Министерство образования Республики Беларусь

Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра программного обеспечения информационных технологий

П.Ю. Бранцевич

Лекционный материал

«Операционные системы и системное программирование»

Часть 1

Раздел 2

Операционные системы

Для студентов специальности
40 01 01
«Программное обеспечение информационных технологий»
дневной формы обучения

Минск 2008

ТЕМА 1 4
1.1 Введение в операционные системы 4
1.1.1 Понятие операционной системы 4
1.1.2 Поколения ОС 5
1.1.3 Функции и свойства ОС 7
1.1.4 Характеристики современных ОС 9
1.1.5 Архитектура микроядра 9
1.1.6 Многопоточность 10
1.1.7 Симметричная многопроцессорность 10
1.1.8 Распределенные ОС 11
1.1.9 Объектно-ориентированный дизайн 11
1.1.10 Концепция ОС на основе микроядра 11
1.1.11 Принципы построения ОС 16
ТЕМА 2 21
2.1 Основы операционных систем 21
2.1.1 Понятие процесса 21
2.1.2 Понятие ресурса 24
2.1.3 Концепция виртуализации 27
2.1.5 Дисциплины распределения ресурсов 30
2.1.6 Концепция прерывания 31
ТЕМА 3 33
3.1 Процессы 33
3.1.1 Состояние процессов 33
3.1.2 Описание процессов 36
3.1.3 Концепция потока как составной части процесса 40
3.1.4 Многопоточность 41
3.1.5 Однопоточная модель процесса 42
3.1.6 Многопоточная модель процесса 43
3.1.7 Функциональность потоков 44
3.1.8 Взаимодействие процессов 45
3.1.9 Синхронизирующие примитивы (семафоры). Применение семафоров для решения задачи взаимного исключения 55
3.1.10 Задача “производитель-потребитель” 56
3.1.11 Взаимодействие через переменные состояния 61
3.1.12 Мониторoподобные средства синхронизации 64
ТЕМА 4 72
4.1 Ресурсы 72
4.1.1 Распределение ресурсов. Проблема тупиков 72
4.1.2 Алгоритм банкира 73
4.1.3 Применение алгоритма банкира 75
ТЕМА 5 77
5.1 Память. Управление памятью 77
5.1.1 Требования к управлению памятью 78

5.1.2 Схемы распределения памяти 79
5.1.3 Система двойников при распределении памяти 80
ТЕМА 6 81
6.1 Организация виртуальной памяти 81
6.1.1 Структуризация адресного пространства виртуальной памяти 81
6.1.2 Задачи управления виртуальной памятью 81
ТЕМА 7 85
7.1 Планирование в операционных системах 85
7.1 Типы планирования процессора 85
7.2 Алгоритмы плинирования 90
7.3 Традиционное планирование в Unix 98
ТЕМА 8 100
8.1 Управление вводом-выводом и файлами 101
8.1.1 Организация функций ввода-вывода 101
8.1.2 Развитие функций ввода-вывода 102
8.1.3 Управление ОС и устройствами ввода-вывода 102
8.1.4 Модели организации ввода-вывода 103
ТЕМА 9 106
9.1 Аппаратно-программные особенности современных 106
процессоров, ориентированные на поддержку многозадачности 106
9.1.1 Сегментация памяти 106
9.1.2 Страничная организация памяти 118
9.1.3 Организация защиты при работе процессора в защищенном режиме 125
9.1.4 Поддержка многозадачности в процессорах архитектуры IA-32 131
9.1.5 Прерывания и особые случаи 140

ТЕМА 1

1.1 Введение в операционные системы

1.1.1 Понятие операционной системы

Операционная система – это набор программ, которые обеспечивают возможность использования аппаратуры компьютера. При этом аппаратные средства представляют собой некоторую вычислительную мощность, а задача операционной системы заключается в том, чтобы сделать аппаратные средства доступными и по возможности удобными для пользователя.

ОС реализует следующие основные функции:

- определяет или обеспечивает интерфейс пользователя (интерфейс – унифицированная система связи между пользователем и машиной);
- обеспечивает разделение аппаратных ресурсов между пользователями;
- дает возможность работать с общими данными в режиме коллективного пользования;
- планирует доступ пользователей к общим ресурсам;
- обеспечивает эффективное выполнение операций ввода-вывода;
- осуществляет восстановление информации и вычислительного процесса в случае ошибки.

ОС управляет следующими основными ресурсами:

- процессорами;
- памятью;
- устройствами ввода-вывода;
- данными.

ОС взаимодействует с:

- операторами ЭВМ;
- прикладными программистами;
- системными программистами;
- административным персоналом;
- пользователями;
- аппаратными средствами и программами.

Пользователи – это абоненты вычислительного комплекса, которые применяют компьютер для выполнения полезной работы.

Операторы ЭВМ – это специально подготовленные люди, которые следят за работой ОС и по запросам системы вмешиваются в работу компьютера для устранения каких-либо препятствий или выполнения необходимых действий.

Системные программисты занимаются сопровождением ОС, осуществляют ее настройку применительно к требованиям конкретной машины, решаемых задач и эксплуатируемых условий. При необходимости осуществляют разработку дополнительных системных программ, которые направлены на расширение функциональных возможностей системы и для обслуживания новых аппаратных устройств.

Администраторы – это люди, устанавливающие принципы и порядок работы на ЭВМ и взаимодействующие с ОС, чтобы обеспечить соблюдение принятого порядка.

1.1.2 Поколения ОС

Выделяют следующие поколения ОС:

Нулевое поколение (40-е гг. XX в). Характеризуется тем, что ОС на первых вычислительных машинах не было. Пользователи имели доступ к машинному языку и все программы писались на машинных кодах.

Первое поколение (50-е гг. XX в). ОС на этом этапе были разработаны с целью ускорения и упрощения перехода с задачи на задачу. Это было началом систем пакетной обработки. Пакетная обработка предусматривает объединение отдельных задач в группы. Каждая задача на этапе выполнения получает все ресурсы машины, а после завершения задачи(нормального или аварийного), управление ресурсами возвращалось ОС, которая очищала машину от последней решенной задачи и обеспечивала ввод и запуск следующей задачи. Считается, что первую ОС в начале 50-х гг. для IBM 701 создала исследовательская лаборатория фирмы General Motors.

Первые ОС ориентировались на сокращение времени, которое затрачивалось на запуск задачи на вычислительной машине и на удаление ее из машины. Т.е. цель – минимизация времени перехода с одной задачи на другую.

Уже в первых ОС появилась концепция имен системных файлов как средство достижения независимости программ от аппаратуры. Это позволило указывать программе не конкретный номер физического устройства, а стандартные системные файлы ввода-вывода.

К концу 50-х гг. были разработаны ОС, обладающие следующими характеристиками:

- пакетная обработка одного потока задач;
- наличие стандартных подпрограмм ввода-вывода. С целью освобождения пользователя от программирования процессов ввода-вывода на машинном языке;
- возможность автоматического перехода от программы к программе;
- наличие средств восстановления после ошибок, которые позволяли запускать следующую задачу при минимальном вмешательстве операторов;
- наличие языков управления заданиями, с помощью которых пользователи описывали свои задания и ресурсы, требуемые для их выполнения.

Второе поколение (первая половина 60-х гг.). Особенностью ОС этого поколения явилось то, что они создавались как системы коллективного пользования с мультипрограммным режимом работы и как первые системы мультипроцессорного типа.

В мультипрограммных системах несколько пользовательских программ одновременно находятся в оперативной памяти компьютера, а центральный процессор быстро переключается с задачи на задачу. Мультипроцессорные системы содержат несколько процессоров с целью повышения вычислительной мощности. В это время разработаны методы, обеспечивающие независимость программирования от внешних устройств, что привело к тому, что пользователь указывал не конкретное физическое устройство ввода-вывода, а определял характеристики, которым должно отвечать устройство ввода-вывода, а ОС сама находила соответствующее устройство и при необходимости давала оператору указание подготовить это устройство к работе.

На этом этапе разработаны первые системы с разделением времени, которые позволяли пользователю непосредственно взаимодействовать с компьютером при помощи пультов-терминалов телетайпного типа. С системой разделения времени пользователи работают в диалоговом или интерактивном режимах. Это позволило значительно повысить эффективность разработки программ. Наиболее распространенная система такого типа – программа SABRE бронирования и продажи билетов на самолеты компании American Airlines.

Чуть позже появились первые системы реального времени, в которых компьютеры применялись для управления технологическими процессами. Для систем реального времени характерно то, что они обеспечивают реакцию на предусмотренные события за время, не превышающее некоторое допустимое для данного события.

Третье поколение (середина 60-х – конец 70-х гг.). Это поколение связано с созданием и развитием больших универсальных машин System360/370 фирмы IBM. Эти машины относились к кассу машин общего назначения. Но в СССР был разработан аналог машин - ЕС.

ОС третьего поколения стали многорежимными системами, т.е. некоторые из них обеспечивали работу практически во всех режимах(пакетная обработка, режим разделения времени, режим реального времени, мультипроцессорный режим).Такие ОС были громоздкими и дорогостоящими. Сроки и затраты на их реализацию значительно превышали планирование.

ОС стали толстой прослойкой между пользователем и вычислительной машиной. Пользователю приходилось изучать сложные языки управления заданиями, чтобы уметь описать задание и требуемые для их выполнения ресурсы.

Четвертое поколение (конец 70-х – начало 90-х гг.). Этот этап имеет следующие особенности:

- широкое распространение вычислительных сетей и средств обработки данных в режиме on-line, т.е. пользователи получают доступ к территориально распределенным компьютерам при помощи терминалов различного типа;

- появление микропроцессоров создало условия микрокомпьютера, который привел к существенным социальным последствиям;

- потребность передачи информации по линиям связи различных типов потребовала создания систем защиты данных, т.е. шифрованию, созданию ключей защиты, разделению прав доступа к информации;

- большое внимание стало уделяться созданию ОС, ориентированных на неподготовленного пользователя, созданию дружественных пользовательских интерфейсов.

В это время появились системы с управлением при помощи меню. Начала широко распространяться концепция виртуальных машин. Важную роль стали играть системы баз данных (БД). Получила распространение концепция о распределенной обработке данных. Классический пример ОС этого поколения – ОС UNIX.

Пятое поколение (начало 90-х гг.). Отличительные особенности этого этапа следующие.

1. Интенсивное развитие и широкое распространение и развитие персонального компьютера.
2. Переход компьютера из сферы производства в бытовую сферу.
3. Развитие локальных и глобальных сетей, распространение Interneta.
4. Рост вычислительной и информационной мощности.
5. Решение нетрадиционных задач для вычислительной техники.
6. Совершенствование пользовательского интерфейса.
7. Развитие графических пользовательских интерфейсов.
8. Появление и развитие новых устройств для взаимодействия пользователя и вычислительной машины (манипуляторы, экраны управления, цифровые камеры, речевые вводы и др.).
9. Развитие интеллектуальных систем и систем искусственного интеллекта.
10. Создание супер-ЭВМ, обладающих огромной вычислительной мощностью.

Получили развитие многозадачные ОС.

1.1.3 Функции и свойства ОС

Операционная система (ОС) – это упорядоченная последовательность системных управляющих программ, совместно с необходимыми информационными массивами, предназначенных для планирования, исполнения пользовательских программ и управления всеми ресурсами вычислительной машины (программами, данными, аппаратурой и другими распределяемыми и управляемыми объектами) с целью предоставления возможности пользователям эффективно, в некотором смысле, решать задачи, сформулированные в терминах вычислительной машины.

ОС – это программа, которая контролирует работу прикладных программ и системных приложений и выполняет роль интерфейса между приложениями и аппаратным обеспечением ЭВМ (это более простое определение).

Можно определить следующие параметры ОС:

1. Удобство (ОС делает использование ПК достаточно простым и удобным)

2. Эффективность (ОС позволяет эффективно использовать ресурсы ПК)

3. Возможность развития (ОС должна быть организована так, чтобы допускать эффективную разработку, тестирование и внедрение новых приложений и системных функций, причем это не должно мешать нормальному функционированию ОС).

Рассматривая ОС как интерфейс между пользователем и компьютером, можно представить иерархическую структуру программного и аппаратного обеспечения, использующегося для представления конечному пользователю возможности работы с приложениями.

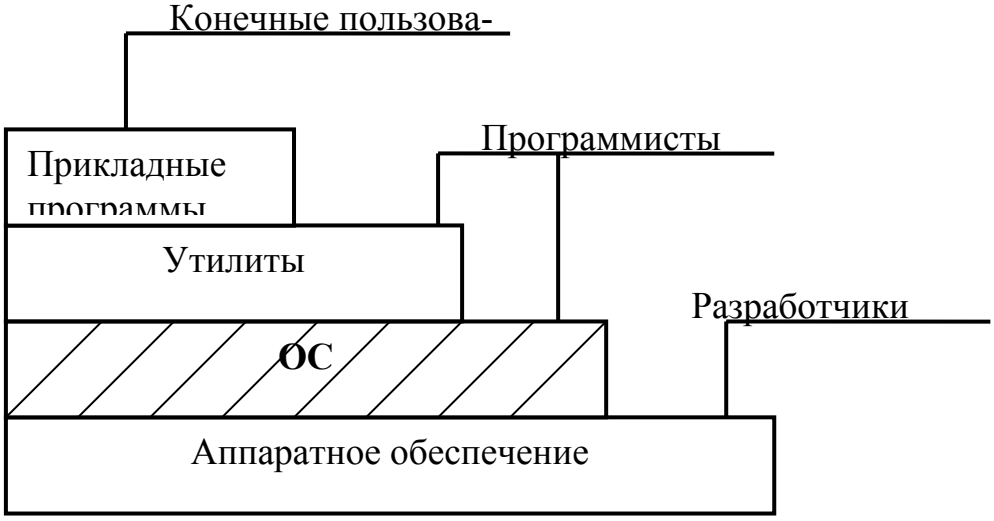


Рис. 1.1. Место операционной системы в вычислительной системе

ОС предоставляют следующий перечень услуг:

1. Разработка программ. ОС предоставляет программисту разнообразные инструменты и сервисы, к примеру, редакторы, отладчики, реализованные в виде программ-утилит, которые поддерживаются ОС, хотя не входят в её ядро. Такие программы называются *инструментами разработки приложений.*

2. Исполнение программ. Для запуска программы требуется загрузить её в основную память (сформировать области команд и данных), инициализировать устройства ввода-вывода и файлы, подготовить ресурсы ЭВМ. ОС выполняет

эти трудоемкие операции. Существуют специальные загрузчики программ для выполнения этих действий.

3. Доступ к устройствам ввода-вывода. Несмотря на то, что для каждого устройства нужен свой набор команд и контроллеров сигналов, ОС предоставляет пользователю единообразный интерфейс, который скрывает эти детали и обеспечивает программисту доступ к устройствам ввода-вывода с помощью простых команд чтения-записи.

4. Контролируемый доступ к файлам. При работе с файлами обеспечивается необходимая структуризация данных, записываемая в файлы, а также работа механизма защиты при обращении к файлу многопользовательскими ОС (классический пример – ОС UNIX).

5. Системный доступ. ОС управляет доступом к отдельным системным ресурсам, а также доступом к вычислительной системе в целом. Она защищает ресурсы и данные от несанкционированного использования и разрешает конфликтные ситуации (например, проблема тупиков).

6. Обнаружение ошибок и их обработка. В ходе работы ЭВМ возможны сбои, например ошибки памяти, нарушение работы отдельных устройств, возможные программные ошибки (переполнения, попытки обращения к недоступным ресурсам). В этих случаях ОС должна выполнять действия, минимизирующие влияние ошибки на работу приложения. Спектр таких действий достаточно широк, от простого уведомления об ошибке до аварийной остановки программы.

7. Учет использования ресурсов. В ОС должны быть средства учета использованных ресурсов и отображения параметров их производительности, а также определение времени использования ресурса отдельными пользователями. Это особенно важно при настройке конфигурации. Наличие таких средств позволяет оптимизировать работу системы в целом и обеспечить наиболее оптимальную загрузку процессора и других ресурсов, т. е. повысить ее производительность.

1.1.4 Характеристики современных ОС

В экспериментальных коммерческих ОС были опробованы различные подходы и структурные элементы, большинство из которых можно объединить в следующие категории:

– архитектура микроядра;
– многопоточность;
– симметричная многопроцессорность;
– распределенные ОС;
– объектно-ориентированный дизайн.

1.1.5 Архитектура микроядра

В настоящее время многие ОС реализованы по принципу большого монолитного ядра. Такое ядро обеспечивает планирование выполнения программ, работу с файловой системой, сетевые функции, работу драйверов различных устройств, управление памятью и многие другие функции.

Монолитное ядро реализуется как единый процесс, все элементы которого реализуют одно и то же адресное пространство.

В архитектуре микроядра к функциям ядра относятся только самые важные, такие как работа с адресным пространством, обеспечения взаимодействия между процессами, основное планирование. Обеспечение остальных услуг ОС возможно на процессы, которые запускаются в пользовательском режиме, и микроядро работает с ними, как с другими приложениями. Такие процессы иногда называются ***серверами.***

Такой подход позволяет разделить задачу разработки ОС на разработку ядра и разработку сервера. Это с одной стороны упрощает реализацию системы, обеспечивает её ёмкость, возможность настройки на требования конкретных приложений и среды, а также этот подход хорошо вписывается в концепцию распределения системы.

1.1.6 Многопоточность

Это технология, при которой процесс, выполняющий приложения, разделяется на несколько одновременно выполняющихся потоков.

Поток – это диспетчиризированная единица работы, включающая контекст процессора, а также свою собственную область стека.

Команды потока выполняется последовательно, и поток может быть прерван при переключении процессора на обработку другого потока. У потоков как правило общее адресное пространство, в отличии от процессов.

Процесс – это набор из одного или нескольких потоков, а также связанных с этими потоками системными ресурсами, такими как область памяти для хранения кода и данных, открытые файлы, различные устройства.

Концепция процесса близка к концепции выполняющейся программы. Разбивая приложения на несколько потоков, программист получает преимущество модульности приложения и возможность управления связанными с приложением временными событиями.

Многопоточность оказывается полезной для приложений, выполняющих несколько независимых заданий, которые не требуют последовательного выполнения.

Потоки иногда называют ***облегченными процессами***.

1.1.7 Симметричная многопроцессорность

Этот термин относится к архитектуре аппаратного обеспечения компьютера, предполагающую наличие нескольких процессов (минимум 2), а также образу ОС, поддерживающей такую аппаратную архитектуру.

Симметричная многопроцессорность определяется следующими характеристиками:

1. В системе имеется несколько процессоров;

2. Процессоры соединены меду собой коммуникационной шиной и совместно используют одну и ту же основную память и одни и те же устройства ввода-вывода;

3. Все процессоры могут выполнять одни и те же функции.

Считается, сто многопроцессорные системы имеют несколько потенциальных преимуществ по сравнению с однопроцессорными:

- производительность;
- надежность;
- наращивание;
- масштабируемость.

В действительности эти преимущества не всегда реализуются. В частности, если задача строго линейна (т.е. не распараллеливается), то и выигрыша никакого не будет.

1.1.8 Распределенные ОС

Сущность этой концепции состоит в том, что создается так называемый кластер, состоящий из нескольких отдельных компьютеров, каждый из которых обладает собственной основной и вторичной памятью и своими устройствами ввода-вывода. Распределенные ОС создают видимость единого пространства основной и вторичной памяти и единой файловой системы.

К таким системам относятся системы типа клиент-сервер, но пока развитие таких систем находится в стадии развития.

1.1.9 Объектно-ориентированный дизайн

Помогает навести порядок в процессе добавления к основному небольшому ядру дополнительных модулей. Такой подход облегчает разработку пользовательских приложений, а также производить настройку ОС не нарушая её целостности. Классический пример таких систем –Windows.

1.1.10 Концепция ОС на основе микроядра

Первые ОС, которые разрабатывались в 50-х годах, можно отнести к классу *монолитных ОС*. В таких системах почти все процедуры могли вызывать одна другую. Такой подход практически не допускал расширяемости ОС.

Следующий этап, который во многом связан с машинами IBM, характерен разработкой *слоистых ОС*, основанных на иерархической организации функ-

ций. Причём взаимодействие возможно только с функциями, находящимися на соседних уровнях.

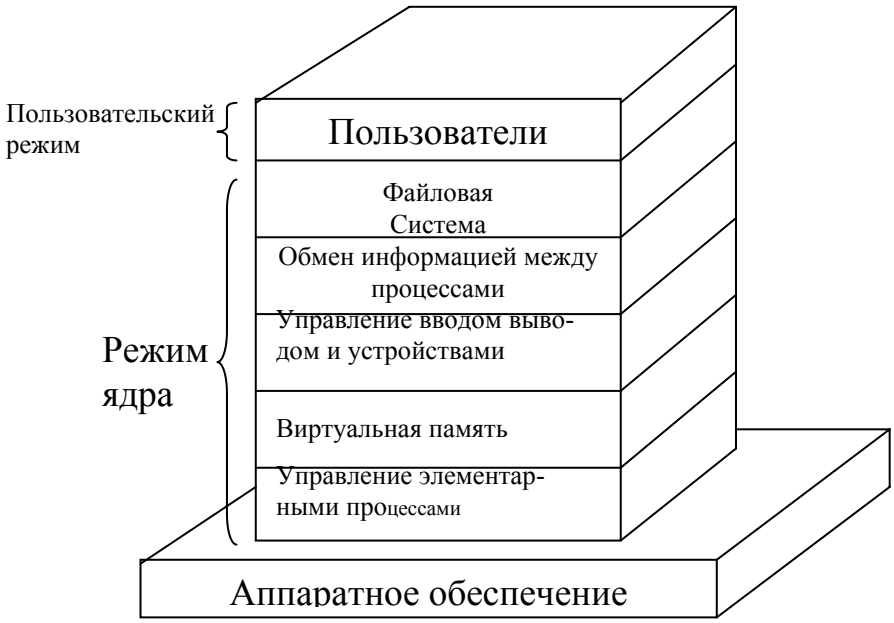


Рис.1.2. Структура слоистых ОС

Такой подход позволяет строить более надёжные ОС, позволяющие осуществлять модификацию, однако если потребуется ввести новые функции в слой, то необходимо вносить изменения и в соседние слои для возможности обращения к этим функциям. Другая проблема – это проблема безопасности, т. к. между слоями много точек обмена.

Основой концепции ОС на основе микроядра является то, что вертикальное расположение уровней заменяется горизонтальным, а приложения и услуги не являющиеся критическими, работают в пользовательском режиме. И хотя выбор того, что должно располагается в ядре, а что выносится за его пределы, зависит от архитектуры системы, общая тенденция такова, что многие службы, которые раньше размещались в ядре, теперь располагаются на уровне внешних подсистем, которые взаимодействуют с ядром и друг с другом, т. е. в ядре должны располагаться только самые важные функции. К таким подсистемам относятся драйвера устройств, файловые системы, менеджер виртуальной памяти, системы управления окнами, служба безопасности.

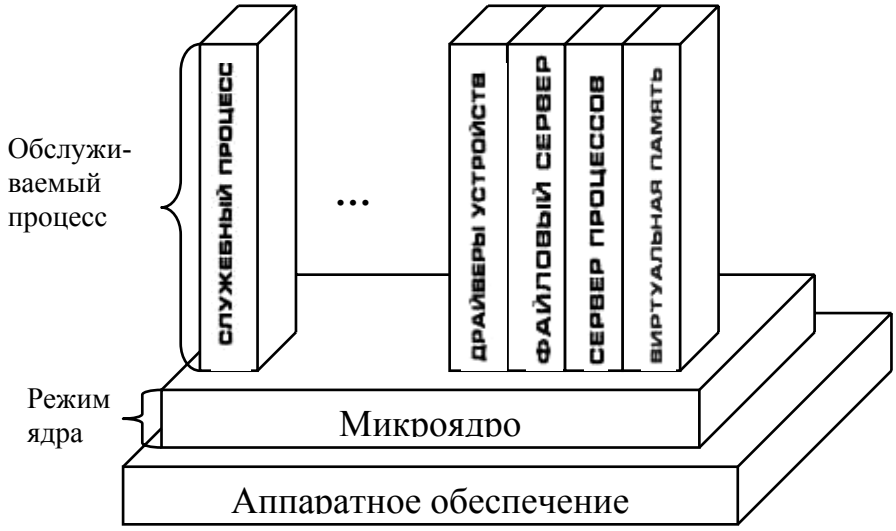


Рис. 1.3. ОС на основе микроядра

Процессы выполняются на пользовательском уровне, взаимодействуя между собой как равноправные, и обычно взаимодействие осуществляется с помощью обмена сообщениями, которые передаются через микроядро. Микроядро выступает в роли посредника: оно подтверждает правильность сообщений, передаёт их от одного компонента другому, предоставляет доступ аппаратному обеспечению. Микро ядро выполняет и защитные функции: оно не пропускает сообщение, если такой обмен не разрешен.

Достоинства концепции микроядра.

1. *Единообразные интерфейсы*. Используются для запросов, генерируемых процессами. Процессам ненужно различать приложения, выполняемые на уровне ядра и на пользовательском уровне, так как доступ ко всем службам осуществляется только с помощью передачи сообщений.

2. *Расширяемость*. Имеется возможность добавлять в ОС новые услуги или сервисы, а также обеспечивать множественную реализацию сервисов в одной и той же функциональной области. Например, можно организовывать несколько различных способов хранения файлов на дисках. Добавление новой службы в ОС требует модификации лишь некоторых других служб и не требует изменять микроядро.

3. *Гибкость*. В ОС можно не только добавлять новые услуги, но и удалять некоторые из них. Это может потребоваться для получения компактной и эффективной версии. Но если пользователю не требуются некоторые подсистемы, занимающие большие объёмы памяти, он может скомпоновать компактную ОС по своим нуждам.

4. *Переносимость*. Программный код, который взаимодействует с аппаратными средствами, или большая его часть, находится в микроядре. Поэтому уменьшаются объёмы работ, связанных с переносом ОС на новую аппаратную платформу (процессор).

5. *Надёжность*. Чем больший код имеет программа, тем труднее её протестировать. Однако небольшое ядро ОС можно тщательно проверить. А небольшое число интерфейсов прикладного программирования позволяет реализовать подсистемы, работающие вне ядра, с достаточно качественным программным кодом. Имея стандартизованный набор интерфейсов, разработчик отдельного приложения не может повлиять на другие системные компоненты.

6. *Микроядро способствует поддержке распределенных ОС*. Сообщение, которое передается от обслуживаемых сервисов к обслуживающим должно содержать идентификатор запрашиваемой услуги. Если система такова, что все процессы и сервисы обладают в ней уникальными идентификаторами, то на уровне микроядра образуется единый образ системы. Процесс может отправлять сообщение, не зная, на какой именно машине выполняется приложение, к которому он обращается.

7. *Подход на основе микроядра хорошо функционирует среди объектно-ориентированных ОС*. Такой подход способствует более строгой разработке ядра и модульных расширений ОС. Перспективным считается подход, в котором сочетаются архитектура с микроядром, принципы объектно-ориентированных систем, которые реализуются с использованием компонентов. Компоненты – объекты с четко заданными интерфейсами, которые могут объединятся, образуя программы по принципу блоков.

Основным потенциальным недостатком микроядер является их низкая производительность. Создание сообщения и отправка его через микроядро с последующим получением и декодированием ответа занимает больше времени, чем непосредственно вызов сервиса.

Многое в обеспечении производительности зависит от функциональных возможностей ядра. Избирательное увеличение функциональности ядра приводит к снижению количества переключений между пользовательским режимом и режимом ядра.

Есть мнения и взгляды на то, чтобы сделать микроядро не больше, а наоборот – меньше. При этом повышается его гибкость и надёжность.

Функции микроядра.

1. ***Низкоуровневое управление памятью.*** Для реализации в микроядре защиты на уровне процессов в нем должен обеспечиваться контроль над аппаратной организацией адресного пространства. Если микроядро будет отвечать лишь за отображение виртуальной страницы на физическую страницу, то блок управления памятью, включая систему защиты адресного пространства одного процесса от другого, а также алгоритм замены страниц и другие логические схемы страничной организации памяти можно реализовать вне ядра. Модуль

виртуальной памяти принимает решение, когда загружать страницу в память, и какую из страниц, находящихся в памяти, следует заменить.

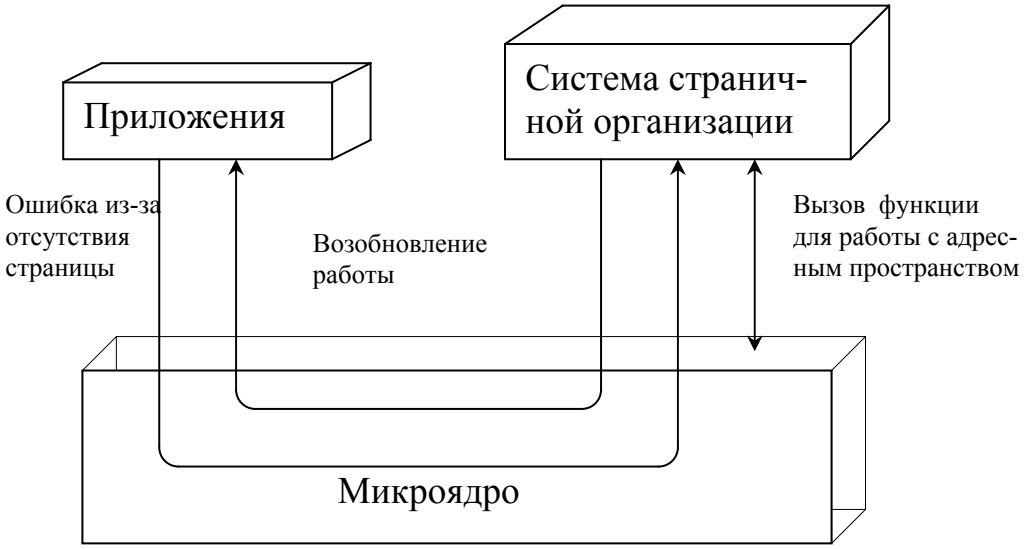


Рис. 1.4. Низкоуровневое управление памятью

Когда приложение обращается к странице, которая отсутствует в основной памяти, возникает прерывание из-за отсутствия страницы и управление перехватывается ядром. Ядро отправляет системе страничной организации памяти сообщение, в котором указывается запрашиваемая страница. Система страничной организации памяти может принять решение о загрузке данной страницы в оперативную память и выделения для этого физической страницы. Система страничной организации и ядро взаимодействуют между собой, чтобы логические операции, выполняемые системой страничной организации памяти, отображались в физическую память. Как только нужная страница станет доступна, то система страничной организации отправляет через микроядро сообщение приложению о том, что приложение может продолжить работу.

Считается, что в микроядре можно оставить только 3 операции для поддержки внешних систем страничной организации памяти и управления внешней памятью.

Предоставление. Процесс, который владеет адресным пространством, может предоставлять некоторые свои страницы другому процессу. Ядро удаляет эти страницы из адресного пространства первого процесса и передаёт их второму процессу.

Отображение. Процесс может отображать любые свои страницы в адресное пространство другого процесса. После чего оба процесса будут иметь доступ к этим страницам, то есть создаётся общая область памяти. Ядро не меняет информации о принадлежности страниц первому процессу, но выполняет отображение, предоставляя другому процессу доступ к этим страницам.

Восстановление. Процесс может восстановить любые страницы, предоставленные другим процессам или отображенные в их адресное пространство.

Вначале ядро определяет всю физическую память как единое адресное пространство, которым управляет основной системный процесс. При создании новых процессов страницы общего адресного пространства могут передаваться или отображаться в эти новые процессы. Такая схема позволяет одновременно поддерживать несколько схем организации виртуальной памяти.

2. ***Взаимодействие между процессами.*** Основной формой взаимодействия между процессами (потоками) являются сообщения. Сообщения включают в себя заголовок, в котором указаны идентификаторы процесса отправителя и процесса получателя, а также указатель на блок данных и некоторые управляющие сведения о процессе.

Взаимодействие между процессами основывается на относящихся к этим процессам *портам*. *Порт* – это очередь сообщений, предназначенная для определённого процесса. С портом связан список возможностей, в которых указано с какими процессами данный процесс может обмениваться информацией. Процесс может разрешить доступ к себе, отправив в ядро сообщение, в котором указана новая возможность порта.

Если адресное пространство в процессах не перекрывается, то передача сообщения от одного процесса другому – это копирование одной области памяти в другую.

3. ***Управление вводом-выводом и прерываниями.*** В микроядре имеется возможность обрабатывать аппаратные прерывания подобно сообщениям и включать в адресное пространство порта ввода-вывода, но не обрабатывает их. Микроядро распознает прерывания, но само не обрабатывает их. Оно генерирует сообщение процессу, работающему на пользовательском уровне и связанному с данным прерыванием, т. е. ядро преобразует прерывание в сообщение, но само в обработке аппаратно-зависимых прерываний не участвует.

В некоторых системах предлагается рассматривать аппаратное обеспечение как набор потоков, которые обладают своими идентификаторами и отправляют сообщение состоящее из идентификатора данного потока соответствующим потокам в пользовательских программах. Поток-получатель выясняет, является ли полученное сообщение прерыванием.

1.1.11 Принципы построения ОС

1. Частотный принцип.

Этот принцип основан на выделении в алгоритмах программ и в обрабатываемых массивах действий и данных по частоте их использования. Действия и данные, используемые часто, располагаются в оперативной памяти, т.к. к ним необходим быстрый доступ. К тому же стремятся наиболее часто выполняемые операции оптимизировать по времени выполнения и по занимаемой памяти.

Следствия от применения частотного принципа – это применение многоуровневого планирования при организации работы ОС. На уровень долгосрочного планирования выносятся редкие и длинные операции управления деятельностью системы. Краткосрочному планированию подвергаются часто используемые и короткие операции.

2. Принцип модульности.

Это принцип в равной степени отражает технологические и эксплуатационные свойства. Наибольший эффект от его использования достигается в том случае, когда принцип одновременно распространен на ОС, аппаратуру и прикладные программы.

Под модулем в общем случае понимают функциональный элемент рассматриваемой системы, имеющей законченное оформление и выполненный в пределах требования ОС, а также средства сопряжения с подобными элементами или элементами более высокого уровня данной или другой системы.

Модуль предполагает лёгкий способ его замены на другой модуль, при наличии заданных интерфейсов. Чаще всего разделение на модули происходит по функциональному признаку.

Модули могут быть восстанавливаемыми и невосстанавливаемыми. Если модуль после окончания работы не восстанавливается в исходное состояние, то он называется однократным. Если модуль в процессе работы искажает своё состояние, но перед окончанием работы восстанавливается в исходное состояние, то его называют многократным. Особое значение при построении ОС имеют модули, называемые параллельно используемыми или реентерабельными. Каждый такой модуль может использоваться одновременно несколькими программами. Это позволяет хранить в памяти только одну копию такого модуля.

3. Принцип функциональной избирательности.

В ОС выделяется некоторая часть особо важных модулей, которые должны быть в оперативной памяти постоянно для эффективной организации вычислительного процесса. Эту часть обычно называют ядром ОС.

При формировании ядра необходимо удовлетворить двум противоречивым требованиям:

– в состав ядра должны войти наиболее часто используемые модули;

– количество модулей должно быть таким, чтобы не занимать много оперативной памяти;

В состав ядра входят модули по управлению системой прерываний, средства перевода процессов с одного состояния в другое, средства распределения оперативной памяти. Программы, входящие в состав ядра, обычно загружаются в оперативную память и называются резидентными.

Помимо резидентных программ существуют транзитные модули или программы, которые загружаются только при необходимости и могут перекрывать в оперативной памяти друг друга.

4. Принцип генерируемости.

Определяет такой способ исходного представления ОС, который позволял бы настраивать эту системную программу, исходя из конкретной конфигурации используемой машины и круга решаемых задач.

Процедура генерации проводится достаточно редко, а процесс генерации осуществляется с помощью специальной программы-генератора и входного языка для неё, позволяющего описывать программные возможности системы и конфигурацию машины. В результате генерации получается полная версия ОС. Исходный набор программ ОС, из которого производится генерация называется дистрибутивом.

5. Принцип функциональной избыточности.

Предполагает возможность проведения одной и той же работы различными средствами, т.е. ОС допускает альтернативное выполнение одних и тех же заданий в различных режимах своего функционирования.

6. Принцип по умолчанию.

Принцип основан на хранении в системе некоторых базовых описании, структур процесса, модулей, конфигурации оборудования и данных, определяющих прогнозируемые объемы требуемой памяти, времени счета программы, потребности во внешних устройствах, которые характеризуют пользовательские программы и условия их выполнения.

Эту информацию пользовательская система использует в качестве заданной, если она не будет определена или сознательно конкретизирована. В целом применение этого принципа позволяет сократить число параметров, устанавливаемых пользователем, когда он работает с системой.

7. Принцип перемещаемости.

Предусматривает построение модулей таким образом, что их исполнение не зависит от места расположения в оперативной памяти. Настройка текста модуля в соответствии с его расположением в памяти осуществляется специальными механизмами либо непосредственно пред выполнением программы, либо по мере по мере ее выполнения.

Настройка заключается в определении фактических адресов, используемых в адресных частях команды, и определяется применяемым в конкретной машине способом адресации и алгоритмом распределения оперативной памяти, принятым для данной ОС.

Этот принцип целесообразно распространять и на пользовательские программы.

8. Принцип защиты.

Определяет необходимость разработки мер, ограждающих программы и данные пользователя от искажения и нежелательного влияния друг на друга, а также пользователя на ОС и наоборот. Особенно трудно обеспечить защиту, когда используется разделение ресурсов.

Программы должны быть гарантированно защищены как при выполнении, так и при хранении, хотя попыток испортить и нанести нежелательный эффект пользовательским программам совершается множество.

Реализуется несколько подходов для обеспечения защиты.

Одним из направлений является реализация двухконтекстности работы процессора: в каждый момент времени процессор может выполнять программу из состава ОС либо прикладную или служебную программу, не входящую в состав ОС.

Второе направление состоит в том, чтобы гарантировать невозможность непосредственного доступа к любому разделяемому ресурсу со стороны пользовательских и служебных программ, для чего в состав машинных команд вводятся специальные привилегированные команды, управляющие распределением и использованием ресурсов. Такие команды разрешается выполнять только ОС. Контроль за выполнением привилегированных команд производится аппаратно.

