Некоммерческое акционерное общество

АЛМАТИНСКИЙ УНИВЕРСИТЕТ ЭНЕРГЕТИКИ И СВЯЗИ

Кафедра инженерной кибернетики

СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ И ПРОГРАММИРОВАНИЕ

Конспект лекций
для студентов специальности
5В070200 – Автоматизация и управление

Алматы 2015

СОСТАВИТЕЛЬ: Н.В.Сябина. Системное программное обеспечение и программирование. Конспект лекций для студентов специальности 5В070200 – Автоматизация и управление. – Алматы: АУЭС, 2015. – 70 с.

Настоящий конспект лекций содержит материалы для ознакомления студентов 3 курса со структурой, особенностями функционирования и использования системного программного обеспечения в вычислительных системах, а именно операционными, файловыми системами, а также системами программирования. В приложения включены необходимые иллюстративные и справочные материалы.

Конспект лекций предназначен для студентов всех форм обучения специальности 5В070200 – Автоматизация и управление.

Ил. 21, библиогр. – 9 назв.

Рецензент: ст. преп., канд. техн. наук Мусапирова Г.Д.

Печатается по плану издания некоммерческого акционерного общества «Алматинский университет энергетики и связи» на 2014 г.

© НАО «Алматинский университет энергетики и связи», 2015 г.

Содержание

Введение 4
Лекция №1. Системное программное обеспечение. Основные понятия. 5
Лекция №2. Операционные системы. Их эволюция. Ядро системы 7
Лекция №3. Операционные системы. Назначение и функции 11
Лекция №4. Прерывания 14
Лекция №5. Вычислительные процессы, задачи и ресурсы 18
Лекция №6. Основные виды ресурсов и возможности их разделения 23
Лекция №7. Управление задачами. Дисциплины диспетчеризации 27
Лекция №8. Взаимодействие процессов. Синхронизация. Тупики 32
Лекция №9. Управление памятью в операционных системах. Память и отображение. Неразрывные методы распределения памяти. 35
Лекция №10. Разрывные методы распределения памяти 38
Лекция №11. Управление вводом-выводом в операционных системах 42
Лекция №12. Файлы и организация работы с ними 46
Лекция №13. Файловые системы, их особенности и реализация 50
Лекция №14. Системы программирования 54
Лекция №15. Библиотеки системы программирования 58
Приложение А Операционные системы 62
Приложение Б Файловые системы 68
Список литературы 70

Введение

Любая современная вычислительная система функционирует на базе *аппаратного обеспечения* (hardware) и решает задачи с помощью *программного обеспечения* (software). Достаточно сложная и неоднозначная структура программного обеспечения на самом деле несколько условная и, как правило, производит классификацию программного обеспечения по назначению программ, хотя существуют и другие критерии, характеризующие программное обеспечение (дружественность, тип использования и т.д.).

Подразделение программного обеспечения на системное и прикладное в некоторой степени устарело. Поэтому современная классификация программного обеспечения предусматривает его разделение на *системное, промежуточное* и *прикладное*. Причем, современная тенденция развития программного обеспечения состоит в снижении объема как системного, так и прикладного программирования, а основная часть работы выполняется в промежуточном программном обеспечении.

Промежуточное программное обеспечение (middleware) определяется как совокупность программ, осуществляющих управление вторичными ресурсами, которые ориентированы на решение определенного класса задач. К нему относятся менеджеры транзакций, серверы БД, серверы коммуникаций и другие программные серверы. С точки зрения инструментальных средств промежуточное программное обеспечение ближе к прикладному, т.к. не работает напрямую с первичными ресурсами, а использует для этого сервисы, предоставляемые системным программным обеспечением. С точки же зрения алгоритмов и технологий разработки оно ближе к системному, т.к. всегда является сложным программным изделием многократного и многоцелевого использования, в котором применяются алгоритмы, сходные с применяемыми в системном программном обеспечении.

Снижение объема системного программирования обусловлено современными концепциями построения операционных систем, а также архитектурой ядра, в соответствии с которыми большая часть функций системы выносится в утилиты, которые можно отнести и к промежуточному программному обеспечению. Снижение объема прикладного программирования связано с тем, что современные продукты промежуточного программного обеспечения предлагают все больший набор инструментальных средств и шаблонов для решения задач своего класса. Значительная часть системного и практически все прикладное программное обеспечение пишется на языках высокого уровня, что обеспечивает сокращение расходов на их разработку/модификацию и переносимость.

Целью настоящего конспекта является ознакомление студентов со структурой, особенностями функционирования и использования системного программного обеспечения в вычислительных системах.

Лекция №1. Системное программное обеспечение. Основные понятия

Цель: получить представление о современном состоянии системного программного обеспечения, составе и основных функциях.

Развитие вычислительной техники и информационных технологий неизбежно влечет за собой с одной стороны их усложнение, а с другой стороны - упрощение интерфейсов взаимодействия с пользователем. Пользователи и их программы взаимодействуют с вычислительной техникой посредством *системного* программного обеспечения.

Системное программное обеспечение — это набор специальных программ, которые управляют компонентами вычислительной системы (процессором, коммуникационными и периферийными устройствами), а также предназначены для обеспечения надежного функционирования и работоспособности системы.

Системное программное обеспечение подразделяется на *системные управляющие* и *системные обслуживающие* программы.

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

Программа *обслуживания* (*утилита*) – программа, предназначенная для оказания услуг общего характера пользователям и обслуживающему персоналу вычислительной системы. Управляющая программа с набором утилит, необходимых для эксплуатации системы, составляют *операционную систему* (ОС). Кроме утилит, входящих в состав ОС, могут существовать и другие, выполняющие *опционное* (дополнительное) обслуживание. Обычно это утилиты, обеспечивающие разработку программного обеспечения для ОС.

Центральной частью ОС является *ядро* (*kernel*), обеспечивающее приложениям эффективный доступ к ресурсам компьютера, таким как процессорное время, память, внешнее аппаратное обеспечение, внешние устройства ввода и вывода, а также предоставляет сервисы файловой системы и сетевых протоколов.

К системному программному обеспечению помимо операционных систем относятся *системы программирования*, *операционные оболочки*, *драйверы* и *утилиты*.

Система программирования – это система, образуемая языком программирования, компилятором или интерпретатором программ, представленных на этом языке, соответствующей документацией, а также вспомогательными средствами для подготовки программ к форме, пригодной для выполнения.

Системы программирования, в которых *исходная* вычислительная система (на которой программа готовится к выполнению) отличается от *целевой* (на которой программа будет выполняться) называются *кросс-системами*. Часто кросс-системы применяются для разработки программного обеспечения встроенных вычислительных систем, для которых характерен малый объем ресурсов: ограничение оперативной памяти, отсутствие внешней памяти или необходимого набора внешних устройств. То есть в том случае, когда ресурсов целевой вычислительной системы просто недостаточно для выполнения на ней системного программного обеспечения подготовки программ, например, интерактивных систем программирования с развитым интерфейсом пользователя.

Операционная оболочка – это часть операционной среды (специальная надстройка над операционной системой), основная задача которой упростить работу пользователя с неграфическими ОС. Например, Norton Commander обеспечивал пользователю наглядную работу с MS DOS.

Драйвер – это специальная программа, которая позволяет обеспечить работоспособность различных технических устройств. Простейшие устройства, например, мышка и флэш–карта не требуют установки драйверов, так как в стандартную операционную систему они уже включены по умолчанию.

Утилиты – это вспомогательные или служебные программы, которые представляют пользователю дополнительные услуги (сервисы). Следует отметить, что часть утилит входит в состав операционной системы, а другая - функционирует автономно и поставляется отдельно. К утилитам относятся:

- файловые менеджеры;
- средства динамического сжатия данных;
- средства просмотра и воспроизведения;
- средства диагностики и контроля;
- средства коммуникаций (коммуникационные программы);
- средства обеспечения компьютерной безопасности;

Файловый менеджер (*file manager*) – это программа, которая предоставляет пользовательский интерфейс для работы с файловой системой и файлами, а именно: позволяет выполнять основные операции над файлами (например, создание, открытие, редактирование, перемещение, переименование, копирование, удаление, изменение атрибутов и свойств, поиск файлов, назначение прав). Помимо основных функций многие файловые менеджеры включают ряд дополнительных возможностей, таких как работа с сетью, резервное копирование, управление принтерами и т.д.

Средства динамического сжатия данных (архиваторы) позволяют увеличить объем информации на диске за счет ее динамического сжатия.

Средства диагностики и контроля позволяют проверить конфигурацию компьютера, работоспособность его устройств, например, жестких дисков.

Коммуникационные программы предназначены для организации обмена информацией между компьютерами в сети.

К *средствам обеспечения компьютерной безопасности* относят программы для резервного копирования, антивирусное программное обеспечение.

Таким образом, большая часть общего (системного) программного обеспечения входит в состав ОС. Другая часть входит в состав самого компьютера (некоторые программы ОС и контролирующие тесты записаны в постоянное запоминающее устройство, установленное на системной плате).

Лекция №2. Операционные системы. Их эволюция. Ядро системы

Цель: получить представление о процессе эволюции операционных систем и ядре операционной системы.

С момента появления первых вычислительных систем техническое и программное обеспечения эволюционировали совместно, оказывая взаимное влияние друг на друга. Появление новых технических возможностей приводило к прорыву в области создания удобных, эффективных и безопасных программ, а свежие идеи в программной области стимулировали поиски новых технических решений.

Программирование первых ламповых вычислительных устройств (1945–1955 гг.) осуществлялось исключительно на машинном языке. Об операционных системах не было и речи, все задачи организации вычислительного процесса решались вручную каждым программистом с пульта управления. При этом существенная часть времени уходила на подготовку запуска программы, а сами программы выполнялись строго последовательно (режим последовательной обработки данных). В результате исследований и разработок появляется первое системное программное обеспечение: в 1951–1952 гг. возникают прообразы первых компиляторов с символических языков (Fortran и др.), а в 1954 г. Нат Рочестер разрабатывает Ассемблер для IBM-701.

В середине 50-х годов одновременно с появлением транзисторов наблюдается бурное развитие алгоритмических языков, появляются первые настоящие компиляторы, редакторы связей, библиотеки математических и служебных подпрограмм. Для повышения эффективности использования компьютера задания с похожими ресурсами начинают собирать вместе, создавая *пакет* заданий. Появляются первые *системы пакетной обработки*, которые просто автоматизируют запуск одной программы из пакета за другой и тем самым увеличивают коэффициент загрузки процессора. При реализации систем пакетной обработки был разработан формализованный язык управления заданиями, с помощью которого программист сообщал системе и оператору, какую работу он хочет выполнить на вычислительной машине. Системы пакетной обработки стали прообразом современных операционных систем и первыми системными программами, предназначенными для управления вычислительным процессом.

Переход к интегральным микросхемам (начало 60-х – 1980 гг.) привел к увеличению сложности и количества задач, решаемых компьютерами. Пакетные системы начинают заниматься планированием заданий: в зависимости от наличия запрошенных ресурсов, срочности вычислений и т.д. на счет выбирается то или иное задание. Появление *мультипрограммных систем* обеспечило более эффективное использование системных ресурсов, но еще долго операционные системы оставались пакетными.

Логическим расширением систем мультипрограммирования стали системы *разделения времени* (time-sharing system), в которых процессор переключается между задачами не только на время операций ввода-вывода, но и просто по прошествии определенного времени. Эти переключения происходят так часто, что пользователи могут взаимодействовать со своими программами во время их выполнения, то есть интерактивно. В результате появляется возможность одновременной работы нескольких пользователей на одной компьютерной системе. Использование механизмов виртуальной памяти позволило создать иллюзию неограниченной оперативной памяти. В системах разделения времени пользователь получил возможность эффективно производить отладку программы в интерактивном режиме и записывать информацию на диск непосредственно с клавиатуры. Появление on-line-файлов привело к необходимости разработки развитых файловых систем. В этот же период появилась идея стандартизации операционных систем.

В середине 80-х стали бурно развиваться сети компьютеров, работающих под управлением *сетевых* или *распределенных* операционных систем. В *сетевых операционных системах* пользователи могут получить доступ к ресурсам другого сетевого компьютера. Каждая машина в сети работает под управлением своей локальной операционной системы, отличающейся от операционной системы автономного компьютера наличием дополнительных средств, которые не меняют структуру операционной системы. *Распределенная система*, напротив, внешне выглядит как обычная автономная система. Пользователь не знает и не должен знать, где его файлы хранятся – на локальной или удаленной машине – и где его программы выполняются. Он может вообще не знать, подключен ли его компьютер к сети. Внутреннее строение распределенной операционной системы имеет существенные отличия от *автономных* (*классических*) операционных систем.

Таким образом, выделяют шесть основных функций, которые в процессе эволюции выполняли классические ОС: планирование процессов; обеспечение программ средствами коммуникации и синхронизации; управление памятью; управление файловой системой; управление вводом-выводом; обеспечение безопасности. Каждая из приведенных функций обычно реализована в виде подсистемы, являющейся структурным компонентом ОС. В каждой операционной системе эти функции, конечно, реализовывались по-своему, в различном объеме. Они не были изначально придуманы как составные части операционных систем, а появились в процессе развития. Операционные системы существуют до сих пор потому, что на данный момент их

существование – это разумный способ использования вычислительных систем.

По сути дела, операционная система – это обычная программа, поэтому было бы логично и организовать ее так же, как устроено большинство программ, то есть составить из процедур и функций. В этом случае компоненты операционной системы являются не самостоятельными модулями, а составными частями одной большой программы. Такая структура операционной системы называется *монолитным ядром* (monolithic kernel). Монолитное ядро – старейший способ организации операционных систем.

Монолитное ядро представляет собой набор работающих в привилегированном режиме процедур, каждая из которых может вызвать каждую. Другими словами, *монолитное ядро* – это такая схема операционной системы, при которой все ее компоненты являются составными частями одной программы, используют общие структуры данных и взаимодействуют друг с другом путем непосредственного вызова процедур. Для монолитной операционной системы ядро совпадает со всей системой.

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

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

Продолжая структуризацию, можно разбить всю вычислительную систему на ряд более мелких уровней с хорошо определенными связями между ними, так чтобы объекты уровня N могли вызывать только объекты уровня N-1. Нижним уровнем в таких системах обычно является hardware, верхним уровнем – интерфейс пользователя. Чем ниже уровень, тем более привилегированные команды и действия может выполнять модуль, находящийся на этом уровне. Впервые такой подход был применен при

создании системы THE (Technishe Hogeschool Eindhoven) Дейкстрой и его студентами в 1968 г. Ее уровни показаны на рисунке А.1. *Слоеные* (*многоуровневые - Layered systems*) *системы* хорошо реализуются. При использовании операций нижнего слоя не нужно знать, как они реализованы, нужно лишь понимать, что они делают. Слоеные системы хорошо тестируются, отлаживаются и модифицируются. При необходимости можно заменить лишь один слой, не трогая остальные. Но слоеные системы сложны для разработки: тяжело правильно определить порядок слоев и что отнести к конкретному слою. Слоеные системы менее эффективны, чем монолитные.

Современная тенденция в разработке операционных систем состоит в перенесении значительной части системного кода на уровень пользователя и одновременной минимизации ядра. Речь идет о подходе к построению ядра, называемом *микроядерной архитектурой* (microkernel architecture) операционной системы, когда большинство ее составляющих являются самостоятельными программами. В этом случае взаимодействие между ними обеспечивает специальный модуль ядра, называемый *микроядром*, которое работает в привилегированном режиме и обеспечивает взаимодействие между программами, планирование использования процессора, первичную обработку прерываний, операции ввода-вывода и базовое управление памятью (рисунок А.2). Остальные компоненты системы взаимодействуют друг с другом путем передачи сообщений через микроядро.

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

Все рассмотренные подходы к построению операционных систем имеют свои достоинства и недостатки. В большинстве случаев современные операционные системы используют различные комбинации этих подходов – *смешанный подход*.

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

представляют собой *гибридные операционные системы*, поскольку в их ядрах тесно переплетены элементы микроядерной архитектуры и монолитного ядра.

Лекция №3. Операционные системы. Назначение и функции

Цель: ознакомиться с назначением, основными функциями и классификацией операционных систем, а также с понятием и особенностями операционной среды.

Операционная система (ОС) представляет собой комплекс системных управляющих и обрабатывающих программ, которые, с одной стороны выступают как интерфейс между аппаратурой компьютера и пользователем с его задачами, а с другой стороны, предназначены для наиболее эффективного расходования ресурсов вычислительной системы и организации надежных вычислений. Под *интерфейсом* понимают комплекс спецификаций, определяющих конкретный способ взаимодействия пользователя с компьютером. Основные функции ОС:

1) *прием от пользователя (или от оператора системы) заданий, или команд, сформулированных на соответствующем языке, и их обработка*. Задания могут передаваться в виде текстовых директив (команд) оператора или в форме указаний, выполняемых с помощью манипулятора (мыши);
2) *загрузка в оперативную память подлежащих исполнению программ*;
3) *распределение памяти, а в большинстве систем и организация виртуальной памяти*;
4) *запуск программы* (передача ей управления, в результате чего процессор исполняет программу);
5) *идентификация всех программ и данных*;
6) *прием и исполнение различных запросов от выполняющихся приложений*. ОС выполняет большое количество системных функций (сервисов), которые могут быть запрошены из выполняющейся программы. Обращение к этим сервисам осуществляется по соответствующим правилам, которые и определяют *интерфейс прикладного программирования* (API – Application Program Interface) этой ОС;
7) *обслуживание всех операций ввода-вывода*;
8) *обеспечение работы систем управлений файлами (СУФ) и/или систем управления базами данных (СУБД)*, что позволяет резко увеличить эффективность всего программного обеспечения;
9) *обеспечение режима мультипрограммирования*, т.е. организация выполнения двух или более программ на одном процессоре, создающая видимость их одновременного исполнения;
10) *планирование и диспетчеризация задач в соответствии с заданными стратегией и дисциплинами обслуживания*;
11) *организация механизмов обмена сообщениями и данными между выполняющимися программами*;

12) для сетевых ОС характерной является *функция обеспечения взаимодействия связанных между собой компьютеров*;

13) *защита одной программы от влияния другой, обеспечение сохранности данных, защита самой ОС от исполняющихся на компьютере приложений*;

14) *аутентификация и авторизация пользователей* (для большинства диалоговых ОС). Под *аутентификацией* понимается процедура проверки имени пользователя и его пароля на соответствие тем значениям, которые хранятся в его учетной записи (т.е. если входное имя (login) и пароль совпадают, то это и есть тот самый пользователь). Термин «*авторизация*» означает, что в соответствии с учетной записью пользователя, который прошел аутентификацию, ему назначаются определенные права (привилегии), определяющие, что ему разрешается и запрещается делать на компьютере;

15) *удовлетворение жестким ограничениям на время ответа в режиме реального времени* (характерно для ОС реального времени);

16) *обеспечение работы систем программирования*, с помощью которых пользователи готовят свои программы;

17) *предоставление услуг на случай частичного сбоя системы*.

ОС изолирует аппаратное обеспечение компьютера от прикладных программ пользователей. И пользователь, и его программы взаимодействуют с компьютером через интерфейсы ОС. Сформировалось относительно небольшое количество классификаций операционных систем: по назначению, по режиму обработки задач, по способу взаимодействия с системой и, наконец, по способам построения (архитектурным особенностям системы).

Традиционно различают ОС *общего* и *специального назначения*. ОС специального назначения, в свою очередь, подразделяются на ОС для носимых микрокомпьютеров и различных встроенных систем, организации и ведения баз данных, решения задач реального времени и т. п. Еще не так давно операционные системы для персональных компьютеров относили к ОС специального назначения. Современные мультизадачные ОС для персональных компьютеров относят к ОС общего назначения, поскольку их можно использовать для самых разнообразных целей.

По режиму обработки задач различают ОС, обеспечивающие однопрограммный и мультипрограммный (мультизадачный) режимы. К однопрограммным ОС относится, например, известная MS DOS. Мультипрограммный режим обеспечивает параллельное выполнение нескольких приложений, и при этом заботу о механизмах организации их параллельной работы берет на себя сама ОС, которая распределяет между выполняющимися приложениями ресурсы вычислительной системы, осуществляет необходимую синхронизацию вычислений и взаимодействие. Мультизадачный режим, наоборот, предполагает, что забота о параллельном выполнении и взаимодействии приложений ложится на прикладных программистов. Современные ОС для персональных компьютеров реализуют и мультипрограммный, и мультизадачный режимы.

Если принимать во внимание *способ взаимодействия* с компьютером, то можно говорить о диалоговых системах и системах пакетной обработки. Доля последних, хоть и не убывает в абсолютном исчислении, но в процентном отношении она существенно сократилась по сравнению с диалоговыми системами. При организации работы с вычислительной системой в диалоговом режиме говорят об однопользовательских (однотерминальных) и мультитерминальных ОС. В мультитерминальных ОС с одной вычислительной системой одновременно могут работать несколько пользователей, каждый со своего терминала. При этом у пользователей возникает иллюзия, что у каждого из них имеется собственная вычислительная система. Для организации мультитерминального доступа к вычислительной системе необходимо обеспечить мультипрограммный режим работы. В качестве примера мультитерминальной операционной системы для персональных компьютеров можно назвать Linux. Имитация мультитерминальных возможностей имеется и в системе Windows XP, поскольку каждый пользователь после регистрации получает свою виртуальную машину. Количество параллельно работающих виртуальных машин определяется имеющимися ресурсами.

