КАСИМОВА Ш.Т.

ОПЕРАЦИОННЫЕ СИСТЕМЫ

МИНИСТЕРСТВО ЦИФРОВЫХ ТЕХНОЛОГИЙ
РЕСПУБЛИКИ УЗБЕКИСТАН

ТАШКЕНТСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ
ТЕХНОЛОГИЙ ИМЕНИ МУХАММАДА АЛ-ХОРАЗМИЙ

КАСИМОВА Ш.Т.

ОПЕРАЦИОННЫЕ СИСТЕМЫ

УЧЕБНОЕ ПОСОБИЕ

Ташкент
"METODIST NASHRIYOTI"
2024

УДК: 004.451(075.8)
ББК: 32.973я7
К 281

Касимова Ш.Т.
Операционные системы. Учебное пособие. – Ташкент: "METODIST NASHRIYOTI", 2024. – 308 стр.

Операционная система (ОС) — это программный комплекс, одной из важнейших задач которого является предоставление пользователю возможности использовать ресурсы компьютера по своему усмотрению в максимально доступном объеме, не отвлекаясь на проблемы управления аппаратными ресурсами, находящиеся за гранью его возможностей.

Операционная система является обязательным компонентом любой вычислительной машины, какие бы задачи перед ней ни стояли — будь то домашний компьютер, узел локальной или глобальной компьютерной сети, сервер баз данных или же комплекс управления технологическим процессом на промышленном предприятии (хотя надо учесть, что как раз на производстве могут использоваться не полноценные компьютеры, а микроконтроллеры без ОС).

Операционная система должна быть достаточно прозрачной для разработчиков программного обеспечения, дабы те могли разрабатывать приложения для расширения функционала ОС и улучшения ее работы.

Представляемое учебное пособие предусматривает получение знаний по эволюции ОС, классификации операционных систем, составу и функционированию ОС, управлению локальными ресурсами, управлению распределенными ресурсами, представлен сравнительный обзор ОС и эволюция пользовательских интерфейсов ОС.

Рецензенты:
К. Разманов
Заведующий кафедрой "Современные информационно-коммуникационные технологии" Международной исламской академии Узбекистана, к.т.н.
С.С. Бекназарова
Заведующая кафедрой АВТ ФТТ проф., д.т.н.

Публикация разрешена на основании приказа Ташкентского университета информационных технологий имени Мухаммада аль-Хоразми от 10 мая 2023 года № 500.

ISBN 978-9910-03-167-0

ВВЕДЕНИЕ

Перед тем как приступать к проектированию операционной системы(ОС), необходимо составить четкое представление о том, для каких задач она будет создана. От степени универсализации будущей операционной системы зависит сложность ее проектирования. Например, операционная система для управления промышленным станком на несколько порядков проще, чем для карманного или настольного компьютера, и тем более — чем для сервера локальной сети или вычислительной сети в целом.

В первом случае операционная система разрабатывается под конкретную аппаратную платформу, в рамках которой будет решать одну задачу. Для таких операционных систем характерен чрезвычайно малый размер ядра и очень высокое быстродействие. Такие системы разрабатывают однозадачными, и задача, для управления ходом решения которой разрабатывается операционная система, решается в режиме реального времени.

Во втором случае операционная система разрабатывается под одну или несколько концепций аппаратных платформ, конкретные реализации которых могут быть выполнены из различных по архитектуре и исполнению компонентов. Таким системам свойственны большие объемы дистрибутивов, поддержка большого количества различных аппаратных модулей, многозадачность, возможность работы в сетях передачи данных и многое другое.

Для универсальных операционных систем характерны следующие четыре цели:

- определение абстракций — процессы, файлы, модели памяти, концепция ввода-вывода и многое другое;
- предоставление примитивных команд для работы с абстракциями;
- защита — как сеансов пользователей, так и вычислительных ресурсов;
- управление аппаратными компонентами.

Сложность проектирования операционных систем зависит и от ряда человеческих факторов. Например, разработчики операционных систем стремятся сохранить обратную совместимость с предыдущими версиями. Если в пределах соседних версий это еще возможно, так как смена поколений операционных систем происходит реже, чем смена поколений вычислительных машин, то для версий, работающих на разных поколениях вычислительных машин, сохранить совместимость уже гораздо сложнее. При смене аппаратного поколения происходит изменение концепций разработки прикладных программ, и зачастую возникает ситуация, когда приложение, написанное для работы на предыдущем аппаратном поколении, плохо работает или не работает вовсе на новой аппаратной платформе.

Вторая проблема заключается в том, что над разработкой операционной системы трудится большое количество людей. И предсказать, как тот или иной компонент системы поведет себя при взаимодействии с другим компонентом, зачастую невозможно.

Третья сложность в том, что система должна обеспечивать параллелизм работы нескольких пользователей и множества устройств. Кроме этого, операционная система должна защищать данные пользователей от несанкционированного доступа таким образом, чтобы не мешать работе с данными их владельцам.

ГЛАВА 1. ВВЕДЕНИЕ В ОС

1.1. ЭВОЛЮЦИЯ ОПЕРАЦИОННЫХ СИСТЕМ

1.1.1. Этапы развития вычислительной техники

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 новые 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[5,6].

Первый период (1945-1955). Ламповые машины. Операционные системы отсутствовали.

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

Первые шаги по созданию электронных вычислительных машин были предприняты в конце второй мировой войны. В середине 40-х были созданы первые ламповые вычислительные устройства, и появился принцип программы, хранимой в памяти машины (John Von Neumann, июнь 1945г). 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, собственно вычисления, размышления программиста). Отладка программ велась с пульта управления с помощью изучения состояния памяти и регистров машины. В конце этого периода появляется первое системное программное обеспечение: в 1951-52 гг. возникают прообразы первых компиляторов с символических языков (Fortran и др.), а в 1954 г. Nat Rochester разрабатывает ассемблер для IBM-701. В целом первый период характеризуется крайне высокой стоимостью вычислительных систем, их малым количеством и низкой эффективностью использования.

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

С середины 50-х годов начался новый 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 они смогли непрерывно работать настолько долго, чтобы на них можно было возложить выполнение действительно практически важных задач. Снизилось потребление вычислительными машинами электроэнергии. Проще стали системы охлаждения. Размеры компьютеров уменьшились. Эксплуатация и обслуживание вычислительной техники подешевели. Началось использование ЭВМ коммерческими фирмами. Одновременно наблюдается бурное развитие алгоритмических языков (ALGOL-58, LISP, COBOL, ALGOL-60, PL-1 и т.д.). Появляются первые настоящие компиляторы, редакторы связей, библиотеки математических и служебных подпрограмм. Упрощается процесс программирования. Пропадает необходимости 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 требуемые для нее 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 большое) время.

Смена запрошенных ресурсов вызывает приостановку выполнения программ. В результате процессор часто простаивает. Для повышения эффективности использования компьютера задания с похожими требуемыми 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 явились прообразом современных операционных систем, они стали первыми системными программами, предназначенными для управления вычислительным процессом[11-15].

Третий период (Начало 60-х - 1980). Компьютеры на основе интегральных микросхем. Первые многозадачные ОС.

Следующий важный период развития вычислительных машин относится к началу 60-х - 1980 годам. 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 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 заданию требуется ввод данных, они читаются с диска. Точно так же выходная информация сначала копируется в системный буфер и записывается на ленту или диск, а реально 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 появления 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 появляется 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 поставленный вопрос (операция ввода-вывода), преподаватель (процессор) выслушивает ответ другого студента (вычисления). Естественно, что такая ситуация требует наличия в комнате нескольких студентов. Точно также 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 целой революции в строении вычислительной системы. Большую роль, здесь играет аппаратная поддержка, наиболее существенные особенности которой:

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. Во-вторых, 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[3,4].

Не менее важна в организации мультипрограммирования роль операционной системы. Наиболее существенные изменения состояли в следующем:

Интерфейс между прикладной программой и ОС был организован при помощи набора системных вызовов.

Организация очереди из заданий в памяти и выделение процессора одному из заданий потребовали планирования заданий.

Для переключения процессора с одного задания на другое возникла потребность в сохранении содержимого регистров и структур данных, необходимых для выполнения задания, иначе говоря, контекста, для обеспечения правильного продолжения вычислений.

Поскольку память является ограниченным ресурсом, оказались нужны стратегии управления памятью, то есть потребовалось упорядочить процессы размещения, замещения и выборки информации из памяти.

Так как программы могут пожелать произвести санкционированный обмен данными, стало необходимо их обеспечить средствами коммуникации. И, наконец, для корректного обмена данными необходимо предусмотреть координацию программами своих действий, т.е. средства синхронизации.

Мультипрограммные пакетные системы дают окружение, в котором различные системные ресурсы (например, процессор, память, периферийные устройства) используются эффективно. И все же 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 электронолучевых дисплеев и переосмысление возможностей применения клавиатур поставили на очередь решение этой проблемы. Логическим расширением систем мультипрограммирования стали time-sharing системы или системы разделения времени **. В них процессор переключается между задачами не только на время операций ввода-вывода, но и

просто по прошествии определенного интервала времени. Эти переключения происходят столь часто, что пользователи могут взаимодействовать со своими программами во время их выполнения, то есть интерактивно. В результате появляется возможность одновременной работы многих 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 необходимый для ее дальнейшего выполнения кусок может быть легко 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 легко и эффективно вести отладку своей программы в интерактивном режиме, 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 в этот период наблюдается и внешняя их эволюция. До начала этого периода вычислительные комплексы были, как правило, несовместимы. Каждый имела свою собственную специальную операционную систему, свою систему команд и т.д. В результате программу, успешно работающую на одном типе машин, необходимо было полностью переписать и заново отладить для другого типа компьютеров. В начале третьего периода появилась идея создания семейств программно-совместимых машин, работающих под управлением одной и той же операционной системы. Первым семейством программно-совместимых машин, построенных на интегральных микросхемах, явилась серия машин IBM/360. Построенное в начале 60-х годов это семейство значительно превосходило машины второго поколения по критерию цена/производительность. За ней последовала линия

компьютеров PDP, несовместимых с линией IBM, кульминацией которой стала PDP-11.

Сила одной семьи была одновременно и ее слабостью. Широкие возможности этой концепции (наличие всех моделей: от миникомпьютеров до гигантских машин; обилие разнообразной периферии; различное окружение; различные пользователи) порождали сложную и огромную операционную систему. Миллионы строчек ассемблера, написанные тысячами программистов, содержали множество ошибок, что вызывало непрерывный поток публикаций о них и попыток их 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 (1980-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 удешевление 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 широко использоваться неспециалистами, что потребовало разработки "дружественного" программного обеспечения, это положило конец кастовости программистов.

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[1,2].

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, при необходимости воспользоваться ресурсами другого сетевого компьютера, должны знать о его 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 существенно 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.

Что мы вынесли из истории развития вычислительных систем?

Просмотрев этапы развития вычислительных систем, мы можем выделить пять основных функций, которые выполняли классические операционные системы в процессе своей 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 конкретной операционной системе эти функции, конечно, реализовывались по-своему, в различном объеме. Они не были придуманы как составные части деятельности операционных систем изначально, а появились в процессе развития, по мере того, как вычислительные системы становились удобнее, эффективнее и безопаснее. 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 настоящий момент их существование - это разумный способ использования вычислительных систем.

1.1.2. Посылки возникновения ОС

Первый период (1945 -1955). В середине 40-х были созданы первые ламповые вычислительные устройства. 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 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. Не было никакого другого системного программного обеспечения, кроме библиотек математических и служебных подпрограмм[10-12].

Второй период (1955 - 1965). С середины 50-х годов начался новый период в развитии вычислительной техники, связанный с появлением новой технической базы – полупроводниковых элементов. Компьютеры второго поколения стали более надежными, теперь они смогли непрерывно работать настолько долго, чтобы на них можно было возложить выполнение действительно практически важных задач. Именно в этот период

произошло разделение персонала на программистов и операторов, эксплуатационников и разработчиков вычислительных машин.

В эти годы появились первые алгоритмические языки, а следовательно и первые системные программы – компиляторы. Стоимость процессорного времени возросла, что потребовало уменьшения непроизводительных затрат времени между запусками программ. Появились первые системы пакетной обработки, которые просто автоматизировали запуск одной программ за другой и тем самым увеличивали коэффициент загрузки процессора. Системы пакетной обработки явились прообразом современных операционных систем, они стали первыми системными программами, предназначенными для управления вычислительным процессом. В ходе 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. Совокупность нескольких заданий, как правило, в виде колоды перфокарт, получила название пакета заданий.

Третий период (1965 - 1980). Следующий важный период развития вычислительных машин относится к 1965-1980 годам. В это время в технической базе произошел переход от отдельных полупроводниковых элементов типа транзисторов к интегральным микросхемам, что дало гораздо большие возможности новому, третьему поколению компьютеров.

Для этого периода характерно также создание семейств программно-совместимых машин. Первым семейством программно-совместимых машин, построенных на интегральных микросхемах, явилась серия машин IBM/360. Построенное в начале 60-х годов это семейство значительно превосходило машины второго поколения по критерию «цена-производительность». Вскоре идея программно-совместимых машин стала общепризнанной.

Программная совместимость требовала и совместимости операционных систем. Такие операционные системы должны были бы работать и на больших, и на малых вычислительных

- Обеспечение безопасности

Каждая из приведенных функций обычно реализована в виде подсистемы, являющейся структурным компонентом ОС. В каждой конкретной операционной системе эти функции, конечно, реализовывались по-своему, в различном объеме. Они не были придуманы как составные части деятельности операционных систем изначально, а появились в процессе развития, по мере того, как вычислительные системы становились удобнее, эффективнее и безопаснее. 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 настоящий момент их существование - это разумный способ использования вычислительных систем.

1.1.2. Посылки возникновения ОС

Первый период (1945 -1955). В середине 40-х были созданы первые ламповые вычислительные устройства. 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 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. Не было никакого другого системного программного обеспечения, кроме библиотек математических и служебных подпрограмм[10-12].

Второй период (1955 - 1965). С середины 50-х годов начался новый период в развитии вычислительной техники, связанный с появлением новой технической базы – полупроводниковых элементов. Компьютеры второго поколения стали более надежными, теперь они смогли непрерывно работать настолько долго, чтобы на них можно было возложить выполнение действительно практически важных задач. Именно в этот период

произошло разделение персонала на программистов и операторов, эксплуатационников и разработчиков вычислительных машин.

В эти годы появились первые алгоритмические языки, а следовательно и первые системные программы – компиляторы. Стоимость процессорного времени возросла, что потребовало уменьшения непроизводительных затрат времени между запусками программ. Появились первые системы пакетной обработки, которые просто автоматизировали запуск одной программ за другой и тем самым увеличивали коэффициент загрузки процессора. Системы пакетной обработки явились прообразом современных операционных систем, они стали первыми системными программами, предназначенными для управления вычислительным процессом. В ходе 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. Совокупность нескольких заданий, как правило, в виде колоды перфокарт, получила название пакета заданий.

Третий период (1965 - 1980). Следующий важный период развития вычислительных машин относится к 1965-1980 годам. В это время в технической базе произошел переход от отдельных полупроводниковых элементов типа транзисторов к интегральным микросхемам, что дало гораздо большие возможности новому, третьему поколению компьютеров.

Для этого периода характерно также создание семейств программно-совместимых машин. Первым семейством программно-совместимых машин, построенных на интегральных микросхемах, явилась серия машин IBM/360. Построенное в начале 60-х годов это семейство значительно превосходило машины второго поколения по критерию «цена-производительность». Вскоре идея программно-совместимых машин стала общепризнанной.

Программная совместимость требовала и совместимости операционных систем. Такие операционные системы должны были бы работать и на больших, и на малых вычислительных

системах, с большим и с малым количеством разнообразной периферии, в коммерческой области и в области научных исследований. Операционные системы, построенные с намерением удовлетворить всем этим противоречивым требованиям, оказались чрезвычайно сложными и громоздкими. Важнейшим достижением ОС данного поколения явилась реализация мультипрограммирования. *Мультипрограммирование* – это способ организации вычислительного процесса, при котором на одном процессоре попеременно выполняются несколько программ. Пока одна программа выполняет операцию ввода-вывода, процессор не простаивает, как это происходило при последовательном выполнении программ (однопрограммный режим), а выполняет другую программу (многопрограммный режим). При этом каждая программа загружается в свой участок оперативной памяти, называемый разделом.

Другое нововведение – спулинг (spooling). *Спулинг* в то время определялся как способ организации вычислительного процесса, в соответствии с которым задания считывались с перфокарт на диск в том темпе, в котором они появлялись в помещении вычислительного центра, а затем, когда очередное задание завершалось, новое задание с диска загружалось в освободившийся раздел.

Наряду с мультипрограммной реализацией систем пакетной обработки появился новый тип ОС – системы разделения времени. Вариант мультипрограммирования, применяемый в системах разделения времени, нацелен на создание для каждого отдельного пользователя иллюзии единоличного использования вычислительной машины.

Четвертый период (1980 - настоящее время). Следующий период в эволюции операционных систем связан с появлением больших интегральных схем (БИС). В эти годы произошло резкое возрастание степени интеграции и удешевление микросхем. Компьютер стал доступен отдельному человеку. С точки зрения архитектуры персональные компьютеры ничем не отличались от класса миникомпьютеров типа PDP-11.

Компьютеры стали широко использоваться неспециалистами, что потребовало разработки «дружественного» (понятного) программного обеспечения.

На рынке операционных систем доминировали две системы: MS-DOS и UNIX. Однопрограммная однопользовательская ОС MS-DOS широко использовалась для компьютеров, построенных на базе микропроцессоров Intel 8088, а затем 80286, 80386 и 80486. Мультипрограммная многопользовательская ОС UNIX доминировала в среде «неинтеловских» компьютеров, особенно построенных на базе высокопроизводительных RISC-процессоров.

В середине 80-х стали бурно развиваться сети персональных компьютеров, работающие под управлением сетевых или распределенных ОС.

В сетевых ОС пользователи должны быть осведомлены о наличии других компьютеров и должны делать логический вход в другой компьютер, чтобы воспользоваться его ресурсами, преимущественно файлами. Каждая машина в сети содержит свою собственную локальную операционную систему, отличающуюся от ОС автономного компьютера наличием дополнительных средств, позволяющих компьютеру работать в сети. Сетевая ОС не имеет фундаментальных отличий от ОС однопроцессорного компьютера, но она обязательно содержит программную поддержку для сетевых интерфейсных устройств (драйвер сетевого адаптера), а также средства для удаленного входа в другие компьютеры сети и средства доступа к удаленным файлам, однако эти дополнения существенно не меняют структуру самой операционной системы.

1.1.3. Определение операционной системы

Операционная система (ОС) в наибольшей степени определяет облик всей вычислительной системы в целом. ОС выполняет две по существу мало связанные функции: обеспечение пользователю-программисту удобств посредством предоставления для него расширенной машины и повышение эффективности использования компьютера путем рационального управления его ресурсами[1-4].

ОС как виртуальная (расширенная) машина. Использование большинства компьютеров на уровне машинного языка затруднительно, особенно это касается ввода-вывода. (Например,

системах, с большим и с малым количеством разнообразной периферии, в коммерческой области и в области научных исследований. Операционные системы, построенные с намерением удовлетворить всем этим противоречивым требованиям, оказались чрезвычайно сложными и громоздкими. Важнейшим достижением ОС данного поколения явилась реализация мультипрограммирования. *Мультипрограммирование* – это способ организации вычислительного процесса, при котором на одном процессоре попеременно выполняются несколько программ. Пока одна программа выполняет операцию ввода-вывода, процессор не простаивает, как это происходило при последовательном выполнении программ (однопрограммный режим), а выполняет другую программу (многопрограммный режим). При этом каждая программа загружается в свой участок оперативной памяти, называемый разделом.

Другое нововведение – спулинг (spooling). *Спулинг* в то время определялся как способ организации вычислительного процесса, в соответствии с которым задания считывались с перфокарт на диск в том темпе, в котором они появлялись в помещении вычислительного центра, а затем, когда очередное задание завершалось, новое задание с диска загружалось в освободившийся раздел.

Наряду с мультипрограммной реализацией систем пакетной обработки появился новый тип ОС – системы разделения времени. Вариант мультипрограммирования, применяемый в системах разделения времени, нацелен на создание для каждого отдельного пользователя иллюзии единоличного использования вычислительной машины.

Четвертый период (1980 - настоящее время). Следующий период в эволюции операционных систем связан с появлением больших интегральных схем (БИС). В эти годы произошло резкое возрастание степени интеграции и удешевление микросхем. Компьютер стал доступен отдельному человеку. С точки зрения архитектуры персональные компьютеры ничем не отличались от класса миникомпьютеров типа PDP-11.

Компьютеры стали широко использоваться неспециалистами, что потребовало разработки «дружественного» (понятного) программного обеспечения.

На рынке операционных систем доминировали две системы: MS-DOS и UNIX. Однопрограммная однопользовательская ОС MS-DOS широко использовалась для компьютеров, построенных на базе микропроцессоров Intel 8088, а затем 80286, 80386 и 80486. Мультипрограммная многопользовательская ОС UNIX доминировала в среде «неинтеловских» компьютеров, особенно построенных на базе высокопроизводительных RISC-процессоров.

В середине 80-х стали бурно развиваться сети персональных компьютеров, работающие под управлением сетевых или распределенных ОС.

В сетевых ОС пользователи должны быть осведомлены о наличии других компьютеров и должны делать логический вход в другой компьютер, чтобы воспользоваться его ресурсами, преимущественно файлами. Каждая машина в сети содержит свою собственную локальную операционную систему, отличающуюся от ОС автономного компьютера наличием дополнительных средств, позволяющих компьютеру работать в сети. Сетевая ОС не имеет фундаментальных отличий от ОС однопроцессорного компьютера, но она обязательно содержит программную поддержку для сетевых интерфейсных устройств (драйвер сетевого адаптера), а также средства для удаленного входа в другие компьютеры сети и средства доступа к удаленным файлам, однако эти дополнения существенно не меняют структуру самой операционной системы.

1.1.3. Определение операционной системы

Операционная система (ОС) в наибольшей степени определяет облик всей вычислительной системы в целом. ОС выполняет две по существу мало связанные функции: обеспечение пользователю-программисту удобств посредством предоставления для него расширенной машины и повышение эффективности использования компьютера путем рационального управления его ресурсами[1–4].

ОС как виртуальная (расширенная) машина. Использование большинства компьютеров на уровне машинного языка затруднительно, особенно это касается ввода-вывода. (Например,

для организации чтения блока данных с гибкого диска программист может использовать 16 различных команд, каждая из которых требует 13 параметров, таких как номер блока на диске, номер сектора на дорожке и т. п. Когда выполнение операции с диском завершается, контроллер возвращает 23 значения, отражающих наличие и типы ошибок, которые, очевидно, надо анализировать.) При работе с диском программисту-пользователю достаточно представлять его в виде некоторого набора файлов, каждый из которых имеет имя. Работа с файлом заключается в его открытии, выполнении чтения или записи, а затем в закрытии файла. Вопросы подобные тому, в каком состоянии сейчас находится двигатель механизма перемещения головок, не должны волновать пользователя. Программа, которая скрывает от программиста все реалии аппаратуры и предоставляет возможность простого, удобного просмотра указанных файлов, чтения или записи – это операционная система. Операционная система также берет на себя такие функции как обработка прерываний, управление таймерами и оперативной памятью и т.д.

С точки зрения пользователя функцией ОС является предоставление пользователю некоторой расширенной или виртуальной машины, которую легче программировать и с которой легче работать, чем непосредственно с аппаратурой, составляющей реальную машину, тем самым скрывая от пользователя детали управления оборудованием (hardware)

Этот принцип иллюстрируется рис.1.1.

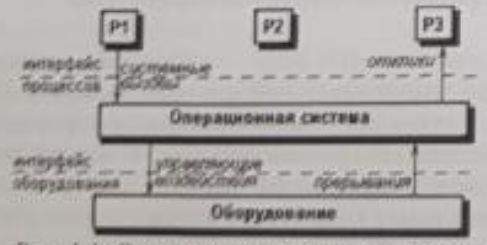


Рис. 1.1. Операционная система, процессы, оборудование

Как видно из рисунка, ОС играет роль "прослойки" между процессами пользователей и оборудованием системы. (Под оборудованием понимаются, как правило, внешние устройства, но можно трактовать этот термин и шире - включая в него все первичные ресурсы). Процессы пользователей не имеют непосредственного доступа к оборудованию и, говоря шире, к системным ресурсам. Если процессу необходимо выполнить операцию с системным ресурсом, в том числе, и с оборудованием, процесс выдает системный вызов. ОС интерпретирует системный вызов, проверяет его корректность, возможно, помещает в очередь запросов и выполняет его. Если выполнение вызова связано с операциями на оборудовании, ОС формирует и выдает на оборудование требуемые управляющие воздействия. Оборудование, выполнив операцию, заданную управляющими воздействиями, сигнализирует об этом прерыванием. Прерывание поступает в ядро ОС, которое анализирует его и формирует отклик для процесса, выдавшего системный вызов. Если выполнение системного вызова не требует операций на оборудовании, отклик может быть сформирован немедленно.

Управляющие воздействия и прерывания составляют интерфейс оборудования, системные вызовы и отклики на них - интерфейс процессов. В качестве синонима интерфейса процессов мы в соответствии со сложившейся в последнее время традицией часто будем употреблять аббревиатуру API (Application Programm Interface - интерфейс прикладной программы).

Отделение процессов пользователя от оборудования преследует две цели.

Во-первых, - безопасность. Если пользователь не имеет прямого доступа к оборудованию и вообще к системным ресурсам, то он не может вывести их из строя или монопольно использовать в ущерб другим пользователям. Обеспечение этой цели нуждается в аппаратной поддержке, рассматриваемой в следующем разделе.

Во-вторых, - обеспечение абстрагирования пользователя от деталей управления оборудованием. Вывод на диск, например, требует сложного программирования контролера дискового

устройства, однако, все пользователи используют для этих целей простое обращение к драйверу устройства. Более того, в большинстве систем имеются библиотеки системных вызовов, обеспечивающие API для языков высокого уровня (прежде всего - для языка С). Можно также говорить о том, что ОС интегрирует ресурсы: из ресурсов низкого (физического) уровня она конструирует более сложные ресурсы, которые с одной стороны сложнее (по функциональным возможностям), а с другой стороны - проще (по управлению) низкоуровневых.

ОС как система управления ресурсами – менеджер ресурсов. С другой стороны, с точки зрения системного программиста, ОС представляет собой набор программ, управляющий всеми частями сложной вычислительной системы (ВС) - ресурсами. Ресурс - "средство системы обработки данных, которое может быть выделено процессу обработки данных на определенный интервал времени". Простыми словами: ресурс - это все те аппаратные и программные средства и данные, которые необходимы для выполнения программы. Ресурсы можно подразделить на первичные и вторичные. К первой группе относятся те ресурсы, которые обеспечиваются аппаратными средствами, например: процессор, память - оперативная и внешняя, устройства и каналы ввода-вывода и т.п. Ко второй группе - ресурсы, порождаемые ОС, например, системные коды и структуры данных, файлы, семафоры, очереди и т.п. В последнее время в связи с развитием распределенных вычислений и распределенного хранения данных все большее значение приобретают такие ресурсы как данные и сообщения.

В соответствии со вторым подходом функцией ОС является распределение процессоров, памяти, устройств и данных между процессами, конкурирующими за эти ресурсы. ОС должна управлять всеми ресурсами вычислительной машины таким образом, чтобы обеспечить максимальную эффективность ее функционирования. Критерием эффективности может быть, например, пропускная способность. Управление ресурсами включает решение двух общих, не зависящих от типа ресурса задач:

– планирование ресурса - то есть определение, кому, когда, а для делимых ресурсов и в каком количестве, необходимо выделить данный ресурс;

– отслеживание состояния ресурса – то есть поддержание оперативной информации о том, занят или не занят ресурс, а для делимых ресурсов – какое количество ресурса уже распределено, а какое свободно.

Для решения этих общих задач управления ресурсами разные ОС используют различные алгоритмы, что, в конечном счете, и определяет их облик в целом, включая характеристики производительности, область применения и даже пользовательский интерфейс. Так, например, алгоритм управления процессором в значительной степени определяет, является ли ОС системой разделения времени, системой пакетной обработки или системой реального времени.

1.1.4. Функции операционных систем

В настоящее время существует большое количество различных типов операционных систем, отличающихся областями применения, аппаратными платформами, способами реализации и др. Назначение операционных систем можно разделить на четыре основные составляющие [5, 10, 13].

1. ***Организация (обеспечение) удобного интерфейса между приложениями и пользователями***, с одной стороны, и аппаратурой компьютера – с другой. Вместо реальной аппаратуры компьютера ОС представляет пользователю расширенную виртуальную машину, с которой удобнее работать и которую легче программировать. Вот список основных сервисов, предоставляемых типичными операционными системами.

- Разработка программ: ОС представляет программисту разнообразные инструменты разработки приложений: редакторы, отладчики и т.п. Ему не обязательно знать, как функционируют различные электронные и электромеханические узлы и устройства компьютера. Часто пользователь не знает даже системы команд процессора, поскольку он может обойтись

мощными высокоуровневыми функциями, которые представляет ОС.

- Исполнение программ. Для запуска программы нужно выполнить ряд действий: загрузить в основную память программу и данные, инициализировать устройства ввода-вывода и файлы, подготовить другие ресурсы. ОС выполняет всю эту рутинную работу вместо пользователя.

- Доступ к устройствам ввода-вывода. Для управления каждым устройством используется свой набор команд. ОС предоставляет пользователю единообразный интерфейс, который скрывает все эти детали и обеспечивает программисту доступ к устройствам ввода-вывода с помощью простых команд чтения и записи. Если бы программист работал непосредственно с аппаратурой компьютера, то для организации, например, чтения блока данных с диска ему пришлось бы использовать более десятка команд с указанием множества параметров. После завершения обмена программист должен был бы предусмотреть еще более сложный анализ результата выполненной операции.

- Контролируемый доступ к файлам. При работе с файлами управление со стороны ОС предполагает не только глубокий учет природы устройства ввода-вывода, но и знание структур данных, записанных в файлах. Многопользовательские ОС, кроме того, обеспечивают механизм защиты при обращении к файлам.

- Системный доступ. ОС управляет доступом к совместно используемой или общедоступной вычислительной системе в целом, а также к отдельным системным ресурсам. Она обеспечивает защиту ресурсов и данных от несанкционированного использования и разрешает конфликтные ситуации.

- Обнаружение ошибок и их обработка. При работе компьютерной системы могут происходить разнообразные сбои за счет внутренних и внешних ошибок в аппаратном обеспечении, различного рода программных ошибок (переполнение, попытка обращения к ячейке памяти, доступ к которой запрещен и др.). В каждом случае ОС выполняет действия, минимизирующие влияние ошибки на работу приложения (от простого сообщения об ошибке до аварийной остановки программы).

- Учет использования ресурсов. Хорошая ОС имеет средства учета использования различных ресурсов и отображения параметров производительности вычислительной системы. Эта информация важна для настройки (оптимизации) вычислительной системы с целью повышения ее производительности.

В результате реальная машина, способная выполнить только небольшой набор элементарных действий (машинных команд), с помощью операционной системы превращается в виртуальную машину, выполняющую широкий набор гораздо более мощных функций. Виртуальная машина тоже управляется командами, но уже командами более высокого уровня, например: удалить файл с определенным именем, запустить на выполнение прикладную программу, повысить приоритет задачи, вывести текст файла на печать и т.д. Таким образом, назначение ОС состоит в предоставлении пользователю (программисту) некоторой расширенной виртуальной машины, которую легче программировать и с которой легче работать, чем непосредственно с аппаратурой, составляющей реальный компьютер, систему или сеть.

2. Организация эффективного использования ресурсов компьютера. ОС не только представляет пользователям и программистам удобный интерфейс к аппаратным средствам компьютера, но и является своеобразным диспетчером ресурсов компьютера. К числу основных ресурсов современных вычислительных систем относятся процессоры, основная память, таймеры, наборы данных, диски, накопители на МЛ, принтеры, сетевые устройства и др. Эти ресурсы распределяются операционной системой между выполняемыми программами. В отличие от программы, которая является статическим объектом, выполняемая программа – это динамический объект, он называется процессом и является базовым понятием современных ОС.

Управление ресурсами вычислительной системы с целью наиболее эффективного их использования является вторым назначением операционной системы. Критерии эффективности, в соответствии с которыми ОС организует управление ресурсами компьютера, могут быть различными. Например, в одних

системах важен такой критерий, как пропускная способность вычислительной систем, в других – время ее реакции. Зачастую ОС должны удовлетворять нескольким, противоречащим друг другу критериям, что доставляет разработчикам серьезные трудности.

Управление ресурсами включает решение ряда общих, не зависящих от типа ресурса задач:

- планирование ресурса – определение, какому процессу, когда и в каком качестве (если ресурс может выделяться частями) следует выделить данный ресурс;
- удовлетворение запросов на ресурсы – выделение ресурса процессам;
- отслеживание состояния и учет использования ресурса – поддержание оперативной информации о занятости ресурса и распределенной его доли;
- разрешение конфликтов между процессами, претендующими на один и тот же ресурс.

Для решения этих общих задач управления ресурсами разные ОС используют различные алгоритмы, особенности которых, в конечном счете, определяют облик ОС в целом, включая характеристики производительности, область применения и даже пользовательский интерфейс. Таким образом, управление ресурсами составляют важное назначение ОС. В отличие от функций расширенной виртуальной машины большинство функций управления ресурсами выполняются операционной системой автоматически и прикладному программисту недоступны.

3. Облегчение процессов эксплуатации аппаратных и программных средств вычислительной системы. Ряд операционных систем имеет в своем составе наборы служебных программ, обеспечивающие резервное копирование, архивацию данных, проверку, очистку и дефрагментацию дисковых устройств и др.

Кроме того, современные ОС имеют достаточно большой набор средств и способов диагностики и восстановления работоспособности системы. Сюда относятся:

- диагностические программы для выявления ошибок в конфигурации ОС;
- средства восстановления последней работоспособной конфигурации;
- средства восстановления поврежденных и пропавших системных файлов и др.

4. Возможность развития. Современные ОС организуются таким образом, что допускают эффективную разработку, тестирование и внедрение новых системных функций, не прерывая процесса нормального функционирования вычислительной системы. Большинство операционных систем постоянно развиваются (нагляден пример Windows). Происходит это в силу следующих причин.

Обновление и возникновение новых видов аппаратного обеспечения. Например, ранние версии ОС UNIX и OS/2 не использовали механизмы страничной организации памяти (что это такое, мы рассмотрим позже), потому, что они работали на машинах, не обеспеченных соответствующими аппаратными средствами.

Новые сервисы. Для удовлетворения пользователей или нужд системных администраторов ОС должны постоянно предоставлять новые возможности. Например, может потребоваться добавить новые инструменты для контроля или оценки производительности, новые средства ввода-вывода данных (речевой ввод). Другой пример – поддержка новых приложений, использующих окна на экране дисплея.

Исправления. В каждой ОС есть ошибки. Время от времени они обнаруживаются и исправляются. Отсюда постоянные появления новых версий и редакций ОС. Необходимость регулярных изменений накладывает определенные требования на организацию операционных систем. Очевидно, что эти системы (как, впрочем, и другие сложные программы системы) должны иметь модульную структуру с четко определенными межмодульными связями (интерфейсами). Важную роль играет хорошая и полная документированность системы.

Перейдем к рассмотрению состава компонентов и функций ОС. Современные операционные системы содержат сотни и тысячи модулей (например, W2000 содержит 29 млн строк

исходного кода на языке С). Функции ОС обычно группируются либо в соответствии с типами локальных ресурсов, которыми управляет ОС, либо в соответствии со специфическими задачами, применимыми ко всем ресурсам. Совокупности модулей, выполняющих такие группы функций, образуют подсистемы операционной системы.

Наиболее важными подсистемами управления ресурсами являются подсистемы управления процессами, памятью, файлами и внешними устройствами, а подсистемами, общими для всех ресурсов, являются подсистемы пользовательского интерфейса, защиты данных и администрирования.

Управление процессами. Подсистема управления процессами непосредственно влияет на функционирование вычислительной системы. Для каждой выполняемой программы ОС организует один или более процессов. Каждый такой процесс представляется в ОС информационной структурой (таблицей, дескриптором, контекстом процессора), содержащей данные о потребностях процесса в ресурсах, а также о фактически выделенных ему ресурсах (область оперативной памяти, количество процессорного времени, файлы, устройства ввода-вывода и др.). Кроме того, в этой информационной структуре хранятся данные, характеризующие историю пребывания процесса в системе: текущее состояние (активное или заблокированное), приоритет, состояние регистров, программного счетчика и др.

В современных мультипрограммных ОС может существовать одновременно несколько процессов, порожденных по инициативе пользователей и их приложений, а также инициированных ОС для выполнения своих функций (системные процессы). Поскольку процессы могут одновременно претендовать на одни и те же ресурсы, подсистема управления процессами планирует очередность выполнения процессов, обеспечивает их необходимыми ресурсами, обеспечивает взаимодействие и синхронизацию процессов.

Управление памятью. Подсистема управления памятью производит распределение физической памяти между всеми существующими в системе процессами, загрузку и удаление программных кодов и данных процессов в отведенные им

области памяти, настройку адресно-зависимых частей кодов процесса на физические адреса выделенной области, а также защиту областей памяти каждого процесса. Стратегия управления памятью складывается из стратегий выборки, размещения и замещения блока программы или данных в основной памяти. Соответственно используются различные алгоритмы, определяющие, когда загрузить очередной блок в память (по запросу или с упреждением), в какое место памяти его поместить и какой блок программы или данных удалить из основной памяти, чтобы освободить место для размещения новых блоков.

Одним из наиболее популярных способов управления памятью в современных ОС является виртуальная память. Реализация механизма виртуальной памяти позволяет программисту считать, что в его распоряжении имеется однородная оперативная память, объем которой ограничивается только возможностями адресации, предоставляемыми системой программирования.

Важная функция управления памятью – *защита памяти*. Нарушения защиты памяти связаны с обращениями процессов к участкам памяти, выделенной другим процессам прикладных программ или программ самой ОС. Средства защиты памяти должны пресекать такие попытки доступа путем аварийного завершения программы-нарушителя.

Управление файлами. Функции управления файлами сосредоточены в файловой системе ОС. Операционная система виртуализирует отдельный набор данных, хранящихся на внешнем накопителе, в виде файла – простой неструктурированной последовательности байтов, имеющих символьное имя. Для удобства работы с данными файлы группируются в каталоги, которые, в свою очередь, образуют группы – каталоги более высокого уровня. Файловая система преобразует символьные имена файлов, с которыми работает пользователь или программист, в физические адреса данных на дисках, организует совместный доступ к файлам, защищает их от несанкционированного доступа.

Управление внешними устройствами. Функции управления внешними устройствами возлагаются на подсистему управления

внешними устройствами, называемую также подсистемой ввода-вывода. Она является интерфейсом между ядром компьютера и всеми подключенными к нему устройствами. Спектр этих устройств очень обширен (принтеры, сканеры, мониторы, модемы, манипуляторы, сетевые адаптеры, АЦП разного рода и др.), сотни моделей этих устройств отличаются набором и последовательностью команд, используемых для обмена информацией с процессором и другими деталями.

Программа, управляющая конкретной моделью внешнего устройства и учитывающая все его особенности, называется драйвером. Наличие большого количества подходящих драйверов во многом определяет успех ОС на рынке. Созданием драйверов занимаются как разработчики ОС, так и компании, выпускающие внешние устройства. ОС должна поддерживать четко определенный интерфейс между драйверами и остальными частями ОС. Тогда разработчики компаний-производителей устройств ввода-вывода могут поставлять вместе со своими устройствами драйверы для конкретной операционной системы.

Защита данных и администрирование. Безопасность данных вычислительной системы обеспечивается средствами отказоустойчивости ОС, направленными на защиту от сбоев и отказов аппаратуры и ошибок программного обеспечения, а также средствами защиты от несанкционированного доступа. Для каждого пользователя системы обязательна процедура логического входа, в процессе которой ОС убеждается, что в систему входит пользователь, разрешенный административной службой. Администратор вычислительной системы определяет и ограничивает возможности пользователей в выполнении тех или иных действий, т.е. определяет их права по обращению и использованию ресурсов системы.

Важным средством защиты являются функции аудита ОС, заключающегося в фиксации всех событий, от которых зависит безопасность системы. Поддержка отказоустойчивости вычислительной системы реализуется на основе резервирования (дисковые RAID-массивы, резервные принтеры и другие устройства, иногда резервирование центральных процессоров, в ранних ОС – дуальные и дуплексные системы, системы с мажоритарным органом и др.). Вообще обеспечение

отказоустойчивости системы – одна из важнейших обязанностей системного администратора, который для этого использует ряд специальных средств и инструментов [7, 10, 13].

Интерфейс прикладного программирования. Прикладные программисты используют в своих приложениях обращения к операционной системе, когда для выполнения тех или иных действий им требуется особый статус, которым обладает только ОС. Возможности операционной системы доступны программисту в виде набора функций, который называется интерфейсом прикладного программирования (Application Programming Interface, API). Приложения обращаются к функциям API с помощью системных вызовов. Способ, которым приложение получает услуги операционной системы, очень похож на вызов подпрограмм.

Способ реализации системных вызовов зависит от структурной организации ОС, особенностей аппаратной платформы и языка программирования.

В ОС UNIX системные вызовы почти идентичны библиотечным процедурам. Ситуация в Windows иная (более подробно это рассмотрим далее).

Пользовательский интерфейс. ОС обеспечивает удобный интерфейс не только для прикладных программ, но и для пользователя (программиста, администратора). В ранних ОС интерфейс сводился к языку управления заданиями и не требовал терминала. Команды языка управления заданиями набивались на перфокарты, а результаты выполнения задания выводились на печатающее устройство.

Современные ОС поддерживают развитые функции пользовательского интерфейса для интерактивной работы за терминалами двух типов: алфавитно-цифрового и графического. При работе за алфавитно-цифровым терминалом пользователь имеет в своем распоряжении систему команд, развитость которой отражает функциональные возможности данной ОС. Обычно командный язык ОС позволяет запускать и останавливать приложения, выполнять различные операции с каталогами и файлами, получать информацию о состоянии ОС, администрировать систему. Команды могут вводиться не только в интерактивном режиме с терминала, но и считываться из так

называемого командного файла, содержащего некоторую последовательность команд.

Программный модуль ОС, ответственный за чтение отдельных команд или же последовательности команд из командного файла, иногда называют командным интерпретатором (в MS-DOS – командным процессором).

Вычислительные системы, управляемые из командной строки, например UNIX-системы, имеют командный интерпретатор, называемый оболочкой (Shell). Она, собственно, не входит в состав ОС, но пользуется многими функциями операционной системы. Когда какой-либо пользователь входит в систему, запускается оболочка. Стандартным терминалом для нее является монитор с клавиатурой. Оболочка начинает работу с печати приглашения (prompt) – знака доллара (или иного знака), говорящего пользователю, что оболочка ожидает ввода команды (аналогично управляется MS-DOS). Если теперь пользователь напечатает какую-либо команду, оболочка создает системный вызов и ОС выполнит эту команду. После завершения оболочка опять печатает приглашение и пытается прочесть следующую входную строку.

Ввод команд может быть упрощен, если операционная система поддерживает графический пользовательский интерфейс. В этом случае пользователь выбирает на экране нужный пункт меню или графический символ (так это происходит, например, в ОС Windows).

1.2. КЛАССИФИКАЦИЯ ОПЕРАЦИОННЫХ СИСТЕМ

1.2.1. Особенности алгоритмов управления ресурсами

Операционные системы различаются особенностями реализации внутренних алгоритмов управления основными ресурсами компьютера (процессорами, памятью, устройствами), особенностями использованных методов проектирования, типами аппаратных платформ, областями использования и многими другими свойствами.

Рассмотрим классификацию операционных систем по нескольким основным признакам.

В зависимости от особенностей использованного алгоритма управления процессором операционные системы делят на:

- многозадачные и однозадачные;
- многопользовательские и однопользовательские;
- системы, поддерживающие многонитевую обработку и не поддерживающие ее;
- многопроцессорные и однопроцессорные.

Поддержка многозадачности. По числу одновременно выполняемых задач операционные системы могут быть разделены на два класса:

- однозадачные (например, *MS-DOS*) и
- многозадачные (*ОС ЕС, OS/2, UNIX, Windows 98*).

Однозадачные операционные системы в основном выполняют функцию предоставления пользователю виртуальной машины, делая более простым и удобным процесс взаимодействия пользователя с компьютером. Однозадачные операционные системы включают средства управления периферийными устройствами, средства управления файлами, средства общения с пользователем.

Многозадачные операционные системы, кроме вышеперечисленных функций, управляют разделением совместно используемых ресурсов, таких как процессор, оперативная память, файлы и внешние устройства.

Поддержка многопользовательского режима. По числу одновременно работающих пользователей операционные системы делятся на:

- однопользовательские (*MS-DOS, Windows 3.x*, ранние версии *OS/2*);
- многопользовательские (*UNIX, Windows NT*).

Главным отличием многопользовательских систем от однопользовательских является наличие средств защиты информации каждого пользователя от несанкционированного доступа других пользователей. Следует заметить, что не всякая многозадачная система является многопользовательской, и не всякая однопользовательская операционная система является однозадачной.

1.2.2. Особенности аппаратных платформ

На свойства операционной системы непосредственное влияние оказывают аппаратные средства, на которые она ориентирована. По типу аппаратуры различают операционные системы персональных компьютеров, мини-компьютеров, мейнфреймов, кластеров и сетей ЭВМ. Среди перечисленных типов компьютеров могут встречаться как однопроцессорные варианты, так и многопроцессорные. В любом случае специфика аппаратных средств, как правило, отражается на специфике операционных систем.

Очевидно, что ОС большой машины является более сложной и функциональной, чем ОС персонального компьютера. Так в ОС больших машин функции по планированию потока выполняемых задач, очевидно, реализуются путем использования сложных приоритетных дисциплин и требуют большей вычислительной мощности, чем в ОС персональных компьютеров. Аналогично обстоит дело и с другими функциями[18,19].

Сетевая ОС имеет в своем составе средства передачи сообщений между компьютерами по линиям связи, которые совершенно не нужны в автономной ОС. На основе этих сообщений сетевая ОС поддерживает разделение ресурсов компьютера между удаленными пользователями, подключенными к сети. Для поддержания функций передачи сообщений сетевые ОС содержат специальные программные

компоненты, реализующие популярные коммуникационные протоколы, такие как IP, IPX, Ethernet и другие.

Многопроцессорные системы требуют от операционной системы особой организации, с помощью которой сама операционная система, а также поддерживаемые ею приложения могли бы выполняться параллельно отдельными процессорами системы. Параллельная работа отдельных частей ОС создает дополнительные проблемы для разработчиков ОС, так как в этом случае гораздо сложнее обеспечить согласованный доступ отдельных процессов к общим системным таблицам, исключить эффект гонок и прочие нежелательные последствия асинхронного выполнения работ.

Другие требования предъявляются к операционным системам кластеров. Кластер - слабо связанная совокупность нескольких вычислительных систем, работающих совместно для выполнения общих приложений, и представляющихся пользователю единой системой. Наряду со специальной аппаратурой для функционирования кластерных систем необходима и программная поддержка со стороны операционной системы, которая сводится в основном к синхронизации доступа к разделяемым ресурсам, обнаружению отказов и динамической реконфигурации системы. Одной из первых разработок в области кластерных технологий были решения компании Digital Equipment на базе компьютеров VAX. Недавно этой компанией заключено соглашение с корпорацией Microsoft о разработке кластерной технологии, использующей Windows NT. Несколько компаний предлагают кластеры на основе UNIX-машин.

Наряду с ОС, ориентированными на совершенно определенный тип аппаратной платформы, существуют операционные системы, специально разработанные таким образом, чтобы они могли быть легко перенесены с компьютера одного типа на компьютер другого типа, так называемые *мобильные* ОС. Наиболее ярким примером такой ОС является популярная система UNIX. В этих системах аппаратно-зависимые места тщательно локализованы, так что при переносе системы на новую платформу переписываются только они. Средством, облегчающем перенос остальной части ОС, является написание ее на машинно-независимом языке, например, на C,

который и был разработан для программирования операционных систем.

1.2.3. Особенности областей использования

Многозадачные ОС подразделяются на три типа в соответствии с использованными при их разработке критериями эффективности:

- системы пакетной обработки (например, ОС ЕС),
- системы разделения времени (UNIX, VMS),
- системы реального времени (QNX, RT/11).

Системы пакетной обработки предназначались для решения задач в основном вычислительного характера, не требующих быстрого получения результатов. Главной целью и критерием эффективности систем пакетной обработки является максимальная пропускная способность, то есть решение максимального числа задач в единицу времени. Для достижения этой цели в системах пакетной обработки используются следующая схема функционирования: в начале работы формируется пакет заданий, каждое задание содержит требование к системным ресурсам; из этого пакета заданий формируется мультипрограммная смесь, то есть множество одновременно выполняемых задач. Для одновременного выполнения выбираются задачи, предъявляющие отличающиеся требования к ресурсам, так, чтобы обеспечивалась сбалансированная загрузка всех устройств вычислительной машины; так, например, в мультипрограммной смеси желательно одновременное присутствие вычислительных задач и задач с интенсивным вводом-выводом. Таким образом, выбор нового задания из пакета заданий зависит от внутренней ситуации, складывающейся в системе, то есть выбирается "выгодное" задание. Следовательно, в таких ОС невозможно гарантировать выполнение того или иного задания в течение определенного периода времени. В системах пакетной обработки переключение процессора с выполнения одной задачи на выполнение другой происходит только в случае, если активная задача сама отказывается от процессора, например, из-за необходимости выполнить операцию ввода-вывода. Поэтому одна задача может

надолго занять процессор, что делает невозможным выполнение интерактивных задач. Таким образом, взаимодействие пользователя с вычислительной машиной, на которой установлена система пакетной обработки, сводится к тому, что он приносит задание, отдает его диспетчеру-оператору, а в конце дня после выполнения всего пакета заданий получает результат. Очевидно, что такой порядок снижает эффективность работы пользователя.

Системы разделения времени призваны исправить основной недостаток систем пакетной обработки - изоляцию пользователя-программиста от процесса выполнения его задач. Каждому пользователю системы разделения времени предоставляется терминал, с которого он может вести диалог со своей программой. Так как в системах разделения времени каждой задаче выделяется только квант процессорного времени, ни одна задача не занимает процессор надолго, и время ответа оказывается приемлемым. Если квант выбран достаточно небольшим, то у всех пользователей, одновременно работающих на одной и той же машине, складывается впечатление, что каждый из них единолично использует машину. Ясно, что системы разделения времени обладают меньшей пропускной способностью, чем системы пакетной обработки, так как на выполнение принимается каждая запущенная пользователем задача, а не та, которая "выгодна" системе, и, кроме того, имеются накладные расходы вычислительной мощности на более частое переключение процессора с задачи на задачу. Критерием эффективности систем разделения времени является не максимальная пропускная способность, а удобство и эффективность работы пользователя.

Системы реального времени применяются для управления различными техническими объектами, такими, например, как станок, спутник, научная экспериментальная установка или технологическими процессами, такими, как гальваническая линия, доменный процесс и т.п. Во всех этих случаях существует предельно допустимое время, в течение которого должна быть выполнена та или иная программа, управляющая объектом, в противном случае может произойти авария: спутник выйдет из зоны видимости, экспериментальные данные, поступающие с

датчиков, будут потеряны, толщина гальванического покрытия не будет соответствовать норме. Таким образом, критерием эффективности для систем реального времени является их способность выдерживать заранее заданные интервалы времени между запуском программы и получением результата (управляющего воздействия). Это время называется временем реакции системы, а соответствующее свойство системы - реактивностью. Для этих систем мультипрограммная смесь представляет собой фиксированный набор заранее разработанных программ, а выбор программы на выполнение осуществляется исходя из текущего состояния объекта или в соответствии с расписанием плановых работ.

Некоторые операционные системы могут совмещать в себе свойства систем разных типов, например, часть задач может выполняться в режиме пакетной обработки, а часть - в режиме реального времени или в режиме разделения времени. В таких случаях режим пакетной обработки часто называют фоновым режимом.

1.2.4.Особенности методов построения

При описании операционной системы часто указываются особенности ее структурной организации и основные концепции, положенные в ее основу.

К таким базовым концепциям относятся:

- Способы построения ядра системы - монолитное ядро или микроядерный подход. Большинство ОС использует монолитное ядро, которое компонуется как одна программа, работающая в привилегированном режиме и использующая быстрые переходы с одной процедуры на другую, не требующие переключения из привилегированного режима в пользовательский и наоборот. Альтернативой является построение ОС на базе микроядра, работающего также в привилегированном режиме и выполняющего только минимум функций по управлению аппаратурой, в то время как функции ОС более высокого уровня выполняют специализированные компоненты ОС - серверы, работающие в пользовательском режиме. При таком построении ОС работает более медленно, так

как часто выполняются переходы между привилегированным режимом и пользовательским, зато система получается более гибкой - ее функции можно наращивать, модифицировать или сужать, добавляя, модифицируя или исключая серверы пользовательского режима. Кроме того, серверы хорошо защищены друг от друга, как и любые пользовательские процессы.

- Построение ОС на базе объектно-ориентированного подхода дает возможность использовать все его достоинства, хорошо зарекомендовавшие себя на уровне приложений, внутри операционной системы, а именно: аккумуляцию удачных решений в форме стандартных объектов, возможность создания новых объектов на базе имеющихся с помощью механизма наследования, хорошую защиту данных за счет их инкапсуляции во внутренние структуры объекта, что делает данные недоступными для несанкционированного использования извне, структуризованность системы, состоящей из набора хорошо определенных объектов.
- Наличие нескольких прикладных сред дает возможность в рамках одной ОС одновременно выполнять приложения, разработанные для нескольких ОС. Многие современные операционные системы поддерживают одновременно прикладные среды MS-DOS, Windows, UNIX (POSIX), OS/2 или хотя бы некоторого подмножества из этого популярного набора. Концепция множественных прикладных сред наиболее просто реализуется в ОС на базе микроядра, над которым работают различные серверы, часть которых реализуют прикладную среду той или иной операционной системы.

Распределенная организация операционной системы позволяет упростить работу пользователей и программистов в сетевых средах. В распределенной ОС реализованы механизмы, которые дают возможность пользователю представлять и воспринимать сеть в виде традиционного однопроцессорного компьютера. Характерными признаками распределенной организации ОС являются: наличие единой справочной службы разделяемых ресурсов, единой службы времени, использование механизма вызова удаленных процедур (RPC) для прозрачного распределения программных процедур по машинам,

многонитевой обработки, позволяющей распараллеливать вычисления в рамках одной задачи и выполнять эту задачу сразу на нескольких компьютерах сети, а также наличие других распределенных служб.

1.3. СОСТАВ И ФУНКЦИОНИРОВАНИЕ ОС

1.3.1. Подсистема управления процессами

Одной из основных подсистем любой современной мультипрограммной компьютера, является подсистема управления процессами и потоками. Основные функции этой подсистемы:

- создание процессов и потоков;
- обеспечение процессов и потоков необходимыми ресурсами;
- изоляция процессов;

планирование выполнения процессов и потоков (вообщеОС, непосредственно влияющей на функционирование

- , следует говорить и о планировании заданий);
- диспетчеризация потоков;
- организация межпроцессного взаимодействия;
- синхронизация процессов и потоков;
- завершение и уничтожение процессов и потоков.

1. К созданию процесса приводят пять основных событий:

1. инициализация ОС (загрузка);

2. выполнение запроса работающего процесса на создание процесса;

3. запрос пользователя на создание процесса, например, при входе в систему в интерактивном режиме;

4. инициирование пакетного задания;

5. создание операционной системой процесса, необходимого для работы каких-либо служб.

Обычно при загрузке ОС создаются несколько процессов. Некоторые из них являются высокоприоритетными процессами, обеспечивающими взаимодействие с пользователями и выполняющими заданную работу. Остальные процессы являются фоновыми, они не связаны с конкретными пользователями, но выполняют особые функции – например, связанные с электронной почтой, Web-страницами, выводом на *печать*, передачей файлов *по* сети, периодическим запуском программ (например, *дефрагментации дисков*) и т.д. Фоновые процессы называют демонами[13-15].

Новый процесс может быть создан *по* запросу текущего процесса. Создание новых процессов полезно в тех случаях, когда выполняемую задачу проще всего сформировать как набор связанных, но, тем не менее, независимых взаимодействующих процессов. В интерактивных системах *пользователь* может запустить программу, набрав на клавиатуре команду или дважды щелкнув на значке программы. В обоих случаях создается новый процесс и *запуск* в нем программы. В *системах пакетной обработки* на мэйнфреймах пользователи посылают задание (возможно, с использованием удаленного доступа), а ОС создает новый процесс и запускает следующее задание из очереди, когда освобождаются необходимые ресурсы.

2. С технической точки зрения во всех перечисленных случаях новый процесс формируется одинаково: текущий процесс выполняет системный *запрос* на создание нового процесса. **Подсистема управления процессами и потоками отвечает за обеспечение процессов необходимыми ресурсами**. ОС поддерживает в памяти специальные информационные структуры, в которые записывает, какие ресурсы выделены каждому процессу. Она может назначить процессу ресурсы в единоличное пользование или совместное пользование с другими процессами. Некоторые из ресурсов выделяются процессу при его создании, а некоторые – динамически*по* запросам во *время выполнения*. Ресурсы могут быть выделены процессу на все время его жизни или только на определенный период. При выполнении этих функций подсистема управления процессами взаимодействует с другими подсистемами ОС, ответственными за *управление ресурсами*, такими как подсистема управления памятью, *подсистема ввода-вывода*, *файловая система*.

3. Для того чтобы процессы не могли вмешаться в *распределение ресурсов*, а также не могли повредить коды и данные друг друга, **важнейшей задачей ОС является изоляция одного процесса от другого**. Для этого *операционная система* обеспечивает каждый процесс отдельным виртуальным адресным пространством, так что ни один процесс не может получить прямого доступа к командам и данным другого процесса.

4. В ОС, где существуют процессы и потоки, процесс рассматривается как заявка на потребление всех видов ресурсов, кроме одного – процессорного времени. Этот важнейший *ресурс* распределяется операционной системой между другими единицами работы – потоками, которые и получили свое название благодаря тому, что они представляют собой последовательности (потоки выполнения) команд. **Переход от выполнения одного потока к другому осуществляется в результате *планирования* и *диспетчеризации*.** Работа *по* определению момента, в который необходимо прервать выполнение текущего потока, и потока, которому следует предоставить возможность выполняться, называется планированием. **Планирование потоков осуществляется на основе информации, хранящейся в описателях процессов и потоков.** При планировании принимается во внимание *приоритет потоков*, время их ожидания в очереди, накопленное *время выполнения*, интенсивность обращения к вводу-выводу и другие факторы.

5. **Диспетчеризация заключается в реализации найденного в результате планирования решения, т.е. в переключении процессора с одного потока на другой.** Диспетчеризация проходит в три этапа:

- сохранение контекста текущего потока;
- загрузка контекста потока, выбранного в результате планирования;
- запуск нового потока на выполнение.

6. Когда в системе одновременно выполняется несколько независимых задач, возникают дополнительные проблемы. **Хотя потоки возникают и выполняются синхронно, у них может возникнуть необходимость во взаимодействии**, например, при обмене данными. Для общения друг с другом процессы и потоки могут использовать широкий спектр возможностей: каналы (в *UNIX*), почтовые ящики (*Windows*), вызов удаленной процедуры, сокеты (в *Windows* соединяют процессы на разных машинах). Согласование скоростей потоков также очень важно для предотвращения эффекта "гонок" (когда несколько потоков пытаются изменить один и тот же *файл*), взаимных блокировок и

• загрузку кодов и данных процессов в отведенные им области памяти;

• настройку адресно-зависимых частей кодов процесса на физические адреса выделенной области;

• защиту областей памяти каждого процесса.

Известно множество алгоритмов распределения ОП. Их отличием может быть, например:

- число выделяемых процессу областей памяти (одной непрерывной или нескольких несмежных);
- степень свободы границы областей (статическая фиксация на всем ИСП или динамическое перемещение при дополнительном увеличении);
- единица и форма выделения (страницами фиксированного размера или сегментами переменной длины).

Популярным способом управления памятью является механизм поддержки виртуальной памяти, позволяющий программисту писать программы так, как будто в его распоряжении имеется однородная ОП достаточно большого размера[6-8].