Для реализации принципов защиты может использовать контекстный механизм защиты данных и текста программ, находящийся в операционной памяти. Для программ пользователей выделяются определенные участки памяти и выход за пределы этих участков приводит к возникновению прерываний по защите. Механизм контроля реализуется аппаратным способом, путем применения ограничительных регистров или ключей памяти.

Третье направление реализует механизм защиты данных, хранящихся в памяти, используются подходы, основанные на разграничении прав доступа, введении паролей, контроле за правильной интерпретацией данных, записанных в файле.

9. Принцип независимости программ от внешних устройств.

Позволяет выполнять операции управления внешними устройствами независимо от их конкретных физических характеристик. Связь программ с конкретными устройствами производится не на уровне трансляции программ, а в период планирования ее выполнения. При подключении новых устройств или замене существующих, текст программ не изменяется, а осуществляется подключение нового устройства к ОС путём подключения специальной программы, обеспечивающей взаимодействие ОС с внешним устройством. Такие программы называются драйверами.

10. Принцип открытой и наращиваемой операционной системы.

Открытая ОС доступна пользователю и специалисту, обслуживающему машину, а возможность наращивания или модификации операционной системы позволяет использовать не только возможности генерации, но вводить в операционную систему новые модули или модифицировать существующие.

ТЕМА 2

2.1 Основы операционных систем

2.1.1 Понятие процесса

В предметной области системного программирования понятие процесса является базовым, но вместе с тем недостаточно формально определено. Существуют определения процесса как формального, так и неформального свойства. Понятие процесса является определенным видом абстракции, которую поразному используют и трактуют разные группы лиц.

Многие архитектуры современных вычислительных машин являются многопроцессорными. Процессор - устройство в составе ЭВМ, способное автоматически выполнять допустимые для него действия в некотором определенном порядке, т.е. по определенной программе, хранящейся в памяти, и непосредственно доступной устройству. Между процессорами в системе существуют информационные и управляющие связи.

Каждый процессор - объект в системе, которым хотели бы воспользоваться одновременно несколько пользователей для исполнения своей программы. В отношении каждого пользователя, претендующего на исполнение программы на некотором процессоре, и системы, распределяющей этот процессор среди многих пользователей, вводится понятие “процесс”.

В общем случае процесс - некоторая деятельность, связанная с исполнением программы на процессоре.

Согласно стандартизованного определения процесс - система действий, реализующая определенную функцию в вычислительной системе и оформленная так, что управляющая программа вычислительной системы может перераспределять ресурсы этой системы в целях обеспечения мультипрограммирования.

При выполнении программы могут потребоваться результаты других процессов или процессоров, или другие ресурсы. Следовательно, ходом развития процесса нужно управлять.

Управление процессами, как и в отношении каждого, так и в отношении их совокупности - функция ОС.

При выполнении программ на центральном процессоре чаще всего различают следующие характерные состояния.

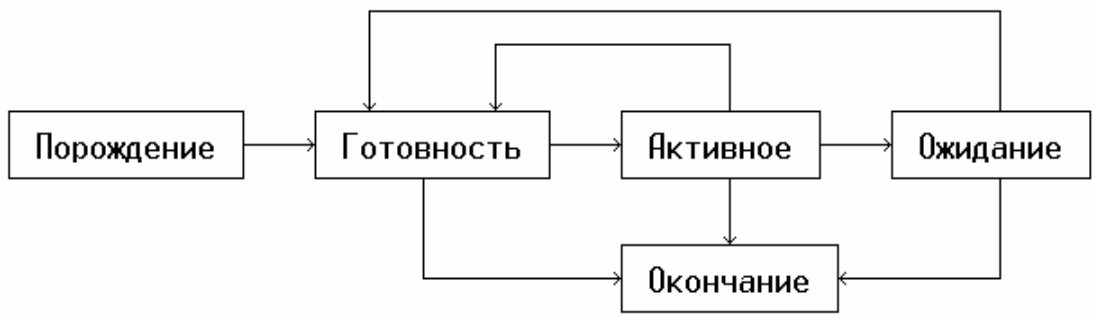


Рис. 2.1. Состояния процесса и переходы

Порождение - подготовка условий для первого исполнения на процессоре.

Активное состояние, или состояние "счет" - программа выполняется на процессоре.

Ожидание - программа не исполняется на процессоре по причине занятости какого-либо требуемого ресурса.

Готовность - программа не исполняется, но для использования предоставлены все необходимые в текущей момент ресурсы, кроме ЦП.

Окончание - нормальное или аварийное окончание исполнения программы, после которого процессор и другие ресурсы ей не предоставляются.

Процесс находится в каждом или некоторых из своих допустимых состояний в течение некоторого времени, после чего переходит в другое допустимое состояние.

Процессы определяются рядом временных характеристик. В некоторый момент времени процесс может быть порожден, а через некоторое время закончен. Интервал между этими моментами называют интервалом существования процесса. Длительность интервала существования процесса, в общем случае, непредсказуема.

Отдельные виды процессов требуют такого планирования, чтобы гарантировать окончание процесса до наступления некоторого конкретного времени. Это процессы реального времени.

В другой класс входят процессы, время существования которых должно быть не более интервала времени допустимой реакции ЭВМ на запросы пользователя. Это интерактивные процессы. Интервал времени является допустимым, если он не раздражает пользователя.

Остальные процессы относятся к классу пакетных.

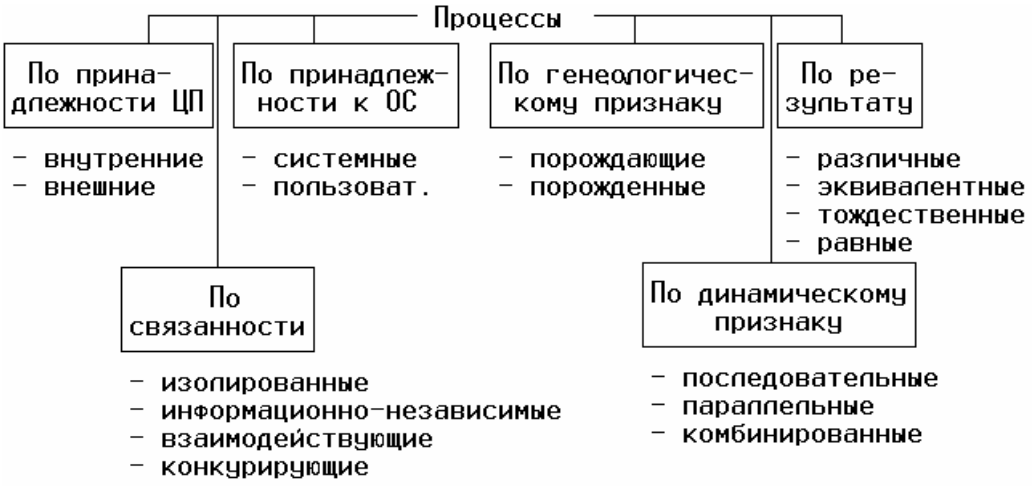


Рис. 2.2. Классификация процессов

Сравнение процессов может быть произведено с использованием понятия "трасса" - порядок и длительность пребывания процесса в допустимых состояниях на интервале существования.

Два процесса, имеющие одинаковый конечный результат обработки одних и тех же исходных данных по одной той же или даже различным программам, называют эквивалентными. Трассы эквивалентных процессов в общем случае не совпадают. Если в каждом из эквивалентных процессов обработка данных происходит по одной и той же программе, но трассы при этом в общем случае не совпадают, то такие процессы называют тождественными. При совпадении трасс у тождественных процессов их называют равными. Во всех остальных случаях процессы всегда различны.

Проблематичность управления процессами заключается в том, что в момент порождения процессов их трассы неизвестны. Также требуется учитывать, как соотносятся во времени интервалы существования процессов.

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

Процессы называются взаимосвязанными, если между ними с помощью системы управления процессами поддерживается какого-либо рода связь: функциональная, пространственно-временная, управляющая, информационная и т.д. В противном случае процессы являются изолированными. При наличии между процессами управляющей связи устанавливается соотношение вида “порождающий-порождённый”.

Если взаимосвязанные процессы при развитии используют совместно некоторые ресурсы, но не связаны между собой информационно, то такие процессы называют информационно-независимыми, а при наличии информационной связи процессы называются взаимодействующими. Схемы механизмов установки таких связей различны, они обусловлены динамикой процессов и выбранным способом связи: явным (явный обмен сообщениями между процессами), или неявным (с помощью разделяемых структур данных). Когда необходимо подчеркнуть связь между взаимосвязанными процессами по ресурсам, их называют конкурирующими.

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

Отношение предшествования для двух процессов означает, что первый процесс должен переходить в активное состояние раньше второго.

Отношение приоритетности означает, что некоторый процесс с определённым приоритетом может быть переведён в активное состояние, если в состоянии готовности нет процессов с большим приоритетом, а процессор либо свободен, либо используется программой с меньшим приоритетом.

Отношение взаимного исключения. Процессы используют общий ресурс. Совокупность действий над этим ресурсом одного процесса называют критической областью. Критическая область одного процесса не должна выполняться одновременно с критической областью другого процесса для этого же ресурса.

Проблема в реализации синхронизирующих правил в составе системы управления процессами обусловлена динамикой процессов, неопределенностью и непредсказуемостью порядка и частотой перехода процессов из состояния в состояние по мере их развития.

В отношении каждой совокупности взаимодействующих процессов приходится решать задачу синхронизации, которая требует определённого порядка выполнения процессов с целью установления требуемого взаимодействия.

2.1.2 Понятие ресурса

Ресурс - всякий потребляемый объект (независимо от формы его существования), обладающий некоторой практической ценностью для потребителя.

Ресурсы различаются по запасу выделяемых единиц ресурса и бывают в этом смысле исчерпаемые и неисчерпаемые. Исчерпаемость ресурса, как правило, приводит к конфликтам в среде потребителей. Для регулирования конфликтов ресурсы должны распределяться между потребителями по каким-то правилам, в наибольшей степени их удовлетворяющим.

Вычислительную систему или ЭВМ можно представить как ограниченную последовательность функциональных элементов, обладающих потенциальными

возможностями выполнения с их помощью или над ними действий, связанных с обработкой, хранением или передачей данных.

Такие элементы потребляются другими элементами, являющимися в общем случае пользователями.

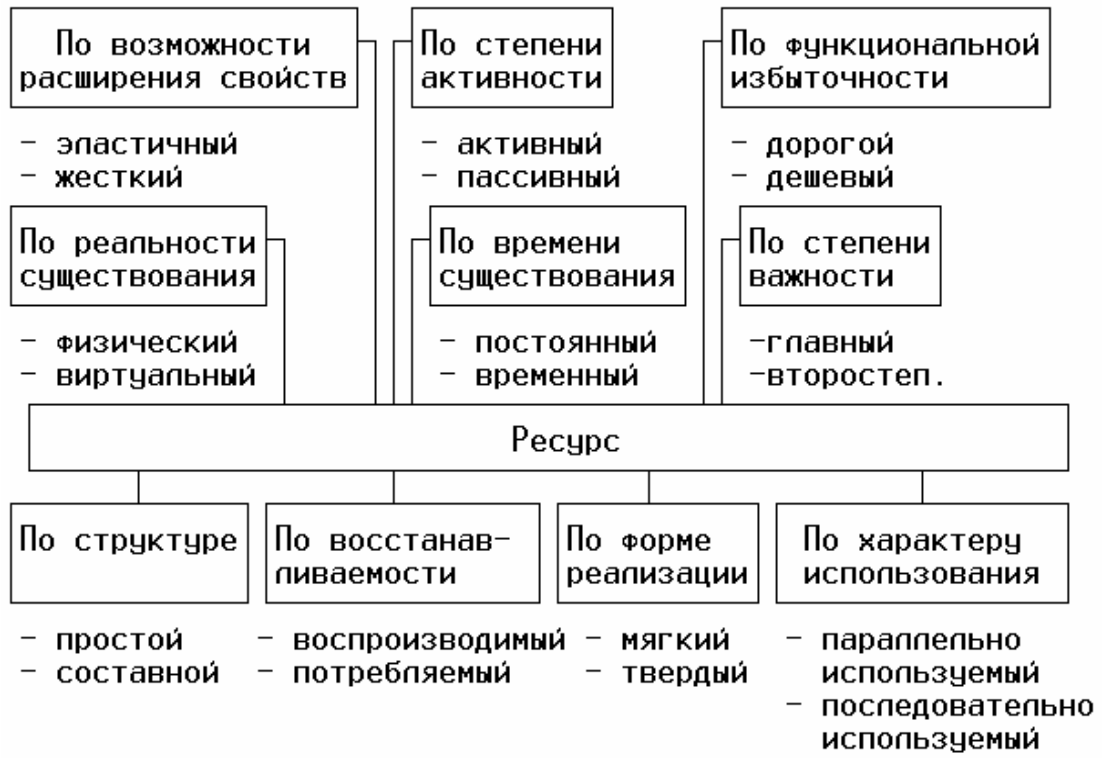


Рис. 2.3. Классификация ресурсов

Уровень детализации элементов, выделяемых по запросам для использования в системе, может быть различным. Считается, что потребителем выделяемых элементов вычислительной системы будут являться процессы, развивающиеся в ней. Все выделяемые по запросам от процессов элементы системы отождествляются с понятием ресурсов.

В соответствие с ГОСТ ресурсом является средство вычислительной системы, которое может быть выделено процессу на определенный интервал времени.

Под физическим понимают ресурс, который реально существует и при распределении его между пользователями обладает всеми присущими ему физическими характеристиками.

Виртуальный ресурс - это некоторая модель физического ресурса. Виртуальный ресурс не существует в том виде, в котором он проявляет себя пользователю. Как модель виртуальный ресурс реализуется в некоторой программно-аппаратной форме. В этом смысле виртуальный ресурс существует. Однако виртуальный ресурс может предоставить пользователю не только часть тех свойств, которые присущи объекту моделирования, т.е. физическому ресурсу, но и свойства, которые ему не присущи. Однако пользователь будет воспринимать их, как реально существующие.

Признак возможности расширения свойств характеризует ресурс с точки зрения возможности построения на его основе некоторого виртуального ресурса. Физический ресурс, который допускает виртуализацию, называется эластичным. Жестким называется физический ресурс, который по своим внутренним свойствам не допускает виртуализации.

При использовании активного ресурса он способен выполнять действия по отношению к другим ресурсам (или даже в отношении самого себя) или процессам, которые в общем случае приводят к изменению ресурса. (Например, ЦП - активный ресурс).

Пассивный ресурс не обладает таким свойством. (Например, память).

Разделение ресурсов на дорогие и дешевые связано с реализацией принципа функциональной избыточности при распределении ресурсов. Перед пользователем стоит задача выбора - получить быстро требуемый ресурс и дорого заплатить за такую услугу, либо подождать выделения требуемого ресурса и после его использования заплатить меньше. При наличии в системе альтернативных ресурсов вводятся и различные цены за их использование.

В отношении каждого ресурса процесс-пользователь выполняет три типа действий: запрос, использование, освобождение.

Если при распределении системой ресурса допускается многократное выполнение действий в последовательности запрос-использование-освобождение, то такой ресурс называют воспроизводимым. Если считать, что при этом ресурс не утрачивает своих свойств, то время его существования равно бесконечности.

В отношении некоторых категорий ресурсов порядок действий таков: освобождение-запрос-использование, после чего ресурс, который в данном случае называют потребляемым, изымается из сферы потребления.

Правила распределения ресурсов обуславливаются параллельной или последовательной схемой использования распределяемого между несколькими процессами ресурса. Последовательная схема предполагает, что в отношение некоторого ресурса, который называют последовательны, допустима строго последовательная во времени последовательность действий: запрос, исполнение, освобождение – каждым процессом-потребителем этого ресурса. Для параллельных процессов такие цепочки действий называются критическими областями и при выполнении должны удовлетворять правилу взаимного исключения.

Последовательно выполняемый ресурс, разделяемый несколькими параллельными процессами, поэтому называется критическим ресурсом. Параллельная же схема предполагает одновременное использование одного и того же ресурса несколькими процессами. Но такое использование не должно вносить каких либо изменений в логику развития каждого из процессов.

По форме реализации различают твердые и мягкие ресурсы. Под твердыми понимаются аппаратные компоненты машины, а также человеческие ресурсы, остальные ресурсы относятся к мягким. Различным по отношению к твердым и мягким ресурсам является подверженность к сбойным или отказным ситуациям, а также последующее восстановление работоспособности.

В классе мягких ресурсов выделено два типа:

- программные;
- информационные.

Если мягкий ресурс допускает копирование и эффект от использования ресурса-оригинала и ресурса-копии идентичен, то такой ресурс называется программным, в обратном случае – информационным. Мягкие информационные ресурсы либо принципиально не допускают копирования, либо допускают копирование, но оно является функцией времени (сообщения, сигналы, запросы на прерывание и т.д.).

2.1.3 Концепция виртуализации

Концепция виртуализации рассматривается в ОС как средство уменьшения конфликтов при взаимодействии процессов и распределении ресурсов, облегчая работу пользователя с вычислительной системой при его обращении к тем или иным ресурсам.

Виртуализация достигается на основе централизованной схемы распределения ресурсов. При такой организации работы обеспечиваются *две формы виртуализации*:

1. Пользователь обеспечивается при обслуживании своего процесса ресурсом, который реально не существует, или существует, но с ухудшенными характеристиками.

2. Для нескольких параллельных процессов создается иллюзия одновременного использования того, что одновременно в реальной системе существовать не может.

Целью виртуализации является предоставление пользователям ресурсов с характеристиками, в наибольшей степени их удовлетворяющими и позволяющими снять ограничения на количество распределяемых ресурсов, что позволяет увеличить скорость развития процесса, организовать более гибкое управление процессами и снять ограничение на их количество.

Виртуализация обеспечивается аппартно-программными средствами.

Примеры виртуализации и виртуальных ресурсов:

1. Построение на основе одного интервального таймера произвольного числа таймеров, которые предоставляются пользователям в монотонное использование.

2. Применение мультиплексных каналов при работе с внешними устройствами. Имеется одно устройство, организующее связь пользовательских программ с внешними устройствами. Для каждого устройства каналом выделяется квант времени. Величина кванта времени для различных устройств может быть разной. Устройствам при таком режиме создаётся видимость, что они непрерывно обмениваются информацией с пользовательскими режимами.

3. Организация на базе одного из устройств, имеющего определенный способ доступа, имитацию работы других устройств с другими способами доступа.

4. Использование так называемых нулевых устройств. Такие устройства реально не существуют, но создаётся видимость их существования. При пере-

даче данных на такое устройство они поглощаются, а при чтении данных с такого устройства сразу выдается признак конца файла.

5. Понятие спулинга при работе с внешними устройствами. При этом каждое внешнее устройство моделируется некоторой областью внешней памяти. Каждая такая область представляет собой буфер между процессом и реальным устройством. Если этот буфер выполняет роль выходного, то в нем накапливаются данные по мере выполнения процессором команд вывода. При этом не производится вывода, но у процесса появляется ощущение, что он что-то вывел. Реальный вывод производится под контролем системы. Входной буфер работает таким же образом. В нем накапливаются данные, вводимые от внешних устройств, а затем эти данные передаются процессу. Типичный пример устройства, в котором применяется спулинг – это устройство печати.

Рассмотренные примеры иллюстрируют возможность построения и использования виртуальных ресурсов. Каждый виртуальный ресурс подобного рода генерируется некоторым процессом. Управление этим процессом осуществляется со стороны ОС, которая в этом смысле выполняет роль распределителя реального ресурса, на базе которого строится один или несколько виртуальных. Если поддерживается несколько виртуальных ресурсов, то в составе распределения реализуют дисциплины распределения таких ресурсов среди процессов-пользователей.

Одним из наиболее типичных или законченных примеров концепции виртуализации является понятие *виртуальной машины*. Любая ОС, являясь средством распределения ресурсов, организует по определённым правилам управление процессами на базе скрытой аппаратной части, создавая у пользователя видимость виртуальной машины.

Типичный пользователь видит и использует виртуальную машину как некоторое устройство, способное воспринимать его программы, написанные на определённом языке программирования, выполнять их и выдавать результаты. При таком подходе пользователя не интересует структура машины и способы эффективного использования её составных частей. Пользователь работает с машиной в терминах, применяемых в языке программирования.

Зачастую виртуальная машина, предоставленная пользователю, воспроизводит реальную архитектуру машины, но элементы архитектуры в таком представлении выступают с новыми или улучшенными характеристиками.

Идеальная в представлении пользователя архитектура виртуальной вычислительной машины имеет следующий состав:

– бесконечная по объему память с произвольно выбираемым наиболее удобным для пользователя доступом к объектам, хранимым в памяти

– один или несколько процессоров, способных выполнять действия, выражаемые пользователем в терминах некоторых удобных для него языков программирования

– произвольное количество внешних устройств с удобным способом доступа и предоставление информации, передаваемой через эти устройства или

хранимой ими без каких-либо существенных ограничений на объем информации

Концепция виртуализации нашла широкое применение при проектировании и реализации ОС. Наиболее рационально представить структуру системы в виде определенного набора планировщиков процессов и распределителей ресурсов, которые часто называются ***мониторами.***

Монитор – распределитель некоторого ресурса, который может на основании некоторой организации работы обеспечить ту или иную степень виртуализации при распределении эластичного ресурса.

Использование концепции виртуализации положено в основу восходящего метода проектирования и разработки ОС. ОС строится как иерархия вложенных друг в друга виртуальных машин. Низшим уровнем иерархии являются аппаратные средства машины. Следующий уровень – программный. Он совместно с нижним уровнем обеспечивает достижение машиной новых свойств (виртуальная машина I уровня). Относительно этой виртуальной машины разрабатывается новый программный слой, и получается виртуальная машина II уровня. Последовательно наращивая уровни, можно реализовать виртуальную машину с требуемыми свойствами. При этом каждый новый слой позволяет расширить функциональные возможности по обработке данных и позволяет организовать достаточно простой доступ к нижним уровням. Применение метода иерархического построения виртуальных машин позволяет систематизировать проект, повысить надежность сложной программной системы, уменьшить сроки разработки. Однако при этом имеется ряд проблем, основная из них: определение свойств и количества уровней виртуализации и определение правил внесения на каждый уровень необходимых частей ОС.

Имеются ряд правил по формированию уровней и способов их взаимодействия.

1. На каждом уровне ничего не известно о свойствах и о существовании более высоких уровней.

2. На каждом уровне ничего не известно о внутреннем строении других уровней. Связь между ними осуществляется только через жесткие, заранее определенные сопряжения.

3. Каждый уровень представляет собой группу модулей, некоторые из которых являются внутренними и не доступны для других уровней. Имена остальных модулей известны на следующем, более высоком уровне, и представляют собой сопряжение с этим уровнем.

4. Каждый уровень располагает определенными ресурсами и либо скрывает их от других уровней, либо представляет другим уровням их абстракции (виртуальные ресурсы).

5. Каждый уровень может обеспечивать некоторую абстракцию данных в системе.

6. Предположения, которые делаются на каждом уровне относительно других уровней, должны быть минимальными.

7. Связь между уровнями ограничена явными аргументами, передаваемыми с одного уровня на другой.

8. Недопустимо совместное использование несколькими уровнями глобальных данных.

9. Каждый уровень должен иметь высокую прочность и слабое сцепление с другими уровнями.

10. Всякая функция, выполняемая уровнем абстракции, должна иметь единственный вход.

2.1.5 Дисциплины распределения ресурсов

Идея мультипрограммирования связана с наличием очередей процессов. Процессор в многозадачных системах по очереди предоставляется процессам.

Использование многими процессами одного ресурса, который в данный момент времени может обслуживать только один процесс, осуществляется с помощью дисциплин распределения ресурсов. Их основой является:

1. *Дисциплины формирования очередей на ресурс* — это совокупность правил, определяющих размещение процессов в очереди.

2. *Дисциплина обслуживания очереди* — совокупность правил извлечения одного из процессов из очереди с последующим предоставлением ему ресурса для использования.

Основным конструктивным и согласующим элементом при реализации дисциплин диспетчеризации является очередь.

Дисциплины формирования очередей разделяют на два класса:

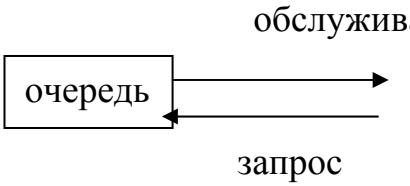
– *статический*: приоритеты назначаются до выполнения задания

– *динамический:* приоритеты назначаются в процессе выполнения задания

На практике чаще всего встречаются следующие *дисциплины распределения или обслуживания ресурсов:*

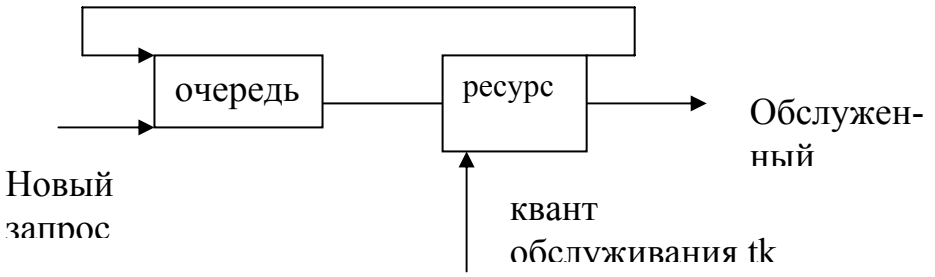
1. Обслуживание в порядке поступления. Все заявки поступают в конец очереди. Первыми обслуживаются заявки из начала очереди.

2. Обслуживание в порядке, обратном порядку поступления. Данная дисциплина является основой построения стековой памяти.



Общим для 1 и 2 является относительная простота и определённая справедливость в обслуживании всех потоков, поступающих в систему. Среднее время ожидания запросов в очереди при некотором установившемся темпе обслуживания и поступления является одинаковым, независимо от характеристик процессов пользователей. Помимо этого, FIFO обеспечивает минимизацию дисперсии времени ожидания.

3. Круговой циклической алгоритм.



4. В основу этой дисциплины обслуживания положена дисциплина FIFO, но время обслуживания каждого процесса здесь ограничено и определяется квантом времени обслуживания. Если запрос был обслужен полностью в течение этого кванта, он покидает очередь, иначе он помещается в конец очереди. А если не успел, то поступает опять в конец очереди. Эта дисциплина широко применяется в системах распределения времени. Хотя в этой дисциплине нет явных приоритетов, автоматически получают преимущество процессы, не требующие долгого обслуживания. Короткие процессы имеют меньшее среднее время пребывания в очереди по сравнению с длинными. Чем меньше квант времени, тем выше благоприятствование коротким процессам, но уменьшение кванта времени ведёт к увеличению затрат, связанных с переключением с процесса на процесс, что особенно неблагоприятно для длинных процессов. Это одноочередная дисциплина.

В ОС также широко используются многоочередные дисциплины.

2.1.6 Концепция прерывания

Реализация многопрограммного режима работы ЭВМ основана на использовании прерываний. Так, программа, обслуживаемая процессором, прерывается из-за отсутствия данных, подлежащих обработке в оперативной памяти. Программа, обслуживаемая процессором, может быть прервана более приоритетной программой. Из всего многообразия различных причин прерывания необходимо выделить два вида системных причин прерывания: первого и второго рода.

Системные причины прерывания первого рода возникают в том случае, когда у процесса, находящегося в активном состоянии, возникает потребность получить некоторый ресурс или отказаться от него либо выполнить над ресурсом

какие-либо действия. Эти причины появляются и тогда, когда процесс порождает, уничтожает и выполняет какие-либо действия в отношении других процессов. При таких прерываниях возникает необходимость в явной форме выражать требование на прерывание процессом самого себя. Необходимо установление связи типа «вызывающий — вызываемый». Установка такой связи реализуется, как правило, в форме макрокоманд, представленных в пользовательской программе. При выполнении таких макрокоманд происходит переключение процессора с обработки программы пакета на работу операционной системы, которая подготавливает и обеспечивает выполнение соответствующего прерывания. К этой группе относятся и так называемые внутренние прерывания, которые связаны с работой процессора и являются синхронными с его операциями. К таким прерываниям относится арифметическое переполнение, исчезновение порядка в операциях с плавающей запятой, обращение к заишщеннрму массиву оперативной памяти и др.

Системная причина прерывания второго рода обусловлена необходимостью проведения синхронизации между параллельными процессами. Процессы, порожденные и подчиненные операционной системе, по мере их окончания или при какой-то другой ситуации вырабатывают сигнал прерывания. Например, по мере окончания внешнего процесса пересылки массива данных с МД в оперативную память вырабатывается сигнал прерывания. Этот сигнал прерывает исполнение обслуживаемой программы процессором, которое происходит «без ее ведома», т. е. асинхронно.

При обработке каждого прерывания необходимо выполнять такую последовательность действий:

1) восприятие запроса на прерывание;

2) запоминание состояния прерванного процесса, определяемое прежде всего значением счетчика команд. Оно должно отражать и признак команды, после выполнения которой произошло прерывание, содержимое регистров общего назначения, режим работы процессора (с позиции обслуживания процессором пользовательской или системной программы в момент прерывания);

3) передача управления прерывающей программе, для чего счетчик команд СК должен быть занесен адрес, который, как правило, является уникальным для каждого типа прерывания;

4) обработка прерывания;

5) восстановление нормальной работы.

В большинстве ЭВМ этапы 1—3 реализуются аппаратными средствами, а этапы 4 и 5 — операционной системой, ее блоком программ обработки прерываний.

Наряду с описанными функциями, реализуемыми на аппаратном уровне, обработка, прерываний реализуется в ОС на уровне системных программ. Такие функции представлены в прерывающих программах. Их количество определяется количеством уровней прерывания.

Например, обработчик внешних прерываний, обработчик программных прерываний. Прерывающая программа, обслуживающая запросы по уровню

ввода-вывода называется супервизором ввода-вывода. Прерывающие программы по уровню ОС называются программами супервизора.

Каждая прерывающая программа соответствует определенному уровню прерываний. Последний объединяет ряд запросов прерываний, обслуживаемых одной прерывающей программой. При таком решении кроме выбора прерывающей программы необходимо выбирать в уровне наиболее приоритетные запросы, требующие обслуживания. Эта функция часто реализуется в системных прерывающих программах.

После окончания обработки прерываний происходит восстановление нормальной работы ЭВМ. Это восстановление происходит передачей управления системной программе, выполняющей функции диспетчера. Последняя получает управление в тех случаях, когда результатом обработки прерывания является вывод некоторой задачи из состояния ожидания в состояние готовности или постановкой текущей задачи в состояние ожидания. Программа «Диспетчер» анализирует состояние задач в системе и передает управление готовой к выполнению задаче с наивысшим приоритетом. Если задачи в состоянии готовности отсутствуют, то центральный процессор переводится в состояние ожидания до возникновения очередного прерывания.

ТЕМА 3

3.1 Процессы

3.1.1 Состояние процессов

Процесс – это система действий, реализующая определенную функцию в вычислительной системе и оформленная так, что управляющая программа вычислительной системы может перераспределять ресурсы этой системы в целях обеспечения мультипрограммирования.

Для процесса при выполнении на центральном процессоре можно выделить следующие характерные отдельные состояния.

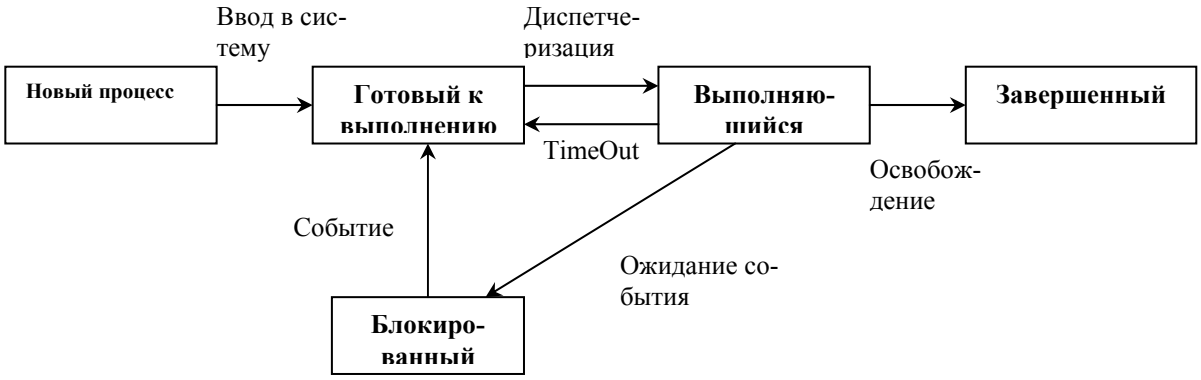


Рис. 3.1. Схема переходов процесса из состояния в состояние

Новый. Только что созданный процесс, который еще не помещен ОС во множество процессов.

Выполняющийся. В данный момент работающий процесс.

Готовый к выполнению. Процесс будет запущен, как только представится возможность.

Блокированный. Процесс, который ожидает некоторого события.

Завершающийся. Процесс, удаленный из множества запущенных процессов.

Запуск процесса состоит из 2-х этапов.

1. Присвоение процессу идентификатора и формирование всех таблиц, необходимых процессу.

2. Распределение памяти под процесс.

Рассмотрим систему переходов из состояние в состояние.

Нулевое состояние- «состояние новый». Этот переход происходит при наступлении одного из следующих событий.

1. Если ОС работает в пакетном режиме и для обработки поступает управляющий поток пакетных заданий.

2. Если система работает в интерактивном режиме и в систему с терминала входит новый пользователь.

3. Создание ОС процесса, необходимого для выполнения служебных функций.

4. Порождение одного процесса другим.

Переход из «новый» в «готовый». ОС осуществляет переход, когда будет готова к обработке дополнительных процессов. В большинстве систем сущест-

вует ограничения на количество выполняющихся процессов или на объем виртуальной памяти.

Переход из «готовый» в «выполняющийся». Происходит, когда ОС выбирает новый процесс для запуска. Выбирается один из процессов, находящихся в состоянии «готовый», в соответствии с какой-то дисциплиной в обслуживании.

Переход из «выполняющийся» в «готовый». Чаще всего происходит, когда процесс отрабатывает максимальный промежуток времени, отведенный для непрерывной работы одного процесса.

Переход из «выполняющийся» в «блокированный». Процесс переводится в заблокированное состояние, если для продолжения его работы требуется какое-либо событие. Например, процесс может запросить какой-либо ресурс, который временно недоступен или требуется выполнить какое-либо действие, необходимое для продолжения работы процесса, например, операцию ввода-вывода.

Переход «заблокированный»-«готовый». Осуществляется, когда происходит ожидаемое событие.

Переход из «выполняющийся» в «завершающийся». Выполняется тогда, когда процесс сигнализирует об окончании своей работы, или ОС прекращает его выполнение в силу каких-то причин.

Основные причины создания процессов следующие.

1. Новое пакетное задание. Готовясь принять на обработку новое задание. ОС считывает очередные команды управления заданием.

2. Вход в систему в интерактивном режиме, когда в систему с терминала подключается новый пользователь.

3. Создание ОС процесса нужной для работы некоторой служебной ОС может создать процесс, для выполнения некоторой операции.

4. Порождение одного процесса другим с целью структуризации программы или использования возможности параллельных вычислений, программа может создавать другие процессы.

Основные причины завершения процессов следующие.

1. Обычное завершение.

2. Превышение программой времени, отведенной на её выполнение.

3. Недостаточный объем памяти.

4. Выход за пределы отведенной области памяти.

5. Ошибки защиты.

6. Арифметические ошибки (деление на ноль; использование чисел, выходящих за разрядную сетку).

7. Излишнее ожидание (т.е. процесс ждет наступление события больше, чем указано в параметрах системы).

8. Ошибки ввода-вывода.

9. Неверная команда.

10. Команда с недоступными привилегиями.

11. Неправильное использование данных.

12. Вмешательство оператора или ОС.

13. По требованию родительского процесса.

ОС может завершить любой процесс, находящийся в любом состоянии по своему усмотрению.

3.1.2 Описание процессов

ОС можно представить как некий механизм, управляющий тем, как процессы используют системные ресурсы. И т.к. одна из задач ОС- управление процессами и ресурсами, то ОС должна располагать информацию о текущем состоянии каждого ресурса и процесса. Для этих целей ОС создает и поддерживает таблицы с информацией по каждому объекту управления. Общая структура таких таблиц может иметь вид, представленный на рис. 3.2.

3.1.2.1 Структуры управления процессами

Для управления ОС должна знать где находится процесс и атрибуты нужные для управления процессами. Набор атрибутов, которые используется ОС называется управляющим блоком процесса. Имеется понятие – образ процесса. В образ входит следующая информация.

1. Данные пользователя – это допускающая изменения часть пользовательского пространства, а также данные программы, пользовательский стек и модифицированный код

2. Пользовательская программа – программный код, который надо выполнить

3. Системный стек. С каждым процессом связаны один или несколько системных стеков. Стек используется для хранения параметров, адресов вызовов процедур и системных функций

4. Управляющий блок процесса – данные нужные ОС для управления процессом

Местонахождение образа процесса зависит от системы управления памятью. Но для управления процессом надо, чтобы какая-то его часть находилась в основной памяти. А чтобы запустить процесс, его надо полностью загрузить в основную и виртуальную память. Образы состоят из блоков, которые необязательно должны располагаться последовательно.

Атрибуты процессов.

1. Идентификаторы процесса:

- идентификатор данного процесса;
- идентификатор родительского процесса;
- идентификатор пользователя.

2. Информация о состоянии процесса:

- регистры, доступные пользователю (те, к которым можно обращаться с помощью машинных команд);
- управляемые регистры и регистры состояния (те, которые управляют работой процесса). Должны быть счетчик команд, который содержит адрес очередной команды, коды условия, которые отражают результат выполнения

арифметической или логической команды и информация о состоянии;

- указатели на стек – указывает на вершину стека, с процессом связывается один или несколько стеков.

3. Управляющая информация процесса:

а) информация по планированию и состоянию: состояние процесса, приоритет, может быть несколько значений приоритета: по умолчанию, текущий, максимально возможный. Информация, связанная с планированием: квант времени в течение, которого процесс выполняется при последовательном запуске. Информация о событии: идентификация события, инициализирующего продолжение работы события;

б) структуры данных: процесс может быть связан с другими процессами с помощью очереди, кольца или другими структурами. Для этого в управляемом блоке должны быть указатели на другие процессы;

в) обмен информацией между процессами: флаги, сигналы, сообщения;

г) привилегии процессов – это права доступа к определенным областям памяти, права выполнять определенные виды команд, возможности использования системных утилит;

д) управление памятью: это указатели на таблицы сегментов или страниц, описывающих распределение процессов в виртуальной памяти;

е) владение ресурсами и их использование: перечень открываемых файлов, сведения об исполнении процесса и других устройств ввода-вывода.

