MyData {
    int val1;
    int val2;
} MYDATA, *PMYDATA;
```

```
DWORD WINAPI ThreadProc( LPVOID lpParam )
{
    HANDLE hStdout;
    PMYDATA pData;

    TCHAR msgBuf[BUF_SIZE];
    size_t cchStringSize;
    DWORD dwChars;

    hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
    if( hStdout == INVALID_HANDLE_VALUE )
        return 1;

    // Cast the parameter to the correct data type.

    pData = (PMYDATA)lpParam;

    // Print the parameter values using thread-safe functions.

    StringCchPrintf(msgBuf, BUF_SIZE, TEXT("Parameters = %d,
%d\n"),
        pData->val1, pData->val2);
    StringCchLength(msgBuf, BUF_SIZE, &cchStringSize);
    WriteConsole(hStdout, msgBuf, cchStringSize, &dwChars,
NULL);

    // Free the memory allocated by the caller for the thread
    // data structure.

    HeapFree(GetProcessHeap(), 0, pData);

    return 0;
}

void main()
{
    PMYDATA pData;
    DWORD dwThreadId[MAX_THREADS];
    HANDLE hThread[MAX_THREADS];
    int i;

    // Create MAX_THREADS worker threads.

    for( i=0; i<MAX_THREADS; i++ )
    {
        // Allocate memory for thread data.

        pData = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY,
                sizeof(MYDATA));

        if( pData == NULL )
            ExitProcess(2);
```

```
        // Generate unique data for each thread.

        pData->val1 = i;
        pData->val2 = i+100;

        hThread[i] = CreateThread(
            NULL,              // default security attributes
            0,                 // use default stack size
            ThreadProc,        // thread function
            pData,             // argument to thread function
            0,                 // use default creation flags
            &dwThreadId[i]);   // returns the thread identifier

        // Check the return value for success.

        if (hThread[i] == NULL)
        {
            ExitProcess(i);
        }
    }

    // Wait until all threads have terminated.

    WaitForMultipleObjects(MAX_THREADS,      hThread,      TRUE,
INFINITE);

    // Close all thread handles upon completion.

    for(i=0; i<MAX_THREADS; i++)
    {
        CloseHandle(hThread[i]);
    }
}
```

Завершение потока

Завершение потока можно организовать четырьмя способами:

- Функция потока возвращает управление (предпочтительный способ)
- Поток самоуничтожается вызовом функции ExitThread
- Один из потоков данного или стороннего процесса вызывает функцию TerminateThread (нежелательный способ)
- Завершается процесс, содержащий данный поток

Адресное пространство процесса и проецирование файлов

В Windows потоки работают в одном адресном пространстве процесса, поэтому обмен данными между ними в большинстве случаев сводится к доступу к общей разделяемой памяти. Иная ситуация в случае взаимодействия потоков, принадлежащим разным процессам: так как адресное пространство процессов изолировано друг от друга, то приходится принимать специальные меры для передачи данных из адресного пространства одного процесса в адресное пространство другого. Доступная пользователю часть адресного пространства процесса выделяется реально не в физической оперативной

памяти, а в файлах, данные которых проецируются на соответствующие фрагменты адресного пространства. Выделение памяти без явного указания проецируемого файла приведет к тому, что область для проецирования будет автоматически выделяться в файле подкачки страниц. Физическая оперативная память в значительной степени является кэшем для данных файлов. Выделение физической оперативной памяти применяется очень редко, например, средствами расширения адресного пространства (Address Windowing Extension, AWE), позволяющими использовать более чем 4Г адресного пространства в 32-х разрядном приложении Win32.

Важная особенность средств управления адресным пространством и проецированием файлов - они используют так называемую гранулярность выделения ресурсов (ее можно узнать с помощью функции GetSystemInfo); в современных версиях Win32 гранулярность составляет 64К. Это означает, что если вы попробуете спроецировать в память файл размером 100 байт, то в адресном пространстве будет занят фрагмент в 65536 байт длиной, из которого реально будут заняты только первые 100 байт. Для управления адресным пространством предназначены функции VirtualAlloc, VirtualFree, VirtualAllocEx, VirtualFreeEx, VirtualLock и VirtualUnlock. С их помощью можно резервировать пространство в адресном пространстве процесса (без передачи физической памяти из файла) и управлять передачей памяти из файла подкачки страниц указанному диапазону адресов.

Механизмы проецирования файлов в память различаются для обычных и для исполняемых файлов. Функции создания процессов (CreateProcess...) и загрузки библиотек (LoadLibrary, LoadLibraryEx), помимо специфичных действий, выполняют проецирование исполняемых файлов в адресное пространство процесса; при этом учитывается их деление на сегменты, наличие секций импорта, экспорта и релокаций и др. Обычные файлы проецируются как непрерывный блок данных на непрерывный диапазон адресов. Для явного проецирования файлов используется специальный объект ядра проекция файла (file mapping object). Этот объект предназначен для описания файла, который может быть спроецирован в память, но реального отображения файла или его части в память при создании проекции не происходит. Описатель объекта "проекция файла" можно получить с помощью функций CreateFileMapping и OpenFileMapping. Для проецирования файла или его части в память предназначены функции MapViewOfFile, MapViewOfFileEx и UnmapViewOfFile:

```
#include <windows.h>
#define DEFAULT_SECURITY  (LPSECURITY_ATTRIBUTES)NULL
int main( void )
{
  HANDLE  hFile, hMapping;
  LPVOID  pMapping;
  LPSTR   p;
  int     i;
  hFile = CreateFile(
    "abc.dat", GENERIC_WRITE|GENERIC_READ,
    FILE_SHARE_WRITE, DEFAULT_SECURITY,
    CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL
  );
  hMapping = CreateFileMapping(
    hFile, DEFAULT_SECURITY, PAGE_READWRITE, 0, 256, NULL
  );
  pMapping = MapViewOfFile( hMapping, FILE_MAP_WRITE, 0,0, 0 );
  for ( p = (LPSTR)pMapping, i=0; i<256; i++ ) *p++ = (char)i;
```

```
  UnmapViewOfFile( hMapping );
  CloseHandle( hMapping );
  CloseHandle( hFile );
  return 0;
}
```

Межпроцессное взаимодействие с использованием проецирования файлов

Проецирование файлов используется для создания разделяемой памяти: для этого один процесс должен создать объект "проекция файла", а другой - открыть его. После этого система будет гарантировать когерентность данных в этой проекции во всех процессах, которые ее используют, хотя проекции могут размещаться в разных диапазонах адресов.

В примере ниже приводится текст двух приложений (first.cpp и second.cpp), которые обмениваются между собой данными через общий объект "проекция файла":

```
/* FIRST.CPP */
#include <windows.h>
#define DEFAULT_SECURITY  (LPSECURITY_ATTRIBUTES)NULL
int main( void )
{
  HANDLE                hMapping;
  LPVOID                pMapping;
  STARTUPINFO               si;
  PROCESS_INFORMATION    pi;
  int               *p;
  int               i;
  memset( &si, 0, sizeof(si) );
  memset( &pi, 0, sizeof(pi) );
  si.cb = sizeof(si);
  hMapping = CreateFileMapping(
    INVALID_HANDLE_VALUE, DEFAULT_SECURITY, PAGE_READWRITE,
    0, 1024, "FileMap-AB-874436342"
  );
  pMapping = MapViewOfFile( hMapping, FILE_MAP_WRITE, 0,0, 0 );
  for (p=(int*)pMapping,i=0; i<256; i++) *p++=i;
  CreateProcess(
    NULL, "second.exe", DEFAULT_SECURITY,
    DEFAULT_SECURITY, FALSE, NORMAL_PRIORITY_CLASS,
    NULL, NULL, &si, &pi);
  CloseHandle( pi.hThread );
  WaitForSingleObject( pi.hProcess, INFINITE );
  CloseHandle( pi.hProcess );
  UnmapViewOfFile( hMapping );
  CloseHandle( hMapping );
  return 0;
}

/* SECOND.CPP */
#include <windows.h>
int main( int ac, char **av )
{
  HANDLE        hMapping;
```

```
  LPVOID        pMapping;
  int           *p;
  int           i;
  hMapping = OpenFileMapping(
    FILE_MAP_READ, FALSE, "FileMap-AB-874436342"
  );
  pMapping = MapViewOfFile( hMapping, FILE_MAP_READ, 0,0, 0 );
  for ( p = (int*)pMapping, i=0; i<256; i++ )
    if ( *p++ != i ) break;
  if ( i != 256 ) { /* ОШИБКА! */ }
  UnmapViewOfFile( hMapping );
  CloseHandle( hMapping );
  return 0;
}
```

Важно отметить, что если разные процессы откроют один и тот же файл, а затем каждый создаст свой собственный объект "проекция файла", то система не будет гарантировать когерентности данных в проекциях; когерентность обеспечивается только в рамках одного объекта "проекция файла". В некоторых случаях можно воспользоваться функцией FlushViewOfFile для явного сброса данных из оперативной памяти в файл, что, однако, еще не гарантирует автоматического обновления данных в других проекциях. Именно механизм проецирования файлов является базовым средством для передачи данных между адресными пространствами процессов. Он является основой для построения многих других средств межпроцессного взаимодействия. Так, например, обмен оконными сообщениями (если в сообщении содержится указатель на данные) между разными процессами приводит к тому, что система создает внутренний объект "проекция", помещает данные в него, проецирует на второй процесс и посылает сообщение получателю с указателем на проекцию в процессе-получателе.

Межпроцессное взаимодействие с использованием общих секций

Еще одна разновидность работы с разделяемыми данными посредством проецирования файлов связана с объявлением специальных разделяемых сегментов в приложении - такие сегменты будут общими для всех копий этого приложения. В своей основе такой способ является частным случаем проецирования - с той оговоркой, что проецируется исполняемый файл (или разделяемая библиотека) и управление проекциями осуществляется декларативным способом.

В Microsoft Visual C++ для объявления разделяемого сегмента и помещаемых в него данных используются директивы #pragma и _ _declspec, как показано в примере ниже:

```
/* FSEC.CPP */
#include <windows.h>
#define DEFAULT_SECURITY  (LPSECURITY_ATTRIBUTES)NULL

#pragma section("SHRD_DATA",read,write,shared)
_ _declspec(allocate("SHRD_DATA")) int shared[ 256 ];

int main( int ac, char **av )
{
  STARTUPINFO           si;
  PROCESS_INFORMATION   pi;
  int                   i;
```

```
  memset( &si, 0, sizeof(si) );
  memset( &pi, 0, sizeof(pi) );
  si.cb = sizeof(si);
  if ( ac != 2 || strcmp( av[1], "slave" ) ) {
    for ( i = 0; i < 256; i++ ) shared[i] = 256-i;
    /* первый экземляр с общими данными */
    CreateProcess(
      NULL, "fsec.exe slave", DEFAULT_SECURITY,
      DEFAULT_SECURITY, FALSE, NORMAL_PRIORITY_CLASS,
      NULL, NULL, &si, &pi
    );
    CloseHandle( pi.hThread );

    WaitForSingleObject( pi.hProcess, INFINITE );
    CloseHandle( pi.hProcess );
  } else {
    /* второй экземляр с общими данными */
    for ( i = 0; i < 256; i++ )
      if ( shared[i] != 256-i ) break;
    if ( i != 256 ) { /* ОШИБКА! */ }
  }
  return 0;
}
```

Некоторое неудобство, связанное с необходимостью использовать одно и то же приложение, можно легко обойти, если разделяемый сегмент поместить в разделяемую библиотеку - тогда один и тот же образ библиотеки может быть подключен к разным приложениям, что позволит им обмениваться данными. В этом варианте надо только учитывать, что адреса, в которые будет помещена библиотека, в разных процессах могут отличаться.

Лекция. Реализация процессов и потоков в .NET

Параллельные операции в .NET

Реализация параллельного выполнения кода в .NET основана на базовых механизмах, предоставляемых ядром операционной системы Windows. Аналогично средствам операционной системы средства .NET Framework могут быть разделены на следующие группы:

1) Обработка асинхронных запросов. Сюда относятся средства для выполнения асинхронных операций ввода-вывода, средства для работы с очередями сообщений, некоторые надстройки для работы в сетевой среде (ASP, XML и др.) и средства поддержания инфраструктуры COM объектов.
2) Организация многопоточных приложений. Сюда относятся средства создания потоков, управления ими, включая пулы потоков, средства взаимной синхронизации, а также организация локальной для потоков памяти. По большей части .NET Framework предоставляет надстройку над средствами операционной системы.
3) Работа с различными процессами и доменами приложений. В основном это надстройки над абстракциями высокого уровня, такими как RPC, COM объекты и пр.

Как и в случае API системы, такое деление не является строгим – реальные средства "пересекают" границы этих категорий. При этом не все механизмы, предоставляемые операционной системой, нашли свое отражение в .NET Framework; равно как многие механизмы, оставаясь внешне схожими с механизмами операционной системы, существенным образом изменились. Так, например, .NET не поддерживает волокна; операции асинхронного ввода-вывода основываются на использовании отдельных потоков, выполняющих фоновые синхронные операции ввода-вывода, и др. Вообще говоря, использование фоновых потоков для выполнения специфических задач поддержки инфраструктуры .NET стало общим местом. При запуске .NET приложения автоматически создается пул потоков, используемый CLR по мере надобности. Этот пул применяется, в частности, в ситуациях, связанных с обработкой асинхронных запросов и асинхронного ввода-вывода, когда в рамках операционной системы реально использовался бы основной поток в состоянии ожидания оповещения.

.NET разрабатывался с учетом возможности переноса на другие платформы. Для облегчения этого процесса в архитектуру CLR включен специальный уровень адаптации к платформе (PAL, Platform Adaptation Layer), являющийся прослойкой между основными механизмами CLR и уровнем операционной системы. В случае платформы Windows уровень PAL достаточно прост - считается, что PAL должен предоставить для CLR функциональность, аналогичную Win32 API. Однако в случае иных платформ PAL может оказаться достаточно сложным и многоуровневым. Например, в случае платформ, не поддерживающих многопоточные приложения, PAL должен самостоятельно реализовать недостающую функциональность.

Потоки и пул потоков в .Net

Потоки в .NET реализованы на основе модели потоков операционной системы и не предусматривают средств управления волокнами. Если система не поддерживает многопоточные приложения, то PAL должен будет реализовать собственный планировщик потоков режима пользователя (и в этом случае окажется, что потоки .NET являются аналогами волокон, а не потоков ядра).

Основные классы для реализации многопоточных приложений определены в пространстве имен System.Threading. Для описания собственных потоков предназначен класс Thread. При создании потока ему необходимо указать делегата, реализующего процедуру потока. К сожалению, в .NET, во-первых, не предусмотрено передачи аргументов в эту процедуру, во-вторых, процедура должна быть статическим методом, и в-третьих, класс Thread является опечатанным. В результате передача каких-либо данных в процедуру потока вызывает определенные трудности и требует явного или косвенного использования статических полей, что не слишком удобно, зачастую нуждается в дополнительной синхронизации и плохо соответствует парадигме ООП. Приведенный ниже пример демонстрирует работу с созданием нескольких потоков для параллельного перемножения двух квадратных матриц. Начальные значения всех элементов матриц равны 1, поэтому результирующая матрица должна быть заполнена числами, равными размерности перемножаемых матриц. В процессе умножения и суммирования элементов матриц синхронизация не выполняется, поэтому при достаточно большом размере матриц гарантированно будут возникать ошибки (необходимо синхронизировать выполнение некоторых действий в потоках, чтобы избежать возникновения ошибок):

```
using System;
using System.Threading;
```

```
namespace TestNamespace {
  class TestApp {
    const int                m_size = 600;
    const int                m_stripsize = 50;
    const int                m_stipmax = 12;
    private static int       m_stripused = 0;
    private static double[,] m_A = new double[m_size,m_size],
                             m_B = new double[m_size,m_size],
                             m_C = new double[m_size,m_size];
    public static void ThreadProc()
    {
      int  i,j,k, from, to;
      from = ( m_stripused++ ) * m_stripsize;
      to = from + m_stripsize;
      if ( to > m_size ) to = m_size;
      for ( i = 0; i < m_size; i++ ) {
        for ( j = 0; j < m_size; j++ ) {
          for ( k = from; k < to; k++ )
            m_C[i,j] += m_A[i,k] * m_B[k,j];
        }
      }
    }
    public static void Main()
    {
      Thread[]  T = new Thread[ m_stripmax ];
      int    i,j,errs;
      for ( i = 0; i < m_size; i++ ) {
        for ( j = 0; j < m_size; j++ ) {
          m_A[i,j] = m_B[i,j] = 1.0;
          m_C[i,j] = 0.0;
        }
      }
      for ( i = 0; i < m_stripmax; i++ ) {
        T[i] = new Thread(new ThreadStart(ThreadProc));
        T[i].Start();
      }
      // дожидаемся завершения всех потоков
      for ( i = 0; i < m_stripmax; i++ ) T[i].Join();
      // проверяем результат
     errs = 0;
     for ( i = 0; i < m_size; i++ )
          for ( j = 0; j < m_size; j++ )
               if ( m_C[i,j] != m_size ) errs++;
     Console.WriteLine("Error count = {0}", errs );
    }
  }
}
```

Поток в .NET может находиться в одном из следующих состояний: незапущенном, исполнения, ожидания, приостановленном, завершенном и прерванном. Возможные переходы между этими состояниями изображены на рис.1.

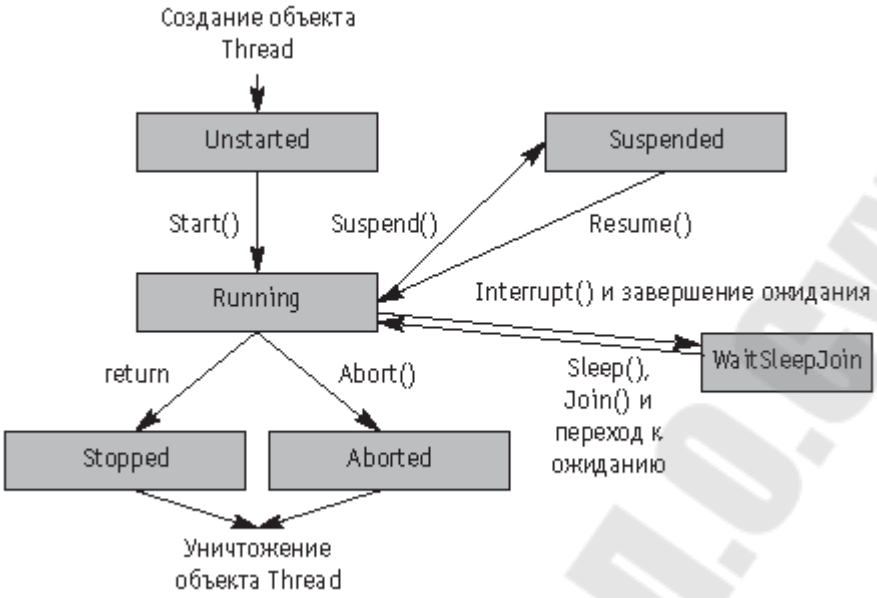

Рис. 1. Состояния потока

Сразу после создания и до начала выполнения потока он находится в незапущенном состоянии (Unstarted). Текущее состояние можно определить с помощью свойства Thread.ThreadState. После запуска поток можно перевести в состояние исполнения (Running) вызовом метода Thread.Start. Работающий поток может быть переведен в состояние ожидания (WaitSleepJoin) явным или неявным вызовом соответствующих методов (Thread.Sleep, Thread.Join и др.) или приостановлен (Suspended) с помощью метода Thread.Suspend(). Исполнение приостановленного потока можно возобновить вызовом метода Thread.Resume. Также можно досрочно вывести поток из состояния ожидания вызовом метода Thread.Interrupt. Завершение функции потока нормальным образом переводит поток в состояние "завершен" (Stopped), а досрочное прекращение работы вызовом метода Thread.Abort переведет его в состояние "прерван" (Aborted). Кроме того, .NET поддерживает несколько переходных состояний (AbortRequested, StopRequested и SuspendRequested). Состояния потока в общем случае могут комбинироваться, например, вполне корректно сочетание состояния ожидания (WaitSleepJoin) и какого-либо переходного, скажем, AbortRequested.

Для выполнения задержек в ходе выполнения потока предназначены два метода - Sleep, переводящий поток в состояние ожидания на заданное время, и SpinWait, который выполняет некоторую задержку путем многократных повторов внутреннего цикла. Этот метод дает высокую загрузку процессора, однако позволяет реализовать очень короткие паузы. К сожалению, продолжительность пауз зависит от производительности и загруженности процессора. Для получения и задания приоритета потока используется свойство Thread.Priority. Приоритеты потока в .NET базируются на подмножестве относительных приоритетов Win32 API так, что при переносе на другие платформы существует возможность предоставить их корректные аналоги. В .NET используются приоритеты Highest, AboveNormal, Normal, BelowNormal и Lowest.

Когда .NET приложение начинает исполняться в среде Windows, CLR создает внутренний пул потоков, используемый средой для реализации асинхронных операций ввода-вывода, вызова асинхронных процедур, обработки таймеров и других целей. Потоки могут добавляться в пул по мере надобности. Этот пул реализуется на основе пула потоков, управляемого операционной системой (построенного на основе порта завершения ввода-вывода). Для взаимодействия с пулом потоков предусмотрен класс ThreadPool, и единственный объект, принадлежащий этому классу, создается CLR при запуске

приложения. Все домены приложений в рамках одного процесса используют общий пул потоков.

Разработчики могут использовать несколько статических методов класса ThreadPool. Так, например, существует возможность связать внутренний порт завершения ввода-вывода с файловым объектом, созданным неуправляемым кодом, для обработки событий, связанных с завершением ввода-вывода этим объектом (см. методы ThreadPool.BindHandle и описание порта завершения ввода-вывода ранее). Можно управлять числом потоков в пуле (методы GetAvailableThreads, GetMaxThreads, GetMinThreads и SetMinThreads), можно ставить в очередь асинхронных вызовов собственные процедуры (метод QueueUserWorkItem) и назначать процедуры, которые будут вызываться при освобождении какого-либо объекта (метод RegisterWaitForSingleObject). Эти два метода имеют "безопасные" и "небезопасные" (Unsafe...) версии; последние отличаются тем, что в стеке вызовов асинхронных методов не будут присутствовать данные о реальном контексте безопасности потока, поставившего в очередь этот вызов, - в подобном случае будет использоваться контекст безопасности самого пула потоков.

Асинхронный ввод-вывод

Для реализации асинхронного ввода-вывода в .NET предназначен абстрактный класс System.IO.Stream. В этом классе определены абстрактные синхронные методы чтения Read и записи Write, а также реализация асинхронных методов BeginRead, EndRead, BeginWrite и EndWrite. Асинхронные методы реализованы с помощью обращения к синхронным операциям фоновыми потоками пула. На основе абстрактного класса Stream в .NET Framework реализуются потомки, осуществляющие взаимодействие с разного рода потоками данных. Так, например, System.IO.FileStream реализует операции с файлами, System.IO.MemoryStream предоставляет возможность использования байтового массива в качестве источника или получателя данных, System.IO.BufferedStream является "надстройкой" над другими объектами, производными от System.IO.Stream, и обеспечивает буферизацию запросов чтения и записи. Некоторые классы вне пространства имен Sytem.IO также являются потомками Stream. Так, например, класс System.NET.Sockets.NetworkStream обеспечивает сетевое взаимодействие.

```
using System;
using System.IO;
namespace TestNamespace {
  class TestApp {
    private const int      m_size = 100000000;
    private static byte[]   m_data = new byte [m_size];

    public TestApp()
    {
      int  i;
      for ( i = 0; i < m_size; i++ ) m_data[i] = (byte)i;
    }

    public static void DoneWritting( IAsyncResult state ) {
    }
    static void Main(string[] args) {
      TestApp        ta = new TestApp();
      IAsyncResult   state;
      Stream         st = new FileStream(
        "test.dat", FileMode.OpenOrCreate,
```

```
        FileAccess.ReadWrite, FileShare.Read, 1, true
      );
      state = st.BeginWrite(
        m_data, 0, m_size,
        new AsyncCallback(DoneWritting), null
      );
      // код в этом месте будет выполняться
      // одновременно с выводом данных
      st.EndWrite( state );
      st.Close();
    }
  }
}
```

Данный пример демонстрирует использование FileStream для выполнения асинхронной операции записи большого объема данных. При реализации собственных потомков класса Stream, возможно, будет иметь смысл переопределить не только абстрактные методы Read и Write, но также некоторые базовые (например, BeginRead, ReadByte и др.), универсальная реализация которых может быть неэффективной в конкретном случае.

Асинхронные процедуры

Для реализации вызова асинхронных процедур в .NET используются фоновые потоки пула, так же как для обработки асинхронных операций ввода-вывода. Класс ThreadPool предлагает два способа для вызова асинхронных процедур: явное размещение вызовов в очереди (QueueUserWorkItem) и связывание вызовов с переводом некоторых объектов в свободное состояние (RegisterWaitForSingleObject). Кроме того, .NET позволяет осуществлять асинхронные вызовы любых процедур с помощью метода BeginInvoke делегатов.

Статический метод ThreadPool.QueueUserWorkItem ставит вызов указанной процедуры в очередь для обработки. Если пул содержит простаивающие потоки, то обработка этой функции начнется немедленно:

```
using System;
using System.Threading;
namespace TestNamespace {
  class GreetingData {
    private string   m_greeting;
    public GreetingData( string text ) { m_greeting = text; }
    public void Invoke() { Console.WriteLine( m_greeting ); }
  }

  class TestApp {
    static void AsyncProc( Object arg )
    {
      GreetingData  gd = (GreetingData)arg;
      gd.Invoke();
    }

    public static void Main()
    {
      GreetingData gd = new GreetingData("Hello, world!");
```

```
      ThreadPool.QueueUserWorkItem(new   WaitCallback(AsyncProc),
gd);
      Thread.Sleep( 1000 );
    }
  }
}
```

При постановке в очередь асинхронного вызова можно указать объект, который является аргументом асинхронной процедуры (при создании собственных потоков передача аргументов процедуре потока затруднительна).

Второй способ вызова асинхронных процедур связан с использованием объектов, производных от класса System.Threading.WaitHandle (это события и мьютексы). При этом вызов асинхронной процедуры связывается с переводом объекта в свободное состояние. Данный метод может быть использован также для организации повторяющегося через определенные интервалы вызова асинхронных процедур - при регистрации делегата указывается максимальный интервал ожидания, и если он исчерпывается, то вызов размещается в очереди пула, даже если объект остался занятым. Если объект по-прежнему остается занятым, то вызов процедуры будет периодически размещаться в очереди после исчерпания каждого интервала ожидания.

```
using System;
using System.Threading;

namespace TestNamespace {
  class GreetingData {
    private string              m_greeting;
    private RegisteredWaitHandle   m_waithandle;
    public GreetingData( string text ) { m_greeting = text; }
    public void Invoke() { Console.WriteLine( m_greeting ); }
    public RegisteredWaitHandle WaitHandle {
      set {
        if (value==null) m_waithandle.Unregister( null );
        m_waithandle = value;
      }
    }
  }
  class TestApp {
    static void AsyncProc( Object arg, bool isTimeout ) {
      GreetingData  gd = (GreetingData)arg;
      if ( !isTimeout ) gd.WaitHandle = null;
      gd.Invoke();
    }
    public static void Main() {
      GreetingData     gd = new GreetingData("Hello");
      AutoResetEvent   ev = new AutoResetEvent(false);
      gd.WaitHandle=ThreadPool.RegisterWaitForSingleObject(
        ev,  new WaitOrTimerCallback(AsyncProc),
        gd,  1000,  false
      );
      Thread.Sleep( 2500 );
      ev.Set();
      Console.ReadLine();
    }
```

```
  }
}
```

Приведенный пример демонстрирует использование периодического вызова асинхронной процедуры - при регистрации делегата (RegisterWaitForSingleObject) указывается максимальное время ожидания 1 секунда (1000 миллисекунд), после чего основной поток переводится в состояние "спячки" на 2.5 секунды. За это время в очередь пула поступает два вызова асинхронных процедур (с признаком вызова по тайм-ауту). Через 2.5 секунды основной поток пробуждается, переводит событие в свободное состояние, и в очередь пула поступает третий вызов. При обработке этого вызова регистрация делегата отменяется.

Последний способ связан с использованием методов BeginInvoke и EndInvoke делегатов. Когда определяется какой-либо делегат функции, для него будут определены методы: BeginInvoke (содержащий все аргументы делегата плюс два дополнительных - AsyncCallback, который может быть вызван по завершении обработки асинхронного вызова, и AsyncState, с помощью которого можно определить состояние асинхронной процедуры) и EndInvoke, содержащий все выходные параметры (т.е. описанные как inout или out), плюс IAsyncResult, позволяющий узнать результат выполнения процедуры.

Таким образом, использование BeginInvoke позволяет не только поставить в очередь вызов асинхронной процедуры, но также связать с завершением ее обработки еще один асинхронный вызов. Метод EndInvoke служит для ожидания завершения обработки асинхронной процедуры:

```
using System;
using System.Threading;

namespace TestNamespace {
  public class GreetingData {
    private string   m_greeting;
    public GreetingData( string text ) { m_greeting = text; }
    public static void Invoke( GreetingData arg ) {
      Console.WriteLine( arg.m_greeting );
    }
  }
  public delegate void AsyncProcCallback ( GreetingData gd );
  class TestApp {
    public static void Main() {
      GreetingData gd = new GreetingData( "Hello!!!" );
      AsyncProcCallback apd = new AsyncProcCallback(
        GreetingData.Invoke );
      IAsyncResult ar = apd.BeginInvoke( gd, null, null );
      ar.AsyncWaitHandle.WaitOne();
    }
  }
}
```

Данный пример иллюстрирует вызов асинхронной процедуры с использованием метода BeginInvoke и альтернативный механизм ожидания завершения - с использованием внутреннего объекта AsyncWaitHandle (класса WaitHandle), благодаря которому, собственно говоря, становится возможен вызов асинхронной процедуры, обслуживающей завершение обработки данной процедуры. В этом смысле асинхронный вызов процедур с помощью BeginInvoke очень близок к обработке асинхронных операций ввода-вывода.

Синхронизация и изоляция потоков

Проблемы, встающие перед разработчиками многопоточных приложений .NET, очень похожи на проблемы разработчиков приложений Win32 API. Соответственно, .NET предоставляет в значительной мере близкий набор средств взаимодействия потоков и их взаимной синхронизации. К этим средствам относятся атомарные операции, локальная для потока память, синхронизирующие примитивы и таймеры. Их применение в основе своей похоже на применение аналогичных средств API.

Атомарные операции

Платформа .NET предоставляет, аналогично базовой операционной системе Windows, набор некоторых основных операций над целыми числами (int и long), которые могут выполняться атомарно. Для этого предусмотрены четыре статических метода класса System.Threading.Interlocked, а именно Increment, Decrement, Exchange и CompareExchange. Применение этих методов аналогично соответствующим Interlocked... процедурам Win32 API.

Возвращаясь к примеру использования потоков для умножения матриц, можно выделить один момент, требующий исправления: самое начало процедуры потока, там, где определяется номер полосы:

```
public static void ThreadProc()
{
  int    i,j,k, from, to;
  from = ( m_stripused++ ) * m_stripsize;
  to = from + m_stripsize;
  ...
```

Здесь потенциально возможна ситуация, когда несколько потоков одновременно начнут выполнять этот код и получат идентичные номера полос. В этом месте самым эффективным было бы использование атомарных операций для увеличения значения поля m_stripused. Для этого фрагмент надо переписать:

```
public static void ThreadProc()
{
 int    i,j,k, from, to;
 from  =  (Interlocked.Increment(ref  m_stripused)  -  1  )  *
m_stripsize;
 to = from + m_stripsize;
  ...
```

Синхронизация потоков

Основные средства взаимной синхронизации потоков в .NET обладают заметным сходством со средствами операционной системы. Среди них можно выделить:

- Мониторы, близкие к критическим секциям Win32 API.
- События и мьютексы, имеющие соответствующие аналоги среди объектов ядра.
- Плюс дополнительный достаточно универсальный синхронизирующий объект, обеспечивающий множественный доступ потоков по чтению и исключительный - по записи.

Последний синхронизирующий объект ReaderWriterLock закрывает очень типичный класс задач синхронизации - для многих объектов является совершенно корректным конкурентный доступ для чтения и требуются исключительные права для изменения данных. Причина в том, что изменение сложных объектов осуществляется не атомарно, поэтому во время постепенного внесения изменений объект кратковременно пребывает в некорректном состоянии - при этом должен быть исключен не только доступ других потоков, пытающихся внести изменения, но даже потоков, осуществляющих чтение.

Мониторы

Мониторы в .NET являются аналогами критических секций в Win32 API. Использование мониторов достаточно эффективно (это один из самых эффективных механизмов) и удобно настолько, что в .NET был предусмотрен механизм, который позволяет использовать практически любой объект, хранящийся в управляемой куче, для синхронизации доступа. Для этого с каждым объектом ссылочного типа сопоставляется запись SyncBlock, являющаяся, по сути, аналогом структуры CRITICAL_SECTION в Win32 API. Добавление такой записи к каждому объекту в управляемой куче чересчур накладно, особенно если учесть, что используются они относительно редко. Поэтому все записи SyncBlock выносятся в отдельный кэш, а в информацию об объекте включается ссылка на запись кэша (см. рис. 2). Такой прием позволяет, с одной стороны, содержать кэш синхронизирующих записей минимального размера, а с другой - любому объекту при необходимости можно сопоставить запись.

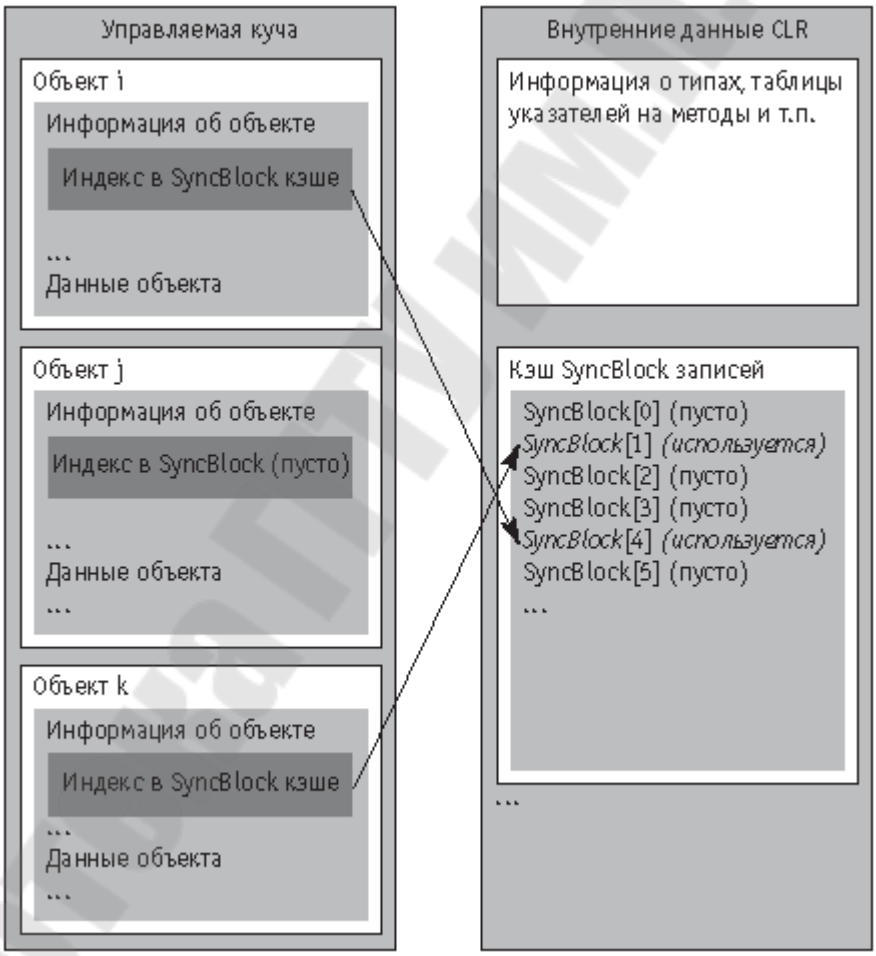

Рис.2. Использование кэша SyncBlock записей объектами управляемой кучи

Обычно объекты не имеют сопоставленной с ними SyncBlock записи, однако она автоматически выделяется при первом использовании монитора.

Класс Monitor, определенный в пространстве имен System.Threading, предлагает несколько статических методов для работы с записями синхронизации. Методы Enter и Exit являются наиболее применяемыми и соответствуют функциям EnterCriticalSection и LeaveCriticalSection операционной системы. Аналогично критическим секциям Win32 API, мониторы могут использоваться одним потоком рекурсивно. Еще несколько методов класса Monitor - Wait, Pulse и PulseAll - позволяют при необходимости временно

разрешить доступ к объекту другому потоку, ожидающему его освобождения, не покидая критической секции.

Продолжим рассмотрение примера с многопоточным умножением матриц. Помимо уже рассмотренной проблемы с назначением полос, в процедуре потока есть еще одно некорректное место - прибавление накоплением к элементу результирующей матрицы произведения двух элементов исходной матрицы:

```
public static void ThreadProc()
{
  int    i,j,k, from, to;
  from = (Interlocked.Increment(ref m_stripused)-1)
    * m_stripsize;
  to = from + m_stripsize;
  if ( to > m_size ) to = m_size;
  for ( i = 0; i < m_size; i++ ) {
    for ( j = 0; j < m_size; j++ ) {
      for ( k = from; k < to; k++ )
        m_C[i,j] += m_A[i,k] * m_B[k,j];
    }
  }
}
```

Так как эта операция выполняется не атомарно, то вполне может быть так, что один поток считывает значение m_C[i,j], прибавляет к нему величину m_A[i,k] * m_B[k,j] и, прежде чем успевает записать в m_C[i,j] результат сложения, прерывается другим потоком. Второй поток успевает изменить величину m_C[i,j], потом первый снова пробуждается и записывает значение, вычисленное для предыдущего состояния элемента m_C[i,j], - то есть некорректную величину. Собственно говоря, именно эта ситуация и приводит к ошибкам, которые можно наблюдать в исходном примере.

Ситуацию можно исправить, используя синхронизацию при доступе к элементу m_C[i,j] с помощью мониторов:

```
...
  for ( j = 0; j < m_size; j++ ) {
    for ( k = from; k < to; k++ ) {
      Monitor.Enter( m_C );
      try {
        m_C[i,j] += m_A[i,k] * m_B[k,j];
      } finally {
        Monitor.Exit( m_C );
      }
    }
  }
...
```

В этом фрагменте надо выделить два существенных момента: во-первых, использование метода Exit в блоке finally, а во-вторых - использование всего массива m_C, а не отдельного элемента m_C[i,j].

Первое надо взять за правило, так как в случае возникновения исключения в критической секции блокировка может остаться занятой (т.е. в случае покидания секции без вызова метода Exit).

Второе связано с тем, что элементы m_C[i,j] являются значениями, а не ссылочными типами. Для типов-значений соответствующее представление в управляемой куче не создается, и у них нет и не может быть ссылок на синхронизирующие записи SyncBlock.

Самое плохое в этой ситуации то, что попытка собрать приложение, использующее типы-значения в качестве аргументов методов Enter и Exit (как в примере ниже), пройдет успешно:

```
...
  for ( j = 0; j < m_size; j++ ) {
    for ( k = from; k < to; k++ ) {
      Monitor.Enter( m_C[i,j] );
      try {
        m_C[i,j] += m_A[i,k] * m_B[k,j];
      } finally {
        Monitor.Exit( m_C[i,j] );
      }
    }
  }
...
```

В прототипах методов Enter и Exit указано, что они должны получать ссылочный тип object; соответственно тип-значение будет упакован, и методу Enter будет передан свой экземпляр упакованного типа-значения, на который будет поставлена блокировка, а методу Exit - свой экземпляр, на котором блокировки никогда не было. Понятно, что все остальные потоки будут создавать и множить свои собственные упакованные представления типов-значений, и никакой синхронизации не произойдет. Поэтому при использовании мониторов важно проследить, чтобы вызовы разных методов в разных потоках использовали один общий объект ссылочного типа.

Можно выделить интересный момент – типы объектов сами являются экземплярами класса Type, и для них выделяется место в управляемой куче. Это позволяет использовать тип объекта в качестве владельца записи SyncBlock:

```
...
  for ( j = 0; j < m_size; j++ ) {
    for ( k = from; k < to; k++ ) {
      Monitor.Enter( typeof(double) );
      try {
        m_C[i,j] += m_A[i,k] * m_B[k,j];
      } finally {
        Monitor.Exit( typeof(double) );
      }
    }
  }
...
```

Возможно неявное использование мониторов в C# с помощью ключевого слова lock:

```
lock ( obj ) { ... }
```

эквивалентна

```
Monitor.Enter( obj ); try { ... }
```

```
finally { Mointor.Exit( obj ); }
```

Использование ключевого слова lock предпочтительно, так как при этом выполняется дополнительная синтаксическая проверка - попытка использовать для блокировки тип-значение приведет к диагностируемой компилятором ошибке, вместо трудно отлавливаемой ошибки во время исполнения:

```
public static void ThreadProc()
{
  int     i,j,k, from, to;
  double  R;
  from   =   (Interlocked.Increment(ref   m_stripused)   -   1)   *
m_stripsize;
  to = from + m_stripsize;
  if ( to > m_size ) to = m_size;
  for ( i = 0; i < m_size; i++ ) {
    for ( j = 0; j < m_size; j++ ) {
      R = 0;
      for ( k = from; k < to; k++ ) R += m_A[i,k]*m_B[k,j];
      lock ( m_C ) { m_C[i,j] += R; }
    }
  }
}
```

Данный пример показывает процедуру потока, осуществляющего пополосное умножение матриц с необходимой синхронизацией. Следует заметить, что синхронизация доступа требует дополнительных ресурсов процессора (в данном случае, качественно превышающих затраты на умножение и сложение двух чисел с плавающей запятой), поэтому целесообразно как можно сильнее сократить число блокировок и время их наложения. В примере для этого использована промежуточная переменная R, накапливающая частичный результат.

Следует особо подчеркнуть, что мониторы и блокировки доступа только лишь позволяют разработчику реализовать соответствующую синхронизацию, но ни в коем случае не осуществляют принудительное ограничение конкурентного обращения к полям и методам объектов. Любой параллельно выполняющийся фрагмент кода сохраняет полную возможность обращаться со всеми объектами, независимо от того, связаны они с какими-либо блокировками или нет. Для синхронизации и блокирования доступа необходимо, чтобы все участники синхронизации явным образом использовали критические секции.

Ожидающие объекты

.NET предоставляет базовый класс WaitHandle, служащий для описания объекта, который находится в одном из двух состояний: занятом или свободном. На основе этого класса строятся другие классы синхронизирующих объектов .NET, такие как события (ManualResetEvent и AutoResetEvent) и мьютексы (Mutex).

Класс WaitHandle является, по сути, оберткой объектов ядра операционной системы, поддерживающих интерфейс синхронизации. Свойство Handle объекта WaitHandle позволяет установить (или узнать) соответствие этого объекта .NET с объектом ядра операционной системы.

Существует три метода класса WaitHandle для ожидания освобождения объекта: метод WaitOne, являющийся методом объекта, и статические методы WaitAny и WaitAll. Метод WaitOne является оберткой вызова WaitForSingleObject Win32 API, а методы WaitAny и WaitAll - вызова WaitForMultipleObjects. Соответственно семантике конкретных объектов ядра, представленных объектом WaitHandle, методы Wait... могут изменять или не изменять состояние ожидаемого объекта. Так, например, для событий с ручным сбросом (ManualResetEvent) состояние не меняется, а события с автоматическим сбросом и мьютексы (AutoResetEvent, Mutex) переводятся в занятое состояние.

Объекты класса WaitHandle и производных от него, представляя объекты ядра операционной системы, могут быть использованы для межпроцессного взаимодействия. Конструкторы производных объектов (событий и мьютексов) позволяют задать имя объекта ядра, предназначенное для организации общего доступа к объектам процессами:

```
using System;
using System.Threading;

namespace TestNamespace {
  public class SomeData {
    public const int          m_queries = 10;
    private static int        m_counter = 0;
    private static Mutex       m_mutex = new Mutex();
    private static ManualResetEvent m_event =
                new ManualResetEvent( false );
    public static void Invoke( int no ) {
      m_mutex.WaitOne();
      m_counter++;
      if ( m_counter >= m_queries ) m_event.Set();
      m_mutex.ReleaseMutex();
      m_event.WaitOne();
    }
  }
  public delegate void AsyncProcCallback( int no );
  class TestApp {
    public static void Main() {
      int               i;
      WaitHandle[]      wh;
      AsyncProcCallback  apd;
      wh = new WaitHandle[ SomeData.m_queries ];
      apd = new AsyncProcCallback( SomeData.Invoke );
      for ( i = 0; i < SomeData.m_queries; i++ ) wh[i] =
           apd.BeginInvoke(i,null,null).AsyncWaitHandle;
      WaitHandle.WaitAll( wh );
    }
  }
}
```

Приведенный пример показывает синхронизацию с использованием мьютекса, события с ручным сбросом и объекта WaitHandle, представляющего состояние асинхронного вызова. В примере делается 10 асинхронных вызовов, после чего приложение ожидает завершения всех вызовов с помощью метода WaitAll. Каждый асинхронный метод в секции кода,

защищаемой мьютексом (здесь было бы эффективнее использовать монитор или блокировку), подсчитывает число сделанных вызовов и переходит к ожиданию занятого события. Самый последний асинхронный вызов установит событие в свободное состояние, после чего все вызовы должны завершиться. Помимо использования разных синхронизирующих объектов, в этом примере интересно поведение CLR: асинхронные вызовы должны обрабатываться в пуле потоков, однако число вызовов превышает число потоков в пуле. CLR по мере необходимости добавляет в пул потоки для обработки поступающих запросов.

Потоки не являются наследниками класса WaitHandle в силу того, что для разных базовых платформ потоки могут быть реализованы в качестве потоков операционной системы или легковесных потоков, управляемых CLR. В последнем случае потоки .NET не будут иметь никаких аналогов среди объектов ядра операционной системы. Для синхронизации с потоками надо использовать метод Join класса Thread.

Один "писатель", много "читателей"

Одной из типичных задач синхронизации потоков является задача, в которой допускается одновременный конкурентный доступ многих объектов для чтения данных ("читатели") и исключительный доступ единственного потока, вносящего в объект изменения ("писатель"). В Win32 API стандартного объекта, реализующего подобную логику, не существует, поэтому каждый раз его надо проектировать и создавать заново. .NET предоставляет весьма эффективное стандартное решение: класс ReaderWriterLock. В приводимом ниже примере демонстрируется применение методов Acquire... и Release... для корректного использования блокировки доступа при чтении и записи. Тестовый класс содержит две целочисленные переменные, которые считываются и увеличиваются на 1 с небольшими задержками по отношению друг к другу. Пока операции синхронизируются, попытка чтения или изменения всегда будет возвращать четный результат, а вот если бы синхронизация не выполнялась, то в некоторых случаях получались бы нечетные числа:

```
using System;
using System.Threading;
namespace TestNamespace {
  public class SomeData {
    public const int         m_queries = 10;
    private ReaderWriterLock  m_rwlock = new ReaderWriterLock();
    private int       m_a = 0, m_b = 0;
    public int summ() {
      int  r;
      m_rwlock.AcquireReaderLock( -1 );
      try {
        r = m_a; Thread.Sleep( 1000 ); return r + m_b;
      } finally {
        m_rwlock.ReleaseReaderLock();
      }
    }
    public int inc() {
      m_rwlock.AcquireWriterLock( -1 );
      try {
        m_a++; Thread.Sleep( 500 ); m_b++;
        return m_a + m_b;
      } finally {
        m_rwlock.ReleaseWriterLock();
```

```
      }
    }
    public static void Invoke( SomeData sd, int no ) {
      if ( no % 2 == 0 ) {
        Console.WriteLine( sd.inc() );
      } else {
        Console.WriteLine( sd.summ() );
      }
    }
  }
  public delegate void AsyncProcCallback(SomeData sd, int no);
  class TestApp {
    public static void Main() {
      int                i;
      SomeData           sd = new SomeData();
      WaitHandle[]       wh;
      AsyncProcCallback  apd;
      wh = new WaitHandle[ SomeData.m_queries ];
      apd = new AsyncProcCallback( SomeData.Invoke );
      for ( i = 0; i < SomeData.m_queries; i++ ) wh[i] =

     apd.BeginInvoke(sd,i,null,null).AsyncWaitHandle;
      WaitHandle.WaitAll( wh );
    }
  }
}
```

Конечно, аналогичного эффекта можно было бы добиться, просто используя блокировку (lock или методы класса Monitor) при доступе к объекту. Однако, такой подход потребует наложить блокировку исключительного доступа при чтении данных, что не эффективно. В обычных условиях вполне допустимо чтение данных несколькими одновременно выполняющимися потоками, что может дать заметное ускорение.

Локальная для потока память

Применение локальной для потока памяти в .NET опирается на TLS память, поддерживаемую операционной системой. Аналогично Win32 API, возможны декларативный и императивный подходы для работы с локальной для потока памятью.

Декларативный подход сводится к использованию атрибута ThreadStaticAttribute перед описанием любого статического поля. Например, в следующем фрагменте:

```
class SomeData {
  [ThreadStatic]
  public static double  xxx;
  ...
```

Поле класса SomeData.xxx будет размещено в локальной для каждого потока памяти.

Императивный подход связан с применением методов AllocateDataSlot, AllocateNamedDataSlot, GetNamedDataSlot, FreeNamedDataSlot, GetData и SetData класса Thread. Использование этих методов очень похоже на использование Tls... функций Win32 API, с той разницей, что вместо целочисленного индекса в TLS массиве потока (как это

было в Win32 API) используется объект типа LocalDataStoreSlot, который выполняет функции прежнего индекса:

```
class SomeData {
  private      static      LocalDataStoreSlot      m_tls      =
Thread.AllocateDataSlot();
  public static void ThreadProc() {
    Thread.SetData( m_tls, ... );
    ...
  }
  public void Main() {
    SomeData  sd = new SomeData();
    ...
    // создание и запуск потоков
  }
}
```

Методы Allocate... и GetNamedDataSlot позволяют выделить новую ячейку в TLS памяти (или получить существующую именованную), методы GetData и SetData позволяют получить или сохранить ссылку на объект в TLS памяти. Использование TLS памяти в .NET менее удобно и эффективно, чем в Win32 API, но это связано не с реализацией TLS, а с реализацией потоков:
Во-первых, возможно размещение данных в TLS памяти только текущего потока, то есть нельзя положить данные до запуска потока.
Во-вторых, процедура потока не получает аргументов, то есть требуется предусмотреть отдельный механизм для передачи данных в функцию потока, а этот неизбежно реализуемый механизм окажется конкурентом существующей реализации TLS памяти.
В-третьих, использование TLS памяти в асинхронно вызываемых процедурах может быть ограничено теми соображениями, что заранее нельзя предугадать поток, который будет выполнять эту процедуру.
В-четвертых, использование методов ООП часто позволяет сохранить специфичные данные в полях объекта, вообще не прибегая к выделению TLS памяти.

Таймеры

.NET предлагает два вида таймеров: один описан в пространстве имен System.Timers, а другой - в пространстве имен System.Threading. Таймер пространства имен System.Threading является опечатанным и предназначен для вызова указанной асинхронной процедуры с заданным интервалом времени. Таймер пространства имен System.Timers может быть использован для создания собственных классов-потомков - в нем вместо процедуры асинхронного вызова применяется обработка события, с которым может быть сопоставлено несколько обработчиков. Кроме того, этот таймер может вызывать обработку события конкретным потоком, а не произвольным потоком пула:

```
using System;
using System.Timers;

namespace TestNamespace {
  class TestTimer : Timer {
    private int m_minimal, m_maximal, m_counter;
    public int count { get{ return m_counter - m_minimal; }}
    public TestTimer( int mn, int mx ) {
      Elapsed += new ElapsedEventHandler(OnElapsed);
```