Защита памяти – это избирательная способность ОС предохранять выполняемую задачу от записи или чтения памяти, назначенной другой задаче. Реальные программы часто содержат ошибки, вызывающие попытки обращения к «чужой» памяти. Средства защиты памяти в ОС должны пресекать несанкционированный доступ процессов к чужим областям памяти.

Таким образом, функциями ОС по управлению памятью являются:

• отслеживание свободной и занятой памяти;

• выделение памяти процессам и освобождение памяти при завершении процессов;

• защита памяти;

• вытеснение процессов из ОП на диск, если основной памяти недостаточно для размещения всех процессов и возвращение их обратно в ОП;

• настройка адресов программы на конкретную область физической памяти.

1.3.4. Файловая подсистема

Способность ОС скрывать сложности реальной аппаратуры ярко проявляется в одной из основных подсистем ОС – **файловой системе** (ФС). ОС представляет отдельный набор данных, хранящийся на внешнем накопителе, в виде файла – простой неструктурированной последовательности байтов, имеющей символьное имя. Для удобства работы файлы группируются в многоуровневые каталоги. Пользователь средствами ОС может выполнять над файлами и каталогами различные известные операции [1-4].

Чтобы представить множество наборов данных, разбросанных по цилиндрам и поверхностям дисков различных типов, в виде удобной иерархии файлов и каталогов, ОС должна решить множество задач. ФС ОС преобразует символьные имена файлов в физические адреса данных на диске, организует совместный доступ к файлам, защищает их от несанкционированного доступа.

При выполнении своих функций ФС тесно взаимодействует с подсистемой ввода-вывода (управления УВВ), которая по запросам ФС осуществляет передачу данных между дисками и ОП.

Подсистема ввода-вывода играет роль интерфейса ко всем УВВ компьютера. Спектр УВВ обширен, их номенклатура насчитывает сотни моделей. Эти модели могут существенно различаться набором и последовательностью команд обмена информацией с процессором и памятью компьютера, скоростью работы, кодировкой передаваемых данных, возможностью совместного использования и другими деталями.

Для управления конкретной моделью УВВ с учетом всех ее особенностей предназначен специальный драйвер. Драйвер может управлять единственной моделью или целой группой однотипных УВВ. Для пользователя очень важно, чтобы ОС включала как можно больше разнообразных драйверов, что гарантирует возможность использования большего числа УВВ разных производителей. От наличия подходящих драйверов во многом зависит успех ОС на рынке.

Созданием драйверов УВВ занимаются как разработчики конкретной ОС, так и производители самих УВВ. ОС должна поддерживать хорошо определенный интерфейс между драйверами и остальной частью ОС, чтобы разработчики УВВ могли поставлять свои драйверы УВВ для данной ОС.

Прикладные программисты могут пользоваться интерфейсом драйверов при разработке своих программ. Но это не очень удобно, поскольку такой интерфейс обычно представляет собой низкоуровневые операции с огромным числом деталей.

Поддержание высокоуровневого унифицированного набора функций так называемого **интерфейса прикладного программирования** (Application Programming Interface, API) для разнородных УВВ является одной из наиболее важных задач ОС. Со времени появления ОС UNIX такой унифицированный API в большинстве ОС строится на основе концепции файлового доступа. Ее суть в том, что обмен с любым УВВ выглядит как обмен с файлом, имеющим имя и представляющим собой неструктурированную последовательность байтов. В качестве файла может выступать реальный файл на диске или иное устройство, и это – самая удобная для пользователя или программиста абстракция [2, 17-22].

1.3.5. Подсистема коммуникации

Связь между процессами — это суть распределенных систем. В определении распределенной системы содержится фраза «координирующая свои действия только путем посылки сообщений». Таким образом, коммуникации — важнейшая подсистема в составе РС.

Распределенные системы создаются поверх компьютерных сетей. При этом взаимодействие в распределенных системах удобно организовывать на основе высокоуровневых моделей групповых коммуникаций, косвенного (*indirect*) взаимодействия, очередей сообщений, удаленных вызовов процедур и объектов, которые базируются на низкоуровневых средствах связи, межпроцессном обмене сообщениями и коммуникационных сетевых и транспортных протоколах. В реальном мире средства

связи не могут гарантировать абсолютно надежного способа обмена сообщениями: при определенных условиях сообщения могут задерживаться, теряться, дублироваться и менять порядок. На основе таких принципиально ненадежных технических средств связи путем использования специализированных протоколов строятся относительно надежные логические средства связи.

Состав коммуникационной подсистемы

Архитектурные блоки коммуникационной подсистемы собираются из коммуникационных элементов. Распределенные системы разбиваются на слои и которые представляют собой «промежуточный слой» и «промежуточный уровень».

Нижний слой — это низкоуровневые сетевые и межсетевые технологии, базирующиеся на сетевых протоколах, которые входят в состав ОС.

Под *взаимодействующими сущностями* в распределенных системах обычно подразумевают процессы, поэтому межпроцессный обмен считается базовой коммуникационной парадигмой. Можно сделать также следующие уточнения. В системах, где процессы дополнены потоками, коммуникации осуществляют потоки. В некоторых примитивных системах, например сенсорных сетях, абстракция процесса не поддерживается. Для таких систем коммуникационные сущности — отдельные узлы.

Межпроцессный обмен относится к относительно низкоуровневой поддержке коммуникаций между процессами в распределенных системах. Он включает примитивы обмена сообщениями, реализованные при помощи прямого доступа к программному интерфейсу, предлагаемому интернет-протоколами (программирование с помощью сокетов). С точки зрения прикладных программ межпроцессного обмена зачастую оказывается недостаточно. Поэтому иногда в качестве элементов взаимодействия в распределенных системах рассматривают проблемно-ориентированные абстракции, например объекты, компоненты, веб-сервисы и др., а в качестве элементов связующего ПО часто выступают удаленные вызовы процедуры и

обращения к удаленным объектам, а также разнообразные косвенные (*indirect*) взаимодействия.

Удаленные вызовы представляют распространенный способ двухсторонних коммуникаций в распределенных системах и обычно базируются на протоколе запрос-ответ (*request-reply*). Данный протокол включает попарный обмен сообщениями между клиентом и сервером. Первое сообщение от клиента к серверу содержит название операции, которую надо исполнить на сервере и ассоциированные аргументы, а ответное сообщение содержит результаты операции, как правило, в виде массива байтов.

Удаленный вызов процедуры (remote procedure call, RPC) предполагает, что процедуры на удаленном компьютере могут быть вызваны, как если бы они были процедурами локального адресного пространства. При этом система скрывает факт распределения, включая кодирование и декодирование параметров и результатов, посылку сообщений и организацию необходимой семантики для вызова процедуры. Этот подход является элегантной поддержкой клиент-серверной архитектуры. При этом обеспечивается прозрачность доступа и локализации ресурсов. *Удаленный вызов объекта (Remote method invocation,* RMI) имеет сходство с RPC, но в мире распределенных объектов. В рамках этого подхода вызывающий объект может вызвать метод удаленного объекта. Как и в случае RPC, нижележащие детали обычно скрыты от пользователя[15,16].

Техника удаленных вызовов базируется на двухсторонних коммуникациях, когда отправитель сообщения (или удаленного вызова) явным образом взаимодействует с соответствующим получателем. Получатели знают о существовании и идентификации отправителей, и в большинстве случаев обе части подсистемы функционируют одновременно.

Есть, однако, много подходов, которые обеспечивают косвенные (опосредованные, *indirect*) коммуникации через третьи сущности, лежащие в основе слабой связи между получателем и отправителем. В частности, при этом:

- отправителю не нужно знать, кому он посылает сообщение (отсутствие непосредственной пространственной связи);

- отправитель и получатель не должны существовать в одно и то же время (отсутствие временной связи).

К методам косвенного взаимодействия можно отнести: *групповые коммуникации (group communications), очереди сообщений (message queues), системы публикации-подписки (publish-subscribe systems) и распределенную разделяемую память (distributed shared memory,* DSM). В данной книге более подробно будут рассмотрены два первых способа. В случае групповых коммуникаций происходит доставка сообщений набору получателей. Групповая коммуникация основывается на абстракции группы, которая представлена в системе идентификатором группы. Получатели могут начать получать сообщения, посланные группе, путем присоединения к группе. Отправители посылают сообщение группе через групповой идентификатор и, следовательно, не нуждаются в знании адресов получателей сообщений. Группа обычно поддерживает механизм присоединения и выхода из группы и включает механизмы обработки отказов членов группы. В случае очередей сообщений процесс-отправитель может послать сообщение в указанную очередь, а процесс-потребитель может получить сообщение из очереди или быть оповещенным о прибытии в очередь нового сообщения.

Контрольные вопросы

1. Выберите из предложенного списка, что может являться критерием эффективности вычислительной системы:

-пропускная способность
-занятость оперативной памяти
-загруженность центрального процессора
-занятость временной памяти

2. Системы пакетной обработки предназначены для решения задач:

-вычислительного характера
-требующих постоянного диалога с пользователем
-занятость оперативной памяти

-требующих решения конкретной задачи за определенный промежуток времени

3. В каких системах гарантируется выполнение задания за определенный промежуток времени:
-пакетной обработки
-разделения времени
-занятость оперативной памяти
-системах реального времени
4. В системах пакетной обработки суммарное время выполнения смеси задач:

-равно сумме времен выполнения всех задач смеси
-меньше или равно суммы времен выполнения всех задач смеси
-больше или равно суммы времен выполнения всех задач смеси
-занятость оперативной памяти
5. В системах реального времени

-набор задач неизвестен заранее
-занятость оперативной памяти
-набор задач известен заранее
-известен или нет набор задач зависит от характера системы
6. Самое неэффективное использование ресурсов вычислительной системы:
-в системах пакетной обработки
-занятость оперативной памяти
-в системах разделения времени
-в системах реального времени

7. В многопоточных системах поток есть –

-заявка на ресурсы
-занятость оперативной памяти
-заявка на ресурс ЦП
-заявка на ресурс ОП
8. Потоки создаются с целью:

-ускорения работы процесса

-защиты областей памяти
-занятость оперативной памяти
-улучшения межпроцессного взаимодействия
9. Как с точки зрения экономии ресурсов лучше распараллелить работу:
-создать несколько процессов
-создать несколько потоков
-занятость оперативной памяти
-оба равнозначны, можно выбирать любой из них
10. Планирование потоков игнорирует:
-приоритет потока
-занятость оперативной памяти
-время ожидания в очереди
-принадлежность некоторому процессу
11. В каких системах тип планирования статический
-реального времени
-разделения времени
-занятость оперативной памяти
-пакетной обработки
12. Состояние, которое не определено для потока в системе:
-выполнение
-синхронизация
-ожидание
-готовность
13. Каких смен состояний не существует в системе:
-выполнение → готовность
-ожидание →выполнение
-ожидание → готовность
-готовность → ожидание
14. Какой из алгоритмов планирования является централизованным:
-вытесняющий
-памятный
-возможный
-невытесняющий
15. При каком кванте времени в системах, использующих алгоритм квантования, время ожидания потока в очереди не зависит от длительности ее выполнения:

-при маленьком кванте времени
-занятость оперативной памяти
-при длительном кванте времени
-при любом кванте времени
16. Приоритет процесса не зависит от:
-того, является ли процесс системным или прикладным
-статуса пользователя
-требуемых процессом ресурсов
-занятость оперативной памяти
17. В каких пределах может изменяться приоритет потока в системе Windows NT:
-от базового приоритета процесса до нижней границы диапазона приоритета потоков реального времени
-от нуля до базового приоритета процесса
-занятость оперативной памяти
-базовый приоритет процесса ± 2
18. Каких классов прерываний нет?
-аппаратных
-асинхронных
-внутренних
-программных

ГЛАВА 2. АРХИТЕКТУРА СОВРЕМЕННЫХ ОС

2.1. УПРАВЛЕНИЯ ЛОКАЛЬНЫМИ РЕСУРСАМИ

2.1.1. Управление и состояние процессов

Процесс (задача) - программа, находящаяся в режиме выполнения.

С каждым процессом связывается его **адресное пространство,** из которого он может читать и в которое он может писать данные.

Адресное пространство содержит:

- саму программу
- данные к программе
- стек программы

С каждым процессом связывается набор **регистров,** например:

- счетчика команд (в процессоре) - регистр в котором содержится адрес следующей, стоящей в очереди на выполнение команды. После того как команда выбрана из памяти, счетчик команд корректируется и указатель переходит к следующей команде.
- указатель стека
- и д.р.

Во многих операционных системах вся информация о каждом процессе, дополнительная к содержимому его собственного адресного пространства, хранится в **таблице процессов** операционной системы.

Некоторые поля таблицы

Таблица 2.1

Управление процессом	**Управление памятью**	**Управление файлами**
Регистры Счетчик команд Указатель стека Состояние процесса Приоритет	Указатель на текстовый сегмент Указатель на сегмент данных Указатель на	Корневой каталог Рабочий каталог Дескрипторы файла Идентификатор пользователя

Параметры планирования Идентификатор процесса Родительский процесс Группа процесса Время начала процесса Использованное процессорное время	сегмент стека	Идентификатор группы

Модель процесса

В многозадачной системе реальный процессор переключается с процесса на процесс, но для упрощения модели рассматривается набор процессов, идущих параллельно (псевдопараллельно).

Рассмотрим схему с четырьмя работающими программами.

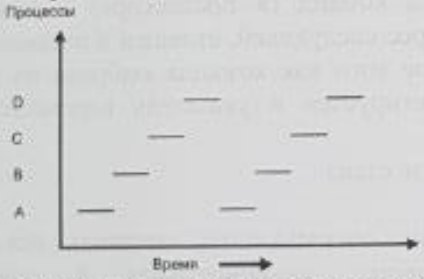


Рис.2.1. Схема с четырьмя работающими программами

В каждый момент времени активен только один процесс

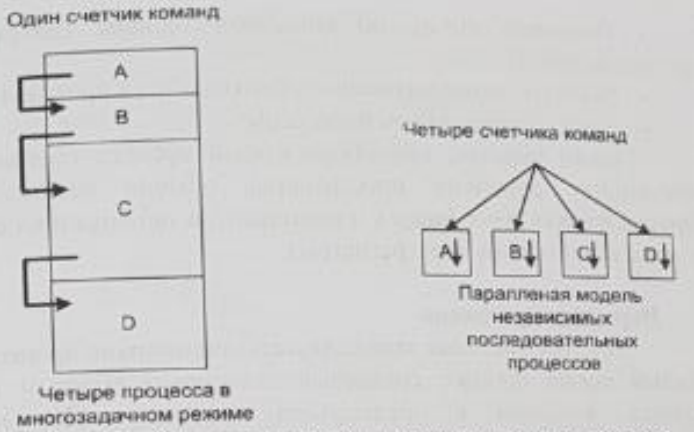


Рис.2.2. Четыре процесса и четыре счетчика команд

Справа представлены параллельно работающие процессы, каждый со своим счетчиком команд. Разумеется, на самом деле существует только один физический счетчик команд, в который загружается логический счетчик команд текущего процесса. Когда время, отведенное текущему процессу, заканчивается, физический счетчик команд сохраняется в памяти, в логическом счетчике команд процесса.

Создание процесса

Три основных события, приводящие к созданию процессов (вызов **fork** или **CreateProcess**)[1,2]:

- Загрузка системы
- Работающий процесс подает системный вызов на создание процесса
- Запрос пользователя на создание процесса

Во всех случаях, активный текущий процесс посылает системный вызов на создание нового процесса.

Каждому процессу присваивается идентификатор процесса PID - Process IDentifier.

Завершение процесса

(вызов **exit** или **ExitProcess**):

- Плановое завершение (окончание выполнения)

- Плановый выход по известной ошибке (например, отсутствие файла)
- Выход по неисправимой ошибке (ошибка в программе)
- Уничтожение другим процессом

Таким образом, приостановленный процесс состоит из собственного адресного пространства, обычно называемого **образом памяти** (**core image**), и компонентов таблицы процессов (в числе компонентов и его регистры).

Иерархия процессов

В UNIX системах заложена жесткая иерархия процессов. Каждый новый процесс созданный системным вызовом fork, является дочерним к предыдущему процессу. Дочернему процессу достаются от родительского переменные, регистры и т.п. После вызова fork, как только родительские данные скопированы, последующие изменения в одном из процессов не влияют на другой, но процессы помнят о том, кто является родительским.

В таком случае в UNIX существует и прародитель всех процессов - процесс **init**.

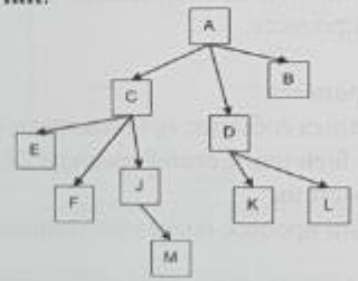


Рис.2.3. Дерево процессов для систем UNIX

В Windows не существует понятия иерархии процессов. Хотя можно задать специальный маркер родительскому процессу, позволяющий контролировать дочерний процесс.

Состояние процессов

Три состояния процесса:

- Выполнение (занимает процессор)
- Готовность (процесс временно приостановлен, чтобы позволить выполняться другому процессу)
- Ожидание (процесс не может быть запущен по своим внутренним причинам, например, ожидая операции ввода/вывода)

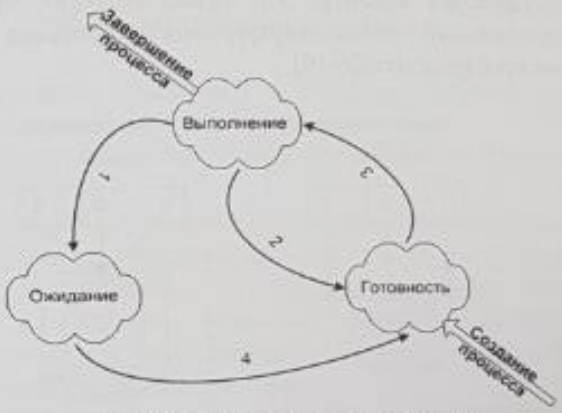


Рис.2.3. Три состояния процесса

Возможные переходы между состояниями:
1. Процесс блокируется, ожидая входных
2. Планировщик выбирает другой процесс
3. Планировщик выбирает этот процесс
4. Поступили входные данные

Переходы 2 и 3 вызываются планировщиком процессов операционной системы, так что сами процессы даже не знают об этих переходах. С точки зрения самих процессов есть два состояния выполнения и ожидания.

На серверах для ускорения ответа на запрос клиента, часто загружают несколько процессов в режим ожидания, и как только сервер получит запрос, процесс переходит из "ожидания" в "выполнение". Этот переход выполняется намного быстрее, чем запуск нового процесса.

Потоки (нити, облегченный процесс)
Понятие потока

Каждому процессу соответствует адресное пространство и одиночный **поток** исполняемых команд. В многопользовательских системах, при каждом обращении к одному и тому же сервису, приходится создавать новый процесс для обслуживания клиента. Это менее выгодно, чем создать квазипараллельный поток внутри этого процесса с одним адресным пространством[6-10].

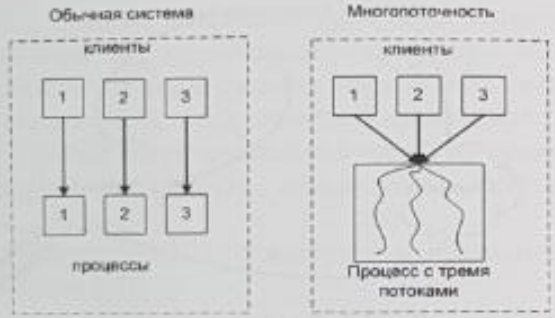


Рис.2.4. Сравнение многопоточной системы с однопоточной

Модель потока

С каждым потоком связывается:

- Счетчик выполнения команд
- Регистры для текущих переменных
- Стек
- Состояние

Потоки делят между собой элементы своего процесса:

- Адресное пространство
- Глобальные переменные
- Открытые файлы
- Таймеры
- Семафоры
- Статистическую информацию.

В остальном модель идентична модели процессов.

В POSIX и Windows есть поддержка потоков на уровне ядра. В Linux есть новый системный вызов **clone** для создания потоков, отсутствующий во всех остальных версиях системы UNIX. В POSIX есть новый системный вызов **pthread_create** для создания потоков. В Windows есть новый системный вызов **Createthread** для создания потоков.

Преимущества использования потоков

1. Упрощение программы в некоторых случаях, за счет использования общего адресного пространства.
2. Быстрота создания потока, по сравнению с процессом, примерно в 100 раз.
3. Повышение производительности самой программы, т.к. есть возможность одновременно выполнять вычисления на процессоре и операцию ввода/вывода. Пример: текстовый редактор с тремя потоками может одновременно взаимодействовать с пользователем, форматировать текст и записывать на диск резервную копию.

Реализация потоков в пространстве пользователя, ядра и смешанное

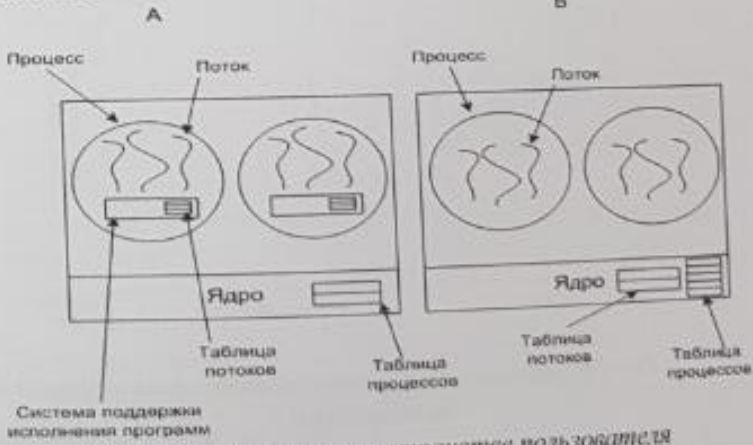


*рис.2.5. **A** - потоки в пространстве пользователя*
__B__ - потоки в пространстве ядра

В случае **А** ядро о потоках ничего не знает. Каждому процессу необходима **таблица потоков**, аналогичная таблице процессов.

Преимущества случая **А**:

- Такую многопоточность можно реализовать на ядре не поддерживающем многопоточность
- Более быстрое переключение, создание и завершение потоков
- Процесс может иметь собственный алгоритм планирования.

Недостатки случая **А**:

- Отсутствие прерывания по таймеру внутри одного процесса
- При использовании блокирующего (процесс переводится в режим ожидания, например: чтение с клавиатуры, а данные не поступают) системного запроса все остальные потоки блокируются.
- Сложность реализации

Рис.2.6. Мультиплексирование потоков пользователя в потоках ядра

Поток ядра может содержать несколько потоков пользователя.

Особенности реализации Windows

Используется четыре понятия:

- Задание - набор процессов с общими квотами и лимитами
- Процесс - контейнер ресурсов (память ...), содержит как минимум один поток.
- Поток - именно исполняемая часть, планируемая ядром.
- Волокно - облегченный поток, управляемый полностью в пространстве пользователя. Один поток может содержать несколько волокон.

Потоки работают в режиме пользователя, но при системных вызовах переключаются в режим ядра. Из-за переключения в режим ядра и обратно, очень замедляется работа системы. Поэтому было введено понятие **волокна**. У каждого потока может быть несколько волокон.

Взаимодействие между процессами

Ситуации, когда приходится процессам взаимодействовать:

- Передача информации от одного процесса другому
- Контроль над деятельностью процессов (например: когда они борются за один ресурс)
- Согласование действий процессов (Например: когда один процесс поставляет данные, а другой их выводит на печать. Если согласованности не будет, то второй процесс может начать печать раньше, чем поступят данные).

Два вторых случая относятся и к потокам. В первом случае у потоков нет проблем, т.к. они используют общее адресное пространство.

Передача информации от одного процесса другому

Передача может осуществляться несколькими способами:

- Разделяемая память
- **Каналы** (трубы), это псевдофайл, в который один процесс пишет, а другой читает.

• **Сокеты** - поддерживаемый ядром механизм, скрывающий особенности среды и позволяющий единообразно взаимодействовать процессам, как на одном компьютере, так и в сети.

• Почтовые ящики (только в Windows), однонаправленные, возможность широковещательной рассылки.

Вызов удаленной процедуры, процесс **A** может вызвать процедуру в процессе **B**, и получить обратно данные.

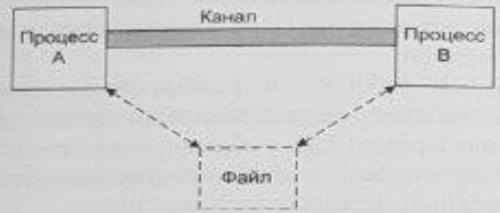


Рис. 2.7. Схема для канала

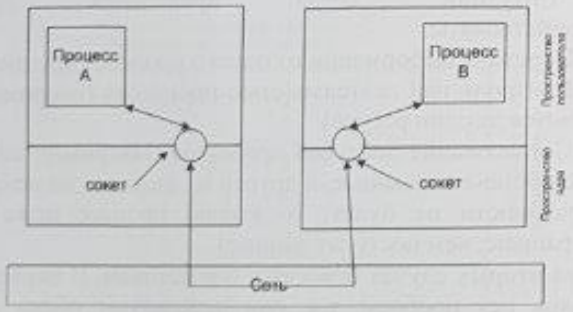


Рис.2.8. Схема для сокетов

Состояние состязания

Состояние состязания - ситуация когда несколько процессов считывают или записывают данные (в память или файл) одновременно.

Рассмотрим пример, когда два процесса пытаются распечатать файл. Для этого им нужно поместить имя файла в спулер печати, в свободный сегмент.

in - переменная, указывающая на следующий свободный сегмент

out - переменная, указывающая на следующее имя файла для печати

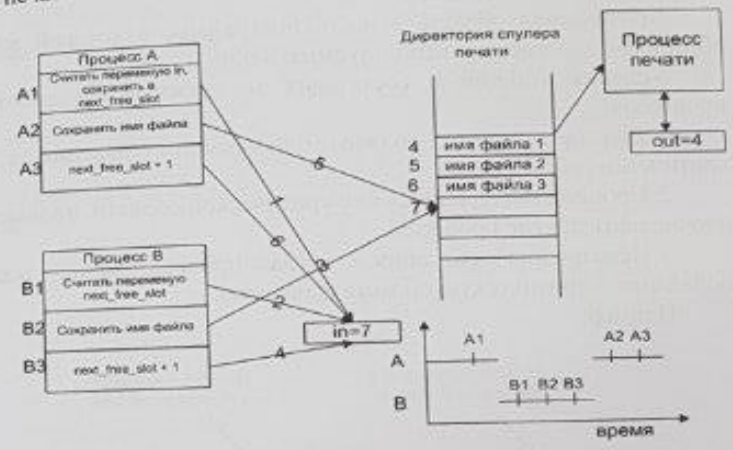


Рис.2.9. Пример состязания

Распишем события по пунктам[1,8].

1. Процесс **A** считывает переменную in (равную 7), и сохраняет ее в своей переменной next_free_slot.

2. Происходит прерывание по таймеру, и процессор переключается на процесс **B**.

3. Процесс **B** считывает переменную in (равную 7), и сохраняет ее в своей переменной next_free_slot.

4. Процесс **B** сохраняет имя файла в сегменте **7**.

5. Процесс **B** увеличивает переменную next_free_slot на единицу (next_free_slot+1), и заменяет значение in на 8.

6. Управление переходит процессу **A**, и продолжает с того места на котором остановился.

7. Процесс **A** сохраняет имя файла в сегменте 7, затирая имя файла процесса **B**.

8. Процесс **A** увеличивает переменную next_free_slot на единицу (next_free_slot+1), и заменяет значение in на 8.

Как видно из этой ситуации, файл процесса **B** не будет напечатан.

Критические области

Критическая область - часть программы, в которой есть обращение к совместно используемым данным.

Условия избегания состязания и эффективной работы процессов:

1. Два процесса не должны одновременно находиться в критических областях.
2. Процесс, находящийся вне критической области, не может блокировать другие процессы.
3. Невозможна ситуация, когда процесс вечно ждет попадания в критическую область (зависает).

Пример:

Рис.2.10. Взаимное исключение с использованием критических областей

Взаимное исключение с активным ожиданием

Рассмотрим методы взаимного исключения

Запрещение прерываний

Заключается в запрещении всех прерываний при входе процесса в критическую область.

Недостаток этого метода в том, что если произойдет сбой процесса, то он не сможет снять запрет на прерывания.

Переменные блокировки

Вводится понятие переменной блокировки, т.е. если значение этой переменной равно, например 1, то ресурс занят другим процессом, и второй процесс переходит в режим ожидания (блокируется) до тех пор, пока переменная не примет значение 0.

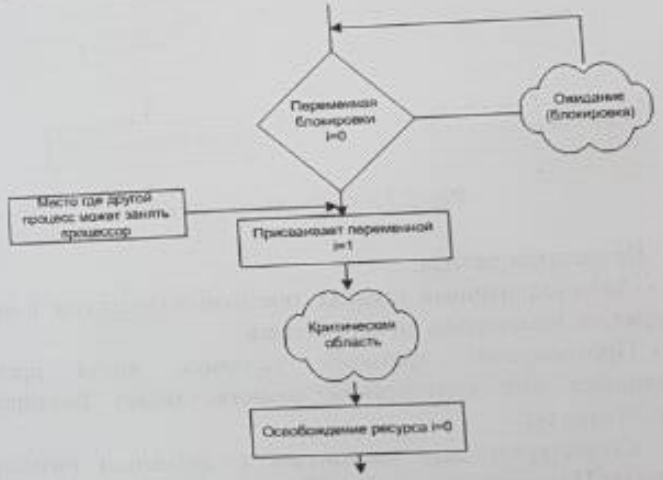


Рис.2.11. Метод блокирующих переменных

Проблема, как и с процессом печати, после того как первый процесс считает 0, второй может занять процессор и тоже считать 0. Заблокированный процесс находится в режиме

Проблема переполненного буфера (проблема производителя и потребителя)

Рассмотрим два процесса, которые совместно используют буфер ограниченного размера, один процесс пишет в буфер, другой считывает данные.

Чтобы первый процесс не писал, когда буфер полный, а второй не считывал, когда он пуст, вводится переменная *count* для подсчета количества элементов в буфере.

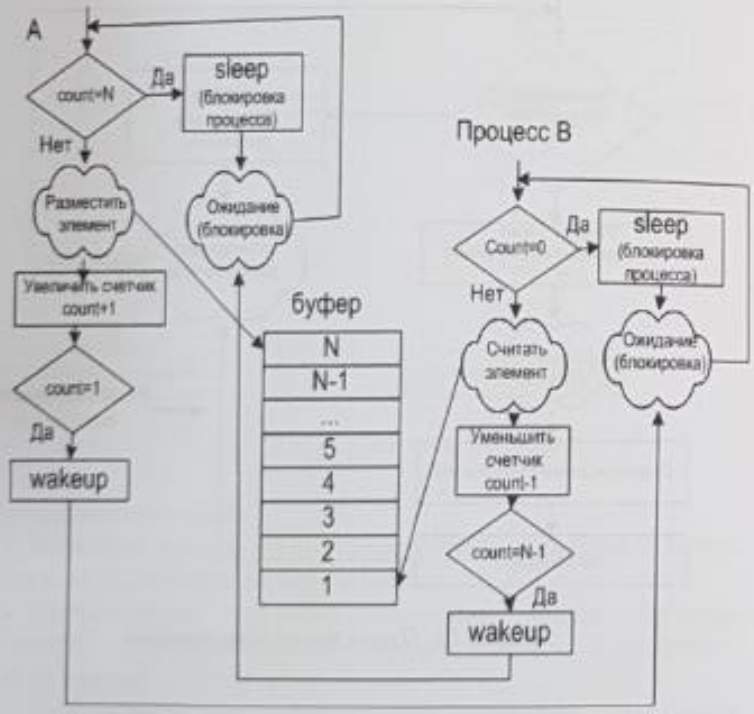


Рис.2.14. Проблема переполненного буфера

В этой ситуации оба процесса могут попасть в состояние ожидания, если пропадет сигнал активации.

Алгоритм такой ситуации:

1. Процесс **B**, считал count=0 (заблокироваться он еще не успел)
2. Планировщик передал управление процессу **A**
3. Процесс **A**, выполнил все вплоть до wakeup, пытаясь разблокировать процесс **B** (но он не заблокирован, wakeup срабатывает впустую)
4. Планировщик передал управление процессу **B**
5. И он заблокировался, и больше сигнала на разблокировку не получит
6. Процесс **A** в конце концов заполнит буфер и заблокируется, но сигнала на разблокировку не получит.

Типы процессов

Системные процессы

Системные процессы являются частью ядра, всегда расположены в ОЗУ и запускаются особым образом при инициализации ядра системы. Выполняемые инструкции и данные этих процессов находятся в ядре системы, таким образом они могут вызывать функции и обращаться к данным, недоступным для остальных процессов. Системными процессами являются: shed (диспетчер свопинга), vhand (диспетчер страничного замещения), bdfflush (диспетчер буферного кэша), kmadaemon (диспетчер памяти ядра). Системный процесс init является прародителем всех остальных процессов в UNIX, но не является частью ядра. Его запуск происходит из исполняемого файла (/etc/init)[15,16].

Демоны

Демоны – это интерактивные процессы, которые запускаются путем загрузки в память исполняемых файлов, и выполняются в фоновом режиме. Демоны запускаются после инициализации ядра при инициализации системы, и обеспечивают работу различных подсистем ОС: системы терминального доступа, системы печати, системы сетевого доступа, сетевых услуг. Демоны не связаны ни с одним пользовательским сеансом работы и не могут непосредственно управляться пользователем. Большую часть времени демоны

ожидают пока тот или иной процесс запросит определенную услугу: доступ к файловому архиву или печать документов.

Прикладные процессы

К прикладным процессам относятся все остальные процессы, выполняющиеся в системе, как правило, порожденные в рамках пользовательского сеанса работы. Например, запуск команды ls(1) породит прикладной процесс просмотра файлов, процессов, которые выполняются. Основной командный интерпретатор (loginsheel), обеспечивает работу в UNIX, он запускается сразу после регистрации в системе, а после завершения работы отключается от системы. Пользовательские процессы могут выполняться как в интерактивном, так и фоновом режимах, но время их жизни ограничено сеансом пользователя. Интерактивные процессы монопольно владеют терминалом, пока такой процесс не завершит свое выполнение, например, ps(1). В то же время интерпретатор sheel считывает пользовательский ввод и запускает задачи.

Атрибуты процессов

Каждый процесс имеет уникальный идентификатор PID, позволяющий ядру системы различать процессы. Когда создается новый процесс, ядро присваивает ему следующий свободный идентификатор по нарастающей до максимально возможного. Когда процесс завершает свою работу, ядро освобождает занятый им идентификатор, (0…65535)=2^{16}.

Идентификатор родительского процесса (PPID-Parent Process ID) -Идентификатор процесса, породившего данный процесс.

Приоритет процесса (Nice Number)

С одной стороны это относительный приоритет процесса, учитываемый планировщиком при определении очередности запуска. Фактическое распределение процессорных ресурсов определяется приоритетом выполнения, зависящим от нескольких факторов, в частности, от заданного относительного приоритета. Относительный приоритет не применяется системой на всем протяжении жизни процесса, но может быть изменен пользователем или администратором. Приоритет выполнения динамически обновляется ядром.

Реальный (RID) и эффективный (EUID) идентификаторы пользователя

Реальный идентификатором пользователя данного процесса является идентификатор пользователя, запустившего процесс. Эффективный идентификатор служит для определения прав доступа процесса к системным ресурсам, в первую очередь к ресурсам файловой системы. Обычно RID и EUID эквивалентны, т.е. процесс имеет в системе те же права, что и пользователь, запустивший его. Существует возможность задать процессу более широкие права, чем права пользователя путем установки флаги SUID, когда эффективному идентификатору присваивается значение идентификатора владельца исполняемого файла.

2.1.2. Алгоритмы планирования процессов

Планирование процессов

Планирование - обеспечение поочередного доступа процессов к одному процессору.

Планировщик - отвечающая за это часть операционной системы.

Алгоритм планирования - используемый алгоритм для планирования.

Ситуации, когда необходимо планирование:

1. Когда создается процесс
2. Когда процесс завершает работу
3. Когда процесс блокируется на операции ввода/вывода, семафоре, и т.д.
4. При прерывании ввода/вывода.

Алгоритм планирования без переключений (неприоритетный) - не требует прерывание по аппаратному таймеру, процесс останавливается только когда блокируется или завершает работу.

Алгоритм планирования с переключениями (приоритетный) - требует прерывание по аппаратному таймеру, процесс работает только отведенный период времени, после

этого он приостанавливается по таймеру, чтобы передать управление планировщику.

Необходимость алгоритма планирования зависит от задач, для которых будет использоваться операционная система.

Основные три системы:

1. Системы пакетной обработки - могут использовать неприоритетный и приоритетный алгоритм (например: для расчетных программ).

2. Интерактивные системы - могут использовать только приоритетный алгоритм, нельзя допустить чтобы один процесс занял надолго процессор (например: сервер общего доступа или персональный компьютер).

3. Системы реального времени - могут использовать неприоритетный и приоритетный алгоритм (например: система управления автомобилем).

Задачи алгоритмов планирования:

1. **Для всех систем**

Справедливость - каждому процессу справедливую долю процессорного времени

Контроль над выполнением принятой политики

Баланс - поддержка занятости всех частей системы (например: чтобы были заняты процессор и устройства ввода/вывода)

2. **Системы пакетной обработки**

Пропускная способность - количество задач в час

Оборотное время - минимизация времени на ожидание обслуживания и обработку задач.

Использование процесса - чтобы процессор всегда был занят.

3. **Интерактивные системы**

Время отклика - быстрая реакция на запросы

Соразмерность - выполнение ожиданий пользователя (например: пользователь не готов к долгой загрузке системы)

4. **Системы реального времени**

Окончание работы к сроку - предотвращение потери данных

Предсказуемость - предотвращение деградации качества в мультимедийных системах (например: потерь качества звука должно быть меньше чем видео).

Планирование в системах пакетной обработки

"Первый пришел - первым обслужен" (FIFO - First In Fist Out)

Процессы ставятся в очередь по мере поступления.

Преимущества:

- Простата
- Справедливость (как в очереди покупателей, кто последний пришел, тот оказался в конце очереди)

Недостатки:

- Процесс, ограниченный возможностями процессора может затормозить более быстрые процессы, ограниченные устройствами ввода/вывода.

"Кратчайшая задача - первая"

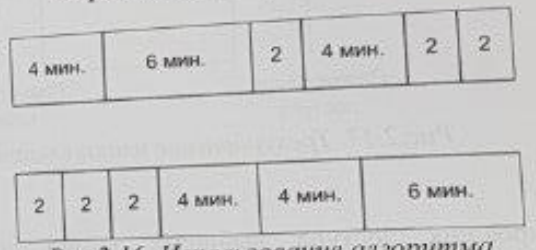


Рис.2.16. Использование алгоритма

Нижняя очередь выстроена с учетом этого алгоритма

Преимущества:

- Уменьшение оборотного времени
- Справедливость (как в очереди покупателей, кто без сдачи проходит вперед)

Недостатки:

- Длинный процесс, занявший процессор, не пустит более новые краткие процессы, которые пришли позже.

этого он приостанавливается по таймеру, чтобы передать управление планировщику.

Необходимость алгоритма планирования зависит от задач, для которых будет использоваться операционная система.

Основные три системы:

1. Системы пакетной обработки - могут использовать неприоритетный и приоритетный алгоритм (например: для расчетных программ).

2. Интерактивные системы - могут использовать только приоритетный алгоритм, нельзя допустить чтобы один процесс занял надолго процессор (например: сервер общего доступа или персональный компьютер).

3. Системы реального времени - могут использовать неприоритетный и приоритетный алгоритм (например: система управления автомобилем).

Задачи алгоритмов планирования:

1. **Для всех систем**

Справедливость - каждому процессу справедливую долю процессорного времени

Контроль над выполнением принятой политики

Баланс - поддержка занятости всех частей системы (например: чтобы были заняты процессор и устройства ввода/вывода)

2. **Системы пакетной обработки**

Пропускная способность - количество задач в час

Оборотное время - минимизация времени на ожидание обслуживания и обработку задач.

Использование процесса - чтобы процессор всегда был занят.

3. **Интерактивные системы**

Время отклика - быстрая реакция на запросы

Соразмерность - выполнение ожиданий пользователя (например: пользователь не готов к долгой загрузке системы)

4. **Системы реального времени**

Окончание работы к сроку - предотвращение потери данных

Предсказуемость - предотвращение деградации качества в мультимедийных системах (например: потерь качества звука должно быть меньше чем видео).

Планирование в системах пакетной обработки

"Первый пришел - первым обслужен" (FIFO - First In Fist Out)

Процессы ставятся в очередь по мере поступления.

Преимущества:

- Простата
- Справедливость (как в очереди покупателей, кто последний пришел, тот оказался в конце очереди)

Недостатки:

- Процесс, ограниченный возможностями процессора может затормозить более быстрые процессы, ограниченные устройствами ввода/вывода.

"Кратчайшая задача - первая"

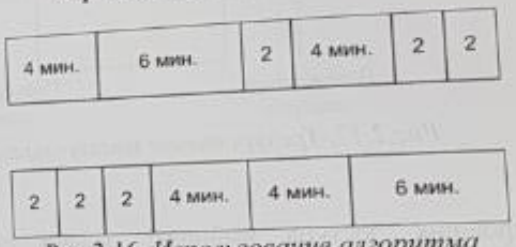


Рис.2.16. Использование алгоритма

Нижняя очередь выстроена с учетом этого алгоритма

Преимущества:

- Уменьшение оборотного времени
- Справедливость (как в очереди покупателей, кто без сдачи проходит вперед)

Недостатки:

- Длинный процесс, занявший процессор, не пустит более новые краткие процессы, которые пришли позже.

Наименьшее оставшееся время выполнение

Аналог предыдущего, но если приходит новый процесс, его полное время выполнения сравнивается с оставшимся временем выполнения текущего процесса.

Трехуровневое планирование

Рис.2.17. Трехуровневое планирование

Планировщик доступа выбирает задачи оптимальным образом(например: процессы, ограниченные процессором и вводом/выводом).

Если процессов в памяти слишком много, планировщик памяти выгружает и загружает некоторые процессы на диск. Количество процессов находящихся в памяти, называется **степенью многозадачности**.

Планирование в интерактивных системах

Циклическое планирование

Самый простой алгоритм планирования и часто используемый.

Каждому процессу предоставляется квант времени процессора. Когда квант заканчивается процесс переводится

планировщиком в конец очереди. При блокировке процессор выпадает из очереди.

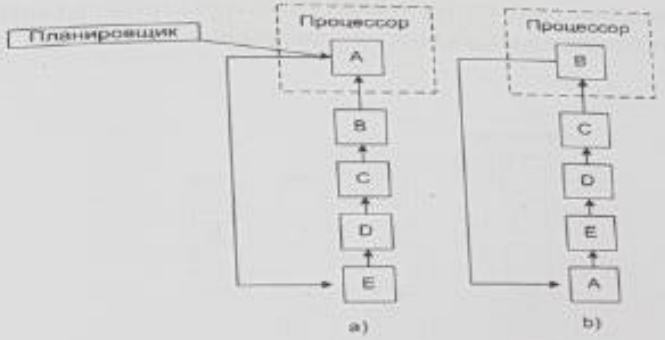


Рис.2.18. Пример циклического планирования

Преимущества:

- Простата
- Справедливость (как в очереди покупателей, каждому только по килограмму)

Недостатки:

- Если частые переключения (квант - 4мс, а время переключения равно 1мс), то происходит уменьшение производительности.
- Если редкие переключения (квант - 100мс, а время переключения равно 1мс), то происходит увеличение времени ответа на запрос.

Приоритетное планирование

Каждому процессу присваивается **приоритет**, и управление передается процессу с самым высоким приоритетом.

Приоритет может быть динамический и статический.

Динамический приоритет может устанавливаться так:

П=1/Т, где Т- часть использованного в последний раз кванта

Если использовано 1/50 кванта, то приоритет 50.

Если использован весь квант, то приоритет 1.

Т.е. процессы, ограниченные вводом/вывода, будут иметь приоритет над процессами ограниченными процессором.

Часто процессы объединяют по приоритетам в группы, и используют приоритетное планирование среди групп, но внутри группы используют циклическое планирование.

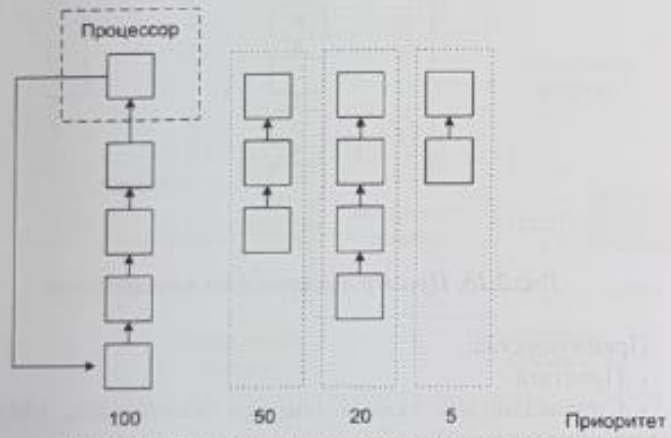


Рис.2.19. Приоритетное планирование 4-х групп

Методы разделения процессов *на группы*

Группы с разным квантом времени

Сначала процесс попадает в группу с наибольшим приоритетом и наименьшим квантом времени, если он использует весь квант, то попадает во вторую группу и т.д. Самые длинные процессы оказываются в группе наименьшего приоритета и наибольшего кванта времени.

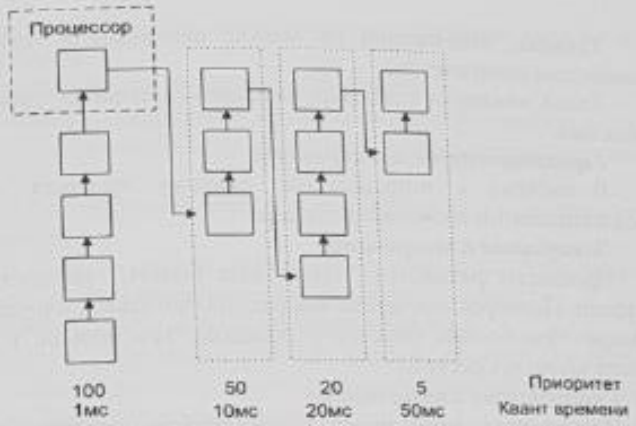


Рис.2.20. Кратчайшая задача - первая

Процесс либо заканчивает работу, либо переходит в другую группу

Этот метод напоминает алгоритм - "Кратчайшая задача - первая".

Группы с разным назначением процессов

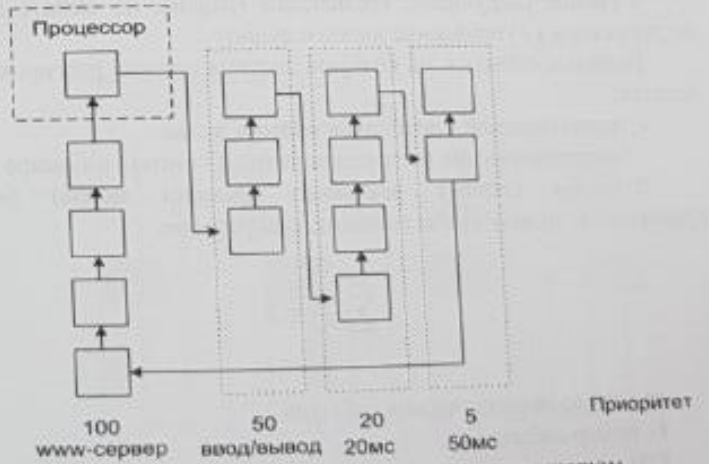


Рис.2.21. Группа с наивысшим приоритетом

Процесс, отвечающий на запрос, переходит в группу с наивысшим приоритетом.

Такой механизм позволяет повысить приоритет работы с клиентом.

Гарантированное планирование

В системе с n-процессами, каждому процессу будет предоставлено 1/n времени процессора.

Лотерейное планирование

Процессам раздаются "лотерейные билеты" на доступ к ресурсам. Планировщик может выбрать любой билет, случайным образом. Чем больше билетов у процесса, тем больше у него шансов захватить ресурс.

Справедливое планирование

Процессорное время распределяется среди пользователей, а не процессов. Это справедливо если у одного пользователя несколько процессов, а у другого один.

Планирование в системах реального времени

Системы реального времени делятся на:

- жесткие (жесткие сроки для каждой задачи) - управление движением
- гибкие (нарушение временного графика не желательны, но допустимы) - управление видео и аудио

Внешние события, на которые система должна реагировать, делятся:

- периодические - потоковое видео и аудио
- непериодические (непредсказуемые) - сигнал о пожаре

Что бы систему реального времени можно было планировать, нужно чтобы выполнялось условие:

$$\sum_{i=1}^{m} \frac{T(i)}{P(i)} \le 1$$

m - число периодических событий
i - номер события
P(i) - период поступления события
T(i) - время, которое уходит на обработку события

Т.е. перегруженная система реального времени является не **планируемой**.

Планирование однородных процессов

В качестве однородных процессов можно рассмотреть видео сервер с несколькими видео потоками (несколько пользователей смотрят фильм).

Т.к. все процессы важны, можно использовать циклическое планирование.

Но так как количество пользователей и размеры кадров могут меняться, для реальных систем он не подходит.

Общее планирование реального времени

Используется модель, когда каждый процесс борется за процессор со своим заданием и графиком его выполнения.

Планировщик должен знать:

- частоту, с которой должен работать каждый процесс
- объем работ, который ему предстоит выполнить
- ближайший срок выполнения очередной порции задания

Рассмотрим пример из трех процессов.

Процесс **A** запускается каждые 30мс, обработка кадра 10мс

Процесс **B** частота 25 кадров, т.е. каждые 40мс, обработка кадра 15мс

Процесс **C** частота 20 кадров, т.е. каждые 50мс, обработка кадра 5мс

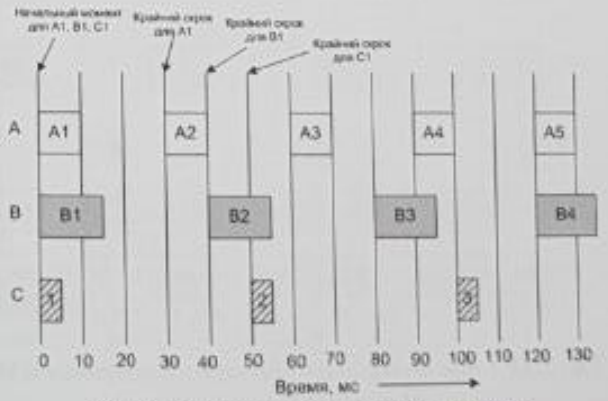


Рис.2.22. Три периодических процесса

Проверяем, можно ли планировать эти процессы.

10/30+15/40+5/50=0.808<1

Условие выполняется, планировать можно.

Будем планировать эти процессы **статическим**(приоритет заранее назначается каждому процессу) и **динамическим** методами.

Статический алгоритм планирования RMS (Rate Monotonic Scheduling)

Процессы должны удовлетворять условиям:

- Процесс должен быть завершен за время его периода
- Один процесс не должен зависеть от другого
- Каждому процессу требуется одинаковое процессорное время на каждом интервале
- У непериодических процессов нет жестких сроков
- Прерывание процесса происходит мгновенно

Приоритет в этом алгоритме пропорционален частоте.

Процессу A он равен 33 (частота кадров)

Процессу B он равен 25

Процессу C он равен 20

Процессы выполняются по приоритету.

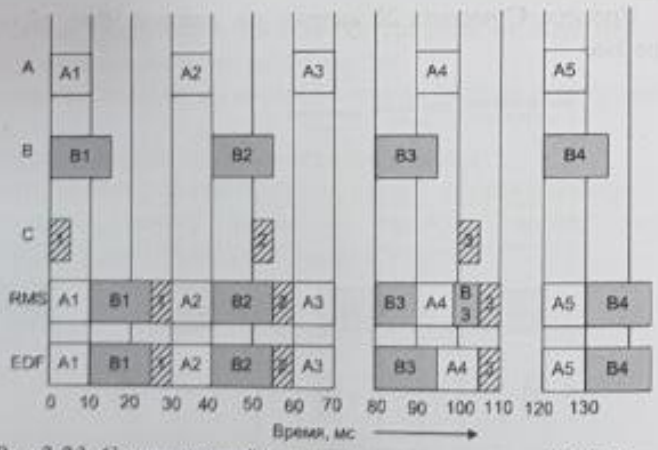


Рис.2.23. Статический алгоритм планирования RMS (Rate Monotonic Scheduling)

Динамический алгоритм планирования EDF (Earliest Deadline First)

Наибольший приоритет выставляется процессу, у которого осталось наименьшее время выполнения.

При больших загрузках системы **EDF** имеет преимущества.

Рассмотрим пример, когда процессу A требуется для обработки кадра - 15мс.

Проверяем, можно ли планировать эти процессы.

15/30+15/40+5/50=0.975<1

Загрузка системы 97.5%

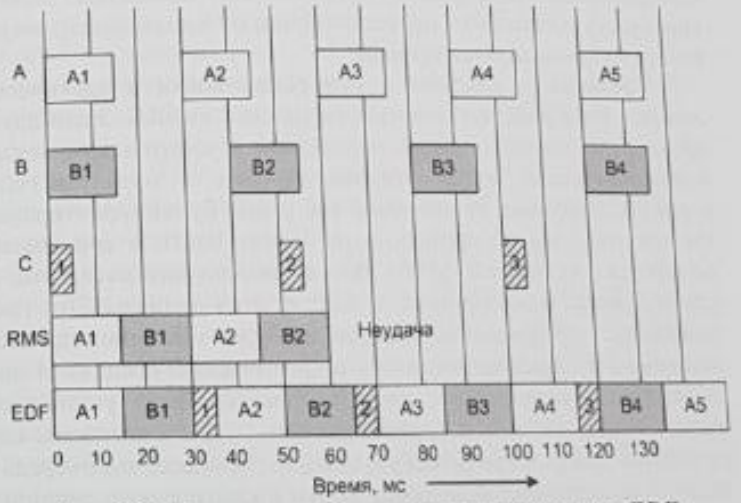


Рис.2.24. Динамический алгоритм планирования EDF (Earliest Deadline First)

Алгоритм планирования **RMS** терпит неудачу.

Алгоритмы синхронизации (алгоритмы корректной организации взаимодействия процессов)

Критическая секция

Критическая секция – часть программы, результат выполнения которой может непредсказуемо меняться, если

переменные, относящиеся к ней, изменяются другими потоками в то время, когда выполнение этой части еще не завершено. В примере критическая секция – файл "заказов", являющийся разделяемым ресурсом для процессов R и S[18,19].

Алгоритм Деккера - первое известное корректное решение проблемы взаимного исключения.

Если два процесса пытаются перейти в критическую секцию одновременно, алгоритм позволит это только одному из них, основываясь на том, чья в этот момент очередь. Если один процесс уже вошёл в критическую секцию, другой будет ждать, пока первый покинет её. Это реализуется при помощи использования двух флагов (индикаторов "намерения" войти в критическую секцию) и переменной turn (показывающей, очередь какого из процессов наступила).

Процессы объявляют о намерении войти в критическую секцию; это проверяется внешним циклом «while». Если другой процесс не заявил о таком намерении, в критическую секцию можно безопасно войти (вне зависимости от того, чья сейчас очередь). Взаимное исключение всё равно будет гарантировано, так как ни один из процессов не может войти в критическую секцию до установки этого флага (подразумевается, что, по крайней мере, один процесс войдёт в цикл «while»). Это также гарантирует продвижение, так как не будет ожидания процесса, оставившего «намерение» войти в критическую секцию. В ином случае, если переменная другого процесса была установлена, входят в цикл «while» и переменная turn будет показывать, кому разрешено войти в критическую секцию. Процесс, чья очередь не наступила, оставляет намерение войти в критическую секцию до тех пор, пока не придёт его очередь (внутренний цикл «while»). Процесс, чья очередь пришла, выйдет из цикла «while» и войдёт в критическую секцию.

+ не требует специальных Test-and-set инструкций, по этому легко переносим на разные языки программирования и архитектуры компьютеров

-Действует только для двух процессов

Алгоритм Петерсона — программный алгоритм взаимного исключения потоков исполнения кода.

Перед тем как начать исполнение критической секции кода (то есть кода, обращающегося к защищаемым совместно используемым ресурсам), поток должен вызвать специальную процедуру (назовем ее EnterRegion) со своим номером в качестве параметра. Она должна организовать ожидание потока своей очереди входа в критическую секцию. После исполнения критической секции и выхода из нее, поток вызывает другую процедуру (назовем ее LeaveRegion), после чего уже другие потоки смогут войти в критическую область. 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.

-Как и алгоритм Деккера, действует только для 2 процессов

+Более простая реализация, чем у алгоритма Деккера

Алгоритм булочной. Алгоритм Петерсона дает нам решение задачи корректной организации взаимодействия двух процессов. Давайте рассмотрим теперь соответствующий алгоритм для n взаимодействующих процессов.

Каждый вновь прибывающий процесс получает метку с номером. Процесс с наименьшим номером метки обслуживается следующим. К сожалению, из-за неатомарности операции вычисления следующего номера алгоритм булочной не гарантирует, что у всех процессов будут метки с разными номерами. В случае равенства номеров меток у двух или более процессов первым обслуживается клиент с меньшим значением имени (имена можно сравнивать в лексикографическом порядке). Разделяемые структуры данных для алгоритма – это два массива

Специальные механизмы синхронизации – семафоры Дейкстры, мониторы Хора, очереди сообщений.

Семафоры

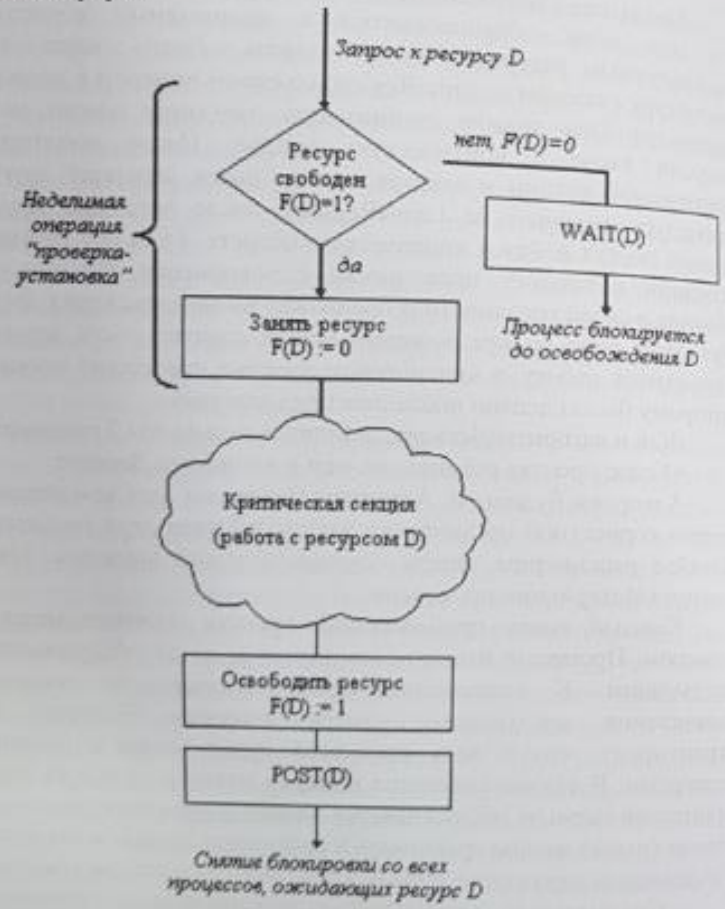


Рис.2.25. Семафоры

Для устранения этого недостатка во многих ОС предусматриваются специальные системные вызовы (аппарат для работы с критическими секциями).

В разных ОС аппарат событий реализован по своему, но в любом случае используются системные функции, которые

условно называют WAIT(x) и POST(x), где x – идентификатор некоторого события (например, освобождение ресурса).

Обобщающее средство синхронизации процессов предложил Дейкстра, который ввел новые примитивы, обозначаемые V ("открытие") и P ("закрытие"), оперирующие над целыми неотрицательными переменными, называемыми семафорами.

Доступ любого процесса к семафору, за исключением момента его инициализации, может осуществляться только через эти две атомарные операции.