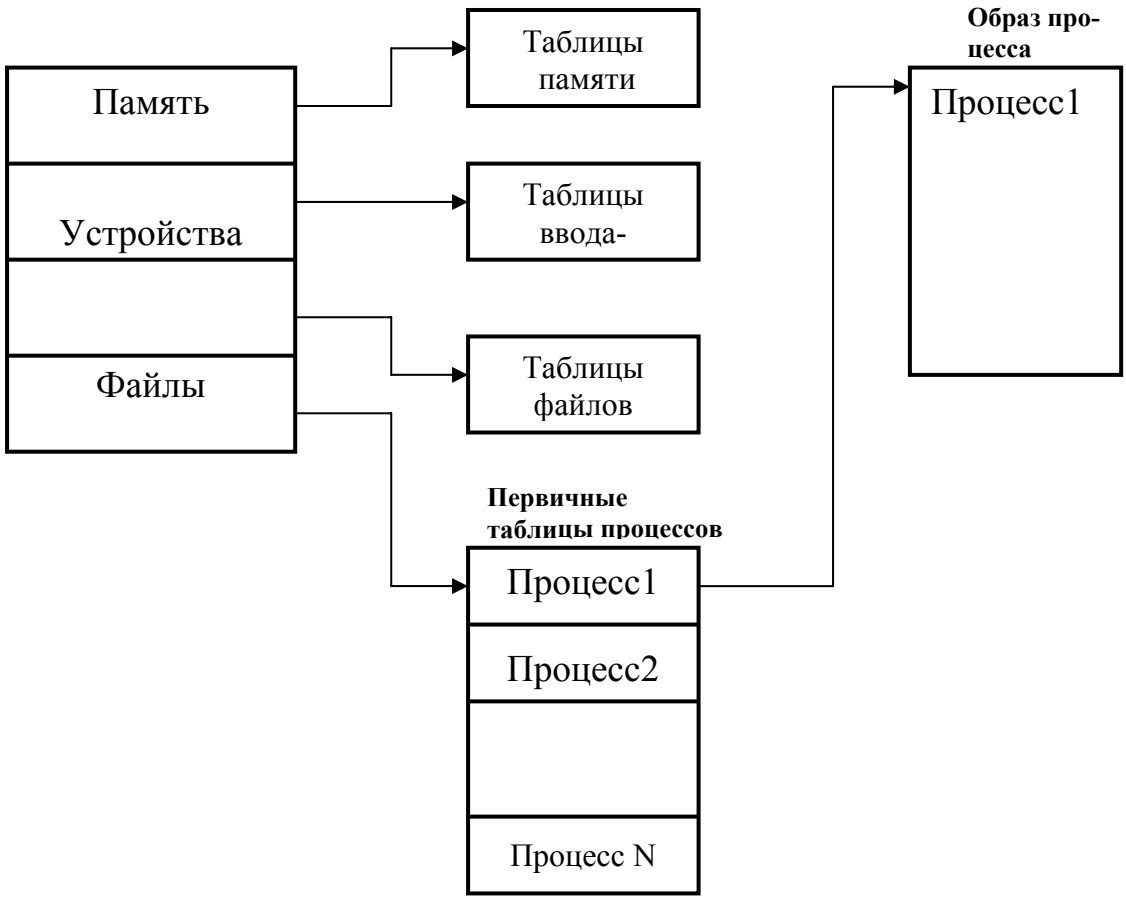


Рис. 3.2. Общая структура таблиц процесса с информацией по каждому объекту управления

Замечания по организации таблиц

1. Хотя таблицы имеют разное назначение, все они должны быть связаны между собой и иметь перекрестные ссылки. Например, доступ к файлам, информация о которых хранится в таблице файлов, осуществляется через устройства ввода-вывода, а информация из файлов будет находиться в основной или виртуальной памяти на протяжении какого-то времени.

2. Для создания таблиц ОС должны иметь информацию по основной конфигурации вычислительной системы (например, такая информация, как объем основной памяти, количество и бит устройств ввода-вывода, идентификаторы устройств и т.д.) Сами таблицы должны быть доступны для ОС, поэтому место для них выделяется системой управления памяти.

Таблицы памяти.

Предназначены для хранения информации об основной ОП и виртуальной памяти. Эти таблицы включают следующую информацию:

1) объем основной памяти**,** отведенной процессу;

2) объем вторичной или виртуальной памяти, отведенной процессу;

3) все атрибуты защиты блоков основной и виртуальной памяти;

4) всю информацию, необходимую для управления виртуальной памятью.

Таблицы ввода-вывода.

Используются для управления устройствами ввода-вывода и каналами компьютерной системы. В каждый момент времени устройство ввода-вывода может быть либо свободным, либо отданным в распоряжение какому-либо процессу. Если выполняется операция ввода-вывода, то должна быть информация о состоянии этой операции. Например, какие адреса ОП задействованы в этой операции, кто является отправителем и получателем отправляемой информации.

Таблицы файлов.

В них находится информация о существующих файлах, их расположение на магнитных носителях, текущем состоянии и других атрибутов. В ОС может быть специальная подсистема управления памятью.

Таблицы процессов.

Содержат сведения о процессах, располагая которыми ОС может управлять процессами.

3.1.2.2 Управление процессами

Модели выполнения.

Большинство процессов поддерживают два режима работы:

1. *Режим с невысокими привилегиями* – пользовательский.
2. *Режим с высокими привилегиями* – системный.

Функции, возлагаемые на ядро ОС.

1. Управление процессами:
а) создание и уничтожение;
б) планирование и диспетчеризация;
в) переключение;
г) синхронизация;
д) организация управления блокировкой.
2. Управление памятью:
а) выделение адреса пространства процесса;
б) свопинг;
в) управление страницами и сегментами.
3. Управление вводом-выводом:
а) управление буферами;
б) выделение процессам каналов и устройств ввода-вывода.
4. Функции поддержки:
а) обработка прерываний;
б) учет использования ресурсов;
в) текущий контроль системы.

Создание процесса.

При создании процесса необходимо выполнить следующие действия.

1. Присвоить началу процесса уникальный идентификатор.

2. Выделить память.

3. Инициализировать управляющий блок процесса.

4. Установить необходимые связи.

5. Создать или расширить другие структуры данных.

Переключение процессов.

Возникают вопросы.

1. Какие моменты должны приводить к переключению процессов?

2. Как установить разницу переключением режима работы и переключением процесса?

Переключение процесса может произойти в любой момент, когда управление от выполняющегося процесса переходит к системе.

Для это существуют следующие причины:

1. Прерывание, внешнее по отношению к текущей команде.

2. Ловушка, связанная с выполнением текущей команды.

3. Вызов супервизора, т.е. функции ОС.

Можно отметить несколько типов прерываний, которые приводят к переключению на систему:

1. *прерывание таймера*, когда процесс отработал свой квант.

2. *прерывание ввода-вывода*, когда процесс ждет ввода или уже вывел.

Переключение режимов.

Переключение режимов происходит при обработке прерываний. При этом процессор выполняет следующие действия:

1. Сохраняет контекст программы, устанавливает в счетчике команд адрес процедуры-обработчика прерывания.

2. Изменяет состояние процесса. При это выполняются следующие действия:

а) по сохранению полного контекста;

б) помещение управляющего блок процессов в соответствующую очередь;

в) выбор следующего для выполнения и восстановления контекста выбранного процесса.

3.1.3 Концепция потока как составной части процесса

Концепция процесса объединяет в себе две отдельные независимые концепции. Одна из них имеет отношение к владению ресурсами, а другая – к выполнению процесса. В некоторых ОС это привело к появлению конструкции, называемой *потоком* (thread).

В классическом понимании концепция процесса характеризуется двумя параметрами.

1. *Владение ресурсами* (resource ownership)

Процесс включает виртуальное адресное пространство, в котором содержится образ процесса, и на протяжении каких-то интервалов времени может владеть такими ресурсами, как основная память, каналы и устройства вво-

да/вывода, файлы, или на некоторое время получать контроль над ними. ОС выполняет защитные функции, предотвращая нежелательное взаимодействие процессов в сфере распределения ресурсов.

2. *Планирование выполнения*

Выполнение процесса осуществляется путём выполнения кода одной или нескольких программ, причём выполнение процессов может чередоваться друг с другом.

Процесс имеет такие параметры, как:

– состояние процесса;

– текущий приоритет, в соответствии с которым происходит планирование выполнения процесса.

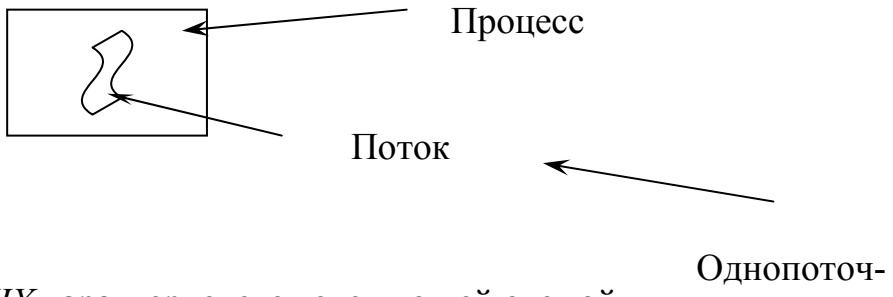
Эти характеристики процесса являются его сущностью, но ОС может рассматривать их отдельно.

Для того, чтобы различать эти две характеристики, единицу диспетчеризации называют *потоком или облегченным процессом (lightweight process)*, а единицу владения ресурсами – *процессом или заданием.* Пока эта терминология не является полностью устоявшейся.

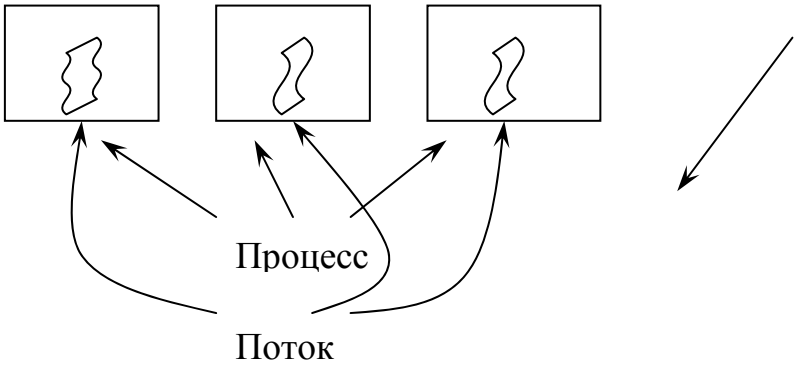
3.1.4 Многопоточность

Многопоточность – это способность ОС поддерживать в рамках одного процесса выполнение нескольких потоков. Процесс, представляющий собой единый поток выполнения, называется *однопоточным.*

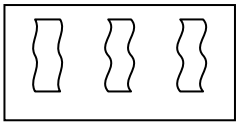
MS-DOS является примером однопоточной ОС.



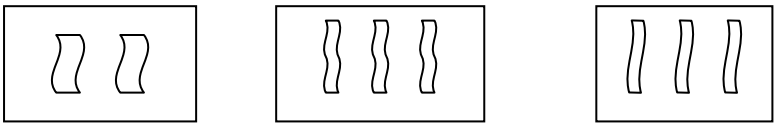
ОС UNIX характеризуется следующей схемой



JAVA является средой, в которой один процесс может расщепляться на несколько потоков.



WINDOWS 2000, OS/2, LINUX являются многопроцессорными, многопоточными системами.



В многопоточной среде процесс определяется, как структурная единица распределения ресурсов, а также как структурная единица защиты.

В рамках процесса может находиться один или несколько потоков, каждый из которых обладает следующими характеристиками:

1. Состояние выполнения потока
2. Сохраненный контекст невыполненного потока (поток имеет независимый счётчик команд, работающий в рамках процесса).
3. Стек выполнения.
4. Статическая память, выделяемая под переменные.
5. Доступ к памяти и ресурсам процесса, которому принадлежит поток. Этот доступ разделяется всеми потоками процесса.

3.1.5 Однопоточная модель процесса

Рис. 3.3. Однопоточная модель процесса

В представление процесса с одним потоком входит:

- блок управления процесса;
- пользовательское адресное пространство;
- стек ядра;

- стек пользователя, с помощью которого осуществляется вызов процедур и возврат из них;

Когда выполнение процесса прерывается, то содержимое регистров сохраняется в памяти.

3.1.6 Многопоточная модель процесса

В многопоточной среде с каждым процессом также связаны управляющий блок и адресное пространство. Но для каждого потока создаётся свой отдельный стек и свой управляющий блок, в котором содержится значение регистров, приоритет и другая информация о состоянии потока.

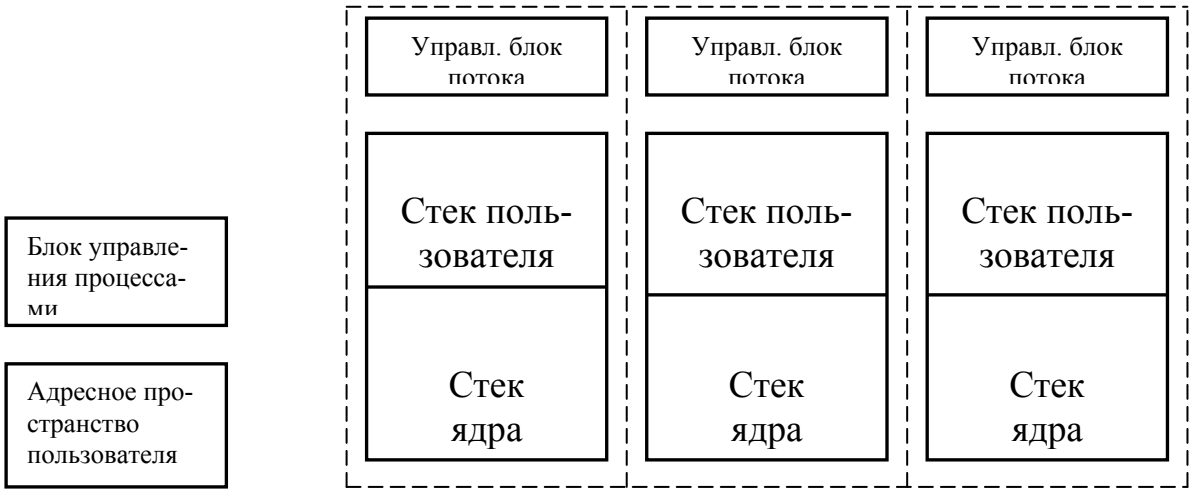


Рис. 3.3. Многопоточная модель процесса

Все потоки процесса разделяют между собой состояние и ресурсы этого процесса. Они находятся в адресном пространстве и имеют доступ к данным. Если один поток изменяет какие-то данные, то другие потоки во время своего доступа к данным могут отследить эти изменения. Если один поток открывает файл, то другие потоки могут читать информацию из этого файла. Порождение новых потоков создаёт новую линию выполнения команд притом, что доступ идёт к одним и тем же данным.

С точки зрения производительности, использование потоков имеет следующие преимущества.

1. Создание нового потока в существующем процессе занимает на много меньше времени, чем создание нового процесса.

2. Поток можно завершить на много быстрее, чем процесс.

3. Переключение потоков в рамках одного процесса происходит на много быстрее, чем переключение процессов.

4. При использовании потоков повышается эффективность обмена информацией между двумя выполняющимися программами. В большинстве ОС об-

мен данными между независимыми процессами происходит с участием ядра. А так как потоки используют одно и тоже адресное пространство, то они могут обмениваться информацией без участия ядра.

Примером приложения, в котором можно применить поток, является файловый сервера. Так как серверу приходится обрабатывать большое количество запросов, то необходимо создание потоков. Если серверная программа работает на многопроцессорной машине, то на разных процессорах в рамках одного процесса могут выполняться одновременно несколько потоков.

3.1.7 Функциональность потоков

Основными состояниями потоков являются:

– состояние выполнения;

– состояние готовности;

– состояние блокировки.

С изменением состояния потоков связаны 4 действия.

Порождение.

Во время создания нового процесса, вместе с ним создаётся и поток этого процесса. В рамках одного и того же процесса поток может породить другой поток, определив его указатель команд и аргумент. Новый поток создаётся со своим собственным контекстом регистров и стековым пространством, после чего он помещается в очередь готовых к выполнению потоков.

Блокирование.

Если потоку нужно подождать, пока не наступит некоторое событие, то он блокируется. При этом сохраняется содержимое его пользовательских регистров, счётчик команд, указатель стека. После этого процессор может перейти к выполнению другого потока.

Разблокирование.

Когда наступает событие, которое ожидал блокированный поток, тогда этот поток переходит в состояние готовности.

Завершение.

После завершения стек потока удаляется.

Синхронизация потоков.

Так как все потоки используют одно и тоже адресное пространство и другие ресурсы, действия различных потоков надо синхронизировать, чтобы они не мешали друг другу и не повредили совместно используемые данные. При синхронизации потоков используются те же методы, что и при синхронизации процессов. Это значит, что решаются задачи потребитель-производитель, чтение-запись и другие.

3.1.8 Взаимодействие процессов

3.1.8.1 Задача взаимного исключения

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

В качестве примера рассматривается два последовательных процесса, которые удобно считать циклическими. В каждом цикле выполнения процесса существует критический интервал. Это означает, что в любой момент времени только один процесс может находиться внутри своего критического интервала. Чтобы осуществить такое взаимное исключение, оба процесса имеют доступ к некоторому числу общих переменных. Операции проверки текущего значения такой общей переменной и присваивание нового значения общей переменной рассматриваются как неделимые. То есть, если два процесса осуществляют присваивание новое значение одной и той же общей переменной одновременно, то присваивание происходит друг за другом и окончательное значение переменной одному из присвоенных значений, но никак не их смеси. Если процесс поверяет значение переменной одновременно с присваиванием ей значения другим процессом, то первый процесс обнаруживает либо старое, либо новое значение, но никак не их смесь.

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

```
      end;
parend;
end;
```

```
int turn=1;
void P0()
{
  while (1)
  {
    while(turn!=0)
    критический интервал 1;
    turn=1;
    ….
  }
}
void P1()
{
  while (1)
  {
    while(turn!=1)
    критический интервал 2;
    turn=0;
    ….
  }
}
void main()
{
  parbegin(P0,P1);
}
```

Недостатки решения:

1. Процессы могут входить в критический интервал строго последовательно. Темпы развития задаются медленным процессом.

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

Второй способ решения:

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

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

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

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

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

Было предложено следующее решение (вариант 3)::

```
begin integer C1,C2;
   C1 := 1;
   C2 := 1;
   parbegin
     процесс 1: begin
   A1: C1 := 0;
   L1: if (C2 = 0) then goto L1;
         критический интервал 1;
         C1 := 1;
         остаток цикла 1;
         goto A1;
       end;
    процесс 2: begin
   A2: C2 := 0;
   L2 if (C1 = 0) then goto L2;
         критический интервал 2;
         C2 := 1;
         остаток цикла 2;
         goto A2;
       end;
parend;
end;
```

```
int flag[2];
void P0()
{
  while (1)
  {
    flag[0]=1;
    while (flag[1]);
    критический интервал 1;
    flag[0]=0;
    ….
```

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

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

Решение 4.

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

```
                                goto L2;
                            end;
        критический интервал 2;
        C2 := 1;
        остаток цикла 2;
        goto L2;
      end;
parend;
end;
```

```
int flag[2];
void P0()
{
  while (1)
  {
    flag[0]=1;
    while (flag[1]);
    {
      flag[0]=0;
      задержка;
      flag[0]=1;
    }
    критический интервал 1;
    flag[0]=0;
    ….
  }
}
void P1()
{
  while (1)
  {
    flag[1]=1;
    while (flag[0]);
    {
      flag[1]=0;
      задержка;
      flag[1]=1;
    }
    критический интервал 2;
    flag[1]=0;
    ….
  }
}
```

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

Решение задачи взаимного исключения. Алгоритм Деккера.

```
begin integer C1,C2,очередь;
   C1 := 1;
   C2 := 1;
   очередь := 1;
   parbegin
     процесс 1: begin
   A1: C1 := 0;
   L1: if (C2 = 0) then begin
                          if (очередь = 1) then goto L1;
                         C1 := 1;
                    B1: if (очередь = 2) then goto B1;
                         goto A1;
                       end;
         критический интервал 1;
         очередь := 2;
         C1 := 1;
         остаток цикла 1;
         goto A1;
       end;
       процесс 2: begin
   A2: C2 := 0;
    L2: if (C1 = 0) then begin
                               if (очередь = 2) then goto L2;
                               C2 := 1;
                          B2: if (очередь = 1) then goto B2;
                               goto A2;
                           end;
         критический интервал 2;
         очередь := 1;
         C2 := 1;
         остаток цикла 2;
         goto A2;
       end;
parend;
end;
```

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

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

Решение задачи взаимного исключения. Алгоритм Пэтерсона..

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

Для доказательства корректности задачи взаимного исключения необходимо проверить три положения.

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

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

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

3.1.8.2 Обобщенная задача взаимного исключения

Даны N процессов, каждый со своим критическим интервалом. Необходимо организовать их функционирование так, чтобы в любой момент самое большее один процесс находился в критическом интервале. Для проверки правильности решения проверяем три условия.

```
begin integer array b, c [0..N];
  integer очередь;
  for очередь = 1 step 1 until N do begin
                                        b [очередь] := 1;
                                        c [очередь] := 1;
                                        end;
  очередь := 0;
  parbegin
    процесс 1: begin … end;
    процесс 2: begin … end;
…
    процесс N: begin … end;
  parend;
end;
    процесс i: begin integer j, k;
      Ai: b [i] := 0;
      Li: if (очередь <> i) then begin
                                        C[i] = 1;
                                        k = очередь;
                                        if (b [k] = 1) then очередь := i;
                                        goto Li;
                                  end;
      C[i] := 0;
      for j := 1 step 1 until N do
                                  if ((j <> i) and (C[j] = 0)) then goto Li;
      Критический интервал i;
      очередь := 0;
      C[i] := 1;
      b[i] := 1;
      Остаток цикла i;
      goto Ai;
    end;
```

3.1.9 Синхронизирующие примитивы (семафоры). Применение семафоров для решения задачи взаимного исключения

Неделимое обращение к общей переменной всегда подразумевает одностороннее движение информации. Отдельный процесс может либо присвоить новое значение, либо проверить текущее значение. Однако такая проверка не оставляет следов для других процессов. Вследствие этого, в то время как процесс желает отреагировать на текущее значение общей переменной, значение этой переменной может быть изменено другими процессами. То есть такое взаимодействие через общие переменные нельзя считать во всех случаях адекватным. Для разрешения такой ситуации:

1. Вводятся специальные целочисленные общие переменные, называемые семафорами.

2. Добавляются к набору действия, из которых состоят процессы, два новых примитива: p-операция и v-операция.

Эти две операции всегда выполняются над семафорами и представляют единственный способ обращения к семафорам со стороны одновременно действующих процессов. Для решения задачи взаимного исключения область значений семафоров 0 и 1 (двоичные семафоры), но существует понятие и общего семафора, при котором он может принимать определенное количество целочисленных значений.

s1 – семафор,

p(s1) или v(s1) – запись операций над семафором.

V-операция – операция с одним аргументом, который должен быть семафором. Ее назначение – увеличение аргумента на единицу. Это действие рассматривается как неделимая операция.

Имеется достаточно принципиальное различие между семафорными и обычными операциями сложения. Если двумя параллельно-развивающимися процессами выполнена семафорная операция v(s1) над общим семафором s1, получим увеличение семафора s1 на два, так как операция v неделимая. А выполнение операции сложения s1 с единицой параллельными процессами может привести к увеличению значения s1 на единицу.

П1	П2
S1 := S1+1	S1 := S1+1

П1	П2
v(S1)	v(S1)

P-операция – операция с одним аргументом, который должен быть семафором. Ее назначение – уменьшение аргумента на единицу, если только результирующее значение не становится отрицательным. Завершение p-операции, то

есть решение о том, что в настоящий момент является подходящим для выполнения уменьшения и последующее уменьшение значения аргумента, рассматривается как неделимая операция.

P-операция определяет потенциальную задержку. Если инициируется p-операция над семафором, который в этот момент равен нулю, то в данном случае p-операция не может завершиться, пока какой-либо другой процесс не выполнит v-операцию над тем же семафором и не присвоит ему значение 1. Несколько процессов могут одновременно начать p-операцию над одним семафором.

Утверждение о том, что завершение p-операции есть неделимое действие, означает, что когда семафор получит значение 1, только одна из начавшихся p-операций над семафором завершится, какая именно – не определено.

```
begin integer свободно;
  свободно := 1;
  очередь := 0;
  parbegin
    процесс 1: begin … end;
    процесс 2: begin … end;
…
    процесс N: begin … end;
  parend;
end;
    процесс i:
    begin
      Li: p(свободно);
      Критический интервал i;
      V(свободно);
      Остаток цикла i;
      goto Li;
    end;
```

3.1.10 Задача “производитель-потребитель”

3.1.10.1 Общие семафоры

Рассматриваются два процесса, которые называются производитель и потребитель. Оба процесса являются циклическими. Производитель при каждом циклическом повторении участка программы производит отдельную порцию информации, которая должна быть обработана потребителем. Потребитель при каждом повторении обрабатывает следующую порцию информации, выработанную производителем. Отношения производитель-потребитель подразумевают односторонний канал связи, по которому могут передаваться порции ин-

формации. С этой целью процессы связаны через буфер неограниченной емкости. То есть, произведенные порции не должны немедленно потребляться, а могут организовывать в буфере очередь. Буфер работает по принципу FIFO.

ЧПБ – число порций в буфере.
РБ – работа с буфером

```
begin integer ЧПБ;
  ЧПБ := 0;
  parbegin
    производитель: begin
      П1: производство новой порции;
      добавление новой порции в буфер;
      v(ЧПБ);
      goto П1;
    end;

    потребитель: begin
      П2: p(ЧПБ);
      взятие порции из буфера;
      обработка взятой порции;
      goto П2;
    end;
parend;
end;
```

При неограниченном буфере производитель никаких ограничений не имеет, а потребитель может читать данные из буфера только если они там есть. Операции добавления порции к буферу и взятия порции из буфера могут помешать друг другу, если не предусмотреть их взаимное исключение во время исполнения. Этого можно избежать с помощью двоичного семафора.

```
begin integer ЧПБ, РБ;
  ЧПБ := 0;
  parbegin
    производитель: begin
      П1: производство новой порции;
      p(РБ)
      добавление новой порции в буфер;
      v(РБ);
      v(ЧПБ);
      goto П1;
    end;
```

```
      потребитель: begin
  П2: p(ЧПБ);
       p(РБ);
        взятие порции из буфера;
        v(РБ);
        обработка взятой порции;
        goto П2;
      end;
parend;
end;
```

При решении стремятся использовать только двоичные семафоры. В таком случае программа будет выглядеть так:

ЗП - задержка потребителя.

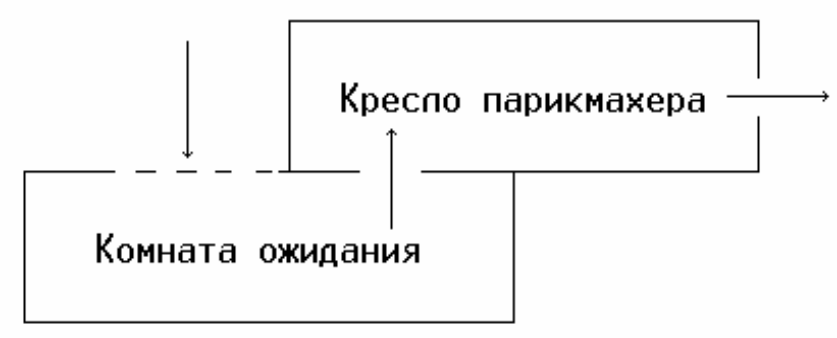
СЧПБ - счетчик порций в буфере.

```
begin integer ЧПБ, РБ, ЗП;
  ЧПБ := 0;
  РБ := 1;
  ЗП := 0;
  parbegin
    производитель: begin
      П1: производство новой порции;
      p(РБ)
      добавление новой порции в буфер;
      ЧПБ := ЧПБ + 1;
      If (ЧПБ = 1) then v(ЗП);
      v(РБ);
      goto П1;
    end;
потребитель: begin integer СЧПБ;
      ждать:          p(ЗП);
      продолжить: p(РБ);
      взятие порции из буфера;
      ЧПБ := ЧПБ - 1;
      СЧПБ := ЧПБ;
      v(РБ);
      обработка взятой порции;
      if (СЧПБ = 0) then goto ждать;
                    else goto продолжить;
    end;
parend;
end;
```

Значение локальной переменной СЧПБ устанавливается при взятии порции из буфера и фиксируется, не была ли эта порция последней. Здесь двоичный семафор РБ решает задачу взаимного исключения при работе с буфером. Введен новый двоичный семафор – задержка потребителя (ЗП). Программа представляет интерес в течение времени, когда буфер пуст. В исходном состоянии семафор ЗП установлен в ноль и открывается производителем только в том случае, когда в пустой буфер записывается порция. Остановка потребителя на семафоре ЗП не блокирует работу с буфером, так как стоит выше операции р(РБ). Смысл ждать есть только в том случае, если буфер пуст.

3.1.10.2 Задача “производитель-потребитель”, буфер неограниченного размера

```
begin
  integer ЧПБ, РБ, ЗП;
  ЧПБ:=0; РБ:=1; ЗП:=0;
  parbegin
   производитель: begin
                n1: производство новой порции;
                   P(РБ);
                   добавление новой порции к буферу;
                   ЧПБ:=ЧПБ+1;
                   if (ЧПБ=0) then begin V(РБ); V(ЗП); end
                                 else V(РБ);
                   goto n1;
               end;
   потребитель:   begin
                n2: P(РБ);
                   ЧПБ:=ЧПБ-1;
                   if (ЧПБ=-)1 then begin V(РБ); P(ЗП); P(РБ); end;
                   взятие порции из буфера;
                   V(РБ);
                   обработка взятой порции;
                   goto n2;
               end;
  parend;
```



Эта программа называется "спящий парикмахер"

Используется два двоичных семафора: работа с буфером и задержка потребителя.

Особенности:

- используются только двоичные семафоры;
- задача взаимного исключения реализуется через семафоры;
- через семафоры реализуется задержка потребителя, которая действует при попытке потребителя читать данные из пустого буфера.

3.1.10.3 Задача “производитель-потребитель”, буфер ограниченного размера

Производитель и потребитель связаны через буфер ограниченной емкости в N порций. ЧПП - число пустых порций.

```
 begin
   integer ЧПБ, ЧПП, РБ;
   ЧПБ:=0;
   ЧПП:=N;
   РБ:=1;
   parbegin
     производитель: begin
                      n1: производство новой порции;
                          P(ЧПП);
                          P(РБ);
                          добавление порции к буферу;
                          V(РБ);
                          V(ЧПБ);
                          goto n1;
                    end;
     потребитель: begin
                    n2: P(ЧПБ);
```

```
            P(РБ);
            взятие порции из буфера;
            V(РБ);
            V(ЧПП);
            обработка взятой порции;
            goto n2;
        end;
  parend;
end;
```

И производитель и потребитель решают через РБ задачу взаимного исключения. Проблема производителя: нельзя писать в заполненный буфер. Проблема потребителя: нельзя читать из пустого буфера. Производитель решает свою проблему с использованием общего семафора ЧПП (число пустых порций), а потребитель — через ЧПБ (число порций в буфере).

3.1.11 Взаимодействие через переменные состояния

3.1.11.1 Постановка задачи

Исходные данные.

Можно определять и формировать исходные процессы.

Процессы могут связываться друг с другом через общие переменные.

Имеются средства синхронизации.

При взаимодействии процессов могут возникать решения, которые касаются более чем одного процесса. Не всегда очевидно, какое решение будет принято. Если не найден какой-то руководящий принцип (например, критерии эффективности), то с целью определённости нужно установить некоторые ограничения.

Если разрабатываются реально работающие системы, то требуется убедиться в корректности решения. При параллельном программировании использование проверяющих тестов затруднено, поэтому вопросы проверки правильности решения должны рассматриваться в самых начальных этапах проектирования системы.

3.1.11.2 Пример применения приоритетного правила

Рассматриваются производители, которые выдают порции информации различного размера и размер порции выражается в некоторых единицах. Потребитель обрабатывает последовательные порции из буфера и умеет обрабатывать порции, размер которых заранее не задан. Максимальный размер порции известен. Максимальный размер буфера определён в единицах информации, а не в количестве порций. Размер буфера - не менее максимального размера пор-

ции информации. Вопрос о возможности размещения в буфере определенной выработанной порции зависит от размера этой порции.

Если имеется более одного производителя и какой-то из них ждет из-за отсутствия достаточного места в буфере, то другие производители могут продолжать работу и достигнуть точки, когда они желают выдать выработанную порцию информации в буфер.

При принятии решения, кому первому поместить информацию в буфер, формируется требование: производитель, предлагающий большую порцию, имеет больший приоритет. Если порции равны, то не имеет значения, кто добавит информацию первым.

Дано: N — производителей; M — потребителей; RB — размер буфера.

```
begin
  integer array желание[1:N], СП[1:N];
  integer ЧПБ, ББ, РБ, ЧСЕБ, i;
  for цикл:=1 step 1 until N do begin
                                  желание[i]:=0;
                                  СП[i]:=0;
                                    end;
  ЧПБ:=0; ЧСЕБ:=RB;
  ББ:=0; РБ:=1;
  parbegin
    производитель 1: begin ... end;
    . . . . . .
     производитель n: begin integer РП;
        цикл n:  производство  новой  порции и установка размера порции  (РП);
        P(РБ);
        if ( (ББ=0) and (ЧСЕБ>=РП) ) then
                                          ЧСЕБ:=ЧСЕБ-РП
                                      else
                                          begin
                                          ББ:=ББ+1;
                                          желание[n]:=РП;
                                          V(РБ);
                                          P(СП[n]);
                                          P(РБ);
                                          end;
        добавление порции к буферу;
        V(РБ);
        V(ЧПБ);
        goto цикл n;
end;
                    ... ... ...
производитель N: begin ... end;
```

```
потребитель 1: begin ... end;
               ... ... ...
потребитель m: begin
                    integer РП, i, max, nmax;
       цикл m:  Р(ЧПБ);
                Р(РБ);
                взятие порции из буфера и установка РП;
                ЧСЕБ:=ЧСЕБ+РП;
        проверка: if (ББ>0) then
                                begin
                                max:=0;
                                for i:=1 step 1 until N do
                                                begin
                                        if (max<желание[i]) then
                                                                begin
                                                                max:=желание[i];
                                                                nmax:=i;
                                                                end
                                                end;
                                if (max=<ЧСЕБ) then
                                                        begin
                                                        ЧСЕБ:=ЧСЕБ-max;
                                                        желание[nmax]:=0;
                                                        ББ:=ББ-1;
                                                        V(СП[nmax]);
                                                        goto проверка;
                                                        end;
                                end;
                V(РБ);
                обработка взятой порции;
                goto цикл m;
             end;
```

Для решения задачи вводится переменная состояния для каждого производителя - “желание”, обозначающая число единиц информации в буфере, необходимых для размещения порции, которая в текущий момент не может быть добавлена к буферу. Если для процесса-производителя эта переменная равна нулю, то процесс-производитель не имеет неудовлетворённых требований на добавление порций.

Для каждого производителя вводится двоичный семафор СП (семафор производителя).

Для буфера вводится двоичный семафор РБ (работа с буфером), предназначенный для взаимного исключения при работе с буфером в широком смыс-

ле, т.е. не только взятие и добавление, но также проверка и модификация связанных с буфером переходных состояний.

Как только в буфер добавляется новая порция, она может быть обработана. Так как безразлично, кто из потребителей её возьмёт, то для определения этого может быть использован общий семафор ЧПБ (число порций в буфере). О свободных областях в буфере производителям сообщается через целочисленную переменную состояния ЧСЕБ (число свободных единиц в буфере). Введена целочисленная переменная ББ (блокировка буфера), значение которой определяет, сколько процессов-производителей имеют желание добавить порцию в буфер, но не смогли разместить свои порции в буфере и она уведомляет производителя в том, что уже есть процессы, которые обнаружили, что ББ>0, то он должен присоединиться к процессам, которые ожидают размещения порции в буфер.

3.1.12 Мониторподобные средства синхронизации

3.1.12.1 Введение

Основной концепцией таких средств является организация контроля правильности установления взаимосвязи между процессами.

Считается, что данные средства синхронизации имеют большую наглядность, чем семафорная техника, что достигается за счет структурирования.

Концепция мониторподобных средств основана на языковом обособлении или локализации средств взаимодействия. Это не только выделение особых языковых конструкций, но и сосредоточение в их составе информации о разделяемых ресурсах, о переменных состояния, характеризующих эти ресурсы, а также допустимые действия над этими ресурсами.

3.1.12.2 Механизм типа «критическая область»

Как следует из названия, это механизм, ориентированный на решение задачи взаимного исключения.

Для построения критической области используется две языковые конструкции. Одна из них VAR V: SHARED T; предназначена для описания критического ресурса, и её используют в начале текста программы в области описания типов переменных. Вторая описывает доступ к ресурсу в тексте программы REGION V DO S.

```
VAR     R: SHARED T1;
        Q: SHARED T2;
Begin
    Parbegin
        процесс 1: . . . L1: REGION R DO S1; . . .
```

процесс 2: . . . L2: REGION Q DO S2; . . .

. . . . . .

процесс i: . . . L_i: REGION R DO S3; . . .

Parend;

End.

Процессы 1-ый и i-ый взаимно разделяют ресурс R и при выполнении программы будет выполнятся только одна из областей.

Описание переменной V: SHARED T означает, что определяется ресурс с именем V, как некоторая переменная, доступная параллельным процессам. Тип ресурса задаётся его описанием T.

Для осуществления доступа к ресурсу V в тексте программы процесса требуется использовать конструкцию вида REGION V DO S. Такая конструкция описывает отдельную критическую область относительно критического ресурса V и определяет действия S, которые будут осуществлены над ресурсом. Такие конструкции при исполнении исключают друг друга относительно критического ресурса.

