ПРИДНЕСТРОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
имени Т.Г. ШЕВЧЕНКО
Физико-математический факультет
Кафедра Нелинейной оптики и КР

Операционные системы

Часть I

Курс лекций

Тирасполь, 2012

УДК 681.3.066 (078.8)
ББК 3973.2 018я73
0-60

Составители:
В.В. Васильев, преподаватель кафедры НО и КР ПГУ имени Т.Г. Шевченко
О.Ф.Васильева, ст. преподаватель кафедры ОФ и МПФ ПГУ имени Т.Г. Шевченко

Рецензенты:
О.В. Коровай, к.ф.-м.н, доцент кафедры НО и КР ПГУ имени Т.Г. Шевченко
А.Л. Макаревич, к.ф.-м.н, доцент кафедры ТТЭМ ПГУ имени Т.Г. Шевченко

Операционные системы. Часть I: Курс лекций/Сост. В.В. Васильев, О.Ф. Васильева – Тирасполь, 2012. – 76 с.

В данном курсе лекций рассматриваются общие принципы функционирования операционных систем (ОС), а также основные алгоритмы и структуры данных, используемые при разработке отдельных подсистем и модулей ОС. Описываются подсистемы управления устройствами, данными, процессами и памятью. В качестве примеров ОС рассмотрены Windows, UNIX и MS-DOS.

Курс лекций будет полезен для студентов физико-математического факультета специальностей «Прикладная математика» и «Прикладная математика и информатика»

УДК 681.3.066 (078.8)
ББК3973.2 018я73
0-60

Утверждено Научно-методическим советом ПГУ им. Т.Г. Шевченко

СОДЕРЖАНИЕ

ПРЕДИСЛОВИЕ

В настоящее время мы являемся свидетелями небывалого расцвета операционных систем, поэтому для их изучения для студентов открываются огромные возможности: выпускаются новые операционные системы (ОС) для настольных компьютеров, кластеров компьютеров и параллельных вычислений, мобильных устройств, облачных вычислений. Бесспорным лидером в данной области является корпорация Microsoft, выпустившая целую серию ОС семейства Windows: Windows XP, Windows 2003, Windows Vista (2007), Windows 2008, Windows 2008 High-Performance Computing (HPC), Windows 7. Развиваются также диалекты ОС Linux (Red Hat, Fedora, Mandrake, Ubuntu, SuSE и др.– сотни диалектов). Linux – операционная система типа UNIX, ядро которой свободно распространяется с исходными кодами.

По мнению Дэвида Проберта, менеджера по разработке ОС Windows (Microsoft), знание операционных систем является основой успешной карьеры в сфере программирования. Предмет ОС сочетает в себе как математические методы, так и методы проектирования современного программного обеспечения, которые используются и во многих других современных областях – при разработке игр, клиент-серверных приложений, бизнес-приложений, Web-технологий и программных инструментов.

Знание ОС способствует становлению зрелого мышления программиста и хорошему знанию сетевых технологий и протоколов, виртуальных машин, методов современного программирования.

1. ВВЕДЕНИЕ. ПОНЯТИЕ ОПЕРАЦИОННОМЫЙ СИСТЕМЫ.

Операционная система (ОС) - комплекс системных и управляющих программ, предназначенных для наиболее эффективного использования всех ресурсов вычислительной системы (ВС) и удобства работы с ней. Это программа, которая выполняет функции посредника между пользователем и компьютером.

Вычислительная система - взаимосвязанная совокупность аппаратных средств вычислительной техники и программного обеспечения, предназначенная для обработки информации.

Назначение ОС - организация вычислительного процесса в вычислительной системе, рациональное распределение вычислительных ресурсов между отдельными решаемыми задачами; предоставление пользователям многочисленных сервисных средств, облегчающих процесс программирования и отладки задач. Операционная система исполняет роль своеобразного интерфейса (Интерфейс - совокупность аппаратуры и программных средств, необходимых для подключения периферийных устройств к ПЭВМ) между пользователем и ВС, т.е. ОС предоставляет пользователю виртуальную ВС. Это означает, что ОС в значительной степени формирует у пользователя представление о возможностях ВС, удобстве работы с ней, ее пропускной способности. Различные ОС на одних и тех же технических средствах могут предоставить пользователю различные возможности для организации вычислительного процесса или автоматизированной обработки данных.

В программном обеспечении ВС операционная система занимает основное положение, поскольку осуществляет планирование и контроль всего вычислительного процесса. Любая из компонент программного обеспечения обязательно работает под управлением ОС.

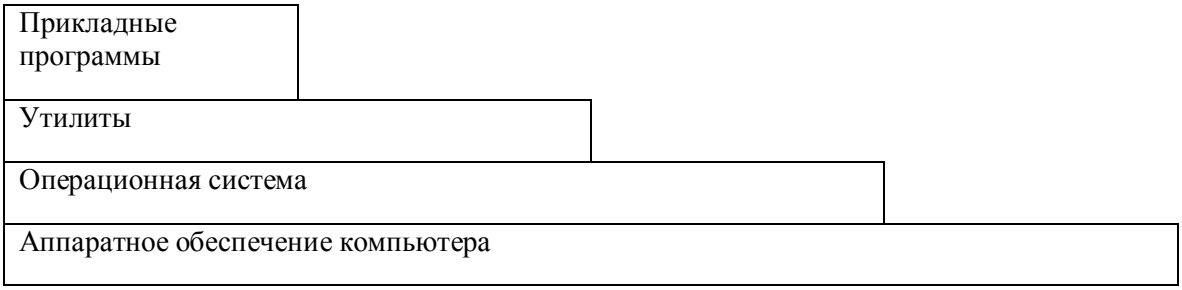

Рис.1

На рисунке 1 показано, что не один из компонентов программного обеспечения, за исключением самой операционной системы, не имеет непосредственного доступа к аппаратуре компьютера. Даже пользователь взаимодействует со своими программами через интерфейс. Любые их команды, прежде чем попасть в прикладные программы, проходят через операционные системы

В любой операционной системе можно выделить 4 основные части: *ядро, файловую структуру, интерпретатор команд пользователя и утилиты*.

1. Ядро - это основная, определяющая часть операционной системы, которая управляет аппаратными средствами и выполнением программ.
2. Файловая структура - это система хранения файлов на запоминающих устройствах.
3. Интерпретатор команд или оболочка - это программа, организующая взаимодействие пользователя с компьютером.

4. Утилиты - это просто отдельные программы, которые, вообще говоря, ничем принципиально не отличаются от других программ, запускаемых пользователем, разве только своим основным назначением - они выполняют служебные функции.

Основные функции операционных систем:

- загрузка приложений в оперативную память и их выполнение;
- стандартизованный доступ к периферийным устройствам (устройства ввода-вывода);
- управление оперативной памятью (распределение между процессами, виртуальная память);
- управление доступом к данным на энергонезависимых носителях (таких как жёсткий диск, компакт-диск и т. д.), как правило с помощью файловой системы;
- пользовательский интерфейс;
- сетевые операции, поддержка стека протоколов
- параллельное или псевдопараллельное выполнение задач (многозадачность);
- взаимодействие между процессами: обмен данными, взаимная синхронизация;
- защита самой системы, а также пользовательских данных и программ от действий пользователей (злонамеренных или по незнанию) или приложений;
- разграничение прав доступа и многопользовательский режим работы (аутентификация, авторизация).

В соответствии с условиями применения различают три режима ОС: пакетной обработки, разделения времени и реального времени. В режиме пакетной обработки ОС последовательно выполняет собранные в пакет задания. В этом режиме пользователь не имеет контакта с ЭВМ, получая лишь результаты вычислений. В режиме разделения времени ОС одновременно выполняет несколько задач, допуская обращение каждого пользователя к ЭВМ. В режиме реального времени ОС обеспечивает управление объектами в соответствии с принимаемыми входными сигналами. Время отклика ЭВМ с ОС реального времени на возмущающее воздействие должно быть минимальным.

2. ИСТОРИЯ РАЗВИТИЯ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ.

Первый период (1945-1955)

В середине 40-х были созданы первые ламповые вычислительные устройства, и появился принцип программы, хранимой в памяти машины. 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. 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.

Как правило, программист, разрабатывающий программу, заказывал ежедневно несколько часов машинного времени и в течение этого времени монопольно использовал машину. Выполнив очередной запуск отлаживаемой программы (которую надо было каждый раз вводить либо с клавиатуры, либо, в лучшем случае, с перфокарт), пользователь получал распечатку (чаще всего в виде массива цифр), анализировал результаты, вносил изменения в программу и снова запускал ее. Таким образом, в ходе сеанса отладки дорогостоящее оборудование простаивало 99% времени, пока программист осмысливал результаты и работал с устройствами ввода/вывода. Кроме того, сбой при вводе одной перфокарты мог потребовать начать сначала всю работу программы.

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. разрабатывается **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).

Применение транзисторов вместо электронных ламп привело к повышению надежности компьютеров. Теперь они смогли непрерывно работать довольно долго. Снизилось потребление 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. Появляются первые настоящие компиляторы, редакторы связей, библиотеки математических и служебных подпрограмм. 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.

Третий период (начало 60-х - 1980).

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.

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. 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. При этом каждая программа загружается в свой участок оперативной памяти, называемый разделом, и не должна влиять на выполнение другой программы

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. 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.
- 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.

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.

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.

Четвертый период (1980-настоящее время).

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 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.

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

3. ХАРАКТЕРИСТИКИ И ВИДЫ ОПЕРАЦИОННЫХ СИСТЕМ

Основные функции операционных систем

При рассмотрении основ функционирования ОС принято выделять четыре основных группы функций, выполняемых системой.

1. **Управление устройствами**. Имеются в виду все периферийные устройства, подключаемые к компьютеру, – клавиатура, монитор, принтеры, диски и т.п.

2. **Управление данными**. Под этим старинным термином сейчас понимается работа с файлами, хотя были времена, когда обращение к данным на магнитных носителях выполнялось путем указания адреса размещения данных на устройстве, а понятия файла не существовало.

3. **Управление процессами**. Эта сторона работы ОС связана с запуском и завершением работы программ, обработкой ошибок, обеспечением параллельной работы нескольких программ на одном компьютере.

4. **Управление памятью**. Оперативная память компьютера – это такой ресурс, которого часто не хватает. В этих условиях разумное планирование использования памяти является важнейшим фактором эффективной работы.

Имеется еще несколько важных обязанностей, ложащихся на ОС, которые трудно втиснуть в рамки традиционной классификации функций. К ним, прежде всего, относятся:

- **Организация интерфейса с пользователем**. Формы интерфейса могут быть разнообразными, в зависимости от типа и назначения ОС: язык управления пакетами заданий, набор диалоговых команд, средства графического интерфейса.

- **Защита данных**. Как только система перестает быть достоянием одного изолированного от внешнего мира пользователя, вопросы защиты данных от несанкционированного доступа приобретают первостепенную важность. ОС, обеспечивающая работу в сети или в системе разделения времени, должна соответствовать имеющимся стандартам безопасности.

- **Ведение статистики**. В ходе работы ОС должна собираться, храниться и анализироваться разнообразная информация: о количестве времени, затраченном различными программами и пользователями, об интенсивности использования ресурсов, о попытках некорректных действий пользователей, о сбоях оборудования и т.п. Собранная информация хранится в системных журналах и в учетных записях пользователей.

Операционная система (ОС) связывает аппаратное обеспечение и прикладные программы. Многие свойства различных программ похожи, и операционная система обычно предоставляет этот общий сервис. Например, практически все программы

считывают и записывают информацию на диск или отображают ее на дисплее. И хотя каждая программа в принципе может содержать инструкции, выполняющие эти повторяющиеся задачи, использование в этих целях операционной системы более практично.

По современным представлениям, ОС должна уметь делать следующее:

- Обеспечивать загрузку пользовательских программ в оперативную память и их исполнение.
- Обеспечивать работу с устройствами долговременной памяти, такими как магнитные диски, ленты, оптические диски и т.д. Как правило, ОС управляет свободным пространством на этих носителях и структурирует пользовательские данные.
- Предоставлять более или менее стандартный доступ к различным устройствам ввода/вывода, таким как терминалы, модемы, печатающие устройства.
- Предоставлять некоторый пользовательский интерфейс. Слово некоторый здесь сказано не случайно - часть систем ограничивается командной строкой, в то время как другие на 90% состоят из средств интерфейса пользователя.
- Параллельное (точнее, псевдопараллельное, если машина имеет только один процессор) исполнение нескольких задач.
- Распределение ресурсов компьютера между задачами.
- Организация взаимодействия задач друг с другом.
- Взаимодействие пользовательских программ с нестандартными внешними устройствами.
- Организация межмашинного взаимодействия и разделения ресурсов.
- Защита системных ресурсов, данных и программ пользователя, исполняющихся процессов и самой себя от ошибочных и зловредных действий пользователей и их программ.

Использование операционной системы делает программное обеспечение более общим: программы могут работать на любом компьютере, на котором можно запустить эту операционную систему, поскольку взаимодействуют с операционной системой, а не с аппаратурой.

Классификация ОС

Существуют различные виды классификации ОС по тем или иным признакам, отражающие разные существенные характеристики систем.

По назначению:

А) Системы общего назначения. Это достаточно расплывчатое название подразумевает ОС, предназначенные для решения широкого круга задач, включая запуск различных приложений, разработку и отладку программ, работу с сетью и с мультимедиа.

Различают три типа операционных систем (ОС) общего назначения: поддерживающие (1) однопрограммный режим работы и диалоговый способ общения, (2) обеспечивающие пакетную обработку задач в режиме мультипрограммирования и (3) операционные системы разделения времени.

1. Операционные системы, поддерживающие однопрограммный режим работы и диалоговый способ общения, включают в себя средства, обеспечивающие ввод и вывод информации, управляют работой системных обрабатывающих программ - трансляторов, редакторов, предоставляют пользователю сведения о ходе выполнения задач,

обеспечивают работу с библиотеками. Обычно такие операционные системы называют мониторными. Они не повышают производительности ЭВМ, но позволяют программисту вмешиваться в ход выполнения задания, что резко повышает производительность его работы, особенно на этапе отладки программ.

2. ОС обеспечивающие пакетную обработку задач в режиме мультипрограммирования. В RAM ЭВМ одновременно находится несколько системных и пользовательских задач, и когда одна из них обрабатывается процессором, то для остальных осуществляются необходимые обмены с внешним устройством (ВУ).

Эффективность использования ВС при этом во многом зависит от состава пакета задач, подлежащих выполнению, так как могут возникать ситуации, когда все задачи находятся в состоянии ожидания и процессор простаивает Эффективность работы пользователя при этом невысокая, так как в условиях пакетной обработки задач он не имеет возможности вмешиваться в процесс выполнения своей программы.

Такая операционная система должна выполнять рациональное планирование работ по обработке всех поступающих задач (комплекс мероприятий по вводу задач в ЭВМ, распознаванию их характеристик, размещению всех входных наборов данных на внешних носителях, организации входных и выходных очередей).

Как правило, задачи из входного потока данных, прочитанного одним из внешних устройств (ВУ), не сразу попадают в RAM ЭВМ, а размещаются на устройствах внешней памяти. В режимах пакетной обработки задачи выстраиваются в очередь (входную очередь), место задачи в очереди определяется ее приоритетом. Перенос задачи из очереди в RAM ЭВМ происходит автоматически.

Если для решения очередной задачи не хватает ресурсов, ОС должна принять одно из следующих решений:

- ✓ отобрать часть ресурсов у какой - либо другой задачи, выполнявшейся в данный момент и менее приоритетной;
- ✓ подождать, пока какая-нибудь из решаемых задач завершится и освободит требуемый ресурс;
- ✓ пропустить вне очереди ту задачу, чья очередь еще не подошла, но для выполнения которой ресурсов достаточно.

3. Операционные системы разделения времени позволяют реализовать возможность повышения производительности труда пользователя за счет его доступа к своей задаче в процессе ее выполнения и повышения производительности ВС за счет мультипрограммирования. Режим разделения времени создает иллюзию одновременного доступа нескольких пользователей ко всем вычислительным ресурсам ВС. Каждый пользователь общается с системой так, как если бы ему одному принадлежали все вычислительные ресурсы: он может остановить выполнение своей задачи в нужном месте, просмотреть требуемые области RAM, с заданного места выполнить свою программу по командам и т.д. На самом же деле каждый пользователь получает для своей задачи достаточную зону RAM, процессор и прочие вычислительные ресурсы только в течение определенного и достаточно малого интервала времени, как уже говорилось выше - кванта.

Пропускная способность ВС в режиме разделения времени ниже, чем при обработке задач в режиме мультипрограммирования, из-за накладных расходов ОС, вызванных частыми переключениями процессора и главным образом многочисленными переносами задач из RAM на жесткий диск и обратно, то есть свопингами. Во многих пользовательских системах режим разделения времени сочетается с пакетной обработкой

задач в режиме мультипрограммирования. В этом случае RAM ЭВМ разделяется на зону для пакетной обработки и на зону (или несколько зон в зависимости от емкости RAM) для выполнения задач в режиме разделения времени. Такое сочетание позволяет загружать процессор даже в ситуациях, когда все пользователи режима разделения времени остановят выполнение своих задач.

Б) Системы реального времени. Этот важный класс систем предназначен для управления технологическими объектами (такими, как летательные аппараты, технологические установки, автомобили, сложная бытовая техника и т.п.). Из подобного назначения вытекают жесткие требования к надежности и эффективности системы. Должно быть обеспечено точное планирование действий системы во времени (управляющие сигналы должны выдаваться в заданные моменты времени, а не просто «по возможности быстро»). Особый подкласс составляют системы, встроенные в оборудование. Такие системы годами могут выполнять фиксированный набор программ, не требуя вмешательства человека-оператора на более глубоком уровне, чем нажатие кнопки «Вкл.».

Операционные системы реального времени иногда делят на два типа:

1. Операционная система, которая может обеспечить требуемое время выполнения задачи реального времени даже в худших случаях, называется операционной системой жёсткого реального времени.
2. Операционная система, которая может обеспечить требуемое время выполнения задачи реального времени в среднем, называется операционной системой мягкого реального времени. Это такие системы, которые не могут гарантировать точное соблюдение временных соотношений, но «очень стараются», т.е. содержат средства для приоритетного выполнения заданий, критичных по времени. Такой системе нельзя доверить управление ракетой, но она вполне справится с демонстрацией видеофильма. Выделение подобных систем в отдельный класс имеет скорее рекламное значение, позволяя таким системам, как Windows NT и некоторые версии UNIX, тоже называть себя «системами реального времени».

В) Прочие специализированные системы. Это различные ОС, ориентированные прежде всего на эффективное решение задач определенного класса, с большим или меньшим ущербом для прочих задач. Можно выделить, например, сетевые системы (такие, как Novell Netware), обеспечивающие надежное и высокоэффективное функционирование локальных сетей.

По характеру взаимодействия с пользователем:

А) Пакетные ОС, обрабатывающие заранее подготовленные задания.

Б) Диалоговые ОС, выполняющие команды пользователя в интерактивном режиме. Красивое слово «интерактивный» означает постоянное взаимодействие системы с пользователем.

В) ОС с графическим интерфейсом. В принципе, их также можно отнести к диалоговым системам, однако использование мыши и всего, что с ней связано (меню, кнопки и т.п.) вносит свою специфику.

Г) Встроенные ОС, не взаимодействующие с пользователем.

По числу одновременно выполняемых задач:

А) Однозадачные ОС. В таких системах в каждый момент времени может существовать не более чем один активный пользовательский процесс. Следует заметить, что одновременно с ним могут работать системные процессы (например, выполняющие запросы на ввод/вывод).

Б) Многозадачные ОС. Они обеспечивают параллельное выполнение нескольких пользовательских процессов. Реализация многозадачности требует значительного усложнения алгоритмов и структур данных, используемых в системе.

По числу пользователей:

А) Однопользовательские ОС. Для них характерен полный доступ пользователя к ресурсам системы. Подобные системы приемлемы в основном для изолированных компьютеров, не допускающих доступа к ресурсам данного компьютера по сети или с удаленных терминалов.

Б) Многопользовательские ОС. Их важной компонентой являются средства защиты данных и процессов каждого пользователя, основанные на понятии владельца ресурса и на точном указании прав доступа, предоставленных каждому пользователю системы.

По аппаратурной основе:

А) Однопроцессорные ОС. В данном курсе будут рассматриваться только они.

Б) Многопроцессорные ОС. В задачи такой системы входит, помимо прочего, эффективное распределение выполняемых заданий по процессорам и организация согласованной работы всех процессоров.

В) Сетевые ОС. Они включают возможность доступа к другим компьютерам локальной сети, работы с файловыми и другими серверами.

Г) Распределенные ОС. Их отличие от сетевых заключается в том, что распределенная система, используя ресурсы локальной сети, представляет их пользователю как единую систему, не разделенную на отдельные машины.

Критерии оценки операционных систем

Надежность

Принято считать самым важным при оценке программного обеспечения

Под надежностью ОС понимается:

1) живучесть, т.е. способность сохранять хотя бы минимальную работоспособность в условиях аппаратных сбоев и программных ошибок. Высокая живучесть особенно важна для ОС компьютеров, встроенных в аппаратуру, когда вмешательство человека затруднено, а отказ компьютерной системы может иметь тяжелые последствия.

2) способность, как минимум, диагностировать, а как максимум, компенсировать хотя бы некоторые типы аппаратных сбоев. Для этого обычно вводится избыточность хранения наиболее важных данных системы.

3) ОС не должна содержать собственных (внутренних) ошибок. Это требование редко бывает выполнимо в полном объеме (программисты давно сумели доказать своим заказчикам, что в любой большой программе всегда есть ошибки, и это в порядке вещей), однако следует хотя бы добиться, чтобы основные, часто используемые или наиболее ответственные части ОС были свободны от ошибок.

4) способность противодействовать явно неразумным действиям пользователя. Обычный пользователь должен иметь доступ только к тем возможностям системы, которые необходимы для его работы. Если же пользователь, даже действуя в рамках своих полномочий, пытается сделать что-то очень странное (например, отформатировать системный диск), то самое малое, что должна сделать ОС, это переспросить пользователя, уверен ли он в правильности своих действий.