Смысл P(S) заключается в проверке текущего значения семафора S, и если S>0, то осуществляется переход к следующей за примитивом операции, иначе процесс переходит в состояние ожидания.

P(S):

Пока S==0

Процесс блокируется;

S=S-1;

Операция V(S) связана с увеличением значения S на 1 и переводом одного или нескольких процессов в состояние готовности к исполнению процессором.

V(S):

S=S+1;

В простом случае, когда семафор работает в режиме 2-х состояний (S>0 и S=0), его алгоритм работы полностью совпадает с алгоритмом работы мьютекса, а S выполняет роль блокирующей переменной.

"+": пассивное ожидание (постановка в очередь и автоматическая выдача ресурсов)

· возможность управления группой однородных ресурсов

"-": не указывают непосредственно на критический ресурс

· некорректное использование операций может привести к нарушению работоспособности (например, переставив местами операции P(e) и P(b) в функции Writer()).

Мониторы

Для облегчения работы программистов при создании параллельных программ без усилий на доказательства правильности алгоритмов и отслеживание взаимосвязанных

объектов (что характерно при использовании семафоров) предложено высокоуровневое средство синхронизации, называемое мониторами.

Мониторы – тип данных, обладающий собственными переменными, значения которых могут быть изменены только с помощью вызова функций-методов монитора.

Функции-методы могут использовать в работе только данные, находящиеся внутри монитора, и свои параметры.

Доступ к мониторам в каждый момент времени имеет только один процесс.

Для организации не только взаимоисключений, но и очередности процессов, подобно семафорам f(full) и e(empty), было введено понятие условных переменных, над которыми можно совершать две операции wait и signal, отчасти похожие на операции P и V над семафорами.

Функция монитора 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 совершает операцию signal над той же самой условной переменной. Это приводит к пробуждению ранее заблокированного процесса, и он становится активным.

Исключение входа нескольких процессов в монитор реализуется компилятором, а не программистом, что делает ошибки менее вероятными.

Требуются специальные языки программирования и компиляторы (встречаются в языках, "параллельный Евклид","параллельный Паскаль",Java).

Следует отметить, что условные переменные мониторов не запоминают предысторию, поэтому операцию signal всегда должна выполняться после операции wait(иначе выполнение операции wait всегда будет приводить к блокированию процесса).

Очереди сообщений

Механизм очередей сообщений позволяет процессам и потокам обмениваться структурированными сообщениями. Один или несколько процессов независимым образом могут посылать сообщения процессу – приемнику.

Очередь сообщений представляет возможность использовать несколько дисциплин обработки сообщений (FIFO, LIFO, приоритетный доступ, произвольный доступ).

При чтении сообщения из очереди удаления сообщения из очереди не происходит, и сообщение может быть прочитано несколько раз.

В очереди присутствуют не сами сообщения, а их адреса в памяти и размер. Эта информация размещается системой в сегменте памяти, доступном для всех задач, общающихся с помощью данной очереди

Основные функции управления очередью:

- Создание новой очереди
- Открытие существующей очереди
- Чтение и удаление сообщений из очереди
- Чтение без последующего удаления
- Добавление сообщения в очередь
- Завершение использование очереди
- Удаление из очереди всех сообщений
- Определение числа элементов в очереди

Задача взаимного исключения

Если двум или более процессам необходимо взаимодействовать друг с другом, то они должны быть связаны, то есть иметь средства для обмена информацией. Предполагается, что процессы связаны слабо. Под этим подразумевается, что кроме достаточно редких моментов явной связи эти процессы рассматриваются как совершенно независимые друг от друга. В частности, не допускаются какие либо предположения об относительных скоростях различных процессов.

В качестве примера рассматривается два последовательных процесса, которые удобно считать циклическими. В каждом цикле выполнения процесса существует критический интервал. Это означает, что в любой момент времени только один процесс может находиться внутри своего критического интервала. Чтобы осуществить такое взаимное исключение, оба процесса имеют доступ к некоторому числу общих переменных. Операции проверки текущего значения такой общей переменной и присваивание нового значения общей переменной рассматриваются как неделимые. То есть, если два процесса

осуществляют присваивание новое значение одной и той же общей переменной одновременно, то присваивание происходит друг за другом и окончательное значение пере-менной одному из присвоенных значений, но никак не их смеси. Если процесс поверяет значение переменной одновременно с присваиванием ей значения другим процессом, то первый процесс обнаруживает либо старое, либо новое значение, но никак не их смесь.

```
    begin integer очередь; очередь := 1;
parbegin
    процесс 1: begin
    L1: if (очередь = 2) then goto L1;
критический интервал 1; очередь := 2;
    остаток цикла 1; goto L1;
    end;
    процесс 2: begin
    L2: if (очередь = 1) then goto L2;
критический интервал 2; очередь := 1;
    остаток цикла 2; goto L2;
    end; parend; end;
    int turn=1; void P0() {
    while (1) {
    while(turn!=0) критический интервал 1;
turn=1;
    …. }
    }
    void P1() {
    while (1) {
    while(turn!=1) критический интервал 2;
turn=0;
    …. }
    }
    void main() {
    parbegin(P0,P1); }
```

Недостатки решения:

1. Процессы могут входить в критический интервал строго последователь-но. Темпы развития задаются медленным процессом.

2. Если кто-то из процессов останется в остатке цикла, то он затормозит и второй процесс

Второй способ решения:

```
    begin integer C1,C2; C1 := 1;
    C2 := 1; parbegin
    процесс 1: begin
    L1: if (C2 = 0) then goto L1; C1 := 0;
    критический интервал 1; C1 := 1;
    остаток цикла 1; goto L1;
    end;
    процесс 2: beginL2: if (C1 = 0) then goto
L2; C2 := 0;
    критический интервал 2; C2 := 1;
    остаток цикла 2; goto L2;
    end; parend; end;
    int flag[2]; void P0() {
    while (1) {
    while (flag[1]); flag[0]=1;
    критический интервал 1; flag[0]=0;
    …. }
    }
    void P1() {
    while (1) {
    while (flag[0]); flag[1]=1;
    критический интервал 2; flag[1]=0;
    …. }
    }
    void main() {

    flag[0]=0; flag[1]=0; parbegin(P0,P1);
    }
```

Недостаток: принципиально при развитии процессов строго синхронно они могут одновременно войти в критический интервал.

Было предложено следующее решение (вариант 3)::

```
    begin integer C1,C2; C1 := 1;
    C2 := 1; parbegin
```

```
процесс 1: begin A1: C1 := 0;
L1: if (C2 = 0) then goto L1; критический
интервал 1; C1 := 1;
остаток цикла 1; goto A1;
end;
процесс 2: begin A2: C2 := 0;
L2 if (C1 = 0) then goto L2; критический
интервал 2; C2 := 1;
остаток цикла 2; goto A2;
end; parend; end;
int flag[2]; void P0() {
while (1) {
flag[0]=1; while (flag[1]);
критический интервал 1; flag[0]=0;
} }
void P1() {
while (1) {
flag[1]=1; while (flag[0]);
критический интервал 2; flag[1]=0;
…. }
}
void main() {
flag[0]=0; flag[1]=0; parbegin(P0,P1);
}
```

Недостаток: возникает другая проблема – может бесконечно долго решаться вопрос о том, кто первым войдет в критический интервал.

Решение 4.

```
begin integer C1,C2; C1 := 1;
C2 := 1; parbegin
процесс 1: begin L1: C1 := 0;
if (C2 = 0) then begin
C1 := 1; goto L1;
end; критический интервал 1; C1 := 1;
остаток цикла 1; goto L1;
end;
процесс 2: begin
L2: C2 := 0;
```



```
if (C1 = 0) then begin
C2 := 1;

goto L2; end;
критический интервал 2; C2 := 1;
остаток цикла 2; goto L2;
end; parend; end;
int flag[2]; void P0() {
while (1) {
flag[0]=1; while (flag[1]); {
flag[0]=0; задержка; flag[0]=1;
}
критический интервал 1; flag[0]=0;
…. }
}
void P1() {
while (1) {
flag[1]=1; while (flag[0]); {
flag[1]=0; задержка; flag[1]=1;
}
критический интервал 2; flag[1]=0;
…. }
}
void main() {
flag[0]=0; flag[1]=0; parbegin(P0,P1);
}
```

Решение задачи взаимного исключения. Алгоритм Деккера.

```
begin integer C1,C2,очередь; C1 := 1;
C2 := 1; очередь := 1; parbegin
процесс 1: begin A1: C1 := 0;
L1: if (C2 = 0) then begin
if (очередь = 1) then goto L1; C1 := 1;
B1: if (очередь = 2) then goto B1; goto A1;
end; критический интервал 1; очередь := 2;
C1 := 1;
остаток цикла 1; goto A1;
end;
процесс 2: begin A2: C2 := 0;
```

```
L2: if (C1 = 0) then begin
if (очередь = 2) then goto L2; C2 := 1;
B2: if (очередь = 1) then goto B2; goto A2;
end; критический интервал 2; очередь := 1;
C2 := 1;
остаток цикла 2; goto A2;
end; parend; end;
int flag[2], turn; void P0()
{
while (1) {
flag[0]=1; while (flag[1]); {
if (turn==1) {
flag[0]=0;
while (turn==1); flag[0]=1;
} }
критический интервал 1; turn=1;
flag[0]=0; ….
} }
```

```
void P1() {
while (1) {
flag[1]=1; while (flag[0]); {
if (turn==0) {
flag[1]=0;
while (turn==0); flag[1]=1;
} }
критический интервал 2; turn=0;
flag[1]=0; ….
} }
void main()
{ flag[0]=0; flag[1]=0; turn=1;
parbegin(P0,P1); }
```

Решение задачи взаимного исключения. Алгоритм Петерсона..

```
int flag[2], turn; void P0()
{
```



```
while (1) {
flag[0]=1; turn=1;
while (flag[1] && (turn==1)); критический
интервал 1; flag[0]=0;
…. }
}
void P1() {
while (1) {
flag[1]=1; turn=0;
while (flag[0] && (turn==0)); критический
интервал 2; flag[1]=0;
…. }
}
void main() {
flag[0]=0; flag[1]=0; parbegin(P0,P1); }
```

Для доказательства корректности задачи взаимного исключения необходимо проверить три положения.

1. Решение безопасно в том смысле, что два процесса не могут одновременно оказаться в своих критических интервалах

2. В случае сомнения, кому из двух процессов первому войти в критический интервал, выяснение этого вопроса не откладывается до бесконечности

3. Остановка какого-либо из процессов в остатке цикла не вызывает блокировки другого процесса.

Взаимоблокировки, тупиковые ситуации, «зависания» системы

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[20-22].

Взаимоблокировка возникает, когда две и более задач постоянно блокируют друг друга в ситуации, когда у каждой задачи заблокирован ресурс, который пытаются заблокировать другие задачи. Например:

- Транзакция А создает общую блокировку строки 1.
- Транзакция Б создает общую блокировку строки 2.
- Транзакция А теперь запрашивает монопольную блокировку строки 2 и блокируется до того, как транзакция Б закончится и освободит общую блокировку строки 2.
- Транзакция Б теперь запрашивает монопольную блокировку строки 1 и блокируется до того, как транзакция А закончится и освободит общую блокировку строки 1.

Транзакция А не может завершиться до того, как завершится транзакция Б, а транзакция Б заблокирована транзакцией А. Такое условие также называется цикличной зависимостью: транзакция А зависит от транзакции Б, а транзакция Б зависит от транзакции А и этим замыкает цикл.

Тупик возникает при перестановке местами операций P(e) и P(b) в примере с процессами "читатель" и "писатель", рассмотренном выше – ни один из этих потоков не сможет завершить начатую работу и возникнет тупиковая ситуация, которая не может разрешиться без внешнего воздействия.

Тупиковые ситуации следует отличать от простых очередей хотя те и другие возникают при совместном использовании ресурсов и внешне выглядят схоже.

Однако очередь – неотъемлемый признак высокого коэффициента использования ресурсов при случайном поступлении запросов а тупик – "неразрешимая" ситуация.

Проблема тупиков требует решения следующих задач:

-распознавание тупиков,

-предотвращение тупиков,

-восстановление системы после тупиков.

Распознавание тупиков

В случаях, когда тупиковая ситуация образована многими процессами, использующими много ресурсов, распознавание

тупика является не тривиальной задачей. Существуют формальные, программно-реализованные методы распознавания тупиков, основанные на ведении таблиц распределения ресурсов и таблиц запросов к занятым ресурсам, анализ которых позволяет обнаружить взаимные блокировки.

2.1.3. Управление памятью

Функции ОС по управлению памятью

Под памятью (*memory*) в данном случае подразумевается оперативная (основная) *память* компьютера. В однопрограммных операционных системах основная *память* разделяется на две части. Одна часть для операционной системы (резидентный *монитор*, *ядро*), а вторая – для выполняющейся в текущий момент времени программы. В многопрограммных ОС "пользовательская" часть памяти – важнейший *ресурс* вычислительной системы – должна быть распределена для размещения нескольких процессов, в том числе процессов ОС. Эта задача распределения выполняется операционной системой динамически специальной подсистемой управления памятью (*memory management*). Эффективное *управление памятью* жизненно важно для многозадачных систем. Если в памяти будет находиться небольшое число процессов, то значительную часть времени процессы будут находиться в состоянии ожидания ввода-вывода и *загрузка* процессора будет низкой[12,13].

В ранних ОС *управление памятью* сводилось просто к загрузке программы и ее данных из некоторого внешнего накопителя (перфоленты, магнитной ленты или магнитного диска) в *ОЗУ*. При этом *память* разделялась между программой и ОС. На рис. 2.34 показаны три варианта такой схемы. Первая модель раньше применялась на мэйнфреймах и мини-компьютерах. Вторая схема сейчас используется на некоторых карманных компьютерах и встроенных системах, третья модель была характерна для ранних персональных компьютеров с *MS-DOS*.

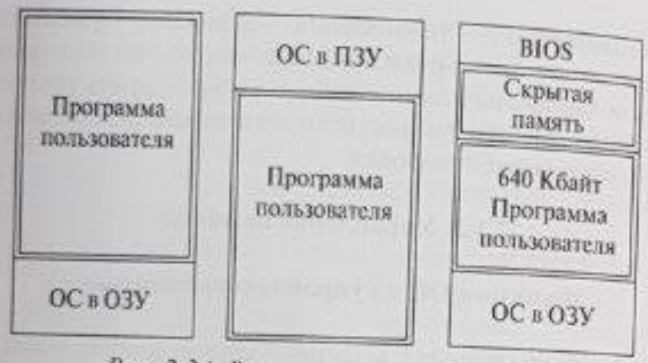


Рис. 2.34. Варианты распределения памяти

С появлением мультипрограммирования задачи ОС, связанные с распределением имеющейся памяти между несколькими одновременно выполняющимися программами, существенно усложнились[7,8].

Функциями ОС *по* управлению памятью в мультипрограммных системах являются:

- отслеживание (учет) свободной и занятой памяти;
- первоначальное и *динамическое выделение памяти* процессам приложений и самой операционной системе и освобождение памяти по завершении процессов;
- настройка адресов программы на конкретную область физической памяти;
- полное или частичное вытеснение кодов и данных процессов из ОП на диск, когда размеры ОП недостаточны для размещения всех процессов, и возвращение их в ОП;
- защита памяти, выделенной процессу, от возможных вмешательств со стороны других процессов;
- *дефрагментация памяти*.

Перечисленные функции особого пояснения не требуют, остановимся только на задаче преобразования адресов программы при ее загрузке в ОП.

Для идентификации переменных и команд на разных этапах жизненного *цикла* программы используются символьные имена, виртуальные (математические, условные, логические – все это синонимы) и физические адреса (рис. 2.35).

Рис. 2.35. Типы адресов

Символьные имена присваивает *пользователь* при написании программ на алгоритмическом языке или ассемблере. Виртуальные адреса вырабатывает *транслятор*, переводящий программу на *машинный язык*. Поскольку во время трансляции неизвестно, в какое *место* оперативной памяти будет загружена *программа*, *транслятор* присваивает переменным и командам виртуальные (условные) адреса, считая *по* умолчанию, что начальным адресом программы будет нулевой *адрес*.

Физические адреса соответствуют номерам ячеек оперативной памяти, где в действительности будут расположены переменные и команды.

Совокупность виртуальных адресов процесса называется виртуальным адресным пространством. *Диапазон* адресов виртуального пространства у всех процессов один и тот же и определяется разрядностью адреса процессора (для Pentium *адресное пространство* составляет объем, равный 2^{32} *байт*, с диапазоном адресов от 0000.0000_{16} до $FFFF.FFFF_{16}$).

Существует два принципиально отличающихся подхода к преобразованию виртуальных адресов в физические. В первом случае такое преобразование выполняется один раз для каждого процесса во время начальной загрузки программы в *память*.

Преобразование осуществляет перемещающий *загрузчик* на основании имеющихся у него данных о начальном адресе физической памяти, в которую предстоит загружать программу, а также информации, предоставляемой транслятором об адресно-зависимых элементах программы.

Второй способ заключается в том, что *программа* загружается в *память* в виртуальных адресах. Во *время выполнения* программы при каждом обращении к памяти *операционная система* преобразует виртуальные адреса в физические.

Распределение памяти

Существует ряд базовых вопросов управления памятью, которые в различных ОС решаются *по*-разному. Например, следует ли назначать каждому процессу одну непрерывную область физической памяти или можно выделять *память* участками? Должны ли *сегменты* программы, загруженные в *память*, находиться на одном месте в течение всего периода выполнения процесса или их можно время от времени сдвигать? Что делать, если *сегменты* программы не помещаются в имеющуюся *память*? Как сократить *затраты* ресурсов системы на *управление памятью*? Имеется и ряд других не менее интересных проблем управления памятью [5, 10, 13, 17].

Ниже приводится классификация методов распределения памяти, в которой выделено два класса методов – с перемещением сегментов процессов между ОП и ВП (диском) и без перемещения, т.е. без привлечения внешней памяти (рис. 2.36). Данная классификация учитывает только основные признаки методов. Для каждого метода может быть использовано несколько различных алгоритмов его реализации.

Рис. 2.36. Классификация методов распределения памяти

На рис. 2.37 показаны два примера фиксированного распределения. Одна возможность состоит в использовании разделов одинакового размера. В этом случае любой процесс, размер которого не превышает размера раздела, может быть загружен в любой доступный раздел. Если все *разделы* заняты и нет ни одного процесса в состоянии готовности или работы, ОС может выгрузить процесс из любого раздела и загрузить другой процесс, обеспечивая тем самым *процессор* работой.

Рис. 2.37. Варианты фиксированного распределения памяти

При использовании разделов с одинаковым размером имеются две проблемы.

1. Программа может быть слишком велика для размещения в разделе. В этом случае программист должен разрабатывать программу, использующую оверлеи, чтобы в любой момент времени требовался только один раздел памяти. Когда требуется модуль, отсутствующий в данный момент в ОП, пользовательская программа должна сама его загрузить в раздел памяти программы. Таким образом, в данном случае управление памятью во многом возлагается на программиста.

2. Использование ОП крайне неэффективно. Любая программа, независимо от ее размера, занимает раздел целиком. При этом могут оставаться неиспользованные участки памяти большого размера. Этот феномен появления неиспользованной памяти называется внутренней фрагментацией (internal fragmentation).

Бороться с этими трудностями (хотя и не устранить полностью) можно посредством использования разделов разных размеров. В этом случае *программа* размером до 8 Мбайт может

обойтись без оверлеев, а *разделы* малого размера позволяют уменьшить внутреннюю фрагментацию при загрузке небольших программ.

В том случае, когда *разделы* имеют одинаковый раздел, *размещение* процессов тривиально – в любой свободный раздел. Если все *разделы* заняты процессами, которые не готовы к немедленной работе, любой из них может быть выгружен для освобождения памяти для нового процесса.

Когда *разделы* имеют разные размеры, есть два возможных подхода к назначению процессов разделам памяти. Простейший *путь* состоит в том, чтобы каждый процесс размещался в наименьшем разделе, способном вместить данный процесс (в этом случае в задании пользователя указывался размер требуемой памяти). При таком подходе для каждого раздела требуется *очередь* планировщика, в которой хранятся выгруженные из памяти процессы, предназначенные для данного раздела памяти. Достоинство такого способа в возможности распределения процессов между разделами ОП так, чтобы минимизировать внутреннюю фрагментацию[13,15,16].

Недостаток заключается в том, что отдельные очереди для разделов могут привести к неоптимальному распределению памяти системы в целом. Например, если в некоторый момент времени нет ни одного процесса размером от 7 до 12 Мбайт, то раздел размером 12 Мбайт будет пустовать, в то время как он мог бы использоваться меньшими процессами. Поэтому более предпочтительным является использование одной очереди для всех процессов. В момент, когда требуется загрузить процесс в ОП, выбирается наименьший доступный раздел, способный вместить данный процесс.

В целом можно отметить, что схемы с *фиксированными разделами* относительно просты, предъявляют минимальные требования к операционной системе; накладные *расходы* работы процессора на *распределение памяти* невелики. Однако у этих схем имеются серьезные недостатки.

1. Количество разделов, определенное в момент генерации системы, ограничивает количество активных процессов (т.е. уровень мультипрограммирования).

2. Поскольку размеры разделов устанавливаются заранее во время генерации системы, небольшие задания приводят к неэффективному использованию памяти. В средах, где заранее известны потребности в памяти всех задач, применение рассмотренной схемы может быть оправдано, но в большинстве случаев эффективность этой технологии крайне низка.

Для преодоления сложностей, связанных с фиксированным распределением, был разработан альтернативный подход, известный как динамическое распределение. В свое время этот подход был применен фирмой *IBM* в операционной системе для мэйнфреймов в OS/MVT (*мультипрограммирование* с переменным числом задач – *Multiprogramming With a Variable* number of Tasks). Позже этот же подход к распределению памяти использован в ОС ЕС ЭВМ [12].

При динамическом распределении образуется переменное количество разделов переменной длины. При размещении процесса в основной памяти для него выделяется строго необходимое количество памяти. В качестве примера рассмотрим использование 64 Мбайт (рис.2.38) основной памяти. Изначально вся *память* пуста, за исключением области, задействованной ОС. Первые три процесса загружаются в *память*, начиная с адреса, где заканчивается ОС, и используют столько памяти, сколько требуется данному процессу. После этого в конце ОП остается свободный участок памяти, слишком малый для размещения четвертого процесса. В некоторый момент времени все процессы в памяти оказываются неактивными, и *операционная система* выгружает второй процесс, после чего остается достаточно памяти для загрузки нового, четвертого процесса.

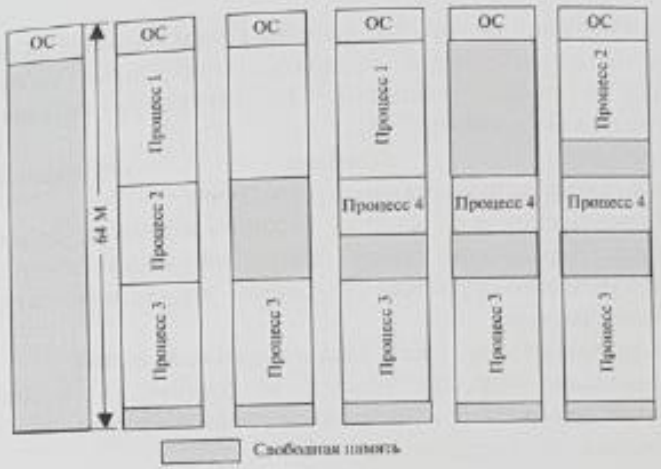


Рис. 2.38. Вариант использования памяти

Поскольку процесс 4 меньше процесса 2, появляется еще свободный участок памяти. После того как в некоторый момент времени все процессы оказались неактивными, но стал готовым к работе процесс 2, свободного места в памяти для него не находится, а ОС вынуждена выгрузить процесс 1, чтобы освободить необходимое *место* и разместить процесс 2 в ОП. Как показывает данный пример, этот метод хорошо начинает работу, но плохо продолжает. В конечном счете, он приводит к наличию *множества* мелких свободных участков памяти, в которых нет возможности разместить какой-либо новый процесс. Это явление называется внешней фрагментацией (*external fragmentation*), что отражает тот факт, что сильно фрагментированной становится *память*, внешняя *по* отношению ко всем разделам.

Один из методов преодоления внешней фрагментации – уплотнение (*compaction*) процессов в ОП. Осуществляется это перемещением всех занятых участков так, чтобы вся свободная *память* образовала единую свободную область. В *дополнение* к функциям, которые ОС выполняет при распределении памяти динамическими разделами, в данном

случае она должна еще время от времени копировать содержимое разделов из одного места в другое, корректируя таблицы свободных и занятых областей. Эта процедура называется уплотнением или сжатием[17,18].

Перечислим функции операционной системы *по* управлению памятью в этом случае.

1. Перемещение всех занятых участков в сторону старших или младших адресов при каждом завершении процесса или для вновь создаваемого процесса в случае отсутствия раздела достаточного размера.
2. Коррекция таблиц свободных и занятых областей.
3. Изменение адресов команд и данных, к которым обращаются процессы при их перемещении в памяти, за счет использования *относительной адресации*.
4. Аппаратная поддержка процесса динамического преобразования относительных адресов в абсолютные адреса основной памяти.
5. Защита памяти, выделяемой процессу, от взаимного влияния других процессов.

Уплотнение может выполняться либо при каждом завершении процесса, либо только тогда, когда для вновь создаваемого процесса нет свободного раздела достаточного размера. В первом случае требуется меньше вычислительной работы при корректировке таблиц свободных и занятых областей, а во втором – реже выполняется процедура сжатия.

Так как *программа* перемещается *по* оперативной памяти в ходе своего выполнения, в данном случае невозможно выполнить настройку адресов с помощью перемещающего загрузчика. Здесь более подходящим оказывается динамическое преобразование адресов. Достоинствами распределения памяти перемещаемыми разделами являются эффективное использование оперативной памяти, *исключение* внутренней и внешней фрагментации, недостатком – дополнительные накладные *расходы* ОС.

При использовании фиксированной схемы распределения процесс всегда будет назначаться одному и тому же разделу памяти после его выгрузки и последующей загрузке в *память*. Это позволяет применять простейший *загрузчик*, который замещает при загрузке процесса все относительные ссылки

абсолютными адресами памяти, определенными на основе базового адреса загруженного процесса.

Ситуация усложняется, если размеры разделов равны (или неравны) и существует единая *очередь* процессов, – процесс *по* ходу работы может занимать разные *разделы*. Такая же ситуация возможна и при динамическом распределении. В этих случаях расположение команд и данных, к которым обращается процесс, не является фиксированным и изменяется всякий раз при выгрузке, загрузке или перемещении процесса. Для решения этой проблемы в программах используются относительные адреса. Это означает, что все ссылки на *память* в загружаемом процессе даются относительно начала этой программы. Таким образом, для корректной работы программы требуется аппаратный механизм, который бы транслировал относительные адреса в физические в процессе выполнения команды, обращающейся к памяти.

Применяемый обычно способ трансляции показан на рис. 2.39. Когда процесс переходит в состояние выполнения, в специальный *регистр* процесса, называемый базовым, загружается начальный *адрес* процесса в основной памяти. Кроме того, используется "граничный" (bounds) *регистр*, в котором содержится *адрес* последней ячейки программы. Эти значения заносятся в регистры при загрузке программы в основную *память*. При выполнении процесса относительные адреса в командах обрабатываются процессором в два этапа. Сначала к относительному адресу прибавляется *значение* базового регистра для получения абсолютного адреса. Затем полученный абсолютный *адрес* сравнивается со значением в граничном регистре. Если полученный абсолютный *адрес* принадлежит данному процессу, *команда* может быть выполнена. В противном случае генерируется соответствующее данной ошибке *прерывание*.

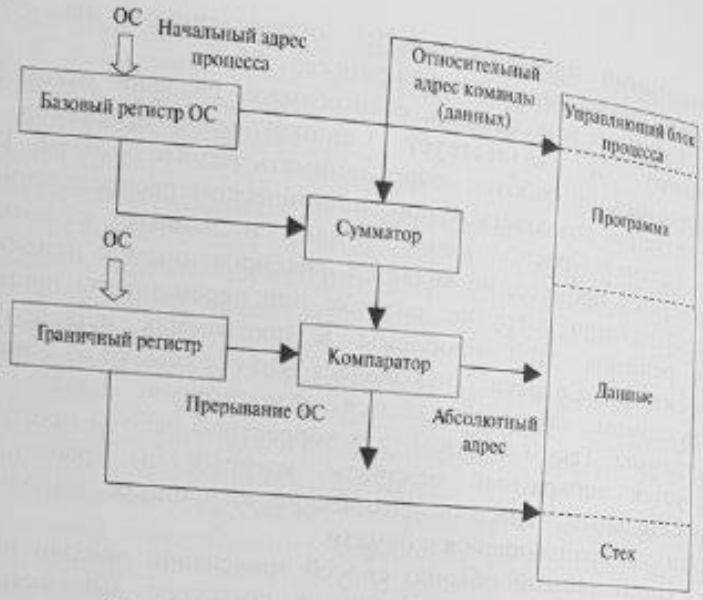


Рис. 2.39. Преобразование адресов

Особенности защищенного режима работы процессора

Адресация памяти в реальном режиме

Для адресации байта памяти в реальном режиме работы используются две 16-разрядные компоненты адреса - сегмент и смещение. Физический адрес, который попадает на шину адреса системной платы компьютера, складывается (в буквальном смысле этого слова) из сдвинутой влево на четыре бита и дополненной справа четырьмя нулевыми битами сегментной компоненты и компоненты смещения. Перед сложением компонента смещения расширяется до 20 бит так, что в старшие четыре бита записываются нули (рис. 2.40).

Рис. 2.40. Получение физического адреса в реальном режиме

Задавая произвольные значения для сегмента и смещения мы можем сконструировать физический адрес для обращения к памяти размером 1 Мбайт плюс 64 Кбайт (и минус 16 байт).

Адрес, состоящий из сегмента и смещения, мы будем называть логическим адресом реального режима. Диапазон логических адресов от 0000h:0000h до FFFFh:000Fh соответствует диапазону физических адресов от 00000h до FFFFFh. Этот диапазон адресов соответствует первому мегабайту оперативной памяти.

Диапазон логических адресов от FFFFh:0010h до FFFFh:FFFFh соответствует так называемой области старшей памяти (High Memory Area). Размер области старшей памяти равен 64 Кбайта без 16 байт, и эта память доступна в реальном режиме для процессора модели 80286 и более старших моделей. Если вы работаете с операционной системой MS-DOS версии 5.0 или 6.2, имеет смысл загрузить ядро MS-DOS в область старших адресов, указав в файле config.sys команду:

DOS=HIGH

Недостатки реального режима работы процессора очевидны. Вы не можете использовать расширенную память, расположенную в адресном пространстве выше области старшей памяти. Если в вашем компьютере установлено 16 Мбайт оперативной памяти, процессор не сможет непосредственно адресовать из них целых 15 Мбайт (рис. 2.41).

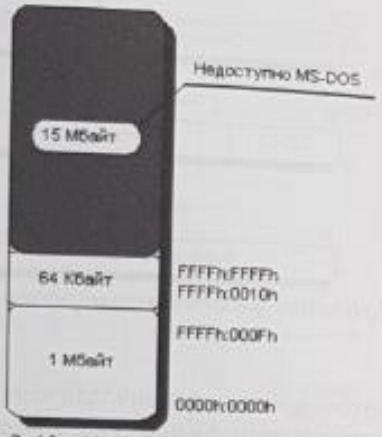


Рис. 2.41. Адресация памяти в MS-DOS

На заре развития персональных компьютеров оперативная память размером в 1 Мбайт считалась достаточно большой для решения любых мыслимых задач. Однако с появлением Windows и внедрением графического пользовательского интерфейса критерии оценки объема памяти резко изменились. Теперь минимальный объем памяти для нормальной работы приложений Windows составляет 4 Мбайта, а для некоторых приложений (например, для системы разработки Borland C++ for Windows версии 4.0 или Microsoft Visual C++) требуется 8 Мбайт или даже 16 Мбайт. Схема адресации реального режима непригодна для работы с такими большими объемами памяти, так как в этой схеме для физического адреса предусмотрено всего 20 разрядов.

Вторым крупным недостатком схемы адресации реального режима является то, что программы, работающие в реальном режиме, имеют полный доступ ко всей адресуемой памяти. Несмотря на то, что в MS-DOS имеются функции управления памятью, с помощью которых программы могут получить в свое распоряжение блоки памяти, ничто не помешает программе выполнить запись за пределами полученного блока или даже в системную область памяти, разрушив MS-DOS.

Если в мультизадачной среде одна задача может писать данные в область памяти, отведенной другой задаче, она может разрушить и эту задачу, и ядро операционной системы. Поэтому

в мультизадачных операционных системах, разработанных для процессоров серии Intel 80xxx или Pentium, применяется только защищенный режим работы процессора.

Адресация памяти в защищенном режиме

В защищенном режиме, как и в реальном, логический адрес состоит из двух компонент. Однако эти компоненты называются не сегмент и смещение, а селектор и смещение. Для вычисления физического адреса в процессоре 80286 используются также две таблицы дескрипторов - глобальная таблица дескрипторов GDT (Global Descriptor Table) и локальная таблица дескрипторов LDT (Local Descroptor Table). Селектор используется для адресации ячейки одной из таблиц дескрипторов, содержащей помимо прочей информации базовый 24-разрядный адрес сегментов. Для получения физического адреса базовый адрес складывается со смещением, расширенным до 24 разрядов (рис. 2.42).

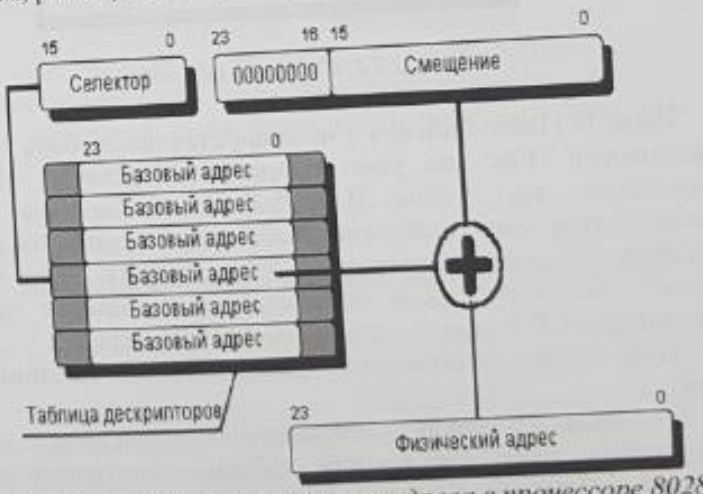


Рис. 2.42. получение физического адреса в процессоре 80286

Согласно этой схеме адресации памяти, селектор содержит номер ячейки таблицы дескрипторов, но не компоненту физического адреса. Программа может задавать не любые

значения селекторов, а только те, которые соответствуют существующим ячейкам таблицы дескрипторов. Разумеется, программа может загрузить в сегментный регистр любое значение, однако при попытке обратиться к сегменту памяти с использованием неправильного селектора работа программы будет прервана.

Таким образом, несмотря на то, что компоненты адреса остались, как и в реальном режиме, 16-разрядными, новая схема адресации защищенного режима процессора 80286 позволяет адресовать до 16 Мбайт памяти, так как в результате преобразования получается 24-разрядный физический адрес.

Кроме индекса, используемого для выбора ячейки дескрипторной таблицы при формировании физического адреса, селектор содержит еще два поля (рис. 2.43).

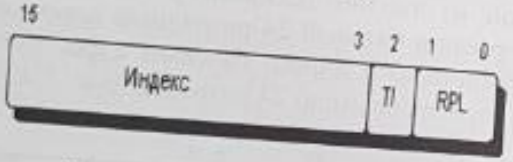


Рис. 2.43. Формат селектора

Поле TI (Table Indicator) используется для выбора таблицы дескрипторов. Как мы уже говорили, существуют таблицы дескрипторов двух типов. В любой момент времени может использоваться одна глобальная таблица дескрипторов и одна локальная таблица дескрипторов. Если бит TI равен 0, для выборки базового адреса используется глобальная таблица дескрипторов GDT, если 1 - локальная LDT[21,22].

Зачем нужно использовать дескрипторные таблицы двух типов?

В мультизадачной операционной системе можно использовать одну глобальную таблицу дескрипторов для описания областей памяти, принадлежащей операционной системе и несколько локальных таблиц дескрипторов для каждой задачи. В этом случае при соответствующей настройке базовых адресов можно изолировать адресные пространства операционной системы и отдельных задач. Если сделать так, что

каждая задача будет пользоваться только своей таблицей дескрипторов, любая задача сможет адресоваться только к своим сегментам памяти, описанным в соответствующей таблице, и к сегментам памяти, описанным в глобальной таблице дескрипторов. В системе может существовать только одна глобальная таблица дескрипторов.

Поле RPL (Requested Privilege Level) селектора содержит уровень привилегий, запрошенный программой при обращении к сегменту. Программа может обращаться только к таким сегментам, которые имеют соответствующий уровень привилегий. Поэтому программа не может, например, воспользоваться глобальной таблицей дескрипторов для получения доступа к описанным в ней системным сегментам, если она не обладает достаточным уровнем привилегий. На этом основана защита системных данных от разрушения (преднамеренного или в результате программной ошибки) со стороны прикладных программ.

Таблица дескрипторов содержит, помимо базового адреса сегмента, другую информацию, описывающую сегмент (рис. 2.44). Точный формат дескриптора, а также других структур данных и системных регистров, имеющих отношение к работе в защищенном режиме, вы можете найти в 6 томе "Библиотеки системного программиста".

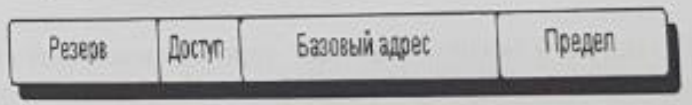


Рис. 2.44. Формат дескриптора сегмента процессора 80286

В частности, дескриптор содержит размер сегмента (предел). При вычислении физического адреса процессор следит за тем, чтобы физический адрес не выходил за пределы, указанные в дескрипторе сегмента. Программа не может обратиться к памяти, лежащей вне пределов, указанных в дескрипторе. Если же она попробует это сделать, ее работа будет прервана. Поэтому, создав, например, сегмент для хранения

массива размером 100 байт, вы не сможете записать в него 101 байт или 10 Кбайт данных.

Поле доступа содержит уровень привилегий сегмента и информацию о типе сегмента. Существуют сегменты кода, сегменты данных и системные сегменты. Кроме того, для сегмента кода можно запретить операцию чтения, а для сегмента данных - операцию записи. Поэтому операционная система может создать сегменты кода, которые можно выполнять, но нельзя читать, и сегменты данных, защищенные от записи.

Что же касается привилегий, в процессорах 80xxx и Pentium существуют четыре уровня привилегий - от 0 до 3, причем наибольшие привилегии соответствуют уровню 0. Уровни привилегий часто называют также кольцами защиты (рис. 2.45).

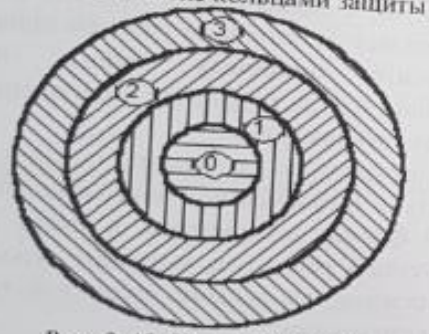


Рис. 2.45. Кольца защиты

В кольце 0 обычно работает ядро операционной системы. Кольцо 1 соответствует уровню привилегий драйверов, кольцо 2 - системам, таким как системы управления базами данных. В наименее привилегированном кольце 3 располагаются прикладные программы, запускаемые пользователем.

Описанная выше схема распределения привилегий может изменяться от одной операционной системы к другой. В операционной системе Windows 3.1 в нулевом кольце располагаются только виртуальные драйверы, все остальные модули Windows, а также приложения, работают в кольце 3.

Процессоры 80386, 80486 и Pentium используют более сложную схему адресации памяти, которая, однако, остается прозрачной для программиста.

Преобразование адреса в этих процессорах является многоступенчатым. Программы адресуют память с помощью логического адреса, состоящего из 16-разрядного селектора и 32-разрядного смещения. Так же, как и в процессоре 80286, селектор используется для выборки базового адреса сегмента из глобальной или локальной таблицы дескрипторов. Отличие заключается в том, что во-первых, используются 32-разрядные базовый адрес и смещение, а во-вторых, результат сложения называется линейным адресом и используется не для непосредственной адресации памяти, а для дальнейших преобразований (рис. 2.46).

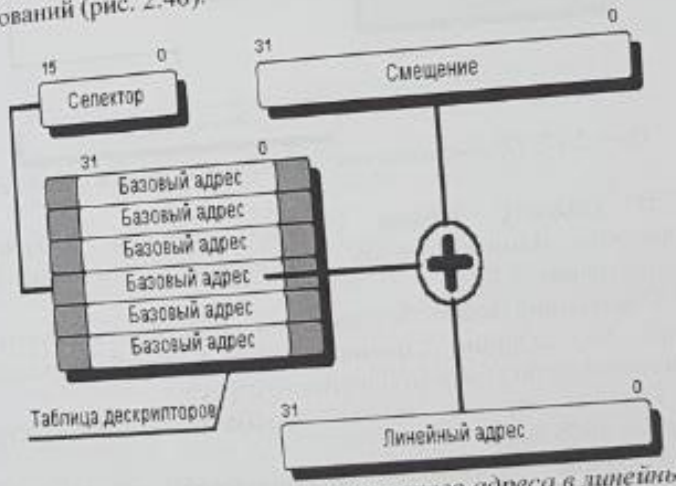


Рис. 2.46. Преобразование логического адреса в линейный

Старшие десять бит линейного адреса используются как индекс в каталоге таблиц страниц (рис. 2.47).

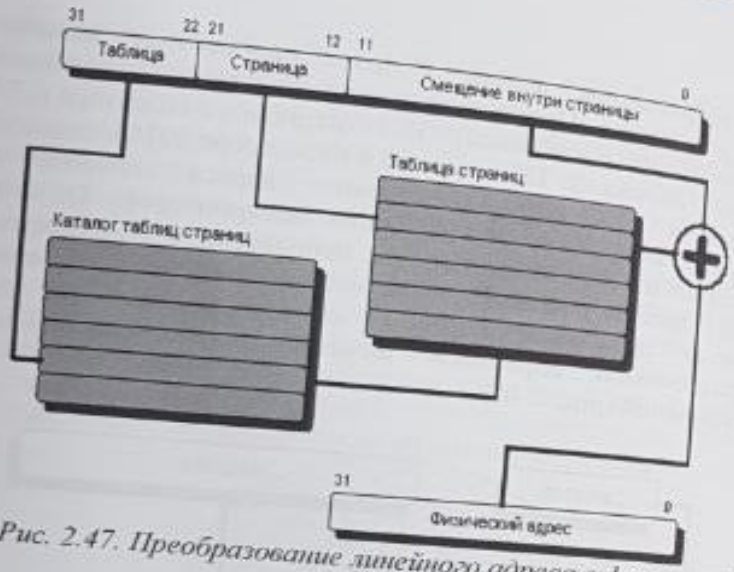


Рис. 2.47. Преобразование линейного адреса в физический

В каталоге таблиц страниц хранятся дескрипторы, содержащие помимо другой информации, физические адреса таблиц страниц[7,8].

Следующие десять бит линейного адреса используются для выбора из таблицы страниц, адрес которой определяется старшими десятью битами линейного адреса.

Таблица страниц может описывать до 1024 страниц размером 4096 байт.

Младшие двенадцать бит линейного адреса содержат смещение адресуемого байта внутри страницы.

Отметим, что преобразование линейного адреса в физический выполняется процессором с помощью каталога таблиц страниц и таблиц страниц, подготовленных операционной системой. Программист, создающий приложения для Windows, никогда не работает с таблицами страниц или каталогом таблиц страниц. Он пользуется логическим адресом в формате <селектор:смещение>, поэтому схема преобразования логического адреса в физический остается для него прозрачной.

Операционная система Microsoft Windows версии 3.1 может работать, в стандартном и расширенном режиме . В первом

случае используется схема адресации процессора 80286, даже если в компьютере установлен процессор 80386. Если Windows работает на процессоре 80386, 80486 или Pentium, при наличии достаточного объема оперативной памяти (больше 2 Мбайт) по умолчанию используется расширенный режим работы и, соответственно, схема преобразования адресов процессора 80386.

Основное преимущество системы управления памятью расширенного режима работы Windows заключается в использовании виртуальной памяти. Виртуальная память работает на уровне страниц (описанных в каталогах страниц) и совершенно прозрачна для программиста. Операционная система Windows полностью управляет виртуальной памятью. Если программа пытается обратиться к странице, отсутствующей в памяти и выгруженной на диск, происходит прерывание работы программы, страница подгружается с диска, вслед за чем работа программы продолжается. Программа может заказывать для себя блоки памяти огромного размера, адресуясь к ним непосредственно, при этом возникает полная иллюзия работы с оперативной памятью большого размера[11-14].

Описанная выше схема адресации в защищенном режиме накладывает ограничения на операции, которое приложение Windows может выполнять над селекторами.

Приложение Windows не должно выполнять над селекторами арифметические операции и операции сравнения

Программируя для реального режима операционной системы MS-DOS, вы, возможно, при адресации блока памяти большого размера (больше 64 Кбайт) изменяли содержимое сегментных регистров. В защищенном режиме вы не можете делать никаких предположений относительно базового адреса следующего или предыдущего дескриптора в локальной или глобальной таблице дескрипторов.

Сказанное не означает, что приложения Windows не могут работать с блоками памяти, занимающими несколько сегментов. В этом случае для адресации вам нужно использовать специальные методы. Однако, если вы составляете приложение на языке программирования C или C++, при определении указателей на блоки памяти размером больше, чем 64 Кбайт, можно воспользоваться ключевым словом huge. Для таких

указателей при необходимости будет автоматически выполняться переключение на нужные селекторы.

Работа приложения Windows не должна зависеть от уровня привилегий, предоставленного приложению операционной системой, так как в новых версиях Windows этот уровень может измениться.

Приведем исходный текст приложения SELECTOR, с помощью которого вы сможете проанализировать структуру селектора сегмента кода и сегмента данных, взятых из регистров CS и DS (листинг 2.1).

Листинг 2.1. Файл selector/selector.cpp

```
#define STRICT
#include <windows.h>

#pragma argsused
int PASCAL
WinMain(HINSTANCE hInstance,
   HINSTANCE hPrevInstance,
   LPSTR     lpszCmdLine, int nCmdShow)
{
  UINT uSelCS, uSelDS, uTICS, uTIDS;
  BYTE szBuf[100];

  // Получаем селектор сегмента кода
  asm mov ax, cs
  asm mov uSelCS, ax

  // Получаем селектор сегмента данных
  asm mov ax, ds
  asm mov uSelDS, ax

  // Выделяем бит TI. Если этот бит
  // равен 1, для адресации используется
  // глобальная таблица дескрипторов,
  // если 0 - локальная
  uTICS = (uSelCS & 4) >> 2;
```



```
  uTIDS = (uSelDS & 4) >> 2;

  // Выводим значения селекторов для
сегментов
  // кода и данных, значения поля TI и
номер
  // кольца защиты
  wsprintf(szBuf, "CS=%0X \tTI=%d\tRING=%d"
                  "\nDS=%0X
\tTI=%d\tRING=%d",
      uSelCS, uTICS, uSelCS & 3,
      uSelDS, uTIDS, uSelDS & 3);

  MessageBox(NULL, (LPSTR)szBuf,
     "CS & DS selector's", MB_OK);
  return 0;
}
```

Это приложение переписывает текущее содержимое регистров процессора CS и DS в переменные uSelCS и uSelDS. Далее содержимое бита TI селекторов, взятых из регистров DS и CS, переписывается в переменные uTICS и uTIDS, соответственно.

Запустив это нехитрое приложение, вы сможете убедиться, что операционная система Windows версии 3.1 предоставляет приложениям самый низкий уровень привилегий, располагая их в третьем кольце защиты (рис. 2.48).

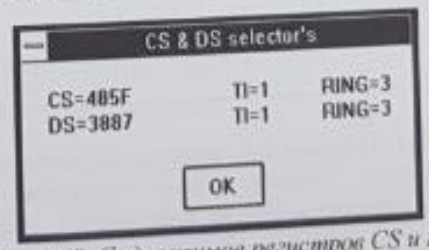


Рис. 2.48. Содержимое регистров CS и DS

Так как содержимое поля TI равно 1, это означает, что для адресации сегмента кода и сегмента данных используется локальная таблица дескрипторов.

В операционной системе Windows версии 3.1 все приложения используют одну общую локальную таблицу дескрипторов, что в принципе не исключает взаимного влияния приложений. Это означает, что адресные пространства приложений Windows не изолированы друг от друга. Поэтому не пытайтесь загружать в сегментные регистры неправильные селекторы. В следующих версиях Windows каждое приложение будет иметь свое собственное адресное пространство.

Используйте только те селекторы, которые получены приложением от операционной системы Windows.

Файл определения модуля приложения SELECTOR ничем не примечателен и приведен в листинге 2.2.

Листинг 2.2. Файл selector/selector.def

```
;============================
; Файл определения модуля
; ============================
NAME          SELECTOR
DESCRIPTION 'Приложение SELECTOR, (C) 1994,
Frolov A.V.'
EXETYPE       windows
STUB          'winstub.exe'
STACKSIZE     8120
HEAPSIZE      1024
CODE          preload moveable discardable
DATA          preload moveable multiple
```

Обработка прерываний в защищенном режиме

В реальном режиме для обработки прерываний используется таблица векторов прерываний, расположенная в первом килобайте адресного пространства. Эта таблица состоит из 256 элементов размером 4 байта, которые содержат полный адрес обработчиков прерывания в формате <сегмент:смещение>.

Как вы знаете, существуют аппаратные и программные прерывания. Аппаратные прерывания вырабатываются периферийными устройствами, как правило, при завершении ими операции ввода/вывода. Эти прерывания являются асинхронными по отношению к запущенным программам. Программные прерывания вызываются командой INT. Программные прерывания являются синхронными, так как они инициируются самой программой[17.18].

В ответ на прерывание любого типа в реальном режиме в регистры CS:IP процессора загружается адрес, взятый из соответствующей ячейки таблицы векторов прерываний, после чего управление передается по этому адресу. Обработчик прерываний, выполнив все необходимые действия, возвращает управление прерванной программе, выполняя команду IRET.

Программы MS-DOS широко используют программные прерывания для получения обслуживания от MS-DOS и BIOS.

Механизм обработки прерываний в защищенном режиме намного сложнее. Для определения адресов обработчиков прерываний в защищенном режиме используется дескрипторная таблица прерываний IDT (Interrupt Descriptor Table), расположение которой определяется содержимым специального системного регистра. Эта таблица содержит дескрипторы специальных типов - вентили прерываний, вентили исключений и вентили задач.

Вентиль прерываний содержат не только логический адрес обработчика прерывания, но и поле доступа. Программа может вызвать прерывание только в том случае, если она имеет для этого достаточный уровень доступа. Таким образом, операционная система, работающая в защищенном режиме, может запретить прикладным программам вызывать некоторые или все программные прерывания.

Обычные приложения Windows никогда не должны вызывать программные прерывания, так как для взаимодействия с операционной системой используется другой механизм, основанный на вызове функций из библиотек динамической загрузки. Тем не менее, некоторые прерывания (например, INT 21h) все же можно использовать. Для таких прерываний

Windows выполняет трансляцию адресов из формата защищенного режима в формат реального режима.

Приложение Windows не должно пытаться изменить дескрипторную таблицу прерываний. Не следует также думать, что эта таблица расположена по адресу 0000h:0000h, селектор 0000h вообще не используется для адресации памяти.

Если в этом нет особой необходимости, приложение Windows не должно вызывать программные прерывания. Для работы с файлами, принтером, для вывода на экран следует вызывать функции программного интерфейса операционной системы Windows.

Защищенный режим работы микропроцессора

В защищенном режиме любой запрос к памяти как со стороны операционной системы, так и со стороны прикладных программ должен быть санкционирован. Микропроцессор *аппаратно* контролирует доступ программ к любому адресу в оперативной памяти.

В защищенном режиме всегда действует сегментный способ организации распределения памяти. Кроме того, может быть включен и страничный механизм (страничная трансляция).

Поддержка сегментированной модели памяти

Дескрипторные таблицы

Микропроцессор аппаратно поддерживает три типа дескрипторных таблиц:

- глобальная дескрипторная таблица (GDT);
- локальная дескрипторная таблица (LDT);
- таблица дескрипторов прерываний (IDT).

1. Таблица GDT (Global Descriptor Table) — *глобальная дескрипторная таблица.* Это основная общесистемная таблица, к которой допускается обращение со стороны программ, обладающих достаточными привилегиями. Расположение таблицы GDT в памяти произвольно; оно локализуется с помощью специального регистра gdtr (48 бит). В таблице GDT могут содержаться следующие типы дескрипторов:
   - дескрипторы сегментов кодов программ;
   - дескрипторы сегментов данных программ;

   - дескрипторы стековых сегментов программ;
   - дескрипторы TSS (Task Segment Status) — специальные системные объекты, называемые сегментами состояния задач;
   - дескрипторы для таблиц LDT;
   - шлюзы вызова;
   - шлюзы задач.

2. Таблица LDT (Local Descriptor Table) — *локальная дескрипторная таблица.* Для любой задачи в системе может быть создана своя дескрипторная таблица подобно общесистемной GDT. Для связи между таблицами GDT и LDT в таблице GDT создается дескриптор, описывающий область памяти, в которой находится LDT. Расположение таблицы LDT в памяти также произвольно и локализуется с помощью специального регистра ldtr (16 бит). В таблице LDT могут содержаться следующие типы дескрипторов:
   - дескрипторы сегментов кодов программ;
   - дескрипторы сегментов данных программ;
   - дескрипторы стековых сегментов программ;
   - шлюзы вызова;
   - шлюзы задач.

3. **Таблица IDT** (Interrupt Descriptor Table) — *дескрипторная таблица прерываний.* Данная таблица также является общесистемной и содержит дескрипторы специального типа, которые определяют местоположение программ обработчиков всех видов прерываний. В качестве аналогии можно привести таблицу векторов прерываний реального режима. Расположение таблицы IDT в памяти произвольно и локализуется с помощью специального регистра idtr (48 бит). Элементы данной таблицы называются *шлюзами.* Отметим, что эти шлюзы бывают трех типов:
   - шлюзы задач;
   - шлюзы прерываний;
   - шлюзы ловушек.

Каждая из дескрипторных таблиц может содержать до 8192 (2^{13}) дескрипторов.

Формат селектора сегмента

В защищенном режиме иначе интерпретируется содержимое сегментных регистров.

Теперь в них содержатся не адреса начала сегмента, а номер соответствующего сегмента (индекс, или селектор) в таблице дескрипторов сегментов. Поэтому теперь сегментные регистры называются ***селекторами сегментов***.

Каждый регистр разбивается на три поля:

- индекс (старшие 13 битов) - Index. Определяет номер сегмента (индекс) в соответствующей таблице дескрипторов.
- индикатор таблицы сегментов - TI (бит 2). Определяет в глобальной или локальной таблице находится дескриптор сегмента: TI=0 - в глобальной, TI=1 - в локальной.
- Уровень привилегий (биты 0 и 1). Указывает запрашиваемый уровень привилегий.

Формат дескриптора сегмента

Каждый сегмент описывается соответствующим дескриптором сегмента.

Структурно дескриптор сегмента представляет собой 8-байтовую структуру.

Обратим внимание на следующее:

- в защищенном режиме размер сегмента не фиксирован, его расположение можно задать в пределах 4 Гбайт.
- поля, определяющие размер сегмента и его начальный (базовый) адрес разорваны, в целях совместимости со старыми микропроцессорами.

Защищенный режим впервые появился в микропроцессоре i80286. Этот микропроцессор имел 24-разрядную адресную шину и, соответственно, мог адресовать в защищенном режиме до 16 Мбайт оперативной памяти. Для этого ему достаточно было иметь в дескрипторе поле базового адреса 24 бита и поле размера сегмента 16 бит. После появления микропроцессора i80386 с 32-разрядной шиной команд и данных в целях совместимости программ разработчики не стали изменять формат дескриптора, а просто использовали свободные поля. Внутри микропроцессора эти поля объединены. Внешне же они остались разделены, и при программировании с этим приходится мириться.

Размер сегмента в защищенном режиме может достигать 4 Гбайт, то есть занимать все возможное физическое пространство памяти.

Как это возможно, если суммарный размер поля размера сегмента всего 20 бит, что соответствует величине 1 Мбайт? Секрет скрыт в поле *гранулярности* — бит G. Если бит G=0, то значение в поле размера сегмента означает размер сегмента в байтах, а если G=1, то в страницах. Размер страницы составляет 4 Кбайт. Нетрудно подсчитать, что когда максимальное значение поля размера сегмента составляет Offfffh, то это соответствует 1 М страниц, что и соответствует величине 1 М * 4 Кб ¯ 4 Гб.

Таким образом, первый тип защиты - **по уровню доступа** - реализуется в защищенном режиме благодаря тому, что информация о базовом адресе сегмента и его размере выведена на уровень микропроцессора □ это позволяет аппаратно контролировать работу программ с памятью и предотвращать обращения по несуществующим адресам либо по адресам, находящимся вне предела, разрешенного полем размера сегмента (limit).

Другой тип защиты - **по привилегиям** - заключается в том, что сегменты неравноправны в правах доступа к ним. Суть этого механизма защиты по привилегиям заключается в том, что конкретный сегмент может находиться на одном из четырех уровней привилегированности с номерами 0, 1, 2 и 3. Самым привилегированным является уровень 0. Существует ряд ограничений на взаимодействие сегментов с различными уровнями привилегий.

Информация о правах доступа к сегменту содержится в специальном байте AR дескриптора.

Наиболее важные поля байта AR — это

1) dpl (2 бита)

2) биты R/W, C/ED и I, которые вместе определяют тип сегмента.

Поле dpl — часть механизма защиты по привилегиям. Содержит значение 0.. 3 привилегированности сегмента.

Тип сегмента определяется тремя битами. Это поле определяет целевое назначение сегмента.

Рассмотрим назначение некоторых комбинаций этих битов.

Комбинации битов	Назначение сегмента
000	Сегмент данных, только для чтения
001	Сегмент данных с разрешением чтения и записи
010	Не определена
011	Не определена
100	Сегмент стека с разрешением чтения и записи
101	Сегмент кода с разрешением только выполнения
110	Сегмент кода с разрешением выполнения и чтения из него
111	Подчиненный сегмент кода с разрешением выполнения

Подчиненный сегмент кода с разрешением выполнения и чтения из него

Замечание. Возможны два принципиально разных вида сегментов: данных и кода. Сегмент стека является разновидностью сегмента данных, но с особой трактовкой поля размера сегмента. Это объясняется спецификой использования стека (он растет в направлении младших адресов памяти). Таким образом, видно, что поле типа ограничивает использование объявленных сегментов. В частности, программные сегменты не могут быть модифицированы без применения специальных приемов. Доступ к сегменту данных также может быть ограничен только на чтение.

Поддержка страничной модели памяти

При страничной организации ОП делится на блоки (***страницы***) фиксированного размера 4 Кб (число, кратное степени двойки, операции сложения можно заменить на операции конкатенации).

Диспетчер памяти для каждой страницы формирует соответствующий дескриптор. Дескрипторы страниц собираются в таблицы.

Таблицы страниц

Есть два типа таблиц страниц:

1. Таблица каталогов таблиц страниц (PDE - page directiry entry)
2. Таблица страниц (PTE - page table entry)

Каждая таблица состоит из 1024 (2^{10}) элементов. Элементами таблиц являются дескрипторы страниц. Размер одного дескриптора - 4б. Для хранения одной таблицы необходима одна страница памяти.

Говорят, что осуществляется ***двухшаговая страничная трансляция*** адресов.

Рассмотрим составляющие этого механизма.

Для текущей задачи создается одна таблица PDE и одна или более страниц PTE.

Все страницы текущей задачи описаны в таблицах страниц PTE - page table entry.

Одна таблица PTE состоит из 1024 (2^{10}) элементов - дескрипторов страниц, одна таблица страниц описывает пространство памяти в 4 Мб.

Если задаче недостаточно памяти в 4 МБ, создается несколько таблиц PTE.