```
      m_minimal = m_counter = mn;
      m_maximal = mx;
      AutoReset = true;
      Interval = 400;
    }
    static void OnElapsed( object src, ElapsedEventArgs e ) {
      TestTimer  tt = (TestTimer)src;
      if ( tt.m_counter < tt.m_maximal ) tt.m_counter++;
      if ( tt.m_counter >= tt.m_maximal ) tt.Stop();
    }
    static void Main(string[] args) {
      TestTimer  tm = new TestTimer( 0, 10 );
      tm.Start();
      Thread.Sleep( 5000 );
      tm.Stop();
    }
  }
}
```

Приведенный выше пример иллюстрирует использование таймера пространства имен System.Timers.

Модуль 4. Управление ресурсами в ОС

Лекции 6ч.
Тема 12. Организация памяти компьютера. Простейшие схемы управления памятью
Тема 13. Виртуальная память. Схемы управления виртуальной памятью
Тема 14. Файловая система ОС

Лекция. Организация памяти компьютера. Простейшие схемы управления памятью

Введение

Главная задача компьютерной системы – выполнять программы. Программы вместе с данными, к которым они имеют доступ, в процессе выполнения должны (по крайней мере частично) находиться в оперативной памяти. Операционной системе приходится решать задачу распределения памяти между пользовательскими процессами и компонентами ОС. Эта деятельность называется управлением памятью. Таким образом, память (storage, memory) является важнейшим ресурсом, требующим тщательного управления. В недавнем прошлом память была самым дорогим ресурсом.

Часть ОС, которая отвечает за управление памятью, называется менеджером памяти.
Физическая организация памяти компьютера

Запоминающие устройства компьютера разделяют, как минимум, на два уровня: основную (главную, оперативную, физическую) и вторичную (внешнюю) память.

Основная память представляет собой упорядоченный массив однобайтовых ячеек, каждая из которых имеет свой уникальный адрес (номер). Процессор извлекает команду из основной памяти, декодирует и выполняет ее. Для выполнения команды могут потребоваться обращения еще к нескольким ячейкам основной памяти. Обычно основная память изготавливается с применением полупроводниковых технологий и теряет свое содержимое при отключении питания.

Вторичную память (это главным образом диски) также можно рассматривать как одномерное линейное адресное пространство, состоящее из последовательности байтов. В отличие от оперативной памяти, она является энергонезависимой, имеет существенно большую емкость и используется в качестве расширения основной памяти.

Эту схему можно дополнить еще несколькими промежуточными уровнями, как показано на рис. 8.1. Разновидности памяти могут быть объединены в иерархию по убыванию времени доступа, возрастанию цены и увеличению емкости.

Рис. 8.1. Иерархия памяти

Многоуровневую схему используют следующим образом. Информация, которая находится в памяти верхнего уровня, обычно хранится также на уровнях с большими номерами. Если процессор не обнаруживает нужную информацию на i-м уровне, он начинает искать ее на следующих уровнях. Когда нужная информация найдена, она переносится в более быстрые уровни.

Локальность

Оказывается, при таком способе организации по мере снижения скорости доступа к уровню памяти снижается также и частота обращений к нему.

Ключевую роль здесь играет свойство реальных программ, в течение ограниченного отрезка времени способных работать с небольшим набором адресов памяти. Это эмпирически наблюдаемое свойство известно как принцип локальности или локализации обращений.

Свойство локальности (соседние в пространстве и времени объекты характеризуются похожими свойствами) присуще не только функционированию ОС, но и природе вообще. В случае ОС свойство локальности объяснимо, если учесть, как пишутся программы и как хранятся данные, то есть обычно в течение какого-то отрезка времени ограниченный фрагмент кода работает с ограниченным набором данных. Эту часть кода и данных удается разместить в памяти с быстрым доступом. В результате реальное время доступа к памяти определяется временем доступа к верхним уровням, что и обусловливает эффективность использования иерархической схемы. Надо сказать, что описываемая организация вычислительной системы во многом имитирует деятельность человеческого мозга при переработке информации. Действительно, решая конкретную проблему, человек работает с небольшим объемом информации, храня не относящиеся к делу сведения в своей памяти или во внешней памяти (например, в книгах).

Кэш процессора обычно является частью аппаратуры, поэтому менеджер памяти ОС занимается распределением информации главным образом в основной и внешней памяти компьютера. В некоторых схемах потоки между оперативной и внешней памятью регулируются программистом (см. например, далее оверлейные структуры), однако это связано с затратами времени программиста, так что подобную деятельность стараются возложить на ОС.

Адреса в основной памяти, характеризующие реальное расположение данных в физической памяти, называются физическими адресами. Набор физических адресов, с которым работает программа, называют физическим адресным пространством.

Логическая память

Аппаратная организация памяти в виде линейного набора ячеек не соответствует представлениям программиста о том, как организовано хранение программ и данных. Большинство программ представляет собой набор модулей, созданных независимо друг от друга. Иногда все модули, входящие в состав процесса, располагаются в памяти один за другим, образуя линейное пространство адресов. Однако чаще модули помещаются в разные области памяти и используются по-разному.

Схема управления памятью, поддерживающая этот взгляд пользователя на то, как хранятся программы и данные, называется сегментацией. Сегмент – область памяти определенного назначения, внутри которой поддерживается линейная адресация. Сегменты содержат процедуры, массивы, стек или скалярные величины, но обычно не содержат информацию смешанного типа.

По-видимому, вначале сегменты памяти появились в связи с необходимостью обобществления процессами фрагментов программного кода (текстовый редактор, тригонометрические библиотеки и т. д.), без чего каждый процесс должен был хранить в своем адресном пространстве дублирующую информацию. Эти отдельные участки памяти, хранящие информацию, которую система отображает в память нескольких процессов, получили название сегментов. Память, таким образом, перестала быть линейной и превратилась в двумерную. Адрес состоит из двух компонентов: номер сегмента, смещение внутри сегмента. Далее оказалось удобным размещать в разных сегментах различные компоненты процесса (код программы, данные, стек и т. д.). Попутно выяснилось, что можно контролировать характер работы с конкретным сегментом, приписав ему атрибуты, например права доступа или типы операций, которые разрешается производить с данными, хранящимися в сегменте.

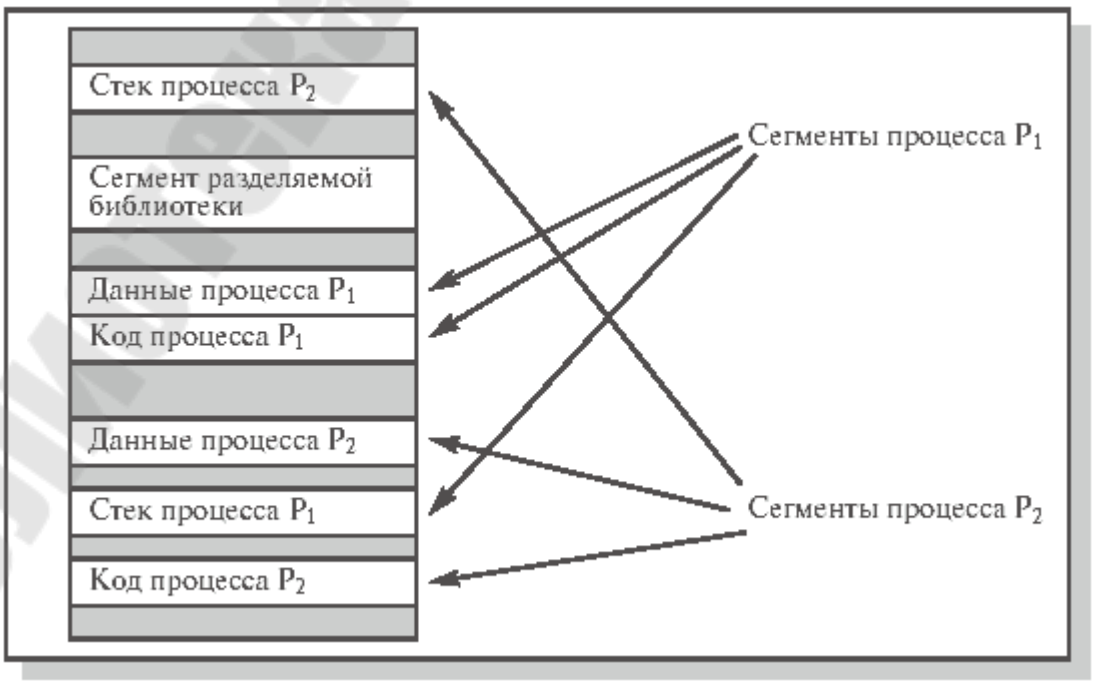

Рис. 8.2. Расположение сегментов процессов в памяти компьютера

Некоторые сегменты, описывающие адресное пространство процесса, показаны на рис. 8.2. Более подробная информация о типах сегментов имеется в лекции 10.

Большинство современных ОС поддерживают сегментную организацию памяти. В некоторых архитектурах (Intel, например) сегментация поддерживается оборудованием.

Адреса, к которым обращается процесс, таким образом, отличаются от адресов, реально существующих в оперативной памяти. В каждом конкретном случае используемые программой адреса могут быть представлены различными способами. Например, адреса в исходных текстах обычно символические. Компилятор связывает эти символические адреса с перемещаемыми адресами (такими, как n байт от начала модуля). Подобный адрес, сгенерированный программой, обычно называют логическим (в системах с виртуальной памятью он часто называется виртуальным) адресом. Совокупность всех логических адресов называется логическим (виртуальным) адресным пространством.

Связывание адресов

Итак логические и физические адресные пространства ни по организации, ни по размеру не соответствуют друг другу. Максимальный размер логического адресного пространства обычно определяется разрядностью процессора (например, 232) и в современных системах значительно превышает размер физического адресного пространства. Следовательно, процессор и ОС должны быть способны отобразить ссылки в коде программы в реальные физические адреса, соответствующие текущему расположению программы в основной памяти. Такое отображение адресов называют трансляцией (привязкой) адреса или связыванием адресов (см. рис. 8.3).

Связывание логического адреса, порожденного оператором программы, с физическим должно быть осуществлено до начала выполнения оператора или в момент его выполнения. Таким образом, привязка инструкций и данных к памяти в принципе может быть сделана на следующих шагах [Silberschatz, 2002].
Этап компиляции (Compile time). Когда на стадии компиляции известно точное место размещения процесса в памяти, тогда непосредственно генерируются физические адреса. При изменении стартового адреса программы необходимо перекомпилировать ее код. В качестве примера можно привести .com программы MS-DOS, которые связывают ее с физическими адресами на стадии компиляции.

Этап загрузки (Load time). Если информация о размещении программы на стадии компиляции отсутствует, компилятор генерирует перемещаемый код. В этом случае окончательное связывание откладывается до момента загрузки. Если стартовый адрес меняется, нужно всего лишь перезагрузить код с учетом измененной величины.

Этап выполнения (Execution time). Если процесс может быть перемещен во время выполнения из одной области памяти в другую, связывание откладывается до стадии выполнения. Здесь желательно наличие специализированного оборудования, например регистров перемещения. Их значение прибавляется к каждому адресу, сгенерированному процессом. Большинство современных ОС осуществляет трансляцию адресов на этапе выполнения, используя для этого специальный аппаратный механизм.

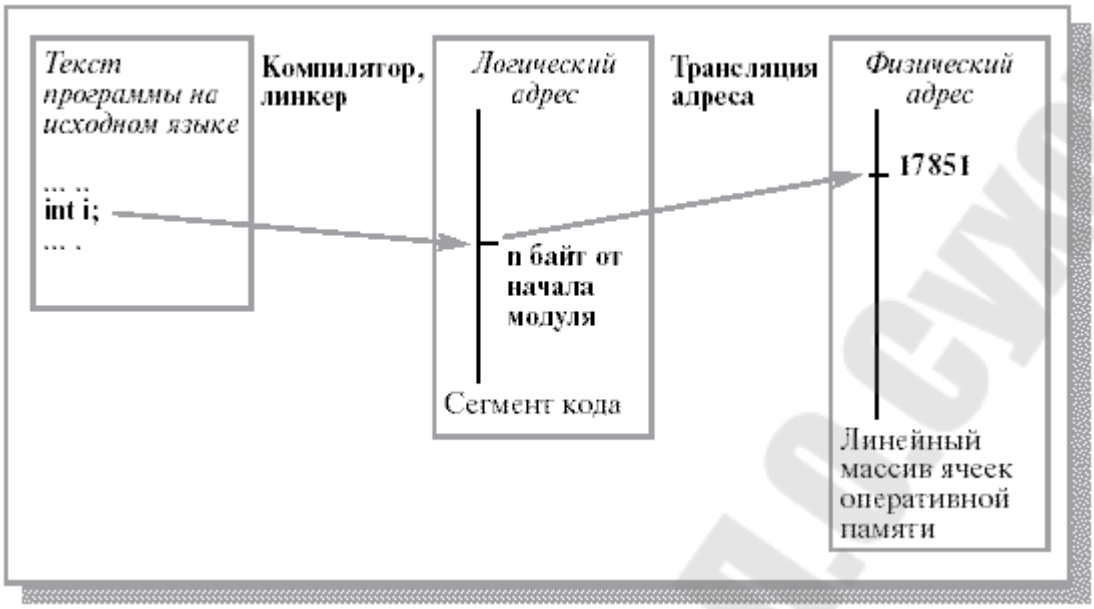

Рис. 8.3. Формирование логического адреса и связывание логического адреса с физическим

Функции системы управления памятью

Чтобы обеспечить эффективный контроль использования памяти, ОС должна выполнять следующие функции:

- отображение адресного пространства процесса на конкретные области физической памяти;
- распределение памяти между конкурирующими процессами;
- контроль доступа к адресным пространствам процессов;
- выгрузка процессов (целиком или частично) во внешнюю память, когда в оперативной памяти недостаточно места;
- учет свободной и занятой памяти.

В следующих разделах лекции рассматривается ряд конкретных схем управления памятью. Каждая схема включает в себя определенную идеологию управления, а также алгоритмы и структуры данных и зависит от архитектурных особенностей используемой системы. Вначале будут рассмотрены простейшие схемы. Доминирующая на сегодня схема виртуальной памяти будет описана в последующих лекциях.

Простейшие схемы управления памятью

Первые ОС применяли очень простые методы управления памятью. Вначале каждый процесс пользователя должен был полностью поместиться в основной памяти, занимать непрерывную область памяти, а система принимала к обслуживанию дополнительные пользовательские процессы до тех пор, пока все они одновременно помещались в основной памяти. Затем появился "простой свопинг" (система по-прежнему размещает каждый процесс в основной памяти целиком, но иногда на основании некоторого критерия целиком сбрасывает образ некоторого процесса из основной памяти во внешнюю и заменяет его в основной памяти образом другого процесса). Такого рода схемы имеют не только историческую ценность. В настоящее время они применяются в учебных и научно-исследовательских модельных ОС, а также в ОС для встроенных (embedded) компьютеров.

Схема с фиксированными разделами

Самым простым способом управления оперативной памятью является ее предварительное (обычно на этапе генерации или в момент загрузки системы) разбиение на несколько разделов фиксированной величины. Поступающие процессы помещаются в тот или иной раздел. При этом происходит условное разбиение физического адресного пространства. Связывание логических и физических адресов процесса происходит на этапе его загрузки в конкретный раздел, иногда – на этапе компиляции.

Каждый раздел может иметь свою очередь процессов, а может существовать и глобальная очередь для всех разделов (см. рис. 8.4).

Эта схема была реализована в IBM OS/360 (MFT), DEC RSX-11 и ряде других систем.

Подсистема управления памятью оценивает размер поступившего процесса, выбирает подходящий для него раздел, осуществляет загрузку процесса в этот раздел и настройку адресов.

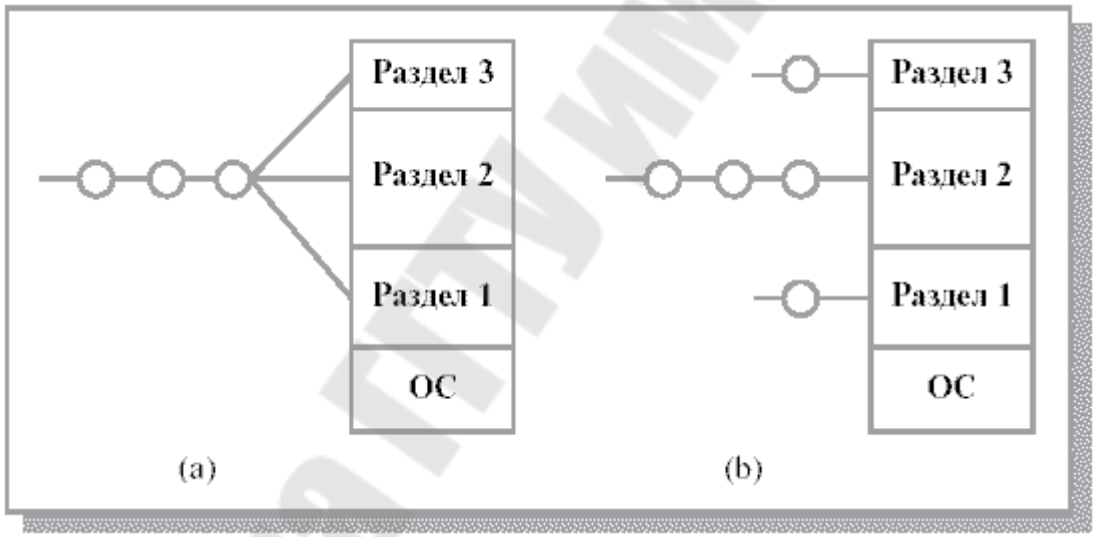

Рис. 8.4. Схема с фиксированными разделами: (a) – с общей очередью процессов, (b) – с отдельными очередями процессов

Очевидный недостаток этой схемы – число одновременно выполняемых процессов ограничено числом разделов.

Другим существенным недостатком является то, что предлагаемая схема сильно страдает от внутренней фрагментации – потери части памяти, выделенной процессу, но не используемой им. Фрагментация возникает потому, что процесс не полностью занимает выделенный ему раздел или потому, что некоторые разделы слишком малы для выполняемых пользовательских программ.

Один процесс в памяти

Частный случай схемы с фиксированными разделами – работа менеджера памяти однозадачной ОС. В памяти размещается один пользовательский процесс. Остается определить, где располагается пользовательская программа по отношению к ОС – в верхней части памяти, в нижней или в средней. Причем часть ОС может быть в ROM (например, BIOS, драйверы устройств). Главный фактор, влияющий на это решение, – расположение вектора прерываний, который обычно локализован в нижней части памяти,

поэтому ОС также размещают в нижней. Примером такой организации может служить ОС MS-DOS.

Защита адресного пространства ОС от пользовательской программы может быть организована при помощи одного граничного регистра, содержащего адрес границы ОС.

Оверлейная структура

Так как размер логического адресного пространства процесса может быть больше, чем размер выделенного ему раздела (или больше, чем размер самого большого раздела), иногда используется техника, называемая оверлей (overlay) или организация структуры с перекрытием. Основная идея – держать в памяти только те инструкции программы, которые нужны в данный момент.

Потребность в таком способе загрузки появляется, если логическое адресное пространство системы мало, например 1 Мбайт (MS-DOS) или даже всего 64 Кбайта (PDP-11), а программа относительно велика. На современных 32-разрядных системах, где виртуальное адресное пространство измеряется гигабайтами, проблемы с нехваткой памяти решаются другими способами (см. раздел "Виртуальная память").

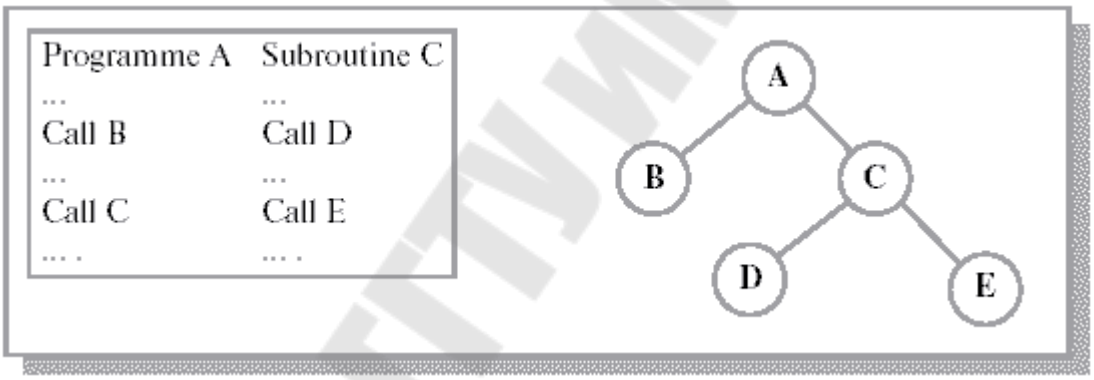

Рис. 8.5. Организация структуры с перекрытием. Можно поочередно загружать в память ветви A-B, A-C-D и A-C-E программы

Коды ветвей оверлейной структуры программы находятся на диске как абсолютные образы памяти и считываются драйвером оверлеев при необходимости. Для описания оверлейной структуры обычно используется специальный несложный язык (overlay description language). Совокупность файлов исполняемой программы дополняется файлом (обычно с расширением .odl), описывающим дерево вызовов внутри программы. Для примера, приведенного на рис. 8.5, текст этого файла может выглядеть так:
A-(B,C)
C-(D,E)

Синтаксис подобного файла может распознаваться загрузчиком. Привязка к физической памяти происходит в момент очередной загрузки одной из ветвей программы.

Оверлеи могут быть полностью реализованы на пользовательском уровне в системах с простой файловой структурой. ОС при этом лишь делает несколько больше операций ввода-вывода. Типовое решение – порождение линкером специальных команд, которые включают загрузчик каждый раз, когда требуется обращение к одной из перекрывающихся ветвей программы.

Тщательное проектирование оверлейной структуры отнимает много времени и требует знания устройства программы, ее кода, данных и языка описания оверлейной структуры. По этой причине применение оверлеев ограничено компьютерами с небольшим логическим адресным пространством. Как мы увидим в дальнейшем, проблема оверлейных сегментов, контролируемых программистом, отпадает благодаря появлению систем виртуальной памяти.

Заметим, что возможность организации структур с перекрытиями во многом обусловлена свойством локальности, которое позволяет хранить в памяти только ту информацию, которая необходима в конкретный момент вычислений.

Динамическое распределение. Свопинг

Имея дело с пакетными системами, можно обходиться фиксированными разделами и не использовать ничего более сложного. В системах с разделением времени возможна ситуация, когда память не в состоянии содержать все пользовательские процессы. Приходится прибегать к свопингу (swapping) – перемещению процессов из главной памяти на диск и обратно целиком. Частичная выгрузка процессов на диск осуществляется в системах со страничной организацией (paging) и будет рассмотрена ниже.

Выгруженный процесс может быть возвращен в то же самое адресное пространство или в другое. Это ограничение диктуется методом связывания. Для схемы связывания на этапе выполнения можно загрузить процесс в другое место памяти.

Свопинг не имеет непосредственного отношения к управлению памятью, скорее он связан с подсистемой планирования процессов. Очевидно, что свопинг увеличивает время переключения контекста. Время выгрузки может быть сокращено за счет организации специально отведенного пространства на диске (раздел для свопинга). Обмен с диском при этом осуществляется блоками большего размера, то есть быстрее, чем через стандартную файловую систему. Во многих версиях Unix свопинг начинает работать только тогда, когда возникает необходимость в снижении загрузки системы.

Схема с переменными разделами

В принципе, система свопинга может базироваться на фиксированных разделах. Более эффективной, однако, представляется схема динамического распределения или схема с переменными разделами, которая может использоваться и в тех случаях, когда все процессы целиком помещаются в памяти, то есть в отсутствие свопинга. В этом случае вначале вся память свободна и не разделена заранее на разделы. Вновь поступающей задаче выделяется строго необходимое количество памяти, не более. После выгрузки процесса память временно освобождается. По истечении некоторого времени память представляет собой переменное число разделов разного размера (рис. 8.6). Смежные свободные участки могут быть объединены.

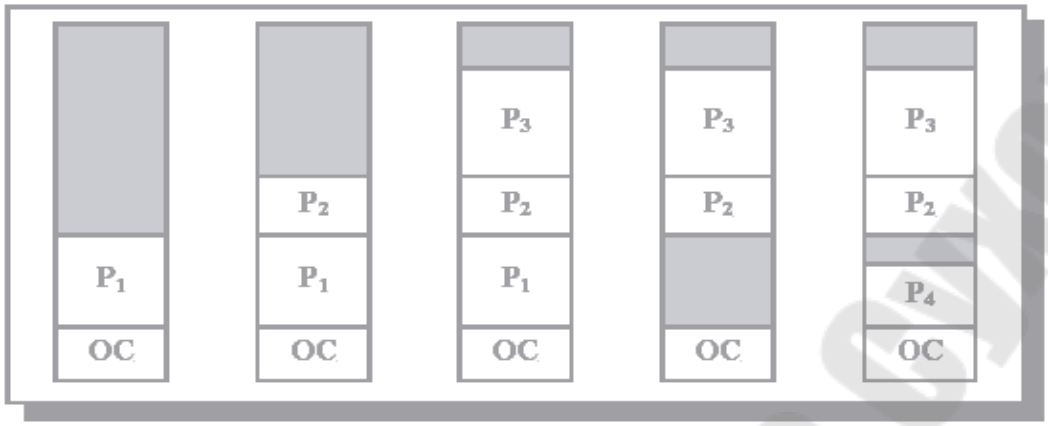

Рис. 8.6. Динамика распределения памяти между процессами (серым цветом показана неиспользуемая память)

В какой раздел помещать процесс? Наиболее распространены три стратегии:

- стратегия первого подходящего (First fit). Процесс помещается в первый подходящий по размеру раздел;
- стратегия наиболее подходящего (Best fit). Процесс помещается в тот раздел, где после его загрузки останется меньше всего свободного места;
- стратегия наименее подходящего (Worst fit). При помещении в самый большой раздел в нем остается достаточно места для возможного размещения еще одного процесса.

Моделирование показало, что доля полезно используемой памяти в первых двух случаях больше, при этом первый способ несколько быстрее. Попутно заметим, что перечисленные стратегии широко применяются и другими компонентами ОС, например для размещения файлов на диске.

Типовой цикл работы менеджера памяти состоит в анализе запроса на выделение свободного участка (раздела), выборе его среди имеющихся в соответствии с одной из стратегий (первого подходящего, наиболее подходящего и наименее подходящего), загрузке процесса в выбранный раздел и последующих изменениях таблиц свободных и занятых областей. Аналогичная корректировка необходима и после завершения процесса. Связывание адресов может осуществляться на этапах загрузки и выполнения.

Этот метод более гибок по сравнению с методом фиксированных разделов, однако ему присуща внешняя фрагментация – наличие большого числа участков неиспользуемой памяти, не выделенной ни одному процессу. Выбор стратегии размещения процесса между первым подходящим и наиболее подходящим слабо влияет на величину фрагментации. Любопытно, что метод наиболее подходящего может оказаться наихудшим, так как он оставляет множество мелких незанятых блоков.

Статистический анализ показывает, что пропадает в среднем 1/3 памяти! Это известное правило 50% (два соседних свободных участка в отличие от двух соседних процессов могут быть объединены).

Одно из решений проблемы внешней фрагментации – организовать сжатие, то есть перемещение всех занятых (свободных) участков в сторону возрастания (убывания) адресов, так, чтобы вся свободная память образовала непрерывную область. Этот метод иногда называют схемой с перемещаемыми разделами. В идеале фрагментация после сжатия должна отсутствовать. Сжатие, однако, является дорогостоящей процедурой,

алгоритм выбора оптимальной стратегии сжатия очень труден и, как правило, сжатие осуществляется в комбинации с выгрузкой и загрузкой по другим адресам.

Страничная память

Описанные выше схемы недостаточно эффективно используют память, поэтому в современных схемах управления памятью не принято размещать процесс в оперативной памяти одним непрерывным блоком.

В самом простом и наиболее распространенном случае страничной организации памяти (или paging) как логическое адресное пространство, так и физическое представляются состоящими из наборов блоков или страниц одинакового размера. При этом образуются логические страницы (page), а соответствующие единицы в физической памяти называют физическими страницами или страничными кадрами (page frames). Страницы (и страничные кадры) имеют фиксированную длину, обычно являющуюся степенью числа 2, и не могут перекрываться. Каждый кадр содержит одну страницу данных. При такой организации внешняя фрагментация отсутствует, а потери из-за внутренней фрагментации, поскольку процесс занимает целое число страниц, ограничены частью последней страницы процесса.

Логический адрес в страничной системе – упорядоченная пара (p,d), где p – номер страницы в виртуальной памяти, а d – смещение в рамках страницы p, на которой размещается адресуемый элемент. Заметим, что разбиение адресного пространства на страницы осуществляется вычислительной системой незаметно для программиста. Поэтому адрес является двумерным лишь с точки зрения операционной системы, а с точки зрения программиста адресное пространство процесса остается линейным.

Описываемая схема позволяет загрузить процесс, даже если нет непрерывной области кадров, достаточной для размещения процесса целиком. Но одного базового регистра для осуществления трансляции адреса в данной схеме недостаточно. Система отображения логических адресов в физические сводится к системе отображения логических страниц в физические и представляет собой таблицу страниц, которая хранится в оперативной памяти. Иногда говорят, что таблица страниц – это кусочно-линейная функция отображения, заданная в табличном виде.

Интерпретация логического адреса показана на рис. 8.7. Если выполняемый процесс обращается к логическому адресу v = (p,d), механизм отображения ищет номер страницы p в таблице страниц и определяет, что эта страница находится в страничном кадре p', формируя реальный адрес из p' и d.

Таблица страниц (page table) адресуется при помощи специального регистра процессора и позволяет определить номер кадра по логическому адресу. Помимо этой основной задачи, при помощи атрибутов, записанных в строке таблицы страниц, можно организовать контроль доступа к конкретной странице и ее защиту.

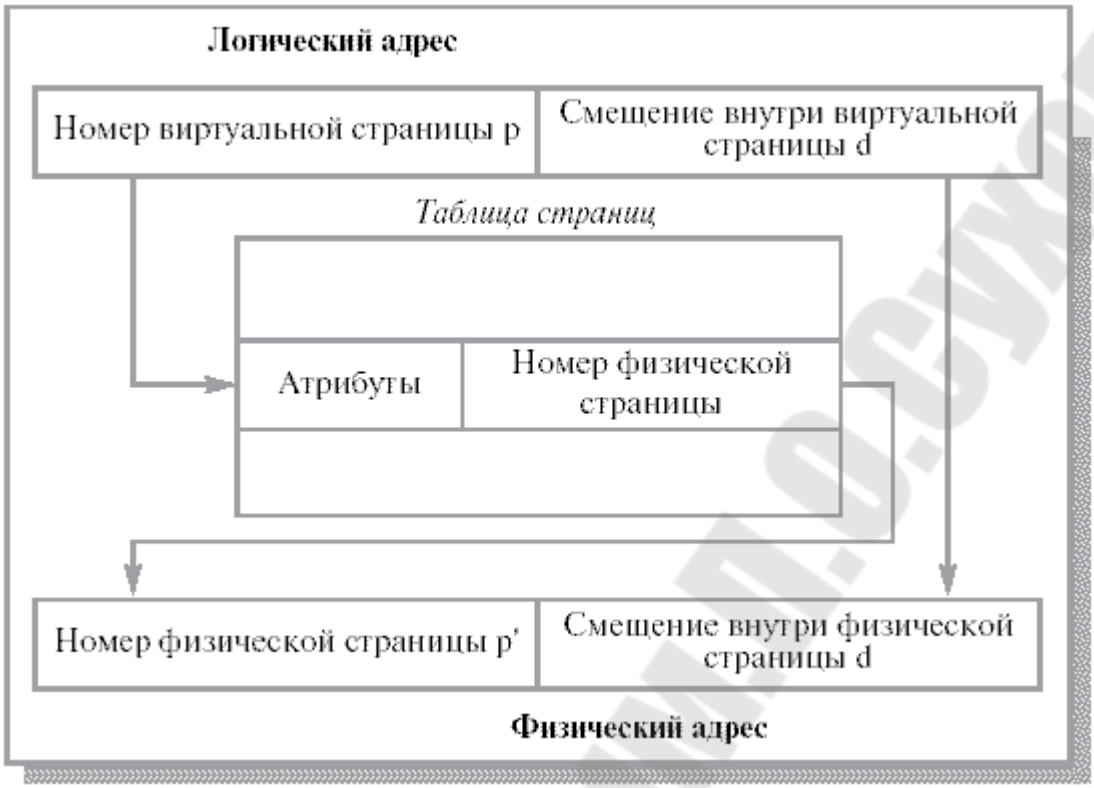

Рис. 8.7. Связь логического и физического адресов при страничной организации памяти

Отметим еще раз различие точек зрения пользователя и системы на используемую память. С точки зрения пользователя, его память – единое непрерывное пространство, содержащее только одну программу. Реальное отображение скрыто от пользователя и контролируется ОС. Заметим, что процессу пользователя чужая память недоступна. Он не имеет возможности адресовать память за пределами своей таблицы страниц, которая включает только его собственные страницы.

Для управления физической памятью ОС поддерживает структуру таблицы кадров. Она имеет одну запись на каждый физический кадр, показывающий его состояние.

Отображение адресов должно быть осуществлено корректно даже в сложных случаях и обычно реализуется аппаратно. Для ссылки на таблицу процессов используется специальный регистр. При переключении процессов необходимо найти таблицу страниц нового процесса, указатель на которую входит в контекст процесса.

Сегментная и сегментно-страничная организация памяти

Существуют две другие схемы организации управления памятью: сегментная и сегментно-страничная. Сегменты, в отличие от страниц, могут иметь переменный размер. Идея сегментации изложена во введении. При сегментной организации виртуальный адрес является двумерным как для программиста, так и для операционной системы, и состоит из двух полей – номера сегмента и смещения внутри сегмента. Подчеркнем, что в отличие от страничной организации, где линейный адрес преобразован в двумерный операционной системой для удобства отображения, здесь двумерность адреса является следствием представления пользователя о процессе не в виде линейного массива байтов, а как набор сегментов переменного размера (данные, код, стек...).

Программисты, пишущие на языках низкого уровня, должны иметь представление о сегментной организации, явным образом меняя значения сегментных регистров (это хорошо видно по текстам программ, написанных на Ассемблере). Логическое адресное пространство – набор сегментов. Каждый сегмент имеет имя, размер и другие параметры (уровень привилегий, разрешенные виды обращений, флаги присутствия). В отличие от страничной схемы, где пользователь задает только один адрес, который разбивается на номер страницы и смещение прозрачным для программиста образом, в сегментной схеме пользователь специфицирует каждый адрес двумя величинами: именем сегмента и смещением.

Каждый сегмент – линейная последовательность адресов, начинающаяся с 0. Максимальный размер сегмента определяется разрядностью процессора (при 32-разрядной адресации это 232 байт или 4 Гбайт). Размер сегмента может меняться динамически (например, сегмент стека). В элементе таблицы сегментов помимо физического адреса начала сегмента обычно содержится и длина сегмента. Если размер смещения в виртуальном адресе выходит за пределы размера сегмента, возникает исключительная ситуация.

Логический адрес – упорядоченная пара v=(s,d), номер сегмента и смещение внутри сегмента.

В системах, где сегменты поддерживаются аппаратно, эти параметры обычно хранятся в таблице дескрипторов сегментов, а программа обращается к этим дескрипторам по номерам-селекторам. При этом в контекст каждого процесса входит набор сегментных регистров, содержащих селекторы текущих сегментов кода, стека, данных и т. д. и определяющих, какие сегменты будут использоваться при разных видах обращений к памяти. Это позволяет процессору уже на аппаратном уровне определять допустимость обращений к памяти, упрощая реализацию защиты информации от повреждения и несанкционированного доступа.

Аппаратная поддержка сегментов распространена мало (главным образом на процессорах Intel). В большинстве ОС сегментация реализуется на уровне, не зависящем от аппаратуры.

Хранить в памяти сегменты большого размера целиком так же неудобно, как и хранить процесс непрерывным блоком. Напрашивается идея разбиения сегментов на страницы. При сегментно-страничной организации памяти происходит двухуровневая трансляция виртуального адреса в физический. В этом случае логический адрес состоит из трех полей: номера сегмента логической памяти, номера страницы внутри сегмента и смещения внутри страницы. Соответственно, используются две таблицы отображения – таблица сегментов, связывающая номер сегмента с таблицей страниц, и отдельная таблица страниц для каждого сегмента.

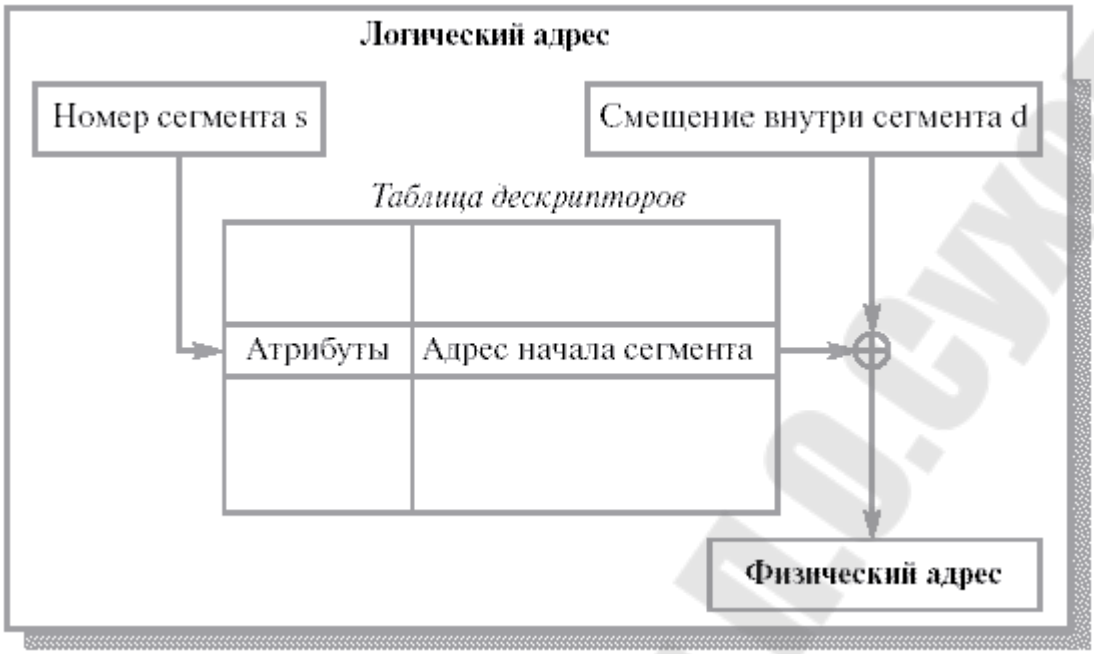

Рис. 8.8. Преобразование логического адреса при сегментной организации памяти

Сегментно-страничная и страничная организация памяти позволяет легко организовать совместное использование одних и тех же данных и программного кода разными задачами. Для этого различные логические блоки памяти разных процессов отображают в один и тот же блок физической памяти, где размещается разделяемый фрагмент кода или данных.

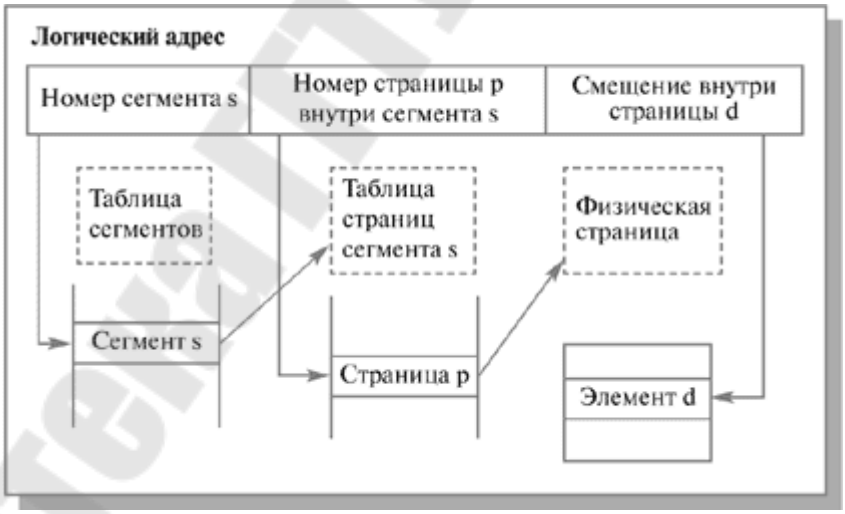

Рис. 8.9. Упрощенная схема формирования физического адреса при сегментно-страничной организации памяти

Лекция. Виртуальная память.

Понятие виртуальной памяти

Разработчикам программного обеспечения часто приходится решать проблему размещения в памяти больших программ, размер которых превышает объем доступной оперативной памяти. Один из вариантов решения данной проблемы – организация структур с перекрытием – рассмотрен в предыдущей лекции. При этом предполагалось активное участие программиста в процессе формирования перекрывающихся частей

программы. Развитие архитектуры компьютеров и расширение возможностей операционной системы по управлению памятью позволило переложить решение этой задачи на компьютер. Одним из главных достижений стало появление виртуальной памяти (virtual memory). Впервые она была реализована в 1959 г. на компьютере «Атлас», разработанном в Манчестерском университете.

Суть концепции виртуальной памяти заключается в следующем. Информация, с которой работает активный процесс, должна располагаться в оперативной памяти. В схемах виртуальной памяти у процесса создается иллюзия того, что вся необходимая ему информация имеется в основной памяти. Для этого, во-первых, занимаемая процессом память разбивается на несколько частей, например страниц. Во-вторых, логический адрес (логическая страница), к которому обращается процесс, динамически транслируется в физический адрес (физическую страницу). И наконец, в тех случаях, когда страница, к которой обращается процесс, не находится в физической памяти, нужно организовать ее подкачку с диска. Для контроля наличия страницы в памяти вводится специальный бит присутствия, входящий в состав атрибутов страницы в таблице страниц.

Таким образом, в наличии всех компонентов процесса в основной памяти необходимости нет. Важным следствием такой организации является то, что размер памяти, занимаемой процессом, может быть больше, чем размер оперативной памяти. Принцип локальности обеспечивает этой схеме нужную эффективность.

Возможность выполнения программы, находящейся в памяти лишь частично, имеет ряд вполне очевидных преимуществ.
Программа не ограничена объемом физической памяти. Упрощается разработка программ, поскольку можно задействовать большие виртуальные пространства, не заботясь о размере используемой памяти.
Поскольку появляется возможность частичного помещения программы (процесса) в память и гибкого перераспределения памяти между программами, можно разместить в памяти больше программ, что увеличивает загрузку процессора и пропускную способность системы.
Объем ввода-вывода для выгрузки части программы на диск может быть меньше, чем в варианте классического свопинга, в итоге каждая программа будет работать быстрее.

Таким образом, возможность обеспечения (при поддержке операционной системы) для программы «видимости» практически неограниченной (характерный размер для 32-разрядных архитектур 232 = 4 Гбайт) адресуемой пользовательской памяти (логическое адресное пространство) при наличии основной памяти существенно меньших размеров (физическое адресное пространство) – очень важный аспект.

Но введение виртуальной памяти позволяет решать другую, не менее важную задачу – обеспечение контроля доступа к отдельным сегментам памяти и, в частности, защиту пользовательских программ друг от друга и защиту ОС от пользовательских программ. Каждый процесс работает со своими виртуальными адресами, трансляцию которых в физические выполняет аппаратура компьютера. Таким образом, пользовательский процесс лишен возможности напрямую обратиться к страницам основной памяти, занятым информацией, относящейся к другим процессам.

Например, 16-разрядный компьютер PDP-11/70 с 64 Кбайт логической памяти мог иметь до 2 Мбайт оперативной памяти. Операционная система этого компьютера тем не менее поддерживала виртуальную память, которая обеспечивала защиту и перераспределение основной памяти между пользовательскими процессами.

Напомним, что в системах с виртуальной памятью те адреса, которые генерирует программа (логические адреса), называются виртуальными, и они формируют виртуальное адресное пространство. Термин «виртуальная память» означает, что программист имеет дело с памятью, отличной от реальной, размер которой потенциально больше, чем размер оперативной памяти.

Хотя известны и чисто программные реализации виртуальной памяти, это направление получило наиболее широкое развитие после соответствующей аппаратной поддержки.

Следует отметить, что оборудование компьютера принимает участие в трансляции адреса практически во всех схемах управления памятью. Но в случае виртуальной памяти это становится более сложным вследствие разрывности отображения и многомерности логического адресного пространства. Может быть, наиболее существенным вкладом аппаратуры в реализацию описываемой схемы является автоматическая генерация исключительных ситуаций при отсутствии в памяти нужных страниц (page fault).

Любая из трех ранее рассмотренных схем управления памятью – страничной, сегментной и сегментно-страничной – пригодна для организации виртуальной памяти. Чаще всего используется сегментно-страничная модель, которая является синтезом страничной модели и идеи сегментации. Причем для тех архитектур, в которых сегменты не поддерживаются аппаратно, их реализация – задача архитектурно-независимого компонента менеджера памяти.

Сегментная организация в чистом виде встречается редко.

Архитектурные средства поддержки виртуальной памяти

Очевидно, что невозможно создать полностью машинно-независимый компонент управления виртуальной памятью. С другой стороны, имеются существенные части программного обеспечения, связанного с управлением виртуальной памятью, для которых детали аппаратной реализации совершенно не важны. Одним из достижений современных ОС является грамотное и эффективное разделение средств управления виртуальной памятью на аппаратно-независимую и аппаратно-зависимую части. Коротко рассмотрим, что и каким образом входит в аппаратно-зависимую часть подсистемы управления виртуальной памятью. Компоненты аппаратно-независимой подсистемы будут рассмотрены в следующей лекции.

В самом распространенном случае необходимо отобразить большое виртуальное адресное пространство в физическое адресное пространство существенно меньшего размера. Пользовательский процесс или ОС должны иметь возможность осуществить запись по виртуальному адресу, а задача ОС – сделать так, чтобы записанная информация оказалась в физической памяти (впоследствии при нехватке оперативной памяти она может быть вытеснена во внешнюю память). В случае виртуальной памяти система отображения адресных пространств помимо трансляции адресов должна предусматривать ведение таблиц, показывающих, какие области виртуальной памяти в данный момент находятся в физической памяти и где именно размещаются.

Страничная виртуальная память

Как и в случае простой страничной организации, страничная виртуальная память и физическая память представляются состоящими из наборов блоков или страниц

одинакового размера. Виртуальные адреса делятся на страницы (page), соответствующие единицы в физической памяти образуют страничные кадры (page frames), а в целом система поддержки страничной виртуальной памяти называется пейджингом (paging). Передача информации между памятью и диском всегда осуществляется целыми страницами.

После разбиения менеджером памяти виртуального адресного пространства на страницы виртуальный адрес преобразуется в упорядоченную пару (p,d), где p – номер страницы в виртуальной памяти, а d – смещение в рамках страницы p, внутри которой размещается адресуемый элемент. Процесс может выполняться, если его текущая страница находится в оперативной памяти. Если текущей страницы в главной памяти нет, она должна быть переписана (подкачана) из внешней памяти. Поступившую страницу можно поместить в любой свободный страничный кадр.

Поскольку число виртуальных страниц велико, таблица страниц принимает специфический вид (см. раздел «Структура таблицы страниц»), структура записей становится более сложной, среди атрибутов страницы появляются биты присутствия, модификации и другие управляющие биты.

При отсутствии страницы в памяти в процессе выполнения команды возникает исключительная ситуация, называемая страничное нарушение (page fault) или страничный отказ. Обработка страничного нарушения заключается в том, что выполнение команды прерывается, затребованная страница подкачивается из конкретного места вторичной памяти в свободный страничный кадр физической памяти и попытка выполнения команды повторяется. При отсутствии свободных страничных кадров на диск выгружается редко используемая страница. Проблемы замещения страниц и обработки страничных нарушений рассматриваются в следующей лекции.

Для управления физической памятью ОС поддерживает структуру таблицы кадров. Она имеет одну запись на каждый физический кадр, показывающий его состояние.

В большинстве современных компьютеров со страничной организацией в основной памяти хранится лишь часть таблицы страниц, а быстрота доступа к элементам таблицы текущей виртуальной памяти достигается, как будет показано ниже, за счет использования сверхбыстродействующей памяти, размещенной в кэше процессора.

Сегментно-страничная организации виртуальной памяти

Как и в случае простой сегментации, в схемах виртуальной памяти сегмент – это линейная последовательность адресов, начинающаяся с 0. При организации виртуальной памяти размер сегмента может быть велик, например может превышать размер оперативной памяти. Повторяя все ранее приведенные рассуждения о размещении в памяти больших программ, приходим к разбиению сегментов на страницы и необходимости поддержки своей таблицы страниц для каждого сегмента.

На практике, однако, появления в системе большого количества таблиц страниц стараются избежать, организуя неперекрывающиеся сегменты в одном виртуальном пространстве, для описания которого хватает одной таблицы страниц. Таким образом, одна таблица страниц отводится для всего процесса. Например, в популярных ОС Linux и Windows 2000 все сегменты процесса, а также область памяти ядра ограничены виртуальным адресным пространством объемом 4 Гбайт. При этом ядро ОС располагается по фиксированным виртуальным адресам вне зависимости от выполняемого процесса.

Структура таблицы страниц

Организация таблицы страниц – один из ключевых элементов отображения адресов в страничной и сегментно-страничной схемах. Рассмотрим структуру таблицы страниц для случая страничной организации более подробно.

Итак, виртуальный адрес состоит из виртуального номера страницы и смещения. Номер записи в таблице страниц соответствует номеру виртуальной страницы. Размер записи колеблется от системы к системе, но чаще всего он составляет 32 бита. Из этой записи в таблице страниц находится номер кадра для данной виртуальной страницы, затем прибавляется смещение и формируется физический адрес. Помимо этого запись в таблице страниц содержит информацию об атрибутах страницы. Это биты присутствия и защиты (например, 0 – read/write, 1 – read only...). Также могут быть указаны: бит модификации, который устанавливается, если содержимое страницы модифицировано, и позволяет контролировать необходимость перезаписи страницы на диск; бит ссылки, который помогает выделить малоиспользуемые страницы; бит, разрешающий кэширование, и другие управляющие биты. Заметим, что адреса страниц на диске не являются частью таблицы страниц.

Основную проблему для эффективной реализации таблицы страниц создают большие размеры виртуальных адресных пространств современных компьютеров, которые обычно определяются разрядностью архитектуры процессора. Самыми распространенными на сегодня являются 32-разрядные процессоры, позволяющие создавать виртуальные адресные пространства размером 4 Гбайт (для 64-разрядных компьютеров эта величина равна 264 байт). Кроме того, существует проблема скорости отображения, которая решается за счет использования так называемой ассоциативной памяти (см. следующий раздел).

Подсчитаем примерный размер таблицы страниц. В 32-битном адресном пространстве при размере страницы 4 Кбайт (Intel) получаем 232/212=220, то есть приблизительно миллион страниц, а в 64-битном и того более. Таким образом, таблица должна иметь примерно миллион строк (entry), причем запись в строке состоит из нескольких байтов. Заметим, что каждый процесс нуждается в своей таблице страниц (а в случае сегментно-страничной схемы желательно иметь по одной таблице страниц на каждый сегмент).

Понятно, что количество памяти, отводимое таблицам страниц, не может быть так велико. Для того чтобы избежать размещения в памяти огромной таблицы, ее разбивают на ряд фрагментов. В оперативной памяти хранят лишь некоторые, необходимые для конкретного момента исполнения фрагменты таблицы страниц. В силу свойства локальности число таких фрагментов относительно невелико. Выполнить разбиение таблицы страниц на части можно по-разному. Наиболее распространенный способ разбиения – организация так называемой многоуровневой таблицы страниц. Для примера рассмотрим двухуровневую таблицу с размером страниц 4 Кбайт, реализованную в 32-разрядной архитектуре Intel.

Таблица, состоящая из 220 строк, разбивается на 210 таблиц второго уровня по 210 строк. Эти таблицы второго уровня объединены в общую структуру при помощи одной таблицы первого уровня, состоящей из 210 строк. 32-разрядный адрес делится на 10-разрядное поле p1, 10-разрядное поле p2 и 12-разрядное смещение d. Поле p1 указывает на нужную строку в таблице первого уровня, поле p2 – второго, а поле d локализует нужный байт внутри указанного страничного кадра (см. рис. 9.1).

При помощи всего лишь одной таблицы второго уровня можно охватить 4 Мбайт (4 Кбайт x 1024) оперативной памяти. Таким образом, для размещения процесса с большим объемом занимаемой памяти достаточно иметь в оперативной памяти одну таблицу первого уровня и несколько таблиц второго уровня. Очевидно, что суммарное количество строк в этих таблицах много меньше 220. Такой подход естественным образом обобщается на три и более уровней таблицы.

Наличие нескольких уровней, естественно, снижает производительность менеджера памяти. Несмотря на то что размеры таблиц на каждом уровне подобраны так, чтобы таблица помещалась целиком внутри одной страницы, обращение к каждому уровню – это отдельное обращение к памяти. Таким образом, трансляция адреса может потребовать нескольких обращений к памяти.

Количество уровней в таблице страниц зависит от конкретных особенностей архитектуры. Можно привести примеры реализации одноуровневого (DEC PDP-11), двухуровневого (Intel, DEC VAX), трехуровневого (Sun SPARC, DEC Alpha) пейджинга, а также пейджинга с заданным количеством уровней (Motorola). Функционирование RISC-процессора MIPS R2000 осуществляется вообще без таблицы страниц. Здесь поиск нужной страницы, если эта страница отсутствует в ассоциативной памяти, должна взять на себя ОС (так называемый zero level paging).

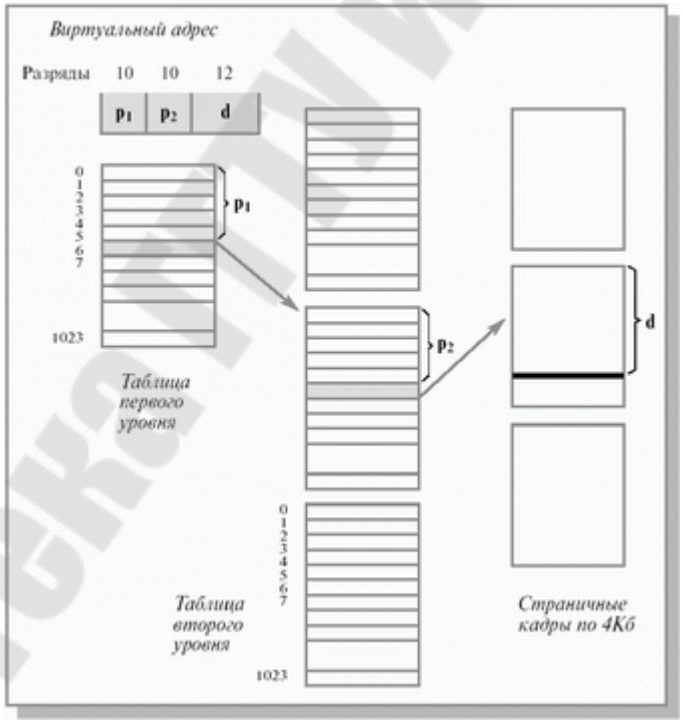

Рис. 9.1. Пример двухуровневой таблицы страниц

Ассоциативная память

Поиск номера кадра, соответствующего нужной странице, в многоуровневой таблице страниц требует нескольких обращений к основной памяти, поэтому занимает много времени. В некоторых случаях такая задержка недопустима. Проблема ускорения поиска решается на уровне архитектуры компьютера.

В соответствии со свойством локальности большинство программ в течение некоторого промежутка времени обращаются к небольшому количеству страниц, поэтому активно используется только небольшая часть таблицы страниц.

Естественное решение проблемы ускорения – снабдить компьютер аппаратным устройством для отображения виртуальных страниц в физические без обращения к таблице страниц, то есть иметь небольшую, быструю кэш-память, хранящую необходимую на данный момент часть таблицы страниц. Это устройство называется ассоциативной памятью, иногда также употребляют термин буфер поиска трансляции (translation lookaside buffer – TLB).

Одна запись таблицы в ассоциативной памяти (один вход) содержит информацию об одной виртуальной странице: ее атрибуты и кадр, в котором она находится. Эти поля в точности соответствуют полям в таблице страниц.

Так как ассоциативная память содержит только некоторые из записей таблицы страниц, каждая запись в TLB должна включать поле с номером виртуальной страницы. Память называется ассоциативной, потому что в ней происходит одновременное сравнение номера отображаемой виртуальной страницы с соответствующим полем во всех строках этой небольшой таблицы. Поэтому данный вид памяти достаточно дорого стоит. В строке, поле виртуальной страницы которой совпало с искомым значением, находится номер страничного кадра. Обычное число записей в TLB от 8 до 4096. Рост количества записей в ассоциативной памяти должен осуществляться с учетом таких факторов, как размер кэша основной памяти и количества обращений к памяти при выполнении одной команды.

Рассмотрим функционирование менеджера памяти при наличии ассоциативной памяти.

Вначале информация об отображении виртуальной страницы в физическую отыскивается в ассоциативной памяти. Если нужная запись найдена – все нормально, за исключением случаев нарушения привилегий, когда запрос на обращение к памяти отклоняется.

Если нужная запись в ассоциативной памяти отсутствует, отображение осуществляется через таблицу страниц. Происходит замена одной из записей в ассоциативной памяти найденной записью из таблицы страниц. Здесь мы сталкиваемся с традиционной для любого кэша проблемой замещения (а именно какую из записей в кэше необходимо изменить). Конструкция ассоциативной памяти должна организовывать записи таким образом, чтобы можно было принять решение о том, какая из старых записей должна быть удалена при внесении новых.

Число удачных поисков номера страницы в ассоциативной памяти по отношению к общему числу поисков называется hit (совпадение) ratio (пропорция, отношение). Иногда также используется термин «процент попаданий в кэш». Таким образом, hit ratio – часть ссылок, которая может быть сделана с использованием ассоциативной памяти. Обращение к одним и тем же страницам повышает hit ratio. Чем больше hit ratio, тем меньше среднее время доступа к данным, находящимся в оперативной памяти.

Предположим, например, что для определения адреса в случае кэш-промаха через таблицу страниц необходимо 100 нс, а для определения адреса в случае кэш-попадания через ассоциативную память – 20 нс. С 90% hit ratio среднее время определения адреса – 0,9x20+0,1x100 = 28 нс.

Вполне приемлемая производительность современных ОС доказывает эффективность использования ассоциативной памяти. Высокое значение вероятности нахождения данных в ассоциативной памяти связано с наличием у данных объективных свойств: пространственной и временной локальности.

Необходимо обратить внимание на следующий факт. При переключении контекста процессов нужно добиться того, чтобы новый процесс «не видел» в ассоциативной памяти информацию, относящуюся к предыдущему процессу, например очищать ее. Таким образом, использование ассоциативной памяти увеличивает время переключения контекста.

Рассмотренная двухуровневая (ассоциативная память + таблица страниц) схема преобразования адреса является ярким примером иерархии памяти, основанной на использовании принципа локальности, о чем говорилось во введении к предыдущей лекции.

Инвертированная таблица страниц

Несмотря на многоуровневую организацию, хранение нескольких таблиц страниц большого размера по-прежнему представляют собой проблему. Ее значение особенно актуально для 64-разрядных архитектур, где число виртуальных страниц очень велико. Вариантом решения является применение инвертированной таблицы страниц (inverted page table). Этот подход применяется на машинах PowerPC, некоторых рабочих станциях Hewlett-Packard, IBM RT, IBM AS/400 и ряде других.

В этой таблице содержится по одной записи на каждый страничный кадр физической памяти. Существенно, что достаточно одной таблицы для всех процессов. Таким образом, для хранения функции отображения требуется фиксированная часть основной памяти, независимо от разрядности архитектуры, размера и количества процессов. Например, для компьютера Pentium с 256 Мбайт оперативной памяти нужна таблица размером 64 Кбайт строк.

Несмотря на экономию оперативной памяти, применение инвертированной таблицы имеет существенный минус – записи в ней (как и в ассоциативной памяти) не отсортированы по возрастанию номеров виртуальных страниц, что усложняет трансляцию адреса. Один из способов решения данной проблемы – использование хеш-таблицы виртуальных адресов. При этом часть виртуального адреса, представляющая собой номер страницы, отображается в хеш-таблицу с использованием функции хеширования. Каждой странице физической памяти здесь соответствует одна запись в хеш-таблице и инвертированной таблице страниц. Виртуальные адреса, имеющие одно значение хеш-функции, сцепляются друг с другом. Обычно длина цепочки не превышает двух записей.

Размер страницы

Разработчики ОС для существующих машин редко имеют возможность влиять на размер страницы. Однако для вновь создаваемых компьютеров решение относительно оптимального размера страницы является актуальным. Как и следовало ожидать, нет одного наилучшего размера. Скорее есть набор факторов, влияющих на размер. Обычно размер страницы – это степень двойки от 29 до 214 байт.

Чем больше размер страницы, тем меньше будет размер структур данных, обслуживающих преобразование адресов, но тем больше будут потери, связанные с тем, что память можно выделять только постранично.

Как следует выбирать размер страницы? Во-первых, нужно учитывать размер таблицы страниц, здесь желателен большой размер страницы (страниц меньше, соответственно и таблица страниц меньше). С другой стороны, память лучше утилизируется с маленьким размером страницы. В среднем половина последней страницы процесса пропадает. Необходимо также учитывать объем ввода-вывода для взаимодействия с внешней памятью и другие факторы. Проблема не имеет идеального решения. Историческая тенденция состоит в увеличении размера страницы.

Как правило, размер страниц задается аппаратно, например в DEC PDP-11 – 8 Кбайт, в DEC VAX – 512 байт, в других архитектурах, таких как Motorola 68030, размер страниц может быть задан программно. Учитывая все обстоятельства, в ряде архитектур возникают множественные размеры страниц, например в Pentium размер страницы колеблется от 4 Кбайт до 8 Кбайт. Тем не менее большинство коммерческих ОС ввиду сложности перехода на множественный размер страниц поддерживают только один размер страниц.

Лекция 11. Управление виртуальной памятью

Исключительные ситуации при работе с памятью

Из материала предыдущей лекции следует, что отображение виртуального адреса в физический осуществляется при помощи таблицы страниц. Для каждой виртуальной страницы запись в таблице страниц содержит номер соответствующего страничного кадра в оперативной памяти, а также атрибуты страницы для контроля обращений к памяти.

Что же происходит, когда нужной страницы в памяти нет или операция обращения к памяти недопустима? Естественно, что операционная система должна быть как-то оповещена о происшедшем. Обычно для этого используется механизм исключительных ситуаций (exceptions). При попытке выполнить подобное обращение к виртуальной странице возникает исключительная ситуация "страничное нарушение" (page fault), приводящая к вызову специальной последовательности команд для обработки конкретного вида страничного нарушения.

Страничное нарушение может происходить в самых разных случаях: при отсутствии страницы в оперативной памяти, при попытке записи в страницу с атрибутом "только чтение" или при попытке чтения или записи страницы с атрибутом "только выполнение". В любом из этих случаев вызывается обработчик страничного нарушения, являющийся частью операционной системы. Ему обычно передается причина возникновения исключительной ситуации и виртуальный адрес, обращение к которому вызвало нарушение.

Нас будет интересовать конкретный вариант страничного нарушения - обращение к отсутствующей странице, поскольку именно его обработка во многом определяет производительность страничной системы. Когда программа обращается к виртуальной странице, отсутствующей в основной памяти, операционная система должна выделить страницу основной памяти, переместить в нее копию виртуальной страницы из внешней памяти и модифицировать соответствующий элемент таблицы страниц.

Повышение производительности вычислительной системы может быть достигнуто за счет уменьшения частоты страничных нарушений, а также за счет увеличения скорости их обработки. Время эффективного доступа к отсутствующей в оперативной памяти странице складывается из:
обслуживания исключительной ситуации (page fault);
чтения (подкачки) страницы из вторичной памяти (иногда, при недостатке места в основной памяти, необходимо вытолкнуть одну из страниц из основной памяти во вторичную, то есть осуществить замещение страницы);
возобновления выполнения процесса, вызвавшего данный page fault.

Для решения первой и третьей задач ОС выполняет до нескольких сот машинных инструкций в течение нескольких десятков микросекунд. Время подкачки страницы близко к нескольким десяткам миллисекунд. Проведенные исследования показывают, что вероятности page fault 5x10-7 оказывается достаточно, чтобы снизить производительность страничной схемы управления памятью на 10%. Таким образом, уменьшение частоты page faults является одной из ключевых задач системы управления памятью. Ее решение обычно связано с правильным выбором алгоритма замещения страниц.

Стратегии управления страничной памятью

Программное обеспечение подсистемы управления памятью связано с реализацией следующих стратегий:

Стратегия выборки (fetch policy) - в какой момент следует переписать страницу из вторичной памяти в первичную. Существует два основных варианта выборки - по запросу и с упреждением. Алгоритм выборки по запросу вступает в действие в тот момент, когда процесс обращается к отсутствующей странице, содержимое которой находится на диске. Его реализация заключается в загрузке страницы с диска в свободную физическую страницу и коррекции соответствующей записи таблицы страниц.

Алгоритм выборки с упреждением осуществляет опережающее чтение, то есть кроме страницы, вызвавшей исключительную ситуацию, в память также загружается несколько страниц, окружающих ее (обычно соседние страницы располагаются во внешней памяти последовательно и могут быть считаны за одно обращение к диску). Такой алгоритм призван уменьшить накладные расходы, связанные с большим количеством исключительных ситуаций, возникающих при работе со значительными объемами данных или кода; кроме того, оптимизируется работа с диском.

Стратегия размещения (placement policy) - в какой участок первичной памяти поместить поступающую страницу. В системах со страничной организацией все просто - в любой свободный страничный кадр. В случае систем с сегментной организацией необходима стратегия, аналогичная стратегии с динамическим распределением.

Стратегия замещения (replacement policy) - какую страницу нужно вытолкнуть во внешнюю память, чтобы освободить место в оперативной памяти. Разумная стратегия замещения, реализованная в соответствующем алгоритме замещения страниц, позволяет хранить в памяти самую необходимую информацию и тем самым снизить частоту страничных нарушений. Замещение должно происходить с учетом выделенного каждому процессу количества кадров. Кроме того, нужно решить, должна ли замещаемая страница принадлежать процессу, который инициировал замещение, или она должна быть выбрана среди всех кадров основной памяти.

Алгоритмы замещения страниц

Итак, наиболее ответственным действием менеджера памяти является выделение кадра оперативной памяти для размещения в ней виртуальной страницы, находящейся во внешней памяти. Напомним, что мы рассматриваем ситуацию, когда размер виртуальной памяти для каждого процесса может существенно превосходить размер основной памяти. Это означает, что при выделении страницы основной памяти с большой вероятностью не удастся найти свободный страничный кадр. В этом случае операционная система в соответствии с заложенными в нее критериями должна:

- найти некоторую занятую страницу основной памяти;
- переместить в случае надобности ее содержимое во внешнюю память;
- переписать в этот страничный кадр содержимое нужной виртуальной страницы из внешней памяти;
- должным образом модифицировать необходимый элемент соответствующей таблицы страниц;
- продолжить выполнение процесса, которому эта виртуальная страница понадобилась.

Заметим, что при замещении приходится дважды передавать страницу между основной и вторичной памятью. Процесс замещения может быть оптимизирован за счет использования бита модификации (один из атрибутов страницы в таблице страниц). Бит модификации устанавливается компьютером, если хотя бы один байт был записан на страницу. При выборе кандидата на замещение проверяется бит модификации. Если бит не установлен, нет необходимости переписывать данную страницу на диск, ее копия на диске уже имеется. Подобный метод также применяется к read-only-страницам, они никогда не модифицируются. Эта схема уменьшает время обработки page fault.

Существует большое количество разнообразных алгоритмов замещения страниц. Все они делятся на локальные и глобальные. Локальные алгоритмы, в отличие от глобальных, распределяют фиксированное или динамически настраиваемое число страниц для каждого процесса. Когда процесс израсходует все предназначенные ему страницы, система будет удалять из физической памяти одну из его страниц, а не из страниц других процессов. Глобальный же алгоритм замещения в случае возникновения исключительной ситуации удовлетворится освобождением любой физической страницы, независимо от того, какому процессу она принадлежала.

Глобальные алгоритмы имеют ряд недостатков. Во-первых, они делают одни процессы чувствительными к поведению других процессов. Например, если один процесс в системе одновременно использует большое количество страниц памяти, то все остальные приложения будут в результате ощущать сильное замедление из-за недостатка кадров памяти для своей работы. Во-вторых, некорректно работающее приложение может подорвать работу всей системы (если, конечно, в системе не предусмотрено ограничение на размер памяти, выделяемой процессу), пытаясь захватить больше памяти. Поэтому в многозадачной системе иногда приходится использовать более сложные локальные алгоритмы. Применение локальных алгоритмов требует хранения в операционной системе списка физических кадров, выделенных каждому процессу. Этот список страниц иногда называют резидентным множеством процесса. В одном из следующих разделов рассмотрен вариант алгоритма подкачки, основанный на приведении резидентного множества в соответствие так называемому рабочему набору процесса.

Эффективность алгоритма обычно оценивается на конкретной последовательности ссылок к памяти, для которой подсчитывается число возникающих page faults. Эта

последовательность называется строкой обращений (reference string). Мы можем генерировать строку обращений искусственным образом при помощи датчика случайных чисел или трассируя конкретную систему. Последний метод дает слишком много ссылок, для уменьшения числа которых можно сделать две вещи:
для конкретного размера страниц можно запоминать только их номера, а не адреса, на которые идет ссылка; несколько подряд идущих ссылок на одну страницу можно фиксировать один раз.

Как уже говорилось, большинство процессоров имеют простейшие аппаратные средства, позволяющие собирать некоторую статистику обращений к памяти. Эти средства обычно включают два специальных флага на каждый элемент таблицы страниц. Флаг ссылки (reference бит) автоматически устанавливается, когда происходит любое обращение к этой странице, а уже рассмотренный выше флаг изменения (modify бит) устанавливается, если производится запись в эту страницу. Операционная система периодически проверяет установку таких флагов, для того чтобы выделить активно используемые страницы, после чего значения этих флагов сбрасываются.

Алгоритм FIFO. Выталкивание первой пришедшей страницы

Простейший алгоритм. Каждой странице присваивается временная метка. Реализуется это просто созданием очереди страниц, в конец которой страницы попадают, когда загружаются в физическую память, а из начала берутся, когда требуется освободить память. Для замещения выбирается старейшая страница. К сожалению, эта стратегия с достаточной вероятностью будет приводить к замещению активно используемых страниц, например страниц кода текстового процессора при редактировании файла. Заметим, что при замещении активных страниц все работает корректно, но page fault происходит немедленно.
Аномалия Билэди (Belady)

На первый взгляд кажется очевидным, что чем больше в памяти страничных кадров, тем реже будут иметь место page faults. Удивительно, но это не всегда так. Как установил Билэди с коллегами, определенные последовательности обращений к страницам в действительности приводят к увеличению числа страничных нарушений при увеличении кадров, выделенных процессу. Это явление носит название "аномалии Билэди" или "аномалии FIFO".

Система с тремя кадрами (9 faults) оказывается более производительной, чем с четырьмя кадрами (10 faults), для строки обращений к памяти 012301401234 при выборе стратегии FIFO.

Рис. 10.1. Аномалия Билэди: (a) - FIFO с тремя страничными кадрами; (b) - FIFO с четырьмя страничными кадрами

Аномалию Билэди следует считать скорее курьезом, чем фактором, требующим серьезного отношения, который иллюстрирует сложность ОС, где интуитивный подход не всегда приемлем.

Оптимальный алгоритм (OPT)

Одним из последствий открытия аномалии Билэди стал поиск оптимального алгоритма, который при заданной строке обращений имел бы минимальную частоту page faults среди всех других алгоритмов. Такой алгоритм был найден. Он прост: замещай страницу, которая не будет использоваться в течение самого длительного периода времени.

Каждая страница должна быть помечена числом инструкций, которые будут выполнены, прежде чем на эту страницу будет сделана первая ссылка. Выталкиваться должна страница, для которой это число наибольшее.

Этот алгоритм легко описать, но реализовать невозможно. ОС не знает, к какой странице будет следующее обращение. (Ранее такие проблемы возникали при планировании процессов - алгоритм SJF).

Зато мы можем сделать вывод, что для того, чтобы алгоритм замещения был максимально близок к идеальному алгоритму, система должна как можно точнее предсказывать обращения процессов к памяти. Данный алгоритм применяется для оценки качества реализуемых алгоритмов.

Выталкивание дольше всего не использовавшейся страницы. Алгоритм LRU

Одним из приближений к алгоритму OPT является алгоритм, исходящий из эвристического правила, что недавнее прошлое - хороший ориентир для прогнозирования ближайшего будущего.

Ключевое отличие между FIFO и оптимальным алгоритмом заключается в том, что один смотрит назад, а другой вперед. Если использовать прошлое для аппроксимации

будущего, имеет смысл замещать страницу, которая не использовалась в течение самого долгого времени. Такой подход называется least recently used алгоритм (LRU). Работа алгоритма проиллюстрирована на рис. рис. 10.2. Сравнивая рис. 10.1 b и 10.2, можно увидеть, что использование LRU алгоритма позволяет сократить количество страничных нарушений.

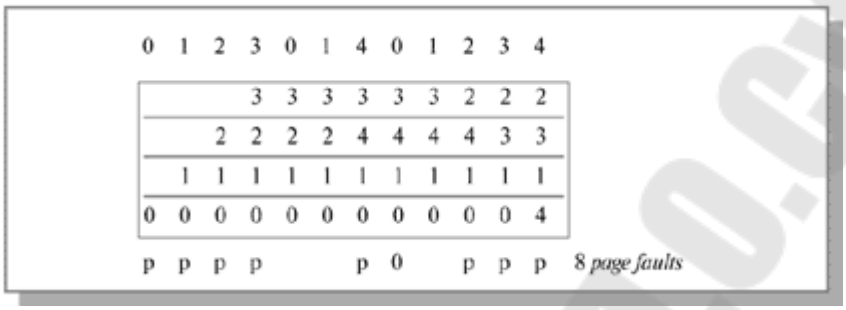

Рис. 10.2. Пример работы алгоритма LRU

LRU - хороший, но труднореализуемый алгоритм. Необходимо иметь связанный список всех страниц в памяти, в начале которого будут хранится недавно использованные страницы. Причем этот список должен обновляться при каждом обращении к памяти. Много времени нужно и на поиск страниц в таком списке.

В [Таненбаум, 2002] рассмотрен вариант реализации алгоритма LRU со специальным 64-битным указателем, который автоматически увеличивается на единицу после выполнения каждой инструкции, а в таблице страниц имеется соответствующее поле, в которое заносится значение указателя при каждой ссылке на страницу. При возникновении page fault выгружается страница с наименьшим значением этого поля.

Как оптимальный алгоритм, так и LRU не страдают от аномалии Билэди. Существует класс алгоритмов, для которых при одной и той же строке обращений множество страниц в памяти для n кадров всегда является подмножеством страниц для n+1 кадра. Эти алгоритмы не проявляют аномалии Билэди и называются стековыми (stack) алгоритмами.

Выталкивание редко используемой страницы. Алгоритм NFU

Поскольку большинство современных процессоров не предоставляют соответствующей аппаратной поддержки для реализации алгоритма LRU, хотелось бы иметь алгоритм, достаточно близкий к LRU, но не требующий специальной поддержки.

Программная реализация алгоритма, близкого к LRU, - алгоритм NFU(Not Frequently Used).

Для него требуются программные счетчики, по одному на каждую страницу, которые сначала равны нулю. При каждом прерывании по времени (а не после каждой инструкции) операционная система сканирует все страницы в памяти и у каждой страницы с установленным флагом обращения увеличивает на единицу значение счетчика, а флаг обращения сбрасывает.

Таким образом, кандидатом на освобождение оказывается страница с наименьшим значением счетчика, как страница, к которой реже всего обращались. Главный недостаток алгоритма NFU состоит в том, что он ничего не забывает. Например, страница, к которой очень часто обращались в течение некоторого времени, а потом обращаться перестали, все

равно не будет удалена из памяти, потому что ее счетчик содержит большую величину. Например, в многопроходных компиляторах страницы, которые активно использовались во время первого прохода, могут надолго сохранить большие значения счетчика, мешая загрузке полезных в дальнейшем страниц.

К счастью, возможна небольшая модификация алгоритма, которая позволяет ему "забывать". Достаточно, чтобы при каждом прерывании по времени содержимое счетчика сдвигалось вправо на 1 бит, а уже затем производилось бы его увеличение для страниц с установленным флагом обращения.

Другим, уже более устойчивым недостатком алгоритма является длительность процесса сканирования таблиц страниц.

Другие алгоритмы

Для полноты картины можно упомянуть еще несколько алгоритмов.

Например, алгоритм Second-Chance - модификация алгоритма FIFO, которая позволяет избежать потери часто используемых страниц с помощью анализа флага обращений (бита ссылки) для самой старой страницы. Если флаг установлен, то страница, в отличие от алгоритма FIFO, не выталкивается, а ее флаг сбрасывается, и страница переносится в конец очереди. Если первоначально флаги обращений были установлены для всех страниц (на все страницы ссылались), алгоритм Second-Chance превращается в алгоритм FIFO. Данный алгоритм использовался в Multics и BSD Unix.

В компьютере Macintosh использован алгоритм NRU (Not Recently-Used), где страница-"жертва" выбирается на основе анализа битов модификации и ссылки. Интересные стратегии, основанные на буферизации страниц, реализованы в VAX/VMS и Mach.

Имеется также и много других алгоритмов замещения. Объем этого курса не позволяет рассмотреть их подробно. Подробное описание различных алгоритмов замещения можно найти в монографиях [Дейтел, 1987], [Цикритис, 1977], [Таненбаум, 2002] и др.

Управление количеством страниц, выделенным процессу. Модель рабочего множества

В стратегиях замещения, рассмотренных в предыдущем разделе, прослеживается предположение о том, что количество кадров, принадлежащих процессу, нельзя увеличить. Это приводит к необходимости выталкивания страницы. Рассмотрим более общий подход, базирующийся на концепции рабочего множества, сформулированной Деннингом [Denning, 1996].

Итак, что делать, если в распоряжении процесса имеется недостаточное число кадров? Нужно ли его приостановить с освобождением всех кадров? Что следует понимать под достаточным количеством кадров?

Трешинг (Thrashing)

Хотя теоретически возможно уменьшить число кадров процесса до минимума, существует какое-то число активно используемых страниц, без которого процесс часто генерирует page faults. Высокая частота страничных нарушений называется трешинг (thrashing, иногда употребляется русский термин "пробуксовка", см. рис. 10.3). Процесс находится в состоянии трешинга, если при его работе больше времени уходит на подкачку страниц,

нежели на выполнение команд. Такого рода критическая ситуация возникает вне зависимости от конкретных алгоритмов замещения.

Рис. 10.3. Частота page faults в зависимости от количества кадров, выделенных процессу

Часто результатом трешинга является снижение производительности вычислительной системы. Один из нежелательных сценариев развития событий может выглядеть следующим образом. При глобальном алгоритме замещения процесс, которому не хватает кадров, начинает отбирать кадры у других процессов, которые в свою очередь начинают заниматься тем же. В результате все процессы попадают в очередь запросов к устройству вторичной памяти (находятся в состоянии ожидания), а очередь процессов в состоянии готовности пустеет. Загрузка процессора снижается. Операционная система реагирует на это увеличением степени мультипрограммирования, что приводит к еще большему трешингу и дальнейшему снижению загрузки процессора. Таким образом, пропускная способность системы падает из-за трешинга.