Основные допущения при построении такого механизма синхронизации заключаются в том, что недопустима обработка переменной, описанной как «разделяемая» в каких-либо языковых конструкциях в программе, отличных от конструкций типа REGION. Попытка сделать это – заведомая ошибка в использовании критического ресурса – может быть выявлена на этапе компиляции программы.

```
TYPE T = ARRAY 1..100 OF INTEGER;
VAR M: SHARED T;
Begin
    Parbegin
Процесс 1: L1: <действие процесса>
        REGION M DO <обработка массива M>;
        GOTO L1;
. . . . . .
    Parend;
End.
```

3.1.12.3 Механизм типа «условная критическая область»

```
TYPE T = ARRAY 1..100 OF INTEGER;
VAR M:  SHARED T;
        СЧИТЫВАНИЕ : BOOLEAN;
BEGIN
    СЧИТЫВАНИЕ :=TRUE;
    PARBEGIN    ПРОЦЕСС 1 : BEGIN
                        M1: <НЕКОТОРЫЕ ДЕЙСТВИЯ>
```

```
                    REGION M DO BEGIN
                    AWAIT(СЧИТЫВАНИЕ);
                    <СЧИТАТЬ ИНФОРМАЦИЮ ИЗ М>
                    СЧИТЫВАНИЕ := TRUE;
                                END;
                    GOTO M1;
                       END;
          ПРОЦЕСС 2 : BEGIN
                    M2: <НЕКОТОРЫЕ ДЕЙСТВИЯ>
                    REGION M DO BEGIN
                    <ЗАПИСЬ ИНФОРМАЦИИ В М>
                    СЧИТЫВАНИЕ := FALSE;
                                END;
                    GOTO M2;
                       END;
      PAREND;
END.
```

Данный механизм является модификацией механизма «критическая область». Но здесь вводится возможность производить работу с критическим ресурсом только тогда, когда он пребывает в определенном состоянии или выполнено условие, допускающее возможность работы с критическим ресурсом.

Для проверки условия и состояния вводится специальный примитив AWAIT. Он может выполняться только в пределах языковой конструкции REGION. В качестве параметра этого примитива используется логическое выражение. Предполагается, что значения компонентов логического выражения могут изменяться другими параллельными процессами, но обязательно в области REGION. Если при выполнении AWAIT окажется, что логическое выражение истинно, то данный процесс временно покидает критическую область. Это даёт возможность одному из других параллельных процессов войти в критическую область и изменить параметры логического выражения. При повторном входе приостановленного процесса в критическую область логическое условие может стать ложным, и процесс получит возможность работать с критическим ресурсом.

Для организации временного выхода из критической области и повторного входа в неё по отношению к критическому ресурсу выстраиваются две очереди:

- в первую или главную очередь попадают те процессы, которые готовы войти в критическую область, и конкурируют за право получить доступ к ресурсу;

- в другую очередь, называемую событийной, попадают процессы, которые, находясь в критической области, обнаружили с помощью примитива AWAIT невозможность обработки критического ресурса.

Процессы, находящиеся в событийной очереди, переносятся в главную очередь после каждого нормального выхода какого-либо из процессов из его критической области по отношению к какому-либо ресурсу.

Такие действия могут быть реализованы компилятором путём применения двоичного семафора, единственного в отношении каждого критического ресурса.

```
TYPE T = ARRAY 1..100 OF INTEGER;
VAR      M : SHARED T;  СЧИТЫВАНИЕ : BOOLEAN;

      BEGIN
      СЧИТЫВАНИЕ :=TRUE;
      PARBEGIN
      ПРОЦЕСС 1 :  BEGIN
                  M1: <ДЕЙСТВИЯ ПРОЦЕССА> // ПРОЦЕСС ЧИТАТЬ
                  REION M DO BEGIN AWAIT (СЧИТЫВАНИЕ);
                                    <СЧИТАТЬ ИНФОРМАЦИЮ ИЗ М>
                                    СЧИТЫВАНИЕ := TRUE;
                                    END;
                  GOTO M1;
                  END;
ПРОЦЕСС 2 :
            BEGIN
                  M2: <ДЕЙСТВИЯ ПРОЦЕССА> //ПРОЦЕСС-ПИСАТЕЛЬ
                  REGION M DO BEGIN
                  <ЗАПИСЬ ИНФОРМАЦИИ В М>
                  СЧИТЫВАНИЕ := FALSE;
                  END;
                  GOTO M2;
            END;
PAREND;
END.
```

Процесс-читатель может выполнить действия только в том случае, если информация в массив записана.

Особенностью данного средства синхронизации является сокрытие от пользователей механизма повторного входа в критические области. Этот механизм является механизмом активного ожидания, поэтому его использование целесообразно, когда обращение к критическим ресурсам происходит редко.

3.1.12.4 Вторая модификация механизма «критическая область» (модификация второго рода)

Сущность модификации второго рода состоит в том, что в составе функции REGION допустимо использование двух примитивов AWAIT и COUSE по отношению к переменной E, называемой событийной.

Для описания этой переменной используется конструкция

VAR E : EVENT R.

Это значит, что E – событийная по отношению к критическому ресурсу R.

Переменная E является структурированной и представляет собой одномерный массив. В этот массив как в очередь попадают процессы, при работе которых был выполнен примитив AWAIT.

При выполнении примитива COUSE по отношению к этой переменной в ходе работы какого-либо из процессов все процессы, находящиеся в составе переменной E, получают возможность повторного входа в свои главные очереди.

При таком подходе процессы находятся в состоянии пассивного ожидания. Логическое условие явно не декларируется. О наступлении ожидаемого события сообщается с помощью примитива COUSE.

Пример.

Распределяется ресурс R, который состоит из M однотипных единиц. Любому процессу должна быть предоставлена возможность при каждом разовом обращении к системе получить из состава R одну единицу. Если в момент обращения нет ни одной свободной единицы, то процесс должен перейти в состояние ожидания до тех пор, пока не появится хотя бы одна свободная единица. После использования единицы ресурса процесс обязан её возвратить в состав ресурса R. О таком возвращении должны быть оповещены все процессы, ожидающие единицу ресурса.

```
BEGIN
TYPE    RES = 1..M,    P = 1..N;
VAR     INF : SHARED RECORD
                      СВОБОД_РЕСУРС : SEQUENCE OF RES;
                      ТРЕБОВАНИЕ : QUEUE OF P;
                      ОЧ_СОБЫТИЙ : ARRAY P OF EVENT R;
                      END;

PROCEDURE RESERVE(ПРОЦЕСС : P; VAR РЕСУРС : RES)
BEGIN
REGION INF DO   BEGIN
                WHILE EMPTY(СВОБОД_РЕСУРС) DO
                    BEGIN
                    ENTER(ПРОЦЕСС , ТРЕБОВАНИЕ);
```

```
                    AWAIT(ОЧ_СОБЫТИЙ(ПРОЦЕСС));
                    END;
                GET (РЕСУРС, СВОБОД_РЕСУРС);

END;

PROCEDURE RELEASE(РЕСУРС : RES)
BEGIN
VAR ПРОЦЕСС : P;
REGION INF DO
                BEGIN
                PUT(РЕСУРС , СВОБОД_РЕСУРС);
                IF (NOT EMPTY(ТРЕБОВАНИЕ)) THEN
                                BEGIN
                        REMOVE(ПРОЦЕСС , ТРЕБОВАНИЕ);
                        COUSE(ОЧ_СОБЫТИЙ(ПРОЦЕСС));
                                END;
        END;
END;

PARBEGIN
ПРОЦЕСС M1:     BEGIN
    VAR R1:RES;
    M1 : <НЕКОТОРЫЕ ДЕЙСТВИЯ>
    RESERVE(1 , VAR R1);
    <ИСПОЛЬЗОВАТЬ ЕДИНИЦУ РЕСУРСА ИЗ R С НОМЕРОМ,
      РАВНЫМ ЗНАЧЕНИЮ R1>
    RELEASE(R1);
    GOTO M1;
        END;
. . .

ПРОЦЕСС MN:     BEGIN
    VAR RN:RES;
    M1 : <НЕКОТОРЫЕ ДЕЙСТВИЯ>
    RESERVE(N , VAR RN);
    <ИСПОЛЬЗОВАТЬ ЕДИНИЦУ РЕСУРСА ИЗ R С НОМЕРОМ,
      РАВНЫМ ЗНАЧЕНИЮ RN>
    RELEASE(R1);
    GOTO MN;
        END;
PAREND;
END.
```

В рассматриваемом примере под критическим ресурсом понимается не ресурс R, который даже не описан, а некоторая совокупность информации, описывающая состояние ресурса и системы процессов, выдавших заявки на получение ресурсов. Для этого объявлена переменная INF, которую можно отнести к типу запись, состоящую из трёх поименованных элементов разного типа. Элемент с именем СВОБОД_РЕСУРС – это переменная типа SEQUENCE (последовательность), состоящая из непоименованных элементов типа RES. Эта переменная предназначена для хранения информации о нераспределённых в текущий момент единицах ресурсов. Все единицы ресурса пронумерованы от 1 до M. Такие элементы можно обрабатывать с помощью примитивов GET и PUT.

GET присваивает значение переменной (1-ый аргумент) одному из элементов переменной типа SEQUENCE по правилу FIFO;

PUT – обратная по смыслу операции GET. При выполнении PUT по правилу FIFO выбирается элемент из переменной типа SEQUENCE и присваивается переменной, указанной первым аргументом.

Второе поле записи – ТРЕБОВАНИЕ. Это переменная типа QUEUE (очередь). Она содержит непоименованные элементы, каждый из которых может принимать значения от 1 до N, где N – максимальное число процессов, которые могут претендовать на использование ресурсов.

Элемент ТРЕБОВАНИЕ используется для хранения заявок, поступающих от процессов на приобретение единиц ресурсов. Заявка – это номер процесса. Все процессы пронумерованы от 1 до N, причём каждый процесс знает о своём номере. Занесение номера процесса в очередь и извлечение номера процесса из очереди осуществляется с помощью примитивов ENTER и REMOVE. Занесение и извлечение необязательно производятся по правилу FIFO, а в соответствие с некоторой дисциплиной обслуживания.

В отношении переменной ТРЕБОВАНИЕ используется процедура EMPTY, значение которой всегда будет истинным при отсутствии элементов в очереди ТРЕБОВАНИЕ.

Элемент ОЧ_СОБЫТИЙ – это массив событийных переменных, каждая из которых указывает явно на один и тот же ресурс. Процессу с номером i соответствует i-ый элемент этого массива.

Для упрощения решения задачи используются процедуры RESERVE (для выделения единицы ресурса) и RELEASE (для возврата использованного ресурса в переменную СВОБОД_РЕСУРС).

При обращении к процедуре RESERVE процесс передаёт в качестве параметра собственный номер и имя переменной, куда будет помещён номер единицы ресурса, выделенной для использования.

Процедура освобождения RELEASE требует один параметр, и в этом параметре указывается номер той единицы ресурса, которая возвращается процессом в состав общего ресурса. Предполагается, что процесс может возвращать только те единицы ресурса, которые он захватил.

В каждой процедуре имеется критическая область в отношении ресурса INF, и только в её пределах происходит обращение к ресурсам.

При необходимости получить ресурс процесс с соответствующим номером обращается к процедуре RESERVE, причём такое обращение может сделать только один их процессов. Остальные по правилу взаимного исключения переводятся в состояние ожидания относительно критического ресурса INF.

Если процесс, который вошёл в критическую область, при выполнении процедуры резервирования обнаружил, что ни одной свободной единицы ресурса нет, то он временно покидает критическую область с помощью примитива AWAIT и переходит в состояние ожидания на собственной событийной переменной в массиве ОЧ_СОБЫТИЙ.

Если процесс, находясь в критической области, обнаружил хотя бы одну свободную единицу ресурса, он узнаёт её номер и производит её захват. При этом корректируется значение переменной СВОБОД_РЕСУРС. Полученной единицей ресурса процесс владеет монопольно, после чего осуществляется выход из критической области, и, следовательно, в неё может войти какой-либо из других процессов.

При выполнении процессом процедуры освобождения указанная им единица ресурса становится свободной, и соответствующая информация заносится в переменную СВОБОД_РЕСУРС. В ходе выполнения этой процедуры также проверяется необходимость реактивации процессов, которые временно вышли из критической области. Эта проверка проводится путём анализа переменной ТРЕБОВАНИЕ, и если в этой переменной имеются запросы, то осуществляется перевод одного из процессов из переменной ОЧ_СОБЫТИЙ в главную очередь.

Предложенный способ обращения к критическому ресурсу путём использования процедур в тексте алгоритма является методологическим приёмом, который позволяет назвать такие средства синхронизации мониторoподобными. Здесь увеличена степень локализации действий относительно критического ресурса INF. Такой подход позволяет разделить процесс проектирования алгоритма на два класса. В одном проектируются средства доступа к ресурсам или система распределения ресурса, а второй – это непосредственно разработка программы, реализующей определённую задачу.

Таким образом, создаётся централизованная схема распределения критических ресурсов. Такая схема называется мониторoподобной.

ТЕМА 4

4.1 Ресурсы

4.1.1 Распределение ресурсов. Проблема тупиков

Рассматривается логическая задача, которая возникает при взаимодействии различных процессов, когда они должны делить ресурсы.

Под процессами понимаются программы, описывающие некоторый вычислительный процесс, выполняемый ЭВМ. Выполнение такого вычислительного процесса требует времени, в течение которого в памяти ЭВМ хранится информация.

О процессах известно следующее.

1. Их требования к объему памяти не будут превышать определенного предела.

2. Каждый вычислительный процесс завершится при условии, что требуемый процессу объем памяти будет предоставлен в его распоряжение. Завершение вычислительного процесса будет означать, что его требование к памяти уменьшилось до нуля.

Имеющаяся память поделена на страницы фиксированного размера, эквивалентные с точки зрения программы.

Фактическое требование нужного процессу объема памяти является функцией времени, то есть изменяется по мере протекания процесса, но не превышает заранее заданную границу. Отдельные процессы запрашивают и выделяют память единицами в одну страницу. Однажды начатый процесс получает возможность рано или поздно завершиться, и исключается ситуация, когда процесс может быть уничтожен в ходе выполнения, напрасно истратив свои ресурсы.

Если на вычислительной машине выполняется один процесс, или они последовательно следуют один за другим, единственным условием является то, чтобы запрашиваемый процессом объём памяти не превышал доступный объём памяти вычислительной машины. Если параллельно развиваются несколько процессов, то могут возникнуть проблемы с выделением им ресурсов и надо предусмотреть выделение памяти таким образом, чтобы все начатые процессы смогли завершить свое выполнение.

Ситуация, когда какой-либо из процессов может быть завершён лишь при условии уничтожения какого-либо другого процесса, называется «смертельными объятиями» или тупиком.

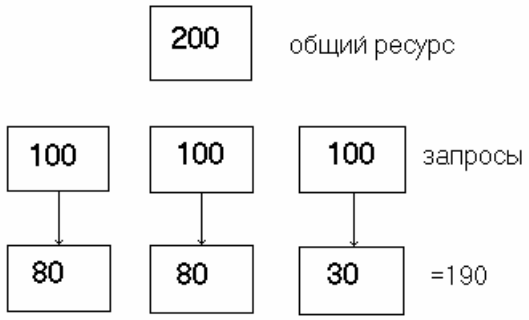


Рис. 4.1. Тупиковая ситуация

Ситуация, приведенная выше, является тупиковой. Из этой ситуации невозможно выйти без уничтожения какого-либо из процессов.

Решаемая проблема состоит в том, как избежать попадания в тупик, не накладывая слишком больших ограничений.

4.1.2 Алгоритм банкира

Банкир обладает конечным капиталом, например, талерами. Он решает принимать клиентов, которые могут занимать у него талеры на следующих условиях:

1. Клиент делает заем для совершения сделки, которая будет завершена за определенный промежуток времени.
2. Клиент должен указать максимальное количество талеров для этой сделки.
3. Пока заем не превысит заранее установленную потребность, клиент может увеличивать или уменьшать свой заем.
4. Клиент, который просит увеличить свой текущий заем, без недовольства воспринимает ответ о том, что необходимо подождать с получением очередного талера, но через некоторое время талер будет обязательно выдан.
5. Гарантия для клиента, что такой момент наступит, основана на предусмотрительности банкира и на том факте, что остальные клиенты работают по таким же правилам.

Основными вопросами при решении такой задачи являются:

1. При каких условиях банкир может заключить контракт с новым клиентом?
2. При каких условиях банкир может выплатить (следующий) запрашиваемый талер клиенту, не опасаясь попасть в тупик?

Ответ на первый вопрос достаточно прост: банкир может принять любого клиента на обслуживание, чья максимальная потребность не превышает капитал банкира. Ответ на второй вопрос достаточно сложный. Сначала нужно формализовать задачу.

потребность[i] <= капитал, для всех i.
0 <= заем[i] <= потребность[i], для всех i.
требование[i] = потребность[i] - заем[i], для всех i.
наличные = капитал - СУММА_по_i заем[i].
0 <= наличные <= капитал.

В такой ситуации алгоритм принятия решения о том, будет ли безопасной выдача следующего талера выглядит след образом:

```
integer Св_Деньги; boolean Безопасно;
boolean array Завершение_под_сомнением [1..N];
    Св_Деньги := наличные;
    for i := 1 step 1 until N do Завершение_под_сомнением[i] := true;
    L: for i :=1 step 1 until N do begin
                        if  ( (Завершение_под_сомнением [i]) and
        (Требован[i] <= Св_Деньги) ) then begin
                        Завершение_под_сомнением [i] := false;
                        Св_Деньги := Св_Деньги + Заем[i];
                        goto L;
                                    end;
                            end;
  if  (Св_Деньги = капитал) then Безопасно := true
                          else Безопасно := false;
```

Проверка возможности выплаты, то есть положение Безопасно, означает, могут ли с гарантией быть завершены все сделки. Алгоритм начинается с проверки, имеет ли, по крайней мере, один клиент требование, не превышающее наличные деньги. Если это так, то этот клиент может завершить свою сделку, и далее исследуются оставшиеся клиенты с учетом того, что первый клиент завершил свою сделку и возвратил свой заём полностью.

Безопасность положения означает, что могут быть закончены все сделки, то есть банкир видит способ получения обратно всех своих денег.

В полном варианте алгоритма банкира эта ситуация должна быть удовлетворена для всех клиентов, принятых на обслуживание и если после завершения цикла, отмеченного меткой L окажется, что капитал банкира полностью восстановлен, то ситуация считается безопасной, в противном случае она определяется как тупиковая и, следовательно, удовлетворять запрос клиента не представляется возможным.

Если более глубоко анализировать эту проблему, то можно показать, что решение о выделении следующего запрашиваемого талера клиенту будет принято тогда, когда хотя бы для одного клиента выполниться условие ((Завершение_под_сомнением[i]) and (Треб[i]<=Св_Деньги)) и это говорит о безопасности ситуации и проверку можно приостановить.

4.1.3 Применение алгоритма банкира

Каждому талеру можно поставить в соответствие жесткий диск или какое-либо устройство. Тогда заем талера будет означать разрешение на использование одного из дисков.

```
begin
    integer array Заем, Требование, Клиент_Сем,
            Клиент_Пер, Номер_Талера,
            Возвращенные_Талеры[1..N];
            integer array Талер_Сем, Талер_Пер, Номер_Клиента[1..M];
     integer Взаимн_искл, наличные, k;
     boolean procedure Попытка_выдать_талер_клиенту (integer j);
     begin
            if (Клиент_Пер[j]=1) then begin
                                    integer i, Своб_Деньги;
                                    boolean array Заверш_под_сомн[1..N];
                                    Своб_Деньги :=наличные -1;
                                    Требование[j]:=Требование[j]-1;
                                    Заем[j]:=Заем[j]+1;
                                    for i:=1 step 1 until N do
                                                Завершен_под_сомн[j]:=true;
                    L0: for i:=1 step 1 until N do begin
                            if  (Завершен_под_сомн[j] and
                            (Требование[i]=<Своб_Деньги) ) then begin
                                    if (i<>j) then begin
                                                Заверш_под_сомн[i]:=false;
                                              Своб_Деньги:=Своб_Деньги+Заем[i];
                                                goto L0;
                                                      end;
                                    else begin
                                            i:=0;
                                            L1:  i:=i+1;
                                            if (Талер_Пер[i] = 0) then goto L1;
                                            Номер_Талера[j]:=i;
                                            Номер_Клиента[i]:=j;
                                            Клиент_Пер[j]:=0;
                                            Талер_Пер[i]:=0;
                                            Наличные:=Наличные-1;
                                    Попытка_Выдать_Талер_Клиенту:=true;
                                            V (Клиент_Сем[j]);
                                            V (Талер_Сем[j]);
                                            goto L2;
```

```
                                    end;
                                                end;
                                                            end;
                                    Требование[j]:=Требование[j]+1;
                                    Заем[j]:=Заем[j]-1;
                                end;
                                Попытка_Выдать_Талер_Клиенту:=false;
                        L2: end;  /* процедуры */

Взаимн_искл:=1; Наличные :=M;
for k:=1 step 1 until N do begin
                                Заем[k]:=0;
                                Клиент_Сем[k]:=0;
                                Клиент_Пер[k]:=0;
                                Требование[k]:=Потребность[k];
                                Возвращенные_Талеры[k]:=Потребность[k];
                                end;
for k:=1 step 1 until M do begin
                                Талер_Сем[k]:=0;
                                Талер_Перем[k]:=1;
                                 end;
        parbegin
                Клиент 1: begin ... end;
                …
                Клиент i: begin ...
                        P(Возвращенные_Талеры[i]);
                        P(Взаимн_искл);
                        Клиент_Пер[i] := 1;
                        Попытка_выдать_талер_клиенту(i);
                        V(Взаимн_искл);
                        P(Клиент_Сем[i]);
                        …
                end;
                Талер 1: begin ... end;
              …
                Талер m: begin integer h;
                   …
                        начало: P(Талер_Сем[m]);
                           P(Взаимн_искл);
Требование[Номер_Клиента[m]] := Требование[Номер_Клиента[m]] - 1;
                        Талер_Перем[m] := 1;
                        наличные := наличные + 1;
                        V(Возвращенные_Талеры[Номер_Клиента[m]]);
                        for h:=1 step 1 until N do begin
```

```
                    if (Попытка_выдать_талер_клиенту(h))
                    then goto выход;
                end;
                выход:V(Взаимн_искл);
                goto начало
                …
            end;
        parend;
end;
```

Сущность: есть клиент и есть талер. Каждый клиент имеет переменную состояния Клиент_Пер. Если Клиент_Пер = 1 – это означает: “хочу получить заем”. Иначе Клиент_Пер = 0. Каждый талер имеет переменную состояния Талер_Пер. Если Талер_Пер = 1 – это означает: “нахожусь среди свободного капитала”. Клиенты пронумерованы от 1 до N, а талеры от 1 до M. С каждым клиентом связывается переменная Номер_Талера, значение которой после очередной выдачи талера клиенту определяет номер только что выделенного талера. В свою очередь с каждым талером связана переменная Номер_Клиента , значение которой указывает клиента, которому выдан этот талер. Имеются семафоры Клиент_Сем, и Талер_Сем. Фактически возврат талера заканчивается после того, как тот действительно присоединится к наличному капиталу банкира: об этом талер будет сообщать клиенту с помощью общего семафора клиента "Возвращенные_талеры". Значение булевой процедуры "Попытка_выдать_талер_клиенту" говорит о том, удовлетворен ли задержанный запрос на талер. В программе для талера используется тот факт, что возврат талера может теперь привести к удовлетворению единственного задержанного запроса на талер (если банкир распоряжается услугами более чем одного вида, то последнее свойство уже не будет выполняться).

ТЕМА 5

5.1 Память. Управление памятью

5.1.1 Требования к управлению памятью

Система управления памятью должна обеспечивать решение следующих проблем.

1. Перемещение. Для максимальной загрузки процессора необходимо иметь набор готовых к выполнению процессов. Для этого требуется реализовать возможные загрузки и выгрузки активных процессов из основной памяти и обратно. При этом весьма желательно, чтобы при таких перезагрузках программа могла перемещаться в различные участки памяти. ОС должна знать местоположение управляющей информации процессов и стека выполнения, а также точку входа в исполняемый код. Т.к. управлением памятью занимается ОС, а также ОС управляет загрузкой процессов в память, соответствующие адреса она получает автоматически. Однако в программе встречаются команды перехода и ветвления, и поэтому должна быть реализована возможность преобразования адресных ссылок в коде программы в реальные физические адреса, соответствующие текущему положению программы в ОП.

2. Защита. Каждый процесс должен быть защищен от воздействия других процессов. Это означает, что из программного кода одного процесса нельзя обращаться к памяти, отведенной под другой процесс. Но обеспечение перемещаемости программы усложняет организацию механизма защиты. Т.к. в программе невозможно предусмотреть контроль за всеми адресными обращениями, то такой контроль возможен только на аппаратном уровне.

3. Совместное использование. Предусматривает возможность обращения нескольких процессов к ОП. Если несколько процессов выполняют один и тот же машинный код, то целесообразно разрешить каждому из процессов работать с одной и той же копией этого кода, а не создавать каждому процессу копию. Система управления памятью должна обеспечить управляемый доступ к разделенным областям памяти, не ослабляя при этом защиту памяти.

4. Логическая организация памяти. И основная память и вторичная (внешняя) организованы как линейное адресное пространство, где элементом адресации является байт или слово. Такая организация отражает систему аппаратного обеспечения, но не соответствует страничной организации программ. Большинство программ организуется из модулей, часть из которых неизменна (используются только для чтения или исполнения), а другие содержат данные, которые могут изменяться.

Если ОС или аппаратное обеспечение ЭВМ могут работать с программами, представленными в виде модулей, то это обеспечивает ряд преимуществ:

- модули могут быть созданы и откомпилированы независимо друг от друга, при этом ссылки из одного модуля во второй разрешается системой во время работы программы;
- разные модули могут получать разные модули защиты;
- возможно применение механизмов совместного использования модулей различными процессами.

Наиболее подходящим способом для решения таких задач является сегментация.

5. Физическая организация памяти. Память в ЭВМ разделяется как минимум на два уровня- основную и вторичную. Основная обеспечивает быстрый доступ по достаточно высокой цене, она энергозависима, следовательно, не обеспечивает долговременного хранения. Вторичная память медленнее и дешевле, и энергонезависима. Следовательно, она может использоваться для долговременного хранения данных и программ. Основная память применяется для хранения программ, используемых в текущее время.

Одной из основных задач ОС является организация потоков информации между основной и вторичной памятью.

5.1.2 Схемы распределения памяти

Основной функцией ОС по управлению памятью является размещение программы в основной памяти для её выполнения процессором. В современных ОС решение этой задачи предполагает использование сложной схемы, называемой ***виртуальной памятью***.

Известно несколько способов распределения памяти:

1. фиксированное распределение. ОП разделяется на ряд статических разделов во время генерации системы. Процесс может быть загружен в раздел равного или большего размера. Положительная сторона- простота реализации и малые системные затраты. Отрицательная сторона – неэффективное использование памяти из-за внутренней фрагментации и фиксированного максимального количества процессов.
2. Динамическое распределение. Разделы создаются динамически, каждый процесс загружается в раздел необходимого раздела. Достоинство- отсутствие внутренней фрагментации, более эффективное использование ОП. Недостаток – существенные затраты процессора на противодействие внешней фрагментации и проведения уплотнения памяти. При выделении памяти, таким образом, применяются три основных алгоритма: наилучший подходящий, первый подходящий, следующий подходящий.
3. Простая страничная организация. ОП разделена на ряд кадров равного размера. Каждый процесс распределен на некоторое количество страниц равного размера, такой же длины, что и кадры памяти. Процесс загружается путем загрузки всех его страниц. Достоинство- отсутствие внешней фрагментации. Недостаток – небольшая внутренняя фрагментация.
4. Простая сегментация. Каждый процесс распределен на ряд сегментов. Процесс загружается путем загрузки всех своих сегментов в динамические, не обязательно смежные, разделы. Достоинство- отсутствие внутренней фрагментации. Недостаток – проблемы с внешней фрагментацией.
5. Страничная организация виртуальной памяти. Подобна простой страничной организации, но не требуется загружать все страницы процесса. Необ-

ходимые нерезидентные страницы автоматически подгружаются в память. Достоинства – отсутствие внешней фрагментации, более высокая степень многозадачности, большое виртуальное адресное пространство. Недостаток – значительные затраты на управление виртуальной памятью.

6. Сегментация виртуальной памяти. Подобна простой сегментации, но не требуется загружать все сегменты процесса. Необходимые нерезидентные сегменты автоматически подгружаются в память. Достоинства - отсутствие внутренней фрагментации, более высокая степень многозадачности, большое виртуальное адресное пространство, поддержка защиты и совместного использования. Недостаток – затраты на управление сложной виртуальной памятью.

5.1.3 Система двойников при распределении памяти

Фиксированное распределение памяти ограничивает количество активных процессов и неэффективно использует память при несоответствии между размерами разделов и процессами. Динамическое распределение реализуется более сложно и включает затраты на уплотнение памяти. Решением в этом плане является ***система двойников***. В ней память распределяется блоками размером 2^k, где 2^l-минимальный размер выделяемого блока, а 2^u-максимальный размер(вся доступная распределенная память). Вначале все доступное для распределения адресное пространство рассматривается как единый блок размером 2^u.

Если запрашивается блок размером s, таким что $2^{u-1} \le s \le 2^u$, то выделяется весь блок памяти, в противном случае, блок будет разделен на два одинаковых подблока (двойника), размерами 2^{u-1}. Если $2^{u-2} \le s \le 2^{u-1}$, то выделяется блок 2^{u-1}, иначе повторяется. Процесс деления продолжается до тех пор, пока не будет сгенерирован наименьший блок, размер которого не меньше s. Система двойников всегда ведет список доступных блоков для каждого размера 2^i, где $i \in (l,u)$. Блок может быть удален из списка путем разделения его пополам и внесения двух свободных блоков в список 2^i. Когда пара свободных блоков в списке 2^i оказывается освобожденной, они удаляются из этого списка и объединяются в единый блок в списке 2^{i+1}.

Такой подход оказался достаточно эффективным и он применен в Linux.

ТЕМА 6

6.1 Организация виртуальной памяти

6.1.1 Структуризация адресного пространства виртуальной памяти

Сложность управления виртуальной памяти объясняется необходимостью устанавливать связи между аппаратным и программным обеспечением ОС. Ключевыми моментами в таком управлении являются следующие характеристики страничной организации и сегментации.

1. Все обращения к памяти в рамках процесса представляют собой логические адреса, которые динамически транслируются в физические во время выполнения. Процесс может находиться в разных местах оперативной памяти (ОП). Логический адрес представляет собой ссылку на ячейку памяти, не зависимо от расположения данных в памяти. Относительный адрес определяется положением от некоторой известной точки, обычно началом программы, и представляет собой частный случай логического адреса. Физический адрес представляет собой действительное расположение ячейки в ОП.

2. Процесс может быть разбит на ряд частей, которые не могут располагаться в ОП единым непрерывным блоком. Это обеспечивается за счет динамической трансляции адресов и используемой таблицы страниц и сегментов.

В результате этого достигаются следующие свойства:

– в ОП может поддерживаться большое количество процессов.

– процесс может быть больше, чем вся ОП.

Процесс выполняется в ОП, которая поэтому называется реальной, но программист имеет дело с потенциально гораздо большей памятью, называемой виртуальной и которая выделяется во внешней памяти. При управлении виртуальной памятью решаются 4 задачи:

- размещение;
- перемещение;
- преобразование;
- замещение.

6.1.2 Задачи управления виртуальной памятью

6.1.2.1 Задача размещения

Её сущность состоит в выборе в адресном пространстве ОП сегментов или страниц, на которые будут отображаться сегменты или страницы виртуального адресного пространства.

При решении этой задачи стараются выбрать максимально простой алгоритм распределения памяти. Алгоритм строится таким образом, чтобы умень-

шить частоту его использования. Для учета свободных и распределенных страниц можно использовать двоичного вектора. Число двоичных разрядов вектора равно числу страниц ОП. Двоичные разряды нумеруются в той же последовательности, что и страницы. Если в разряде храниться 0 - это значит, что страница свободна и 1 - страница распределена. Если свободных страниц нет, а запрос на страницу поступает, то решается задача замещения и какую-то страницу перемещают во внешнюю память.

6.1.2.2 Задача перемещения

Состоит в том, к произвести выборку информации, которая хранится во внешней виртуальной памяти, для переноса в ОП. Используется 2 основных варианта выборки информации во внешней памяти:

– по требованию;

– предварительно.

При выборке по требованию страница передается в ОП только тогда, когда выполняется обращение к ячейке памяти, расположенной на этой странице. Когда процесс только запускается, число обращений к внешним страницам, распределенных во внешней памяти, достаточно велико, но постепенно начинает срабатывать принцип локализации, и все большее число обращений начинает происходить к уж загруженным страницам.

В случае предварительной выборки, загружается не только страница, вызвавшая прерывание. Если страница процесса расположена последовательно во внешней памяти, то бывает эффективной загрузка нескольких страниц за один раз. Предварительная выборка ориентирована на физическую организацию внешней памяти, и здесь учитываются такие факторы, как время поиска страницы и задержки, связанные с позиционированием устройств чтения.

6.1.2.3 Задача преобразования

Заключается в нахождении абсолютного физического адреса основной памяти по его виртуальному адрес

6.1.2.4 Задача замещения

Целью этой задачи является выбор среди пространства ОП той страницы, которую следует переместить во внешнюю память. Задача решается, когда обнаруживается отсутствие страницы в ОП, а вместе с тем следует запрос на размещение новых страниц.

Первый, наиболее простой подход, заключается в остановке процесса, потребовавшего страницу ОП в ситуации, когда вся память распределена. В этом случае управление передается другому процессов, у которого нет требований на дополнительную память, а в отношении приостановленного процесса либо не принимается никаких действий, либо все его страницы выгружаются во

внешнюю память. Недостаток этого подхода- дискриминации подвергается тот процесс, в ходе выполнения которого возникла потребность замещения.

Идеальная стратегия замещения: должна быть замещена та страница, к которой дольше всего не будет обращений в будущем.

Существуют следующие стратегии:

- для замещения выбирается страница случайным образом;
- выбирается страница, которая дольше всего была в ОП;
- FIFO (удаляется та страница, которая раньше всех была распределена какому-либо процессу). Фактически, это реализация предыдущей стратегии, но она может быть реализована для различных процессов;
- алгоритм удаления дольше всех неиспользовавшейся страницы. Если долго обращения к странице не было, следовательно, в будущем тоже не предвидится.

В простейшем случае, с каждой страницей для этой стратегии связывается бит использования. Этот бит установлен в 1 при обращении к странице, а способ сброса бита в 0 и определяет способы реализации данной стратегии.

В настоящее время используется варианты «часового» алгоритма. Всего n ячеек.

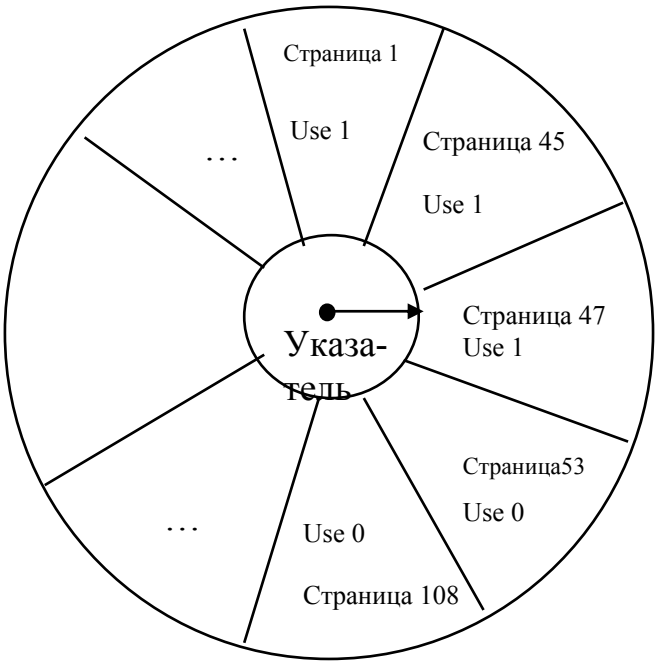


Рис. 6.1. Иллюстрация “Часового” алгоритма

Имеется циклический буфер размерности n, в каждом элементе хранится номер страницы и бит использования. Бит использования установлен в 1, когда к странице произведено обращение, если при первой загрузке страницы в ОП. Указатель буфера указывает на последнюю замещенную страницу. Когда возникает вопрос решить задачу замещения, указатель перемещается на следующий элемент буфера. Если бит использования установлен в 0, то производится замещение соответствующей страницы. Если же окажется, что бит использования =1, то страница не замещается, бит использования устанавливается в 0, а указатель перемещается на следующий элемент буфера. Перемещение указате-

ля осуществляется до тех пор, пока не будет обнаружена страница с 0 битом использования.

Повысить эффективность часового алгоритма можно путем увеличения количества используемых при его работе битов. Практически, во всех системах страничной организации со страницей связывается бит модификации. Этот бит указывает, что страница не может быть замещена до тех пор, пока её содержимое не будет перепсано во внешнюю память. Соответственно может быть 4 комбинации битов использования и битов модификации:

N m	
0 0	*n:0-давно использован*
1 0	*1-недавно использован*
0 1	*m:0- не модифицирован*
1 1	*1- модифицирован*

Часовой алгоритм выглядит следующим образом:

1. Сканируем буфер, начиная с текущего положения. В процессе сканирования бит использования не изменяется. Первая страница состояния (0,0) замещается.

2. Если такой страницы нет, то ищем страницу с параметром (0,1). Если такая страница найдена, она замещается. В процессе выполнения данного шага у всех проанализированных страниц сбрасывается бит использования.

3. Если такой страницы нет, значит, у всех страниц будет сброшен бит использования, указатель буфера вернется в исходное положение, затем повторяем шаг 1 и , при необходимости, 2.

Явление пробуксовки наблюдается тогда, когда ОП имеет небольшие размеры, а программы велики по размеру. В этом случае может возникнуть ситуация частого замещения страниц, и большая часть процессорного времени тратится на выполнение служебных функций. Следовательно, резко замедляется выполнение пользовательских программ.

ТЕМА 7

7.1 Планирование в операционных системах

В многозадачных системах в основной памяти одновременно содержится код нескольких процессов. В работе каждого процесса периоды использования процессора чередуются с ожиданием завершения выполнения операций ввода-вывода или некоторых внешних событий. Процессор (или процессоры) занят выполнением одного процесса, в то время как остальные находятся в состоянии ожидания.