Пусть текущая задача использует 50 Мб памяти (например, графический редактор) для описания этой памяти надо иметь 14 таблиц PTE.

Для таблиц PTE текущей задачи создается таблица PDE. В ней каждый дескриптор указывает местонахождение одной таблицы PTE. Таблица PDE также состоит из 1024 (2^{10}) элементов. (Остальные дескрипторы не используются.)

Обратим внимание, что для описания 50 Мб памяти одной задачи требуется 15 страниц = 60Кб памяти (такие потери считаются приемлемыми).

Формат дескриптора страницы

Каждая страница описывается дескриптором (32 бита).

Старшие 20 битов - номер страницы. По существу, это адрес страницы, т.к. приписывание в качестве младших разрядов 12 нулей приводит к получению начального адреса страницы;

Количество битов, отводимое под номер страницы, определяет объем возможной ОП, которой может пользоваться программа.

- старшие 10 битов определяют номер (индекс) таблицы страниц PTE в таблице PDE

- младшие 10 битов - номер (индекс) дескриптора страницы в таблице PTE (а из этого дескриптора уже выбирается номер физической страницы).

Остальные биты

- (0): present - самый младший (нулевой) бит - так называемый бит присутствия. If present=0, то страница отсутствует в ОП - прерывание с передачей управления специальной программе, которая должна загрузить отсутствующую страницу;
- (1): read/write - для защиты памяти
- (2): User/supervisor - для защиты памяти
- (3,4) - нулевые
- (5): access - бит обращения, показывает, что к странице осуществлялся доступ
- (6): dirty - «грязный» бит - отмечает, что данную страницу модифицировали и при замещении ее следующей необходимо сохранить во внешней памяти
- (7,8) - нулевые
- (9,10,11) - зарезервированы для разработки системными программистами подсистемы организации виртуальной памяти.

Переход микропроцессора в защищенный режим

Вспомним кратко процесс загрузки ОС и подчеркнем момент перехода в защищенный режим.

Сразу после включения питания или нажатия кнопки сброса микропроцессор начинает свою работу в реальном режиме. В этом режиме он производит действия по тестированию аппаратуры компьютера. После успешного завершения тестирования микропроцессор выполняет начальную загрузку

системы, используя программу начальной загрузки, хранящейся на нулевой дорожке диска. Программа начальной загрузки считывает с диска программу инициализации операционной системы и передает ей управление. Действие этой программы зависит от того, в каком режиме работы микропроцессора будет осуществляться дальнейшее функционирование системы. Если в реальном режиме, то операционная система формирует среду и структуры данных для работы в этом режиме. Если же загружаемая операционная система собирается дальше работать в защищенном режиме, то она должна в него специальным образом перейти.

Но прежде чем сделать это, операционная система формирует системные структуры данных (в частности, рассмотренные нами дескрипторные таблицы) для работы в защищенном режиме. Затем может быть осуществлен переход в защищенный режим и выполнение дальнейших действий.

2.1.4. Методы распределения памяти

Понятие об организации и управлении физической памятью в операционных системах

Организация и управление основной (первичной, физической, реальной) памятью вычислительной машины - один из важнейших факторов, определяющих построение операционных систем. В англоязычной технической литературе память обозначается синонимами *memory* и *storage*.

В операционных системах различают два вида памяти: основная (первичная) и внешняя (вторичная).

Основная память (main storage) - оперативная память центрального процессора или ее часть, представляющее собой единое пространство памяти.

Внешняя память (external storage) - память, данные в которой доступны центральному процессору посредством операций ввода-вывода.

Для непосредственного выполнения программ или обращения к данным необходимо, чтобы они размещались в основной памяти. Внешняя память имеет, как правило, гораздо

большую емкость, чем основная, стоит дешевле и позволяет хранить данные и программы, которые должны быть наготове для обработки.

Кроме основной и внешней памяти в современных ЭВМ существует дополнительная быстродействующая память, называемая *кэш-памятью*.

Все три перечисленных вида памяти образуют *иерархию памяти* вычислительной машины (рис.2.49.).

Рис.2.49. Иерархия памяти

Операционным системам с несколькими уровнями иерархии памяти свойственна высокая интенсивность челночных обменов программами и данными между физическими устройствами памяти различных уровней. Такие обмены отнимают системные ресурсы (например, время центрального процессора), которые можно было бы использовать более продуктивно.

Основная память представляет собой один из самых дорогостоящих ресурсов. Главной задачей при разработке ОС считается оптимальное использование основной памяти на основе рациональной организации и управления ею.

Под *организацией памяти* понимается то, каким образом представляется и как используется основная память.

В операционных системах применяются следующие виды представления основной памяти:

- фиксированными блоками равного размера;
- фиксированными разделами неодинакового размера;
- динамическими разделами, размеры которых изменяются в ходе работы вычислительной системы.

Использование основной памяти может осуществляться следующими способами:

- размещение в памяти единовременно только одной программы пользователей;
- размещение в памяти одновременно нескольких программ пользователей;
- размещение программ пользователей в конкретном заранее заданном разделе основной памяти;
- размещение каждой программы пользователя в одном непрерывном (односвязном) пространстве основной памяти;
- размещение программы пользователя в несмежных областях оперативной памяти (при этом ОС осуществляет разбиение размещаемых там программ на отдельные блоки и обеспечивает связь этих блоков между собой).

В операционных системах может применяться любая комбинация перечисленных видов представления и способов использования основной памяти ЭВМ[12,13].

Независимо от того, какая схема организации памяти принята для конкретной ОС, необходимо решить, какие стратегии следует применять для достижения оптимальных характеристик.

Стратегии управления памятью определяют, как будет работать память с конкретной схемой организации при различных подходах к решению следующих вопросов:

- когда следует поместить новую программу в память;
- в какое место основной памяти будет размещаться очередная программа;
- как разместить очередную программу в памяти (с минимизацией потерь памяти или с максимизацией скорости размещения);
- какую из находящихся в памяти программ следует вывести из памяти, если необходимо обязательно разместить новую программу, а память уже заполнена.

В существующих ОС реализованы стратегии управления, по-разному отвечающие на перечисленные выше вопросы, что в немалой степени обусловлено имеющимися в распоряжении разработчиков аппаратурными и программными средствами.

Стратегии управления памятью делятся на следующие категории:

- стратегии выборки;
- стратегии размещения;
- стратегии замещения.

В свою очередь стратегии выборки разделяют на две подкатегории:

- стратегии выборки по запросу (по требованию);
- стратегии упреждающей выборки.

Стратегии выборки ставят своей целью определить, когда следует "втолкнуть" очередную программу (или блок программы) или данные в основную память.

Стратегии размещения ставят своей целью определить, в какое место основной памяти следует размещать поступающую программу. Наиболее распространенными являются стратегии размещения, реализующие принципы занятия "первого подходящего", "наиболее подходящего" и "наименее подходящего" по размерам свободного участка памяти.

Стратегии замещения ставят своей целью определить, какой блок программы или данных следует вывести ("вытолкнуть") из основной памяти, чтобы освободить место для размещения вновь поступающих программ или данных.

При реализации стратегий размещения операционные системы часто учитывают требования связного распределения памяти для программ и данных.

Связное распределение памяти - такое распределение основной памяти ЭВМ, при котором каждая программа занимает один непрерывный (связный) блок ячеек памяти.

Несвязное распределение памяти - такое распределение основной памяти ЭВМ, при котором программа пользователя разбивается на ряд блоков (сегментов, страниц), которые могут размещаться в основной памяти в участках, не обязательно соседствующих друг с другом (в несмежных участках). В этом случае обеспечивается более эффективное использование пространства основной памяти.

Эффективность той или иной стратегии размещения можно оценить с помощью коэффициента использования памяти h

$$h = \frac{V_П}{V_{ОП} - V_{ОС}} = \frac{V_П}{V_О} \qquad (2.2)$$

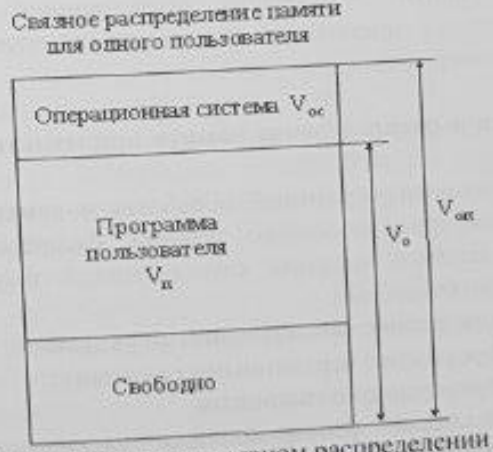
где $V_п$ - объем памяти, занимаемый программами пользователя; $V_{оп}$ - полный объем основной памяти; $V_{ос}$ - объем памяти, занимаемый операционной системой; $V_о$ - объем памяти, доступный для распределения.

Связное распределение памяти для одного пользователя

Связное распределение памяти для одного пользователя, называемое также одиночным непрерывным распределением, применяется в ЭВМ, работающих в пакетном однопрограммном режиме под управлением простейшей ОС.

Вся основная часть ЭВМ, не занятая программами операционной системы, выделяется программе единственного на данном отрезке времени пользователя. Размер программы в этом случае ограничивается размером доступной основной памяти, однако существует возможность выполнения программ, размер которых превышает размер основной памяти, используя механизм оверлеев.

Организация памяти при связном распределении для одного пользователя показана на рис. 2.50.



2.50. Память при связном распределении

Коэффициент использования памяти для рассматриваемого случая вычисляется по формуле

$$h_{cf}=V_n/V_o, \qquad (2.3)$$

где V_n - размер программы пользователя;

V_o - объем доступной для распределения основной памяти ЭВМ.

Функциями ОС в данном случае являются:

- выделение программе необходимого пространства памяти;
- защита памяти;
- освобождение памяти.

Функция выделения памяти сводится к предоставлению программе всей доступной памяти ЭВМ.

Защита памяти в однопрограммных системах заключается в установке защиты областей памяти, занятых операционной системой, от воздействия программ пользователя. Эта функция реализуется при помощи одного *регистра границы*, встроенного в центральный процессор. Регистр границы содержит либо старший адрес команды, относящийся к операционной системе, либо младший адрес доступной программе основной памяти (адрес начала программы). Если программа пользователя пытается войти в область операционной системы, то вырабатывается прерывание по защите памяти, и программа аварийно завершается.

Связное распределение памяти при мультипрограммной обработке

При мультипрограммной обработке в памяти компьютера размещается сразу несколько заданий. Распределение памяти между заданиями в этом случае может быть выполнено следующими способами:

- распределение фиксированными разделами;
- распределение переменными разделами;
- распределение со свопингом.

Распределение фиксированными разделами имеет две модификации:

а) с загрузкой программ в абсолютных адресах;

б) с загрузкой перемещаемых модулей.

При загрузке перемещаемых модулей вся оперативная память машины разбивается на некоторое количество *разделов* фиксированного размера. Размеры разделов могут не совпадать. В каждом разделе может быть размещено только одно задание.

В случае загрузки программ в абсолютных адресах при их подготовке указывается начальный адрес загрузки программ, совпадающий с начальным адресом раздела, в котором эта программа будет выполняться[19,20].

В случае загрузки перемещаемых модулей раздел, в котором будет размещено задание, либо автоматически определяется операционной системой в соответствии с реализованной в нем стратегией выбора раздела ("первый подходящий", "самый подходящий", "самый неподходящий"), либо указывается операционной системе специальными командами языка управления заданиями.

В обоих случаях задание монопольно владеет всем объемом оперативной памяти раздела, в который оно было помещено операционной системой.

Основным недостатком распределения памяти фиксированными разделами является неэффективное использование ресурсов вычислительной системы из-за возможного появления длинных очередей заданий, ожидающих освобождения конкретного раздела в то время, как остальные разделы пусты. Подобная ситуация изображена на рис.2.51. Задания, ожидающие освобождения раздела С, могли бы разместиться и в разделах А или В, однако операционная система не позволяет им это сделать, т.к. в управляющей информации указан конкретный раздел, в котором эти задания должны выполняться - раздел С.

Способ распределения памяти фиксированными разделами используется в операционных системах ОС ЕС и IBM/360 в режиме MFT, в котором загрузка программ выполняется перемещаемыми модулями.

Защита памяти при распределении фиксированными разделами выполняется аналогично защите памяти для одного пользователя, только теперь необходимо наличие нескольких

граничных регистров - по два регистра на каждый раздел. В одном из граничных регистров указывается нижняя граница раздела, а во втором - его верхняя граница. Если программа пользователя пытается обратиться к данным, расположенным вне области адресов данного раздела, то вырабатывается прерывание по защите памяти.

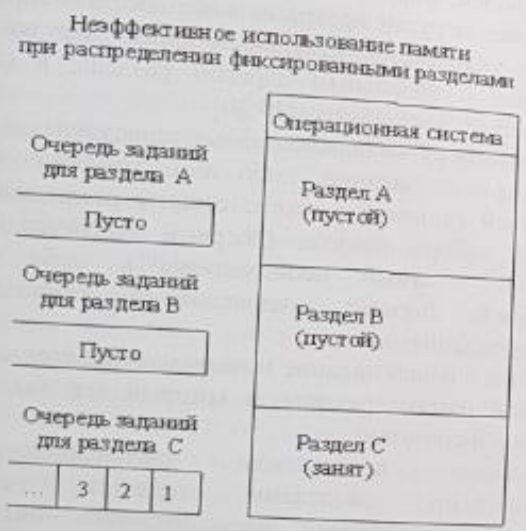


Рис.2.51.Распределение фиксированными разделами

В мультипрограммных системах с фиксированными разделами наблюдается явление *фрагментации памяти.*

Фрагментация памяти - появление в памяти вычислительной машины чередования занятых и незанятых (свободных) участков оперативной памяти.

При распределении фиксированными разделами появление фрагментации обусловлено тем, что либо задания пользователей не полностью занимают выделенные им разделы, либо часть разделов остается незанятой[1,4,6].

На рис.2.52. показано проявление фрагментации оперативной памяти.

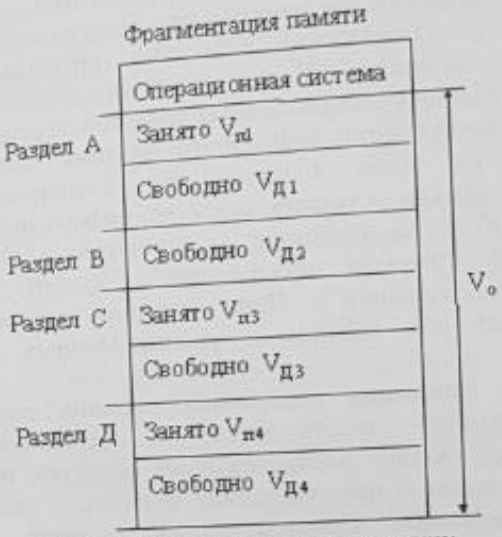


Рис.2.52. Фрагментация памяти

Фрагментация памяти представляет собой нарушение односвязности пространства свободной памяти ЭВМ, что приводит к снижению эффективности использования памяти как одного из основных ресурсов вычислительной машины.

Распределение памяти переменными разделами предназначено для повышения эффективности использования оперативной памяти ЭВМ. Суть способа распределения памяти переменными разделами состоит в том, что заданиям, когда они поступают, выделяется такой объем памяти, который им требуется, т.е. размер раздела оперативной памяти, выделяемой каждому заданию, в точности соответствует размеру этого задания. Поэтому "перерасхода" памяти, как это происходит при распределении фиксированными разделами, в данном способе не наблюдается.

Имеется две модификации способа распределения переменными разделами:

- распределение переменными неперемещаемыми разделами;

• распределение переменными перемещаемыми разделами.

При распределении памяти *переменными неперемещаемыми разделами* (динамическими разделами) операционная система создает две таблицы: таблицу учета распределенных областей памяти и таблицу учета свободных областей памяти ("дыр").

При поступлении очередного задания память для него отводится на этапе долгосрочного планирования, причем выделение памяти осуществляется по информации из таблицы учета "дыр" в соответствии с принятой в ОС стратегией размещения ("первый подходящий", "самый подходящий", "самый неподходящий"). При успешном распределении ОС корректирует обе таблицы - распределенных и свободных областей.

После окончания какого-либо задания занимаемый им участок памяти освобождается, и операционная система корректирует таблицу распределенных областей, вычеркивая из нее информацию о закончившемся задании, а также заносит в таблицу свободных областей данные о вновь появившейся "дыре".

Рассмотрим следующий пример. Пусть начальное распределение памяти переменными разделами выполнено так, как показано в табл.2.1, 2.2 и на рис.2.53а. После размещения заданий А, В, С и Д осталась свободная область такого размера, что ни одна из программ, продолжающих стоять в очереди, в эту область не помещается.

Таблица 2.1. Таблица распределенных областей

Номер раздела, ключ защиты	Имя раздела	Размер	Адрес	Состояние
1	А	100К	50К	Распределен
2	В	200К	150К	Распределен
3	С	100К	350К	Распределен
4	Д	400К	450К	Распределен
5	Е	100К	850К	Распределен

Таблица 2.2. Таблица свободных областей

Номер свободной области	Размер	Адрес	Состояние
1	100К	950К	Доступна

Предположим, что через некоторое время закончились задания А и С (рис.2.53б). Таблицы областей приобретают вид, показанный в табл. 2.3 и 2.4.

Таблица 2.3. Таблица распределенных областей: закончилось задание А

Номер раздела, ключ защиты	Имя раздела	Размер	Адрес	Состояние
1	-	-	-	Пусто
2	В	200К	150К	Распределен
3	-	-	-	Пусто
4	Д	400К	450К	Распределен
5	Е	100К	850К	Распределен

Таблица 2.4. Таблица свободных областей: закончилось задание А

Номер свободной области	Размер	Адрес	Состояние
1	100К	100К	Доступна
2	100К	350К	Доступна
3	100К	950К	Доступна

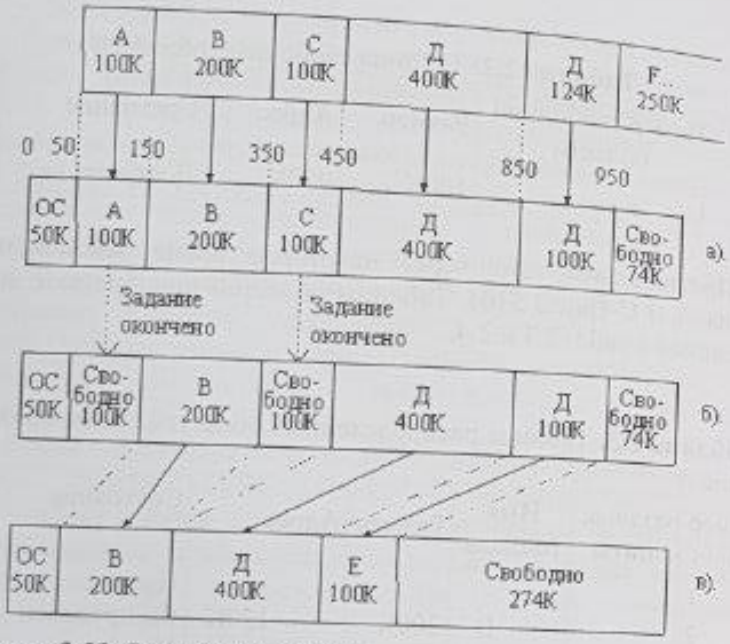


рис. 2.53. Распределение памяти переменными разделами

Можно видеть, что несмотря на наличие 274К свободной памяти, достаточной для размещения задания Е, стоящего первым в очереди, ОС не имеет возможности это сделать, т.к. свободная память разбита на два фрагмента по 100К каждый, в которых разместить программы, стоящие в очереди, невозможно. Этот пример иллюстрирует главный недостаток способа распределения переменными неперемещаемыми разделами - склонность к фрагментации основной памяти, что снижает эффективность работы вычислительной системы.

При распределении памяти *переменными перемещаемыми разделами* операционная система осуществляет действия, называемые *уплотнением памяти*, состоящими в перемещении всех занятых участков к одному или другому краю основной памяти. Благодаря этому вместо большого количества небольших "дыр", образующихся при использовании распределения переменными неперемещаемыми разделами, формируется

единый (связный) участок свободной памяти. На рис.2.53в показан результат уплотнения, когда находящиеся в основной памяти программы В, Д и Е перемещены на свободные участки после окончания работы программ А и С. Свободная память теперь представляет собой непрерывную область размером 274К, в которую ОС может поместить стоящее в очереди задание F. Этот процесс называют также *дефрагментацией* памяти.

Дефрагментация памяти, применяемая при распределении перемещаемыми разделами, имеет свои недостатки:

- требуются дополнительные затраты времени;
- во время уплотнения памяти система должна прекращать (приостанавливать) все другие работы, что зачастую может оказаться неприемлемым;
- необходимость перемещения заданий в памяти требует хранения значительного объема информации, связанной с размещением программ в памяти, что увеличивает требования к памяти со стороны ОС;
- при интенсивном потоке коротких программ может возникнуть необходимость частой дефрагментации памяти, так что затачиваемые на эти цели системные ресурсы могут оказаться неоправданными получаемой выгодой.

Распределение памяти со свопингом (от англ. swapping - подкачка) характеризуется тем, что в отличие от рассмотренных ранее способов распределения программы пользователей не остаются в основной памяти до момента их завершения. В простейшей системе со свопингом в каждый момент времени только одно задание пользователя находится в основной памяти и занимает ее до тех пор, пока оно может выполняться, а затем освобождает как память, так и центральный процессор для следующего задания. Таким образом, вся память целиком на короткий период выделяется одному заданию, затем в некоторый момент времени это задание выводится (выталкивается, т.е. осуществляется "откачка"), а очередное задание вводится (вталкивается, т.е. осуществляется "подкачка"). В обычном случае каждое задание, еще до своего завершения, будет много раз перекачиваться из внешней памяти в основную и обратно.

Для обеспечения свопинга во внешней памяти ОС создает один или несколько *файлов подкачки*, где хранятся образы

оперативной памяти находящихся в работе заданий пользователей. Способ распределения памяти со свопингом применяется в простейших ОС, работающих в режиме разделения времени.

Стратегии размещения информации в памяти

Стратегии размещения информации в памяти предназначены для того, чтобы определить, в какое место основной памяти следует помещать поступающие программы и данные при распределении памяти неперемещаемыми разделами. Наиболее часто применяются следующие стратегии:

- размещение с выбором первого подходящего (стратегия "первый подходящий");
- размещение с выбором наиболее подходящего (стратегия "самый подходящий");
- алгоритм с выбором наименее подходящего (стратегия "самый неподходящий").

Стратегия "первый подходящий" состоит в выполнении следующих шагов:

- упорядочить таблицу свободных областей в порядке *возрастания адресов*;
- поместить информацию в первый встретившийся участок основной памяти размером не менее требуемого.

Стратегия "самый подходящий" реализует следующую последовательность действий:

- упорядочить таблицу свободных областей в порядке *возрастания размеров* свободных областей;
- поместить информацию в первый встретившийся участок свободной памяти размером не менее требуемого.

Стратегия "самый неподходящий" выполняет следующие действия:

- упорядочить таблицу свободных областей в порядке *убывания размеров* областей;
- поместить информацию в первый встретившийся участок свободной памяти размером не менее требуемого.

Строгих доказательств преимущества той или иной стратегии перед остальными не существует, так что их

применение в операционных системах основано на интуитивных аргументах разработчиков ОС.

Организация виртуальной памяти

Термин *виртуальная память* обычно ассоциируется с возможностью адресовать пространство памяти, гораздо большее, чем емкость первичной (реальной, физической) памяти конкретной вычислительной машины. Концепция виртуальной памяти впервые была реализована в машине, созданной в 1960 г. в Манчестерском университете (Англия). Однако широкое распространение системы виртуальной памяти получили лишь в ЭВМ четвертого и последующих поколений.

Существует два наиболее известных способа реализации виртуальной памяти - *страничная* и *сегментная*. Применяется также их комбинация - *странично-сегментная* организация виртуальной памяти.

Все системы виртуальной памяти характеризуются тем, что адреса, формируемые выполняемыми программами, не обязательно совпадают с адресами первичной памяти. Виртуальные адреса, как правило, представляют гораздо большее множество адресов, чем имеется в первичной памяти.

Суть концепции виртуальной памяти заключается в том, что адреса, к которым обращается выполняющийся процесс, отделяются от адресов, реально существующих в первичной памяти.

Адреса, на которые делает ссылки выполняющийся процесс, называются *виртуальными адресами*.

Адреса, которые реально существуют в первичной памяти, называются *реальными (физическими) адресами*.

Диапазон виртуальных адресов, к которым может обращаться выполняющийся процесс, называется *пространством виртуальных адресов* V этого процесса.

Диапазон реальных адресов, существующих в конкретной вычислительной машине, называется *пространством реальных адресов R* этой ЭВМ.

Несмотря на то, что процессы обращаются только к виртуальным адресам, в действительности они должны работать

с реальной памятью. Для установления соответствия между виртуальными и реальными адресами разработаны механизмы динамического преобразования адресов ДПА (или ДАТ - от англ.Dynamics Adress Transformation), обеспечивающие преобразование виртуальных адресов в реальные во время выполнения процесса. Все подобные системы обладают общим свойством (см.рис.2.54) - *смежные адреса виртуального адресного пространства процесса не обязательно будут смежными в реальной памяти.*

Это свойство называют "искусственной смежностью". Тем самым пользователь освобождается от необходимости рассматривать физическую память с ее уникальными характеристиками.

Виртуальная память строится, как правило, по двухуровневой схеме (см.рис.2.54).

Первый уровень - это реальная память, в которой находятся выполняемые процессы и в которой должны размещаться данные, к которым обращаются эти процессы.

Второй уровень - это внешняя память большой емкости, например, накопители на магнитных дисках, способные хранить программы и данные, которые не могут все сразу уместиться в реальной памяти из-за ограниченности ее объема. Память второго уровня называют *вторичной* или *внешней.*

В мультипрограммных режимах реальная память разделяется между многими процессами. Поскольку каждый процесс может иметь гораздо большее пространство виртуальных адресов, чем реальная память, то в текущий момент времени в реальной памяти имеется возможность держать лишь небольшую часть программных кодов и данных каждого процесса, причем даже эти небольшие части кодов и данных не обязательно будут размещаться сплошным массивом реальной памяти (свойство "*искусственной смежности*").

Механизм динамического преобразования адресов ведет учет того, какие ячейки виртуальной памяти в данный момент находятся в реальной памяти и где именно они размещаются. Это осуществляется с помощью таблиц отображения, ведущихся механизмом ДПА.

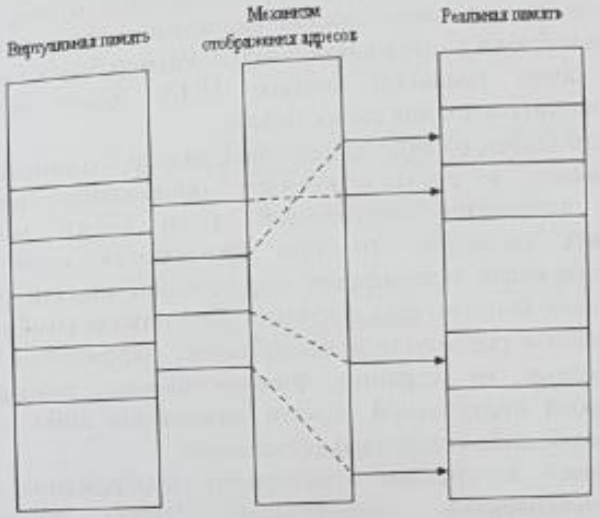


Рис.2.54. Искусственная смежность

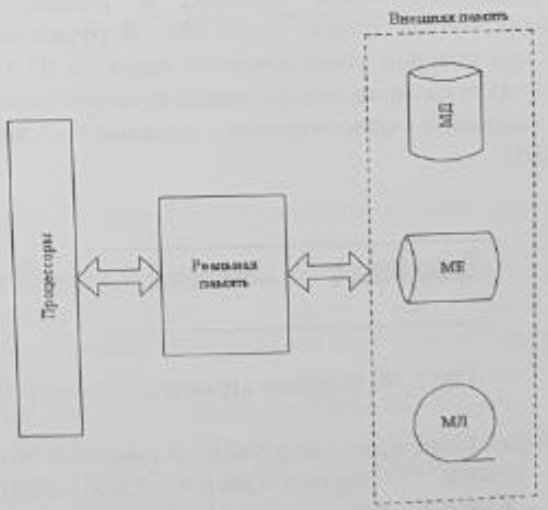


Рис.2.55. Двухуровневая память

Информация, перемещаемая из виртуальной памяти в реальную, механизмом ДПА группируется в *блоки*, и система следит за тем, в каких местах реальной памяти размещаются различные блоки виртуальной памяти. Размер блока влияет на то, какую долю реальной памяти ДПА будет использовать непроизводительно, для своих целей.

Если блоки имеют одинаковый размер, то они называются *страницами*, а соответствующая организация виртуальной памяти называется *страничной*. Если блоки могут быть различных размеров, то они называются *сегментами*, а соответствующая организация виртуальной памяти называется *сегментной*. В некоторых системах оба подхода комбинируются, т.е. сегменты реализуются как объекты переменных размеров, формируемые из страниц фиксированного размера. Такая организация виртуальной памяти называется либо *сегментно-страничной*, либо *странично-сегментной*.

Адреса в системе поблочного отображения являются *двухкомпонентными (двумерными)*. Чтобы обратиться к конкретному элементу данных, программа указывает блок, в котором расположен этот элемент, и смещение элемента относительно начала блока (рис.2.56). Виртуальный адрес n указывает при помощи упорядоченной пары (b, d), где b- номер блока, в котором размещается соответствующий элемент данных, а d - смещение относительно начального адреса этого блока[19,20].

Рис.2.56. Формат виртуального адреса

Преобразование адреса виртуальной памяти n =(b, d) в адрес реальной памяти r осуществляется следующим образом (рис.2.57). Каждый процесс имеет собственную *таблицу отображения блоков*, которую операционная система ведет в реальной памяти. Реальный адрес a этой таблицы загружается в

специальный регистр центрального процессора, называемый *регистром начального адреса таблицы отображения блоков процесса*.

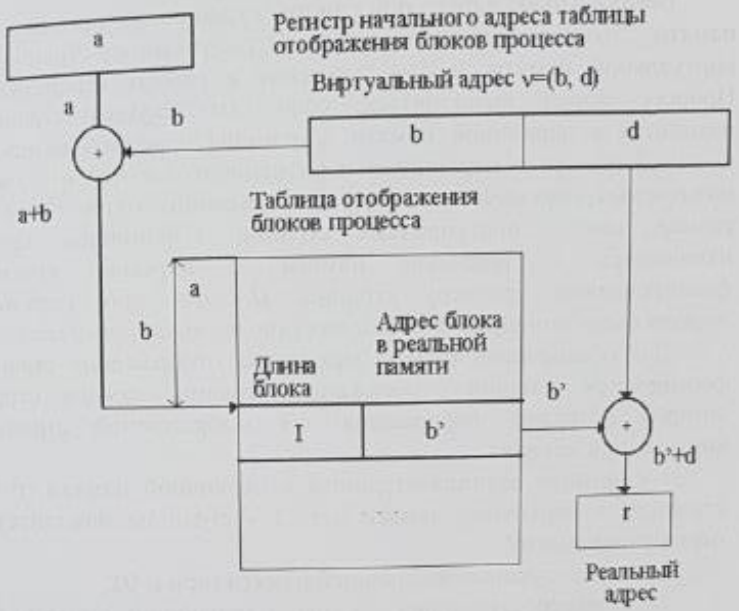


Рис.2.57. Преобразование виртуального адреса

Таблицы отображения блоков содержат по одной строке для каждого блока процесса, причем эти блоки идут последовательно: сначала блок 0, затем блок 1 и т.д. Номер блока b суммируется с начальным адресом a таблицы, образуя реальный адрес строки таблицы для блока b. Найденная строка содержит реальный адрес b начала блока b в реальной памяти. К этому начальному адресу b прибавляется смещение d, так что образуется искомый реальный адрес r=b'+d.

Все методы поблочного отображения, применяемые в системах с сегментной, страничной и комбинированной

странично-сегментной организациями, подобны схеме отображения, называемой схемой прямого отображения.

Страничная организация виртуальной памяти

Виртуальный адрес при чисто страничной организации памяти _ это упорядоченная пара (p, d), где p - номер страницы в виртуальной памяти, а d - смещение в рамках страницы p. Процесс может выполняться, если его текущая страница находится в первичной памяти. Страницы переписываются из внешней памяти в первичную и размещаются в ней в блоках, называемых *страничными кадрами* и имеющих точно такой же размер, как у поступающих страниц. Страничные кадры начинаются в реальной памяти с адресов, кратных фиксированному размеру страниц. *Поступающая страница может быть помещена в любой свободный страничный кадр.*

Для обеспечения работы механизма отображения страниц формируется таблица отображения страниц, каждая строка которой содержит информацию об отображаемой странице виртуальной памяти:

r - признак наличия страницы в первичной памяти (r=0 - страницы в первичной памяти нет; 1 - страница находится в первичной памяти):

S - адрес страницы во внешней памяти (при r=0):

p' - номер страничного кадра в первичной памяти, где размещена виртуальная страница с номером p.

Сегментная организация виртуальной памяти

Виртуальный адрес при сегментной организации виртуальной памяти - это упорядоченная пара n = (s, d) , где s - номер сегмента виртуальной памяти, а d - смещение в рамках этого сегмента. Процесс может выполняться только в том случае, если его текущий сегмент находится в первичной памяти. Сегменты передаются из внешней памяти в первичную целиком. Все ячейки, относящиеся к сегменту, занимают смежные адреса первичной памяти. Для размещения поступающих из внешней памяти сегментов в свободные участки первичной памяти

применяются те же стратегии размещения, как и при распределении переменными неперемещаемыми разделами - "первый подходящий", "самый подходящий", "самый неподходящий". Динамическое преобразование виртуальных адресов в реальные адреса осуществляется в соответствии со схемой прямого отображения.

Странично-сегментная организация виртуальной памяти

Системы со странично-сегментной организацией обладают достоинствами обоих способов реализации виртуальной памяти. Сегменты обычно содержат целое число страниц, причем не обязательно, чтобы все страницы сегмента находились в первичной памяти одновременно, а смежные страницы виртуальной памяти не обязательно должны оказаться смежными в первичной памяти. В системе со странично-сегментной организацией применяется трехкомпонентная (трехмерная) адресация. Виртуальный адрес n здесь определяется как упорядоченная тройка n =(s, p, d), где s - номер сегмента, p - номер страницы, а d - смещение в рамках страницы, где находится нужный элемент.

Операционная система для каждого процесса формирует, во-первых, одну таблицу сегментов процесса, и, во-вторых, таблицы страниц сегментов (по одной на каждый сегмент процесса).

Таблица сегментов процесса содержит в своих строках информацию о количестве страниц в сегменте и о начальных адресах s' размещения таблиц страниц сегментов в первичной памяти ЭВМ.

Каждая страница таблиц сегмента содержит в своих строках информацию о начальном адресе p' размещения в первичной памяти страничного кадра для данной страницы виртуальной памяти.

Динамическое преобразование виртуальных адресов в системах со странично-сегментной организацией отличается от преобразования по схеме наличием еще одного уровня вычисления адреса, как это показано на схеме рис.2.58, и появлением таблиц страниц для каждого сегмента процесса.

Стратегии управления виртуальной памятью

Стратегии управления виртуальной памятью, так же как и стратегии управления физической памятью, разделяются на три категории: стратегии вталкивания, стратегии размещения и стратегии выталкивания.

Целью *стратегий вталкивания* является определить, в какой момент следует переписать страницу или сегмент из вторичной памяти в первичную.

Целью *стратегий размещения* является определить, в какое место первичной памяти помещать поступающую страницу или сегмент.

Целью *стратегий выталкивания* является решить, какую страницу или сегмент следует удалить из первичной памяти, чтобы освободить место для помещения поступающей страницы или сегмента, если первичная память полностью занята.

Большинство стратегий управления виртуальной памятью базируется на концепции *локальности*, суть которой заключается в том, что *распределение запросов процессов на обращение к памяти имеет, как правило, неравномерный характер с высокой степенью локальной концентрации.*

Свойство локальности проявляется как во времени, так и в пространстве.

Локальность во времени означает, что к ячейкам памяти, к которым недавно производилось обращение, с большой вероятностью будет обращение в ближайшем будущем.

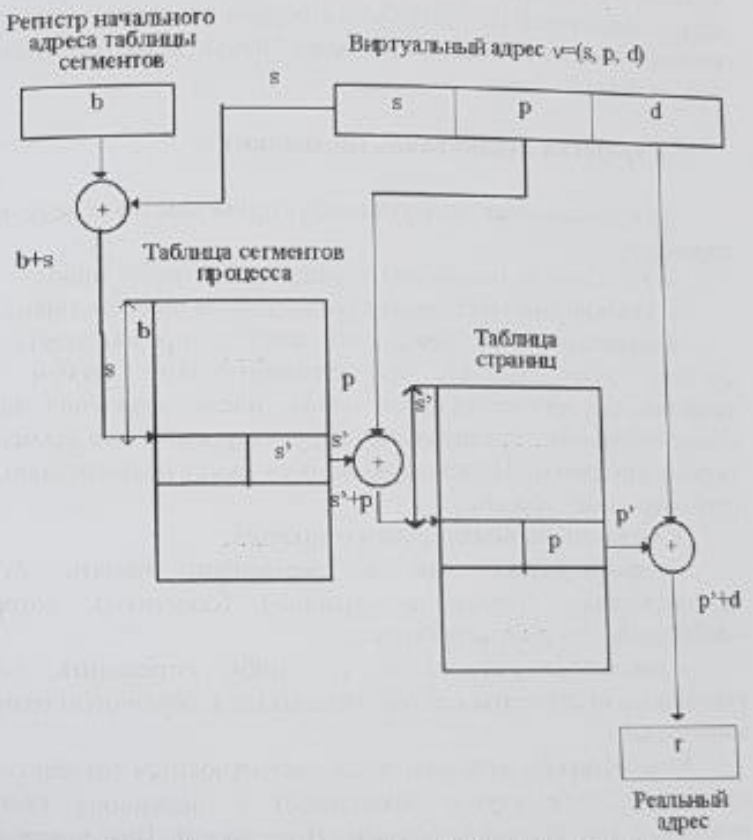


Рис.2.58. Странично –сегментная организация

Локальность в пространстве означает, что обращения к памяти, как правило, концентрируются так, что в случае обращения к некоторой ячейке памяти с большой вероятностью можно ожидать обращение к близлежащим ячейкам.

Свойство локальности наблюдается не только в прикладных программах, но и в работе программ операционной системы. Свойство это скорее эмпирическое (наблюдаемое на практике), чем теоретически обоснованное. Локальность никак нельзя гарантировать, однако ее вероятность достаточно велика. Самым

важным следствием локализации является то, что программа может эффективно работать, если в первичной памяти находится подмножество, включающее наиболее "популярные" ее страницы или сегменты.

Стратегии вталкивания (подкачки)

Для управления вталкиванием применяются следующие стратегии:

- вталкивание (подкачка) по запросу (по требованию);
- вталкивание (подкачка) с упреждением (опережением).

Вталкивание (подкачка) по запросу предполагает, что система ждет ссылки на страницу или сегмент от выполняющегося процесса и только после появления такой ссылки начинает переписывать данную страницу или сегмент в первичную память. Подкачка по запросу имеет положительные и отрицательные стороны.

К положительным сторонам относятся:

- гарантировано, что в первичную память будут переписываться только те страницы (сегменты), которые необходимы для работы процесса;
- накладные расходы на то, чтобы определить, какие страницы или сегменты следует передавать в первичную память, минимальны.

К недостаткам подкачки по запросу относится тот факт, что процесс в этом случае накапливает в первичной памяти требуемые ему страницы (сегменты) по одной. При появлении ссылки на каждую новую страницу (сегмент) процессу приходится ждать, когда эта страница (или сегмент) будет передана в первичную память. В зависимости от того, сколько страниц (сегментов) данного процесса уже находится в первичной памяти, эти периоды ожидания будут обходиться все более дорого, поскольку ожидающие процессы будут занимать все больший объем памяти.

Вталкивание (подкачка) с упреждением предполагает, что система пытается заблаговременно определить, к каким страницам или сегментам будет обращаться процесс. Если вероятность обращения высока и в первичной памяти имеется

свободное место, то соответствующие страницы или сегменты будут переписываться в первичную память еще до того, как к ним будет явно производиться обращение. При правильном выборе страниц (сегментов) для упреждающей подкачки удается существенно сократить общее время выполнения данного процесса и уменьшить значение показателя "пространство-время".

К недостаткам стратегии подкачки с упреждением можно отнести тот факт, что, согласно теории вычислимости, точно предсказать путь, по которому будет развиваться процесс, в общем случае невозможно. Поэтому вполне возможны ситуации, когда решения о выборе станиц (сегментов) для упреждающей подкачки будет в большинстве случаев приниматься неверно для одного или нескольких процессов, развивающихся в системе, что в свою очередь приведет к резкому снижению скорости работы этих процессов из-за увеличения времени ожидания необходимых им страниц или сегментов.

Стратегии размещения

В системах со *страничной* организацией виртуальной памяти решение о размещении вновь загружаемых страниц принимается достаточно просто: *новая страница может быть помещена в любой свободный страничный кадр.*

Для систем с *сегментной* организацией виртуальной памяти применяются такие же стратегии размещения, какие используются в системах распределения памяти переменными разделами, а именно:

- размещение с выбором первого подходящего свободного участка;
- размещение с выбором самого подходящего свободного участка;
- размещение с выбором наименее подходящего свободного участка.

Стратегии выталкивания

В мультипрограммных системах вся первичная память бывает, как правило, занята. В этом случае программа

управления памятью должна решать, какую страницу или какой сегмент следует удалить из первичной памяти, чтобы освободить место для поступающей страницы или сегмента. В настоящее время применяются следующие стратегии выталкивания (откачки) страниц (сегментов):

- выталкивание случайных страниц или сегментов;
- выталкивание первой пришедшей страницы или сегмента (FIFO);
- выталкивание дольше всего не использовавшихся страниц или сегментов (LRU);
- выталкивание наименее часто использовавшихся страниц или сегментов (LFU);
- выталкивание не использовавшихся в последнее время страниц или сегментов (NUR).

Стратегия выталкивания случайных страниц или сегментов является наиболее простой в реализации, обладает малыми издержками и не является дискриминационной по отношению к каким-либо процессам, работающим в системе. В соответствии с этой стратегией любые страницы или сегменты, находящиеся в первичной памяти, могут быть выбраны для выталкивания с равной вероятностью, в том числе даже следующая страница или сегмент, к которым будет производиться обращение (и которые, естественно, удалять из памяти наиболее нецелесообразно). Поскольку подобная стратегия, по сути, рассчитана на "слепое" везение, в реальных системах она применяется редко.

Стратегия выталкивания первой пришедшей страницы или сегмента (FIFO-стратегия) реализует принцип "первый пришел - первый ушел". В этом случае в момент поступления каждой страницы (сегмента) в первичную память ей (ему) присваивается метка времени. Когда появляется необходимость удалить из первичной памяти какую-либо страницу (сегмент), выбирается та страница (сегмент), у которой метка времени имеет наименьшее значение. Аргументом в пользу такой стратегии выталкивания является довод, что у данной страницы уже были возможности "использовать свой шанс", и пора дать подобные возможности другой странице. Однако стратегия FIFO с большой вероятностью будет приводить к удалению из первичной памяти

активно используемых страниц (сегментов), поскольку тот факт, что страница (сегмент) находится в первичной памяти в течение длительного времени, вполне может означать, что эта страница или сегмент постоянно находится в работе.

Стратегия выталкивания дольше всего не использовавшихся страниц или сегментов (LRU-стратегия) предусматривает, что для выталкивания следует выбирать те страницы (сегменты), которые не использовались дольше других. Стратегия LRU требует, чтобы при каждом обращении к страницам (сегментам) их метки времени обновлялись. Это может быть сопряжено с существенными издержками, поэтому LRU-стратегия, несмотря на свою привлекательность, в современных операционных системах реализуется достаточно редко. Кроме того, при реализации LRU-стратегии может быть так, что страница (сегмент), к которой дольше всего не было обращений, в действительности станет следующей используемой страницей (сегментом), если программа к этому моменту очередной раз пройдет большой цикл, охватывающий несколько страниц или сегментов.

Стратегия выталкивания реже всего используемых страниц или сегментов (LFU-стратегия) является одной из наиболее близких к рассмотренной выше LRU-стратегии. В соответствии с LFU-стратегией из первичной памяти выталкиваются наименее часто (наименее интенсивно) использовавшиеся к данному времени страницы или сегменты. Здесь контролируется интенсивность использования страниц (сегментов). Для этого каждой странице (сегменту) назначается счетчик, значение которого увеличивается на единицу при каждом обращении к данной странице (сегменту). LFU-стратегия, будучи интуитивно оправданной, имеет те же недостатки, что и стратегия LRU: во-первых, велика вероятность того, что из первичной памяти будут удалены страницы или сегменты, которые потребуются процессам при следующем обращении к памяти и, во-вторых, ее реализация может быть сопряжена со значительными затратами на организацию контроля интенсивности использования страниц или сегментов.

Стратегия выталкивания не использовавшихся в последнее время страниц или сегментов (NUR-стратегия) также является

близкой к стратегии LRU и характеризуется относительно небольшими издержками на свою реализацию. Согласно NUR-стратегии из первичной памяти выталкиваются те страницы (сегменты), к которым не было обращений в последнее время. В соответствии со свойством локальности во времени к страницам (сегментам), не использовавшимся в последнее время, вряд ли будет обращение в ближайшем будущем, так что их можно заменить на вновь поступающие страницы.

Поскольку желательно заменять те страницы (сегменты), которые в период нахождения в основной памяти не изменялись, реализация NUR-стратегии предусматривает введение двух аппаратных бит-признаков на страницу (сегмент):

- бит-признак b_0 обращения к странице (сегменту);
- бит-признак b_1 модификации страницы (сегмента).

Первоначально все b_0 и b_1 устанавливаются в 0. При обращении к странице (сегменту) соответствующий бит-признак b_0 устанавливается в 1. В случае изменения содержимого страницы (сегмента) соответствующий бит-признак b_1 устанавливается в 1. NUR-стратегия предусматривает существование четырех групп страниц (сегментов), показанных в табл. 2.5.

Таблица 2.5. Группы страниц (сегментов)

Группа	b_0	b_1
1	0	0
2	1	0
3	0	1
4	1	1

В первую очередь из первичной памяти выталкиваются страницы (сегменты), принадлежащие группам с меньшими номерами.

Учет времени, в течение которого к страницам (сегментам) не было обращений, осуществляется периодическим сбрасыванием в 0 всех битов-признаков, выполняемым операционной системой.

Практически любая стратегия выталкивания страниц (сегментов) не исключает опасности нерациональных решений. Это объясняется тем, операционная система не может точно прогнозировать будущее поведение любого из процессов, поступивших к ней на обработку.

2.1.5. Управление вводом-выводом

Внешние устройства, сопровождающие операции ввода-вывода, могут быть объединены в три группы.

- **Работающие с пользователем**. Используются для связи с пользователем компьютера. В качестве примера можно привести принтеры и видеотерминалы, состоящие из дисплея, клавиатуры, а также другие устройства — на пример, манипулятор "мышь".
- **Работающие с компьютером**. Используются для связи с электронным оборудованием. К ним можно отнести дисковые устройства и устройства с магнитной лентой, датчики, контроллеры и преобразователи.
- **Коммуникации**. Используются для связи с удаленными устройствами. К ним относятся модемы и драйверы цифровых линий.

Имеются существенные различия как между устройствами ввода-вывода, принадлежащими к разным классам, так и в рамках каждого класса. Отметим следующие из этих различий.

- **Скорость передачи данных**. Скорость передачи данных может отличаться на несколько порядков (рис. 2.59).
- **Применение**. Каждое действие, поддерживаемое устройством, оказывает влияние на программное обеспечение и стратегии операционной системы. Так, на пример, использующийся для хранения файлов диск требует наличия программного обеспечения для управления файлами. Диск, используемый в качестве внешнего запоминающего устройства для страниц виртуальной памяти, зависит от программных и аппаратных средств виртуальной памяти. Кроме того, данные приложения оказывают воздействия и на алгоритмы дискового планирования (этот вопрос рассматривается в настоящей главе позже). В качестве еще одного примера можно привести терминал, который может использоваться как обычным

близкой к стратегии LRU и характеризуется относительно небольшими издержками на свою реализацию. Согласно NUR-стратегии из первичной памяти выталкиваются те страницы (сегменты), к которым не было обращений в последнее время. В соответствии со свойством локальности во времени к страницам (сегментам), не использовавшимся в последнее время, вряд ли будет обращение в ближайшем будущем, так что их можно заменить на вновь поступающие страницы.

Поскольку желательно заменять те страницы (сегменты), которые в период нахождения в основной памяти не изменялись, реализация NUR-стратегии предусматривает введение двух аппаратных бит-признаков на страницу (сегмент):

- бит-признак b_0 обращения к странице (сегменту);
- бит-признак b_1 модификации страницы (сегмента).

Первоначально все b_0 и b_1 устанавливаются в 0. При обращении к странице (сегменту) соответствующий бит-признак b_0 устанавливается в 1. В случае изменения содержимого страницы (сегмента) соответствующий бит-признак b_1 устанавливается в 1. NUR-стратегия предусматривает существование четырех групп страниц (сегментов), показанных в табл. 2.5.

Таблица 2.5. Группы страниц (сегментов)

Группа	b_0	b_1
1	0	0
2	1	0
3	0	1
4	1	1

В первую очередь из первичной памяти выталкиваются страницы (сегменты), принадлежащие группам с меньшими номерами.

Учет времени, в течение которого к страницам (сегментам) не было обращений, осуществляется периодическим сбрасыванием в 0 всех битов-признаков, выполняемым операционной системой.

Практически любая стратегия выталкивания страниц (сегментов) не исключает опасности нерациональных решений. Это объясняется тем, операционная система не может точно прогнозировать будущее поведение любого из процессов, поступивших к ней на обработку.

2.1.5. Управление вводом-выводом

Внешние устройства, сопровождающие операции ввода-вывода, могут быть объединены в три группы.

- **Работающие с пользователем**. Используются для связи с пользователем компьютера. В качестве примера можно привести принтеры и видеотерминалы, состоящие из дисплея, клавиатуры, а также другие устройства — на пример, манипулятор "мышь".
- **Работающие с компьютером**. Используются для связи с электронным оборудованием. К ним можно отнести дисковые устройства и устройства с магнитной лентой, датчики, контроллеры и преобразователи.
- **Коммуникации**. Используются для связи с удаленными устройствами. К ним относятся модемы и драйверы цифровых линий.

Имеются существенные различия как между устройствами ввода-вывода, принадлежащими к разным классам, так и в рамках каждого класса. Отметим следующие из этих различий.

- **Скорость передачи данных**. Скорость передачи данных может отличаться на несколько порядков (рис. 2.59).
- **Применение**. Каждое действие, поддерживаемое устройством, оказывает влияние на программное обеспечение и стратегии операционной системы. Так, на пример, использующийся для хранения файлов диск требует наличия программного обеспечения для управления файлами. Диск, используемый в качестве внешнего запоминающего устройства для страниц виртуальной памяти, зависит от программных и аппаратных средств виртуальной памяти. Кроме того, данные приложения оказывают воздействия и на алгоритмы дискового планирования (этот вопрос рассматривается в настоящей главе позже). В качестве еще одного примера можно привести терминал, который может использоваться как обычным

пользователем, так и системным администратором — при этом требуются не только различные уровни привилегий, но и, вероятно, различные уровни приоритетов операционной системы.

- **Сложность управления.** Для принтера требуется относительно простой интерфейс управления, диску же необходим намного более сложный интерфейс. Влияние этих отличий на операционную систему сглаживается усложнением контроллеров ввода-вывода.
- **Единицы передачи данных.** Данные могут передаваться как поток байтов или символов (например, при терминальном вводе-выводе), и блоками (например, при выполнении дисковых операций ввода-вывода).
- **Представление данных.** Различные устройства используют разные схемы кодирования данных, включая разную кодировку символов и контроль четности.
- **Условия ошибок.** Природа ошибок, способ сообщения о них, их последствия и возможные ответы резко отличаются при переходе от одного устройства к другому.

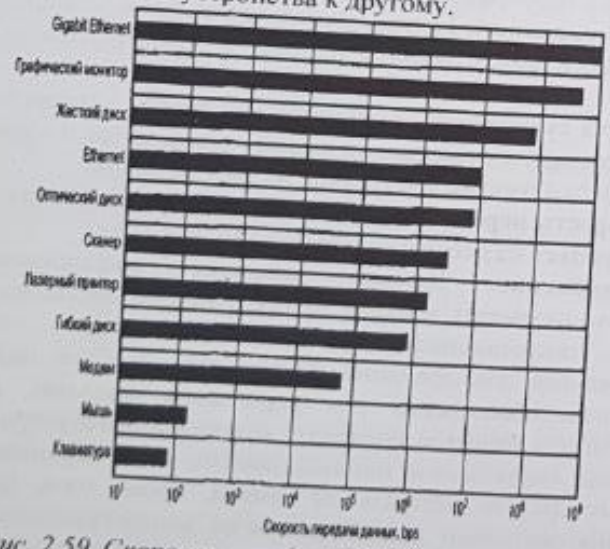


Рис. 2.59. Скорость передачи данных типичных устройств ввода-вывода

Такое разнообразие приводит тому, что, по сути, невозможна разработка единого и согласованного подхода к проблеме ввода-вывода как с точки зрения операционной системы, так и с точки зрения пользовательских процессов.

Организация функций
Три способа осуществления ввода-вывода

- Программируемый ввод-вывод. Процессор посылает необходимые команды контроллеру ввода-вывода; после этого процесс находится в состоянии ожидания завершения операции ввода-вывода.
- Ввод-вывод, управляемый прерываниями. Процессор посылает необходимые команды контроллеру ввода-вывода и продолжает выполнение следующих команд. Выполнение процесса прерывается контроллером ввода- вывода, когда последний выполнит свое задание. Выполняющийся процессором код может быть кодом процесса, обратившегося к устройству ввода- вывода, если нет необходимости в ожидании выполнения операции ввода- вывода. В противном случае процесс приостанавливается до получения прерывания, и процессор переключается на выполнение другого процесса.
- Прямой доступ к памяти. Модуль прямого доступа к памяти управляет обменом данных между основной памятью и контроллером ввода-вывода. Процессор посылает запрос на передачу блока данных модулю прямого доступа к памяти, а прерывание происходит только после передачи всего блока данных.

В большинстве компьютерных систем основным способом передачи данных, поддерживаемым операционной системой, является прямой доступ к памяти.

Параллельно с развитием компьютерных систем возрастает сложность и интеллектуальность их отдельных компонентов, что заметнее всего в области ввода-вывода. **Этапы развития функциональности устройств ввода-вывода можно охарактеризовать следующим образом:**

1. Процессор непосредственно управляет периферийным устройством.

2. К устройству добавляется контроллер или модуль ввода-вывода. Процессор использует программируемый ввод-вывод без прерываний. На этом этапе процессор становится в некоторой степени отделенным от конкретных деталей интерфейсов внешних устройств.

3. Применяется та же конфигурация, что и в пункте 2, только с использованием прерываний. В результате процессору нет необходимости расходовать время на ожидание выполнения операций ввода-вывода, что приводит к увеличению производительности.

4. Модуль ввода-вывода получает возможность непосредственной работы с памятью с использованием DMA. Появляется возможность перемещения блока данных в память или из нее без использования процессора (за исключением моментов начала и окончания передачи данных).

5. Модуль ввода-вывода совершенствуется и становится отдельным процессором, обладающим специализированной системой команд, предназначенных для ввода-вывода. Центральный процессор дает задание процессору ввода- вывода выполнить программу ввода-вывода, находящуюся в основной памяти. Процессор ввода-вывода производит выборку и выполнение соответствующих команд без участия центрального процессора. Такая процедура позволяет центральному процессору определить последовательность выполняемых функций ввода-вывода и быть прерванным только при выполнении всей последовательности.

6. Модуль ввода-вывода обладает своей локальной памятью и является, по сути, отдельным компьютером. При такой архитектуре управление многочисленными устройствами ввода-вывода может осуществляться при минимальном вмешательстве центрального процессора. Обычно такая архитектура используется для управления связью с интерактивными терминалами. Процессор ввода-вывода берет на себя большинство задач, связанных с управлением терминалами.

Если проследить описанный выше путь развития устройств ввода-вывода, то можно заметить, что вмешательство процессора в функции ввода-вывода становится все менее заметным. Центральный процессор все больше и больше освобождается от

задач, связанных с вводом-выводом, что приводит к повышению общей производительности. Этапы 5 и 6 отражают изменение концепции устройства ввода-вывода — отныне он способен к самостоятельному выполнению программы.

Обратите внимание на терминологию. Для всех модулей, описанных в пунктах 4-6, вполне применим термин *"прямой доступ к памяти"*, поскольку каждый из них использует непосредственное управление основной памятью модулем ввода-вывода. Модуль ввода-вывода, описанный в пункте 5, часто называется также каналом ввода-вывода, а модуль, описанный в пункте 6, — процессором ввода-вывода. Впрочем, иногда в литературе каждый из этих терминов используется и для описания другого типа устройств. В оставшейся части главы мы используем термин *"канал ввода-вывода"* для обоих типов модулей ввода-вывода.

Прямой доступ к памяти

На рис. 2.60 представлена логическая схема прямого доступа к памяти. Устройство прямого доступа к памяти способно дублировать функции процессора, в частности получать от процессора управление системой. Эта возможность необходима ему для передачи данных по системной шине — как в память, так и из нее. Обычно модуль DMA использует системную шину лишь в том случае, когда процессор не нуждается в ней (в противном случае ему придется вынудить процессор временно приостановить свою работу). Этот способ наиболее распространен и именуется захватом цикла, так как модуль DMA выполняет захват цикла шины.

Рассмотрим работу схемы прямого доступа к памяти. В тот момент, когда процессору необходимо произвести считывание или запись блока данных, он выполняет запрос к модулю DMA, передавая ему следующую информацию.

- Какая из операций — чтения или записи — запрашивается. В зависимости от этого будет использоваться либо управляющая линия чтения, либо записи между процессором и модулем DMA.

подключенного к данных. используемого устройства ввода-вывода,

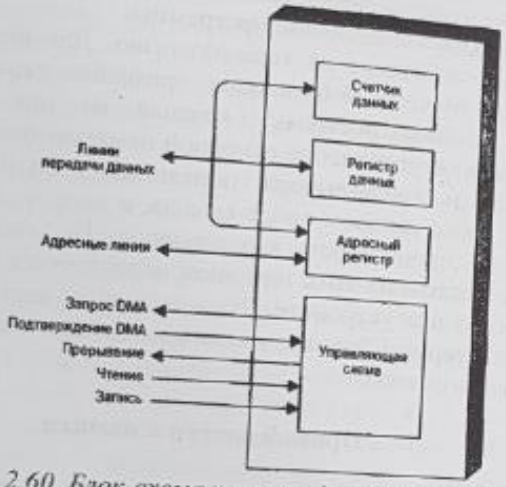


Рис. 2.60. Блок-схема прямого доступа к памяти

- Начальный адрес считываемой (или записываемой) области памяти, хранящийся в адресном регистре модуля DMA.
- Какое количество слов необходимо прочесть или записать. Эта величина хранится в регистре счетчика данных модуля DMA.

После этого процессор продолжает свою работу с другим заданием, передав управление операцией ввода-вывода модулю DMA. В свою очередь модуль DMA, минуя процессор, передает весь блок данных непосредственно в память (или считывает данные из нее). После выполнения передачи данных модуль DMA посылает процессору сигнал прерывания. Таким образом, процессор включается в этот процесс лишь в начале и в конце передачи данных.

На рис. 2.61 показаны позиции цикла команд, в которых работа процессора может быть приостановлена. В любом случае приостановка работы процессора происходит только при необходимости использования шины. После этого устройство

DMA выполняет передачу одного слова и возвращает управление процессору. Обратите внимание на то, что это не прерывание: процессор не сохраняет контекст с переходом к выполнению другого задания. Процессор просто делает паузу на время одного цикла шины. Общее влияние DMA состоит в несколько более замедленной работе процессора. Тем не менее для передачи большого блока данных модулем ввода-вывода метод прямого доступа к памяти более эффективен, чем метод с использованием прерываний или программируемый ввод-вывод.