Основной особенностью операционных систем *реального времени* (ОСРВ) является обеспечение обработки поступающих заданий в течение заданных интервалов времени, которые нельзя превышать. Поток заданий в общем случае не является планомерным и не может регулироваться оператором, так как задания поступают в непредсказуемые моменты времени и без всякой очередности. В ОСРВ накладные расходы процессорного времени на этапе инициирования задач могут отсутствовать, так как набор задач обычно фиксирован, и вся информация о них известна еще до поступления запросов. Лучшие характеристики по производительности для систем реального времени обеспечиваются однотерминальными ОСРВ. Средства организации мультитерминального режима всегда замедляют работу системы в целом, но расширяют функциональные возможности системы. Одной из наиболее известных ОСРВ для персональных компьютеров является ОС QNX.

По основному архитектурному принципу операционные системы разделяются на микроядерные и макроядерные (монолитные). В некоторой степени это разделение тоже условно, однако в качестве яркого примера микроядерной ОС можно привести ОСРВ QNX, тогда как в качестве монолитной можно назвать Windows 95/98 или ОС Linux.

Операционной средой называется:

- программная среда, в которой выполняются прикладные программы пользователей, образуемая операционной системой;
- соответствующие интерфейсы, необходимые программам и пользователям для обращения к управляющей (супервизорной) части ОС с целью получить определенные сервисы;
- системное программное окружение, в котором могут выполняться программы, созданные по правилам работы этой среды.

Системные функции определяют те возможности, которые ОС предоставляет выполняющимся под ее управлением приложениям. Системные запросы (вызовы системных операций или функций) либо явно прописываются в тексте программы программистами, либо подставляются автоматически самой системой программирования на этапе трансляции исходного текста разрабатываемой программы. Каждая ОС имеет свое множество системных функций, которые вызываются по принятым в системе правилам. Совокупность системных вызовов и правил, по которым следует их использовать, определяет *интерфейс прикладного программирования* (API). Поэтому программа, созданная для работы в некоторой ОС, не будет работать в другой, поскольку API у этих систем, как правило, различаются. Для преодоления этого ограничения создаются программные среды.

Программная (системная) среда – это некоторое системное программное окружение, позволяющее выполнять все системные запросы от прикладной программы. Различают *основную* (*нативную – native*) системную программную среду, которая непосредственно образуется кодом ОС, и *дополнительную* системную программную среду, которая организована путем эмуляции другой ОС. Если в ОС организована работа с различными операционными средами, то в такой системе можно выполнять программы, созданные не только для данной, но и для других операционных систем. Операционная среда может включать несколько пользовательских и программных интерфейсов.

Лекция №4. Прерывания

Цель: ознакомиться с механизмом прерываний, используемым в операционных системах.

Прерывания представляют собой механизм, позволяющий координировать параллельное функционирование отдельных устройств вычислительной системы и реагировать на особые состояния, возникающие при работе процессора, т.е. – это принудительная передача управления от выполняемой программы к системе (а через нее – к соответствующей программе обработки прерывания), происходящая при возникновении определенного события.

Основная *цель введения прерываний* – реализация асинхронного режима функционирования и распараллеливание работы отдельных устройств вычислительного комплекса.

Механизм прерываний реализуется аппаратно-программными средствами. Структуры систем прерывания (в зависимости от аппаратной архитектуры) могут быть самыми разными, но имеют одну общую особенность – прерывание непременно влечет за собой изменение порядка выполнения команд процессором.

Механизм обработки прерываний независимо от архитектуры вычислительной системы подразумевает выполнение некоторой последовательности действий:

- *шаг 1*: установление факта прерывания (прием сигнала запроса на прерывание) и идентификация прерывания;
- *шаг 2*: запоминание состояния прерванного процесса вычислений. Состояние процесса выполнения программы определяется значением счетчика команд, содержимым регистров процессора, а также может содержать спецификацию режима и другую информацию;
- *шаг 3*: управление аппаратно передается на подпрограмму обработки прерывания. В простейшем случае в счетчик команд заносится начальный адрес подпрограммы обработки прерываний, а в соответствующие регистры – информация из слова состояния;
- *шаг 4*: сохранение информации о прерванной программе, которую не удалось спасти (на шаге 2) с помощью аппаратуры;
- *шаг 5*: собственно выполнение программы, связанной с обработкой прерывания. Эта работа может быть выполнена той же подпрограммой, на которую было передано управление на шаге 3, но в ОС достаточно часто она реализуется путем последующего вызова соответствующей подпрограммы;
- *шаг 6*: восстановление информации, относящейся к прерванному процессу (этап, обратный шагу 4);
- *шаг 7*: возврат в прерванную программу.

Шаги 1-3 реализуются аппаратно, шаги 4-7 – программно.

На рисунке 4.1 показано, что при возникновении запроса на прерывание естественный ход вычислений нарушается и управление передается на программу обработки возникшего прерывания. При этом средствами аппаратуры сохраняется (с помощью механизмов стековой памяти) адрес той команды, с которой следует продолжить выполнение прерванной программы. После выполнения программы обработки прерывания управление возвращается на прерванную ранее программу посредством занесения в указатель команд сохраненного адреса команды, которую нужно было бы выполнить, если бы не возникло прерывание. Однако такая схема используется только в самых простых программных средах. В мультипрограммных ОС обработка прерываний происходит по более сложным схемам.

Главные функции механизма прерываний:

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

Переход от прерываемой программы к обработчику и обратно должен выполняться как можно быстрей. Одним из самых простых и быстрых методов является использование таблицы, содержащей перечень всех допустимых прерываний и адреса соответствующих обработчиков. Для корректного возвращения к прерванной программе перед передачей

управления обработчику прерываний содержимое регистров процессора запоминается либо в памяти с прямым доступом, либо в *системном стеке* (system stack).

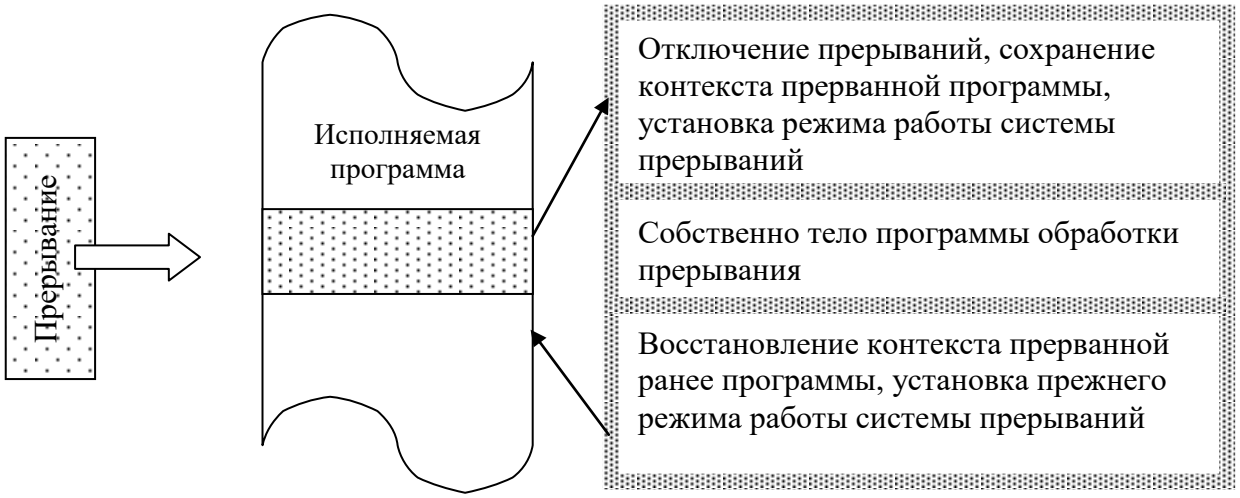

Рисунок 4.1 - Обработка прерывания

Прерывания, возникающие при работе вычислительной системы, можно разделить на два основных класса: *внешние* (*асинхронные*) и *внутренние* (*синхронные*).

Внешние прерывания вызываются асинхронными событиями, которые происходят вне прерываемого процесса:

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

Внутренние прерывания вызываются событиями, которые связаны с работой процессора и являются синхронными с его операциями:

1) при нарушении адресации;
2) при наличии в поле кода операции незадействованной двоичной комбинации;
3) при делении на ноль;
4) вследствие переполнения или исчезновения порядка;
5) от средств контроля (обнаружение ошибки четности, ошибок в работе различных устройств);
6) попытка использования команды, запрещенной в данном режиме (привилегированные команды – команды ввода-вывода, переключения режима работы центрального процессора, команды инициализации некоторых системных регистров процессора).

Программные прерывания - это прерывания, которые происходят по соответствующей команде прерывания, т.е. процессор осуществляет практически те же действия, что и при обычных внутренних прерываниях.

Этот механизм введен специально для того, чтобы, прежде всего, обеспечить автоматическое переключение процессора в привилегированный режим с возможностью исполнения любых команд.

Сигналы, вызывающие прерывания, формируются вне процессора или в самом процессоре и могут возникать одновременно. Выбор одного из них для обработки осуществляется на основе приоритетов, приписанных каждому прерыванию. На рисунке А.3 изображен обычный порядок (*приоритеты*) обработки прерываний в зависимости от типа прерываний.

Учет приоритета может быть встроен в технические средства, а также определяться ОС, т.е. допускается программно-аппаратное управление порядком обработки сигналов прерывания. Другой способ позволяет применять различные дисциплины обслуживания прерываний.

Наличие сигнала прерывания не обязательно должно вызывать прерывание исполняющейся программы. Процессор может обладать средствами защиты от прерываний: отключение системы прерываний, маскирование (запрет) отдельных сигналов прерывания. Программное управление этими средствами позволяет ОС регулировать обработку сигналов прерывания, заставляя процессор обрабатывать их сразу по приходу; откладывать обработку на некоторое время; полностью игнорировать прерывания. Обычно операция прерывания выполняется только после завершения выполнения текущей команды. Поскольку сигналы прерывания возникают в произвольные моменты времени, то на момент прерывания может существовать несколько сигналов, которые могут быть обработаны только последовательно в соответствии с назначенными приоритетами. Программное управление *специальными регистрами маски* (маскирование сигналов прерывания) позволяет реализовать различные дисциплины обслуживания:

- *с относительными приоритетами*, т.е. обслуживание не прерывается даже при наличии запросов с более высокими приоритетами. После окончания обслуживания данного запроса обслуживается запрос с наивысшим приоритетом для организации такой дисциплины необходимо в программе обслуживания данного запроса наложить маски на все остальные сигналы прерывания или просто отключить систему прерываний;

- *с абсолютными приоритетами*, т.е. всегда обслуживается прерывание с наивысшим приоритетом. На время обработки маскируются все запросы с более низким приоритетом, причем возможно многоуровневое прерывание (т.е. прерывание программ обработки прерываний);

- *по принципу стека* (по дисциплине LCFS – Last Come First Served – последним пришел, первым обслужен), т.е. запросы с более низким приоритетом могут прерывать обработку прерывания с более высоким приоритетом. Для этого маска не накладывается ни на один из сигналов прерывания и не выключается система прерываний.

Для правильной реализации последних двух дисциплин нужно обеспечить полное маскирование системы прерываний при выполнении шагов

1-4 и 6-7. Это необходимо для того, чтобы не потерять запрос и правильно его обслужить. Многоуровневое прерывание должно происходить на этапе обработки прерывания, а не на этапе перехода с одного процесса вычислений на другой.

Лекция №5. Вычислительные процессы, задачи и ресурсы

Цель лекции: ознакомиться с назначением и функциями вычислительных процессов и задач, классификацией ресурсов и общей схемой их выделения, с диаграммой состояний процесса и его дескрипторами, а также с основными принципами мультипрограммирования, многопользовательского режима работы и режима разделения времени.

Последовательный процесс (*задача – task*) – это отдельная программа с ее данными, выполняющаяся на последовательном процессоре (т.е. таком процессоре, в котором текущая команда выполняется после завершения предыдущей).

В современных процессорах для повышения скорости вычислений возможно параллельное выполнение нескольких команд, что достигается двумя основными способами – *организацией конвейерного механизма выполнения команды* и *созданием нескольких конвейеров*. Однако в подобных процессорах обязательно достигается логическая последовательность в выполнении команд, предусмотренная программой.

Концепция процесса предполагает два аспекта:

1) процесс является носителем данных;
2) процесс выполняет задачи, связанные с обработкой данных.

В качестве примеров процессов (задач) можно назвать прикладные программы пользователей, утилиты и другие системные обрабатывающие программы. Процессом может быть редактирование текста, трансляция исходной программы, ее компоновка, исполнение. Трансляция какой-нибудь исходной программы является одним процессом, а трансляция следующей исходной программы – другим, т.к. транслятор выступает как одна и та же программа, но обрабатываемые данные являются разными. Основная цель процесса - выработать механизмы распределения и управления ресурсами.

Ресурсы – это многократно используемые, относительно стабильные и часто недостающие объекты, которые запрашиваются, задействуются или освобождаются в период их активности, т.е. всякий объект, который может распределяться внутри системы. Ресурсы могут быть *разделяемыми*, когда несколько процессов используют их *одновременно* или *параллельно* (попеременно в течение некоторого интервала времени), и *неделимыми* (рисунок А.4).

В первых вычислительных системах любая программа могла выполняться только после полного завершения предыдущей. Поскольку все эти системы были построены в соответствии с принципами фон Неймана, все

подсистемы и устройства компьютера управлялись исключительно центральным процессором. Соответственно, пока осуществлялся обмен данными между оперативной памятью и внешними устройствами, процессор не мог выполнять вычисления. Введение в состав вычислительной машины специальных контроллеров позволило *совместить во времени* (*распараллелить*) операции вывода полученных данных и последующие вычисления на центральном процессоре. Однако все равно процессор часто и долго простаивал, дожидаясь завершения очередной операции ввода-вывода. Поэтому возникла необходимость в организации *мультипрограммного* (*мультизадачного*) режима работы.

Суть *мультипрограммного* режима работы заключается в том, что пока одна программа ожидает завершения очередной операции ввода-вывода, другая задача может быть поставлена на решение. На рисунке А.5 изображена ситуация, при которой благодаря совмещению во времени двух вычислительных процессов общее время их выполнения получается меньше, чем, если бы их выполняли по очереди. При мультипрограммировании повышается пропускная способность системы, но отдельный процесс никогда не может быть выполнен быстрее, чем если бы он выполнялся в однопрограммном режиме (всякое разделение ресурсов замедляет работу одного из участников за счет дополнительных затрат времени на ожидание освобождения ресурса).

Под *коэффициентом мультипрограммирования* (*m*) понимают количество параллельно выполняемых программ. Для загрузки центрального процессора до уровня 90% необходимо, чтобы m был не менее 4-5, а чтобы наиболее эффективно использовать и остальные ресурсы системы, желательно иметь m на уровне 10-15.

Хотя понятия мультипрограммного и мультизадачного режимов работы достаточно близки, это не одно и то же. *Мультипрограммный режим* предполагает, что ОС организует параллельное выполнение нескольких вычислительных процессов на одном компьютере, но при этом они не зависят друг от друга, у них не может быть ни общих файлов, ни общих переменных, они могут принадлежать разным пользователям. Каждому процессу выделяются затребованные ресурсы и он выполняется как бы на отдельной виртуальной машине. Средства защиты системы должны обеспечить невмешательство одного процесса в другой. При необходимости совместить процессы или обеспечить возможность тесного взаимодействия между осуществляемыми вычислениями используется *мультизадачный режим*, который предполагает организацию параллельного выполнения вычислений при наличии специальных механизмов для передачи данных, синхросигналов, каких-либо сообщений между взаимодействующими вычислениями.

Упрощение пользовательского интерфейса и развитие интерфейсных функций ОС позволило реализовать *диалоговый* режим работы, который предполагает взаимодействие пользователя с компьютером без посредника.

Но этот режим может быть реализован и без мультипрограммирования (например, ОС PC-DOS).

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

Общая схема выделения ресурсов. При необходимости использовать какой-либо ресурс (ОП, УВВ, массив данных и т.п.) задача путем обращения к супервизору ОС посредством специальных вызовов (команд, директив) сообщает о своем требовании. При этом указывается вид ресурса, и если нужно, его объем. Команда обращения к ОС передает ей управление, переводя процессор в привилегированный режим работы, если такой существует.

Большинство компьютеров имеют два (и более) режимов работы: привилегированный (режим супервизора) и пользовательский. Ресурс может быть выделен задаче, обратившейся к ОС с соответствующим запросом, если:

1) ресурс свободен и в системе нет запросов от задач более высокого приоритета к этому же ресурсу;

2) текущий запрос и ранее выданные запросы допускают совместное использование ресурсов;

3) ресурс используется задачей низшего приоритета и может быть временно отобран (разделяемый ресурс).

Получив запрос, ОС либо удовлетворяет его и возвращает управление задаче, либо, если ресурс занят, ставит задачу в очередь к ресурсу, переводя ее в состояние ожидания (блокируя).

После окончания работы с ресурсом задача опять с помощью специального вызова супервизора сообщает ОС об отказе от ресурса, либо ОС забирает ресурс сама, если управление возвращается супервизору после выполнения какой-либо системной функции.

Супервизор освобождает ресурс и проверяет, имеется ли очередь к освободившемуся ресурсу. Если очередь есть, то в зависимости от принятой дисциплины обслуживания и приоритетов заявок, он выводит из состояния ожидания задачу, ожидающую ресурс, и переводит ее в состояние готовности к выполнению, после чего, либо передает ей управление, либо возвращает управление задаче, только что освободившей ресурс.

При выдаче запроса на ресурс задача может указать, хочет ли она владеть ресурсом монопольно или допускает совместное использование с другими задачами.

Если в системе имеется некоторая совокупность ресурсов, то управлять их использованием можно на основе некоторой стратегии, которая

подразумевает четкую формулировку целей, следуя которым можно добиться эффективного распределения ресурсов.

Процесс может находиться в *активном* и *пассивном* состоянии. В активном состоянии процесс может конкурировать за ресурсы вычислительной системы, а в пассивном состоянии он известен системе, но за ресурсы не конкурирует. В свою очередь, активный процесс может быть в одном из состояний:

1) *выполнения* – все затребованные процессом ресурсы выделены;

2) *готовности к выполнению* – ресурсы могут быть представлены, тогда процесс перейдет в состояние выполнения;

3) *блокирования* (*ожидания*) – затребованные ресурсы не могут быть предоставлены, или не завершена операция ввода-вывода.

Для ОСРВ характерно наличие пассивного состояния процесса, которого нет в большинстве других ОС. Поскольку при проектировании систем реального времени состав выполняемых ею задач и их параметры известны заранее, для них заранее создают *дескрипторы задач* (процессов) – специальную информационную структуру, сопровождающую каждый процесс, чтобы не тратить время на организацию дескрипторов и поиски для них необходимых ресурсов. Таким образом, многие процессы в ОСРВ находятся в состоянии *бездействия*.

Процесс из состояния бездействия в состояние готовности может перейти в случаях: по команде оператора: при выборе из очереди планировщиком; при вызове из другой задачи; по прерыванию от внешнего инициативного устройства; при наступлении запланированного запуска программы.

Из состояния выполнения процесс может выйти по одной из причин:

1) процесс завершается, при этом он посредством обращения к супервизору передает управление ОС и сообщает о своем завершении. В результате этих действий супервизор переводит его в состояние бездействия или уничтожает;

2) процесс переводится супервизором ОС в состояние готовности в связи с появлением более приоритетных задач или в связи с окончанием выделенного ему кванта времени;

3) процесс блокируется (переводится в состояние ожидания) либо вследствие операции ввода-вывода, либо по причине невозможности предоставить ему ресурс, запрошенный в настоящий момент, либо по команде оператора на приостановление задачи, либо по требованию через супервизор от другой задачи.

При наступлении соответствующего события процесс *деблокируется* и переводится в состояние готовности к исполнению.

Для управления процессами ОС должна располагать необходимой информацией. С этой целью на каждый процесс заводится специальная информационная структура, называемая *дескриптором процесса* (описателем

процесса, блоком управления задачей). Обычно дескриптор содержит следующую информацию:

- *идентификатор* процесса;
- *тип* (*класс*) процесса, который определяет для супервизора некоторые правила предоставления ресурсов;
- *приоритет* процесса, в соответствии с которым супервизор предоставляет ресурсы;
- *переменную состояния*, которая определяет, в каком состоянии находится процесс;
- *контекст задачи*, т.е. защищенную область памяти, в которой хранятся текущие значения регистров процессора, когда процесс прерывается, не закончив работы;
- *информацию о ресурсах*, которыми владеет процесс и/или имеет право пользоваться;
- *место* (*адрес*) для организации общения с другими процессами;
- *параметры времени запуска*;
- *адрес задачи на диске* в ее исходном состоянии в случае отсутствия системы управления файлами и адрес на диске, куда она выгружается из ОП, если ее вытесняет другая задача.

Дескрипторы задач постоянно находятся в оперативной памяти с целью ускорения работы супервизора, который организует их в *списки* (*очереди*) и отображает изменение состояния процесса. В некоторых ОС количество дескрипторов определяется жестко и заранее (например, в конфигурационном файле).