Эффективность

Как известно, эффективность любой программы определяется двумя группами показателей, которые можно обобщенно назвать «время» и «память». При разработке

системы приходится принимать много непростых решений, связанных с оптимальным балансом этих показателей.

Важнейшим показателем временной эффективности является производительность системы, т.е. усредненное количество полезной вычислительной работы, выполняемой в единицу времени. С другой стороны, для диалоговых ОС не менее важно время реакции системы на действия пользователя. Эти показатели могут в некоторой степени противоречить друг другу. Например, в системах разделения времени увеличение кванта времени повышает производительность (за счет сокращения числа переключений процессов), но ухудшает время реакции.

В программировании известна аксиома: выигрыш во времени достигается за счет проигрыша в памяти, и наоборот. Это в полной мере относится к ОС, разработчикам которых постоянно приходится искать баланс между затратами времени и памяти.

Забота от эффективности долгое время стояла на первом месте при разработке программного обеспечения, и особенно ОС. К сожалению, оборотной стороной стремительного увеличения мощности компьютеров стало ослабление интереса к эффективности программ. В настоящее время эффективность является первостепенным требованием разве что в отношении систем реального времени.

Удобство

Этот критерий наиболее субъективен. Можно предложить, например, такой подход: система или ее часть удобна, если она позволяет легко и просто решать те задачи, которые встречаются наиболее часто, но в то же время содержит средства для решения широкого круга менее стандартных задач (пусть даже эти средства не столь просты). Пример: такое частое действие, как копирование файла, должно выполняться при помощи одной простой команды или легкого движения мыши; в то же время для изменения разделов диска не грех почитать руководство, поскольку это может понадобиться даже не каждый год.

Разработчики каждой ОС имеют собственные представления об удобстве, и каждая ОС имеет своих приверженцев, считающих именно ее идеалом удобства.

Масштабируемость

Довольно странный термин «масштабируемость» (scalability) означает возможность настройки системы для использования в разных вариантах, в зависимости от мощности вычислительной системы, от набора конкретных периферийных устройств, от роли, которую играет конкретный компьютер (сервер, рабочая станция или изолированный компьютер) от назначения компьютера (домашний, офисный, исследовательский и т.п.).

Гарантией масштабируемости служит продуманная модульная структура системы, позволяющая в ходе установки системы собирать и настраивать нужную конфигурацию. Возможен и другой подход, когда под общим названием объединяются, по сути, разные системы, обеспечивающие в разумных пределах программную совместимость. Примером могут служить версии Windows NT/2000/XP, Windows 95/98 и Windows CE.

Понятие ядра ОС

Для понимания работы ОС необходимо уметь выделять основные части системы и их связи, т.е. описывать структуру системы. Для разных ОС их структурное деление может быть весьма различным.

Наиболее общими видами структуризации можно считать два. С одной стороны, можно считать, что ОС разделена на подсистемы, выполняющие свои группы функции. Такое деление достаточно обосновано, программные модули ОС действительно в

основном можно отнести к одной из этих подсистем. Другое важное структурное деление связано с понятием ядра системы.

Ядро, как можно понять из названия, это основная, «самая системная» часть операционной системы. Имеются разные определения ядра. Согласно одному из них, ядро – это резидентная часть системы, т.е. к ядру относится тот программный код, который постоянно находится в памяти в течение всей работы системы. Остальные модули ОС являются транзитными, т.е. подгружаются в память с диска по мере необходимости на время своей работы. К транзитным частям системы относятся:

- **утилиты** (utilities) – отдельные системные программы, решающие частные задачи, такие как форматирование и проверку диска, поиск данных в файлах, мониторинг (отслеживание) работы системы и многое другое;
- **системные библиотеки подпрограмм**, позволяющие прикладным программам использовать различные специальные возможности, поддерживаемые системой (например, библиотеки для графического вывода, для работы с мультимедиа и т.п.);
- **интерпретатор команд** – программа, выполняющая ввод команд пользователя, их анализ и вызов других модулей для выполнения команд;
- **системный загрузчик** – программа, которая при запуске ОС (например, при включении питания) обеспечивает загрузку системы с диска, ее инициализацию и старт.

Не менее важным является определение ядра, основанное на различении режимов работы компьютера. Все современные процессоры поддерживают, как минимум, два режима: **привилегированный режим** (он же режим ядра, kernel mode) и **непривилегированный** (режим задачи, режим пользователя, user mode). Программы, работающие в режиме ядра, имеют полный, неограниченный доступ ко всем ресурсам компьютера: его командам, адресам, портам ввода/вывода и т.п. В режиме задачи возможности программы ограничены, она, в частности, не может выполнить некоторые специальные команды. Аппаратное разграничение возможностей является абсолютно необходимым условием реализации надежной защиты данных в многопользовательской системе. Отсюда вытекает и определение ядра как части ОС, работающей в режиме ядра. Все остальные программы, как системные утилиты, так и программы пользователей, работают в режиме пользователя и должны обращаться к ядру для выполнения многих системных действий.

Следует сказать, что переходы из режима пользователя в режим ядра и обратно – это действия, требующие определенного времени, и слишком частое их выполнение может привести к заметному снижению скорости работы программ. В связи с этим определение того, какие функции должны поддерживаться ядром, а какие лучше выполнять в режиме пользователя – это непростая и важная задача, которую должны решить разработчики ОС.

Особую роль в структуре системы играют драйверы устройств. Эти программы, предназначенные для обслуживания конкретных периферийных устройств, несомненно, можно отнести к ядру системы: они почти всегда являются резидентными и работают в режиме ядра. Но в отличие от самого ядра, которое изменяется только при появлении новой версии ОС, набор используемых драйверов весьма мобилен и зависит от набора устройств, подключенных к данному компьютеру. В некоторых системах (например, в ранних версиях UNIX) для подключения нового драйвера требовалось перекомпилировать все ядро. В большинстве современных ОС драйверы подключаются к ядру в процессе загрузки системы, а иногда разрешается даже загрузка и выгрузка драйверов в ходе работы системы.

В качестве программного интерфейса системы, т.е. средств для обращения прикладных программ к услугам ОС, используется документированный набор системных вызовов или функций API (Applied Programming Interface). Между этими двумя терминами есть некоторая разница. Под системными вызовами понимаются функции, реализуемые непосредственно программами ядра системы. При их выполнении происходит переход из режима пользователя в режим ядра, а затем обратно. В отличие от этого, API-функции определяются как функции, описанные в документации ОС, независимо от того, выполняются ли они ядром или же системными библиотеками, работающими в режиме пользователя. В Windows часто несколько разных API-функций обращаются к одному и тому же недокументированному системному вызову, но имеют различные обрамляющие части, работающие в режиме пользователя.

Там, где различие между двумя этими понятиями несущественно, можно использовать нейтральный термин «системные функции»

3. КРАТКАЯ ХАРАКТЕРИСТИКА СОВРЕМЕННЫХ ОПЕРАЦИОННЫХ СИСТЕМ

MS-DOS

Операционная система MS-DOS (Disk Operating System), созданная фирмой Microsoft, появилась в 1981 г. практически одновременно с 16-ти разрядным ПК фирмы IBM (процессор intel 8080).

Точнее, вместе с IBM PC поставляется вариант этой системы, имеющий название PC DOS. С того времени DOS (так в дальнейшем будем называть PC DOS или MS DOS) довольно сильно изменилась. Первоначальный вариант DOS был очень похож на ОС CP/M-S6

MS DOS в течении 15 лет являлась наиболее широко распространенной операционной системой для персональных компьютеров. Число прикладных программ для MS-DOS составляет десятки тысяч. Такого большого количества программ в быстроразвивающейся компьютерной сфере никогда не было. Это разнообразие программ позволяло удерживать операционной системе MS DOS ведущее положение среди более прогрессивных и более мощных операционных систем. Естественно, что эта ОС имеет свои достоинства и недостатки.

Ограничения DOS:

-Отсутствие графического интерфейса.

-Ограничение памяти, доступной DOS-программ - 640 Кб. На самом деле DOS может использовать до 1 Mb ОЗУ, но архитектура IBM PC сокращает доступную память до 640 К. Есть множество обходных путей - отображаемая память, расширенная память, расширители DOS, блоки верхней памяти, создаваемые диспетчерами памяти для процессора 80386.

-Большинство основных прикладных программ общаются с экраном, клавиатурой и принтером в обход DOS, так как предоставляемые ею услуги по организации интерфейса с этими и другими устройствами совершенно недостаточны. DOS, например, не поддерживает ввод-вывод через последовательный порт по прерываниям.

-Не так-то просто обучиться пользоваться различными DOS-программами. В DOS нет стандартного интерфейса для прикладных программ, поэтому то, что работает в одной программе, вероятнее всего, не будет работать в другой.

-Под имя файла отводится 12 байт.

-Работа с жестким диском большого объема приводит к значительной фрагментации файлов.

-Полное отсутствие мультизадачности. DOS предназначена для одновременного выполнения только одной программы, и попытки заставить ее работать по-другому (за исключением некоторых очень специфичных случаев) чреваты крахом системы. Даже резидентные программы (TSR), являющиеся ограниченным, но все же весьма полезным исключением из правила, осложняют дело, когда конфликтуют друг с другом или с другими элементами системы. Имеется большое количество изделий различных фирм, обеспечивающих мультизадачность или переключение задач в системах, базирующихся на DOS, но ни одна из них не являлась особо эффективной.

Плюсы ОС DOS

-Одна из наиболее очевидных сильных сторон DOS - умеренные требования к оборудованию.

-DOS-программы работают быстро, по большей части благодаря тому, что большинство из них использует текстовый режим дисплея. Даже графические DOS-программы, как правило, в несколько раз быстрее своих Windows-аналогов, так как над ними не довлеет GDI (Graphics Device Interface, интерфейс графического устройства, компонент Windows, который используется программами для вывода на экран).

-Всегда под рукой.

-Работают на любом совместимом с Intel 8080 оборудованием

Windows 3.1x

Операционная оболочка Windows 3.1 - это разработанная фирмой Microsoft надстройка над операционной системой DOS, обеспечивающая большое количество возможностей и удобств для пользователей и программистов. (1990-1991 год, Именно в это время компьютеры, построенные на базе процессора Intel 386, на глазах становились всё доступней.).

ОС Windows 3.11 является 32-х разрядной ОС, с поддержкой 16-ти разрядных программ.

Ключевой идеей Windows является обеспечение полной независимости программ от аппаратуры. ОС Windows 3.1 изначально создавалась так, чтобы полностью взять на себя общение с конкретным типом дисплея или принтера. Как пользователю, так и программисту, создающему приложение под Windows, предоставлены универсальные средства, снимающие проблему обеспечения совместимости с конкретной аппаратурой (аппаратная совместимость) и программным обеспечением (программная совместимость).

Унифицированный единый графический интерфейс с пользователем облегчает изучение новых программных продуктов. Реализована возможность 3D-графики.

Одним из средств, обеспечивающих программную совместимость, является механизм обмена данными между различными приложениями. Специальный "почтовый ящик" (clipboard) Windows 3.1 позволяет пользователю переносить информацию из одного приложения в другое, не заботясь о ее формате и представлении. В отличие от профессиональных операционных систем, где механизм обмена данных между программами доступен только программисту, в Windows 3.1 это делается очень просто и наглядно для пользователя.

Механизм обмена данных между приложениями - жизненно важное свойство многозадачной среды. И в настоящее время производители программного обеспечения

пришли уже к выводу, что для переноса данных из одного приложения в другое одного "почтового ящика" явно недостаточно.

Появился новый, более универсальный механизм - OLE (Object Linking Embedded - Встроенная Объектная Связь), который позволяет переносить из одного приложения в другое разнородные данные.

Windows не только позволяет работать с привычным программным продуктом, но и предлагает дополнительные возможности (запуск нескольких программ одновременно, быстрое переключение с одной программы на другую, обмен данными между ними и т.п.). Обеспечена возможность работы со всеми прикладными программами MS-DOS (текстовыми процессорами, СУБД, электронными таблицами и пр.).

Windows 3.1 может работать в одном из трех режимов: Real (реальном), Standart (стандартном), 386 Enhanced (расширенном). В процессе установки Windows анализирует имеющиеся аппаратные ресурсы и автоматически устанавливает режим, наиболее полно использующий возможности имеющейся аппаратуры.

Последующие версии Windows были направлены на повышение надежности, а также поддержку средств мультимедиа (версия 3.1) и работу в компьютерных сетях (версия 3.11).

В реальном режиме Windows 3.1 не использует аппаратные возможности, не поддерживаемые MS-DOS (этот режим является единственно возможным для машин с процессором 8086/8088): как и в MS-DOS, пользователь ограничен оперативной памятью в 640 Кбайт.

Windows 95

Windows 95 представляет собой продукт эволюционного развития системы Windows 3.1x и содержит множество её исходных кодов.

Однако в основе архитектуры Windows 95 лежит ОС, обладающая совместимостью с MS-DOS, а не MS-DOS , при которой Windows работает как подсистема.

Результатом стало появление гибридной ОС, способной работать с 16-разрядными прикладными программами Windows, программами, унаследованными от DOS, windows 3.1x и Windows NT, и старыми драйверами устройств реального режима и в то же время совместимой с истинными 32-разрядными прикладными программами и 32-разрядными драйверами виртуальных устройств.

Наиболее важные усовершенствования, появившиеся в Windows 95:

- изначально заложенная в ней способность работать с 32-разрядными многопотоковыми прикладными программами,
- защищенные адресные пространства,
-вытесняющая многозадачность,
-намного более широкое и эффективное использование драйверов виртуальных устройств.
- Windows 95 должна была стать первой операционной системой, поддерживающей стандарт PLUG & PLAY.
- В сетевой среде невозможно обойтись без электронной почты, поэтому оболочка непосредственно поддерживает интерфейс электронной почты.
- Длинные имена файлов.
- Встроенные средства просмотра файлов.

- Отдельное адресное пространство. ПриложениеWin32 выполняется в своей собственной, защищённой области памяти, и никакая другая программа не может нарушить целостность ее кода или данных.

Недостатки:

- Относительно слабая защищенность от плохо работающих программ, содержащих ошибки.

- примущества Windows достигаются за счет значительного увеличения нагрузки на аппаратные средства компьютера.

Каждая собственная прикладная программа Windows 95 видит неструктурированное 4 Gb-ное адресное пространство, в котором размещается она сама плюс системный код и драйверы Windows 95. Каждая 32-разрядная прикладная программа выполняется так, как будто она монопольно использует весь ПК. Код прикладной программы загружается в это адресное пространство между отметками 2 и 4 Gb. Хотя 32-разрядные прикладные программы "не видят" друг друга, они могут обмениваться данными через буфер обмена (Clipboard), механизмы DDE и OLE. Все 32-разрядные прикладные программы выполняются в соответствии с моделью вытесняющей многозадачности, основанной на управлении отдельными потоками. Планировщик потоков, представляющий собой составную часть системы управления виртуальной памятью (VMM), распределяет системное время среди группы одновременно выполняемых потоков на основе оценки текущего приоритета каждого потока и его готовности к выполнению. Вытесняющее планирование позволяет реализовать намного более плавный и надежный механизм многозадачности, чем кооперативный метод, используемый в Windows 3.1x.

Адреса памяти ниже 4 Mb е отображаются в адресное пространство каждой прикладной программы и совместно используются всеми процессами. Благодаря этому становится возможной совместимость с существующими драйверами реального режима, которым необходим доступ к этим адресам. Это делает еще одну область памяти незащищенной от случайной записи. К самым нижним 64 К этого адресного пространства 32-разрядные прикладные программы обращаться не могут, что дает возможность перехватывать неверные указатели, но 16-разрядные программы, которые, возможно, содержат ошибки, могут записывать туда данные.

Семейство Windows NT

Windows NT по существу представляет собой операционную систему сервера, приспособленную для использования на рабочей станции. Этим обусловлена архитектура, в которой абсолютная защита прикладных программ и данных берет верх над соображениями скорости и совместимости. Чрезвычайная надежность Windows NT обеспечивается ценой высоких системных затрат, поэтому для получения приемлемой производительности необходимы быстродействующий процессор и по меньшей мере 16 Mb ОЗУ. В системе Windows NT безопасность нижней памяти достигается за счет отказа от совместимости с драйверами устройств реального режима. В среде Windows NT работают собственные 32-разрядные NT-прикладные программы, а также большинство прикладных программ Windows 95. Система Windows NT позволяет выполнять в своей среде 16-разрядные Windows- и DOS-программы.

Операционная система Windows NT поддерживает две файловые системы:

- файловую систему FAT (File Allocation Table),

- файловую систему NTFS.

ОС NT характеризуется поддержкой следующих механизмов:

1. модель модифицированного микроядра;
2. эмуляция нескольких ОС;
3. независимость от архитектуры процессора;
4. объектная модель;
5. многопоточность;
6. вытесняющая многозадачность;
7. виртуальная память с подкачкой страниц по требованию;
8. мультипроцессорная обработка;
9. интегрированная поддержка сети.

Операционная система реализует отдельное адресное пространство для всех выполняемых задач. Такая архитектура защищает приложения от повреждения. В отличие от Windows 9x программы не имеют возможности вносить изменения в системные файлы. Имеется возможность контролировать занимаемую задачей память и загрузку компьютера.

История создания, особенности и отличия Windows 98

Операционная система Windows 98 эволюционно развивает Windows 95. В июне 1997 года появилась первая публичная версия продукта, который аккумулировал в себя все, что было наработано за два года развития Windows 95, и назывался "Microsoft Memphis". Очень скоро проект Memphis получил официальное название Windows 98.

Windows 98 содержит все обновления предыдущих промежуточных выпусков Windows 95 (Service Pack 1 и OSR 2) в сочетании с поддержкой шины Universal Serial Bus (USB), интерфейсом Internet Explirer 4 и рядом новых функций. В систему добавлены новые свойства и новые приложения, интерфейс системы улучшен, найденные за три года ошибки по возможности исправлены, но по-прежнему базовым кодом для Windows 98, служит код Windows 95. Это 32-разрядная операционная система, поддерживающая работу с аппаратурой по стандарту Plug and Play, совместимая с Windows 95, но более высокопроизводительная и устойчивая.

По сравнению |с Windows 95 операционная система Windows 98 включает *более совершенную подсистему управления электропитанием, работает с несколькими мониторами, более эффективно работает с памятью, а также поддерживает новые типы оборудования* и множество новых моделей устройств, - например, более новых моделей принтеров. Система оптимизирована для нового интерфейса ACPI (Advanced Configuration and Power Interface) компьютеров, с системой BIOS, поддерживающей быструю загрузку. Поддержка спецификации OnNow позволяет переводить компьютер в режим низкого энергопотребления.

Специально для поддержки новых больших (с объемом более 2 Gb) жестких дисков в Windows 98 реализована ***новая файловая система FAT32***, которая впервые появилась в выпуске Windows 95 OSR2. FAT32 развивает прежнюю систему на основе 16-разрядной таблицы размещения файлов FAT (в одном контексте с FAT32 ее теперь чаще называют FAT16). FAT32 работает с жесткими дисками объемом до 2 Tb, при этом размер кластеров на разделах FAT32 сравнительно мал, что существенно экономит дисковое пространство.

Кроме того, Windows 98 включает поддержку новой модели драйверов WDM (Win32 Driver Model), которая позволяет драйверам устройств работать как под Windows 98, так и с будущими версиями Windows NT.

Программа установки Windows 98 проще, чем Windows 95. В комплект поставки включены новые средства сетевой и пакетной установки операционной системы - Microsoft Batch 98, INF Installer и Dbset.exe. Программа установки самостоятельно определяет оборудование и автоматически перезагружает компьютер.

Windows 98 имеет развитые средства работы с Интернетом, включая новый обозреватель Interriet Explorer 4.

Средство Microsoft WebTV для Windows 98 позволяет пользователю просматривать телевизионные передачи на экране компьютера, оборудованного платой ТВ-тюнера. На момент выпуска Windows 98 WebTV позволяет просматривать только кабельные и вещательные телепередачи в формате NTSC, анонсирована поддержка спутникового телевидения и других стандартов телевещания.

Windows 98 поддерживает безопасные удаленные сетевые соединения посредством виртуальных частных сетей (virtual private networking, VPN). Для этого в Windows 98 включены драйвер адаптера удаленного доступа с поддержкой VPN и протокол сетевого интерфейса WAN-сетей (network driver interface specification wide area network, NDISWAN).

Продолжает совершенствоваться поддержка мультимедиа. В Windows 98 встроена текущая версия DirectX - набора API, позволяющего приложениям работать напрямую с мультимедиа-устройствами.

Windows 98 работает устройствами графического ввода, такими как сканеры и цифровые камеры. Поддержка стандарта Image Color Matching (ICM) версии 2.0 гарантирует, что оригинальные цвета вводимого изображения будут правильно интерпретированы устройством. ввода и корректно отображены на мониторе, цветном принтере или в графическом файле.

Windows 98 содержит множество новых и усовершенствованных утилит, таких как программа резервного копирования, дефрагментатор, ускоряющий запуск прикладных программ, конвертор FAT16-дисков в формат FAT32 и др.

Семейство операционных систем UNIX,Linux.

UNIX имеет долгую и интересную историю.UNIX зародился в лаборатории Bell Labs фирмы AT&T в 1969 г. Имя UNIX (Uniplex Information and Computing Services) было дано ей еще одним сотрудником Bell Labs, Брайаном Керниганом, который первоначально назвал ее UNICS, подчеркивая ее отличие от многопользовательской MULTICS. Вскоре UNICS начали называть UNIX.