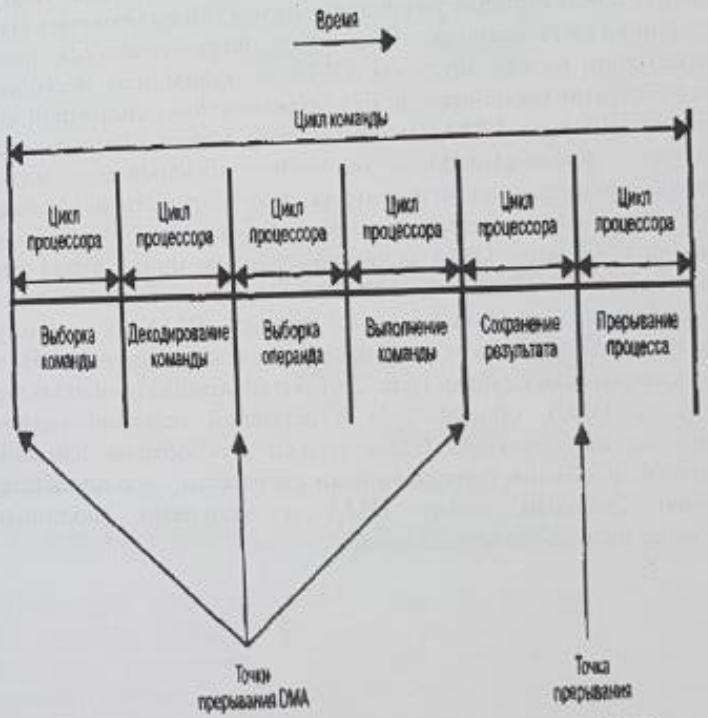


Рис. 2.61. DMA и точки прерывания цикла команд

Конфигурирование прямого доступа к памяти может быть выполнено различными способами; некоторые из них представлены на рис. 2.62. В первом примере все модули

подключены к одной и той же системной шине. Модуль DMA, выступающий в качестве дублера процессора, использует программируемый ввод-вывод для обмена данными между памятью и устройством ввода-вывода с участием модуля DMA. Несмотря на достоинство такой конфигурации, заключающееся в относительной дешевизне, она малоэффективна. Поскольку используется программируемый ввод-вывод под управлением процессора, на передачу каждого слова затрачиваются два цикла шины (после запроса на передачу следует передача данных).

Число необходимых циклов шины может быть в значительной степени уменьшено путем интегрирования DMA и функций ввода-вывода. При этом подразумевается наличие магистрали между модулем DMA и одним или несколькими устройствами ввода-вывода без подключения системной шины. Логический узел DMA на самом деле может быть как частью модуля ввода-вывода, так и отдельным модулем, контролирующим один или несколько устройств ввода-вывода. Эту идею можно развивать путем добавления модулей ввода-вывода к модулю DMA с использованием шины ввода-вывода (рис. 2.61,в). Такая схема позволяет свести количество интерфейсов ввода-вывода в модуле DMA к одному и предусматривает легкое расширение этой конфигурации. Во всех представленных случаях (рис. 2.61,б и в) совместно используемая модулем DMA, процессором и основной памятью системная шина служит модулю DMA только для обмена данными с памятью и обмена управляющими сигналами — с процессором. Обмен данными между DMA и модулями ввода-вывода происходит вне системной шины.

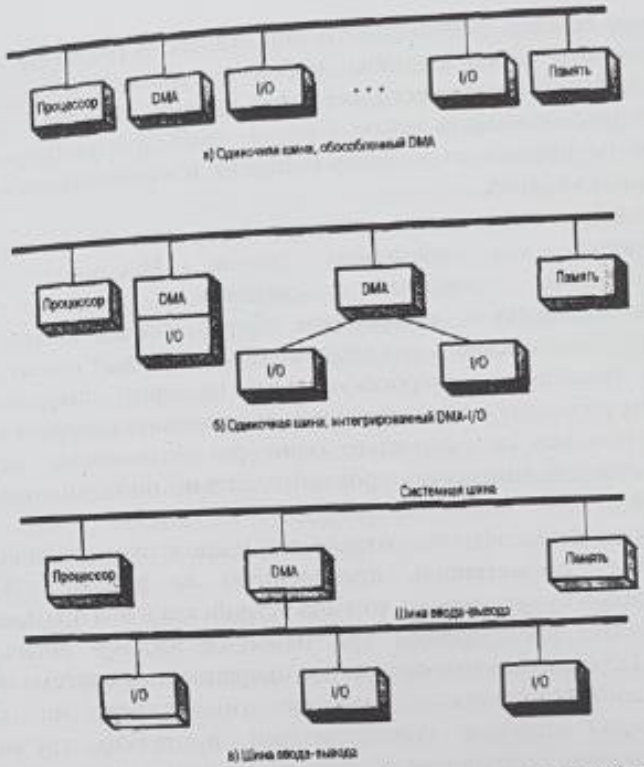


Рис. 2.61. Альтернативные конфигурации прямого доступа к памяти

Логическая структура функций ввода-вывода

Философия иерархии состоит в том, что функции операционной системы следует разделять в соответствии с их сложностью, характерной временной шкалой и уровнем абстракции. Следуя этому подходу, можно прийти к такой организации операционной системы, при которой она будет представлена серией уровней. Каждый уровень представляет связанное подмножество функций, необходимых операционной системе. Выполнение примитивных функций передается более низкому уровню; при этом более высокий уровень не знает

никаких деталей выполнения поставленных задач более низким уровнем. Со своей стороны любой из уровней обеспечивает обслуживание следующего верхнего уровня. В идеале уровни нужно организовывать таким образом, чтобы изменения в одном уровне не приводили к необходимости внесения изменений в остальных уровнях.

Вообще говоря, чем ниже уровень, тем более короткой оказывается его временная шкала. Некоторые части операционной системы должны непосредственно взаимодействовать с аппаратным оборудованием компьютера, где продолжительность различных событий может оказаться на уровне нескольких микросекунд; другие части операционной системы работают с пользователем, ввод команд которым может осуществляться со скоростью один раз в несколько секунд. Использование множества уровней идеально подходит для этих условий.

Такая философия по отношению к средствам ввода-вывода определяет организацию, приведенную на рис.2.62. Детали организации будут зависеть от типа устройства и его применения. На рисунке представлены три наиболее важные логические схемы. Естественно, что конкретная операционная система может и не соответствовать в точности этим схемам, но общие положения остаются справедливыми в любом случае, и большинство операционных систем используют ввод-вывод приблизительно таким образом.

Рассмотрим в первую очередь самый простой случай, когда локальное периферийное устройство осуществляет связь посредством потока байтов или записей (рис.2.62,а). В этом случае уровни будут следующими[15,16].

• *Логический ввод-вывод.* Модуль логического ввода-вывода обращается с устройством как с логическим ресурсом и не обращает внимания на детали фактического управления устройством. Логический модуль ввода-вывода работает посредником между пользовательскими процессами (предоставляя им набор высокоуровневых функций) и устройством.

• *Устройство ввода-вывода.* Запрошенные операции и данные (буферизированные символы, записи и т.п.)

конвертируются в соответствующие последовательности инструкций ввода-вывода, команды управления каналом и команды контроллера. Для более эффективного использования устройства может быть применена буферизация.

• *Планирование и контроль.* На этом уровне происходит реальная организация очередей и планирование операций ввода-вывода, а также управление выполнением операций. Осуществляется работа с прерываниями, получение и передача информации о состоянии устройства. Это уровень программного обеспечения, которое непосредственно взаимодействует с контроллером ввода-вывода, а следовательно, с аппаратным обеспечением устройства.

Для устройств связи структура ввода-вывода (рис.2.62,б) выглядит почти так же, как и рассмотренная выше. Принципиальное отличие состоит в том, что логический модуль ввода-вывода заменяется коммуникационной архитектурой, которая, в свою очередь, может состоять из некоторого количества уровней.

На рис.2.62,в представлена характерная структура управления вводом-выводом во внешнее запоминающее устройство, поддерживающее файловую систему. Здесь имеется три уровня, с которыми мы не сталкивались ранее.

Управление каталогами. На этом уровне происходит преобразование символьных имен файлов в идентификаторы, указывающие на файл — непосредственно или косвенно, с использованием файлового дескриптора или индексной таблицы. Этот уровень также связан с такими пользовательскими операциями с каталогами файлов, как их добавление, удаление или реорганизация.

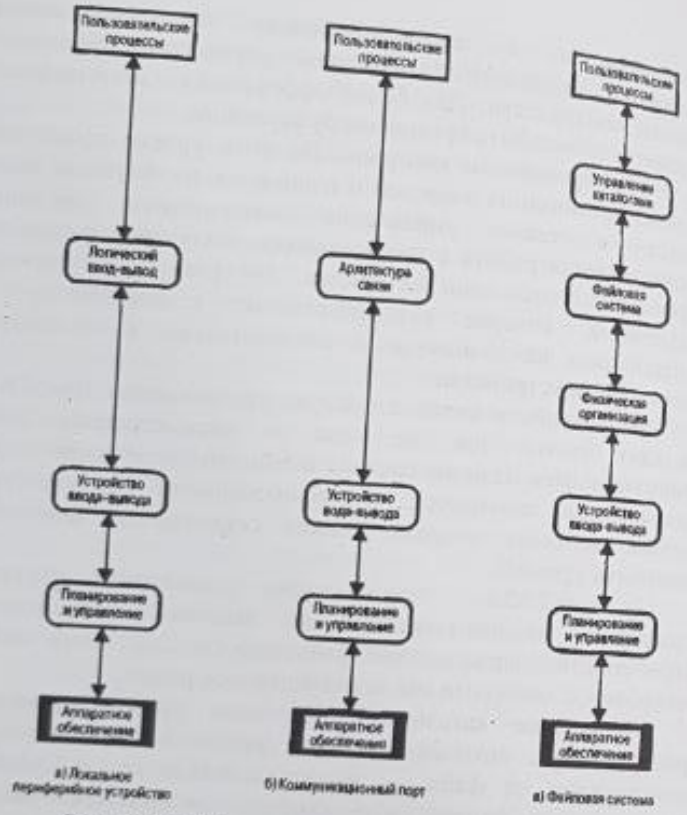


Рис. 2.62. Модель организации ввода-вывода

• **Файловая система**. Этот уровень работает с логической структурой файлов и операциями с ними, такими, как открытие, закрытие, чтение и запись. Кроме того, управление правами доступа также происходит на этом уровне.

• **Физическая организация.** Как адреса виртуальной памяти должны быть преобразованы в физические адреса основной памяти с учетом сегментации и страничной организации, так и логические ссылки на файлы и записи должны быть конвертированы в физические адреса конкретного внешнего запоминающего устройства с учетом физической структуры

дорожек и секторов внешнего запоминающего устройства. На этом же уровне происходит общее управление распределением пространства внешней памяти и буферов основной памяти[12,14].

Буферизация операций ввода-вывода

Предположим, что пользовательскому процессу необходимо выполнить считывание блоков данных длиной по 512 байт по одному с магнитной ленты. Данные будут считаны в область внутри адресного пространства пользовательского процесса с виртуальным адресом от 1000 до 1511. Наиболее простой путь решения этой задачи — выполнение команды ввода-вывода (что-то наподобие Read_Block [1000, tape]) и ожидание того момента, когда данные станут доступными. Ожидание может быть либо активным, т.е. будет происходить непрерывное тестирование состояния устройства, либо, что более практично, процесс будет приостановлен до прерывания.

При таком подходе имеются две проблемы. Первая представляет собой приостановку программы для ожидания выполнения относительно медленного ввода-вывода. Вторая проблема состоит в том, что такой подход к вводу-выводу мешает свопингу. Виртуальные адреса с 1000 по 1511 должны находиться в основной памяти при считывании блока (в противном случае часть данных будет утеряна). При использовании страничной организации памяти по крайней мере одна страница (содержащая целевой адрес) должна быть заблокирована в основной памяти. Поэтому, несмотря на то что часть задания может быть выгружена на диск, полный свопинг процесса окажется невозможным, даже если это необходимо для операционной системы. Следует также учесть возможность взаимоблокировки. При генерации процессом команды ввода-вывода он приостанавливается и выгружается на диск до начала выполнения операции ввода-вывода. Далее процесс ожидает, когда будет выполнена запрошенная им операция ввода-вывода, которая, в свою очередь, ожидает, когда процесс будет возвращен в основную память, поскольку место в основной памяти для считывания данных попросту отсутствует. Для того чтобы избежать взаимоблокировки, пользовательская память, вовлеченная в операцию ввода-вывода, должна быть заблокирована в основной памяти сразу же после выдачи запроса

на ввод-вывод, даже если операция ввода-вывода ставится в очередь и может быть выполнена только через некоторое время.

То же рассуждение применимо и к операции вывода. Если блок пересылается из адресного пространства пользовательского процесса в модуль ввода-вывода, то на время этой передачи процесс блокируется и не может быть выгружен на диск.

Чтобы уменьшить накладные расходы и увеличить эффективность, иногда удобно выполнить чтение данных заранее, до реального запроса (а запись данных — немного позже реального запроса). Эта методика известна как буферизация. В данном разделе мы рассмотрим некоторые схемы буферизации, поддерживаемые операционными системами для повышения производительности.

При рассмотрении различных методов буферизации важно учитывать, что существуют устройства ввода-вывода двух типов: блочно-ориентированные и поточно-ориентированного. Блочно-ориентированные устройства сохраняют информацию блоками, обычно фиксированного размера, и выполняют передачу данных поблочно. Как правило, при этом можно ссылаться на данные с использованием номера блока. Диски и магнитные ленты относятся к блочно-ориентированным устройствам ввода-вывода. Поточно-ориентированные устройства выполняют передачу данных в виде неструктурированных потоков байтов. К этой группе устройств относятся терминалы, принтеры, коммуникационные порты, манипулятор "мышь" и другие указывающие устройства, а также большинство устройств, не являющихся внешними запоминающими устройствами.

Одинарный буфер

Простейшим типом поддержки со стороны операционной системы является одинарный буфер. В тот момент, когда пользовательский процесс выполняет запрос ввода-вывода, операционная система назначает ему буфер в системной части основной памяти.

Схема одинарного буфера для блочно-ориентированных устройств может быть описана следующим образом. Сначала осуществляется передача входных данных в системный буфер.

Когда она завершается, процесс перемещает блок в пользовательское пространство и немедленно производит запрос следующего блока. Такая процедура называется опережающим считыванием, или упреждающим вводом; она выполняется в предположении, что этот блок со временем будет затребован. Для многих типов задач этот метод в большинстве случаев неплохо работает, поскольку доступ к данным обычно осуществляется последовательно (только при окончании последовательности обработки считывание блока будет излишним).

Такой подход, по сравнению с отсутствием буферизации, обеспечивает повышение быстродействия. Пользовательский процесс может обрабатывать один блок данных в то время, когда происходит считывание следующего блока. Операционная система при этом может осуществить выгрузку процесса, поскольку выполняется операция считывания данных в системную память, а не в память пользовательского процесса. Однако такая технология усложняет функционирование операционной системы, которая должна следить за назначением системных буферов пользовательским процессам. Влияет буферизация и на схему подкачки: когда операция ввода-вывода работает с тем же диском, который используется и для свопинга, теряется смысл в организации очереди операций записи. Выгрузка процесса и освобождение основной памяти не начнется до тех пор, пока не завершится запрошенная операция ввода-вывода — а тогда выгрузка процесса больше не будет иметь смысла.

Дисковое планирование

На протяжении последних 30 лет увеличение скорости процессоров и основной памяти осуществляется с большим опережением по сравнению со скоростью доступа к диску. Приблизительно можно сказать, что рост скорости работы процессора и основной памяти на два порядка соответствует росту скорости работы диска на один порядок. В результате скорость обращения к дискам сейчас по меньшей мере на четыре порядка меньше скорости обращения к основной памяти, и

разрыв этот, похоже, в обозримом будущем будет только увеличиваться. Поэтому производительность дисковой системы является жизненно важным вопросом, и множество исследовательских работ направлено на поиск схем ее улучшения[21,22].

Параметры производительности диска

Конкретные детали дисковой операции ввода-вывода зависят от компьютерной системы, операционной системы, природы канала ввода-вывода и аппаратного обеспечения контроллера диска. Обобщенная временная диаграмма передачи данных дисковым устройством ввода-вывода представлена на рис.2.64.

При работе диска его скорость вращения постоянна. Для того чтобы выполнить чтение или запись, головка должна находиться над искомой дорожкой, а кроме того — над началом искомого сектора на этой дорожке. Процедура выбора дорожки включает в себя перемещение головки (в системе с подвижными головками) или электронный выбор нужной головки (в системе с неподвижными головками). В системе с подвижными головками на позиционирование головки над дорожкой затрачивается время, известное как время поиска. В любом случае после выбора дорожки контроллер диска ожидает момент, когда начало искомого сектора достигнет головки. Время, необходимое для достижения головки началом сектора, известно как время задержки из-за вращения, или время ожидания вращения. Сумма времен поиска (если таковой выполняется) и времени задержки из-за вращения составляет время доступа — время, которое требуется для позиционирования для чтения или записи. Как только головка попадает в искомую позицию, выполняется операция чтения или записи, осуществляемая во время движения сектора под головкой, — это и есть непосредственная передача данных при выполнении операции ввода-вывода.

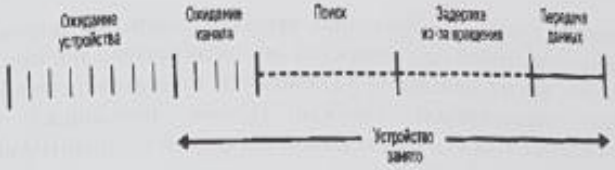


Рис. 2.64. Временная диаграмма работы диска

Кроме этого, существует ряд других задержек, обычно присутствующих в дисковой операции ввода-вывода. Когда процесс выполняет запрос на ввод-вывод, последний должен быть размещен в очереди устройства. После этого выполняется назначение устройства процессу. Если устройство использует каналы ввода-вывода совместно с другими дисками, необходимо дополнительное ожидание доступности канала. И только после этого осуществляется непосредственный доступ к диску, рассмотренный ранее.

В некоторых мейнфреймах используется методика, известная как вращательное позиционное считывание (rotational positional sensing — RPS), работающая следующим образом. При выполнении команды поиска происходит освобождение канала для обработки других операций ввода-вывода. После выполнения поиска устройство определяет момент, когда данные окажутся под головкой. Как только этот сектор подходит к головке, устройство пытается восстановить связь с узлом. Если либо контроллер, либо канал заняты другой операцией ввода-вывода, то попытка восстановления связи оказывается неуспешной и диск совершает полный оборот перед повторной попыткой. Это дополнительный элемент, который следует добавить к полному времени ожидания (рис.2.64).

Время поиска

Время поиска представляет собой время, необходимое для перемещения головки к нужной дорожке; к сожалению, очень трудно установить этот параметр количественно. Время поиска состоит из двух ключевых компонентов: времени начального запуска и времени, необходимого на пересечение дорожек в

процессе поиска. К сожалению, время пересечения дорожек не является линейной функцией от их количества, но при этом включает время начальной установки и принятия решения для каждой пересекаемой дорожки (время, прошедшее после установления головки над искомой дорожкой до идентификации последней).

Значительно улучшает характеристики диска уменьшение и облегчение его компонентов. Не так давно типичный диск имел в диаметре 14 дюймов (36 см), в то время как сегодня самый распространенный размер составляет 3.5 дюйма (8.9 см), что существенно уменьшает расстояния перемещения головок. Типичное среднее время поиска в современных дисках составляет от 5 до 10 ms.

Задержка из-за вращения

Жесткие магнитные диски, в отличие от гибких, имеют скорость вращения в диапазоне от 5400 до 10000 об/мин (последнее значение соответствует одному обороту за 6ms). Поэтому при скорости вращения 10000 об/мин средняя задержка из-за вращения составляет 3 ms. Гибкие диски обычно имеют скорость вращения в пределах 300-600 об/мин, что соответствует средней задержке от 100 до 200 ms.

Время передачи данных

Время передачи данных на диск или считывания с него зависит от скорости вращения диска следующим образом:

$$T = \frac{b}{rN},$$

где

T— время передачи данных; b — количество передаваемых байтов; N — количество байтов в дорожке; r — скорость вращения (об/с). Таким образом, итоговое среднее время доступа можно выразить как

$$T_a = T_s + \frac{1}{2r} + \frac{b}{rN},$$

где T_s — среднее время поиска.

Оценка времени

Рассмотрим теперь две операции ввода-вывода, показывающие, как опасно полагаться на средние значения. Пусть у нас имеется обычный диск, у которого среднее время поиска составляет 10 ms, скорость вращения — 10000 об/мин, и диск разбит на сектора по 512 байт, с 320 секторами на одной дорожке. Предположим, что нам необходимо выполнить чтение файла, состоящего из 2560 секторов, общим размером 1.3 Мбайт. Мы хотим оценить общее время передачи данных.

Сначала предположим, что файл сохранен настолько компактно, насколько это возможно, — т.е. файл занимает все секторы на 8 соседних дорожках (8 дорожек * 320 секторов на дорожке = 2560 секторов). Такое размещение называется последовательным. В этом случае время, необходимое для чтения первой дорожки, определяется следующим образом:

Среднее время поиска 10 ms
Задержка на вращение 3 ms
Чтение 320 секторов 6 ms

Предположим, что остальные дорожки могут быть считаны последовательно, без затраты времени на поиск. Другими словами, операция ввода-вывода не отстает от потока данных с диска. Значит, для каждой последующей дорожки остается только задержка из-за вращения; соответственно, каждая дорожка считывается за 3 + 6 = 9 ms. Итак, для чтения всего файла нам потребуется

Общее время = 19 / 7 * 9 = 82 ms = 0.082 s

Теперь рассчитаем время, необходимое для чтения тех же данных, но при случайном, а не последовательном доступе — т.е. секторы с содержимым файла распределены на диске случайным образом. Тогда для каждого сектора:

Среднее время поиска 10 ms
Задержка на вращение 3 ms
Чтение 1 сектора 0.01875 ms
13.01875 ms

Общее время = 2560 * 13.01875 = 33328 ms = 33.328 s

Очевидно, что порядок чтения секторов с диска оказывает огромное влияние на производительность дискового ввода-вывода. Если при доступе к файлу с диска считывается (или записывается на него) несколько секторов, имеется возможность определенного контроля над использованием секторов с данными, но об этом мы поговорим в следующей главе. Однако в многозадачной среде всегда будут в наличии конкурирующие между собой запросы на операции ввода-вывода с одним и тем же диском, так что избежать случайного доступа не удастся. Таким образом, следует изучить способы повышения производительности дискового ввода-вывода при случайном доступе.

Стратегии дискового планирования

В только что рассмотренном примере причина разницы в производительности может быть объяснена продолжительностью поиска. Если выполнение обращений к секторам включает выбор дорожек случайным образом, производительность дискового ввода-вывода окажется чрезвычайно низкой. Для ее повышения нам необходимо уменьшить время, затрачиваемое на поиск дорожки[7].

Рассмотрим типичную ситуацию в многозадачной среде, когда операционная система поддерживает очередь запросов для каждого устройства ввода-вывода. Соответственно, в очереди одного диска будет находиться некоторое количество запросов на ввод-вывод (чтение или запись) от различных процессов. Если выбирать запросы из очереди случайным образом, то следует ожидать, что искомые дорожки будут располагаться в произвольном порядке, это приведет к очень низкой производительности. Такое случайное распределение может служить точкой отсчета для оценки других методик.

Простейшей формой планирования является планирование "первым вошел — первым вышел" (FIFO), что просто означает обработку запросов из очереди в порядке их поступления. Преимущество такой стратегии — в ее беспристрастности. На рис. 2.65,а показано перемещение головки при использовании стратегии FIFO (в этом примере мы полагаем, что на диске имеется 200 дорожек, а в очереди находятся запросы к дорожкам, поступившие случайным образом: 55, 58, 39, 18, 90, 160, 150, 38, 184). В табл. 2.7,а приведены количественные результаты.

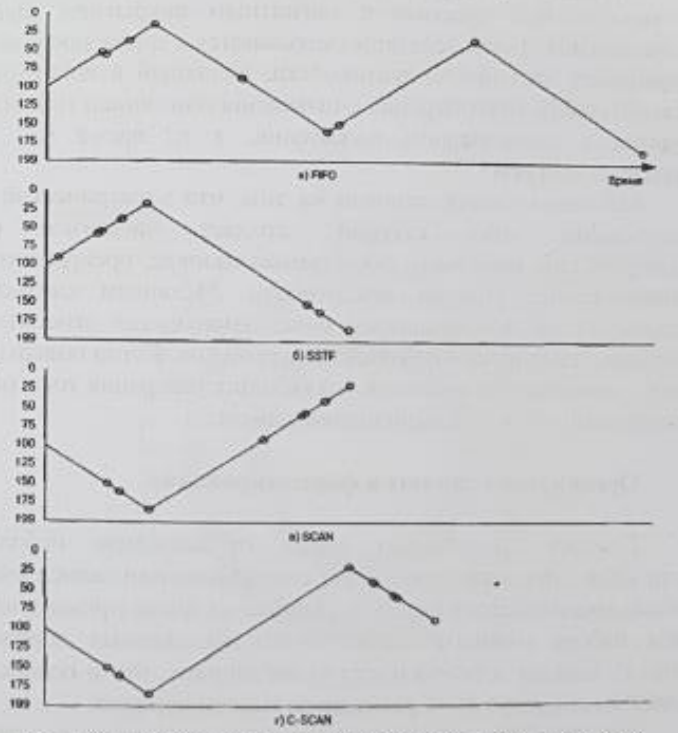


Рис. 2.65. Сравнение различных алгоритмов дискового планирования

При использовании стратегии FIFO надеяться на высокую производительность можно только при небольшом количестве

процессов и запросах в основном к близким группам секторов. Однако при работе большого количества процессов производительность будет почти такой же, как и при случайном планировании. Поэтому следует обратиться к более интеллектуальным стратегиям планирования.

Дисковые устройства Магнитный диск

Диск представляет собой круглую пластину, выполненную из металла или пластика с магнитным покрытием. Данные записываются, а впоследствии считываются с диска посредством проводящей катушки индуктивности, входящей в конструкцию головки (head). При операции считывания или записи головка находится в стационарном положении, в то время как диск вращается под ней.

Механизм записи основан на том, что электрический ток, проходящий через катушку, создает магнитное поле. Электрические импульсы, посылаемые головке, превращаются в намагниченные участки поверхности. Механизм считывания использует то, что магнитное поле, движущееся относительно катушки, генерирует в ней электрический ток. Когда поверхность диска движется под головкой, происходит генерация тока той же полярности, что и при выполнении записи.

Организация данных и форматирование

Головка представляет собой относительно небольшое устройство, предназначенное для считывания или записи участка диска, вращающегося под ней. Данные на диске организуются в виде набора концентрических колец, называемых дорожками (tracks). Каждая дорожка имеет ту же ширину, что и головка. На поверхности могут быть размещены тысячи дорожек.

На рис.2.66 представлена схема размещения данных. Соседние дорожки разделены промежутками (gaps). Это предотвращает (или, по крайней мере, минимизирует) возникновение ошибок, вызванных некорректным положением головки или простой интерференцией магнитных полей. Для упрощения электронной схемы обычно на каждой дорожке

сохраняется одинаковое количество битов. Следовательно, плотность записи, в битах на единицу длины, возрастает при переходе от внешней дорожки ко внутренней (такое явление происходит и при записи звука на пластинку).

Данные пересылаются на диск и из него блоками (blocks). Обычно размер блока меньше, чем емкость дорожки. Соответственно, данные сохраняются в областях, имеющих размер, равный размеру блока; они называются секторами (sectors). Обычно на одной дорожке располагается несколько сотен секторов, и они могут быть либо фиксированной, либо переменной длины. Для большинства дисков используется фиксированный размер сектора — 512 байт. Во избежание ошибок соседние секторы также разделены интервалами.

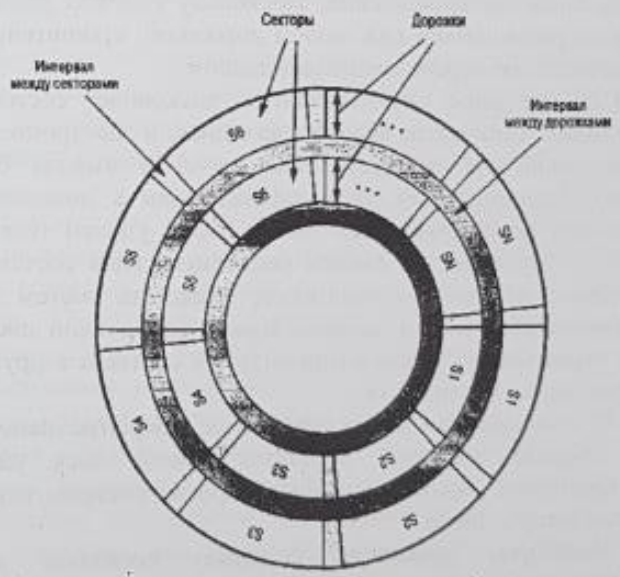


Рис. 2.66. Размещение данных на диске

Для определения позиций секторов на дорожке требуются специальные средства. Ясно, что на дорожке должна существовать некая начальная точка, и должен быть способ идентификации начала и конца каждого сектора. Эти требования

удовлетворяются посредством контроля за записанными на диск данными. Так, диск форматируется с некоторыми дополнительными данными, используемыми только дисководом и недоступными пользователю.

Физические характеристики

Головка диска может быть либо фиксирована, либо перемещаема (в радиальном направлении пластины диска). В дисках с фиксированной головкой на одну дорожку приходится одна головка. Все головки смонтированы на жестком кронштейне, который располагается над всеми дорожками. В диске с перемещающейся головкой имеется только одна головка, закрепленная на кронштейне. Поскольку головка должна быть способна размещаться над любой дорожкой, кронштейн должен обеспечивать ее перемещение над диском.

Сам же диск смонтирован в дисководе, состоящем из кронштейна, шпинделя, вращающего диск, и электронной схемы, необходимой для осуществления ввода и вывода бинарных данных. Стационарный диск зафиксирован в дисководе, в то время как переносной диск может быть удален или заменен другим. Преимущество дисков последнего типа состоит в том, что при ограниченном количестве дисковых систем доступен неограниченный объем данных. Кроме того, такой диск может быть перенесен из одной компьютерной системы в другую (при условии их совместимости).

В большинстве дисков магнитное покрытие наносится на обе стороны дисковой пластины (такой диск называется двусторонним). Некоторые, менее дорогие системы используют односторонние диски.

Некоторые дисководы содержат несколько дисковых пластин, размещенных на вертикальной оси с интервалом в несколько дюймов. При этом предусматривается наличие нескольких кронштейнов. Набор дисковых пластин называется пакетом дисков (рис.2.66). Многопластинчатые диски снабжены перемещаемыми головками, по одной на одну поверхность пластины. Головки механически зафиксированы таким образом, что все они находятся на одинаковом расстоянии от центра диска

и перемещаются одновременно. Таким образом, в любой момент времени все головки оказываются расположенными над дорожками, равноудаленными от центра диска. Множество дорожек, находящихся в одной и той же позиции по отношению к пластине, называется цилиндром (cylinder). Например, все закрашенные дорожки, показанные на рис.2.67, являются частью одного цилиндра.

Возможна еще одна классификация дисков по типам головок. Обычно головка чтения/записи располагается на фиксированном расстоянии над дисковой пластиной. В головках второго типа при считывании или записи осуществляется реальный физический контакт с поверхностью. Такой механизм используется в гибких дисках, имеющих небольшой размер, гибкую пластину и являющихся самыми дешевыми из всех типов дисков.

Чтобы объяснить принцип работы дисков третьего типа, необходимо прокомментировать соотношение между плотностью данных и размером воздушного промежутка. Головка должна генерировать или улавливать электромагнитное поле достаточной для нормального считывания или записи величины. Чем *уже* головка, тем ближе она должна быть расположена к поверхности пластины. Сужение головки предполагает сужение дорожки и, естественно, увеличение плотности записи, что очень желательно. Однако чем ближе головка расположена к поверхности пластины, тем больше вероятность возникновения ошибок, вызванных загрязнениями или физическими дефектами. Дальнейшее развитие технологии привело к появлению винчестерных дисков, головки которых используются в герметичных дисководах, защищенных от загрязнений. Они сконструированы для работы на расстоянии от поверхности пластины, меньшем, чем расстояние при работе с обычными жестко фиксированными головками, что приводит к более высокой плотности записи данных. Головка представляет собой своеобразную аэродинамическую фольгу, парящую над поверхностью пластины при вращении диска. Воздушное давление, создаваемое при вращении диска, достаточно для создания зазора между поверхностью и головкой. Такая

бесконтактная схема обеспечивает работу очень узких, близко расположенных к поверхности головок.

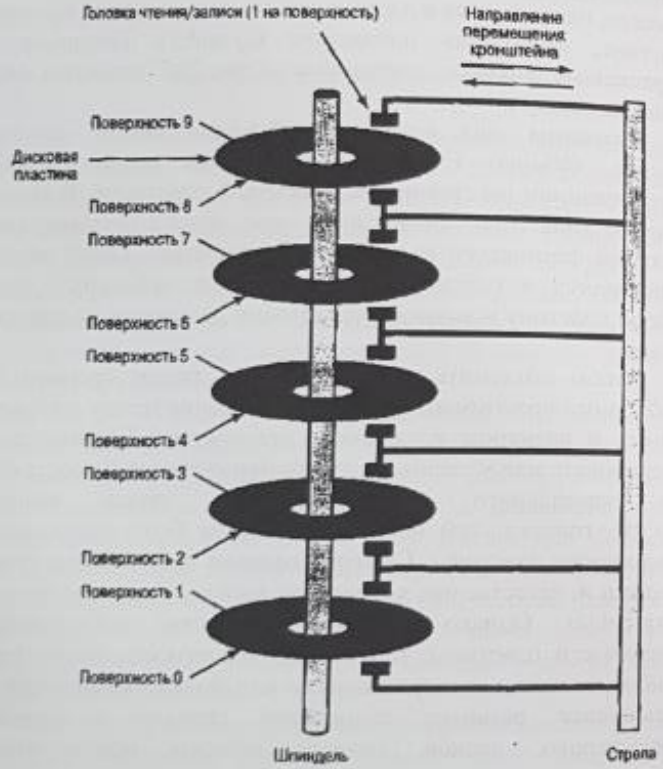


Рис. 2.67. Компоненты магнитного диска

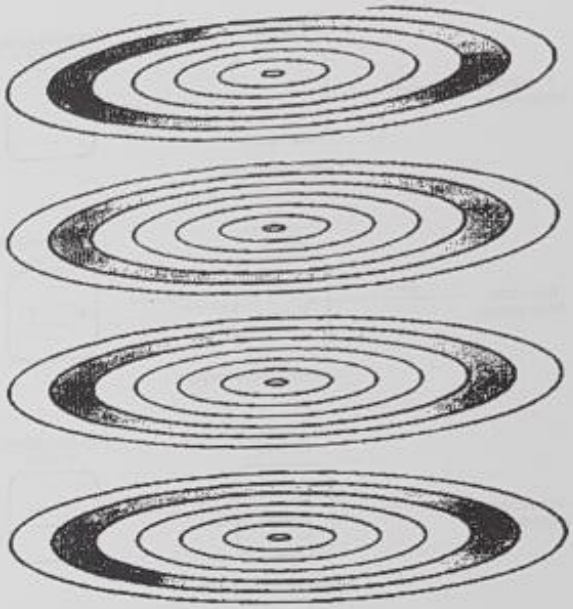


Рис. 2.68. Дорожки и цилиндры

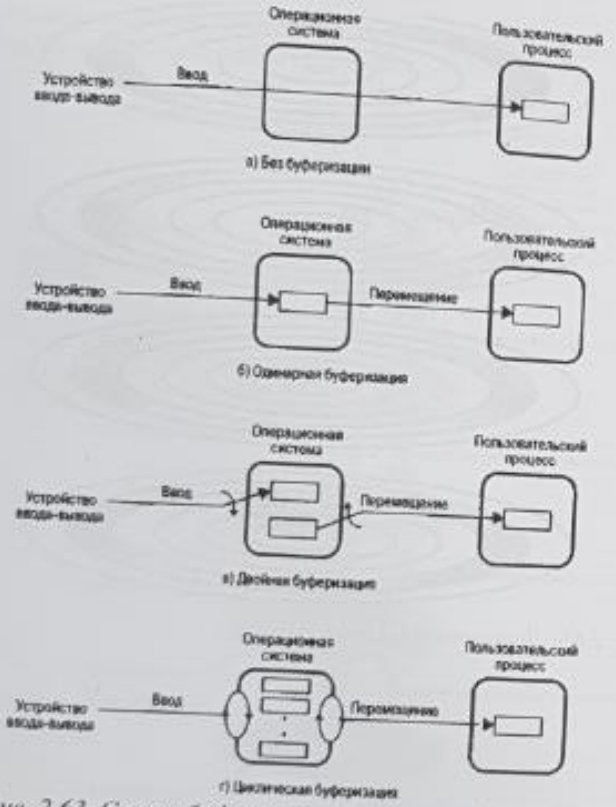


Рис. 2.63. Схемы буферизации ввода-вывода (ввод)

Похожие рассуждения применимы и к блочно-ориентированному выводу. Если данные передаются на устройство, то сначала они копируются из пользовательского пространства в системный буфер, из которого они в конечном счете будут записаны на устройство. В этой ситуации выводящий данные процесс может продолжать работу сразу же после передачи данных в системный буфер.

Схема одинарного буфера может быть применена и при поточно-ориентированном вводе-выводе — построчно или побайтово. Построчная буферизация применима, например, в

неинтеллектуальных терминалах, где пользователь вводит данные построчно, завершая строки символом возврата каретки, сигнализируя об окончании строки; вывод на терминал происходит таким же образом — построчно. Другим примером может служить строчный принтер. Побайтовые операции применяются при использовании терминалов с формами, а также многих других периферийных устройств, когда каждое нажатие клавиши является значимым[22].

В случае построчного ввода-вывода буфер может быть использован для хранения одной строки. Пользовательский процесс приостанавливается на время ввода, ожидая поступления целой строки. При операции вывода пользовательский процесс может разместить строку в буфере и продолжить работу. Необходимость приостановления этого процесса возникает только в том случае, если требуется вывод второй строки, в то время как первая еще не покинула буфер.

Двойной буфер

Улучшить схему одинарной буферизации можно путем использования двух системных буферов (рис. 2.63,в). Теперь процесс выполняет передачу данных в один буфер (или считывание из него), в то время как операционная система освобождает (или заполняет) другой. Эта технология известна как двойная буферизация или сменный буфер.

Время выполнения при блочно-ориентированной передаче данных можно грубо оценить как max[C, T]. Таким образом, если $C<T$, то блочно-ориентированное устройство может работать с максимальной скоростью. Если же $C>T$, то двойная буферизация избавляет процесс от необходимости ожидания завершения ввода-вывода. В любом случае достигается преимущество перед одинарной буферизацией. Это улучшение буферизации осуществляется за счет увеличения ее сложности.

При поточно-ориентированном вводе мы снова обращаемся к двум альтернативным режимам работы. Необходимость приостановления процесса при построчном выводе возникает только в том случае, если при выводе очередной строки оба буфера не пусты. При побайтовых операциях двойной буфер не

прерываний, а в соответствующие регистры – информация из слова состояния. В более развитых процессорах, например в том же i80286 и последующих 32-битовых микропроцессорах, начиная с i80386, осуществляется достаточно сложная процедура определения начального адреса соответствующей подпрограммы обработки прерывания и не менее сложная процедура инициализации рабочих регистров процессора.

4. Сохранение информации о прерванной программе, которую не удалось спасти на шаге 2 с помощью действий аппаратуры. В некоторых вычислительных системах предусматривается запоминание довольно большого объема информации о состоянии прерванного процесса.

5. Обработка прерывания. Эта работа может быть выполнена той же подпрограммой, которой было передано управление на шаге 3, но в ОС чаще всего она реализуется путем последующего вызова соответствующей подпрограммы.

6. Восстановление информации, относящейся к прерванному процессу (этап, обратный шагу 4).

7. Возврат в прерванную программу.

Шаги 1–3 реализуются аппаратно, а шаги 4–7 — программно.

На рис.2.69 показано, что при возникновении запроса на прерывание естественный ход вычислений нарушается и управление передается программе обработки возникшего прерывания. При этом средствами аппаратуры сохраняется (как правило, с помощью механизмов стековой памяти) адрес той команды, с которой следует продолжить выполнение прерванной программы. После выполнения программы обработки прерывания управление возвращается прерванной ранее программе посредством занесения в указатель команд сохраненного адреса команды. Однако такая схема используется только в самых простых программных средах. В мультипрограммных операционных системах обработка прерываний происходит по более сложным схемам, о чем будет более подробно написано ниже.

Итак, главные функции механизма прерываний:

– распознавание или классификация прерываний;

– передача управления соответственно обработчику прерываний;

– корректное возвращение к прерванной программе.

Переход от прерываемой программы к обработчику и обратно должен выполняться как можно быстрей. Одним из быстрых методов является использование таблицы, содержащей перечень всех допустимых для компьютера прерываний и адреса соответствующих обработчиков. Для корректного возвращения к прерванной программе перед передачей управления обработчику прерываний содержимое регистров процессора запоминается либо в памяти с прямым доступом, либо в системном стеке – system stack.

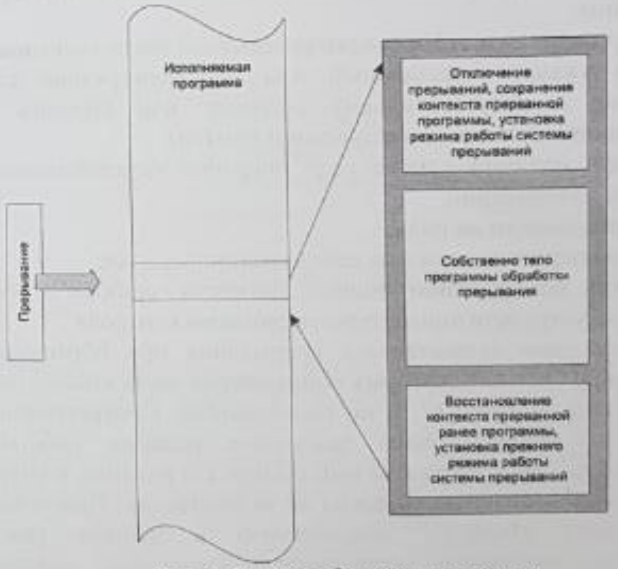


Рис. 2.69. Обработка прерывания

Прерывания, возникающие при работе вычислительной системы, можно разделить на два основных класса: внешние (их иногда называют асинхронными) и внутренние (синхронные).

Внешние прерывания вызываются асинхронными событиями, которые происходят вне прерываемого процесса, например:

– прерывания от таймера;

– прерывания от внешних устройств (прерывания по вводу/выводу);

– прерывания по нарушению питания;

– прерывания с пульта оператора вычислительной системы;

– прерывания от другого процессора или другой вычислительной системы.

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

– при нарушении адресации (в адресной части выполняемой команды указан запрещенный или несуществующий адрес, обращение к отсутствующему сегменту или странице при организации механизмов виртуальной памяти);

– при наличии в поле кода операции незадействованной двоичной комбинации;

– при делении на ноль;

– при переполнении или исчезновении порядка;

– при обнаружении ошибок четности, ошибок в работе различных устройств аппаратуры средствами контроля.

Могут еще существовать прерывания при обращении к супервизору ОС – в некоторых компьютерах часть команд может использовать только ОС, а не пользователи. Соответственно в аппаратуре предусмотрены различные режимы работы, и пользовательские программы выполняются в режиме, в котором эти привилегированные команды не исполняются. При попытке использовать команду, запрещенную в данном режиме, происходит внутреннее прерывание и управление передается супервизору ОС. К привилегированным командам относятся и команды переключения режима работа центрального процессора.

Наконец, существуют собственно **программные прерывания**. Эти прерывания происходят по соответствующей команде прерывания, то есть по этой команде процессор осуществляет практически те же действия, что и при обычных

внутренних прерываниях. Данный механизм был специально введен для того, чтобы переключение на системные программные модули происходило не просто как переход в подпрограмму, а точно таким же образом, как и обычное прерывание. Этим обеспечивается автоматическое переключение процессора в привилегированный режим с возможностью исполнения любых команд.

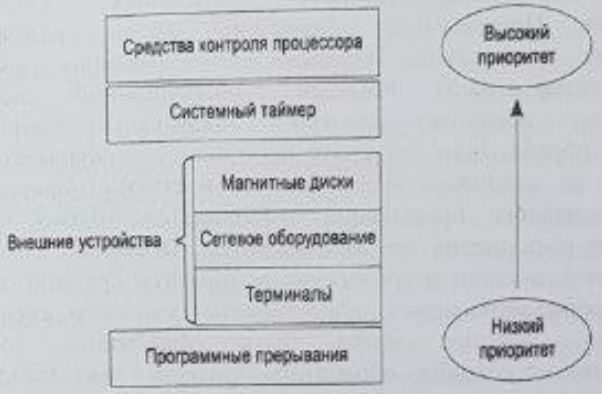


Рис. 2.70. Распределение прерываний по уровням приоритета

Сигналы, вызывающие прерывания, формируются вне процессора или в самом процессоре; они могут возникать одновременно. Выбор одного из них для обработки осуществляется на основе приоритетов, приписанных каждому типу прерывания. Очевидно, что прерывания от схем контроля процессора должны обладать наивысшим приоритетом (если аппаратура работает неправильно, то не имеет смысла продолжать обработку информации). На рис.2.70 изображен обычный порядок (приоритеты) обработки прерываний в зависимости от типа прерываний. Учет приоритета может быть встроен в технические средства, а также определяться операционной системой, то есть кроме аппаратно реализованных приоритетов прерывания большинство вычислительных машин и

комплексов допускают программно-аппаратное управление порядком обработки сигналов прерывания. Второй способ, дополняя первый, позволяет применять различные дисциплины обслуживания прерываний.

Наличие сигнала прерывания не обязательно должно вызывать прерывание исполняющейся программы. Процессор может обладать средствами защиты от прерываний: *отключение* системы прерываний, *маскирование* (запрет) отдельных сигналов прерывания. Программное управление этими средствами (существуют специальные команда для управления работой системы прерываний) позволяет операционной системе регулировать обработку сигналов прерывания, заставляя процессор обрабатывать их сразу по приходу, откладывать их обработку на некоторое время или полностью игнорировать. Обычно операция прерывания выполняется только после завершения выполнения текущей команды. Поскольку сигналы прерывания возникают в произвольные моменты времени, то на момент прерывания может существовать несколько сигналов прерывания, которые могут быть обработаны только последовательно. Чтобы обработать сигналы прерывания в разумном порядке им (как уже отмечалось) присваиваются приоритеты. Сигнал с более высоким приоритетом обрабатывается в первую очередь, обработка остальных сигналов прерывания откладывается[15].

Программное управление специальными регистрами маски (маскирование сигналов прерывания) позволяет реализовать различные дисциплины обслуживания:

– с **относительными приоритетами**, то есть обслуживание не прерывается даже при наличии запросов с более высокими приоритетами. После окончания обслуживания данного запроса обслуживается запрос с наивысшим приоритетом. Для организации такой дисциплины необходимо в программе обслуживания данного запроса наложить маски на все остальные сигналы прерывания или просто отключить систему прерываний;

– с **абсолютными приоритетами**, то есть всегда обслуживается прерывание с наивысшим приоритетом. Для реализации этого режима необходимо на время обработки

прерывания замаскировать все запросы с более низким приоритетом. При этом возможно многоуровневое прерывание, то есть прерывание программ обработки прерываний. Число уровней прерывания в этом режиме изменяется и зависит от приоритета запроса;

- по **принципу стека**, или, как иногда говорят, по дисциплине LCFS (last come first served – последним пришел – первым обслужен), то есть запросы с более низким приоритетом могут прерывать обработку прерывания с более высоким приоритетом. Для этого необходимо не накладывать маски ни на один сигнал прерывания и не выключать систему прерываний.

Следует особо отметить, что для правильной реализации последних двух дисциплин нужно обеспечить полное маскирование системы прерываний при выполнении шагов 1-4 и 6-7. Это необходимо для того, чтобы не потерять запрос и правильно его обслужить. Многоуровневое прерывание должно происходить на этапе собственно обработки прерывания, а не на этапе перехода с одного процесса на другой.

Управление ходом выполнения задач со стороны ОС заключается в организации реакций на прерывания, в организации обмена информацией (данными и программами), предоставлении необходимых ресурсов, в динамике выполнения задачи и в организации сервиса. Причины прерываний определяет ОС (модуль, который называют *супервизором прерываний*), она же и выполняет действия, необходимые при данном прерывании и в данной ситуации. Поэтому в состав любой ОС реального времени прежде всего входят программы управления системой прерываний, контроля состояний задач и событий, синхронизации задач, средства распределения памяти и управления ею, а уже потом средства организации данных (с помощью файловых систем и т. д.). Следует, однако, заметить, что современная ОС реального времени должна вносить в аппаратно-программный комплекс нечто большее, нежели просто обеспечение быстрой реакции на прерывания.

Как было указано ранее, при появлении запроса на прерывание система прерываний идентифицирует сигнал и, если прерывания разрешены, управление передается на соответствующую подпрограмму обработки. Из рис.2.69 видно,

что в подпрограмме обработки прерывания имеются две служебные секции. Это – первая секция, в которой осуществляется сохранение контекста прерванной задачи, который не смог быть сохранен на 2-м шаге, и последняя, заключительная секция, в которой, наоборот, осуществляется восстановление контекста. Для того чтобы система прерываний не среагировала повторно на сигнал запроса на прерывание, она обычно автоматически «закрывает» (отключает) прерывания, поэтому необходимо потом в подпрограмме обработки прерываний вновь включать систему прерываний. Установка рассмотренных режимов обработки прерываний (с относительными и абсолютными приоритетами, и по правилу LCFS) осуществляется в конце первой секции подпрограммы обработки. Таким образом, на время выполнения центральной секции (в случае работы в режимах с абсолютными приоритетами и по дисциплине LCFS) прерывания разрешены. На время работы заключительной секции подпрограммы обработки система прерываний должна быть отключена и после восстановления контекста вновь включена. Поскольку эти действия необходимо выполнять практически в каждой подпрограмме обработки прерываний, во многих операционных системах первые секции подпрограмм обработки прерываний выделяются в специальный системный программный модуль, называемый супервизором прерываний.

Супервизор прерываний прежде всего сохраняет в дескрипторе текущей задачи рабочие регистры процессора, определяющие контекст прерываемого вычислительного процесса. Далее он определяет ту подпрограмму, которая должна выполнить действия, связанные с обслуживанием настоящего (текущего) запроса на прерывание. Наконец, перед тем как передать управление этой подпрограмме, супервизор прерываний устанавливает необходимый режим обработки прерывания. После выполнения подпрограммы обработки прерывания управление вновь передается супервизору ОС, на этот раз уже на тот модуль, который занимается диспетчеризацией задач. И уже диспетчер задач, в свою очередь, в соответствии с принятым режимом распределения процессорного времени (между выполняющимися процессами) восстановит контекст той задачи,

которой будет решено выделить процессор. Рассмотренная схема проиллюстрирована на рис.2.71.

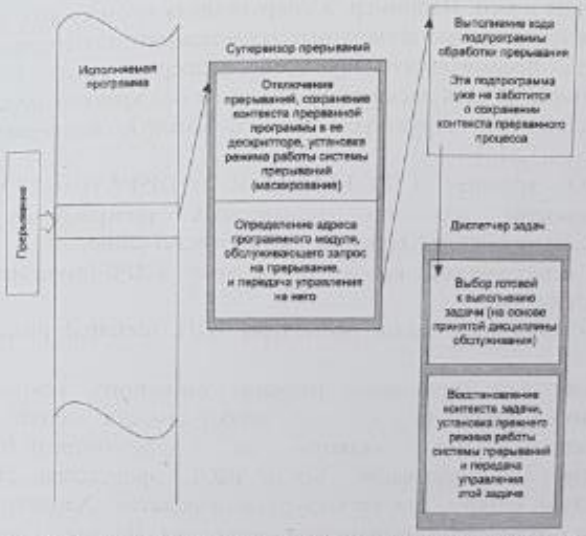


Рис. 2.71. Обработка прерывания при участии супервизоров ОС

Как видно из рис.2.71, здесь нет непосредственного возврата в прерванную ранее программу непосредственно из самой подпрограммы обработки прерывания. Для прямого непосредственного возврата достаточно адрес возврата сохранить в стеке, что и делает аппаратура процессора. При этом стек легко обеспечивает возможность возврата в случае вложенных прерываний, поскольку он всегда реализует дисциплину LCFS (last come – first served).

Однако если бы контекст процессов сохранялся просто в стеке, как это обычно реализуется аппаратурой, а не в описанных выше дескрипторах задач, то не было бы возможности гибко подходить к выбору той задачи, которой нужно передать процессор после завершения работы подпрограммы обработки

прерывания. Естественно, что это только общий принцип. В конкретных процессорах и в конкретных ОС могут существовать некоторые отступления от рассмотренной схемы и/или дополнения к ней. Например, в современных процессорах часто имеются специальные аппаратные возможности для сохранения контекста прерываемого процесса непосредственно в его дескрипторе, то есть дескриптор процесса (по крайней мере его часть) становится структурой данных, которую поддерживает аппаратура[12,13].

IRQL уровни 1 (*APC* LEVEL) и 2 (DISPATCH LEVEL) предназначены для так называемых программных (в терминологии Microsoft) прерываний соответственно:

- асинхронный вызов процедуры - *APC* (asynchronous procedure call) и
- отложенный вызов процедуры -*DPC* (deferred procedure call).

Если ядро принимает решение выполнить некоторую системную процедуру, но нет необходимости делать это немедленно, оно ставит ее в очередь *DPC* и генерирует *DPC* прерывание. Когда IRQL процессора станет достаточно низким, эта процедура выполняется. Характерный пример - отложенная операция планирования. Из этого следует, что код, выполняемый на IRQL уровне, выше или равном 2, не подвержен операции планирования.

1) DPC предназначен для обслуживания программных прерываний, исходящих от модулей ядра. Сюда помещаются программные запросы, вызывающие диспетчер потоков (т.к. ситуация обработки прерываний часто требует перепланирования потоков, но эта процедура менее критичная, чем собственно обработка прерывания.)

2) Кроме того, на этом уровне ожидают своей очереди не только вызовы планировщика/диспетчера, но и другие отложенные вызовы от других процедур.

Асинхронный вызов процедур - механизм, аналогичный механизму *DPC*, но более общего назначения, в частности, доступный пользовательским процессам. Эти процедуры могут прерывать выполнение обыкновенного кода, вызываются они с

помощью программного прерывания и выполняются в текущем контексте.

Структура драйвера. Точки входа в драйвер

В отличие от прикладной программы, драйвер не является процессом и не имеет потока исполнения. Вместо этого любая функция драйвера выполняется в контексте того потока и процесса, в котором она была вызвана. При этом вызов может происходить от прикладной программы или драйвера, либо возникать в результате прерывания. В первом случае контекст исполнения драйвера точно известен - это прикладная программа. В третьем случае контекст исполнения случайный, поскольку прерывание (и, соответственно, исполнение кода драйвера) может произойти при выполнении любой прикладной программы. Во втором случае контекст исполнения может быть как известным, так и случайным - это зависит от контекста исполнения функции вызывающего драйвера.

Под **вызовом драйвера** здесь подразумевается не обычный вызов функции, а передача так называемого запроса ввода/вывода.

При написании любого драйвера необходимо помнить четыре основных момента:

1. возможные точки входа драйвера;
2. контекст, в котором могут быть вызваны точки входа драйвера;
3. последовательность обработки типичных запросов;
4. Уровень IRQL, при котором вызывается точка входа, и, следовательно, ограничения на использование некоторых функций ОС: каждая (!!!) функция ОС может быть вызвана только при определенных уровнях IRQL (см. описание любой функции в DDK, там всегда указаны эти уровни).

Архитектура драйвера Windows NT использует модель точек входа, в которой Диспетчер Ввода/вывода вызывает специфическую подпрограмму в драйвере, когда требуется, чтобы драйвер выполнил специфическое действие. В каждую точку входа передается определенный набор параметров для драйвера, чтобы дать возможность ему выполнить требуемую

функцию. Базовая структура драйвера состоит из набора точек входа, наличие которых обязательно, плюс некоторое количество точек входа, наличие которых зависит от назначения драйвера.

Далее перечисляются точки входа либо классы точек входа драйвера:

1. DriverEntry. Диспетчер Ввода/вывода вызывает эту функцию драйвера при первоначальной загрузке драйвера. Внутри этой функции драйверы выполняют инициализацию как для себя, так и для любых устройств, которыми они управляют. Эта точка входа требуется для всех NT драйверов.

2. Диспетчерские (Dispatch) точки входа. Точки входа Dispatch драйвера вызываются Диспетчером Ввода/вывода, чтобы запросить драйвер инициировать некоторую операцию ввода/вывода.

3. Unload. Диспетчер Ввода/вывода вызывает эту точку входа, чтобы запросить драйвер подготовиться к немедленному удалению из системы. Только драйверы, которые поддерживают выгрузку, должны реализовывать эту точку входа. В случае вызова этой функции, драйвер будет выгружен из системы при выходе из этой функции вне зависимости от результата ее работы.

4. Fast I/O. Вместо одной точки входа, на самом деле это набор точек входа. Диспетчер Ввода/вывода или Диспетчер Кэша вызывают некоторую функцию быстрого ввода/вывода (Fast I/O), для инициирования некоторого действия "Fast I/O". Эти подпрограммы поддерживаются исключительно драйверами файловой системы.

5. Управление очередями запросов IRP (сериализация - процесс выполнения различных транзакций в нужной последовательности). Два типа очередей: Системная очередь (StartIo) и очереди, управляемые драйвером.

6. Reinitialize. Диспетчер Ввода/вывода вызывает эту точку входа, если она была зарегистрирована, чтобы позволить драйверу выполнить вторичную инициализацию.

7. Точка входа процедуры обработки прерывания (ISR-Interrupt Service Routine). Эта точка входа присутствует, только если драйвер поддерживает обработку прерывания.

8. Точки входа вызовов отложенных процедур (DPC - Deferred Procedure Call). Два типа DPC: DpcForIsr и CustomDpc. Драйвер использует эти точки входа, чтобы завершить работу, которая должна быть сделана в результате появления прерывания или другого специального условия[15,16].

Объект, описывающий драйвер. Объект, описывающий устройство. Объект, описывающий файл. Взаимосвязь объектов.

Файловый объект - это объект, видимый из режима пользователя, который представляет всевозможные открытые источники или приемники ввода/вывода: файл на диске или устройство (физическое, логическое, виртуальное). Физическим устройством может быть, например, последовательный порт, физический диск; логическим -логический диск; виртуальным - виртуальный сетевой адаптер, именованный канал, почтовый ящик. Всякий раз, когда некоторый поток открывает файл, создается новый файловый объект с новым набором атрибутов. В любой момент времени сразу несколько файловых объектов могут быть ассоциированы с одним разделяемым виртуальным файлом, но каждый такой файловый объект имеет уникальный описатель, корректный только в контексте процесса, поток которого инициировал открытие файла. Файловые объекты, как и другие объекты, имеют иерархические имена, охраняются объектной защитой, поддерживают синхронизацию и обрабатываются системными сервисами.

Объект-драйвер представляет в системе некоторый драйвер Windows NT и хранит для диспетчера ввода/вывода адреса стандартных процедур (точки входа), которые драйвер может или должен иметь в зависимости от того, является ли он драйвером верхнего или нижнего уровней. Объект-драйвер описывает также, где драйвер загружен в физическую память и размер драйвера. Объект-драйвер описывается частично документированной структурой данных DRIVER_OBJECT. Этот объект создается менеджером ввода/вывода при загрузке драйвера в систему, после чего диспетчер вызывает процедуру инициализации драйвера DriverEntry и передает ей указатель на объект-драйвер. Объект-драйвер является скрытым для кода пользовательского уровня, то есть только определенные

компоненты уровня ядра (в том числе и диспетчер ввода/вывода) знают внутреннюю структуру этого типа объекта и могут получать доступ ко всем данным, содержащимся в объекте, напрямую.