Понятие процесса было введено для реализации идей мультипрограммирования. Для реализации мультизадачности характерны понятия *легковесных процессов – потоков выполнения*, *нитей* или *тредов*. Каждый процесс всегда состоит, по крайней мере, из одного *потока выполнения*, и только если имеется внутренний параллелизм, программист может «расщепить» один поток на несколько параллельных. Все потоки имеют одно и то же виртуальное адресное пространство своего процесса. Это означает, что они разделяют одни и те же глобальные переменные. Поскольку каждый поток может иметь доступ к каждому виртуальному адресу, один поток может использовать стек другого потока. Собственными являются программный счетчик, стек, рабочие регистры процессора, потоки-потомки, состояние. Вследствие того, что потоки, относящиеся к одному процессу, выполняются в одном и том же виртуальном адресном пространстве, между ними легко организовать тесное взаимодействие, в отличие от процессов, для которых нужны специальные механизмы обмена сообщениями и данными. Более того, программист, создающий многопоточное приложение, может заранее продумать работу множества потоков процесса таким образом, чтобы они могли взаимодействовать наиболее выгодным способом, а не конкурировать за ресурсы тогда, когда этого можно избежать.

Для того, чтобы можно было эффективно организовать параллельное выполнение рассмотренных сущностей (процессов и потоков), в архитектуру современных процессоров включены средства для работы со специальной информационной структурой, описывающей ту или иную сущность. Для этого уже на уровне архитектуры микропроцессора используется понятие *задача* (*task*). Оно как бы объединяет в себе и обычный процесс, и поток выполнения (тред). Это понятие и поддерживаемая для него на уровне аппаратуры информационная структура позволяют в дальнейшем при разработке операционной системы строить соответствующие дескрипторы как для задач, так и для процессов. И отличаться эти дескрипторы будут, прежде всего тем, что дескриптор задачи может хранить только контекст приостановленного вычислительного процесса, тогда как дескриптор процесса должен содержать поля, описывающие тем или иным способом ресурсы, выделенные этому процессу.

Лекция №6. Основные виды ресурсов и возможности их разделения

Цель: рассмотреть основные виды ресурсов, их особенности и возможности их разделения.

Рассмотрим кратко основные виды ресурсов вычислительной системы и способы их разделения.

Прежде всего, одним из важнейших ресурсов является сам *процессор*, точнее — *процессорное время*. Процессорное время делится попеременно (параллельно). Имеется множество методов разделения этого ресурса.

Вторым видом ресурсов вычислительной системы можно считать *память*. Память — очень интересный вид ресурса. Дело в том, что в каждый конкретный момент времени процессор при выполнении вычислений обращается к очень ограниченному числу ячеек оперативной памяти. С этой точки зрения желательно память выделять для возможно большего числа параллельно исполняемых задач. С другой стороны, как правило, чем больше оперативной памяти может быть выделено для конкретного текущего вычислительного процесса, тем лучше будут условия его выполнения. Поэтому проблема эффективного разделения оперативной памяти между параллельно выполняемыми вычислительными процессами является одной из самых актуальных.

Оперативная память (ОП) может делиться и одновременно (то есть в памяти одновременно может располагаться несколько задач или, по крайней мере, текущих фрагментов, участвующих в вычислениях), и попеременно (в разные моменты оперативная память может предоставляться для разных вычислительных процессов). Достаточно подробно вопросы распределения памяти между параллельно выполняющимися процессами рассмотрены позже.

Внешняя память тоже является ресурсом, который часто необходим для выполнения вычислений. Когда говорят о внешней памяти (например, памяти на магнитных дисках), то собственно *память* и *доступ* к ней считаются разными видами ресурса. Каждый из этих ресурсов может предоставляться независимо от другого. Но для полноценной работы с внешней памятью необходимо иметь оба этих ресурса. Собственно внешняя память может разделяться и одновременно, а вот доступ к ней всегда разделяется попеременно.

Если говорить о *внешних устройствах*, то они, как правило, могут разделяться параллельно, если используются механизмы прямого доступа. Если же устройство работает с последовательным доступом, то оно не может считаться разделяемым ресурсом.

Простыми и наглядными примерами внешних устройств, которые не могут быть разделяемыми, являются принтер и накопитель на магнитной ленте. Действительно, если допустить, что принтер можно разделять между двумя процессами, которые смогут его использовать (управлять его работой) попеременно, то результаты печати, скорее всего, окажутся негодными — фрагменты выведенного текста могут перемешаться таким образом, что будет не понятно, что есть что. Аналогично и для накопителя на магнитной ленте. Если один процесс начнет что-то читать или писать, а второй при этом запросит перемотку ленты на ее начало, то оба вычислительных процесса не смогут выполнить свои вычисления. Здесь следует заметить, что при работе с устройствами печати мы, тем не менее, явно наблюдаем возможность печатать из разных программ, выполняющихся параллельно. Однако необходимо знать, что это реализуется за счет того, что каждый вычислительный процесс получает свой виртуальный принтер, который он ни с кем не разделяет. Операционная система, получив задания на печать от выполняющихся задач, сама упорядочивает эти задания и передает очередное задание на принтер только после полного завершения предыдущего задания.

Очень важным видом ресурсов являются *программные модули*. Прежде всего, рассмотрим *системные программные модули*, поскольку именно они обычно считаются программными ресурсами и поэтому в принципе могут распределяться между выполняющимися процессами.

Как известно, программные модули могут быть *однократно используемыми* и *многократно* (или *повторно*) *используемыми*.

Однократно используемыми называют такие программные модули, которые могут быть правильно выполнены только один раз, то есть в процессе своего выполнения они могут испортить себя: либо повреждается часть кода, либо исходные данные, от которых зависит ход вычислений. Очевидно, что однократно используемые программные модули являются неделимым ресурсом. Более того, их, как правило, вообще не распределяют как ресурс системы. Системные однократно используемые программные модули, как правило, задействуются только на этапе загрузки операционной системы.

Собственно двоичные файлы, которые обычно хранятся на системном диске и в которых и записаны однократно используемые модули, не портятся, а потому могут быть повторно использованы при следующем запуске операционной системы. *Повторно используемые* программные модули, в свою очередь, могут быть непривилегированными, привилегированными и реентерабельными. Все они допускают корректное повторное выполнение программного кода при обращении к нему из другой программы.

Привилегированные программные модули работают в так называемом привилегированном режиме, то есть при отключенной системе прерываний (часто говорят, что прерывания закрыты), когда никакие внешние события не могут нарушить естественный порядок вычислений. Как результат, программный модуль выполняется до своего конца, после чего он может быть вновь вызван на исполнение из другой задачи (другого вычислительного процесса). С позиций стороннего наблюдателя по отношению к вычислительным процессам, которые попеременно (причем, возможно, неоднократно) в течение срока своей «жизни» вызывают некоторый привилегированный программный модуль, такой модуль будет выступать как попеременно разделяемый ресурс. В первой секции программного модуля выключается система прерываний. Следовательно, при выполнении вычислений в первой секции ничто не может их прервать, и беспокоиться о промежуточных переменных нет необходимости. В последней секции, напротив, система прерываний включается. Даже если тут же возникнет прерывание и другой процесс запросит этот же привилегированный модуль, все равно все вычисления уже выполнены и ничто не сможет их испортить.

Непривилегированные программные модули - это обычные программные модули, которые могут быть прерваны во время своей работы. Следовательно, такие модули в общем случае нельзя считать разделяемыми, потому что если после прерывания выполнения такого модуля, исполняемого в рамках одного вычислительного процесса, запустить его еще раз по требованию другого вычислительного процесса, то промежуточные результаты для прерванных вычислений могут быть потеряны.

В противоположность этому, *реентерабельные* программные модули допускают повторное многократное прерывание своего исполнения и повторный их запуск по обращению из других задач (вычислительных процессов). Для этого реентерабельные программные модули должны быть созданы таким образом, чтобы было обеспечено сохранение промежуточных результатов для прерываемых вычислений и возврат к ним, когда вычислительный процесс возобновляется с прерванной ранее точки. Это может быть реализовано двумя способами: с помощью *статических* и *динамических* методов выделения памяти под сохраняемые значения. Основным и наиболее часто используемым является динамический способ выделения памяти для сохранения всех промежуточных результатов вычисления, относящихся к реентерабельному программному модулю.

Основная идея построения и работы реентерабельного программного модуля заключается в том, что в первой (головной) своей части путем обращения из системной привилегированной секции осуществляется запрос на получение в системной области памяти блока ячеек, необходимого для размещения всех текущих (промежуточных) данных. При этом на вершину стека помещается указатель на начало области данных и ее объем. Все текущие переменные реентерабельного программного модуля в этом случае располагаются в системной области памяти. Адресация этих переменных осуществляется относительно вершины стека. Поскольку в конце привилегированной секции система прерываний включается, то во время работы центральной (основной) части реентерабельного модуля возможно ее прерывание. Если прерывания не возникает, то в третьей (заключительной) секции осуществляется запрос на освобождение используемого блока системной области памяти.

При освобождении этой области памяти модифицируется значение стека. Если же во время работы центральной секции возникает прерывание, и другой вычислительный процесс обращается к тому же самому реентерабельному программному модулю, то для этого нового процесса вновь заказывается новый блок памяти в системной области памяти, и на вершину стека записывается новый указатель. Очевидно, что возможно многократное повторное вхождение в реентерабельный программный модуль до тех пор, пока в области системной памяти, выделяемой специально для реентерабельной обработки, есть свободные области, объема которых достаточно для выделения нового блока.

Что касается статического способа выделения памяти, то здесь речь может идти, например, о том, что заранее для фиксированного числа вычислительных процессов резервируются области памяти, в которых будут располагаться переменные реентерабельных программных модулей: для каждого процесса - своя область памяти. Чаще всего в качестве таких процессов выступают процессы ввода-вывода, и речь идет о реентерабельных драйверах.

Кроме реентерабельных программных модулей существуют еще *повторно входимые* (*re-entrance*). Этим термином называют программные модули, которые тоже допускают свое многократное параллельное использование, но, в отличие от реентерабельных, их нельзя прерывать. Повторно входимые программные модули состоят из привилегированных секций, и повторное обращение к ним возможно только после завершения какой-нибудь из этих секций. После выполнения очередной такой привилегированной секции управление может быть передано супервизору, который может предоставить возможность выполняться другой задаче, а значит, становится возможным повторное вхождение в рассматриваемый программный модуль. Другими словами, в повторно входимых программных модулях четко предопределены все *допустимые* (возможные) *точки входа*.

Следует отметить, что повторно входимые программные модули встречаются гораздо чаще реентерабельных (повторно прерываемых).

Наконец, существуют *информационные ресурсы*, то есть в качестве ресурсов могут выступать *данные*. Информационные ресурсы могут существовать как в виде переменных, находящихся в оперативной памяти, так и в виде файлов. Если процессы используют данные только для чтения, то такие информационные ресурсы можно разделять. Если же процессы могут изменять информационные ресурсы, то необходимо специальным образом организовывать работу с такими данными.

Лекция №7. Управление задачами. Дисциплины диспетчеризации

Цель: ознакомиться с понятиями планирования и диспетчеризации процессов и задач, с классификацией основных дисциплин диспетчеризации и возможностями их использования.

Каждый поток выполнения получает статус *задачи* и для него создается соответствующий дескриптор. При этом операционная система выполняет следующие основные функции: создание и удаление задач; планирование процессов и диспетчеризация задач; синхронизация задач и обеспечение их средствами коммуникации. Создание задачи сопряжено с формированием соответствующей информационной структуры, а ее удаление – с расформированием. Создание и удаление задач осуществляется по соответствующим запросам от пользователей или от самих задач. Задача может породить другую задачу («родители» - «потомки»). Родитель может приостановить или удалить дочернюю задачу, тогда как потомок не может управлять родителем.

Процессор является одним из самых необходимых ресурсов для выполнения вычислений. Поэтому способы распределения времени центрального процессора между выполняющимися задачами сильно влияют на скорость выполнения отдельных вычислений и на общую эффективность вычислительной системы.

Основным подходом в организации того или иного метода управления процессами, обеспечивающего эффективную загрузку ресурсов, является организация очередей процессов и ресурсов. При распределении процессорного времени между задачами также используется *механизм очередей*. Решение вопросов, связанных с тем, какой задаче следует предоставить процессорное время в данный момент, возлагается на специальный модуль ОС, называемый *диспетчером задач*. Вопросы подбора вычислительных процессов, которые можно и целесообразно решать параллельно, возлагаются на планировщик процессов. На распределение ресурсов влияют конкретные потребности тех задач, которые должны выполняться параллельно.

Например, пусть всем выполняющимся ресурсам требуется устройство с последовательным доступом. Т.к. такое устройство не может разделяться между параллельно выполняющимися процессами, то процессы будут вынуждены долго ждать своей очереди, т.е. недоступность одного ресурса может привести к тому, что длительное время не будут использоваться многие другие ресурсы. Если же взять такой набор процессов, которые не будут конкурировать между собой за неразделяемые ресурсы при своем параллельном выполнении, то процессы смогут выполниться быстрее (из-за отсутствия дополнительных ожиданий), кроме того, имеющиеся в системе ресурсы будут использоваться более эффективно. Таким образом, возникает задача подбора такого множества процессов, которые при своем выполнении будут как можно реже конфликтовать за имеющиеся в системе ресурсы – *планирование* вычислительных процессов.

В настоящее время на первый план вышли задачи *динамического* (*краткосрочного планирования*) – текущего наиболее эффективного распределения ресурсов, возникающего практически по каждому событию. Задачи динамического планирования стали называть *диспетчеризацией*. *Краткосрочный планировщик* решает, какая из задач, находящихся в очереди готовых к выполнению, должна быть передана на исполнение.

Долгосрочный планировщик решает, какой из процессов, находящихся во входной очереди, в случае освобождения ресурсов памяти должен быть переведен в очередь процессов, готовых к выполнению. Он выбирает процесс из входной очереди с целью создания неоднородной мультипрограммной смеси. В большинстве распространенных современных ОС долгосрочный планировщик отсутствует.

Основное различие между краткосрочным и долгосрочным планировщиками заключается в частоте из запуска (например, краткосрочный может запускаться каждые 30 или 100 мс, долгосрочный – один раз в несколько минут).

При рассмотрении *стратегий планирования*, как правило, идет речь о диспетчеризации. Иногда используется термин *стратегия обслуживания*. Стратегия планирования определяет, какие процессы планируются на выполнение, чтобы достичь поставленной цели. Среди стратегий выбора процесса, которому необходимо предоставить процессор, можно выделить следующие:

- по возможности заканчивать вычислений (вычислительные процессы) в том же порядке, в котором они были начаты;
- отдавать предпочтение более коротким вычислительным задачам;
- предоставлять всем пользователям (процессам пользователей) одинаковые услуги, в том числе и одинаковое время ожидания.

Дисциплины диспетчеризации (*обслуживания*) – это правила формирования очереди готовых к выполнению задач, в соответствии с которыми формируется эта очередь (список). Различают два больших класса дисциплин обслуживания: *бесприоритетные* и *приоритетные*. При

бесприоритетном обслуживании выбор задач производится в некотором заранее установленном порядке без учета их относительной важности и времени обслуживания. При реализации приоритетных дисциплин обслуживания отдельным задачам предоставляется преимущественное право попасть в состояние исполнения. В концепции приоритетов имеются следующие варианты: приоритет, присвоенный задаче, является величиной постоянной; приоритет изменяется в течение времени решения задачи (динамический приоритет).

Диспетчеризация с динамическими приоритетами требует дополнительных расходов на вычисление значений приоритетов исполняющихся задач, поэтому во многих ОС реального времени используются методы диспетчеризации на основе абсолютных приоритетов. Это позволяет сократить время реакции системы на очередное событие, однако требует детального анализа всей системы для правильного присвоения соответствующих приоритетов всем исполняющимся задачам, чтобы гарантировать обслуживание. Перечень дисциплин обслуживания и их классификация приведен на рисунке А.6. Рассмотрим основные наиболее часто используемые дисциплины диспетчеризации.

Самой простой в реализации является *дисциплина FCFS* (First Come First Served – первый пришел, первым обслужен), согласно которой задачи обслуживаются «в порядке очереди», т.е. в порядке их появления (рисунок А.7). Те задачи, которые были заблокированы в процессе работы (попали в состояние ожидания, например из-за операций ввода-вывода) после перехода в состояние готовности вновь ставятся в эту очередь готовности. При этом возможны два варианта:

1) ставить разблокированную задачу в конец очереди готовых к выполнению задач (самый простой, применяется чаще всего);

2) диспетчер помещает разблокированную задачу перед теми задачами, которые еще не выполнялись – т.е. образуется две очереди: одна из новых задач, а другая – из ранее выполнявшихся задач, попавших в состояние ожидания.

Дисциплина не требует внешнего вмешательства в ход вычислений, при ней не происходит перераспределения процессорного времени. Она относится к не вытесняющим дисциплинам. Достоинства: простота реализации и малые расходы системных ресурсов на формирование очереди задач. Недостаток: при увеличении загрузки вычислительной системы растет среднее время ожидания обслуживания, причем короткие задания вынуждены ожидать столько же, сколько и трудоемкие. Избежать этого недостатка позволяют дисциплины SJN и SRT.

Дисциплина обслуживания SJN (Shortest Job Next – следующим выполняется самое короткое задание) требует, чтобы для каждого задания была известна оценка в потребностях машинного времени. Она предполагает, что имеется только одна очередь заданий, готовых к выполнению. Задания, которые были заблокированы в процессе своего исполнения, вновь попадают

в конец очереди готовых к выполнению наравне с вновь поступающими. Это приводило к тому, что те задания, которым требовалось немного времени для завершения, вынуждены были ожидать процессор наравне с длительными работами.

Дисциплина обслуживания SRT (Shortest Remaining Time – следующим выполняется задание, которому осталось меньше всего времени выполняться на компьютере) была предложена для устранения этого недостатка.

Все три дисциплины обслуживания могут использоваться для пакетных режимов обработки, когда пользователю не нужно ждать реакции системы – он просто сдает свое задание и через несколько часов получает результаты вычислений. Для интерактивных вычислений необходимо обеспечить приемлемое время реакции системы. Если система является мультитерминальной, желательно, чтобы она обеспечивала еще и равенство в обслуживании. Для систем разделения времени основной является система обслуживания, согласно которой главным является равенство обслуживания при приемлемом времени обслуживания (этой стратегии соответствуют Unix-системы). Если же система однопользовательская с возможностью мультипрограммной обработки, то желательно, чтобы программы, с которыми непосредственно работает пользователь, имели лучшее время реакции, нежели фоновые задания. При этом желательно, чтобы некоторые приложения, выполняясь без непосредственного участия пользователя (например, получение электронной почты с использованием модема и коммутирующих линий для передачи данных), гарантированно получали необходимую им долю процессорного времени.

Для решения перечисленных проблем используется дисциплина обслуживания, называемая *карусельной* (Round Robin, RR), и приоритетные методы обслуживания. Дисциплина RR (рисунок А.8) предполагает, что каждая задача получает процессорное время *порциями* (*квантами* времени q – time slice). После окончания кванта времени q задача снимается с процессора, и он передается следующей задаче. Снятая задача ставится в конец очереди задач, готовых к выполнению. Это - одна из самых распространенных дисциплин.

Для оптимальной работы системы необходимо правильно выбрать закон, по которому кванты времени выделяются задачам. Величина кванта времени выбирается как компромисс между приемлемым временем реакции системы на запросы пользователей и накладными расходами на частую смену контекста задач. В некоторых ОС есть возможность указывать в явном виде q или диапазон возможных значений, тогда система будет стараться выбирать оптимальное значение сама. Дисциплина карусельной диспетчеризации более подходит для случая, когда все задачи имеют одинаковые права на использование ресурсов центрального процессора. Однако одни задачи всегда нужно решать в первую очередь, тогда как остальные могут подождать. Это реализуется за счет того, что одной задаче присваивается один приоритет, а другой задаче – другой. Задачи в очереди будут располагаться в соответствии

с их приоритетами. Формирует очередь диспетчер задач. Процессор в первую очередь предоставляется задаче с более высоким приоритетом, и только если ее потребности удовлетворены или она попала в состояние ожидания, диспетчер может предоставить его следующей задаче. Основная идея карусельной дисциплины обслуживания и механизм приоритетов по-разному используются в различных дисциплинах обслуживания.

Бывают ситуации, когда ОС не поддерживает в явном виде дисциплину RR. Например, в некоторых системах реального времени используется диспетчер задач, работающий по принципу абсолютных приоритетов (процессор предоставляется задаче с максимальным приоритетом, а при равенстве приоритетов он действует по принципу очередности). Т.о. причиной снятия задачи с выполнения может быть только появление задачи с более высоким приоритетом. В своей простейшей реализации дисциплина RR предполагает, что все задачи имеют одинаковый приоритет. Если же необходимо ввести механизм приоритетного обслуживания, то это делается за счет организации нескольких очередей. Процессорное время предоставляется дисциплинам, которые стоят в самой привилегированной очереди. Если она пустая, то диспетчер задач начинает посматривать остальные очереди. Именно по такому алгоритму действует диспетчер задач в ОС OS/2, Windows 9x, Windows NT/2000/XP и многих других. Отличия заключаются в количестве очередей и правилах перемещения задач из одной очереди в другую.