Широкое распространение UNIX получил с 1974 года, после описания этой системы Томпсоном и Ритчи в компьютерном журнале CACM. UNIX получил широкое распространение в университетах, так как для них он поставлялся бесплатно вместе с исходными кодами на С. Широкое распространение эффективных С-компиляторов сделало UNIX уникальной для того времени ОС из-за возможности переноса на различные компьютеры. Университеты внесли значительный вклад в улучшение UNIX и дальнейшую его популяризацию. Еще одним шагом на пути получения признания UNIX как стандартизованной среды стала разработка Денисом Ритчи библиотеки ввода-вывода stdio. Благодаря использованию этой библиотеки для компилятора С, программы для UNIX стали легко переносимыми.

Большинство современных UNIX-систем являются коммерческими версиями исходных дистрибутивов UNIX. Solaris от Sun, HP-UX Hewlett-Packard, AIX® от IBM являются лучшими представителями UNIX, которые, кроме того, имеют свои собственные уникальные элементы и свои собственные фундаментальные решения. Например, Sun Solaris - это UNIX, но, кроме того, она содержит много инструментов и расширений, разработанных специально в расчете на рабочие станции и серверы производства Sun.

Исходный код ядра коммерческих дистрибутивов UNIX является собственностью их разработчиков и не распространяется за пределы компании.

Широкое распространение UNIX породило проблему несовместимости его многочисленных версий. Очевидно, что для пользователя весьма неприятен тот факт, что пакет, купленный для одной версии UNIX, отказывается работать на другой версии UNIX. Периодически делались и делаются попытки стандартизации UNIX, но они пока имеют ограниченный успех. Процесс сближения различных версий UNIX и их расхождения носит циклический характер. Перед лицом новой угрозы со стороны какой-либо другой операционной системы различные производители UNIX-версий сближают свои продукты, но затем конкурентная борьба вынуждает их делать оригинальные улучшения и версии снова расходятся. В этом процессе есть и положительная сторона - появление новых идей и средств, улучшающих как UNIX, так и многие другие операционные системы, перенявшие у него за долгие годы его существования много полезного.

Наибольшее распространение получили две весьма несовместимые линии версий UNIX: линия AT&T - UNIX System V, и линия университета Berkeley-BSD. Многие фирмы на основе этих версий разработали и поддерживают свои версии UNIX: SunOS и Solaris фирмы Sun Microsystems, UX фирмы Hewlett-Packard, XENIX фирмы Microsoft, AIX фирмы IBM, UnixWare фирмы Novell (проданный теперь компании SCO), и список этот можно еще долго продолжать.

В 80-е годы были попытки превратить UNIX в коммерческую систему. Однако в 1991-1994 гг. Линус Торвальдс, в то время студент-программист из Хельсинки, заново написал систему, соответствующую стандартам POSIX, но отличающуюся от традиционной UNIX большей надежностью и эффективностью. Эта система получила название Linux.

Исходные тексты Linux свободно распространяются, что позволяет, как во времена молодости UNIX, развивать систему общими усилиями огромного сообщества заинтересованных программистов. Эффективной координации этих усилий очень способствует Интернет. Несколько позднее был открыт свободный доступ к текстам известной версии UNIXFreeBSD.

ОС Unix/Linux можно условно разделить на три уровня:

-Аппаратные средства - самый низкий уровень;

-Ядро – программа с включёнными в неё драйверами устройств, обеспечивающая доступ к аппаратным средствам системы для других программ;

-Пользовательские программы.

Ядро является главным исполняемым файлом. Оно стартует первым после загрузчиков, обеспечивает запуск командного интерпретатора и продолжает выполняться в течении всего сеанса. В задачи ядра входит управление всеми процессами.

Взаимодействую с аппаратной частью ядро реализует следующие элементы ОС:

-Процессы

-Сигналы
-Виртуальную память
-Межзадачное взаимодействие
-Файловую систему

В ОС Linux используется монолитные ядра. Они оптимизированы для более высокой производительности с минимальными контекстными переключениями. Такая архитектура упрощает поддержку кода ядра для разработчиков, но требует перекомпиляции ядра при добавлении новых устройств. ОС Windows 95/98 используют монолитное (monolithic) ядро.

Windows 2000/XP построены на архитектуре микроядра (microkernel architecture). Микроядра являются сравнительно небольшими и модульными. Благодаря последнему новые устройства зачастую добавляются как модули, которые можно загружать/выгружать на этапе исполнения без перекомпиляции ядра. На архитектуре микроядра построены также FreeBSD и Mac OS X.

В Unix/Linux графическая система существует отдельно от ядра и функционирует как обычное приложение. В операционных системах Windows графическая система интегрирована в ядро. В случае использования операционной системы на рабочей станции, особенно при запуске графикоемких приложений, возможно, лучше, когда графическая система входит в ядро - в этом случае она может быстрее работать. А при работе на сервере предпочтительней отделение графической системы от ядра ОС, так как она загружает память и процессор. В случае Unix/Linux графическую систему можно просто отключить, к тому же, если системный администратор ее все-таки хочет использовать, в Linux есть несколько графических оболочек на выбор, некоторые из них (например, WindowMaker) достаточно слабо загружают машину. Эта же особенность Unix-образных операционных систем позволяет запускать эти ОС на машинах с весьма скромными объемами ОЗУ и т.п. В случае Windows же графическая система слишком тесно интегрирована в ОС, поэтому она должна запускаться даже на тех серверах, на которых она вовсе не нужна.

Независимо от версии, общими для UNIX чертами являются:
- многопользовательский режим со средствами защиты данных от несанкционированного доступа,
- реализация мультипрограммной обработки в режиме разделения времени, основанная на использовании алгоритмов вытесняющей многозадачности (preemptive multitasking),
- использование механизмов виртуальной памяти и свопинга для повышения уровня мультипрограммирования,
- унификация операций ввода-вывода на основе расширенного использования понятия "файл",
- иерархическая файловая система, образующая единое дерево каталогов независимо от количества физических устройств, используемых для размещения файлов,
- переносимость системы за счет написания ее основной части на языке С,
- разнообразные средства взаимодействия процессов, в том числе и через сеть,
- кэширование диска для уменьшения среднего времени доступа к файлам.

OS/2 Warp

OS/2 v.2.0 была первой доступной и работающей 32-х битной операционной системой для персональных компьютеров.

В апреле 1987 г. компании IBM и Microsoft объявили о совместных планах по созданию новой операционной системы: OS/2. Прошло несколько лет, и мир стал свидетелем "бракоразводного процесса", в результате чего у OS/2 остался один родитель - компания IBM, а фирма Microsoft отдала все симпатии любимому детищу, имя которому Windows. Важно помнить, что OS/2 - это новая операционная система с графическим интерфейсам пользователя (ГИП), в то время как Windows представляла собой ГИП, работающий "поверх" DOS.

OS/2 является полностью защищенной операционной системой, благодаря чему невозможны конфликты между программами в памяти. Многозадачная система OS/2 способна выполнять одновременно несколько прикладных программ: например, Вы можете начать пересчет электронной таблицы, запустить печать документа в текстовом редакторе, связной пакет для приема/передачи сообщений электронной почты, а затем продолжить поиск записей в базе данных.

Система OS/2 поддерживает многопроцессные прикладные программы, рассчитанные на одновременное выполнение нескольких внутренних функций. Примерами могут служить текстовый редактор, в котором печать документа и проверка правописания осуществляются параллельно; электронная таблица с возможностью одновременного выполнения функций пересчета и просмотра или база данных, в которой можно совмещать функции обновления и поиска записей.

Архитектура OS/2 Warp Connect 3.0 во многом похожа на архитектуру Windows 95, но в ее концепции заложено меньше компромиссов, связанных с использованием старого 16-разрядного кода. В результате появилась ОС с лучшими, чем у Windows 95, средствами защиты, в которой можно выполнять программы OS/2, Win16 и DOS, однако несовместимая с 16-разрядными драйверами устройств. 32-разрядные прикладные программы Windows не могут выполняться в среде OS/2 Warp.

Собственным 32-разрядным прикладным программам OS/2 доступно 4 Gb-ное отдельное адресное пространство. Код прикладных программ отображается в диапазон адресов от 0 до 512 Mb, системный код OS/2 отображается в пространство от 512 Mb до 4 Gb. Эта область системного кода используется совместно всеми процессами. Исполняемые 32-разрядные прикладные программы изолированы друг от друга, хотя они могут общаться между собой с помощью средств вырезания и вставки (cut-and-paste) или механизма DDE OS/2. В системе OS/2 Warp применяется модель вытесняющей многозадачности собственных прикладных программ, основанная на управлении отдельными потоками.

Такая организация обладает во многом теми же достоинствами и недостатками, что свойственны Windows 95. Выделение системных ресурсов происходит гладко, а вызовы, направляемые в системные API, могут обслуживаться без существенных накладных расходов, так как системные DLL расположены в тех же адресных пространствах, что и вызывающая прикладная программа. Размер рабочего множества тоже удерживается в разумных пределах, так как не нужно создавать множественные экземпляры системных DLL. Но защита не гарантируется, поскольку плохо работающие прикладные программы все же могут испортить важные системные области.

5. ПРОЦЕССЫ И ИХ ПОДДЕРЖКА В ОПЕРАЦИОННОЙ СИСТЕМЕ

Понятие процесса

Понятие процесса характеризует некоторую совокупность набора исполняющихся команд, ассоциированных с ним ресурсов (выделенная для исполнения память или

адресное пространство, стеки, используемые файлы и устройства ввода-вывода и т. д.) и текущего момента его выполнения (значения регистров, программного счетчика, состояние стека и значения переменных), находящуюся под управлением операционной системы. Не существует взаимно однозначного соответствия между процессами и программами, обрабатываемыми вычислительными системами. В некоторых операционных системах для работы определенных программ может организовываться более одного процесса или один и тот же процесс может исполнять последовательно несколько различных программ. Более того, даже в случае обработки только одной программы в рамках одного процесса, нельзя считать, что процесс представляет собой просто динамическое описание кода исполняемого файла, данных и выделенных для них ресурсов.

Состояния процесса.

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

Видно, каждый процесс может находиться как минимум в двух состояниях: **процесс исполняется** и **процесс не исполняется**. Диаграмма состояний процесса в такой модели изображена на рисунке 2.

Процесс, находящийся в состоянии **процесс исполняется**, может через некоторое время завершиться или быть приостановлен операционной системой и снова переведен в состояние **процесс не исполняется**.

Рис 2. Простейшая диаграмма состояний процесса.

Приостановка процесса происходит по одной из двух причин:

1 – для его дальнейшей работы потребовалось возникновение какого-либо события (например, завершения операции ввода-вывода);

2 – истек временной интервал, отведенный операционной системой для работы этого процесса.

После этого операционная система по определенному алгоритму выбирает для исполнения один из процессов, находящихся в состоянии **процесс не исполняется**, и переводит его в состояние **процесс исполняется**. Новый процесс, появляющийся в системе, первоначально помещается в состояние **процесс не исполняется.**

Для того, чтобы избежать выбора операционной системой процесса не готового к исполнению (например ждёт данных), необходимо состояние **процесс не исполняется** разделить на два новых состояния - **готовность** и **ожидание** (рис.3).

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

Рис.3. Подробная диаграмма состояний процесса

Покинуть это состояние процесс может по трем причинам:

-либо он заканчивает свою деятельность;

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

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

Для полноты картины в модели нам необходимо ввести еще два состояния процессов**: рождение** и **закончил исполнение**

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

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

Операции над процессами и связанные с ними понятия. Набор операций.

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

--Создание процесса — завершение процесса;

--Приостановка процесса (перевод из состояния **исполнение** в состояние **готовность**) — запуск процесса (перевод из состояния **готовность** в состояние **исполнение**);

--Блокирование процесса (перевод из состояния **исполнение** в состояние **ожидание**) - разблокирование процесса (перевод из состояния **ожидание** в состояние **готовность**);

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

Операции создания и завершения процесса являются одноразовыми, так как применяются к процессу не более одного раза (некоторые системные процессы никогда не завершаются при работе вычислительной системы). Все остальные операции, связанные с изменением состояния процессов, будь то запуск или блокировка, как правило, являются многоразовыми.

Process Control Block (PCB) и контекст процесса.

Для того чтобы операционная система могла выполнять операции над процессами, каждый процесс представляется в ней некоторой структурой данных. Эта структура содержит информацию, специфическую для данного процесса:

-состояние, в котором находится процесс;

-программный счетчик процесса или, другими словами, адрес команды, которая должна быть выполнена для него следующей;

-содержимое регистров процессора;

-данные, необходимые для планирования использования процессора и управления памятью (приоритет процесса, размер и расположение адресного пространства и т. д.);

-учетные данные (идентификационный номер процесса, какой пользователь инициировал его работу, общее время использования процессора данным процессом и т. д.);

-информацию об устройствах ввода-вывода, связанных с процессом (например, какие устройства закреплены за процессом, таблицу открытых файлов);

Конкретный ее состав и строение зависят, конечно, от конкретной операционной системы. Во многих операционных системах информация, характеризующая процесс, хранится не в одной, а в нескольких связанных структурах данных. Эти структуры могут иметь различные наименования, содержать дополнительную информацию или, наоборот, лишь часть описанной информации.

Для простоты изложения будем считать, что она хранится в одной структуре данных. Мы будем называть ее PCB (Process Control Block) или **блоком управления процессом**. Блок управления процессом является моделью процесса для операционной системы. Любая операция, производимая операционной системой над процессом, вызывает определенные изменения в PCB. В рамках принятой модели состояний процессов содержимое PCB между операциями остается постоянным.

Информацию, для хранения которой предназначен блок управления процессом, удобно для дальнейшего изложения разделить на две части. Содержимое всех регистров процессора (включая значение программного счетчика) будем называть *регистровым контекстом процесса*, а все остальное – *системным контекстом* процесса. Знания регистрового и системного контекстов процесса достаточно для того, чтобы управлять его поведением в операционной системе, совершая над ним операции. С точки зрения пользователя, наоборот, наибольший интерес представляет содержимое адресного пространства процесса, возможно наряду с регистровым контекстом, определяющее последовательность преобразования данных и полученные результаты. Код и данные, находящиеся в адресном пространстве процесса, будем называть его *пользовательским контекстом*. Совокупность регистрового, системного и пользовательского контекстов процесса для краткости принято называть просто контекстом процесса.

В любой момент времени процесс полностью характеризуется своим *контекстом*.

Одноразовые операции

Одноразовые операции приводят к изменению количества процессов, находящихся под управлением операционной системы, и всегда связаны с выделением или освобождением определенных ресурсов.

Любая операционная система, поддерживающая концепцию процессов, должна обладать средствами для их создания. В очень простых системах все процессы могут быть порождены на этапе старта системы. Более сложные операционные системы создают процессы динамически, по мере необходимости. Инициатором рождения нового процесса после старта операционной системы может выступить либо процесс пользователя, совершивший специальный системный вызов, либо сама операционная система, то есть, в конечном итоге, тоже некоторый процесс. Процесс, инициировавший создание нового процесса, принято называть процессом-родителем (parent process), а вновь созданный процесс - процессом-ребенком (child process). Процессы-дети могут, в свою очередь, порождать новых детей и т. д., образуя, в общем случае, внутри системы набор генеалогических деревьев процессов - генеалогический лес. Пример генеалогического леса изображен на рисунке 4.

При рождении процесса система заводит новый PCB с состоянием процесса рождение и начинает его заполнение. Новый процесс получает свой собственный уникальный идентификационный номер. Поскольку для хранения идентификационного номера процесса в операционной системе отводится ограниченное количество бит, то для соблюдения уникальности номеров количество одновременно присутствующих в ней процессов должно быть ограничено. После завершения какого-либо процесса его

освободившийся идентификационный номер может быть повторно использован для другого процесса.

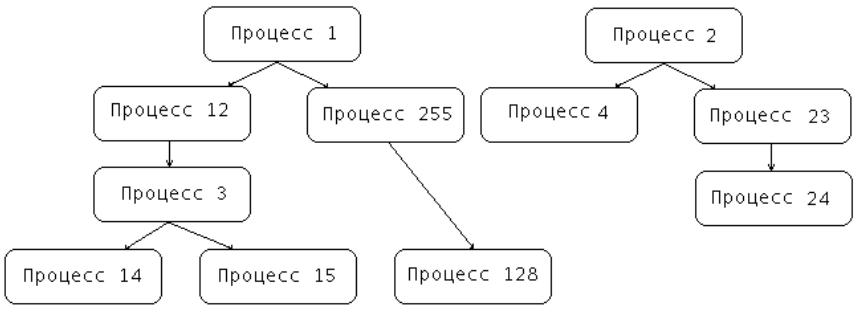

Рис.4. Упрощенный генеалогический лес процессов.
Стрелочка означает отношение родитель-ребенок.

Обычно для выполнения своих функций процесс-ребенок требует определенных ресурсов: памяти, файлов, устройств ввода-вывода и т. д. Существует два подхода к их выделению. Новый процесс может получить в свое пользование некоторую часть родительских ресурсов, возможно, разделяя с процессом-родителем и другими процессами-детьми права на них, или может получить свои ресурсы непосредственно от операционной системы.

После наделения процесса-ребенка ресурсами необходимо занести в его адресное пространство программный код, значения данных, установить программный счетчик. Здесь также возможны два решения. В первом случае процесс-ребенок становится дубликатом процесса-родителя по регистровому и пользовательскому контекстам, при этом должен существовать способ определения кто для кого из процессов-двойников является родителем. Во втором случае процесс-ребенок загружается новой программой из какого-либо файла. Операционная система UNIX разрешает порождение процесса только первым способом; для запуска новой программы необходимо сначала создать копию процесса-родителя, а затем процесс-ребенок должен заменить свой пользовательский контекст с помощью специального системного вызова. Операционные системы VAX/VMS и WINDOWS NT допускают только второе решение.

Порождение нового процесса как дубликата процесса-родителя приводит к возможности существования программ (т. е. исполняемых файлов), для работы которых организуется более одного процесса. Возможность замены пользовательского контекста процесса по ходу его работы (т. е. загрузки для исполнения новой программы) приводит к тому, что в рамках одного и того же процесса могут быть последовательно выполнены несколько различных программ.

После того как процесс наделен содержанием, в PCB дописывается оставшаяся информация и состояние нового процесса изменяется на **готовность**

Мы не будем в деталях останавливаться на причинах, которые могут привести к окончанию жизненного цикла процесса. После того, как процесс завершил свою работу, операционная система переводит его в состояние **закончил исполнение** и освобождает

все ассоциированные с ним ресурсы, делая соответствующие записи в блоке управления процессом. При этом сам PCB не уничтожается, а остается в системе еще некоторое время. Это связано с тем, что процесс-родитель после завершения процесса-ребенка может запросить операционную систему о причине произошедшей смерти порожденного им процесса и/или статистическую информацию об его работе. Подобная информация сохраняется в PCB мертвого процесса до запроса процесса-родителя или до конца его деятельности, после чего все следы умершего процесса окончательно исчезают из системы. В операционной системе UNIX процессы, находящиеся в состоянии **закончил исполнение**, принято называть процессами зомби.

В ряде операционных систем гибель процесса-родителя приводит к завершению работы всех его детей. В других операционных системах (например, в UNIX) процессы-дети продолжают свое существование и после окончания работы процесса-родителя. При этом возникает необходимость изменения информации в PCB процессов-детей о породившем их процессе для того, чтобы генеалогический лес процессов оставался целостным.

Многоразовые операции

Многоразовые операции не приводят к изменению количества процессов в операционной системе и не обязаны быть связанными с выделением или освобождением ресурсов.

Запуск процесса. Из числа процессов, находящихся в состоянии **готовность**, операционная система выбирает один процесс для последующего исполнения. Критерии и алгоритмы такого выбора зависят от системы **планирования процессов**. Для избранного процесса операционная система обеспечивает наличие в оперативной памяти информации, необходимой для его дальнейшего выполнения. Далее состояние процесса изменяется на **исполнение**, восстанавливаются значения регистров для данного процесса, и управление передается команде, на которую указывает счетчик команд процесса. Все данные, необходимые для этого восстановления контекста, извлекаются из PCB процесса, над которым совершается операция.

Приостановка процесса. Работа процесса, находящегося в состоянии **исполнение**, приостанавливается в результате какого-либо прерывания. Процессор автоматически сохраняет счетчик команд и, возможно, один или несколько регистров в стеке исполняемого процесса и передает управление по специальному адресу обработки данного прерывания. На этом деятельность hardware по обработке прерывания завершается. По указанному адресу обычно располагается одна из частей операционной системы. Она сохраняет динамическую часть системного и регистрового контекстов процесса в его PCB, переводит процесс в состояние **готовность** и приступает к обработке прерывания, то есть к выполнению определенных действий, связанных с возникшим прерыванием.

Блокирование процесса. Процесс блокируется, когда он не может продолжать свою работу, не дождавшись возникновения какого-либо события в вычислительной системе.

Для этого он обращается к операционной системе с помощью определенного системного вызова. Операционная система обрабатывает системный вызов (инициализирует операцию ввода-вывода, добавляет процесс в очередь процессов, дожидающихся освобождения устройства или возникновения события, и т. д.) и, при необходимости, сохранив необходимую часть контекста процесса в его PCB, переводит процесс из состояния **исполнение** в состояние **ожидание**.

Разблокирование процесса. После возникновения в системе какого-либо события, операционной системе нужно точно определить какое именно событие произошло. Затем операционная система проверяет: находился ли некоторый процесс в состоянии **ожидание** для данного события и, если находился, переводит его в состояние **готовность**, выполняя необходимые действия, связанные с наступлением события (инициализация операции ввода-вывода для очередного ожидающего процесса и т. п.).

Переключение контекста

До сих пор мы рассматривали операции над процессами изолированно, независимо друг от друга. В действительности же деятельность мультипрограммной операционной системы состоит из цепочек операций, выполняемых над различными процессами, и сопровождается переключением процессора с одного процесса на другой.