Диспетчер ввода/вывода определяет тип объекта - **объект-устройство**, используемый для представления физического, логического или виртуального устройства, чей драйвер был загружен в систему. Формат объекта-устройство определяется частично документированной структурой данных DEVICE_OBJECT. Хотя объект-устройство может быть создан в любое время посредством вызова функции loCreateDeviceQ, обычно он создается внутри DriverEntry. Объект-устройство описывает характеристики устройства, такие как требование по выравниванию буферов и местоположение очереди устройства, для хранения поступающих пакетов запросов ввода/вывода.

При создании объекта-устройства также может быть указано его имя, которое будет видимо в директории «\Device» пространства имен диспетчера объектов. Объекты- устройства используются в Windows NT как точки входа в пространства имен, не контролируемые менеджером объектов. Если при разборе имени объекта диспетчер объектов встречает объект-устройство, то он вызывает метод разбора, связанный с этим устройством.

Для обращения к именованному объекту-устройству из подсистемы Win32 посредством функции CreateFile() должны быть предприняты дополнительные действия. Функция CreateFile() ищет имя устройства в директории Диспетчера Объектов «\??» (NT 4.0 и Win2000), либо «\DosDevices»(NT 3.51). Поэтому в соответствующей директории, а для большей совместимости это должна быть «\DosDevices», должен быть создан объект (**символическая связь**), указывающий на имя устройства в директории «\Device». Обычно связь создается в самом драйвере, хотя это можно сделать и из прикладной программы пользовательского режима функцией DefineDosDevice().

2.1.7. Организация файлов и модель доступа

Требования к хранению информации:

- возможность хранения больших объемов данных
- информация должна сохраняться после прекращения работы процесса
- несколько процессов должны иметь одновременный доступ к информации

Именование файлов

Длина имени файла зависит от ОС, может быть от 8 (MS-DOS) до 255 (Windows, LINUX) символов.

ОС могут различать прописные и строчные символы. Например, WINDOWS и windows для MS-DOS одно и тоже, но для UNIX это разные файлы.

Во многих ОС имя файла состоит из двух частей, разделенных точкой, например windows.exe. Часть после точки называют **расширением файла**. По нему система различает тип файла.

У MS-DOS расширение составляет 3 символа. По нему система различает тип файла, а также можно его исполнять или нет.

У UNIX расширение ограничено размером имени файла в 255 символов, также у UNIX может быть несколько расширений, но расширениями пользуются больше прикладные программы, а не ОС. По расширению UNIX не может определить исполняемый это файл или нет.

Структура файла

Три основные структуры файлов:

1. **Последовательность байтов** - ОС не интересуется содержимым файла, она видит только байты. Основное преимущество такой системы, это гибкость использования. Используются в Windows и UNIX[14].
2. **Последовательность записей** - записей фиксированной длины (например, перфокарта), считываются последовательно. Сейчас не используются.
3. **Дерево записей** - каждая запись имеет ключ, записи считываются по ключу. Основное преимущество такой системы, это скорость поиска. Пока еще используется на мэйнфреймах.

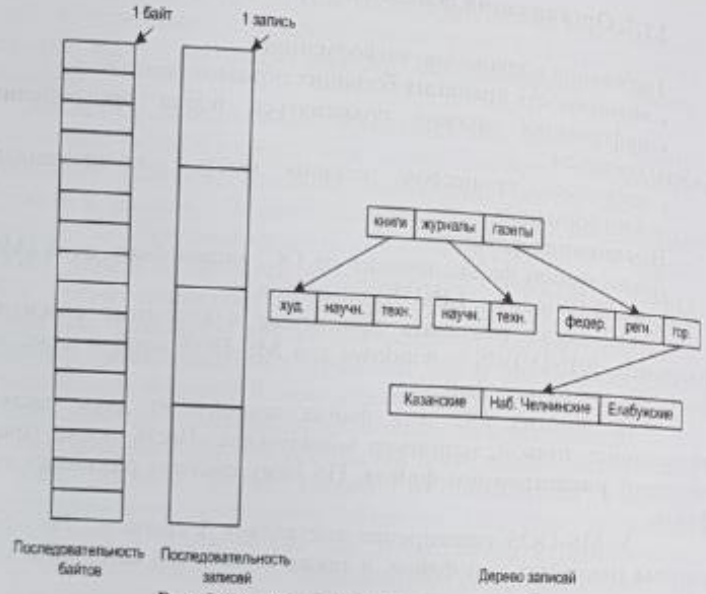


Рис.2.72. Три типа структур файла.

Типы файлов

Основные типы файлов:

- **Регулярные** - содержат информацию пользователя. Используются в Windows и UNIX.
- **Каталоги** - системные файлы, обеспечивающие поддержку структуры файловой системы. Используются в Windows и UNIX.
- **Символьные** - для моделирования ввода-вывода. Используются только в UNIX.
- **Блочные** - для моделирования дисков. Используются только в UNIX.

Основные типы регулярных файлов:

- **ASCII файлы** - состоят из текстовых строк. Каждая строка завершается возвратом каретки (Windows), символом перевода строки (UNIX) и используются оба варианта (MS-DOS).

Поэтому если открыть текстовый файл, написанный в UNIX, в Windows, то все строки сольются в одну большую строку, но под MS-DOS они не сольются (*это достаточно частая ситуация*). Основные преимущества ASCII файлов:

- могут отображаться на экране, и выводится на принтер без преобразований
- могут редактироваться почти любым редактором

- **Двоичные файлы** - остальные файлы (не ASCII). Как правило, имеют внутреннюю структуру.

Основные типы двоичных файлов:

- **Исполняемые** - программы, их может обрабатывать сама операционная система, хотя они записаны в виде последовательности байт.
- **Неисполняемые** - все остальные.

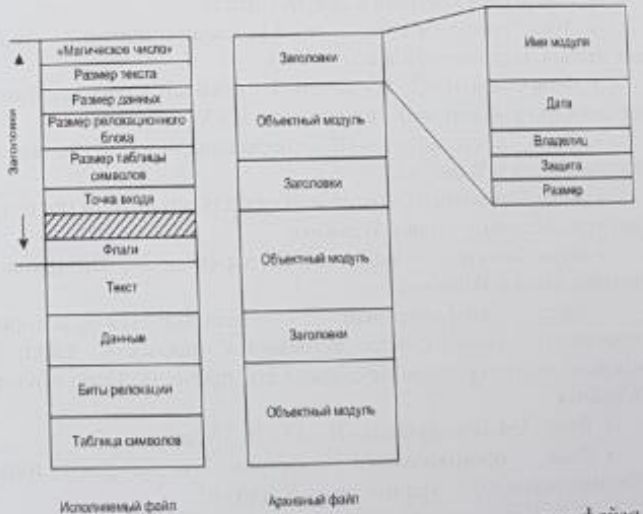


Рис.2.73. Примеры исполняемого и не исполняемого файла

Доступ к файлам

Основные виды доступа к файлам:

• **Последовательный** - байты читаются по порядку. Использовались, когда были магнитные ленты.

• **Произвольный** - файл можно читать с произвольной точки. Основное преимущество возникает, когда используются большие файлы (например, баз данных) и надо считать только часть данных из файла. Все современные ОС используют этот доступ.

Атрибуты файла

Основные атрибуты файла:

• Защита - кто, и каким образом может получить доступ к файлу (пользователи, группы, чтение/запись). Используются в Windows и UNIX.

• Пароль - пароль к файлу

• Создатель - кто создал файл

• Владелец - текущий владелец файла

• Флаг "только чтение" - 0 - для чтения/записи, 1 - только для чтения. Используются в Windows.

• Флаг "скрытый" - 0 - виден, 1 - невиден в перечне файлов каталога (по умолчанию). Используются в Windows.

• Флаг "системный" - 0 - нормальный, 1 - системный. Используются в Windows.

• Флаг "архивный" - готов или нет для архивации (не путать сжатием). Используются в Windows.

• Флаг "сжатый" - файл сжимается (подобие zip архивов). Используются в Windows.

• Флаг "шифрованный" - используется алгоритм шифрования. Если кто-то попытается прочесть файл, не имеющий на это прав, он не сможет его прочесть. Используются в Windows.

• Флаг ASCII/двоичный - 0 - ASCII, 1 - двоичный

• Флаг произвольного доступа - 0 - только последовательный, 1 - произвольный доступ

• Флаг "временный" - 0 - нормальный, 1 - для удаления файла по окончании работы процесса

• Флаг блокировки - блокировка доступа к файлу. Если он занят для редактирования.

• Время создания - дата и время создания. Используются UNIX.

• Время последнего доступа - дата и время последнего доступа

• Время последнего изменения - дата и время последнего изменения. Используются в Windows и UNIX.

• Текущий размер - размер файла. Используются в Windows и UNIX.

Операции с файлами

Основные системные вызовы для работы с файлами:

• Create - создание файла без данных.

• Delete - удаление файла.

• Open - открытие файла.

• Close - закрытие файла.

• Read - чтение из файла, с текущей позиции файла.

• Write - запись в файл, в текущею позицию файла.

• Append - добавление в конец файла.

• Seek - устанавливает файловый указатель в определенную позицию в файле.

• Get attributes - получение атрибутов файла.

• Set attributes - установить атрибутов файла.

• Rename - переименование файла.

Файлы, отображаемые на адресное пространство памяти

Иногда удобно файл отобразить в памяти (не надо использовать системные вызовы ввода-вывода для работы с файлом), и работать с памятью, а потом записать измененный файл на диск[11].

При использовании страничной организации памяти, файл целиком не загружается, а загружаются только необходимые страницы.

При использовании сегментной организации памяти, файл загружают в отдельный сегмент.

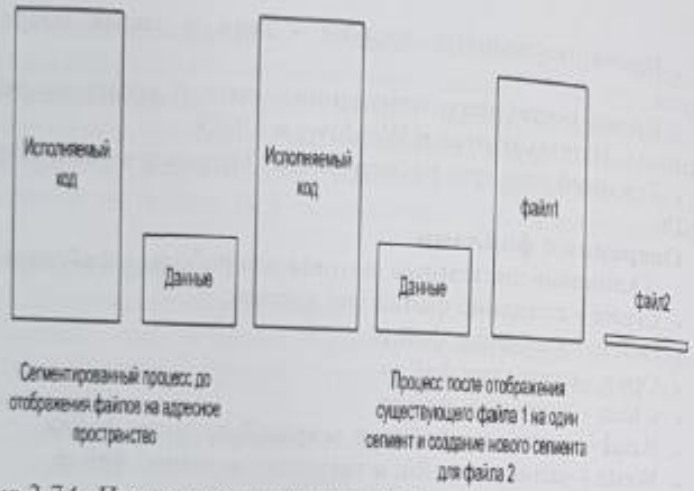


Рис.2.74. Пример копирования файла через отображение в памяти.

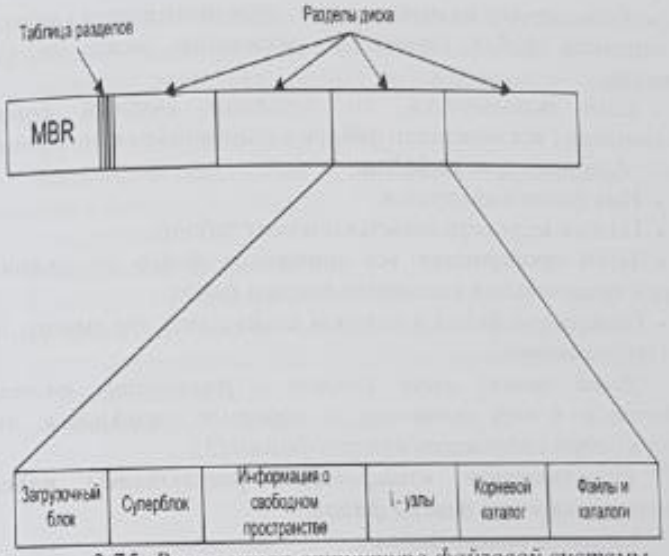
Алгоритм:

1. Создается сегмент для файла 1
2. Файл отображается в памяти
3. Создается сегмент для файла 2
4. Сегмент 1 копируется в сегмент 2
5. Сегмент 2 сохраняется на диске

Недостатки этого метода:

- Тяжело определить длину выходного файла
- Если один процесс отобразил файл в памяти и изменил его, но файл еще не сохранен, второй процесс откроет это же файл, и будет работать с устаревшим файлом.
- Файл может оказаться большим, больше сегмента или виртуального пространства.

Структура файловой системы



2.75. Возможная структура файловой системы

Все что до "Загрузочного блока" и включая его одинаково у всех ОС. Дальше начинаются различия.

Суперблок - содержит ключевые параметры файловой системы.

Ускорение поиска файлов

Если каталог очень большой (несколько тысяч файлов), последовательное чтение каталога мало эффективно.

1 Использование хэш-таблицы для ускорения поиска файла.

Алгоритм записи файла:

- Создается хэш-таблица в начале каталога, с размером n (n записей).
- Для каждого имени файла применяется хэш-функция, такая, чтобы при хэшировании получалось число от 0 до n-1.
- Исследуется элемент таблицы соответствующий хэш-коду.

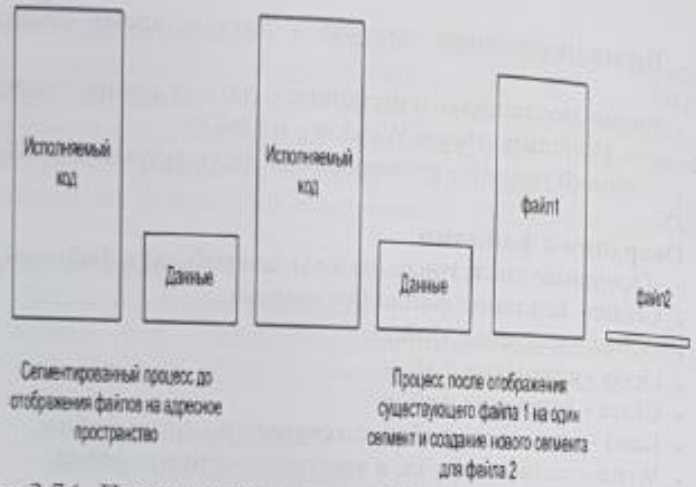


Рис.2.74. Пример копирования файла через отображение в памяти.

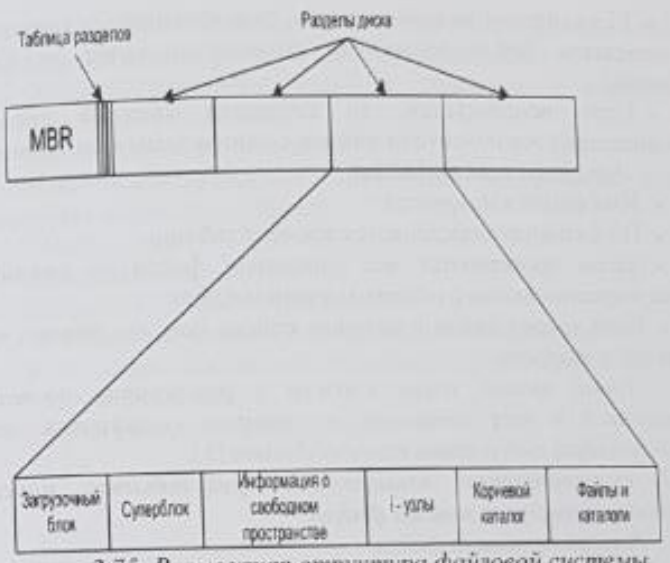
Алгоритм:

1. Создается сегмент для файла 1
2. Файл отображается в памяти
3. Создается сегмент для файла 2
4. Сегмент 1 копируется в сегмент 2
5. Сегмент 2 сохраняется на диске

Недостатки этого метода:

- Тяжело определить длину выходного файла
- Если один процесс отобразил файл в памяти и изменил его, но файл еще не сохранен, второй процесс откроет это же файл, и будет работать с устаревшим файлом.
- Файл может оказаться большим, больше сегмента или виртуального пространства.

Структура файловой системы



2.75. Возможная структура файловой системы

Все что до "Загрузочного блока" и включая его одинаково у всех ОС. Дальше начинаются различия.

Суперблок - содержит ключевые параметры файловой системы.

Ускорение поиска файлов

Если каталог очень большой (несколько тысяч файлов), последовательное чтение каталога мало эффективно.

1 Использование хэш-таблицы для ускорения поиска файла.

Алгоритм записи файла:

- Создается хэш-таблица в начале каталога, с размером n (n записей).
- Для каждого имени файла применяется хэш-функция, такая, чтобы при хэшировании получалось число от 0 до n-1.
- Исследуется элемент таблицы соответствующий хэш-коду.

• Если элемент не используется, туда помещается указатель на описатель файла (описатели размещены вслед за хэш-таблицей).

• Если используется, то создается связный список, объединяющие все описатели файлов с одинаковым хэш-кодом.

Алгоритм поиска файла:

• Имя файла хэшируется

• По хэш-коду определяется элемент таблицы

• Затем проверяются все описатели файла из связного списка и сравниваются с искомым именем файла

• Если имени файла в связном списке нет, это значит, что файла нет в каталоге.

Такой метод очень сложен в реализации, поэтому используется в тех системах, в которых ожидается, что каталоги будут содержать тысячи файлов[5].

2 Использование кэширования результатов поиска файлов для ускорения поиска файла.

Алгоритм поиска файла:

• Проверяется, нет ли имени файла в кэше

• Если нет, то ищется в каталоге, если есть, то берется из кэша

Такой способ дает ускорение только при частом использовании одних и тех же файлов.

Надежность файловой системы

Резервное копирование

Случаи, для которых необходимо резервное копирование:

• Аварийные ситуации, приводящие к потере данных на диске

• Случайное удаление или программная порча файлов

Основные принципы создания резервных копий:

• Создавать несколько копий - ежедневные, еженедельные, ежемесячные, ежеквартальные.

• Как правило, необходимо сохранять не весь диск, а толь выборочные каталоги.

• Применять **инкрементные резервные копии** - сохраняются только измененные файлы

• Сжимать резервные копии для экономии места

• Фиксировать систему при создании резервной копии, чтобы вовремя резервирования система не менялась.

• Хранить резервные копии в защищенном месте, не доступном для посторонних.

Существует две стратегии:

• Физическая архивация - поблочное копирование диска (копируются блоки, а не файлы)

Недостатки:

- копирование пустых блоков
- проблемы с дефектными блоками
- не возможно применять инкрементное копирование
- не возможно копировать отдельные каталоги и файлы

Преимущества:

- высокая скорость копирования

2.2. УПРАВЛЕНИЕ РАСПРЕДЕЛЕННЫМИ РЕСУРСАМИ

2.2.1. Средства коммуникации и классификация примитивов

Единственным по-настоящему важным отличием распределенных систем от централизованных является межпроцессная взаимосвязь. В централизованных системах связь между процессами, как правило, предполагает наличие разделяемой памяти. Типичный пример - проблема "поставщик-потребитель", в этом случае один процесс пишет в разделяемый буфер, а другой - читает из него. Даже наиболее простая форма синхронизации - семафор - требует, чтобы хотя бы одно слово (переменная самого семафора) было разделяемым. В распределенных системах нет какой бы то ни было разделяемой памяти, таким образом вся природа межпроцессных коммуникаций должна быть продумана заново[5-7].

Основой этого взаимодействия может служить только передача по сети сообщений. В самом простом случае системные средства обеспечения связи могут быть сведены к двум основным системным вызовам (примитивам), один - для посылки сообщения, другой - для получения сообщения. В дальнейшем на их базе могут быть построены более мощные средства сетевых коммуникаций, такие как распределенная файловая система или вызов удаленных процедур, которые, в свою очередь, также могут служить основой для построения других сетевых сервисов.

Несмотря на концептуальную простоту этих системных вызовов - ПОСЛАТЬ и ПОЛУЧИТЬ - существуют различные варианты их реализации, от правильного выбора которых зависит эффективность работы сети. В частности, эффективность коммуникаций в сети зависит от способа задания адреса, от того, является ли системный вызов блокирующим или неблокирующим, какие выбраны способы буферизации сообщений и насколько надежным является протокол обмена сообщениями.

Способы адресации

Для того, чтобы послать сообщение, необходимо указать адрес получателя. В очень простой сети адрес может задаваться в виде константы, но в более сложных сетях нужен и более изощренный способ адресации.

Одним из вариантов адресации на верхнем уровне является использование физических адресов сетевых адаптеров. Если в получающем компьютере выполняется только один процесс, то ядро будет знать, что делать с поступившим сообщением - передать его этому процессу. Однако, если на машине выполняется несколько процессов, то ядру не известно, какому из них предназначено сообщение, поэтому использование сетевого адреса адаптера в качестве адреса получателя приводит к очень серьезному ограничению - на каждой машине должен выполняться только один процесс.

Альтернативная адресная система использует имена назначения, состоящие из двух частей, определяющие номер машины и номер процесса. Однако адресация типа "машина-процесс" далека от идеала, в частности, она не гибка и не прозрачна, так как пользователь должен явно задавать адрес машины-получателя. В этом случае, если в один прекрасный день машина, на которой работает сервер, отказывает, то программа, в которой жестко используется адрес сервера, не сможет работать с другим сервером, установленном на другой машине.

Другим вариантом могло бы быть назначение каждому процессу уникального адреса, который никак не связан с адресом машины. Одним из способов достижения этой цели является использование централизованного механизма распределения адресов процессов, который работает просто, как счетчик. При получении запроса на выделение адреса он просто возвращает текущее значение счетчика, а затем наращивает его на единицу. Недостатком этой схемы является то, что централизованные компоненты, подобные этому, не обеспечивают в достаточной степени расширяемость систем. Еще один метод назначения процессам уникальных идентификаторов заключается в разрешении каждому процессу выбора своего собственного идентификатора из очень большого адресного пространства,

такого как пространство 64-х битных целых чисел. Вероятность выбора одного и того же числа двумя процессами является ничтожной, а система хорошо расширяется. Однако здесь имеется одна проблема: как процесс-отправитель может узнать номер машины процесса-получателя. В сети, которая поддерживает широковещательный режим (то есть в ней предусмотрен такой адрес, который принимают все сетевые адаптеры), отправитель может широковещательно передать специальный пакет, который содержит идентификатор процесса назначения. Все ядра получат эти сообщения, проверят адрес процесса и, если он совпадает с идентификатором одного из процессов этой машины, пошлют ответное сообщение "Я здесь", содержащее сетевой адрес машины[9-11].

Хотя эта схема и прозрачна, но широковещательные сообщения перегружают систему. Такой перегрузки можно избежать, выделив в сети специальную машину для отображения высокоуровневых символьных имен. При применении такой системы процессы адресуются с помощью символьных строк, и в программы вставляются эти строки, а не номера машин или процессов. Каждый раз перед первой попыткой связаться, процесс должен послать запрос специальному отображающему процессу, обычно называемому сервером имен, запрашивая номер машины, на которой работает процесс-получатель.

Совершенно иной подход - это использование специальной аппаратуры. Пусть процессы выбирают свои адреса случайно, а конструкция сетевых адаптеров позволяет хранить эти адреса. Теперь адреса процессов не обнаруживаются путем широковещательной передачи, а непосредственно указываются в кадрах, заменяя там адреса сетевых адаптеров.

Блокирующие и неблокирующие примитивы

Примитивы бывают блокирующими и неблокирующими, иногда они называются соответственно синхронными и асинхронными. При использовании блокирующего примитива, процесс, выдавший запрос на его выполнение, приостанавливается до полного завершения примитива.

Например, вызов примитива ПОЛУЧИТЬ приостанавливает вызывающий процесс до получения сообщения.

При использовании неблокирующего примитива управление возвращается вызывающему процессу немедленно, еще до того, как требуемая работа будет выполнена. Преимуществом этой схемы является параллельное выполнение вызывающего процесса и процесса передачи сообщения. Обычно в ОС имеется один из двух видов примитивов и очень редко - оба. Однако выигрыш в производительности при использовании неблокирующих примитивов компенсируется серьезным недостатком: отправитель не может модифицировать буфер сообщения, пока сообщение не отправлено, а узнать, отправлено ли сообщение, отправитель не может. Отсюда сложности в построении программ, которые передают последовательность сообщений с помощью неблокирующих примитивов.

Имеется два возможных выхода. Первое решение - это заставить ядро копировать сообщение в свой внутренний буфер, а затем разрешить процессу продолжить выполнение. С точки зрения процесса эта схема ничем не отличается от схемы блокирующего вызова: как только процесс снова получает управление, он может повторно использовать буфер.

Второе решение заключается в прерывании процесса-отправителя после отправки сообщения, чтобы проинформировать его, что буфер снова доступен. Здесь не требуется копирование, что экономит время, но прерывание пользовательского уровня делает программирование запутанным, сложным, может привести к возникновению гонок.

Вопросом, тесно связанным с блокирующими и неблокирующими вызовами, является вопрос тайм-аутов. В системе с блокирующим вызовом ПОСЛАТЬ при отсутствии ответа вызывающий процесс может заблокироваться навсегда. Для предотвращения такой ситуации в некоторых системах вызывающий процесс может задать временной интервал, в течение которого он ждет ответ. Если за это время сообщение не поступает, вызов ПОСЛАТЬ завершается с кодом ошибки.

Буферизуемые и небуферизуемые примитивы

Примитивы, которые были описаны, являются небуферизуемыми примитивами. Это означает, что вызов ПОЛУЧИТЬ сообщает ядру машины, на которой он выполняется, адрес буфера, в который следует поместить пребывающее для него сообщение.

Эта схема работает прекрасно при условии, что получатель выполняет вызов ПОЛУЧИТЬ раньше, чем отправитель выполняет вызов ПОСЛАТЬ. Вызов ПОЛУЧИТЬ сообщает ядру машины, на которой выполняется, по какому адресу должно поступить ожидаемое сообщение, и в какую область памяти необходимо его поместить. Проблема возникает тогда, когда вызов ПОСЛАТЬ сделан раньше вызова ПОЛУЧИТЬ. Каким образом сможет узнать ядро на машине получателя, какому процессу адресовано вновь поступившее сообщение, если их несколько? И как оно узнает, куда его скопировать?

Один из вариантов - просто отказаться от сообщения, позволить отправителю взять тайм-аут и надеяться, что получатель все-таки выполнит вызов ПОЛУЧИТЬ перед повторной передачей сообщения. Этот подход не сложен в реализации, но, к сожалению, отправитель (или скорее ядро его машины) может сделать несколько таких безуспешных попыток. Еще хуже то, что после достаточно большого числа безуспешных попыток ядро отправителя может сделать неправильный вывод об аварии на машине получателя или о неправильности использованного адреса.

Второй подход к этой проблеме заключается в том, чтобы хранить хотя бы некоторое время, поступающие сообщения в ядре получателя на тот случай, что вскоре будет выполнен соответствующий вызов ПОЛУЧИТЬ. Каждый раз, когда поступает такое "неожидаемое" сообщение, включается таймер. Если заданный временной интервал истекает раньше, чем происходит соответствующий вызов ПОЛУЧИТЬ, то сообщение теряется.

Хотя этот метод и уменьшает вероятность потери сообщений, он порождает проблему хранения и управления преждевременно поступившими сообщениями. Необходимы

буферы, которые следует где-то размещать, освобождать, в общем, которыми нужно управлять. Концептуально простым способом управления буферами является определение новой структуры данных, называемой почтовым ящиком.

Процесс, который заинтересован в получении сообщений, обращается к ядру с запросом о создании для него почтового ящика и сообщает адрес, по которому ему могут поступать сетевые пакеты, после чего все сообщения с данным адресом будут помещены в его почтовый ящик. Такой способ часто называют буферизуемым примитивом.

Надежные и ненадежные примитивы

Ранее подразумевалось, что когда отправитель посылает сообщение, адресат его обязательно получает. Но реально сообщения могут теряться. Предположим, что используются блокирующие примитивы. Когда отправитель посылает сообщение, то он приостанавливает свою работу до тех пор, пока сообщение не будет послано. Однако нет никаких гарантий, что после того, как он возобновит свою работу, сообщение будет доставлено адресату.

Для решения этой проблемы существует три подхода. Первый заключается в том, что система не берет на себя никаких обязательств по поводу доставки сообщений. Реализация надежного взаимодействия становится целиком заботой пользователя.

Второй подход заключается в том, что ядро принимающей машины посылает квитанцию-подтверждение ядру отправляющей машины на каждое сообщение. Посылающее ядро разблокирует пользовательский процесс только после получения этого подтверждения. Подтверждение передается от ядра к ядру. Ни отправитель, ни получатель его не видят.

Третий подход заключается в использовании ответа в качестве подтверждения в тех системах, в которых запрос всегда сопровождается ответом. Отправитель остается заблокированным до получения ответа. Если ответа нет слишком долго, то посылающее ядро может переслать запрос специальной службе предотвращения потери сообщений.

Концепция удаленного вызова процедур

Идея вызова удаленных процедур *(Remote Procedure Call - RPC)* состоит в расширении хорошо известного и понятного механизма передачи управления и данных внутри программы, выполняющейся на одной машине, на передачу управления и данных через сеть. Средства удаленного вызова процедур предназначены для облегчения организации распределенных вычислений. Наибольшая эффективность использования RPC достигается в тех приложениях, в которых существует интерактивная связь между удаленными компонентами с небольшим временем ответов и относительно малым количеством передаваемых данных. Такие приложения называются RPC-ориентированными.

Характерными чертами вызова локальных процедур являются:

- Асимметричность, то есть одна из взаимодействующих сторон является инициатором;
- Синхронность, то есть выполнение вызывающей процедуры при останавливается с момента выдачи запроса и возобновляется только после возврата из вызываемой процедуры.

Реализация удаленных вызовов существенно сложнее реализации вызовов локальных процедур. Начнем с того, что поскольку вызывающая и вызываемая процедуры выполняются на разных машинах, то они имеют разные адресные пространства, и это создает проблемы при передаче параметров и результатов, особенно если машины не идентичны. Так как RPC не может рассчитывать на разделяемую память, то это означает, что параметры RPC не должны содержать указателей на ячейки нестековой памяти и что значения параметров должны копироваться с одного компьютера на другой. Следующим отличием RPC от локального вызова является то, что он обязательно использует нижележащую систему связи, однако это не должно быть явно видно ни в определении процедур, ни в самих процедурах. Удаленность вносит дополнительные проблемы. Выполнение вызывающей программы и вызываемой локальной процедуры в одной машине реализуется в рамках

единого процесса. Но в реализации RPC участвуют как минимум два процесса - по одному в каждой машине. В случае, если один из них аварийно завершится, могут возникнуть следующие ситуации: при аварии вызывающей процедуры удаленно вызванные процедуры станут "осиротевшими", а при аварийном завершении удаленных процедур станут "обездоленными родителями" вызывающие процедуры, которые будут безрезультатно ожидать ответа от удаленных процедур.

Кроме того, существует ряд проблем, связанных с неоднородностью языков программирования и операционных сред: структуры данных и структуры вызова процедур, поддерживаемые в каком-либо одном языке программирования, не поддерживаются точно так же во всех других языках[21,22].

Эти и некоторые другие проблемы решает широко распространенная технология RPC, лежащая в основе многих распределенных операционных систем.

Базовые операции RPC

Чтобы понять работу RPC, рассмотрим вначале выполнение вызова локальной процедуры в обычной машине, работающей автономно. Пусть это, например, будет системный вызов

count=read (fd,buf,nbytes);

где fd - целое число, buf - массив символов, nbytes - целое число.

Чтобы осуществить вызов, вызывающая процедура заталкивает параметры в стек в обратном порядке (рис. 2.76). После того, как вызов read выполнен, он помещает возвращаемое значение в регистр, перемещает адрес возврата и возвращает управление вызывающей процедуре, которая выбирает параметры из стека, возвращая его в исходное состояние. Заметим, что в языке С параметры могут вызываться или по ссылке (by name), или по значению (by value). По отношению к вызываемой процедуре параметры-значения являются инициализируемыми локальными переменными. Вызываемая процедура может изменить их, и это не повлияет на значение оригиналов этих переменных в вызывающей процедуре.

Если в вызываемую процедуру передается указатель на переменную, то изменение значения этой переменной вызываемой процедурой влечет изменение значения этой переменной и для вызывающей процедуры. Этот факт весьма существенен для RPC.

Существует также другой механизм передачи параметров, который не используется в языке С. Он называется call-by-copy/restore и состоит в необходимости копирования вызывающей программой переменных в стек в виде значений, а затем копирования назад после выполнения вызова поверх оригинальных значений вызывающей процедуры.

Решение о том, какой механизм передачи параметров использовать, принимается разработчиками языка. Иногда это зависит от типа передаваемых данных. В языке С, например, целые и другие скалярные данные всегда передаются по значению, а массивы - по ссылке.

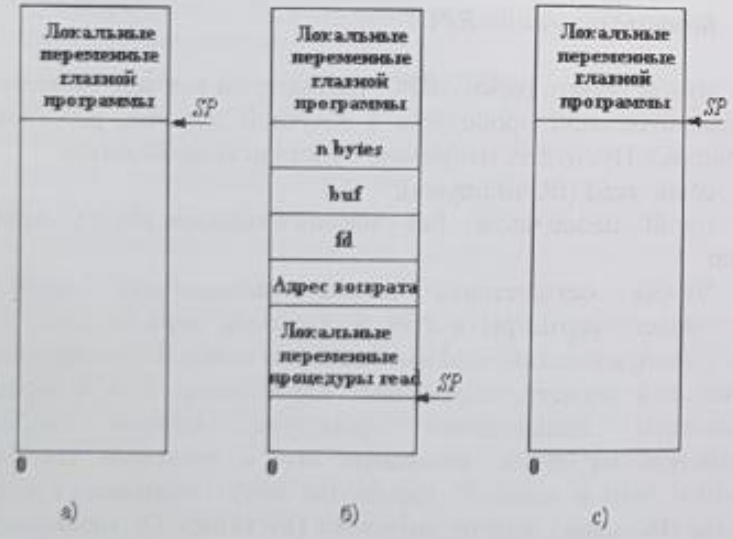


Рис. 2.76. а) Стек до выполнения вызова read; б) Стек во время выполнения процедуры; в) Стек после возврата в вызывающую программу

Идея, положенная в основу RPC, состоит в том, чтобы сделать вызов удаленной процедуры выглядящим по возможности также, как и вызов локальной процедуры. Другими словами - сделать RPC прозрачным: вызывающей процедуре не требуется знать, что вызываемая процедура находится на другой машине, и наоборот.

RPC достигает прозрачности следующим путем. Когда вызываемая процедура действительно является удаленной, в библиотеку помещается вместо локальной процедуры другая версия процедуры, называемая клиентским стабом (stub - заглушка). Подобно оригинальной процедуре, стаб вызывается с использованием вызывающей последовательности (как на рис. 2.76), так же происходит прерывание при обращении к ядру. Только в отличие от оригинальной процедуры он не помещает параметры в регистры и не запрашивает у ядра данные, вместо этого он формирует сообщение для отправки ядру удаленной машины.

Этапы выполнения RPC

Взаимодействие программных компонентов при выполнении удаленного вызова процедуры иллюстрируется рисунком 2.77. После того, как клиентский стаб был вызван программой-клиентом, его первой задачей является заполнение буфера отправляемым сообщением. В некоторых системах клиентский стаб имеет единственный буфер фиксированной длины, заполняемый каждый раз с самого начала при поступлении каждого нового запроса. В других системах буфер сообщения представляет собой пул буферов для отдельных полей сообщения, причем некоторые из этих буферов уже заполнены. Этот метод особенно подходит для тех случаев, когда пакет имеет формат, состоящий из большого числа полей, но значения многих из этих полей не меняются от вызова к вызову.

Затем параметры должны быть преобразованы в соответствующий формат и вставлены в буфер сообщения. К этому моменту сообщение готово к передаче, поэтому выполняется прерывание по вызову ядра.

Рис. 2.77. Remote Procedure Call

Когда ядро получает управление, оно переключает контексты, сохраняет регистры процессора и карту памяти (дескрипторы страниц), устанавливает новую карту памяти, которая будет использоваться для работы в режиме ядра. Поскольку контексты ядра и пользователя различаются, ядро должно точно скопировать сообщение в свое собственное адресное пространство, так, чтобы иметь к нему доступ, запомнить адрес назначения (а, возможно, и другие поля заголовка), а также оно должно передать его сетевому интерфейсу. На этом завершается работа на клиентской стороне. Включается таймер передачи, и ядро может либо выполнять циклический опрос наличия ответа, либо передать управление планировщику, который выберет какой-либо другой процесс на выполнение. В первом случае ускоряется выполнение запроса, но отсутствует мультипрограммирование.

На стороне сервера поступающие биты помещаются принимающей аппаратурой либо во встроенный буфер, либо в оперативную память. Когда вся информация будет получена, генерируется прерывание. Обработчик прерывания проверяет правильность данных пакета и определяет, какому стабу следует их передать. Если ни один из стабов не ожидает этот пакет, обработчик должен либо поместить его в буфер, либо вообще

отказаться от него. Если имеется ожидающий стаб, то сообщение копируется ему. Наконец, выполняется переключение контекстов, в результате чего восстанавливаются регистры и карта памяти, принимая те значения, которые они имели в момент, когда стаб сделал вызов receive[17,18].

Теперь начинает работу серверный стаб. Он распаковывает параметры и помещает их соответствующим образом в стек. Когда все готово, выполняется вызов сервера. После выполнения процедуры сервер передает результаты клиенту. Для этого выполняются все описанные выше этапы, только в обратном порядке.

Рисунок 2.78 показывает последовательность команд, которую необходимо выполнить для каждого RPC-вызова, а рис. 2.79 - какая доля общего времени выполнения RPC тратится на выполнение каждого их описанных 14 этапов. Исследования были проведены на мультипроцессорной рабочей станции DEC Firefly, и, хотя наличие пяти процессоров обязательно повлияло на результаты измерений, приведенная на рисунке гистограмма дает общее представление о процессе выполнения RPC.

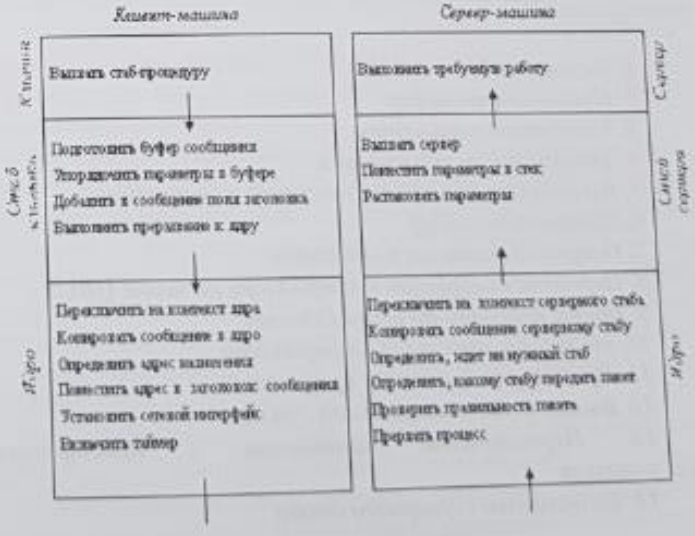


Рис. 2.78. Этапы выполнения процедуры RPC

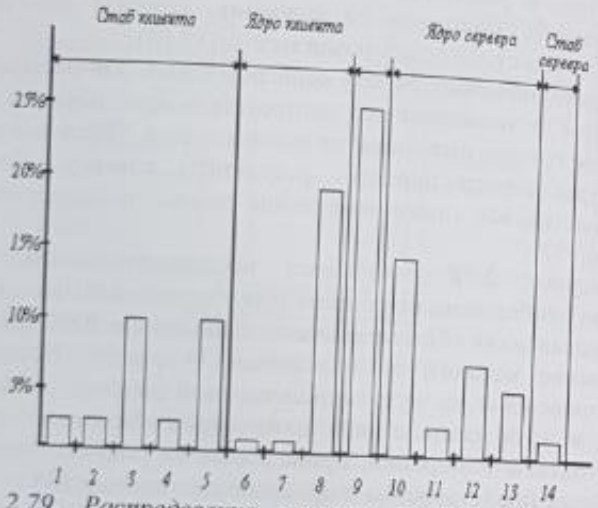


Рис. 2.79. Распределение времени между 14 этапами выполнения RPC

1. Вызов стаба
2. Подготовить буфер
3. Упаковать параметры
4. Заполнить поле заголовка
5. Вычислить контрольную сумму в сообщении
6. Прерывание к ядру
7. Очередь пакета на выполнение
8. Передача сообщения контроллеру по шине QBUS
9. Время передачи по сети Ethernet
10. Получить пакет от контроллера
11. Процедура обработки прерывания
12. Вычисление контрольной суммы
13. Переключение контекста в пространство пользователя
14. Выполнение серверного стаба

Динамическое связывание

Рассмотрим вопрос о том, как клиент задает месторасположение сервера. Одним из методов решения этой проблемы является непосредственное использование сетевого адреса сервера в клиентской программе. Недостаток такого подхода - его чрезвычайная негибкость: при перемещении сервера, или при увеличении числа серверов, или при изменении интерфейса во всех этих и многих других случаях необходимо перекомпилировать все программы, которые использовали жесткое задание адреса сервера. Для того, чтобы избежать всех этих проблем, в некоторых распределенных системах используется так называемое динамическое связывание.

Начальным моментом для динамического связывания является формальное определение (спецификация) сервера. Спецификация содержит имя файл-сервера, номер версии и список процедур-услуг, предоставляемых данным сервером для клиентов. Для каждой процедуры дается описание ее параметров с указанием того, является ли данный параметр входным или выходным относительно сервера. Некоторые параметры могут быть одновременно входными и выходными - например, некоторый массив, который посылается клиентом на сервер, модифицируется там, а затем возвращается обратно клиенту (операция copy/ restore).

Формальная спецификация сервера используется в качестве исходных данных для программы-генератора стабов, которая создает как клиентские, так и серверные стабы. Затем они помещаются в соответствующие библиотеки. Когда пользовательская (клиентская) программа вызывает любую процедуру, определенную в спецификации сервера, соответствующая стаб-процедура связывается с двоичным кодом программы. Аналогично, когда компилируется сервер, с ним связываются серверные стабы[12-14].

При запуске сервера самым первым его действием является передача своего серверного интерфейса специальной программе, называемой binder'ом. Этот процесс, известный как процесс регистрации сервера, включает передачу сервером своего имени,

номера версии, уникального идентификатора и описателя местонахождения сервера. Описатель системно независим и может представлять собой IP, Ethernet, X.500 или еще какой-либо адрес. Кроме того, он может содержать и другую информацию, например, относящуюся к аутентификации.

Когда клиент вызывает одну из удаленных процедур первый раз, например, read, клиентский стаб видит, что он еще не подсоединен к серверу, и посылает сообщение binder-программе с просьбой об импорте интерфейса нужной версии нужного сервера. Если такой сервер существует, то binder передает описатель и уникальный идентификатор клиентскому стабу.

Клиентский стаб при посылке сообщения с запросом использует в качестве адреса описатель. В сообщении содержатся параметры и уникальный идентификатор, который ядро сервера использует для того, чтобы направить поступившее сообщение в нужный сервер в случае, если их несколько на этой машине.

Этот метод, заключающийся в импорте/экспорте интерфейсов, обладает высокой гибкостью. Например, может быть несколько серверов, поддерживающих один и тот же интерфейс, и клиенты распределяются по серверам случайным образом. В рамках этого метода становится возможным периодический опрос серверов, анализ их работоспособности и, в случае отказа, автоматическое отключение, что повышает общую отказоустойчивость системы. Этот метод может также поддерживать аутентификацию клиента. Например, сервер может определить, что он может быть использован только клиентами из определенного списка.

Однако у динамического связывания имеются недостатки, например, дополнительные накладные расходы (временные затраты) на экспорт и импорт интерфейсов. Величина этих затрат может быть значительна, так как многие клиентские процессы существуют короткое время, а при каждом старте процесса процедура импорта интерфейса должна быть снова выполнена. Кроме того, в больших распределенных системах может стать узким местом программа binder, а создание нескольких программ аналогичного назначения также увеличивает накладные расходы на создание и синхронизацию процессов.

Семантика RPC в случае отказов

В идеале RPC должен функционировать правильно и в случае отказов. Рассмотрим следующие классы отказов:

1. Клиент не может определить местонахождения сервера, например, в случае отказа нужного сервера, или из-за того, что программа клиента была скомпилирована давно и использовала старую версию интерфейса сервера. В этом случае в ответ на запрос клиента поступает сообщение, содержащее код ошибки.

2. Потерян запрос от клиента к серверу. Самое простое решение - через определенное время повторить запрос.

3. Потеряно ответное сообщение от сервера клиенту. Этот вариант сложнее предыдущего, так как некоторые процедуры не являются идемпотентными. Идемпотентной называется процедура, запрос на выполнение которой можно повторить несколько раз, и результат при этом не изменится. Примером такой процедуры может служить чтение файла. Но вот процедура снятия некоторой суммы с банковского счета не является идемпотентной, и в случае потери ответа повторный запрос может существенно изменить состояние счета клиента. Одним из возможных решений является приведение всех процедур к идемпотентному виду. Однако на практике это не всегда удается, поэтому может быть использован другой метод - последовательная нумерация всех запросов клиентским ядром. Ядро сервера запоминает номер самого последнего запроса от каждого из клиентов, и при получении каждого запроса выполняет анализ - является ли этот запрос первичным или повторным.

4. Сервер потерпел аварию после получения запроса. Здесь также важно свойство идемпотентности, но к сожалению не может быть применен подход с нумерацией запросов. В данном случае имеет значение, когда произошел отказ - до или после выполнения операции. Но клиентское ядро не может распознать эти ситуации, для него известно только то, что время ответа истекло. Существует три подхода к этой проблеме:

• Ждать до тех пор, пока сервер не перезагрузится и пытаться выполнить операцию снова. Этот подход гарантирует,

что RPC был выполнен до конца по крайней мере один раз, а возможно и более.

• Сразу сообщить приложению об ошибке. Этот подход гарантирует, что RPC был выполнен не более одного раза.

• Третий подход не гарантирует ничего. Когда сервер отказывает, клиенту не оказывается никакой поддержки. RPC может быть или не выполнен вообще, или выполнен много раз. Во всяком случае этот способ очень легко реализовать.

Ни один из этих подходов не является очень привлекательным. А идеальный вариант, который бы гарантировал ровно одно выполнение RPC, в общем случае не может быть реализован по принципиальным соображениям. Пусть, например, удаленной операцией является печать некоторого текста, которая включает загрузку буфера принтера и установку одного бита в некотором управляющем регистре принтера, в результате которой принтер стартует. Авария сервера может произойти как за микросекунду до, так и за микросекунду после установки управляющего бита. Момент сбоя целиком определяет процедуру восстановления, но клиент о моменте сбоя узнать не может. Короче говоря, возможность аварии сервера радикально меняет природу RPC и ясно отражает разницу между централизованной и распределенной системой. В первом случае крах сервера ведет к краху клиента, и восстановление невозможно. Во втором случае действия по восстановлению системы выполнить и возможно, и необходимо[5,6].

1. Клиент потерпел аварию после отсылки запроса. В этом случае выполняются вычисления результатов, которых никто не ожидает. Такие вычисления называют "сиротами". Наличие сирот может вызвать различные проблемы: непроизводительные затраты процессорного времени, блокирование ресурсов, подмена ответа на текущий запрос ответом на запрос, который был выдан клиентской машиной еще до перезапуска системы.

Как поступать с сиротами? Рассмотрим 4 возможных решения.

• *Уничтожение.* До того, как клиентский стаб посылает RPC-сообщение, он делает отметку в журнале, оповещая о том, что он будет сейчас делать. Журнал хранится на диске или в другой памяти, устойчивой к сбоям. После аварии система

перезагружается, журнал анализируется и сироты ликвидируются. К недостаткам такого подхода относятся, во-первых, повышенные затраты, связанные с записью о каждом RPC на диск, а, во-вторых, возможная неэффективность из-за появления сирот второго поколения, порожденных RPC-вызовами, выданными сиротами первого поколения.

• *Перевоплощение.* В этом случае все проблемы решаются без использования записи на диск. Метод состоит в делении времени на последовательно пронумерованные периоды. Когда клиент перезагружается, он передает широковещательное сообщение всем машинам о начале нового периода. После приема этого сообщения все удаленные вычисления ликвидируются. Конечно, если сеть сегментированная, то некоторые сироты могут и уцелеть.

• *Мягкое перевоплощение* аналогично предыдущему случаю, за исключением того, что отыскиваются и уничтожаются не все удаленные вычисления, а только вычисления перезагружающегося клиента.

• *Истечение срока.* Каждому запросу отводится стандартный отрезок времени T, в течение которого он должен быть выполнен. Если запрос не выполняется за отведенное время, то выделяется дополнительный квант. Хотя это и требует дополнительной работы, но если после аварии клиента сервер ждет в течение интервала T до перезагрузки клиента, то все сироты обязательно уничтожаются.

На практике ни один из этих подходов не желателен, более того, уничтожение сирот может усугубить ситуацию. Например, пусть сирота заблокировал один или более файлов базы данных. Если сирота будет вдруг уничтожен, то эти блокировки останутся, кроме того уничтоженные сироты могут остаться стоять в различных системных очередях, в будущем они могут вызвать выполнение новых процессов и т.п.

2.2.2. Синхронизация в распределенных системах и алгоритмы синхронизации

К вопросам связи процессов, реализуемой путем передачи сообщений или вызовов RPC, тесно примыкают и вопросы

синхронизации процессов. Синхронизация необходима процессам для организации совместного использования ресурсов, таких как файлы или устройства, а также для обмена данными.

В однопроцессорных системах решение задач взаимного исключения, критических областей и других проблем синхронизации осуществлялось с использованием общих методов, таких как семафоры и мониторы. Однако эти методы не совсем подходят для распределенных систем, так как все они базируются на использовании разделяемой оперативной памяти. Например, два процесса, которые взаимодействуют, используя семафор, должны иметь доступ к нему. Если оба процесса выполняются на одной и той же машине, они могут иметь совместный доступ к семафору, хранящемуся, например, в ядре, делая системные вызовы. Однако, если процессы выполняются на разных машинах, то этот метод не применим, для распределенных систем нужны новые подходы.

Алгоритм синхронизации логических часов

В централизованной однопроцессорной системе, как правило, важно только относительное время и не важна точность часов. В распределенной системе, где каждый процессор имеет собственные часы со своей точностью хода, ситуация резко меняется: программы, использующие время (например, программы, подобные команде make в UNIX, которые используют время создания файлов, или программы, для которых важно время прибытия сообщений и т.п.) становятся зависимыми от того, часами какого компьютера они пользуются. В распределенных системах синхронизация физических часов (показывающих реальное время) является сложной проблемой, но с другой стороны очень часто в этом нет никакой необходимости: то есть процессам не нужно, чтобы во всех машинах было правильное время, для них важно, чтобы оно было везде одинаковое, более того, для некоторых процессов важен только правильный порядок событий. В этом случае мы имеем дело с логическими часами.

Введем для двух произвольных событий отношение "случилось до". Выражение a ® b читается "a случилось до b" и означает, что все процессы в системе считают, что сначала произошло событие a, а потом - событие b. Отношение

"случилось до" обладает свойством транзитивности: если выражения a ® b и b ® c истинны, то справедливо и выражение a ® c. Для двух событий одного и того же процесса всегда можно установить отношение "случилось до", аналогично может быть установлено это отношение и для событий передачи сообщения одним процессом и приемом его другим, так как прием не может произойти раньше отправки. Однако, если два произвольных события случились в разных процессах на разных машинах, и эти процессы не имеют между собой никакой связи (даже косвенной через третьи процессы), то нельзя сказать с полной определенностью, какое из событий произошло раньше, а какое позже.

Ставится задача создания такого механизма ведения времени, который бы для каждого события a мог указать значение времени T(a), с которым бы были согласны все процессы в системе. При этом должно выполняться условие: если a ® b , то T(a)<T(b). Кроме того, время может только увеличиваться и, следовательно, любые корректировки времени могут выполняться только путем добавления положительных значений, и никогда - путем вычитания.

Рассмотрим алгоритм решения этой задачи, который предложил Lamport. Для отметок времени в нем используются события. На рисунке 2.80 показаны три процесса, выполняющихся на разных машинах, каждая из которых имеет свои часы, идущие со своей скоростью. Как видно из рисунка, когда часы процесса 0 показали время 6, в процессе 1 часы показывали 8, а в процессе 2 - 10. Предполагается, что все эти часы идут с постоянной для себя скоростью.

В момент времени 6 процесс 0 посылает сообщение A процессу 1. Это сообщение приходит к процессу 1 в момент времени 16 по его часам. В логическом смысле это вполне возможно, так как 6<16. Аналогично, сообщение B, посланное процессом 1 процессу 2 пришло к последнему в момент времени 40, то есть его передача заняла 16 единиц времени, что также является правдоподобным.

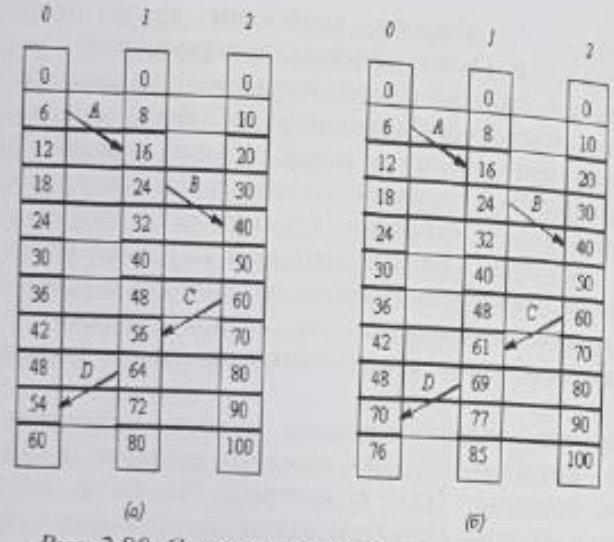


Рис. 2.80. Синхронизация логических часов
а - три процесса, каждый со своими собственными часами;
б - алгоритм синхронизации логических часов

Ну а далее начинаются весьма странные вещи. Сообщение C от процесса 2 к процессу 1 было отправлено в момент времени 64, а поступило в место назначения в момент времени 54. Очевидно, что это невозможно. Такие ситуации необходимо предотвращать. Решение Lamport'a вытекает непосредственно из отношений "случилось до". Так как C было отправлено в момент 60, то оно должно дойти в момент 61 или позже. Следовательно, каждое сообщение должно нести с собой время своего отправления по часам машины-отправителя. Если в машине, получившей сообщение, часы показывают время, которое меньше времени отправления, то эти часы переводятся вперед, так, чтобы они показали время, большее времени отправления сообщения. На рисунке 3.6,б видно, что C поступило в момент 61, а сообщение D - в 70.

Этот алгоритм удовлетворяет сформулированным выше требованиям.

Системы, состоящие из нескольких процессов, часто легче программировать, используя так называемые критические секции. Когда процессу нужно читать или модифицировать некоторые разделяемые структуры данных, он прежде всего входит в критическую секцию для того, чтобы обеспечить себе исключительное право использования этих данных, при этом он уверен, что никакой процесс не будет иметь доступа к этому ресурсу одновременно с ним. Это называется взаимным исключением. В однопроцессорных системах критические секции защищаются семафорами, мониторами и другими аналогичными конструкциями. Рассмотрим, какие алгоритмы могут быть использованы в распределенных системах.

Централизованный алгоритм

Наиболее очевидный и простой путь реализации взаимного исключения в распределенных системах - это применение тех же методов, которые используются в однопроцессорных системах. Один из процессов выбирается в качестве координатора (например, процесс, выполняющийся на машине, имеющей наибольшее значение сетевого адреса). Когда какой-либо процесс хочет войти в критическую секцию, он посылает сообщение с запросом к координатору, оповещая его о том, в какую критическую секцию он хочет войти, и ждет от координатора разрешение. Если в этот момент ни один из процессов не находится в критической секции, то координатор посылает ответ с разрешением. Если же некоторый процесс уже выполняет критическую секцию, связанную с данным ресурсом, то никакой ответ не посылается; запрашивавший процесс ставится в очередь, и после освобождения критической секции ему отправляется ответ-разрешение. Этот алгоритм гарантирует взаимное исключение, но вследствие своей централизованной природы обладает низкой отказоустойчивостью.

Распределенный алгоритм

Когда процесс хочет войти в критическую секцию, он формирует сообщение, содержащее имя нужной ему критической секции, номер процесса и текущее значение времени. Затем он посылает это сообщение всем другим процессам. Предполагается, что передача сообщения надежна, то есть получение каждого сообщения сопровождается подтверждением. Когда процесс получает сообщение такого рода, его действия зависят от того, в каком состоянии по отношению к указанной в сообщении критической секции он находится. Имеют место три ситуации:

1. Если получатель не находится и не собирается входить в критическую секцию в данный момент, то он отсылает назад процессу-отправителю сообщение с разрешением.

2. Если получатель уже находится в критической секции, то он не отправляет никакого ответа, а ставит запрос в очередь.

3. Если получатель хочет войти в критическую секцию, но еще не сделал этого, то он сравнивает временную отметку поступившего сообщения со значением времени, которое содержится в его собственном сообщении, разосланном всем другим процессам. Если время в поступившем к нему сообщении меньше, то есть его собственный запрос возник позже, то он посылает сообщение-разрешение, в обратном случае он не посылает ничего и ставит поступившее сообщение-запрос в очередь.

Процесс может войти в критическую секцию только в том случае, если он получил ответные сообщения-разрешения от всех остальных процессов. Когда процесс покидает критическую секцию, он посылает разрешение всем процессам из своей очереди и исключает их из очереди.

Алгоритм Token Ring

Совершенно другой подход к достижению взаимного исключения в распределенных системах иллюстрируется рисунком 2.81. Все процессы системы образуют логическое кольцо, т.е. каждый процесс знает номер своей позиции в кольце,

а также номер ближайшего к нему следующего процесса. Когда кольцо инициализируется, процессу 0 передается так называемый токен. Токен циркулирует по кольцу. Он переходит от процесса n к процессу n+1 путем передачи сообщения по типу "точка-точка". Когда процесс получает токен от своего соседа, он анализирует, не требуется ли ему самому войти в критическую секцию. Если да, то процесс входит в критическую секцию. После того, как процесс выйдет из критической секции, он передает токен дальше по кольцу. Если же процесс, принявший токен от своего соседа, не заинтересован во вхождении в критическую секцию, то он сразу отправляет токен в кольцо. Следовательно, если ни один из процессов не желает входить в критическую секцию, то в этом случае токен просто циркулирует по кольцу с высокой скоростью[7,8].

Сравним эти три алгоритма взаимного исключения. Централизованный алгоритм является наиболее простым и наиболее эффективным. При его использовании требуется только три сообщения для того, чтобы процесс вошел и покинул критическую секцию: запрос и сообщение-разрешение для входа и сообщение об освобождении ресурса при выходе. При использовании распределенного алгоритма для одного использования критической секции требуется послать (n-1) сообщений-запросов (где n - число процессов) - по одному на каждый процесс и получить (n-1) сообщений-разрешений, то есть всего необходимо 2(n-1) сообщений. В алгоритме Token Ring число сообщений переменно: от 1 в случае, если каждый процесс входил в критическую секцию, до бесконечно большого числа, при циркуляции токена по кольцу, в котором ни один процесс не входил в критическую секцию.

К сожалению все эти три алгоритма плохо защищены от отказов. В первом случае к краху приводит отказ координатора, во втором - отказ любого процесса (парадоксально, но распределенный алгоритм оказывается менее отказоустойчивым, чем централизованный), а в третьем - потеря токена или отказ процесса.

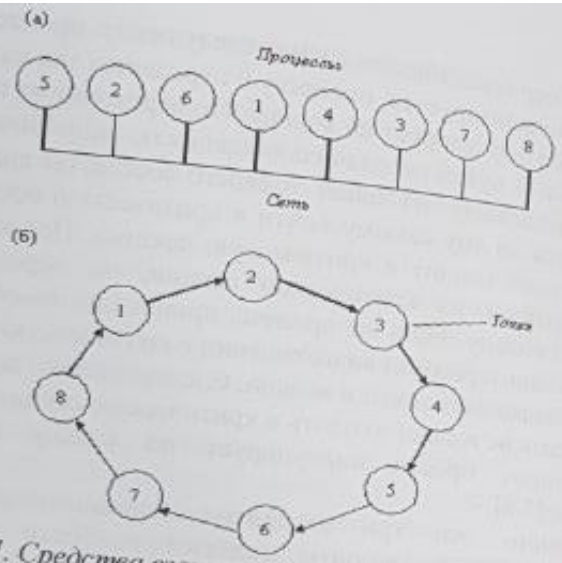


Рис. 2.81. Средства взаимного исключения в распределенных системах

а - неупорядоченная группа процессов в сети;

б - логическое кольцо, образованное программным обеспечением

Неделимые транзакции

Все средства синхронизации, которые были рассмотрены ранее, относятся к нижнему уровню, например, семафоры. Они требуют от программиста детального знания алгоритмов взаимного исключения, управления критическими секциями, умения предотвращать клинчи (взаимные блокировки), а также владения средствами восстановления после краха. Однако существуют средства синхронизации более высокого уровня, которые освобождают программиста от необходимости вникать во все эти подробности и позволяют ему сконцентрировать свое внимание на логике алгоритмов и организации параллельных вычислений. Таким средством является неделимая транзакция.