Известные дисциплины диспетчеризации могут применять или не применять еще одно правило, касающееся перераспределения процессорного времени между выполняющимися задачами. Диспетчеризация без перераспределения процессорного времени, т.е. *невытесняющая* (non-preemptive multitasking) или *кооперативная многозадачность* – это такой способ диспетчеризации задач, при котором активная задача выполняется до тех пор, пока она сама «по собственной инициативе» не отдаст управление диспетчеру задач для того, чтобы он выбрал из очереди другой, готовый к выполнению поток или процесс. Это – дисциплины FCFS, SJN, SRT.

Есть дисциплины, в которых процессор может быть принудительно отобран у текущей задачи. Такие дисциплины называются *вытесняющими*, поскольку одна задача вытесняется другой. Диспетчеризация с перераспределением процессорного времени между задачами, т.е. *вытесняющая многозадачность* (preemptive multitasking) – это такой способ, при котором решение о переключении процессора с выполнения одной задачи на другую принимается диспетчером задач. При вытесняющей многозадачности механизм диспетчеризации задач целиком сосредоточен в ОС, которая выполняет функции: определяет момент снятия с выполнения текущей задачи; сохраняет ее контекст в дескрипторе задачи; выбирает из очереди готовых задач следующую и запускает ее на выполнение, загружая ее контекст. Дисциплина RR и другие, созданные на ее основе, относятся к вытесняющим.

Лекция №8. Взаимодействие процессов. Синхронизация. Тупики

Цель: ознакомиться с понятием кооперации процессов, средствами обмена информацией, синхронизацией процессов и ее механизмами, а также причинами возникновения, способами обнаружения тупиков и возможностями их устранения.

Для нормального функционирования процессов операционная система старается максимально обособить их друг от друга. Каждый процесс имеет собственное адресное пространство, нарушение которого, как правило, приводит к аварийной остановке процесса. Каждому процессу по возможности предоставляются свои дополнительные ресурсы.

Тем не менее, для достижения поставленной цели различные процессы могут исполняться псевдопараллельно на одной вычислительной системе или параллельно на разных вычислительных системах, взаимодействуя между собой. Причинами совместной деятельности процессов обычно являются: необходимость ускорения решения задачи, совместное использование обновляемых данных, удобство работы или модульный принцип построения программных комплексов. Процессы не могут взаимодействовать, не обмениваясь информацией, при этом их поведение в зависимости от полученной информации, может изменяться. Если деятельность процессов остается неизменной при любой принятой ими информации, то это означает, что они на самом деле в "общении" не нуждаются.

Процессы, которые влияют на поведение друг друга путем обмена информацией, принято называть *кооперативными* или *взаимодействующими* процессами, в отличие от независимых процессов, не оказывающих друг на друга никакого воздействия. По объему передаваемой информации и степени возможного воздействия одного процесса на поведение другого все средства обмена можно разделить на три категории:

1) *сигнальные* - передается минимум информации (1 бит) и используются для извещения процесса о наступлении какого-либо события, при этом взаимовлияние процессов минимально;

2) *канальные* – обмен информацией происходит через линии связи, предоставленные операционной системой, а объем передаваемой информации в единицу времени ограничивается пропускной способностью линий связи;

3) *разделяемая память* - несколько процессов могут совместно использовать некоторую область адресного пространства, при этом возможность обмена информацией максимальна, как и влияние на поведение другого процесса, но требует повышенной осторожности. Созданием разделяемой памяти занимается операционная система.

Допустим, что в вычислительной системе находятся два взаимодействующих процесса: один из них – H – с высоким приоритетом, другой – L – с низким приоритетом. Пусть планировщик устроен так, что процесс с высоким приоритетом вытесняет низкоприоритетный процесс

всякий раз, когда он готов к исполнению, и занимает процессор, до того момента, пока не появится процесс с еще большим приоритетом. Тогда в случае, если процесс L находится в своей критической секции, а процесс H, получив процессор, подошел ко входу в критическую область, мы получаем тупиковую ситуацию. Процесс H не может войти в критическую область, находясь в цикле, а процесс L не получает управления, чтобы покинуть критический участок. Чтобы не допустить возникновения подобных проблем и повысить производительность вычислительных систем, были разработаны различные *механизмы синхронизации*.

Для организации синхронизации процессов могут применяться специальные механизмы высокого уровня, блокирующие процесс, ожидающий входа в критическую секцию или наступления своей очереди для использования совместного ресурса. К таким механизмам относятся *семафоры*, *мониторы* и *сообщения*.

Способы синхронизации процессов позволяют процессам успешно кооперироваться, однако в некоторых случаях могут возникнуть непредвиденные затруднения. Предположим, что несколько процессов конкурируют за обладание конечным числом ресурсов. Если запрашиваемый процессом ресурс недоступен, ОС переводит данный процесс в состояние ожидания. В случае, когда требуемый ресурс удерживается другим ожидающим процессом, первый процесс не сможет сменить свое состояние. Такая ситуация называется *тупиком* (*deadlock*). Говорят, что в мультипрограммной системе процесс находится в *состоянии тупика*, если он ожидает события, которое никогда не произойдет. *Системная тупиковая ситуация* («з*ависание системы*»), является следствием того, что один или более процессов находятся в состоянии тупика. Иногда подобные ситуации называют *взаимоблокировками*. В общем случае проблема тупиков эффективного решения не имеет.

Одной из причин бесконечного ожидания может быть дискриминационная политика по отношению к некоторым процессам. Однако чаще всего событие, которого ждет процесс в тупиковой ситуации, – освобождение ресурса. К взаимоблокировке может привести использование как неделимых, так и разделяемых ресурсов. Например, чтение с разделяемого диска может одновременно осуществляться несколькими процессами, тогда как запись предполагает исключительный доступ к данным на диске. Можно считать, что часть диска, куда происходит запись, выделена конкретному процессу. Следовательно, тупики связаны с выделенными ресурсами, то есть возникают, когда процессу предоставляется эксклюзивный доступ к устройствам, файлам и другим ресурсам. Как правило, борьба с тупиками – очень дорогостоящее мероприятие. Тем не менее, для ряда систем, например, для систем реального времени, иного выхода нет.

Условия возникновения тупиков были сформулированы Коффманом, Элфиком и Шошани в 1970 г. Для образования тупика необходимым и достаточным является выполнение всех четырех условий:

1) *условие взаимоисключения*: одновременно использовать ресурс может только один процесс;

2) *условие ожидания ресурсов*: процессы удерживают ресурсы, уже выделенные им, и могут запрашивать другие ресурсы;

3) *условие неперераспределяемости*: ресурс, выделенный ранее, не может быть принудительно отобран у процесса;

4) *условие кругового ожидания*: существует кольцевая цепь процессов, в которой каждый процесс ждет доступа к ресурсу, удерживаемому другим процессом цепи.

Первые три условия формируют правила, существующие в системе, тогда как четвертое условие описывает ситуацию, которая может сложиться при определенной неблагоприятной последовательности событий. Поэтому методы предотвращения взаимоблокировок ориентированы, главным образом, на нарушение первых трех условий путем введения ряда ограничений на поведение процессов и способы распределения ресурсов. Методы обнаружения и устранения менее консервативны и сводятся к поиску и разрыву цикла ожидания ресурсов:

1) *игнорирование проблемы в целом*;
2) *предотвращение тупиков*;
3) *обнаружение тупиков*;
4) *восстановление после тупиков*.

Самый простой и наиболее распространенный способ устранить тупик – завершить выполнение одного или более процессов, чтобы впоследствии использовать его ресурсы. Тогда в случае удачи остальные процессы смогут выполняться. Если это не помогает, можно ликвидировать еще несколько процессов.

После каждой ликвидации должен запускаться алгоритм обнаружения тупика. По возможности лучше ликвидировать тот процесс, который может быть без ущерба возвращен к началу (такие процессы называются *идемпотентными*). Примером такого процесса может служить компиляция. С другой стороны, процесс, который изменяет содержимое базы данных, не всегда может быть корректно запущен повторно.

В некоторых случаях можно временно отобрать ресурс у текущего владельца и передать его другому процессу. Возможность отнять ресурс у процесса, дать его другому процессу и затем без ущерба вернуть назад зависит от природы ресурса. Подобное восстановление часто затруднительно.

В ряде систем реализованы средства отката и перезапуска или рестарта с *контрольной точки* (сохранение состояния системы в какой-то момент времени). Если проектировщики системы знают, что тупик вероятен, они могут периодически организовывать для процессов контрольные точки.

Когда тупик обнаружен, видно, какие ресурсы вовлечены в цикл кругового ожидания. Чтобы осуществить восстановление, процесс, который владеет таким ресурсом, должен быть отброшен к моменту времени, предшествующему его запросу на этот ресурс.

Таким образом, возникновение тупиков является потенциальной проблемой любой операционной системы. Они возникают, когда имеется группа процессов, каждый из которых пытается получить исключительный доступ к некоторым ресурсам и претендует на ресурсы, принадлежащие другому процессу. В итоге все они оказываются в состоянии бесконечного ожидания. С тупиками можно бороться, можно их обнаруживать, избегать и восстанавливать систему после тупиков. Однако цена подобных действий высока и соответствующие усилия должны предприниматься только в системах, где игнорирование тупиковых ситуаций приводит к катастрофическим последствиям.

Лекция №9. Управление памятью в операционных системах. Память и отображение. Неразрывные методы распределения памяти

Цель лекции: ознакомиться с механизмами отображения пространства имен на физическую память компьютера, а также с неразрывными методами распределения памяти.

Оперативная память – это важнейший ресурс любой вычислительной системы, поскольку без нее невозможно выполнение ни одной программы. Поскольку память является распределяемым ресурсом, от выбранных механизмов распределения памяти между выполняющимися процессами в значительной степени зависит эффективной использования ресурсов системы, ее производительность и возможности, которыми могут пользоваться программисты при создании свои программ.

В общем случае множество переменных в программе не упорядочено, т.к. обращение к памяти осуществляется посредством некоторого набора *логических имен* (чаще всего символьных, а не числовых), хотя отдельные переменные могут иметь частичную упорядоченность (массивы). Имена переменных и входных точек программных модулей составляют пространство символьных имен – *логическое пространство*. С другой стороны, при выполнении программы процессор работает с *физической памятью*, извлекая из нее команды и данные и помещая в нее результаты вычислений. *Физическая память* представляет собой упорядоченное множество ячеек реально существующей оперативной памяти, все они пронумерованы, т.е. к каждой из них можно обратиться, указав ее порядковый номер (адрес). Количество ячеек физической памяти ограничено и фиксировано.

Системное программное обеспечение должно связать каждое символьное имя, указанное пользователем, с физической ячейкой памяти, т.е. осуществить *отображение пространства имен на физическую память компьютера*. В общем случае это отображение осуществляется в два этапа (рисунок А.9): сначала системой программирования, а затем операционной системой (с помощью подсистемы управления памятью). Между этапами обращения к памяти имеют форму *виртуального адреса*. При этом множество

всех допустимых значений виртуального адреса для некоторой программы определяет ее *виртуальное адресное пространство* или *виртуальную память*. Виртуальное адресное пространство зависит от архитектуры процессора и от системы программирования и практически не зависит от объема реальной физической памяти компьютера.

Система программирования осуществляет трансляцию и компоновку программы, используя библиотечные модули. В результате работы системы программирования полученные виртуальные адреса могут иметь как *двоичную*, так и *символьно-двоичную* формы. В случае, когда имеется полная тождественность виртуального адресного пространства физической памяти, речь идет о генерации *абсолютной двоичной программы* (т.е. программы будет исполняться только тогда, когда виртуальные адреса будут соответствовать физическим). Часть программных модулей любой ОС обязательно должна быть абсолютными двоичными программами, которые размещаются по фиксированным адресам физической памяти (например, программы загрузки операционной системы).

Термин *виртуальная память* фактически относится к системам, которые сохраняют виртуальные адреса во время исполнения. Так как второе отображение осуществляется во время исполнения задачи, то адреса физических ячеек могут изменяться. В зависимости от соотношения объемов виртуального адресного пространства *Vv* и физической памяти *Vp* используются разные эффективных методы распределения памяти.

Память задаче может выделяться одним сплошным участком (*методы неразрывного распределения памяти*) или несколькими порциями, которые могут быть размещены в разных областях памяти (*методы разрывного распределения*).

Простое непрерывное распределение – это самая простая схема, согласно которой память может быть условно разделена на три области: область, занимаемая ОС; область, в которой размещается исполняемая задача; область памяти, незанятая ничем (свободная). Эта схема является самой первой схемой и продолжает оставаться достаточно распространенной. Схема предполагает, что ОС не поддерживает мультипрограммирование, поэтому не возникает проблемы распределения памяти между несколькими задачами. Программные модули, необходимые для всех программ, располагаются в области самой ОС, а вся оставшаяся память может быть предоставлена задаче. Эта область памяти получается непрерывной, что облегчает систему программирования. Привязка виртуальных адресов программы к физическому адресному пространству осуществляется на этапе загрузки задачи в память. Чтобы отвести задачам как можно больший объем памяти, ОС строится таким образом, чтобы постоянно в оперативной памяти располагалась только самая нужная ее часть – ядро.

Если есть необходимость создать программу, логическое адресное пространство которой должно быть больше, чем свободная область памяти (или чем весь возможный объем ОП), то используется *распределение с*

перекрытием – так называемые *оверлейные структуры*. Этот метод распределения предполагает, что вся программа может быть разбита на части – *сегменты*. Каждая оверлейная программа имеет одну главную часть (main) и несколько сегментов (segments), в памяти одновременно могут находиться только ее главная часть или несколько не перекрывающихся сегментов. Пока в ОП располагаются выполняющиеся сегменты, остальные находятся во внешней памяти. После того, как текущий сегмент завершит свое выполнения, возможны варианты: либо он сам обращается к ОС с указанием, какой сегмент должен быть загружен следующим; либо он возвращает управление главному сегменту задачи, который обращается с указанием к ОС. Простейшие схемы сегментирования предполагают, что в каждый момент времени в памяти может располагаться только один сегмент вместе с главным модулем. Более сложные схемы позволяют располагать в памяти несколько сегментов. Например, в однопрограммной ОС MS DOS распределение памяти построено по схеме простого непрерывного распределения с перекрытием (оверлейные структуры).

Для организации мультипрограммного и/или мультизадачного режима необходимо обеспечить одновременное расположение в оперативной памяти нескольких задач (целиком или частями). Самая простая схема распределения памяти между несколькими задачами предполагает, что память, не занятая ядром ОС, может быть разбита на несколько непрерывных частей – *разделов*. Разделы характеризуются именем, типом, границами (указываются начало раздела и его длина). Разбиение памяти на несколько непрерывных (неразрывных) разделов может быть *фиксированным* (*статическим*) либо *динамическим* (т.е. процесс выделения нового раздела происходит непосредственно при появлении новой задачи). Разбиение всего объема ОП на несколько разделов может осуществляться единовременно (т.е. в процессе генерации нового варианта ОС, который потом эксплуатируется) или по мере необходимости оператором системы. Пример разбиения памяти на несколько разделов приведен на рисунке А.10. В каждом разделе в каждый момент времени может располагаться по одной программе (задаче). В этом случае по отношению к каждому разделу можно применить все те методы создания программ, которые применяются для однопрограммных систем. Возможно использование оверлейных структур, что позволяет создавать большие сложные программы и в то же время поддерживать коэффициент мультипрограммирования на должном уровне.

При небольшом объеме памяти и, следовательно, небольшом количестве разделов увеличить число параллельно выполняемых приложений можно за счет замены их в памяти (*свопинга – swapping*). При свопинге задача может быть целиком выгружена на магнитный диск (перемещена во внешнюю память), а на ее место загружается либо более привилегированная, либо просто готовая к выполнению другая задача, находившаяся на диске в приостановленном состоянии. При свопинге из основной памяти во внешнюю (и обратно) перемещается вся программа, а не ее отдельная часть. Основным

недостатком рассматриваемого способа является наличие достаточно большого объема неиспользуемой памяти, которая может быть в каждом из разделов. Такие потери памяти называют *фрагментацией памяти*.

В отдельных разделах потери памяти могут быть значительными, однако использовать фрагменты свободной памяти при таком способе распределения невозможно.

Чтобы избавиться от фрагментации, можно размещать в ОП задачи плотно, одну за другой, выделяя ровно столько памяти, сколько задача требует. Для этого система использует специальный планировщик, который ведет список адресов свободной ОП. При появлении новой задачи диспетчер памяти просматривает этот список и выделяет для задачи раздел, объем которой либо равен необходимому, либо чуть больше, если память выделяется не ячейками, а некими дискретными единицами. Список при этом изменяется. При освобождении раздела диспетчер памяти пытается объединить освобождающийся раздел с одним из свободных участков, если таковой является смежным. Список свободных участков может быть упорядочен либо по адресам, либо по объему. Память под новый раздел выделяется одним из способов:

1) *первый подходящий участок* – список свободный областей упорядочивается по адресам. Диспетчер просматривает список и выделяет задаче раздел в той области, которая первой подойдет по объему;

2) *самый подходящий участок* – список свободных областей упорядочен по возрастанию объема фрагментов. В этом случае при просмотре списка для нового раздела будет использован фрагмент свободной памяти, объем которой наиболее точно соответствует требуемому;

3) *самый неподходящий участок* – как ни странно является наиболее эффективным способом. Список свободных адресов упорядочивается по убыванию объема свободного фрагмента. Следовательно, подходящий объем будет найден сразу, а оставшийся неиспользованный фрагмент может оказаться еще пригодным для дальнейшего использования.

При любой дисциплине обслуживания в памяти всегда будет наблюдаться сильная фрагментация. При этом возможны ситуации, когда из-за сильной фрагментации памяти диспетчер задач не сможет образовать новый раздел, хотя суммарный объем свободных областей будет больше, чем это необходимо для задачи. В этой ситуации организуется *уплотнение памяти*, для чего все вычисления приостанавливаются, и диспетчер памяти корректирует свои списки, перемещая разделы в начало памяти. Недостаток – потеря времени на уплотнение и невозможность при этом выполнять вычисления.

Лекция №10. Разрывные методы распределения памяти

Цель лекции: ознакомиться с разрывными методами распределения памяти.

Методы распределения памяти, при которых задаче уже не может предоставляться сплошная (непрерывная) область памяти, называются *разрывными*. Подход требует для своей реализации больше ресурсов и намного сложнее, однако, выделение задаче памяти фрагментами, а не сплошной областью, позволяет уменьшить фрагментацию памяти.

Первым среди разрывных методов распределения памяти был *сегментный*. Для этого метода программу необходимо разбивать на части и уже каждой такой части выделять физическую память. Естественным способом разбиения программы на части является разбиение ее на логические элементы – так называемые *сегменты*. В принципе, каждый программный модуль может быть принят как отдельный сегмент, а вся программа будет представлять собой множество сегментов. Каждый сегмент размещается в памяти как до определенной степени самостоятельная единица. Логически обращение к элементам программы в этом случае будет состоять из имени сегмента и смещения относительно начала этого сегмента. Физически имя сегмента (или порядковый номер) будет соответствовать некоторому адресу, с которого этот сегмент начинается при его размещении в памяти, и смещение должно прибавляться к этому базовому адресу. Преобразование имени сегмента в его порядковый номер осуществит система программирования. Для каждого сегмента система указывает его объем. ОС будет размещать сегменты в памяти и для каждого сегмента она должна вести учет о местонахождении этого сегмента (рисунок А.11). Информация о текущем размещении сегментов задачи в памяти сводится в таблицу сегментов (таблицу дескрипторов сегментов задачи). Каждая задача имеет свою таблицу сегментов.

Увеличение коэффициента мультипрограммирования позволяет увеличить загрузку системы и более эффективно использовать ресурсы вычислительной системы. Однако увеличивать количество задач можно только до определенного предела, ибо если в памяти не будет хватать места для часто используемых сегментов, то производительность системы резко упадет, т.к. сегмент, находящийся вне ОП, для участия в вычислениях должен быть перемещен в ОП. При этом, если в памяти есть свободное пространство, то необходимо найти нужный сегмент во внешней памяти и загрузить его в оперативную. А если свободного места нет, принять решение – на место какого из сегментов будет загружаться требуемый. Перемещение сегментов из ОП на жесткий диск и обратно называют *свопингом сегментов*. Для решения проблемы используются следующие дисциплины:

1) *правило FIFO* (*First In First Out* – первый пришедший первым и выбывает) - для замещения выбирается сегмент, первым попавший в память;

2) *правило LRU* (*Least Recently Used* – дольше других неиспользуемый);

3) *правило LFU* (*Least Frequently Used* – реже других используемый);

4) *случайный* (*random*) выбор сегмента.

Для реализации дисциплин 2 и 3 необходимо, чтобы процессор имел дополнительные аппаратные средства. Диспетчер памяти время от времени

просматривает таблицы дескрипторов исполняющихся задач и собирает для соответствующей обработки статистическую информацию об обращениях к сегментам. В результате составляется список, упорядоченный либо по длительности простоя (2), либо по частоте использования (3). Дисциплины 1 и 4 являются самыми простыми в реализации, но не учитывают, насколько часто используется тот или иной сегмент, следовательно, диспетчер памяти может выгрузить или расформировать тот сегмент, к которому в ближайшем будущем будет обращение. При этом вероятность ошибки в первом и четвертом случаях больше, чем во втором и третьем.