Эффект трешинга, возникающий при использовании глобальных алгоритмов, может быть ограничен за счет применения локальных алгоритмов замещения. При локальных алгоритмах замещения если даже один из процессов попал в трешинг, это не сказывается на других процессах. Однако он много времени проводит в очереди к устройству выгрузки, затрудняя подкачку страниц остальных процессов.

Критическая ситуация типа трешинга возникает вне зависимости от конкретных алгоритмов замещения. Единственным алгоритмом, теоретически гарантирующим отсутствие трешинга, является рассмотренный выше не реализуемый на практике оптимальный алгоритм.

Итак, трешинг - это высокая частота страничных нарушений. Необходимо ее контролировать. Когда она высока, процесс нуждается в кадрах. Можно, устанавливая желаемую частоту page faults, регулировать размер процесса, добавляя или отнимая у него кадры. Может оказаться целесообразным выгрузить процесс целиком. Освободившиеся кадры выделяются другим процессам с высокой частотой page faults.

Для предотвращения трешинга требуется выделять процессу столько кадров, сколько ему нужно. Но как узнать, сколько ему нужно? Необходимо попытаться выяснить, как много кадров процесс реально использует. Для решения этой задачи Деннинг использовал модель рабочего множества, которая основана на применении принципа локальности.

Модель рабочего множества

Рассмотрим поведение реальных процессов.

Процессы начинают работать, не имея в памяти необходимых страниц. В результате при выполнении первой же машинной инструкции возникает page fault, требующий подкачки порции кода. Следующий page fault происходит при локализации глобальных переменных и еще один - при выделении памяти для стека. После того как процесс собрал большую часть необходимых ему страниц, page faults возникают редко.

Таким образом, существует набор страниц (P1, P2, ... Pn), активно использующихся вместе, который позволяет процессу в момент времени t в течение некоторого периода T производительно работать, избегая большого количества page faults. Этот набор страниц называется рабочим множеством W(t,T) (working set) процесса. Число страниц в рабочем множестве определяется параметром T, является неубывающей функцией T и относительно невелико. Иногда T называют размером окна рабочего множества, через которое ведется наблюдение за процессом (см. рис. 10.4).

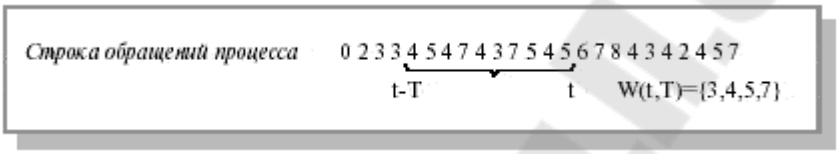

Рис. 10.4. Пример рабочего множества процесса

Легко написать тестовую программу, которая систематически работает с большим диапазоном адресов, но, к счастью, большинство реальных процессов не ведут себя подобным образом, а проявляют свойство локальности. В течение любой фазы вычислений процесс работает с небольшим количеством страниц.

Когда процесс выполняется, он двигается от одного рабочего множества к другому. Программа обычно состоит из нескольких рабочих множеств, которые могут перекрываться. Например, когда вызвана процедура, она определяет новое рабочее множество, состоящее из страниц, содержащих инструкции процедуры, ее локальные и глобальные переменные. После ее завершения процесс покидает это рабочее множество, но может вернуться к нему при новом вызове процедуры. Таким образом, рабочее множество определяется кодом и данными программы. Если процессу выделять меньше кадров, чем ему требуется для поддержки рабочего множества, он будет находиться в состоянии трешинга.

Принцип локальности ссылок препятствует частым изменениям рабочих наборов процессов. Формально это можно выразить следующим образом. Если в период времени (t-T, t) программа обращалась к страницам W(t,T), то при надлежащем выборе T с большой вероятностью эта программа будет обращаться к тем же страницам в период времени (t, t+T). Другими словами, принцип локальности утверждает, что если не слишком далеко заглядывать в будущее, то можно достаточно точно его прогнозировать исходя из прошлого. Понятно, что с течением времени рабочий набор процесса может изменяться (как по составу страниц, так и по их числу).

Наиболее важное свойство рабочего множества - его размер. ОС должна выделить каждому процессу достаточное число кадров, чтобы поместилось его рабочее множество. Если кадры еще остались, то может быть инициирован другой процесс. Если рабочие множества процессов не помещаются в память и начинается трешинг, то один из процессов можно выгрузить на диск.

Решение о размещении процессов в памяти должно, следовательно, базироваться на размере его рабочего множества. Для впервые инициируемых процессов это решение

может быть принято эвристически. Во время работы процесса система должна уметь определять: расширяет процесс свое рабочее множество или перемещается на новое рабочее множество. Если в состав атрибутов страницы включить время последнего использования ti (для страницы с номером i), то принадлежность i-й страницы к рабочему набору, определяемому параметром T в момент времени t будет выражаться неравенством: t-T < ti < t. Алгоритм выталкивания страниц WSClock, использующий информацию о рабочем наборе процесса, описан в [Таненбаум, 2002].

Другой способ реализации данного подхода может быть основан на отслеживании количества страничных нарушений, вызываемых процессом. Если процесс часто генерирует page faults и память не слишком заполнена, то система может увеличить число выделенных ему кадров. Если же процесс не вызывает исключительных ситуаций в течение некоторого времени и уровень генерации ниже какого-то порога, то число кадров процесса может быть урезано. Этот способ регулирует лишь размер множества страниц, принадлежащих процессу, и должен быть дополнен какой-либо стратегией замещения страниц. Несмотря на то что система при этом может пробуксовывать в моменты перехода от одного рабочего множества к другому, предлагаемое решение в состоянии обеспечить наилучшую производительность для каждого процесса, не требуя никакой дополнительной настройки системы.

Страничные демоны

Подсистема виртуальной памяти работает производительно при наличии резерва свободных страничных кадров. Алгоритмы, обеспечивающие поддержку системы в состоянии отсутствия трешинга, реализованы в составе фоновых процессов (их часто называют демонами или сервисами), которые периодически "просыпаются" и инспектируют состояние памяти. Если свободных кадров оказывается мало, они могут сменить стратегию замещения. Их задача - поддерживать систему в состоянии наилучшей производительности.

Примером такого рода процесса может быть фоновый процесс - сборщик страниц, реализующий облегченный вариант алгоритма откачки, основанный на использовании рабочего набора и применяемый во многих клонах ОС Unix (см., например,[Bach, 1986]). Данный демон производит откачку страниц, не входящих в рабочие наборы процессов. Он начинает активно работать, когда количество страниц в списке свободных страниц достигает установленного нижнего порога, и пытается выталкивать страницы в соответствии с собственной стратегией.

Но если возникает требование страницы в условиях, когда список свободных страниц пуст, то начинает работать механизм свопинга, поскольку простое отнятие страницы у любого процесса (включая тот, который затребовал бы страницу) потенциально вело бы к ситуации thrashing, и разрушало бы рабочий набор некоторого процесса. Любой процесс, затребовавший страницу не из своего текущего рабочего набора, становится в очередь на выгрузку в расчете на то, что после завершения выгрузки хотя бы одного из процессов свободной памяти уже может быть достаточно.

В ОС Windows 2000 аналогичную роль играет менеджер балансного набора (Working set manager), который вызывается раз в секунду или тогда, когда размер свободной памяти опускается ниже определенного предела, и отвечает за суммарную политику управления памятью и поддержку рабочих множеств.

Программная поддержка сегментной модели памяти процесса

Реализация функций операционной системы, связанных с поддержкой памяти, - ведение таблиц страниц, трансляция адреса, обработка страничных ошибок, управление ассоциативной памятью и др. - тесно связана со структурами данных, обеспечивающими удобное представление адресного пространства процесса. Формат этих структур сильно зависит от аппаратуры и особенностей конкретной ОС.

Чаще всего виртуальная память процесса ОС разбивается на сегменты пяти типов: кода программы, данных, стека, разделяемый и сегмент файлов, отображаемых в память (см. рис. 10.5).

Сегмент программного кода содержит только команды. Сегмент программного кода не модифицируется в ходе выполнения процесса, обычно страницы данного сегмента имеют атрибут read-only. Следствием этого является возможность использования одного экземпляра кода для разных процессов.

Сегмент данных, содержащий переменные программы и сегмент стека, содержащий автоматические переменные, могут динамически менять свой размер (обычно данные в сторону увеличения адресов, а стек - в сторону уменьшения) и содержимое, должны быть доступны по чтению и записи и являются приватными сегментами процесса.

С целью обобществления памяти между несколькими процессами создаются разделяемые сегменты, допускающие доступ по чтению и записи. Вариантом разделяемого сегмента может быть сегмент файла, отображаемого в память. Специфика таких сегментов состоит в том, что из них откачка осуществляется не в системную область выгрузки, а непосредственно в отображаемый файл. Реализация разделяемых сегментов основана на том, что логические страницы различных процессов связываются с одними и теми же страничными кадрами.

Сегменты представляют собой непрерывные области (в Linux они так и называются - области) в виртуальном адресном пространстве процесса, выровненные по границам страниц. Каждая область состоит из набора страниц с одним и тем же режимом защиты. Между областями в виртуальном пространстве могут быть свободные участки. Естественно, что подобные объекты описаны соответствующими структурами (см., например, структуры mm_struct и vm_area_struct в Linux).

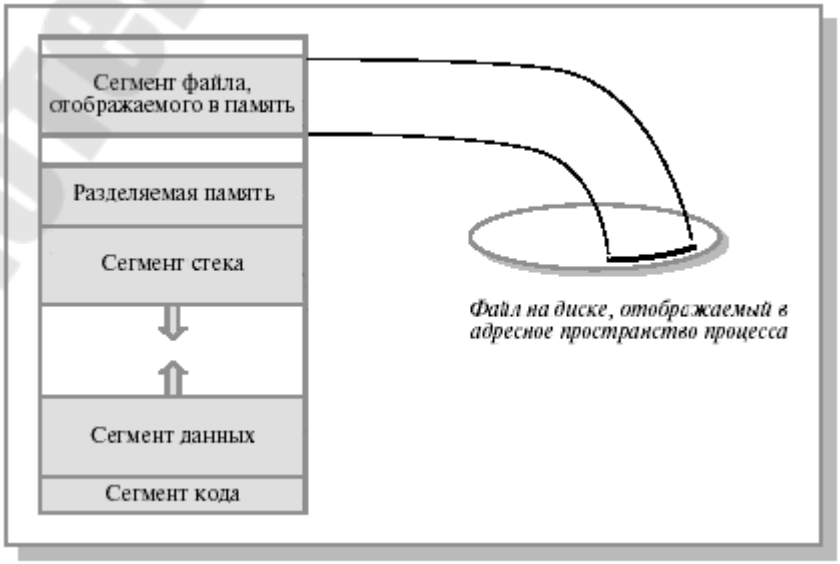

Рис. 10.5. Образ процесса в памяти

Часть работы по организации сегментов может происходить с участием программиста. Особенно это заметно при низкоуровневом программировании. В частности, отдельные области памяти могут быть поименованы и использоваться для обмена данными между процессами. Два процесса могут общаться через разделяемую область памяти при условии, что им известно ее имя (пароль). Обычно это делается при помощи специальных вызовов (например, map и unmap), входящих в состав интерфейса виртуальной памяти.

Загрузка исполняемого файла (системный вызов exec) осуществляется обычно через отображение (mapping) его частей (кода, данных) в соответствующие сегменты адресного пространства процесса. Например, сегмент кода является сегментом отображаемого в память файла, содержащего исполняемую программу. При попытке выполнить первую же инструкцию система обнаруживает, что нужной части кода в памяти нет, генерирует page fault и подкачивает эту часть кода с диска. Далее процедура повторяется до тех пор, пока вся программа не окажется в оперативной памяти.

Как уже говорилось, размер сегмента данных динамически меняется. Рассмотрим, как организована поддержка сегментов данных в Unix. Пользователь, запрашивая (библиотечные вызовы malloc, new) или освобождая (free, delete) память для динамических данных, фактически изменяет границу выделенной процессу памяти через системный вызов brk (от слова break), который модифицирует значение переменной brk из структуры данных процесса. В результате происходит выделение физической памяти, граница brk смещается в сторону увеличения виртуальных адресов, а соответствующие строки таблиц страниц получают осмысленные значения. При помощи того же вызова brk пользователь может уменьшить размер сегмента данных. На практике освобожденная пользователем виртуальная память (библиотечные вызовы free, delete) системе не возвращается. На это есть две причины. Во-первых, для уменьшения размеров сегмента данных необходимо организовать его уплотнение или "сборку мусора". А во-вторых, незанятые внутри сегмента данных области естественным образом будут вытолкнуты из оперативной памяти вследствие того, что к ним не будет обращений. Ведение списков занятых и свободных областей памяти в сегменте данных пользователя осуществляется на уровне системных библиотек.

Более подробно информация об адресных пространствах процессов в Unix изложена в [Кузнецов], [Bach, 1986].

Отдельные аспекты функционирования менеджера памяти

Корректная работа менеджера памяти помимо принципиальных вопросов, связанных с выбором абстрактной модели виртуальной памяти и ее аппаратной поддержкой, обеспечивается также множеством нюансов и мелких деталей. В качестве примера такого рода компонента рассмотрим более подробно локализацию страниц в памяти, которая применяется в тех случаях, когда поддержка страничной системы приводит к необходимости разрешить определенным страницам, хранящим буферы ввода-вывода, другие важные данные и код, быть блокированными в памяти.

Рассмотрим случай, когда система виртуальной памяти может вступить в конфликт с подсистемой ввода-вывода. Например, процесс может запросить ввод в буфер и ожидать его завершения. Управление передастся другому процессу, который может вызвать page fault и, с отличной от нуля вероятностью, спровоцировать выгрузку той страницы, куда должен быть осуществлен ввод первым процессом. Подобные ситуации нуждаются в

дополнительном контроле, особенно если ввод-вывод реализован с использованием механизма прямого доступа к памяти (DMA). Одно из решений данной проблемы - вводить данные в не вытесняемый буфер в пространстве ядра, а затем копировать их в пользовательское пространство.

Второе решение - локализовать страницы в памяти, используя специальный бит локализации, входящий в состав атрибутов страницы. Локализованная страница замещению не подлежит. Бит локализации сбрасывается после завершения операции ввода-вывода.

Другое использование бита локализации может иметь место и при нормальном замещении страниц. Рассмотрим следующую цепь событий. Низкоприоритетный процесс после длительного ожидания получил в свое распоряжение процессор и подкачал с диска нужную ему страницу. Если он сразу после этого будет вытеснен высокоприоритетным процессом, последний может легко заместить вновь подкачанную страницу низкоприоритетного, так как на нее не было ссылок. Имеет смысл вновь загруженные страницы помечать битом локализации до первой ссылки, иначе низкоприоритетный процесс так и не начнет работать.

Использование бита локализации может быть опасным, если забыть его отключить. Если такая ситуация имеет место, страница становится неиспользуемой. SunOS разрешает использование данного бита в качестве подсказки, которую можно игнорировать, когда пул свободных кадров становится слишком маленьким.

Другим важным применением локализации является ее использование в системах мягкого реального времени. Рассмотрим процесс или нить реального времени. Вообще говоря, виртуальная память - антитеза вычислений реального времени, так как дает непредсказуемые задержки при подкачке страниц. Поэтому системы реального времени почти не используют виртуальную память. ОС Solaris поддерживает как реальное время, так и разделение времени. Для решения проблемы page faults, Solaris разрешает процессам сообщать системе, какие страницы важны для процесса, и локализовать их в памяти. В результате возможно выполнение процесса, реализующего задачу реального времени, содержащего локализованные страницы, где временные задержки страничной системы будут минимизированы.

Помимо системы локализации страниц, есть и другие интересные проблемы, возникающие в процессе управления памятью. Так, например, бывает непросто осуществить повторное выполнение инструкции, вызвавшей page fault. Представляют интерес и алгоритмы отложенного выделения памяти (копирование при записи и др.). Ограниченный объем данного курса не позволяет рассмотреть их более подробно.

Лекция . Файловая система

Файловая система - это часть операционной системы, назначение которой состоит в том, чтобы организовать эффективную работу с данными, хранящимися во внешней памяти, и обеспечить пользователю удобный интерфейс при работе с такими данными. Организовать хранение информации на магнитном диске непросто. Это требует, например, хорошего знания устройства контроллера диска, особенностей работы с его регистрами. Непосредственное взаимодействие с диском - прерогатива компонента системы ввода-вывода ОС, называемого драйвером диска. Для того чтобы избавить пользователя компьютера от сложностей взаимодействия с аппаратурой, была придумана ясная абстрактная модель файловой системы. Операции записи или чтения файла концептуально проще, чем низкоуровневые операции работы с устройствами.

Основная идея использования внешней памяти состоит в следующем. ОС делит память на блоки фиксированного размера, например, 4096 байт. Файл, обычно представляющий собой неструктурированную последовательность однобайтовых записей, хранится в виде последовательности блоков (не обязательно смежных); каждый блок хранит целое число записей. В некоторых ОС (MS-DOS) адреса блоков, содержащих данные файла, могут быть организованы в связный список и вынесены в отдельную таблицу в памяти. В других ОС (Unix) адреса блоков данных файла хранятся в отдельном блоке внешней памяти (так называемом индексе или индексном узле). Этот прием, называемый индексацией, является наиболее распространенным для приложений, требующих произвольного доступа к записям файлов. Индекс файла состоит из списка элементов, каждый из которых содержит номер блока в файле и сведения о местоположении данного блока. Считывание очередного байта осуществляется с так называемой текущей позиции, которая характеризуется смещением от начала файла. Зная размер блока, легко вычислить номер блока, содержащего текущую позицию. Адрес же нужного блока диска можно затем извлечь из индекса файла. Базовой операцией, выполняемой по отношению к файлу, является чтение блока с диска и перенос его в буфер, находящийся в основной памяти.

Файловая система позволяет при помощи системы справочников (каталогов, директорий) связать уникальное имя файла с блоками вторичной памяти, содержащими данные файла. Иерархическая структура каталогов, используемая для управления файлами, может служить другим примером индексной структуры. В этом случае каталоги или папки играют роль индексов, каждый из которых содержит ссылки на свои подкаталоги. С этой точки зрения вся файловая система компьютера представляет собой большой индексированный файл. Помимо собственно файлов и структур данных, используемых для управления файлами (каталоги, дескрипторы файлов, различные таблицы распределения внешней памяти), понятие "файловая система" включает программные средства, реализующие различные операции над файлами.

Основные функции файловой систем:

- идентификация файлов
- связывание имени файла с выделенным ему пространством внешней памяти
- распределение внешней памяти между файлами. Для работы с конкретным файлом пользователю не требуется иметь информацию о местоположении этого файла на внешнем носителе информации. Например, для того чтобы загрузить документ в редактор с жесткого диска, нам не нужно знать, на какой стороне какого магнитного диска, на каком цилиндре и в каком секторе находится данный документ.
- обеспечение надежности и отказоустойчивости. Стоимость информации может во много раз превышать стоимость компьютера.

- обеспечение защиты от несанкционированного доступа.
- обеспечение совместного доступа к файлам, так чтобы пользователю не приходилось прилагать специальных усилий по обеспечению синхронизации доступа.
- обеспечение высокой производительности.

Иногда говорят, что файл - это поименованный набор связанной информации, записанной во вторичную память. Для большинства пользователей файловая система - наиболее видимая часть ОС. Она предоставляет механизм для онлайнового хранения и доступа как к данным, так и к программам для всех пользователей системы. С точки зрения пользователя, файл - единица внешней памяти, то есть данные, записанные на диск, должны быть в составе какого-нибудь файла.

Важный аспект организации файловой системы - учет стоимости операций взаимодействия с вторичной памятью. Процесс считывания блока диска состоит из позиционирования считывающей головки над дорожкой, содержащей требуемый блок, ожидания, пока требуемый блок сделает оборот и окажется под головкой, и собственно считывания блока. Для этого требуется значительное время (десятки миллисекунд). В современных компьютерах обращение к диску осуществляется примерно в 100 000 раз медленнее, чем обращение к оперативной памяти. Таким образом, критерием вычислительной сложности алгоритмов, работающих с внешней памятью, является количество обращений к диску.

Общие сведения о файлах

Имена файлов

Файлы представляют собой абстрактные объекты. Их задача - хранить информацию, скрывая от пользователя детали работы с устройствами. Когда процесс создает файл, он дает ему имя. После завершения процесса файл продолжает существовать и через свое имя может быть доступен другим процессам.

Правила именования файлов зависят от ОС. Многие ОС поддерживают имена из двух частей (имя+расширение), например progr.c (файл, содержащий текст программы на языке Си) или autoexec.bat (файл, содержащий команды интерпретатора командного языка). Тип расширения файла позволяет ОС организовать работу с ним различных прикладных программ в соответствии с заранее оговоренными соглашениями. Обычно ОС накладывают некоторые ограничения, как на используемые в имени символы, так и на длину имени файла. В соответствии со стандартом POSIX, популярные ОС оперируют удобными для пользователя длинными именами (до 255 символов).

Типы файлов

Важный аспект организации файловой системы и ОС - следует ли поддерживать и распознавать типы файлов. Если да, то это может помочь правильному функционированию ОС, например не допустить вывода на принтер бинарного файла.

Основные типы файлов: регулярные (обычные) файлы и директории (справочники, каталоги). Обычные файлы содержат пользовательскую информацию. Директории - системные файлы, поддерживающие структуру файловой системы. В каталоге содержится перечень входящих в него файлов и устанавливается соответствие между файлами и их характеристиками (атрибутами). Мы будем рассматривать директории ниже.

Напомним, что хотя внутри подсистемы управления файлами обычный файл представляется в виде набора блоков внешней памяти, для пользователей обеспечивается представление файла в виде линейной последовательности байтов. Такое представление позволяет использовать абстракцию файла при работе с внешними устройствами, при организации межпроцессных взаимодействий и т. д. Так, например, клавиатура обычно рассматривается как текстовый файл, из которого компьютер получает данные в символьном формате. Поэтому иногда к файлам приписывают другие объекты ОС, например специальные символьные файлы и специальные блочные файлы, именованные каналы и сокеты, имеющие файловый интерфейс. Эти объекты рассматриваются в других разделах данного курса.

Обычные (или регулярные) файлы реально представляют собой набор блоков (возможно, пустой) на устройстве внешней памяти, на котором поддерживается файловая система. Такие файлы могут содержать как текстовую информацию (обычно в формате ASCII), так и произвольную двоичную (бинарную) информацию.

Текстовые файлы содержат символьные строки, которые можно распечатать, увидеть на экране или редактировать обычным текстовым редактором.

Другой тип файлов - нетекстовые, или бинарные, файлы. Обычно они имеют некоторую внутреннюю структуру. Например, исполняемый файл в ОС Unix имеет пять секций: заголовок, текст, данные, биты реаллокации и символьную таблицу. ОС выполняет файл, только если он имеет нужный формат. Другим примером бинарного файла может быть архивный файл. Типизация файлов не слишком строгая.

Обычно прикладные программы, работающие с файлами, распознают тип файла по его имени в соответствии с общепринятыми соглашениями. Например, файлы с расширениями .c, .pas, .txt - ASCII-файлы, файлы с расширениями .exe - выполнимые, файлы с расширениями .obj, .zip - бинарные и т. д.

Атрибуты файлов

Кроме имени ОС часто связывают с каждым файлом и другую информацию, например дату модификации, размер и т. д. Эти другие характеристики файлов называются атрибутами. Список атрибутов в разных ОС может варьироваться. Обычно он содержит следующие элементы: основную информацию (имя, тип файла), адресную информацию (устройство, начальный адрес, размер), информацию об управлении доступом (владелец, допустимые операции) и информацию об использовании (даты создания, последнего чтения, модификации и др.).

Список атрибутов обычно хранится в структуре директорий (см. следующую лекцию) или других структурах, обеспечивающих доступ к данным файла.

Организация файлов и доступ к ним

Программист воспринимает файл в виде набора однородных записей. Запись - это наименьший элемент данных, который может быть обработан как единое целое прикладной программой при обмене с внешним устройством. Причем в большинстве ОС размер записи равен одному байту. В то время как приложения оперируют записями, физический обмен с устройством осуществляется большими единицами (обычно блоками). Поэтому записи объединяются в блоки для вывода и разблокируются - для

ввода. Вопросы распределения блоков внешней памяти между файлами рассматриваются в следующей лекции.

ОС поддерживают несколько вариантов структуризации файлов.

Последовательный файл

Простейший вариант - так называемый последовательный файл. То есть файл является последовательностью записей. Поскольку записи, как правило, однобайтовые, файл представляет собой неструктурированную последовательность байтов.

Обработка подобных файлов предполагает последовательное чтение записей от начала файла, причем конкретная запись определяется ее положением в файле. Такой способ доступа называется последовательным (модель ленты). Если в качестве носителя файла используется магнитная лента, то так и делается. Текущая позиция считывания может быть возвращена к началу файла (rewind).

Файл прямого доступа

В реальной практике файлы хранятся на устройствах прямого (random) доступа, например на дисках, поэтому содержимое файла может быть разбросано по разным блокам диска, которые можно считывать в произвольном порядке. Причем номер блока однозначно определяется позицией внутри файла.

Здесь имеется в виду относительный номер, специфицирующий данный блок среди блоков диска, принадлежащих файлу. О связи относительного номера блока с абсолютным его номером на диске рассказывается в следующей лекции.

Естественно, что в этом случае для доступа к середине файла просмотр всего файла с самого начала не обязателен. Для специфицирования места, с которого надо начинать чтение, используются два способа: с начала или с текущей позиции, которую дает операция seek. Файл, байты которого могут быть считаны в произвольном порядке, называется файлом прямого доступа.

Таким образом, файл, состоящий из однобайтовых записей на устройстве прямого доступа, - наиболее распространенный способ организации файла. Базовыми операциями для такого рода файлов являются считывание или запись символа в текущую позицию. В большинстве языков высокого уровня предусмотрены операторы посимвольной пересылки данных в файл или из него.

Подобную логическую структуру имеют файлы во многих файловых системах, например в файловых системах ОС Unix и MS-DOS. ОС не осуществляет никакой интерпретации содержимого файла. Эта схема обеспечивает максимальную гибкость и универсальность. С помощью базовых системных вызовов (или функций библиотеки ввода/вывода) пользователи могут как угодно структурировать файлы. В частности, многие СУБД хранят свои базы данных в обычных файлах.

Другие формы организации файлов

Известны как другие формы организации файла, так и другие способы доступа к ним, которые использовались в ранних ОС, а также применяются сегодня в больших мэйнфреймах (mainframe), ориентированных на коммерческую обработку данных.

Первый шаг в структурировании - хранение файла в виде последовательности записей фиксированной длины, каждая из которых имеет внутреннюю структуру. Операция чтения производится над записью, а операция записи переписывает или добавляет запись целиком. Ранее использовались записи по 80 байт (это соответствовало числу позиций в перфокарте) или по 132 символа (ширина принтера). В ОС CP/M файлы были последовательностями 128-символьных записей. С введением CRT-терминалов данная идея утратила популярность.

Другой способ представления файлов - последовательность записей переменной длины, каждая из которых содержит ключевое поле в фиксированной позиции внутри записи (см. рис. 11.1). Базисная операция в данном случае - считать запись с каким-либо значением ключа. Записи могут располагаться в файле последовательно (например, отсортированные по значению ключевого поля) или в более сложном порядке. Метод доступа по значению ключевого поля к записям последовательного файла называется индексно-последовательным.

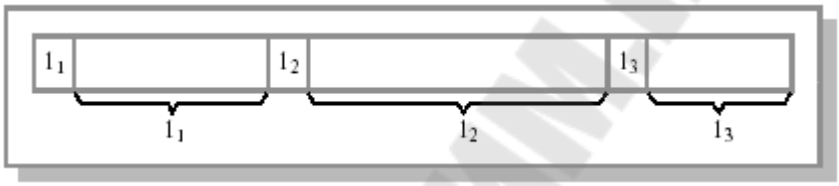

Рис. 11.1. Файл как последовательность записей переменной длины

В некоторых системах ускорение доступа к файлу обеспечивается конструированием индекса файла. Индекс обычно хранится на том же устройстве, что и сам файл, и состоит из списка элементов, каждый из которых содержит идентификатор записи, за которым следует указание о местоположении данной записи. Для поиска записи вначале происходит обращение к индексу, где находится указатель на нужную запись. Такие файлы называются индексированными, а метод доступа к ним - доступ с использованием индекса.

Предположим, у нас имеется большой несортированный файл, содержащий разнообразные сведения о студентах, состоящие из записей с несколькими полями, и возникает задача организации быстрого поиска по одному из полей, например по фамилии студента. Рис. 11.2 иллюстрирует решение данной проблемы - организацию метода доступа к файлу с использованием индекса.

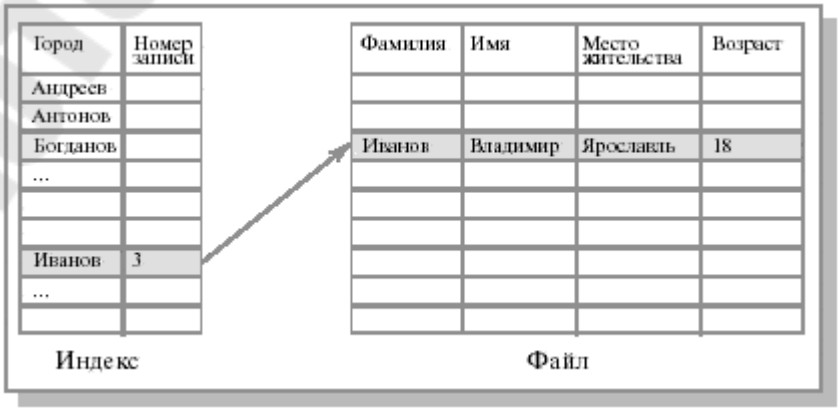

Рис. 11.2. Пример организации индекса для последовательного файла

Следует отметить, что почти всегда главным фактором увеличения скорости доступа является избыточность данных.

Способ выделения дискового пространства при помощи индексных узлов, применяемый в ряде ОС (Unix и некоторых других, см. следующую лекцию), может служить другим примером организации индекса.

В этом случае ОС использует древовидную организацию блоков, при которой блоки, составляющие файл, являются листьями дерева, а каждый внутренний узел содержит указатели на множество блоков файла. Для больших файлов индекс может быть слишком велик. В этом случае создают индекс для индексного файла (блоки промежуточного уровня или блоки косвенной адресации).

Операции над файлами

Операционная система должна предоставить в распоряжение пользователя набор операций для работы с файлами, реализованных через системные вызовы. Чаще всего при работе с файлом пользователь выполняет не одну, а несколько операций. Во-первых, нужно найти данные файла и его атрибуты по символьному имени, во-вторых, считать необходимые атрибуты файла в отведенную область оперативной памяти и проанализировать права пользователя на выполнение требуемой операции. Затем следует выполнить операцию, после чего освободить занимаемую данными файла область памяти.

Рассмотрим в качестве примера основные файловые операции ОС Unix [Таненбаум, 2002]:

- Создание файла, не содержащего данных. Смысл данного вызова - объявить, что файл существует, и присвоить ему ряд атрибутов. При этом выделяется место для файла на диске и вносится запись в каталог.
- Удаление файла и освобождение занимаемого им дискового пространства.
- Открытие файла. Перед использованием файла процесс должен его открыть. Цель данного системного вызова - разрешить системе проанализировать атрибуты файла и проверить права доступа к нему, а также считать в оперативную память список адресов блоков файла для быстрого доступа к его данным. Открытие файла является процедурой создания дескриптора или управляющего блока файла. Дескриптор (описатель) файла хранит всю информацию о нем. Иногда, в соответствии с парадигмой, принятой в языках программирования, под дескриптором понимается альтернативное имя файла или указатель на описание файла в таблице открытых файлов, используемый при последующей работе с файлом . Например, на языке Си операция открытия файла fd=open(pathname,flags,modes); возвращает дескриптор fd, который может быть задействован при выполнении операций чтения (read(fd,buffer,count);) или записи.
- Закрытие файла. Если работа с файлом завершена, его атрибуты и адреса блоков на диске больше не нужны. В этом случае файл нужно закрыть, чтобы освободить место во внутренних таблицах файловой системы.
- Позиционирование. Дает возможность специфицировать место внутри файла, откуда будет производиться считывание (или запись) данных, то есть задать текущую позицию.
- Чтение данных из файла. Обычно это делается с текущей позиции. Пользователь должен задать объем считываемых данных и предоставить для них буфер в оперативной памяти.

- Запись данных в файл с текущей позиции. Если текущая позиция находится в конце файла, его размер увеличивается, в противном случае запись осуществляется на место имеющихся данных, которые, таким образом, теряются.

Есть и другие операции, например переименование файла, получение атрибутов файла и т. д.

Существует два способа выполнить последовательность действий над файлами [Олифер, 2001].

В первом случае для каждой операции выполняются как универсальные, так и уникальные действия (схема stateless). Например, последовательность операций может быть такой: open, read1, close, ... open, read2, close, ... open, read3, close.

Альтернативный способ - это когда универсальные действия выполняются в начале и в конце последовательности операций, а для каждой промежуточной операции выполняются только уникальные действия. В этом случае последовательность вышеприведенных операций будет выглядеть так: open, read1, ... read2, ... read3, close.

Большинство ОС использует второй способ, более экономичный и быстрый. Первый способ более устойчив к сбоям, поскольку результаты каждой операции становятся независимыми от результатов предыдущей операции; поэтому он иногда применяется в распределенных файловых системах (например, Sun NFS).

Директории. Логическая структура файлового архива

Количество файлов на компьютере может быть большим. Отдельные системы хранят тысячи файлов, занимающие сотни гигабайтов дискового пространства. Эффективное управление этими данными подразумевает наличие в них четкой логической структуры. Все современные файловые системы поддерживают многоуровневое именование файлов за счет наличия во внешней памяти дополнительных файлов со специальной структурой - каталогов (или директорий).

Каждый каталог содержит список каталогов и/или файлов, содержащихся в данном каталоге. Каталоги имеют один и тот же внутренний формат, где каждому файлу соответствует одна запись в файле директории (см., например, рис.11.3).

Число директорий зависит от системы. В ранних ОС имелась только одна корневая директория, затем появились директории для пользователей (по одной директории на пользователя). В современных ОС используется произвольная структура дерева директорий.

Имя файла (каталога)	Тип файла (обычный или каталог)	
Anti	K	атрибуты
Games	K	атрибуты
Autoexec.bat	O	атрибуты
mouse.com	O	атрибуты

Рис. 11.3. Директории

Таким образом, файлы на диске образуют иерархическую древовидную структуру (см. рис. 11.4).

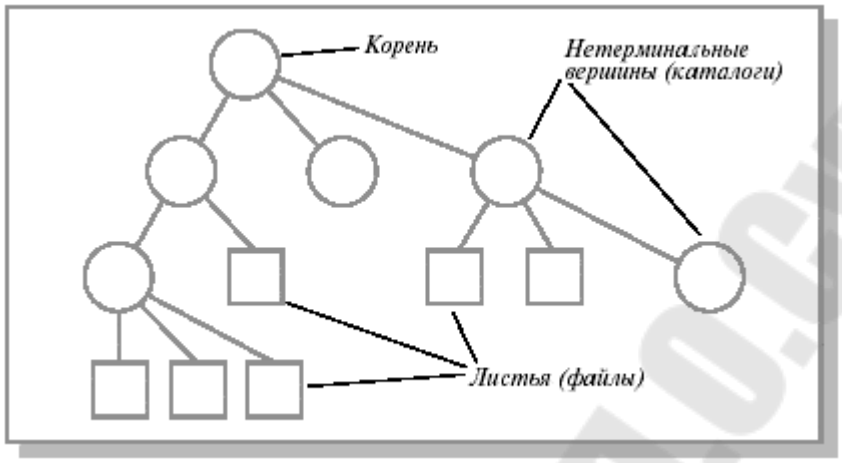

Рис. 11.4. Древовидная структура файловой системы

Существует несколько эквивалентных способов изображения дерева. Структура перевернутого дерева, приведенного на рис. 11.4, наиболее распространена. Верхнюю вершину называют корнем. Если элемент дерева не может иметь потомков, он называется терминальной вершиной или листом (в данном случае является файлом). Нелистовые вершины - справочники или каталоги содержат списки листовых и нелистовых вершин. Путь от корня к файлу однозначно определяет файл.

Подобные древовидные структуры являются графами, не имеющими циклов. Можно считать, что ребра графа направлены вниз, а корень - вершина, не имеющая входящих ребер. Как мы увидим в следующей лекции, связывание файлов, которое практикуется в ряде операционных систем, приводит к образованию циклов в графе.

Внутри одного каталога имена листовых файлов уникальны. Имена файлов, находящихся в разных каталогах, могут совпадать. Для того чтобы однозначно определить файл по его имени (избежать коллизии имен), принято именовать файл так называемым абсолютным или полным именем (pathname), состоящим из списка имен вложенных каталогов, по которому можно найти путь от корня к файлу плюс имя файла в каталоге, непосредственно содержащем данный файл. То есть полное имя включает цепочку имен - путь к файлу, например /usr/games/doom. Такие имена уникальны. Компоненты пути разделяют различными символами: "/" (слэш) в Unix или обратными слэшем в MS-DOS (в Multics - ">"). Таким образом, использование древовидных каталогов минимизирует сложность назначения уникальных имен.

Указывать полное имя не всегда удобно, поэтому применяют другой способ задания имени - относительный путь к файлу. Он использует концепцию рабочей или текущей директории, которая обычно входит в состав атрибутов процесса, работающего с данным файлом. Тогда на файлы в такой директории можно ссылаться только по имени, при этом поиск файла будет осуществляться в рабочем каталоге. Это удобнее, но, по существу, то же самое, что и абсолютная форма.

Для получения доступа к файлу и локализации его блоков система должна выполнить навигацию по каталогам. Рассмотрим для примера путь /usr/linux/progr.c. Алгоритм одинаков для всех иерархических систем. Сначала в фиксированном месте на диске находится корневая директория. Затем находится компонент пути usr, т. е. в корневой директории ищется файл /usr. Исследуя этот файл, система понимает, что данный файл

является каталогом, и блоки его данных рассматривает как список файлов и ищет следующий компонент linux в нем. Из строки для linux находится файл, соответствующий компоненту usr/linux/. Затем находится компонент progr.c, который открывается, заносится в таблицу открытых файлов и сохраняется в ней до закрытия файла.

Отклонение от типовой обработки компонентов pathname может возникнуть в том случае, когда этот компонент является не обычным каталогом с соответствующим ему индексным узлом и списком файлов, а служит точкой связывания (принято говорить "точкой монтирования") двух файловых архивов. Этот случай рассмотрен в следующей лекции.

Многие прикладные программы работают с файлами, находящимися в текущей директории, не указывая явным образом ее имени. Это дает пользователю возможность произвольным образом именовать каталоги, содержащие различные программные пакеты. Для реализации этой возможности в большинстве ОС, поддерживающих иерархическую структуру директорий, используется обозначение "." - для текущей директории и ".." - для родительской.

Разделы диска. Организация доступа к архиву файлов.

Задание пути к файлу в файловых системах некоторых ОС отличается тем, с чего начинается эта цепочка имен.

В современных ОС принято разбивать диски на логические диски (это низкоуровневая операция), иногда называемые разделами (partitions). Бывает, что, наоборот, объединяют несколько физических дисков в один логический диск (например, это можно сделать в ОС Windows NT). Поэтому в дальнейшем изложении мы будем игнорировать проблему физического выделения пространства для файлов и считать, что каждый раздел представляет собой отдельный (виртуальный) диск. Диск содержит иерархическую древовидную структуру, состоящую из набора файлов, каждый из которых является хранилищем данных пользователя, и каталогов или директорий (то есть файлов, которые содержат перечень других файлов, входящих в состав каталога), необходимых для хранения информации о файлах системы.

В некоторых системах управления файлами требуется, чтобы каждый архив файлов целиком располагался на одном диске (разделе диска). В этом случае полное имя файла начинается с имени дискового устройства, на котором установлен соответствующий диск (буквы диска). Например, c:\util\nu\ndd.exe. Такой способ именования используется в файловых системах DEC и Microsoft.

В других системах (Multics) вся совокупность файлов и каталогов представляет собой единое дерево. Сама система, выполняя поиск файлов по имени, начиная с корня, требовала установки необходимых дисков.

В ОС Unix предполагается наличие нескольких архивов файлов, каждый на своем разделе, один из которых считается корневым. После запуска системы можно "смонтировать" корневую файловую систему и ряд изолированных файловых систем в одну общую файловую систему.

Технически это осуществляется с помощью создания в корневой файловой системе специальных пустых каталогов (см. также следующую лекцию). Специальный системный вызов mount ОС Unix позволяет подключить к одному из этих пустых каталогов корневой каталог указанного архива файлов. После монтирования общей файловой системы

именование файлов производится так же, как если бы она с самого начала была централизованной. Задачей ОС является беспрепятственный проход точки монтирования при получении доступа к файлу по цепочке имен. Если учесть, что обычно монтирование файловой системы производится при загрузке системы, пользователи ОС Unix обычно и не задумываются о происхождении общей файловой системы.

Операции над директориями

Как и в случае с файлами, система обязана обеспечить пользователя набором операций, необходимых для работы с директориями, реализованных через системные вызовы. Несмотря на то что директории - это файлы, логика работы с ними отличается от логики работы с обычными файлами и определяется природой этих объектов, предназначенных для поддержки структуры файлового архива. Совокупность системных вызовов для управления директориями зависит от особенностей конкретной ОС. Напомним, что операции над каталогами являются прерогативой ОС, то есть пользователь не может, например, выполнить запись в каталог начиная с текущей позиции. Рассмотрим в качестве примера некоторые системные вызовы, необходимые для работы с каталогами [Таненбаум, 2002]:

- Создание директории. Вновь созданная директория включает записи с именами '.' и '..', однако считается пустой.
- Удаление директории. Удалена может быть только пустая директория.
- Открытие директории для последующего чтения. Например, чтобы перечислить файлы, входящие в директорию, процесс должен открыть директорию и считать имена всех файлов, которые она включает.
- Закрытие директории после ее чтения для освобождения места во внутренних системных таблицах.
- Поиск. Данный системный вызов возвращает содержимое текущей записи в открытой директории. Вообще говоря, для этих целей может использоваться системный вызов Read, но в этом случае от программиста потребуется знание внутренней структуры директории.
- Получение списка файлов в каталоге.
- Переименование. Имена директорий можно менять, как и имена файлов.
- Создание файла. При создании нового файла необходимо добавить в каталог соответствующий элемент.
- Удаление файла. Удаление из каталога соответствующего элемента. Если удаляемый файл присутствует только в одной директории, то он вообще удаляется из файловой системы, в противном случае система ограничивается только удалением специфицируемой записи.

Очевидно, что создание и удаление файлов предполагает также выполнение соответствующих файловых операций. Имеется еще ряд других системных вызовов, например связанных с защитой информации.

Защита файлов

Информация в компьютерной системе должна быть защищена как от физического разрушения (reliability), так и от несанкционированного доступа (protection).

Контроль доступа к файлам

Наличие в системе многих пользователей предполагает организацию контролируемого доступа к файлам. Выполнение любой операции над файлом должно быть разрешено только в случае наличия у пользователя соответствующих привилегий. Обычно контролируются следующие операции: чтение, запись и выполнение. Другие операции, например копирование файлов или их переименование, также могут контролироваться. Однако они чаще реализуются через перечисленные. Так, операцию копирования файлов можно представить как операцию чтения и последующую операцию записи.

Списки прав доступа

Наиболее общий подход к защите файлов от несанкционированного использования - сделать доступ зависящим от идентификатора пользователя, то есть связать с каждым файлом или директорией список прав доступа (access control list), где перечислены имена пользователей и типы разрешенных для них способов доступа к файлу. Любой запрос на выполнение операции сверяется с таким списком. Основная проблема реализации данного способа - список может быть длинным. Чтобы разрешить всем пользователям читать файл, необходимо всех их внести в список. У такой техники есть два нежелательных следствия.

- Конструирование подобного списка может оказаться сложной задачей, особенно если мы не знаем заранее пользователей системы.
- Запись в директории должна иметь переменный размер (включать список потенциальных пользователей).

Для решения этих проблем создают классификации пользователей, например, в ОС Unix все пользователи разделены на три группы.

1. Владелец (Owner).
2. Группа (Group). Набор пользователей, разделяющих файл и нуждающихся в типовом способе доступа к нему.
3. Остальные (Univers).

Это позволяет реализовать конденсированную версию списка прав доступа. В рамках такой ограниченной классификации задаются только три поля (по одному для каждой группы) для каждой контролируемой операции. В итоге в Unix операции чтения, записи и исполнения контролируются при помощи 9 бит (rwxrwxrwx).

Лекция. Реализация файловой системы

Для организации хранения информации на диске пользователь вначале обычно выполняет его форматирование, выделяя на нем место для структур данных, которые описывают состояние файловой системы в целом. Затем пользователь создает нужную ему структуру каталогов (или директорий), которые, по существу, являются списками вложенных каталогов и собственно файлов. И наконец, он заполняет дисковое пространство файлами, приписывая их тому или иному каталогу. Таким образом, ОС должна предоставить в распоряжение пользователя совокупность системных вызовов, которые обеспечивают его необходимыми сервисами. Кроме того, файловые службы могут решать проблемы проверки и сохранения целостности файловой системы, проблемы повышения производительности и ряд других.

Общая структура файловой системы

Нижний уровень - оборудование. Это в первую очередь магнитные диски с подвижными головками - основные устройства внешней памяти, представляющие собой пакеты

магнитных пластин (поверхностей), между которыми на одном рычаге двигается пакет магнитных головок. Шаг движения пакета головок является дискретным, и каждому положению пакета головок логически соответствует цилиндр магнитного диска. Цилиндры делятся на дорожки (треки), а каждая дорожка размечается на одно и то же количество блоков (секторов) таким образом, что в каждый блок можно записать по максимуму одно и то же число байтов. Следовательно, для обмена с магнитным диском на уровне аппаратуры нужно указать номер цилиндра, номер поверхности, номер блока на соответствующей дорожке и число байтов, которое нужно записать или прочитать от начала этого блока. Таким образом, диски могут быть разбиты на блоки фиксированного размера и можно непосредственно получить доступ к любому блоку (организовать прямой доступ к файлам).

Непосредственно с устройствами (дисками) взаимодействует часть ОС, называемая ***системой ввода-вывода***. Система ввода-вывода предоставляет в распоряжение более высокоуровневого компонента ОС - файловой системы - используемое дисковое пространство в виде непрерывной последовательности блоков фиксированного размера. Система ввода-вывода имеет дело с физическими блоками диска, которые характеризуются адресом, например диск 2, цилиндр 75, сектор 11. Файловая система имеет дело с логическими блоками, каждый из которых имеет номер (от 0 или 1 до N). Размер логических блоков файла совпадает или является кратным размеру физического блока диска и может быть задан равным размеру страницы виртуальной памяти, поддерживаемой аппаратурой компьютера совместно с операционной системой.

В структуре системы управления файлами можно выделить базисную подсистему, которая отвечает за выделение дискового пространства конкретным файлам, и более высокоуровневую логическую подсистему, которая использует структуру дерева директорий для предоставления модулю базисной подсистемы необходимой ей информации, исходя из символического имени файла. Она также ответственна за авторизацию доступа к файлам (см. лекции 11 и 16).

Стандартный запрос на открытие (open) или создание (create) файла поступает от прикладной программы к логической подсистеме. Логическая подсистема, используя структуру директорий, проверяет права доступа и вызывает базовую подсистему для получения доступа к блокам файла. После этого файл считается открытым, он содержится в таблице открытых файлов, и прикладная программа получает в свое распоряжение дескриптор (или handle в системах Microsoft) этого файла. Дескриптор файла является ссылкой на файл в таблице открытых файлов и используется в запросах прикладной программы на чтение-запись из этого файла. Запись в таблице открытых файлов указывает через систему выделения блоков диска на блоки данного файла. Если к моменту открытия файл уже используется другим процессом, то есть содержится в таблице открытых файлов, то после проверки прав доступа к файлу может быть организован совместный доступ. При этом новому процессу также возвращается дескриптор - ссылка на файл в таблице открытых файлов. Далее в тексте подробно проанализирована работа наиболее важных системных вызовов.

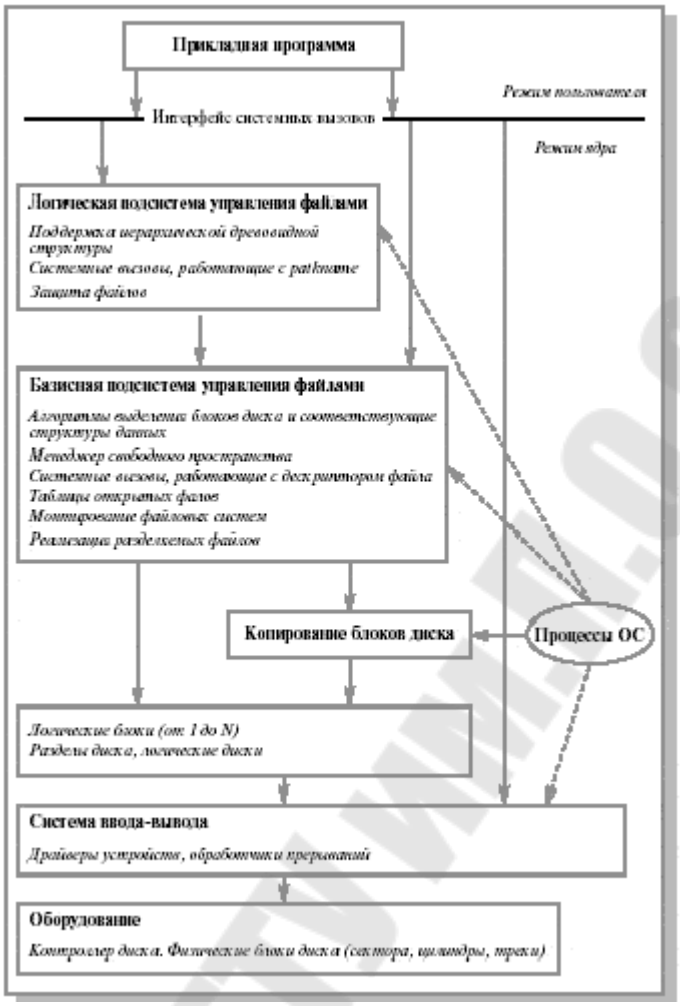

Рис. 12.1. Блок-схема файловой системы

Управление внешней памятью

Прежде чем описывать структуру данных файловой системы на диске, необходимо рассмотреть алгоритмы выделения дискового пространства и способы учета свободной и занятой дисковой памяти. Эти задачи связаны между собой.

Методы выделения дискового пространства

Ключевым, безусловно, является вопрос, какой тип структур используется для учета отдельных блоков файла, то есть способ связывания файлов с блоками диска. В ОС используется несколько методов выделения файлу дискового пространства. Для каждого из методов запись в директории, соответствующая символьному имени файла, содержит указатель, следуя которому можно найти все блоки данного файла.

Выделение непрерывной последовательностью блоков

Простейший способ - хранить каждый файл как непрерывную последовательность блоков диска. При непрерывном расположении файл характеризуется адресом и длиной (в блоках). Файл, стартующий с блока b, занимает затем блоки b+1, b+2, ... b+n-1.

Эта схема имеет два преимущества. Во-первых, ее легко реализовать, так как выяснение местонахождения файла сводится к вопросу, где находится первый блок. Во-вторых, она

обеспечивает хорошую производительность, так как целый файл может быть считан за одну дисковую операцию.

Непрерывное выделение используется в ОС IBM/CMS, в ОС RSX-11 (для выполняемых файлов) и в ряде других.

Этот способ распространен мало, и вот почему. В процессе эксплуатации диск представляет собой некоторую совокупность свободных и занятых фрагментов. Не всегда имеется подходящий по размеру свободный фрагмент для нового файла. Проблема непрерывного расположения может рассматриваться как частный случай более общей проблемы выделения блока нужного размера из списка свободных блоков. Типовыми решениями этой задачи являются стратегии первого подходящего, наиболее подходящего и наименее подходящего (сравните с проблемой выделения памяти в методе с динамическим распределением). Как и в случае выделения нужного объема оперативной памяти в схеме с динамическими разделами (см. лекцию 8), метод страдает от внешней фрагментации, в большей или меньшей степени, в зависимости от размера диска и среднего размера файла.

Кроме того, непрерывное распределение внешней памяти неприменимо до тех пор, пока неизвестен максимальный размер файла. Иногда размер выходного файла оценить легко (при копировании). Чаще, однако, это трудно сделать, особенно в тех случаях, когда размер файла меняется. Если места не хватило, то пользовательская программа может быть приостановлена с учетом выделения дополнительного места для файла при последующем рестарте. Некоторые ОС используют модифицированный вариант непрерывного выделения - основные блоки файла + резервные блоки. Однако с выделением блоков из резерва возникают те же проблемы, так как приходится решать задачу выделения непрерывной последовательности блоков диска теперь уже из совокупности резервных блоков.

Единственным приемлемым решением перечисленных проблем является периодическое уплотнение содержимого внешней памяти, или "сборка мусора", цель которой состоит в объединении свободных участков в один большой блок. Но это дорогостоящая операция, которую невозможно осуществлять слишком часто.

Таким образом, когда содержимое диска постоянно изменяется, данный метод нерационален. Однако для стационарных файловых систем, например для файловых систем компакт-дисков, он вполне пригоден.

Связный список

Внешняя фрагментация - основная проблема рассмотренного выше метода - может быть устранена за счет представления файла в виде связного списка блоков диска. Запись в директории содержит указатель на первый и последний блоки файла (иногда в качестве варианта используется специальный знак конца файла - EOF). Каждый блок содержит указатель на следующий блок (см. рис. 12.2).

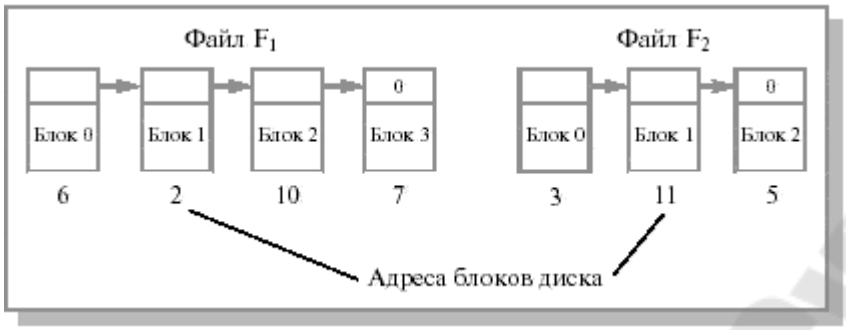

Рис. 12.2. Хранение файла в виде связного списка дисковых блоков

Внешняя фрагментация для данного метода отсутствует. Любой свободный блок может быть использован для удовлетворения запроса. Заметим, что нет необходимости декларировать размер файла в момент создания. Файл может расти неограниченно.
Связное выделение имеет, однако, несколько существенных недостатков:

- при прямом доступе к файлу для поиска i-го блока нужно осуществить несколько обращений к диску, последовательно считывая блоки от 1 до i-1, то есть выборка логически смежных записей, которые занимают физически несмежные секторы, может требовать много времени. Здесь мы теряем все преимущества прямого доступа к файлу.
- данный способ не очень надежен. Наличие дефектного блока в списке приводит к потере информации в оставшейся части файла и потенциально к потере дискового пространства, отведенного под этот файл.
- для указателя на следующий блок внутри блока нужно выделить место, что не всегда удобно. Емкость блока, традиционно являющаяся степенью двойки (многие программы читают и пишут блоками по степеням двойки), таким образом, перестает быть степенью двойки, так как указатель отбирает несколько байтов.

Поэтому метод связного списка обычно в чистом виде не используется.

Таблица отображения файлов

Одним из вариантов предыдущего способа является хранение указателей не в дисковых блоках, а в индексной таблице в памяти, которая называется таблицей отображения файлов (FAT - file allocation table) (см. рис. 12.3). Этой схемы придерживаются многие ОС (MS-DOS, OS/2, MS Windows и др.)

По-прежнему существенно, что запись в директории содержит только ссылку на первый блок. Далее при помощи таблицы FAT можно локализовать блоки файла независимо от его размера. В тех строках таблицы, которые соответствуют последним блокам файлов, обычно записывается некоторое граничное значение, например EOF.

Главное достоинство данного подхода состоит в том, что по таблице отображения можно судить о физическом соседстве блоков, располагающихся на диске, и при выделении нового блока можно легко найти свободный блок диска, находящийся поблизости от других блоков данного файла. Минусом данной схемы может быть необходимость хранения в памяти этой довольно большой таблицы.

Рис. 12.3. Метод связного списка с использованием таблицы в оперативной памяти

Индексные узлы

Наиболее распространенный метод выделения файлу блоков диска - связать с каждым файлом небольшую таблицу, называемую индексным узлом (i-node), которая перечисляет атрибуты и дисковые адреса блоков файла (см. рис 12.4). Запись в директории, относящаяся к файлу, содержит адрес индексного блока. По мере заполнения файла указатели на блоки диска в индексном узле принимают осмысленные значения.

Индексирование поддерживает прямой доступ к файлу, без ущерба от внешней фрагментации. Индексированное размещение широко распространено и поддерживает как последовательный, так и прямой доступ к файлу.

Обычно применяется комбинация одноуровневого и многоуровневых индексов. Первые несколько адресов блоков файла хранятся непосредственно в индексном узле, таким образом, для маленьких файлов индексный узел хранит всю необходимую информацию об адресах блоков диска. Для больших файлов один из адресов индексного узла указывает на блок косвенной адресации. Данный блок содержит адреса дополнительных блоков диска. Если этого недостаточно, используется блок двойной косвенной адресации, который содержит адреса блоков косвенной адресации. Если и этого не хватает, используется блок тройной косвенной адресации.

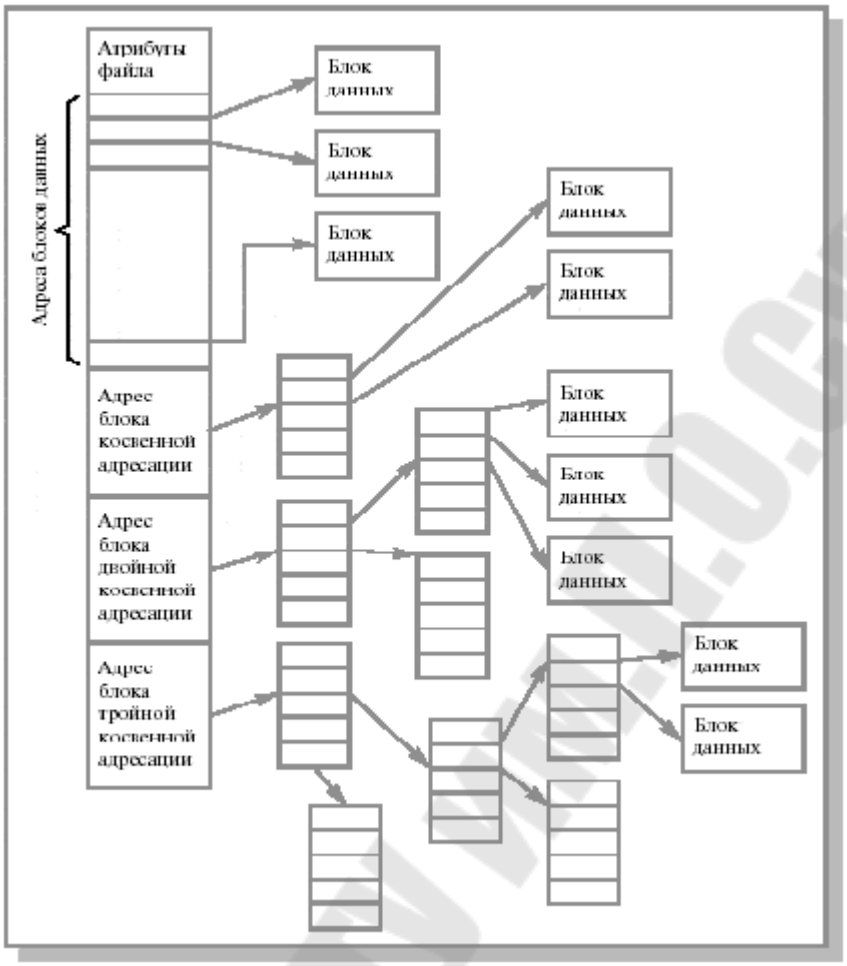

Рис. 12.4. Структура индексного узла

Данную схему используют файловые системы Unix (а также файловые системы HPFS, NTFS и др.). Такой подход позволяет при фиксированном, относительно небольшом размере индексного узла поддерживать работу с файлами, размер которых может меняться от нескольких байтов до нескольких гигабайтов. Существенно, что для маленьких файлов используется только прямая адресация, обеспечивающая максимальную производительность.

Управление свободным и занятым дисковым пространством

Дисковое пространство, не выделенное ни одному файлу, также должно быть управляемым. В современных ОС используется несколько способов учета используемого места на диске. Рассмотрим наиболее распространенные.

Учет при помощи организации битового вектора

Часто список свободных блоков диска реализован в виде битового вектора (bit map или bit vector). Каждый блок представлен одним битом, принимающим значение 0 или 1, в зависимости от того, занят он или свободен. Например, 00111100111100011000001 ... .

Главное преимущество этого подхода состоит в том, что он относительно прост и эффективен при нахождении первого свободного блока или n последовательных блоков на диске. Многие компьютеры имеют инструкции манипулирования битами, которые могут использоваться для этой цели. Например, компьютеры семейств Intel и Motorola имеют

инструкции, при помощи которых можно легко локализовать первый единичный бит в слове.

Описываемый метод учета свободных блоков используется в Apple Macintosh.

Несмотря на то что размер описанного битового вектора наименьший из всех возможных структур, даже такой вектор может оказаться большого размера. Поэтому данный метод эффективен, только если битовый вектор помещается в памяти целиком, что возможно лишь для относительно небольших дисков. Например, диск размером 4 Гбайт с блоками по 4 Кбайт нуждается в таблице размером 128 Кбайт для управления свободными блоками. Иногда, если битовый вектор становится слишком большим, для ускорения поиска в нем его разбивают на регионы и организуют резюмирующие структуры данных, содержащие сведения о количестве свободных блоков для каждого региона.

Учет при помощи организации связного списка

Другой подход - связать в список все свободные блоки, размещая указатель на первый свободный блок в специально отведенном месте диска, попутно кэшируя в памяти эту информацию.

Подобная схема не всегда эффективна. Для трассирования списка нужно выполнить много обращений к диску. Однако, к счастью, нам необходим, как правило, только первый свободный блок.

Иногда прибегают к модификации подхода связного списка, организуя хранение адресов n свободных блоков в первом свободном блоке. Первые n-1 этих блоков действительно используются. Последний блок содержит адреса других n блоков и т. д.

Существуют и другие методы, например, свободное пространство можно рассматривать как файл и вести для него соответствующий индексный узел.

Размер блока

Размер логического блока играет важную роль. В некоторых системах (Unix) он может быть задан при форматировании диска. Небольшой размер блока будет приводить к тому, что каждый файл будет содержать много блоков. Чтение блока осуществляется с задержками на поиск и вращение, таким образом, файл из многих блоков будет читаться медленно. Большие блоки обеспечивают более высокую скорость обмена с диском, но из-за внутренней фрагментации (каждый файл занимает целое число блоков, и в среднем половина последнего блока пропадает) снижается процент полезного дискового пространства.

Для систем со страничной организацией памяти характерна сходная проблема с размером страницы.

Проведенные исследования показали, что большинство файлов имеют небольшой размер. Например, в Unix приблизительно 85% файлов имеют размер менее 8 Кбайт и 48% - менее 1 Кбайта.

Можно также учесть, что в системах с виртуальной памятью желательно, чтобы единицей пересылки диск-память была страница (наиболее распространенный размер страниц

памяти - 4 Кбайта). Отсюда обычный компромиссный выбор блока размером 512 байт, 1 Кбайт, 2 Кбайт, 4 Кбайт.

Структура файловой системы на диске

Рассмотрение методов работы с дисковым пространством дает общее представление о совокупности служебных данных, необходимых для описания файловой системы. Структура служебных данных типовой файловой системы, например Unix, на одном из разделов диска, таким образом, может состоять из четырех основных частей (см. рис. 12.5).

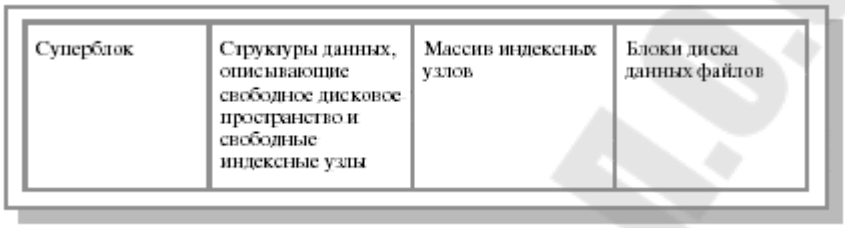

Рис. 12.5. Примерная структура файловой системы на диске

В начале раздела находится суперблок, содержащий общее описание файловой системы, например:

- тип файловой системы;
- размер файловой системы в блоках;
- размер массива индексных узлов;
- размер логического блока.

Описанные структуры данных создаются на диске в результате его форматирования (например, утилитами format, makefs и др.). Их наличие позволяет обращаться к данным на диске как к файловой системе, а не как к обычной последовательности блоков.

В файловых системах современных ОС для повышения устойчивости поддерживается несколько копий суперблока. В некоторых версиях Unix суперблок включал также и структуры данных, управляющие распределением дискового пространства, в результате чего суперблок непрерывно подвергался модификации, что снижало надежность файловой системы в целом. Выделение структур данных, описывающих дисковое пространство, в отдельную часть является более правильным решением.

Массив индексных узлов (ilist) содержит список индексов, соответствующих файлам данной файловой системы. Размер массива индексных узлов определяется администратором при установке системы. Максимальное число файлов, которые могут быть созданы в файловой системе, определяется числом доступных индексных узлов.

В блоках данных хранятся реальные данные файлов. Размер логического блока данных может задаваться при форматировании файловой системы. Заполнение диска содержательной информацией предполагает использование блоков хранения данных для файлов директорий и обычных файлов и имеет следствием модификацию массива индексных узлов и данных, описывающих пространство диска. Отдельно взятый блок данных может принадлежать одному и только одному файлу в файловой системе.

Реализация директорий

Как уже говорилось, директория или каталог - это файл, имеющий вид таблицы и хранящий список входящих в него файлов или каталогов. Основная задача файлов-директорий - поддержка иерархической древовидной структуры файловой системы. Запись в директории имеет определенный для данной ОС формат, зачастую неизвестный пользователю, поэтому блоки данных файла-директории заполняются не через операции записи, а при помощи специальных системных вызовов (например, создание файла).

Для доступа к файлу ОС использует путь (pathname), сообщенный пользователем. Запись в директории связывает имя файла или имя поддиректории с блоками данных на диске (см. рис. 12.6). В зависимости от способа выделения файлу блоков диска (см. раздел "Методы выделения дискового пространства") эта ссылка может быть номером первого блока или номером индексного узла. В любом случае обеспечивается связь символьного имени файла с данными на диске.

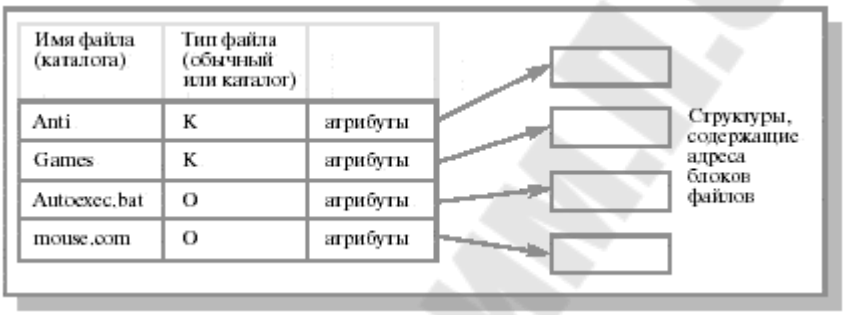

Рис. 12.6. Реализация директорий

Когда система открывает файл, она ищет его имя в директории. Затем из записи в директории или из структуры, на которую запись в директории указывает, извлекаются атрибуты и адреса блоков файла на диске. Эта информация помещается в системную таблицу в главной памяти. Все последующие ссылки на данный файл используют эту информацию. Атрибуты файла можно хранить непосредственно в записи в директории, как показано на рис. 12.6. Однако для организации совместного доступа к файлам удобнее хранить атрибуты в индексном узле, как это делается в Unix.

Примеры реализации директорий в некоторых ОС

Директории в ОС MS-DOS

В ОС MS-DOS типовая запись в директории имеет вид, показанный на рис. 12.7.

Рис. 12.7. Вариант записи в директории MS-DOS

В ОС MS-DOS, как и в большинстве современных ОС, директории могут содержать поддиректории (специфицируемые битом атрибута), что позволяет конструировать произвольное дерево директорий файловой системы.

Номер первого блока используется в качестве индекса в таблице FAT. Далее по цепочке в этой таблице могут быть найдены остальные блоки.

Директории в ОС Unix

Структура директории проста. Каждая запись содержит имя файла и номер его индексного узла (см. рис. 12.8). Вся остальная информация о файле (тип, размер, время модификации, владелец и т. д. и номера дисковых блоков) находится в индексном узле.

Рис. 12.8. Вариант записи в директории Unix

В более поздних версиях Unix форма записи претерпела ряд изменений, например имя файла описывается структурой. Однако суть осталась прежней.

Поиск в директории

Список файлов в директории обычно не является упорядоченным по именам файлов. Поэтому правильный выбор алгоритма поиска имени файла в директории имеет большое влияние на эффективность и надежность файловых систем.

Линейный поиск

Существует несколько стратегий просмотра списка символьных имен. Простейшей из них является линейный поиск. Директория просматривается с самого начала, пока не встретится нужное имя файла. Хотя это наименее эффективный способ поиска, оказывается, что в большинстве случаев он работает с приемлемой производительностью. Например, авторы Unix утверждали, что линейного поиска вполне достаточно. По-видимому, это связано с тем, что на фоне относительно медленного доступа к диску некоторые задержки, возникающие в процессе сканирования списка, несущественны.

Метод прост, но требует временных затрат. Для создания нового файла вначале нужно проверить директорию на наличие такого же имени. Затем имя нового файла вставляется в конец директории (если, разумеется, файл с таким же именем в директории не существует, в противном случае нужно информировать пользователя). Для удаления файла нужно также выполнить поиск его имени в списке и пометить запись как неиспользуемую.

Реальный недостаток данного метода - последовательный поиск файла. Информация о структуре директории используется часто, и неэффективный способ поиска будет заметен пользователями. Можно свести поиск к бинарному, если отсортировать список файлов. Однако это усложнит создание и удаление файлов, так как требуется перемещение большого объема информации.

Хеш-таблица

Хеширование (см. например, [Ахо, 2001]) - другой способ, который может использоваться для размещения и последующего поиска имени файла в директории. В данном методе имена файлов также хранятся в каталоге в виде линейного списка, но дополнительно используется хеш-таблица. Хеш-таблица, точнее построенная на ее основе хеш-функция, позволяет по имени файла получить указатель на имя файла в списке. Таким образом, можно существенно уменьшить время поиска.

В результате хеширования могут возникать коллизии, то есть ситуации, когда функция хеширования, примененная к разным именам файлов, дает один и тот же результат. Обычно имена таких файлов объединяют в связные списки, предполагая в дальнейшем осуществление в них последовательного поиска нужного имени файла. Выбор подходящего алгоритма хеширования позволяет свести к минимуму число коллизий. Однако всегда есть вероятность неблагоприятного исхода, когда непропорционально большому числу имен файлов функция хеширования ставит в соответствие один и тот же результат. В таком случае преимущество использования этой схемы по сравнению с последовательным поиском практически утрачивается.

Другие методы поиска

Помимо описанных методов поиска имени файла, в директории существуют и другие. В качестве примера можно привести организацию поиска в каталогах файловой системы NTFS при помощи так называемого B-дерева, которое стало стандартным способом организации индексов в системах баз данных (см. [Ахо, 2001]).

Монтирование файловых систем

Так же как файл должен быть открыт перед использованием, и файловая система, хранящаяся на разделе диска, должна быть смонтирована, чтобы стать доступной процессам системы.

Функция mount (монтировать) связывает файловую систему из указанного раздела на диске с существующей иерархией файловых систем, а функция umount (демонтировать) выключает файловую систему из иерархии. Функция mount, таким образом, дает пользователям возможность обращаться к данным в дисковом разделе как к файловой системе, а не как к последовательности дисковых блоков. Процедура монтирования состоит в следующем. Пользователь (в Unix это суперпользователь) сообщает ОС имя устройства и место в файловой структуре (имя пустого каталога), куда нужно присоединить файловую систему (точка монтирования) (см. рис. 12.9 и рис. 12.10). Например, в ОС Unix библиотечный вызов mount имеет вид:

```
mount(special pathname,directory pathname,options);
```

где special pathname - имя специального файла устройства (в общем случае имя раздела), соответствующего дисковому разделу с монтируемой файловой системой, directory pathname - каталог в существующей иерархии, где будет монтироваться файловая система (другими словами, точка или место монтирования), а options указывает, следует ли монтировать файловую систему "только для чтения" (при этом не будут выполняться такие функции, как write и create, которые производят запись в файловую систему). Затем ОС должна убедиться, что устройство содержит действительную файловую систему ожидаемого формата с суперблоком, списком индексов и корневым индексом.

Некоторые ОС осуществляют монтирование автоматически, как только встретят диск в первый раз (жесткие диски на этапе загрузки, гибкие - когда они вставлены в дисковод), ОС ищет файловую систему на устройстве. Если файловая система на устройстве имеется, она монтируется на корневом уровне, при этом к цепочке имен абсолютного имени файла (pathname) добавляется буква раздела.

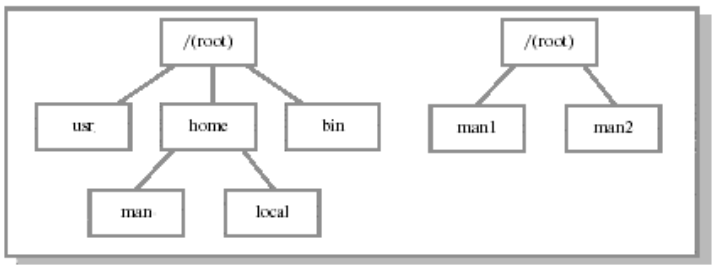

Рис. 12.9. Две файловые системы до монтирования

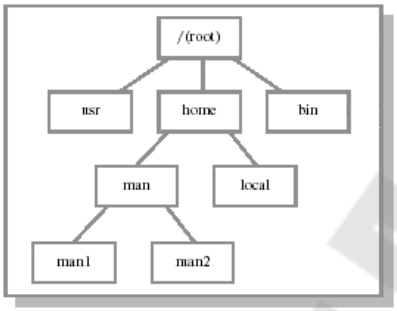

Рис. 12.10. Общая файловая система после монтирования

Ядро поддерживает таблицу монтирования с записями о каждой смонтированной файловой системе. В каждой записи содержится информация о вновь смонтированном устройстве, о его суперблоке и корневом каталоге, а также сведения о точке монтирования. Для устранения потенциально опасных побочных эффектов число линков (см. следующий раздел) к каталогу - точке монтирования - должно быть равно 1. Занесение информации в таблицу монтирования производится немедленно, поскольку может возникнуть конфликт между двумя процессами. Например, если монтирующий процесс приостановлен для открытия устройства или считывания суперблока файловой системы, а тем временем другой процесс может попытаться смонтировать файловую систему.

Наличие в логической структуре файлового архива точек монтирования требует аккуратной реализации алгоритмов, осуществляющих навигацию по каталогам. Точку монтирования можно пересечь двумя способами: из файловой системы, где производится монтирование, в файловую систему, которая монтируется (в направлении от глобального корня к листу), и в обратном направлении. Алгоритмы поиска файлов должны предусматривать ситуации, в которых очередной компонент пути к файлу является точкой монтирования, когда вместо анализа индексного узла очередной директории приходится осуществлять обработку суперблока монтированной системы.

Связывание файлов

Иерархическая организация, положенная в основу древовидной структуры файловой системы современных ОС, не предусматривает выражения отношений, в которых потомки связываются более чем с одним предком. Такая негибкость частично устраняется возможностью реализации связывания файлов или организации линков (link).

Ядро позволяет пользователю связывать каталоги, упрощая написание программ, требующих пересечения дерева файловой системы (см. рис. 12.11). Часто имеет смысл хранить под разными именами одну и ту же команду (выполняемый файл). Например, выполняемый файл традиционного текстового редактора ОС Unix vi обычно может вызываться под именами ex, edit, vi, view и vedit файловой системы. Соединение между директорией и разделяемым файлом называется "связью" или "ссылкой" (link). Дерево файловой системы превращается в циклический граф.

Это удобно, но создает ряд дополнительных проблем.

Простейший способ реализовать связывание файла - просто дублировать информацию о нем в обеих директориях. При этом, однако, может возникнуть проблема совместимости в случае, если владельцы этих директорий попытаются независимо друг от друга изменить содержимое файла. Например, в ОС CP/M запись в директории о файле непосредственно содержит адреса дисковых блоков. Поэтому копии тех же дисковых адресов должны быть сделаны и в другой директории, куда файл линкуется. Если один из пользователей что-то добавляет к файлу, новые блоки будут перечислены только у него в директории и не будут "видны" другому пользователю.

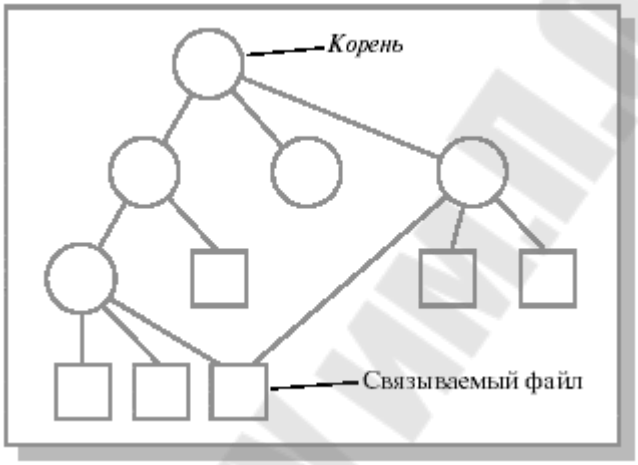

Рис. 12.11. Структура файловой системы с возможностью связывания файла с новым именем

Проблема такого рода может быть решена двумя способами. Первый из них - так называемая жесткая связь (hard link). Если блоки данных файла перечислены не в директории, а в небольшой структуре данных (например, в индексном узле), связанной собственно с файлом, то второй пользователь может связаться непосредственно с этой, уже существующей структурой.

Альтернативное решение - создание нового файла, который содержит путь к связываемому файлу. Такой подход называется символической линковкой (soft или symbolic link). При этом в соответствующем каталоге создается элемент, в котором имени связи сопоставляется некоторое имя файла (этот файл даже не обязан существовать к моменту создания символической связи). Для символической связи может создаваться отдельный индексный узел и даже заводиться отдельный блок данных для хранения потенциально длинного имени файла.

Каждый из этих методов имеет свои минусы. В случае жесткой связи возникает необходимость поддержки счетчика ссылок на файл для корректной реализации операции удаления файла. Например, в Unix такой счетчик является одним из атрибутов, хранящихся в индексном узле. Удаление файла одним из пользователей уменьшает количество ссылок на файл на 1. Реальное удаление файла происходит, когда число ссылок на файл становится равным 0.

В случае символической линковки такая проблема не возникает, так как только реальный владелец имеет ссылку на индексный узел файла. Если собственник удаляет файл, то он разрушается, и попытки других пользователей работать с ним закончатся провалом. Удаление символического линка на файл никак не влияет. Проблема организации символической связи - потенциальное снижение скорости доступа к файлу. Файл

символического линка хранит путь к файлу, содержащий список вложенных директорий, для прохождения по которому необходимо осуществить несколько обращений к диску.

Символический линк имеет то преимущество, что он может использоваться для организации удобного доступа к файлам удаленных компьютеров, если, например, добавить к пути сетевой адрес удаленной машины.

Циклический граф - структура более гибкая, нежели простое дерево, но работа с ней требует большой аккуратности. Поскольку теперь к файлу существует несколько путей, программа поиска файла может найти его на диске несколько раз. Простейшее практическое решение данной проблемы - ограничить число директорий при поиске. Полное устранение циклов при поиске - довольно трудоемкая процедура, выполняемая специальными утилитами и связанная с многократной трассировкой директорий файловой системы.

Кооперация процессов при работе с файлами

Когда различные пользователи работают вместе над проектом, они часто нуждаются в разделении файлов.

Разделяемый файл - разделяемый ресурс. Как и в случае любого совместно используемого ресурса, процессы должны синхронизировать доступ к совместно используемым файлам, каталогам, чтобы избежать тупиковых ситуаций, дискриминации отдельных процессов и снижения производительности системы.

Например, если несколько пользователей одновременно редактируют какой-либо файл и не принято специальных мер, то результат будет непредсказуем и зависит от того, в каком порядке осуществлялись записи в файл. Между двумя операциями read одного процесса другой процесс может модифицировать данные, что для многих приложений неприемлемо. Простейшее решение данной проблемы - предоставить возможность одному из процессов захватить файл, то есть блокировать доступ к разделяемому файлу других процессов на все время, пока файл остается открытым для данного процесса. Однако это было бы недостаточно гибко и не соответствовало бы характеру поставленной задачи.

Рассмотрим вначале грубый подход, то есть временный захват пользовательским процессом файла или записи (части файла между указанными позициями).

Системный вызов, позволяющий установить и проверить блокировки на файл, является неотъемлемым атрибутом современных многопользовательских ОС. В принципе, было бы логично связать синхронизацию доступа к файлу как к единому целому с системным вызовом open (т. е., например, открытие файла в режиме записи или обновления могло бы означать его монопольную блокировку соответствующим процессом, а открытие в режиме чтения - совместную блокировку). Так поступают во многих операционных системах (начиная с ОС Multics). В ОС Unix это не так, что имеет исторические причины.

В первой версии системы Unix, разработанной Томпсоном и Ричи, механизм захвата файла отсутствовал. Применялся очень простой подход к обеспечению параллельного (от нескольких процессов) доступа к файлам: система позволяла любому числу процессов одновременно открывать один и тот же файл в любом режиме (чтения, записи или обновления) и не предпринимала никаких синхронизационных действий. Вся ответственность за корректность совместной обработки файла ложилась на использующие

его процессы, и система даже не предоставляла каких-либо особых средств для синхронизации доступа процессов к файлу. Однако впоследствии для того, чтобы повысить привлекательность системы для коммерческих пользователей, работающих с базами данных, в версию V системы были включены механизмы захвата файла и записи, базирующиеся на системном вызове fcntl.

Допускается два варианта синхронизации: с ожиданием, когда требование блокировки может привести к откладыванию процесса до того момента, когда это требование может быть удовлетворено, и без ожидания, когда процесс немедленно оповещается об удовлетворении требования блокировки или о невозможности ее удовлетворения в данный момент.

Установленные блокировки относятся только к тому процессу, который их установил, и не наследуются процессами-потомками этого процесса. Более того, даже если некоторый процесс пользуется синхронизационными возможностями системного вызова fcntl, другие процессы по-прежнему могут работать с тем файлом без всякой синхронизации. Другими словами, это дело группы процессов, совместно использующих файл, - договориться о способе синхронизации параллельного доступа.

Более тонкий подход заключается в прозрачной для пользователя блокировке отдельных структур ядра, отвечающих за работу с файлами части пользовательских данных. Например, в ОС Unix во время системного вызова, осуществляющего ту или иную операцию с файлом, как правило, происходит блокирование индексного узла, содержащего адреса блоков данных файла. Может показаться, что организация блокировок или запрета более чем одному процессу работать с файлом во время выполнения системного вызова является излишней, так как в подавляющем большинстве случаев выполнение системных вызовов и так не прерывается, то есть ядро работает в условиях невытесняющей многозадачности. Однако в данном случае это не совсем так. Операции чтения и записи занимают продолжительное время и лишь инициируются центральным процессором, а осуществляются по независимым каналам, поэтому установка блокировок на время системного вызова является необходимой гарантией атомарности операций чтения и записи. На практике оказывается достаточным заблокировать один из буферов кэша диска, в заголовке которого ведется список процессов, ожидающих освобождения данного буфера. Таким образом, в соответствии с семантикой Unix изменения, сделанные одним пользователем, немедленно становятся "видны" другому пользователю, который держит данный файл открытым одновременно с первым.

Примеры разрешения коллизий и тупиковых ситуаций

Логика работы системы в сложных ситуациях может проиллюстрировать особенности организации мультидоступа.

Рассмотрим в качестве примера образование потенциального тупика при создании связи (link), когда разрешен совместный доступ к файлу [Bach, 1986].

Два процесса, выполняющие одновременно следующие функции:

процесс A: `link("a/b/c/d","e/f/g");`

процесс B: `link("e/f","a/b/c/d/ee");`

могут зайти в тупик. Предположим, что процесс А обнаружил индекс файла "a/b/c/d" в тот самый момент, когда процесс В обнаружил индекс файла "e/f". Фраза "в тот же самый момент" означает, что системой достигнуто состояние, при котором каждый процесс получил искомый индекс. Когда же теперь процесс А попытается получить индекс файла "e/f", он приостановит свое выполнение до тех пор, пока индекс файла "f" не освободится. В то же время процесс В пытается получить индекс каталога "a/b/c/d" и приостанавливается в ожидании освобождения индекса файла "d". Процесс А будет удерживать заблокированным индекс, нужный процессу В, а процесс В, в свою очередь, будет удерживать заблокированным индекс, необходимый процессу А.

Для предотвращения этого классического примера взаимной блокировки в файловой системе принято, чтобы ядро освобождало индекс исходного файла после увеличения значения счетчика связей. Тогда, поскольку первый из ресурсов (индекс) свободен при обращении к следующему ресурсу, взаимной блокировки не происходит.

Поводов для нежелательной конкуренции между процессами много, особенно при удалении имен каталогов. Предположим, что один процесс пытается найти данные файла по его полному символическому имени, последовательно проходя компонент за компонентом, а другой процесс удаляет каталог, имя которого входит в путь поиска. Допустим, процесс А делает разбор имени "a/b/c/d" и приостанавливается во время получения индексного узла для файла "c". Он может приостановиться при попытке заблокировать индексный узел или при попытке обратиться к дисковому блоку, где этот индексный узел хранится. Если процессу В нужно удалить связь для каталога с именем "c", он может приостановиться по той же самой причине, что и процесс А. Пусть ядро впоследствии решит возобновить процесс В раньше процесса А. Прежде чем процесс А продолжит свое выполнение, процесс В завершится, удалив связь каталога "c" и его содержимое по этой связи. Позднее процесс А попытается обратиться к несуществующему индексному узлу, который уже был удален. Алгоритм поиска файла, проверяющий в первую очередь неравенство значения счетчика связей> нулю, должен сообщить об ошибке.

Надежность файловой системы

Жизнь полна неприятных неожиданностей, а разрушение файловой системы зачастую более опасно, чем разрушение компьютера. Поэтому файловые системы должны разрабатываться с учетом подобной возможности. Помимо очевидных решений, например своевременное дублирование информации (backup), файловые системы современных ОС содержат специальные средства для поддержки собственной совместимости.

Целостность файловой системы

Важный аспект надежной работы файловой системы - контроль ее целостности. В результате файловых операций блоки диска могут считываться в память, модифицироваться и затем записываться на диск. Причем многие файловые операции затрагивают сразу несколько объектов файловой системы. Например, копирование файла предполагает выделение ему блоков диска, формирование индексного узла, изменение содержимого каталога и т. д. В течение короткого периода времени между этими шагами информация в файловой системе оказывается несогласованной.

И если вследствие непредсказуемой остановки системы на диске будут сохранены изменения только для части этих объектов (нарушена атомарность файловой операции), файловая система на диске может быть оставлена в несовместимом состоянии. В

результате могут возникнуть нарушения логики работы с данными, например появиться "потерянные" блоки диска, которые не принадлежат ни одному файлу и в то же время помечены как занятые, или, наоборот, блоки, помеченные как свободные, но в то же время занятые (на них есть ссылка в индексном узле) или другие нарушения.

В современных ОС предусмотрены меры, которые позволяют свести к минимуму ущерб от порчи файловой системы и затем полностью или частично восстановить ее целостность.

Порядок выполнения операций

Очевидно, что для правильного функционирования файловой системы значимость отдельных данных неравноценна. Искажение содержимого пользовательских файлов не приводит к серьезным (с точки зрения целостности файловой системы) последствиям, тогда как несоответствия в файлах, содержащих управляющую информацию (директории, индексные узлы, суперблок и т. п.), могут быть катастрофическими. Поэтому должен быть тщательно продуман порядок выполнения операций со структурами данных файловой системы.

Рассмотрим пример создания жесткой связи для файла [Робачевский, 1999]. Для этого файловой системе необходимо выполнить следующие операции:

- создать новую запись в каталоге, указывающую на индексный узел файла;
- увеличить счетчик связей в индексном узле.

Если аварийный останов произошел между 1-й и 2-й операциями, то в каталогах файловой системы будут существовать два имени файла, адресующих индексный узел со значением счетчика связей, равному 1. Если теперь будет удалено одно из имен, это приведет к удалению файла как такового. Если же порядок операций изменен и, как прежде, останов произошел между первой и второй операциями, файл будет иметь несуществующую жесткую связь, но существующая запись в каталоге будет правильной. Хотя это тоже является ошибкой, но ее последствия менее серьезны, чем в предыдущем случае.

Журнализация

Другим средством поддержки целостности является заимствованный из систем управления базами данных прием, называемый журнализация (иногда употребляется термин "журналирование"). Последовательность действий с объектами во время файловой операции протоколируется, и если произошел останов системы, то, имея в наличии протокол, можно осуществить откат системы назад в исходное целостное состояние, в котором она пребывала до начала операции. Подобная избыточность может стоить дорого, но она оправданна, так как в случае отказа позволяет реконструировать потерянные данные.

Для отката необходимо, чтобы для каждой протоколируемой в журнале операции существовала обратная. Например, для каталогов и реляционных СУБД это именно так. По этой причине, в отличие от СУБД, в файловых системах протоколируются не все изменения, а лишь изменения метаданных (индексных узлов, записей в каталогах и др.). Изменения в данных пользователя в протокол не заносятся. Кроме того, если протоколировать изменения пользовательских данных, то этим будет нанесен серьезный ущерб производительности системы, поскольку кэширование потеряет смысл.

Журнализация реализована в NTFS, Ext3FS, ReiserFS и других системах. Чтобы подчеркнуть сложность задачи, нужно отметить, что существуют не вполне очевидные проблемы, связанные с процедурой отката. Например, отмена одних изменений может затрагивать данные, уже использованные другими файловыми операциями. Это означает, что такие операции также должны быть отменены. Данная проблема получила название каскадного отката транзакций [Брукшир, 2001]

Проверка целостности файловой системы при помощи утилит

Если же нарушение все же произошло, то для устранения проблемы несовместимости можно прибегнуть к утилитам (fsck, chkdsk, scandisk и др.), которые проверяют целостность файловой системы. Они могут запускаться после загрузки или после сбоя и осуществляют многократное сканирование разнообразных структур данных файловой системы в поисках противоречий.

Возможны также эвристические проверки. Например, нахождение индексного узла, номер которого превышает их число на диске или поиск в пользовательских директориях файлов, принадлежащих суперпользователю.

К сожалению, приходится констатировать, что не существует никаких средств, гарантирующих абсолютную сохранность информации в файлах, и в тех ситуациях, когда целостность информации нужно гарантировать с высокой степенью надежности, прибегают к дорогостоящим процедурам дублирования.

Управление "плохими" блоками

Наличие дефектных блоков на диске - обычное дело. Внутри блока наряду с данными хранится контрольная сумма данных. Под "плохими" блоками обычно понимают блоки диска, для которых вычисленная контрольная сумма считываемых данных не совпадает с хранимой контрольной суммой. Дефектные блоки обычно появляются в процессе эксплуатации. Иногда они уже имеются при поставке вместе со списком, так как очень затруднительно для поставщиков сделать диск полностью свободным от дефектов. Рассмотрим два решения проблемы дефектных блоков - одно на уровне аппаратуры, другое на уровне ядра ОС.

Первый способ - хранить список плохих блоков в контроллере диска. Когда контроллер инициализируется, он читает плохие блоки и замещает дефектный блок резервным, помечая отображение в списке плохих блоков. Все реальные запросы будут идти к резервному блоку. Следует иметь в виду, что при этом механизм подъемника (наиболее распространенный механизм обработки запросов к блокам диска) будет работать неэффективно. Дело в том, что существует стратегия очередности обработки запросов к диску (подробнее см. лекцию "ввод-вывод"). Стратегия диктует направление движения считывающей головки диска к нужному цилиндру. Обычно резервные блоки размещаются на внешних цилиндрах. Если плохой блок расположен на внутреннем цилиндре и контроллер осуществляет подстановку прозрачным образом, то кажущееся движение головки будет осуществляться к внутреннему цилиндру, а фактическое - к внешнему. Это является нарушением стратегии и, следовательно, минусом данной схемы.

Решение на уровне ОС может быть следующим. Прежде всего, необходимо тщательно сконструировать файл, содержащий дефектные блоки. Тогда они изымаются из списка свободных блоков. Затем нужно каким-то образом скрыть этот файл от прикладных программ.

Производительность файловой системы

Поскольку обращение к диску - операция относительно медленная, минимизация количества таких обращений - ключевая задача всех алгоритмов, работающих с внешней памятью. Наиболее типичная техника повышения скорости работы с диском - кэширование.

Кэширование

Кэш диска представляет собой буфер в оперативной памяти, содержащий ряд блоков диска (см. рис. 12.12). Если имеется запрос на чтение/запись блока диска, то сначала производится проверка на предмет наличия этого блока в кэше. Если блок в кэше имеется, то запрос удовлетворяется из кэша, в противном случае запрошенный блок считывается в кэш с диска. Сокращение количества дисковых операций оказывается возможным вследствие присущего ОС свойства локальности (о свойстве локальности много говорилось в лекциях, посвященных описанию работы системы управления памятью).

Аккуратная реализация кэширования требует решения нескольких проблем:

- емкость буфера кэша ограничена. Когда блок должен быть загружен в заполненный буфер кэша, возникает проблема замещения блоков, то есть отдельные блоки должны быть удалены из него. Здесь работают те же стратегии и те же FIFO, Second Chance и LRU-алгоритмы замещения, что и при выталкивании страниц памяти.

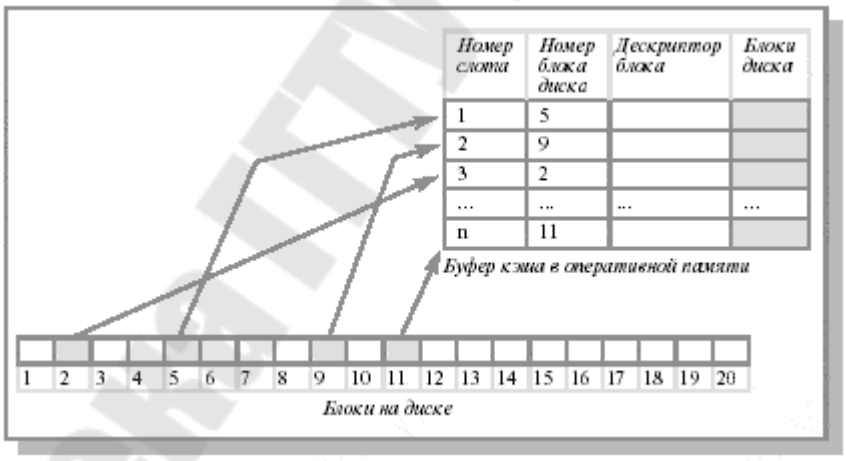

Рис. 12.12. Структура блочного кэша

- замещение блоков должно осуществляться с учетом их важности для файловой системы. Блоки должны быть разделены на категории, например: блоки индексных узлов, блоки косвенной адресации, блоки директорий, заполненные блоки данных и т. д., и в зависимости от принадлежности блока к той или иной категории можно применять к ним разную стратегию замещения.
- поскольку кэширование использует механизм отложенной записи, при котором модификация буфера не вызывает немедленной записи на диск, серьезной проблемой является "старение" информации в дисковых блоках, образы которых находятся в буферном кэше. Несвоевременная синхронизация буфера кэша и диска может привести к очень нежелательным последствиям в случае отказов оборудования или программного обеспечения. Поэтому стратегия и порядок отображения информации из кэша на диск должна быть тщательно продумана. Так, блоки, существенные для совместимости файловой системы (блоки индексных

узлов, блоки косвенной адресации, блоки директорий), должны быть переписаны на диск немедленно, независимо от того, в какой части LRU-цепочки они находятся. Необходимо тщательно выбрать порядок такого переписывания. В Unix имеется для этого вызов SYNC, который заставляет все модифицированные блоки записываться на диск немедленно. Для синхронизации содержимого кэша и диска периодически запускается фоновый процесс-демон. Кроме того, можно организовать синхронный режим работы с отдельными файлами, задаваемый при открытии файла, когда все изменения в файле немедленно сохраняются на диске.

- проблема конкуренции процессов на доступ к блокам кэша решается ведением списков блоков, пребывающих в различных состояниях, и отметкой о состоянии блока в его дескрипторе. Например, блок может быть заблокирован, участвовать в операции ввода-вывода, а также иметь список процессов, ожидающих освобождения данного блока.

Оптимальное размещение информации на диске

Кэширование - не единственный способ увеличения производительности системы. Другая важная техника - сокращение количества движений считывающей головки диска за счет разумной стратегии размещения информации. Например, массив индексных узлов в Unix стараются разместить на средних дорожках. Также имеет смысл размещать индексные узлы поблизости от блоков данных, на которые они ссылаются и т. д.

Кроме того, рекомендуется периодически осуществлять дефрагментацию диска (сборку мусора), поскольку в популярных методиках выделения дисковых блоков (за исключением, может быть, FAT) принцип локальности не работает, и последовательная обработка файла требует обращения к различным участкам диска.

Реализация некоторых операций над файлами

Системные вызовы, работающие с символическим именем файла

Системные вызовы, связывающие pathname с дескриптором файла

Это функции создания и открытия файла. Например, в ОС Unix