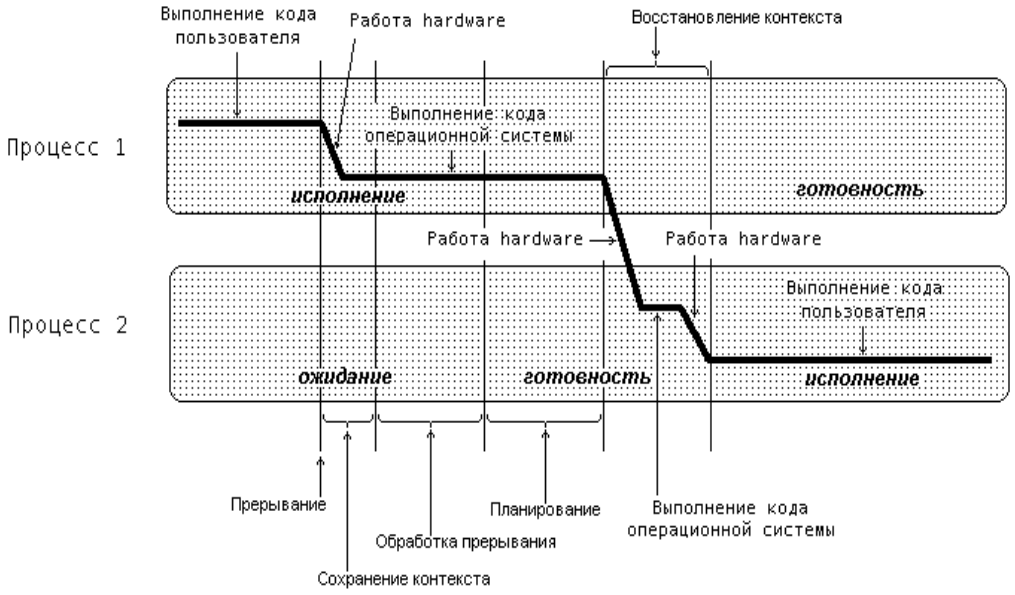

Рис.5. Выполнение операции разблокирования процесса.

Давайте для примера упрощенно рассмотрим, как в реальности может проистекать операция разблокирования процесса, ожидающего ввода-вывода (рис. 5). При исполнении процессором некоторого процесса (на рисунке 5 - процесс 1) возникает прерывание от устройства ввода-вывода, сигнализирующее об окончании операций на устройстве. Над выполняющимся процессом производится операция приостановка. Далее, операционная система разблокирует процесс, инициировавший запрос на ввод-вывод (на рисунке 5- процесс 2), и осуществляет запуск приостановленного или нового процесса, выбранного при выполнении планирования (на рисунке 5 был выбран разблокированный процесс). Как видим, в результате обработки информации об

окончании операции ввода-вывода возможна смена процесса, находящегося в состоянии исполнение.

Для корректного переключения процессора с одного процесса на другой необходимо сохранить контекст исполнявшегося процесса и восстановить контекст процесса, на который будет переключен процессор. Такая процедура сохранения/восстановления работоспособности процессов называется переключением контекста. Время, затраченное на переключение контекста, не используется вычислительной системой для совершения полезной работы и представляет собой накладные расходы, снижающие производительность системы. Оно меняется от машины к машине и обычно находится в диапазоне от 1 до 1000 микросекунд.

Нити исполнения (потоки)

В традиционных операционных системах у каждого процесса есть адресное пространство и единственный поток управления. Фактически это почти что определение процесса. Тем не менее нередко возникают ситуации, когда неплохо было бы иметь несколько потоков управления в одном и том же адресном пространстве, выполняемых квазипараллельно, как будто они являются чуть ли не обособленными процессами (за исключением общего адресного пространства).

Введем новую абстракцию внутри понятия “процесс” – нить исполнения или просто нить (поток) (в англоязычной литературе используется термин thread). Нити процесса разделяют его программный код, глобальные переменные и системные ресурсы, но каждая нить имеет свой собственный программный счетчик, свое содержимое регистров и свой собственный стек. Теперь процесс представляется как совокупность взаимодействующих нитей и выделенных ему ресурсов (рис.6).

Рис.6 Несколько процессов, у каждого процесса несколько потоков, у каждого потока несколько подпотоков.

Процесс, содержащий всего одну нить исполнения, идентичен процессу в том смысле, который мы употребляли ранее. Для таких процессов мы в дальнейшем будем использовать термин “традиционный процесс”. Иногда нити называют облегченными процессами или мини-процессами, так как во многих отношениях они подобны традиционным процессам. Нити, как и процессы, могут порождать нити-потомки, правда, только внутри своего процесса, и переходить из состояния в состояние.

Состояния нитей аналогичны состояниям традиционных процессов. Из состояния рождение процесс приходит содержащим всего одну нить исполнения. Другие нити процесса будут являться потомками этой нити прародительницы. Мы можем считать, что процесс находится в состоянии готовность, если хотя бы одна из его нитей находится в состоянии готовность и ни одна из нитей не находится в состоянии исполнение. Мы можем считать, что процесс находится в состоянии исполнение, если одна из его нитей находится в состоянии исполнение. Процесс будет находиться в состоянии ожидание, если все его нити находятся в состоянии ожидание. Наконец, процесс находится в состоянии завершил исполнение, если все его нити находятся в состоянии завершили исполнение. Пока одна нить процесса заблокирована, другая нить того же процесса может выполняться.

Зачем нам нужна какая-то разновидность процесса внутри самого процесса? Необходимость в подобных мини-процессах, называемых потоками, обусловливается целым рядом причин. Основная причина использования потоков заключается в том, что во многих приложениях одновременно происходит несколько действий, часть которых может периодически быть заблокированной. Модель программирования упрощается за счет разделения такого приложения на несколько последовательных потоков, выполняемых в квазипараллельном режиме. Вместо того чтобы думать о прерываниях, таймерах и контекстных переключателях, мы можем думать о параллельных процессах. Но только теперь, рассматривая потоки, мы добавляем новый элемент: возможность использования параллельными процессами единого адресного пространства и всеми имеющимися данными. Эта возможность играет весьма важную роль для тех приложений, которым не подходит использование нескольких процессов (с их раздельными адресными пространствами).

Второй аргумент в пользу потоков - легкость (то есть быстрота) их создания и ликвидации по сравнению с более «тяжеловесными» процессами. Во многих системах создание потоков осуществляется в 10-100 раз быстрее, чем создание процессов. Это свойство особенно пригодится, когда потребуется быстро и динамично изменять количество потоков.

Третий аргумент - когда потоки работают в рамках одного центрального процессора, они не приносят никакого прироста производительности, но когда проводятся значительные вычисления, а также значительная часть времени тратится на ожидание ввода-вывода, наличие потоков позволяет этим действиям перекрываться по времени, ускоряя работу приложения.

И наконец, потоки весьма полезны для систем, имеющих несколько центральных процессоров, где есть реальная возможность параллельных вычислений.

Понять, в чем состоит польза от применения потоков, проще всего на конкретных примерах. Рассмотрим в качестве первого примера текстовый процессор. Обычно эти программы отображают создаваемый документ на экране в том виде, в каком он будет выводиться на печать. В частности, все концы строк и концы страниц находятся именно там, где они в результате и появятся на бумаге, чтобы пользователь мог при

необходимости их проверить и подправить (например, убрать на странице начальные и конечные висячие строки, имеющие неэстетичный вид). Предположим, что пользователь пишет какую-то книгу. С авторской точки зрения проще всего всю книгу иметь в одном файле, облегчая поиск тем, осуществление глобальных замен и т. д. Теперь представим себе, что происходит, когда пользователь вдруг удаляет одно предложение на первой странице 800-страничного документа. Теперь, проверив внесенные изменения, он хочет внести еще одну поправку на 600-й странице и набирает команду, предписывающую текстовому процессору перейти на эту страницу (возможно, за счет поиска фразы, которая только там и встречается). Теперь текстовый процессор вынужден немедленно переформатировать всю книгу вплоть до 600-й страницы, поскольку он не знает, какой будет первая строка на 600-й странице, пока не проведет обработку всех предыдущих страниц. Перед отображением 600-й страницы может произойти существенная задержка, вызывающая недовольство пользователя.

И здесь на помощь могут прийти потоки. Предположим, что текстовый процессор написан как двухпоточная программа. Один из потоков взаимодействует с пользователем, а другой занимается переформатированием в фоновом режиме. Как только предложение с первой страницы будет удалено, поток, отвечающий за взаимодействие с пользователем, приказывает потоку, отвечающему за формат, переформатировать всю книгу. Пока взаимодействующий поток продолжает отслеживать события клавиатуры и мыши, реагируя на простые команды вроде прокрутки первой страницы, второй поток с большой скоростью проводит вычисления. Если немного повезет, то переформатирование закончится как раз перед тем, как пользователь запросит просмотр 600-й страницы, которая тут же сможет быть отображена. Ну, раз уж начали, то почему бы не добавить и третий поток? Многие текстовые процессоры обладают свойством автоматического сохранения всего файла на диск каждые несколько минут, чтобы уберечь пользователя от утраты его дневной работы в случает программных или системных сбоев или отключения электропитания. Третий поток может заниматься созданием резервных копий на диске, не мешая первым двум.

6. ПЛАНИРОВАНИЕ ПРОЦЕССОВ

Уровни планирования

В рассматриваемой модели существуют два вида планирования в вычислительных системах: **планировании заданий** и **планировании** использования **процессора**. Планирование заданий появилось в пакетных системах после того, как для хранения сформированных пакетов заданий начали использоваться магнитные диски. Изменяя порядок загрузки заданий в вычислительную систему, можно повысить эффективность ее использования. Процедуру выбора очередного задания для загрузки в машину, т. е. для порождения соответствующего процесса, мы и назвали планированием заданий. Планирование использования процессора впервые возникает в мультипрограммных вычислительных системах, где в состоянии **готовность** могут одновременно находиться

несколько процессов. Именно для процедуры выбора из них одного процесса, который получит процессор в свое распоряжение, т.е. будет переведен в состояние **исполнение**, мы использовали это словосочетание.

Планирование заданий выступает в качестве долгосрочного планирования процессов. Оно отвечает за порождение новых процессов в системе, определяя ее *степень мультипрограммирования*, т. е. количество процессов, одновременно находящихся в ней. Если степень мультипрограммирования системы поддерживается постоянной, т. е. среднее количество процессов в компьютере не меняется, то новые процессы могут появляться только после завершения ранее загруженных. Поэтому долгосрочное планирование осуществляется достаточно редко, между появлением новых процессов могут проходить минуты и даже десятки минут. В некоторых операционных системах долгосрочное планирование сведено к минимуму или совсем отсутствует. Так, например, во многих интерактивных системах разделения времени порождение процесса происходит сразу после появления соответствующего запроса.

Планирование использования процессора выступает в качестве краткосрочного планирования процессов. Оно проводится, к примеру, при обращении исполняющегося процесса к устройствам ввода-вывода или просто по завершении определенного интервала времени. Поэтому краткосрочное планирование осуществляется весьма часто, как правило, не реже одного раза в 100 миллисекунд. Выбор нового процесса для исполнения оказывает влияние на функционирование системы до наступления очередного аналогичного события, т. е. в течение короткого промежутка времени, что и обусловило название этого уровня планирования — краткосрочное.

В некоторых вычислительных системах бывает выгодно для повышения их производительности временно удалить какой-либо частично выполнившийся процесс из оперативной памяти на диск, а позже вернуть его обратно для дальнейшего выполнения. Такая процедура в англоязычной литературе получила название ***swapping***, что можно перевести на русский язык как перекачка, хотя в профессиональной литературе оно употребляется без перевода — свопинг. Когда и какой из процессов нужно перекачать на диск и вернуть обратно, решается дополнительным промежуточным уровнем планирования процессов — среднесрочным.

Критерии планирования и требования к алгоритмам

Для каждого уровня планирования процессов можно предложить много различных алгоритмов. Выбор конкретного алгоритма определяется классом задач, решаемых вычислительной системой, и целями, которых мы хотим достичь, используя планирование. К числу таких целей можно отнести:

--Справедливость: гарантировать каждому заданию или процессу определенную часть времени использования процессора в компьютерной системе, стараясь не допустить возникновения ситуации, когда процесс одного пользователя постоянно занимает процессор, в то время как процесс другого пользователя фактически не приступал к выполнению.

--Эффективность: постараться занять процессор на все 100% рабочего времени, не позволяя ему простаивать в ожидании процессов готовых к исполнению. В реальных вычислительных системах загрузка процессора колеблется от 40 до 90 процентов.

--Сокращение полного времени выполнения (turnaround time): обеспечить минимальное время между стартом процесса или постановкой задания в очередь для загрузки и его завершением.

--Сокращение времени ожидания (waiting time): минимизировать время, которое проводят процессы в состоянии готовность и задания в очереди для загрузки.

--Сокращение времени отклика (response time): минимизировать время, которое требуется процессу в интерактивных системах для ответа на запрос пользователя.

Независимо от поставленных целей планирования желательно также, чтобы алгоритмы обладали следующими свойствами:

--Были предсказуемыми. Одно и то же задание должно выполняться приблизительно за одно и то же время.

--Имели минимальные накладные расходы, связанные с их работой. Если на каждые 100 миллисекунд, выделенных процессу для использования процессора, будет приходиться 200 миллисекунд на определение того, какой именно процесс получит процессор в свое распоряжение, и на переключение контекста, то такой алгоритм, очевидно, использовать не стоит.

--Равномерно загружали ресурсы вычислительной системы, отдавая предпочтение тем процессам, которые будут занимать малоиспользуемые ресурсы.

--Обладали масштабируемостью, т. е. не сразу теряли работоспособность при увеличении нагрузки. Например, рост количества процессов в системе в два раза не должен приводить к увеличению полного времени выполнения процессов на порядок.

Многие из приведенных выше целей и свойств являются противоречивыми. Улучшая работу алгоритма с точки зрения одного критерия, мы ухудшаем ее с точки зрения другого. Приспосабливая алгоритм под один класс задач, мы тем самым дискриминируем задачи другого класса.

Параметры планирования

Все параметры планирования можно разбить на две большие группы: статические параметры и динамические параметры. Статические параметры не изменяются в ходе функционирования вычислительной системы, динамические же, напротив, подвержены постоянным изменениям.

К статическим параметрам вычислительной системы можно отнести предельные значения ее ресурсов (размер оперативной памяти, максимальное количество памяти на диске для осуществления свопинга, количество подключенных устройств ввода-вывода и т. п.).

К статическим параметрам процессов относятся характеристики, как правило, присущие заданиям уже на этапе загрузки:

--Каким пользователем запущен процесс или сформировано задание.

--Насколько важной является поставленная задача, т. е. каков приоритет ее выполнения.

--Сколько процессорного времени запрошено пользователем для решения задачи.

--Каково соотношение процессорного времени и времени, необходимого для осуществления операций ввода-вывода.

--Какие ресурсы вычислительной системы (оперативная память, устройства ввода-вывода, специальные библиотеки и системные программы и т. д.) и в каком количестве необходимы заданию.

Динамические параметры системы описывают количество свободных ресурсов в текущий момент времени.

Алгоритмы долгосрочного планирования используют в своей работе статические и динамические параметры вычислительной системы и статические параметры процессов (динамические параметры процессов на этапе загрузки заданий еще не известны). Алгоритмы краткосрочного и среднесрочного планирования дополнительно учитывают и динамические характеристики процессов.

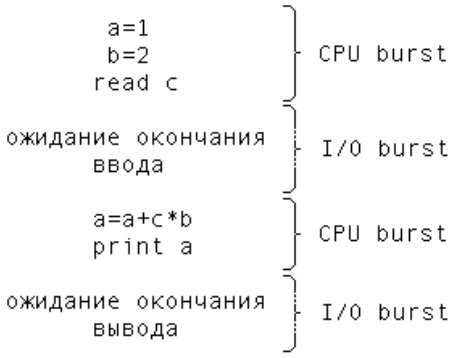

Рис. 7. Фрагмент деятельности процесса с выделением промежутков непрерывного использования процессора и ожидания ввода-вывода.

Для краткосрочного планирования нам понадобится ввести еще два динамических параметра. Деятельность любого процесса можно представить как последовательность циклов использования процессора и ожидания завершения операций ввода-вывода. Промежуток времени непрерывного использования процессора носит на английском языке название **CPU burst**, а промежуток времени непрерывного ожидания ввода-вывода – **I/O burst**. На рисунке 7 показан фрагмент деятельности некоторого процесса на псевдоязыке программирования с выделением указанных промежутков. Для краткости изложения мы будем использовать термины CPU burst и I/O burst без перевода.

Вытесняющее и невытесняющее планирование

Процесс планирования осуществляется частью операционной системы, называемой ***планировщиком***. Планировщик может принимать решения о выборе для исполнения нового процесса, из числа находящихся в состоянии готовность, в следующих четырех случаях:

1.Когда процесс переводится из состояния исполнение в состояние завершение.

2.Когда процесс переводится из состояния исполнение в состояние ожидание.

3.Когда процесс переводится из состояния исполнение в состояние готовность (например, после прерывания от таймера).

4.Когда процесс переводится из состояния ожидание в состояние готовность (завершилась операция ввода-вывода или произошло другое событие).

В случаях 1 и 2 процесс, находившийся в состоянии исполнение, не может дальше исполняться, и для выполнения всегда необходимо выбрать новый процесс. В случаях 3 и 4 планирование может не проводиться, процесс, который исполнялся до прерывания, может продолжать свое выполнение после обработки прерывания. Если планирование осуществляется только в случаях 1 и 2, говорят, что имеет место невытесняющее (nonpreemptive) планирование. В противном случае говорят о вытесняющем (preemptive) планировании. Термин “вытесняющее планирование” возник потому, что исполняющийся процесс помимо своей воли может быть вытеснен из состояния исполнение другим процессом.

Невытесняющее планирование используется, например, в MS Windows 3.1 и ОС Apple Macintosh. При таком режиме планирования процесс занимает столько процессорного времени, сколько ему необходимо. При этом переключение процессов возникает только при желании самого исполняющегося процесса передать управление (для ожидания завершения операции ввода-вывода или по окончании работы). Этот метод планирования относительно просто реализуем и достаточно эффективен, так как позволяет использовать большую часть процессорного времени на работу самих процессов и до минимума сократить затраты на переключение контекста. Однако при невытесняющем планировании возникает проблема возможности полного захвата процессора одним процессом, который вследствие каких-либо причин (например, из-за ошибки в программе) зацикливается и не может передать управление другому процессу. В такой ситуации спасает только перезагрузка всей вычислительной системы.

Вытесняющее планирование обычно используется в системах разделения времени. В этом режиме планирования процесс может быть приостановлен в любой момент своего исполнения. Операционная система устанавливает специальный таймер для генерации сигнала прерывания по истечении некоторого интервала времени — кванта. После прерывания процессор передается в распоряжение следующего процесса. Временные прерывания помогают гарантировать приемлемые времена отклика процессов для пользователей, работающих в диалоговом режиме, и предотвращают “зависание” компьютерной системы из-за зацикливания какой-либо программы.

Алгоритмы планирования

First-Come, First-Served (FCFS)

Простейшим алгоритмом планирования является алгоритм FCFS (First Come, First Served-первым пришел, первым обслужен). Представим себе, что процессы, находящиеся

в состоянии готовность, организованы в очередь. Когда процесс переходит в состояние готовность, он, а точнее ссылка на его PCB, помещается в конец этой очереди. Выбор нового процесса для исполнения осуществляется из начала очереди с удалением оттуда ссылки на его PCB. Очередь подобного типа имеет в программировании специальное наименование FIFO — сокращение от First In, First Out (первым вошел, первым вышел).

Такой алгоритм выбора процесса осуществляет невытесняющее планирование. Процесс, получивший в свое распоряжение процессор, занимает его до истечения своего текущего CPU burst. После этого для выполнения выбирается новый процесс из начала очереди.

Процесс¤	p0¤	p1¤	p2¤	¤
Продолжительность очередного CPU burst¤	13¤	4¤	1¤	¤

Рис.8 Очередь процесса со временами CPU burst

Преимуществом алгоритма FCFS является легкость его реализации, в то же время он имеет и много недостатков. Рассмотрим следующий пример. Пусть в состоянии готовность находятся три процесса p0, p1 и p2, для которых известны времена их очередных CPU burst. Эти времена приведены в таблице на рис.8 в некоторых условных единицах. Для простоты будем полагать, что вся деятельность процессов ограничивается использованием только одного промежутка CPU burst, что процессы не совершают операций ввода-вывода, и что время переключения контекста пренебрежимо мало.

Если процессы расположены в очереди процессов готовых к исполнению в порядке p0, p1, p2, то картина их выполнения выглядит так, как показано на рисунке 9. Первым для выполнения выбирается процесс p0, который получает процессор на все время своего CPU burst, т. е. на 13 единиц времени.

Рис.9 Последовательность исполнения процессов

После его окончания в состояние исполнение переводится процесс p1, занимая процессор на 4 единицы времени. И, наконец, возможность работать получает процесс p2. Время ожидания для процесса p0 составляет 0 единиц времени, для процесса p1 — 13 единиц, для процесса p2 — 13 + 4 = 17 единиц. Таким образом, среднее время ожидания в этом случае — (0 + 13 + 17)/3 = 10 единиц времени. Полное время выполнения для процесса p0 составляет 13 единиц времени, для процесса p1 — 13 + 4 = 17 единиц, для процесса p2 — 13 + 4 + 1 = 18 единиц. Среднее полное время выполнения оказывается равным (13 + 17 + 18)/3 = 16 единицам времени. Если те же самые процессы

расположены в порядке p2, p1, p0, то картина их выполнения будет соответствовать рисунку 10

Рис.10 Последовательность исполнения процессов.

Время ожидания для процесса p0 равняется 5 единицам времени, для процесса p1 — 1 единице, для процесса p2 — 0 единиц. Среднее время ожидания составит (5 + 1 + 0)/3 = 2 единицы времени. Это в 5 (!) раз меньше, чем в предыдущем случае. Полное время выполнения для процесса p0 получается равным 18 единицам времени, для процесса p1 — 5 единицам, для процесса p2 — 1 единице. Среднее полное время выполнения составляет (18 + 5 + 1)/3 = 6 единиц времени, что почти в 2,7 раза меньше чем при первой расстановке процессов.