Ключом к многозадачности является планирование. Обычно используются четыре типа планирования (табл. 7.1)

Таблица 7.1 - Типы планирования

Долгосрочное планирование	Решение о добавлении процесса в пул выполняемых процессов
Среднесрочное планирование	Решение о добавлении процесса к числу процессов, полностью или частично размещённых в памяти
Краткосрочное планирование	Решение о том, какой из доступных процессов будет выполнятся процессором
Планирование ввода-вывода	Решение о том, какой из запросов процессоров на операции ввода-вывода будет обработан свободным устройством ввода-вывода

7.1 Типы планирования процессора

Цель планирования процессора состоит в распределении во времени процессов, выполняемых процессором (или процессорами) таким образом, чтобы удовлетворять требованиям системы, таким, как время отклика, пропускная способность и эффективность работы процессора. Во многих системах планирование разбивается на три отдельные функции — долгосрочного, среднесрочного краткосрочного планирования. Их названия соответствуют временным масштабам выполнения этих функций.

На рис. 7.1 функции планирования привязаны к диаграмме переходов стояния процесса. Долгосрочное планирование осуществляется при создании нового процесса и представляет собой решение о добавлении нового процесса к множеству активных в настоящий момент процессов. Среднесрочное планирование является частью свопинга и представляет собой решение о добавлении

процесса к множеству по крайней мере частично расположенных в основной памяти (и, следовательно, доступных для выполнения) процессов. Краткосрочное планирование является решением о том, какой из готовых к выполнению процессов будет выполняться следующим. На рис. 7.2 диаграмма переходов реорганизовала таким образом, чтобы показать вложенность функций планирования.

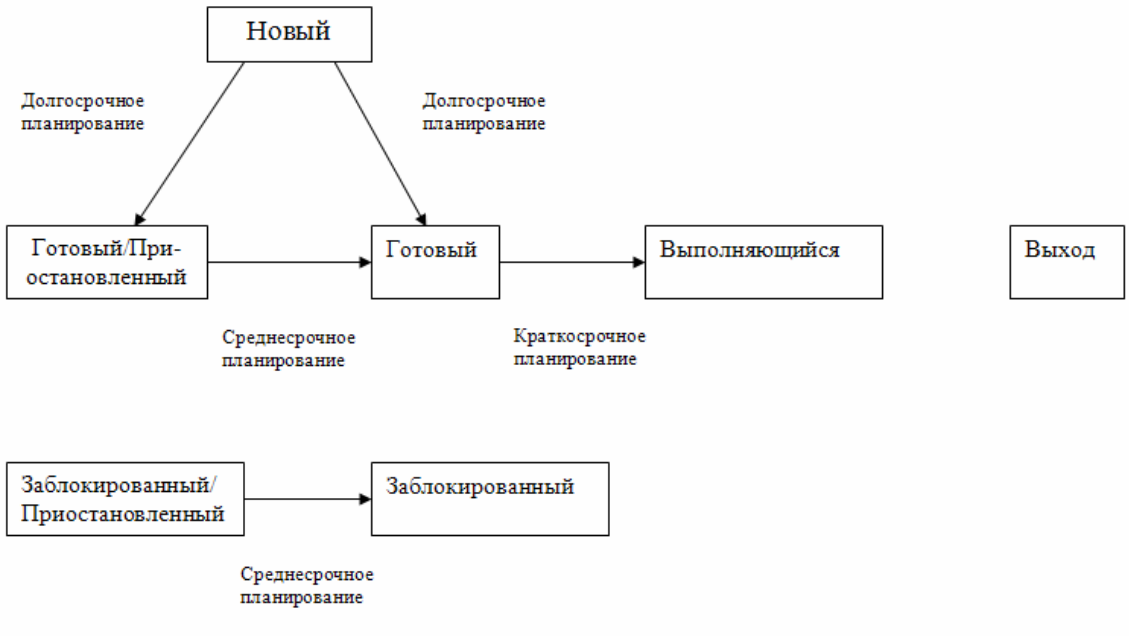


Рис. 7.1 Место планирования в диаграмме переходов состояний процесса

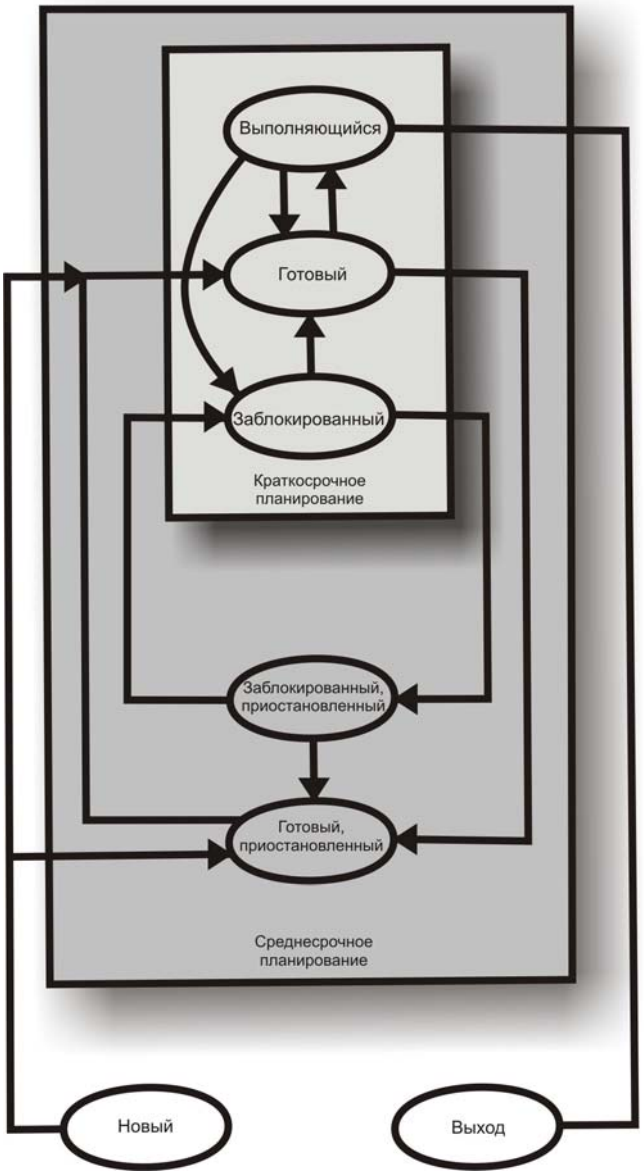


Рис 7.2. Уровни планирования

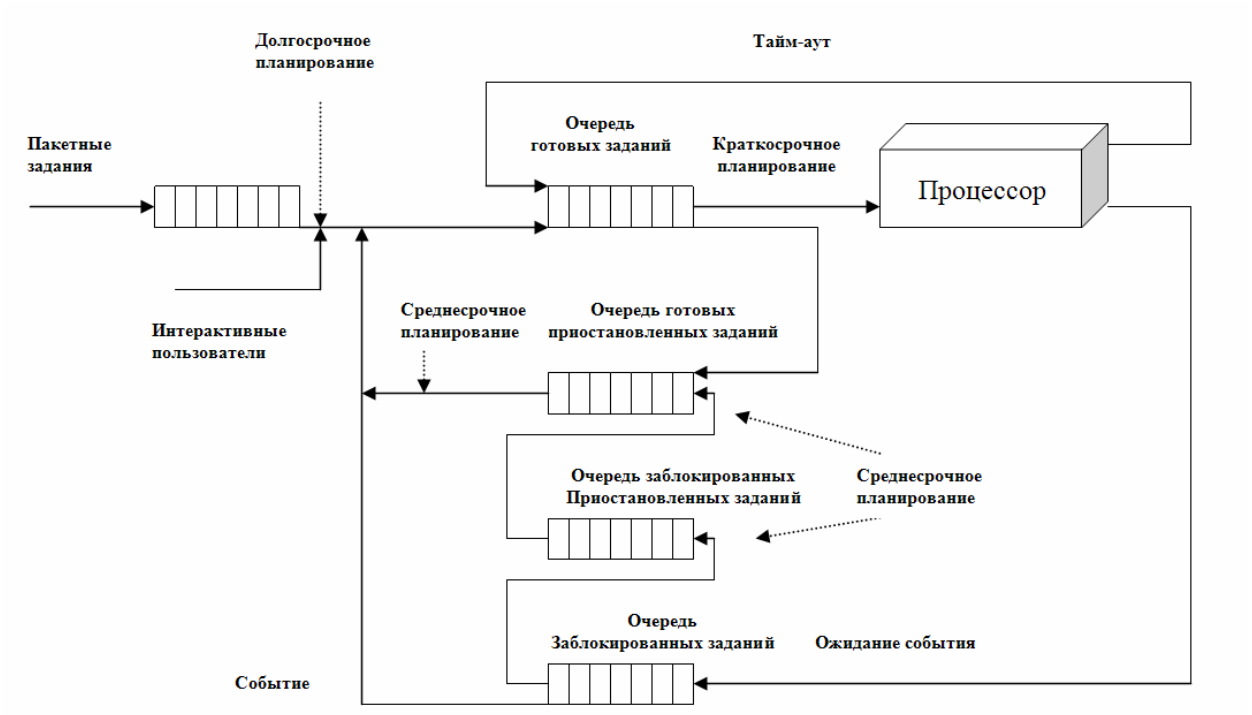


Рис. 7.3. Диаграмма планирования с участием очередей

Долгосрочное планирование.

Долгосрочное: планирование указывает, какие программы допускаются к выполнению системой, и тем самым определяет степень многозадачности. Будучи допущенным к выполнению, задание (или пользовательская программа) становится процессом, который добавляется в очередь для краткосрочного планирования. В некоторых системах вновь созданный процесс добавляется к очереди среднесрочного планировщика, будучи целиком сброшенным на диск.

В пакетных системах (или в пакетной части операционной системы общего назначения) новое задание направляется на диск и хранится в очереди пакетных заданий, а долгосрочный планировщик по возможности создает процессы для заданий из очереди. В такой ситуации планировщик должен принять решение, во-первых, о том, способна ли операционная система работать с дополнительными процессами, а во-вторых, о том, какое именно задание (или задания) следует превратить процесс (процессы).

Решение о том, когда следует создавать новый процесс, в общем определяется желаемым уровнем многозадачности. Чем больше процессов будет создано, тем меньший процент времени будет тратиться на выполнение каждого из них (поскольку в борьбе за одно и то же время конкурирует большое количество процессов). Таким образом, долгосрочный планировщик может ограничить степень многозадачности, с тем чтобы обеспечить удовлетворительный уровень обслуживания текущего множества процессов. Каждый раз при завершении за-

дания планировщик решает, следует ли добавить в систему один или несколько новых процессов. Кроме того, долгосрочный планировщик может быть вызван в случае, когда относительное время простоя процессора превышает некоторый предопределенный порог.

Решение о том, какое из заданий должно быть добавлено в систему, может основываться на простейшем принципе «первым поступил − первым обслужен»; кроме того, для управления производительностью системы может использоваться и специальный инструментарий. Используемые в последнем случае критерии могут включать приоритет заданий, ожидаемое время выполнения и требования для работы устройств ввода-вывода. Например, если заранее доступна детальная информация о процессах, планировщик может пытаться поддерживать в системе смесь из процессов, ориентированных на вычисления и загружающих процессор, и процессов с высокой интерактивностью ввода-вывода и малой загрузкой процессора. Принимаемое решение может также зависеть от того, какие именно ресурсы ввода-вывода будут запрашиваться процессом.

В случае использования интерактивных программ в системах с разделением времени запрос на запуск процесса может генерироваться действиями пользователя по подключению к системе. Пользователи не просто вносятся в очередь в ожидании, когда система обработает их запрос на подключение. Вместо этого операционная система принимает всех зарегистрированных пользователей до насыщения системы (пороговое значение которого определяется заранее). После достижения состояния насыщения на все запросы на вход в систему будет получено сообщение о заполненности системы и временном прекращении доступа к ней с предложением повторить операцию входа попозже.

Среднесрочное планирование.

Среднесрочное планирование является частью системы свопинга. Обычно решение о загрузке процесса в память принимается в зависимости от степени многозадачности; кроме того, в системе с отсутствием виртуальной памяти среднесрочное планирование также тесно связано с вопросами управления памятью. Таким образом, решение о загрузке процесса в память должно учитывать требования к памяти выгружаемого процесса.

Краткосрочное планирование.

Рассматривая частоту работы планировщика, можно сказать, что долгосрочное планирование выполняется сравнительно редко, среднесрочное — несколько чаще. Краткосрочный же планировщик, известный также как диспетчер (dispatcher), работает чаще всего, определяя, какой именно процесс будет выполняться следующим.

Краткосрочный планировщик вызывается при наступлении события, которое может приостановить текущий процесс или предоставить возможность прекратить выполнение данного процесса в пользу другого. Вот некоторые примеры таких событий:

- прерывание таймера;
- прерывания ввода-вывода;
- вызовы операционной системы;

- сигналы.

7.2 Алгоритмы плинирования

Критерии краткосрочного планирования

Основная цель краткосрочного планирования состоит в распределении прицессорного времени таким образом, чтобы оптимизировать один или несколько аспектов поведения системы. Вообще говоря, имеется множество критериев оценки различных стратегий планирования.

Наиболее распространенные критерии могут быть классифицированы в двух плоскостях. Во-первых, мы можем разделить их на пользовательские и системные. Пользовательские критерии связаны с поведением системы по отношению к отдельному пользователю или процессу. В качестве примера можно привести время отклика в интерактивной системе. Время отклика представляет собой интервал между передачей запроса и началом ответа на него. Его пользователь ощущает непосредственно, и, само собой, продолжительность интервала очень интересует его

Системные критерии ориентированы на эффективность и полноту использования процессора. В качестве примера можно привести пропускную способность, которая представляет собой скорость завершения процессов. Это, безусловно, эффективная мера производительности системы, которая должна быть максимизирована. Однако она в большей степени ориентирована на производительность системы, а не на обслуживание пользователя, так что и удовлетворять она будет системного администратора, а не пользователей системы.

В то время как пользовательские критерии важны почти для всех систем, системные критерии для однопользовательских систем не так значимы. В этом случае, пожалуй, достижение высокой эффективности использования процессора или высокая производительность не так существенны, как скорость ответа системы приложению пользователя.

Еще один способ разделения критериев — на те, которые связаны с производительностью, и те, которые с производительностью непосредственно не связаны. Ориентированные на производительность критерии выражаются числовыми значениями и обычно достаточно легко измеримы — примерами их могут служить время отклика и пропускная способность. Критерии, не связанные с производительностью непосредственно, либо качественны по своей природе, либо трудно поддаются измерениям и анализу. Примером такого критерия служит предсказуемость. Желательно, чтобы предоставляемые пользователю сервисы в разное время имели одни и те же характеристики, не зависящие от других задач, выполняемых в настоящее время системой. До некоторой степени этот критерий является измеримым − путём вычисления отклонений как функции от загрузки системы. Однако провести такие измерения оказывается вовсе не просто.

В табл. 7.2 приведены ключевые критерии планирования. Все они взаимозависимы, и достичь оптимального результата по каждому из них одновременно невозможно. Например, обеспечение удовлетворительного отклика может потребовать применения алгоритма с высокой частотой переключения процессов, что повысит накладные расходы и, соответственно, снизит пропускную способность системы. Следовательно, разработка стратегии планирования представляет собой поиск компромисса среди противоречивых требований; относительный вес каждого из критериев определяется природой и предназначением разрабатываемой системы.

В большинстве интерактивных операционных систем с одним пользователем или с разделяемым временем критичным требованием является время отклика.

Таблица 7.2. Критерии планирования

	Пользовательские, связанные с производительностью
Время оборота	Интервал времени между подачей процесса и его завершением. Включает время выполнения, а также время, затраченное на ожидание ресурсов, в том числе и процессора. Критерий вполне применим для пакетных заданий.
Время отклика	В интерактивных процессах это время, истекшее между подачей запроса и началом получения ответа на него. Зачастую процесс может начать вывод информации пользователю, ещё не окончив полной обработки запроса, так что описанный критерий − наиболее подходящий с точки зрения пользователя. Стратегия планирования должна пытаться сократить время получения ответа при максимализации количества интерактивных пользователей, время отклика для которых не выходит за заданные пределы.
Предельный срок	При указании предельного срока завершения процесса планирование должно подчинить ему все прочие цели максимализации количества процессов, завершающихся в срок.
	Пользовательские, иные
Предсказуемость	Данное задание должно выполняться примерно за одно и то же количество времени и с одной и той же стоимостью, независимо от загрузки системы. Большие вариации времени исполнения или времени отклика дезориентируют пользователей. Это явление может сигнализировать о больших колебаниях загрузки или о необходимости дополнительной настройки системы для устранения нестабильности ее работы.
	Системные, связанные с производительностью
Пропускная способность	Стратегия планирования должна пытаться максимизировать количество процессов, завершающихся за единицу времени, что является мерой количества выполненной системой работы. Очевидно, что эта величина зависит от средней продолжительности процесса; однако на нее влияет и используемая стратегия планирования.
Использование процессора	Этот показатель представляет собой процент времени, в течение которого процессор оказывается занят. Для дорогих совместно используемых систем этот критерий достаточно важен; в однопользовательских же и некоторых других системах (типа систем реального времени) этот критерий менее важен по сравнению с рядом других.
	Системные, иные
Беспристрастность	При отсутствии дополнительных указаний от пользователя или системы все процессы должны рассматриваться как равнозначные и ни один процесс не должен подвергнуться голоданию.
Использование приоритетов	Если процессам назначены приоритеты, стратегия планирования должна отдавать предпочтение процессам с более высоким приоритетом.
Баланс ресурсов	Стратегия планирования должна поддерживать занятость системных ресурсов. Предпочтение должно быть отдано процессу, который недостаточно использует важные ресурсы. Этот критерий включает использование долгосрочного и среднесрочного планирования.

Использование приоритетов

Во многих системах каждому процессу присвоен некоторый приоритет, и планировщик всегда должен среди процессов выбирать тот, у которого приоритет наибольший. На рис. 7.3 показано использование приоритетов. Вместо од-

ной очереди готовых к исполнению процессов у нас имеется их множество, упорядоченное по убыванию приоритета: RQ0, RQ1, ..., RQn, т.е.

Приоритет[RQ*i*]> *Приоритет*[RQ*j*] при *i*<*j*

При выборе процесса планировщик начинает с очереди процессов с наивысшим приоритетом (RQ0). Если в очереди имеются один или несколько процессов, процесс для работы выбирается с использованием некоторой стратегии планирования. Если очередь RQ0 пуста, рассматривается очередь RQ1 и т.д.

Одна из основных проблем в такой чисто приоритетной схеме планирования состоит в том, что процессы с низким приоритетом могут оказаться в состоянии голодания. Это будет происходить при постоянном поступлении новых готовых к выполнению процессов с высоким приоритетом. Если такое поведение нежелательно, приоритет процесса может снижаться при его выполнении.

Альтернативные стратегии планирования

В табл. 7.3 представлена некоторая информация о различных стратегиях планирования. Функция планирования определяет, какой из готовых к выполнению процессов будет выбран следующим для выполнения. Функция может быть основана на приоритете, требованиях к ресурсам или характеристиках выполнения процессов. В этом случае имеют значение три величины:

w − время, затраченное к этому моменту системой (ожидание и выполнение);

e − время, затраченное к этому моменту на выполнение;

s − общее время обслуживания, требующееся процессу, включая e (обычно эта величина оценивается или задаётся пользователем).

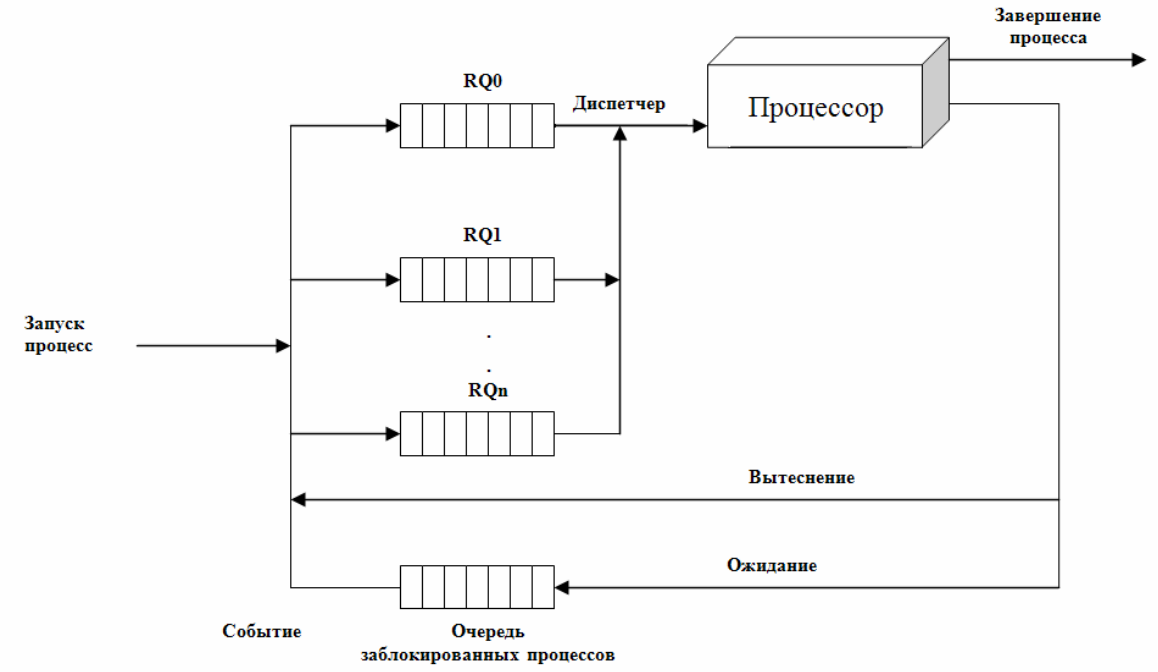


Рис 7.4. Планирование с учетом приоритетов

Например, выбор функции max[w] определяет стратегию "первым поступил - первым обслужен" (first-come-first-served — FCFS).

Режим решения определяет, в какие моменты времени выполняется функция выбора. Режимы решения подразделяются на две основные категории.

Невытесняющие. В этом случае находящийся в состоянии выполнения процесс продолжает выполнение до тех пор, пока он не завершится или пока не окажется в заблокированном состоянии ожидания завершения операции ввода-вывода или запроса некоторого системного сервиса.

Вытесняющие. Выполняющийся в настоящий момент процесс может быть прерван и переведен операционной системой в состояние готовности к выполнению. Решение о вытеснении может приниматься при запуске нового процесса по прерыванию, которое переводит заблокированный процесс в состояние готовности к выполнению, или периодически — на основе прерываний таймера.

Вытесняющие стратегии приводят к повышенным накладным расходам по сравнению с невытесняющими, но при этом обеспечивают лучший уровень обслуживания всего множества процессов, поскольку предотвращают монопольное использование процессора в течение продолжительного времени одним из процессов. Кроме того, использование эффективных механизмов переключения процессов (по возможности реализованное аппаратно) и большой объем основной памяти (для хранения в ней как можно большего количества процессов) позволяющих поддерживать относительно небольшую стоимость вытеснения.

Первым поступил − первым обслужен

Простейшая стратегия планирования "первым поступил — первым обслужен" (first-come-first-served − FCFS) известна также как схема "первым пришел— первым вышел", или схема строгой очередности. Как только процесс становится готовым к выполнению, он присоединяется к очереди готовых процессов. При прекращении выполнения текущего процесса для выполнения выбирается процесс, который находился в очереди дольше других.

Для однопроцессорных систем FCFS — не самая подходящая стратегия, но она часто комбинируется с использованием приоритетов. В этом случае планировщик поддерживает ряд очередей, по одной для каждого уровня приоритета, и работает с процессами в каждой очереди в соответствии со стратегией FCFS.

	Функция выбора	Режим решения	Пропускная способность	Время отклика	Накладные расходы	Влияние на процессы	Голодание
FCFS	max[w]	Невытесняющий	Не важна	Может быть большим, в особенности при больших отклонениях во времени исполнения про-	Минимальны	Плохо сказывается на коротких процессах и процессах с интенсивным вводом-	Отсутствует
Круговая	const	Вытесняющий	Может быть низкой при малом времени	Обеспечивает хорошее время отклика для коротких процессов	Минимальны	Беспристрастна	Отсутствует
SPN	min[s]	Невытесняющий	Высокая	Обеспечивает хорошее время отклика для коротких процессов	Могут быть высокими	Плохо сказывается на длинных процессах	Возможно
SRT	Min[s-e]	Вытесняющий (по решению)	Высокая	Обеспечивает хорошее время отклика	Могут быть высокими	Плохо сказываются на длинных процессах	Возможно
HRRN	min([w+s]/s)	Невытесняющий	Высокая	Обеспечивает хорошее время отклика	Могут быть высокими	Хороший баланс	Отсутствует
Со снижением приоритета	См.текст	Вытесняющий (по времени	Не важна	Не важно	Могут быть высокими	Может привести к предпочтению процессов с интенсивным вводом-выводом	Возможно

Круговое планирование

Очевидный путь повышения эффективности работы с короткими процессами в схеме FCFS — использование вытеснения на основе таймера. Простейшая стратегия, основанная на этой идее, — стратегия кругового (карусельного) планирования (round robin — RR). Таймер генерирует прерывания через определенные интервалы времени. При каждом прерывании исполняющийся в настоящий момент процесс помещается в очередь готовых к выполнению процессов, и начинает выполняться очередной процесс, выбираемый в соответствии со стратегией FCFS. Эта методика известна также как **квантование времени** (time slicing), поскольку перед тем как оказаться вытесненным, каждый процесс получает квант Івремени для выполнения.

При круговом планировании принципиальным становится вопрос о продолжительности кванта времени. При малом кванте времени короткие процессы будут относительно быстро проходить через систему, но при этом возрастают накладные расходы, связанные с обработкой прерывания и выполнением функций планирования. Следовательно, очень коротких квантов времени следует избегать. Одно из полезных правил в этом случае звучит так: квант времени должен быть немного больше, чем время, требующееся для типичного полного обслуживания. Если квант оказывается меньшего размера, большинство процессов потребует как минимум два кванта времени.

Выбор самого короткого процесса

Еще один путь к снижению перекоса в пользу длинных процессов — использование стратегии выбора самого короткого процесса (shortest process next — SPN). Это невытесняющая стратегия, при которой для выполнения выбирается процесс с наименьшим ожидаемым временем исполнения.

Основная трудность в применении стратегии SPN состоит в том, что для ее осуществления необходима по меньшей мере оценка времени выполнения требующегося каждому процессу.

Основной риск при использовании стратегии SPN заключается в возможном голодании длинных процессов при стабильной работе коротких процессов. Кроме того, хотя SPN снижает перекос в пользу длинных процессов, его применение нежелательно в системах с разделением времени или системах обработки транзакций из-за отсутствия вытеснения

Наименьшее остающееся время

Стратегия наименьшего остающегося времени (shortest remaining time — SRT) представляет собой вытесняющую версию стратегии SPN. В этом случае планировщик выбирает процесс с наименьшим ожидаемым временем до окончания процесса. При присоединении нового процесса к очереди готовых к исполнению процессов может оказаться, что его оставшееся время в действительности меньше, чем оставшееся время выполняемого в настоящий момент процесса. Планировщик, соответственно, может применить вытеснение при готовности нового процесса. Как и при использовании стратегии SPN, планировщик для корректной работы функции выбора должен оценивать время выполнения процесса; в этом случае также имеется риск голодания длинных процессов.

В случае использования стратегии SRT нет таких больших перекосов в пользу длинных процессов, как при использовании стратегии FCFS; в отличие от стратегии RR, здесь не генерируются дополнительные прерывания, что снижает накладные расходы. Тем не менее в этом случае происходит увеличение накладных расходов из-за необходимости фиксировать и записывать время выполнения процессов. В связи с тем что короткие задания немедленно получают преимущество перед выполняющимися длинными заданиями, стратегия SRT существенно выигрывает у стратегии SPN во времени оборота.

Справедливое планирование

Все рассмотренные алгоритмы планирования рассматривают множество готовых к выполнению процессов как единый пул, из которого выбирается очередной процесс для выполнения. Этот пул может быть разделен по степени приоритета процессов, но в противном случае он остается гомогенным.

Однако в многопользовательских системах при организации приложений или заданий отдельных пользователей как множества процессов (или потоков) у них имеется структура, не распознаваемая традиционными планировщиками. С точки зрения пользователя, важно не то, как будет выполняться отдельный процесс, а то, как будет выполняться множество процессов, составляющих единое приложение. Таким образом, было бы неплохо, если бы планирование осуществлялось с учетом наличия таких множеств процессов. Данный подход в целом известен как справедливое (fair-share) планирование. Эта же концепция может быть распространена на группы пользователей, даже если каждый из пользователей представлен единственным процессом. Например, в системе с разделением: времени мы можем рассматривать всех пользователей данного отдела как членов одной группы. Планировщик принимает решения с учетом необходимости предоставить каждой группе пользователей по возможности одинаковый сервис. Таким образом, если в системе находится много пользователей из одного отдела, то изменение времени отклика должно в первую очередь коснуться именно пользователей этого отдела, не затрагивая прочих пользователей.

Термин *справедливое планирование* указывает на философию, лежащую в основе такого планирования. Каждому пользователю назначен определенный вес, который определяет долю использования системных ресурсов данным пользователем. В частности, каждый пользователь использует процессор. Данная схема работает более или менее линейно, так что. если вес пользователя А в два раза превышает вес пользователя В, то в течение достаточно длительного промежутка времени пользователь А должен выполнить в два раза большую работу, чем пользователь В. Цель справедливого планирования состоит в отслеживании использования ресурсов и предоставлении меньшего количества ресурсов тому пользователю, который уже получил лишнее, и большего количества — тому, чья доля оказалась меньше справедливой.

Был предложен ряд алгоритмов справедливого планирования. В этом разделе мы рассмотрим схему планирования, реализованную в ряде систем UNIX. Эта схема известна как справедливый планировщик (fair-share scheduler —

FSS). FSS при принятии решения рассматривает историю выполнения связанной группы процессов вместе с индивидуальными историями выполнения каждого процесса. Система разделяет пользовательское сообщество на множество групп со справедливым планированием и распределяет процессорное время между ними. Так, если у нас имеется четыре группы, каждая из них получит по 25% процессорного времени. В результате каждая группа обеспечивается виртуальной системой, работающей, соответственно, медленнее, чем система в целом.

Планирование осуществляется исходя из приоритетов с учетом приоритета процесса, недавнего использования им процессора и недавнего использования процессора группой, к которой он принадлежит. Чем больше числовое значение приоритета, тем ниже сам приоритет. Для процесса j из группы k применимы следующие формулы:

$$CPU_j(i) = \frac{CPU_j(i-1)}{2}$$

$$GCPU_j(i) = \frac{CPU_j(i-1)}{2}$$

$$P_j(i) = Base_j + \frac{CPU_j(i-1)}{2} + \frac{GCPU_k(i-1)}{4W_k}$$

где $CPU_j(i)$ — мера загруженности процессора процессом j на интервале i;

$GCPU_k(i)$ — мера загруженности процессора группой k на интервале i;

$P_j(i)$ — приоритет процесса j в начале интервала i (меньшее значение соответствует большему приоритету);

$Base_j$ — базовый приоритет процесса j;

W_k — вес, назначенный группе k ($0 \le W_k \le 1$ и $\sum_k W_k = 1$).

Каждому процессу назначается базовый приоритет. Приоритет процесса снижается по мере использования им процессора, так же как и по мере использования процессора группой в целом. В случае использования процессора группой среднее значение нормализуется делением на вес группы. Чем больший вес назначен группе, тем меньше использование ею процессора влияет на приоритет.

7.3 Традиционное планирование в Unix

В этом разделе рассматривается традиционное планирование UNIX, используемое как в SVR3, так и в 4.3 BSD UNIX. Эти системы в первую очередь предназначены для работы в интерактивной среде с разделением времени. Алгоритм планирования разработан таким образом, чтобы обеспечить приемлемое время отклика для интерактивных пользователей, в то же время гарантируя отсутствие голодания низкоприоритетных заданий. Хотя описываемый алгоритм и был заменен в более современных версиях UNIX, его изучение как представителя практически используемых алгоритмов с разделением времени не лишено основания. Схема планирования SVR4 соответствует требованиям реального времени.

Традиционный планировщик UNIX использует многоуровневый возврат с применением кругового планирования в пределах очередей каждого приоритета, а также односекундное вытеснение. Таким образом, если текущий процесс не блокируется или не завершается в пределах одной секунды, он вытесняется. Приоритет основан на типе процесса и истории выполнения. Применяются следующие формулы:

$$CPU_j(i) = \frac{CPU_j(i-1)}{2}$$

$$P_j(i) = Base_j + \frac{CPU_j(i-1)}{2} + nice_j$$

где $CPU_j(i)$ — мера использования процессора процессом j на протяжении интервала j;

$P_j(i)$ — приоритет процесса *j* в начале интервала *i* (меньшее значение соответствует большему приоритету);

$Base_j$ — базовый приоритет процесса *j;*

$nice_j$ — указываемый пользовательский коэффициент.

Приоритет каждого процесса пересчитывается один раз в секунду, в момент принятия решения о том, какой процесс будет выполняться следующим. Назначение базового приоритета состоит в разделении процессов на фиксированные группы уровней приоритетов. Значения компонентов CPU и nice ограничены требованием того, чтобы процесс не мог выйти из назначенной ему на основании базового приоритета группы. Эти группы используются для оптимизации доступа к блочным устройствам (например, к диску) и обеспечения быстрого отклика операционной системы на системные вызовы. Имеются следующие группы приоритетов (приведены в порядке снижения приоритетов):

- программа свопинга;
- управление блочными устройствами ввода-вывода;
- управление файлами;
- управление символьными устройствами ввода-вывода;
- пользовательские процессы.

Такая иерархия должна обеспечить наиболее эффективное использование устройств ввода-вывода. В группе пользовательских процессов использование

истории исполнения приводит к применению штрафных санкций к процессам, ориентированным на вычисления, что также должно способствовать повышению эффективности системы. В сочетании с круговой схемой с вытеснением данная стратегия удовлетворяет требованиям общего назначения с разделением

ТЕМА 8

8.1 Управление вводом-выводом и файлами

Внешние устройства, по отношению к которым выполняются операции ввода-вывода могут быть объединены в три группы:

1) работающие с пользователем – связывают пользователя с вычислительной машиной (видеотерминалы, манипулятор, мышь, принтер);

2) работающие с компьютером (дисковые устройства, датчики, контроллеры), которые служат для связи с электронным оборудованием;

3) устройство коммуникации служит для связи с удаленным устройством (модемы, драйверы цифровых линий и т.д.);

Устройства ввода-вывода классифицируются по следующим характеристикам.

1. Скорость передачи данных.

2. Применение. Каждое действие, поддерживаемое устройством, оказывает влияние на программное обеспечение и стратегии операционной системы. Для размещения файла на диске требуется управление файловой системы. Для организации виртуальной памяти нужны программные средства, реализующие функции подкачки. С видеотерминалом может работать обычный пользователь и администратор.

3. Сложность управления.

4. Единицы передачи данных. Данные могут передаваться блоками или как поток байтов.

5. Представление данных. Разные устройства используют различные схемы кодирования данных, включая контроль четности и помехоустойчивые коды.

6. Условие ошибок. Природа ошибок, способ сообщения о них последствия, возможные ответы существующего отличия при переходе от одного устройства к другому.

8.1.1 Организация функций ввода-вывода

Существует три способа ввода-вывода.

1. Программируемый ввод-вывод. Препроцессор посылает необходимые команды контроллеру ввода-вывода и после этого процессор находится в состоянии ожидания завершения операции ввода-вывода;

2. Ввод-вывод, управляемый прерываниями. Процессор посылает нужные команды контроллеру ввода-вывода и продолжает выполнение следующих команд. Выполнение процесса прерывается контроллером ввода-вывода, когда устройство готово к передаче данных. После посылки команды, процессор выполнит или текущий процесс, если не требуется ожидание данных от устройств ввода-вывода, или процессор переключается на выполнение другого процесса;

3. Прямой доступ к памяти. Модуль прямого доступа к памяти управляет обменом данных между основной памятью и контроллером ввода-вывода. Процессор посылает запрос на передачу блока данных модуля прямого доступа к памяти. После завершения обмена может инициировать прерывание либо про-

веряется бит готовности;

8.1.2 Развитие функций ввода-вывода

Этапы развития функциональности устройств ввода-вывода.

1) Процессор непосредственно управляет периферийными устройствами;

2) К устройству добавляется контроллер или модуль ввода-вывода. Процессор использует программируемый ввод-вывод без прерываний. На этом этапе процессор отдаляется от конкретных деталей внешних устройств;

3) Та же конфигурация, что и на втором этапе, но еще используются прерывания. Повышение производительности достигается за счет того, что не требуется завершить оперативный ввод-вывод. Ожидается появление запроса на прерывание;

4) Модель ввода-вывода получает возможность работы с памятью через прямой доступ к памяти DMA или DP. На этом этапе блоки данных перемещаются без использования процессора за исключением начальных этапов;

5) Модуль ввода-вывода совершается и становится отдельным процессором, обладающим специальной системой команд для ввода-вывода. Центральный процессор задает процессору ввода-вывода задание выполнить программу ввода-вывода, находящуюся в основной памяти. Процессор ввода-вывода производит выборку и выполнение соответствующих команд без участия центрального процессора. Это позволит центральному процессору определить последовательность выполняемых функций ввода-вывода;

6) Модуль ввода-вывода обладает своей локальной памятью и, по сути, является отдельным компьютером. Здесь управление ввода-вывода различными устройствами осуществляется при минимальном вмешательстве центрального процессора. Такая архитектура используется для управления связью с интерактивными терминалами. Процессор ввода-вывода выполняет большинство задач, связанных с управлением терминала. Этот путь развития устройств ввода-вывода ориентирован на то, чтобы вмешательство центрального процессора в функции ввода-вывода становилось минимальным. Изменилась концепция устройства ввода-вывода, в результате чего устройство ввода-вывода способно само выполнять программу.

8.1.3 Управление ОС и устройствами ввода-вывода

При проектировании средств ввода-вывода имеется 2 цели: эффективность и универсальность.