Модель неделимой транзакции пришла из бизнеса. Представьте себе переговорный процесс двух фирм о продаже-покупке некоторого товара. В процессе переговоров условия

договора могут многократно меняться, уточняться. Пока договор еще не подписан обеими сторонами, каждая из них может от него отказаться. Но после подписания контракта сделка (transaction) должна быть выполнена.

Компьютерная транзакция полностью аналогична. Один процесс объявляет, что он хочет начать транзакцию с одним или более процессами. Они могут некоторое время создавать и уничтожать разные объекты, выполнять какие-либо операции. Затем инициатор объявляет, что он хочет завершить транзакцию. Если все с ним соглашаются, то результат фиксируется. Если один или более процессов отказываются (или они потерпели крах еще до выработки согласия), тогда измененные объекты возвращается точно к тому состоянию, в котором они находились до начала выполнения транзакции. Такое свойство "все-или-ничего" облегчает работу программиста.

Для программирования с использованием транзакций требуется некоторый набор примитивов, которые должны быть предоставлены программисту либо операционной системой, либо языком программирования. Примеры примитивов такого рода:

BEGIN_TRANSACTION	команды, которые следуют за этим примитивом, формируют транзакцию.
END_TRANSACTION	завершает транзакцию и пытается зафиксировать ее.
ABORT_TRANSACTION	прерывает транзакцию, восстанавливает предыдущие значения.
READ	читает данные из файла (или другого объекта)
WRITE	пишет данные в файл (или другой объект).

Первые два примитива используются для определения границ транзакции. Операции между ними представляют собой тело транзакции. Либо все они должны быть выполнены, либо ни одна из них. Это может быть системный вызов, библиотечная

процедура или группа операторов языка программирования, заключенная в скобки.

Транзакции обладают следующими свойствами: упорядочиваемостью, неделимостью, постоянством.

Упорядочиваемость гарантирует, что если две или более транзакции выполняются в одно и то же время, то конечный результат выглядит так, как если бы все транзакции выполнялись последовательно в некотором (в зависимости от системы) порядке.

Неделимость означает, что когда транзакция находится в процессе выполнения, то никакой другой процесс не видит ее промежуточные результаты.

Постоянство означает, что после фиксации транзакции никакой сбой не может отменить результатов ее выполнения.

Если программное обеспечение гарантирует вышеперечисленные свойства, то это означает, что в системе поддерживается механизм транзакций.

Рассмотрим некоторые подходы к реализации механизма транзакций.

В соответствии с первым подходом, когда процесс начинает транзакцию, то он работает в индивидуальном рабочем пространстве, содержащем все файлы и другие объекты, к которым он имеет доступ. Пока транзакция не зафиксируется или не прервется, все изменения данных происходят в этом рабочем пространстве, а не в "реальном", под которым мы понимаем обычную файловую систему. Главная проблема этого подхода состоит в больших накладных расходах по копированию большого объема данных в индивидуальное рабочее пространство, хотя и имеются несколько приемов уменьшения этих расходов.

Второй общий подход к реализации механизма транзакций называется списком намерений. Этот метод заключается в том, что модифицируются сами файлы, а не их копии, но перед изменением любого блока производится запись в специальный файл - журнал регистрации, где отмечается, какая транзакция делает изменения, какой файл и блок изменяется и каковы старое и новое значения изменяемого блока. Только после успешной записи в журнал регистрации делаются изменения в исходном

файле. Если транзакция фиксируется, то и об этом делается запись в журнал регистрации, но старые значения измененных данных сохраняются. Если транзакция прерывается, то информация журнала регистрации используется для приведения файла в исходное состояние, и это действие называется откатом.

В распределенных системах фиксация транзакций может потребовать взаимодействия нескольких процессов на разных машинах, каждая из которых хранит некоторые переменные, файлы, базы данных. Для достижения свойства неделимости транзакций в распределенных системах используется специальный протокол, называемый протоколом двухфазной фиксации транзакций. Хотя он и не является единственным протоколом такого рода, но он наиболее широко используется.

Суть этого протокола состоит в следующем. Один из процессов выполняет функции координатора (рис.2.82). Координатор начинает транзакцию, делая запись об этом в своем журнале регистрации, затем он посылает всем подчиненным процессам, также выполняющим эту транзакцию, сообщение "подготовиться к фиксации". Когда подчиненные процессы получают это сообщение, то они проверяют, готовы ли они к фиксации, делают запись в своем журнале и посылают координатору сообщение-ответ "готов к фиксации". После этого подчиненные процессы остаются в состоянии готовности и ждут от координатора команду фиксации. Если хотя бы один из подчиненных процессов не откликнулся, то координатор откатывает подчиненные транзакции, включая и те, которые подготовились к фиксации.

Выполнение второй фазы заключается в том, что координатор посылает команду "фиксировать" (commit) всем подчиненным процессам. Выполняя эту команду, последние фиксируют изменения и завершают подчиненные транзакции. В результате гарантируется одновременное синхронное завершение (удачное или неудачное) распределенной транзакции.

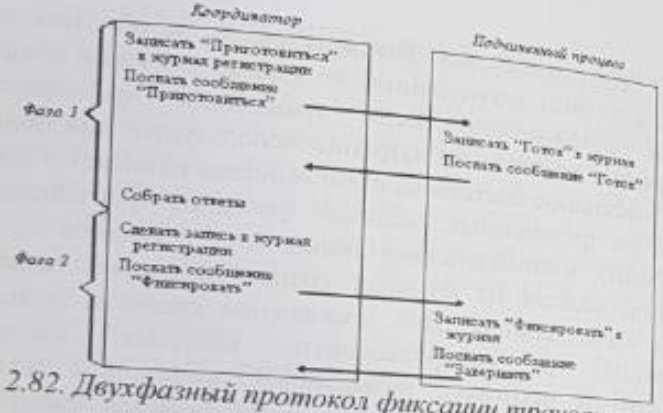


Рис. 2.82. Двухфазный протокол фиксации транзакции

2.2.3. Распределенная файловая система

Ключевым компонентом любой распределенной системы является файловая система. Как и в централизованных системах, в распределенной системе функцией файловой системы является хранение программ и данных и предоставление доступа к ним по мере необходимости. Файловая система поддерживается одной или более машинами, называемыми файл-серверами. Файл-серверы перехватывают запросы на чтение или запись файлов, поступающие от других машин (не серверов). Эти другие машины называются клиентами. Каждый посланный запрос проверяется и выполняется, а ответ отсылается обратно. Файл-серверы обычно содержат иерархические файловые системы, каждая из которых имеет корневой каталог и каталоги более низких уровней. Рабочая станция может подсоединять и монтировать эти файловые системы к своим локальным файловым системам. При этом монтируемые файловые системы остаются на серверах.

Важно понимать различие между файловым сервисом и файловым сервером. Файловый сервис - это описание функций, которые файловая система предлагает своим пользователям. Это описание включает имеющиеся примитивы, их параметры и функции, которые они выполняют. С точки зрения пользователей файловый сервис определяет то, с чем пользователи могут работать, но ничего не говорит о том, как все это реализовано. В

сущности, файловый сервис определяет интерфейс файловой системы с клиентами.

Файловый сервер - это процесс, который выполняется на отдельной машине и помогает реализовывать файловый сервис. В системе может быть один файловый сервер или несколько, но в хорошо организованной распределенной системе пользователи не знают, как реализована файловая система. В частности, они не знают количество файловых серверов, их месторасположение и функции. Они только знают, что если процедура определена в файловом сервисе, то требуемая работа каким-то образом выполняется, и им возвращаются требуемые результаты. Более того, пользователи даже не должны знать, что файловый сервис является распределенным. В идеале он должен выглядеть также, как и в централизованной файловой системе.

Так как обычно файловый сервер - это просто пользовательский процесс (или иногда процесс ядра), выполняющийся на некоторой машине, в системе может быть несколько файловых серверов, каждый из которых предлагает различный файловый сервис. Например, в распределенной системе может быть два сервера, которые обеспечивают файловые сервисы систем UNIX и MS-DOS соответственно, и любой пользовательский процесс пользуется подходящим сервисом.

Файловый сервис в распределенных файловых системах (впрочем как и в централизованных) имеет две функционально различные части: собственно файловый сервис и сервис каталогов. Первый имеет дело с операциями над отдельными файлами, такими, как чтение, запись или добавление, а второй - с созданием каталогов и управлением ими, добавлением и удалением файлов из каталогов и т.п.

Интерфейс файлового сервиса

Для любого файлового сервиса, независимо от того, централизован он или распределен, самым главным является вопрос, что такое файл? Во многих системах, таких как UNIX и MS DOS, файл - это неинтерпретируемая последовательность байтов. Значение и структура информации в файле является

заботой прикладных программ, операционную систему это не интересует[1-3].

В ОС мейнфреймов поддерживаются разные типы логической организации файлов, каждый с различными свойствами. Файл может быть организован как последовательность записей, и у операционной системы имеются вызовы, которые позволяют работать на уровне этих записей. Большинство современных распределенных файловых систем поддерживают определение файла как последовательности байтов, а не последовательности записей. Файл характеризуется атрибутами: именем, размером, датой создания, идентификатором владельца, адресом и другими.

Важным аспектом файловой модели является возможность модификации файла после его создания. Обычно файлы могут модифицироваться, но в некоторых распределенных системах единственными операциями с файлами являются СОЗДАТЬ и ПРОЧИТАТЬ. Такие файлы называются неизменяемыми. Для неизменяемых файлов намного легче осуществить кэширование файла и его репликацию (тиражирование), так как исключается все проблемы, связанные с обновлением всех копий файла при его изменении.

Файловый сервис может быть разделен на два типа в зависимости от того, поддерживает ли он модель загрузки-выгрузки или модель удаленного доступа. В модели загрузки-выгрузки пользователю предлагаются средства чтения или записи файла целиком. Эта модель предполагает следующую схему обработки файла: чтение файла с сервера на машину клиента, обработка файла на машине клиента и запись обновленного файла на сервер. Преимуществом этой модели является ее концептуальная простота. Кроме того, передача файла целиком очень эффективна. Главным недостатком этой модели являются высокие требования к дискам клиентов. Кроме того, неэффективно перемещать весь файл, если нужна его маленькая часть.

Другой тип файлового сервиса соответствует модели удаленного доступа, которая предполагает поддержку большого количества операций над файлами: открытие и закрытие файлов, чтение и запись частей файла, позиционирование в файле,

проверка и изменение атрибутов файла и так далее. В то время как в модели загрузки-выгрузки файловый сервер обеспечивал только хранение и перемещение файлов, в данном случае вся файловая система выполняется на серверах, а не на клиентских машинах. Преимуществом такого подхода являются низкие требования к дисковому пространству на клиентских машинах, а также исключение необходимости передачи целого файла, когда нужна только его часть.

Интерфейс сервиса каталогов

Природа сервиса каталогов не зависит от типа используемой модели файлового сервиса. В распределенных системах используются те же принципы организации каталогов, что и в централизованных, в том числе многоуровневая организация каталогов.

Принципиальной проблемой, связанной со способами именования файлов, является обеспечение прозрачности. В данном контексте прозрачность понимается в двух слабо различимых смыслах. Первый - прозрачность расположения - означает, что имена не дают возможности определить месторасположение файла. Например, имя */server1/dir1/dir2/x* говорит, что файл x расположен на сервере 1, но не указывает, где расположен этот сервер. Сервер может перемещаться по сети, а полное имя файла при этом не меняется. Следовательно, такая система обладает прозрачностью расположения.

Предположим, что файл x очень большой, а на сервере 1 мало места, предположим далее, что на сервере 2 места много. Система может захотеть переместить автоматически файл x на сервер 2. К сожалению, когда первый компонент всех имен - это имя сервера, система не может переместить файл на другой сервер автоматически, даже если каталоги *dir1* и *dir2* находятся на обоих серверах. Программы, имеющие встроенные строки имен файлов, не будут правильно работать в этом случае. Система, в которой файлы могут перемещаться без изменения имен, обладает свойством независимости от расположения. Распределенная система, которая включает имена серверов или

машин непосредственно в имена файлов, не является независимой от расположения. Система, базирующаяся на удаленном монтировании, также не обладает этим свойством, так как в ней невозможно переместить файл из одной группы файлов в другую и продолжать после этого пользоваться старыми именами. Независимости от расположения трудно достичь, но это желаемое свойство распределенной системы.

Большинство распределенных систем используют какую-либо форму двухуровневого именования: на одном уровне файлы имеют символические имена, такие как prog.c, предназначенные для использования людьми, а на другом - внутренние, двоичные имена, для использования самой системой. Каталоги обеспечивают отображение между двумя этими уровнями имен. Отличием распределенных систем от централизованных является возможность соответствия одному символьному имени нескольких двоичных имен. Обычно это используется для представления оригинального файла и его архивных копий. Имея несколько двоичных имен, можно при недоступности одной из копий файла получить доступ к другой. Этот метод обеспечивает отказоустойчивость за счет избыточности.

Семантика разделения файлов

Когда два или более пользователей разделяют один файл, необходимо точно определить семантику чтения и записи, чтобы избежать проблем. В централизованных системах, разрешающих разделение файлов, таких как UNIX, обычно определяется, что, когда операция ЧТЕНИЕ следует за операцией ЗАПИСЬ, то читается только что обновленный файл. Аналогично, когда операция чтения следует за двумя операциями записи, то читается файл, измененный последней операцией записи. Тем самым система придерживается абсолютного временного упорядочивания всех операций, и всегда возвращает самое последнее значение. Будем называть эту модель семантикой UNIX'а. В централизованной системе (и даже на мультипроцессоре с разделяемой памятью) ее легко и понять, и реализовать.

Семантика UNIX может быть обеспечена и в распределенных системах, но только, если в ней имеется лишь один файловый сервер, и клиенты не кэшируют файлы. Для этого все операции чтения и записи направляются на файловый сервер, который обрабатывает их строго последовательно. На практике, однако, производительность распределенной системы, в которой все запросы к файлам идут на один сервер, часто становится неудовлетворительной. Эта проблема иногда решается путем разрешения клиентам обрабатывать локальные копии часто используемых файлов в своих личных кэшах. Если клиент сделает локальную копию файла в своем локальном кэше и начнет ее модифицировать, а вскоре после этого другой клиент прочитает этот файл с сервера, то он получит неверную копию файла. Одним из способов устранения этого недостатка является немедленный возврат всех изменений в кэшированном файле на сервер. Такой подход хотя и концептуально прост, но не эффективен.

Другим решением является введение так называемой сессионной семантики, в соответствии с которой изменения в открытом файле сначала виды только процессу, который модифицирует файл, и только после закрытия файла эти изменения могут видеть другие процессы. При использовании сессионной семантики возникает проблема одновременного использования одного и того же файла двумя или более клиентами. Одним из решений этой проблемы является принятие правила, в соответствии с которым окончательным является тот вариант, который был закрыт последним. Менее эффективным, но гораздо более простым в реализации, является вариант, при котором окончательным результирующим файлом на сервере может оказаться любой из этих файлов.

Следующий подход к разделению файлов заключается в том, чтобы сделать все файлы неизменяемыми. Тогда файл нельзя открыть для записи, а можно выполнять только операции СОЗДАТЬ и ЧИТАТЬ. Тогда для изменения файла остается только возможность создать полностью новый файл и поместить его в каталог под именем старого файла. Следовательно, хотя файл и нельзя модифицировать, его можно заменить (автоматически) новым файлом. Другими словами, хотя файлы и

нельзя обновлять, но каталоги обновлять можно. Таким образом, проблема, связанная с одновременным использованием файла, просто исчезнет.

Четвертый способ работы с разделяемыми файлами в распределенных системах - это использование механизма неделимых транзакций.

Итак, было рассмотрено четыре различных подхода к работе с разделяемыми файлами в распределенных системах.

1. **Семантика UNIX**. Каждая операция над файлом немедленно становится видимой для всех процессов.
2. **Сессионная семантика**. Изменения не видны до тех пор, пока файл не закрывается.
3. **Неизменяемые файлы**. Модификации невозможны, разделение файлов и репликация упрощаются.
4. **Транзакции**. Все изменения делаются по принципу "все или ничего".

Вопросы разработки структуры файловой системы

Рассмотрим прежде всего вопрос о распределении серверной и клиентской частей между машинами. В некоторых системах (например, NFS) нет разницы между клиентом и сервером, на всех машинах работает одно и то же базовое программное обеспечение, так что любая машина, которая хочет предложить файловый сервис, свободно может это сделать. Для этого ей достаточно экспортировать имена выбранных каталогов, чтобы другие машины могли иметь к ним доступ.

В других системах файловый сервер - это только пользовательская программа, так что система может быть сконфигурирована как клиент, как сервер или как клиент и сервер одновременно. Третьим, крайним случаем, является система, в которой клиенты и серверы - это принципиально различные машины, как в терминах аппаратуры, так и в терминах программного обеспечения. Серверы могут даже работать под управлением другой операционной системы.

Вторым важным вопросом реализации файловой системы является структуризация сервиса файлов и каталогов. Один подход заключается в комбинировании этих двух сервисов на одном сервере. При другом подходе эти сервисы разделяются. В

последнем случае при открытии файла требуется обращение к серверу каталогов, который отображает символьное имя в двоичное, а затем обращение к файловому серверу с двоичным именем для действительного чтения или записи файла[13,15].

Аргументом в пользу разделения сервисов является тот факт, что они на самом деле слабо связаны, поэтому их раздельная реализация более гибкая. Например, можно реализовать сервер каталогов MS-DOS и сервер каталогов UNIX, которые будут использовать один и тот же файловый сервер для физического хранения файлов. Разделение этих функций также упрощает программное обеспечение. Недостатком является то, что использование двух серверов увеличивает интенсивность сетевого обмена.

Постоянный поиск имен, особенно при использовании нескольких серверов каталогов, может приводить к большим накладным расходам. В некоторых системах делается попытка улучшить производительность за счет кэширования имен. При открытии файла кэш проверяется на наличие в нем нужного имени. Если оно там есть, то этап поиска, выполняемый сервером имени, пропускается, и двоичный адрес извлекается из кэша.

Последний рассматриваемый здесь структурный вопрос связан с хранением на серверах информации о состоянии клиентов. Существует две конкурирующие точки зрения.

Первая состоит в том, что сервер не должен хранить такую информацию (сервер stateless). Другими словами, когда клиент посылает запрос на сервер, сервер его выполняет, отсылает ответ, а затем удаляет из своих внутренних таблиц всю информацию о запросе. Между запросами на сервере не хранится никакой текущей информации о состоянии клиента. Другая точка зрения состоит в том, что сервер должен хранить такую информацию (сервер statefull).

Рассмотрим эту проблему на примере файлового сервера, имеющего команды ОТКРЫТЬ, ПРОЧИТАТЬ, ЗАПИСАТЬ и ЗАКРЫТЬ файл. Открывая файлы, statefull-сервер должен запоминать, какие файлы открыл каждый пользователь. Обычно при открытии файла пользователю дается дескриптор файла или другое число, которое используется при последующих вызовах для его идентификации. При поступлении вызова, сервер

использует дескриптор файла для определения, какой файл нужен. Таблица, отображающая дескрипторы файлов на сами файлы, является информацией о состоянии клиентов.

Для сервера stateless каждый запрос должен содержать исчерпывающую информацию (полное имя файла, смещение в файле и т.п.), необходимую серверу для выполнения требуемой операции. Очевидно, что эта информация увеличивает длину сообщения.

Однако при отказе statefull-сервера теряются все его таблицы, и после перезагрузки неизвестно, какие файлы открыл каждый пользователь. Последовательные попытки провести операции чтения или записи с открытыми файлами будут безуспешными. Stateless-серверы в этом плане являются более отказоустойчивыми, и это аргумент в их пользу.

Преимущества обоих подходов можно обобщить следующим образом:

Stateless-серверы:

- отказоустойчивы;
- не нужны вызовы OPEN/CLOSE;
- меньше памяти сервера расходуется на таблицы;
- нет ограничений на число открытых файлов;
- отказ клиента не создает проблем для сервера.

Statefull-серверы:

- более короткие сообщения при запросах;
- лучше производительность;
- возможно опережающее чтение;
- легче достичь идемпотентности;
- возможна блокировка файлов.

Кэширование

В системах, состоящих из клиентов и серверов, потенциально имеется четыре различных места для хранения файлов и их частей: диск сервера, память сервера, диск клиента (если имеется) и память клиента. Наиболее подходящим местом для хранения всех файлов является диск сервера. Он обычно имеет большую емкость, и файлы становятся доступными всем клиентам. Кроме того, поскольку в этом случае существует

только одна копия каждого файла, то не возникает проблемы согласования состояний копий[19,20].

Проблемой при использовании диска сервера является производительность. Перед тем, как клиент сможет прочитать файл, файл должен быть переписан с диска сервера в его оперативную память, а затем передан по сети в память клиента. Обе передачи занимают время.

Значительное увеличение производительности может быть достигнуто за счет кэширования файлов в памяти сервера. Требуются алгоритмы для определения, какие файлы или их части следует хранить в кэш-памяти.

При выборе алгоритма должны решаться две задачи. Во-первых, какими единицами оперирует кэш. Этими единицами могут быть или дисковые блоки, или целые файлы. Если это целые файлы, то они могут храниться на диске непрерывными областями (по крайней мере в виде больших участков), при этом уменьшается число обменов между памятью и диском а, следовательно, обеспечивается высокая производительность. Кэширование блоков диска позволяет более эффективно использовать память кэша и дисковое пространство.

Во-вторых, необходимо определить правило замены данных при заполнении кэш-памяти. Здесь можно использовать любой стандартный алгоритм кэширования, например, алгоритм LRU (least recently used), соответствии с которым вытесняется блок, к которому дольше всего не было обращения.

Кэш-память на сервере легко реализуется и совершенно прозрачна для клиента. Так как сервер может синхронизировать работу памяти и диска, с точки зрения клиентов существует только одна копия каждого файла, так что проблема согласования не возникает.

Хотя кэширование на сервере исключает обмен с диском при каждом доступе, все еще остается обмен по сети. Существует только один путь избавиться от обмена по сети - это кэширование на стороне клиента, которое, однако, порождает много сложностей.

Так как в большинстве систем используется кэширование в памяти клиента, а не на его диске, то мы рассмотрим только этот случай. При проектировании такого варианта имеется три

возможности размещения кэша (рис.2.83). Самый простой состоит в кэшировании файлов непосредственно внутри адресного пространства каждого пользовательского процесса. Обычно кэш управляется с помощью библиотеки системных вызов. По мере того, как файлы открываются, закрываются, читаются и пишутся, библиотека просто сохраняет наиболее часто используемые файлы. Когда процесс завершается, все модифицированные файлы записываются назад на сервер. Хотя эта схема реализуется с чрезвычайно низкими издержками, она эффективна только тогда, когда отдельные процессы часто повторно открывают и закрывают файлы. Таким является процесс менеджера базы данных, но обычные программы чаще всего читают каждый файл однократно, так что кэширование с помощью библиотеки в этом случае не дает выигрыша.

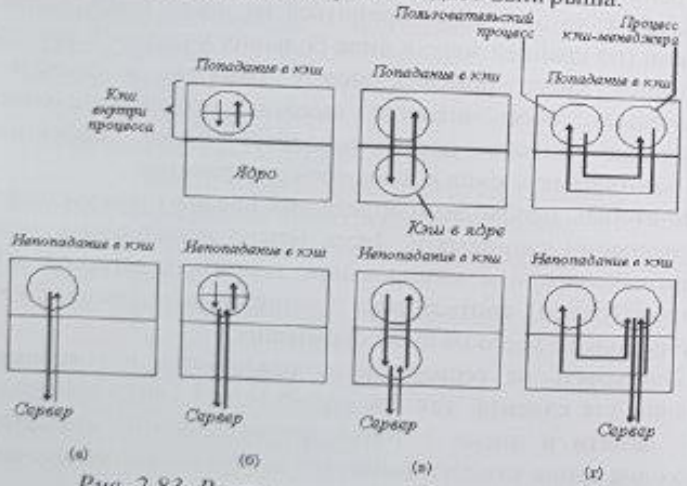


Рис. 2.83. Различные способы выполнения кэша в клиентской памяти
а - без кэширования;
б - кэширование внутри каждого процесса;
в - кэширование в ядре;
г - кэш-менеджер как пользовательский процесс

Другим местом кэширования является ядро. Недостатком этого варианта является то, что во всех случаях требуется

выполнять системные вызовы, даже в случае успешного обращения к кэш-памяти (файл оказался в кэше). Но преимуществом является то, что файлы остаются в кэше и после завершения процессов. Например, предположим, что двухпроходный компилятор выполняется, как два процесса. Первый проход записывает промежуточный файл, который читается вторым проходом. На рисунке 2.83,в показано, что после завершения процесса первого прохода промежуточный файл, вероятно, будет находиться в кэше, так что вызов сервера не потребуется.

Третьим вариантом организации кэша является создание отдельного процесса пользовательского уровня - кэш-менеджера. Преимущество этого подхода заключается в том, что ядро освобождается от кода файловой системы и тем самым реализуются все достоинства микроядер.

С другой стороны, когда ядро управляет кэшем, оно может динамически решить, сколько памяти выделить для программ, а сколько для кэша. Когда же кэш-менеджер пользовательского уровня работает на машине с виртуальной памятью, то понятно, что ядро может решить выгрузить некоторые, или даже все страницы кэша на диск, так что для так называемого "попадания в кэш" требуется подкачка одной или более страниц. Нечего и говорить, что это полностью дискредитирует идею кэширования. Однако, если в системе имеется возможность фиксировать некоторые страницы в памяти, то такая парадоксальная ситуация может быть исключена.

Как и везде, нельзя получить что-либо, не заплатив чем-то за это. Кэширование на стороне клиента вносит в систему проблему несогласованности данных.

Одним из путей решения проблемы согласования является использование алгоритма сквозной записи. Когда кэшируемый элемент (файл или блок) модифицируется, новое значение записывается в кэш и одновременно посылается на сервер. Теперь другой процесс, читающий этот файл, получает самую последнюю версию.

Один из недостатков алгоритма сквозной записи состоит в том, что он уменьшает интенсивность сетевого обмена только при чтении, при записи интенсивность сетевого обмена та же

самая, что и без кэширования. Многие разработчики систем находят это неприемлемым и предлагают следующий алгоритм, использующий отложенную запись: вместо того, чтобы выполнять запись на сервер, клиент просто помечает, что файл изменен. Примерно каждые 30 секунд все изменения в файлах собираются вместе и отсылаются на сервер за один прием. Одна большая запись обычно более эффективна, чем много маленьких[13,17].

Следующим шагом в этом направлении является принятие сессионной семантики, в соответствии с которой запись файла на сервер производится только после его закрытия. Этот алгоритм называется "запись-по-закрытию". Как мы видели раньше, этот путь приводит к тому, что если две копии одного файла кэшируются на разных машинах и последовательно записываются на сервер, то второй записывается поверх первого. Однако это не так уж плохо, как кажется на первый взгляд. В однопроцессорной системе два процесса могут открыть и читать файл, модифицировать его в своих адресных пространствах, а затем записать его назад. Следовательно, алгоритм "запись-по-закрытию", основанный на сессионной семантике, не намного хуже варианта, уже используемого в однопроцессорной системе.

Совершенно отличный подход к проблеме согласования - это использование алгоритма централизованного управления (этот подход соответствует семантике UNIX). Когда файл открыт, машина, открывшая его, посылает сообщение файловому серверу, чтобы оповестить его об этом факте. Файл-сервер сохраняет информацию о том, кто открыл какой файл, и о том, открыт ли он для чтения, для записи, или для того и другого. Если файл открыт для чтения, то нет никаких препятствий для разрешения другим процессам открыть его для чтения, но открытие его для записи должно быть запрещено. Аналогично, если некоторый процесс открыл файл для записи, то все другие виды доступа должны быть предотвращены. При закрытии файла также необходимо оповестить файл-сервер для того, чтобы он обновил свои таблицы, содержащие данные об открытых файлах. Модифицированный файл также может быть выгружен на сервер в такой момент.

Четыре алгоритма управления кэшированием обобщаются следующим образом:

1. *Сквозная запись.* Этот метод эффективен частично, так как уменьшает интенсивность только операций чтения, а интенсивность операций записи остается неизменной.

2. *Отложенная запись.* Производительность лучше, но результат чтения кэшированного файла не всегда однозначен.

3. *"Запись-по-закрытию".* Удовлетворяет сессионной семантике.

4. *Централизованное управление.* Ненадежен вследствие своей централизованной природы.

Подводя итоги обсуждения проблемы кэширования, нужно отметить, что кэширование на сервере несложно реализуется и почти всегда дает эффект, независимо от того, реализовано кэширование у клиента или нет. Кэширование на сервере не влияет на семантику файловой системы, видимую клиентом. Кэширование у клиента напротив дает увеличение производительности, но увеличивает и сложность семантики.

Репликация

Распределенные системы часто обеспечивают репликацию (тиражирование) файлов в качестве одной из услуг, предоставляемых клиентам. Репликация - это асинхронный перенос изменений данных исходной файловой системы в файловые системы, принадлежащие различным узлам распределенной файловой системы. Другими словами, система оперирует несколькими копиями файлов, причем каждая копия находится на отдельном файловом сервере. Имеется несколько причин для предоставления этого сервиса, главными из которых являются:

1. Увеличение надежности за счет наличия независимых копий каждого файла на разных файл-серверах.

2. Распределение нагрузки между несколькими серверами.

Как обычно, ключевым вопросом, связанным с репликацией является прозрачность. До какой степени пользователи должны быть в курсе того, что некоторые файлы реплицируются? Должны ли они играть какую-либо роль в процессе репликации

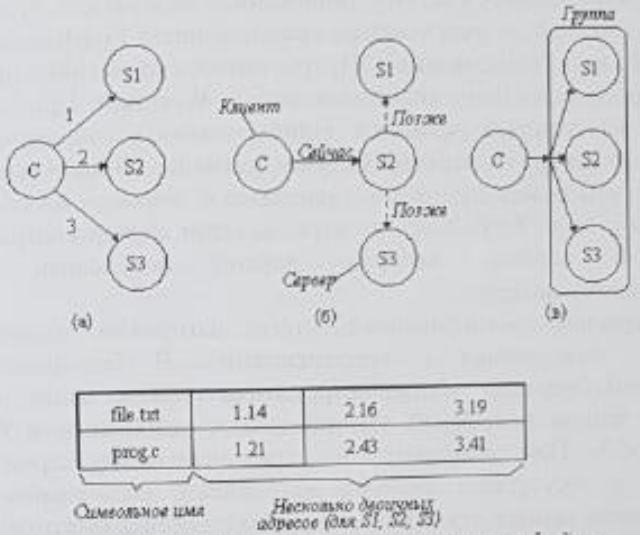
или репликация должна выполняться полностью автоматически? В одних системах пользователи полностью вовлечены в этот процесс, в других система все делает без их ведома. В последнем случае говорят, что система репликационно прозрачна.

На рисунке 2.84 показаны три возможных способа репликации. При использовании первого способа (а) программист сам управляет всем процессом репликации. Когда процесс создает файл, он делает это на одном определенном сервере. Затем, если пожелает, он может сделать дополнительные копии на других серверах. Если сервер каталогов разрешает сделать несколько копий файла, то сетевые адреса всех копий могут быть ассоциированы с именем файла, как показано на рисунке снизу, и когда имя найдено, это означает, что найдены все копии. Чтобы сделать концепцию репликации более понятной, рассмотрим, как может быть реализована репликация в системах, основанных на удаленном монтировании, типа UNIX. Предположим, что рабочий каталог программиста имеет имя */machine1/usr/ast*. После создания файла, например, */machine1/usr/ast/xyz*, программист, процесс или библиотека могут использовать команду копирования для того, чтобы сделать копии */machine2/usr/ast/xyz* и *machine3/usr/ast/xyz*. Возможно программа использует в качестве аргумента строку */usr/ast/xyz* и последовательно попытается открывать копии, пока не достигнет успеха. Эта схема хотя и работает, но имеет много недостатков, и по этим причинам ее не стоит использовать в распределенных системах.

На рисунке 2.84,б показан альтернативный подход - ленивая репликация. Здесь создается только одна копия каждого файла на некотором сервере. Позже сервер сам автоматически выполнит репликации на другие серверы без участия программиста. Эта система должна быть достаточно быстрой для того, чтобы обновлять все эти копии, если потребуется.

Последним рассмотрим метод, использующий групповые связи (рис. 2.80,в). В этом методе все системные вызовы ЗАПИСАТЬ передаются одновременно на все серверы, таким образом копии создаются одновременно с созданием оригинала. Имеется два принципиальных различия в использовании групповых связей и ленивой репликации. Во-первых, при

ленивой репликации адресуется один сервер, а не группа. Во-вторых, ленивая репликация происходит в фоновом режиме, когда сервер имеет промежуток свободного времени, а при групповой репликации все копии создаются в одно и то же время.



file.txt	1.14	2.16	3.19
prog.c	1.21	2.43	3.41

Символьное имя — Несколько двоичных адресов (для S1, S2, S3)

Рис. 2.84. а) Точная репликация файла; б) Ленивая репликация файла; в) Репликация файла, использующая группу

Рассмотрим, как могут быть изменены существующие реплицированные файлы. Существует два хорошо известных алгоритма решения этой проблемы.

Первый алгоритм, называемый "репликация первой копии", требует, чтобы один сервер был выделен как первичный. Остальные серверы являются вторичными. Когда реплицированный файл модифицируется, изменение посылается на первичный сервер, который выполняет изменения локально, а затем посылает изменения на вторичные серверы.

Чтобы предотвратить ситуацию, когда из-за отказа первичный сервер не успевает оповестить об изменениях все вторичные серверы, изменения должны быть сохранены в

постоянном запоминающем устройстве еще до изменения первичной копии. В этом случае после перезагрузки сервера есть возможность сделать проверку, не проводились ли какие-нибудь обновления в момент краха. Недостаток этого алгоритма типичен для централизованных систем - пониженная надежность. Чтобы избежать его, используется метод, предложенный Гиффордом и известный как "голосование". Пусть имеется n копий, тогда изменения должны быть внесены в любые W копий. При этом серверы, на которых хранятся копии, должны отслеживать порядковые номера их версий. В случае, когда какой-либо сервер выполняет операцию чтения, он обращается с запросом к любым R серверам. Если R+W > n, то, хотя бы один сервер содержит последнюю версию, которую можно определить по максимальному номеру.

Интересной модификацией этого алгоритма является алгоритм "голосования с приведениями". В большинстве приложений операции чтения встречаются гораздо чаще, чем операции записи, поэтому R обычно делают небольшим, а W - близким к N. При этом выход из строя нескольких серверов приводит к отсутствию кворума для записи. Голосование с приведениями решает эту проблему путем создания фиктивного сервера без дисков для каждого отказавшего или отключенного сервера. Фиктивный сервер не участвует в кворуме чтения (прежде всего, у него нет файлов), но он может присоединиться к кворуму записи, причем он просто записывает в никуда передаваемый ему файл. Запись только тогда успешна, когда хотя бы один сервер настоящий.

Когда отказавший сервер перезапускается, то он должен получить кворум чтения для обнаружения последней версии, которую он копирует к себе перед тем, как начать обычные операции. В остальном этот алгоритм подобен основному.

Контрольные вопросы

1. Часто единственным достоинством виртуальной памяти называют возможность обеспечить для процесса объем виртуального адресного пространства, превышающий объем реальной памяти. Назовите другие достоинства виртуальной памяти.

2. В чем достоинства и недостатки преобразования виртуальных адресов в реальные во время выполнения программы? Какая часть работы по этому преобразованию выполняется аппаратным обеспечением, а какая - ОС?

3. Иногда считают, что виртуальная память может быть обеспечена только в системах с аппаратной поддержкой динамической трансляции адреса. Докажите, что это не так.

4. Почему при поиске свободной памяти стратегия "самый подходящий" оказывается хуже, чем "первый подходящий".

5. Сравните сегментную и страничную модели виртуальной памяти. Какая из них представляется Вам лучшей и почему?

6. Дополните приведенные в разделе 3.5. соображения по поводу выбора размера страницы.

7. Смоделируйте ситуацию применения дисциплины вытеснения FCFS, в которой увеличение числа реальных страниц приведет к увеличению числа страничных отказов.

8. Что такое кластерная подкачка страниц? Почему в современных ОС она становится все более популярной?

9. Каким образом ОС может определять, к каким страницам будут обращения в ближайшее время?

10. Большой размер виртуальной памяти процесса может приводить к тому, что даже таблица страниц не будет помещаться в реальной памяти. Какими путями решается эта проблема в современных ОС?

11. Каким образом снижение стоимости памяти влияет на дисциплины управления памятью?

12. Какие принципиальные изменения в концепции памяти может повлечь за собой увеличение разрядности адреса?

ГЛАВА 3. СРАВНИТЕЛЬНЫЙ ОБЗОР ОС

3.1. СОВРЕМЕННЫЕ КОНЦЕПЦИИ И ТЕХНОЛОГИИ ПРОЕКТИРОВАНИЯ ОС

3.1.1. Семейства ОС

Когда говорят об операционной системе, имеют в виду прежде всего семейство операционных систем. Формально операционные системы в пределах семейства отличаются номером версии. Что касается устройства, функций и возможностей операционных систем в пределах семейства, то радикальных отличий между операционными системами в пределах семейства нет (основные черты одни и те же).

Например, любая операционная система семейства *Windows* (*Windows* 3.1, *Windows* 95, *Windows* 2000,*XP* и т.д.) управляется по принципу "укажи и щелкни", характеризуется графическим интерфейсом и т.д.

Однако, когда в пределах семейства невозможно обойтись без радикальных изменений, из данного семейства вырастает новое (яркий пример: семейство *Windows* выросло из *MS DOS*, когда работа в текстовом диалоговом режиме изжила себя).

Итак, различные версии операционных систем в пределах семейства различаются функциональными возможностями, но не методологией построения.

Список операционных систем (основные ОС)

Это список известных операционных систем. Операционные системы могут быть классифицированы по базовой технологии *(UNIX*-подобные, пост-*UNIX*/потомки *UNIX*), типу лицензии (проприетарная или открытая), развивается ли в настоящее время (устаревшие или современные), по назначению (универсальные, ОС встроенных систем, ОС *PDA*, ОС реального времени, для рабочих станций или для серверов), а также по множеству других признаков.

Список операционных систем:

Apple:
A/UX
Apple Darwin
Apple DOS

GS/OS
Mac OS
Mac OS 8
Mac OS 9
Mac OS X
Cheetah
Puma
Jaguar
Panther
Tiger
Leopard
Snow Leopard
Lion
IOS
ProDOS
SOS
DEC/Compaq/HP:
AIS
OS-8
ITS (для *PDP-6* и *PDP-10*)
TOPS-10 (для *PDP-10*)
TOPS-20 (для *PDP-10*)
WAITS
TENEX (от *BBN*)
RSTS/E (работала на нескольких типах машин, в основном *PDP-11*)
RSX-11 (многопользовательская многозадачная ОС для *PDP-11*)
RT-11 (однопользовательская для *PDP-11*)
RTE-II (система реального времени для *HP-2000/2100* и *ДОС РВ* для *M-6000/7000*, *СМ-1*)
VMS (от *DEC* для серии компьютеров *VAX*, позднее переименована в *OpenVMS*)
HP-UX от *HP*
NonStop OS – разработана компанией *Tandem Computers*, впоследствии приобретённой фирмой *Compaq*
OSF/1 (от *DEC*; дважды переименована, сначала в *Digital UNIX*, затем в *Tru*64 *UNIX*)

ГЛАВА 3. СРАВНИТЕЛЬНЫЙ ОБЗОР ОС
3.1. СОВРЕМЕННЫЕ КОНЦЕПЦИИ И ТЕХНОЛОГИИ ПРОЕКТИРОВАНИЯ ОС

3.1.1. Семейства ОС

Когда говорят об операционной системе, имеют в виду прежде всего семейство операционных систем. Формально операционные системы в пределах семейства отличаются номером версии. Что касается устройства, функций и возможностей операционных систем в пределах семейства, то радикальных отличий между операционными системами в пределах семейства нет (основные черты одни и те же).

Например, любая операционная система семейства *Windows* (*Windows* 3.1, *Windows* 95, *Windows* 2000,*XP* и т.д.) управляется по принципу "укажи и щелкни", характеризуется графическим интерфейсом и т.д.

Однако, когда в пределах семейства невозможно обойтись без радикальных изменений, из данного семейства вырастает новое (яркий пример: семейство *Windows* выросло из *MS DOS*, когда работа в текстовом диалоговом режиме изжила себя).

Итак, различные версии операционных систем в пределах семейства различаются функциональными возможностями, но не методологией построения.

Список операционных систем (основные ОС)

Это список известных операционных систем. Операционные системы могут быть классифицированы по базовой технологии *(UNIX*-подобные, пост-*UNIX*/потомки *UNIX*), типу лицензии (проприетарная или открытая), развивается ли в настоящее время (устаревшие или современные), по назначению (универсальные, ОС встроенных систем, ОС *PDA*, ОС реального времени, для рабочих станций или для серверов), а также по множеству других признаков.

Список операционных систем:

Apple:
A/UX
Apple Darwin
Apple DOS

GS/OS
Mac OS
Mac OS 8
Mac OS 9
Mac OS X
Cheetah
Puma
Jaguar
Panther
Tiger
Leopard
Snow Leopard
Lion
IOS
ProDOS
SOS
DEC/Compaq/HP:
AIS
OS-8
ITS (для *PDP-6* и *PDP-10*)
TOPS-10 (для *PDP-10*)
TOPS-20 (для *PDP-10*)
WAITS
TENEX (от *BBN*)
RSTS/E (работала на нескольких типах машин, в основном *PDP-11*)
RSX-11 (многопользовательская многозадачная ОС для *PDP-11*)
RT-11 (однопользовательская для *PDP-11*)
RTE-II (система реального времени для *HP-2000/2100* и *ДОС РВ* для *M-6000/7000*, *СМ-1*)
VMS (от *DEC* для серии компьютеров *VAX*, позднее переименована в *OpenVMS*)
HP-UX от *HP*
NonStop OS – разработана компанией *Tandem Computers*, впоследствии приобретённой фирмой *Compaq*
OSF/1 (от *DEC*; дважды переименована, сначала в *Digital UNIX*, затем в *Tru*64 *UNIX*)

IBM:
IBSYS
OS/2
OS/2 v1.0 – Выпущена в декабре 1987 года. Одна из первых операционных систем с поддержкой многозадачного режима процессора 80286.
OS/2 v1.10SE – Выпущена в октябре 1988 года. *SE* = *Standard Edition*.
OS/2 v1.10EE – 1989 год.
OS/2 v1.20 – 1989 год. Редакции *SE* и *EE*. Улучшенный Presentation Manager.
OS/2 v1.30 – 1991 год. Также редакции *SE* и *EE*.
OS/2 v2.00 – Весна 1992 года. Первая версия *OS/2*, которой для работы необходим процессор 80386 с его защищённым режимом.
OS/2 v2.10 – Май 1993 года.
OS/2 v2.11 – Конец 1993 года. Не содержит подсистемы *Win-OS/2* и устанавливается поверх *Windows 3.1*. Стоит дешевле других версий *OS/2*.
OS/2 *v3.0 «Warp»* и *«Warp Connect»* – Октябрь 1994 года.
OS/2 v4.0 «Merlin» – Сентябрь 1996 года.
OS/2 *Warp 4.5* Server for E-business «Aurora» – Апрель 1999 года. Дальнейшие обновления получили имена *CP1* и *CP2 (Convenience Package)* и базировались на *Aurora*.
eComStation
AIX – *Unix*-подобная ОС
AIX/RT
AIX/6000
AIX PC
AIX/ESA
AIX/370
AIX 5L
DYNIX – *Unix*-подобная ОС, разработана компанией *Sequent Computer Systems*, которая позже была поглощена IBM
OS/400
VM

DOS/360
DOS/VSE
OS/360 – первая ОС для архитектуры *System 360*
MFT – позднее переименована в *OS/VS1*
MVT – позднее переименована в *OS/VS2*
SVS
MVS – разновидность *MVT*
TPF
ALCS
OS/390
z/OS – следующая версия после *IBM OS/390*
z/VM – разновидность *VM*
z/VSE – разновидность *VSE*
Basic Operating System – первая ОС для архитектуры *System 360*
PC DOS – *OEM*-версия *MS-DOS*, впоследствии дорабатывалась самостоятельно.
ОС ЕС, СВМ, МВС, ДОС ЕС, МОС ЕС – *IBM*-совместимые операционные системы (клоны) советского производства
Microsoft:
MSX-DOS
MS-DOS
Xenix – лицензированная версия *Unix*; продана *SCO* в 1990-х
Microsoft Windows
Windows 1.0
Windows 2.0 (для 80286)
Windows 3.0 – первая версия, имевшая коммерческий успех
Windows 3.1 – выпущена 18 марта 1992 года
Windows for Workgroups 3.11
Windows 9x – версии *Windows 4.x*, новое семейство, сохранявшее преемственность с *Windows 3.x*
Windows 95 (*версия Windows 4.00.950*)
Windows 98 (*версия Windows 4.10.1998*)
Windows Me (*версия Windows 4.90.3000*)
Windows NT – ОС, разрабатываемая в Майкрософт с 1988 года командой во главе с Дэвидом Катлером под рабочим названием *OS/2 Version3*.

Windows NT 3.1 – первая версия Windows *NT*, выпущена 27 июля 1993

Windows NT 3.5 (варианты поставки: *Workstation* – для рабочих станций и *Server* – для серверов)

Windows NT 3.51 – отлаженная версия *Windows NT 3.5*

Windows NT 4.0 (варианты поставки: *Workstation* – для рабочих станций и *Server* – для серверов)

Windows 2000 (версия *Windows NT 5.0*, варианты поставки: *Professional* - для рабочих станций, *Server*, *Advanced Server* и *Datacenter Server* - для серверов)

Windows XP (версия *Windows NT 5.1* – внутренне основана на ядре *Windows 2000*); варианты поставки: *Home*, *Professional*, *Tablet PC Edition*, *Media Center Edition*, *Embedded*

Windows Server 2003 (версия *Windows NT 5.2*) – вариант *Windows XP* для работы на серверах

Windows Compute Cluster Server 2003 – вариант *Windows XP* для работы в кластерных системах

Windows XP Embedded – вариант *Windows XP* для встраиваемых систем

Windows Vista (версия *Windows NT 6.0*)

Windows Server 2008 (версия *Windows NT 6.0*) – вариант *Windows Vista* для работы на серверах

Windows HPC Server 2008 – замена *Windows Compute Cluster Server 2003* для кластерных систем

Windows Home Server

Windows Vista for Embedded Systems – вариант *Windows Vista* для встраиваемых систем

Windows 7 (версия *Windows NT 6.1*)

Windows Server 2008 R2 (версия *Windows NT 6.1*) – вариант *Windows 7* для работы на серверах

Windows CE (compact edition – компактная редакция) – Операционная система реального времени для встраиваемых систем, мобильных телефонов, наладонных компьютеров и даже роботов.

Windows Mobile, Pocket PC – версии *Windows CE* для мобильных телефонов и наладонных компьютеров.

Windows Embedded – версии *Windows CE* для встраиваемых систем, включая роботов.

Windows 8

Другие *Unix*-подобные и*POSIX*-совместимые:

Aegis/OS (Apollo Computer)

CLIX от Intergraph

Cromix (Unix-emulating OS from Cromemco)

Coherent (Эмулирующая *Unix* ОС от *Mark Williams Co.* для персональных компьютеров)

DNIX

DYNIX – *Unix*-подобная ОС, разработана компанией *Sequent Computer Systems*, которая позже была поглощена *IBM*

Idris

IRIX от *SGI*

NeXTStep – свое развитие получила в ОС *Mac OS X* после объединения компаний *NeXT* и *Apple*

OPENSTEP

OS-9 – *Unix*-подобная *RTOS*, эмулирующая *Unix* от *Microware* для процессора *Motorola 6809*

OS-9/68k (Эмулирующая *Unix* от *Microware* для процессора *Motorola 680x0*; создана из *OS-9*)

OS-9000 (портативная эмуляция *Unix* от *Microware*; одна из реализаций предназначена для *Intel x86*)

QNX (POSIX, микроядерная операционная система; используется, в основном, во встроенных системах реального времени)

Rhapsody

RiscOS

SCO UNIX (от *Santa Cruz Operation*, куплена компанией *Caldera*, позже переименованной в *SCO*)

System V (реализация *AT&T Unix*, *'SVr4'* 4й релиз). Фактически последний «чистый» *UNIX*. Всё остальное обычно называют *UNIX* подобным…

UNIflex (Эмулирующая *Unix* ОС от *TSC* для *DMA*-совместимых, *Motorola 6809* с расширенной адресацией; например, *SWTPC*, *GIMIX*, …)

Ultrix (первая версия *Unix* для *VAX* и *PDP-11* от *DEC*, основана на *BSD*)

Unicos (*Unix* для суперкомпьютеров *Cray Research Inc.*)

Venix

Смартфоны:
Linux
Google Android;
Palm webOS в *Palm Pre;*
Maemo в *Nokia 770 Internet Tablet, Nokia N800, N810* и *Nokia N900;*
OpenMoko в устройствах *Neo 1973* и *Neo FreeRunner;*
MontaVista Mobilinux в *Motorola A760, E680;*
EZX Linux в *Motorola A1200, A1600, E6;*
MOTOMAGX в *Motorola ZINE ZN5;*
LiMo Platform;
Access Linux Platform в *Edelweiss;*
MeeGo;
bada – (ошибочно считается *OS Linux*) *OS* разработанная компанией *Samsung[6]*
Symbian OS
Apple iOS – ОС для *Apple iPhone, iPod touch, iPad*
BlackBerry OS
JavaFX Mobile
Windows Mobile на базе *Windows CE*
Windows Phone 7
Нетбуки, смартбуки, *MID:*
Linux
Slackware;
Xandros Linux;
Xubuntu;
Ubuntu;
Slax;
Puppy Linux;
Eeebuntu;
Linpus Linux Lite в *Acer Aspire One;*
Google Android;
Ubuntu Netbook Remix;
Moblin for Netbooks;
Google Chrome OS;
Jolicloud.
EPOC32 Release 5 в *Psion netBook* 1999 года;

Microsoft Windows CE в *Psion Teklogix netBook Pro* 2003 года, в *Elonex Smartbook* и др.
Microsoft Windows CE MID в *Toshiba JournE;*

Программное обеспечение ЭВМ подразделяют на прикладное, инструментальное и системное. Прикладные программы ориентированы на решение конечных пользовательских задач. Инструментальные программные системы предназначены для разработки программных средств, их наиболее типичными представителями являются среды языков программирования. Системные программы обеспечивают оптимальные условия для функционирования аппаратно-программной системы.

Системные программы могут быть как органической частью операционной среды (наиболее типичные представители – драйверы), так и самостоятельной надстройкой. Однако в любом случае системные программы должны быть спроектированы в соответствии с требованиями, предъявляемыми операционной средой.

Операционная среда состоит из подсистем – модулей и компонент – драйверов. Драйверы, органически встраиваемые в операционную систему, расширяют ее возможности.

Классификация операционных систем

Существует несколько схем классификации операционных систем. Ниже приведена классификация по некоторым признакам с точки зрения пользователя (табл. 3.1).

Таблица 3.1. Классификация ОС

№ п/п	Признак классификации	Разделения
1.	По числу одновременно выполняемых задач	1. Однозадачные 2. Многозадачные
2.	По числу одновременно работающих пользователей	1. Однопользовательские 2. Многопользовательские

3.	По числу одновременно управляемых процессоров	1. Однопроцессорные 2. Многопроцессорные
4.	По режиму работы	1. Пакетной обработки 2. Разделения времени 3. Реального времени 4. Многорежимные

Многозадачная ОС, решая проблемы распределения ресурсов и конкуренции, полностью реализует мультипрограммный режим в соответствии с определенными требованиями.

Приблизительность классификации по числу одновременно выполняемых задач очевидна. Так, в ОС *MS-DOS* можно организовать запуск дочерней задачи и одновременное сосуществование в памяти двух и более задач. Однако эта ОС традиционно считается однозадачной, главным образом из-за отсутствия защитных механизмов и коммуникационных возможностей.

Что касается классификации по числу одновременно работающих пользователей, то следует отметить: наиболее существенно отличие заключается в наличии у многопользовательских систем механизмов защиты персональных данных каждого пользователя.

Многопроцессорные системы состоят из двух или более центральных процессоров, 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 в ряде других[17,18].

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. В асимметричных ОС процессоры неравноправны. Обычно существует главный процессор (*master*)

и подчиненные (*slave*), загрузку и характер работы которых определяет главный процессор.

Рассмотрим подробнее классификацию ОС по режиму работы. Существует три категории ОС, которые характеризуются определенным типом взаимодействия между пользователем и его заданием: ОС пакетной обработки, в которых задание пользователя обрабатывается как последовательность пакетов, а возможность взаимодействия между пользователем и его заданием во время выполнения отсутствует; ОС разделения времени, которые обеспечивают одновременное обслуживание многих пользователей, позволяя каждому взаимодействовать со своими заданиями; ОС реального времени, которые обслуживают внешние процессы в темпе, соизмеримом с темпом их поступления (в настоящее время широкое распространение получили многорежимные ОС).

В разряд многозадачных ОС, наряду с пакетными системами и системами разделения времени, включаются также системы **реального времени**. 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 те 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.

Приведенная классификация ОС не является исчерпывающей.

3.2. ЭВОЛЮЦИЯ ПОЛЬЗОВАТЕЛЬСКИХ ИНТЕРФЕЙСОВ ОС

3.2.1. Задачи пользовательского интерфейса ОС

Разновидность интерфейсов можно разложить на характеристики по структуре связей, способа подключения и передачи данных, принципов управления и синхронизации.

Рассмотрим самые распространённые виды интерфейсов.

Внутримашинный интерфейс - система связи и средств сопряжения узлов и блоков ЭВМ между собой. Внутримашинный интерфейс представляет собой совокупность электрических линий связи (проводов), схем сопряжения с компонентами компьютера, протоколов (алгоритмов) передачи и преобразования сигналов.

Внешний интерфейс - система связи системного блока с периферийными устройствами ЭВМ или с другими ЭВМ.

Для человека, как пользователю, ближе всего пользовательский интерфейс (ПИ) или его ещё называют интерфейс «человек-машина», «человек-компьютер». Данный интерфейс позволяет нам общаться с компьютером и получать в ответ информацию в понятном для обычного пользователя виде, легко воспринимающейся информации в виде текста, аудио и видео информации.

Машинная часть интерфейса - часть интерфейса, реализованная в машине (аппаратно-программной ее части) с использованием возможностей вычислительной техники.

Человеческая часть интерфейса - это часть интерфейса, реализуемая человеком с учетом его возможностей, слабостей, привычек, способности к обучению и других факторов.

Самый первый пользовательский интерфейс был весьма примитивен, он позволял ввести пользователю некоторые данные, которые по заданному алгоритму обрабатывались, и система выдавала ответ (по принципу калькулятора). В данном случае пользователь должен был обладать определёнными знаниями и принципом работы программного продукта, с которым он работал.

Одной из задач, современного пользовательского интерфейса, заключается в том, чтобы общение пользователя с компьютером не требовало особых знаний, могла работать с любым уровнем подготовленности и обученности человека, позволила работать пользователю любыми удобными средствами (голосовое управление, выдача команд движение рук, глаз и т.п.).

В связи с этим появились специальные пользовательские интерфейсы, изменились концепции построения пользовательских интерфейсов и предложено несколько методик их создания.

Пользовательский интерфейс представляет собой совокупность программных и аппаратных средств, обеспечивающих взаимодействие пользователя с компьютером. Основу такого взаимодействия составляют диалоги. То есть процесс общения пользователя с компьютером — это обмен данными (сообщениями). При этом данные или сообщения бывают входящими и выходящими:

- входные сообщения, генерируемые человеком при помощи любых периферийных устройств (джойстик, клавиатура, мышь и т.п.);
- выходные сообщения, генерируются компьютером в виде картинок, текстов, аудио, видео, графических изображениях в ответ на запрос пользователя.

В качестве примеров входных сообщений можно привести пользовательские запросы при поиске (ввод параметров поиска), расчётов (ввод расчётных параметров и функций), запросы к базе данных и т.п.

Для выходных сообщений, можно привести примеры выдачи подсказок, результатов запросов, индикацию и сигнализацию БИОС, и т.п.

Классификация пользовательских интерфейсов и их элементов

Согласно принципам программирования, программный интерфейс можно разделить на группы.

Процедурные интерфейсы – это пользовательский интерфейс, способный вводить определённые данные через набор

действий, каждое из которых подразумевает определённый набор данных, функций. Процедурные ПИ можно разделить на три подкатегории:

Консольный интерфейс – этот ПИ организует общение пользователя с машиной через запросы «вопрос-ответ». Такой интерфейс реализует как правило конкретный сценарий. Программа сама определяет последовательность действий.

Интерфейс-меню. Общение происходит через специальный список действий, который предлагает программа пользователю на выбор. Данный интерфейс может быть простым – одноуровневым, когда запрос небольшой и простой (открыть, копировать, закрыть и т.п.), и иерархическим, когда варианты операций сложны и многоуровневые (работа с файлами, с базами данных и т.п.). При таком интерфейсе уже пользователь определяет последовательность действий.

Многоуровневое меню состоит из уровней, в каждом из которых одноуровневое меню.

Изначально программа находится в режиме ожидания команды пользователя из представленного меню. После выбора действия, пользователь его вводит в программу и теперь сам находится в режиме ожидания, пока программа обработает его запрос и выдаст ответ.

Меню для пользователя может выглядеть по-разному. Самое простое меню позволяет сделать выбор при помощи ввода порядкового номера действия из меню с клавиатуры. Этот метод прост, но требует определённой внимательности, чтобы не допустить ошибки. Меню посложнее, позволяют передвигаться по меню при помощи клавиатуры или мыши, при этом выделенный элемент будет выделяться цветом. Данное меню более информативней, приятней на вид и проще в пользовании.

Интерфейс со свободной навигацией. Данный интерфейс позволяет пользователю выбирать любые операции, находящиеся в зоне доступа, через с интерфейсные компоненты. Благодаря стандартизации, интерфейсные компоненты широко применяются в различных программах, в различных операционных системах. Принцип управления данными компонентами хорошо известен и понятен любому пользователю,

что является безусловным превосходством перед другими видами интерфейсов.

Наиболее распространёнными компонентами интерфейса со свободной навигацией в системе ОС Windows являются:

- поле ввода (edit box),
- кнопка (button),
- опция, флажок (checkbox),
- переключатель (radio button),
- индикатор хода выполнения задачи (progress bar),
- ползунок (slider),
- наборный счетчик (spin control, up/down control),
- панель инструментов (toolbar),
- списки: линейный (list box), выпадающий (combo box), древовидный (tree control);
- меню (menu).

Одним из достоинств интерфейса со свободной навигацией, является активация блокировки элементов в процессе диалога. Элементы, работа которых на текущий момент не корректна или недоступна по каким-либо причинам, будут не активны и никакое действие через эти элементы невозможно будет назначить. Данная особенность помогает пользователю ориентироваться при диалоге с программой.

Интерфейсы такого типа можно реализовать, применив визуальные среды программирования или на процедурно-ориентированном языке (СИ). Для этого необходимо событийное программирование и объектно-ориентированные библиотеки.

По уровню взаимодействия ПИ, его можно разделить на однонаправленные и двунаправленные.

- однонаправленные ПИ –обусловлен тем, что здесь происходит только ввод данных или действий от пользователя в программу;

- двунаправленные ПИ –в отличии от однонаправленного, он ещё осуществляет вывод данных или действий в ответ на запрос, управляющие воздействия пользователя.