Как видим, среднее время ожидания и среднее полное время выполнения для этого алгоритма существенно зависят от порядка расположения процессов в очереди. Если у нас есть процесс с длительным CPU burst, то короткие процессы, перешедшие в состояние готовность после длительного процесса, будут очень долго ждать начала своего выполнения. Поэтому алгоритм FCFS практически неприменим для систем разделения времени. Слишком большим получается среднее время отклика в интерактивных процессах.

Round Robin (RR)

Модификацией алгоритма FCFS является алгоритм, получивший название Round Robin (Round Robin – это вид детской карусели в США) или сокращенно RR. По сути дела это алгоритм **FCFS**, только реализованный в режиме вытесняющего планирования. Можно представить себе все множество готовых процессов организованным циклически — процессы сидят на карусели. Карусель вращается так, что каждый процесс находится около процессора небольшой фиксированный квант времени, обычно 10 - 100 миллисекунд (см. рисунок 101). Пока процесс находится рядом с процессором, он получает процессор в свое распоряжение и может исполняться.

Реализуется такой алгоритм так же, как и предыдущий, с помощью организации процессов, находящихся в состоянии готовность, в очередь FIFO. Планировщик выбирает для очередного исполнения процесс, расположенный в начале очереди, и устанавливает таймер для генерации прерывания по истечении определенного кванта времени. При выполнении процесса возможны два варианта:

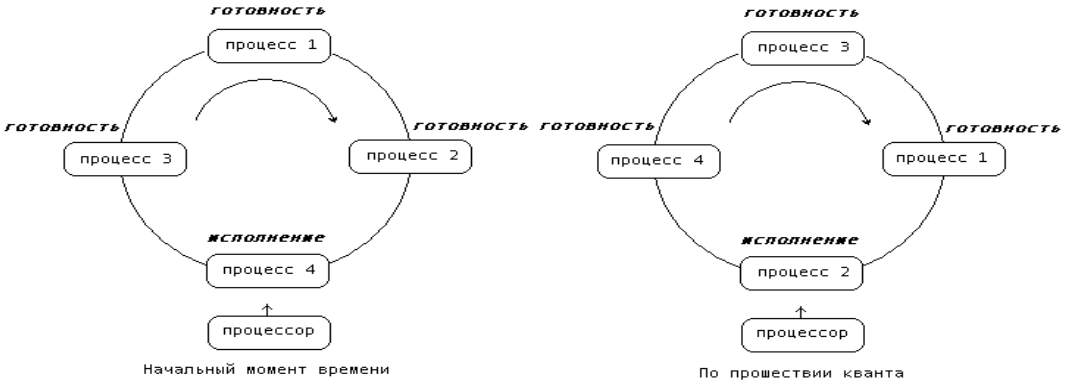

Рис.11 Циклическая организация очереди процессов.

1. Время непрерывного использования процессора, требующееся процессу, (остаток текущего CPU burst) меньше или равно продолжительности кванта времени. Тогда процесс по своей воле освобождает процессор до истечения кванта времени, на исполнение выбирается новый процесс из начала очереди и таймер начинает отсчет кванта заново.
2. Продолжительность остатка текущего CPU burst процесса больше, чем квант времени. Тогда по истечении этого кванта процесс прерывается таймером и помещается в конец очереди процессов готовых к исполнению, а процессор выделяется для использования процессу, находящемуся в ее начале.

Рассмотрим предыдущий пример с порядком процессов p0, p1, p2 и величиной кванта времени равной 4. Выполнение этих процессов иллюстрируется таблицей на рис.12. Обозначение “И” используется в ней для процесса, находящегося в состоянии исполнение, обозначение “Г” — для процессов в состоянии готовность, пустые ячейки соответствуют завершившимся процессам. Состояния процессов показаны на протяжении соответствующей единицы времени, т. е. колонка с номером 1 соответствует промежутку времени от 0 до 1.

время	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
p_0	И	И	И	И	Г	Г	Г	Г	Г	И	И	И	И	И	И	И	И	И
p_1	Г	Г	Г	Г	И	И	И	И										
p_2	Г	Г	Г	Г	Г	Г	Г	Г	И									

Рис.12 Алгоритм планирования Round Robin с квантом времени равным 4.

Первым для исполнения выбирается процесс p0. Продолжительность его CPU burst больше, чем величина кванта времени, и поэтому процесс исполняется до истечения кванта, т. е. в течение 4 единиц времени. После этого он помещается в конец очереди готовых к исполнению процессов, которая принимает вид p1, p2, p0. Следующим начинает выполняться процесс p1. Время его исполнения совпадает с величиной выделенного кванта, поэтому процесс работает до своего завершения. Теперь очередь процессов в состоянии готовность состоит из двух процессов p2, p0. Процессор выделяется процессу p2. Он завершается до истечения отпущенного ему процессорного

времени, и очередные кванты отмеряются процессу p0 — единственному, не закончившему к этому моменту свою работу. Время ожидания для процесса p0 (количество символов “Г” в соответствующей строке) составляет 5 единиц времени, для процесса p1 — 4 единицы времени, для процесса p2 — 8 единиц времени. Таким образом, среднее время ожидания для этого алгоритма получается равным (5 + 4 + 8)/3 = 5,6(6) единицы времени. Полное время выполнения для процесса p0 (количество непустых столбцов в соответствующей строке) составляет 18 единиц времени, для процесса p1 — 8 единиц, для процесса p2 — 9 единиц. Среднее полное время выполнения оказывается равным (18 + 8 + 9)/3 = 11,6(6) единицам времени.

Легко видеть, что среднее время ожидания и среднее полное время выполнения для обратного порядка процессов не отличаются от соответствующих времен для алгоритма FCFS и составляют 2 и 6 единиц времени соответственно.

На производительность алгоритма RR сильно влияет величина кванта времени. Рассмотрим тот же самый пример с порядком процессов p0, p1, p2 для величины кванта времени равной 1 (см. таблицу на рис.13). Время ожидания для процесса p0 составит 5 единиц времени, для процесса p1 — тоже 5 единиц, для процесса p2 — 2 единицы. В этом случае среднее время ожидания получается равным (5 + 5 + 2)/3 = 4 единицам времени. Среднее полное время исполнения составит (18 + 9 + 3)/3 = 10 единиц времени.

время	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
p_0	И	Г	Г	И	Г	И	Г	И	Г	И	И	И	И	И	И	И	И	И
p_1	Г	И	Г	Г	И	Г	И	Г	И									
p_2	Г	Г	И															

Рис.13 Алгоритм планирования Round Robin с квантом времени равным 1.

При очень больших величинах кванта времени, когда каждый процесс успевает завершить свой CPU burst до возникновения прерывания по времени, алгоритм RR вырождается в алгоритм FCFS. При очень малых величинах создается иллюзия того, что каждый из n процессов работает на своем собственном виртуальном процессоре с производительностью ~ 1/n от производительности реального процессора. Правда, это справедливо лишь при теоретическом анализе при условии пренебрежения временами переключения контекста процессов. В реальных условиях при слишком малой величине кванта времени и, соответственно, слишком частом переключении контекста, накладные расходы на переключение резко снижают производительность системы.

Shortest-Job-First (SJF)

При рассмотрении алгоритмов FCFS и RR мы видели, насколько существенным для них является порядок расположения процессов в очереди процессов готовых к исполнению. Если короткие задачи расположены в очереди ближе к ее началу, то общая производительность этих алгоритмов значительно возрастает. Если бы мы знали время следующих CPU burst для процессов, находящихся в состоянии готовность, то могли бы

выбрать для исполнения не процесс из начала очереди, а процесс с минимальной длительностью CPU burst. Если же таких процессов два или больше, то для выбора одного из них можно использовать уже известный нам алгоритм FCFS. Квантование времени при этом не применяется. Описанный алгоритм получил название “кратчайшая работа первой” или Shortest Job First (SJF).

SJF алгоритм краткосрочного планирования может быть как вытесняющим, так и невытесняющим. При невытесняющем SJF планировании процессор предоставляется избранному процессу на все требующееся ему время, независимо от событий происходящих в вычислительной системе. При вытесняющем SJF планировании учитывается появление новых процессов в очереди готовых к исполнению (из числа вновь родившихся или разблокированных) во время работы выбранного процесса. Если CPU burst нового процесса меньше, чем остаток CPU burst у исполняющегося, то исполняющийся процесс вытесняется новым.

Рассмотрим пример работы невытесняющего алгоритма SJF. Пусть в состоянии готовность находятся четыре процесса p0, p1, p2 и p3, для которых известны времена их очередных CPU burst. Эти времена приведены в таблице на рис.14. Как и прежде, будем полагать, что вся деятельность процессов ограничивается использованием только одного промежутка CPU burst, что процессы не совершают операций ввода-вывода, и что время переключения контекста пренебрежимо мало.

Процесс	p_0	p_1	p_2	p_3
Продолжительность очередного CPU burst	5	3	7	1

Рис.14 Очередь процессов

При использовании невытесняющего алгоритма SJF первым для исполнения будет выбран процесс p3, имеющий наименьшее значение очередного CPU burst. После его завершения для исполнения выбирается процесс p1, затем p0 и, наконец, p2. Вся эта картина изображена в таблице на рис.15.

время	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
p_0	Г	Г	Г	Г	И	И	И	И	И							
p_1	Г	И	И	И												
p_2	Г	Г	Г	Г	Г	Г	Г	Г	Г	И	И	И	И	И	И	И
p_3	И															

Рис.15 Алгоритм Shortest-Job-First с невытесняющим планированием

Как видим, среднее время ожидания для алгоритма SJF составляет (4 + 1 + 9 + 0)/4 = 3,5 единицы времени. Легко посчитать, что для алгоритма FCFS при порядке процессов p0, p1, p2, p3 эта величина будет равняться (0 + 5 + 8 + 15)/4 = 7 единицам времени, т. е. будет в 2 раза больше, чем для алгоритма SJF. Можно показать, что для заданного набора процессов (если в очереди не появляются новые процессы) алгоритм SJF является оптимальным с точки зрения минимизации среднего времени ожидания среди класса всех невытесняющих алгоритмов.

Для рассмотрения примера **вытесняющего** SJF планирования мы возьмем ряд процессов p0, p1, p2 и p3с различными временами CPU burst и различными моментами их появления в очереди процессов готовых к исполнению (см. таблицу на рис.16).

Процесс	Время появления в очереди	Продолжительность очередного CPU burst
p_0	0	6
p_1	2	2
p_2	6	7
p_3	0	5

Рис.16 Алгоритм Shortest-Job-First с вытесняющим планированием

В начальный момент времени в состоянии готовность находятся только два процесса p0 и p4. Меньшее время очередного CPU burst оказывается у процесса p3, поэтому он и выбирается для исполнения (см. таблицу на рис.17.).

время	1	2	3	4	5	6	7	8	9	10
p_0	Г	Г	Г	Г	Г	Г	Г	И	И	И
p_1			И	И						
p_2							Г	Г	Г	Г
p_3	И	И	Г	Г	И	И	И			

время	11	12	13	14	15	16	17	18	19	20
p_0	И	И	И							
p_1										
p_2										
p_3	Г	Г	Г	И	И	И	И	И	И	И

Рис.17 Алгоритм Shortest-Job-First с вытесняющим планированием

По прошествии 2-х единиц времени в систему поступает процесс p1. Время его CPU burst меньше, чем остаток CPU burst у процесса p3, который вытесняется из состояния исполнение и переводится в состояние готовность. По прошествии еще 2-х единиц времени процесс p1 завершается, и для исполнения вновь выбирается процесс p3. В момент времени t = 6 в очереди процессов готовых к исполнению появляется процесс p2, но поскольку ему для работы нужно 7 единиц времени, а процессу p3 осталось трудиться всего 2 единицы времени, то процесс p3 остается в состоянии исполнение. После его завершения в момент времени t = 7 в очереди находятся процессы p0 и p2, из которых выбирается процесс p0. Наконец, последним получит возможность выполняться процесс p2.

Гарантированное планирование.

При одновременной работе **N** пользователей в вычислительной системе можно применить алгоритм планирования, который гарантирует, что каждый из пользователей будет иметь в своем распоряжении примерно **1/N** часть процессорного времени.

Пронумеруем всех пользователей от 1 до N. Для каждого пользователя с номером i введем две величины: **Ti** - время нахождения пользователя в системе (другими словами длительность сеанса его общения с машиной), и **ti** - суммарное процессорное время уже выделенное всем его потокам в течение сеанса. Справедливым для пользователя было бы получение **Ti/N** процессорного времени. Если $\mathbf{t_i << T_i / N}$, то i - й пользователь несправедливо обделен процессорным временем. Если же $\mathbf{t_i >> T_i/N}$, то система явно благоволит к пользователю с номером i.

Суть гарантированного планирования заключается в том, что для каждого пользовательского потока вычисляется коэффициент справедливости $(t_i\ N)/Ti$ и очередной квант времени предоставляется процессу с наименьшим коэффициентом.

К недостаткам этого алгоритма можно отнести невозможность предугадать поведение пользователей. Если некоторый пользователь отправится на пару часов пообедать и поспать, не прерывая сеанса работы, то по возвращении его процессы будут получать неоправданно много процессорного времени.

Приоритетное планирование

При приоритетном планировании каждому процессу присваивается определенное числовое значение — приоритет, в соответствии с которым ему выделяется процессор. Процессы с одинаковыми приоритетами планируются в порядке FCFS. Для алгоритма SJF в качестве такого приоритета выступает оценка продолжительности следующего CPU burst. Чем меньше значение этой оценки, тем более высокий приоритет имеет процесс. Для алгоритма гарантированного планирования приоритетом служит вычисленный коэффициент справедливости. Чем он меньше, тем больше приоритет у процесса.

Принципы назначения приоритетов могут опираться как на внутренние критерии вычислительной системы, так и на внешние по отношению к ней. Внутренние используют различные количественные и качественные характеристики процесса для вычисления его приоритета (например, определенные ограничения по времени использования процессора, требования к размеру памяти, число открытых файлов и используемых устройств ввода-вывода). Внешние критерии исходят из таких параметров, как важность процесса для достижения каких-либо целей, стоимость оплаченного процессорного времени и других политических факторов.

Планирование с использованием приоритетов может быть как вытесняющим, так и невытесняющим. При вытесняющем планировании процесс с более высоким приоритетом, появившийся в очереди готовых процессов, вытесняет исполняющийся процесс с более низким приоритетом. В случае невытесняющего планирования он просто становится в начало очереди готовых процессов. Давайте рассмотрим примеры использования различных режимов приоритетного планирования.

Пусть в очередь процессов, находящихся в состоянии готовность, поступают те же процессы, что и в примере для вытесняющего алгоритма SJF, только им дополнительно еще присвоены приоритеты (см. таблицу на рис.18). В вычислительных системах не существует определенного соглашения, какое значение приоритета - 1 или 4 считать более приоритетным. Во избежание путаницы, во всех наших примерах мы будем предполагать, что большее значение соответствует меньшему приоритету.

Процесс	Время появления в очереди	Продолжительность очередного CPU burst	Приоритет
p_0	0	6	4
p_1	2	2	3
p_2	6	7	2
p_3	0	5	1

Рис.18 Очередь процессов с различными приоритетами.

Как будут вести себя процессы при использовании невытесняющего приоритетного планирования? Первым для выполнения в момент времени t = 0 выбирается процесс p3, как обладающий наивысшим приоритетом. После его завершения в момент времени t = 5 в очереди процессов готовых к исполнению окажутся два процесса p0 и p1. Больший приоритет из них у процесса p1 он и начнет выполняться (см. таблицу на рис.18). Затем в момент времени t = 8 для исполнения будет избран процесс p2 и лишь потом процесс p0.

время	1	2	3	4	5	6	7	8	9	10
p_0	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г
p_1			Г	Г	Г	И	И			
p_2							Г	И	И	И
p_3	И	И	И	И	И					

время	11	12	13	14	15	16	17	18	19	20
p_0	Г	Г	Г	Г	И	И	И	И	И	И
p_1										
p_2	И	И	И	И						
p_3										

Рис.19 Исполнение процессов с различными приоритетами с невытесняющим планированием.

Иным будет предоставление процессора процессам в случае вытесняющего приоритетного планирования (см. таблицу на рис.20). Первым, как и в предыдущем случае, исполняться начнет процесс p3, а по его окончании процесс p1. Однако в момент времени t = 6 он будет вытеснен процессом p2 и продолжит свое выполнение только в момент времени t = 13. Последним, как и раньше будет исполнен процесс p0.

время	1	2	3	4	5	6	7	8	9	10
p_0	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г
p_1			Г	Г	Г	И	Г	Г	Г	Г
p_2							И	И	И	И
p_3	И	И	И	И	И					

время	11	12	13	14	15	16	17	18	19	20
p_0	Г	Г	Г	Г	И	И	И	И	И	И
p_1	Г	Г	Г	И						
p_2	И	И	И							
p_3										

Рис.20 Исполнение процессов с различными приоритетами с вытесняющим планированием.

В рассмотренном выше примере приоритеты процессов не изменялись с течением временем. Такие приоритеты принято называть статическими. Механизмы статической приоритетности легко реализовать, и они сопряжены с относительно небольшими

издержками на выбор наиболее приоритетного процесса. Однако статические приоритеты не реагируют на изменения ситуации в вычислительной системе, которые могут сделать желательной корректировку порядка исполнения процессов.

Более гибкими являются динамические приоритеты процессов, изменяющие свои значения по ходу исполнения процессов. Начальное значение динамического приоритета, присвоенное процессу, действует в течение лишь короткого периода времени, после чего ему назначается новое, более подходящее значение. Изменение динамического приоритета процесса является единственной операцией над процессами, которую мы до сих пор не рассмотрели.

Главная проблема приоритетного планирования заключается в том, что при ненадлежащем выборе механизма назначения и изменения приоритетов низкоприоритетные процессы могут быть не запущены неопределенно долгое время. Обычно случается одно из двух. Или они все же дожидаются своей очереди на исполнение. Или вычислительную систему приходится выключать, и они теряются (при остановке IBM 7094 в Массачусетском технологическом институте в 1973 году были найдены процессы, запущенные в 1967 году и ни разу с тех пор не исполнявшиеся). Решение этой проблемы может быть достигнуто с помощью увеличения со временем значения приоритета процесса, находящегося в состоянии готовность. Пусть изначально процессам присваиваются приоритеты от 128 до 255. Каждый раз, по истечении определенного промежутка времени, значения приоритетов готовых процессов уменьшаются на 1. Процессу, побывавшему в состоянии исполнение, восстанавливается первоначальное значение приоритета. Даже такая грубая схема гарантирует, что любому процессу в разумные сроки будет предоставлено право на исполнение.

Алгоритм планирования Windows NT

В Windows NT реализована вытесняющая многозадачность. Планировщик использует для определения порядка выполнения потоков алгоритм, основанный на приоритетах, в соответствии с которым каждому потоку присваивается число - приоритет, и потоки с более высоким приоритетом выполняются раньше потоков с меньшим приоритетом. Процесс получает базовый приоритет в тот момент, когда его создает подсистема той или иной прикладной среды. Значение базового приоритета присваивается процессу системой по умолчанию, или системным администратором, или указывается при вызове родительским процессом. Поток наследует этот базовый приоритет и может изменить его, немного увеличив или уменьшив. На основании получившегося в результате приоритета, называемого приоритетом планирования, производится планирование выполнения потоков.

Windows NT поддерживает 32 уровня приоритетов, разделенных на два класса - класс реального времени и класс переменных приоритетов. Так называемые потоки реального времени, приоритеты которых находятся в диапазоне от 16 до 31, являются более приоритетными и используются для выполнения задач, критичных ко времени. Каждый раз, когда необходимо выбрать поток для выполнения, диспетчер прежде всего

просматривает очередь готовых к выполнению потоков реального времени и обращается к другим потокам, только когда очередь потоков реального времени пуста. Большинство потоков в системе попадают в класс потоков с переменными приоритетами, диапазон значений - от 0 до 15. Этот класс имеет название "переменные приоритеты", потому что диспетчер настраивает систему, изменяя (понижая или повышая) приоритеты потоков этого класса.

Для того чтобы обеспечить хорошее время реакции системы, алгоритм планирования использует концепцию абсолютных приоритетов, в соответствии с которой при появлении в очереди готовых потоков такого, у которого приоритет выше, чем у выполняющегося в данный момент, происходит смена выполняемого потока на поток с самым высоким приоритетом.

Использование динамических приоритетов, изменяющихся во времени, позволяет реализовать адаптивное планирование, при котором не дискриминируются интерактивные задачи, часто выполняющие операции ввода-вывода и недоиспользующие выделенные им кванты. Например, приоритет потока, окно которого владеет фокусом ввода, в зависимости от настроек системы может быть увеличен на 1 или 2; приоритет потока, который начинает обработку события клавиатурного ввода, временно увеличивается на 2. Если поток полностью исчерпал свой квант, то его приоритет понижается на некоторую величину. В то же время приоритет потоков, которые перешли в состояние ожидания, не использовав полностью выделенный им квант, повышается. Приоритет не изменяется, если поток вытеснен более приоритетным потоком.

В многопроцессорных системах при планировании выполнения потоков играет роль их процессорная совместимость: после того, как планировщик выбрал поток с наивысшим приоритетом, он проверяет, какой процессор может выполнить данный поток и, если атрибут потока "процессорная совместимость" не позволяет потоку выполняться ни на одном из свободных процессоров, то выбирается следующий в порядке приоритетов поток.

Алгоритм планирования UNIX

ОС UNIX изначально была многозадачной системой, ее алгоритм планирования с самого начала разрабатывался так, чтобы обеспечить хорошую реакцию в интерактивных процессах. У этого алгоритма два уровня. Низкоуровневый алгоритм выбирает следующий процесс из набора процессов в памяти и готовых к работе. Высокоуровневый алгоритм перемещает процессы из памяти на диск и обратно, что предоставляет всем процессам возможность попасть в память и быть запущенными.