Эффективность важна в том смысле, что операции ввода-вывода часто является основным тормозом производительности всей системы, т.к. они работают медленнее, чем центральный процессор. Один из подходов решения – многозадачность.

Универсальность служит для снижения вероятности возникновения ошибок и упрощения работы с устройствами ввода-вывода желательно иметь воз-

можность одинакового управления различными устройствами, как со стороны пользовательских процессов, так и со стороны ОС. Реально достичь большей универсальности не удается, но применение модульного подхода при разработке функций ввода-вывода возможно. ОС представляет определенный уровень иерархии, и каждый уровень представляет некоторое подмножество функций необходимых операционной системе. Некоторые части ОС должны взаимодействовать непосредственно с аппаратным оборудованием компьютера, где продолжительность события в проявлении нескольких микросекунд, а другие части ОС работают с пользователем, который вводит команды один раз в несколько секунд. Исходя из такого подхода, рассматривается несколько моделей организации ввода-вывода.

8.1.4 Модели организации ввода-вывода

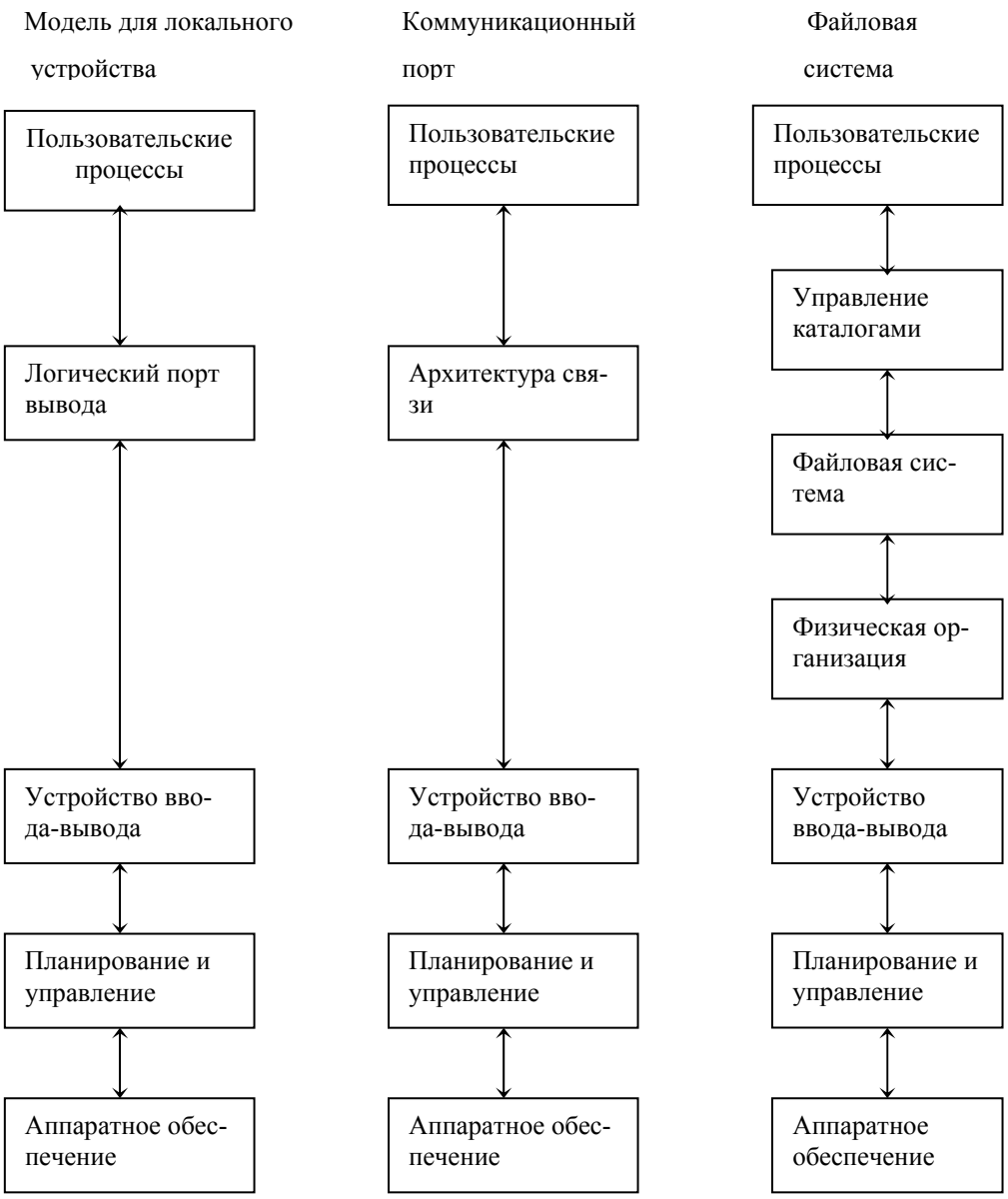


Рис. 8.1.

Функции ОС распределяются по уровням. Выполнение примитивных функций передается более низкому уровню. Более высокий уровень не знает деталей выполняющихся задач на низком уровне. С другой стороны, любой уровень обеспечивает обслуживание верхнего уровня.

Логический ввод-вывод. На этом уровне ведется обращение к устройствам как к логическим ресурсам и не уделяется внимание деталям фактического управления устройства. Уровень является посредником между пользовательскими процессами и устройством, предоставляя процессам набор высокоуровневых функций.

Устройство ввода-вывода. Запрошенные операции и данные преобразуются в соответствующие последовательности инструкции ввода-вывода, команды управления каналом, команды контроллера. Здесь может применять буферизация.

Планирование и управление. На этом уровне производится реальная организация очередей и планирование операций ввода-вывода, а также управление выполнением операций. Осуществляется работа с прерываниями, получение информации о состоянии устройства. Программное обеспечение на этом уровне непосредственно взаимодействует с аппаратурой устройства.

Управление каталогами. На этом уровне происходит преобразование символьных имен файлов в идентификаторы, указывающие на файл непосредственно или косвенно с использованием файлового дескриптора или индекса таблицы. На этом уровне организованы пользовательские операции работы с каталогами файлов: добавление, удаление, реорганизация.

Файловая система. Работает с логической структурой файлов и операциями над ними: открытие, закрытие, чтение, запись. Управление правами доступа также определяется на этом уровне.

ТЕМА 9

9.1 Аппаратно-программные особенности современных процессоров, ориентированные на поддержку многозадачности

9.1.1 Сегментация памяти

Под **сегментом** понимается блок смежных ячеек памяти в адресном пространстве 1Mb, с максимальным размером в 64Kb и начальным (базовым) адресом, находящимся на 16-байтной границе (**параграфе**). Для обращения к памяти необходимо определить базу сегмента и 16-битное расстояние от базы, называемое **смещением** .

9.1.1.1 Сегментация памяти в процессорах 8086

Схема сегментации памяти для этого случая приведена на рис. 9.1.

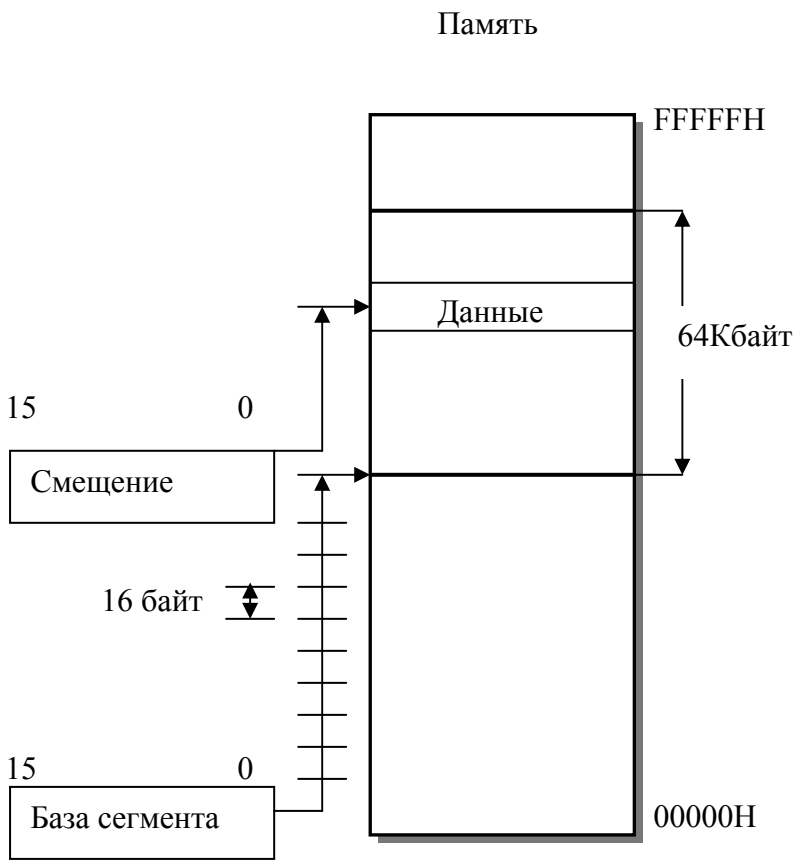


Рис. 9.1. Сегментация памяти в процессоре 8086

СЕГМЕНТ : СМЕЩЕНИЕ логический (виртуальный) адрес

Физический адрес ФА = СЕГМЕНТ *16 + СМЕЩЕНИЕ (*)

Чтобы упростить обращения к памяти, за каждой командой закрепляется сегментный регистр по умолчанию, т.е. команды адресуются через ***CS:IP***, стек через ***SS:SP***, данные через ***DS:[смещение]*** и ***ES:[смещение].***

Преобразование пары СЕГМЕНТ : СМЕЩЕНИЕ в физический производится в соответствии с выше приведенным выражением и, в результате, получаем 20-битный физический адрес. Переход от логического адреса в физический однозначен, а из физического адреса в логический неоднозначен, т.е. каждому физическому адресу может быть сопоставлено в соответствие 4Kb логических адресов.

При таком подходе есть неприятный момент, называемый ***заворачиванием адреса,*** возникающий, когда рассчитываемое значение физического адреса больше 1Mb.

Сегментация памяти процессора 8086 имеете особенности, усложняющие разработку многозадачных систем:

1. Сегменты памяти имеют всего два атрибута:

- начальный адрес (на границе параграфа);
- максимальный размер (64 Kb).

Никаких аппаратных средств контроля правильности использования сегментов нет.

2. Размещение сегментов в памяти произвольно, т.е. они могут частично или полностью перекрываться или не иметь общих частей.

3. Программа может обратиться к любому сегменту для выполнения операций чтения/записи или для выборки команды. Программа может обратиться по любому физическому адресу, а для защиты памяти от несанкционированного доступа требуются внешние схемы.

4. Нет препятствий для обращения к даже несуществующей физической памяти. Если программа выдаёт адрес несуществующей памяти, то результат работы зависит от аппаратных особенностей конкретной вычислительной машины.

9.1.1.2 Сегментация памяти в процессорах архитектуры IA-32

Сегменты описываются отдельными информационными структурами, называемыми ***дескрипторами.***

Основное отличие данной модели сегментации памяти заключается в том, что пользовательская программа не может свободно обращаться по любому адресу в пространстве памяти. Программа, в зависимости от уровня привилегий, не может обращаться к сегменту до тех пор, пока он не описан для данной программы.

Определение сегментов в многозадачном режиме производится на системном уровне, причем определяется следующая информация для каждого сегмента:

- базовый адрес;
- размер сегмента;
- целевое использование;
- уровень привилегий;
- другие параметры.

Базовый адрес может быть произвольным, т.е. выравнивание не требуется на границу параграфа или страницы. Размер сегмента может кодироваться от 1b до 4Gb. Это значит, что с помощью 32-битного индекса можно пройти всё адресное пространство, не модифицируя сегментный регистр.

9.1.1.3 Дескриптор сегмента

Каждый сегмент характеризуется 8-байтная структурой данных, которая называется ***дескриптор сегмента (Segment Descriptor).***

Рис. 9.2. Формат дескриптора сегмента

Поля дескриптора имеют следующие названия.

AR *- Access Rights*
DPL *- Descriptor Privilege Level*
D *- Default size*
U *- User*
A *- Accessed*
G *- Granularity*
P *- Present*
S *- System(segment)*
AVL - *Available*

Число дескрипторов в системе практически неограниченно. Если какую-то область адресного пространства не описать дескриптором, соответствующий диапазон адресов оказывается недоступным и процессор блокирует доступ к этим адресам.

Поля

Базовый адрес **-** это 32-х битное поле. Занимает 2,3,4,7 байты дескриптора и определяет начальный адрес сегмента в линейном адресном пространстве 4 Gb. Этот адрес формирует процессор при нулевом смещении.

Предел - 0,1 байты и 4 бита- граница сегмента(20 бит). Он равен размеру сегмента в байтах минус 1. Предел задает последнюю адресуемую единицу в сегменте. 20-битное поле предела позволяет определить размер в 1 Mb элементов.

Бит G (*гранулярность)*- определяет единицы измерения элементов памяти:

- *G=0,* то поле предела измеряется в байтах и соответственно имеет размер 1 Mb;

- *G=1,* то единицей измерения является страница, каждая из которых имеет размер 4Kb, следовательно максимальный размер 1 M*4Kb = 4 Gb.

Бит A- *бит доступа.* Автоматически устанавливается в 1, когда производится обращение к памяти, описанной данным дескриптором. Этот бит можно использовать для того, чтобы определить те сегменты, к которым долго не было обращения.

Бит D- размер по умолчанию. Он используется для совместимости с процессором 286.

Бит U- пользовательский. Его называют AVL. Предназначен для использования при системном программировании.

Бит AR *- байт прав доступа.*

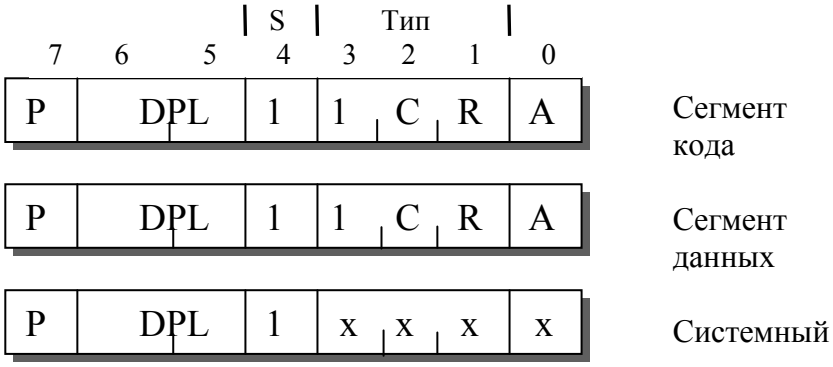


Рис. 9.3. Формат байта прав доступа

Байт прав доступа определяет тип сегмента и те возможности, которые предоставляются программе при работе с этим сегментом.

Для всех типов дескрипторов в байте доступа имеется *бит P (присутствие*):

P=1, когда описываемый дескриптором сегмент находится в физической памяти. В системе с виртуальной организацией памяти часть сегментов располагается на вешней дисковой памяти, бит *P = 0* , и если программа обращается к сегменту, у которого в дескрипторе бит *P = 0*, то возникает особый случай неприсутствия сегмента. В таком случае ОС должна найти свободную область в ОП, скопировать с диска содержимое сегмента в эту область, загрузить в дескриптор новый базовый адрес сегмента, установить *P = 1* и осуществить повторный запуск команды, вызвавшей особый случай. Процесс обращения к не присутствующему сегменту с последующей передачами сегмента между основной памятью и диском называется ***свопингом (подкачкой-swapping)***. Если свободного места для размещения сегмента в ОП нет, то ОС должна выгрузить некоторые сегменты из ОП на дескрипторы этих сегментов и *P = 0.*

DPL *– уровень привилегий дескриптора сегмента* (2 бита). Определяет уровень привилегий, ассоциируемый с той областью памяти, которую описывает дескриптор. Поэтому его следовало бы назвать уровень привилегий сег-

мента. Наивысший уровень привилегий: *DPL =00* (ноль), а наименьший – *DPL =11* (три). Поле *DPL* - составная часть механизма защиты в процессоре.

Бит S *(системный, бит сегмента)*. В дескрипторах сегментов памяти этот бит всегда равен 1. А в дескрипторах системных объектов *S=0.*

Трех битное поле типа определяет целевое использование сегмента, задавая допустимые в сегменте операции.

Тип:

000 – сегмент данных, разрешено только считывание;

001 – сегмент данных, разрешено считывание и запись;

010 – сегмент стека, разрешено только считывание;

011 – сегмент стека, разрешено считывание и запись;

100 – сегмент кода, разрешено только выполнение;

101 – сегмент кода, разрешено выполнение и считывание;

110 – подчиненный сегмент кода, разрешено только выполнение;

111 – подчиненный сегмент кода, разрешено выполнение и считывание.

Отдельные биты:

R - *Read;*

ED- *Expand Down;*

W - *Write;*

C - *Conforming.*

R – показывает возможность считывания кода как данных с помощью префикса замены сегмента. Если *R = 1*, то можно описать другой дескриптор и адресацию можно производить не через *CS*, а через *DS* и *ES.*

C-бит *подчинения*. Если *C = 1*, то соответствующий сегмент кода лишается защиты по уровню привилегий и при обращении к этому сегменту (фактически запуск программы, описывающей этот сегмент) его уровень привилегий переводится на уровень привилегий вызывающей программы. Таким образом он позволяет создать системные библиотеки доступные пользовательским программам.

Для сегмента данных ***W-бит*** показывает возможность записи в сегмент данных или, по-другому, возможность изменения его содержимого. Если этот бит равен нулю, то из сегмента можно читать данные, т.е. описывать виртуальное пространство как постоянное запоминающее устройство, похоже на ПЗУ.

Бит ED*-бит расширения вниз*. С помощью этого бита можно определить сегмент стека (ED=1) и сегмент данных (ED=0). Этот бит управляет интерпретацией поля предела. Размер сегмента памяти равен 8 Кб.

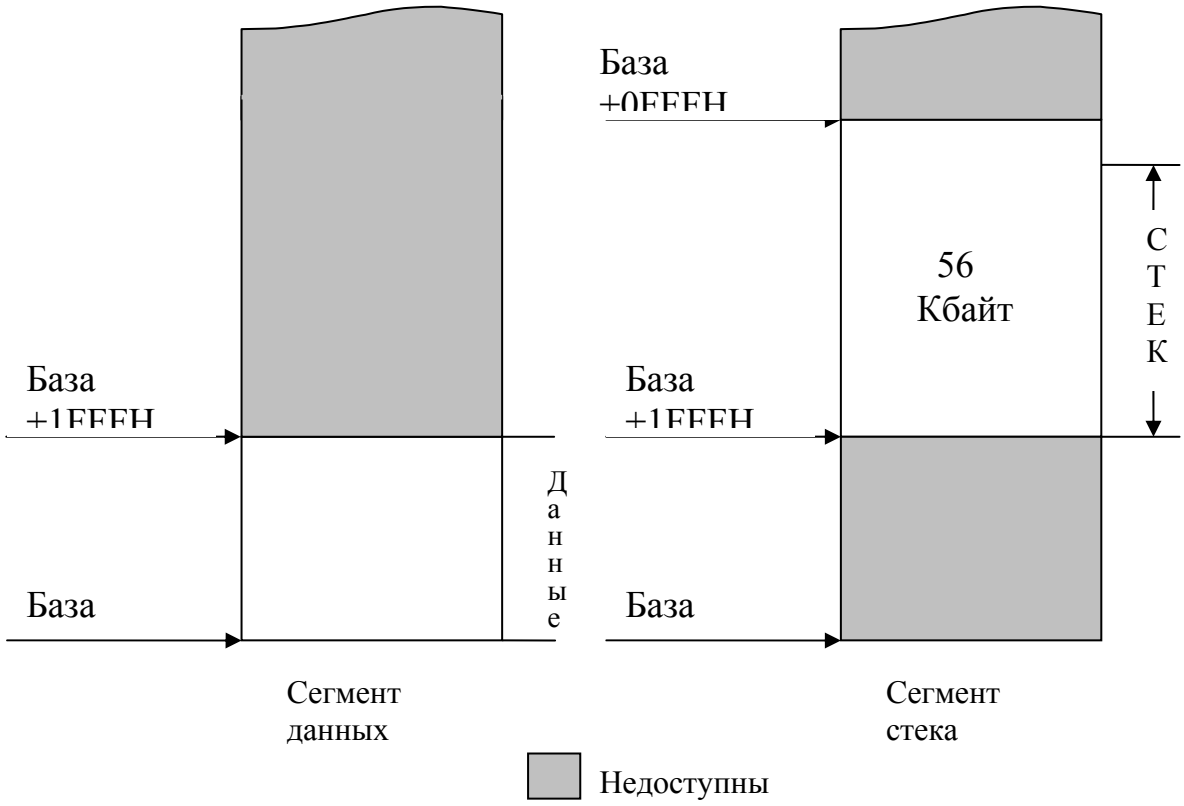


Рис. 9.4. Интерпретация поля предела для сегментов данных и стека

Таким образом определяется ситуация, при которой максимальный адрес находится вверху, а минимальный адрес (0) – внизу. Минимальный адрес задается для сегмента стека полем предела, а значение максимального адреса зависит от значения *бита* ***D*** *(размер по умолчанию*). Если *D=0*, то максимальный адрес: база + FFFFh, т.е. 64K-1. А если бит *D=1*, то максимальный адрес равен: база + const FFFFFFFFh.

Бит X – резервный.

Такая организация памяти позволяет сделать достаточно надежную систему защиты (изолированность кодов и данных).

9.1.1.4 Дескрипторные таблицы

Дескрипторные таблицы – области памяти, предназначенные для хранения 8-байтных дескрипторов. Порядок расположения дескрипторов в таблице

не имеет значения, а максимальное число дескрипторов в таблице равно 8192, т.е. максимальный размер дескрипторной таблицы – 64 Kb.

Процессор предусматривает использование трех типов дескрипторных таблиц.

1. *глобальные дескрипторные таблицы* ***Global Description Table (GDT)***
2. *дескрипторная таблица прерываний* ***Interrupt Description Table (IDT)***
3. *локальные дескрипторные таблицы* ***Local Description Table (LDT)***

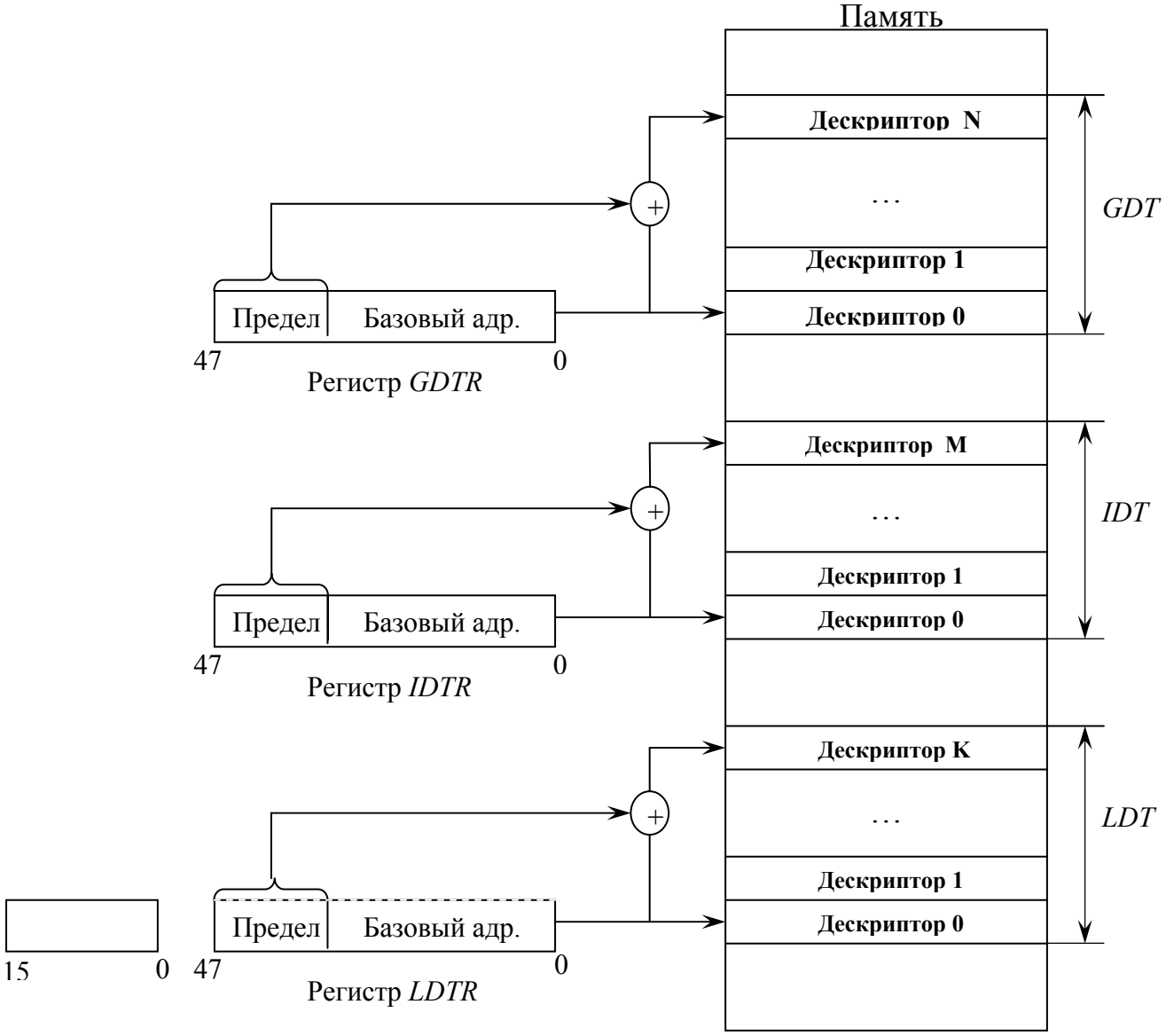


Рис. 9.5. Выбор дескриптора из таблицы

Главной общесистемной таблицей дескрипторов является глобальная дескрипторная таблица GDT. Все программы, выполняющиеся в системе, могут использовать эту таблицу дескрипторов для обращения к сегментам памяти. Место размещения *GDT* в памяти хранится в регистре глобальной дескрипторной таблицы *GDTR*. *GDTR* – 48-битный. В нем хранится 32-битный базовый адрес

и 16-битное поле предела. Предел имеет размер в байтах и если необходимо, например, в таблице хранить *N* дескрипторов, то поле предела будет равно *8·N-1*.

Дескрипторная таблица прерываний (*IDT*) является тоже общесистемной. Она содержит специальные системные объекты или дескрипторы специальных системных объектов, которые называются *шлюзами*, определяющими точки входа подпрограмм обработки прерываний и особых случаев, т. е. эта таблица заменяет таблицу векторов прерываний процессора 8086. Месторасположение этой таблицы определено в регистре *IDTR*.

В многозадачной системе для каждой задачи можно создать свою дескрипторную таблицу (*LDT*), в которой будут определены сегменты, доступные только конкретной задаче. Такую *LDT* определяет 16-битный регистр *LDTR*, что позволяет хранить дескрипторы, описываемые *LDT,* в *GDT*.

С регистром LDTR связан *теневой* регистр. При загрузке селектора в регистр *LDTR* в этот теневой регистр загружается дескриптор из *LDT*, что позволяет ускорит обращение к дескрипторам *LDT*. Это также позволяет рассматривать *LDT* как обычные сегменты памяти и применять к ним операции своппинга.

Система команд процессора имеет всего шесть команд (привилегированных), которые могут выполнять операции с регистрами дескрипторных таблиц. Эти команды позволяют записывать данные в эти регистры и считывать данные из этих регистров.

LGT	*mem48*	*L-Load-загрузить*
LIDT	*mem48*	*S-Save-сохранить*
SGDT	*mem48*	
SIDT	*mem48*	
LLDT	*reg16/mem16*	
SLDT	*reg16/mem16*	

Операнд *mem48* – это 48-битная структура памяти, первые 16 бит которой содержат предел, а следующие 32 бита - линейный базовый адрес.

Для *LLDT* и *SLDT* (команд работы с LDT) аргументом должен быть правильный селектор *LDT*.

9.1.1.5 Селекторы сегментов в защищенном режиме

Для выбора дескриптора из дескрипторной таблицы используются данные, которые загружаются в сегментные регистры. В защищенном режиме размер сегментного регистра 16 бит, однако его содержание интерпретируется следующим образом.

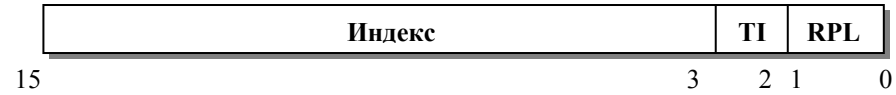


Рис. 9.6. Селектор сегмента в защищенном режиме

RPL - *Requested Privilege Level.*

TI - *Table Indicator.*

2-битное поле ***RPL*** (поле уровня запрашиваемых привилегий) в выборе дескриптора не участвует, а используется для контроля привилегий в механизме защиты.

Бит ***TI*** – *идентификатор таблицы* – показывает из какой дескрипторной таблицы следует выбрать дескриптор. Если *TI=0*, то из *GDT*, а если *TI=1*, то из *LDT*.

Последовательность действий по преобразованию логического адреса в линейный адрес.

1. Взять селектор из сегментного регистра.

2. Если бит *TI=0*, т.е. показывает на *GDT*, то из регистра *GDTR* выбрать адрес дескрипторной таблицы и перейти к шагу 4.

3. Если бит *TI=1* , т.е. показывает на *LDT*, то:

3.1. Взять селектор сегмента *LDT* из регистра *LDTR*;

3.2. Выделить в селекторе поле индекса, умножить этот индекс на 8 и т.о. получить смещение внутри *GDT*, которое указывает на расположение дескриптора в *LDTR*;

3.3. Прибавить полученное значение к адресу *GDT* из регистра *GDTR*;

3.4. Считать из памяти адресуемый дескриптор;

3.5. Выделить из этого дескриптора базовый адрес сегмента, содержащего таблицу LDT, полученный адрес является адресом;

4. Выделить значение из поля исходного индекса селектора, умножить его на 8 и прибавить к адресу дескрипторной таблицы и считать дескриптор по этому адресу;

5. Выделить из дескриптора базовый адрес сегмента;

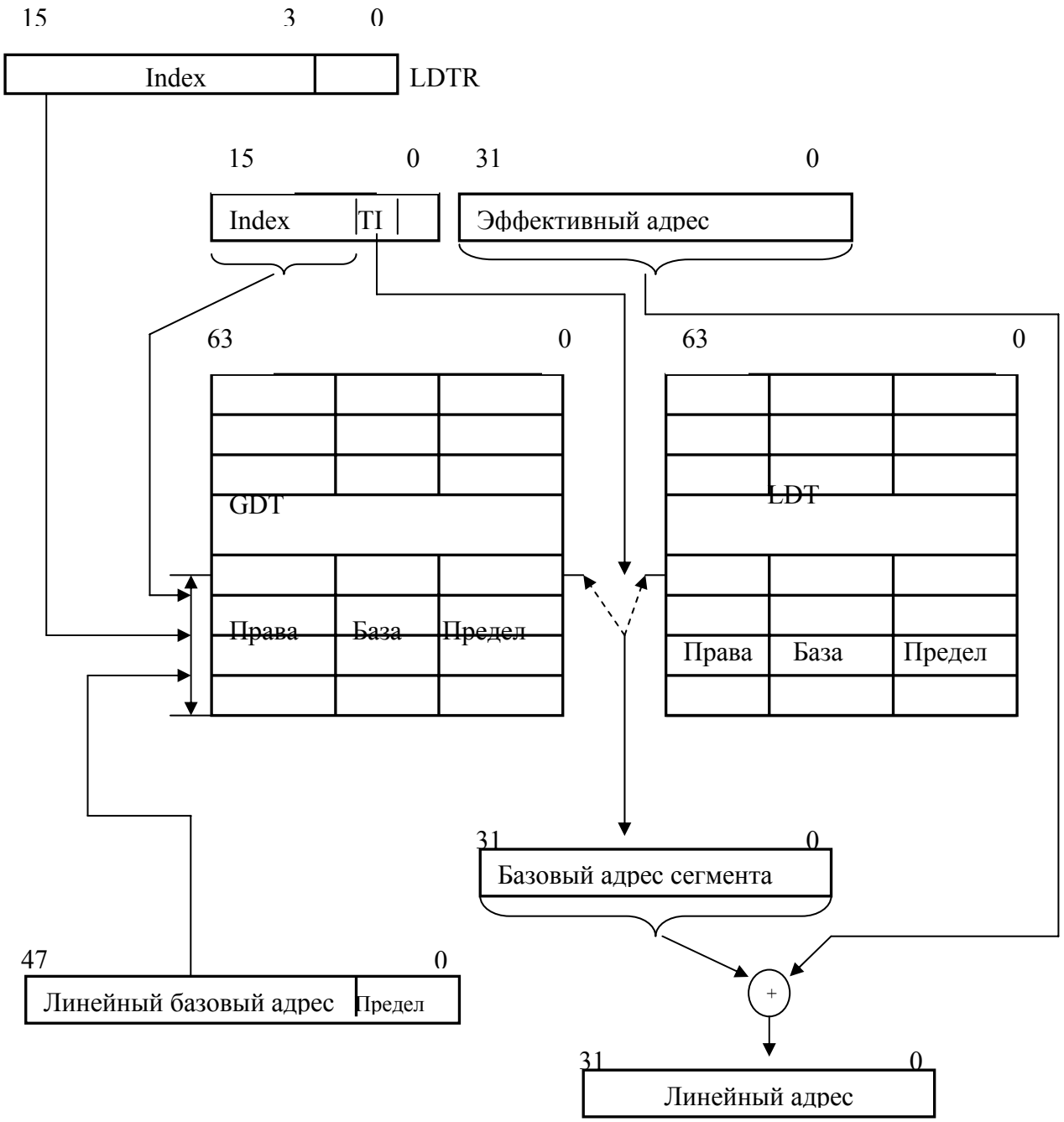
6. Прибавить значение смещения к базовому адресу сегмента, в результате получим линейный адрес

7. Осуществить обращение к памяти в соответствие с полученным линейным адресом

Если отмеченные действия производить каждый раз при преобразовании адреса, то время преобразования адреса было бы значительно больше, чем при работе процессора в реальном режиме.

Для ускорения таких преобразования адреса в процессоре имеются специальные ***теневые регистры***, связанные с сегментными регистрами и *LDTR*. Раз

Рис. 9.7. Схема преобразования адреса



мер этих регистров 64 бита и при загрузке нового значения либо в сегментный регистр, либо в *LDTR,* в соответствующий теневой регистр помещается в выбранный дескриптор из дескрипторной таблицы. Применение теневых регистров позволяет время преобразования адреса при работе в защищенном режиме сделать почти равным времени преобразования адреса при работе процессора в реальном режиме.

Некоторые особенности загрузки селекторов в сегментные регистры.

До загрузки селектора в сегментный регистр и соответствующего выбора дескриптора процессор выполняет несколько проверок. Ряд этих проверок связан с контролем уровня привилегий в механизме защиты, а другие предотвращают загрузку бессмысленных селекторов.

1. Процессор проверяет, чтобы поле индекса селектора находилось в пределах таблицы, определяемой полем TI, поэтому пределы дескрипторных таблиц хранятся вместе с их базовыми адресами.

2. При загрузке селекторов в сегментные регистры данных тип дескриптора должен разрешать считывание из сегмента. Только выполняемые дескрипторы для этих регистров не допускаются.

3. При загрузке регистра ***SS*** (сегментный регистр стека) в сегменте должны быть разрешены операции чтения и записи.

4. При загрузке регистра ***CS*** сегмент должен быть обязательно выполняемым.

5. Если эти проверки кончились успешно, то процессор анализирует бит присутствия в дескрипторе, и только если этот бит равен 1, разрешается загрузка сегментного регистра и загрузка соответствующего дескриптора в теневой регистр. В противном случае, процессор инициализирует прерывания по особым случаям и загрузка селектора не производится. Когда выбираемый селектором дескриптор находится вне предела дескрипторной таблицы или дескриптор имеет неверный тип, процессор формирует нарушение общей защиты (прерывание 13h). Если селектор отмечен как не присутствующий, то генерируется нарушение неприсутствия (прерывание 11h). Если ошибка связана с регистром SS, то генерируется прерывание 12h. При генерировании нарушений, ошибочный селектор включается в стек, и процедуры обработки особых случаев могут выяснить причину нарушения. В отличие от процессора 8086 в защищенном режиме по селектору не возможно узнать какое адресное пространство определяют адресные регистры. Селектор определяет только номер дескриптора, описывающего адресное пространство.

Для упрощения анализа дескриптора и с целью допущения инициирования заведомо запрещенных действий имеется несколько команд, через которые можно получить информацию о дескрипторах.

LAR reg16/32, reg16/mem16 *Load Access Rights Загрузка прав доступа*
LSL reg16/32, reg16/mem16 *Load Segment Limit Загрузка предела*
VERR reg16/mem16 *Verity for Read Проверка на считывание*
VERW reg16/mem16 *Verity for Write Проверка на запись*

При выполнении этих команд процессор учитывает уровень привилегий текущих программ и тех сегментов, которые проверяются с помощью этих команд. При нарушении прав привилегий информация не возвращается.

9.1.1.6 Локальные дескрипторные таблицы

Локальная дескрипторная таблица представляет собой массив восьмибайтных дескрипторов.

В любой момент времени процессор работает лишь с одной LDT, а при переключении задач изменяется и активная LDT, т.к. дескрипторы, описывающие

сегменты, где хранятся локальные таблицы, заносятся в GDT, то для локализации или определения используемой в текущий момент времени LDT, необходим только селектор. И для его хранения в процессоре используется 16-битный регистр LDTR. Поле предела занимает 20 бит, поэтому можно создать таблицу более 64 килобайт. Но на практике этого не требуется. В дескрипторе также задействован бит присутствия, и процессор не разрешает загрузить в регистр LDTR селектор, не присутствующий в LDT, генерируется особый случай неприсутствия.

Для начала работы с LDT в регистр LDTR помещается селектор выбираемой таблицы. При попытке загрузить в селектор недопустимое значение процессор формирует общее нарушение защиты. Допускается загрузка в LDTR пустого селектора. Нулевой селектор означает, что таблица LDT не используется.

К LDI обращение на прямую не допускается.

Дескриптор таблицы LDTR подобен общей структуре дескриптора.