```
fd = creat(pathname,modes);
fd = open(pathname,flags,modes);
```

Другие операции над файлами, такие как чтение, запись, позиционирование головок чтения-записи, воспроизведение дескриптора файла, установка параметров ввода-вывода, определение статуса файла и закрытие файла, используют значение полученного дескриптора файла.

Рассмотрим работу системного вызова open.

Логическая файловая подсистема просматривает файловую систему в поисках файла по его имени. Она проверяет права на открытие файла и выделяет открываемому файлу запись в таблице файлов. Запись таблицы файлов содержит указатель на индексный узел открытого файла. Ядро выделяет запись в личной (закрытой) таблице в адресном пространстве процесса, выделенном процессу (таблица эта называется таблицей пользовательских дескрипторов открытых файлов) и запоминает указатель на данную запись. В роли указателя выступает дескриптор файла, возвращаемый пользователю. Запись в таблице пользовательских файлов указывает на запись в глобальной таблице файлов (см. рис. 12.13).

Первые три пользовательских дескриптора (0, 1 и 2) именуются дескрипторами файлов стандартного ввода, стандартного вывода и стандартного файла ошибок. Процессы в системе Unix по договоренности используют дескриптор файла стандартного ввода при чтении вводимой информации, дескриптор файла стандартного вывода при записи выводимой информации и дескриптор стандартного файла ошибок для записи сообщений об ошибках.

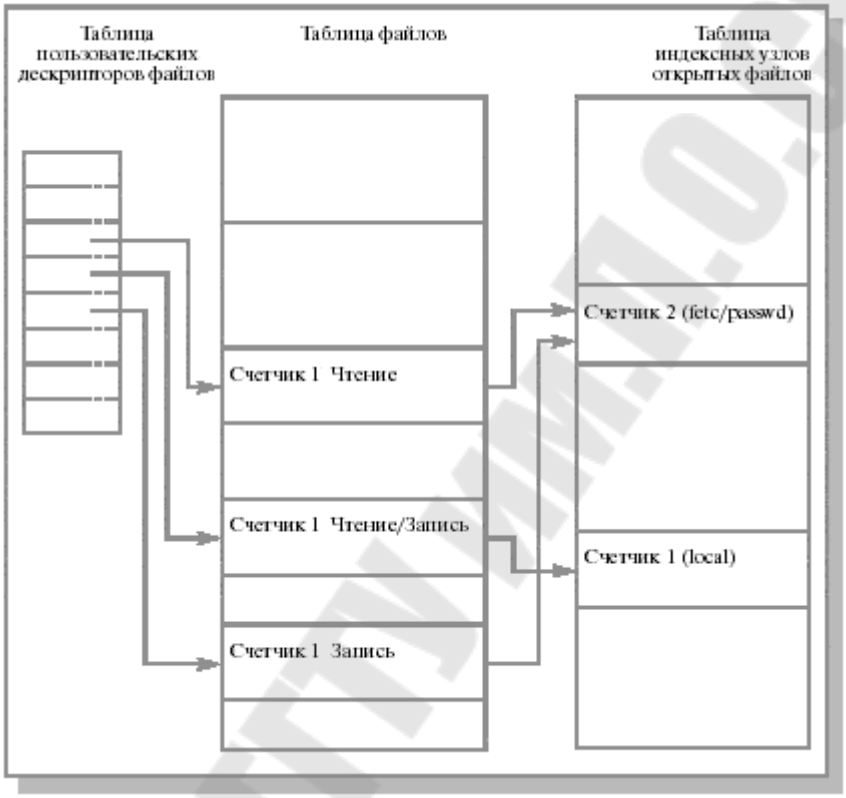

Рис. 12.13. Структуры данных после открытия файлов

Связывание файла

Системная функция link связывает файл с новым именем в структуре каталогов файловой системы, создавая для существующего индекса новую запись в каталоге. Синтаксис вызова функции link:

```
link(source file name, target file name);
```

где source file name - существующее имя файла, a target file name - новое (дополнительное) имя, присваиваемое файлу после выполнения функции link.

Сначала ОС определяет местонахождение индекса исходного файла и увеличивает значение счетчика связей в индексном узле. Затем ядро ищет файл с новым именем; если он существует, функция link завершается неудачно, и ядро восстанавливает прежнее значение счетчика связей, измененное ранее. В противном случае ядро находит в родительском каталоге свободную запись для файла с новым именем, записывает в нее новое имя и номер индекса исходного файла.

Удаление файла

В Unix системная функция unlink удаляет из каталога точку входа для файла. Синтаксис вызова функции unlink:

```
unlink(pathname);
```

Если удаляемое имя является последней связью файла с каким-либо каталогом, ядро в итоге освобождает все информационные блоки файла. Однако если у файла было несколько связей, он остается все еще доступным под другими именами.

Для того чтобы забрать дисковые блоки, ядро в цикле просматривает таблицу содержимого индексного узла, освобождая все блоки прямой адресации немедленно. Что касается блоков косвенной адресации, то ядро освобождает все блоки, появляющиеся на различных уровнях косвенности, рекурсивно, причем в первую очередь освобождаются блоки с меньшим уровнем.

Системные вызовы, работающие с файловым дескриптором

Открытый файл может использоваться для чтения и записи последовательностей байтов. Для этого поддерживаются два системных вызова read и write, работающие с файловым дескриптором (или handle в терминологии Microsoft), полученным при ранее выполненных системных вызовах open или creat.

Функции ввода-вывода из файла

Системный вызов read выполняет чтение обычного файла