Достоинства, появляющиеся при сегментном способе организации виртуальной памяти: при загрузке программы на исполнение можно размещать ее в памяти не целиком, а «по мере необходимости»; некоторые модули могут быть разделяемыми (т.к. они являются сегментами, легко организовать доступ к этим общим сегментам с помощью указателей). Недостатки метода: для доступа к искомой ячейке тратится много времени, для устранения используется кэширование – размещение в сверхоперативной памяти; несмотря на то, что наблюдается существенно меньшая фрагментация, чем при неразрывном распределении памяти, она все же есть; на размещение и обработку дескрипторных таблиц тратится много процессорного времени и памяти.

Для устранения недостатков сегментного способа организации виртуальной памяти был разработан следующий метод разрывного распределения памяти – *страничный*. *Страничный способ организации виртуальной памяти* – это способ, при котором все фрагменты задачи считаются равными (одинаковыми по размеру), причем длина фрагмента в идеале должна быть кратна степени двойки, чтобы операции сложения можно было заменить операциями конкатенации (слияния). Одинаковыми полагаются и единицы памяти, которые предоставляются для размещения фрагментов программы. Эти одинаковые части называют *страницами*, ОП разбивается на физические страницы, а программа – на виртуальные единицы. Часть виртуальных страниц размещается в ОП, а часть – во внешней памяти. Место во внешней памяти (чаще всего НМД – быстродействующие устройства с прямым доступом) называют *файлом подкачки* или *страничным файлом* (paging file) или *swap-файлом*, тем самым подчеркивая, что записи этого файла – страницы – замещают друг друга в ОП. В Unix-системах для этого выделяется специальный раздел.

Разбиение памяти на страницы одинаковой величины приводит к тому, что вместо одномерного адресного пространства появляется двухмерное. Первая координата – номер страницы, вторая – номер ячейки внутри выбранной страницы (индекс). Т.о. физический адрес определяется парой (Pp,i), а виртуальный - парой (Pv,i), где Pp – номер физической страницы; Pv – номер виртуальной страницы; i – индекс ячейки внутри страницы. Количество битов, отводимое под индекс, определяет размер страницы, а отводимое под

номер виртуальной страницы – объем потенциально доступной для программы виртуальной памяти.

Для отображения виртуального адресного пространства задачи на физическую память необходимо иметь *таблицу страниц*. Для описания каждой страницы заводится дескриптор, который в отличие от дескриптора сегмента не имеет поля длины, т.к. размер страниц одинаковый. Защита страничной памяти основана на контроле уровня доступа к каждой странице. При обращении к виртуальной странице, не оказавшейся в данный момент в ОП, возникает прерывание, и управление передается диспетчеру памяти, который должен найти свободное место. Обычно предоставляется первая же свободная страница. Если свободной страницы нет, то используется одна из дисциплин замещения.

Если объем физической памяти небольшой и даже часто требуемые страницы не удается разместить в ОП, возникает так называемая «*пробуксовка*». *Пробуксовка* – это ситуация, при которой загрузка нужной страницы вызывает перемещение во внешнюю память той страницы, с которой идет активная работа. Чтобы не допустить этого, желательно увеличить объем ОП, уменьшить количество параллельно выполняемых задач или использовать более эффективные дисциплины замещения.

Для большинства современных ОС характерна LRU как наиболее эффективная (OS/2, Linux). Однако в ОС Windows NT/2000/XP разработчики, желая их сделать максимально независимыми от аппаратных возможностей процессора, отказались от LRU и применили FIFO. А чтобы скомпенсировать неэффективность FIFO, была введена *буферизация* тех страниц, которые должны быть записаны в файл подкачки на диск или расформированы. *Принцип буферизации* заключается в следующем: прежде чем замещаемая страница действительно окажется во внешней памяти или просто будет расформирована, она помечается как кандидат на выгрузку. Если в следующий раз произойдет обращение к странице, находящейся в «буфере», то страница не выгружается и уходит в конец списка FIFO. В противном случае страница действительно выгружается, а на ее место в буфер попадает следующий кандидат.

В ряде ОС с пакетным режимом работы для борьбы с пробуксовкой используется *метод «рабочего множества»*. *Рабочее множество* – это множество (среднее количество) активных страниц задачи за некоторый интервал T, то есть тех страниц, к которым было обращение за этот интервал времени.

Достоинство метода – минимальная фрагментация. Недостатки: существенные накладные расходы (требуется разместить в памяти и обработать все страницы); программы разбиваются на страницы случайно, без учета логических взаимосвязей, имеющихся в коде (межстраничные переходы осуществляются чаще, чем межсегментные).

Чтобы избежать второго недостатка, сохранив достоинства страничного метода, был разработан способ – *сегментно-страничный* (однако,

увеличились затраты на реализацию). Программа разбивается на сегменты; виртуальный адрес содержит указание на номер соответствующего сегмента. Другая составляющая – смещение относительно начала сегмента состоит из двух полей: виртуальной страницы и индекса.

Недостаток – задержка доступа к памяти увеличивается в 3 раза. Чтобы этого избежать вводится *кэширование по ассоциативному принципу*.

Достоинства: сегменты размещаются в памяти целиком; сегменты разбиты на страницы, все страницы сегмента загружаются в память, что позволяет сократить число обращений к отсутствующим страницам, т.к. вероятность выхода за пределы сегмента меньше вероятности выхода за пределы страницы; страницы исполняемого сегмента находятся в памяти «россыпью»; наличие сегментов облегчает разделение программных модулей между параллельными процессами; минимальная фрагментация.

Способ требует значительных затрат вычислительных ресурсов и сложный в реализации, поэтому используется редко, в дорогих мощных вычислительных системах.

Лекция №11. Управление вводом-выводом в операционных системах

Цель: ознакомиться с режимами управления вводом-выводом, устройствами ввода-вывода и их закреплением, а также синхронным и асинхронным способами ввода-вывода.

Необходимость предоставлять программам средства обмена данными с внешними устройствами, которые бы не требовали непосредственного включения в каждую программу двоичного кода, управляющего вводом-выводом, привела к созданию системного программного обеспечения, в том числе операционных систем. Программирование ввода-вывода является наиболее сложным и трудоемким, поэтому код, реализующий операции ввода-вывода, начали оформлять в виде библиотечных процедур, в затем вообще вывели из систем программирования, включив в ОС. Это позволило не писать код в каждой программе, а лишь обращаться к нему. Таким образом, управление вводом-выводом – одна из основных функций ОС.

При организации ввода-вывода требуется не только обеспечить эффективное управление разнообразными устройствами ввода-вывода (УВВ), но и создать удобный и эффективный виртуальный интерфейс УВВ, позволяющий прикладным программистам считывать или сохранять данные, не обращая внимания на специфику устройств и проблемы их распределения между одновременно выполняющимися программами. Поэтому основным является следующий принцип: *любые операции по управлению вводом-выводом объявляются привилегированными и могут выполняться только кодом самой системы.* Для обеспечения этого принципа в большинстве процессоров вводятся *режимы пользователя* и *супервизора* (режим ядра,

привилегированный режим). В режиме супервизора выполнение команд ввода-вывода разрешено, а в режиме пользователя – запрещено. Обращение к командам ввода-вывода в пользовательском режиме вызывает *исключение* – определенный вид внутреннего прерывания.

Основные причины запрета отдельной пользовательской программе обращаться к внешним устройствам непосредственно: необходимость разрешать возможные конфликты в доступе к устройствам ввода-вывода; желание увеличить эффективность использования ресурсов ввода-вывода; необходимость избавить программы ввода-вывода от ошибок. Таким образом, управление вводом-выводом осуществляется компонентом ОС – *супервизором ввода-вывода*, на который возлагаются задачи получения и проверки на корректность запросов на выполнение операций ввода-вывода от пользовательских задач, супервизора задач или программных модулей самой ОС; вызова соответствующих распределителей каналов и контроллеров, планирования ввода-вывода (постановка запросов в очередь на выполнение); инициирования операций ввода-вывода (передача управления соответствующим драйверам) и, в случае управления вводом-выводом с использованием прерываний, предоставления процессора диспетчеру задач для передачи его очередной задаче; идентификации сигналов при получении сигналов прерывания от УВВ и передача управления программам обработки прерываний; передачи сообщений об ошибках, возникающих в процессе управления операциями ввода-вывода и о завершении операции ввода-вывода задаче, запросившей эту операцию.

Различают *разделяемые* УВВ и *неразделяемые*. В случае если УВВ является *инициативным* (например, набор датчиков), управление со стороны супервизора заключается в активизации соответствующего процесса.

Существует два основных режима ввода-вывода: *режим обмена с опросом готовности УВВ* и *режим обмена с прерываниями*.

Пусть центральный процессор посылает команду устройству управления (рисунок А.12), требующую, чтобы УВВ выполнило некоторое действие (например, команда позиционирования магнитных головок). При этом работает программный канал обмена данными между внешним устройством и ОП. Устройство управления исполняет команду, транслируя сигналы, понятные ему и центральному устройству, в сигналы, понятные УВВ. После выполнения команды, УВВ (или его устройство управления) выдает сигнал готовности, который сообщает процессору о том, что можно выдать новую команду для продолжения обмена данными. Поскольку быстродействие УВВ намного меньше быстродействия центрального процессора, сигнал готовности приходится ожидать очень долго, постоянно опрашивая соответствующую линию интерфейса на наличие или отсутствие внешнего сигнала. Посылать новую команду, не дождавшись сигнала готовности, сообщающего об исполнении предыдущей команды, бессмысленно. В режиме опроса готовности драйвер, управляющий процессом обмена данными с внешним устройством, в цикле выполняет

команду «*проверить наличие сигнала готовности*» и, пока сигнал не появится, драйвер ничего другого не делает. При этом нерационально используется время центрального процессора. Поэтому выгоднее, выдав команду ввода-вывода, забыть об УВВ и перейти на выполнение другой программы, а появление сигнала готовности трактовать как сигнал запроса на прерывание от устройства ввода-вывода.

Режим обмена с прерываниями является режимом *асинхронного* управления. Чтобы не потерять связь с устройством, может быть запущен отсчет времени, в течение которого должна быть выполнена команда и получен сигнал запроса на прерывание. Максимальный интервал времени, в течение которого УВВ или его контроллер должны выдать сигнал запроса на прерывание, называют *установкой тайм-аута*. Если время истекло, а сигнал не получен, считается, что связь с УВВ потеряна, управлять им больше нет возможности, выдается диагностическое сообщение пользователю.

Драйверы, работающие в режиме прерываний, представляют собой сложный комплекс программных модулей и могут иметь несколько *секций*: *секцию запуска*, одну или несколько *секций продолжения* и *секцию завершения*. Секция запуска инициирует операцию ввода-вывода. Эта секция запускается для включения УВВ или просто для инициализации очередной операции ввода-вывода. Секция продолжения осуществляет основную работу по передаче данных и является основным обработчиком прерывания. Секция завершения выключает УВВ или просто завершает операцию.

Многие устройства, прежде всего, *устройства с последовательным доступом*, не допускают совместного использования. Такие устройства могут стать *закрепленными* за процессом на все время его жизни. Это приводит к тому, что вычислительные процессы не могут выполняться параллельно, так как ожидают освобождения УВВ. Для организации совместного использования многими параллельными задачами неразделяемых УВВ, вводится понятие *виртуальных устройств*. Имитация работы с устройством в режиме непосредственного подключения к нему называется *спулингом* (*spooling*). Основное назначение спулинга – создать видимость разделения УВВ, которое фактически является устройством с последовательным доступом и должно использоваться только монопольно и быть закрепленным за процессом. Каждому вычислительному процессу предоставляется не реальный, а виртуальный принтер, и поток выводимых символов сначала направляется в специальный файл на диске (*спул-файл – spool-file*) и только потом, по окончании виртуальной печати, содержимое спул-файла выводится на принтер. Системные процессы, управляющие спул-файлами, называют *спулером чтения* (*spool-reader*) или *спулером записи* (*spool-writer*).

Таблицы ввода-вывода - это информационные структуры, которые используются ОС для управления операциями ввода-вывода и отслеживания состояния ресурсов, занятых в обмене данными. Каждая ОС имеет свои таблицы ввода-вывода, состав которых может отличаться, хотя в некоторых системах вместо таблиц создаются списки. Исходя из принципа управления

вводом-выводом исключительно через супервизор ОС и учитывая, что драйверы УВВ используют механизм прерываний для установления обратной связи центральной части с внешними устройствами, можно сделать вывод о необходимости создания следующих системных таблиц (рисунок А.13):

а) *таблица оборудования* (*UCB – Unit Control Block*) – содержит информацию обо всех УВВ, подключенных к вычислительной системе;

б) *таблица виртуальных логических устройств* (*DRT*) – предназначена для установления связи между виртуальными и реальными устройствами;

в) *таблица прерываний* – необходима для организации обратной связи между центральной частью и УВВ.

Запрос на операцию ввода-вывода от выполняющейся программы поступает на супервизор задач (рисунок А.14, шаг 1). Этот запрос представляет собой обращение к ОС и указывает на конкретную функцию API. Супервизор задач проверяет системный вызов на соответствие принятым спецификациям и в случае ошибки возвращает задаче соответствующее сообщение (шаг 1-1). Если запрос корректен, то он перенаправляется в супервизор ввода-вывода (шаг 2), который по логическому имени с помощью таблицы DRT находит соответствующий элемент UCB в таблице оборудования. Если устройство уже занято, то описатель задачи, запрос которой обрабатывается, помещается в список задач, ожидающих это устройство. Если УВВ свободно, то супервизор ввода-вывода определяет из UCB тип устройства и при необходимости запускает препроцессор, позволяющий получить управляющие коды и данные, которые сможет правильно понять и отработать устройство (шаг 3). Когда «программа» управления операцией ввода-вывода будет готова, супервизор передает управление соответствующему драйверу на секцию запуска (шаг 4). Драйвер инициализирует операцию управления, обнуляет счетчик тайм-аута и возвращает управление диспетчеру задач с тем, чтобы он поставил на процессор готовую к исполнению задачу (шаг 5). Как только УВВ отработает посланную ему команду, оно выставляет сигнал запроса на прерывание, по которому через таблицу прерываний управление передается на секцию продолжения (шаг 6). Получив новую команду, устройство вновь начинает ее обрабатывать, а управление процессором опять передается диспетчеру задач, и процессор продолжает выполнять полезную работу. Результатом является параллельная обработка задач, на фоне которой процессор осуществляет управление операциями ввода-вывода.

Различают *синхронный* и *асинхронный* ввод-вывод данных. Задача, выдавшая запрос на операцию ввода-вывода, переводится супервизором в состояние ожидания завершения заказанной операции. Когда супервизор получает от секции завершения сообщение о том, что операция завершилась, он переводит задачу в состояние готовности к выполнению, и она продолжает выполняться. Такая ситуация соответствует *синхронному вводу-выводу*, который является стандартным для большинства ОС. Для увеличения

скорости выполнения приложений при необходимости может использоваться *асинхронный ввод-вывод*.

Простейшим вариантом является *буферизованный* вывод данных на внешнее устройство, при котором данные из приложения передаются в специальный *системный буфер* – область памяти, отведенную для временного размещения передаваемых данных. В этом случае логически операция вывода для приложения считается выполненной сразу же, и задача может не ожидать окончания действительного процесса передачи данных на устройство. Для организации асинхронного ввода следует не только выделить область памяти для временного хранения считываемых данных и связать выделенный буфер с задачей, заказавшей операцию, но и сам запрос на операцию ввода-вывода разбить на две части (первая - на считывание данных, вторая – на завершение считывания). Асинхронный ввод-вывод характерен для мультипрограммных ОС, использующих механизм потоков выполнения.

Лекция №12. Файлы и организация работы с ними

Цель: рассмотреть вопросы структуры, именования, защиты файлов; выявить операции, которые разрешается производить над файлами; получить представление об организации файлового архива (полного дерева справочников).

Файлы представляют собой абстрактные объекты, задача которых хранить информацию, скрывая от пользователя детали работы с устройствами. Когда процесс создает файл, он дает ему имя. После завершения процесса файл продолжает существовать и через свое имя может быть доступен другим процессам. Правила именования файлов зависят от ОС. Многие ОС поддерживают имена из *двух частей* (имя + расширение). В соответствии со стандартом POSIX, популярные ОС оперируют удобными для пользователя длинными именами (до 255 символов). *Основные типы файлов*: *регулярные* (обычные) *файлы* и *директории* (справочники, каталоги).

Обычные файлы содержат пользовательскую информацию. *Директории* - системные файлы, поддерживающие структуру *файловой системы*. В каталоге содержится перечень входящих в него файлов и устанавливается соответствие между файлами и их характеристиками (атрибутами). Несмотря на то, что внутри подсистемы управления файлами обычный файл представляется в виде набора блоков внешней памяти, для пользователей обеспечивается представление файла в виде линейной последовательности байтов. Иногда к файлам приписывают другие объекты ОС, имеющие файловый интерфейс, названные *каналами* и *сокетами*.

Обычные (*регулярные*) файлы реально представляют собой набор блоков на устройстве внешней памяти, на котором поддерживается файловая система. Такие файлы могут содержать как текстовую информацию (в формате ASCII), так и произвольную двоичную (бинарную) информацию.

Текстовые файлы содержат символьные строки, которые можно распечатать, увидеть на экране или редактировать обычным текстовым редактором. *Нетекстовые* (*бинарные*) файлы имеют некоторую внутреннюю структуру. ОС выполняет файл, только если он имеет нужный формат. Пример бинарного файла - архивный файл. Типизация файлов не слишком строгая.

Кроме имени ОС часто связывают с каждым файлом и другую информацию, например дату модификации, размер и т. д. Эти характеристики файлов называются *атрибутами*, список которых может варьироваться. Типовые элементы: *основную информацию* (имя, тип файла), *адресную информацию* (устройство, начальный адрес, размер), *информацию об управлении доступом* (владелец, допустимые операции) и *информацию об использовании* (даты создания, последнего чтения, модификации). Список атрибутов обычно хранится в структуре директорий или других структурах, обеспечивающих доступ к данным файла.

Запись - это наименьший элемент данных, который может быть обработан как единое целое прикладной программой при обмене с внешним устройством. Причем в большинстве ОС размер записи равен одному байту. В то время как приложения оперируют записями, физический обмен с устройством осуществляется большими *единицами* (*блоками*). Поэтому записи объединяются в блоки для вывода и разблокируются - для ввода.

ОС поддерживают несколько вариантов структуризации файлов:

1) *последовательный файл* является последовательностью записей, а поскольку записи, как правило, однобайтовые, файл представляет собой неструктурированную последовательность байтов;

2) *файл прямого доступа* – это файл, байты которого могут быть считаны в произвольном порядке, то есть содержимое файла может быть разбросано по разным блокам диска, причем номер блока однозначно определяется позицией внутри файла.

В некоторых системах ускорение доступа к файлу обеспечивается конструированием *индекса* файла. Индекс обычно хранится на том же устройстве, что и сам файл, и состоит из списка элементов, каждый из которых содержит идентификатор записи, за которым следует указание о местоположении данной записи. Для поиска записи вначале происходит обращение к индексу, где находится указатель на нужную запись. Такие файлы называются *индексированными*, а метод доступа к ним - *доступ с использованием индекса*. Почти всегда главным фактором увеличения скорости доступа является *избыточность* данных.

Набор основных операций для работы с файлами, реализованных через системные вызовы, включает:

1) *создание файла, не содержащего данных*. Смысл данного вызова - объявить, что файл существует, и присвоить ему ряд атрибутов. При этом выделяется место для файла на диске и вносится запись в каталог;

2) *удаление файла* и освобождение занимаемого им дискового пространства;

3) *открытие файла*. Перед использованием файла процесс должен его открыть. Цель системного вызова - разрешить системе проанализировать атрибуты файла и проверить права доступа к нему, а также считать в оперативную память список адресов блоков файла для быстрого доступа к его данным. Открытие файла является процедурой создания *дескриптора* или управляющего блока файла;

4) *закрытие файла*. При завершении работы с файлом его атрибуты и адреса блоков на диске больше не нужны, поэтому файл нужно закрыть, чтобы освободить место во внутренних таблицах файловой системы;

5) *позиционирование*;

6) *чтение данных из файла*;

7) *запись данных в файл с текущей позиции* и т.д.

Способы выполнения последовательности действий над файлами:

1) для каждой операции выполняются как универсальные, так и уникальные действия (*схема stateless*);

2) универсальные действия выполняются в начале и в конце последовательности операций, а для каждой промежуточной операции выполняются только уникальные действия.

Большинство ОС использует второй способ, более экономичный и быстрый. Первый способ более устойчив к сбоям, поскольку результаты каждой операции становятся независимыми от результатов предыдущей операции, поэтому он иногда применяется в распределенных файловых системах (например, Sun NFS).

Эффективное управление файлами подразумевает наличие в системе четкой *логической структуры*. Все современные файловые системы поддерживают *многоуровневое именование файлов* за счет наличия во внешней памяти дополнительных файлов со специальной структурой - *каталогов* (или *директорий*). Каждый каталог содержит список подкаталогов и/или файлов, содержащихся в данном каталоге. Каталоги имеют один и тот же внутренний формат, где каждому файлу соответствует одна запись в файле директории. Число директорий зависит от системы. В ранних ОС имелась только одна корневая директория, затем появились директории для пользователей (по одной директории на пользователя). В современных ОС используется произвольная структура *дерева директорий*. Таким образом, файлы на диске образуют иерархическую древовидную структуру.