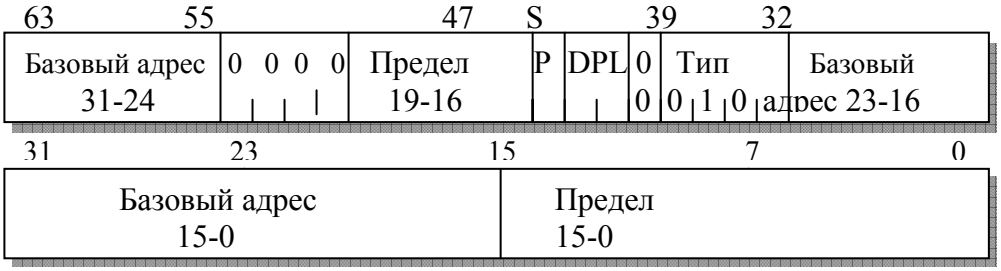


Рис. 9.8. Формат дескриптора таблицы LDT

9.1.1.7 Особенности сегментации

Разрешается создание сегментов, в которых допускаются операции только считывания, только исполнения, сегменты считывания - записи, сегменты исполнения - считывания. Не предусмотрено создания сегментов для всех трех функций. Однако путем перекрытия сегментов или введения дескрипторов с альтернативными наименованиями позволяют определять участки памяти, по отношению к которым можно выполнять все эти 3 действия одновременно. CPU обеспечивает мощные средства для сегментации памяти (защита, фиксация недопустимых операций и т.д.)

9.1.2 Страничная организация памяти

Внутреннее устройство управления памяти в процессоре, наряду с обязательной сегментацией, реализует еще один уровень косвенности в формировании физического адреса памяти. Это страничная организация памяти.

Основной причиной применения страничного преобразования является возможность реализации виртуальной памяти, которая позволяет программисту использовать объем памяти больше чем размер физической ОП.

Страничное преобразование было реализовано и до процессоров INTEL, но здесь оно реализовано как внутреннее устройство. В связи с этим появился ряд преимуществ:

- сокращается загрузка внешней шины;
- гарантируется правильность взаимодействия процессора и страничного устройства управления памятью;
- стандартизируются средства поддержки ОС виртуальной памяти;
- допускается программное разрешение и запрещение страничного преобразования адреса;
- снижается общая стоимость системы.

Если при сегментации базовым объектом памяти является сегмент практически любого размера, то при страничной организации базовым объектом памяти является блок с фиксированным размером 4 kb, который называется *страницей*. При линейном адресном пространстве 4 Gb получается 1 MB страниц. Реальная физическая память меньше 4 Gb, следовательно, при выполнении программы в любой момент времени только часть страниц будет присутствовать в ОП. Такой подход в организации памяти называется *виртуальной памятью.* Страницы, которые не поместились в ОП, располагаются на внешней памяти.

Фиксированный размер всех страниц позволяет загрузить любую виртуальную страницу в любую физическую страницу. Это и объясняет применение страничного преобразования в случае виртуальной памяти.

Понятие виртуальной памяти с заменой страниц по требованию подчёркивает тот факт, что прикладная программа не касается процесса страничного преобразования памяти и может использовать всё доступное адресное пространство.

Процессор автоматически формирует особый случай не присутствия, если обращение идёт к странице, отсутствующей в физической памяти. При возникновении этого особого случая ОС автоматически загружает затребованную страницу из внешней памяти.

В общем случае сегменты не являются кратными размеру страницы. Но рекомендуется выравнивать небольшие сегменты так, чтобы они полностью находились внутри одной страницы.

В процессе страничного преобразования старшие 20 бит 32-битного линейного адреса замещаются другим 20-битным значением, задающим адрес физической страницы.

Для такого преобразования используются таблицы страниц. В процессорах реализуется двухэтапное преобразование линейного адреса в физический. Можно реализовать и одноэтапное преобразование, но тогда для одной таблицы

страниц нужно 4 Мб. Поэтому целесообразнее использование двухэтапного преобразования.

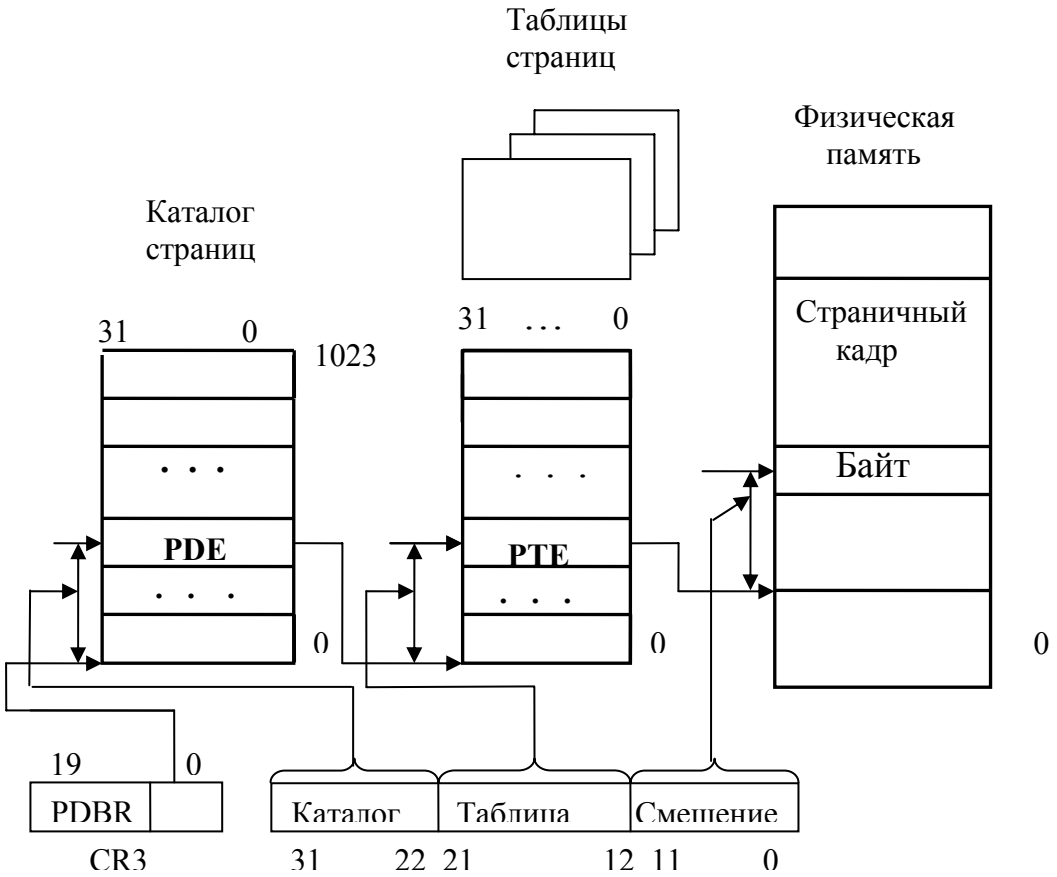


Рис. 9.9. Двухэтапное преобразование линейного адреса в физический

PDE – Page Directory Entry.
PTE – Page Table Entry.
PDBR – Page Directory Base Register.

Основой страничного преобразования является регистр CR3, который содержит 20-битный физический адрес каталога страниц в текущей задаче, и называется регистром базового адреса каталога страниц (PDBR). Это единственный внутренний регистр процессора, который содержит физический адрес памяти. Младшие 12 бит считаются нулевыми, т.е. каталог выровнен на границу страницы.

Предполагается, что каталог страниц всегда находится в памяти и не участвует в свопинге.

Каталог страниц содержит 1024 32-битных дескриптора, называемых элементами каталога страниц. Каждый из этих элементов адресует подчинённую таблицу страниц второго уровня. В свою очередь, каждая из этих страниц содержит 1024 32-битных дескриптора элементов таблицы страниц, а каждый из этих элементов адресует страничный кадр в физической памяти.

Преобразование линейного адреса в физический состоит из следующих действий.

1. Старшие 10 бит линейного адреса дополняются двумя младшими нулями и служат индексом каталога страниц, выбирая один из элементов в каталоге. Выбранный элемент каталога страниц определяет 20-ти битный адрес таблицы страниц.

2. Средние 10 бит линейного адреса дополняются двумя младшими нулями и индексируют таблицу страниц, выбирая из нее PTE. Этот элемент содержит 20-ти битный базовый адрес страничного кадра в физической памяти.

3. Базовый адрес страничного кадра объединяется с младшими 12-ю битами линейного адреса. В результате получается 32-х битный физический адрес памяти, по которому и производится обращение.

При обращении к элементам каталога и таблицы страниц, производится проверка защиты и присутствия страниц в памяти.

Структура элемента каталога страницы и таблицы страниц одинакова и имеет вид.

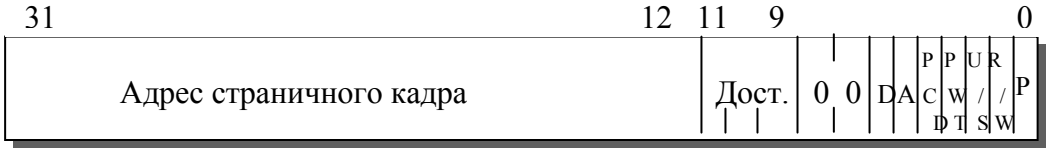


Рис. 9.10. Формат элемента таблицы страниц

A – Accessed
D – Dirty
PWT – Page Write Through
PCD – Page Cache Disable

В каждом элементе страниц хранится адрес страничного кадра. В этом поле находится физический базовый адрес страницы, младшие 12 бит адреса считаются нулевыми (выравниваются на границу страницы), биты 9, 10, 11 процессор не использует, и они могут использоваться при разработке операционных систем.

Бит присутствия P показывает, отображается ли адрес страничного кадра на страницу в физической памяти. Если P = 1, то страница находится в памяти.

Бит пользовательский или системный U/S определяет права доступа, т. е. уровни привилегий.

Бит обращения A и бит D содержат информацию об использовании страницы. Бит D сообщает об обращении к таблице для записи, а бит A сообщает об

обращении для чтения или записи к странице или таблице страниц второго уровня. За исключением бита D в элементе каталога страниц, эти биты устанавливаются в единицу аппаратно, но самостоятельно не сбрасываются, т. е. сбрасываются программно. Бит A используется для нахождения наиболее часто используемых страниц, а бит D используется при перезагрузке страниц. Анализируя бит D можно определить изменена ли страница в ОП.

Биты R/W (чтения и записи) применяются в механизме защиты применительно к страницам.

Биты PCD, PWT (запрещение кэширования страниц и сквозная запись) применяются для управления кэшированием на уровне страниц.

Механизм сегментации не касается страничного преобразования адреса. В процессе сегментации участвуют только логические и линейные адреса, и нет ни одного обращения к памяти по физическому адресу. Когда действует страничное преобразование, линейный и физический адреса обозначают совершенно различные объекты. Адреса физической памяти содержатся только в регистре CR3 и 20-битных полях базовых адресов элементов каталога страниц и таблиц страниц.

При разработке ОС, когда реализуется функции создания дескрипторов, речь идет о линейных адресах, а при создании таблиц для страничного преобразования уже преобразуются и физические адреса.

9.1.2.1 Страничный дескриптор

Страничный дескриптор (рис. 9.10) содержит следующую информацию.

P – вид присутствия. Он показывает, находится ли данная страница в физической памяти. Если P установлен в 0, то при попытке использовать данный элемент возникает особый случай страничного нарушения и операционная система, обрабатывая это страничное нарушение, должна выполнить действие по размещению физической операционной памяти выбранной страницы. После того, как страница загружена, в него загружается адрес, бит P устанавливается в 1. После этого повторяется команда, приведшая к страничному нарушению.

r/w – чтения и записи. Показывает какие действия можно выполнить над страницей;

u/s – бит прав доступа. Если при сегментации используется и привилегии, то для страничного только или на системном уровне или на пользовательском.

D и A используются для определения использования страницы. A = 1, если было обращение к странице. D = 1, если осуществилась запись в станицу, т.е. в странице изменились данные.

Эти биты используются при организации виртуальной памяти. D показывает, что если страница не изменилась, то нет смысла ее перезаписывать.

Механизм сегментации не касается страничного преобразования адреса. В процессе сегментации участвуют только логические и линейные адреса и нет обращения к памяти по физическому адресу. Когда включено страничное пре-

образование, линейные и физические адреса обозначают совершенно различные объекты. Хотя в принципе можно так создать каталог страниц и таблицу страниц, когда линейные адреса будут отражаться точно в такие же адреса – такая ситуация называется тождественным отображением. Прикладные программы имеют дело только с логическими адресами. При разработке операционной системы, когда реализовываются функции создания дескрипторов сегментов, речь идет о линейных адресах. А при создании таблиц для страничного преобразования уже требуются знания физических адресов. Каталог страниц и таблицы страниц размещаются в удобном месте и требуют таких дескрипторов сегментов, которые имеют их как данные с разрешенными операциями чтения и записи.

Современные процессоры поддерживают режим расширенного физического адреса до 64 бит. А общий размер памяти до 64 Гбайт.

PAE – Physical Address Extensions

В этом режиме блок страничной операции оперирует 64 битными элементами. Причем сохраняя режим поддержки и 32 битных элементов. В этом режиме поддерживается размер страницы 2 ГБ и схема преобразования адреса имеет несколько другую структуру.

9.1.2.2 Разрешение и запрещение страничного преобразования

Перед переходом в режим страничного преобразования адресов, создаются каталог страниц и таблицы страниц, которые размещаются в удобном месте ОП. Таблицы занимают блоки линейного адресного пространства и для их описания применяются дескрипторы, которые описывают эти блоки, с разрешенными операциями чтения и записи. Страничное преобразование включается последним из основных средств процессора. Для этого устанавливается в 1 старший бит в регистре CR0. И после этого, со следующей команды начинается страничное преобразование. Для того, чтобы не произошел переход после включения страничного преобразования в другой диапазон адресов, необходимо выделить область памяти, где осуществляется прямое отображение.

Для того, чтобы преобразование произошло корректно, рекомендуется выполнить следующие 3 действия:

1. Запретить аппаратные прерывания, включая и немаскируемые, если это возможно.
2. Разрешать страничное преобразование только из страницы с тождественным отображением.
3. После команды mov, которая установит в 1 бит PG, следует очистить очередь команд устройства предвыборки.

```
pushfd              ; сохранить состояние флага if
cli                  ; сброс if
```

```
mov eax, dir_base ;загрузить в бит cr3 базовый адрес каталога страниц
mov cr3, eax
mov eax, cr0   ; установить в 1 бит PG, разрешить страничное преобразование
bts eax, 31          ; установлен в 1 31-й бит
mov cr0, eax
jmp next
next: nop          ; очищается очередь предвыборки команд
popfd               ; восстановить if
```

Переход от страничного преобразования также должен идти с предосторожностями. При страничном преобразовании активно используется ассоциативный буфер страничного преобразования, в котором осуществляется кэширование элементов таблиц страниц и каталога страниц.

9.1.2.3 Сравнение сегментной организации памяти и страничной организации памяти

Достоинства сегментации.

1. Возможность реализации виртуальной памяти
2. Разработка надежных и живучих операционных систем на базе механизма защиты по привилегиям
3. Автоматическое обнаружение и обработка программных ошибок, вызванных неверными указателями или нарушением предела.
4. Сегментация имеет гибкую схему реализации(можно создать 1 сегмент кода и 1 сегмент данных, которые включают в себя все адресное пространство). При этом, программа видит только свое адресное пространство, а с сегментами работает только ОС. Защита осуществляется только по задачам, и можно создать полностью сегментную систему, когда каждому большому объему данных или кода выделяется отдельный сегмент. В результате получается надежная среда выполнения программ, но ухудшается быстродействие, по причине частой перезагрузки сегментных регистров.

Основным недостатком сегментации является необходимость выполнения служебных функций при загрузке селекторов в системные регистры. В ходе выполнения этой операции необходимо загружать в теневой регистр 8-байтный дескриптор сегмента из соответствующей дескрипторной таблицы, а также должна производится установка бита A в 1, в соответствующем дескрипторе дескрипторной таблицы. А также к недостаткам сегментации относится фрагментация памяти.

Основным достоинством страничной организации памяти по сравнению с сегментацией является фиксированный размер страницы. Это позволяет:

– решить проблему организации памяти в общем (удобно отганизовывать виртуальную память);

– согласовать и обеспечить высокую скорость передачи данных между внешней памятью и ОП.

Страничная организация памяти не видна прикладному программисту.

Недостатки страничной организации памяти.

1. Имеет место эффект *внутренней фрагментации*, когда сегменты памяти имеют размеры, не кратные размеру страниц;

2. Увеличение времени преобразования адреса, т.к. обращение к каталогу страниц и таблиц страниц требуется для каждой операции обращения к памяти (хотя компенсируется аппаратной организацией).

9.1.3 Организация защиты при работе процессора в защищенном режиме

При работе в защищенном режиме допускается одновременное выполнение нескольких прикладных программ, но они изолированы друг от друга таким образом, что ошибки одной программы не влияли на другие программы и операционную систему.

Когда программа совершила неожиданное обращение к недопустимому для неё пространству памяти, механизм защиты блокирует обращение и сообщает о его возникновении. Однако следует понимать, что в однопроцессорной системе можно реализовать только виртуальную многозадачность, т.е. выполнение только одной задачи в один момент, но имеется возможность быстрого переключения между задачами. Для целей защиты предусматривается как минимум два режима работы:

– системный режим (режим супервизора);

– пользовательский режим;

В режиме супервизора, в котором работает операционная система, доступны все ресурсы системы. При работе в пользовательском режиме накладывается ряд ограничений по выполнению некоторых команд процессора, влияющих на общие системные ресурсы (некоторые команды ввода-вывода, управление системными ресурсами, прерываниями и т.п.).

В процессорах Intel обеспечивается аппаратная поддержка защиты по 4 уровням привилегий.

Средства защиты должны предотвращать:

– неразрешенное взаимодействие пользователей друг с другом;

– несанкционированный доступ к данным;

– повреждение программ и данных из-за ошибок в других программах;

– преднамеренные попытки разрушить целостность системы;

– случайное искажение данных.

Механизм защиты процессоров Intel делится на две части:

- управление памятью;

- защита по привилегиям.

Схемы управления памятью обнаруживают большинство программных ошибок (например, формирование неверных адресов, нахождение индекса за

пределами массива, искажения стека и т.д.), а защита по привилегиям позволяет выявить более тонкие ошибки и преднамеренные попытки нарушить функционирование системы.

При работе в защищенном режиме процессор постоянно контролирует достаточно ли привилегированна текущая программа для того, чтобы:

- выполнять некоторые команды;
- обращаться к данным других программ;
- передавать управление внешнему коду по отношению к самой программе с помощью команд дальней передачи управления: ***far call*** или ***far jmp***.

9.1.3.1 Привилегированные команды

К ним относятся те, которые модифицируют состояние флажка *if* , изменяют сегментацию, или изменяют сам механизм защиты, а также команды ввода/вывода.

Команды, воздействующие на механизм сегментации и защиты, могут выполнятся только на нулевом уровне привилегий:

hlt	;остановка процессора;
clts	;сброс флага переключенной задачи;
lgdt, lidt, lldt	;загрузка регистров дескрипторной таблицы;
ltr	;загрузка регистра задач;
lmsw	;загрузка слова состояния машины;

К этой группе также относятся команды передачи данных в регистры управления и проверки.

Вторую группу образовывают команды, связанные с изменением флажка прерываний и команды, производящие ввод/вывод.

Эти команды могут быть выполнены программой, у которых уровень привилегий меньше либо равен уровню *IOPL (*это поле, находящееся в регистре флажков). Их еще называют *IOPL-чувствительными командами.* Изменить значение этого бита флажка прерывания и поля *IOPL*, который находится в регистре флажков. Казалось бы, можно обходным путем с помощью команды занесения регистра флажков в стек и извлечения из стека, которые не относятся к привилегированным. Но процессор всё равно контролирует уровень привилегий программы, которые выполняет эти команды и, если обнаруживается, что попытку изменить указанные биты предпринимает программа, которая не имеет на это право (не относящихся к нулевому уровню), но процессор запрещает модификацию этих битов. Для того, чтобы программа могла выполнять IOPL – чувствительные команды надо, чтобы CPL<=IOPL.

9.1.3.2 Защита доступа к данным

Для работы любой программы требуется адресное пространство данных и стека. Процессор не разрешает обращаться к данным, которые более привилегированны, чем выполняемая программа.

Основное правило защиты доступа к данным формулируется следующим образом: $CPL \le DPL$. Т.е. текущий уровень привилегий выполняемой программы должен быть меньше или равен уровню привилегий дескриптора сегмента данных, к которому идет обращение.

При выполнении команд обращения к данным процессор

1. проверяет привилегии при загрузке селектора в один из сегментных регистров данных (*DS, ES, FS, GS*). Если условие $CPL \le DPL$ не выполняется, то селектор не загружается, и формируется ситуация нарушения общей защиты и изменение сегментного регистра не производится (блокировка).

2. После успешной загрузки селектора, при использовании его для фактического обращения к памяти, процессор контролирует, чтобы запрашиваемая операция чтения или записи для этого сегмента была разрешена.

При загрузке селектора в сегментный регистр стека правила защиты ужесточаются, а именно $CPL = DPL$, т.е. запрещается использовать стек даже с меньшим уровнем привилегий. Если P=1, то выбираемый сегмент должен обязательно присутствовать в памяти.

$\max(CPL, RPL) \le DPL$ - это правило при обращении к сегменту данных.

RPL- *Requested Privilege Level (запрашиваемый уровень привилегий)*

DPL*- Descriptor Privilege Level*

EPL*- Effective Privilege Level (эффективный уровень привилегий)*

CPL*- Current Privilege Level*

В системе защиты используется поле RPL. Защита по принципу CPL<=DPL позволяет контролировать код и данные на различных уровнях привилегий. А для предотвращения использования ошибочных указателей, которые передают более привилегированным программ используется понятие эффективного уровня привилегий max(CPL,RPL) = EPL и с учетом этого должно выполняться условие EPL <= DPL. Если RPL < CPL, то поле RPL значения не имеет и RPL = 0. Если RPL = 0, то контроль идет по текущему уровню привилегий того сегмента, к которому идет обращение.

9.1.3.3 Защита сегмента кода

Процессоры Intel запрещают передачу управления сегменту кода, находящемуся на другом уровне привилегий. Это самый сложный механизм защиты и самый важный. Помимо прочего он имеет несколько исключений. Ограничивая передачу управления в пределах одного кольца защиты, процессор предотвращает изменений уровней привилегий. Как известно, передачу управления в другой сегмент выполняют команды дальнего перехода, дальнего вызова процедуры и возврата из процедуры (***far call, far jmp u ret*** соответственно). Адрес передачи управления задается с помощью 48-битного указателя селек-

тор:смещение, который содержится либо в самой команде, либо берётся из памяти. При выполнении этих команд изменяется значение регистра **CS** и **EIP.**

С точки зрения процессора контроль межсегментной передачи управления заключается в проверке достоверности селектора, загружаемого в регистр *CS*. При загрузке селектора в регистр *CS* процессор выполняет следующие проверки.

1. Проверяет, что целевой дескриптор определяет сегмент кода, т.е. имеет атрибут выполняемого сегмента.

2. *CPL* должен быть равен *DPL* целевого сегмента.

3. Целевой сегмент кода должен быть отмечен присутствующим и новое значение регистра указателя команд должно находится в пределах нового сегмента кода.

Если какая-либо проверка даёт отрицательный результат, то формируется ошибка общей защиты (int). При невыполнении этих проверок формируется нарушения общей защиты или исключение не присутствия.

9.1.3.4 Определение текущего уровня привилегий

CPL – это уровень привилегий выполняющегося кода. Он задается полем *RPL* селектора в регистре *CS*. Поэтому имеется возможность выполняемой программе определить уровень привилегий, на которых она выполняется.

```
mov ax, cs          или      push cs
and ax, 03h                  pop ax
and ax, 03h
```

Два способа передачи:

1) использование подчиненных сегментов;
2) использование специальных дескрипторов называется шлюзами вызова.

9.1.3.5 Передача управления между уровнями привилегий

Когда пользовательские программы взаимодействуют с операционной системой, возникает потребность передачи управления с низкого уровня привилегий на уровень привилегий операционной системы. Для таких передач есть два способа. Первый более простой и называется использование подчиненных сегментов. Второй более сложный – использование специальных дескрипторов, названных шлюзами вызовов.

9.1.3.6 Подчиненные сегменты

Сегмент кода определяется как подчиненный, если бит с в байте прав доступа дескриптора сегмента установлен в 1. при обращении к таким сегментам обычное правило защиты CPL = DPL не действует, действует тока правило, что CPL >= DPL, е.у. можно передавать управление на более высокий или текущий уровень привилегий. При передаче управления на подчиненный сегмент два младших бита регистра CS не изменяются. Таким образом, выполнение программы будет производится на том же уровне, на котором выполнялась вызывающая программа. Такой способ передачи управления используется при обращении к функциям операционной системы, которая не требует изменить состояние системы и не требует работы с внешними устройствами. Передача управления через подчиненные сегменты может осуществляться только во внутренние, более защищенные уровни привилегий.

9.1.3.7 Шлюзы вызова

Шлюзы вызова позволяют реализовать фактическое изменение уровня привилегий.

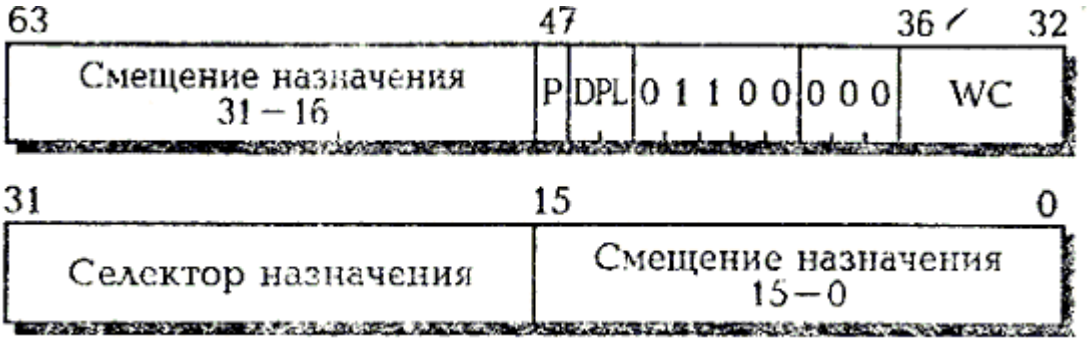


Рис. 9.11. Формат дескриптора шлюза вызова

Поле *WC(Word Count) - счетчик слов,* определяет число параметров, которые передаются в вызываемую программу; максимально можно передать в вызываемую программу через шлюз вызова 128 байт.

Шлюз вызова определяет точку входа программе.

Дескриптор шлюза вызова определяет полный указатель (селектор + смещение) точки входа в процедуру назначения, которой передается управление. Дескриптор шлюза вызова – это своеобразный интерфейсный слой между сегментами кода, находящимися на различных уровнях привилегий.

Шлюзы вызова определяют разрешенные точки входа в более привилегированный код и являются единственным средством смены уровня привилегии.

Дескрипторы шлюзов вызова не определяют никакого адресного пространства, поэтому у них нет полей базы и предела. По своей сути это даже не дескрипторы, но их размещают либо в глобальной дескрипторной таблице, либо, при необходимости, в локальных дескрипторных таблицах. Селекторы для выбора дескрипторов шлюзов вызова необходимо загружать только в сегментный регистр *CS* и ни в какие другие сегментные регистры.

Адресовать шлюз вызова можно только в команде межсегментного вызова ***far call,*** использование ***far jmp*** запрещено. Сама команда ***call*** должна адресовать шлюз вызова, а не сегмент кода назначения.

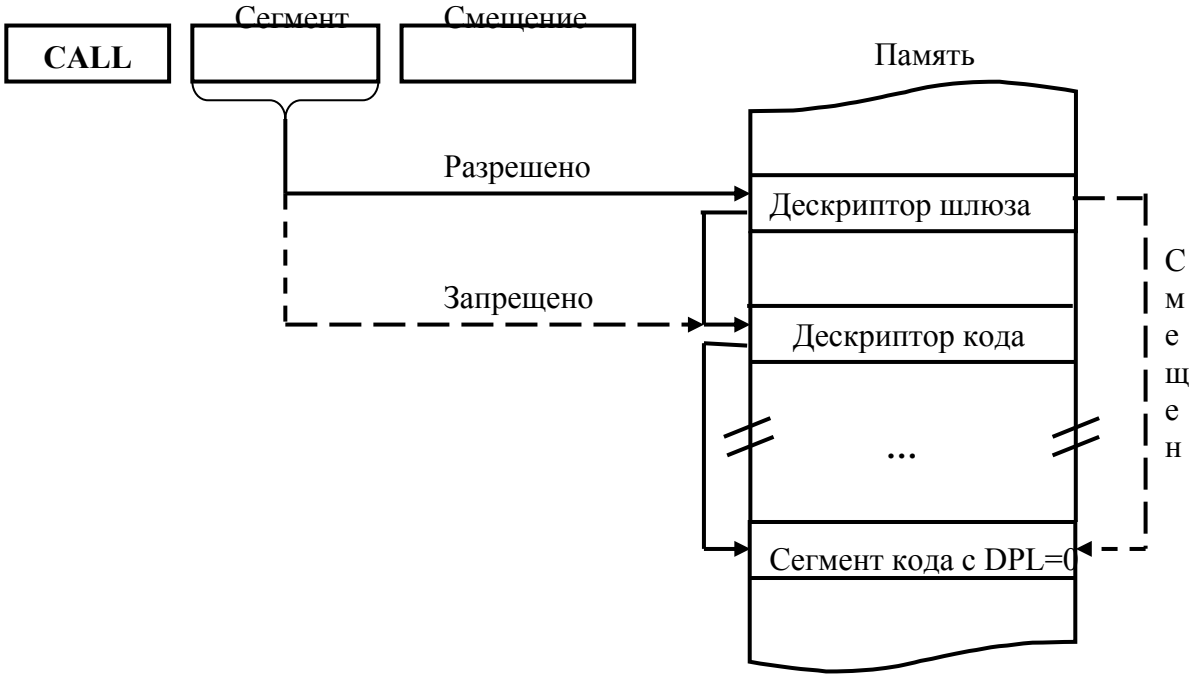


Рис. 9.12. Разрешенный и запрещенный вызовы более привилегированного кода

Реализованный в Intel процессорах косвенный вызов привилегированных процедур имеет несколько преимуществ:

1. Привилегированный код сильно защищен, и вызывающие его программы не могут его разрушить. При этом предполагается, что сам код такой процедуры тщательно отлажен и не содержит ошибок.

2. Шлюзы вызова делают привилегированные процедуры невидимыми для программ на внешних уровнях привилегий.

3. Так как вызывающая программа прямо адресует только шлюз, реализуемые процедурой функции можно изменить или переместить их в адресном пространстве, не затрагивая интерфейс со шлюзом.

Правила защиты при использовании шлюза вызова

1. $DPL_{\text{шлюза вызова}} \geq CPL$;
2. $DPL_{\text{шлюза вызова}} \geq RPL_{\text{селектора шлюза}}$;
3. $DPL_{\text{шлюза вызва}} \geq DPL_{\text{целевого сегмента кода}}$.Это правило предотвращает передачу на более низкий уровень привилегий;
4. $DPL_{\text{целевого сегмента кода}} \leq CPL$.

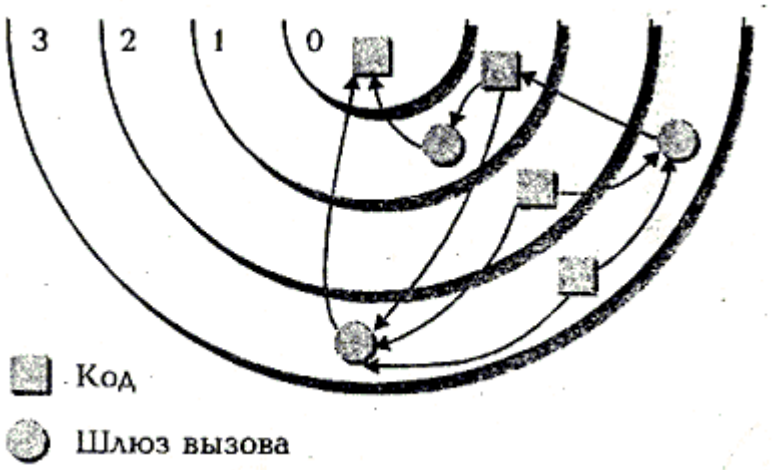


Рис. 9.13. Разрешенные варианты передачи управления через шлюз вызова

9.1.4 Поддержка многозадачности в процессорах архитектуры IA-32

Под задачей понимается программа или группа связанных программ, которая выполняется в многозадачном (мультипрограммном) режиме и её выполнение не должно подвергаться воздействиям извне (в смысле другими программами), и в свою очередь, она не должна оказывать влияния на действия, выполняемые другими программами. В процессорах Intel архитектуры задача определяется как совокупность кода и данных, которым назначен сегмент состояния задачи, т.е. сегмент состояния задачи является эквивалентным контекстной памяти, в которой хранится информация о задачи, когда она не выполняется и откуда она берется при повторном старте задачи. Сегмент состояния задачи – небольшой сегмент данных с разрешенными операциями чтения и записи, доступ к которым запрещен всем программам и к сегменту состояния задачи может обращаться только сам процессор.

Для управления многозадачностью в процессоре нет специальных команд. Вместо этого в некоторых случаях по-другому интерпретируются команды межсегментной передачи управления. Переключение задачи может быть вызва-

но командами межсегментной передачи управления (***far call*** и ***far jump***). Помимо этого новая задача может активизироваться прерыванием или особым случаем. Когда реализуется одна из таких форм передачи управления, по типу адресуемого дескриптора процессор определяет что ему нужно сделать: выполнить обычную межсегментную передачу или переключить задачу.

Имеется два типа дескрипторов, относящихся к задачам:

1. Дескриптор сегмента состояния задачи.
2. Шлюз задачи.

При каждом переключении задачи процессор может перейти к другой локальной дескрипторной таблице.

Переключение задачи похоже на вызов процедуры, но при этом сохраняется намного больше информации. Сохраненная информация должна обеспечивать возобновление работы задачи с той же точки и в том же объеме что и когда задача была прервана.

Для поддержки многозадачности в процессоре имеются:

- регистр задачи;
- сегмент состояния задачи;
- дескриптор сегмента состояния задачи;
- дескриптор шлюза задачи.

Типичным примером многозадачной работы являются системы разделения времени. В ходе функционирования такой системы требуются средства, позволяющие приостановить выполнение задачи, временно сохранить ее состояние, восстановить состояние задачи, на которую осуществляется переключение и инициировать ее продолжение работы.

9.1.4.1 Сегмент состояния задачи

Сегмент состояния задачи представляет собой контекстную память, в которой сохраняется информация о задаче, когда она не выполняется и откуда берется информация при рестарте задачи.

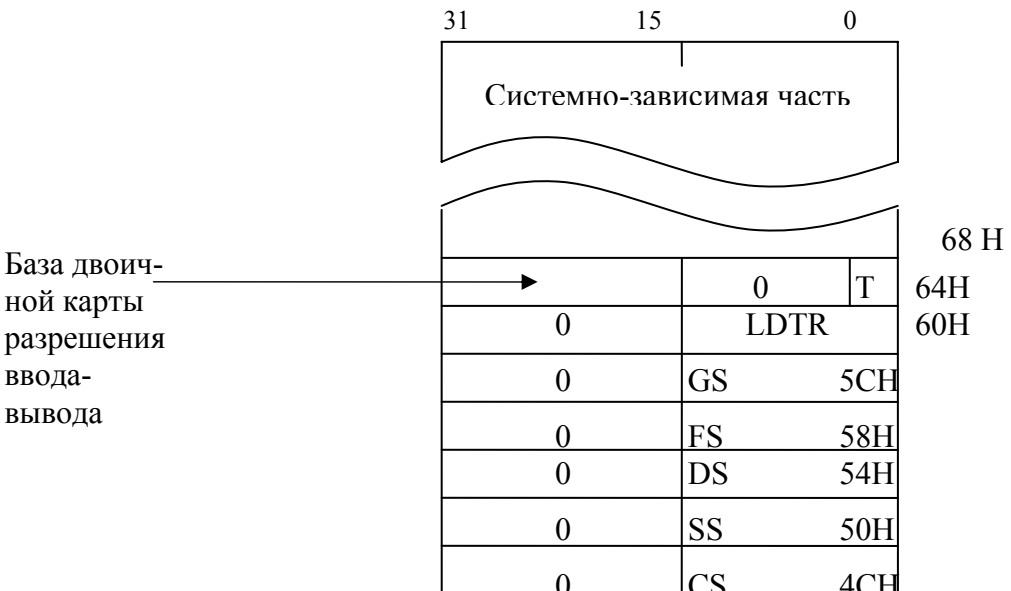


Рис. 9.14. Формат 32-битного сегмента TSS

Часть сегмента состояния задачи размером в 104 байта является обязательной. Эту часть процесса сохраняет и восстанавливает автоматически. Как и любой другой сегмент TSS определяется дескриптором, который называется дескриптором сегмента состояния задачи, который может находится тока в глобальной дескрипторной таблице.

Сохранение регистров флагов позволяет восстановить условия выполнения предыдущей команды. Бит *T* – бит ловушки. Он используется при отладке.

В поле обратной связи хранится селектор TSS задачи, которая выполнялась перед текущей. С его помощью может быть организована цепь вложенных задач подобно цепочке вложенных подпрограмм.

Когда процессор начинает выполнение новой задачи, он считывает всю информацию из первых 104 байтов TSS (минимальный размер сегмента состояния задачи). После этого выполняется команда, которая адресуется регистрами *CS* и *EIP*. При переключении задач между задачами никакой информации не

передается. Этим исключается искажения задач и обеспечивается возможность переключения задач в любой момент времени.