У каждой версии UNIX свой слегка отличающийся низкоуровневый алгоритм планирования, но у большинства этих алгоритмов есть много общих черт, которые мы опишем. В низкоуровневом алгоритме используется несколько очередей. С каждой очередью связан диапазон непересекающихся значений приоритетов. Процессы, выполняющиеся в режиме пользователя, имеют положительные значения приоритетов. У

процессов, выполняющихся в режиме ядра (или обратившихся к системным вызовам), значения приоритетов отрицательные. Отрицательные значения приоритетов считаются наивысшими, а положительные - наоборот, минимальными. В очередях располагаются только процессы, находящиеся в памяти и готовые к работе.

Когда запускается (низкоуровневый) планировщик, он просматривает очереди, начиная с самого высокого приоритета, пока не находит очередь, в которой есть хотя бы один процесс. После этого из этой очереди он выбирает и запускает первый процесс. Процессу разрешается работать в течение некоего максимального кванта времени (обычно 100 мс) или пока он не заблокируется. Если процесс использует весь свой квант времени, он помещается обратно, в конец очереди, а алгоритм планирования запускается снова. Таким образом, процессы, входящие в одну группу приоритетов, используют центральный процессор в порядке циклической очереди.

Раз в секунду приоритет каждого процесса пересчитывается по следующей формуле: priority = CPU_usage + nice + base

На основе сосчитанного нового приоритета каждый процесс прикрепляется к соответствующей очереди. Для получения номера очереди приоритет, как правило, делится на некую константу. Рассмотрим компоненты этой формулы:

A.CPU_usage – использование центрального процессора – представляет собой среднее значение тиков таймера в секунду, которые процесс работал в течение нескольких последних секунд. При каждом тике таймера счетчик активного процесса увеличивается на единицу. Для того, чтобы избежать неограниченного накапливания слагаемого CPU_usage, его величина со временем уменьшается. В различных версиях это выполнено по-разному. Один из способов состоит в делении CPU_usage в конце каждой секунды на 2, другой – в выполнении в конце секунды операции

CPU_usage = CPU_usage * 2*n/(2*n+1), где n – число процессов в системе.

Б. nice – показатель «любезности» процесса, может принимать значения, например, от -20 до +19. Пользователь может назначить показатель nice в диапазоне от 0 до +19. Только системный администратор может назначить процессу показатель nice от -1 до-20.

С. base – если процесс эмулирует прерывания для выполнения системного вызова, он переходит в состояние ожидания. Когда процесс блокируется, он удаляется из структуры очереди до тех пор, пока он не будет снова готов к работе. Параметр base представляет собой жестко прошитое в операционной системе повышение приоритета процесса, выполнившего системный вызов. В основе такой схемы лежит идея как можно более быстрого удаления процессов из ядра

7.КООПЕРАЦИЯ ПРОЦЕССОВ И ОСНОВНЫЕ АСПЕКТЫ ЕЕ ЛОГИЧЕСКОЙ ОРГАНИЗАЦИИ

Взаимодействующие процессы

Для достижения поставленной цели различные процессы (возможно, даже принадлежащие разным пользователям) могут исполняться псевдопараллельно на одной

вычислительной системе или параллельно на разных вычислительных системах, взаимодействуя между собой.

--Одной из причин кооперации является повышение скорости работы. Когда один процесс ожидает наступления некоторого события (например, окончания операции ввода-вывода), другие в это время могут заниматься полезной работой, направленной на решение общей задачи. В многопроцессорных вычислительных системах программа разделяется на отдельные кусочки, каждый из которых будет исполняться на своем процессоре.

--Второй причиной является совместное использование данных. Различные процессы могут, к примеру, работать с одной и той же динамической базой данных или с разделяемым файлом, совместно изменяя их содержимое.

--Третьей причиной является модульная конструкция какой-либо системы. Типичным примером может служить микроядерный способ построения операционной системы, когда ее различные части представляют собой отдельные процессы, общающиеся путем передачи сообщений через микроядро.

Общение процессов обычно приводит к изменению их поведения в зависимости от полученной информации. Процессы, которые влияют на поведение друг друга путем обмена информацией, принято называть **кооперативными** или взаимодействующими процессами, в отличие от независимых процессов, не оказывающих друг на друга никакого воздействия и ничего не знающих о взаимном сосуществовании в вычислительной системе.

Работа одного процесса не должна приводить к нарушению работы другого процесса. Для этого разделены их адресные пространства и системные ресурсы, и для обеспечения корректного взаимодействия процессов требуются специальные средства и действия операционной системы. Давайте рассмотрим основные аспекты организации совместной работы процессов.

Категории средств обмена информацией

Процессы могут взаимодействовать друг с другом только обмениваясь информацией. По объему передаваемой информации и степени возможного воздействия на поведение другого процесса все средства такого обмена можно разделить на три категории:

****Сигнальные.** Передается минимальное количество информации — один бит, “да” или “нет”. Используются, как правило, для извещения процесса о наступлении какого-либо события. Степень воздействия на поведение процесса, получившего информацию, минимальна.

****Канальные.** Общение процессов происходит через линии связи, предоставленные операционной системой. Объем передаваемой информации в единицу времени ограничен пропускной способностью линий связи.

****Разделяемая память.** Два или более процессов могут совместно использовать некоторую область адресного пространства. Созданием разделяемой памяти занимается операционная система. Общение процессов напоминает совместное проживание студентов в одной комнате общежития. Использование разделяемой памяти для

передачи/получения информации осуществляется с помощью средств обычных языков программирования, в то время как сигнальным и канальным средствам коммуникации для этого необходимы специальные системные вызовы. Разделяемая память представляет собой наиболее быстрый способ взаимодействия процессов в одной вычислительной системе.

Различают два способа адресации: прямую и непрямую. В случае прямой адресации взаимодействующие процессы непосредственно общаются друг с другом, при каждой операции обмена данными явно указывая имя или номер процесса, которому информация предназначена или от которого она должна быть получена. Если и процесс, от которого данные исходят, и процесс, принимающий данные, оба указывают имена своих партнеров по взаимодействию, то такая схема адресации называется симметричной прямой адресацией. Ни один другой процесс не может вмешаться в процедуру симметричного прямого общения двух процессов, перехватить посланные или подменить ожидаемые данные. Если только один из взаимодействующих процессов, например передающий, указывает имя своего партнера по кооперации, а второй процесс в качестве возможного партнера рассматривает любой процесс в системе, например, ожидает получения информации от произвольного источника, то такая схема адресации называется асимметричной прямой адресацией.

При непрямой адресации данные помещаются передающим процессом в некоторый промежуточный объект для хранения данных, имеющий свой адрес, из которого они могут быть затем изъяты каким-либо другим процессом.

При использовании прямой адресации связь между процессами в классической операционной системе устанавливается автоматически, без дополнительных инициализирующих действий.

Под однонаправленной связью мы будем понимать связь, при которой каждый процесс, ассоциированный с ней, может использовать средство связи либо только для приема информации, либо только для ее передачи. При двунаправленной связи, каждый процесс, участвующий в общении, может использовать связь и для приема, и для передачи данных. В коммуникационных системах принято называть однонаправленную связь симплексной, двунаправленную связь с поочередной передачей информации в разных направлениях — полудуплексной, а двунаправленную связь с возможностью одновременной передачи информации в разных направлениях — дуплексной.

Особенности передачи информации с помощью линий связи

Буферизация

Можно выделить три принципиальных варианта буферизации:

1.Буфер нулевой емкости или отсутствует. Никакая информация не может сохраняться на линии связи. В этом случае процесс, посылающий информацию, должен ожидать, пока процесс, принимающий информацию, не соблаговолит ее получить, прежде чем заниматься своими дальнейшими делами.

2.Буфер ограниченной емкости. Размер буфера равен n, то есть линия связи не может хранить до момента получения более чем n единиц информации. Если в момент передачи данных в буфере хватает места, то передающий процесс не должен ничего ожидать. Информация просто копируется в буфер. Если же в момент передачи данных буфер заполнен или места не достаточно, то необходимо задержать работу процесса отправителя до появления в буфере свободного пространства.
3.Буфер неограниченной емкости. Теоретически это возможно, но практически вряд ли реализуемо. Процесс, посылающий информацию, никогда не ждет окончания ее передачи и приема другим процессом.

При использовании канального средства связи с непрямой адресацией под емкостью буфера обычно понимается количество информации, которое может быть помещено в промежуточный объект для хранения данных.

Поток ввода/вывода и сообщения

Существует две модели передачи данных по каналам связи — поток ввода-вывода и сообщения. При передаче данных с помощью потоковой модели, операции передачи/приема информации вообще не интересуются содержимым данных. Процесс, прочитавший 100 байт из линии связи, не знает и не может знать, были ли они переданы одновременно, т. е. одним куском, или порциями по 20 байт, пришли они от одного процесса или от разных процессов. Данные представляют собой простой поток байт, без какой-либо их интерпретации со стороны системы. Примерами потоковых каналов связи могут служить pipe и FIFO, описанные ниже.

Одним из наиболее простых способов передачи информации между процессами по линиям связи является передача данных через pipe (канал, трубу или, как его еще называют в литературе, конвейер). Представим себе, что у вас есть некоторая труба в вычислительной системе, в один из концов которой процессы могут сливать информацию, а из другого конца принимать полученный поток. Естественно, что такой способ реализует потоковую модель ввода/вывода. Информацией о расположении трубы в операционной системе обладает только процесс, создавший ее. Этой информацией он может поделиться исключительно со своими наследниками — процессами-детьми и их потомками. Поэтому использовать pipe для связи между собой могут только родственные процессы, имеющие общего предка, создавшего этот канал связи.

Если разрешить процессу, создавшему трубу, сообщать об ее точном расположении в системе другим процессам, сделав вход и выход трубы каким-либо образом видимыми для всех остальных, например, зарегистрировав ее в операционной системе под определенным именем, мы получим объект, который принято называть FIFO или именованный pipe. Именованный pipe может использоваться для связи между любыми процессами в системе.

В модели сообщений процессы налагают на передаваемые данные некоторую структуру. Весь поток информации они разделяют на отдельные сообщения, вводя между данными, по крайней мере, границы сообщений. Примером границ сообщений являются точки между предложениями в сплошном тексте или границы абзаца. Кроме

того, к передаваемой информации может быть присоединены указания на то, кем конкретное сообщение было послано и для кого оно предназначено.

И потоковые линии связи, и каналы сообщений могут иметь или не иметь буфер. Когда мы будем говорить о емкости буфера для потоков данных, мы будем измерять ее в байтах. Когда мы будем говорить о емкости буфера для сообщений, мы будем измерять ее в сообщениях.

8. ТУПИКИ. БОРЬБА С ТУПИКАМИ

В предыдущих главах рассмотрены способы синхронизации процессов, которые позволяют процессам успешно кооперироваться. Однако если средствами синхронизации пользоваться неосторожно, то могут возникнуть непредвиденные затруднения. Предположим, что несколько процессов конкурируют за обладание конечным числом ресурсов. Если запрашиваемый процессом ресурс недоступен, процесс переходит в состояние ожидания. В случае если требуемый ресурс удерживается другим ожидающим процессом, то первый процесс не сможет сменить свое состояние. Такая ситуация называется тупиком. Говорят, что в мультипрограммной системе процесс находится в состоянии тупика, дедлока (deadlock) или клинча, 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.

Рассмотрим пример. Предположим, что два процесса осуществляют вывод с ленты на принтер. Один из них успел монополизировать ленту и претендует на принтер, а другой наоборот. После этого оба процесса оказываются заблокированными в ожидании второго ресурса (см. рис. 21)

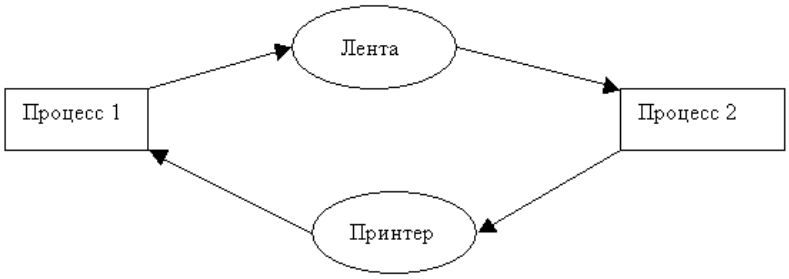

Рис. 21. Пример тупиковой ситуации.

Определение. Множество процессов находится в тупиковой ситуации, если каждый процесс из множества ожидает события, которое только другой процесс данного множества может вызвать. Так как все процессы чего-то ожидают, то ни один из них не сможет инициировать событие, которое разбудило бы другого члена множества и, следовательно, все процессы будут спать вместе. Обычно событие, которого ждет процесс в тупиковой ситуации - освобождение ресурса.

Условия возникновения тупиков

В 1971 г. Коффман, Элфик и Шошани сформулировали следующие четыре условия для возникновения тупиков.

1. Условие взаимоисключения (Mutual exclusion). Каждый ресурс выделен в точности одному процессу или доступен. Процессы требуют предоставления им монопольного управления ресурсами, которые им выделяются.
2. Условие ожидания ресурсов (Hold and wait). Процессы удерживают за собой ресурсы, уже выделенные им, ожидая в то же время выделения дополнительных ресурсов (которые при этом обычно удерживаются другими процессами).
3. Условие неперераспределяемости (No preemtion). Ресурс, данный ранее, не может быть принудительно забран у процесса. Освобождены они могут быть только процессом, который их удерживает.
4. Условие кругового ожидания (Circular wait). Существует кольцевая цепь процессов, в которой каждый процесс удерживает за собой один или более ресурсов, требующихся другим процессам цепи.

Для тупика необходимо выполнение всех четырех условий.

Основные направления борьбы с тупиками.

Основные направления борьбы с тупиками:

1 Игнорировать данную проблему

2 Обнаружение тупиков

3 Восстановление после тупиков

4 Предотвращение тупиков за счет тщательного выделения ресурсов или нарушения одного из условий возникновения тупиков.

9. УПРАВЛЕНИЕ ПАМЯТЬЮ.

Память является важнейшим ресурсом, требующим тщательного управления со стороны мультипрограммной операционной системы. Распределению подлежит вся оперативная память, не занятая операционной системой. Обычно ОС располагается в самых младших адресах, однако может занимать и самые старшие адреса.

Функциями ОС по управлению памятью являются: отслеживание свободной и занятой памяти, выделение памяти процессам и освобождение памяти при завершении процессов, вытеснение процессов из оперативной памяти на диск, когда размеры основной памяти не достаточны для размещения в ней всех процессов, и возвращение их в оперативную память, когда в ней освобождается место, а также настройка адресов программы на конкретную область физической памяти.

Программы, в течение выполнения, вместе с данными, должны находиться в главной (оперативной) памяти. Таким образом, оперативная память (storage, memory) является важнейшим ресурсом, требующим тщательного управления.

Часть ОС, которая управляет памятью, называется ***менеджером памяти***.

Типы адресов

Для идентификации переменных и команд используются *символьные имена* (метки), *виртуальные адреса* и *физические адреса* (рисунок 22).

Символьные имена присваивает пользователь при написании программы на алгоритмическом языке или ассемблере.

Виртуальные адреса вырабатывает транслятор, переводящий программу на машинный язык. Так как во время трансляции в общем случае не известно, в какое место оперативной памяти будет загружена программа, то транслятор присваивает переменным и командам виртуальные (условные) адреса, обычно считая, что программа будет размещена, начиная с нулевого адреса. Совокупность виртуальных адресов процесса называется виртуальным адресным пространством. Каждый процесс имеет собственное виртуальное адресное пространство. Максимальный размер виртуального адресного пространства ограничивается разрядностью адреса, присущей данной архитектуре компьютера, и, как правило, не совпадает с объемом физической памяти, имеющимся в компьютере.

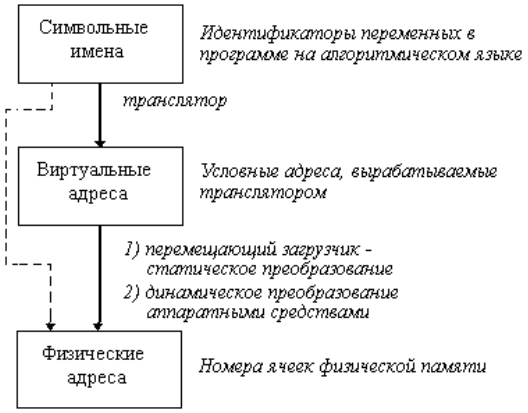

Рис.22

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

Переход от виртуальных адресов к физическим может осуществляться двумя способами. В первом случае замену виртуальных адресов на физические делает специальная системная программа - перемещающий загрузчик. Перемещающий загрузчик на основании имеющихся у него исходных данных о начальном адресе физической памяти, в которую предстоит загружать программу, и информации, предоставленной транслятором об адресно-зависимых константах программы, выполняет загрузку программы, совмещая ее с заменой виртуальных адресов физическими.

Второй способ заключается в том, что программа загружается в память в неизмененном виде в виртуальных адресах, при этом операционная система фиксирует смещение действительного расположения программного кода относительно виртуального адресного пространства. Во время выполнения программы при каждом обращении к оперативной памяти выполняется преобразование виртуального адреса в физический. Второй способ является более гибким, он допускает перемещение программы во время ее выполнения, в то время как перемещающий загрузчик жестко привязывает программу к первоначально выделенному ей участку памяти. Вместе с тем использование перемещающего загрузчика уменьшает накладные расходы, так как преобразование каждого виртуального адреса происходит только один раз во время загрузки, а во втором случае - каждый раз при обращении по данному адресу.

Необходимо различать максимально возможное виртуальное адресное пространство процесса и назначенное (выделенное) процессу виртуальное адресное пространство. В первом случае речь идет о максимальном размере виртуального адресного пространства, определяемом архитектурой компьютера, на котором работает ОС, и, в частности, разрядностью его схем адресации (32-битная, 64-битная и т. п.). Например, при работе на компьютерах с 32-разрядными процессорами Intel Pentium операционная система может предоставить каждому процессу виртуальное адресное пространство до 4 Гбайт (2^{32}). Однако это значение представляет собой только потенциально возможный размер виртуального адресного пространства, который редко на практике бывает необходим процессу. Процесс использует только часть доступного ему виртуального адресного пространства.

Одна из функций управления памятью отображение информации в память. Отображение обычно понимается как преобразование адресных пространств.

Пользовательская программа не видит реальных физических адресов, а имеет дело с логическими адресами, которые являются результатом трансляции символьных имен программы.

Назначенное виртуальное адресное пространство представляет собой набор виртуальных адресов, действительно нужных процессу для работы. Эти адреса первоначально назначает программе транслятор на основании текста программы, когда создает кодовый (текстовый) сегмент, а также сегмент или сегменты данных, с которыми программа работает. Затем при создании процесса ОС фиксирует назначенное виртуальное адресное пространство в своих системных таблицах. В ходе своего выполнения процесс может увеличить размер первоначального назначенного ему виртуального адресного пространства, запросив у ОС создания дополнительных сегментов или увеличения размера существующих. В любом случае операционная система обычно следит за корректностью использования процессом виртуальных адресов — процессу не разрешается оперировать с виртуальным адресом, выходящим за пределы назначенных ему сегментов.

Максимальный размер виртуального адресного пространства ограничивается только разрядностью адреса, присущей данной архитектуре компьютера, и, как правило, не совпадает с объемом физической памяти, имеющимся в компьютере.

Методы распределения памяти без использования дискового пространства

Все методы управления памятью могут быть разделены на два класса: методы, которые используют перемещение процессов между оперативной памятью и диском, и методы, которые не делают этого (рисунок 23). Начнем с последнего, более простого класса методов.

Рис.23 Методы распределения памяти

Распределение памяти фиксированными разделами

Самым простым способом управления оперативной памятью является разделение ее на несколько разделов фиксированной величины. Это может быть выполнено вручную оператором во время старта системы или во время ее генерации. Очередная задача, поступившая на выполнение, помещается либо в общую очередь (рисунок 23а), либо в очередь к некоторому разделу (рисунок 23б).

Подсистема управления памятью в этом случае выполняет следующие задачи: сравнивая размер программы, поступившей на выполнение, и свободных разделов, выбирает подходящий раздел, осуществляет загрузку программы и настройку адресов.

В какой раздел помещать программу? Распространены три стратегии:

1.Стратегия первого подходящего (First fit). Задание помещается в первый подходящий по размеру раздел.

2.Стратегия наиболее подходящего (Best fit). Задание помещается в тот раздел, где ему наиболее тесно.

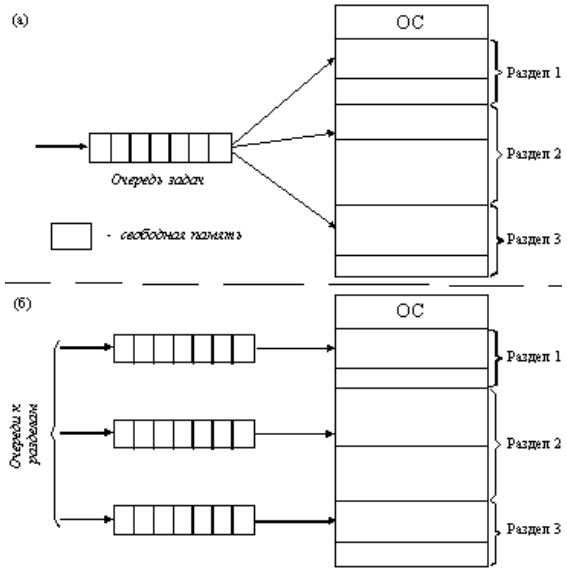

Рис.23 Распределение памяти фиксированными разделами

3.Стратегия наименее подходящего (Worst fit). При помещении в самый большой раздел в нем остается достаточно места для возможного размещения еще одного процесса.

Преимущество - простота реализации

Недостатки:

- жесткость. Так как в каждом разделе может выполняться только одна программа, то уровень мультипрограммирования заранее ограничен числом разделов не зависимо от того, какой размер имеют программы.
- схема сильно страдает от внешней фрагментации потери памяти, не используемой ни одним процессом. Фрагментация возникает потому, что процесс не полностью занимает выделенный ему раздел или вследствие не использования некоторых разделов, которые слишком малы для выполняемых пользовательских программ.