Существует несколько эквивалентных способов изображения дерева. Наиболее распространена структура *перевернутого дерева*, где верхнюю вершину называют *корнем*. Если элемент дерева не может иметь потомков, он называется *терминальной вершиной* или *листом* (файлом). *Нелистовые вершины* - справочники или каталоги содержат списки листовых и нелистовых вершин. Путь от корня к файлу *однозначно* определяет файл. Внутри одного каталога имена листовых файлов уникальны. Имена файлов, находящихся в разных каталогах, могут совпадать. Для того, чтобы однозначно определить файл по его имени (избежать коллизии имен), принято именовать файл так

называемым *абсолютным* или *полным именем (pathname)*, состоящим из списка имен вложенных каталогов, по которому можно найти путь от корня к файлу плюс имя файла в каталоге, непосредственно содержащем данный файл. То есть полное имя включает цепочку имен - *путь к файлу*, например, /usr/games/doom. Такие имена уникальны. Использование древовидных каталогов минимизирует сложность назначения уникальных имен.

Указывать полное имя не всегда удобно, поэтому применяют другой способ задания имени - *относительный* путь к файлу. Он использует концепцию рабочей или текущей директории, которая обычно входит в состав атрибутов процесса, работающего с данным файлом. Тогда на файлы в такой директории можно ссылаться только по имени, при этом поиск файла будет осуществляться в рабочем каталоге.

Для получения доступа к файлу и локализации его блоков система должна выполнить навигацию по каталогам. Отклонение от типовой обработки компонентов pathname может возникнуть в том случае, когда этот компонент является не обычным каталогом с соответствующим ему индексным узлом и списком файлов, а служит точкой *связывания* (точкой *монтирования*) двух файловых архивов.

В современных ОС принято разбивать диски на *логические диски*, иногда называемые *разделами* (partitions). Диск содержит иерархическую древовидную структуру, состоящую из набора файлов и директорий. В некоторых системах управления файлами требуется, чтобы каждый архив файлов целиком располагался в одном разделе. В этом случае полное имя файла начинается с имени дискового устройства, на котором установлен соответствующий диск (буквы диска). Например, c:\util\nu\ndd.exe. Такой способ именования используется в файловых системах DEC и Microsoft.

Как и в случае с файлами, система обязана обеспечить пользователя набором операций, необходимых для работы с директориями, реализованных через системные вызовы. Несмотря на то, что директории - это файлы, логика работы с ними отличается от логики работы с обычными файлами и определяется природой этих объектов, предназначенных для поддержки структуры файлового архива. Совокупность системных вызовов для управления директориями зависит от особенностей конкретной ОС.

Наличие в системе многих пользователей предполагает организацию *контролируемого доступа к файлам*. Выполнение любой операции над файлом должно быть разрешено только в случае наличия у пользователя соответствующих привилегий. Основной подход к защите файлов от несанкционированного использования - связать с каждым файлом или директорией *список прав доступа* (access control list), где перечислены имена пользователей и типы разрешенных для них способов доступа к файлу. Основная проблема реализации данного способа - список может быть длинным. Чтобы разрешить всем пользователям читать файл, необходимо всех их внести в список.

Лекция №13. Файловые системы, их особенности и реализация

Цель: получить представление о структуре и основных характеристиках файловых систем, а также о принципах реализации и их архитектуре.

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

Общая *структура файловой системы* представлена на рисунке Б.1. *Нижний уровень* - оборудование. Это в первую очередь магнитные диски с подвижными головками - основные устройства внешней памяти, представляющие собой пакеты магнитных *пластин* (поверхностей), между которыми на одном рычаге двигается пакет магнитных головок. Шаг движения пакета головок является *дискретным*, и каждому положению пакета головок логически соответствует *цилиндр* магнитного диска. Цилиндры делятся на *дорожки* (*треки*), а каждая дорожка размечается на одно и то же количество *блоков* (*секторов*) так, что в каждый блок можно максимально записать одно и то же число байтов. Следовательно, для обмена с магнитным диском на уровне аппаратуры нужно указать номер цилиндра, номер поверхности, номер блока на соответствующей дорожке и число байтов, которое нужно записать или прочитать от начала этого блока. Таким образом, диски могут быть разбиты на блоки фиксированного размера и можно получить доступ к любому блоку (организовать прямой доступ к файлам).

Непосредственно с устройствами (дисками) взаимодействует *система ввода-вывода*, которая предоставляет в распоряжение файловой системы используемое дисковое пространство в виде непрерывной последовательности блоков фиксированного размера. Система ввода-вывода работает 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 (1) до N. Размер логических блоков файла совпадает или является кратным размеру физического блока диска и может быть задан равным размеру страницы виртуальной памяти, поддерживаемой аппаратурой компьютера совместно с ОС.

В структуре системы управления файлами можно выделить *базисную подсистему*, которая отвечает за выделение дискового пространства конкретным файлам, и более *высокоуровневую логическую подсистему*, которая использует структуру дерева директорий для предоставления модулю базисной подсистемы необходимой ей информации, исходя из символического имени файла. Она также ответственна за *авторизацию* доступа к файлам.

Непосредственное взаимодействие с диском осуществляется с помощью компонента системы ввода-вывода ОС, называемого *драйвером диска*. Чтобы

избавить пользователя от сложностей взаимодействия с аппаратурой, была придумана *абстрактная модель файловой системы*.

В ОС используются несколько *методов выделения файлу дискового пространства*, для каждого из которых запись в директории, соответствующая символьному имени файла, содержит указатель, следуя которому можно найти все блоки данного файла. Различают методы:

1) *выделение непрерывной последовательности блоков* - каждый файл хранится как непрерывная последовательность блоков диска, при этом файл характеризуется адресом и длиной (в блоках). Файл, стартующий с блока b, занимает затем блоки b+1, b+2, ... b+n-1. Преимущества схемы: легко реализуется, так как достаточно определить, где находится первый блок; обеспечивает хорошую производительность, так как целый файл может быть считан за одну дисковую операцию. Недостатки: в зависимости от размера диска и среднего размера файла в большей или меньшей степени проявляется внешняя фрагментация; неприменим, если неизвестен максимальный размер файла; а поскольку содержимое диска изменяется, метод нерационален;

2) *связные списки* - запись в директории содержит указатель на первый и последний блоки файла. Каждый блок содержит указатель на следующий блок (рисунок Б.2). Внешняя фрагментация для данного метода отсутствует. Любой свободный блок может быть использован для удовлетворения запроса. Не нужно декларировать размер файла в момент создания, файл может расти неограниченно. Недостатки: при прямом доступе к файлу выборка логически смежных записей, физически занимающих несмежные секторы, требует много времени; способ ненадежен, поскольку наличие дефектного блока в списке приводит к потере информации в оставшейся части файла и потенциально к потере дискового пространства, отведенного под файл. Метод обычно не используется в чистом виде;

3) *таблица отображения файлов* - указатели хранятся в индексной таблице в памяти, которая называется *таблицей отображения файлов* (*FAT - file allocation table*). Запись в директории содержит только ссылку на первый блок. При помощи таблицы *FAT* можно локализовать блоки файла независимо от его размера. В тех строках таблицы, которые соответствуют последним блокам файлов, обычно записывается некоторое граничное значение, например, EOF. Достоинство: по таблице отображения можно судить о физическом соседстве блоков, располагающихся на диске, и при выделении нового блока можно легко найти свободный блок диска, находящийся поблизости от других блоков данного файла. Недостаток: необходимость хранения в памяти этой довольно большой таблицы;

4) *индексные узлы* - с каждым файлом связывается таблица (*индексный узел - i-node*), которая перечисляет атрибуты и дисковые адреса блоков файла. Запись в директории, относящаяся к файлу, содержит адрес индексного блока. По мере заполнения файла указатели на блоки диска в индексном узле принимают осмысленные значения. Индексирование поддерживает прямой доступ к файлу, без ущерба от внешней фрагментации. Индексированное

размещение широко распространено и поддерживает как последовательный, так и прямой доступ к файлу. Обычно применяется комбинация одноуровневого и многоуровневых индексов. Схему используют файловые системы Unix, а также файловые системы HPFS, NTFS. Подход позволяет при фиксированном, относительно небольшом размере индексного узла поддерживать работу с файлами, размер которых может меняться от нескольких байтов до нескольких гигабайтов.

Дисковое пространство, не выделенное ни одному файлу, также должно быть управляемым. В современных ОС используется несколько *способов учета используемого места на диске*. Наиболее распространенные среди них:

1) *учет при помощи организации битового вектора* - список свободных блоков диска реализован в виде битового вектора (bit map или bit vector). Каждый блок представлен одним битом, принимающим значение 0 или 1, в зависимости от того, занят он или свободен. Преимущество: метод относительно прост и эффективен при нахождении первого свободного блока или n последовательных блоков на диске. Используется в Apple Macintosh;

2) *учет при помощи организации связного списка* - в список связываются все свободные блоки, размещая указатель на первый свободный блок в специально отведенном месте диска, попутно кэшируя в памяти эту информацию. Недостаток: схема не всегда эффективна, поскольку для трассирования списка нужно выполнить много обращений к диску.

Существуют и другие методы, например, свободное пространство можно рассматривать как файл и ввести для него индексный узел.

Важную роль играет *размер логического блока*. В некоторых системах (Unix) он может быть задан при форматировании диска. Небольшой размер блока будет приводить к тому, что каждый файл будет содержать много блоков. Чтение блока осуществляется с задержками на поиск и вращение, поэтому файл из многих блоков будет читаться медленно. Большие блоки обеспечивают более высокую скорость обмена с диском, но из-за внутренней фрагментации снижается процент полезного дискового пространства.

Структура служебных данных типовой файловой системы на одном из разделов диска может состоять из четырех основных частей (рисунок Б.3). В начале раздела находится *суперблок*, содержащий общее описание файловой системы. Описанные структуры данных создаются на диске в результате его *форматирования*. Их наличие позволяет обращаться к данным на диске как к файловой системе, а не как к обычной последовательности блоков.

Файловая система позволяет при помощи системы справочников (*каталогов, директорий*) связать уникальное имя файла с блоками вторичной памяти, содержащими данные файла. *Иерархическая структура каталогов,* используемая для управления файлами, может служить другим примером индексной структуры. В этом случае каталоги или папки играют роль индексов, каждый из которых содержит ссылки на свои подкаталоги. Основными функциями файловой системы являются:

1) *идентификация файлов*;

2) *распределение внешней памяти между файлами*;
3) *обеспечение надежности и отказоустойчивости*;
4) *обеспечение защиты от несанкционированного доступа*;
5) *обеспечение совместного доступа к файлам*;
6) *обеспечение высокой производительности*.

Надежность – одна из важнейших характеристик файловой системы, поскольку ее разрушение зачастую более опасно, чем разрушение компьютера. Поэтому файловые системы должны разрабатываться с учетом подобной возможности. Помимо своевременного *дублирования информации* (*backup*), файловые системы современных ОС содержат специальные средства для поддержки собственной совместимости. Для надежной работы файловой системы важен *контроль ее целостности*. В современных ОС предусмотрены меры, которые позволяют свести к минимуму ущерб от порчи файловой системы, а затем полностью или частично восстановить ее целостность. К ним относятся:

1) *порядок выполнения операций*;
2) *журнализация*;
3) *проверка целостности файловой системы при помощи утилит*.

К сожалению, не существует средств, гарантирующих абсолютную сохранность информации в файлах, поэтому в ситуациях, когда целостность информации нужно гарантировать с высокой степенью надежности, прибегают к дорогостоящим *процедурам дублирования*.

Наличие *дефектных* блоков на диске - обычное дело. Внутри блока наряду с данными хранится *контрольная сумма данных*. Под «*плохими*» блоками понимают блоки диска, для которых вычисленная контрольная сумма *считываемых данных* не совпадает с *хранимой* контрольной суммой. Дефектные блоки обычно появляются в процессе эксплуатации. Существуют следующие решения проблемы дефектных блоков:

1) *хранить список плохих блоков в контроллере диска*;
2) *решение на уровне ОС*.

Для большинства пользователей файловая система - наиболее видимая часть ОС. Она предоставляет механизм для *онлайнового* хранения и доступа как к данным, так и к программам для всех пользователей системы. Наиболее типичная техника повышения скорости работы с диском - *кэширование*. *Кэш* диска представляет собой буфер в оперативной памяти, содержащий ряд блоков диска. Сокращение количества дисковых операций оказывается возможным вследствие присущего ОС свойства *локальности*. Аккуратная реализация кэширования требует решения нескольких проблем:

1) *емкость буфера кэша ограничена*. Когда блок должен быть загружен в заполненный буфер кэша, возникает *проблема замещения* блоков;

2) *«старение» информации в дисковых блоках*, образы которых находятся в буферном кэше. Несвоевременная *синхронизация* буфера кэша и диска может привести к нежелательным последствиям в случае отказов оборудования или программного обеспечения;

3) *проблема конкуренции процессов на доступ к блокам кэша* решается ведением списков блоков, пребывающих в различных состояниях, и отметкой о состоянии блока в его дескрипторе.

Рекомендуется периодически выполнять *дефрагментацию* диска (сборку мусора), поскольку в популярных методиках выделения дисковых блоков (кроме FAT) принцип локальности не работает, и последовательная обработка файла требует обращения к различным участкам диска.

Современные операционные системы предоставляют пользователю возможность работать сразу с несколькими файловыми системами. Файловая система в традиционном понимании становится частью более общей многоуровневой структуры (рисунок Б.4).

На верхнем уровне располагается так называемый *диспетчер файловых систем*. Он связывает запросы прикладной программы с конкретной файловой системой. Каждая файловая система на этапе инициализации регистрируется у диспетчера, сообщая ему точки входа, для последующих обращений к данной файловой системе. Та же идея поддержки нескольких файловых систем в рамках одной ОС может быть реализована по-другому, например, исходя из концепции *виртуальной файловой системы*. Виртуальная файловая система (vfs) представляет собой независимый от реализации уровень и опирается на реальные файловые системы. При этом возникают структуры данных виртуальной файловой системы типа виртуальных индексных узлов, которые обобщают индексные узлы конкретных систем.

Лекция №14. Системы программирования

Цель: получить представление о классификации, основных компонентах и особенностях функционирования типовой системы программирования.

Система программирования - это совокупность программных средств автоматизации разработки программ, их отладки и подготовки к выполнению. К основным компонентам системы относят *трансляторы* с их языками программирования, *редактор связей*, *загрузчик*, средства отладки, библиотеки трансляторов и утилиты. Система программирования работает под управлением ОС и может иметь с ней общие компоненты, например, загрузчик, библиотеки, утилиты.

По отношению к другим программным комплексам системы программирования подразделяются на *самостоятельные* и *встроенные*.

Самостоятельные системы программирования являются системами общего назначения и могут быть организованы как *программные комплексы с автономными компонентами* или как *интегрированные* системы программирования.

Вызов каждого автономного компонента осуществляется посредством указаний на командном языке ОС, поэтому их называют *инструментами командной строки.*

Все компоненты *интегрированной* системы программирования окружены единой программной оболочкой, которая обеспечивает графический интерфейс с пользователем, конструктивно собраны в единой выполняемой программе и поставляются вместе с инструментами командной строки. Некорректно отождествлять интегрированную систему программирования с компилятором.

Встроенная система программирования является составной частью другого программного комплекса, например, СУБД и наиболее развитых пакетах прикладных программ. По своей организации и возможностям они аналогичны интегрированным система программирования.

Несмотря на все различия, каждая система программирования выполняет определенный набор основных функций, присущих классической самостоятельной системе программирования с автономными компонентами. На рисунке 14.1 показана схема функционирования типовой системы программирования.

Современная система программирования основана на *модульном* принципе программирования, что позволяет разделять программы на функционально независимые части – *модули*.

Исходный модуль (ИМ) записывается на входном языке транслятора. Причем программа может состоять из одного или нескольких исходных модулей, которые транслируются независимо друг от друга или с использованием пакетной обработки. Все действия по подготовке и коррекции исходных текстов выполняются *текстовым редактором*. Тексты на исходном языке могут храниться в специальных *файлах включения* и при необходимости включатся в исходные модули. Отдельные объектные модули накапливаются в *библиотеках объектных модулей* (БОМ), для обслуживания которых служит специальная программа – *библиотекарь*. Она позволяет создавать и пополнять библиотеки, заменять и удалять модули, печатать оглавление.

Каждый исходный модуль перерабатывается транслятором в перемещаемый *объектный* модуль (ОМ), который содержит программу на машинном языке и вспомогательную информацию. Объектный модуль невозможно выполнить без дополнительной обработки, которая осуществляется *редактором связей*, в результате чего вырабатывается *загрузочный* модуль (ЗМ). Загрузочный модуль снабжается служебной информацией, которая обеспечивает настройку программы при ее загрузке для выполнения. Загрузка, настройка и инициализация выполняются при помощи *загрузчика*, который может на время выполнения программы включить в нее динамически вызываемые модули.

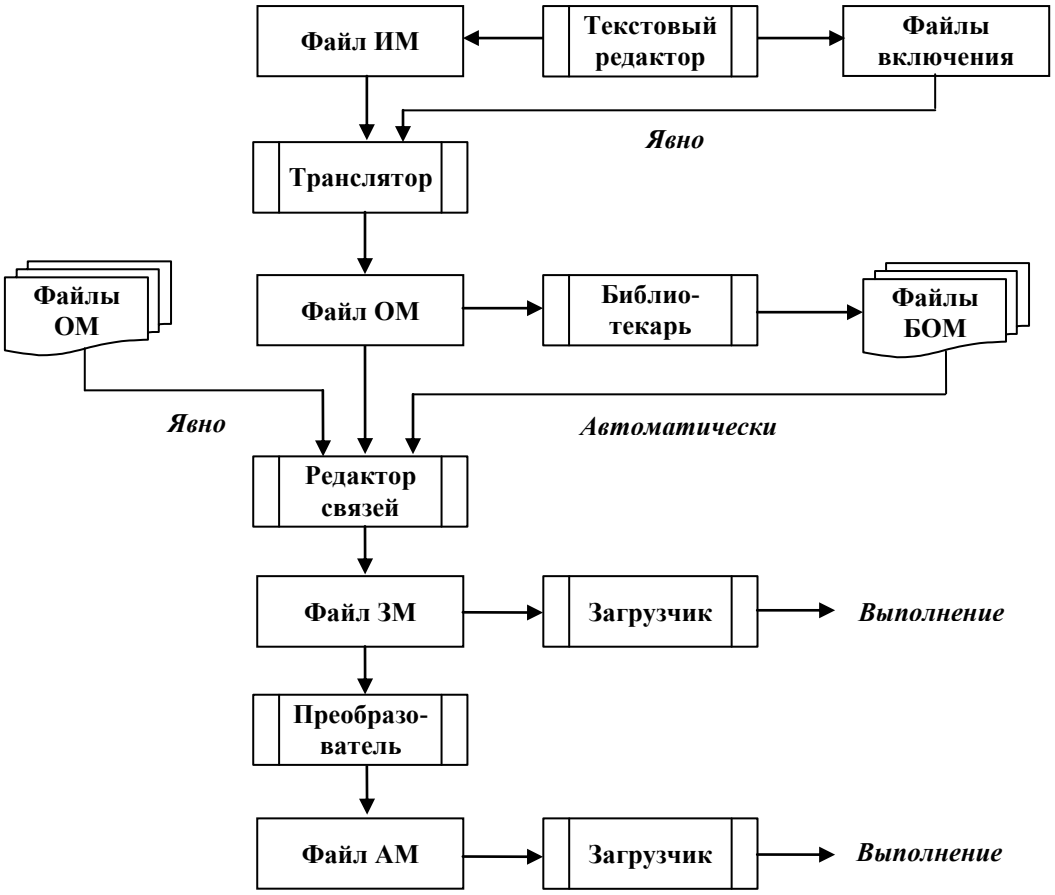

Рисунок 14.1 – Схема функционирования системы программирования

Особым образом написанные программы могут быть преобразованы в *абсолютный* модуль (АМ), который не содержит служебной информации и не нуждается в корректировке при размещении в памяти для выполнения. В современных системах программирования эта функция передана редактору связей.

Программа *актуализации* позволят автоматизировать сравнение дат создания или последней модификации исходного, объектного и загрузочного модулей, чтобы привести в соответствие все формы хранения одной программы.

Все основные компоненты классической системы программирования можно разделить на *трансляторы* и *компоновщики*.

Трансляторы – это общее название системных программ, которые выполняют преобразование программы, представленной на одном языке, в эквивалентную ей программу, представленную на другом языке. В зависимости от назначения трансляторы реализуются в разных вариантах, а именно:

1) *компилятор* осуществляет перевод программы с исходного языка на язык более низкого уровня, в результате чего формируется сохраняемый перемещаемый объектный модуль;

2) *ассемблер* – компилятор для языка ассемблера;

3) *интерпретатор* выполняет программу в машинно-независимой форме с помощью входящих в его состав подпрограмм, предварительно переводя ее в более удобную промежуточную несохраняемую программу;

4) *инкрементный транслятор* служит для повторной трансляции фрагментов программы и дополнений к ней без повторной трансляции всей программы;

5) *компилятор переднего плана* переводит исходную программу в сохраняемую промежуточную программу на машинно-независимом языке;

6) *препроцессор* реализуется как префиксная часть транслятора и осуществляет перевод программы с макрорасширения входного языка транслятора на этот входной язык;

7) *макрогенератор* – это препроцессор для ассемблера;

8) *кросс-компилятор*, транслируя программу на одной платформе, формирует объектный модуль для другой платформы;

9) *компилятор компиляторов* осуществляет перевод формального описания языка программирования в транслятор для этого языка;

10) *детранслятор* восстанавливает исходную программу на языке программирования по ее эквиваленту на машинном языке;

11) *дисассемблер* – детранслятор для языка ассемблера.