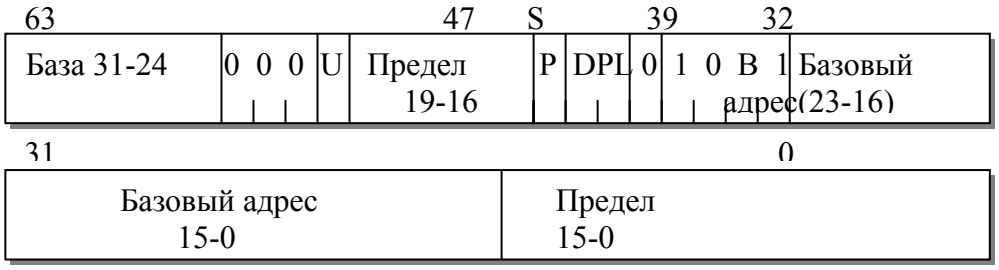


Рис. 9.15. Формат дескриптора сегмента состояния задачи

Сохранение состояния выполняемой задачи требует сохранения содержимого всех регистров процессора, некоторых переменных и адреса команды, которая должна выполняться после рестарта задачи. Это информация по задаче называется её *контекстом*, а действия процессора по сохранению состояния задачи и рестарта другой задачи называются *переключением контекста.*

В ЭВМ выделяется область памяти, доступная только ОС в которую записывается контекст задачи. Требуется чтобы всем задачам, работающим в системе, выделялись такие области памяти, и они постоянно находились в ОП машины. Это обеспечит быстрое переключение задач, но требует определенных затрат ОП (Поле дескриптора TSS стандартное).

9.1.4.2 Дескриптор сегмента TSS

Дескриптор состояния задачи определяет сегмент состояния. Поле *DPL* этого дескриптора показывает, какие программы по уровню привилегий могут обращаться к задаче, определенной данным дескриптором TSS, т.е. функции этого поля аналогичны функциям *DPL* в шлюзе вызова.

Поле типа показывает активна задача или нет. Задачи не являются реентерабельными (т.е. не допускается вызов самой себя или повторный вызов), поэтому бит *B* имеет существенное значение. Он показывает занята задача или нет, если бит в 1 – это значит что задача выполняется, т. е. занята.

Поле предела имеет размер не менее 67h, это минимальный допустимый размер сегмента состояния задачи. При попытке переключения на задачу, дескриптор которой имеет меньший предел, чем 67h, то в дескрипторе возникает особый случай, но больший предел может быть(когда применяется база двоичной карты ввода/вывода).

Обращение к дескриптору сегмента состояния инициализирует переключение задачи. Во многих ОС, поле DPL этих дескрипторов должно содержать 0, поэтому переключение задач могут производить высоко привилегированные программы. Программам не предоставляется возможность считать или модифицировать этот дескриптор. Модификацию можно произвести, применяя до-

полнительные дескриптор данных, описывающий ту же область памяти, дескрипторы TSS могут располагаться только в GDT.

Загрузка дескриптора в сегментный регистр приводит к особому случаю. Инициализируется особый случай, когда производится обращение к дескриптору сегмента состояния задачи с установленным битом *TI*. Не предусмотрено сохранение значения сопроцессора, не все задачи использует сопроцессор и его сохранение нужно производить только когда следующая задача начинает изменять его состояние.

9.1.4.3 Сегмент состояния задач TSS

Большая часть этого сегмента отведена для хранения внутренних регистров процессора и для ряда сегментных регистров, таких, как DTR CR3 и другие.

В этом сегменте также хранятся 3 указателя на стеки, для 0, 1 и 2-го уровня привилегий. Для каждой задачи допускается образование своего каталога страниц и LDT. В поле обратной связи сохраняется селектор состояния той задачи, которая выполнялась перед текущей. С помощью этого поля можно организовать цепочку вложенности задач.

Бит T – это бит ловушки, который применяется при отладке программы. Кроме того, сохраняется в TSS 16-битное слово, которое определяет смещение начала двоичной карты размещений разрешения ввода-вывода. Эта карта помещается в памяти произвольно, но обязательно вблизи сегмента TSS. Эта карта используется при контроле привилегий для команд ввода-вывода. Следовательно, она может занимать 8 Кб памяти. Если бит, для соответствующего порта установлен в 1, то попытка задачи обратиться к этому порту приводит к формированию особого случая нарушения общей защиты, т. е. таким образом можно разделить порты между отдельными задачами. При переключении задач между ними, никакой информации не передается, поэтому они максимально друг от друга, кроме того, при переключении задач сохраняется весь контекст старой задачи, но перепись указателей на стеки не производится в целях экономии времени, потому что эти указатели считаются постоянными на всё время существования задачи.

9.1.4.4 События, которые могут вызвать переключение задачи

Используются термины "выходящая (старая) задача" и "входящая (новая) задача".

Переключение задач может быть вызвано четырьмя событиями.

1. Выходящая задача выполняет команду ***far call*** или ***far jmp***, и селектор выбирает шлюз задачи.

2. Выходящая задача выполняет команду ***far call*** или ***far jmp***, и селектор выбирает дескриптор *TSS*.

3. Выходящая задача выполняет команду ***IRET*** для возврата в предыдущую задачу, которая приводит к переключению задачи, если в регистре флагов установлен бит *NT = 1* (бит вложенности).

4. Возникло аппаратное или программное прерывание, и соответствующий элемент дескрипторной таблицы прерываний содержит шлюз задачи.

До перехода в многозадачный режим необходимо определить дескрипторы TSS, разместить сегменты TSS в адресном пространстве и правильно их инициализировать. Для работы с сегментами TSS используется альтернативное именование. При этом создается сам дескриптор TSS и дескриптор сегмента данных, описывающий ту же область памяти. Эти два дескриптора рекомендуется располагать в GDT один за одним. Перед первым запуском задачи в сегменте TSS (в регистре CS : EIP) должен содержаться адрес первой команды, а сегментные регистры данных должны содержать селекторы сегментов данных для данной задачи. В регистр SS следует загрузить селектор сегмента стека с правильным уровнем привилегий. Регистры общего назначения могут содержать нули, если не определять конкретные начальные значения. Если задача имеет LDT, то должен быть определен селектор для этой таблицы, а если будет поддерживаться страничное преобразование, то и регистр CR3. Страничное преобразование и условие работы с математическим сопроцессором являются обязательными для всех задач.

В регистре задачи TR находится селектор дескриптора сегмента состояния задачи. Поэтому вначале переключения задачи процессор знает, куда ему нужно сохранить текущее состояние задачи, а затем в регистр TR новой задачи устанавливается бит занятости новой задачи и устанавливается бит TS задачи в регистре CR0. Селектор новой задачи берется или из команд **JMP** и **CALL,** либо берется из шлюза задачи, если адресуется шлюз задачи. Потом загружается состояние входящей задачи из сегмента TSS и продолжается выполнение следующей команды. Бит TS используется системными правилами, с целью организации использования математического сопроцессора. Если бит TS установлен и идет обращение к сопроцессору, то ОС знает, что необходимо сохранить содержимое и установить новое значение. Уровни привилегий между собой не связаны.

9.1.4.5 Формат шлюза задач

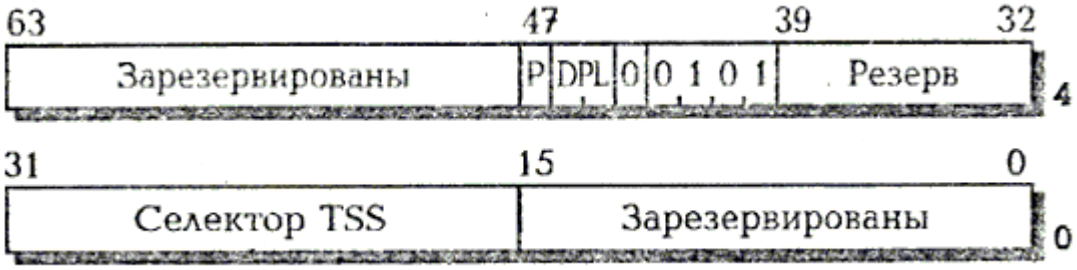


Рис. 9.16. Формат дескриптора шлюза задачи

Правила защиты при переключении задач выглядят так:

Max $(CPL, RPL) \leq DPL_{\text{шлюза задачи}}$
Max $(CPL, RPL) \leq DPL_{\text{сегмента TSS}}$

9.1.4.6. Общая схема переключения задач

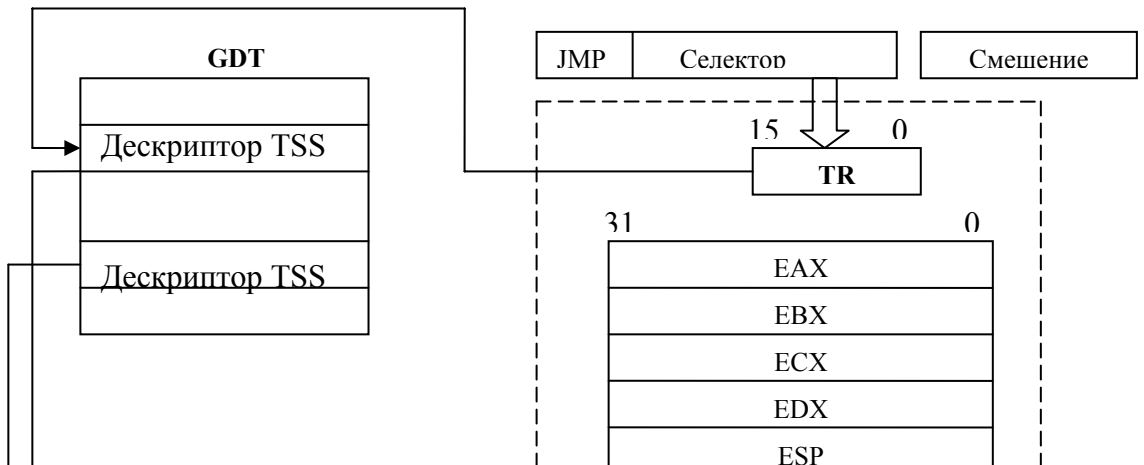


Рис. 9.17 Общая схема переключения задач

9.1.4.7 Особые случаи при переключении задач

При переключении задач могут возникнуть следующие особые случаи:
- неприсутствие;
- нарушение общей защиты;
- неверный сегмент TSS;
- нарушение стека.

Задачи являются нереентерабельными (т.е. без повторного вхождения, нельзя переключится задаче самой на себя). О занятости задачи информирует бит занятости в дескрипторе сегмента состояния задачи. Переключение на занятую задачу процессор не производит, а генерирует особый случай.

9.1.4.8 Вложенность задач

Когда переключение задач инициируется командой ***far call***, аппаратным прерыванием или особым случаем, задача, на которую инициируется переключение, считается вложенной в ту задачу, из которой произошло переключение. Это похоже на вызов подпрограмм.

Когда вложенная задача выполнила команду ***iret***, процессор автоматически переключается на прерванную задачу. Глубина вложенности не ограничивается.

В качестве механизма связи вложенных задач используется поле обратной связи в сегменте *TSS*, в котором сохраняется старое содержимое регистра задачи. Кроме того, процессор устанавливает в единицу бит вложенной задачи *NT* в регистре *EFLAGS* , который свидетельствует о том, поле обратной связи содержит информативное значение.

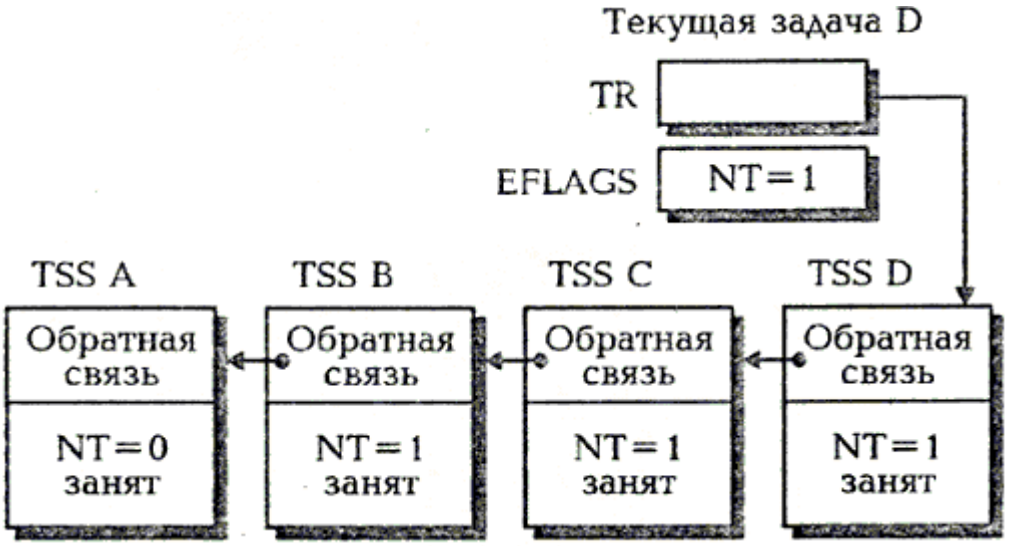


Рис. 9.18. Цепь из трех вложенных задач

Переключение на задачу А было проведено с помощью команды ***far jmp***, т.к. бит *NT=0*, а далее эта задача вызвала следующую с помощью команды ***far call***. Биты вложенности устанавливается в 1, а поле обратной связи указывает на предыдущую задачу. Но для всех этих задач остается включенным бит занятости в дескрипторе сегмента *TSS.*

Фактически вложенность задач учитывает только команда *IRET*, обычная команда возврата из подпрограммы *RET* вложенности не учитывает.

9.1.4.9 Двоичная карта разрешения ввода-вывода

Последнее 16-битное слово в базовом *TSS* содержит смещение начала двоичной карты разрешения ввода-вывода. Она является дополнительным средством механизма защиты по привилегиям. Такая карта создается для каждой задачи и участвует в контроле привилегии команд ввода-вывода.

В двоичной карте каждый бит соответствует одному адресу ввода-вывода, то есть байтному порту ввода-вывода. Младший бит первого байта карты отно-

сится к нулевому адресу и далее- по возрастающей. Полная карта ввода-вывода может занимать 8 Кб памяти. Но она может занимать меньше, если не все порты нужно закрывать на разрешение доступа.

Двоичная карта ввода-вывода должна располагаться вблизи сегмента *TSS*.

9.1.5 Прерывания и особые случаи

В ходе работы ЭВМ могут возникать ситуации настолько важные, что заставят процессор приостановить текущую программу и переключиться на выполнение другой, более срочной и важной. Причинами прерывания текущей программы могут быть:

1. Внешний сигнал по входу маскируемых прерываний *INTR* или немаскируемого прерывания *NMI;*

2. Ненормальная ситуация, сложившаяся при выполнении конкретной команды и препятствующая нормальному ходу выполнения программы;

3. Находящаяся в программе команда прерывания *INT n*, где n – номер прерывания.

Реагируя на внешнее прерывание, процессор должен:

1) определить его источник;

2) сохранить минимальный контекст текущей программы (по крайней мере адрес возврата);

3) переключиться на специальную программу – обработчик прерывания, которым может быть процедура или задача. Обработчик должен выполнять действия, нормализующие ситуацию, после чего процессор возвратится к прерванной программе и она возобновиться так, будто прерывания не было.

Программные прерывания обычно называются ***особыми случаями (exeptions)***. Реакция процессора на особые случаи подобна реакции на аппаратное прерывание, а действия обработчика зависят от условий, при которых возник особый случай.

Особые случаи бывают следующих типов:

– нарушение (*fault);*

– ловушка (*trap*);

– авария (*abort*).

Нарушение – это такой особый случай, который процессор может обнаружить до возникновения фактической ошибки. Например, нарушение правил привилегий или выход за границы сегмента. После корректной обработки нарушения можно продолжить программу, осуществив повторное выполнение команды, вызвавшей особый случай.

Ловушка – это такой особый случай, который обнаруживается после окончания виновной команды. После его обработки процессор возобновляет действия с той командой, которая находится после команды, приведшей к особому случаю. Примером таких команд являются команды, вызвавшие переполнение или команда INT.

Авария – очень серьезная ошибка, в результате которой часть программы или вся программа теряется или искажается и продолжить ее невозможно. Причину аварии установить невозможно, поэтому restart программы не удается и программу необходимо прекратить. К авариям относятся аппаратные ошибки или серьезные искажения системных данных или системных программ.

9.1.5.1 Прерывания и особые случаи в процессоре 8086

Всем источникам прерываний назначается номер от 0 до 255 (**вектор**), который позволяет процессору выбрать нужный обработчик. Некоторые вектора зарезервированы для специальных целей, а незанятые вектора предоставляются в распоряжение программы.

Таблица векторов прерываний находится в памяти и состоит из 4-байтных элементов и начинается с нулевого физического адреса памяти, занимая максимально 1Кб. Любой элемент таблицы представляет собой полный указатель – селектор смещения точки входа в обработчик прерывания

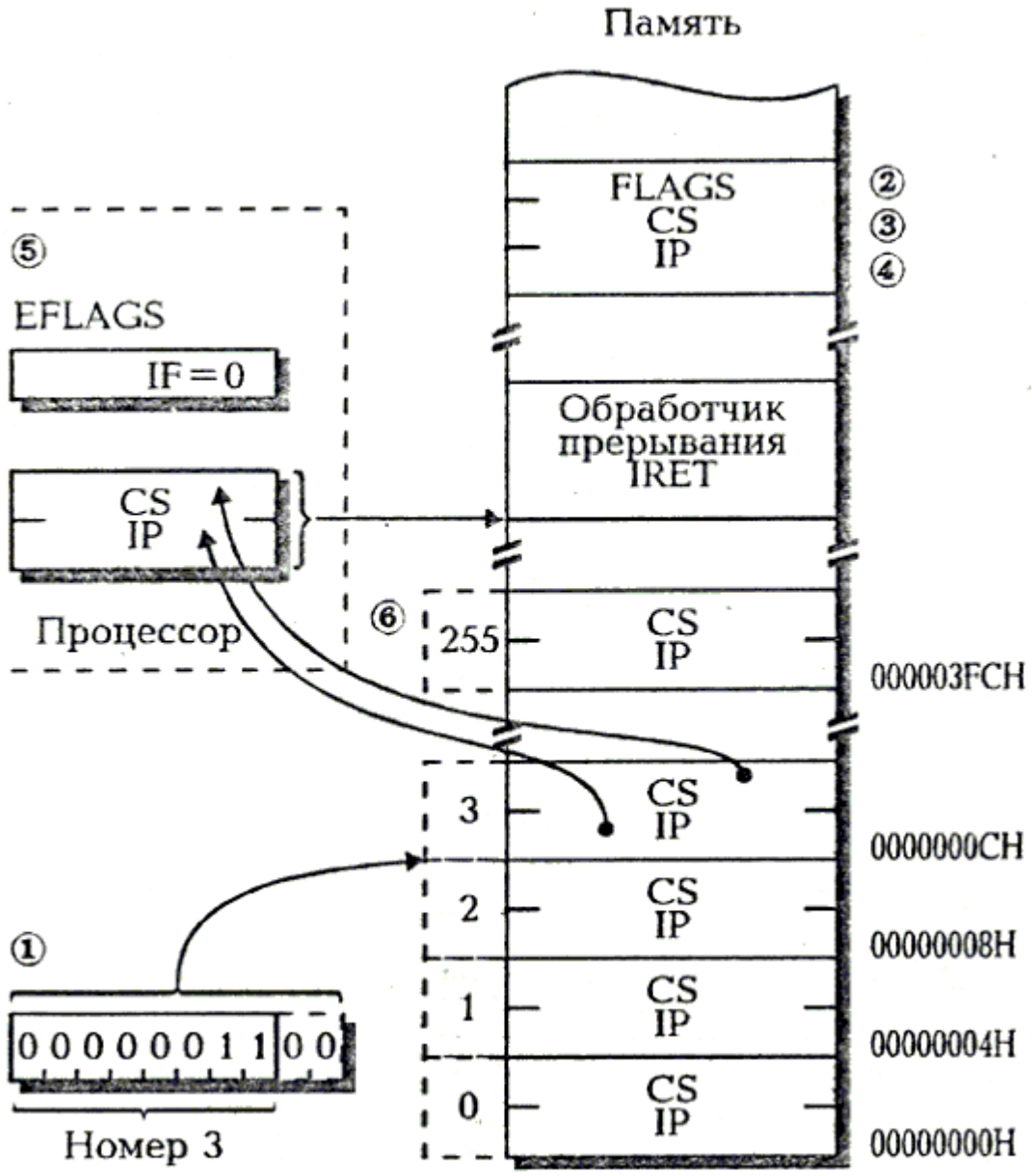


Рис. 9.19. Схема обработки прерываний при работе процессора в реальном режиме

При обработке аппаратных прерываний выполняются следующие действия.

1. Вводится номер прерывания от программируемого контроллера прерываний. Для этого требуется два цикла шины на подтверждение прерывания. Контроллер прерываний – это микросхема, выполняющая вставку вектора.

2. Включается в стек содержимое регистра флагов (16 бит).

3. Включается в стек содержимое регистра *CS*.

4. Включается в стек содержимое регистра *IP*.

5. Сбрасывается в ноль флажок прерывания *IF*, запрещая восприятие дальнейших прерываний до момента установки в 1 этого флажка программой обработки прерывания.

6. По номеру прерывания производится обращение к соответствующему элементу таблицы векторов, из которой извлекается содержимое регистров *CS* и *IP*.

7. Начинается выполнение обработчика прерывания с точки входа, определяемое регистрами *CS:IP*.

Когда обработчик прерывания закончил свои действия, он выполняет команду возврата из прерывания (*IRET*) и управление передается прерванной подпрограмме.

При обработке команды *INT n* номер прерывания содержится в команде, соответственно не нужны циклы шины, а флажок I*F* не сбрасывается в 0, что разрешает обработку аппаратных прерываний.

Особый случай - это внутреннее событие при работе процессора. Для особых случаев назначаются фиксированные номера в диапазоне от 0 до 31.

Процессор 8086 распознает только 5 особых случаев и всегда реагирует на них независимо от состояния флажков прерывания:

(0) – особый случай деления на ноль;

(1) – особый случай покомандной или пошаговой обработки;

(2) – особый случай немаскируемых прерываний NMI;

(3) – особый случай контрольной точки ***int 3;***

(4) – особый случай прерывания при переполнении (фиксируется, если при выполнении команды *INTO* устанавливается флаг *OF=1*).

9.1.5.2 Прерывания в защищенном режиме

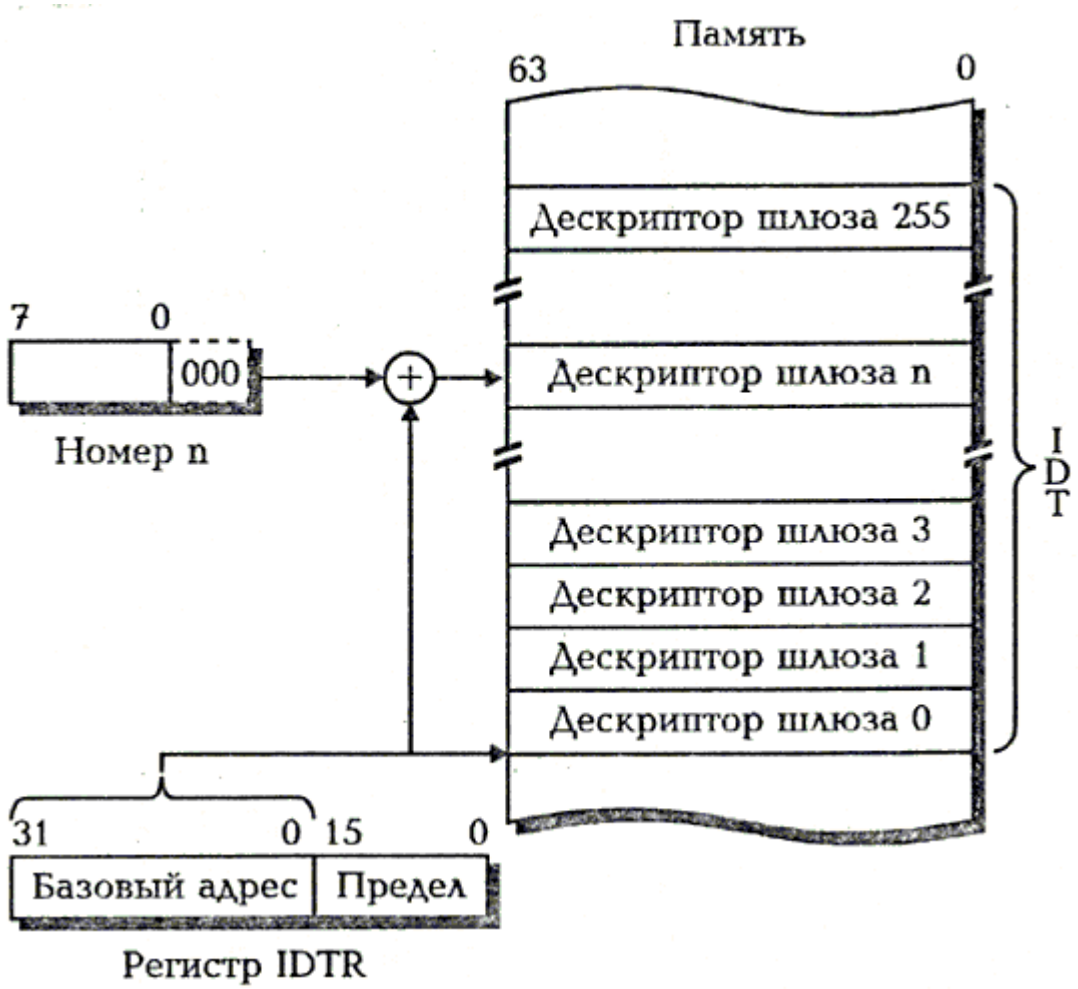


Рис. 9.20. Схема обработки прерываний при работе процессора в защищенном режиме

Механизм обработки прерываний и особых случаев при работе в защищенном режиме усовершенствован по сравнению с реальным режимом, а именно:

1. Таблица векторов прерываний трансформирована в дескрипторную таблицу прерываний (*IDT*).

2. Усложнен процесс перехода к обработке прерывания или особых случаев.

3. Имеется возможность передачи дополнительной информации о причине возникновения особых случаев обработки прерываний.

4. В защищенном режиме процессор распознает большее число особых случаев (например: 0 – деление на нуль, 1 – пошаговая обработка, 2 – критическая точка).

Каждому аппаратному прерыванию и программному особому случаю назначается индивидуальный номер, по которому процессор обращается к деск-

рипторной таблице прерываний. Таблица прерываний находится в памяти и её линейный базовый адрес хранится в регистре IDTR. Она представляет собой массив шлюзов, через который осуществляется передача управления обработчику прерываний или особых случаев. В этом регистре также хранится предел, определяющий размер таблицы. Таблица прерываний является общей для всех задач и процессов. Поэтому никакой информации о ней не сохраняется в сегменте состояния задач. Программы не имеют возможность напрямую обращаться к *IDT* , т.е. они могут выбирать через селектор только *LDT* или *GDT*.

Процессор до передачи управления включает в стек обработки минимум 12 байт (адрес возврата и содержимое регистра флажков).

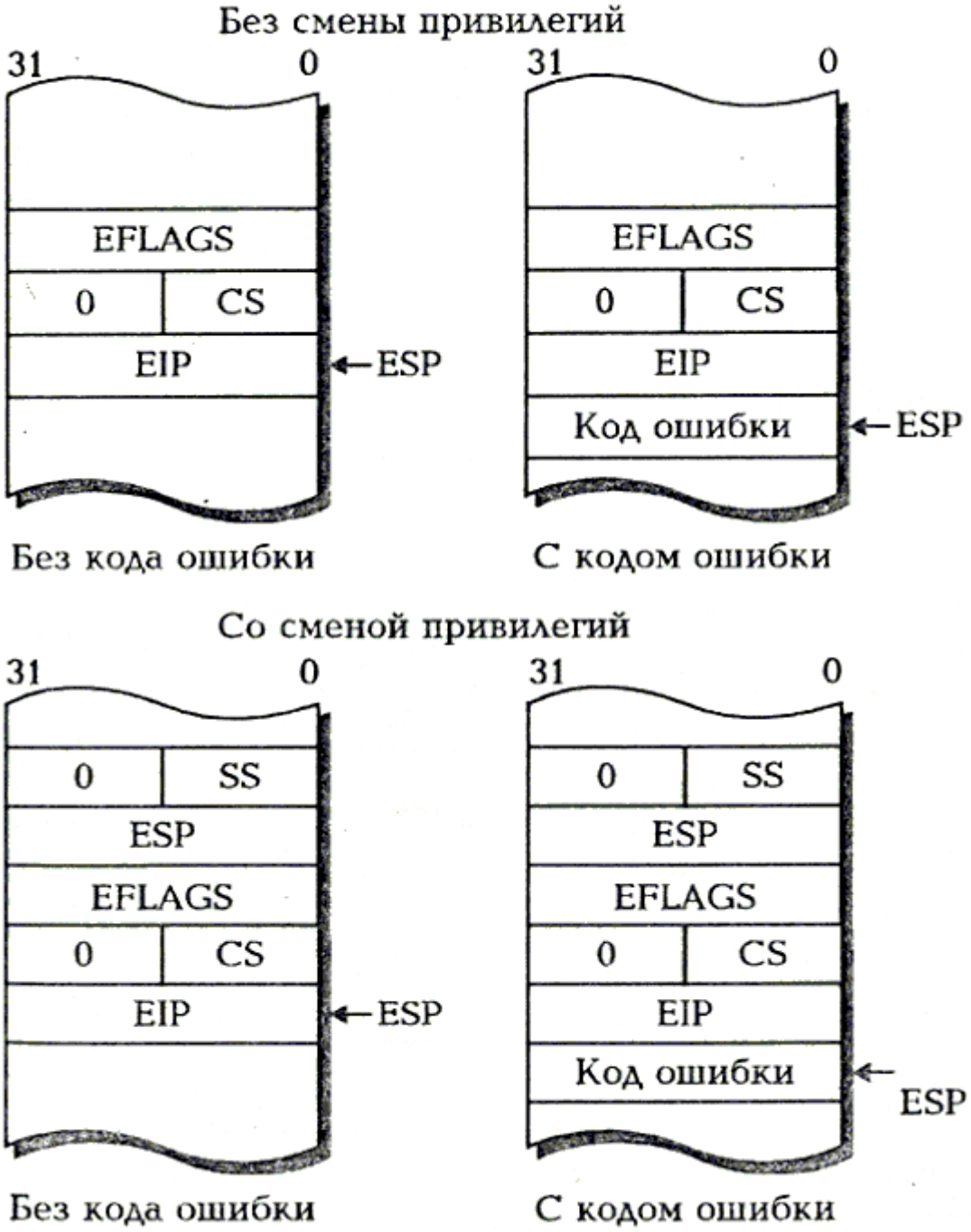


Рис. 9.21. Форматы стека при передаче управления обработчику прерываний

В некоторых программных особых случаях процессор включает в стек код ошибки, по которому можно уточнить причину особого случая. Уровень привилегий обработчика определяется полем *DPL* в дескрипторе его сегмента кода. Обработчик может обращаться к любому сегменту памяти через глобальные или локальные (для текущей задачи) таблицы на своем уровне привилегий и передавать управление с памяти команд *far jmp* и *far call*, изменять уровень привилегий с помощью шлюза вызова и производить ввод-вывод. Управление возвращается прерванной программе командой *IRET*.

Особых случаев при работе в защищенном режиме больше, чем при работе в реальном режиме. Если на границе команды возникает несколько особых случаев, то процессор обслужит их в порядке их приоритетности. Менее приоритетные особые случаи уничтожаются; менее приоритетные прерывания остаются ждать. Уничтоженные особые случаи формируются вновь, когда обработчик прерываний возвращает управление в точку прерывания.

Особые случаи имеют следующую приоритетность(в порядке убывания):

- особые случаи отладки;
- немаскируемые прерывания;
- маскируемые прерывания;
- нарушение при выборке команды и её декодировании;
- особый случай недоступности сопроцессора;
- нарушения неприсутствия сегмента;
- нарушения стека;
- нарушения общей защиты для операндов в памяти;
- нарушение выравнивания операндов в памяти;
- страничное нарушение для операндов в памяти.

9.1.5.3 Дескрипторная таблица прерываний

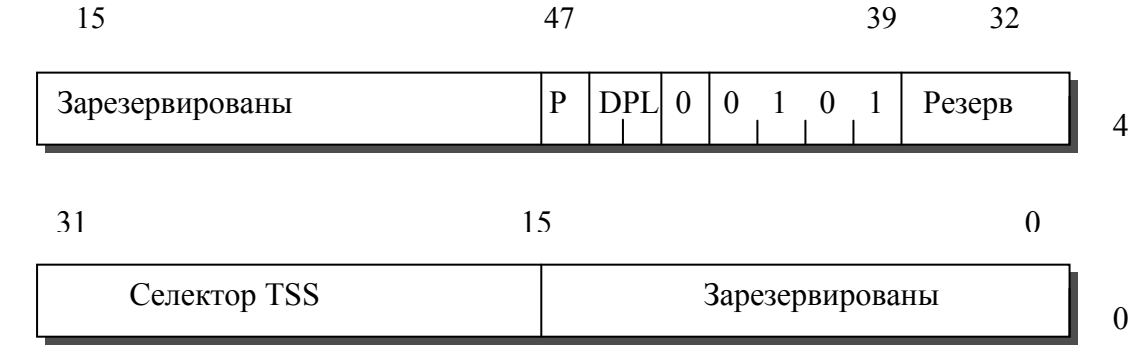
Дескрипторная таблица прерываний представляет массив восьмибайтных дескрипторов. В ней должны быть определены 256 обработчиков. Поэтому ее максимальный предел должен иметь значение 2047 или 7FFh. Если возникает особый случай и его номер выбирается за пределами таблицы, генерируется нарушение общей защиты, т. е. процессор переходит в режим отключения. В этом режиме он прекращает выполнение команд до получения немаскируемых прерываний или при сбросе процессора. Переход в этот режим процессор показывает специальным циклом шины и к этому циклу можно прикрепить какой-либо аппаратный индикатор.

В отличие от глобальной таблицы, нулевой дескриптор является действительным.

В дескрипторной таблице прерываний могут храниться три вида дескрипторов:

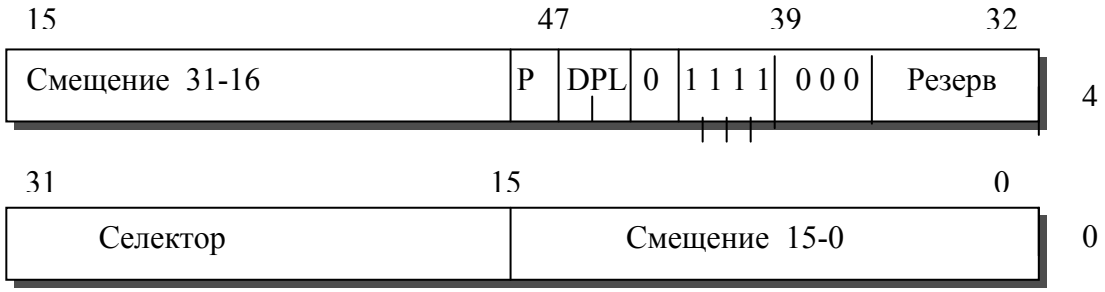
- шлюз задачи;
- шлюз прерывания;

- шлюз ловушки.



Шлюз задачи

Шлюз прерывания



Шлюз ловушки

Рис. 9.22. Форматы дескрипторов шлюзов, разрешенных в дескрипторной таблице прерываний

Шлюзы ловушки и прерывания очень похожи на шлюз вызова, за исключением того, что в них нет поля счетчика слов, т. к. обработчикам прерываний параметры не передаются. Эти шлюзы однозначно определяют точку входа в обработчик. Поле уровня привилегий *DPL* в этих дескрипторах определяет тот

минимальный уровень, который необходим для передачи управления через шлюз. Рекомендуется это поле устанавливать равным 3, чтобы дескрипторы из таблицы прерываний были доступны всем задачам, выполняемым в системе.

9.1.5.4 Шлюз ловушки

Когда возникает особый случай и по его номеру выбирается шлюз ловушки, процессор сохраняет часть своего состояния в соответствие с приведенным ранее рисунком, а затем осуществляет межсегментную передачу управления, используя селектор и смещение из шлюза ловушки.

Селектор должен выбирать сегмент кода, который может адресовать через *GDT* или *LDT*. Но рекомендуется хранить дескрипторы всех обработчиков в *GDT*.

Целевой сегмент кода может быть отмечен как отсутствующий. В этом случае вырабатывается особый случай неприсутствия, по которому должно быть выполнено перемещение обработчика из внешней памяти в оперативную. По коду *IRET* процессор извлекает из стека адрес возврата и содержимое регистра флажков, а, если происходит смена привилегий, то и адрес внешнего стека, после чего прерванная задача возобновляется.

9.1.5.5 Шлюз прерывания

Действует, как и шлюз ловушки, но в этом случае процессор сбрасывает в ноль флажки прерываний. Этот сброс осуществляется после включения регистра флажков в стек, но до выполнения первой команды обработчика. Это значит, что блокируются вновь поступившие аппаратные прерывания до тех пор, пока текущий обработчик не завершит свою работу. Но немаскируемые и программные прерывания не запрещаются.

9.1.5.6 Шлюз задачи

Когда номер прерывания выбирает в таблице прерываний шлюз задачи, процессор переключается на новую задачу, определенную селектором сегмента *TSS* задачи в шлюзе задачи. Такой шлюз не может содержать селектор сегмента *TSS* из *LDT* и селектор шлюза еще одной задачи.

При переключении задачи селектор *TSS* текущие задачи помещает в поле обратной связи *TSS* новой задачи и в ее регистре флажков устанавливает *NT = 1*. Когда обработка завершается командой *IRET* осуществляется переключение на прерванную задачу с использованием поля обратной связи.

Обработка особых случаев через шлюз задачи имеет следующие преимущества:

- автоматически сохраняется весь контекст прерванной задачи;
- обработчик особых случаев не может исказить прерванную задачу, так как она полностью изолирован от нее;
- обработчик прерываний может работать на любом уровне привилегий в заведомо правильной среде. Может иметь свое локальное адресное пространство, благодаря LDT.

Недостатки применения шлюза задачи:

- реакция процессора несколько замедлена, т.к. необходимо сохранять большее количество информации при переключении задач;
- в шлюзе задачи невозможно определить начальную точку выполнения задачи;
- сложно получить информацию о прерванной задаче. Эту информацию можно получить, используя только поле обратной связи.

Когда для обработки особых случаев привлекаются шлюзы задач, необходимо обратить внимание на то, чтобы избежать рекурсии особых случаев, т.к. и задача-обработчик, и задача, которая была прервана, будут отмечены как занятые, а на занятые задачи переключиться нельзя.