Распределение памяти разделами переменной величины

В этом случае память машины не делится заранее на разделы. Сначала вся память свободна. Каждой вновь поступающей задаче выделяется необходимая ей память. Если достаточный объем памяти отсутствует, то задача не принимается на выполнение и стоит в очереди. После завершения задачи память освобождается, и на это место может быть загружена другая задача. Таким образом, в произвольный момент времени оперативная память представляет собой случайную последовательность занятых и свободных участков (разделов) произвольного размера. На рисунке 24 показано состояние памяти в различные моменты времени при использовании динамического распределения. Так в момент t0 в памяти находится только ОС, а к моменту t1 память разделена между 5 задачами, причем задача П4, завершаясь, покидает память. На освободившееся после задачи П4 место загружается задача П6, поступившая в момент t3.

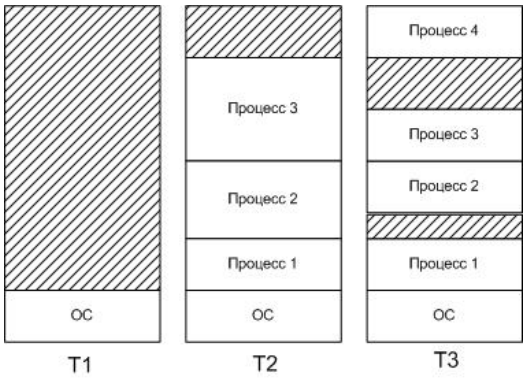

Рис.24 Распределение памяти разделами переменой величины

Задачами операционной системы при реализации данного метода управления памятью является:

1.ведение таблиц свободных и занятых областей, в которых указываются начальные адреса и размеры участков памяти,

2.при поступлении новой задачи - анализ запроса, просмотр таблицы свободных областей и выбор раздела, размер которого достаточен для размещения поступившей задачи,
3.загрузка задачи в выделенный ей раздел и корректировка таблиц свободных и занятых областей,
4.после завершения задачи корректировка таблиц свободных и занятых областей.

Программный код не перемещается во время выполнения, то есть может быть проведена единовременная настройка адресов посредством использования перемещающего загрузчика.

Выбор раздела для вновь поступившей задачи может осуществляться по разным правилам, таким, например, как "первый попавшийся раздел достаточного размера", или "раздел, имеющий наименьший достаточный размер", или "раздел, имеющий наибольший достаточный размер". Все эти правила имеют свои преимущества и недостатки.

По сравнению с методом распределения памяти фиксированными разделами данный метод обладает гораздо большей гибкостью, но ему присущ очень серьезный недостаток - фрагментация памяти. Фрагментация - это наличие большого числа несмежных участков свободной памяти очень маленького размера (фрагментов). Настолько маленького, что ни одна из вновь поступающих программ не может поместиться ни в одном из участков, хотя суммарный объем фрагментов может составить значительную величину, намного превышающую требуемый объем памяти.

Перемещаемые разделы

Одним из методов борьбы с фрагментацией является перемещение всех занятых участков в сторону старших либо в сторону младших адресов, так, чтобы вся свободная память образовывала единую свободную область

Это один из методов борьбы с фрагментацией. Но на него уходит много времени.

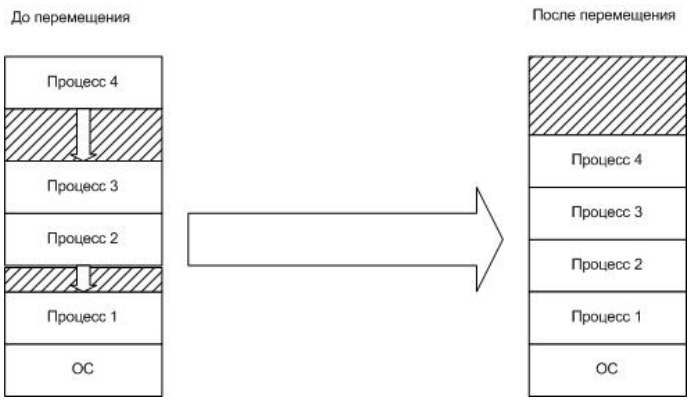

Рис.25 Перемещаемые разделы памяти

В дополнение к функциям, которые выполняет ОС при распределении памяти динамическими разделами в данном случае, она должна еще время от времени копировать содержимое разделов из одного места памяти в другое (рис.25), корректируя таблицы свободных и занятых областей. Эта процедура называется сжатием.

Методы с использованием внешней памяти (свопинг и виртуальная память)

В условиях, когда для обеспечения приемлемого уровня мультипрограммирования имеющейся оперативной памяти недостаточно, был предложен метод, при котором образы некоторых процессов целиком или частично временно выгружаются на диск.

В мультипрограммном режиме помимо активного процесса (то есть процесса, коды которого в настоящий момент интерпретируются процессором) имеются приостановленные процессы, находящиеся в ожидании завершения ввода-вывода или освобождения ресурсов, а также процессы в состоянии готовности, стоящие в очереди к процессору. Образы таких неактивных процессов могут быть временно, до следующего цикла активности, выгружены на диск. Несмотря на то, что коды и данные процесса отсутствуют в оперативной памяти, ОС «знает» о его существовании и в полной мере учитывает это при распределении процессорного времени и других системных ресурсов. К моменту, когда подходит очередь выполнения выгруженного процесса, его образ возвращается с диска в оперативную память. Если при этом обнаруживается, что свободного места в оперативной памяти не хватает, то на диск выгружается другой процесс.

Виртуализация оперативной памяти осуществляется совокупностью программных модулей ОС и аппаратных схем процессора и включает решение следующих задач:

- размещение данных в запоминающих устройствах разного типа, например часть кодов программы — в оперативной памяти, а часть — на диске;
- выбор образов процессов или их частей для перемещения из оперативной памяти на диск и обратно;
- перемещение по мере необходимости данных между памятью и диском; Q преобразование виртуальных адресов в физические.

Виртуализация памяти может быть осуществлена на основе двух различных подходов:

-свопинг (swapping) — образы процессов выгружаются на диск и возвращаются в оперативную память целиком;

-виртуальная память (virtual memory) — между оперативной памятью и диском перемещаются части (сегменты, страницы и т. п.) образов процессов.

Понятие виртуальной памяти

Разработчикам программного обеспечения часто приходится решать проблему размещения в памяти больших программ, размер которых превышает объем доступной оперативной памяти. Развитие архитектуры компьютеров и расширение возможностей операционной системы по управлению памятью позволило переложить решение этой задачи на компьютер. Одним из главных достижений стало появление ***виртуальной памяти* (*virtual memory*).** Впервые она была реализована в 1959 г. на компьютере "Атлас", разработанном в Манчестерском университете.

Суть концепции ***виртуальной памяти*** заключается в следующем. Информация, с которой работает активный процесс, должна располагаться в оперативной памяти. В схемах ***виртуальной памяти*** у процесса создается иллюзия того, что вся необходимая

ему информация имеется в основной памяти. Для этого, во-первых, занимаемая процессом память разбивается на несколько частей, например страниц. Во-вторых, логический адрес (логическая страница), к которому обращается процесс, динамически транслируется в физический адрес (физическую страницу). И, наконец, в тех случаях, когда страница, к которой обращается процесс, не находится в физической памяти, нужно организовать ее подкачку с диска. Для контроля наличия страницы в памяти вводится специальный ***бит присутствия***, входящий в состав атрибутов страницы в ***таблице страниц***.

Таким образом, в наличии всех компонентов процесса в основной памяти необходимости нет. Важным следствием такой организации является то, что размер памяти, занимаемой процессом, может быть больше, чем размер оперативной памяти.

Возможность выполнения программы, находящейся в памяти лишь частично, имеет ряд вполне очевидных преимуществ.

- Программа не ограничена объемом физической памяти. Упрощается разработка программ, поскольку можно задействовать большие виртуальные пространства, не заботясь о размере используемой памяти.
- Поскольку появляется возможность частичного помещения программы (процесса) в память и гибкого перераспределения памяти между программами, можно разместить в памяти больше программ, что увеличивает загрузку процессора и пропускную способность системы.
- Объем ввода-вывода для выгрузки части программы на диск может быть меньше, чем в варианте классического свопинга, в итоге каждая программа будет работать быстрее.

Таким образом, возможность обеспечения (при поддержке операционной системы) для программы "видимости" практически неограниченной (характерный размер для 32-разрядных архитектур 2^{32}= 4 Гбайт) адресуемой пользовательской памяти (логическое адресное пространство) при наличии основной памяти существенно меньших размеров (физическое адресное пространство) – очень важный аспект.

Но введение ***виртуальной памяти*** позволяет решать другую, не менее важную задачу – обеспечение контроля доступа к отдельным сегментам памяти и, в частности, защиту пользовательских программ друг от друга и защиту ОС от пользовательских программ. Каждый процесс работает со своими ***виртуальными адресами***, трансляцию которых в физические выполняет аппаратура компьютера. Таким образом, пользовательский процесс лишен возможности напрямую обратиться к страницам основной памяти, занятым информацией, относящейся к другим процессам.

Напомним, что в системах с ***виртуальной памятью*** те адреса, которые генерирует программа (логические адреса), называются виртуальными, и они формируют виртуальное адресное пространство. Термин "***виртуальная память***" означает, что программист имеет дело с памятью, отличной от реальной, размер которой потенциально больше, чем размер оперативной памяти.

Хотя известны и чисто программные реализации ***виртуальной памяти***, это направление получило наиболее широкое развитие после соответствующей аппаратной поддержки.

Страничное распределение

Виртуальное адресное пространство каждого процесса делится на части одинакового, фиксированного для данной системы размера, называемые виртуальными страницами (virtual pages). В общем случае размер виртуального адресного пространства процесса не кратен размеру страницы, поэтому последняя страница каждого процесса дополняется фиктивной областью.

Вся оперативная память машины также делится на части такого же размера, называемые физическими страницами (или блоками, или кадрами). Размер страницы выбирается равным степени двойки: 512, 1024, 4096 байт и т. д. Это позволяет упростить механизм преобразования адресов.

При создании процесса ОС загружает в оперативную память несколько его виртуальных страниц (начальные страницы кодового сегмента и сегмента данных). Копия всего виртуального адресного пространства процесса находится на диске. Смежные виртуальные страницы не обязательно располагаются в смежных физических страницах. Для каждого процесса операционная система создает таблицу страниц — информационную структуру, содержащую записи обо всех виртуальных страницах процесса.

Запись таблицы, называемая дескриптором страницы, включает следующую информацию:

- номер физической страницы, в которую загружена данная виртуальная страница;
- признак присутствия, устанавливаемый в единицу, если виртуальная страница находится в оперативной памяти;
- признак модификации страницы, который устанавливается в единицу всякий раз, когда производится запись по адресу, относящемуся к данной странице;
- признак обращения к странице, называемый также битом доступа, который устанавливается в единицу при каждом обращении по адресу, относящемуся к данной странице.

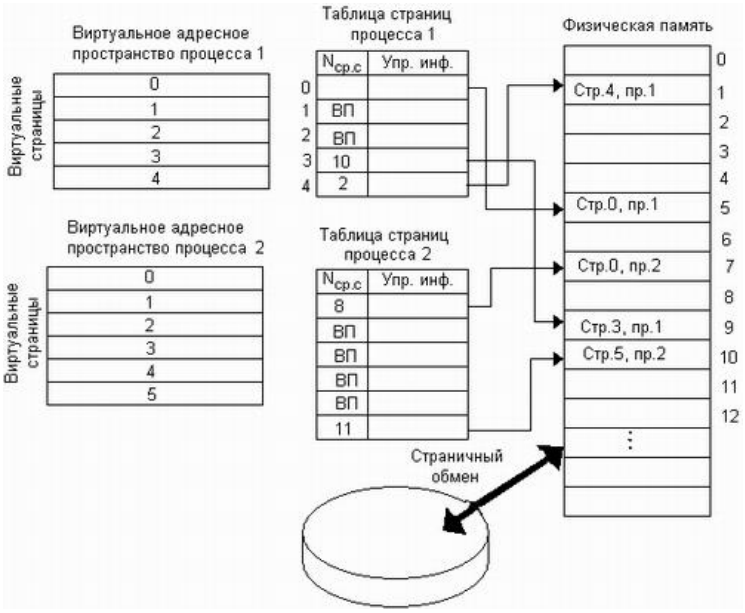

Рис.26 Страничное распределение памяти.

При каждом обращении к памяти выполняется поиск номера виртуальной страницы, содержащей требуемый адрес, затем по этому номеру определяется нужный элемент таблицы страниц, и из него извлекается описывающая страницу информация1. Далее анализируется признак присутствия, и, если данная виртуальная страница находится в оперативной памяти, то выполняется преобразование виртуального адреса в физический, то есть виртуальный адрес заменяется указанным в записи таблицы физическим адресом. Если же нужная виртуальная страница в данный момент выгружена на диск, то происходит так называемое страничное прерывание. Выполняющийся процесс переводится в состояние ожидания, и активизируется другой процесс из очереди процессов, находящихся в состоянии готовности. Параллельно программа обработки страничного прерывания находит на диске требуемую виртуальную страницу (для этого операционная система должна помнить положение вытесненной страницы в страничном файле диска) и пытается загрузить ее в оперативную память. Если в памяти имеется свободная физическая страница, то загрузка выполняется немедленно, если же свободных страниц нет, то на основании принятой в данной системе стратегии замещения страниц решается вопрос о том, какую страницу следует выгрузить из оперативной памяти.

После того как выбрана страница, которая должна покинуть оперативную память, обнуляется ее бит присутствия и анализируется ее признак модификации. Если выталкиваемая страница за время последнего пребывания в оперативной памяти была модифицирована, то ее новая версия должна быть переписана на диск. Если нет, то принимается во внимание, что на диске уже имеется предыдущая копия этой виртуальной страницы, и никакой записи на диск не производится. Физическая страница объявляется свободной. Из соображений безопасности в некоторых системах освобождаемая страница обнуляется, с тем чтобы невозможно было использовать содержимое выгруженной страницы.

Сегментное распределение

При страничной организации виртуальное адресное пространство процесса делится на равные части механически, без учета смыслового значения данных. В одной странице могут оказаться и коды команд, и инициализируемые переменные, и массив исходных данных программы. Такой подход не позволяет обеспечить дифференцированный доступ к разным частям программы, а это свойство могло бы быть очень полезным во многих случаях. Например, можно было бы запретить обращаться с операциями записи в сегмент программы, содержащий коды команд, разрешив эту операцию для сегментов данных.

Кроме того, разбиение виртуального адресного пространства на «осмысленные» части делает принципиально возможным совместное использование фрагментов программ разными процессами. Пусть, например, двум процессам требуется одна и та же подпрограмма, которая к тому же обладает свойством реентерабельности . Тогда коды этой подпрограммы могут быть оформлены в виде отдельного сегмента и включены в

виртуальные адресные пространства обоих процессов. При отображении в физическую память сегменты, содержащие коды подпрограммы из обоих виртуальных пространств, проецируются на одну и ту же область физической памяти. Таким образом оба процесса получат доступ к одной и той же копии подпрограммы

Итак, виртуальное адресное пространство процесса делится на части — сегменты, размер которых определяется с учетом смыслового значения содержащейся в них информации. Отдельный сегмент может представлять собой подпрограмму, массив данных и т. п. Деление виртуального адресного пространства на сегменты осуществляется компилятором на основе указаний программиста или по умолчанию, в соответствии с принятыми в системе соглашениями. Максимальный размер сегмента определяется разрядностью виртуального адреса, например при 32-разрядной организации процессора он равен 4 Гбайт. При этом максимально возможное виртуальное адресное пространство процесса представляет собой набор из N (определенного количества) виртуальных сегментов, каждый размером по 4 Гбайт. В каждом сегменте виртуальные адреса находятся в диапазоне от 0000000016 до FFFFFFFF16. Виртуальный логический адрес - упорядоченная пара v=(s,d), номер сегмента и смещение внутри сегмента.

Недостатком сегментного распределения является избыточность. При сегментной организации единицей перемещения между памятью и диском является сегмент, имеющий в общем случае объем больший, чем страница. Однако во многих случаях для работы программы вовсе не требуется загружать весь сегмент целиком, достаточно было бы одной или двух страниц. Аналогично при отсутствии свободного места в памяти не стоит выгружать целый сегмент, когда можно обойтись выгрузкой нескольких страниц.

Но главный недостаток сегментного распределения — это фрагментация, которая возникает из-за непредсказуемости размеров сегментов. В процессе работы системы в памяти образуются небольшие участки свободной памяти, в которые не может быть загружен ни один сегмент. Суммарный объем, занимаемый фрагментами, может составить существенную часть общей памяти системы, приводя к ее неэффективному использованию.

Странично-сегментное распределение

Как видно из названия, данный метод представляет собой комбинацию страничного и сегментного распределения памяти и, вследствие этого, сочетает в себе достоинства обоих подходов. Виртуальное пространство процесса делится на сегменты, а каждый сегмент в свою очередь делится на виртуальные страницы, которые нумеруются в пределах сегмента. Оперативная память делится на физические страницы. Загрузка процесса выполняется операционной системой постранично, при этом часть страниц размещается в оперативной памяти, а часть на диске. Для каждого сегмента создается своя таблица страниц, структура которой полностью совпадает со структурой таблицы страниц, используемой при страничном распределении. Для каждого процесса создается таблица сегментов, в которой указываются адреса таблиц страниц для всех сегментов данного процесса. Адрес таблицы сегментов загружается в специальный регистр процессора, когда активизируется соответствующий процесс.

Иерархия памяти

Все предыдущие рассуждения справедливы и для других пар запоминающих устройств, например, для оперативной памяти и внешней памяти. В этом случае уменьшается среднее время доступа к данным, расположенным на диске, и роль кэш-памяти выполняет буфер в оперативной памяти.

Рассмотренная нами схема трехуровневой памяти (ассоциативная, основная, вторичная) является частным случаем многоуровневой памяти. Например, как показано на рис. 27, разновидности памяти могут быть организованы в иерархию по убыванию скорости доступа и возрастанию цены.

Считается, что затраты, связанные с переписью информации из одной памяти в другую окажутся меньше выигрыша в быстродействии, который получается за счет сокращения времени выборки из более быстрых слоев памяти. Информация о странице, которая находится в памяти верхнего уровня, хранится также на уровнях с большими номерами. Если процессор не обнаруживает нужную страницу на i-м уровне, он

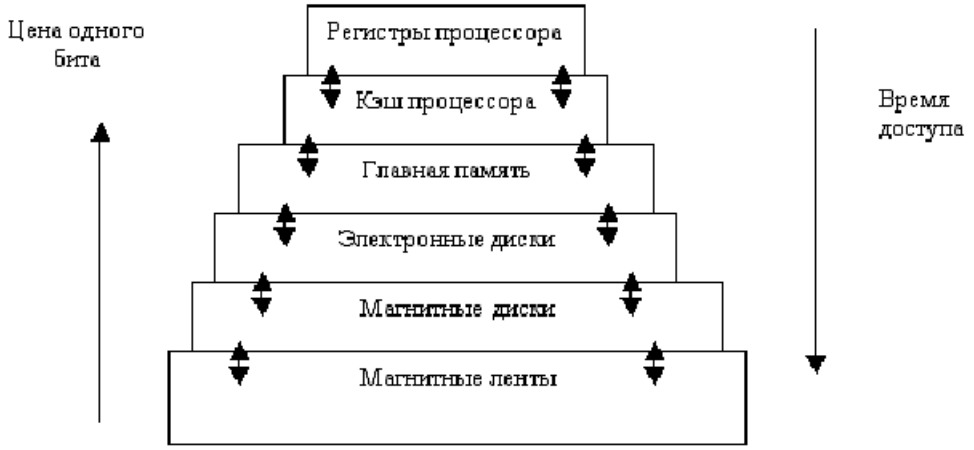

Рис.27 Иерархия памяти в вычислительной системе.

начинает искать ее на последующих уровнях. Когда нужная страница найдена, она переносится в более быстрые уровни. При этом происходит вытеснение какой-то старой страницы, обычно той, которая дольше всего не использовалась. Идея состоит в том, чтобы те страницы, которые чаще всего нужны в настоящее время, находились в более быстрых частях памяти. Эффективность такой схемы обусловлена все тем же свойством локальности (подробнее о локальности и связанным с ним понятием рабочего множества страниц будет рассказано в следующей главе). В результате среднее время доступа для многоуровневой схемы памяти оказывается весьма близким ко времени доступа первого уровня.

10. ФАЙЛОВЫЕ СИСТЕМЫ

Файловая система - это часть операционной системы, назначение которой - организовать эффективную работу с данными, хранящимися во внешней памяти и обеспечить пользователю удобный интерфейс при работе с этими данными.

Организовать хранение информации на магнитном диске непросто. Это требует хорошего знания устройства контроллера диска, особенностей работы с его регистрами и.т. д. Для того чтобы избавить пользователя компьютера от сложностей взаимодействия с аппаратурой и была придумана ясная абстрактная модель файловой системы.

Понятие «файловая система» включает

-совокупность всех файлов на диске,

-наборы структур данных, используемых для управления файлами, такие, например, как каталоги файлов, дескрипторы файлов, таблицы распределения свободного и занятого пространства на диске,

-комплекс системных программных средств, реализующих управление файлами, в частности: создание, уничтожение, чтение, запись, именование, поиск и другие операции над файлами.

Основные функции файловой системы:

1.Идентификация файлов. Связывание имени файла с выделенным ему пространством внешней памяти.

2.Распределение внешней памяти между файлами. Для работы с конкретным файлом не требуется иметь информацию о местоположении этого файла на внешнем носителе информации. Например, для того, чтобы загрузить документ в редактор с жесткого диска нам не требуется знать на какой стороне какого магнитного диска и на каком цилиндре и в каком секторе находится требуемый документ.

3.Обеспечение надежности и отказоустойчивости. Стоимость информации может во много раз превышать стоимость компьютера.