Здесь под управляющими воздействиями подразумеваем ввод пользователем данных, параметров, которые повлияют на настройку ПИ и его параметров.

WIMP-интерфейс – вид графического интерфейса со своими специфическими особенностями:

- работа с программами происходит в окнах;
- появилось понятие «иконка», при нажатии на иконку происходит открытие файла;
- действия с объектами происходят через меню и является основным элементом;
- главным средством управления становится манипулятор.

WIMP-интерфейс – в виду своей сложности, повысил требования к программным средствам и обеспечению (высокая производительность, большая память и т.д.).

SILK- интерфейс (speech, image, language, knowledge (речь, образ, язык, знания)). Самый приближённый интерфейс на текущий момент к пользователю. Его можно назвать – естественно языковым интерфейсом. Общение происходит при помощи обычных слов, как при общении с человеком. Программа по определённым признакам её анализирует и переводит во внутремашинный код, обрабатывает полученную информацию и выводит пользователю на языке понятный пользователю.

Данный интерфейс очень сложен в программном смысле и требователен в аппаратном смысле. Но развитие информационных технологий позволяет всё больше и больше использовать данный интерфейс (голосовой поиск Google).

SILK- интерфейс для общения человека с машиной использует:

- речевую технологию;
- биометрическую технологию (мимический интерфейс);
- семантический (общественный) интерфейс.

Речевая технология получила своё развитие в 90-х годах. Она требовала чёткого произношения определённых слов, на которые программа отвечала соответствующими действиями. На текущий момент времени, данная технология шагнула далеко вперёд, уже нет жёстких требований при общении с программой.

Биометрическая технология появилась в конце 90-х годов. Этому способствовал процесс развития компьютерных технологий. Данная технология определяет биометрические данные человека и вводит их в систему обработки. К ним можно отнести работу со зрачком (определение взгляда, направления,

определение структуры радужной оболочки), работа с лицом человека (распознавание личности), отпечатки пальцев, спектр излучения человека.

Семантический (общественный) интерфейс возник еще в конце 70-х годов XX века, с развитием искусственного интеллекта. Этот интерфейс объединяет в себе сразу несколько групп интерфейсов: командной строки, и графический, речевой, мимический интерфейсы. Его главная особенность от остальных – отсутствие команд. Запрос формируется на естественном языке, в виде связанного текста и образов. По сути - это моделирование общения человека с компьютером. В настоящее время используется для военных целей. Такой интерфейс применяется при ведении воздушного боя.

Эргономичность интерфейса

Под эргономичностью интерфейса понимается удобство общения пользователя с программным продутом. Чтобы оценить эти удобства введены критерии эргономичности интерфейса, это интуитивность (естественность), непротиворечивость (последовательность), визуализация, система навигации, гибкость, поддержка пользователя. Все эти критерии в большей степени зависят от интеллектуальности самой программы, но это уже другая история.

Рассмотрим эти критерии поподробней.

Интуитивность иди естественность – это свойство программного продукта, адаптироваться под требования пользователя, а именно:

- общение происходит при помощи языка пользователя (или приближен к нему);
- контекстные подсказки (по ходу написания или какой-либо другой работы, программа выдаёт пользователю подсказки, советы, пояснения);
- отсутствуют жёсткие требования к порядку ведения диалога пользователя с машиной (пользователь сам строит диалог по мере решения задачи);

• не требуется предварительная обработка данных перед вводом их пользователем в систему (это влияет на быстродействие и исключает появления ошибок).

Непротиворечивость или последовательность ведения диалога гарантирует единство общих принципов работы с системой. Данный критерий содержит:

• последовательность в интерпретации команд; мнемоническое обозначение должны иметь только одинаковые команды;

• последовательность в использовании форматов данных – в одном формате должны представляться аналогичные;

• последовательность в размещении информации на экране –информативность сообщения, должна предоставляться пользователю по степени важности (предупреждение об ошибке появится в центре экрана, а вспомогательная информация в нижнем правом углу).

Выделение элементов интерфейса актуализирует внимание пользователя на конкретной информации. Но стоит учитывать, что при большом объёме выделенной информации, актуализация размывается.

Элементы можно выделить следующими способами:

1) движение (мигание или изменение позиции). Очень эффективный метод, поскольку глаз имеет специальный детектор для движущихся элементов;

2) яркость. Не очень эффективный метод, поскольку люди могут обнаружить всего лишь несколько уровней яркости;

3) цвет. Очень эффективный метод. Основное его назначение - создание интерфейсов, более интересных для пользователя. Он используется для группировки информации, выделения различий между информацией, выделения простых сообщений (ошибки, состояния). Важно отметить, что 9% людей не различают цвета (обычно красно-зеленые сочетания). Однако эти люди могут отличать черно-белые оттенки, поэтому проектировщики интерфейса должны проверять, не нарушает ли восприятие пользователей этой категории использование различных цветов;

4) форма (вид символа, шрифт, начертание, размер). При выделении объектов обычно используют увеличение в 1.5 раза.

5) оттенение (различная текстура объектов);

6) окружение (подчеркивание, рамки, инвертированное изображение).

Система навигации обеспечивает пользователю способность перемещаться между различными экранами, информационными единицами и подпрограммами в ходе ведения диалога. Тип системы навигации существенно зависит от принятого вида интерфейса: для интерфейса языка команд очень мало способов обеспечения полноценной навигации; в интерфейсах с меню можно использовать иерархически структурированные меню, которые будут «направлять» пользователя. Общие принципы проектирования системы навигации включают: использование заголовков страниц для каждого экрана; использование номеров страниц, номеров строк и столбцов; отображение текущего имени файла вверху страницы.

Поддержка пользователя во время диалога - это мера помощи, которую диалог оказывает пользователю при его работе с системой. Она включает в себя:

1) инструкции пользователю - необходимы для направления пользователя в нужную сторону, подсказок и предупреждений для выполнения необходимых действий на пути решения задачи. Инструкции могут быть обеспечены в форме диалога, экранных заставок, справочной информации и т.п. Они могут предложить пользователю: выбрать из предложенных альтернатив некую опцию или набор опций; ввести некоторую информацию; выбрать опцию из набора опций, которые могут изменяться в зависимости от текущего контекста; подтвердить фрагмент введенной информации перед продолжением ввода. Инструкции могут быть помещены в модальные диалоговые окна, которые вынуждают пользователя ответить на вопрос прежде, чем может быть предпринято любое другое действие, потому что все другие средства управления заморожены. Это может быть полезно, когда система должна вынудить пользователя принять решение перед продолжением работы. Немодальные диалоговые окна позволяют работать с другими элементами интерфейса, в то время как само окно может игнорироваться;

2) подтверждение действий системы - используется, чтобы пользователь мог убедиться, что система выполняет, выполнила

или будет выполнять требуемое действие (либо требуемые действия по каким-то причинам не выполнены). В полноценной системе пользователь также может всегда получить информацию о состоянии системы, процесса или активной подпрограмме;

3) сообщения об ошибках - должны объяснить, в чем ошибка, и указать, как ее исправить.

Ошибки могут быть классифицированы различным образом, примеры таких классификаций можно найти в соответствующей литературе по инженерной психологии. Там же можно найти информацию о техниках защиты от ошибок и методах их устранения применительно к пользовательским интерфейсам. Подробное рассмотрение этого вопроса выходит за рамки курсовой работы.

4) Гибкость диалога - это мера того, насколько хорошо диалог соответствует различным уровням подготовки и производительности труда пользователя. При этом диалог может подстраивать свою структуру или входные данные. Гибкость диалога проявляется в способности диалоговых систем адаптироваться либо с помощью пользователя, либо самостоятельно к любому возможному уровню подготовки оператора. Этот параметр влияет на эргономичность опосредовано (через показатель осваиваемости), на качество деятельности достаточно хорошо подготовленного оператора влияния не оказывает (по материалам лекций кафедры «Автоматика и информационные технологии» а также из курса прикладной эргономики). Потому подробное рассмотрение также выходит за рамки представленной курсовой работы.

5) Информативность

Студия Артемия Лебедева утверждает следующее: информация, передаваемая человеку от устройства или программы, содержит в себе смысловую часть. Качество, характеризующее долю полезной информации в общем объеме сообщения, можно назвать информативностью.

Хороший интерфейс передает суть информации минимальными средствами. Зачастую объем передаваемой информации может быть сокращен в разы без ущерба для смысла. И наоборот, информационная ценность может быть повышена без увеличения объема сообщения. Такая оптимизация

должна быть проведена на уровне используемых языковых формулировок, визуальных средств и общей структуры интерфейса.

Информация, передаваемая человеку, должна быть не только полной, но и наглядной. Например, для визуализации трехмерных поверхностей кроме имитации освещения часто используется цветовая шкала (псевдоспектр), которой кодируется высота каждой точки, - это позволяет определять области с одинаковой высотой. Каждый следующий цвет в такой шкале должен выглядеть более светлым, чем предыдущий. К сожалению, в большинстве случаев для этого применяется либо физический спектр («радуга»), либо случайно выбранные градиенты, не подходящие для визуализации плавного изменения значений.

3.2.2. История развития современных пользовательских интерфейсов ОС

Операционная система обеспечивает определённый способ общения пользователя с устройствами компьютера - интерфейс. Интерфейс бывает:

- командный – управление устройствами осуществляется с помощью текстовых команд. Текстовая команда вводится с клавиатуры и завершается нажатием клавиши. Окно появляется если набрать в строке Поиска «cmd» и нажать Enter. Примером операционной системы с таким интерфейсом является MS DOS. Недостатки командного интерфейса: необходимость знать все текстовые команды на английском языке, а также потери времени при вводе команды с клавиатуры[19].
- графический – управление устройствами осуществляется с помощью элементов управления, отображаемых на экране. Это тип интерфейса, в котором в качестве органа управления кроме клавиатуры может использоваться мышь или адекватное устройство позиционирования. Работа с такой системой основана на взаимодействии активных (указатель мыши) и пассивных (экранные кнопки, значки, флажки) экранных элементов управления. Указатель помещается на пункт меню (надпись или

картинку) и нажимается клавиша Enter клавиатуры или левая кнопка мыши. Операционные системы с графическим интерфейсом ещё называются операционными средами (ОС). Типичной операционной средой является среда Windows. Работа в среде Windows предполагает наличие и активное использование мыши. В настоящее время, с появлением мощных компьютеров, широкое распространение получили два типа ОС. К первому типу относятся достаточно похожие ОС семейства Windows компании Microsoft. Они многозадачные и имеют многооконный графический интерфейс. На рынке персональных компьютеров с Windows конкурируют ОС типа UNIX. Это многозадачная многопользовательская ОС с командным интерфейсом. В настоящее время разработаны расширения UNIX, обеспечивающие многооконный графический интерфейс. UNIX развивалась в течение многих лет разными компаниями, но до недавнего времени она не использовалась на персональных компьютерах, т.к. требует очень мощного процессора, весьма дорога и сложна, её установка и эксплуатация требуют высокой квалификации. В последние годы ситуация изменилась. Компьютеры стали достаточно мощными, появилась некоммерческая, бесплатная версия системы UNIX для персональных компьютеров - система Linux. ОС Linux считаются UNIX-подобными. Это связано с 4 тем, что Линус Торвальдс и его единомышленники использовали при создании своей бесплатной операционной системы ключевые концепции, реализованные в семействе ОС — UNIX. По мере роста популярности Linux, в ней появились дополнительные компоненты, облегчающие её установку и эксплуатацию. Немалую роль в росте популярности этой системы сыграла мировая компьютерная сеть Internet. Хотя освоение Linux гораздо сложнее освоения систем типа Windows, Linux - более гибкая и в то же время бесплатная система, что и привлекает к ней многих пользователей. Операционные системы Linux отлично приспособлены для администрирования серверов. Поэтому наибольшую востребованность данные ОС имеют среди корпораций — в частности, в сфере предоставления услуг хостинга, в сегменте облачных решений. В сегменте ПК, ориентированных на частных пользователей, популярность Linux

значительно уступает Windows, несмотря на то, что по базовым функциям современные дистрибутивы Linux, в принципе, сопоставимы с возможностями ОС от Microsoft. На базе Linux вместе с тем разработана самая популярная ОС для смартфонов и планшетов — Android. Которая, в свою очередь, по распространенности значительно опережает мобильную версию Windows. Apple производит компьютеры Macintosh с ОС MacOS. Эти компьютеры используются преимущественно издателями и художниками. Фирма IBM производит ОС OS/2. Особой популярностью в качестве домашней ОС никогда не пользовалась, оставаясь в тени Windows. OS/2 представляет собой самостоятельную линию развития операционных систем, отличаясь от Windows NT существенно меньшей требовательностью к ресурсам компьютера, а от Linux/UNIX – принципиальной разницей в подходе к разработке и большей схожестью графического интерфейса пользователя с Windows. Операционная система OS/2 такого же класса надёжности и защиты, как и Windows NT. Сейчас ОС семейства UNIX, как и Linux, в основном задействуются в среде корпораций — как инструмент управления серверами. Однако значительна распространенность соответствующих решений также и в сегменте ПК для частных пользователей, поскольку платформа Mac управляется OS X либо ее предшественницей — ОС Mac OS, базирующимися на UNIX. Мобильные гаджеты компании Apple (iPhone, iPad) управляются iOS, также относящейся к UNIX-системам. Основные свойства современных операционных систем

- графический интерфейс;
- наличие почти полного набора системных программных средств;
- устойчивость в работе;
- упрощенная настройка и подключение новых периферийных устройств;
- многозадачность.

Эволюция Windows

Эволюция операционных систем связана с эволюцией компьютерных технологий. Когда вычислительная мощность компьютеров была мала, действия над данными совершались с помощью набора команд. Эти команды обрабатывались специальной программой, находящейся в оперативной памяти. Одновременно могла обрабатываться только одна задача. С увеличением мощности компьютеров был применён объектно-ориентированный метод, когда указанному объекту назначается действие и реализована многозадачность (кооперативная и вытесняющая). В случае кооперативной многозадачности все запущенные приложения образовывали очередь на предмет выделения операционной системой ресурсов. Если приложение обращалось к системе чаще, чем другие, то выполнялось только оно, а остальные приложения только занимали оперативную память и замедляли работу в целом. В случае вытесняющей многозадачности ресурсы системы эффективно распределены между приложением, с которым пользователь работает непосредственно, и приложениями, работающими в фоновом режиме. Заканчивающие работу приложения вытесняются и их ресурсы предаются активным приложениям. В начале 80-х был лишь один компьютер, с которым рядовой пользователь мог общаться на «ты» — Lisa от Apple. Проект оказался провальным, но, с одной стороны, он подготовил почву для Mac, а с другой — создал прецедент использования графического пользовательского интерфейса. Изначально идея интерфейса WIMP (Windows, Icons, Menu, Pointer) принадлежала Xerox. Но, не 5 сделав ничего толкового, Xerox передала идею Apple, которая превратила окна, меню, иконки и курсор в главную «фишку» потребительских ПК. DOS (Dick Operating System) — 16-разрядная однозадачная операционная система обладала интерфейсом командной строки и могла работать только с 640 килобайтами оперативной памяти. С ростом оперативной памяти компьютеров и появлением программ, которым требовался для работы весь объём памяти, DOS перестала удовлетворять предъявляемым к ней требованиям. Компания Microsoft работала над операционной системой MS-DOS, командная строка которой была посредником между человеком, «железом» и программами. Пользовательский опыт от MS-DOS был диаметрально противоположным тому, что

предлагала система Apple. И, конечно же, Microsoft не могла закрыть на это глаза. В 1983 году было объявлено о начале работы над графическим интерфейсом для MS-DOS, а в ноябре 1985 года официально вышла программная оболочка для MS-DOS под названием. Windows 1.0 Вторая версия Windows 2.0 выходит с улучшенной графикой. На рабочем столе появились значки, возможность запуска нескольких окон, которые можно наложить друг на друга, теперь пользователь может взаимодействовать с системой используя «горячие» комбинации клавиш. Windows 3.0 – графическая надстройка над DOS упростила работу пользователя благодаря полноценному графическому интерфейсу. С неё начинается успех системы. 1990 года Windows 3.0 начинает комплектоваться пакетом офисных приложений Microsoft Office, который включал в себя Word, Excel и Powerpoint. Windows 3.1 и Windows 3.11. стали следующим этапом в развитии Windows. Разработчики называли ее операционной системой, хотя в действительности она таковой не являлась, а представляла собой оболочку установленную поверх DOS. В Windows 3.x было введено несколько весьма существенных решений, которые позволили сделать работу в этой версии системы более удобной быстрой. Система включала относительную поддержку мультимедиа и работу в локальной сети, исчез пресловутый барьер 640 килобайт и компьютер мог использовать всю установленную в нём оперативную память. Недостатком Windows 3.x. являлось неустойчивость при выполнении дисковых операций чтения/записи в многозадачном режиме. Windows 95 – новая 32-разрядная версия операционной системы. Её отличала абсолютная поддержка мультимедиа, т.к. в неё был интегрирован программно-драйверный комплекс, предоставляющий приложениям Windows прямой доступ к аппаратным устройствам (звуковой карте, видеокарте и т.д.), новый существенно более легкий в применении интерфейс, использование длинных имен файлов. Система работала не совсем стабильно, да и среда совершенна новая. Windows 95 была самой разрекламированной системой, это не помещало пользователям дать ей новое имя «Маздай» (от англ. «must die» — должен умереть) и WinDoze (Сонные окна). Рядовым пользователям приходилось привыкать к особенностям Windows

95, периодически переустанавливать её, перейти было больше некуда, только на популярные среди программистов системы WinNT и Linux, OS/2 в которых рядовой пользователь не мог разобраться. Windows 98 не являлась чем-то принципиально новым. Можно подумать, что это та же Windows 95, но с установленным Internet Explorer. Внешне версии этих операционных систем достаточно отличаются, но из "внутренних" различий следует отметить улучшенный механизм управления оперативной памятью, улучшенные средства управления Windows и восстановления после сбоев и многое другое. Windows 98 содержала массу новых программ и утилит – полный комплект программного обеспечения для работы в Интернет и утилиту конвертации файловой системы FAT 16 в новую версию FAT 32. Для работы Windows 98 комфортный объём оперативной памяти – 64 Мб. Windows 95/98 не являются в абсолютном понимании операционными системами, т.к. их работа требует установленной DOS и загрузка системы начинается именно с загрузки DOS. Windows ME — следующий этап в развитии семейства Windows 3.x/9x/ME. Она стала первой «домашней» системой, отказавшейся от поддержки DOS. В состав Windows ME вошли новая версия Internet Explorer, пакет для редактирования видео, универсальный проигрыватель, ряд новых инструментов обеспечения сохранности конфигурации и системных файлов, введена поддержка цифровых устройств ввода (цифровых фото- и видеокамер). Полный комплект Windows ME занимает 6 на жёстком диске 400 Мб – втрое больше места, чем Windows 98, и оперативной памяти требует 96 Мб. Windows NT/2000, следует рассматривать отдельно от Windows 3.x/95/98/ME. Разработка NT велась отдельно от остальных систем Windows. Все началось, когда Microsoft и IBM начали совместную разработку OS/2, но что-то пошло не так и компании разошлись, тогда OS/2 пришлось переделать в Windows NT. Эти 32-разрядные системы являются истинными операционными системами, поскольку имеют собственное ядро загрузки. Они были разработаны для управления компьютерными сетями, являются более сложными в настройках и поддерживают собственную файловую систему — NTFS, несовместимую с FAT16/32, которую используют Windows 3.x/95/98/ME,

отличаются высокой стабильностью работы и более высокими требованиями к ресурсам компьютера (64 Мб оперативной памяти и процессора Pentium II-300 им уже недостаточно). Однако обыкновенный пользователь, работая с программами под управлением Windows NT/2000, может и не заметить ощутимой разницы, т.к. интерфейс у этих систем, практически одинаков. Windows XP. Появилась красивая, понятная, быстрая, стабильная Windows, которая сразу же сожгла все мосты, заставив позабыть все (от Windows 95 до Windows ME) как страшный сон. 32 и 64 разрядная операционная система с полностью настраиваемым интерфейсом, поддерживает запись компакт-дисков на уровне самой ОС, содержит множество новых и обновлённых программ, обеспечивает стабильность и удобство работы, но требует не меньше 256 Мб оперативной памяти, процессора с частотой не менее 800 МГц и 2 Гб дискового пространства[15,16].

Теперь, все плюсы семейства NT доступны домашним пользователям, в самом лучшем виде. Windows XP оставалась самой популярной операционной системой до 2012 года. Windows Vista – самая большая неудача компании Microsoft, которая была медленной, с глюками, нестабильная, но с красивым интерфейсом. Windows 7 можно считать эволюционным развитием Windows Vista, поэтому заметных для пользователя внешних различий между этими системами не так много. Быстрая, стабильная, с красивым интерфейсом Aero. Главное внимание разработчиков было направлено на развитие и «шлифовку» уже выбранных решений, повышение производительности и обеспечение совместимости. Windows 7 научилась правильно работать с различными сетями, самостоятельно устанавливать драйвера к подключаемым устройствам, получила такую совершенную систему безопасности, которая снимает необходимость устанавливать сторонний антивирус. Также в Windows 7 появилась поддержка сенсорных экранов, которая полностью реализовалась лишь в следующей версии. Windows 7 – это общее имя для целого семейства операционных систем, ориентированных на разные задачи и различные аппаратные платформы, а поэтому выпускающихся в нескольких редакциях. В настоящее время компания Microsoft делает основной акцент на продвижение трёх

общедоступных редакций: Windows 7 Домашней расширенной, ориентированной на широкий круг домашних пользователей; Windows 7 Профессиональной, предназначенной для бизнеса; Windows 7 Максимальной, имеющей все реализованные возможности. Windows 8, в отличие от своих предшественников — Windows 7 и Windows XP, — использует новый интерфейс под названием Metro (произносится мэ́тро). Этот интерфейс появляется первым после запуска системы; он схож по функциональности с рабочим столом — стартовый экран имеет плитки приложений (сродни ярлыкам и иконкам), по нажатию на которые запускается приложение, открывается сайт или папка (в зависимости от того, к какому элементу или приложению привязана плитка). Также в системе присутствует и «классический» рабочий стол, в виде отдельного приложения. Вместо меню «Пуск» в интерфейсе используется «активный угол», нажатие на который открывает стартовый экран. Прокрутка в Metro-интерфейсе идет горизонтально. Также, если сделать жест уменьшения (или нажать на минус внизу экрана), будет виден весь стартовый экран. Плитки на стартовом экране можно перемещать и группировать, давать группам имена и изменять размер плиток (доступно только для плиток, которые были изначально большими). В зависимости от разрешения экрана система автоматически определяет количество строк для плиток — на стандартных планшетных компьютерах три ряда плиток. Цвет стартового экрана меняется в новой панели управления, также меняется и орнамент на заднем фоне. Однако сразу же после релиза, компания Microsoft начала разрабатывать новую ОС, не желая останавливаться на достигнутом. Обновление получило официальное название Windows 8.1. 7 Windows 10 — новая операционная система, единая для устройств, таких как компьютер, телефон, планшет или любой другой гаджет. Для всех этих устройств Microsoft создает единую платформу разработки и единый магазин приложений. Пользователи настольной версии Windows 10 получат возможность создавать несколько рабочих столов и переключаться между ними. Windows 10 позволяет легко использовать знакомое меню "Пуск", панель задач и рабочий стол. Живые плитки обеспечивают мгновенную потоковую

передачу самой важной информации. Играть с помощью Windows 10 стало еще удобнее: уже установленные игры работают быстрее, а еще можно подключаться к другим людям с устройствами Xbox One и Windows 10, чтобы играть вместе с ними. Microsoft Edge — новый браузер, который предоставляет расширенные возможности просмотра веб-страниц и делает работу в Интернете удобней и эффективней. Теперь писать и печатать можно непосредственно на веб-страницах. В Windows 10 есть все для выполнения ваших задач. Она позволяет не отвлекаться от дел благодаря простым способам закрепления приложений на месте и оптимизации пространства экрана. Просматривайте открытые задачи на одном экране и создавайте виртуальные рабочие столы, если вам нужно больше места или вы хотите сгруппировать элементы по проектам, например приложения Office для работы и игры — для развлечений. Windows 10 — самая безопасная из когда-либо созданных систем Windows. С первой загрузки и на протяжении всего срока поддержки устройства вы будете под защитой улучшенных функций безопасности, которые помогают бороться с вирусами, вредоносными программами и фишингом. Windows Hello распознает вас, чтобы только вы могли разблокировать устройство с помощью биометрических данных. Этот метод очень индивидуальный, а значит более защищенный. Необходимость вводить пароль при входе отпадает. Для Windows Hello требуется специализированное оборудование, в том числе сканер отпечатков пальцев, ИК-датчик с подсветкой или другие биометрические датчики. Лицензионное соглашение Windows 10 позволяет компании Microsoft собирать многочисленные сведения о пользователе, историю его интернет-деятельности, пароли к точкам доступа, данные, набираемые на клавиатуре и многое другое. Microsoft собирает данные, используемые для улучшения продуктов и служб. Примерами таких данных являются имя, адрес электронной почты, предпочтения и интересы, журнал браузера, журнал поиска и история файлов, данные телефонных звонков и SMS-сообщений, конфигурация устройств и данные с датчиков, а также данные об использовании приложений. Также могут собираться все данные, вводимые с

клавиатуры, рукописно или через системы распознавания речи[20-22].

Сбор данных может происходить при установке программ, использовании голосового поиска, открытии файлов, вводе текстов. Собранные данные могут передаваться третьей стороне с согласия пользователя для предоставления запрошенных услуг, а также предоставляться изготовителям оборудования. Требованиями к ресурсам компьютера: Процессор как минимум 1 ГГц или SoC. ОЗУ 1 ГБ (для 32-разрядных систем) или 2 ГБ (для 64-разрядных систем). Место на жестком диске 16 ГБ (для 32-разрядных систем) или 20 ГБ (для 64-разрядных систем). Видеоадаптер DirectX версии не ниже 9 с драйвером WDDM 1.0. Дисплей 800 x 600. Разрядность операционной системы Разрядность – способность одновременно обрабатывать какое-то количество битов. чем же отличаются дистрибутивы Windows x32 от Windows x64 и стоит ли вообще переходить на 64-бита. Основное отличие x64 от x32 в том, что версия x64 может работать с памятью вплоть до 32 Гбайт и запускать одновременно и 64-битные, и 32-битные.

3.2.3. Проблемы проектирования пользовательских интерфейсов ОС

User interface (UI) — это инструмент взаимодействия человека с устройством, позволяющий решать конкретные задачи. Поэтому и эволюцию интерфейсов стоит рассматривать в контексте развития пользовательских сценариев.

Аналоговый компьютер и первые терминалы

Первые вычислительные машины использовали преимущественно для лабораторных исследований. Учёные пытались оптимизировать свои трудозатраты и поэтому программировали компьютер на выполнение определённых математических операций с одной или несколькими переменными. Терминалы для ввода команд аналоговых компьютеров напоминали центр управления полётами: чтобы

запустить вычисления пользователю приходилось вручную создавать правильную последовательность действий при помощи тумблеров и проводов. Иногда этот процесс занимал несколько суток, а из-за риска человеческой ошибки достоверность полученных данных оставалась под вопросом[22].

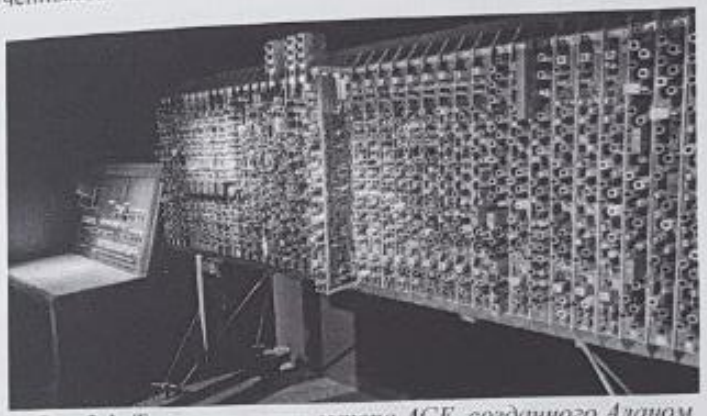


Рис.3.1. Терминал компьютера ACE, созданного Аланом Тьюрингом, 1950 год

Результат вычислений можно было увидеть на панели с лампочками-индикаторами, которые загорались в соответствии с полученными значениями, или на дисплее осциллографа — устройства, преобразующего электрические импульсы в визуальный график.

На начальном этапе существовали только два пользовательских сценария — программирование алгоритма вычислений и ввод переменных для расчёта.

От перфокарты к текстовым интерфейсам (TUI)

Первые цифровые компьютеры, как и аналоговые, предназначались исключительно для автоматизации лабораторных исследований — например, для расчёта допустимой вибрации при проектировании самолётов или обработки результатов переписи населения. Такие компьютеры управлялись при помощи перфокарт — тонких листов картона

в нужном порядке отверстиями, по которым считыватель определял двоичный код и выполнял соответствующую команду.

По мере усложнения вычислительных задач стали развиваться и устройства ввода-вывода — теперь часть данных для обработки можно было загрузить через прикрученную к компьютеру клавиатуру. Для вывода результатов вычислений подключили телетайп — электронную печатную машинку. Именно эта связка, перфолента + телетайп, легла в основу базового текстового интерфейса — интерфейса командной строки, в котором пользователь вводит текстовую команду и следующей строкой видит результат её выполнения.

Рис.3.2. IBM 360 и интерфейс консоли для редактирования гипертекста HES, 1969 год

В современных текстовых интерфейсах до сих пор сохранилась пасхалка от создателей первых перфокарт — максимальное количество знаков по ширине экрана соответствует количеству знаков на стандартной перфокарте.

Текстовые интерфейсы возникли как ответ на потребность пользователей в появлении новых сценариев — быстрого переключения между функциями и программами компьютера, поиска и добавления программ, получения обратной связи о различных видах ошибок.

Оконная революция и графические интерфейсы (GUI)

Переломный момент в развитии интерфейсов произошёл благодаря двум факторам: появлению первых персональных компьютеров и совмещению их с полноценным видеодисплеем. Оба эти события повлияли на решение Дугласа Энгельбарта создать среду управления, действительно понятную и удобную для обычного пользователя.

В то время, когда секретари набирали тексты на печатных машинках, менеджеры рисовали слайды вручную, а компьютеры использовались исключительно для лабораторных вычислений, Энгельбарт создал прототипы современных офисных программ для работы с текстами и графикой, навигации по файловой системе и даже для проведения видеоконференций с возможностью совместного редактирования документов.

Парадоксально, но разработанная им система NLS продолжала задействовать перфоленту для промежуточных операций и при этом уже имела базовый графический элемент всех современных ОС — окна. Вместе с этим Дуглас создал ещё один инструмент, который мы используем до сих пор — компьютерную мышь. Она стала связующим звеном между человеком и созданным графическим интерфейсом.

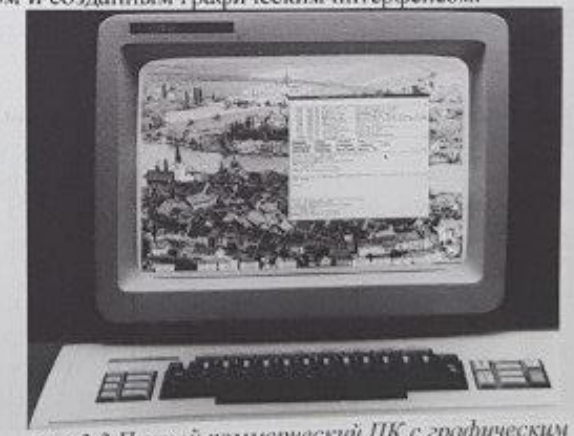


Рис.3.3.Первый коммерческий ПК с графическим интерфейсом Xerox Star 8010, 1981 год

Apple против Microsoft

Противостояние двух IT-гигантов началось именно с появления первых графических интерфейсов. В 1988 году Apple выдвинула иск к Microsoft о нарушении авторских прав. К разбирательству подключилась и компания Xerox, чья операционная система Xerox Star стала прототипом для Apple Lisa и Apple Macintosh.

Примечательно, что за 10 лет до этого Apple получила от Xerox лицензию на использование концепции окон, а затем выдала Microsoft несколько лицензий на отдельные графические элементы в Windows 1. В итоге суд отказал обеим компаниям.

Идеи Энгельбарта воплотились в первом персональном компьютере с графическим интерфейсом — Xerox Alto. Появление этого девайса вдохновило Стива Джобса и Билла Гейтса на разработку собственных операционных систем, построенных на концепции WIMP (windows, icons, menus, pointing device).

Первые графические интерфейсы позволили реализовать многие сложные пользовательские сценарии, но главный из них — работа в режиме многозадачности, когда в нескольких окнах открыты различные программы и документы.

На кончиках пальцев: материальные интерфейсы

Ключевая роль материальных интерфейсов состоит в сокращении времени на ввод команд и принятие решений на основе полученной от компьютера информации. В период холодной войны такая задача прежде всего стояла перед операторами оборонных комплексов.

Поэтому, пока графические интерфейсы только готовились захватить мир, в военных лабораториях шли разработки, призванные дополнить визуальные элементы программного интерфейса неким материальным продолжением. Первыми дополнениями стали световые указки и стилусы — с их помощью воздушные диспетчеры выделяли на мониторе объект, координаты которого необходимо было передать пилоту истребителя.

В 1963 году в распоряжение военных поступил графический планшет со стилусом, предназначенный специально для работы с картами и указания на них цели. А в 1965-м — появился ёмкостный тачскрин, который ещё несколько десятилетий авиадиспетчеры использовали для управления полётами.

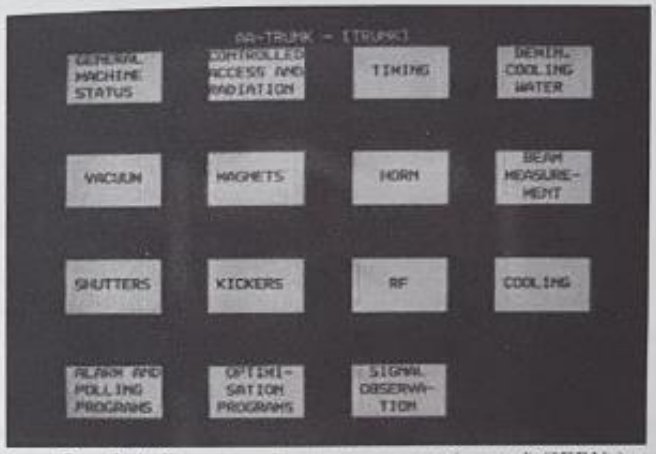


Рис.3.4. Сенсорный терминал, созданный CERN для управления антипротонным аккумулятором (AA), 1976 год.

Для более широкого круга пользователей материальные интерфейсы адаптировали намного позже — отчасти это связано с отсутствием такого запроса, ведь компьютеры ещё долгое время воспринимались исключительно как устройства для работы с данными и управления сложными аппаратами.

Заслуга в популяризации материальных интерфейсов принадлежит одному из создателей упомянутого выше Xerox Alto, Алану Кею. Вдохновившись работами по когнитивной психологии, уже в 1968 году он придумал концепцию образовательного планшета для детей — Dynabook. Центральная идея Dynabook заключается в возможности управлять им без дополнительных «костылей» — писать прямо на поверхности экрана, в два касания выбирать и просматривать образовательный

контент. Именно эту идею воплощал Стив Джобс, создавая iPhone и iPad[17-20].

Рис.3.5.Браслет Cicret проецирует на руку пользователя экран устройства и считывает команды при помощи датчиков движения, 2015 год

Сегодня материальные интерфейсы органично дополняют и расширяют возможности графических — на смартфонах, планшетах и компьютерах. Если 20 лет назад приходилось целиться указателем мыши в кнопку «X», чтобы закрыть ненужное окно, то теперь достаточно просто свайпнуть его. Причём последнее поколение кинетических интерфейсов позволяет манипулировать виртуальными объектами, не прикасаясь к экрану, — датчики распознают жесты и мимику пользователя и выполняют нужную команду.

Возникновение материальных интерфейсов связано со сценариями, предполагающими максимальную точность и скорость управления виртуальными объектами, — указание или выбор, масштабирование, пролистывание, перемещение.

Интерфейс-невидимка

Голосовые технологии стали если не ещё одной ступенью в эволюции среды «человек — компьютер», то как минимум достойным внимания ответвлением. Их, как и материальные интерфейсы, создавали для оптимизации трудозатрат —

преимущественно для преобразования устной речи в текст. И только в начале нулевых, в эпоху развития смартфонов, технологии распознавания речи интегрировали в операционную систему — так они превратились в интерфейс-невидимку, позволяющий управлять устройством бесконтактно.

Как всё начиналось

Прототипом современных голосовых интерфейсов стала машина «Одри» (Audrey — automatic digit recognition). Её создали в 1952 году в лаборатории Белла, того самого изобретателя телефона. «Одри» должна была распознавать произносимые абонентом цифры от 0 до 9, чтобы автоматизировать маршрутизацию звонков.

Качество распознавания сильно зависело от конкретного пользователя, так что полноценно внедрить голосовое управление команде Белла так и не удалось. Но именно это изобретение стало началом развития голосовых технологий.

Голосовой помощник — логичное продолжение идеи материального интерфейса, призванного минимизировать количество посредников между пользователем и машиной. Теперь компьютер доступен любому человеку, освоившему речь, — это самая интуитивно понятная форма взаимодействия с софтом из всех существующих, ведь она не требует каких-то специальных навыков и устройств для ввода команд.

Вычислительная машина IBM Shoebox распознаёт голосовую команду и выводит результат математической операции, 1962 год. Фото: IBM Archives

При этом, несмотря на активную популяризацию идеи умного помощника, который выполнит любую голосовую команду, совершить ещё одну революцию Алексе и Сири пока не удалось. Возможно, это связано с несовершенством технологии распознавания голоса — даже топовые интерфейсы пока что спотыкаются на особенностях дикции и мышления у разных пользователей. А может быть, проблема лежит намного глубже и появление голосового помощника ассоциируется у людей с сюжетами научно-фантастических фильмов, где поумневшие машины выходят из-под контроля и пытаются уничтожить человечество.

Голосовые интерфейсы позволяют реализовать глобальную группу сценариев, связанных с бесконтактным управлением устройством — когда пользователь физически находится на расстоянии от компьютера, у него заняты руки или нарушена координация движений.

Будущее здесь: нейроинтерфейсы

В конце концов, зачем нужен виртуальный секретарь, если можно передавать команды компьютеру силой мысли. Первые нейроинтерфейсы создавались для людей с ограниченными возможностями — ещё в 1970-х годах появились устройства, которые использовались как проводники информации и должны были компенсировать утраченные нейронные связи в организме человека.

Отважный невролог

Создателем первого вживлённого в мозг человека нейроинтерфейса стал невролог Филипп Кеннеди. Его мозговой имплант позволил парализованному пациенту управлять курсором мыши и вводить текст при помощи виртуальной клавиатуры. В следующем эксперименте Кеннеди установил в мозг другого пациента устройство, которое считывало мысленно произносимые звуки и воспроизводило простейшие слова.

Вскоре после этого министерство здравоохранения запретило Кеннеди проводить эксперименты на людях и прекратило финансирование. Тогда учёный решил на личном примере доказать государству, что такие операции безопасны — он установил электроды в собственный мозг и запустил эксперименты по совершенствованию звукового декодера. Но спустя несколько недель электроды пришлось удалить. На обе операции Кеннеди потратил около 100 тысяч долларов.

Сегодня технологии «мозг — компьютер» эволюционировали от инвазивных — то есть требующих непосредственной установки считывающих устройств в нервную систему человека — до неинвазивных. Например, шлемы с электродами, которые принимают сигналы мозга и преобразуют их в понятные компьютеру импульсы. Особенно футуристично выглядит нейроинтерфейс в сочетании с дополненной или виртуальной реальностью.

Благодаря нейроинтерфейсам время на передачу команды компьютеру можно сократить до доли секунды и при этом практически полностью исключить вероятность ошибки, когда вы случайно нажали не на ту кнопку или голосовой ассистент неверно распознал вашу речь. Это особенно актуально сейчас, когда все жизненные и рабочие процессы ускоряются, а пользователи закрывают сайт или приложение, если им приходится ждать отклика дольше пяти секунд.

Рис.3.6. Система «НейроЧат» использует нейроинтерфейс для общения и тренировки мозга, 2018 год.

Что дальше?

Интерфейсы, а точнее их создатели, проделали большой путь от терминалов с проводами, лампочками и инструкцией по эксплуатации в 10 томах к интуитивно понятной среде, которой можно управлять практически силой мысли. В каком направлении они будут развиваться дальше и какие возможности откроют перед пользователем — полностью зависит от фантазии нового поколения разработчиков.

10 правил проектирования интерфейсов, которые нельзя нарушать

В жизни есть определенные правила, которые нельзя нарушать. В дизайне пользовательского интерфейса тоже есть правила, по которым нужно жить. Их называют «эвристикой» или общими принципами, улучшающими юзабилити интерфейсов. Это повторяемые паттерны, проверенные временем и помогающие пользователям ориентироваться в интерфейсе. Хорошо спроектированный интерфейс всегда учитывает изложенные ниже принципы. В не очень хорошо спроектированном интерфейсе наверняка не хватает одного или нескольких принципов. Вы UI дизайнер, так зачем вам нарушать хотя бы одно из этих правил и создавать проблемы для пользователей[15-19]?

1. Видимость состояния системы

Система всегда должна информировать пользователей о том, что происходит, посредством соответствующего и своевременного фидбека.

Всегда предоставляйте пользователям соответствующую информацию, подсказки и контекст, чтобы они знали свое местоположение в системе. Это позволяет пользователю ощущать контроль и знать, что делать дальше. Товар был добавлен в корзину? Правка была сохранена? Сколько времени займет этот процесс? Каков статус моего заказа? Что сейчас происходит? Всегда отвечайте на подобные вопросы пользователям и никогда не оставляйте их в неведении и не заставляйте гадать.

2. Соответствие между системой и реальным миром

Система должна говорить на языке пользователей, используя знакомые им слова, фразы и концепции, а не системные термины. Следуйте правилам реального мира, чтобы информация отображалась в естественном и логическом порядке.

Используйте знакомые слова и язык. Не усложняйте формулировку. Значение слова или иконки на экране должны быть понятны вашей целевой аудитории. Люди приходят на ваш сайт или в приложение, имея сформированные ментальные модели и опыт, позволяющим им интерпретировать паттерны.

Одно из величайших достижений в технологии произошло с появлением графического пользовательского интерфейса (GUI). До него экран компьютера был ограничен непонятными текстовыми командами, которые было нужно запоминать и повторять всякий раз, когда вы хотели выполнить действие. Потом все изменилось. На экране отображались маленькие изображения папок и файлов и курсор в виде руки. Все это были визуальные символы, которые люди мгновенно понимали. Их не нужно объяснять, потому что они ссылаются на ментальные модели реального мира.

3. Последовательность и стандарты

Пользователи не должны задаваться вопросом, означают ли разные слова, ситуации или действия одно и то же.

Есть два типа последовательности: внутренняя и внешняя. Внутренняя последовательность относится к паттернам на вашем сайте или в приложении. Она может быть простой, например, ссылки одного цвета на всех страницах или одна иконка для одной концепции. Внешняя последовательность относится к соглашениям, применяемым в других программах и системах, используемых большинством людей. Например, корзина для покупок. Большинство людей знакомы с принципом работы корзины покупок. Не нужно изобретать велосипед. В противном случае пользователям будет сложнее узнать, как работает ваша корзина. Сохраняйте последовательность и избавьте пользователей от ненужной путаницы.

4. Контроль и свобода пользователей

Пользователи часто выбирают системные функции по ошибке, и им потребуется четко обозначенный «аварийный выход», чтобы выйти из нежелательного состояния без

необходимости проходить расширенный диалог. Добавьте функции отмены и повтора.

Всегда предоставляйте выход. Никогда не заставляйте пользователей выполнять ненужную функцию, и не заводите их в тупик. Например, если вы проектируете схему оформления заказа, позвольте пользователям продолжить делать покупки, если они того пожелают. Если они попытались совершить действие в приложении, позвольте отменить действие, если в последнюю минуту они засомневаются.

5. Предотвращение ошибок

Лучше хорошего сообщения об ошибках только тщательно продуманный дизайн, который в первую очередь предотвращает возникновение проблемы. Либо устраните условия, подверженные ошибкам, либо проверьте их и предоставьте пользователям подтверждения действия.

Когда системные операции критически важны, например, удаление файлов или рассылка письма 1000 получателям, убедитесь, что пользователи знают, что они делают нечто важное. Прежде чем совершить действие, покажите им диалоговое окно подтверждения или предоставьте дополнительную информацию, четко определяющую, что произойдет. Это помешает им двигаться дальше, если они не уверены в своих действиях. Кроме того, это избавит их от сожаления.

6. Пользователи должны узнавать, а не вспоминать

Сведите к минимуму нагрузку на память пользователя, сделав объекты, действия и параметры заметными. Пользователь не должен вспоминать информацию из одной части диалогового окна в другой. Инструкции по использованию системы должны быть заметными или доступными, когда это необходимо.

Одна из целей UI дизайнера – снизить когнитивную нагрузку на пользователей. Психическая память – это огромный ограниченный ресурс. Память работает двумя способами:

узнавание и воспоминание. Узнавание – это то, что вам сразу знакомо. Как лицо человека. Вы смотрите на лицо друга и сразу понимаете, что видели его раньше. Механизм воспоминания работает иначе. Это то, что вам нужно извлечь из памяти, например, имя человека. Воспоминание обычно требует больше времени и усилий, потому что разуму нужно обрабатывать больше информации, чтобы расшифровать то, на что он смотрит. С другой стороны, узнавание происходит мгновенно. Мы хотим больше узнаваемости в интерфейсе и меньше воспоминания. Хороший пример этого принципа – использование для функций универсально узнаваемых кнопок и иконок, например, дом для «ДОМОЙ» или карандаша для «РЕДАКТИРОВАТЬ». А если вам нужно спроектировать для интерфейса новые иконки, которые большинство людей никогда раньше не видели, используйте текстовый дескриптор, чтобы объяснить их и уменьшить когнитивную нагрузку.

7. Гибкость и эффективность использования

Ускорители, невидимые для начинающего пользователя, часто могут ускорить взаимодействие для опытного пользователя, потому система может обслуживать как неопытных, так и опытных пользователей. Разрешите пользователям настраивать частые действия.

Когда в приложении или системе определенные задачи повторяются снова и снова, вы можете сделать взаимодействие более эффективным для пользователей. Например, используйте в мобильном приложении свайп, чтобы сохранить или удалить элементы из списка. Обычный способ удалить элемент – открыть его, а затем нажать кнопку «Удалить». Продвинутый (и более эффективный) способ – просто свайпнуть и мгновенно удалить элемент из списка.

8. Минималистский дизайн и эстетика

Диалоговые окна не должны содержать неактуальную или редко используемую информацию. Каждая дополнительная единица информации в диалоговом окне конкурирует с

релевантными единицами информации и снижает их относительную заметность.

При проектировании для искусства не имеет значения, идем ли мы в сторону барокко и заполняем экран артефактами, текстурами и изображениями. Но при проектировании взаимодействия мы стремимся снизить соотношение сигнал / шум. Это делает интерфейс более понятным для пользователей. Вы можете применить этот принцип, просто уменьшив до минимума контент, отображаемый на экране, будь то изображения или текст, чтобы пользователь мог не отвлекаясь сосредоточиться на текущей задаче.

9. Помогите пользователям распознавать, диагностировать и устранить ошибки

Сообщения об ошибках должны быть изложены простым языком (без кодов), точно указывать на проблему и конструктивно предлагать решение.

Ошибки будут. Это неизбежно. Обязанность UI дизайнера, определить, что произойдет после того, как пользователь обнаружит ошибку. Таким образом, мы можем помочь пользователям, проектируя понятные страницы ошибок и предупреждения, предоставляющие варианты решения проблемы. Например, давайте рассмотрим широко распространенную страницу 404. Мы, как дизайнеры, знаем, что означает страница с ошибкой 404, но обычно пользователи этого не знают. Чтобы помочь им, мы должны перевести код 404 на простой язык, добавив текст, поясняющий, что только что произошло. Например: «Извините, но мы не смогли найти страницу, которую вы искали. Вот несколько страниц с похожим контентом…».

10. Справка и документация

Хотя лучше, если систему можно использовать без документации, пользователю может потребоваться помощь. Подобная информация должна быть удобной для поиска, ориентированной на задачу, содержать список конкретных шагов, которые необходимо выполнить, и не должна быть слишком большой.

Сделайте помощь и справку всегда доступной. Разместите ее на видном месте на верхней панели или в основной области навигации. Когда пользователи сталкиваются с проблемой и не могут легко найти решение, их необходимо направить в раздел, где они смогут его найти. Это может быть страница часто задаваемых вопросов с окном поиска, содержащим возможные предложения и ответы. В случае отсутствия ответа система должна предоставить возможность напрямую связаться со службой поддержки для получения дополнительной помощи, либо через систему тикетов, либо по электронной почте, либо по телефону.

КОНТРОЛЬНЫЕ ВОПРОСЫ

Вопрос 1

Какие ОС называются мультипрограммными

1. обеспечивающие одновременную работу нескольких пользователей
2. поддерживающие сетевую работу компьютеров
3. обеспечивающие запуск одновременно нескольких программ
4. состоящие более чем из одной программы

Вопрос 2

Какие существуют способы реализации ядра системы?

1. многоуровневая (многослойная) организация
2. микроядерная организация
3. реализация распределеннная
4. монолитная организация

Вопрос 3

Что обычно входит в состав ядра ОС

1. высокоуровневые диспетчеры ресурсов
2. аппаратная поддержка функций ОС процессором
3. базовые исполнительные модули
4. набор системных API-функций

Вопрос 4

Какие особенности характерны для современных универсальных операционных систем?

1. поддержка многозадачности
2. поддержка сетевых функций
3. обеспечение безопасности и защиты данных
4. предоставление большого набора системных функций разработчикам приложений

Вопрос 5

Какие утверждения относительно понятия «API-функция» являются правильными?

1. API-функции определяют прикладной программный интерфейс
2. API-функции используются при разработке приложений для доступа к ресурсам компьютера

3. API-функции реализуют самый нижний уровень ядра системы
4. API-функции — это набор аппаратно реализованных функций системы

Вопрос 6

характерны для ОС Unix

1. открытость и доступность исходного кода
2. ориентация на использование оконного графического интерфейса
3. использование языка высокого уровня С
4. возможность достаточно легкого перехода на другие аппаратные платформы

Вопрос 7

Какие типы операционных систем используются наиболее часто в настоящее время?

1. системы семейства Windows
2. системы семейства Unix/Linux
3. системы семейства MS DOS
4. системы семейства IBM OS 360/370

Вопрос 8

Какие задачи необходимо решать при создании мультипрограммных ОС

1. защита кода и данных разных приложений, размещенных вместе в основной памяти
2. централизованное управление ресурсами со стороны ОС
3. переключение процессора с одного приложения на другое
4. необходимость размещения в основной памяти кода и данных сразу многих приложений

Вопрос 9

Какое соотношение между используемыми на СЕРВЕРАХ операционными системами сложилось в настоящее время?

1. примерно поровну используются системы семейств Windows и Unix/Linux
2. около 10 % — системы семейства Windows, около 90 % — системы смейства Unix/Linux
3. около 90 % — системы семейства Windows, около 10 % — системы семейства Unix/Linux

4. около 30 % — системы семейства Windows, около 30 % — системы семейства Unix/Linux, около 40 % — другие системы

Вопрос 10

Какие утверждения относительно понятия «Ядро операционной системы» являются правильными?

1. ядро реализует наиболее важные функции ОС
2. подпрограммы ядра выполняются в привилегированном режиме работы процессора
3. ядро в сложных ОС может строиться по многоуровневому принципу
4. ядро всегда реализуется на аппаратном уровне

Вопрос 11

Какие сообщения возникают при нажатии на клавиатуре алфавитно-цифровой клавиши?

1. WM_KeyDown
2. WM_Char
3. WM_KeyUp
4. WM_KeyPress

Вопрос 12

Какие шаги в алгоритме взаимодействия приложения с системой выполняются операционной системой

1. формирование сообщения и помещение его в системную очередь
2. распределение сообщений по очередям приложений
3. вызов оконной функции для обработки сообщения
4. извлечение сообщения из очереди приложения

Вопрос 13

Что представляет собой понятие "сообщение" (message)?

1. небольшую структуру данных, содержащую информацию о некотором событии
2. специальную API-функцию, вызываемую системой при возникновении события
3. однобайтовое поле с кодом происшедшего события
4. небольшое окно, выводящее пользователю информацию о возникшем событии

Вопрос 14

Какие утверждения относительно иерархии окон являются справедливыми

1. главное окно может содержать любое число подчиненных окон
2. любое подчиненное окно может содержать свои подчиненные окна
3. подчиненные окна могут быть двух типов – дочерние и всплывающие
+ 4. приложение может иметь несколько главных окон

Вопрос 15

Как можно узнать координаты текущего положения мыши при нажатии левой кнопки

1. с помощью события WM_LbuttonDown и его поля LPARAM
2. с помощью события WM_LbuttonDown и его поля WPARAM
3. с помощью события WM_LbuttonDown и его полей WPARAM и LPARAM
4. с помощью события WM_LbuttonCoordinates

Вопрос 16

Какие функции можно использовать для получения контекста устройства?

1. GetDC
2. BeginPaint
3. ReleaseDC
4. CreateContext

Вопрос 17

Какая инструкция (оператор) является основной при написании оконной функции?

1. инструкция множественного выбора типа Case — Of
2. условная инструкция if – then
3. инструкция цикла с известным числом повторений
4. инструкция цикла с неизвестным числом повторений

Вопрос 18

Какой вызов позволяет добавить строку в элемент-список?

1. SendMessage (MyEdit, lb_AddString, 0, строка)
2. SendMessage ("Edit", lb_AddString, 0, строка)
3. SendMessage (MyEdit, AddString, 0, строка)
4. SendMessage (MyEdit, строка, lb_AddString, 0)

Вопрос 19

Какие утверждения относительно оконной функции являются правильными

1. оконная функция принимает 4 входных параметра
2. тело оконной функции – это инструкция выбора с обработчиками событий
3. оконная функция обязательно должна обрабатывать сообщение wm_Destroy
4. оконная функция явно вызывается из основной функции приложения

Вопрос 20

Какие сообщения возникают при нажатии на клавиатуре функциональной клавиши?

1. WM_KeyDown
2. WM_KeyUp
3. WM_KeyPress
4. WM_Char

Вопрос 21

Что может быть причиной появления внутреннего прерывания

1. попытка деления на ноль
2. попытка выполнения запрещенной команды
3. попытка обращения по несуществующему адресу
4. щелчок кнопкой мыши

Вопрос 22

Какие операции определяют взаимодействие драйвера с контроллером

1. проверка состояния устройства
2. запись данных в регистры контроллера
3. чтение данных из регистров контроллера
4. обработка прерываний от устройства

Вопрос 23

Какие операции включает в себя вызов обработчика нового прерывания

1. обращение к таблице векторов прерываний для определения адреса первой команды вызываемого обработчика
2. сохранение контекста для прерываемого программного кода

3. занесение в счетчик команд начального адреса вызываемого обработчика
4. внесение необходимых изменений в таблицу векторов прерываний

Вопрос 24

Что входит в программный уровень подсистемы ввода/вывода

1. драйверы
2. диспетчер ввода/вывода
3. системные вызовы
4. контроллеры

Вопрос 25

Что определяет понятие “порт ввода/вывода”

1. порядковый номер или адрес регистра контроллера
2. машинную команду ввода/вывода
3. устройство ввода/вывода
4. контроллер устройства ввода/вывода

ЛИТЕРАТУРА

1. Таненбаум Э. Современные операционные системы, 4-е издание/ Э. Таненбаум, Х Бос. --СПб.: Питер, 2021. -- 1120с.

2. Гордеев А. Операционные системы. Учебник для ВУЗов, 2-е издание/ А. Гордеев. -- СПб.: Питер, 2018. -- 416с.

3. Иртегов Д. Введение в операционные системы, 2-е издание /Д. Иртегов. -- М.: BHV, 2019. -- 1040 с.

4. Назаров С.В. Современные операционные системы: учеб. пособие/ С.В. Назаров. – М.:Интернет-университет информационных технологий, 2018, -- 368с.

5. Харви Дейтел Операционные системы. Том 1. Основы и принципы /Д. Харви [и др.]. -- М.: Бином-Пресс, 2020. -- 1024с.

6. Харви Дейтел, Пол Дейтел, Дэвид Р. Чофнес Операционные системы. Том 2. Распределенные системы, сети, безопасность/ Д. Харви [и др.]. -- М.: Бином-Пресс, 2011. -- 704с.

7. Стивенс, Р.UNIX. Профессиональное программирование / Р. Стивенс, С. Раго. –СПб.: Символ-Плюс, 2010. – 1040 с.

8. Иванов, Н. Программирование в Linux. Самоучитель / Н. Иванов. – СПб.:BHV, 2012. –400 с.

9. Руссинович, М. Внутреннее устройство MicrosoftWindows /М. Руссинович,Д. Соломон; пер. с англ.– 4-е изд. – М: Издательско-торговый дом «Русская редакция»; СПб.: «Питер», 2005. – 992 с.

10. Рихтер, Дж.Windows для профессионалов: создание эффективных Win32 приложений с учетом специфики 64-разрядной версии Windows/Дж. Рихтер; пер. с англ.– 4-е изд. – СПб; Питер; М.: Издательско-торговый дом "Русская Редакция", 2001. – 752 с.

11. Dabak, P. Undocumented Windows NT / P.Dabak, S.Phadke, M. Borate. –IDG Books Worldwide, Inc.; M&T Books. – 327 с.

12. Nebbett, G. Windows NT/2000 Native API Reference / G.Nebbett.– MTP – 482 с.

13. Солдатов, В.П. Программирование драйверов Windows / В.П. Солдатов. – М.: Бином-Пресс, 2006. – 576 с.

14. Комиссарова, В. Программирование драйверов для Windows / В. Комиссарова. – СПб.: БХВ-Петербург, 2007. – 256 с.

15. Driver development with Visual Studio Express / Microsoft Corporation [Электронныйресурс]. –2014. – Режим доступа http://www.microsoft.com/whdc/default.mspx

16. The Undocumented Functions Microsoft Windows NT/2000 / NTAPI Undocumented Functions [Электронныйресурс]. –2014. – Режим доступаhttp://undocumented.ntinternals.net

17. Дейтел, Р. Введение в операционные системы: в 2 т. / Р. Дейтел. –М.:Мир, – Т.1, Т.2. –1987.

18. Кейслер, С. Проектирование операционных систем для малых ЭВМ / С. Кейслер; пер. с англ. –М.: Мир, 1986. – 680 с.

19. Немет, Э.UNIX: руководство системного администратора / Э. Немет, Г. Снайдер, С. Сибасс, Т. Хейн.; Пер. с англ. – К.: BHV, 1996. – 832 с.

20. Керниган, Б.В. UNIX– универсальная среда программирования /Б.В. Керниган, Р. Пайк; пер. с англ., – М.: Финансы и статистика, 1992. – 304 с.

21. Краковяк, С. Основы организации и функционирования ОС ЭВМ / С. Краковяк. –М.: Мир, –1988 с.

22. Бек, Л. Введение в системное программирование / Л. Бек.–М.: Мир.–1988 с.

23. www.docsity.com;

24. www.docplayer.ru.

ОГЛАВЛЕНИЕ

КАСИМОВА Ш.Т.

ОПЕРАЦИОННЫЕ СИСТЕМЫ

УЧЕБНОЕ ПОСОБИЕ

Ташкент - "METODIST NASHRIYOTI" - 2024

Muharrir: Bakirov Nurmuhammad

Texnik muharrir: Tashatov Farrux
Musahhih: Shoumarova Oqila
Dizayner: Ochilova Zarnigor

Bosishga 1.04.2024.da ruxsat etildi.
Bichimi 60x90. "Times New Roman" garniturasi.
Ofset bosma usulida bosildi.
Shartli bosma tabog'i 20. Nashr bosma tabog'i 19,25.
Adadi 300 nusxa.

"METODIST NASHRIYOTI" MCHJ matbaa bo'limida chop etildi.
Manzil: Toshkent shahri, Shota Rustaveli 2-vagon tor ko'chasi, 1-uy.

+99893 552-11-21

ISBN 978-9910-03-167-0