```
number = read(fd,buffer,count);
```

где fd - дескриптор файла, возвращаемый функцией open, buffer - адрес структуры данных в пользовательском процессе, где будут размещаться считанные данные в случае успешного завершения выполнения функции read, count - количество байтов, которые пользователю нужно прочитать, number - количество фактически прочитанных байтов.

Синтаксис вызова системной функции write (писать):

```
number = write(fd,buffer,count);
```

где переменные fd, buffer, count и number имеют тот же смысл, что и для вызова системной функции read. Алгоритм записи в обычный файл похож на алгоритм чтения из обычного файла. Однако если в файле отсутствует блок, соответствующий смещению в байтах до места, куда должна производиться запись, ядро выделяет блок и присваивает ему номер в соответствии с точным указанием места в таблице содержимого индексного узла.

Обычное использование системных функций read и write обеспечивает последовательный доступ к файлу, однако процессы могут использовать вызов системной функции lseek для указания места в файле, где будет производиться ввод-вывод, и осуществления произвольного доступа к файлу. Синтаксис вызова системной функции:

```
position = lseek(fd,offset,reference);
```

где fd - дескриптор файла, идентифицирующий файл, offset - смещение в байтах, а reference указывает, является ли значение offset смещением от начала файла, смещением от текущей позиции ввода-вывода или смещением от конца файла. Возвращаемое значение, position, является смещением в байтах до места, где будет начинаться следующая операция чтения или записи.

Современные архитектуры файловых систем

Современные ОС предоставляют пользователю возможность работать сразу с несколькими файловыми системами (Linux работает с Ext2fs, FAT и др.). Файловая система в традиционном понимании становится частью более общей многоуровневой структуры (см. рис. 12.14).

На верхнем уровне располагается так называемый диспетчер файловых систем (например, в Windows 95 этот компонент называется installable filesystem manager). Он связывает запросы прикладной программы с конкретной файловой системой.

Каждая файловая система (иногда говорят - драйвер файловой системы) на этапе инициализации регистрируется у диспетчера, сообщая ему точки входа, для последующих обращений к данной файловой системе.

Та же идея поддержки нескольких файловых систем в рамках одной ОС может быть реализована по-другому, например исходя из концепции виртуальной файловой системы. Виртуальная файловая система (vfs) представляет собой независимый от реализации уровень и опирается на реальные файловые системы (s5fs, ufs, FAT, NFS, FFS. Ext2fs s). При этом возникают структуры данных виртуальной файловой системы типа виртуальных индексных узлов vnode, которые обобщают индексные узлы конкретных систем.

Рис. 12.14. Архитектура современной файловой системы

Процессы и потоки.

Потоки позволяют в рамках одной программы решать несколько задач одновременно. потоки — это объекты, получающие время процессора. Время процессора выделяется квантами (quantum, time slice). Квант времени — это интервал, имеющийся в распоряжении потока до тех пор. пока время не будет передано в распоряжение другого потока. Кванты выделяются не программам или процессам, а порожденным ими потокам. Как минимум, каждый процесс имеет хотя бы один (главный) поток

В архитектуре клиент/сервер, когда требуется обслуживание нового клиента, сервер может запустить специально для этого отдельный поток. Такие потоки принято называть симметричными потоками (symmetric threads) — они имеют одинаковое предназначение, исполняют один и тот же код и могут разделять одни и те же ресурсы. Более того, приложения, рассчитанные на серьезную нагрузку, могут поддерживать пул (pool)

однотипных потоков. Поскольку создание потока требует определенного времени, для ускорения работы желательно заранее иметь нужное число готовых потоков и активизировать их по мере подключения очередного клиента.

Асимметричные потоки (asymmetric threads) — это потоки, решающие различные задачи и, как правило, не разделяющие совместные ресурсы. Необходимость в асимметричных потоках возникает: когда в программе необходимы длительные вычисления, при этом необходимо сохранить нормальную реакцию на ввод; когда нужно обрабатывать асинхронный ввод/вывод с использованием различных устройств (COM-порта, звуковой карты, принтера и т. п.); когда вы хотите создать несколько окон и одновременно обрабатывать ввод в них

Когда мы говорим "программа" (application), то обычно имеем в виду понятие, в терминологии операционной системы обозначаемое как "процесс". Процесс состоит из виртуальной памяти, исполняемого кода, потоков и данных. Процесс может содержать много потоков, но обязательно содержит, по крайней мере, один. Поток, как правило, имеет "в собственности" минимум ресурсов; он зависит от процесса, который и распоряжается виртуальной памятью, кодом, данными, файлами и другими ресурсами ОС.

Почему мы используем потоки вместо процессов, хотя, при необходимости, приложение может состоять и из нескольких процессов? Дело в том, что переключение между процессами — значительно более трудоемкая операция, чем переключение между потоками. Другой довод в пользу использования потоков — то, что они специально задуманы для разделения ресурсов; разделить ресурсы между процессами (имеющими раздельное адресное пространство) не так-то просто.

В Delphi возможность создать **фоновую процедуру** реализована через событие Onldle .объекта Application!

```
type TIdleEvent = procedure (Sender: TObject; var Done: Boolean) of object;
property Onldle: TIdleEvent;
```

Обработчик этого события вы можете написать, поместив на форму компонент TApplicationEvents со страницы Additional Палитры компонентов.

Чтобы сделать в фоновом режиме какую-то работу, следует разбить ее на кванты и выполнять по одному кванту каждый вызов Onldle — иначе приложение будет плохо реагировать на внешние воздействия.

Приоритет потока — величина, складывающаяся из двух составных частей: приоритета породившего поток процесса и собственно приоритета потока. Когда поток создается, ему назначается приоритет, соответствующий приоритету породившего его процесса.

В свою очередь, процессы могут иметь следующие классы приоритетов.

Real time;
Normal;
High;
Below normal;
Above normal;
Idle.

Приоритеты имеют значения от 0 до 31. Процесс, породивший поток, может впоследствии изменить его приоритет; в этой ситуации программист имеет возможность управлять скоростью отклика каждого потока.

Операционная система имеет различные очереди готовых к выполнению потоков — для каждого уровня приоритета свой. В момент распределения нового кванта времени она просматривает очереди — от высшего приоритета к низшему. Готовый к выполнению поток, стоящий первым в очереди, получает этот квант и перемещается в хвост очереди. Поток будет исполняться всю продолжительность кванта, если не произойдет одно из двух событий:

выполняющийся поток остановился для ожидания;

появился готовый к выполнению поток с более высоким приоритетом.

Нрмальная практика для асимметричных потоков — это назначение потоку, обрабатывающему ввод, более высокого приоритета, а всем остальным — более низкого или даже приоритета idle, если этот поток должен выполняться только во время простоя системы.

Класс TThread

Метод Execute определяет исполняемый в создаваемом вами потоке TThread код.

procedure Execute; virtual; abstract;

Мастер создания нового объекта TThread создает для вас пустой шаблон этого метода.

Переопределяя метод Execute, мы можем тем самым закладывать в новый потоковый класс то, что будет выполняться при его запуске. Если поток был создан с аргументом CreateSuspended, равным False, то метод Execute выполняется немедленно, в противном случае Execute выполняется после вызова метода Resume .

Если поток рассчитан на однократное выполнение каких-либо действий, то никакого специального кода завершения внутри Execute писать не надо.

Если же в потоке будет выполняться какой-то цикл, и поток должен завершиться вместе с приложением, то условия окончания цикла должны быть примерно такими:

```
procedure TMyThread.Execute;
begin
repeat
DoSomething;
Until CancelCondition or Terminated;
end;
```

Здесь CancelCondition — ваше личное условие завершения потока (исчерпание данных, окончание вычислений, поступление на вход того или иного символа и т. п.), а свойство Terminated сообщает о завершении потока (это свойство может быть установлено как изнутри потока, так и извне; скорее всего, завершается породивший его процесс).

Конструктор объекта:

constructor Create(CreateSuspended: Boolean);

получает параметр CreateSuspended. Если его значение равно True, вновь созданный поток не начинает выполняться до тех пор, пока не будет сделан вызов метода Resume. В случае, если параметр CreateSuspended имеет значение False, конструктор завершается и только затем поток начинает исполнение.

destructor Destroy; override;

Деструктор Destroy вызывается, когда необходимость в созданном потоке отпадает. Деструктор завершает его и высвобождает все ресурсы, связанные с объектом TThread.

function Terminate: Integer;

Для окончательного завершения потока (без последующего запуска) существует метод Terminate. Но если вы думаете, что этот метод делает какие-то принудительные действия по остановке потока, вы ошибаетесь. Все, что происходит, — это установка свойства

property Terminated: Boolean;

в значение True. Таким образом, Terminate — это указание потоку завершиться, выраженное "в мягкой форме", с возможностью корректно освободить ресурсы. Если вам нужно немедленно завершить поток, используйте функцию Windows API TerminateThread.

Метод Terminate автоматически вызывается и из деструктора объекта. Поток— объект VCL будет дожидаться, пока завершится поток— объект операционной системы. Таким образом, если поток не умеет завершаться корректно, вызов деструктора потенциально может привести к зависанию всей программы.

property FreeOnTerminate: Boolean;

Если это свойство равно True, то деструктор потока будет вызван автоматически по его завершении. Это очень удобно для тех случаев, когда вы в своей программе не уверены точно, когда именно завершится поток, и хотите использовать его по принципу "выстрелил и забыл" (fire and forget).

function WaitFor: Integer;

Метод WaitFor предназначен для синхронизации и позволяет одному потоку дождаться момента, когда завершится другой поток. Если вы внутри потока FirstThread пишите код

Code := SecondThread.WaitFor;

то это означает, что поток FirstThread останавливается до момента завершения потока SecondThread. Метод WaitFor возвращает код завершения ожидаемого потока (см. свойство Returnvalue).

property Handle: THandle read FHandle;

property ThreadID: THandle read FThreadID;

Свойства Handle и ThreadID дают программисту непосредственный доступ к потоку средствами API Win32. Если разработчик хочет обратиться к потоку и управлять им, минуя возможности класса TThread, значения Handle и ThreadID могут быть использованы в качестве аргументов функций Win32 API. Например, если программист хочет перед продолжением выполнения приложения дождаться завершения сразу нескольких потоков,

он должен вызвать функцию API waitForMuitipieObjects; для ее вызова необходим массив дескрипторов потоков.

property Priority: TThreadPriority;

Свойство Priority позволяет запросить и установить приоритет потоков. Приоритеты потоков в деталях описаны выше. Допустимыми значениями приоритета для объектов TThread являются tpidle, tpLowest, tpLower, tpNormai, tpHigher, tpHighest и tpTimeCritical.

procedure Synchronize(Method: TThreadMethod);

Этот метод относится к секции protected, т. е. может быть вызван только из потомков TThread. Delphi предоставляет программисту метод Synchronize для безопасного вызова методов VCL внутри потоков. Во избежание конфликтных ситуаций, метод synchronize дает гарантию, что к каждому объекту VCL одновременно имеет доступ только один поток. Аргумент, передаваемый в метод Synchronize, — это имя метода, который производит обращение к VCL; вызов Synchronize с этим параметром — это то же, что и вызов самого метода. Такой метод (класса TThreadMethod) не должен иметь никаких параметров и не должен возвращать никаких значений. К примеру, в основной форме приложения нужно предусмотреть функцию

procedure TMainForm.SyncShowMessage; begin

ShowMessagedntToStr (ThreadListl. Count)) ; // другие обращения к VCL

end;

а в потоке для показа сообщения писать не

ShowMessage(IntToStr(ThreadListl.Count));

и даже не

MainForm.SyncShowMessage;

а только так:

Synchronize(MainForm.SyncShowMessage);

Примечание

Производя любое обращение к объекту VCL из потока, убедитесь, что при этом используется метод Synchronize; в противном случае результаты могут оказаться непредсказуемыми. Это верно даже в том случае, если вы используете средства синхронизации, описанные ниже.

procedure Resume;

Метод Resume класса TThread вызывается, когда поток возобновляет выполнение после остановки, или для явного запуска потока, созданного с параметром CreateSuspended, равным True.

procedure Suspend;

Вызов метода Suspend приостанавливает поток с возможностью повторного запуска впоследствии. Метод suspend приостанавливает поток вне зависимости от кода, исполняемого потоком в данный момент; выполнение продолжается с точки останова.

property Suspended: Boolean;

Свойство suspended позволяет программисту определить, не приостановлен ли поток. С помощью этого свойства можно также запускать и останавливать поток. Установив свойство suspended в значение True, вы получите тот же результат, что и при вызове метода Suspend — приостановку. Наоборот, установка свойства Suspended в значение False возобновляет выполнение потока, как и вызов метода Resume.

property ReturnValue: Integer;

Свойство ReturnValue позволяет узнать и установить значение, возвращаемое потоком по его завершении. Эта величина полностью определяется пользователем. По умолчанию поток возвращает ноль, но если программист захочет вернуть другую величину, то простая переустановка свойства ReturnValue внутри потока позволит получить эту информацию другим потокам. Это, к примеру, может пригодиться, если внутри потока возникли проблемы, или с помощью свойства ReturnValue нужно вернуть число не прошедших орфографическую проверку слов.

Проблемы при синхронизации потоков — это тупики (deadlocks) и гонки (race conditions)

Тупики имеют место, когда поток ожидает ресурс, который в данный момент принадлежит другому потоку. Рассмотрим пример. Поток 1 захватывает ресурс А, и для того чтобы продолжать работу, ждет возможности захватить ресурс Б. В то же время Поток 2 захватывает ресурс Б и ждет возможности захватить ресурс А. Развитие этого сценария заблокирует оба потока; ни один из них не будет исполняться. Ресурсами могут выступать любые совместно используемые объекты системы — файлы, массивы в памяти, устройства ввода/вывода и т. п.

Ситуация гонок возникает, когда два или более потока пытаются получить доступ к общему ресурсу и изменить его состояние. Рассмотрим следующий пример. Пусть Поток 1 получил доступ к ресурсу и изменил его в своих интересах; затем активизировался Поток 2 и модифицировал этот же ресурс до завершения Потока 1. Поток 1 полагает, что ресурс остался в том же состоянии, в каком был до переключения. В зависимости от того, когда именно был изменен ресурс, результаты могут варьироваться — иногда код будет выполняться нормально, иногда нет. Программисты не должны строить никаких гипотез относительно порядка исполнения потоков, т. к. планировщик ОС может запускать и останавливать их в любое время.

Средства синхронизации потоков

Проще всего говорить о синхронизации, если создаваемый поток не взаимодействует с ресурсами других потоков и не обращается к VCL. Допустим, у вас на компьютере несколько процессоров, и вы хотите "распараллелить" вычисления. Тогда вполне уместен следующий код:

MyCompThread := TComputationThread.Create(False);

// Здесь можно что-нибудь делать, пока второй поток производит вычисления

DoSomeWork;

// Теперь ожидаем его завершения

MyCompThread.WaitFor;

Приведенная схема совершенно недопустима, если во время своей работы поток MyCompThread обращается к VCL посредством метода synchronize. В этом случае поток ждет главный поток для обращения к VCL, а тот, в свою очередь, его — классический тупик.

Главные понятия для понимания механизмов синхронизации — функции ожидания и объекты синхронизации. В Windows API предусмотрен ряд функций, позволяющих приостановить выполнение вызвавшего эту функцию потока вплоть до того момента, как будет изменено состояние какого-то объекта, называемого объектом синхронизации (под этим термином здесь понимается не объект Delphi, а объект операционной системы). Простейшая из этих функций — waitForSingieCbject — предназначена для ожидания одного объекта.

К возможным вариантам относятся четыре объекта, которые разработаны специально для синхронизации: событие (event), взаимное исключение (mutex), семафор (semaphore) и таймер (timer).

Но кроме специальных объектов можно организовать ожидание и других объектов, дескриптор которых используется в основном для иных целей, но может применяться и для ожидания. К ним относятся: процесс (process), поток (thread), оповещение об изменении в файловой системе (change notification) и консольный ввод (console input).

Косвенно к этой группе может быть добавлена критическая секция (critical section).

Перечисленные выше средства синхронизации в основном инкапсулированы в состав классов Delphi. У программиста есть две альтернативы. С одной стороны, в состав библиотеки VCL включен модуль SYNCOBJS.PAS, содержащий классы для события (TEvent) и критической секции (TCriticalSection). С другой, с Delphi поставляется отличный пример IPCDEMOS, который иллюстрирует проблемы взаимодействия процессов и содержит модуль IPCTHRD.PAS с аналогичными классами — для того же события, взаимного исключения (TMutex), а также совместно используемой памяти (TSharedMem).

Событие

Класс TEvent (модуль SYNCOBJS.PAS) имеет два метода: setEvent и ResetEvent, которые переводят объект в активное и пассивное состояние соответственно. Конструктор имеет следующий вид:

constructor Create(EventAttributes: PSecurityAttributes;

ManualReset, InitialState: Boolean; const Name: string);

Здесь параметр initialstate — начальное состояние объекта, ManualReset — способ его сброса (перевода в пассивное состояние). Если этот параметр равен True, событие должно быть сброшено вручную. В противном случае событие сбрасывается по мере того, как стартует хоть один поток, ждавший данный объект.

TWaitResult = (wrSignaled, wrTimeout, wrAbandoned, wrError);

function WaitFor(Timeout: DWORD): TWaitResult;

Он дает возможность ожидать активизации события в течение Timeout миллисекунд. Как вы могли догадаться, внутри этого метода происходит вызов функции waitFotsingieObject. Типичных результатов на выходе waitFor два — wrsignaied, если произошла активизация события, и wrTimeout, если за время тайм-аута ничего не произошло. Если нужно (и допустимо!) ждать бесконечно долго, следует установить параметр Timeout в значение INFINITE.

Взаимные исключения

Объект типа взаимное исключение (mutex) позволяет только одному потоку в данное время владеть им. Если продолжать аналогии, то этот объект можно сравнить с эстафетной палочкой.

Класс, инкапсулирующий взаимное исключение, — TMutex — находится в модуле IPCTHRD.PAS (пример IPCDEMOS). Конструктор:

constructor Create (const Name: string);

задает имя создаваемого объекта. Первоначально он не принадлежит никому. (Но функция API createMutex, вызываемая в нем, позволяет передать созданный объект тому потоку, в котором это произошло.) Далее метод

function Get(TimeOut: Integer): Boolean;

производит попытку в течение Timeout миллисекунд завладеть объектом (в этом случае результат равен True). Если объект более не нужен, следует вызвать метод

function Release: Boolean;

Программист может использовать взаимное исключение, чтобы избежать считывания и записи общей памяти несколькими потоками одновременно.

Семафор

Семафор (semaphore) подобен взаимному исключению. Разница между ними в том, что семафор может управлять количеством потоков, которые имеют к нему доступ. Семафор устанавливается на предельное число потоков, которым доступ разрешен. Когда это число достигнуто, последующие потоки будут приостановлены, пока один или более потоков не отсоединятся от семафора и не освободят доступ.

Критическая секция

Работая в Delphi, программист может также использовать объект типа критическая секция (critical section). Критические секции подобны взаимным исключениям по сути, однако между ними существуют два главных отличия:

взаимные исключения могут быть совместно использованы потоками в различных процессах, а критические секции — нет;

если критическая секция принадлежит другому потоку, ожидающий поток блокируется вплоть до освобождения критической секции. В отличие от этого, взаимное исключение разрешает продолжение по истечении тайм-аута.

Критические секции и взаимные исключения очень схожи. На первый взгляд, выигрыш от использования критической секции вместо взаимного исключения не очевиден. Критические секции, однако, более эффективны, чем взаимные исключения, т. к. используют меньше системных ресурсов. Взаимные исключения могут быть установлены на определенный интервал времени, по истечении которого выполнение продолжается; критическая секция всегда ждет столько, сколько потребуется.

Возьмем класс TCriticalSection (модуль SYNCOBJS.PAS). Логика использования его проста — "держать и не пущать". В многопотоковом приложении создается и инициализируется общая для всех потоков критическая секция. Когда один из потоков достигает критически важного участка кода, он пытается захватить секцию вызовом метода Enter:

```
MySection.Enter; try DoSomethingCritical;
finally
MySection.Leave;
end;
```

Когда другие потоки доходят до оператора захвата секции Enter и обнаруживают, что она уже захвачена, они приостанавливаются вплоть до освобождения секции первым потоком путем вызова метода Leave. Обратите внимание, что вызов Leave помещен в конструкцию try. .finally — здесь требуется стопроцентная надежность. Критические секции являются системными объектами и подлежат обязательному освобождению — впрочем, как и остальные рассматриваемые здесь объекты.

Локальные данные потока

Интересная проблема возникает, если в приложении будет несколько одинаковых потоков. Как избежать совместного использования одних и тех же переменных несколькими потоками? Первое, что приходит на ум, — добавить и использовать поля объекта — потомка TThread, которые можно добавить при его создании. Каждый поток соответствует отдельному экземпляру объекта, и их данные пересекаться не будут. (Кстати, это одно из больших удобств использования класса TThread.) Но есть функции API, которые знать не знают об объектах Delphi и их полях и свойствах. Для поддержки разделения данных между потоками на нижнем уровне в язык Object Pascal введена специальная директива — threadvar, которая отличается от директивы описания переменных var тем, что применяется только к локальным данным потока. Следующее описание:

```
Var
datal: Integer; threadvar
data2: Integer;
```

означает, что переменная datal будет использоваться всеми потоками данного приложения, а переменная data2 будет у каждого потока своя.

Алгоритм

1. В среде Delphi откройте меню File и выберите пункт New Application.
2. Расположите на форме необходимые компоненты
3. Сохраняем проект
4. Откройте меню File и выберите пункт New. Затем дважды щелкните на объекте типа поток (значок Thread Object). Откроется диалоговое окно New Items.
5. В диалоговом окне вводим имя объекта поток. Помимо этого, при желании, можно присвоить создаваемому потоку имя, установив флажок Named Thread и задав имя в поле Thread Name. Delphi создаст новый модуль и поместит в него шаблон для нового потока.
6. В метод Execute пишем код для создаваемого потока
7. Сохраняем модуль с потоком.
8. Подключаем к списку используемых модулей в секции интерфейса главной формы модуль с потоком.
9. В секции public главной формы добавляем ссылку на создаваемый поток
10. В обработчике какого-либо события пишем код, создающий вычислительный поток:

```
TestThread := TTestThread.Create(True);
TestThread.FreeOnTerminate := True;
TestThread.Priority := tpLower;
TestThread.Resume;
```

11. В обработчике какого-либо события пишем код, уничтожающий вычислительный поток:

```
if Assigned(TestThread) then TestThread.Terminate;
```

12.

Модуль 5. Безопасность ОС

Лекции 2ч.
Тема 15. Методы и защитные механизмы ОС