Компоновщики – общее название системных программ, выполняющих подготовку объектных и загрузочных модулей к выполнению, которые различаются набором функций.

Типовые функции компоновщиков:

- *перемещение* – настройка сегментов на их размещение в новом адресном пространстве;
- *связывание* – объединение двух и более модулей в единую программу;
- *перекрытие* – настройка модулей программы на выполнение с перекрытиями групп сегментов в основной памяти;
- *загрузка* – размещение программы в основной памяти для выполнения.

Разновидности компоновщиков:

1) *редактор связей* – формирует загрузочный модуль из объектного, реализуют функции связывания, перемещения, иногда перекрытия;

2) *связующий загрузчик* реализует все типовые функции кроме перекрытия и создает немедленно выполняющуюся несохраняемую программу;

3) *абсолютный загрузчик* реализует только функцию загрузки программы, имеющей форму абсолютного модуля;

4) *перемещающий загрузчик* выполняет функции перемещения и загрузки, готовит загрузочный модуль к выполнению в выделенной области основной памяти и инициирует программу.

На рисунке 14.2 показаны схемы применения различных компоновщиков.

В типовой системе программирования обычно предусмотрено наличие редактора связей и перемещающего загрузчика. В системах

программирования с развитым механизмом динамического связывания дополнительно используются специальные динамические загрузчики, которые осуществляют связывание и перемещение во время выполнения программы.

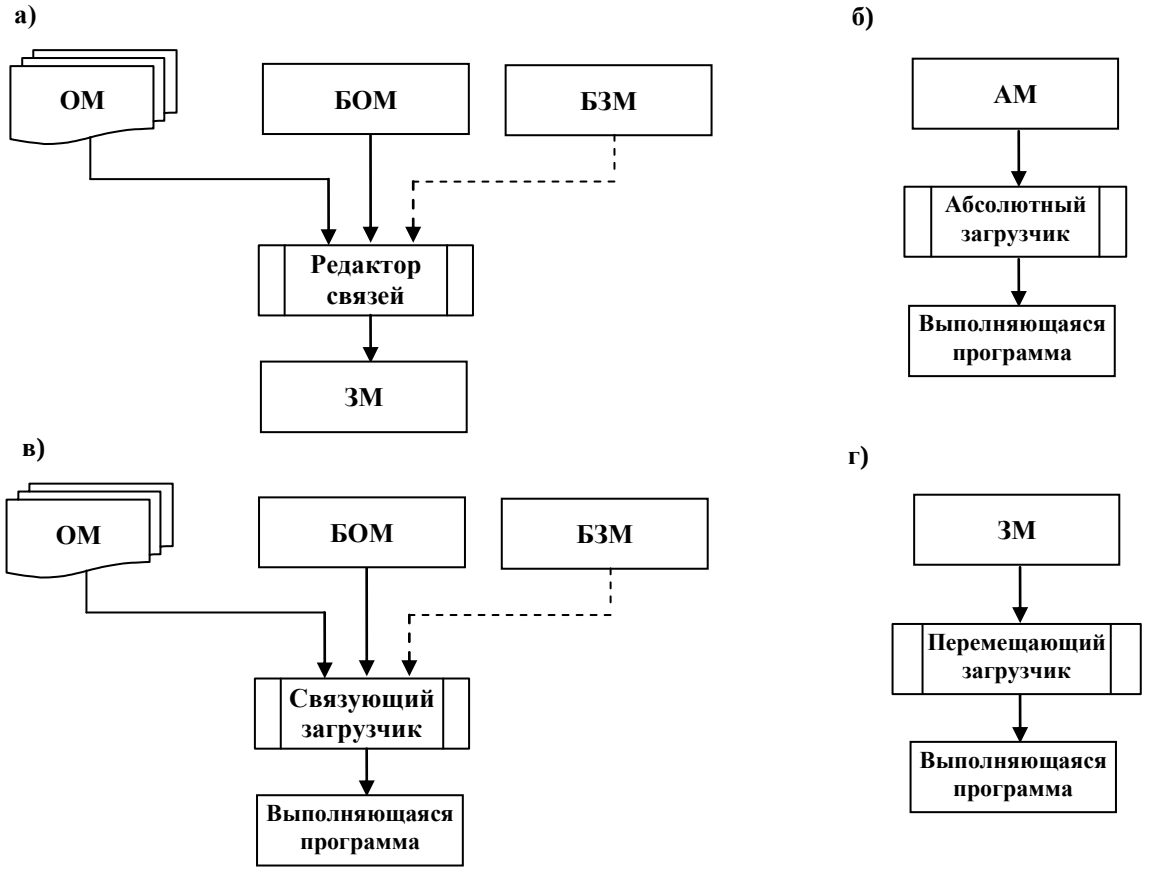

Рисунок 14.2 – Схемы применения а) редактора связей; б) абсолютного загрузчика; в) связующего загрузчика; г) перемещающего загрузчика.

Лекция №15. Библиотеки системы программирования

Цель: ознакомиться с назначением, составом и технологией применения наиболее используемых библиотек системы программирования.

Рассмотрим возможный состав библиотек на примере системы программирования *Borland C++*. Все библиотеки, которые *Borland C++* предоставляет своим пользователям, можно разделить на две части: стандартную библиотеку и дополнительные библиотеки. *Стандартная библиотека* системы программирования C++ организационно представляет собой множество библиотек, в котором выделяются три основные подмножества: стандартная библиотека языка C, графическая библиотека, библиотеки классов C++.

Стандартная библиотека языка C состоит из ряда библиотек, содержащих математические функции, функции обработки строк, функции стандартного ввода-вывода, функции управления памятью, различные

вспомогательные функции. Многие из перечисленных задач более эффективно решаются средствами других доступных библиотек, поэтому эта библиотека в значительной степени служит средством поддержания преемственности языков программирования, аппаратных компьютерных решений и операционных систем.

Графическая библиотека пришла в системы программирования из *MS DOS*. Она служит для формирования графического интерфейса пользователя (*GUI* – Graphic User Interface), а также обеспечивает управление графической системой, построение изображений на экране и получение параметров изображения. Функции этой библиотеки также перекрываются более развитыми средствами современных операционных систем.

Библиотека классов C++ представляет собой ряд библиотек: библиотеку классов ввода-вывода, ориентированного на поток; библиотеку контейнерных классов; библиотеку специализированных классов. В программах на языке C++ для организации ввода-вывода можно использовать функции как стандартной библиотеки C, так и библиотеки классов C++. Особенность применения классов состоит в том, что форма ввода-вывода объекта определяется не форматами, которые должен указать программист, а автоматически - классом объекта. Контейнерные классы обеспечивают работу с динамически изменяемыми данными (списками, очередями, стеками и т.п.). Специализированные классы ориентированы на обработку строковых данных и комплексных данных.

Дополнительные библиотеки системы программирования C++ являются либо принадлежностью операционной системы, либо ее специальными расширениями. Так, *библиотека объектов Windows* (*OWL* – Object Windows Library) входит в состав операционной системы, она поддерживает единый прикладной программный интерфейс (*API* - Application Program Interface) *Windows*, известный как *Win16 API* и *Win32 API*. Система программирования дает пользователям средства этой библиотеки для формирования внешнего оконного интерфейса разрабатываемых программ. *Библиотека* поддержки технологии *связывания и внедрения объектов* (*OLE* – Object Linking and Embedding) представляет собой расширение операционной системы. *OLE* - технология позволяет устанавливать «нетрадиционные» связи между программами. Каждая программа может искать и использовать необходимые ей программные ресурсы в других доступных программах, а также готова предоставить свои ресурсы другим программам. Технология ориентирована на поддержку гибких средств обеспечения внутреннего интерфейса.

В зависимости от времени, когда происходит подключение библиотечных функций к программе, различают библиотеки статические и динамические. Необходимые программе функции *статических библиотек* помещает в нее редактор связей. Эти функции составляют часть выполняемого модуля. Функции *динамических библиотек* подключаются к вызывающей их программе только во время ее выполнения. Библиотеки, функции которых работают во время выполнения программы, называют *библиотеками времени*

выполнения (*RTL* - Run Time Library). Многообразие библиотек *RTL* определяется широким спектром задач, для которых они предназначены, а также множеством возможных версий одной и той же библиотеки:

- статические или динамические библиотеки;
- 16- или 32-разрядные библиотеки;
- библиотеки для плоской (*f - flat*), средней (*m - middle*), большой (*l - large*) или других моделей памяти;
- библиотеки с диагностикой или без диагностики;
- библиотеки для однозадачных или мультизадачных приложений.

Формируемые на основе комбинаций признаков варианты библиотек обладают различными свойствами, предполагающими их применение в той или иной операционной и технической среде. Актуальные варианты библиотеки существуют одновременно, менее актуальные можно при необходимости достаточно быстро генерировать.

Статическая библиотека системы программирования Borland C++ хранится как библиотека объектных модулей, размещенная в *lib*-файле. Необходимые функции вызываются в программе явно в соответствии с общими правилами языка программирования и описаниями функций. Вызовы многих библиотечных функций включает в программу компилятор, они служат для формирования типовых фрагментов объектных программ. Правильность вызова функции компилятор проверяет по прототипам заголовков функций. Прототипы хранятся в заголовочных файлах с расширением *h*, которые включаются в программу с помощью препроцессорного оператора *#include*.

Динамическая библиотека системы программирования Borland C++ хранится как совокупность функций в форме, подобной загрузочному модулю, и чаще всего располагается в *dll*-файле. Вызов необходимых функций осуществляется с помощью соответствующих заголовочных файлов посредством *#include*, по аналогии со статической библиотекой. Однако если в программе не отметить каким-либо особым образом функции динамических библиотек, то компоновщик будет пытаться обрабатывать ссылки на них как ссылки на функции статических библиотек и обнаружит неудовлетворенные внешние связи. Идентификация запрашиваемых динамических функций осуществляется следующим образом. Из динамических библиотек заблаговременно создают особую статическую библиотеку, содержащую имена и некоторую другую информацию о функциях динамических библиотек. Такая библиотека называется *библиотекой импорта* и хранится в *lib*-файле. В Borland C++ заготовлены библиотеки импорта для многих вариантов динамических библиотек, например *import.lib* и *import32.lib* для библиотек Windows. Функция динамической библиотеки, используемая в программе, называется *импортируемой*. Эту же функцию в самой библиотеке называют *экспортируемой*. Библиотека импорта содержит информацию о функциях, которые соответствующие динамические библиотеки могут экспортировать. Обычно библиотеки импорта располагают в одном каталоге

со статическими библиотеками. Используя библиотеки импорта, компоновщик формально удовлетворяет в программе внешние ссылки на функции динамических библиотек и готовит для динамического загрузчика информацию о фактически вызываемых функциях. Импортируемые программой функции можно указать компоновщику другим способом: в секции импорта особого текстового файла - файла определения модуля с расширением *def*. Загрузочный модуль, оформленный как приложение *Windows* (в *PE*-формате), содержит таблицу импорта, специально предназначенную для динамического загрузчика. Когда программа запускается, включенные в нее функции статических библиотек выполняются обычным образом, как части этой программы. Динамически вызываемые функции могут быть загружены в память либо все одновременно (сразу после запуска программы), либо по одной (только в момент вызова в ходе выполнения программы).

Аналогичный набор библиотек имеется во всех современных системах программирования.

Приложение А

Операционные системы

Интерфейс пользователя
Управление вводом-выводом
Драйвер устройства связи оператора и консоли
Управление памятью
Планирование задач и процессов
Hardware