4. Обеспечение защиты от НСД.

5. Обеспечение совместного доступа к файлам, не требуя от пользователя специальных усилий по обеспечению синхронизации доступа.

6.Обеспечение высокой производительности.

Файловая система позволяет программам обходиться набором простых операций для выполнения действий над некоторым абстрактным объектом, представляющим файл. При этом пользователям не нужно иметь дело с деталями действительного расположения данных на диске, буферизацией данных и другими низкоуровневыми проблемами передачи данных с долговременного запоминающего устройства. Все эти функции файловая система берет на себя.

Иерархическая структура файловой системы

Большинство файловых систем имеет иерархическую структуру, в которой уровни создаются за счет того, что каталог более низкого уровня может входить в каталог более высокого уровня (рис. 28).

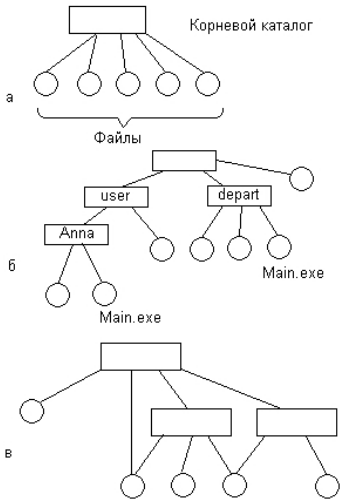

Рис.28 Иерархическая структура файловой системы.

Граф, описывающий иерархию каталогов, может быть деревом или сетью. Каталоги образуют дерево, если файлу разрешено входить только в один каталог (рис. 28 б), и сеть — если файл может входить сразу в несколько каталогов (рис. 28в). Например, в MS-DOS и Windows каталоги образуют древовидную структуру, а в UNIX — сетевую. В древовидной структуре каждый файл является листом. Каталог самого верхнего уровня называется корневым каталогом, или корнем (root).

Иерархическая структура удобна для многопользовательской работы: каждый пользователь со своими файлами локализуется в своем каталоге или поддереве каталогов, и вместе с тем все файлы в системе логически связаны.

Имена файлов

Файл — это именованная область внешней памяти, в которую можно записывать и из которой можно считывать данные. Файлы – абстрактные объекты, предоставляющие пользователям возможность работать с информацией, скрывая от него детали того, как и где она хранится и то, как диски в действительности работают.

Все типы файлов имеют символьные имена. В иерархически организованных файловых системах обычно используются три типа имен -файлов: простые, составные и относительные.

Простое, или короткое, символьное имя идентифицирует файл в пределах одного каталога. Простые имена присваивают файлам пользователи, при этом они должны учитывать ограничения ОС как на номенклатуру символов, так и на длину имени. Так, в системе FAT длина имен ограничивались по следующей схеме:8 символов — собственно имя, 3 символа — расширение имени. В файловой системе s5, поддерживаемой многими версиями ОС UNIX, простое символьное имя не могло содержать более 14 символов. Современные файловые системы, поддерживают длинные простые символьные имена файлов. Например, в NTFS и FAT32 имя файла может содержать до 255 символов.

Полное имя представляет собой цепочку простых символьных имен всех каталогов, через которые проходит путь от корня до данного файла. Таким образом, полное имя является составным, в котором простые имена отделены друг от друга принятым в ОС

разделителем. Часто в качестве разделителя используется прямой или обратный слеш, при этом принято не указывать имя корневого каталога.

В древовидной файловой системе между файлом и его полным именем имеется взаимно однозначное соответствие «один файл — одно полное имя». В файловых системах, имеющих сетевую структуру, файл может входить в несколько каталогов, а значит, иметь несколько полных имен; здесь справедливо соответствие «один файл — много полных имен». В обоих случаях файл однозначно идентифицируется полным именем.

Относительное имя файла определяется через понятие «текущий каталог». Для каждого пользователя в каждый момент времени один из каталогов файловой системы является текущим, причем этот каталог выбирается самим пользователем по команде ОС. Файловая система фиксирует имя текущего каталога, чтобы затем использовать его как дополнение к относительным именам для образования полного имени файла. При использовании относительных имен пользователь идентифицирует файл цепочкой имен каталогов, через которые проходит маршрут от текущего каталога до данного файла. Например, если текущим каталогом является каталог /user, то относительное имя файла /user/anna/main.exe выглядит следующим образом: anna/ main.exe

Многие ОС поддерживают имена из двух частей (имя+расширение), например progr.c(файл, содержащий текст программы на языке Си) или autoexec.bat (файл, содержащий команды интерпретатора командного языка). Тип расширения файла позволяет ОС организовать работу с ним различных прикладных программ в соответствии с заранее оговоренными соглашениями.

Типы и атрибуты файлов

Атрибуты — это информация, описывающая свойства файла. Примеры возможных атрибутов файла:

-тип файла (обычный файл, каталог, специальный файл и т. п.);

-владелец файла;

-создатель файла;

-пароль для доступа к файлу;

-информация о разрешенных операциях доступа к файлу;

-времена создания, последнего доступа и последнего изменения;

-текущий размер файла;

-максимальный размер файла;

признак «только для чтения»;

-признак «скрытый файл»;

-признак «системный файл»;

-признак «архивный файл»;

-признак «двоичный/символьный»;

-признак «временный» (удалить после завершения процесса);

-признак блокировки.

К типам файлов, поддерживаемых современными ОС, относят регулярные (обычные) файлы и директории. Обычные (регулярные) файлы содержат пользовательскую информацию. Директории (справочники, каталоги) - системные файлы, поддерживающие структуру файловой системы. В каталоге содержится перечень файлов, входящих в него, и устанавливается соответствие между файлами и их характеристиками (атрибутами). Мы будем рассматривать директории ниже.

Обычные (или регулярные) файлы реально представляют собой набор блоков на устройстве внешней памяти. Такие файлы могут содержать как текстовую информацию (обычно в формате ASCII), так и произвольную двоичную информацию.

Обычные регулярные файлы бывают - ASCII (American Standard Code for Information Interchange)и бинарные.

ASCII представляет собой кодировку для представления десятичных цифр, латинского и национального алфавитов, знаков препинания и управляющих символов. ASCII файлы содержат строки текста, которые можно распечатать, увидеть на экране или редактировать обычным текстовым редактором.

Другой тип файлов – бинарные файлы, в этом контексте противопоставляются текстовым файлам, и представляют собой набор байтов (в отличие от текстовых, которые представляют собой набор символов), которые, в свою очередь, состоят из битов. А бит - это может принимать значение 0 или 1.

Обычно они имеют некоторую внутреннюю структуру. Например, выполнимый Unix файл имеет пять секций: заголовок, текст, данные, биты реаллокации и символьную таблицу. ОС выполняет файл, только если он имеет нужный формат. Другим примером бинарного файла может быть архивный файл.

Типизация файлов не слишком строгая.

Обычно прикладные программы, работающие с файлами, распознают тип файла по его имени в соответствии с общепринятыми соглашениями. Например, файлы с расширениями .c, .pas, .txt – ASCII файлы, файлы с расширениями .exe – выполнимые, файлы с расширениями .obj, .zip – бинарные и т.д.

Операции над файлами

Операционная система должна предоставить в распоряжение пользователя набор операций для работы с файлами, реализованных через системные вызовы. Чаще всего при работе с файлом пользователь выполняет не одну, а несколько операций. Во-первых, нужно найти данные файла и его атрибуты по его символьному имени, во-вторых, считать необходимые атрибуты файла в отведенную область оперативной памяти и проанализировать права пользователя на выполнение требуемой операции. Затем выполнить операцию, после чего освободить занимаемую данными файла область памяти. Рассмотрим в качестве примера основные файловые операции ОС Unix:

Create. Создание файла, не содержащего данных. Смысл данного вызова - объявить, что файл существует и присвоить ему ряд атрибутов.

Delete. Удаление файла и освобождение занятого им дискового пространства.

Open. Перед использованием файла процесс должен его открыть. Цель данного системного вызова разрешить системе проанализировать атрибуты файла и проверить права доступа к файлу, а также считать в оперативную память список адресов блоков файла для быстрого доступа к его данным.

Close. Если работа с файлом завершена, его атрибуты и адреса блоков на диске больше не нужны. В этом случае файл нужно закрыть, чтобы освободить место во внутренних таблицах файловой системы.

Seek. Дает возможность специфицировать место внутри файла, откуда будет производиться считывание (или запись) данных, то есть задать текущую позицию.

Read. Чтение данных из файла. Обычно это происходит с текущей позиции. Пользователь должен задать объем считываемых данных и предоставить буфер для них.

Write. Запись данных в файл с текущей позиции. Если текущая позиция находится в конце файла, его размер увеличивается, в противном случае запись осуществляется на место имеющихся данных, которые, таким образом, теряются.

Get attributes. Предоставляет процессам нужные им сведения об атрибутах файла. В качестве примера можно привести, утилиту make, которая использует информацию о времени последней модификации файлов.

Set attributes. Дает возможность пользователю установить некоторые атрибуты. Наиболее очевидный пример - установка режима доступа к файлу.

Rename. Возможность переименования файла создает дополнительные удобства для пользователя. Данная операция может быть смоделирована копированием данного файла в файл с новым именем и последующим его удалением.

Реализация файловой системы

Общая структура файловой системы

Нижний уровень - оборудование. Это в первую очередь, магнитные диски с подвижными головками - основные устройства внешней памяти, представляющие собой пакеты магнитных пластин (поверхностей), между которыми на одном рычаге двигается пакет магнитных головок (рис.29). Шаг движения пакета головок является дискретным и каждому положению пакета головок логически соответствует цилиндр магнитного диска. Цилиндры делятся на дорожки (треки), а каждая дорожка размечается на одно и то же количество блоков (секторов), таким образом, что в каждый блок можно записать по максимуму одно и то же число байтов. Следовательно, для произведения обмена с магнитным диском на уровне аппаратуры нужно указать номер цилиндра, номер поверхности, номер блока на соответствующей дорожке и число байтов, которое нужно записать или прочитать от начала этого блока.

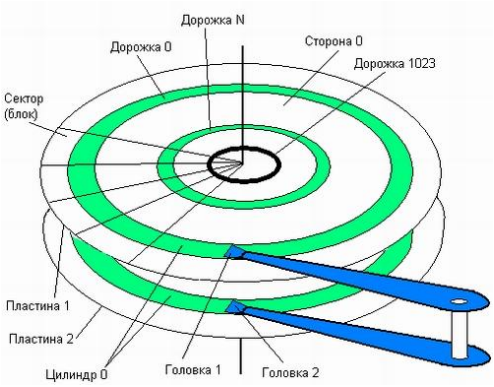

Рис.29 Принцип работы жёсткого диска

Непосредственно с устройствами (дисками) взаимодействует часть ОС, называемая система ввода-вывода (см. соответствующую главу). Система ввода-вывода (она состоит из драйверов устройств и обработчиков прерываний для передачи информации между памятью и дисковой системой) предоставляет в распоряжение более высокоуровневого компонента ОС - файловой системы используемое дисковое пространство в виде непрерывной последовательности блоков фиксированного размера. Система ввода-вывода имеет дело с физическими блоками диска, которые характеризуются адресом, например, диск 2, цилиндр 75, сектор 11. Файловая система имеет дело с логическими блоками, каждый из которых имеет номер (от 0 или 1 до N). Размер этих логических блоков файла совпадает или кратен размеру физического блока диска и может быть задан равным размеру страницы виртуальной памяти, поддерживаемой аппаратурой компьютера совместно с операционной системой.

Дорожки и секторы создаются в результате выполнения процедуры физического, или низкоуровневого, форматирования диска, предшествующей использованию диска. Для определения границ блоков на диск записывается идентификационная информация. Низкоуровневый формат диска не зависит от типа операционной системы, которая этот диск будет использовать.

Разметку диска под конкретный тип файловой системы выполняют процедуры высокоуровневого, или логического, форматирования. При высокоуровневом форматировании определяется размер кластера и на диск записывается информация, необходимая для работы файловой системы, в том числе информация о доступном и неиспользуемом пространстве, о границах областей, отведенных под файлы и каталоги, информация о поврежденных областях. Кроме того, на диск записывается загрузчик операционной системы — небольшая программа, которая начинает процесс инициализации операционной системы после включения питания или рестарта компьютера.

Прежде чем форматировать диск под определенную файловую систему, он может быть разбит на разделы. Раздел — это непрерывная часть физического диска, которую операционная система представляет пользователю как логическое устройство (используются также названия логический диск и логический раздел)1. Логическое

устройство функционирует так, как если бы это был отдельный физический диск. Именно с логическими устройствами работает пользователь, обращаясь к ним по символьным именам, используя, например, обозначения A, B, C, SYS и т. п. Операционные системы разного типа используют единое для всех них представление о разделах, но создают на его основе логические устройства, специфические для каждого типа ОС. Так же как файловая система, с которой работает одна ОС, в общем случае не может интерпретироваться ОС другого типа, логические устройства не могут быть использованы операционными системами разного типа. На каждом логическом устройстве может создаваться только одна файловая система.

11. МЕТОДЫ ВЫДЕЛЕНИЯ ДИСКОВОГО ПРОСТРАНСТВА.

Выделение непрерывной последовательностью блоков

Простейший способ - хранить каждый файл, как непрерывную последовательность блоков диска. При непрерывном расположении файл характеризуется адресом и длиной (в блоках). Файл, стартующий с блока b, занимает затем блоки b+1, b+2, ... b+n-1.

Эта схема имеет два преимущества. Во-первых, ее легко реализовать, т.к. выяснение местонахождения файла сводится к вопросу, где находится первый блок. Во-вторых, она обеспечивает хорошую производительность, т.к. целый файл может быть считан за одну дисковую операцию.

Непрерывное выделение используется в ОС IBM/CMS, в ОС RSX-11 (для выполняемых файлов) и в ряде других.

Основная проблема, вследствие чего этот способ мало распространен - трудно найти место для нового файла. В процессе эксплуатации диск представляет собой некоторую совокупность свободных и занятых фрагментов. Проблема непрерывного расположения может рассматриваться как частный случай более общей проблемы выделения n блоков из списка свободных дыр. Наиболее распространенные стратегии решения этой проблемы - first fit, best fit и worst fit (ср. с проблемой выделения памяти). Таким образом, метод страдает от внешней фрагментации, в зависимости от размера диска и среднего размера файла, в большей или меньшей степени. (Внутренняя подразумевает пустоты внутри страницы. Внешняя – непоследовательность связей страниц.)

Кроме того, непрерывное распределение внешней памяти не применимо до тех пор, пока не известен максимальный размер файла. Иногда размер выходного файла оценить легко (при копировании). Чаще, однако, это трудно сделать. Если места не хватило, то пользовательская программа может быть приостановлена, предполагая выделение дополнительного места для файла при последующем рестарте.

Связный список

Данный метод устраняет внешнюю фрагментацию. Каждый файл - связный список блоков диска (рис.30). Запись в директории содержит указатель на первый и последний блоки файла. Каждый блок содержит указатель на следующий блок.

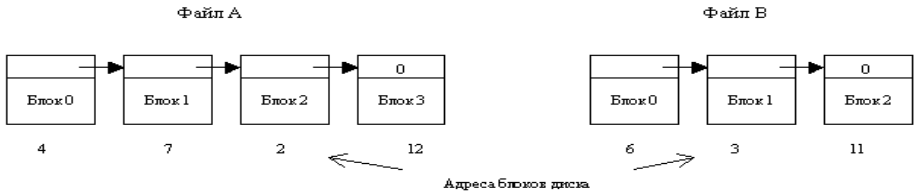

Рис.30 Связный список блоков

Внешняя фрагментация для данного метода отсутствует. Любой свободный блок может быть использован для удовлетворения запроса. Заметим, что нет необходимости декларировать размер файла в момент создания. Файл может неограниченно расти.

Связное выделение имеет, однако, несколько существенных недостатков.

Во-первых, при прямом доступе к файлу для поиска i-го блока нужно осуществить несколько обращений к диску, последовательно считывая блоки от 1 до i-1, то есть выборка логически смежных записей, которые занимают физически несмежные секторы, может требовать много времени.

Прямым следствием этого является низкая надежность. Наличие дефектного блока в списке приводит к потере информации в остаточной части файла и, потенциально, к потере дискового пространства отведенного под этот файл.

Во-вторых, для указателя на следующий блок внутри блока нужно выделить место. Емкость блока, традиционно являющаяся степенью двойки (многие программы читают и пишут блоками по степеням двойки), таким образом, перестает быть степенью двойки, т.к. указатель отбирает несколько байтов.

Поэтому метод связного списка обычно не используется в чистом виде.

Связный список с использованием индекса

Недостатки предыдущего способа могут быть устранены путем изъятия указателя из каждого дискового блока и помещения его в индексную таблицу в памяти, которая называется FAT (file allocation table). Этой схемы придерживаются многие ОС (MS-DOS, OS/2, MS Windows и др.)

Рис. 31 Индексная таблица памяти

По-прежнему, существенно, что запись в директории содержит только ссылку на первый блок и т.о. можно локализовать файл независимо от его размера.

Минусом этой схемы может быть необходимость поддержки в памяти этой довольно большой таблицы.

Индексные узлы

Четвертый метод выяснения принадлежности блока к файлу - связать с каждым файлом маленькую таблицу, называемую индексным узлом (i-node), которая перечисляет атрибуты и дисковые адреса блоков файла (см. рис 32).

Каждый файл имеет свой собственный индексный блок, который содержит адреса блоков данных. Запись в директории, относящаяся к файлу, содержит адрес индексного блока. По мере заполнения файла указатели на блоки диска в индексном узле принимают осмысленные значения.

Индексирование поддерживает прямой доступ к файлу, без ущерба от внешней фрагментации.

Первые несколько адресов блоков файла хранятся непосредственно в индексном узле, т.о. для маленьких файлов индексный узел хранит всю необходимую информацию, которая копируется с диска в память, в момент открытия файла. Для больших файлов один из адресов индексного узла указывает на блок косвенной адресации.

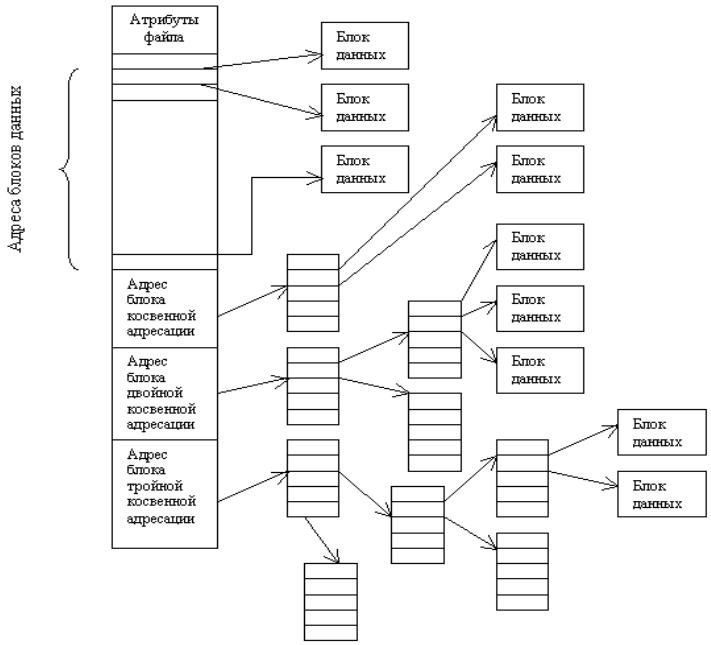

Рис. 32 Индексные узлы

Этот блок содержит адреса дополнительных блоков диска. Если этого недостаточно используется блок двойной косвенной адресации, который содержит адреса блоков косвенной адресации. Если и этого недостаточно используют блок тройной косвенной адресации.

Эту схему использует Unix (а также файловые системы HPFS, NTFS и др.). Такой подход позволяет при фиксированном, относительно небольшом размере индексного узла, поддерживать работу с файлами, размер которых может меняться от нескольких

байт до нескольких гигабайт. Существенно, что для маленьких файлов используется только прямая адресация, обеспечивающая максимальную производительность.

ЛИТЕРАТУРА

1. **Коньков К.А., Карпов В.Е**. Основы операционных систем // Интернет университет электронных технологий.

2.**Таненбаум Э.С.** Современные операционные системы, - СПб.: «Питер», 2007.

3. **Дейтел Г**. Введение в операционные системы: В 2-х т. Т.1. Пер с англ. - М.:Мир, 1998.

4. **Данкан Р.** Профессиональная работа в MS-DOS: Пер. с англ. -М.: Мир, 1993.

5. **Назаров С.В., Широков А.И.** Современные операционные системы. Учеб. пособие. – М.: Интернет-Университет Информационных Технологий

6. **Гордеев А. В**. Операционные системы: Учебник для вузов. — 2-е изд. — СПб.: Питер, 2007.

7. **Деннинг П. Дж., Браун Р. Л.** Операционные системы // Современный компьютер. — М., 1986.

8.**Иртегов Д. В**. Введение в операционные системы. — 2-е изд. — СПб.: BHV-СПб, 2007

9. **Олифер В. Г., Олифер Н. А.** Сетевые операционные системы. — СПб.: Питер, 2002.

10. **Столлингс У.** Операционные системы = Operating Systems: Internals and Design Principles. — М.: Вильямс, 2004.

11. **Таненбаум Э. С.** Многоуровневая организация ЭВМ = Structured Computer Organization. — М.: Мир, 1979.

12. **Таненбаум Э. С., Вудхалл А. С**. Операционные системы. Разработка и реализация = Operating Systems: Design and Implementation. — 3-е изд. — СПб.: Питер, 2007.

13. **Шоу А.** Логическое проектирование операционных систем = The Logical Design of Operating Systems. — М.: Мир, 1981.

14. **Рэймонд Э. С.** Искусство программирования для UNIX = The Art of UNIX Programming. — М.: Вильямс, 2005.

15. **Mark G**. Sobell. UNIX System V. A Practical Guide. — 3rd ed. — 1995.