Рисунок А.1 - Слоеная система THE

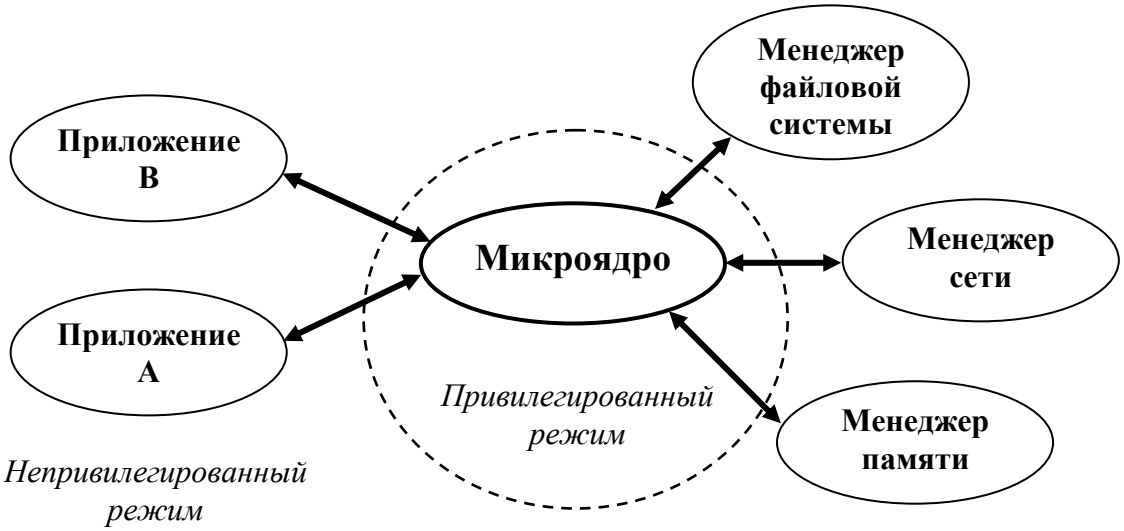

Рисунок А.2 - Микроядерная архитектура операционной системы

Рисунок А.3 – Распределение прерываний по уровням приоритета

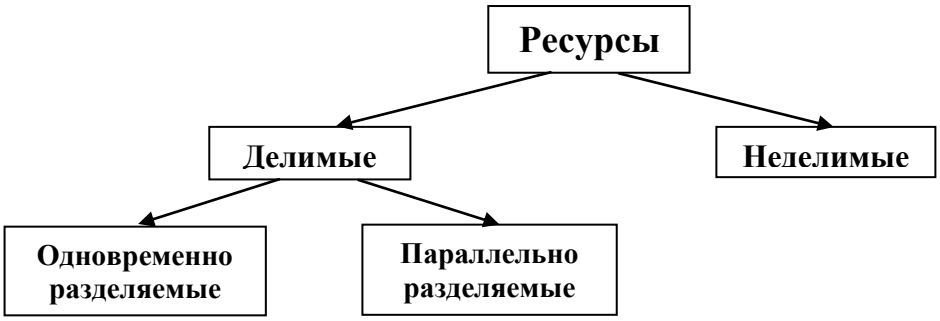

Рисунок А.4 – Классификация ресурсов

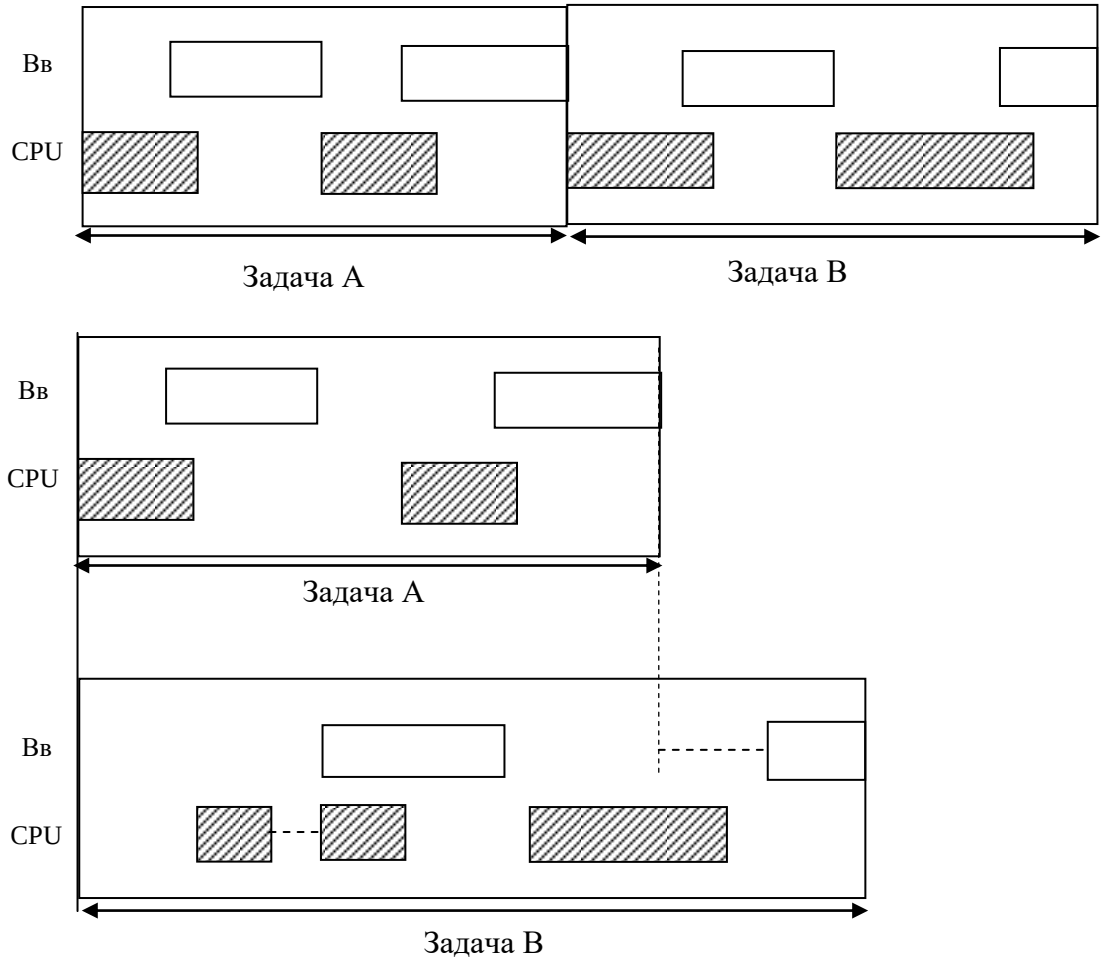

Рисунок А.5 – Выполнение программ в мультипрограммном режиме

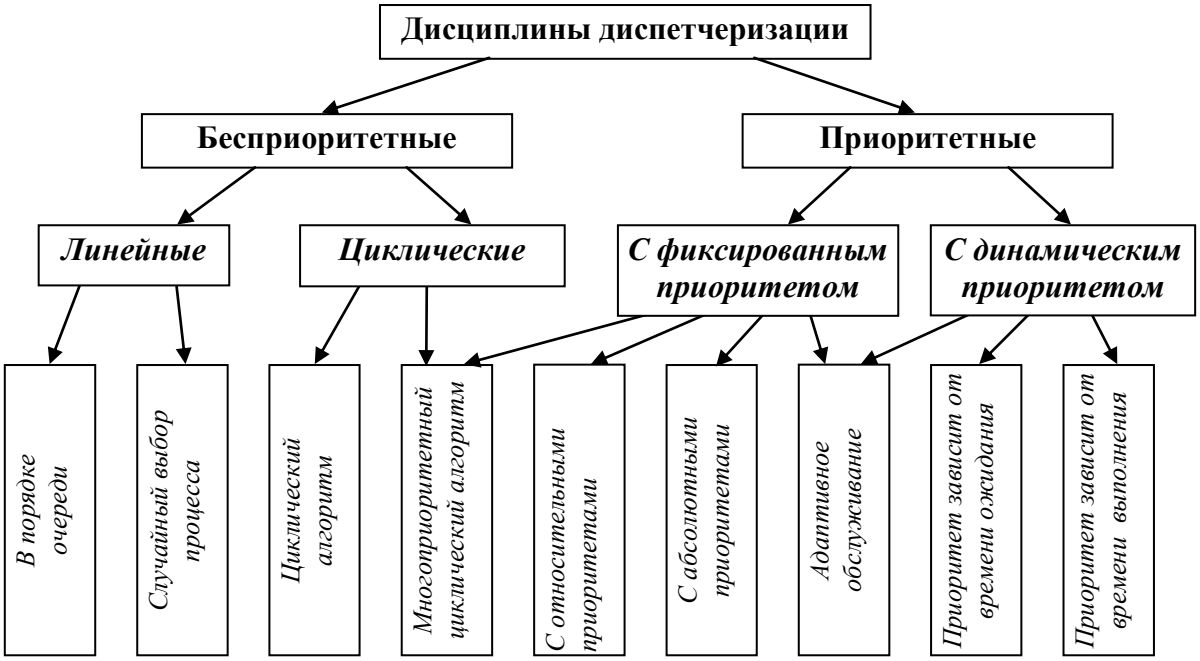

Рисунок А.6 – Классификация дисциплин обслуживания

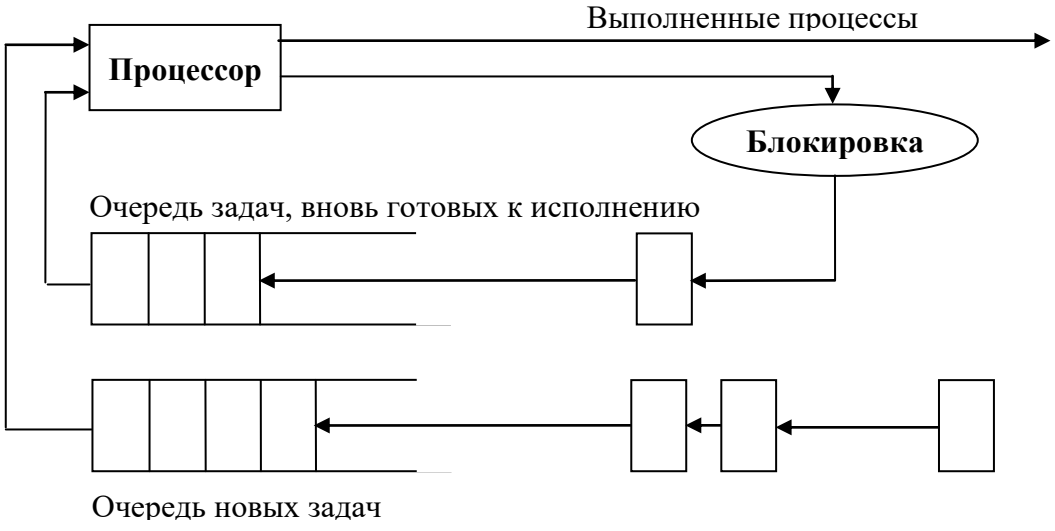

Рисунок А.7 – Дисциплина диспетчеризации FCFS

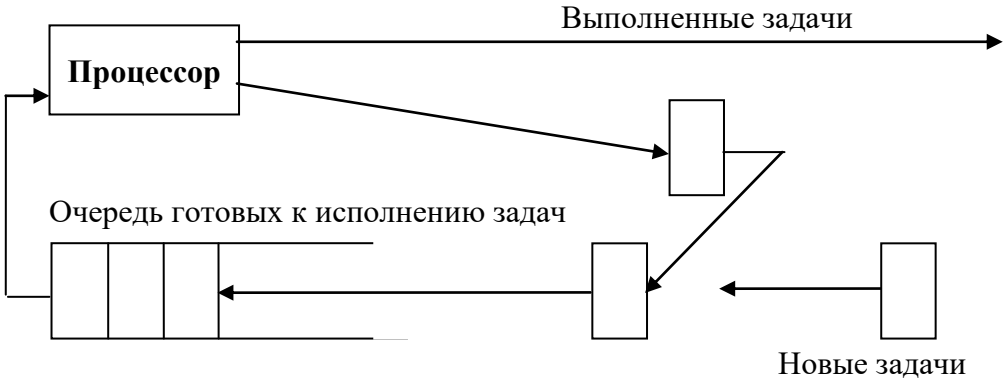

Рисунок А.8 – Карусельная дисциплина диспетчеризации

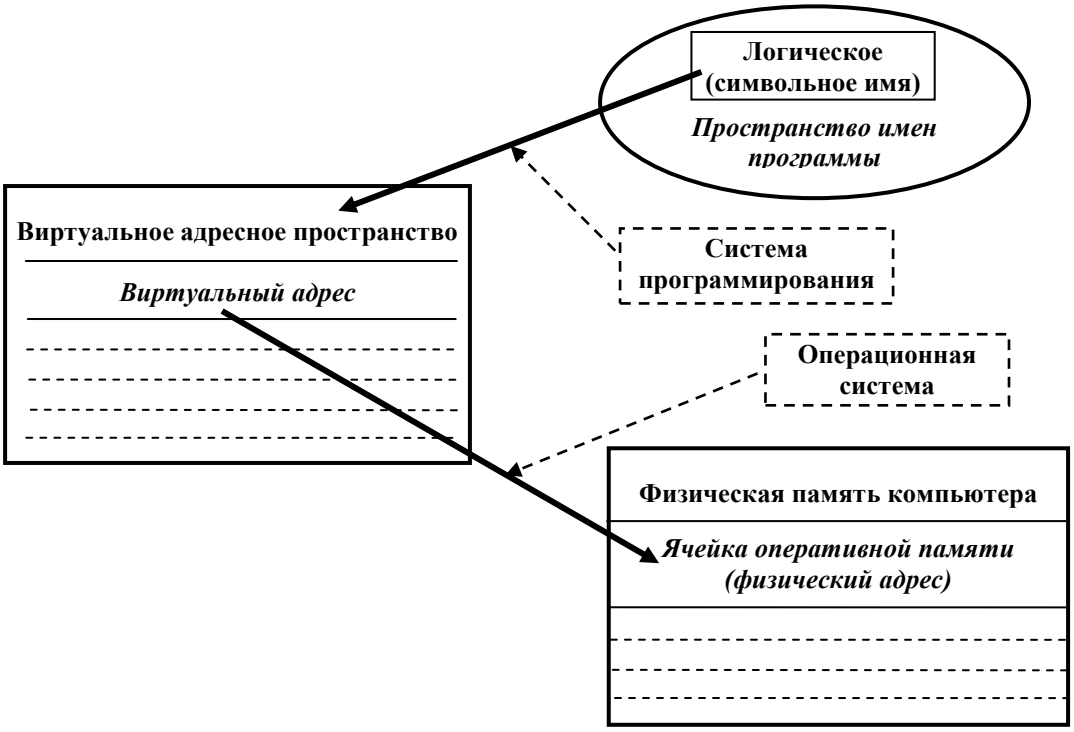

Рисунок А.9 – Память и отображения

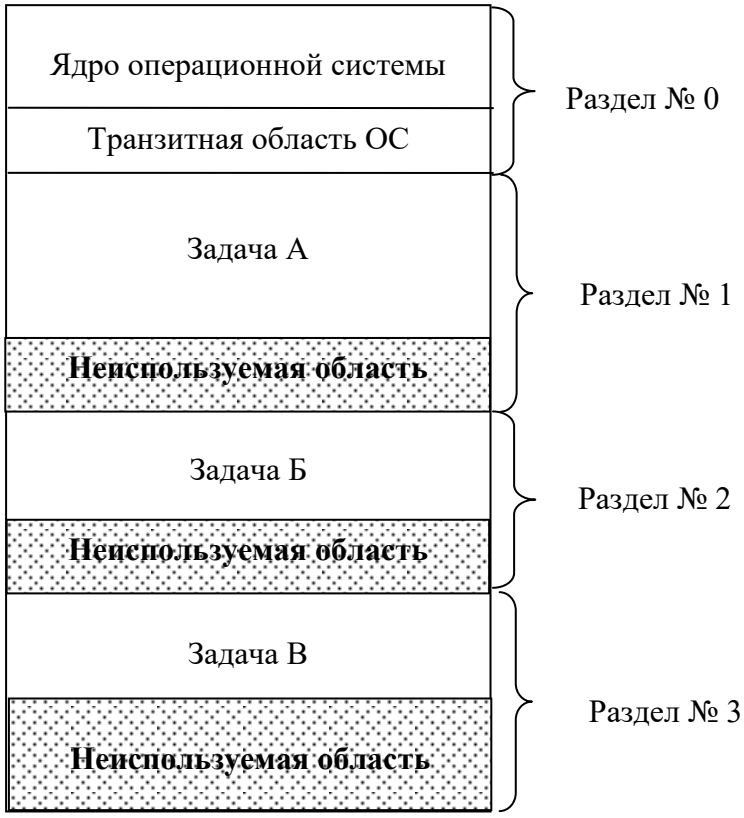

Рисунок А.10 - Распределение разделами с фиксированными границами

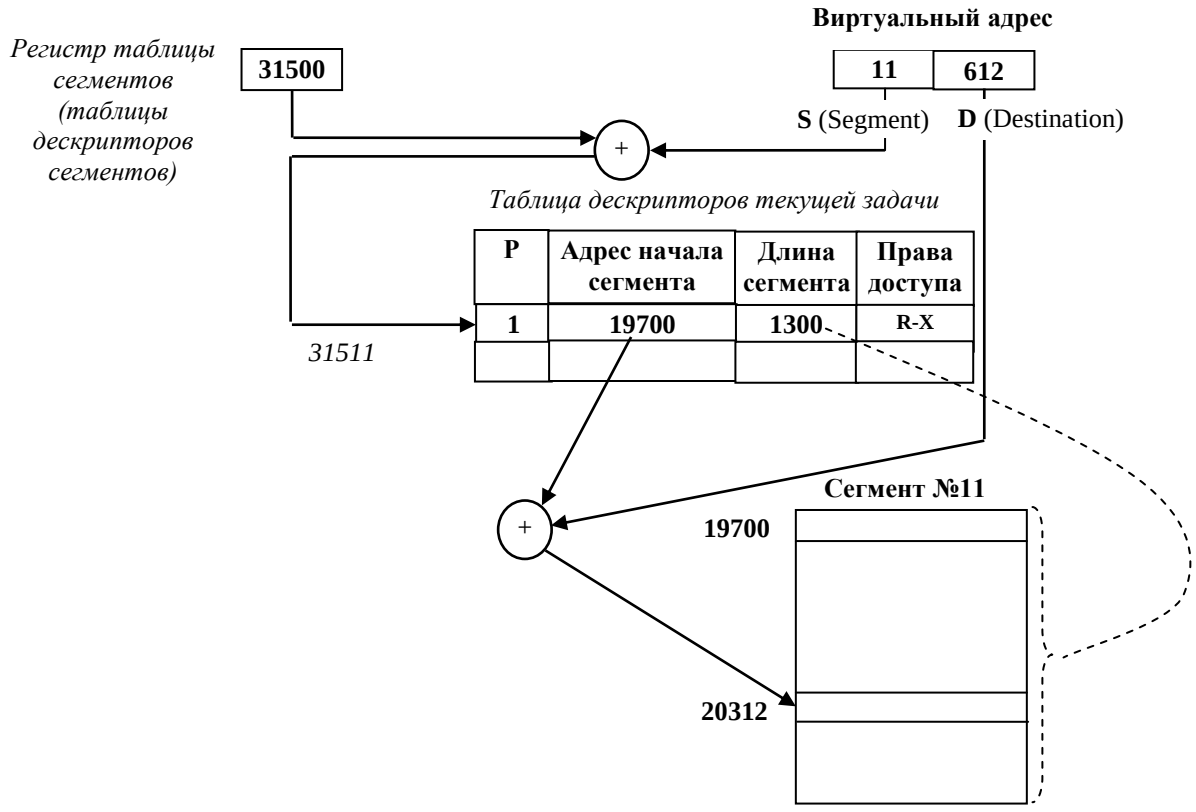

Рисунок А.11 – Сегментный способ организации виртуальной памяти

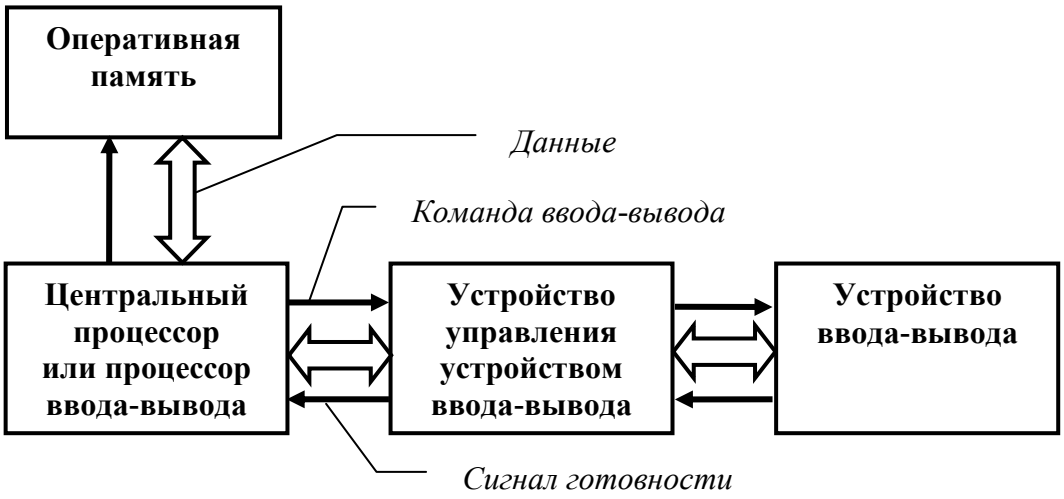

Рисунок А.12 – Управление вводом-выводом

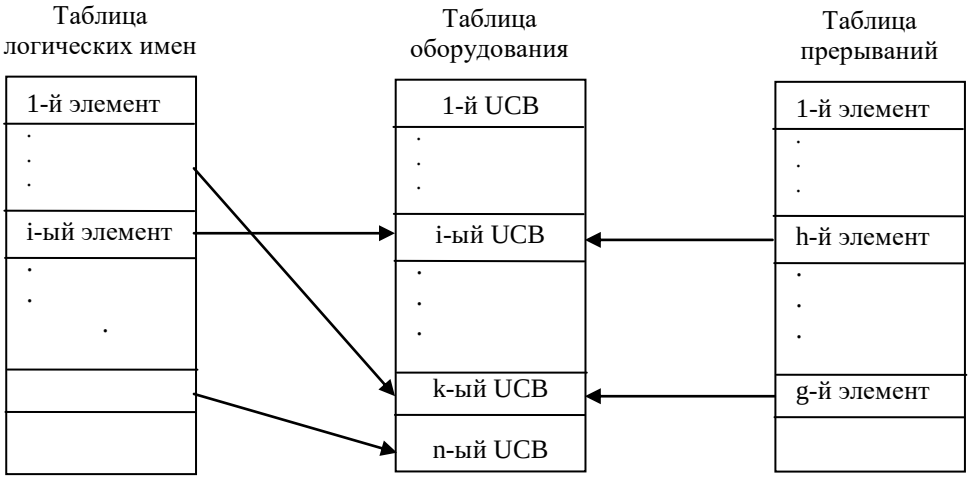

Рисунок А.13 – Взаимосвязи системных таблиц ввода-вывода

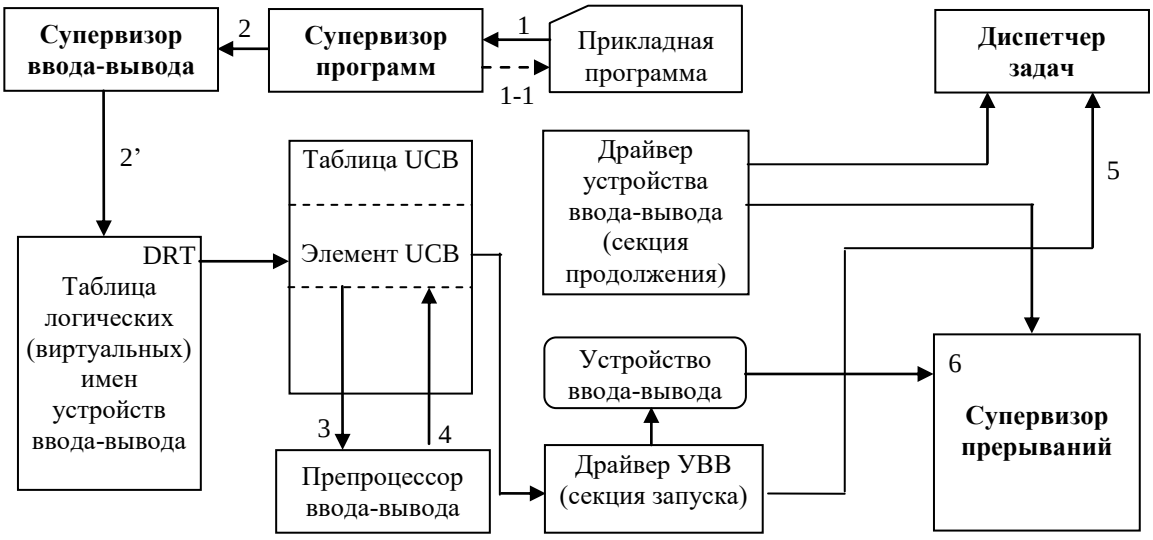

Рисунок А.14 – Процесс управления вводом-выводом

Приложение Б

Файловые системы

Рисунок Б.1 - Блок-схема файловой системы

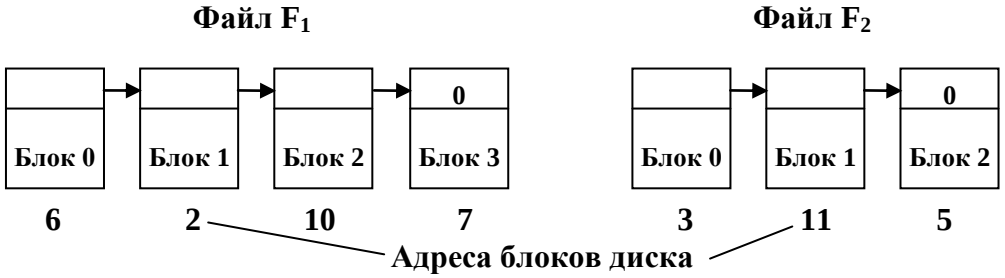

Рисунок Б.2 - Хранение файла в виде связного списка дисковых блоков

Суперблок	**Структуры данных, описывающие свободное дисковое пространство и свободные индексные узлы**	**Массив индексных узлов**	**Блоки диска данных файлов**

Рисунок Б.3 - Примерная структура файловой системы на диске

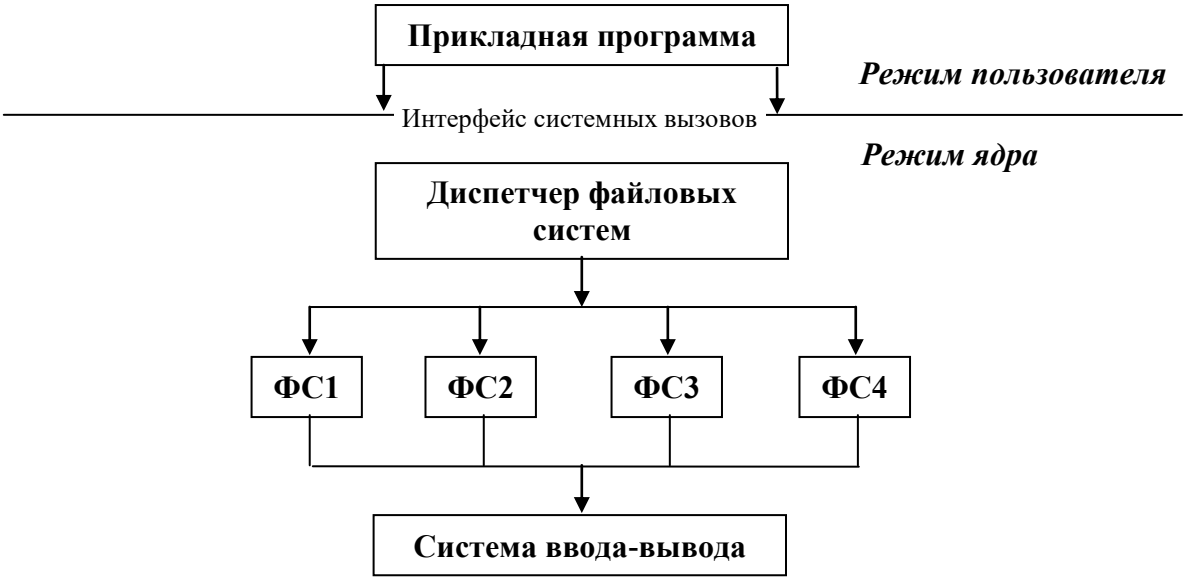

Рисунок Б.4 - Архитектура современной файловой системы

Список литературы

1 Мартемьянов Ю.Ф. Системное программное обеспечение. Концепции построения и обеспечения безопасности.- М.: «Горячая линия -Телеком», 2011.

2 Сафонов В.О. Основы современных операционных систем.-М.: «БИНОМ», 2011.

3 Молчанов А.Ю. Системное программное обеспечение. - СПб.: Питер, 2010.

4 Иртегов Д.В. Введение в Системное программное обеспечение. – СПб.:БХВ-Петербург, 2008.

5 Таненбаум Э. Современные операционные системы. - СПб.: Питер, 2010.

6 Гордеев В.А. Операционные системы. – СПб.: Питер, 2011.

7 Назаров С.В. Операционные среды, системы и оболочки. Основы структурной и функциональной организации. – М., 2007.

8 Олифер В.Г. Сетевые операционные системы. - СПб., 2008.

9 Соколов А.П. Системы программирования: теория, методы, алгоритмы: Учеб пособие. – М.: Финансы и статистика, 2004.

Сводный план 2014г., поз. 321

Наталья Валерьевна Сябина

СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ И ПРОГРАММИРОВАНИЕ

Конспект лекций
для студентов специальности
5В070200 – Автоматизация и управление

Редактор Н.М.Голева
Специалист по стандартизации Н.К.Молдабекова

Подписано в печать __.__.__.
Тираж 75 экз.
Объем 4,7 уч.-изд. л.

Формат 60x84 1/16
Бумага типографская №1
Заказ _____. Цена 2350 тг.

Копировально-множительное бюро
некоммерческого акционерного общества
«Алматинский университет энергетики и связи»
050013 Алматы, Байтурсынова, 126